Techniques for Overloading any_invocable

Filipp Gelman

C++ now
2021
MAY 2-7
Aspen, Colorado, USA

Techniques For Overloading `any_invocable`

C++Now 2021

May 5, 2021

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

Bloomberg Engineering

TechAtBloomberg.com

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

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;

    template <typename... CARGS>
    derived(CARGS&&... cargs) :
        object(std::forward<CARGS>(cargs)...) {}

    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<std::decay_t<T>>>(
        std::forward<T>(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>
any_invocable(T&& object)
    requires std::is_invocable_r_v<
        RET, std::decay_t<T>&, ARGS&&...> :
    ptr_(std::make_unique<derived<std::decay_t<T>>>(
        std::forward<T>(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?~~

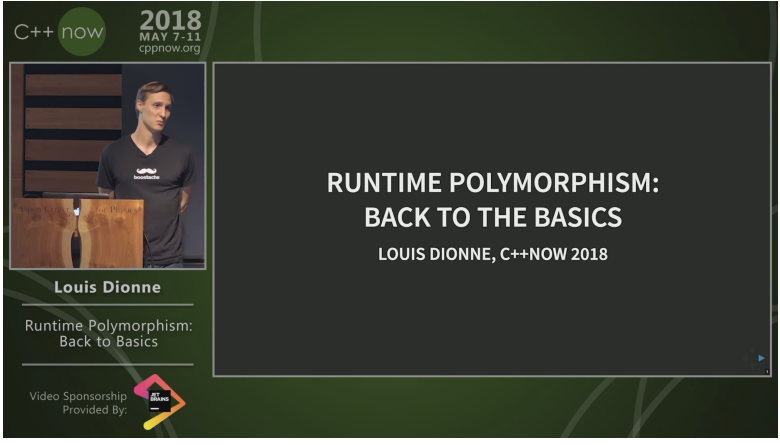
How can we build this?

