Taming Lambdas

Dawid Zalewski

zaldawid@gmail.com
github.com/zaldawid
linkedin.com/in/dawidzalewski

Taming Lambdas' Uniqueness

Dawid Zalewski

zaldawid@gmail.com
github.com/zaldawid
linkedin.com/in/dawidzalewski

What this talk is not about

- Functional programming
  (recursion, higher-order functions, y-combinator)
- Type theory
  (type systems, empty vs. bottom type, λ-calculus)
- Metaprogramming
  (metafunctions, traits, decltype, parameter packs)
- Performance optimization
  (benchmarking, dynamic vs. static dispatch)

Lambdas that aren't the same

```
[](){} != [](){}
```

Lambdas that aren't the same

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

int main() {

  std::cout << std::is_same_v< decltype([](){}), decltype([](){}) >;

}
```

```
> ./same.a
0
```

Lambdas that aren't the same

```
std::vector sinks {
                   [](std::string_view str){ std::cout << str; },
                   [](std::string_view str){ std::cerr << ts() << str; },
                   [](std::string_view str){ std::ofstream{"out"} << str; }
                  };
```

Lambdas that aren't the same

```
struct ImageFilter
{

  using filter_alg_t = decltype([](ImageData& img){});

  filter_alg_t alg;

  ImageFilter(filter_alg_t alg) : alg(std::move(alg)) {}

};
```

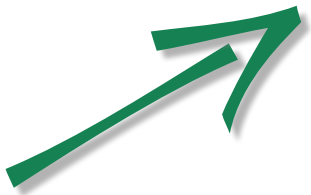
Lambdas that aren't the same

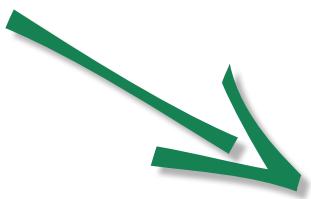
[expr.prim.lambda.closure]

The type of a lambda-expression (which is also the type of the closure object) is a **unique**, unnamed non-union **class type**, called the closure type (…).

Closures ~~Lambdas~~ that aren't the same

```
int main() {

  [](){};

  [](){};

}
```





```
class __lambda_main_3_3 {
public:
  constexpr __main_lambda_3_3() = default;

  inline constexpr void operator()() const { }
};

class __lambda_main_5_3 {
public:
  constexpr __main_lambda_5_3() = default;

  inline constexpr void operator()() const { }
};
```

Simple lambdas behave well

```
std::vector sinks {
                    [](std::string_view str){ std::cout << str; },
                    [](std::string_view str){ std::cerr << ts() << str; },
                    [](std::string_view str){ std::ofstream{"out"} << str; }
                  };
```

Non-capturing lambdas behave well

```
std::vector sinks {
                  [](std::string_view str){ std::cout << str; },
                  [](std::string_view str){ std::cerr << ts() << str; },
                  [](std::string_view str){ std::ofstream{"out"} << str; }
                };
```

Non-capturing lambdas behave well

```
std::vector sinks {
                   +[](std::string_view str){ std::cout << str; },
                   +[](std::string_view str){ std::cerr << ts() << str; },
                   +[](std::string_view str){ std::ofstream{"out"} << str; }
                  };
```

Non-capturing lambdas behave well

```
decltype( +[](std::string_view str){ std::cout << str; } )
```

==

```
void(*)(std::string_view)
```

Non-capturing lambdas behave well

```
[](std::string_view str)
{
  std::cout << str;
};
```

```
class __lambda_dd {
public:
  void operator()(std::string_view str) const {
    std::cout << str;
  }

  using func_ptr_t = void(*)(std::string_view);

  operator func_ptr_t() const noexcept {
      return apply;
  }

private:
  static void apply(std::string_view str) {
      return __lambda_dd{}.operator()(str);
  }
};
```

Non-capturing lambdas behave well

```
std::vector sinks
                  {
                    +[](std::string_view str){ std::cout << str; },
                    +[](std::string_view str){ std::cerr << ts() << str; },
                    +[](auto str){ std::ofstream{"out"} << str; }
                  };
```

Non-capturing, generic lambdas behave well

```
std::vector<void(*)(std::string_view)> sinks
                  {
                    +[](std::string_view str){ std::cout << str; },
                    +[](std::string_view str){ std::cerr << ts() << str; },
                    +[](auto str){ std::ofstream{"out"} << str; }
                  };
```

Non-capturing, generic lambdas behave well

```
std::vector<void(*)(std::string_view)> sinks
                  {
                     [](std::string_view str){ std::cout << str; },
                     [](std::string_view str){ std::cerr << ts() << str; },
                     [](auto str){ std::ofstream{"out"} << str; }
                  };
```

Non-capturing, generic lambdas behave well

```
std::ofstream out{"output.txt"};

std::vector sinks {
                    +[](std::string_view str){ std::cout << str; },
                    +[](std::string_view str){ std::cerr << ts() << str; },
                    +[&out](std::string_view str){ out << str; }
                  };
```

Capturing lambdas are not simple

```
std::ofstream out{"output.txt"};
decltype( +[&out](std::string_view str){ out << str; } )
```

==

```
error: no match for 'operator+'
 (operand type is 'main()::<lambda(std::string_view)>')
```

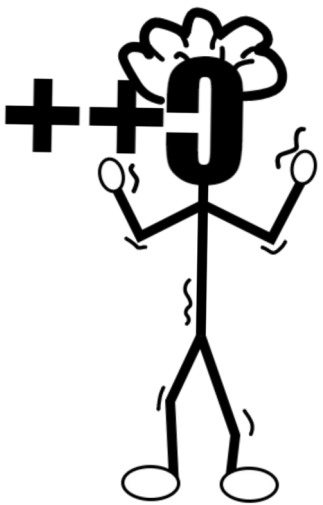
Capturing lambdas are not simple

```
[&out](std::string_view str)
{
  out << str;
};
```

```
class __lambda_dd {
public:
  void operator()(std::string_view str) const {
    out << str;
  }

  __lambda_dd(std::ofstream& out_):
    out{ out_ } {}

private:

  std::ofstream& out;

  static void apply(std::string_view str) {
      return __lambda_dd{ ??? }.operator()(str);
  }
};
```

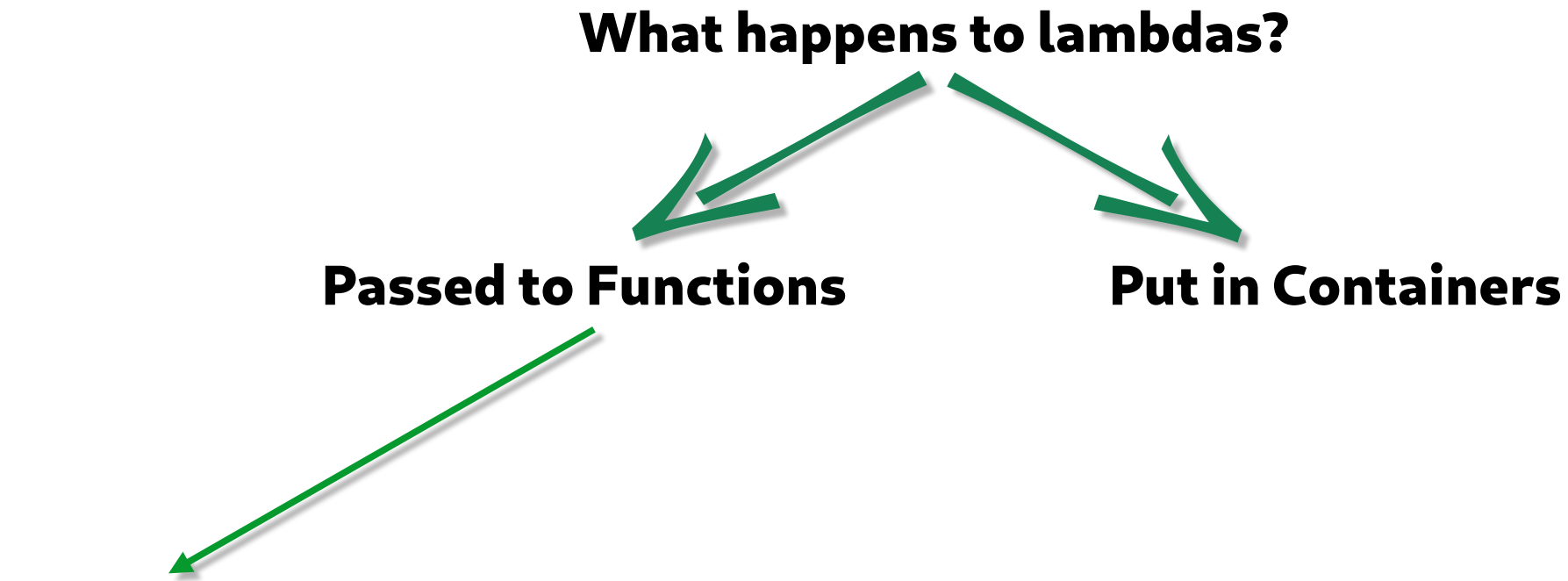
You don’t want function pointers

```
template <auto N>
void write(void(*(&fns)[N])(std::string_view), int ind, std::string_view str)
{
  (*(fns + ind))(str);
}
```



The troublemakers

