unique_pseudofunction

N overloads for the price of 1

Filipp Gelman, P.E.

fgelman1@bloomberg.net

Bloomberg LP

September 20, 2019

Outline

What is `unique_pseudofunction`?

It is `unique_function` with N overloads of `operator()` instead of 1.

```
unique_function      <void(int, float)>
// vs
unique_pseudofunction<void(int, float),
                      bool(char, double, long),
                      short(void)>
```

Why would you use it?

You have callable objects that do many things.

```
struct Minus {
    int operator()(int x) { return -x; }
    int operator()(int x, int y) { return x - y; }
};

struct Plus {
    int operator()(int x) { return x; }
    int operator()(int x, int y) { return x + y; }
};
```

Why would you use it?

You want their interface, not their type.

```
using Operator =
    unique_pseudofunction<int(int),
                          int(int, int)>;

// A function prototype.
void use_operator(Operator const& op);
```

Elsewhere:

```
if (condition)
    use_operator(Plus{});
else
    use_operator(Minus{});
```

Why would you use it?

Your callable objects act on many types.

```
using Minus = std::minus<void>;

using Plus = std::plus<void>;
```

Why would you use it?

You know what types you need.

```
using Operator = unique_pseudofunction<
        int   (int   , int),
        double(double, double)>;

void use_operator(Operator const& op) {
    int i{op(1, 2)};
    double d1{op(3.0, 4.0)};
    double d2{op(5, 6.0)};
}
```

Why would you use it?

You want a type erased variant visitor.

```
void visit(
    unique_pseudofunction<
        void(char  ),
        void(int   ),
        void(double)> const& vis,
    std::variant<char, int, double> const& var) {
    std::visit(vis, var);
}
```

Now `visit` does not need to be a template.

The visitor can be the result of `std::overload`:

`https://wg21.link/p0051`

What problems does it solve?

`std::function` doesn't satisfy my requirements:

- it requires *`CopyConstructible`*
- it has only one overload of `operator()`

`std::function` requires *CopyConstructible*

```
auto lambda =
    [foo = std::make_unique<Foo>()]() {
        (*foo)();
    };

std::function<void(void)> fun(std::move(lambda));
```

`std::function` requires *CopyConstructible*

This doesn't work:

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

It can be worked around:

```
auto lambda =
    [foo = std::make_shared<Foo>()]() {
        (*foo)();
    };
```

But isn't this horrible? I **like** unique ownership.

Class `MyUniqueFunction`

Let's write a class to fix this problem:

```
class MyUniqueFunction {
    struct Base;  // Abstract base.
    template <typename T> struct Derived;  // Will override.

    std::unique_ptr<Base> ptr_;

public:
    template <typename T>
    explicit MyUniqueFunction(T&&);
    void operator()() const;
};
```

Class MyUniqueFunction

The nested structs in MyUniqueFunction:

```
struct Base {
    virtual void operator()() = 0;
    virtual ~Base() noexcept = default;
};

template <typename T>
struct Derived : Base {
    T value;

    template <typename U>
    explicit Derived(U&& value) :
        value(std::forward<U>(value)) {}

    void operator()() override {
        std::invoke(value);
    }
}
```

Class `MyUniqueFunction`

The public interface:

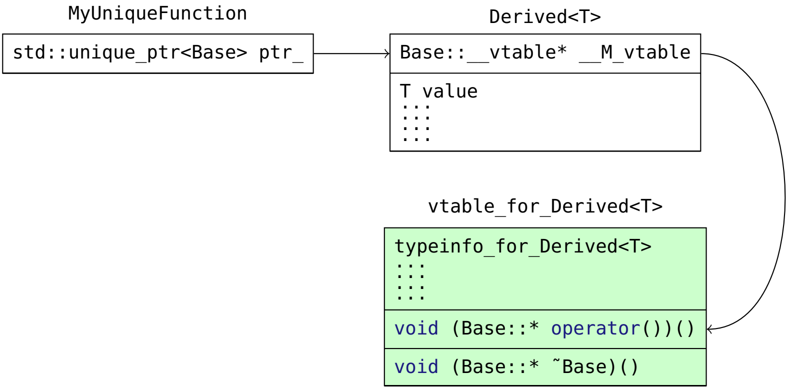
```
template <typename T>
explicit MyUniqueFunction(T&& value) : ptr_(
    std::make_unique<
        Derived<std::remove_cvref_t<T>>>(
            std::forward<T>(value))) {}

void operator()() const { (*ptr_)(); }
```

