unique_pseudofunction

N overloads for the price of one

Bloomberg

Engineering

C++Russia 2020

November 12, 2020

Filipp Gelman, P.E.
fgelman1@bloomberg.net

TechAtBloomberg.com

unique_pseudofunction

N overloads for the price of one

Filipp Gelman, P.E.

fgelman1@bloomberg.net

C++Russia 2020

November 12, 2020

~~unique_pseudofunction~~

Overloading any_invocable

Filipp Gelman, P.E.

fgelman1@bloomberg.net

C++Russia 2020

November 12, 2020

Outline

What is overloaded `any_invocable`?

- Type erasing container for function objects like `std::function`.
- Move only like `any_invocable`.
- **Has N overloads of `operator()` instead of 1.**

```
any_invocable<int   (int   , int   )>

// vs

any_invocable<int   (int   , int   ),
              float (float , float ),
              double(double, double)>
```

What is a callback?

Suppose you have an async interface.

```
int sendAsync(Request const& request,
              void (*on_response)(int error_code,
                                  Response const* resp,
                                  void* data),
              void* data);
```

- `int error_code` is `0` and `resp` points to the received response.
- `int error_code` is non-zero and `resp` is `NULL`.
- Opaque `data` is forwarded to `on_response`.

What is `std::function`?

```
int sendAsync(Request const& request,
              std::function<void(int error_code,
                                 Response const* resp)> on_response);
```

- `int error_code` is `0` and `resp` points to the received response.
- `int error_code` is non-zero and `resp` is `NULL`.
- ~~Opaque `data` is forwarded to `on_response`~~.
- How many times is `on_response` copied?

What is `std::function`?

```
auto callback =
    [resource = std::make_unique<Resource>()]
    (int error_code, Response const* resp) {
        // ...
    };

sendAsync(request, std::move(callback));
```

What is `std::function`?

```
auto callback =
    [resource = std::make_unique<Resource>()]
    (int error_code, Response const* resp) {
        // ...
    };

sendAsync(request, std::move(callback));
```

This doesn't work:

```
functional:99:99: error: use of deleted function
    'lambda::lambda(lambda const&)'
```

What is `any_invocable`?

http://wg21.link/p0288

```
int sendAsync(Request const& request,
              any_invocable<void(int error_code,
                                 Response const* resp)> on_response);
```

- `int error_code` is `0` and `resp` points to the received response.
- `int error_code` is non-zero and `resp` is `NULL`.
- ~~Opaque `data` is forwarded to `on_response`~~.
- ~~How many times is `on_response` copied~~?

What is overloaded `any_invocable`?

```
int sendAsync(Request const& request,
              any_invocable<void(int error_code),
                            void(Response const& resp)> on_response);
```

- ~~`error_code` is `0` and `resp` points to the received response~~.
- ~~`error_code` is non-zero and `resp` is `NULL`~~.
- ~~Opaque `data` is forwarded to `on_response`~~.
- ~~How many times is `on_response` copied~~?
- Invoked with either `error_code` or `Response`.

What is overloaded `any_invocable`?

```
struct Callback {
    void operator()(int error_code) { /* handle error */ }

    void operator()(Response const& resp) { /* process response */ }
};

any_invocable<void(int), void(Response const&)> f = Callback{};
```

What is overloaded `any_invocable`?

```
any_invocable<void(int), void(Response const&)> f =
    std::overload(
        [](int error_code) { /* handle error */ },
        [](Resonse const& resp) { /* process response */ });
```

https://wg21.link/p0051

Motivation

Why overload `any_invocable`?

- Type erased overload sets are cool and useful.
- Demonstrate one implementation of overloaded `any_invocable`.
- **Demonstrate techniques**.
- ~~Include in standard library~~.

Building any_invocable

```
template <typename> class any_invocable;

template <typename RET, typename... ARGS>
class any_invocable<RET(ARGS...)> {
    // ...

public:
    // special member functions

    template</* ... */>
    any_invocable(T object);

    RET operator()(ARGS...);
};
```

Building any_invocable

```
template <typename RET, typename... ARGS>
class any_invocable<RET(ARGS...)> {
    struct base {
        virtual RET operator()(ARGS&&...) = 0;
        virtual ~base() = default;
    };
    // ...

    std::unique_ptr<base> ptr_;

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

Building any_invocable

```
template <typename RET, typename... ARGS>
class any_invocable<RET(ARGS...)> {
    // struct base

    template <typename T>
    struct derived : base {
        // ...
    };

    unique_ptr<base> ptr_;  // points to derived<T>

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

Building any_invocable

```
template <typename T>
struct derived : base {
    T object;

    explicit derived(T object) : object(std::move(object)) {}

    RET operator()(ARGS&&... args) override {
        return std::invoke(object, std::forward<ARGS>(args)...);
    }
};
```

Building any_invocable

```
class any_invocable<RET(ARGS...)> {
    // struct base ...
    // struct derived ...

    std::unique_ptr<base> ptr_;

public:
    // public interface
};
```

Building any_invocable

```
class any_invocable<RET(ARGS...)> {
    // ...

public:
    any_invocable() noexcept = default;
    any_invocable(any_invocable&&) noexcept = default;
    any_invocable& operator=(any_invocable&&) noexcept = default;
    ~any_invocable() = default;

    // converting constructor, operator()
};
```

Building any_invocable

```
// Converting constructor
template <typename T /* , ... */>
any_invocable(T object) :
    ptr_(std::make_unique<derived<T>>(std::move(object))) {}
```

Constraining the constructor is important!

```
void test(std::string);
void test(any_invocable);

void call_test() {
    test("Hi");
}
```

Building any_invocable

```
source:99:99 error: call of overloaded 'test(char const[3])' is ambiguous
    test(Test());
source:99:99: note: candidate 'void test(std::string)'
    'void test(std::string);'
source:99:99: note: candidate 'void test(any_invocable)'
    'void test(any_invocable);'
```