```cpp
auto to_cout = [](std::string_view str){ std::cout << str; };

auto to_err = [&out=std::cerr](std::string_view str) { out << str; };

std::ofstream log{"output.txt"};
auto to_log = [out=std::move(log)](std::string_view str) mutable { out << str; };
```



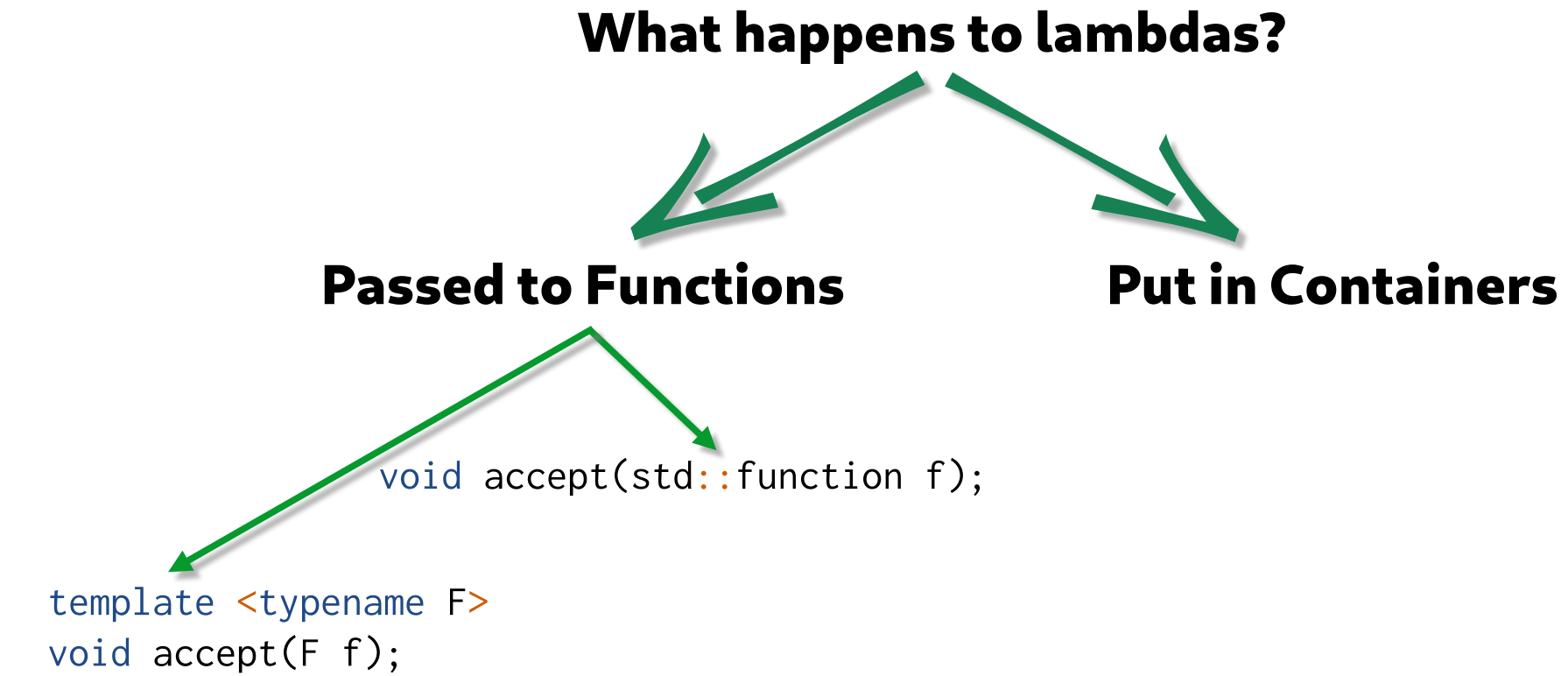

```
template <typename F>
void accept(F f);
```

Just passing using templates (by ref)

```
template <typename Sink>
void accept(Sink& sink, std::string_view str)
{
  sink(str);
}

accept(to_cout, "Lambdas");

accept(to_err, "are");

accept(to_log, "sooo cool!");
```

Just passing using templates (by copy)

```
template <typename Sink>
void accept(Sink sink, std::string_view str)
{
  sink(str);
}

accept(to_cout, "Lambdas");

accept(to_err, "are");

accept(std::move(to_log), "sooo cool!");
```


What happens to lambdas?
Passed to Functions
Put in Containers
void accept(std::function f);
template <typename F>
void accept(F f);

Just passing as std::function

```cpp
using sink_type = std::function<void(std::string_view)>;

void accept(sink_type sink, std::string_view str)
{
  sink(str);
}

accept(to_cout, "Lambdas");

accept(to_err, "are");

accept(std::move(to_log), "sooo cool!");
```

to_log is not copy-constructible

Just passing as std::move_only_function

```cpp
using sink_type = std::move_only_function<void(std::string_view)>;

void accept(sink_type sink, std::string_view str)
{
  sink(str);
}

accept(to_cout, "Lambdas");

accept(to_err, "are");

accept(std::move(to_log), "sooo cool!");
```

Passing around, costs

1

```
long f1(long, long);
...
long f16(long, long);

f1(a, b);
...
f16(a,b)
```

2

```
auto l1  = [](long, long) -> long {...};
...
auto l16 = [](long, long) -> long {...};

l1(a, b);
...
l16(a,b);
```

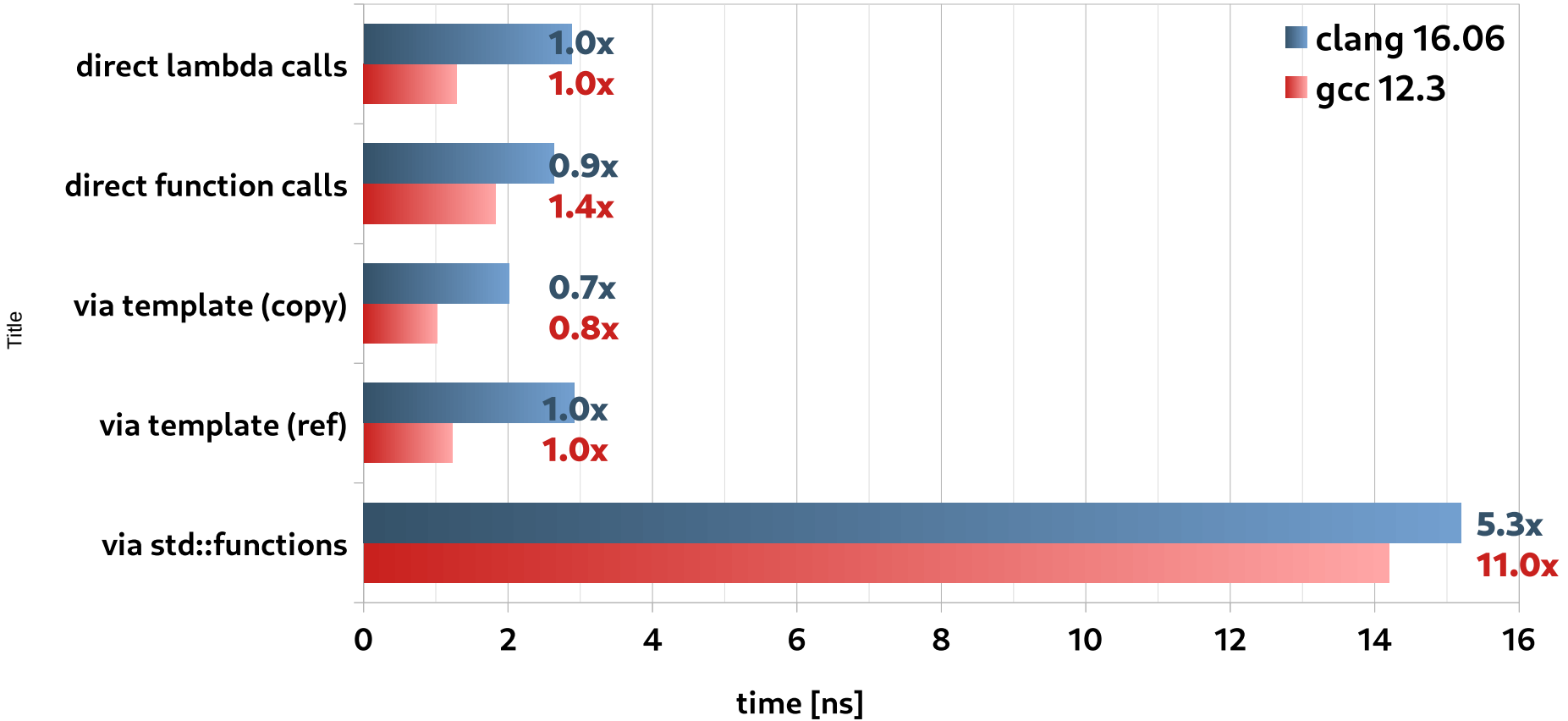
3

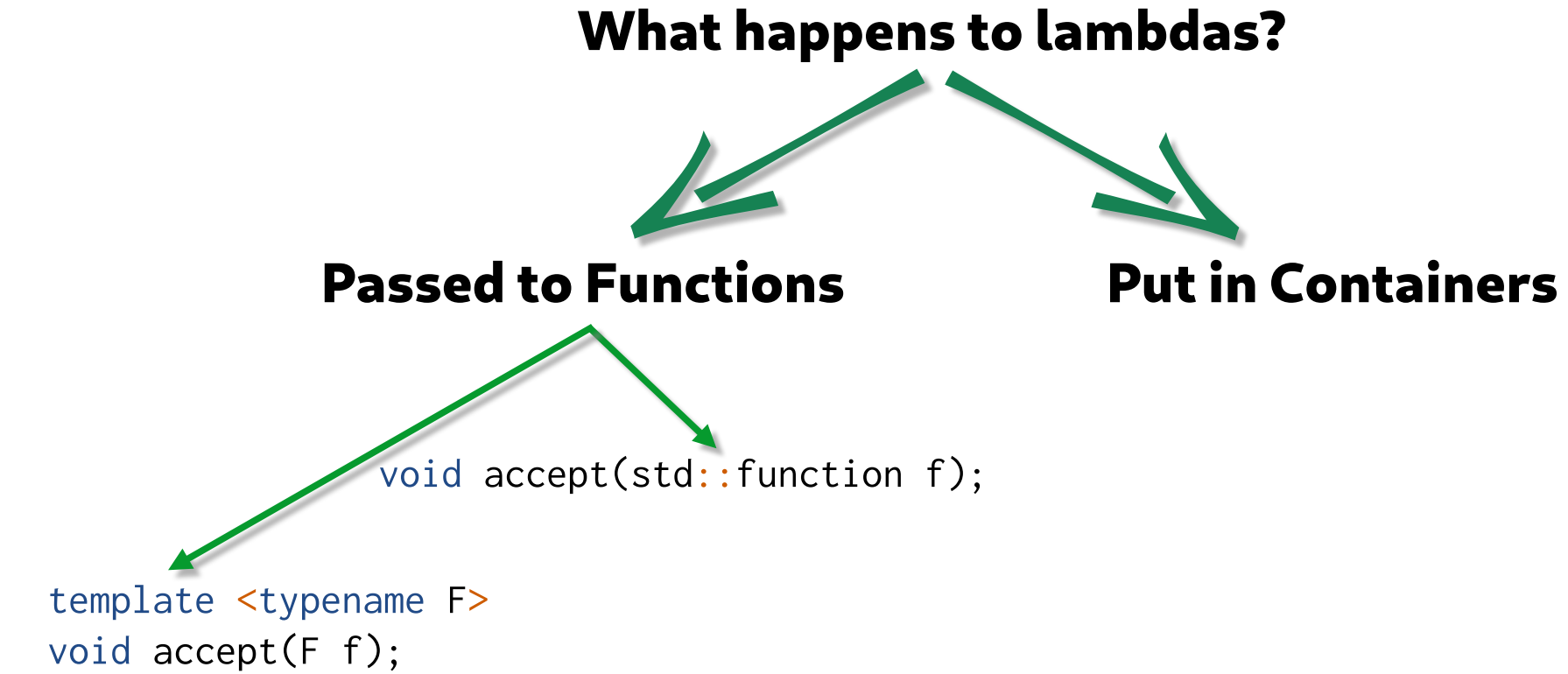
```
auto l1  = [](long, long) -> long {...};
...
auto l16 = [](long, long) -> long {...};

