Higher-order Functions in C++: Techniques and Applications

Michał Dominiak
Nokia Networks
griwes@griwes.info

Outline

Definitions

Definitions

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

Definitions

(Wikipedia:) 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),

Definitions

(Wikipedia:) 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.

The need of higher-order functions

```
auto sum = std::accumulate(std::begin(numbers), std::end(numbers), 0);
```

The need of higher-order functions

```
auto sum = std::accumulate(std::begin(numbers), std::end(numbers), 0);

auto item = std::accumulate(std::begin(keys), std::end(keys), root,
    [](auto && a, auto && b){ return a[b]; }
);
```

The need of higher-order functions

```
auto result_fut = get_calculation().then([](auto && calculation) {
    return calculation.evaluate();
});
```

The need of higher-order functions

```
auto fut = std::accumulate(std::begin(statements), std::end(statements),
    make_ready_future(),
    [](auto && future, auto && stmt) {
        return future.then([=](auto && result) {
            if (!result.should_continue()) {
                return result;
            }
            return stmt->simplify();
        });
    }
);
```

Function pointers are not enough

```
template<typename T>
using comparison_type = bool (*)(const T &, const T &);
```

Function pointers are not enough

```
template<typename T>
using comparison_type = bool (*)(const T &, const T &);

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);
```

Function pointers are not enough

```
template<typename T>
using comparison_type = bool (const T &, const T &);

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);
```

Function pointers are not enough

```
template<typename T>
using comparison_type = bool (const T &, const T &);

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

bool operator==(const foo &, const foo &);
```

Function pointers are not enough

```
template<typename T>
using comparison_type = bool (const T &, const T &);

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

bool operator==(const foo &, const foo &);

do_if_true(foo1, foo2, &operator==);
```

Function pointers are not enough

```
template<typename T>
using comparison_type = bool (const T &, const T &);

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

struct foo {
    bool operator==(const foo &) const;
};
```

Function pointers are not enough

```
template<typename T>
using comparison_type = bool (const T &, const T &);

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

struct foo {
    bool operator==(const foo &) const;
};

do_if_true(a, b, &foo::operator==); // doesn't work!
```

Function pointers are not enough

```
template<typename T>
using comparison_type = bool (const T &, const T &);

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

struct foo {
    bool operator==(const foo &) const;
};

do_if_true(a, b, [](auto && lhs, auto && rhs){ return lhs == rhs; });
```

Function pointers are not enough

```
template<typename T>
using comparison_type = bool (const T &, const T &);

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

bool operator==(foo, foo);
```

Function pointers are not enough

```
template<typename T>
using comparison_type = bool (const T &, const T &);

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

bool operator==(foo, foo);

do_if_true(a, b, &operator==); // doesn't work either!
```

Function pointers are not enough

```
template<typename T>
using comparison_type = bool (const T &, const T &);

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

bool operator==(foo, foo);

do_if_true(a, b, [](auto && lhs, auto && rhs){ return lhs == rhs; });
```

Outline

How it looks like

```
#include <functional>
template<typename T>
using comparison_type = std::function<bool (const T &, const T &)>;
```

How it looks like

```
#include <functional>
template<typename T>
using comparison_type = std::function<bool (const T &, const T &)>;

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);
```

How it looks like

```
#include <functional>
template<typename T>
using comparison_type = std::function<bool (const T &, const T &)>;

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

bool operator==(const foo &, const foo &);
do_if_true(a, b, &operator==);
```

How it looks like

```
#include <functional>
template<typename T>
using comparison_type = std::function<bool (const T &, const T &)>;

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

struct foo {
    bool operator==(const foo &) const;
};
do_if_true(a, b, &foo::operator==);
```

How it looks like

```
#include <functional>
template<typename T>
using comparison_type = std::function<bool (const T &, const T &)>;

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

bool operator==(foo, foo);
do_if_true(a, b, &operator==);
```

How it works

Type erasure.