Adding more overloads

https://youtu.be/gVGtNFg4ay0

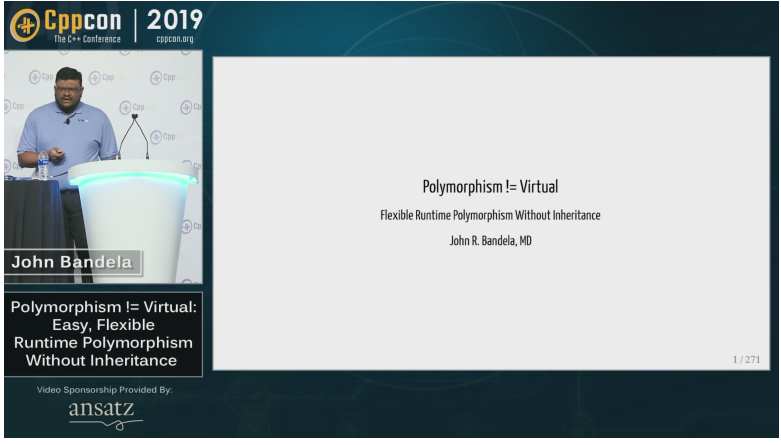
Adding more overloads

https://youtu.be/OtU51Ytfe04



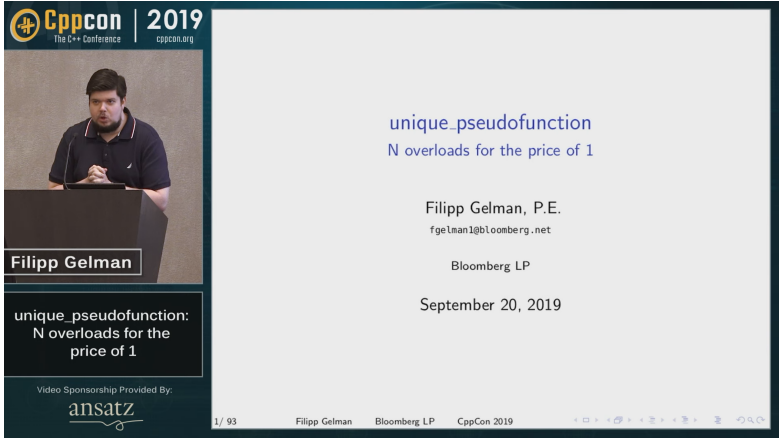
Adding more overloads

https://youtu.be/PSxo85L2lC0

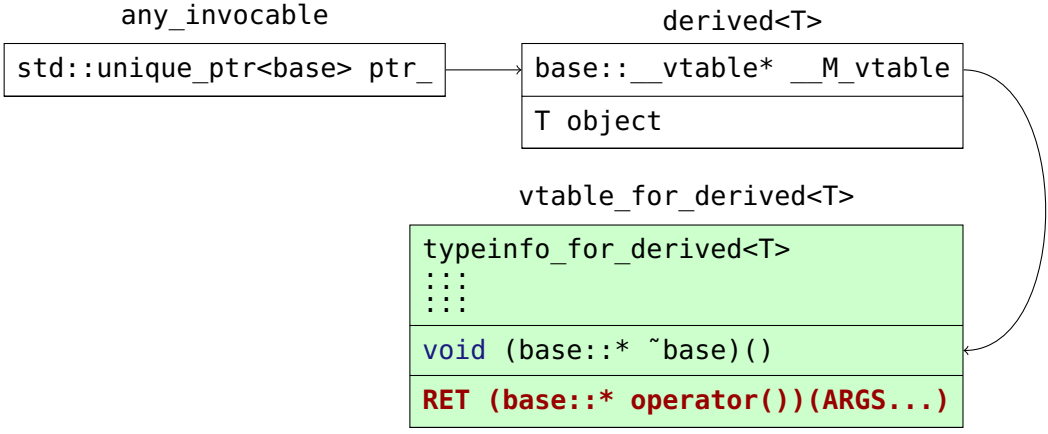


Adding more overloads

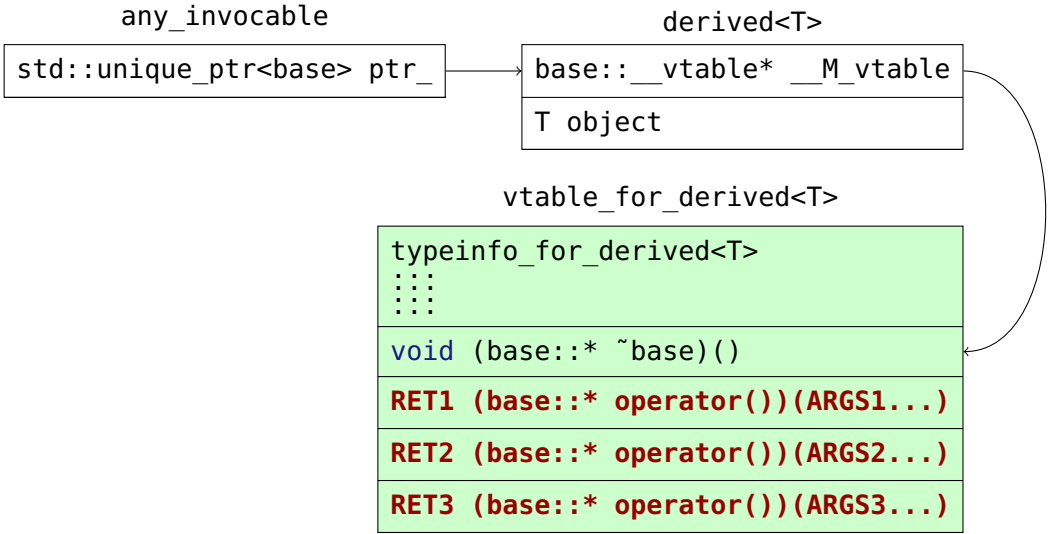
https://youtu.be/3Ms0gi5GfL0



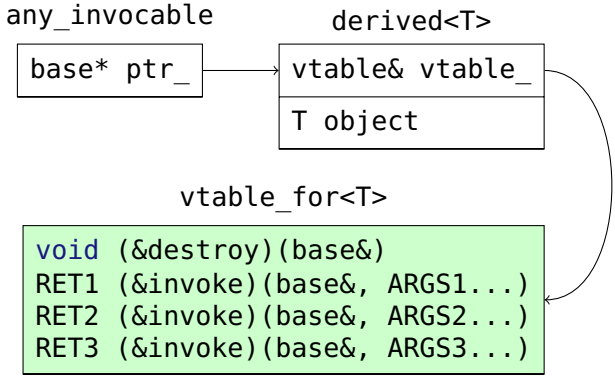
Adding more overloads



Adding more overloads



Adding more overloads



Adding more overloads

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

Adding more overloads

```
template <typename> struct vtable_entry;

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

    fun_t invoke;
};
```

Adding more overloads

```
struct vtable_dtor {
    using fun_t = void (&)(base&) noexcept;
    fun_t destroy;
};

struct vtable : vtable_dtor, vtable_entry<FNS...> {
    constexpr explicit vtable(
        vtable_dtor::fun_t dtor,
        typename vtable_entry<FNS>::fun_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;

    static void destroy(base&) noexcept;

    template <typename RET, typename... ARGS>
    static RET invoke(base&, ARGS... args);

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

    // constructor
};
```