See Sean Parent's talk "Better Code: Runtime Polymorhpism":

https://youtu.be/QGcVXgEVMJg

Class MyUniqueFunction



Class `MyUniqueFunction`

Now I can use move-only objects!

```
MyUniqueFunction f(std::move(lambda));
```

But not implicitly:

```
MyUniqueFunction f = std::move(lambda);
```

```
source:99:99: error: conversion from '<lambda()>' to
        non-scalar type 'MyUniqueFunction' requested
    MyUniqueFunction f = [](){};
```

Class MyUniqueFunction

The problem is

```
/****************/
/**/ explicit /**/ MyUniqueFunction(T&& value)
/****************/
```

What if I remove it?

Class MyUniqueFunction

This causes another problem:

```
void test(int);
void test(MyUniqueFunction);
struct Test { operator int(); };

void test() {
    test(Test());
}
```

```
source:99:99 error: call of overloaded 'test(Test)' is ambiguous
    test(Test());
source:88:6: note: candidate 'void test(int)'
    'void test(int);'
source:89:6: note: candidate 'void test(MyUniqueFunction)'
    'void test(MyUniqueFunction);'
```

Class `MyUniqueFunction`

Let's fix it my making the constructor conditional:

```
template <typename T,
          typename = std::enable_if_t<
              std::is_invocable_v<
                  std::remove_cvref_t<T>>>>
MyUniqueFunction(T&& value)
```

Same effect, but using concepts:

```
template <typename T>
MyUniqueFunction(T&& value)
    requires std::is_invocable_v<std::remove_cvref_t<T>>
```

Now the following works:

```
test(Test());  // calls test(int)
test([](){});  // calls test(MyUniqueFunction)
```

Class `unique_function`

The class `MyUniqueFunction` implements only `unique_function<void(void)>`.

Let's implement `unique_function<RET(ARGS...)>`.

Class unique_function

```
template <typename>
class unique_function;

template <typename RET, typename... ARGS>
class unique_function<RET(ARGS...)> {
    struct Base;
    struct Derived;

    std::unique_ptr<Base> ptr_;

public:
    template <typename T /*, enable_if_t<...> */>
    unique_function(T&& value) /* or requires ... */;
    RET operator()(ARGS&&... args) const;
};
```

Class unique_function

The nested structs in unique_function:

```
struct Base {
    virtual RET operator()(ARGS...) = 0;
    virtual ~Base() noexcept = default;
};

template <typename T>
struct Derived : Base {
    T value;

    template <typename U>
    explicit Derived(U&& value) :
        value(std::forward<U>(value)) {}

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

Class `unique_function`

The public interface:

```
template <typename T, typename = std::enable_if_t<
    std::is_invocable_r_v<RET, std::remove_cvref_t<T>, ARGS...>>>
unique_function(T&& value) : ptr_(
    std::make_unique<
        Derived<std::remove_cvref_t<T>>>(
            std::forward<T>(value))) {}
}

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

Multiple overloads of `operator()`

Both `std::function` and `unique_function` have just one overload of `operator()`.

Can this implementation of `unique_function` be extended?

Multiple overloads of `operator()`

```
template <typename... FUNCS>
class unique_pseudofunction {
    struct Base {
        virtual ~Base() noexcept = default;
        /*
        virtual RET0 operator()(ARGS0...) = 0;
        virtual RET1 operator()(ARGS1...) = 0;
        ...
        */
    };
    // ...
};
```

This would be easy Herb Sutter's "Metaclass functions": `https://wg21.link/p0707`

I wasn't smart enough to do it in plain C++17.

What if we don't use `virtual`?

Class `unique_pseudofunction`

```
template <typename... FUNCS>
class unique_pseudofunction {
    void* data_;
    vtable<FUNCS...> const* vtable_;

public:
    // Public interface.
};
```

`vtable<FUNCS...>` will contain the function pointers from the hypothetical `Base`.

struct vtable

Let's declare some function templates. `vtable` will point to specific versions of these.

```
inline void destroy_impl(void*) noexcept {}

template <typename T>
void destroy_impl(void* data) noexcept {
    delete static_cast<T*>(data);
}

template <typename RET, typename... ARGS>
RET invoke_impl(void*, ARGS...) {
    throw std::bad_function_call();
}

template <typename T, typename RET, typename... ARGS>
RET invoke_impl(void* data, ARGS... args) {
    return std::invoke(*static_cast<T*>(data),
                       std::forward<ARGS>(args)...);
}
```