How it works

```
template<typename Function>
class function;
template<typename R, typename... Args>
class function<R (Args...)> {









};
```

How it works

```
template<typename Function>
class function;
template<typename R, typename... Args>
class function<R (Args...)> {
    class base {
        virtual ~base() = default;
        virtual R call(Args &&...) = 0;
    };

    std::unique_ptr<base> data;

};
```

How it works

```
template<typename Function>
class function;
template<typename R, typename... Args>
class function<R (Args...)> {
    class base {
        virtual ~base() = default;
        virtual R call(Args &&...) = 0;
    };

    std::unique_ptr<base> data;
public:

    R operator()(Args &&... args) const { return data->call(std::forward<Args>(args)...); }
};
```

How it works

```
template<typename Function>
class function;
template<typename R, typename... Args>
class function<R (Args...)> {
    class base {
        virtual ~base() = default;
        virtual R call(Args &&...) = 0;
    };







    std::unique_ptr<base> data;
public:
    template<typename T>
    function(T t) : data{ std::make_unique<impl<T>>(std::move(t)) } {}
    R operator()(Args &&... args) const { return data->call(std::forward<Args>(args)...); }
};
```

How it works

```
template<typename Function>
class function;
template<typename R, typename... Args>
class function<R (Args...)> {
    class base {
        virtual ~base() = default;
        virtual R call(Args &&...) = 0;
    };
    template<typename T>
    class impl : public base {
        T value;
        impl(T t) : value{ std::move(t) } {}
        virtual R call(Args &&... args) override { return std::invoke(t, std::forward<Args>(args)...); }
    };
    std::unique_ptr<base> data;
public:
    template<typename T>
    function(T t) : data{ std::make_unique<impl<T>>(std::move(t)) } {}
    R operator()(Args &&... args) const { return data->call(std::forward<Args>(args)...); }
};
```

How it works

```
template<typename Function>
class function;
template<typename R, typename... Args>
class function<R (Args...)> {









};
```

How it works

```
template<typename Function>
class function;
template<typename R, typename... Args>
class function<R (Args...)> {

    using invoke_t = R (*)(void *, Args &&...);
    invoke_t invoke = nullptr;









};
```

How it works

```
template<typename Function>
class function;
template<typename R, typename... Args>
class function<R (Args...)> {

    using invoke_t = R (*)(void *, Args &&...);
    invoke_t invoke = nullptr;

    std::unique_ptr<void, void (*)(void *)> data{ nullptr, +[](void *){} };

};
```

How it works

```
template<typename Function>
class function;
template<typename R, typename... Args>
class function<R (Args...)> {

    using invoke_t = R (*)(void *, Args &&...);
    invoke_t invoke = nullptr;

    std::unique_ptr<void, void (*)(void *)> data{ nullptr, +[](void *){} };

public:






    R operator()(Args &&... args) const { return invoke(data.get(), std::forward<Args>(args)...); }

};
```

How it works

```
template<typename Function>
class function;
template<typename R, typename... Args>
class function<R (Args...)> {

    using invoke_t = R (*)(void *, Args &&...);
    invoke_t invoke = nullptr;

    std::unique_ptr<void, void (*)(void *)> data{ nullptr, +[](void *){} };

public:

    template<typename F>
    function(F f) {
        invoke = +[](void * data){ return std::invoke(*reinterpret_cast<F *>(data), std::forward<Args>(args)...); }
        data = { new F(std::move(f)), +[](void * ptr){ delete reinterpret_cast<F *>(ptr); } };
    }

    R operator()(Args &&... args) const { return invoke(data.get(), std::forward<Args>(args)...); }

};
```

Problems with std::function: move-only objects

- `std::function` requires Copyable to be constructed

Problems with std::function: move-only objects

- `std::function` requires Copyable to be constructed
- it also requires it for copyability of `std::function` itself

Problems with std::function: move-only objects

- `std::function` requires Copyable to be constructed
- it also requires it for copyability of `std::function` itself
- this precludes the use of it for move-only types

Problems with std::function: move-only objects

- `std::function` requires Copyable to be constructed
- it also requires it for copyability of `std::function` itself
- this precludes the use of it for move-only types
- opinion: the default function type shouldn't be copyable;

Problems with std::function: move-only objects

- `std::function` requires Copyable to be constructed
- it also requires it for copyability of `std::function` itself
- this precludes the use of it for move-only types
- opinion: the default function type shouldn't be copyable; instead we should have a separate type for copyable functions

Problems with std::function: const-correctness

```
struct some_function_object {
    int i = 0;
    int operator()() { return ++i; }
    int operator()() const { return i; }
};
```

Problems with std::function: const-correctness

```
struct some_function_object {
    int i = 0;
    int operator()() { return ++i; }
    int operator()() const { return i; }
};

some_function_object f1;
assert(f1() == 1); // ok; non-const overload selected
```