Building any_invocable

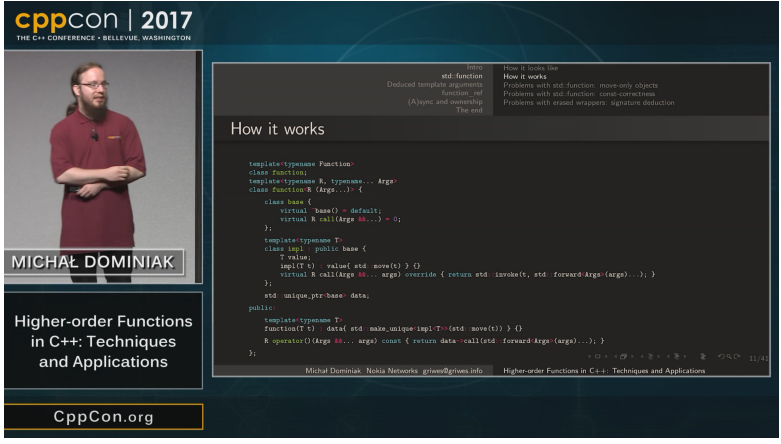
```
// Converting constructor
template <typename T, typename = std::enable_if_t<
    std::is_invocable_r_v<RET, T&, ARGS&&...>>>
any_invocable(T object) :
    ptr_(std::make_unique<derived<T>>(std::move(object))) {}
```

Building any_invocable

```
template <typename RET, typename... ARGS>
class any_invocable<RET(ARGS...)> {
    // ...

public:
    // ...
    RET operator()(ARGS... args) {
        return (*ptr_)(std::forward<ARGS>(args)...);
    }
};
```

Building any_invocable

https://youtu.be/EbnRt-omrFY



Adding more overloads

```
struct base {
    // ...
    virtual RET operator()(ARGS&&...) = 0;
    // ...
};
```

Adding more overloads

```
struct base {
    virtual RET1 operator()(ARGS1&&...) = 0;
    virtual RET2 operator()(ARGS2&&...) = 0;
    virtual RET3 operator()(ARGS3&&...) = 0;
    // ...
};
```

Can we build this?

Adding more overloads

```
struct base {
    virtual RET1 operator()(ARGS1&&...) = 0;
    virtual RET2 operator()(ARGS2&&...) = 0;
    virtual RET3 operator()(ARGS3&&...) = 0;
    // ...
};
```

~~Can we build this?~~

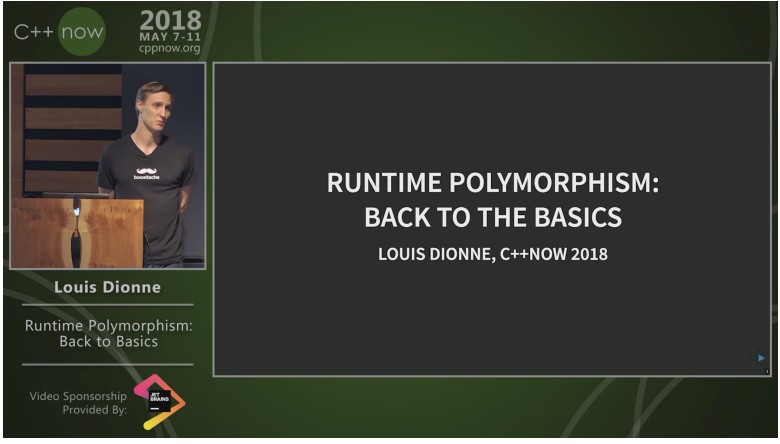
How can we build this?