How will `vtable` destroy the object?

```
struct destroyer {
    void (&destroy)(void*) noexcept;

    constexpr destroyer() noexcept :
        destroy(destroy_impl) {}

    template <typename T>
    constexpr explicit destroyer(
            std::in_place_type_t<T> const&) noexcept :
        destroy(destroy_impl<T>) {}
};
```

struct vtable

How will `vtable` invoke the object?

```
template <typename>
struct vtable_entry;  // Not defined.

template <typename RET, typename... ARGS>
struct vtable_entry<RET(ARGS...)> {
    RET (&invoke)(void*, ARGS...);

    constexpr vtable_entry() noexcept :
        invoke(invoke_impl) {}

    template <typename T, typename = std::enable_if_t<
        std::is_invocable_r_v<RET, T, ARGS...>>>
    constexpr explicit vtable_entry(
            std::in_place_type_t<T> const&) noexcept :
        invoke(invoke_impl<T>) {}

    RET operator()(void* data, ARGS... args) const {
        return invoke(data, std::forward<ARGS>(args)...);
    }
};
```

Let's put them together:

```
template <typename... FUNCS>
struct vtable : destroyer, vtable_entry<FUNCS>... {
    constexpr vtable() noexcept = default;

    template <typename T, typename = std::enable_if_t<
        (... && std::is_constructible_v<
            vtable_entry<FUNCS>,
            std::in_place_type_t<T> const&>)>>
    constexpr explicit vtable(
            std::in_place_type_t<T> const& ipt) noexcept :
        destroyer(ipt), vtable_entry<FUNCS>(ipt)... {}

    using vtable_entry<FUNCS>::operator()...;
};
```

Let's statically allocate some `vtable`s:

```
template <typename... FUNCS>
constexpr inline vtable<FUNCS...> empty_vtable =
        vtable<FUNCS...>();

template <typename T, typename... FUNCS>
constexpr inline vtable<FUNCS...> vtable_for =
        vtable<FUNCS...>(std::in_place_type<T>);
```

Class `unique_pseudofunction`

Now let's implement `unique_pseudofunction`.
Here are the members and default constructor:

```
template <typename... FUNCS>
class unique_pseudofunction {
    void* data_;
    vtable<FUNCS...> const* vtable_;

public:
    unique_pseudofunction() noexcept :
        data_(nullptr),
        vtable_(&empty_vtable<FUNCS...>) {}

    // Continues on next slide.
```

Class `unique_pseudofunction`

The move constructor:

```
// Continues from previous slide.
unique_pseudofunction(
        unique_pseudofunction const&) = delete;

unique_pseudofunction(
        unique_pseudofunction&& other) noexcept :
    data_(std::exchange(other.data_, nullptr)),
    vtable_(std::exchange(other.vtable_,
                          &empty_vtable<FUNCS...>)) {}

// Continues on next slide.
```

Class `unique_pseudofunction`

The move assignment operator:

```
    // Continues from previous slide.
    unique_pseudofunction& operator=(
            unique_pseudofunction const&) = delete;

    unique_pseudofunction& operator=(
            unique_pseudofunction&& other) noexcept {
        unique_pseudofunction(std::move(other)).swap(*this);
        return *this;
    }
    // Continues on next slide.
```

Class `unique_pseudofunction`

The destructor and swap:

```
// Continues from previous slide.
~unique_pseudofunction() noexcept {
    vtable_->destroy(data_);
}

void swap(unique_pseudofunction& other) noexcept {
    std::swap(data_, other.data_);
    std::swap(vtable_, other.vtable_);
}
// Continues on next slide.
```

Class `unique_pseudofunction`

The converting constructor:

```
// Continues from previous slide.
template <typename T, typename = std::enable_if_t<
    /* ... */>>
unique_pseudofunction(T&& data) :
    data_(new std::remove_cvref_t<T>(std::forward<T>(data))),
    vtable_(&vtable_for<std::remove_cvref_t<T>, FUNCS...>) {}
// Continues on the slide after next
```

Class `unique_pseudofunction`

Let's look at the conditions inside `enable_if_t`:

Turn this constructor off if `T` is a
(possibly cv-qualified) `unique_pseudofunction`:

```
!std::is_same_v<
    unique_pseudofunction,
    std::remove_cvref_t<T>>
```