Adding more overloads

```
// Inside derived
static void destroy(struct base& base) noexcept {
    delete &static_cast<derived&>(base);
}

template <typename RET, typename... ARGS>
static RET invoke(struct base& base, ARGS... args) {
    return std::invoke(
        static_cast<derived&>(base).object,
        std::forward<ARGS>(args)...);
}
```

Adding more overloads

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

Adding more overloads

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

Adding more overloads

```
// Inside derived, converting constructor
template <typename... CARGS>
derived(CARGS&&... cargs) :
    base{vtable},
    object(std::forward<CARGS>(cargs)...) {}
```

Adding more overloads

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

    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_->vtable_.destroy(*ptr_);
}
```

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 (& invoke)(base&, ARGS1...) =
        vt.vtable_entry<RET1(ARGS1...)>::invoke;

    // 3. Call that function on the base.
    return invoke(*ptr_, 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_->vtable::vtable_entry<RET1(ARGS1...)>::invoke(
        *ptr_, std::forward<ARGS1>(args)...);
}
```

Adding more overloads

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


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

Adding more overloads

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

Adding more overloads

```
// Inside invocable_interface
RET operator()(ARGS... args) {
    any_invocable<FNS...>& self =
        static_cast<any_invocable<FNS...>&>(*this);

    return self.ptr_->vtable_.vtable_entry<RET(ARGS...)>::invoke(
        *sefl.ptr_,
        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...>>... {
    //...

    /**
     * RET1 operator()(ARGS1... args)
     * RET2 operator()(ARGS2... args)
     */

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

Adding more overloads

```
template <typename, typename>
inline constexpr bool is_invocable_v = false;

template <typename T, typename RET, typename... ARGS>
inline constexpr bool is_invocable_v<T, RET(ARGS...)> =
    std::is_invocable_r_v<RET, T, ARGS&&...>;
};
```

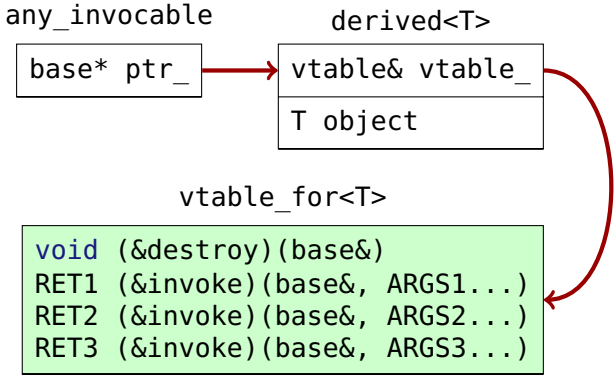
Adding more overloads

```
// Inside any_invocable
template <typename T>
any_invocable(T&& object)
    requires (is_invocable_v<std::decay_t<T>&, FNS> && ...):
    ptr_(new derived<std::decay_t<T>>(std::forward<T>(object))) {}
};
```

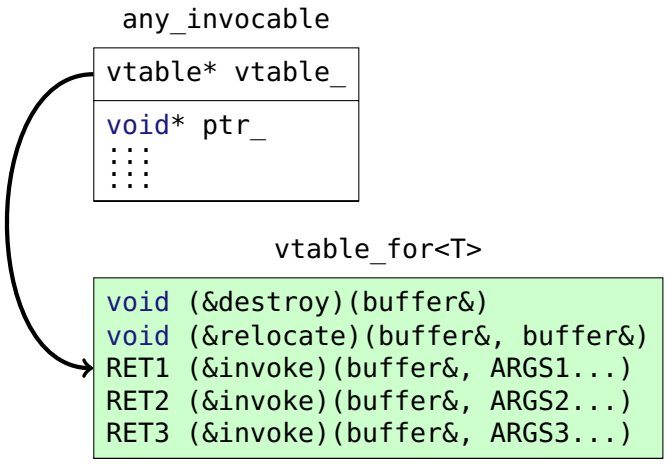
Adding more overloads

```
any_invocable() noexcept;
any_invocable(any_invocable&&) noexcept;
any_invocable& operator=(any_invocable&&) noexcept;
~any_invocable();

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

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

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>
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>;
}
```

Small Object Optimization

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

Small Object Optimization

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

Small Object Optimization

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

Small Object Optimization

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

Small Object Optimization

`buffer` does not enforce semantics!

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

Small Object Optimization

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

Small Object Optimization

```
template <typename RET, typename... ARGS>
struct vtable_entry<RET(ARGS...)> {
    using fun_t = RET (&)(buffer&, ARGS&&...);
    // ...
};
```

Small Object Optimization