Problems with std::function: const-correctness

```
struct some_function_object {
    int i = 0;
    int operator()() { return ++i; }
    int operator()() const { return i; }
};

some_function_object f1;
assert(f1() == 1); // ok; non-const overload selected

const some_function_object f2;
assert(f2() == 0); // ok; const overload selected
```

Problems with std::function: const-correctness

```
struct some_function_object {
    int i = 0;
    int operator()() { return ++i; }
    int operator()() const { return i; }
};

std::function<int ()> f1 = some_function_object{};
assert(f1() == 1); // ok; non-const overload selected

const some_function_object f2;
assert(f2() == 0); // ok; const overload selected
```

Problems with std::function: const-correctness

```
struct some_function_object {
    int i = 0;
    int operator()() { return ++i; }
    int operator()() const { return i; }
};

std::function<int ()> f1 = some_function_object{};
assert(f1() == 1); // ok; non-const overload selected

const std::function<int ()> f2 = some_function_object{};
assert(f2() == 0); // failed; non-const overload selected
```

Problems with std::function: const-correctness

```
struct some_function_object {
    int i = 0;
    int operator()() { return ++i; }
    int operator()() const { return i; }
};

std::function<int ()> f1 = some_function_object{};
assert(f1() == 1); // ok; non-const overload selected

const std::function<int ()> f2 = some_function_object{};
assert(f2() == 0); // failed; non-const overload selected

template<typename R, typename... Args>
class function<R (Args...)> {
    // ...
    R operator()(Args &&... args) const { return invoke(data.get(), std::forward<Args>(args)...); }
    // ...
};
```

Problems with std::function: const-correctness

```
struct some_function_object {
    int i = 0;
    int operator()() { return ++i; }
    int operator()() const { return i; }
};

std::function<int ()> f1 = some_function_object{};
assert(f1() == 1); // ok; non-const overload selected

std::function<int () const> f2 = some_function_object{}; // note: not currently valid
assert(f2() == 0); // ok; the const overload selected
```

Problems with std::function: const-correctness

```
struct some_function_object {
    int i = 0;
    int operator()() { return ++i; }
    int operator()() const { return i; }
};

std::function<int ()> f1 = some_function_object{};
assert(f1() == 1); // ok; non-const overload selected

std::function<int () const> f2 = some_function_object{}; // note: not currently valid
assert(f2() == 0); // ok; the const overload selected
```

- P0045 – "Qualified std::function signatures" by David Krauss

Problems with erased wrappers: signature deduction

- it's not possible to deduce any part of the call signatures when using runtime wrappers

Problems with erased wrappers: signature deduction

- it's not possible to deduce any part of the call signatures when using runtime wrappers
- this is not particularly problematic with argument types, since we have to know those anyway...

Problems with erased wrappers: signature deduction

- it's not possible to deduce any part of the call signatures when using runtime wrappers
- this is not particularly problematic with argument types, since we have to know those anyway...
- ...but it is problematic with the return types

Problems with erased wrappers: signature deduction

- it's not possible to deduce any part of the call signatures when using runtime wrappers
- this is not particularly problematic with argument types, since we have to know those anyway...
- ...but it is problematic with the return types

```
template<typename Ret>
void foo(std::function<Ret (int, int)>);
```

Problems with erased wrappers: signature deduction

- it's not possible to deduce any part of the call signatures when using runtime wrappers
- this is not particularly problematic with argument types, since we have to know those anyway...
- ...but it is problematic with the return types

```
template<typename Ret>
void foo(std::function<Ret (int, int)>);

int bar(int, int);
```

Problems with erased wrappers: signature deduction

- it's not possible to deduce any part of the call signatures when using runtime wrappers
- this is not particularly problematic with argument types, since we have to know those anyway...
- ...but it is problematic with the return types

```
template<typename Ret>
void foo(std::function<Ret (int, int)>);

int bar(int, int);

foo(&bar); // nope
```

Problems with erased wrappers: signature deduction

- it's not possible to deduce any part of the call signatures when using runtime wrappers
- this is not particularly problematic with argument types, since we have to know those anyway...
- ...but it is problematic with the return types

```
template<typename Ret>
void foo(std::function<Ret (int, int)>);

int bar(int, int);

foo(std::function<decltype(bar)>(&bar)); // ugly
```