template <typename F>
long call(F f) { return f(a, b); }

call(l1);
...
call(l16);
```

4

```
using func_t = std::function<long(long,long)>;

auto l1  = func_t{ (long, long) -> long {...} };
...
auto l16 = func_t{ [](long, long) -> long {...} };

long call(func_t f) { return f(a, b); }

call(l1);
...
call(l16);
```

Passing around, the costs




What happens to lambdas?
Passed to Functions
Put in Containers
void accept(std::function f);
template <typename F>
void accept(F f);

Lambdas and their interfaces

```cpp
auto to_cout = [](std::string_view str){ std::cout << str; };
```

```cpp
auto to_err = [&out=std::cerr](std::string_view str) { out << str; };

std::ofstream oss{};
auto to_str = [&out=oss](std::string_view str) { out << str; };
```

```cpp
std::ofstream log{"output.txt"};
auto to_log = [out=std::move(log)](std::string_view str) mutable { out << str; };
```

```cpp
std::ofstream sev_log{"sev_output.txt"};
auto to_sev =
     [out=std::move(sev_log)](std::string_view str, severity sev) mutable
     {
       out << sev << str;
     };
```

Lambdas and their interfaces

	Different type signature	Same type signature
Same body		*Factory!*
Different body		

Same body, same type signature

```
std::ofstream oss{};

auto to_err = [&out=std::cerr](std::string_view str) { out << str; };

auto to_str = [&out=oss]      (std::string_view str) { out << str; };

std::vector sinks{ to_err, to_str };
```

```
error: class template argument deduction failed
note:   couldn't deduce template parameter '_Tp'
```

Same body, same type signature

```
std::ofstream oss{};


auto make_output  = [](std::ostream& out)
{
  return [&out](std::string_view str){ out << str; };
};

std::vector sinks{
                    make_output(std::cerr),
                    make_output(oss)
                  };
```

Just one unique lambda!

Same body, same type signature

```
std::ofstream log{"output.txt"};
std::ofstream errors{"error_log.txt"};


auto make_output  = [](std::ofstream&& out)
{
  return [out=std::move(out)](std::string_view str) mutable { out << str; };
};

std::vector sinks{
                    make_output(std::move(log)),
                    make_output(std::move(errors))
                  };
```

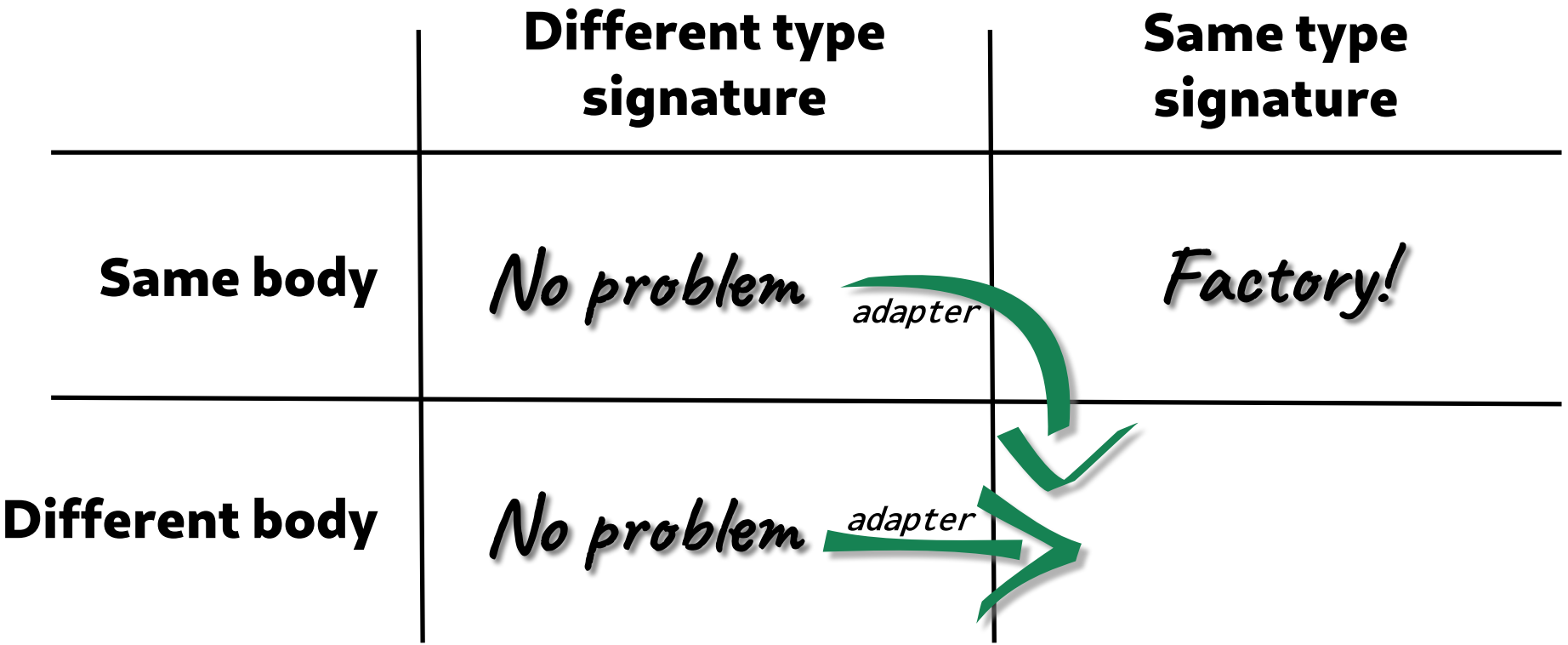
Same body, same type signature

```
std::array streams {
  std::ofstream{"output.txt"},
  std::ofstream{"error_log.txt"}
};

auto make_output  = [](std::ofstream&& out)
{
  return [out=std::move(out)](std::string_view str) mutable { out << str; };
};

std::vector<decltype(make_output(std::declval<std::ofstream>()))> outputs { };

for (auto&& stream : streams)
  outputs.push_back( make_output( std::move(stream)) );
```

Same body, same type signature

```
std::array streams {
  std::ofstream{"output.txt"},
  std::ofstream{"error_log.txt"}
};

auto make_output  = [](std::ofstream&& out)
{
  return [out=std::move(out)](std::string_view str) mutable { out << str; };
};

auto outputs = streams
   | std::ranges::views::as_rvalue
   | std::ranges::views::transform(make_output)
   | std::ranges::views::as_rvalue
   | std::ranges::to<std::vector>();
```

Lambdas and their interfaces



Different body, different type signature

```
std::ofstream log{"output.txt"};
auto to_log = [out=std::move(log)](std::string_view str) mutable { out << str; };
```

