

Higher-order functions in Modern C++

Existing techniques and `function_ref`

Vittorio Romeo

vittorioromeo.info

vittorio.romeo@outlook.com

vromeo5@bloomberg.net

[@supahvee1234](#)

CppCon 2019

2019/09/20

Aurora, CO

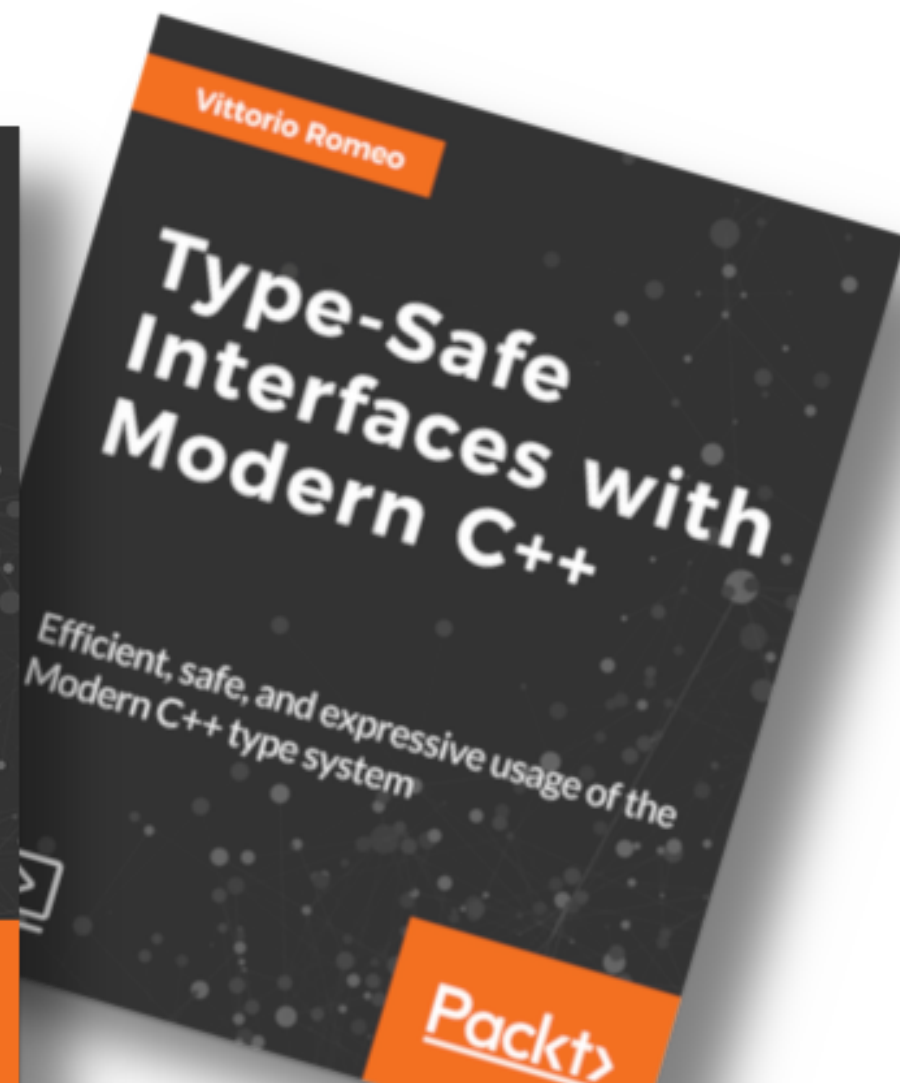
Bloomberg

(C) 2019 Bloomberg Finance L.P. All rights reserved.

About me



Software Engineer @ Bloomberg L.P.



Mastering C++ Standard Library Features

Harness the power of the C++ STL and make full use of its components



4.3 (122 ratings)

1,077 students enrolled



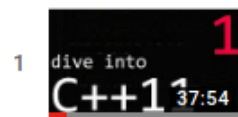
Dive into C++11/14

8 videos • 24,566 views • Last updated on 29 Nov 2015



Vittorio Romeo

EDIT



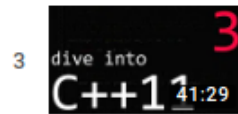
Dive into C++11 - [1] - Arkanoid clone in 160~ lines of code (SFML 2.1)

Vittorio Romeo



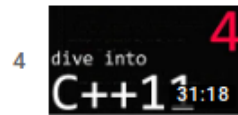
Dive into C++11 - [2] - Frametime, FPS, constexpr, uniform initialization