Outline

1. Introduction
2. std::function
3. Deduced template arguments
4. function_ref
5. (A)synchronous callbacks and ownership model
6. The end

How it looks like

```
template<typename T, typename Comparator>
void do_if_true(const T &, const T &, Comparator);
```

How it looks like

```
template<typename T, typename Comparator>
void do_if_true(const T &, const T &, Comparator);

bool operator==(const foo &, const foo &);
do_if_true(a, b, &operator==);
```

How it looks like

```
template<typename T, typename Comparator>
void do_if_true(const T &, const T &, Comparator);

struct foo {
    bool operator==(const foo &) const;
};
do_if_true(a, b, &foo::operator==);
```

How it looks like

```
template<typename T, typename Comparator>
void do_if_true(const T &, const T &, Comparator);

bool operator==(foo, foo);
do_if_true(a, b, &operator==);
```

How it works

A separate function is generated at compile time for each type of the argument it is called with.

Deduction is your friend

- you deduce the actual argument type, not a call signature

Deduction is your friend

- you deduce the actual argument type, not a call signature
- this allows us to deduce the return type for a given set of arguments

Deduction is your friend

- you deduce the actual argument type, not a call signature
- this allows us to deduce the return type for a given set of arguments

```
template<typename F>
void foo(F && f) {
    using R = decltype(std::forward<F>(f)(1, 2, 17));
    // ...
}
```

Deduction is your friend

- you deduce the actual argument type, not a call signature
- this allows us to deduce the return type for a given set of arguments

```
template<typename F>
void foo(F && f) {
    using R = decltype(std::forward<F>(f)(1, 2, 17));
    // ...
}
```

- ...you can also try to deduce the arguments

Deduction is your friend... but not always

```
template<typename F>
void foo(F && f);
```

Deduction is your friend... but not always

```
template<typename F>
void foo(F && f);

void bar(int);
void bar(std::string);
```

Deduction is your friend... but not always

```
template<typename F>
void foo(F && f);

void bar(int);
void bar(std::string);

foo(&bar); // oops
```

Deduction is your friend... but not always

```
template<typename F>
void foo(F && f);

void bar(int);
void bar(std::string);

foo(static_cast<void(*)(int)>(&bar)); // ugly
```

Deduction is your friend... but not always

```
template<typename F>
void foo(F && f);

void bar(int);
void bar(std::string);

foo([](auto &&... args)
    -> decltype(bar(std::forward<decltype(args)>(args)...))
    { return bar(std::forward<decltype(args)>(args)...); });
```

Deduction is your friend... but not always

```
template<typename F>
void foo(F && f);

void bar(int);
void bar(std::string);

foo([](auto &&... args)
    -> decltype(bar(std::forward<decltype(args)>(args)...))
    { return bar(std::forward<decltype(args)>(args)...); });

foo([]bar); // so much nicer
```

Problems with template arguments: code bloat

- a separate function is generated for each type it's called with

Problems with template arguments: code bloat

- a separate function is generated for each type it's called with
- this allows for aggressive inlining and deeper optimization than other schemes

Problems with template arguments: code bloat

- a separate function is generated for each type it's called with
- this allows for aggressive inlining and deeper optimization than other schemes
- but also makes the resulting binary bigger

Problems with template arguments: only static polymorphism

- templates implement static polymorphism

Problems with template arguments: only static polymorphism

- templates implement static polymorphism
- static polymorphism works at compile time

Problems with template arguments: only static polymorphism

- templates implement static polymorphism
- static polymorphism works at compile time
- if you need runtime polymorphism for your higher-order functions...

Problems with template arguments: only static polymorphism

- templates implement static polymorphism
- static polymorphism works at compile time
- if you need runtime polymorphism for your higher-order functions... you're going to need to pass a runtime wrapper (like `std::function`) as an argument

Problems with template arguments: constraining the signature

```
template<typename T, typename Comparator>
void do_if_true(const T &, const T &, Comparator);
```

Problems with template arguments: constraining the signature

```
template<typename T, typename Comparator>
void do_if_true(const T &, const T &, Comparator);

void foo() {}
```

Problems with template arguments: constraining the signature

```
template<typename T, typename Comparator>
void do_if_true(const T &, const T &, Comparator);

void foo() {}

do_if_true(1, 2, &foo);
```

Problems with template arguments: constraining the signature

```
template<typename T, typename Comparator




>
void do_if_true(const T &, const T &, Comparator);

void foo() {}

do_if_true(1, 2, &foo);
```