```
(std::string_view) -> void
```

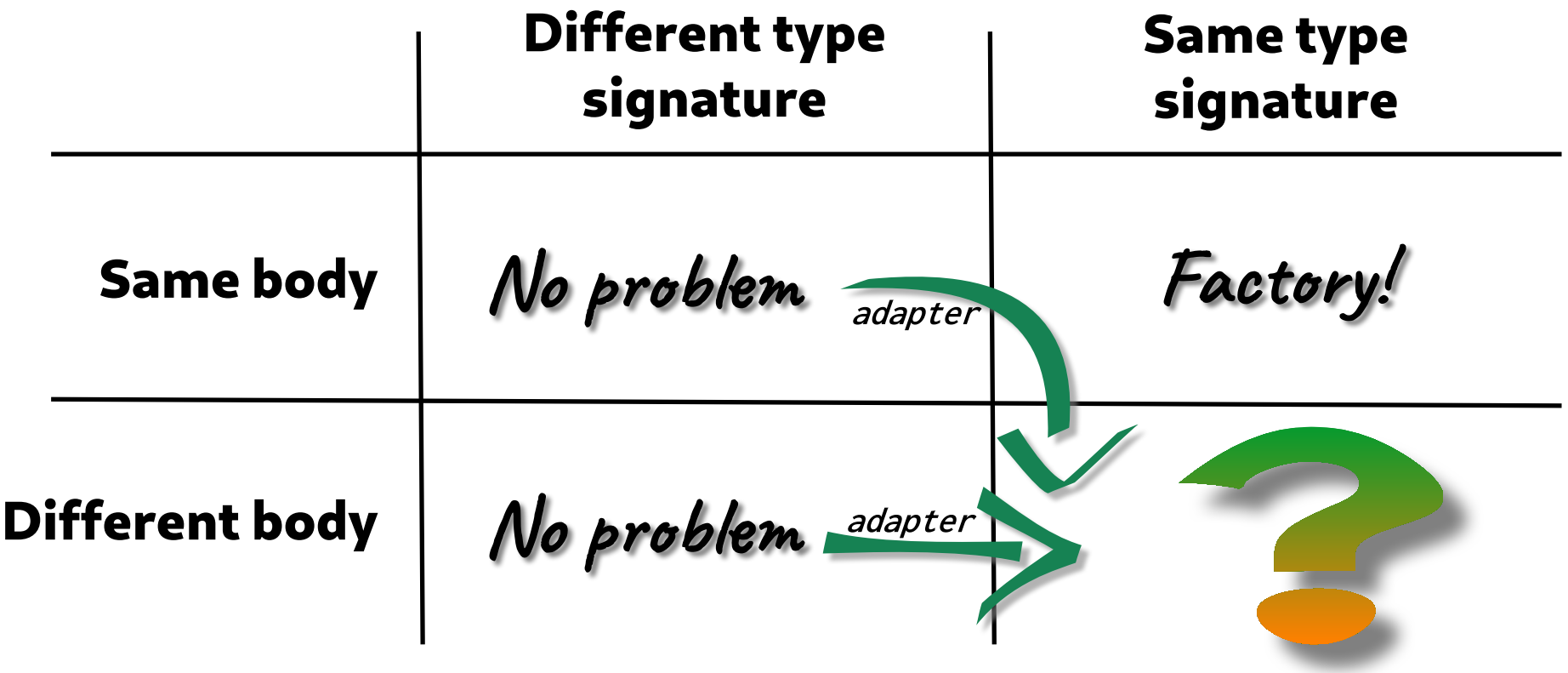
Different body, different type signature

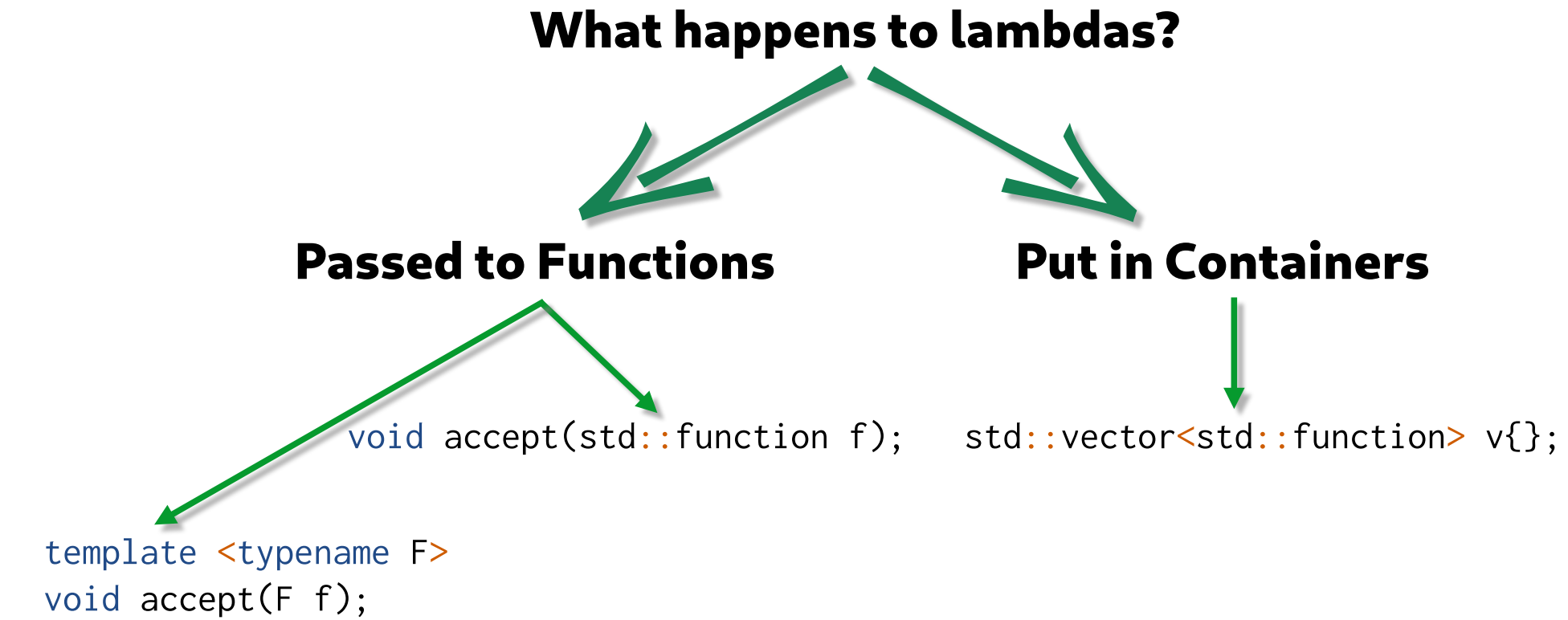
```
std::ofstream sev_log{"sev_output.txt"};
auto to_sev =
     [out=std::move(sev_log)](std::string_view str, severity sev) mutable
     {
       out << sev << str;
     };

auto to_sev_wrapped = [alg=std::move(to_sev)](std::string_view str) mutable
{
  alg(str, severity::unknown);
};
```

Different body, same type signature

```
auto to_cout = [](std::string_view){ ... };

auto to_log = [out=std::move(log)](std::string_view) mutable { ... };

auto to_sev_wrapped = [alg=std::move(to_sev)](std::string_view) mutable {...};
```

Lambdas and their interfaces




What happens to lambdas?
Passed to Functions
Put in Containers
void accept(std::function f);
std::vector<std::function> v{};
template <typename F>
void accept(F f);

Wrapping in std::function

```
using sink_type = std::function<void(std::string_view)>;

std::vector<sink_type> sinks {};

sinks.push_back(to_cout);
sinks.push_back(std::move(to_log));
sinks.push_back(std::move(to_sev_wrapped));

for(auto&& sink : sinks)
  sink("Lambdas are sooo cool!");
```

Nope, not copy-constructible 😱

Wrapping in std::move_only_function

```
using sink_type = std::move_only_function<void(std::string_view)>;

std::vector<sink_type> sinks {};

sinks.push_back(to_cout);
sinks.push_back(std::move(to_log));
sinks.push_back(std::move(to_sev_wrapped));

for(auto&& sink : sinks)
  sink("Lambdas are sooo cool!");
```

Yes, both copy- and move-constrcuctible functors are fine!

Lambdas in containers, costs

1

```
auto l1 = [](long, long) -> long {...};
...
auto l16 = [](long, long) -> long {...};

l1(a, b);
...
l16(a,b);
```

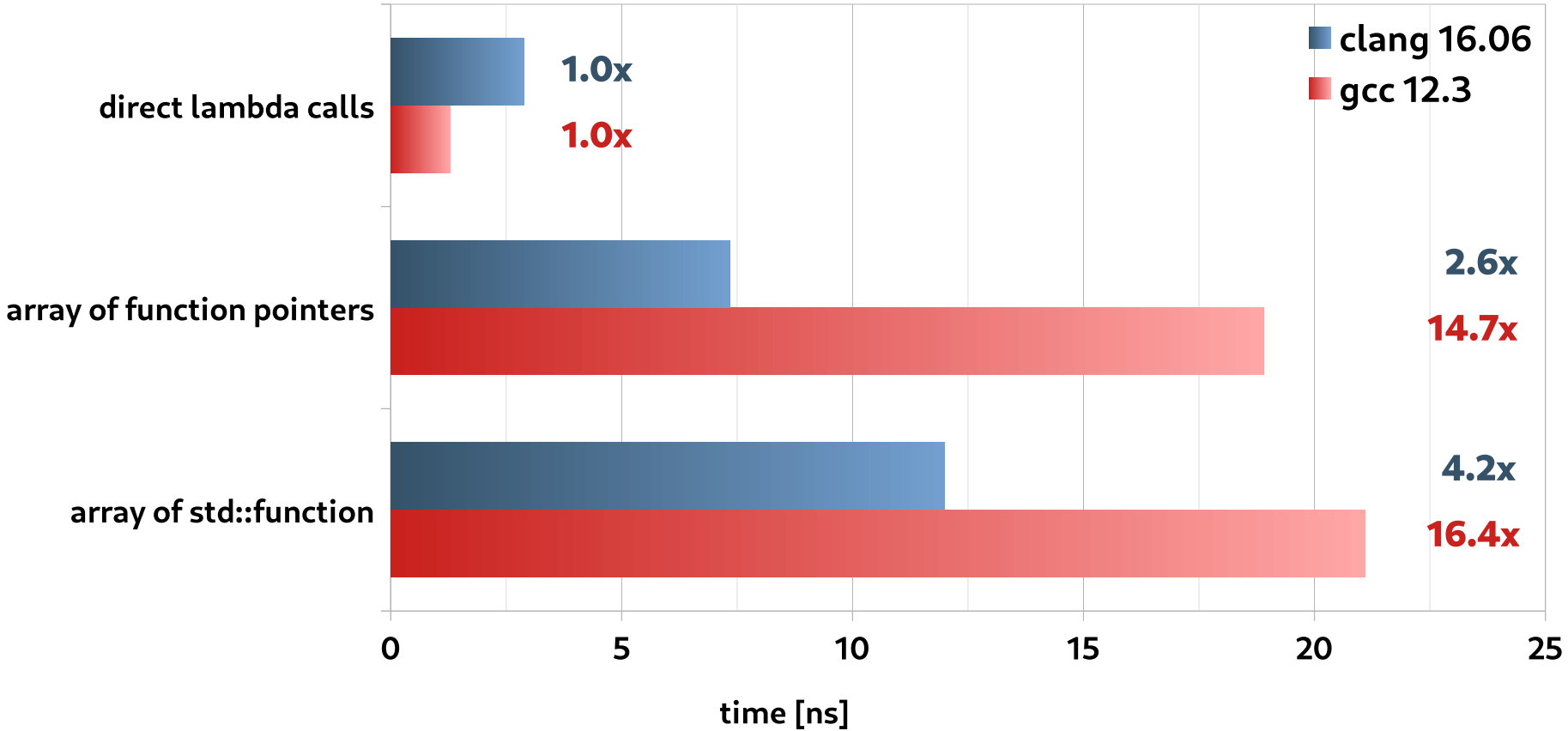
2

```
auto l1 = [](long, long) -> long {...};
...
auto l16 = [](long, long) -> long {...};

std::array<long(*)(long, long)> fs{l1,...,l16};
...
for (auto&& f: fs)
  f(a, b);
```

3

```
auto l1 = [](long, long) -> long {...};
...
auto l16 = [](long, long) -> long {...};

using func_t = std::function<long(long,long)>;
std::array<func_t> fs{l1,...,l16};

for (auto&& f: fs)
  f(a,b);
```

Lambdas in containers, costs



What makes it expensive?

- Array/ pointer indirection
- Type erasure
  - Heap memory
  - Virtual function call
  - Function pointer

Type erasure 101