Vittorio Romeo



Dive into C++11 - [3] - Automatic lifetime, pointers, dynamic allocation

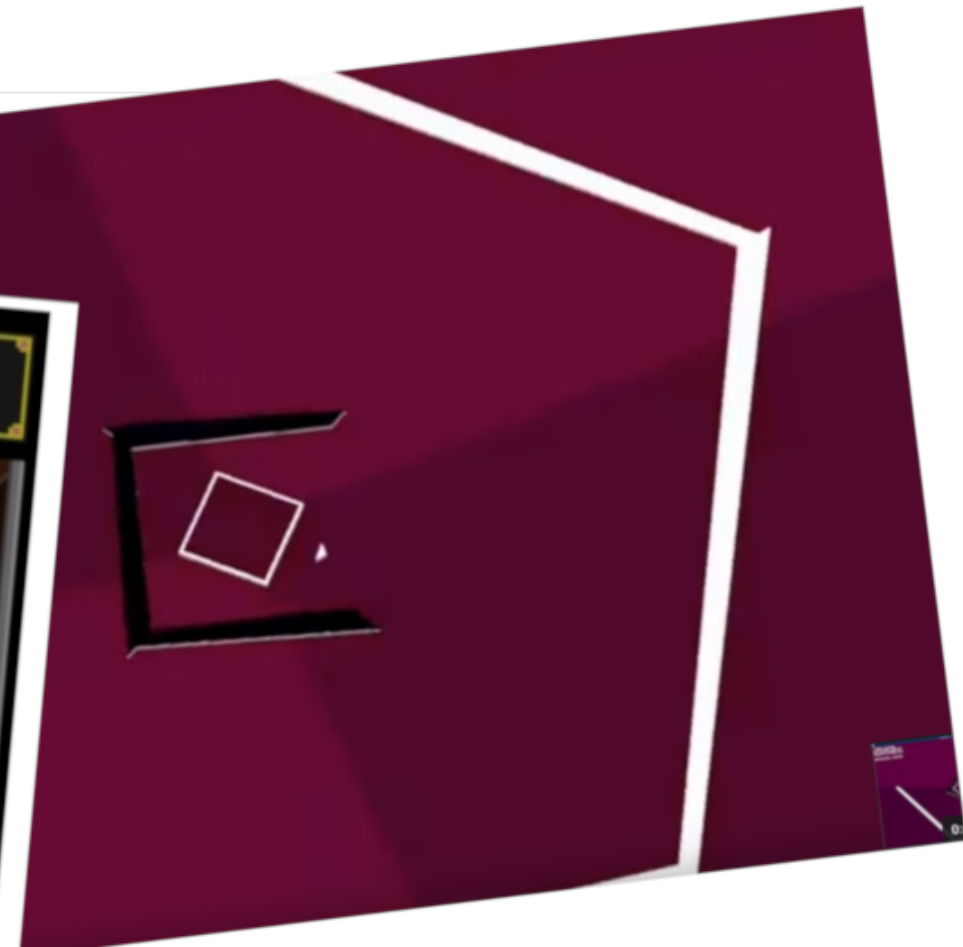
Vittorio Romeo



Dive into C++11 - [4] - Smart pointers

Vittorio Romeo

TIME: 17.69
OFFICIAL MODE
SWAP ENABLED
CONGRUENCE



about me

Hello! My name is Vittorio.

I'm a modern C++ enthusiast who loves to share his knowledge by creating video tutorials and participating to conferences. I have a BS in Computer Science from the University of ...

Document number	P0792R3
Date	2018-10-07
Reply-to	Vittorio Romeo <vittorio.romeo@outlook.com>
Audience	Library Working Group (LWG)
Project	ISO JTC1/SC22/WG21: Programming Language C++

function_ref: a non-owning reference to a Callable

Abstract

This paper proposes the addition of function_ref<R> owning references to Callable objects.

Changelog and polls

Concept-constrained auto

Abstract

home page

Welcome to my blog.

compile-time iteration with C++20 lambdas

16 april 2018

c++ c++20 lambda tutorial

In one of my previous articles, "[compile-time repeat & noexcept-correctness](#)", I have covered the design and implementation of a simple `repeat<n>(f)` function that, when invoked, expands to `n` calls to `f` during compilation. E.g.

```
repeat<4>([&f, &data] { f(data); });
```

```
);  
);  
);  
);
```

rm of *compile-time iteration*. When writing generic code, n order to express the following actions:

[... read more](#)

holder type for variables. The primary goal is to



Vittorio Romeo

SuperV1234

0000;

📍 London, UK

✉ vittorio.romeo@outlook.com

🌐 <http://vittorioromeo.info>

Edit

Organizations



Overview

Repositories 153

Stars 788

Followers 627

Following 94

Pinned repositories

≡ [ecst](#)

[WIP] Experimental C++14 multithreaded compile-time entity-component-system library.

● C++ ★ 333 🍴 32

≡ [Tutorials](#)

Repository for my YouTube tutorials.

● C++ ★ 195 🍴 46

≡ [SSVOpenHexagon](#)

C++14 FOSS clone of "Super Hexagon". SSVStart, SSVEntitySystem, SSVLuaWr, SSVMenuSystem, JSONcpp, SFML2.0. F customizable game files, a soundtrack.

● C++ ★ 298 🍴 33

196 contributions in the last year

Feb Mar Apr May

Customize your pinned repositories

≡ [vr_core](#)

- 5 Meeting C++ Lightning Talks - Vittorio Romeo - 'static_if' in C++14
Meeting Cpp 9:36
 - 6 ++it Meetup Firenze - Introduzione al Game Development in C++14 (Vittorio Romeo)
Italian C++ Community 54:41
 - 7 Vittorio Romeo: Implementation of a multithreaded compile-time ECS in C++14
BoostCon 1:32:35
 - 8 Vittorio Romeo: Implementing 'static' control flow in C++14
BoostCon 1:15:58
 - 9 CppCon 2016: Vittorio Romeo "Implementing 'static' control flow in C++14"
CppCon 1:03:03
 - 10 Implementing 'static' control flow in C++14 - Vittorio Romeo - Meeting C++ 2016
Meeting Cpp 1:02:10
 - 11 Implementation of a multithreaded compile-time ECS in C++14 - Vittorio Romeo
Meeting Cpp 1:07:57
 - 12 Implementing variant Visitation Using Lambdas - Vittorio Romeo [ACCU 2017]
ACCU Conference 1:30:56
- C++Now 2017: Vittorio Romeo "Implementing 'variant' visitation using lambdas"
BoostCon

My presentations/talks

17 videos • 844 views • Last updated on Nov 10, 2018



Vittorio Romeo

EDIT



57,864 • 17 • 156 • 299

Introduction

- Higher-order functions
 - What they are
 - Use cases and implementation techniques
- `function_ref`
 - Motivation
 - Specification and usage examples
 - Implementation
 - Benchmarks
 - Design decisions and controversies

- This is *not* a talk on functional programming
- We are going to look at:
 - *Practical* every day uses of higher-order functions
 - Existing "functional" facilities in the language
 - Design and implementation of a ISO C++20 proposal

- You are somewhat familiar with:
 - Lambda expressions
 - Templates
 - Modern C++ features
- Do not hesitate to ask questions

Higher order functions

In mathematics and computer science, a higher-order function is a function that does at least one of the following:

- **takes one or more functions as arguments** (i.e. procedural parameters),
- **returns a function as its result.**

- [Wikipedia](#)


```
template <typename F>
void call_twice(F&& f)
{
    f();
    f();
}

call_twice([]{ std::cout << "hello"; });
```

- Takes a *FunctionObject* as an argument
- Implementation technique: *template parameter*

```
auto greater_than(int threshold)
{
    return [threshold](int x)
    {
        return x > threshold;
    };
}
```

```
std::vector<int> v{0, 4, 1, 11, 5, 9};
assert(std::count_if(v.begin(), v.end(), greater_than(5)) == 2);
```

(on wandbox.org)

- Returns a `FunctionObject` invocable with an `int`
- Implementation technique: *closure* + `auto` return type

Do we have any higher-order function in the C++ Standard?