Adding more overloads

https://youtu.be/gVGtNFg4ay0

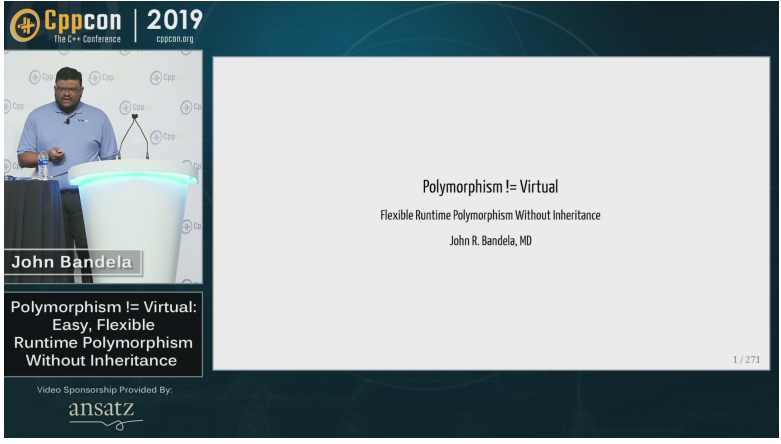
Adding more overloads

https://youtu.be/OtU51Ytfe04



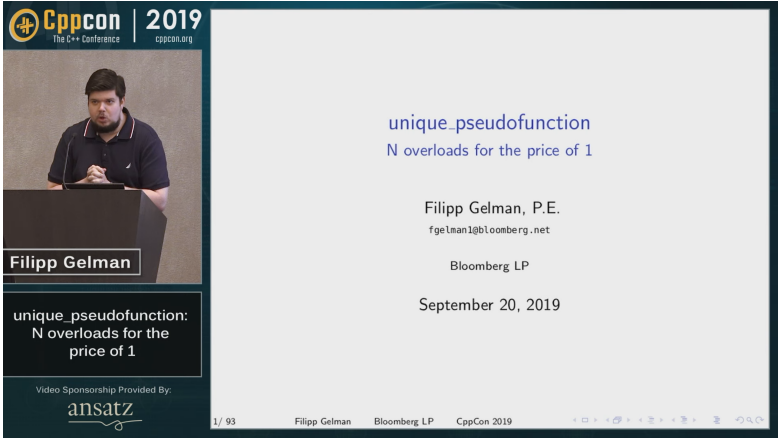
Adding more overloads

https://youtu.be/PSxo85L2lC0

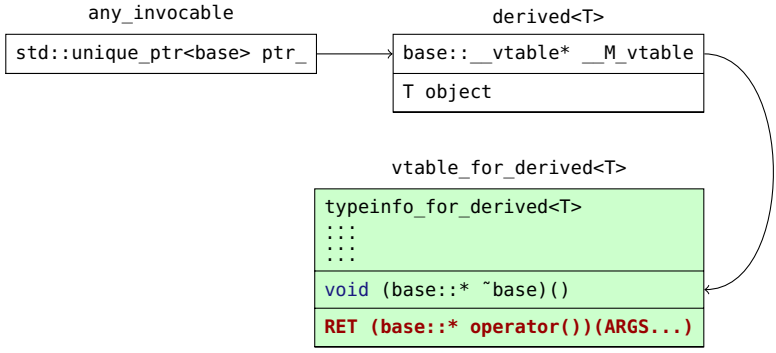


Adding more overloads

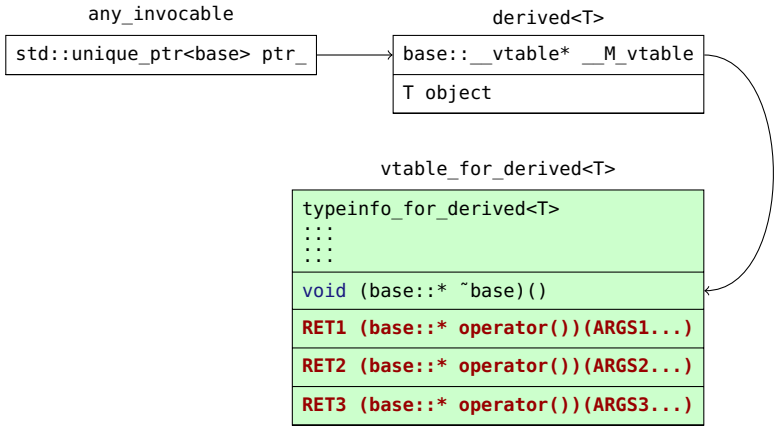
https://youtu.be/3Ms0gi5GfL0



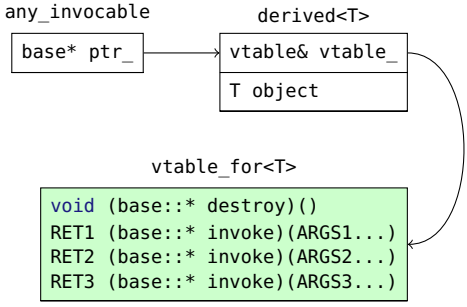
Adding more overloads



Adding more overloads



Adding more overloads



Adding more overloads

```
struct vtable {
    void (base::* destroy)();
    RET1 (base::* invoke1)(ARGS1...);
    RET2 (base::* invoke2)(ARGS2...);
    RET3 (base::* invoke3)(ARGS3...);
};
```

Adding more overloads

```
template <typename> struct vtable_entry;

template <typename RET, typename... ARGS>
struct vtable_entry<RET(ARGS...)> {
    using ptr_t = RET (base::*)(ARGS&&...);

    template <typename T>
    using is_invocable = std::is_invocable_r_v<RET, T&, ARGS&&...>;

    ptr_t const invoke;
};
```

Adding more overloads

```
struct vtable_dtor {
    using ptr_t = void (base::*)() noexcept;
    ptr_t const destroy;
};

struct vtable : vtable_dtor, vtable_entry<FNS...> {
    constexpr explicit vtable(
            vtable_dtor::ptr_t dtor,
            typename vtable_entry<FNS>::ptr_t... invoke) noexcept :
        vtable_dtor{dtor},
        vtable_entry<FNS>{invoke}... {}
};
```

Adding more overloads

```
template <typename... FNS>
class any_invocable {
    // template struct vtable_entry
    // struct vtable_dtor
    // struct vtable

    struct base {
        vtable const& vtable_;
    };

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

Adding more overloads

```
template <typename T>
struct derived : base {
    T object;

    void destroy() noexcept { delete this; }

    template <typename RET, typename... ARGS>
    RET operator()(ARGS... args) {
        return std::invoke(object, std::forward<ARGS>(args)...);
    }

    static inline constexpr struct vtable const vtable{ /* ... */ };

    // constructor
};
```

Adding more overloads

```
// Inside derived
static inline constexpr struct vtable const vtable{
    &derived::destroy,
    &derived::operator()<RET1, ARGS1&&...>,
    &derived::operator()<RET2, ARGS2&&...>,
    &derived::operator()<RET3, ARGS3&&...>};
```

Adding more overloads

```
// Inside derived
static inline constexpr struct vtable const vtable{
    static_cast<vtable_dtor::ptr_t>(&derived::destroy),
    static_cast<typename vtable_entry<FNS>::ptr_t>(
        &derived::operator())...};
```

Adding more overloads

```
// Inside derived, converting constructor
explicit derived(T object) :
    base{vtable},
    object(std::move(object)) {}
```

Adding more overloads

```
template <typename... FNS>
class any_invocable {
    // template struct vtable_entry
    // struct vtable_dtor
    // struct vtable
    // struct base
    // template struct derived

    base* ptr_;
public:
    any_invocable() noexcept;
    any_invocable(any_invocable&& other) noexcept;
    any_invocable& operator=(any_invocable&& other) noexcept;
    ~any_invocable();
    // ...
};
```

Adding more overloads

```
any_invocable() noexcept : ptr_(nullptr) {}

any_invocable(any_invocable&& other) noexcept :
    ptr_(std::exchange(other.ptr, nullptr)) {}

any_invocable& operator=(any_invocable&& other) noexcept {
    any_invocable(std::move(other)).swap(*this);
    return *this;
}

~any_invocable() {
    if (ptr_) (ptr_->*(ptr_->vtable_.destroy))();
}
```

Adding more overloads

```
template <typename... FNS>
class any_invocable {
    // ...
public:
    // constructors, assignment operators

    RET1 operator()(ARGS1...);
    RET2 operator()(ARGS2...);
    RET3 operator()(ARGS3...);
    // ...
};
```

Can we build this?

Adding more overloads

```
template <typename... FNS>
class any_invocable {
    // ...
public:
    // constructors, assignment operators

    RET1 operator()(ARGS1...);
    RET2 operator()(ARGS2...);
    RET3 operator()(ARGS3...);
    // ...
};
```

~~Can we build this?~~

How can we build this?

Adding more overloads