```
template <typename R, typename...Args>
struct inplace_lambda
{
  std::byte storage_[16]{};

  using invoke_ptr = R(*)(std::byte*, Args...);
  using destruct_ptr = void(*)(std::byte*);

  invoke_ptr invoke_;
  destruct_ptr destruct_;

  constexpr R operator()(Args...args) {
    return invoke_(storage_, std::forward<Args>(args)...);
  }

  constexpr ~inplace_lambda() noexcept {
    destruct_(storage_);
  }
};
```

Type erasure 101

```
template <typename R, typename...Args>
struct inplace_lambda
{
  template <typename Lambda, typename Lambda_ = std::remove_reference_t<Lambda> >
  constexpr inplace_lambda(Lambda&& lambda) :
    invoke_ {
      [](std::byte* storage, Args...args) {
        return (reinterpret_cast<Lambda_&>(*storage))(std::forward<Args>(args)...);  }
    },

    destruct_{
      [](std::byte* storage){ (reinterpret_cast<Lambda_&>(*storage)).~Lambda_(); }
    }

    {
      new (storage_) Lambda_{std::forward<Lambda>(lambda)};
    }
};
```

Lambdas in containers, costs

1

```
auto l1 = [](long, long) -> long {...};
...
auto l16 = [](long, long) -> long {...};

l1(a, b);
...
l16(a,b);
```

2

```
auto l1 = [](long, long) -> long {...};
...
auto l16 = [](long, long) -> long {...};

std::array<long(*)(long, long)> fs{l1,...,l16};
...
for (auto&& f: fs)
  f(a, b);
```

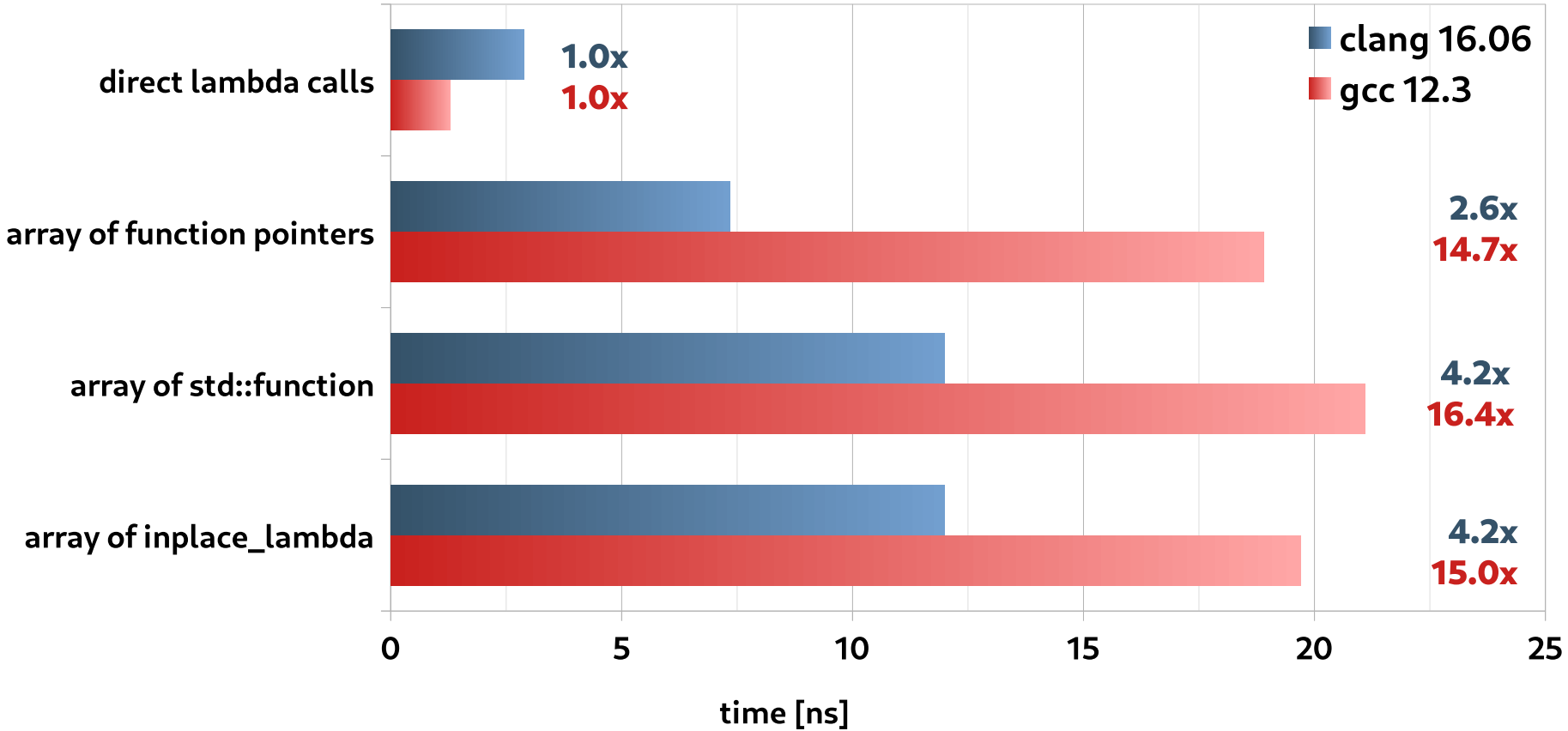
3

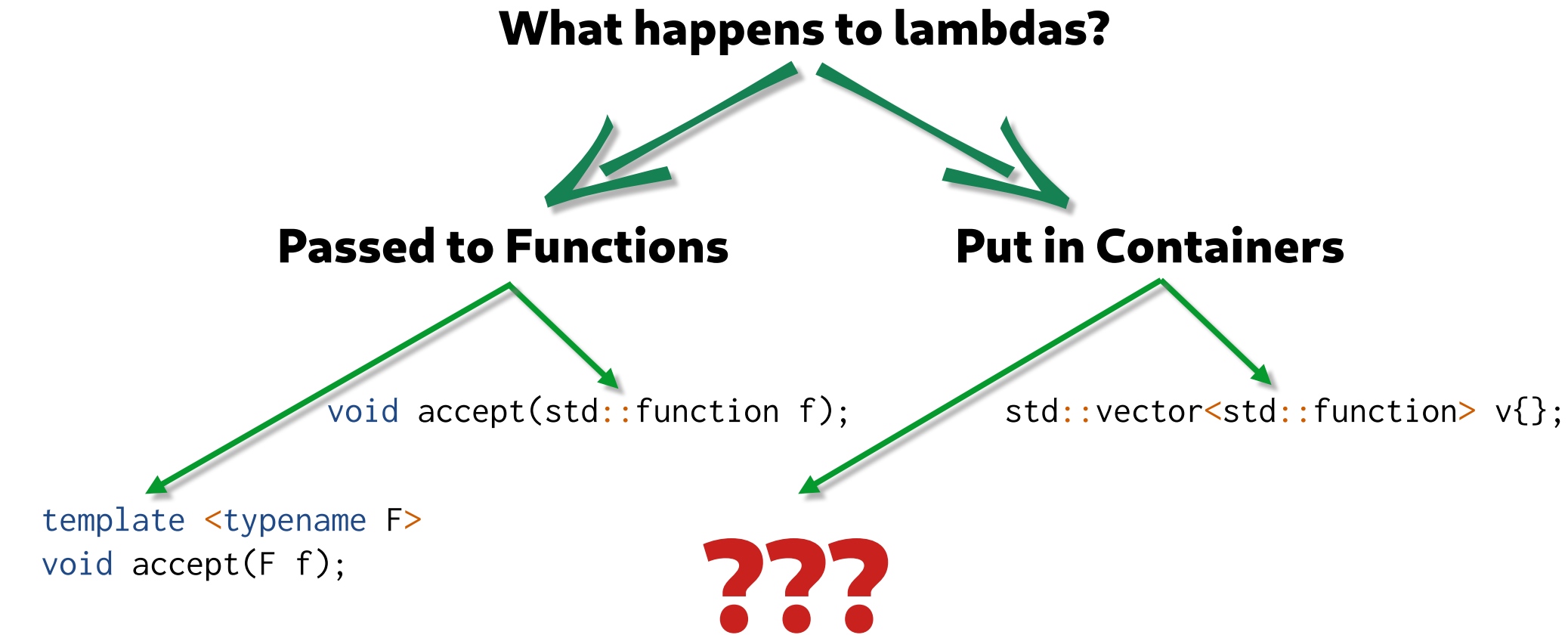
```
auto l1 = [](long, long) -> long {...};
...
auto l16 = [](long, long) -> long {...};

using func_t = std::function<long(long,long)>;
std::array<func_t> fs{l1,...,l16};

for (auto&& f: fs)
  f(a,b);
```

4

```
auto l1 = [](long, long) -> long {...};
...
auto l16 = [](long, long) -> long {...};

using func_t = inplace_lambda<long(long,long)>;
std::array<func_t> fs{l1,...,l16};

for (auto&& f: fs)
  f(a,b);
```

Lambdas in containers, costs




What happens to lambdas?
Passed to Functions
Put in Containers
void accept(std::function f);
std::vector<std::function> v{};
template <typename F>
void accept(F f);
???

Calling all the lambdas

```
functors funs {
               [](long a, long b){ return a + b; },
               [](long y, long z){ return y * z; },
               [](long i, long k){ return 42 * (k - i); }
              };


auto results = funs(19, 23);          // -> std::array{42, 437, 168}
```


Parameter packs
LIGHTYEAR
Parameter packs everywhere

Remember overloaded?

```
template<typename...Ts>
struct overloaded : Ts...
{
  using Ts::operator()...;
};

template<typename...Ts>
overloaded(Ts...) -> overloaded<Ts...>;

overloaded ov{
             [](std::string_view sv){ return std::cout << sv; },
             [](int a, int b){ return a + b; }
           };

ov("The answer is: ") << ov(19, 23);
```

Overloaded inherits from Ts...

Types and packs...

```
struct packs_everywhere
{
  template <typename Lambda>
  auto function(Lambda&& lambda);

  template <typename...Lambdas>
  auto function(Lambdas&&...lambdas);
};
```

A parameter pack

Types and packs...