Do we have any higher-order function in the C Standard?

- `std::qsort` , `std::bsearch`
- `std::atexit` , `std::at_quick_exit`
- `std::signal`

std::signal

Defined in header `<csignal>`

```
/*signal-handler*/ signal(int sig, /*signal-handler*/ handler); (1)
```

```
extern "C" using /*signal-handler*/ = void(int); // exposition-only (2)
```

Sets the handler for signal `sig`. The signal handler can be set so that default handling will occur, signal is ignored, or a user-defined function is called.

Parameters

sig - the signal to set the signal handler to. It can be an implementation-defined value or one of the following values:

SIGABRT
SIGFPE
SIGILL defines signal types
SIGINT (macro constant)
SIGSEGV
SIGTERM

handler - the signal handler. This must be one of the following:

- **SIG_DFL** macro. The signal handler is set to default signal handler.
- **SIG_IGN** macro. The signal is ignored.
- pointer to a function. The signature of the function must be equivalent to the following:

```
extern "C" void fun(int sig);
```

```
#include <csignal>

int main()
{
    std::signal(SIGINT, [](int signal_num)
    {
        std::cout << "signal: " << signal_num << '\n';
    });
}
```

(on godbolt.org)

- *Lambda expressions* work great with higher-order functions
- Stateless *closures* are implicitly convertible to *function pointers*

- `std::set_terminate`
- `std::visit` , `std::apply` , `std::invoke`
- `std::bind` , `std::bind_front` (C++20)
- `<numeric>` and `<algorithm>`
- ...

```
std::vector<entity> entities;

entities.erase(
    std::remove_if(entities.begin(),
                   entities.end(),
                   [](const entity& e){ return !e._active; }),
    entities.end()
);
```

(on godbolt.org)

- "Erase-remove idiom":
 - Moves elements to keep towards the beginning of the range
 - Relative order of elements is preserved


```
using event = std::variant<connect, disconnect, heartbeat>;

void process(event&& e)
{
    std::visit(
        overload([ ](connect)    { std::cout << "process connect\n"; },
                  [ ](disconnect){ std::cout << "process disconnect\n"; },
                  [ ](heartbeat) { std::cout << "process heartbeat\n"; }),
        e);
}

process(event{connect{}});
process(event{heartbeat{}});
process(event{disconnect{}});
```

(on wandbox.org)

- *"Implementing variant visitation using lambdas"* @ [ACCU 2017](#), [C++Now 2017](#)

- Avoiding repetition
- Inversion of control flow
- Asynchronicity
- Compile-time metaprogramming
- ...

- Code repetition leads to bugs and maintenance overhead
- Sometimes, it is trivial to avoid

```
void test_routing(context& ctx)
{
    const auto machine0 = ctx.reserve_port("127.0.0.1");
    const auto machine1 = ctx.reserve_port("127.0.0.1");
    const auto machine2 = ctx.reserve_port("127.0.0.1");
}
```



```
void test_routing(context& ctx)
{
    const auto get_port = [&]{ return ctx.reserve_port("127.0.0.1"); };
    const auto machine0 = get_port();
    const auto machine1 = get_port();
    const auto machine2 = get_port();
}
```

- Other times it can be more complicated

```
void widget::update()  
{  
    for (auto& c : this→_children)  
        if (c→visible())  
            c→recalculate_focus();  
  
    for (auto& c : this→_children)  
        if (c→visible())  
            c→recalculate_bounds();  
  
    for (auto& c : this→_children)  
        if (c→visible())  
            c→update();  
}
```



```
void widget::update()  
{  
    const auto for_visible_children = [this](auto f)  
    {  
        for (auto& c : this->_children)  
            if(c->visible())  
                std::invoke(f, *c);  
    };  
  
    for_visible_children(&widget::recalculate_focus);  
    for_visible_children(&widget::recalculate_bounds);  
    for_visible_children(&widget::update);  
}
```

(on wandbox.org)

- Pass an *action/predicate* to a function which deals with the control flow
- Separate *what happens* from *how it happens*
- *Example*: C++17 parallel algorithms

```
struct physics_component
{
    vec2f _pos, _vel, _acc;
};

std::vector<physics_component> components{ /* ... */ };

std::for_each(std::execution::par_unseq,
              components.begin(),
              components.end(),
              [](auto& c)
              {
                  c._vel += c._acc;
                  c._pos += c._vel;
              });
```

- Decoupling *control flow* from the desired *action*
 - Can be reused & tested separately
- *Example*: printing a comma-separated list of elements

- Initial version

```
template <typename T>
void print(const std::vector<T>& v)
{
    if(std::empty(v)) { return; }
    std::cout << *v.begin();

    for(auto it = std::next(v.begin()); it != v.end(); ++it)
    {
        std::cout << ", ";
        std::cout << *it;
    }
}
```

(on wandbox.org)

- Identify the structure

```
template <typename T>
void print(const std::vector<T>& v)
{
    if(std::empty(v)) { return; }
    /* action */

    for(auto it = std::next(v.begin()); it != v.end(); ++it)
    {
        /* separation */
        /* action */
    }
}
```

- Create an abstraction

```
template <typename Range, typename F, typename FSep>
void for_separated(Range&& range, F&& f_action, FSep&& f_separation)
{
    if(std::empty(range)) { return; }
    f_action(*range.begin());

    for(auto it = std::next(range.begin()); it != range.end(); ++it)
    {
        f_separation();
        f_action(*it);
    }
}
```

- Redefine `print`

```
template <typename T>
void print(const std::vector<T>& v)
{
    for_separated(v,
        [](const auto& x){ std::cout << x; },
        []{ std::cout << ", "; });
}
```

(on wandbox.org)