```
RET1 operator()(ARGS1... args) {
    // 1. Get the vtable.
    vtable const& vt = ptr_->vtable;

    // 2. Get the function pointer.
    RET1 (base::* invoke)(ARGS1...) =
        vt.vtable_entry<RET1(ARGS1...)>::invoke;

    // 3. Call that function on the base.
    return (ptr_->*invoke)(std::forward<ARGS>(args)...);
}
```

Adding more overloads

```
RET1 operator()(ARGS1... args) {
    // 1. Get the vtable.
    // 2. Get the function pointer.
    // 3. Call that function pointer on the base.
    return (ptr_->*(ptr_->vtable::vtable_entry<RET1(ARGS1...)>::invoke))(
        std::forward<ARGS1>(args)...);
}
```

Adding more overloads

```
template <typename... FNS>
class any_invocable {
    // ...
    RET1 operator()(ARGS1... args) {
        return (ptr_->*(ptr_->vtable_.vtable_entry<RET1(ARGS1...)>::invoke))(
            std::forward<ARGS1>(args)...);
    }


    RET2 operator()(ARGS2... args) {
        return (ptr_->*(ptr_->vtable_.vtable_entry<RET2(ARGS2...)>::invoke))(
            std::forward<ARGS1>(args)...);
    }
};
```

Adding more overloads

```
template <typename RET, typename... ARGS>
struct invocable_interface<RET(ARGS...)> {
    RET operator()(ARGS... args) {
        return (ptr_->*(ptr_->vtable::vtable_entry<RET1(ARGS1...)>::invoke))(
            std::forward<ARGS1>(args)...);
    }
};
```

How can `invocable_interface` access `ptr_` from `any_invocable`?

Adding more overloads

CRTP!

```
template <typename, typename>
struct invocable_interface;

template <typename RET, typename... ARGS, typename... FNS>
struct invocable_interface<RET(ARGS...), any_invocable<FNS...>> {
    RET operator()(ARGS... args) {
        any_invocable<FNS...>& self =
            static_cast<any_invocable<FNS...>&>(*this);

        return (self.ptr_->*(self.ptr_->vtable_.vtable_entry<RET(ARGS...)>::invoke))(
            std::forward<ARGS>(args)...)
    }
};
```

Adding more overloads

```
template <typename... FNS>
class any_invocable : invocable_interface<FNS, any_invocable<FNS...>>... {
    //...

    template <typename, typename>
    friend struct invocable_interface;
public:
    // ...

    using invocable_interface<FNS, any_invocable<FNS...>>::operator()...;
};
```

Adding more overloads

```
template <typename... FNS>
class any_invocable : invocable_interface<FNS, any_invocable<FNS...>>... {
    //...

public:
    /**
     * RET1 operator()(ARGS1... args) {
     *     return (ptr_->*ptr_->vtable.@vtable_entry<FNS1>::invoke@)(
     *         std::forward<ARGS1>(args)...);
     * }
     *
     *
     * RET2 operator()(ARGS2... args) {
     *     return (ptr_->*ptr_->vtable.@vtable_entry<FNS2>::invoke@)(
     *         std::forward<ARGS2>(args)...);
     * }
     */

    using invocable_interface<FNS, any_invocable<FNS...>>::operator()...;
};
```

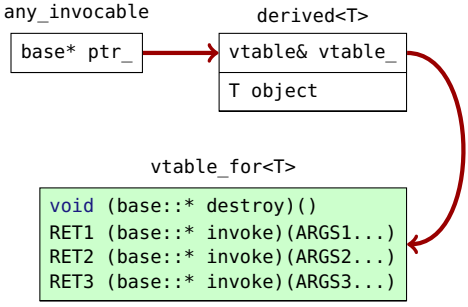
Adding more overloads

```
template <typename... FNS>
class any_invocable : invocable_interface<FNS, any_invocable<FNS...>>... {
    // ...

public:
    // ...
    template <typename T, typename = std::enable_if_t<
        (vtable_entry<FNS>::template is_invocable<T>::value && ...)>>
    any_invocable(T object) :
        ptr_(new derived<T>(std::move(object))) {}
};
```

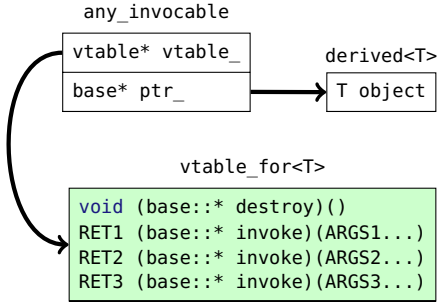
Adding more overloads

```
template <typename... FNS>
class any_invocable : invocable_interface<FNS, any_invocable<FNS...>>... {
   /* ... */

public:
    any_invocable() noexcept;
    any_invocable(any_invocable&&) noexcept;
    any_invocable& operator=(any_invocable&&) noexcept;
    ~any_invocable();

    template</* ... */>
    any_invocable(T object);

    using invocable_interface<FNS, any_invocable<FNS...>>::operator()...;
};
```

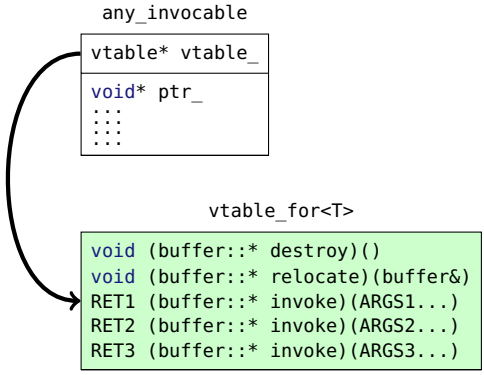
Adding more overloads



Adding more overloads



Small Object Optimization



Small Object Optimization

- Many objects are small and cheap to move.
- Heap allocations can be expensive.

Small Object Optimization

```
union buffer {
    char here_[N];
    void* there_;

    // fits<T>()
    // construct<T>(from)
    // destroy<T>()
    // relocate<T>(to)
    // invoke<T, RET, ARGS...>(...)
}
```

Small Object Optimization