Problems with template arguments: constraining the signature

```
template<typename T, typename Comparator
    typename = typename std::enable_if<std::is_convertible_v<
        std::result_of_t<std::decay_t<Comparator>&(const T &, const T &)>,
        bool
    >>::type
>
void do_if_true(const T &, const T &, Comparator);

void foo() {}

do_if_true(1, 2, &foo);
```

Outline

1. Introduction
2. std::function
3. Deduced template arguments
4. function_ref
5. (A)synchronous callbacks and ownership model
6. The end

How it looks like

```
template<typename T>
using comparison_type = function_ref<bool (const T &, const T &)>;
```

How it looks like

```
template<typename T>
using comparison_type = function_ref<bool (const T &, const T &)>;

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);
```

How it looks like

```
template<typename T>
using comparison_type = function_ref<bool (const T &, const T &)>;

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

bool operator==(const foo &, const foo &);
do_if_true(a, b, &operator==);
```

How it looks like

```
template<typename T>
using comparison_type = function_ref<bool (const T &, const T &)>;

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

struct foo {
    bool operator==(const foo &) const;
};
do_if_true(a, b, &foo::operator==);
```

How it looks like

```
template<typename T>
using comparison_type = function_ref<bool (const T &, const T &)>;

template<typename T>
void do_if_true(const T &, const T &, comparison_type<T>);

bool operator==(foo, foo);
do_if_true(a, b, &operator==);
```

How it works

Type erasure...

How it works

Type erasure... again.

How it works

```
template<typename Function>
class function;
template<typename R, typename... Args>
class function<R (Args...)> {

};
```

How it works

```
template<typename Function>
class function_ref;
template<typename R, typename... Args>
class function_ref<R (Args...)> {

};
```

How it works

```
template<typename Function>
class function_ref;
template<typename R, typename... Args>
class function_ref<R (Args...)> {

    using invoke_t = R (*)(void *, Args &&...);
    invoke_t invoke = nullptr;

};
```

How it works

```
template<typename Function>
class function_ref;
template<typename R, typename... Args>
class function_ref<R (Args...)> {

    using invoke_t = R (*)(void *, Args &&...);
    invoke_t invoke = nullptr;

    std::unique_ptr<void, void (*)(void *)> data{ nullptr, +[](void *){} };

};
```

How it works

```
template<typename Function>
class function_ref;
template<typename R, typename... Args>
class function_ref<R (Args...)> {

    using invoke_t = R (*)(void *, Args &&...);
    invoke_t invoke = nullptr;

    void * data = nullptr;

};
```

How it works

```
template<typename Function>
class function_ref;
template<typename R, typename... Args>
class function_ref<R (Args...)> {

    using invoke_t = R (*)(void *, Args &&...);
    invoke_t invoke = nullptr;

    void * data = nullptr;

public:

    R operator()(Args &&... args) const { return invoke(data.get(), std::forward<Args>(args)...); }

};
```

How it works

```
template<typename Function>
class function_ref;
template<typename R, typename... Args>
class function_ref<R (Args...)> {

    using invoke_t = R (*)(void *, Args &&...);
    invoke_t invoke = nullptr;

    void * data = nullptr;

public:

    R operator()(Args &&... args) const { return invoke(data, std::forward<Args>(args)...); }

};
```

How it works

```
template<typename Function>
class function_ref;
template<typename R, typename... Args>
class function_ref<R (Args...)> {

    using invoke_t = R (*)(void *, Args &&...);
    invoke_t invoke = nullptr;

    void * data = nullptr;

public:

    template<typename F>
    function(F f) {
        invoke = +[](void * data){ return std::invoke(*reinterpret_cast<F *>(data), std::forward<Args>(args)...); }
        data = { new F(std::move(f)), +[](void * ptr){ delete reinterpret_cast<F *>(ptr); } };
    }

    R operator()(Args &&... args) const { return invoke(data, std::forward<Args>(args)...); }

};
```

How it works

```
template<typename Function>
class function_ref;
template<typename R, typename... Args>
class function_ref<R (Args...)> {

    using invoke_t = R (*)(void *, Args &&...);
    invoke_t invoke = nullptr;

    void * data = nullptr;

public:

    template<typename F>
    function_ref(F && f) {
        invoke = +[](void * data){ return std::invoke(*reinterpret_cast<F *>(data), std::forward<Args>(args)...); }
        data = reinterpret_cast<void *>(std::addressof(f));
    }

    R operator()(Args &&... args) const { return invoke(data, std::forward<Args>(args)...); }

};
```