Turn this constructor off unless `T` is invocable in all the right ways:

```
std::is_constructible_v<
    vtable<FUNCS...>,
    std::in_place_type_t<std::remove_cvref_t<T>> const&>>>
```

Class unique_pseudofunction

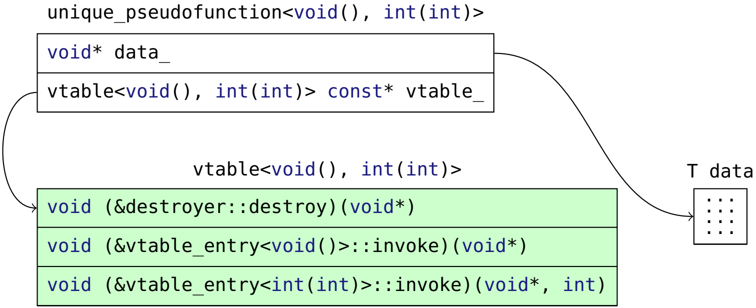
The call operator:

```
    // Continues from the slide before last
    template <typename... ARGS>
    decltype(std::declval<vtable<FUNCS...> const&>()(
        std::declval<void* const&>(),
        std::declval<ARGS>()...))
    operator()(ARGS&&... args) const {
        return (*vtable_)(data_,
                          std::forward<ARGS>(args)...);
    }
};  // close class unique_pseudofunction
```

See Vittorio Romeo's talk “You must type it three times”:

https://youtu.be/I3T4lePH-yA

Class `unique_pseudofunction`



Class `unique_pseudofunction`

We have implemented the class template `unique_pseudofunction`:

- Move only
- Multiple overloads of call operator
- Construct from any invocable object

`base64 -d | gzip -d`

Remaining Problems

Here are some remaining unsolved problems:

- Small object optimization
- Imperfect forwarding
- Compatible interoperability

Imperfect forwarding

Consider this example:

```
struct S {};
void overloaded( );
void overloaded(S);

void test() {
    overloaded({});
}
```

This correctly calls `overloaded(S)`.

Imperfect forwarding

But if we use `unique_pseudofunction`:

```
void test(unique_pseudofunction<
        void( ),
        void(S)> const& overloaded) {
    overloaded({});
}
```

```
source:99:99: error: no match for call to
    '(const unique_pseudofunction<void(), void(S)>)(
        <brace-enclosed initializer list>)'
```

The compiler tries to deduce `ARGS...` because `operator()` is a function template, not an overload set.

Imperfect forwarding

Let's turn `operator()` into an overload set.

```
template <typename RET, typename... ARGS>
struct vtable_entry<RET(ARGS...)> {
    /* ... */

    RET operator()(void*, ARGS...) const;
};
```

Remove this call operator.

Imperfect forwarding

```
template <typename... FUNCS>
struct vtable : destroyer, vtable_entry<FUNCS>... {
    /* ... */

    using vtable_entry<FUNCS>::operator()...;
};
```

Remove this too.

Imperfect forwarding

Add the CRTP-like base class `pseudo_overload`:

```
template <typename... FUNCS>
class unique_pseudofunction;

template <typename RET, typename... ARGS, typename... FUNCS>
struct pseudo_overload<RET(ARGS...), FUNCS...> {
    RET operator()(ARGS... args) const {
        unique_pseudofunction<FUNCS...> const& f =
            static_cast<unique_pseudofunction<FUNCS...> const&>(*this);
        return f.vtable_->vtable_entry<RET(ARGS...)>::invoke(
            f.data_,
            std::forward<ARGS>(args)...);
    }
};
```

Each `pseudo_overload<FUNC, FUNCS...>` directly calls `vtable_entry<FUNC>::invoke`!

Imperfect forwarding

```
template <typename... FUNCS>
class unique_pseudofunction : pseudo_overload<FUNCS, FUNCS...>... {
    template <typename...>
    friend struct pseudo_overload;
    // ...

public:
    // ...

    // Replace operator() template with
    using pseudo_overload<FUNCS, FUNCS...>::operator()...;
};
// Continues on next slide.
```

Imperfect forwarding

Now `operator()` is an overload set:

```
void test(unique_pseudofunction<
        void( ),
        void(S)> const& overloaded) {
    overloaded({});
}
```

This correctly calls `overloaded(S)`.