- `for_separated` is reusable
 - It provides the *control flow*
 - The user provides the *actions*


```
const auto wide_print = [](const auto& sentence)
{
    for_separated(sentence,
        [](const auto& x){ std::cout << x; },
        []{ std::cout << ' '; });
};

wide_print("helloworld"s);
```

(on wandbox.org)

helloworld

```
template <typename Range, typename Pred, typename F>
void consume_if(Range&& range, Pred&& pred, F&& f)
{
    for(auto it = std::begin(range); it  $\neq$  std::end(range);)
    {
        if(pred(*it))
        {
            f(*it);
            it = range.erase(it);
        }
        else { ++it; }
    }
}
```

```
consume_if(_systems,
           [](auto& system){ return system.is_initialized(); },
           change_state_to(state::ready_to_sync));
```

- Currently the easiest way to express asynchronous callbacks
 - `std::future` , `std::thread` , ...
- Many use cases might be superseded by *coroutines*

```
auto graph = all
{
    []{ return http_get_request("animals.com/cat/0.png"); },
    []{ return http_get_request("animals.com/dog/0.png"); }
}
.then([](std::tuple<data, data> payload)
{
    std::apply(stitch, payload);
}));
```

- *"Zero-allocation & no type erasure futures"* @ [ACCU 2018](#), [C++Now 2018](#)

- *"zero-overhead C++17 currying & partial application"*
- *"compile-time iteration with C++20 lambdas"*

```
enumerate_types<int, float, char>([]<typename T, auto I>()  
{  
    std::cout << I << ": " << typeid(T).name() << '\n';  
});
```

0: i

1: f

2: c

- Sometimes other abstractions can be used to achieve the same goals
 - RAI guards
 - Iterators
 - ...

- *Example:* thread-safe access to an object via `synchronized<T>`

```
class foo { /* ... */ };  
synchronized<foo> s_foo;
```

// some way to access contents of `s_foo` in a thread-safe manner

- What interface should `synchronized` expose?
 - i. RAII guards
 - ii. Higher-order functions

```
synchronized<foo> s_foo;  
  
{  
    auto f = s_foo.access();  
    f→some_foo_method();  
}
```

- (+) Friendly to control flow

```
synchronized<foo> s_foo;  
  
s_foo.access([])(foo& f)  
{  
    f.some_foo_method();  
});
```

- (+) Simpler implementation
- (−) Might require captures
- (−) Unfriendly to control flow


```
template <typename T>
class synchronized
{
    T _obj;
    std::mutex _mtx;

public:
    auto access()
    {
        struct access_guard
        {
            std::lock_guard<std::mutex> _guard;
            T* operator→();
            // ... constructors, etc ...
        };

        return access_guard{*this};
    }
};
```

```
template <typename T>
class synchronized
{
    T _obj;
    std::mutex _mtx;

public:
    template <typename F>
    auto access(F&& f)
    {
        std::lock_guard guard{_mtx};
        return std::forward<F>(f)(_obj);
    }
};
```

(on wandbox.org)

- *Example:* benchmarking a function

```
template <typename F>
auto benchmark(F&& f)
{
    const auto t = std::chrono::high_resolution_clock::now();
    f();
    return std::chrono::high_resolution_clock::now() - t;
}
```

- *Example:* iterating over filtered range

```
std::vector<int> ints{/* ... */};  
  
for(int x : filtered(ints, even))  
{  
    /* ... */  
}
```

- (+) Friendly to control flow
- (+) More composable with `std`
- (−) Complicated implementation

```
std::vector<int> ints{/* ... */};  
  
for_filtered(ints, is_even,  
    [](int x){ /* ... */ });
```

- (+) Simpler implementation
- (−) Might require captures
- (−) Unfriendly to control flow

- From `Boost.Iterator`

`filter_iterator` synopsis

```
template <class Predicate, class Iterator>
class filter_iterator
{
public:
    typedef iterator_traits<Iterator>::value_type value_type;
    typedef iterator_traits<Iterator>::reference reference;
    typedef iterator_traits<Iterator>::pointer pointer;
    typedef iterator_traits<Iterator>::difference_type difference_type;
    typedef /* see below */ iterator_category;

    filter_iterator();
    filter_iterator(Predicate f, Iterator x, Iterator end = Iterator());
    filter_iterator(Iterator x, Iterator end = Iterator());
    template<class OtherIterator>
    filter_iterator(
        filter_iterator<Predicate, OtherIterator> const& t
        , typename enable_if_convertible<OtherIterator, Iterator>::type* = 0 // exposition
    );
    Predicate predicate() const;
    Iterator end() const;
    Iterator const& base() const;
    reference operator*() const;
    filter_iterator& operator++();
private:
    Predicate m_pred; // exposition only
    Iterator m_iter; // exposition only
    Iterator m_end; // exposition only
};
```

```
for_filtered(ints, is_even, [](int x){ /* ... */ });
```



```
template <typename Range, typename Pred, typename F>
void for_filtered(Range&& range, Pred&& pred, F&& f)
{
    for(auto&& x : range)
        if(pred(x))
            f(x);
}
```

- HOF iteration can have performance benefits compared to ranges
- *Example:* iterating over interleaved (or concatenated) ranges

```
for (std::variant<text, separator> x : interleave(names, separators))  
{  
    // ...  
}  
  
// vs  
  
for_interleaved(overload([](text) { /* ... */ },  
                        [](separator) { /* ... */ }), names, separators);
```

- The latter does not require run-time dispatch

- Very powerful: many different use cases
- Easier to write than existing alternatives
 - When you need a quick testable/reusable abstraction that doesn't have to be composable, they're great
 - Language alternatives might come: *coroutines*, *ranges*, ...
- Do not play nicely with *control flow* on the caller side
 - Consider `return` / `break` / `continue` in a lambda body
- Even more powerful in C++17 and C++20
- Some proposals might have helped... - e.g. [P0573](#)

function_ref

What options do we have to implement *higher-order functions*?