Problems with function_ref: lifetime

- `function_ref` is functionally a type-erasing reference

Problems with function_ref: lifetime

- `function_ref` is functionally a type-erasing reference
- it's easy to pass in a temporary object

Problems with function_ref: lifetime

- `function_ref` is functionally a type-erasing reference
- it's easy to pass in a temporary object
- you need to either (a) not save the reference, or (b) tell the user explicitly you're going to store it

Problems with function_ref: lifetime

- `function_ref` is functionally a type-erasing reference
- it's easy to pass in a temporary object
- you need to either (a) not save the reference, or (b) tell the user explicitly you're going to store it
- otherwise - dangling references

Problems with function_ref: lifetime

- `function_ref` is functionally a type-erasing reference
- it's easy to pass in a temporary object
- you need to either (a) not save the reference, or (b) tell the user explicitly you're going to store it
- otherwise - dangling references
- opinion: we need language-level lifetimes

Problems with function_ref: lifetime

- `function_ref` is functionally a type-erasing reference
- it's easy to pass in a temporary object
- you need to either (a) not save the reference, or (b) tell the user explicitly you're going to store it
- otherwise - dangling references
- opinion: we need language-level lifetimes (but for the love of all that's holy, don't make them *look* like Rust's)

Problems with function_ref: forced reference semantics

- (related to the previous slide, but the other side of the problem)
- reference semantics are enforced

Problems with function_ref: forced reference semantics

- (related to the previous slide, but the other side of the problem)
- reference semantics are enforced
- that means the caller **can not** call the function with a copy very easily

Problems with function_ref: forced reference semantics

- (related to the previous slide, but the other side of the problem)
- reference semantics are enforced
- that means the caller **can not** call the function with a copy very easily
- (rvalue references supported, but no easy syntax to copy a value in-place in the language)

Problems with function_ref: forced reference semantics

- (related to the previous slide, but the other side of the problem)
- reference semantics are enforced
- that means the caller **can not** call the function with a copy very easily
- (rvalue references supported, but no easy syntax to copy a value in-place in the language)

```
void foo(function_ref<void ()>);
```

Problems with function_ref: forced reference semantics

- (related to the previous slide, but the other side of the problem)
- reference semantics are enforced
- that means the caller **can not** call the function with a copy very easily
- (rvalue references supported, but no easy syntax to copy a value in-place in the language)

```
void foo(function_ref<void ()>);

some_callable bar;
```

Problems with function_ref: forced reference semantics

- (related to the previous slide, but the other side of the problem)
- reference semantics are enforced
- that means the caller **can not** call the function with a copy very easily
- (rvalue references supported, but no easy syntax to copy a value in-place in the language)

```
void foo(function_ref<void ()>);

some_callable bar;

foo(bar);
```

Problems with function_ref: forced reference semantics

- (related to the previous slide, but the other side of the problem)
- reference semantics are enforced
- that means the caller **can not** call the function with a copy very easily
- (rvalue references supported, but no easy syntax to copy a value in-place in the language)

```
void foo(function_ref<void ()>);

some_callable bar;

auto copy = bar;
foo(copy);
```

Problems with function_ref: forced reference semantics

- (related to the previous slide, but the other side of the problem)
- reference semantics are enforced
- that means the caller **can not** call the function with a copy very easily
- (rvalue references supported, but no easy syntax to copy a value in-place in the language)

```
void foo(function_ref<void ()>);

some_callable bar;

foo(auto(bar)); // invalid; only usable in new expressions: new auto(1)
```

Outline

1. Introduction
2. std::function
3. Deduced template arguments
4. function_ref
5. (A)synchronous callbacks and ownership model
6. The end

Ownership and (a)sync

- whether we care about ownership for the sake of correctness or not depends on a kind of an algorithm we are passing a function into

Ownership and (a)sync

- whether we care about ownership for the sake of correctness or not depends on a kind of an algorithm we are passing a function into
- for synchronous algorithms, both semantics lead to legal behavior

Ownership and (a)sync

- whether we care about ownership for the sake of correctness or not depends on a kind of an algorithm we are passing a function into
- for synchronous algorithms, both semantics lead to legal behavior
- for asynchronous algorithms, reference semantics can lead to illegal behavior (dangling references)