```
template <typename T>
constexpr bool fits() noexcept {
    return sizeof(T) <= sizeof(buffer)
        && alignof(T) <= alignof(buffer)
        && std::is_nothrow_move_constructible_v<T>
        && std::is_nothrow_destructible_v<T>;
}
```

Small Object Optimization

```
union buffer {
    // ...
    template <typename T>
    void construct(T object) {
        if constexpr (fits<T>()) {
            new (here_) T(std::move(object));
        } else {
            there_ = new T(std::move(object));
        }
    }
    // ...
}
```

Small Object Optimization

```
union buffer {
    // ...
    template <typename T>
    void destroy() noexcept {
        if constexpr (fits<T>()) {
            reinterpret_cast<T&>(here_).~T();
        } else {
            delete static_cast<T*>(there_);
        }
    }
    // ...
}
```

Small Object Optimization

```
union buffer {
    // ...
    template <typename T>
    void relocate(buffer& dest) noexcept {
        if constexpr (fits<T>()) {
            new (dest.here_) T(reinterpret_cast<T&&>(here_));
            reinterpret_cast<T&>(here_).~T();
        } else {
            dest.there_ = there_;
        }
    }
    // ...
}
```

Small Object Optimization

```
union buffer {
    // ...
    template <typename T, typename RET, typename... ARGS>
    RET invoke(ARGS... args) {
        if constexpr (fits<T>()) {
            return std::invoke(reinterpret_cast<T&>(here_),
                std::forward<ARGS>(args)...);
        } else {
            return std::invoke(*static_cast<T*>(there_),
                std::forward<ARGS>(args)...);
        }
    }
}
```

Small Object Optimization

`buffer` does not enforce semantics!

```
buffer b;
b.construct<Foo>(Foo{});
b.invoke<Foo, void, int>(42);
b.invoke<Foo, void, long, char>(10l, 'a');
b.destroy<Foo>();
```

Small Object Optimization

```
buffer a, b;
a.construct<Foo>(Foo{});
a.relocate<Foo>(b);
b.destroy<Foo>();
// NO a.destroy<Foo>()!!!
```

Small Object Optimization

```
template <typename RET, typename... ARGS>
struct vtable_entry<RET(ARGS...)> {
    using ptr_t = RET (buffer::*)(ARGS&&...);

    // is_invocable
    // invoke
};
```

Small Object Optimization

```
struct vtable_dtor {
    using ptr_t = void (buffer::*)() noexcept;
    ptr_t const destroy;
};

struct vtable_rltor {
    using ptr_t = void(buffer::*)(buffer&) noexcept;
    ptr_t const relocate;
};
```

Small Object Optimization

```
struct vtable : vtable_dtor, vtable_rltor, vtable_entry<FNS...> {
    constexpr explicit vtable(
            vtable_dtor::ptr_t dtor,
            vtable_rltor::ptr_t rltor,
            typename vtable_entry<FNS>::ptr_t... entry) noexcept :
        vtable_dtor{dtor},
        vtable_rltor{rltor},
        vtable_entry<FNS>{entry}... {}
};
```

Small Object Optimization

```
// Inside any_invocable
template <typename T>
static inline constexpr vtable const vtable_for{
    &buffer::destroy<T>,
    &buffer::relocate<T>,
    static_cast<typename vtable_entry<FNS>::ptr_t>(
        &buffer::invoke<T>)...};
```

Small Object Optimization

```
template <typename... FNS>
class any_invocable {
    // template struct vtable_entry
    // struct vtable_dtor
    // struct vtable_rltor
    // struct vtable
    // template constexpr vtable vtable_for

    vtable const* vtable_;
    buffer buffer_;

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

Small Object Optimization

```
any_invocable() noexcept : vtable_(nullptr) {}

template <typename T /* , ... */>
any_invocable(T object) : vtable_(&vtable_for<T>) {
    buffer_.construct(std::move(T));
}

~any_invocable() {
    if (vtable_) (buffer_.*(vtable_->destroy))();
}
```

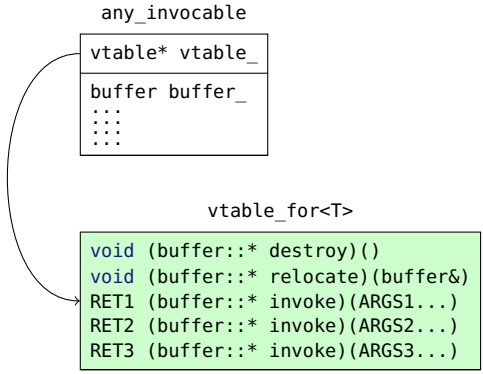
Small Object Optimization

```
any_invocable(any_invocable&& other) noexcept :
        vtable_(std::exchange(other.vtable_, nullptr)) {
    if (vtable_) (other.buffer_.*(vtable_->relocate))(buffer_);
}

any_invocable& operator=(any_invocable&& other) noexcept {
    if (vtable_) (buffer_.*(vtable_->destroy))();
    vtable_ = std::exchange(other.vtable_, nullptr);
    if (vtable_) (other.buffer_.*(vtable_->relocate))(buffer_);
}
```

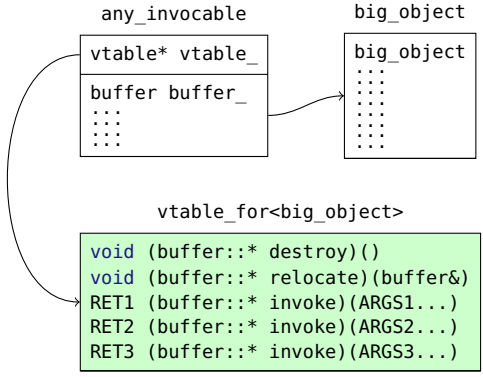
Small Object Optimization

```
template <typename RET, typename... ARGS, typename... FNS>
struct invocable_interface<RET(ARGS...), any_invocable<FNS...>> {
    RET operator()(ARGS... args) {
        any_invocable<FNS...>& self =
            static_cast<any_invocable<FNS...>&>(*this);
        // Call the correct vtable_entry.
        return (self.buffer_.*(self.vtable_->vtable_entry<RET(ARGS...)>::invoke))(
            std::forward<ARGS>(args)...);
    }
};
```

Small Object Optimization



Small Object Optimization



Standardize?