Pointers to functions

```
int operation(int(*f)(int, int))  
{  
    return f(1, 2);  
}
```

- Works with *non-member functions* and *stateless closures*
- Doesn't work with *stateful* `Callable` objects
- Small run-time overhead (easily inlined in the same TU)
- Constrained, with obvious signature

Template parameters

```
template <typename F>  
auto operation(F&& f) → decltype(std::forward<F>(f)(1, 2))  
{  
    return std::forward<F>(f)(1, 2);  
}
```

- Works with any `FunctionObject` (or `Callable`, using `std::invoke`)
- Zero-cost abstraction
- Hard to constrain (less true in C++20)
- Might degrade compilation time

std::function

```
int operation(const std::function<int(int, int)>& f)
{
    return f(1, 2);
}
```

- Works with any `FunctionObject` or `Callable`
- Significant run-time overhead (hard to inline/optimize)
- Constrained, with obvious signature
- Unclear semantics: can be both *owning* or *non-owning*

function_ref

```
int operation(function_ref<int(int, int)> f)
{
    return f(1, 2);
}
```

- Works with any `FunctionObject` or `Callable`
- Small run-time overhead (easily inlined in the same TU)
- Constrained, with obvious signature
- Clear *non-owning* semantics
- Lightweight - think of "`string_view` for `Callable` objects"

- `function_ref<R(Args ...)>` is a *non-owning reference* to a `Callable`
- Parallel:
 - `std::string` → `std::string_view`
 - `std::function` → `std::function_ref`
- Doesn't *own* or *extend the lifetime* of the referenced `Callable`
- Lightweight, friendly to `noexcept` and optimizations, `TriviallyCopyable`
- Proposed by me in [P0792](#) - currently in LWG
 - Many thanks to: *Agustín Bergé, Dietmar Kühl, Eric Niebler, Tim van Deurzen, and Alisdair Meredith*

- Why use `function_ref` instead of `std::function` ?
 - Performance
 - "Clear" reference semantics
- Why use `function_ref` instead of *template parameters*?
 - Easier to write/read/teach
 - Usable in *polymorphic hierarchies*
 - Better compilation times

function_ref

synopsis

```
template <typename Signature>
class function_ref
{
    void* object;           // exposition only
    R(*erased_function)(Args ... ); // exposition only

public:
    function_ref(const function_ref&) noexcept = default;

    template <typename F>
    function_ref(F&&);

    function_ref& operator=(const function_ref&) noexcept = default;

    template <typename F>
    function_ref& operator=(F&&);

    void swap(function_ref&) noexcept;

    R operator()(Args ... ) const noexcept(*see below*);
};
```

```
class replay_map
{
    std::unordered_map<command_id, ref_counted<command>> _items;
    std::unordered_map<queue_id, std::deque<command_id>> _queues;

    void iterate(const queue_id& id,
                 const function_ref<void(const command&)> f) const
    {
        const auto queue_it = _queues.find(id);
        if(queue_it == std::end(_queues)) { return; }

        const auto& q = queue_it->second;
        for(auto it = q.rbegin(); it != q.rend(); ++it)
        {
            f(_items.at(*it).get());
        }
    }
};
```

```
struct packet_cache
{
    using replay_cb = function_ref<void(const packet&)>;
    using consume_cb = function_ref<void(packet&&)>;

    virtual void replay(replay_cb cb) const = 0;
    virtual void consume(consume_cb cb) = 0;

    virtual ~packet_cache() { }
};
```

- ...

```
struct contiguous_packet_cache : packet_cache
{
    // ...

    void replay(replay_cb cb) const override
    {
        for (const auto& p : _packets)
            cb(p);
    }

    void consume(consume_cb cb) override
    {
        for (auto& p : _packets)
            cb(std::move(p));

        clear();
    }
};
```

```
using state_change_cb =  
    function_ref<void(const node_id&, const state_transition&)>;  
  
void node_monitor::sweep(const state_change_cb cb,  
                        const timestamp&      ts)  
{  
    for(auto it = std::begin(_data); it != std::end(_data);)  
    {  
        if (    (it->second._state != node_state::down)  
            && (ts - it->second._last_heartbeat ≥ 10s))  
        {  
            cb(it->first, change_state_to(node_state::down));  
            it = _data.erase(it);  
        }  
        else { ++it; }  
    }  
}
```

```
using state_change_cb =  
    function_ref<void(const node_id&, const state_transition&)>;  
  
void node_monitor::sweep(const state_change_cb cb,  
                        const timestamp&      ts)  
{  
    consume_if(_data,  
               [](const auto& p)  
               {  
                   return (p.second._state == node_state::down)  
                       || (ts - p.second._last_heartbeat < 10s);  
               },  
               [](const auto& p)  
               {  
                   cb(p.first, change_state_to(node_state::down));  
               }));  
}
```

- "Match" a signature through template specialization:

```
template <typename Signature>
class function_ref;

template <typename Return, typename ... Args>
class function_ref<Return(Args ... )>
{
    // ...
};
```

- Store *pointer to* `Callable` object and *pointer to erased function*:

```
template <typename Return, typename ... Args>
class function_ref<Return(Args ... )>
{
private:
    void* _ptr;
    Return (*_erased_fn)(void*, Args ... );

public:
    // ...
};
```


private:

```
void* _ptr;  
Return (*_erased_fn)(void*, Args ... );
```

- On construction, set the pointers:

```
template <typename F, /* ... constraints ... */>  
function_ref(F&& f) noexcept : _ptr{(void*) &f}  
{  
    _erased_fn = [] (void* ptr, Args ... xs) → Return  
    {  
        return (*static_cast<F*>(ptr))(std::forward<Args>(xs) ... );  
    };  
}
```

```
private:  
    void* _ptr;  
    Return (*_erased_fn)(void*, Args ... );
```

- On invocation, go through `_erased_fn` :

```
Return operator()(Args ... xs) const noexcept( /* ... */) {  
    return _erased_fn(_ptr, std::forward<Args>(xs) ... );  
}
```

function_ref

implementation

```

template <typename Return, typename ... Args>
class function_ref<Return(Args ... )>
{
    void* _ptr;
    Return (*_erased_fn)(void*, Args ... );

public:
    template <typename F, /* ... constraints ... */>
    function_ref(F&& f) noexcept : _ptr{(void*) &f}
    {
        _erased_fn = [](void* ptr, Args ... xs) → Return {
            return (*static_cast<F*>(ptr))(std::forward<Args>(xs) ... );
        };
    }

    Return operator()(Args ... xs) const noexcept( /* ... */)
    {
        return _erased_fn(_ptr, std::forward<Args>(xs) ... );
    }
};

```

```
template <typename F>  
function_ref(F&& f);
```

- *Constraints:* `is_same_v<remove_cvref_t<F>, function_ref>` is `false` and
 - If `Signature` has a `noexcept` specifier:
`is_nothrow_invocable_r_v<R, cv-qualifiers F&, Args ... >` is `true` ;
 - Otherwise:
`is_invocable_r_v<R, cv-qualifiers F&, Args ... >` is `true` .