Ownership and (a)sync

- whether we care about ownership for the sake of correctness or not depends on a kind of an algorithm we are passing a function into
- for synchronous algorithms, both semantics lead to legal behavior
- for asynchronous algorithms, reference semantics can lead to illegal behavior (dangling references)
- for synchronous algorithms, we would like to benefit from performance improvements of `function_ref`

Ownership and (a)sync

- whether we care about ownership for the sake of correctness or not depends on a kind of an algorithm we are passing a function into
- for synchronous algorithms, both semantics lead to legal behavior
- for asynchronous algorithms, reference semantics can lead to illegal behavior (dangling references)
- for synchronous algorithms, we would like to benefit from performance improvements of `function_ref`
- for asynchronous algorithms, we would like to be able to select which semantics we want

Value vs reference semantics

- value vs reference semantics should *not* be tied to an algorithm

Value vs reference semantics

- value vs reference semantics should *not* be tied to an algorithm
- an algorithm should be easy to be used correctly and hard to use incorrectly

Value vs reference semantics

- value vs reference semantics should *not* be tied to an algorithm
- an algorithm should be easy to be used correctly and hard to use incorrectly
- when faced with two different correct choices...

Value vs reference semantics

- value vs reference semantics should *not* be tied to an algorithm
- an algorithm should be easy to be used correctly and hard to use incorrectly
- when faced with two different correct choices... we should default to the one that is more inconvenient as the opt-in vs as the opt-out

Value vs reference semantics

- value vs reference semantics should *not* be tied to an algorithm
- an algorithm should be easy to be used correctly and hard to use incorrectly
- when faced with two different correct choices... we should default to the one that is more inconvenient as the opt-in vs as the opt-out
- opt-in pass-by-value is hard – need to spell the type, use `decltype` or use `auto` in a separate statement to make a copy

Value vs reference semantics

- value vs reference semantics should *not* be tied to an algorithm
- an algorithm should be easy to be used correctly and hard to use incorrectly
- when faced with two different correct choices... we should default to the one that is more inconvenient as the opt-in vs as the opt-out
- opt-in pass-by-value is hard – need to spell the type, use `decltype` or use `auto` in a separate statement to make a copy
- opt-in pass-by-reference is easy – just support `std::reference_wrapper`

Value vs reference semantics

```
template<typename T>
struct reference_wrapper {
    reference_wrapper(T &);
    reference_wrapper(T &&) = delete;
    T & get() const;
    operator T &() const;
    template<typename... Args>
    auto operator()(Args &&...) const;
};
```

Ownership erasure

```
some_callable foo;
std::function<void ()> bar = std::ref(foo);
```

Ownership erasure

```
template<typename Function>
class my_function;
template<typename R, typename... Args>
class my_function<R (Args...)> {
    using invoke_t = R (*)(void *, Args &&...);
    invoke_t invoke = nullptr;
    std::unique_ptr<void, void (*)(void *)> data{ nullptr, +[](void *){} };
public:
    template<typename F>
    my_function(F f) {
        invoke = +[](void * data){ return (*reinterpret_cast<F *>(data))(std::forward<Args>(args)...); }
        data = { new F(std::move(f)), +[](void * ptr){ delete reinterpret_cast<F *>(ptr); } };
    }




    R operator()(Args &&... args) const { return invoke(data.get(), std::forward<Args>(args)...); }
};
```

Ownership erasure

```
template<typename Function>
class my_function;
template<typename R, typename... Args>
class my_function<R (Args...)> {
    using invoke_t = R (*)(void *, Args &&...);
    invoke_t invoke = nullptr;
    std::unique_ptr<void, void (*)(void *)> data{ nullptr, +[](void *){} };
public:
    template<typename F>
    my_function(F f) {
        invoke = +[](void * data){ return (*reinterpret_cast<F *>(data))(std::forward<Args>(args)...); }
        data = { new F(std::move(f)), +[](void * ptr){ delete reinterpret_cast<F *>(ptr); } };
    }

    template<typename F>
    my_function(std::reference_wrapper<F> f) {
        invoke = +[](void * data){ return (*reinterpret_cast<F *>(data))(std::forward<Args>(args)...); }
        data = { std::addressof(f.get()), +[](void *){} };
    }

    R operator()(Args &&... args) const { return invoke(data.get(), std::forward<Args>(args)...); }
};
```