Disclaimer: opinions are my own.

Should overloaded `any_invocable` be in the standard library?

No.

Standardize?

`http://wg21.link/p0288`

`any_invocable` is ready.

Extending now might delay.

Extending later might break.

`lock_guard` vs `scoped_lock`.

No 99% solution.

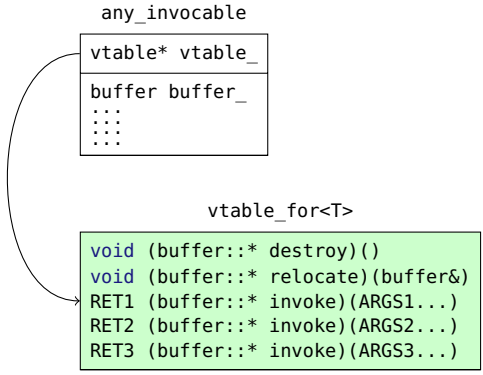
Standardize?

```
union buffer {
    char here_[N];
    void* there_;
};
```

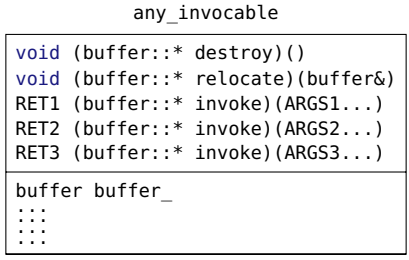
No “correct” value for N.

Baked into the ABI, can’t change.

Standardize?



Standardize?



Standardize?

Should overloaded `any_invocable` be in the **standard** library? **No**.

Should overloaded `any_invocable` be in **another** library? **Maybe**.

Standardize?

Louis Dionne's Dyno: `https://github.com/ldionne/dyno`

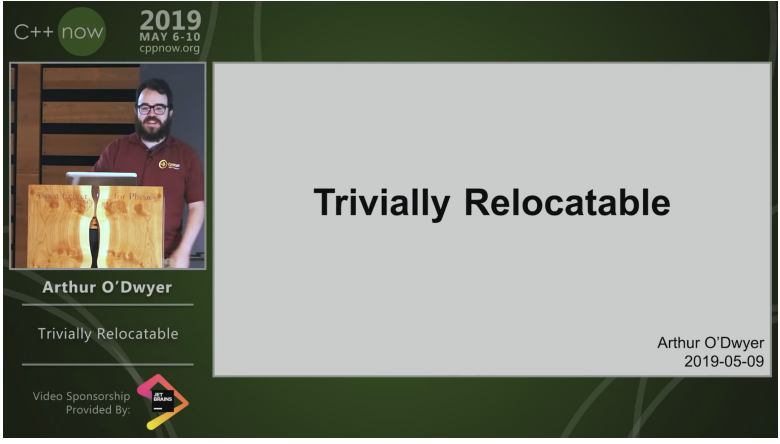
Folly Poly: `https://github.com/facebook/folly/blob/master/folly/docs/Poly.md`

Abseil: `https://github.com/abseil/abseil-cpp/tree/master/absl/functional`

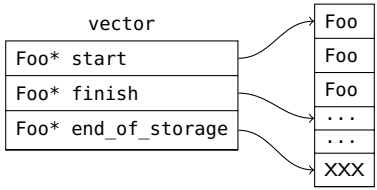
Boost: `https://github.com/boostorg/function`

Trivially Relocatable

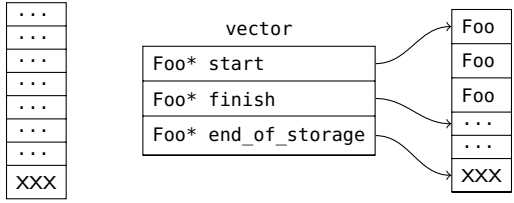
https://youtu.be/SGdfPextuAU



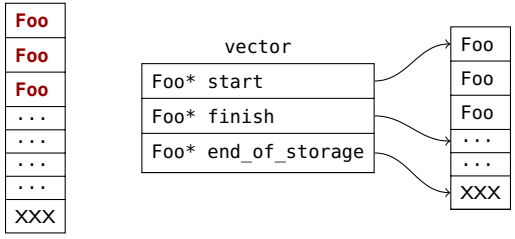
Trivially Relocatable



Trivially Relocatable

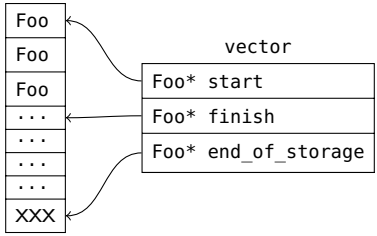


Trivially Relocatable



Trivially Relocatable

Trivially Relocatable

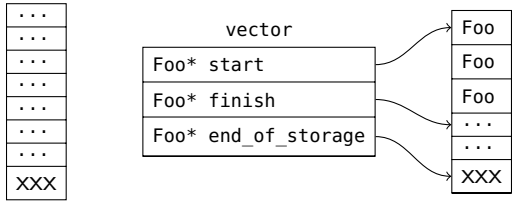


Trivially Relocatable

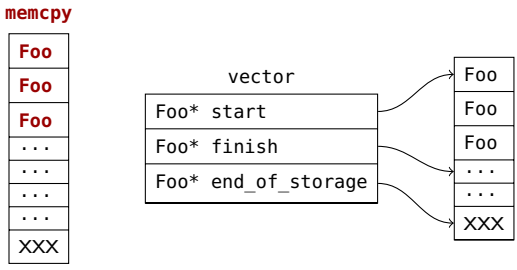
`https://wg21.link/p1144`

Move + Destroy = `memcpy`

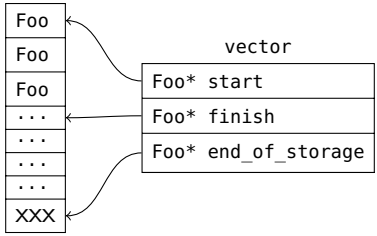
Trivially Relocatable



Trivially Relocatable



Trivially Relocatable



Trivially Relocatable

```
template <typename T>
constexpr bool fits() noexcept {
    return sizeof(T) <= sizeof(buffer)
        && alignof(T) <= alignof(buffer)
        && is_trivially_relocatable_v<T>;
}
```