```
template <typename R, typename...Args>
struct packs_everywhere
{
  template <closure_of<R(Args...) Lambda>
  auto function(Lambda&& lambda);

  template <closure_of<R(Args...)>...Lambdas>
  auto function(Lambdas&&...lambdas);
};

template <typename Lambda, typename R_Args>
concept closure_of =
  std::is_class_v<std::remove_cvref_t<Lambda>> &&
  std::is_member_function_pointer_v<decltype(&std::remove_cvref_t<Lambda>::operator())> &&
  std::is_same_v<R_Args, lambda_type_t<Lambda>>;
```

Function type like this one:

R func(Args...)

Let's build a lambda container

```
template <typename...>
struct functors;

template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{
  template <closure_of<R(Args...)...Ls>
  constexpr functors(Ls&&...lambdas):
    Lambdas{ std::forward<Ls>(lambdas) }...
    {}

  constexpr std::size_t size() const { return sizeof...(Lambdas); }

};
```

Let's build a lambda container

```
                                head_t<int, double, std::string_view> -> int

template <typename...Lambdas>
functors(Lambdas...) -> functors< lambda_type_t< head_t<Lambdas...> >, Lambdas...>;

                        auto lambda = [](long n){ return 42.0 * n; };

                        lambda_type_t<decltype(lambda)> -> double(long)
```

Let's build a lambda container

```
auto to_cout = [](std::string_view) mutable { ... };

auto to_log = [out=std::move(log)](std::string_view) mutable { ... };

functors two_funs{
                    to_cout,
                    std::move(to_log),
                  };
```

```
decltype(two_funs)

==

functors< void(std::string_view), decltype(to_cout), decltype(to_log)>
```

Lambda container, push_back

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{
  template <closure_of<R(Args...)...MoreLambdas>
  constexpr functors<R(Args...), Lambdas..., std::remove_cvref_t<MoreLambdas>...>
  push_back(MoreLambdas&&...more_lambdas) &
  {
    return {
              static_cast<Lambdas const&>(*this)...,
              std::forward<MoreLambdas>(more_lambdas)...
            };
  }
};
```

Lambda container, push_back

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{
  template <closure_of<R(Args...)...MoreLambdas>
  constexpr functors<R(Args...), Lambdas..., std::remove_cvref_t<MoreLambdas>...>
  push_back(MoreLambdas&&...more_lambdas) &&
  {
    return {
              std::move(static_cast<Lambdas&>(*this))...,
              std::forward<MoreLambdas>(more_lambdas)...
            };
  }
};
```

Lambda container, push_back

```
auto to_cout = [](std::string_view) mutable { ... };
auto to_log = [out=std::move(log)](std::string_view) mutable { ... };
auto to_sev_wrapped = [alg=std::move(to_sev)](std::string_view) mutable {...};

functors two_funs{
               to_cout,
               std::move(to_log),
             };

auto three_funs = std::move(two_funs).push_back( std::move(to_sev_wrapped) );
```

Lambda container, push_back

```
auto to_cout = [](std::string_view) mutable { ... };
auto to_log = [out=std::move(log)](std::string_view) mutable { ... };
auto to_sev_wrapped = [alg=std::move(to_sev)](std::string_view) mutable {...};

functors two_funs{
               to_cout,
               std::move(to_log),
             };

auto three_funs = std::move(two_funs).push_back( std::move(to_sev_wrapped) );

three_funs("Hello, Lambdas!");
```

Calling all the lambdas

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{

  using Lambdas::operator()...;

};
```

Calling all the lambdas

```
template <typename R, typename...Args, typename   Lambda >
struct functors<R(Args...), Lambda    >  : private Lambda
{

  constexpr void operator()(std::string_view str)

  {


        static_cast<Lambda &>(*this)

  }

};
```

Calling all the lambdas

```
template <typename R, typename...Args, typename   Lambda >
struct functors<R(Args...), Lambda    >  : private Lambda
{

  constexpr void operator()(std::string_view str)

  {


        static_cast<Lambda &>(*this)(str)

  }

};
```

Calling all the lambdas

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{

  constexpr void operator()(std::string_view str)

  {


        static_cast<Lambdas&>(*this)(str)

  }

};
```

Calling all the lambdas

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{

  constexpr void operator()(std::string_view str)

  {


      ( static_cast<Lambdas&>(*this)(str), ... );

  }

};
```

Calling all the lambdas

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{

  constexpr void operator()(Args...args)

  {


      ( static_cast<Lambdas&>(*this)(std::forward<Args>(args)...), ... );

  }

};
```

Calling all the lambdas

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{

  constexpr void operator()(Args...args)
    requires (std::is_same_v<void, R>)
  {


      ( static_cast<Lambdas&>(*this)(std::forward<Args>(args)...), ... );

  }

};
```

Calling all the lambdas

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{

  constexpr auto operator()(Args...args)
    requires (!std::is_same_v<void, R>)
  {
    return std::array
      {
        static_cast<Lambdas&>(*this)(std::forward<Args>(args)...)...
      };
  }

};
```

Calling all the lambdas

```
functors funs {
               [](long a, long b){ return a + b; },
               [](long y, long z){ return y * z; },
               [](long i, long k){ return 42 * (k - i); }
              };
```

```
auto results = funs(19, 23);          // -> std::array{42, 437, 168}
```

N-th lambda in the pack

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{

  template <auto I>
  constexpr auto& get()
    requires ( I < sizeof...(Lambdas) )
  {
    return static_cast< nth_type_t<I, Lambdas...>& >(*this);
  }

};
```

```
nth_type_t<I, Types...>

nth_type_t<2, int, std::string, double> -> std::string
```