Imperfect forwarding

We have implemented the class template `unique_pseudofunction`:

- Move only
- Multiple overloads of call operator
  - With regular overload resolution
- Construct from any invocable object

`base64 -d | gzip -d`

Compatible Interoperability

Suppose I have a highly capable `unique_pseudofunction`:

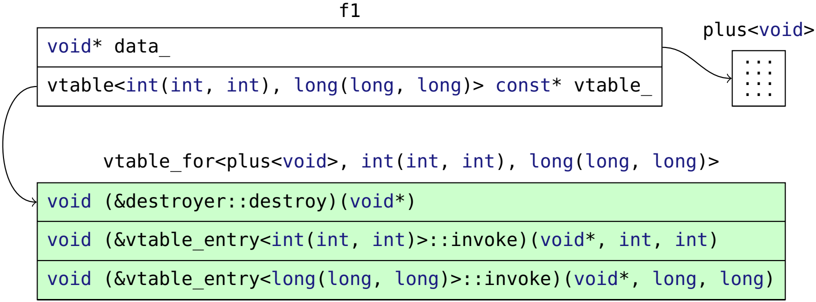
```
unique_pseudofunction<
        int (int,  int),
        long(long, long)> f1(std::plus<void>{});
```

Suppose also that I wish to store it in a less capable one:

```
unique_pseudofunction<int(int, int)> f2(std::move(f1));
```

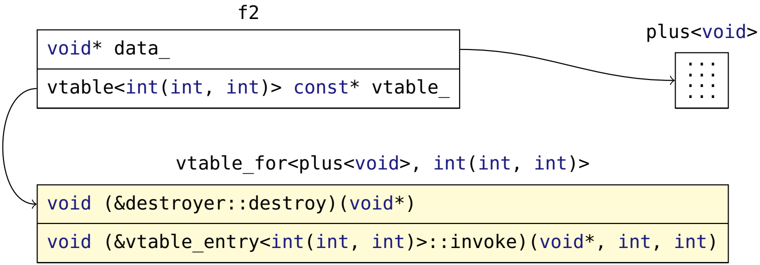
Compatible Interoperability

I start with `f1`:



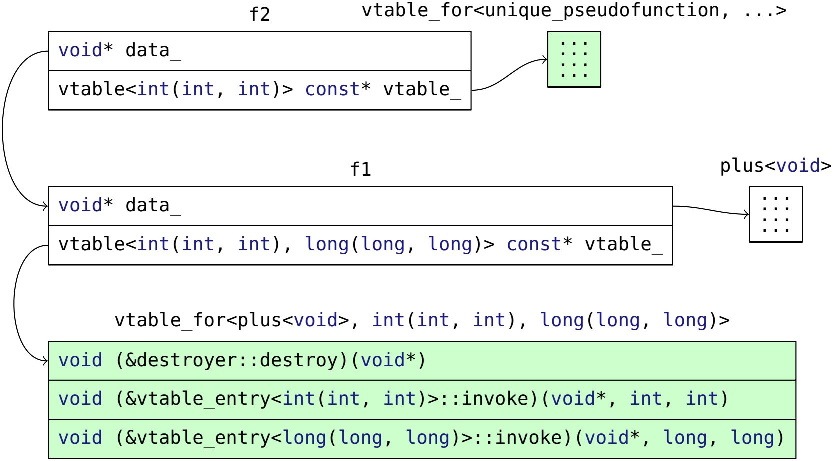
Compatible Interoperability

I want to get `f2`:



Compatible Interoperability

Instead I get:



Compatible Interoperability

- `vtable_` points to a statically allocated `vtable<int(int, int), long(long, long)>`.
- I want `f2.vtable_` to point to a new `vtable<int(int, int)>` that contains a subset of the function references from `*f1.vtable_`.
- This new `vtable<int(int, int)>` must also be statically allocated and constructed at compile time.
- *But to which `vtable(int(int, int), long(long, long)>` points `f1.vtable_` is not known until* **runtime!**

Compatible Interoperability

I don't know how to solve this while keeping statically allocated `vtable<...>`.

Input would be appreciated!

Instead, let's solve this by putting function pointers directly into `unique_pseudofunction`.

Compatible Interoperability

We will abuse multiple inheritance:

```
struct unique_lifetime;
template <typename> struct vtable_entry;
template <typename, typename...> struct pseudo_overload;
```

These are the base classes that will hold all the data members.