Trivially Relocatable

```
any_invocable(any_invocable&& other) noexcept :
        vtable_(std::exchange(other.vtable_, nullptr)) {
    if (vtable_) buffer_ = other.buffer_;
}

any_invocable& operator=(any_invocable&& other) noexcept {
    if (vtable_) (buffer_.*vtable->destroy)();
    vtable_ = std::exchange(other.vtable_, nullptr);
    if (vtable_) buffer_ = other.buffer_;
}
```

Trivially Relocatable

```
template <typename... FNS>
class [[trivially_relocatable]] any_invocable {
    // ...
};
```

Trivially Relocatable

P1144R5
Object relocation in terms of move plus destroy

Published Proposal, 2020-03-01

Issue Tracking:
Inline In Spec

Author:
Arthur O'Dwyer

Audience:
LEWG, EWG

Project:
ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++

Current Source:
github.com/Quuxplusone/draft/blob/gh-pages/d1144-object-relocation.bs

Current:
rawgit.com/Quuxplusone/draft/gh-pages/d1144-object-relocation.html

---

`https://wg21.link/p1144`

Optimization Opportunity

A queue of `any_invocable` is a common use case.

```
std::queue<any_invocable<void()>> queue;

queue.push_back([](){ printf("Hello,"); });
queue.push_back([](){ printf("world!"); });
// ...
queue.push_back([large_state](){ use(large_state); });
```

Optimization Opportunity

```
class WorkQueue {
    std::queue<any_invocable<void(),
               std::list<any_invocable<void()>>> queue_;
    std::mutex mutex_;
    std::condition_variable cond_;

public:
    void push(any_invocable<void()>&& work);
    any_invocable pop();
};
```

Optimization Opportunity

```
void push(any_invocable<void()>&& work) {
    {
        std::lock_guard<std::mutex> lock(mutex_);  // mutex_.lock();
        queue_.push(std::move(work));
            // 1. Allocate memory for element.
            // 2. Move work into that memory.
            // 3. Connect element to container.
        // mutex_.unlock();
    }
    cond_.notify_one();
}
```

Optimization Opportunity

```
any_invocable<void()> pop() {
    any_invocable<void()> ret;
    std::unique_lock<std::mutex> lock(mutex_);  // mutex_.lock();
    while (queue_.empty()) cond_.wait(lock);
    ret = std::move(queue_front());  // Move work into ret.
    queue_.pop();
        // 1. Disconnect element from container.
        // 2. Deallocate memory.
    return ret;
    // mutex_.unlock();
}
```

Optimization Opportunity

- Allocates memory while holding `mutex_`.
- Deallocates memory while holding `mutex_`.
- **Critical region has operations that are not critical!**
- Large objects need separate memory allocation.

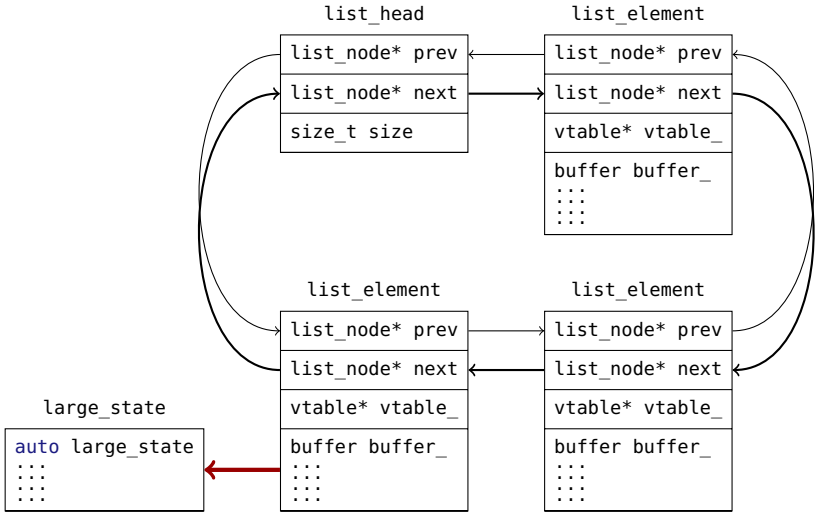
Can this be done better?

Optimization Opportunity

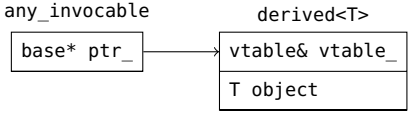
- Allocates memory while holding `mutex_`.
- Deallocates memory while holding `mutex_`.
- **Critical region has operations that are not critical!**
- Large objects need separate memory allocation.

~~Can this be done better?~~
How can this be done better?

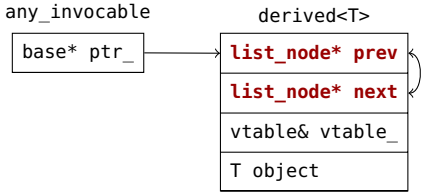
Intrusive Polymorphic Queue



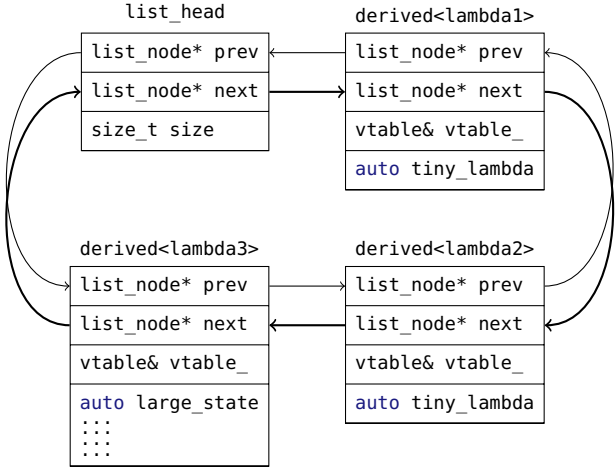
Intrusive Polymorphic Queue



Intrusive Polymorphic Queue



Intrusive Polymorphic Queue



Intrusive Polymorphic Queue

```
template <typename... FNS>
class any_invocable {
    // vtable_dtor
    // vtable_entry
    // vtable
    // base
    // derived

public:
    // special member functions
    // operator()

    class queue;
};
```