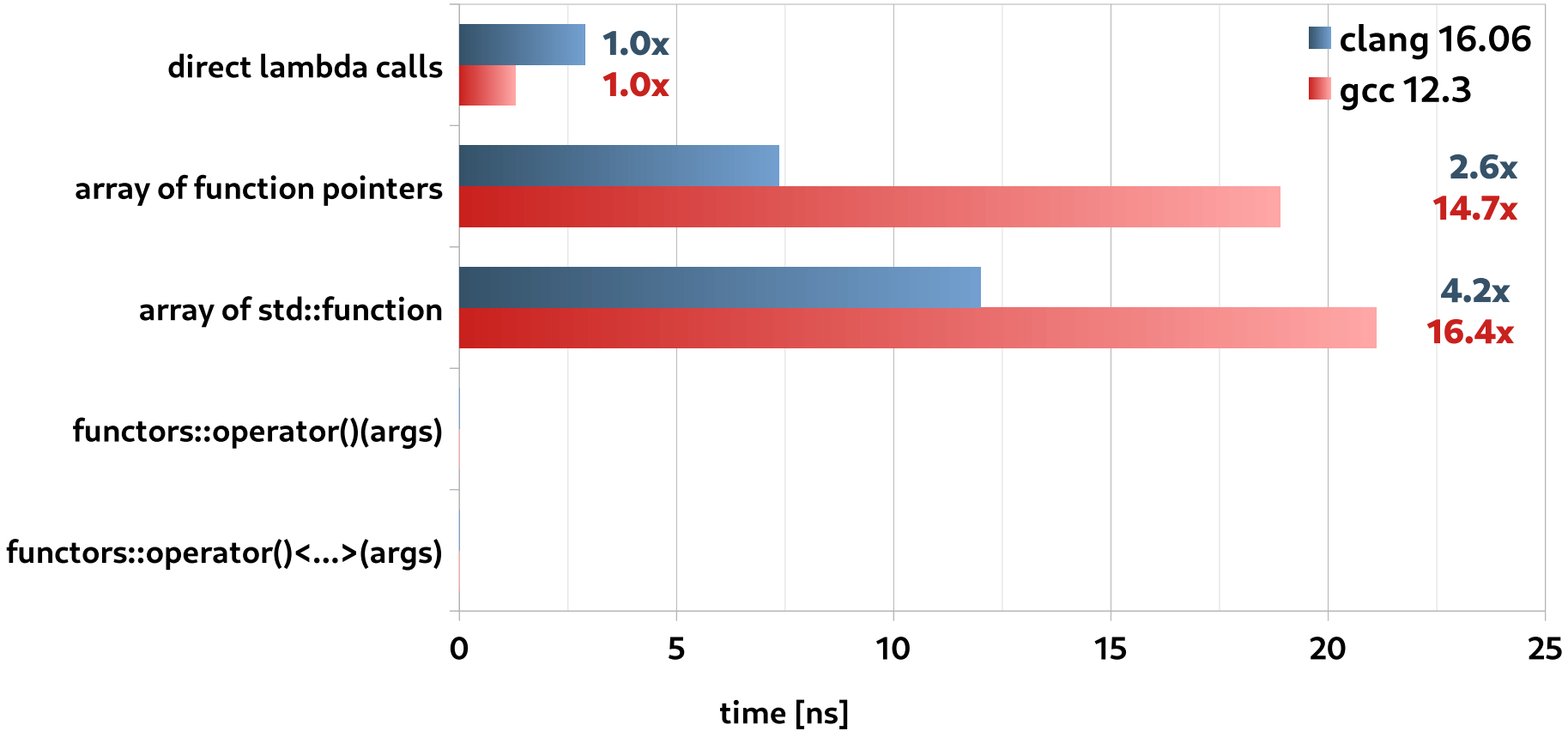
Calling some lambdas

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{
  template <auto I, auto...Is>
  constexpr auto operator()(Args...args) requires (!std::is_same_v<void, R>)
  {
    if constexpr (sizeof...(Is) == 0)
    {
      return get<I>()(std::forward<Args>(args)...);
    }
    else
    {
      return std::array {  get<I>()(std::forward<Args>(args)...),
                           get<Is>()(std::forward<Args>(args)...)...  };
    }
  }
};
```

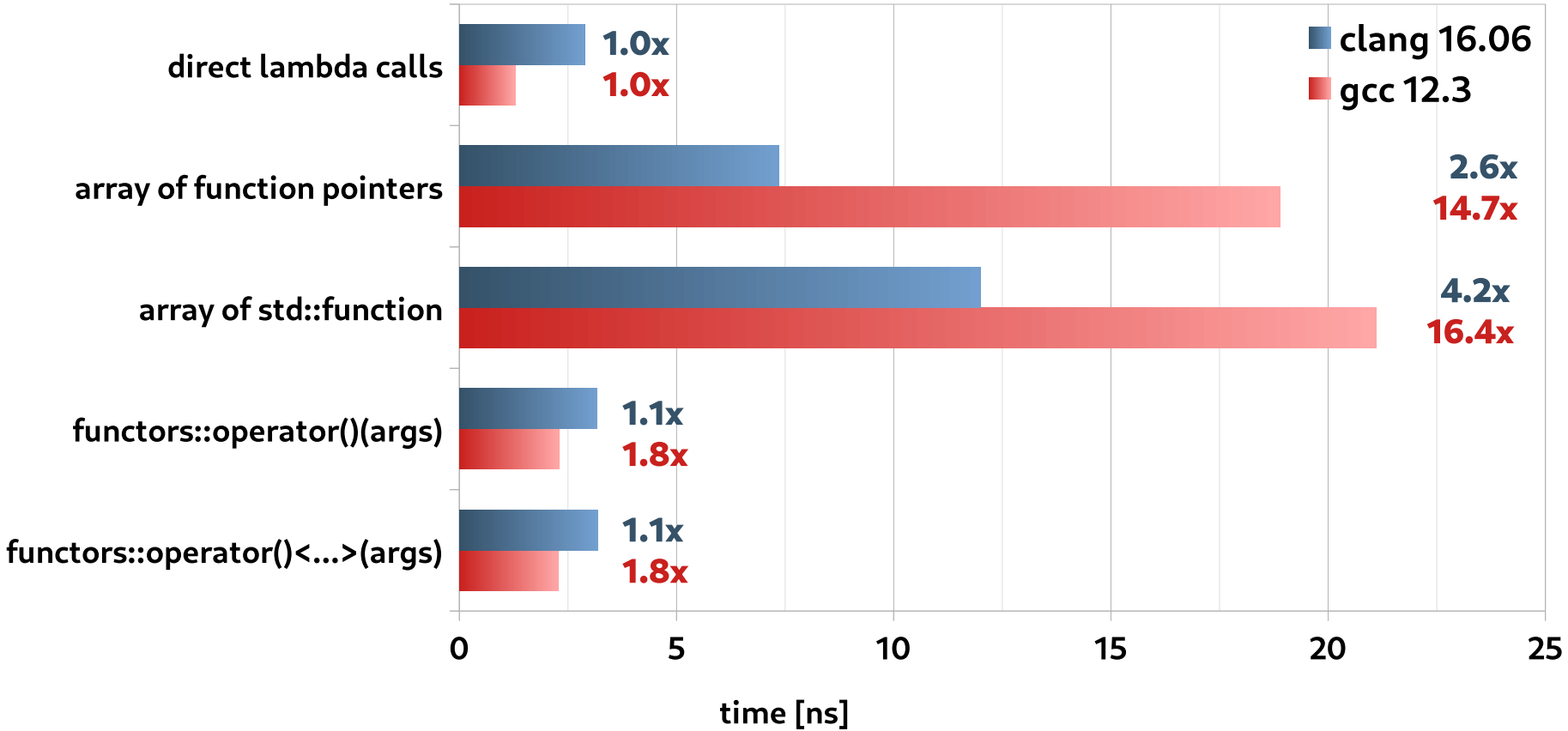
Calling some lambdas

```
functors funs {
               [](long a, long b){ return a + b; },
               [](long y, long z){ return y * z; },
               [](long i, long k){ return 42 * (k - i); }
              };


auto results = funs(19, 23);          // -> std::array{42, 437, 168}

auto results = funs<0, 2>(19, 23);    // -> std::array{42, 168}
```

Lambdas in containers, costs



Lambdas in containers, costs



Calling lambdas

```
functors funs {
               [](long a, long b){ return a + b; },
               [](long y, long z){ return y * z; },
               [](long i, long k){ return 42 * (k - i); }
              };


auto results = funs(19, 23);          // -> std::array{42, 437, 168}

auto results = funs<0, 2>(19, 23);    // -> std::array{42, 168}
```

Calling lambdas dynamically

```
functors funs {
               [](long a, long b){ return a + b; },
               [](long y, long z){ return y * z; },
               [](long i, long k){ return 42 * (k - i); }
              };

long a{19}, b{23};

for (auto i{0u}; i < funs.size(); ++i)
{

  funs[i](a, b);

  ++a; --b;

}
```

Calling lambdas dynamically

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{
  constexpr auto operator[](std::integral auto i) requires (!std::is_same_v<void, R>)
  {


    return [this, i](Args...args)
           {
             return invoke_map[i]( *this, std::forward<Args>(args)... );
           };

  }
};
```

Calling lambdas dynamically

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{
  constexpr static
  std::array<R(*)(functors&, Args...), sizeof...(Lambdas)> invoke_map
  {
    []<auto...Is>(std::index_sequence<Is...>)
    {
      return std::array<R(*)(functors&, Args...), sizeof...(Is)>
      {
        [](functors& self, Args...args)
        {
          return static_cast<nth_type_t<Is, Lambdas...>&>(self).operator()
                                                  (std::forward<Args>(args)...);
        }...
      };
    }(std::index_sequence_for<Lambdas...>{})
  };
};
```

Calling lambdas dynamically

```
functors funs {
               [](long a, long b){ return a + b; },
               [](long y, long z){ return y * z; },
               [](long i, long k){ return 42 * (k - i); }
              };


auto results = funs(19, 23);          // -> std::array{42, 437, 168}

auto results = funs<0, 2>(19, 23);    // -> std::array{42, 168}

auto result = funs[1](19, 23);        // -> 437

auto results = funs[0, 2](19, 23);    // -> std::array{42, 168}
```

Calling lambdas dynamically

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{
  constexpr auto operator[](auto i, auto...idx) requires (!std::is_same_v<void, R>)
  {
    if constexpr (sizeof...(idx) == 0)
    {
      return [this, i](Args...args)
             {
               return invoke_map[i]( *this, std::forward<Args>(args)... );
             };
    }
    else { /*...*/ }
  }
};
```

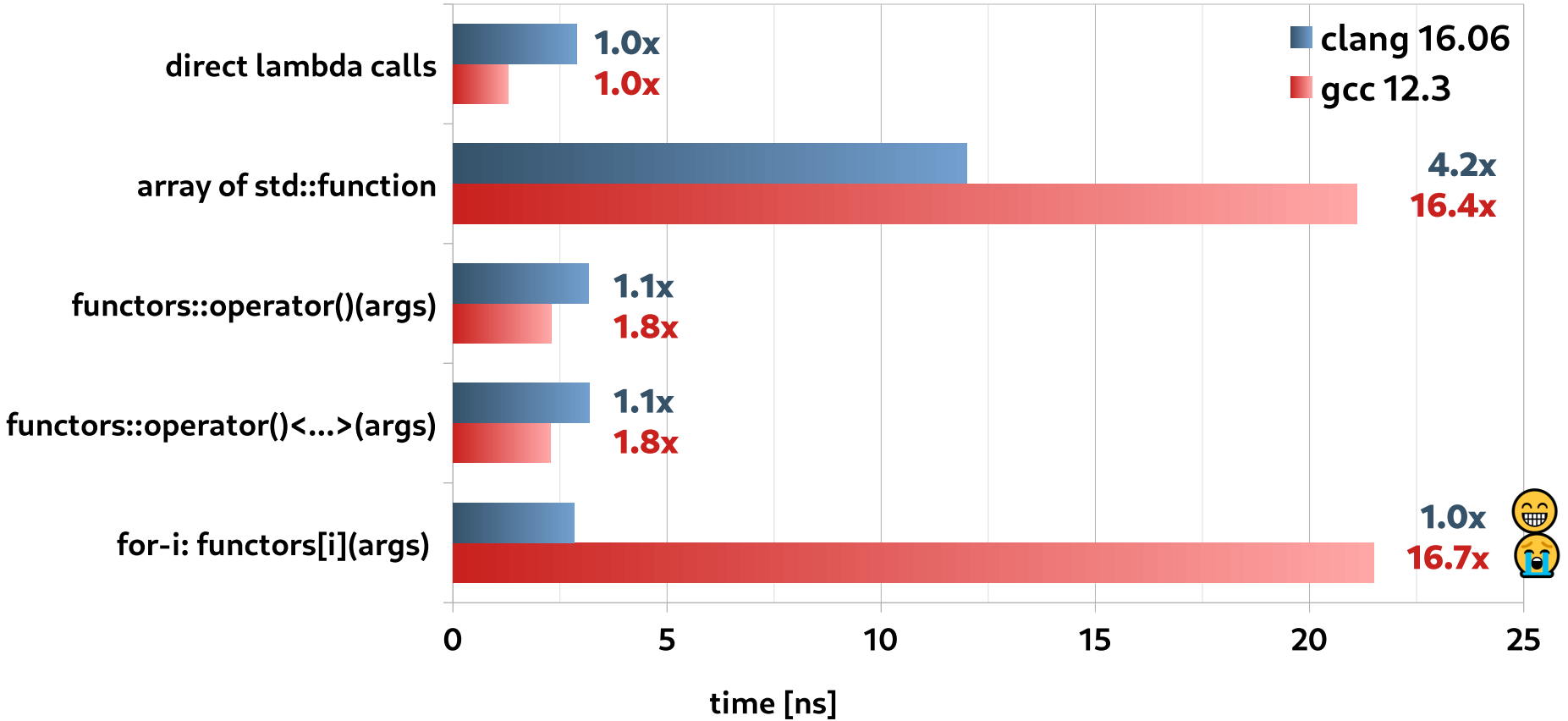
Calling lambdas dynamically

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{
  constexpr auto operator[](auto i, auto...idx) requires (!std::is_same_v<void, R>)
  {
    if constexpr (sizeof...(idx) == 0) {/*...*/}
    else
    {
      return [this, i, idx...](Args...args)
             {
               return std::array
                      {
                        invoke_map[ i ](*this, std::forward<Args>(args)...),
                        invoke_map[idx](*this, std::forward<Args>(args)...)...
                      };
             };
    }
  }
};
```

Calling all the lambdas

```
functors funs {
               [](long a, long b){ return a + b; },
               [](long y, long z){ return y * z; },
               [](long i, long k){ return 42 * (k - i); }
              };


auto results = funs(19, 23);          // -> std::array{42, 437, 168}

auto results = funs<0, 2>(19, 23);    // -> std::array{42, 168}

auto result = funs[1](19, 23);        // -> 437

auto results = funs[0, 2](19, 23);    // -> std::array{42, 168}
```

Lambdas in containers, costs



Making compiler think deeper