Compatible Interoperability

```
template <typename... FUNCS>
class unique_pseudofunction :
        unique_lifetime, pseudo_overload<FUNCS, FUNCS...>... {
    template <typename, typename...>
    friend struct pseudo_overload;
    template <typename...>
    friend class unique_pseudofunction;

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

    using pseudo_overload<FUNCS, FUNCS...>::operator()...;

    // swap and converting constructor later.
};
```

Compatible Interoperability

This part is common among all kinds of `unique_pseudofunction`:

```
struct unique_lifetime {
    void* data;
    void (*destroy)(void*) noexcept;

    unique_lifetime() noexcept :
        data(nullptr),
        destroy(destroy_impl) {}

    // Continues on next slide.
```

Compatible Interoperability

The move constructor:

```
    // Continues from last slide.

    unique_lifetime(unique_lifetime const&) = delete;

    unique_lifetime(unique_lifetime&& other) noexcept :
        data(std::exchange(other.data, nullptr)),
        destroy(std::exchange(other.destroy, destroy_impl)) {}

    // Continues on next slide.
```

Compatible Interoperability

The assignment operator:

```
    // Continues from last slide.
    unique_lifetime& operator=(unique_lifetime const&) = delete;

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

    // Continues on next slide.
```

Compatible Interoperability

Destroy and swap:

```
    // Continues from last slide.
    ~unique_lifetime() noexcept {
        destroy(data);
    }

    void swap(unique_lifetime& other) noexcept {
        std::swap(data, other.data);
        std::swap(destroy, other.destroy);
    }

    // Continues on next slide.
```

Compatible Interoperability

The converting constructor:

```
    // Continues from last slide.
    template <typename T, typename U>
    unique_lifetime(std::in_place_type_t<T> const&, U&& u) :
        data(new T(std::forward<U>(u))),
        destroy(destroy_impl<T>) {}
};
```

Compatible Interoperability

The `struct vtable_entry` now holds a function pointer instead of reference:

```
template <typename RET, typename... ARGS>
struct vtable_entry<RET(ARGS...)> {
    RET (*invoke)(void*, ARGS...);

    constexpr vtable_entry() noexcept :
        invoke(&invoke_impl) {}

    template <typename T, typename = std::enable_if_t<
        std::is_invocable_r_v<RET, T, ARGS...>>>
    constexpr explicit vtable_entry(
            std::in_place_type_t<T> const&) noexcept :
        invoke(&invoke_impl<T>) {}

    // Continues on next slide.
```

Compatible Interoperability

It also has move semantics:

```
    // Continues from last slide.
    vtable_entry(vtable_entry const&) = delete;

    constexpr vtable_entry(vtable_entry&& other) noexcept :
        invoke(std::exchange(other.invoke, &invoke_impl)) {}

    vtable_entry& operator=(vtable_entry const&) = delete;

    constexpr vtable_entry& operator=(
            vtable_entry&& other) noexcept {
        invoke = std::exchange(other.invoke, &invoke_impl);
        return *this;
    }

    void swap(vtable_entry& other) noexcept {
        std::swap(invoke, other.invoke);
    }
};
```

Compatible Interoperability

Let's store the `vtable_entry`s directly in `unique_pseudofunction` as part of each `pseudo_overload` base subobject:

```
template <typename RET, typename... ARGS, typename... FUNCS>
struct pseudo_overload<RET(ARGS...), FUNCS...> :
        vtable_entry<RET(ARGS...)> {
    using vtable_entry<RET(ARGS...)>::vtable_entry;

    constexpr pseudo_overload(
            vtable_entry<RET(ARGS...)>&& other) noexcept :
        vtable_entry<RET(ARGS...)>(std::move(other)) {}

    constexpr pseudo_overload& operator=(
            vtable_entry<RET(ARGS...)>&& other) noexcept {
        vtable_entry<RET(ARGS...)>::operator=(std::move(other));
        return *this;
    }

    // Continues on next slide.
```

Compatible Interoperability

```
    // Continues from last slide.

    RET operator()(ARGS... args) const {
        unique_pseudofunction<FUNCS...> const& f =
            static_cast<unique_pseudofunction<FUNCS...> const&>(*this);
        return vtable_entry<RET(ARGS...)>::invoke(
            f.data,
            std::forward<ARGS>(args)...);
    }
};
```