[...]

```
void might_throw();  
void well_behaved() noexcept;  
  
void f(function_ref<void() noexcept>);  
  
f(&might_throw);    // Compile-time error  
f(&well_behaved);    // OK
```

(on godbolt.org)

- Behavior described in [P0045](#): "Qualified `std::function` signatures" (D. Krauss)

```
const function_ref<int()> get_number = []{ return 42; };  
std::cout << get_number() << '\n'; // Undefined behavior (!)
```

- `function_ref` has the same lifetime concerns as `std::string_view`
 - When used as a function parameter, everything is usually fine
 - In other cases, care is needed to avoid *dangling references*

- `constexpr` was removed in the latest revision (R4), as it's not implementable
 - A `reinterpret_cast` (or cast from `void*`) is required
- Unfortunately this forces users to either use templates or write code twice
- Can we do anything about this?
 - `constexpr`-friendly cast from `void*` to `T*` ?
 - Compiler magic for `function_ref` ?

function_ref

Benchmarks

- Used quick-bench.com with Simon Brand's `function_ref` implementation
 - Internally uses Google Benchmark
- Scenario: invoke simple higher-order function in a loop
- Test with:
 - *template parameter*
 - `function_ref`
 - `std::function`
- Also with and without inlining

```
template <typename F>
void templateParameter(F&& f)
{
    benchmark::DoNotOptimize(f());
}

void stdFunction(const std::function<int()>& f)
{
    benchmark::DoNotOptimize(f());
}

void functionRef(const tl::function_ref<int()>& f)
{
    benchmark::DoNotOptimize(f());
}
```

```
template <typename F>
void __attribute__((noinline)) noInlineTemplateParameter(F&& f)
{
    benchmark::DoNotOptimize(f());
}

void __attribute__((noinline)) noInlineStdFunction(const std::function<int()>& f)
{
    benchmark::DoNotOptimize(f());
}

void __attribute__((noinline)) noInlineFunctionRef(const tl::function_ref<int()>& f)
{
    benchmark::DoNotOptimize(f());
}
```

```
static void TemplateParameter(benchmark::State& state) {  
    for (auto _ : state) {  
        templateParameter([]{ return 1; });  
    }  
}
```

```
BENCHMARK(TemplateParameter);
```

```
static void FunctionRef(benchmark::State& state) {  
    for (auto _ : state) {  
        functionRef([]{ return 1; });  
    }  
}
```

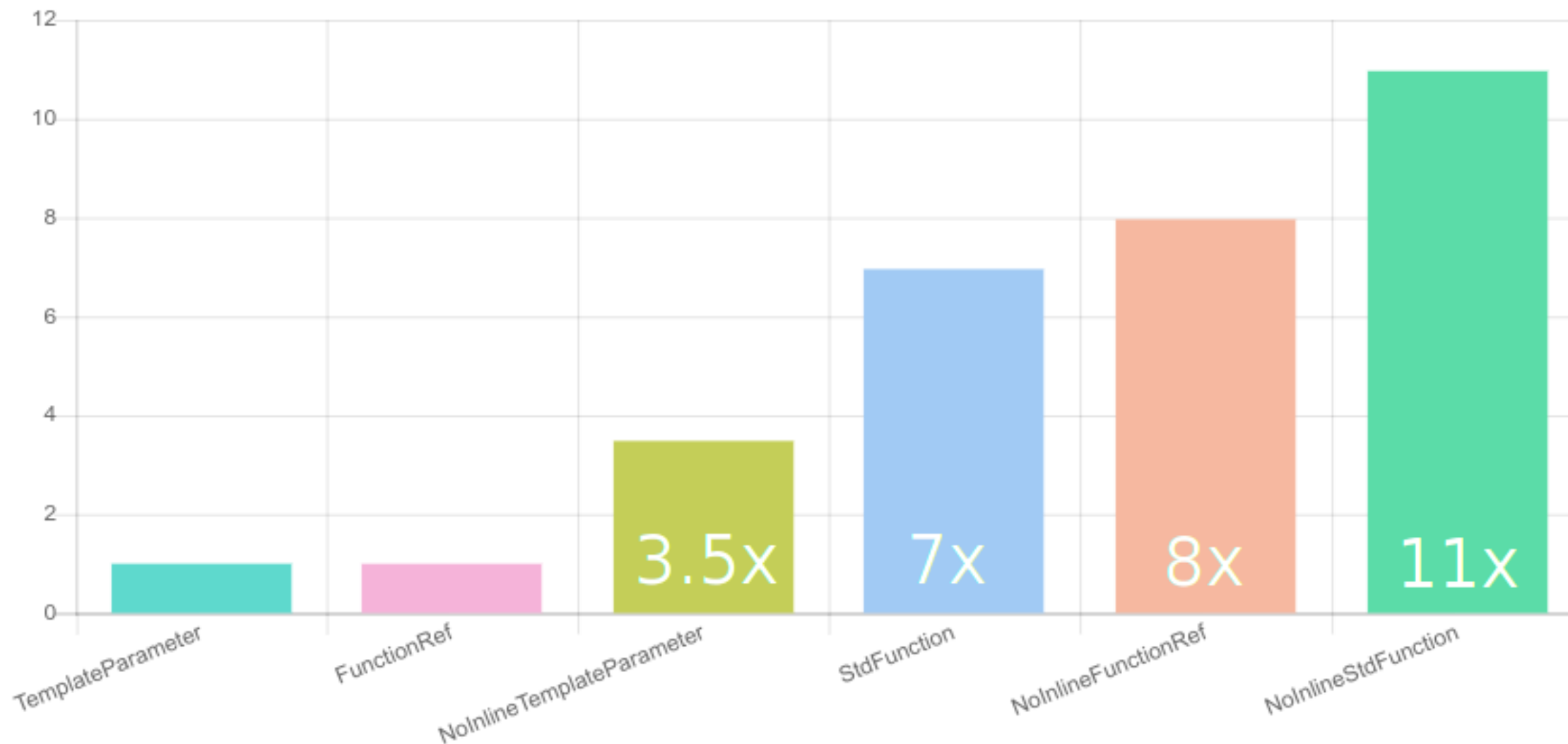
```
BENCHMARK(FunctionRef);
```

```
// ... and so on ...
```