```
functors funs {
               [](long a, long b){ return a + b; },
               [](long y, long z){ return y * z; },
               [](long i, long k){ return 42 * (k - i); }
              };


auto invokers = funs.get_invokers<0, 2>();

auto results = invokers(19, 23);   // -> std::array{42, 168}

auto invokers = funs.get_invokers(0, 2);

auto results = invokers(19, 23);   // -> std::array{42, 168}
```

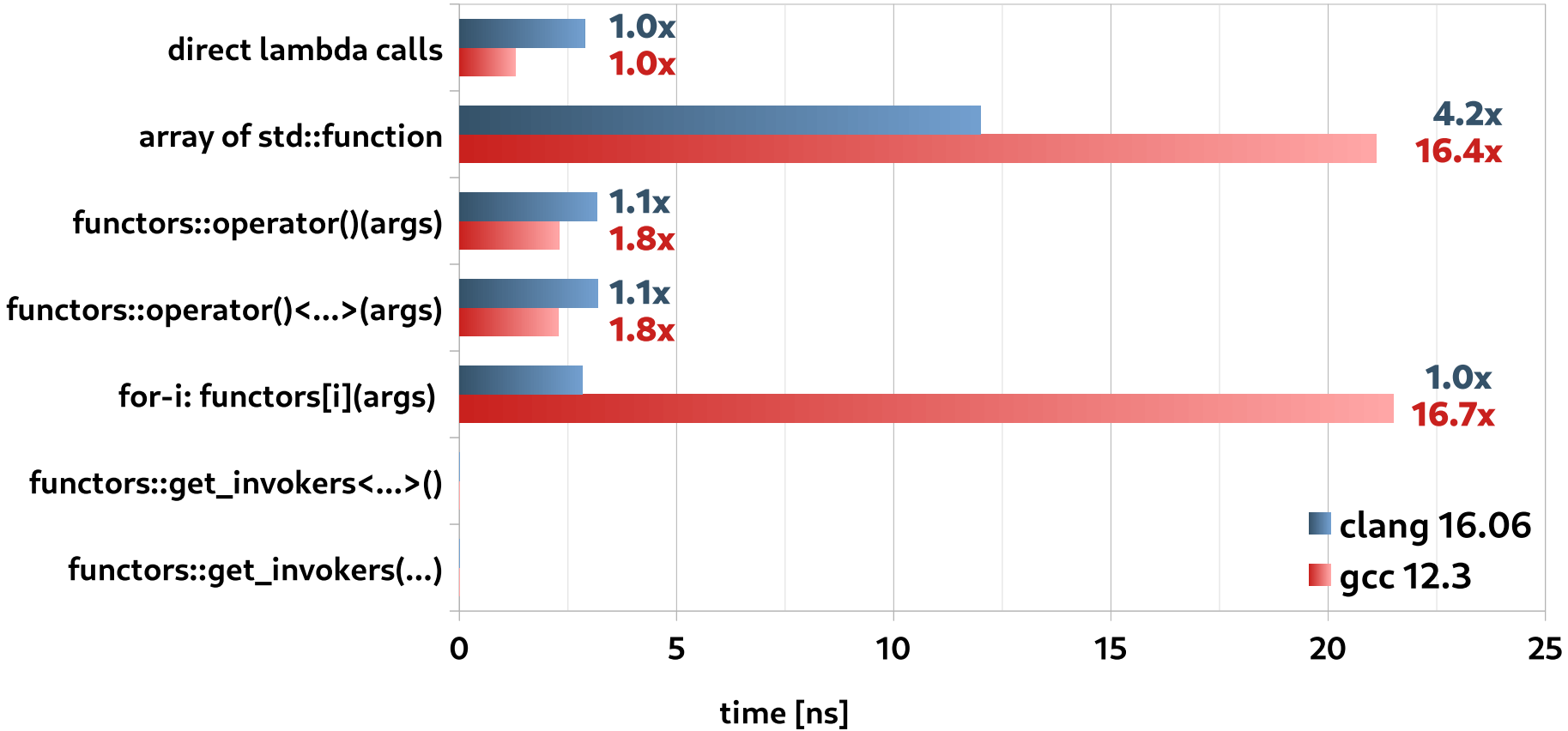
Calling through invokers

```
template <typename R, typename...Args, typename...Lambdas>
struct functors<R(Args...), Lambdas...>  : private Lambdas...
{

  template <auto...Is>
  constexpr auto get_invokers() {
    return [this](Args...args){
      return std::array{ get<Is>()(std::forward<Args>(args)...)... };
    };
  }

  constexpr auto get_invokers(std::integral auto...idx) {
    return [this, idx...](Args...args){
        return std::array{ invoke_map[idx](*this, std::forward<Args>(args)...)... };
    };
  }

};
```

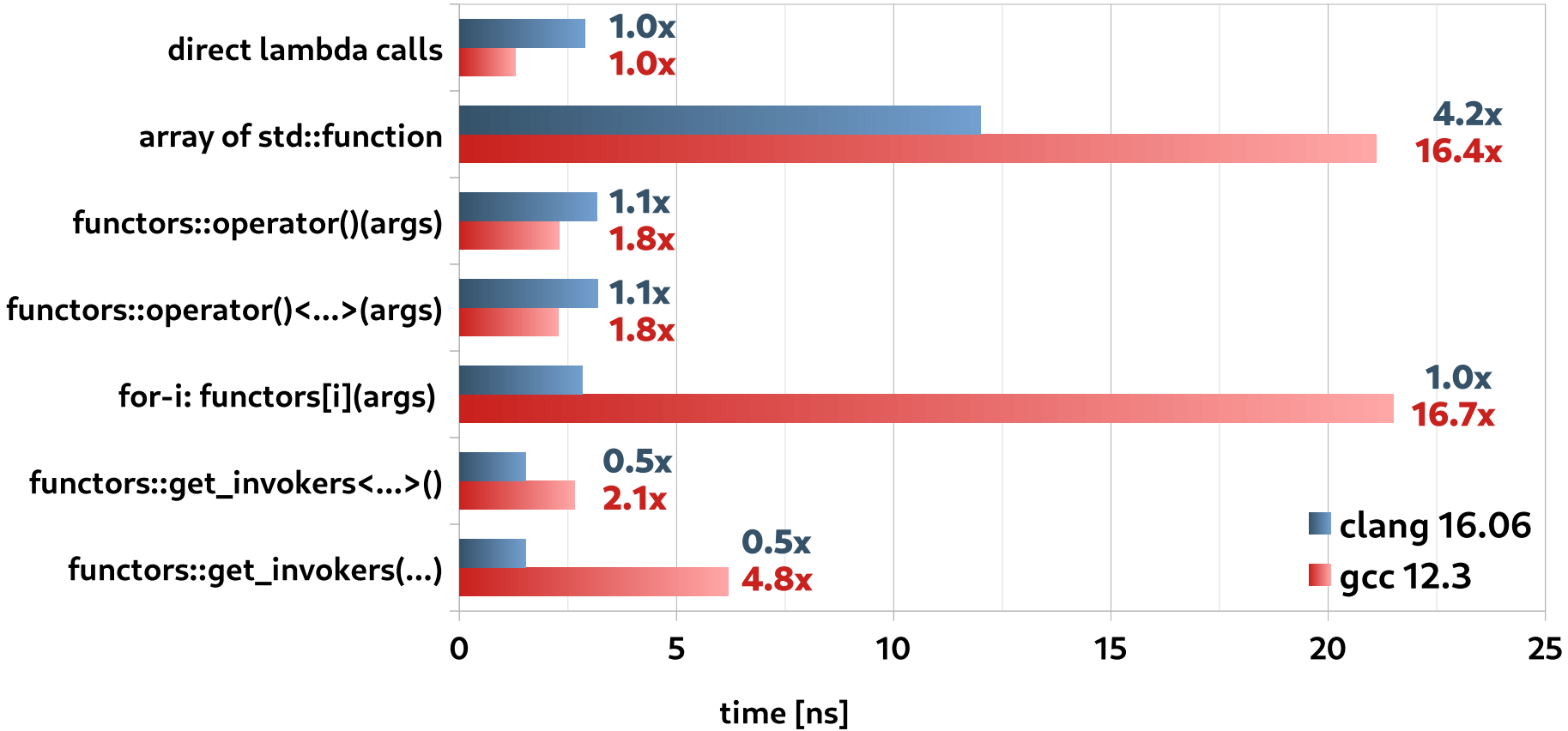
Calling through invokers

```
functors funs {
               [](long a, long b){ return a + b; },
               [](long y, long z){ return y * z; },
               [](long i, long k){ return 42 * (k - i); }
              };


auto invokers = funs.get_invokers<0, 2>();

auto results = invokers(19, 23);    // -> std::array{42, 168}

auto invokers = funs.get_invokers(0, 2);

auto results = invokers(19, 23);    // -> std::array{42, 168}
```

Lambdas in containers, costs



Lambdas in containers, costs



Passing around, the costs

	gcc 12.3	clang 16.06
direct lambda calls	1.0x	1.0x
array of function pointers	14.7x	2.6x
array of std::function	16.4x	4.2x
array of inplace_lambda	15.3x	4.2x
functors::operator()(args)	1.8x	1.1x
functors::operator()<...>(args)	1.8x	1.1x
for-i: functors[i](args)	16.7x	1.0x
functors::get_invokers<...>	2.1x	0.5x
functors::get_invokers(...)	4.8x	0.5x

```
constexpr auto get_invokers(std::integral auto...idx) {
  return [this, idx...](Args...args){
    return std::array{ invoke_map[idx](*this,
std::forward<Args>(args)...)... };
  };
}
```

```
template <typename Sink>
void accept(Sink sink, std::string_view
{
  sink(str);
}
```

Thank you!

```
constexpr auto operator[](auto i, auto...idx)
{
  if constexpr (sizeof...(idx) == 0)
  {
    return [this, i](Args...args)
           {
             return invoke_map[i]( *this, std::forward<Args>(args)... );
           };
  }
  else { /*...*/ }
}
```


zaldawid@gmail.com
github.com/zaldawid
linkedin.com/in/dawidzalewski