Removing static information is not always good

- using `my_function` instead of `function_ref` in an interface loses some information

Removing static information is not always good

- using `my_function` instead of `function_ref` in an interface loses some information
- using `my_function` instead of `std::function` does not lose information

Removing static information is not always good

- using `my_function` instead of `function_ref` in an interface loses some information
- using `my_function` instead of `std::function` does not lose information – no new capabilities, just a performance increase

Removing static information is not always good

- using `my_function` instead of `function_ref` in an interface loses some information
- using `my_function` instead of `std::function` does not lose information – no new capabilities, just a performance increase
- can lead to more subtle and harder to debug dangling references

Removing static information is not always good

- using `my_function` instead of `function_ref` in an interface loses some information
- using `my_function` instead of `std::function` does not lose information – no new capabilities, just a performance increase
- can lead to more subtle and harder to debug dangling references
- already a problem with `std::function`

The search for unique_function

- `my_function` doesn't require copyability

The search for unique_function

- `my_function` doesn't require copyability
- this is sometimes very helpful (think lambdas that captured non-copyable objects)

The search for unique_function

- `my_function` doesn't require copyability
- this is sometimes very helpful (think lambdas that captured non-copyable objects)
- no current standard class that does this

The search for unique_function

- `my_function` doesn't require copyability
- this is sometimes very helpful (think lambdas that captured non-copyable objects)
- no current standard class that does this
- P0288 - "A polymorphic wrapper for all Callable objects" by David Krauss (again!)

Outline

1. Introduction
2. std::function
3. Deduced template arguments
4. function_ref
5. (A)synchronous callbacks and ownership model
6. The end

Takeaways

- you can have dynamic polymorphism on top of static polymorphism, but not the other way around

Takeaways

- you can have dynamic polymorphism on top of static polymorphism, but not the other way around
- use static polymorphism (templates!) if you don't actually *require* runtime type erasure

Takeaways

- you can have dynamic polymorphism on top of static polymorphism, but not the other way around
- use static polymorphism (templates!) if you don't actually *require* runtime type erasure
- `function_ref`-like type is a good way to convey the meaning of "only references (probably don't store me)"

Takeaways

- you can have dynamic polymorphism on top of static polymorphism, but not the other way around
- use static polymorphism (templates!) if you don't actually *require* runtime type erasure
- `function_ref`-like type is a good way to convey the meaning of "only references (probably don't store me)"
- `std::function` looks like it only keeps values...

Takeaways

- you can have dynamic polymorphism on top of static polymorphism, but not the other way around
- use static polymorphism (templates!) if you don't actually *require* runtime type erasure
- `function_ref`-like type is a good way to convey the meaning of "only references (probably don't store me)"
- `std::function` looks like it only keeps values... but due to `std::reference_wrapper` and similar classes, you can still have it lead to dangling references

Takeaways

- you can have dynamic polymorphism on top of static polymorphism, but not the other way around
- use static polymorphism (templates!) if you don't actually *require* runtime type erasure
- `function_ref`-like type is a good way to convey the meaning of "only references (probably don't store me)"
- `std::function` looks like it only keeps values... but due to `std::reference_wrapper` and similar classes, you can still have it lead to dangling references
- approaches like "ownership erasure" gives you mostly the same, but by knowing about `std::reference_wrapper`, can avoid data duplication and performance loss due to double indirection

Takeaways

- you can have dynamic polymorphism on top of static polymorphism, but not the other way around
- use static polymorphism (templates!) if you don't actually *require* runtime type erasure
- `function_ref`-like type is a good way to convey the meaning of "only references (probably don't store me)"
- `std::function` looks like it only keeps values... but due to `std::reference_wrapper` and similar classes, you can still have it lead to dangling references
- approaches like "ownership erasure" gives you mostly the same, but by knowing about `std::reference_wrapper`, can avoid data duplication and performance loss due to double indirection
- `std::function` sometimes requires too much – consider types like `unique_function` if you don't actually *require* copyability

Related links

- `https://wg21.link/p0045`
- `https://wg21.link/p0288`
- `https://vittorioromeo.info/index/blog/passing_functions_to_functions.html`
- `https://gist.github.com/SuperV1234/a1742d0ef143e17c30c65bc1d3d79a58`

Questions and answers

Questions and answers

Thank you!