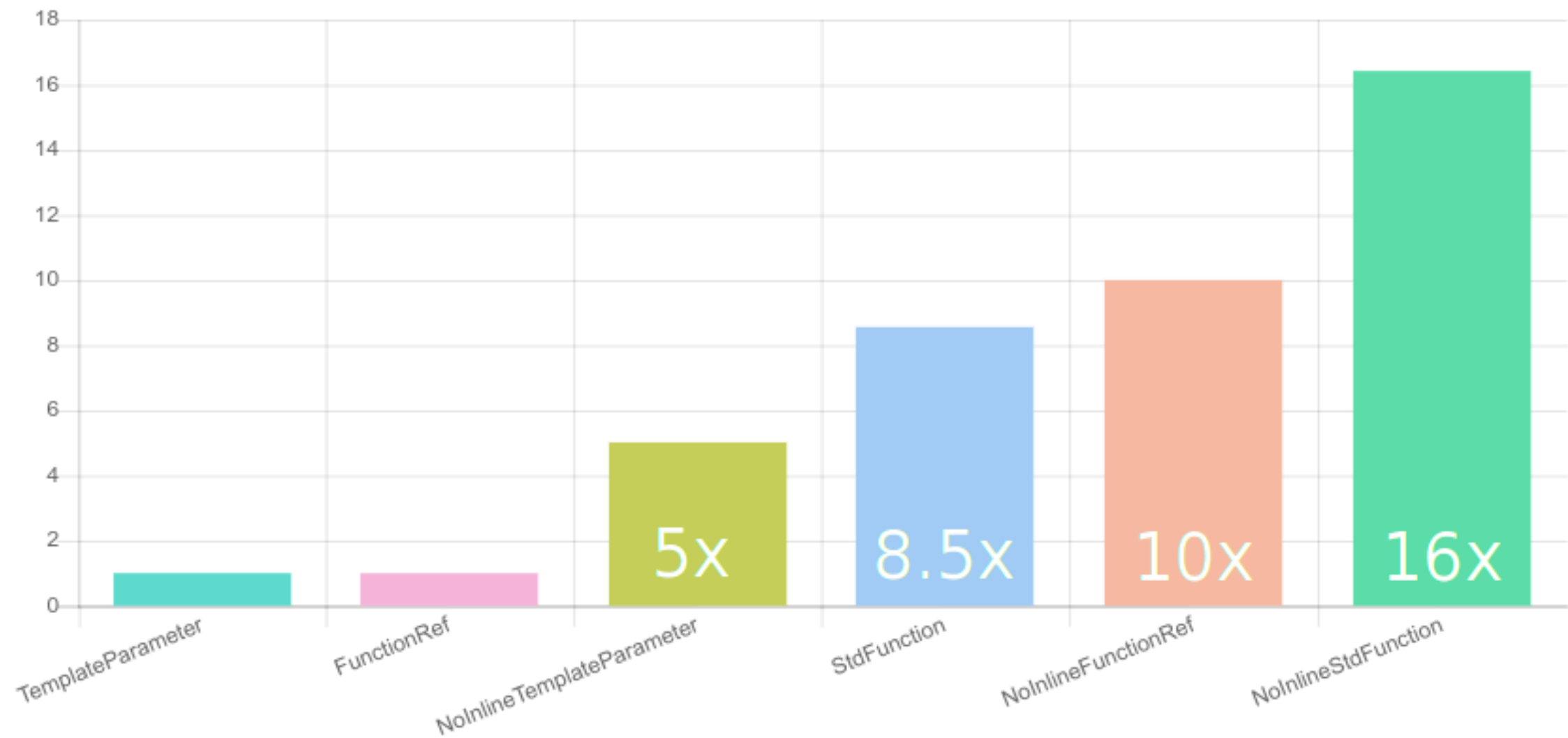
function_ref

benchmarks - results (0/2)