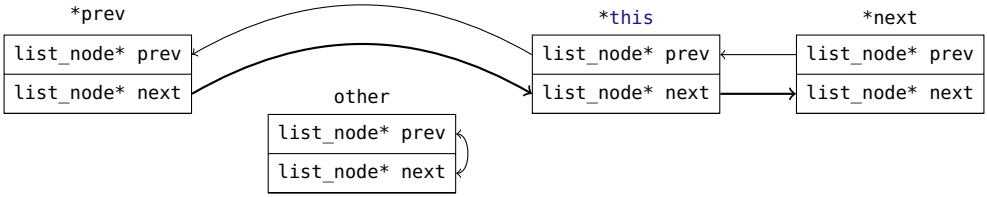
Intrusive Polymorphic Queue

```
class queue {
    // ...

public:
    bool empty() const noexcept;
    void push(any_invocable&&) noexcept;
    any_invocable pop() noexcept;
};
```

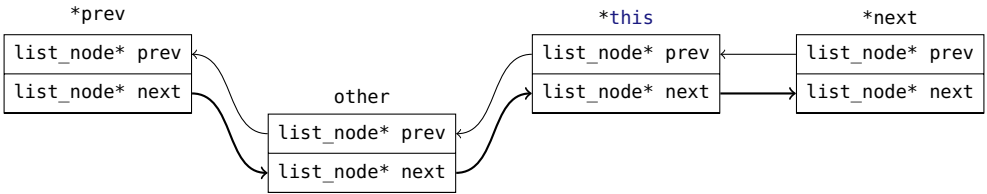
Intrusive Polymorphic Queue

```
struct list_node {
    list_node* prev = this;
    list_node* next = this;

    void hook(list_node& other) noexcept;
    void unhook() noexcept;
    bool is_hooked() const noexcept;
}
```

Intrusive Polymorphic Queue



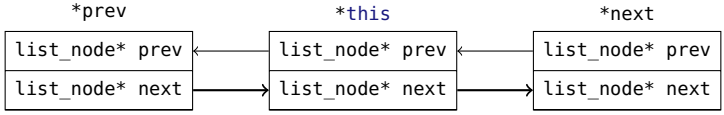
Intrusive Polymorphic Queue



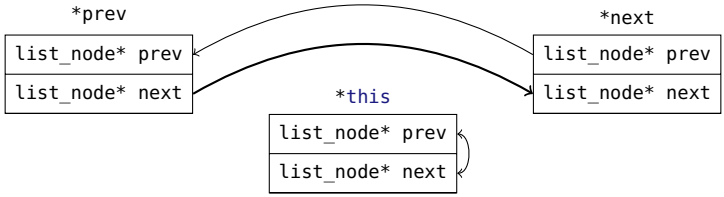
Intrusive Polymorphic Queue

```
void hook(list_node& other) noexcept {
    other.prev = prev;
    other.next = this;
    prev->next = prev = &other;
}
```

Intrusive Polymorphic Queue



Intrusive Polymorphic Queue



Intrusive Polymorphic Queue

```
void unhook() noexcept {
    prev->next = next;
    next->prev = prev;
    prev = next = this;
}
```

Intrusive Polymorphic Queue

```
bool is_hooked() const noexcept {
    return next != this;
    // return prev != this;
}
```

Intrusive Polymorphic Queue

```
struct list_node {
    // prev, next

    // Connect other to the left of *this.
    void hook(list_node& other) noexcept;

    // Disconnect *this.
    void unhook() noexcept;

    // Is *this connected to something?
    bool is_hooked() const noexcept;
}
```

Intrusive Polymorphic Queue

```
template <typename... FNS>
class any_invocable {
    // ...

    struct base : list_node {
        vtable const& vtable_;
    };

public:
    // ...

    class queue;
};
```

Intrusive Polymorphic Queue

```
class queue : list_node {
public:
    queue() noexcept;
    queue(queue&& other) noexcept;
    ~queue();

    bool empty() const noexcept;
    void push(any_invocable&& object) noexcept;
    any_invocable pop() noexcept;
};
```

Intrusive Polymorphic Queue

```
bool empty() const noexcept {
    return !is_hooked();
}

void push(any_invocable&& object) noexcept {
    base& data = *std::exchange(object.ptr_, nullptr);
    hook(data);
}
```

Intrusive Polymorphic Queue

```
// private:
any_invocable pop() noexcept {
    any_invocable ret;
    if (is_hooked()) {
        ret.ptr_ = static_cast<base*>(next);
        ret.ptr_->unhook();
    }
    return ret;
}
```

Intrusive Polymorphic Queue

```
class WorkQueue {
    any_invocable<void()>::queue queue_;
    std::mutex mutex_;
    std::condition_variable cond_;

public:
    void push(any_invocable<void()>&& work) noexcept;
    any_invocable pop() noexcept;
};
```

Intrusive Polymorphic Queue

```
void push(any_invocable<void()>&& work) noexcept {
    // Memory is already allocated and owned by work.
    {
        std::lock_guard<std::mutex> lock(mutex_);  // mutex_.lock();
        queue_.push(std::move(work));  // Modify a few pointers!
        // mutex_.unlock();
    }
    cond_.notify_one();
}
```

Intrusive Polymorphic Queue

```
any_invocable<void()> pop() noexcept {
    std::unique_lock<std::mutex> lock(mutex_);  // mutex_.lock();
    while (queue_.empty()) cond_.wait(lock);
    return queue_.pop();  // Modify a few pointers!
    // mutex_.unlock();
    // Memory is owned by returned any_invocable.
}
```

Intrusive Polymorphic Queue

- Memory allocated before locking `mutex_`.
- Memory freed after unlocking `mutex_`.
- **Critical region has only critical operations!**
- Each allocation is exactly large enough.