Compatible Interoperability

Now we finish `unique_pseudofunction`:

```
// Inside unique_pseudofunction.

void swap(unique_pseudofunction& other) noexcept {
    unique_lifetime::swap(other);
    (pseudo_overload<FUNCS, FUNCS...>::swap(other), ...);
}

// Continues on next slide.
```

Compatible Interoperability

This is the existing converting constructor:

```
// Continues from last slide.

template <typename T, typename = std::enable_if_t<
    !std::is_same_v<
        unique_pseudofunction,
        std::remove_cvref_t<T>> &&
    (... && std::is_constructible_v<
        vtable_entry<FUNCS>,
        std::in_place_type_t<std::remove_cvref_t<T>> const&>)>>
unique_pseudofunction(T&& data) :
    unique_lifetime(
            std::in_place_type<std::remove_cvref_t<T>>,
            std::forward<T>(data)),
    pseudo_overload<FUNCS, FUNCS...>(
            std::in_place_type<std::remove_cvref_t<T>>)... {}

// Continues on next slide.
```

Compatible Interoperability

This is the new converting constructor:

```
// Continues from last slide.

template <typename... OTHER_FUNCS, typename = std::enable_if_t<
    (... && std::is_base_of_v<
            vtable_entry<FUNCS>,
            unique_pseudofunction<OTHER_FUNCS...>>)>>
unique_pseudofunction(
        unique_pseudofunction<OTHER_FUNCS...>&& other) noexcept :
    unique_lifetime(std::move(static_cast<unique_lifetime&>(other))),
    pseudo_overload<FUNCS, FUNCS...>(
        std::move(static_cast<vtable_entry<FUNCS>&>(other)))... {}
```

Compatible Interoperability

Now the class looks like this:

```
unique_pseudofunction<void(), int(int)>
```

`void* data_`
`void (*unique_lifetime::destroy)(void*)`
`void (*vtable_entry<void()>::invoke)(void*)`
`void (*vtable_entry<int(int)>::invoke)(void*, int)`

`T data`

Compatible Interoperability

Converting between compatible `unique_pseudofunction`s now requires only changing pointers:

f2
`void* data_`
`void (*unique_lifetime::destroy)(void*)`
`void (*vtable_entry<void()>::invoke)(void*)`

T data

Compatible Interoperability

We have implemented the class template `unique_pseudofunction`:

- Move only
- Multiple overloads of call operator
  - With regular overload resolution
- Construct from any invocable object
- Efficiently convert to overload subset

base64 -d | gzip -d

Small object optimization

This `unique_pseudofunction` allocates everything on the heap.

Many objects are small and cheap to move.

We could keep them in place.

Small object optimization

Where will we keep the object?

```
union buffer {
    mutable char here[32];
    void* there;

    template <typename T>
    static constexpr bool fits() noexcept;

    template <typename T, typename... ARGS>
    void construct(ARGS&&... args);

    template <typename T>
    void destroy() noexcept;

    template <typename T>
    void move_to(buffer& other) noexcept;

    template <typename T>
    T& get() const noexcept;
};
```

Small object optimization

```
template <typename T>
static 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>;
}
```

A `T` fits only if it is small enough, not too aligned, and some operations don't throw.

Alternatively, I could use Arthur O'Dwyer's "relocate":

`https://wg21.link/p1144`

```
return is_trivially_relocatable_v<T>;
```

Small object optimization

Construct a `T` in the buffer if it fits, on the heap otherwise:

```
template <typename T, typename... ARGS>
void construct(ARGS&&... args) {
    if constexpr (fits<T>()) {
        new (here) T(st::forward<ARGS>(args)...);
    } else {
        there = new T(std::forward<ARGS>(args...));
    }
}

template <typename T>
void destroy() noexcept {
    if constexpr (fits<T>()) {
        reinterpret_cast<T*>(here)->˜T();
    } else {
        delete static_cast<T*>(there);
    }
}
```

Small object optimization

Move a `T` to another buffer:

```
template <typename T>
void move_to(buffer& other) noexcept {
    if constexpr (fits<T>()) {
        new (other.here)
            T(std::move(*reinterpret_cast<T*>(here)));
    } else {
        other.there = std::exchange(here, nullptr);
    }
}
```

Access the contained `T`:

```
template <typename T>
T& get() const noexcept {
    if constexpr (fits<T>()) {
        return *reinterpret_cast<T*>(here);
    } else {
        return *static_cast<T*>(there);
    }
}
```