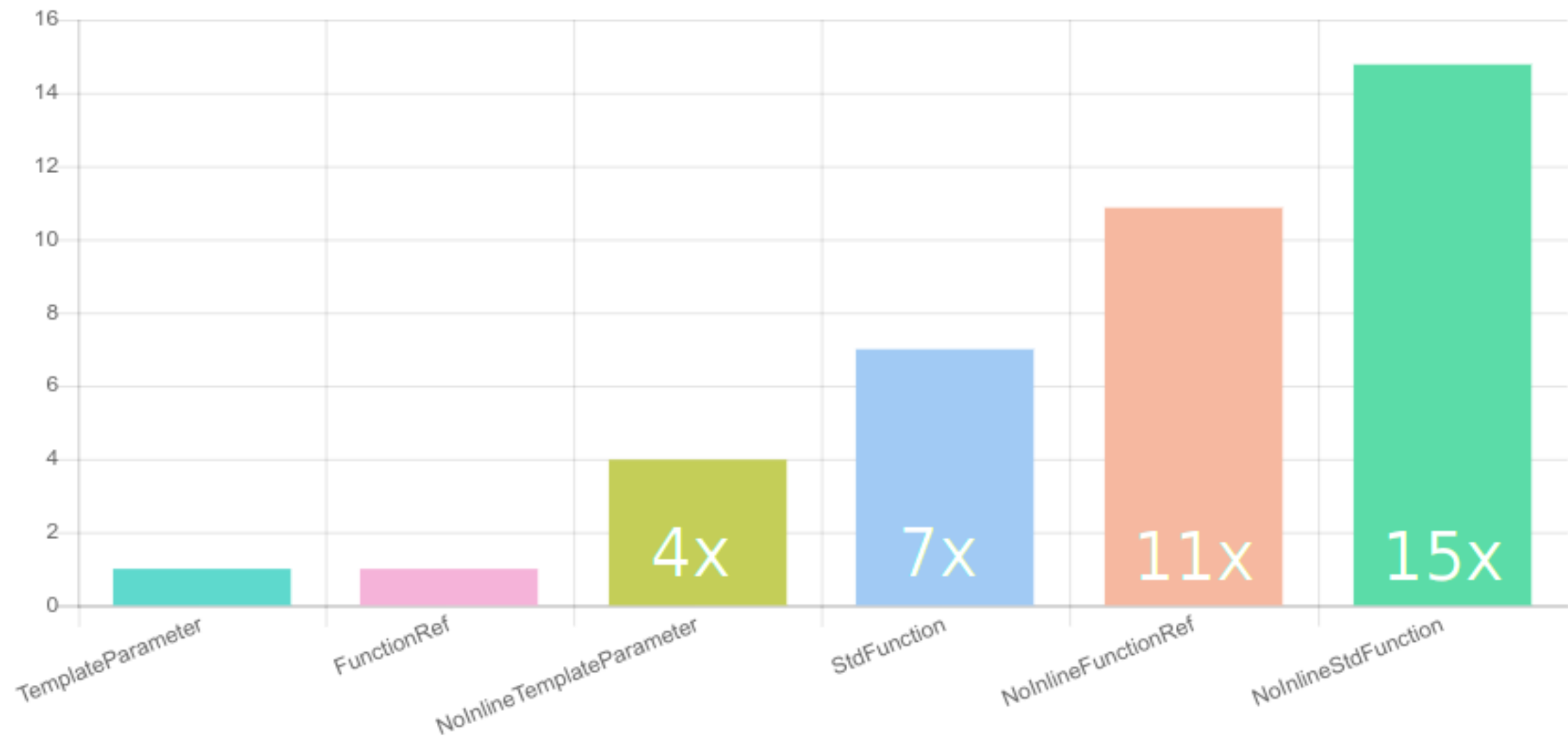
- `g++ 8.x`, `-O3`, `libstdc++` - [\(link\)](#)



- clang++ 7.x, -O3, libstdc++ - [\(link\)](#)



- clang++ 7.x, -O3, libc++ - [\(link\)](#)



- When inlining happens:
 - `function_ref` is as fast as a *template parameter*
 - `std::function` is at least **7x slower** than `function_ref`
- When inlining doesn't happen:
 - `function_ref` is around **2x slower** than a *template parameter*
 - `std::function` is around **1.5x slower** than `function_ref`
- `function_ref` is optimizer-friendly and thrives with inlining
- `function_ref` is always faster than `std::function`

function_ref

Design decisions and controversies

function_ref*size and pointer-to-member-function lifetime*

- `sizeof(function_ref)` is required to be `sizeof(void*) * 2`
 - This makes `function_ref` lightweight and likely to be passed in registers
 - Vocal opposition to bigger sizes
- Consequence: PMFs must outlive a `function_ref` instance

```
function_ref<void(const Service&)> fr = &Service::connect;  
    // Points to the temporary `&Service::connect` instance.  
  
fr();  
    // UB, as the temporary `&Service::connect` is dead.
```

- The previous case could technically be well-defined for a *pointer-to-function*
 - For consistency, it is not

```
void connect(const Service&);  
  
function_ref<void(const Service&)> fr = &connect;  
    // Points to the temporary `&connect` instance.  
  
fr();  
    // UB, as the temporary `&connect` is dead.
```

- LEWG approved this design

- The main use case for `function_ref` is to be used as a function parameter
 - Yet, it exposes an *assignment operator*
- This could be misleading

```
void choice0(context& ctx);  
void choice1(context& ctx);  
  
void menu(context& ctx, int choice)  
{  
    auto default_choice = &choice0;  
    function_ref<void(context&)> choice{default_choice};  
  
    if (choice == 1) { choice = choice1; }  
    else if (choice == 2) { choice = choice2; }  
  
    choice(ctx); // UB!  
}
```

function_ref

std::reference_wrapper

- `function_ref` does not automatically unwrap a `std::reference_wrapper`
 - This leads to double indirection and affects the semantics

```
auto l0 = []{ std::cout << "l0\n"; };
auto l1 = []{ std::cout << "l1\n"; };

std::reference_wrapper rw{std::ref(l0)};
function_ref<void()> fr(rw); // Points to `rw` itself.

fr(); // Prints out `l0`.

rw = l1;

fr(); // Prints out `l1`.
```

- Discussed in LEWG, unanimous dissent to automatically unwrap

- Idea suggested by Adam N. (Yakk)
 - Allow multiple signatures at the same time

```
function_ref<void(int), double(>> fr;  
fr = overload([](int){ },  
              []() → double { }));
```

- Can be implemented more efficiently compared to `overload` of `function_ref`
- Thoughts?
 - Backwards-compatible future addition?
 - `multi_function_ref` ?
 - Complexity not justifiable?

Conclusion

- Any function accepting or returning another is an "*higher-order function*"
 - Many examples in both the C and C++ Standards
- Varied use cases: *avoiding repetition, inverting control flow, ...*
- Highly usable thanks to *lambda expressions*
- Easier to implement compared to some alternatives
 - At the cost of introducing an extra function scope
- **You don't have to go *fully functional* to benefit from them!**

- Non-owning reference to any `Callable` with a given signature
- Missed C++20, hopefully in C++23
- Lightweight, trivial for the compiler to optimize
- Clearer semantics and higher performance compared to `std::function`
- You can start using **`function_ref`** today!
 - [P0792](#)
 - [github:TartanLlama/function_ref](https://github.com/TartanLlama/function_ref)

Thanks!

<https://vittorioromeo.info>

<https://github.com/SuperV1234/cppcon2019>

vittorio.romeo@outlook.com

vromeo5@bloomberg.net

[@supahvee1234](#)

Bloomberg