Small object optimization

Now our functions take a `buffer` instead of `void*`:

```
template <typename RET, typename... ARGS>
RET invoke_impl(buffer const&, ARGS...) {
    throw std::bad_function_call();
}

template <typename T, typename RET, typename... ARGS>
RET invoke_impl(buffer const& data, ARGS... args) {
    return std::invoke(data.get<T>(),
                       std::forward<ARGS>(args)...);
}
```

Small object optimization

Change `destroy_impl` to `destroy_move_impl`:

```
inline void destroy_move_impl(
        buffer&,
        buffer*) noexcept {}

template <typename T>
void destroy_move_impl(
        buffer& from,
        buffer* to) noexcept {
    if (to) {
        from.move_to<T>(*to);
    }
    from.destroy<T>();
}
```

This would be simpler with `[[trivially_relocatable]]`.

Small object optimization

Let's change `unique_lifetime`:

```
// Continues from last slide.

struct unique_lifetime {
    buffer data_;
    void (*destroy_move)(buffer&, buffer*) noexcept;

    unique_lifetime() noexcept :
        destroy_move(&destroy_move_impl) {}

    // Continues on next slide.
```

Small object optimization

Move constructor:

```
// Continues from last slide.

unique_lifetime(unique_lifetime const&) = delete;

unique_lifetime(unique_lifetime&& other) noexcept :
    destroy_move_(std::exchange(
        other.destroy_move_,
        &destroy_move_impl)) {
    destroy_move_(other.data_, &data_);
}

// Continues on next slide.
```

Small object optimization

Move assignment:

```
    // Continues from last slide.

    unique_lifetime& operator=(
            unique_lifetime const&) = delete;

    unique_lifetime& operator=(
            unique_lifetime&& other) noexcept {
        if (&other != this) {
            destroy_move_(data_, nullptr);
            destroy_move_ =
                std::exchange(other.destroy_move_,
                              &destroy_move_impl_);
            destroy_move_(other.data_, &data);
        }
        return *this;
    }

    // Continues on next slide.
```

Small object optimization

The destructor:

```
    // Continues from last slide.

    ~unique_lifetime() noexcept {
        destroy_move_(data_, nullptr);
    }

    // Continues on next slide.
```

Small object optimization

The converting constructor:

```
    // Continues from last slide.
    template <typename T, typename... ARGS>
    explicit unique_lifetime(
        std::in_place_type_t<T> const&,
        ARGS&&... args) :
            destroy_move(destroy_move_impl<T>) {
        data_.construct<T>(std::forward<ARGS>(args)...);
    }
};
```

Small object optimization

`vtable_entry` doesn't change except for the type of the `invoke` data member.

`pseudo_overload` remains completely unchanged.

`unique_pseudofunction` must change slightly.

Small object optimization

The converting constructor:

```
// Inside unique_pseudofunction.

    template <typename T, typename = std::enable_if_t<
        /* ... */>>
    unique_pseudofunction(T&& data) :
        unique_lifetime(
                std::in_place_type<std::remove_cvref_t<T>>,
                std::forward<T>(data)),
        pseudo_overload<FUNCS, FUNCS...>(
                std::in_place_type<std::remove_cvref_t<T>>)... {}

// Continues on next slide.
```

Small object optimization

The interoperability constructor:

```
// Continues from last slide.

    template <typename... OTHER_FUNCS, typename = std::enable_if_t<
        /* ... */>>
    unique_pseudofunction(
            unique_pseudofunction<OTHER_FUNCS...>&& other) noexcept :
        unique_lifetime(std::move(
            static_cast<unique_lifetime&>(other))),
        pseudo_overload<FUNCS, FUNCS...>(
            std::move(static_cast<vtable_entry<FUNCS>&>(other)))... {}
```

Small object optimization

We have finally implemented the class template `unique_pseudofunction`:

- Move only
  - Potentially `[[trivially_relocatable]]`
- Multiple overloads of call operator
  - With regular overload resolution
- Construct from any invocable object
  - Store it in place when possible
- Efficiently convert to overload subset

Closing

Thanks to:

- Peter Muldoon for suggesting I present this class.
- Russel Seehafer for giving permission and tolerating my shenanigans.
- Others:
  - Bjarne Stroustrup
  - Sean Parent
  - Luis Dionne
  - Vittorio Romeo
  - Dietmar Kühl
- **YOU!**

base64 -d | gzip -d