```
struct vtable_dtor {
    using fun_t = void (&)(buffer&) noexcept;
    // ...
};

struct vtable_rltor {
    using fun_t = void(&)(buffer&, buffer&) noexcept;
    fun_t relocate;
};
```

Small Object Optimization

```
struct vtable : vtable_dtor, vtable_rltor, vtable_entry<FNS...> {
    constexpr explicit vtable(
            vtable_dtor::fun_t dtor,
            vtable_rltor::fun_t rltor,
            typename vtable_entry<FNS>::fun_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 {
    // ...

    vtable const* vtable_;
    buffer buffer_;

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

Small Object Optimization

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

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

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

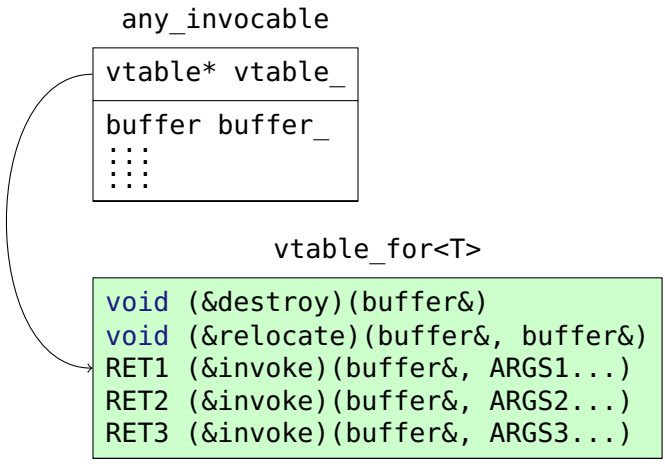
Small Object Optimization

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

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

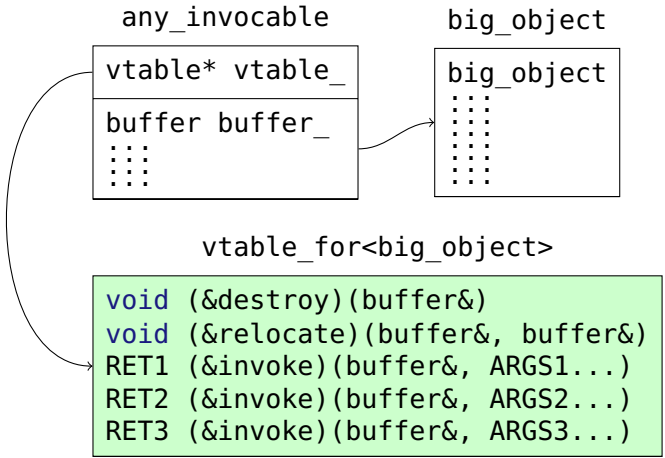
Small Object Optimization

```
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.vtable_->vtable_entry<RET(ARGS...)>::invoke
            (self.buffer_);
    }
};
```

Small Object Optimization

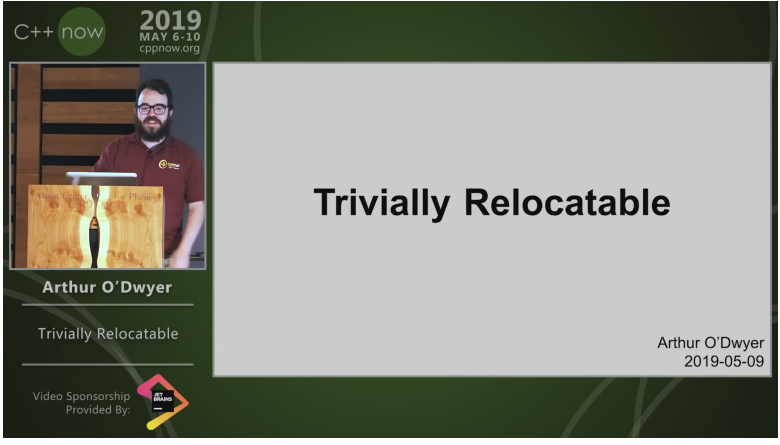


Small Object Optimization



Trivially Relocatable

https://youtu.be/SGdfPextuAU



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`

Disassembly

```
int test(std::function<int(int)> fun) {
    return fun(42);
}
```

Disassembly

```
test(std::function<int (int)>):
  subq $24, %rsp
  cmpq $0, 16(%rdi)  // <-- check for empty
  movl $42, 12(%rsp)
  je .L5
  leaq 12(%rsp), %rsi
  call *24(%rdi)  // <-- indirect call
  addq $24, %rsp
  ret
.L5:
  call std::__throw_bad_function_call()
```

Disassembly

```
int test(any_invocable<int(int)> fun) {
    return fun(42);
}
```

Disassembly

```
test(any_invocable<int (int)>):
  subq $24, %rsp
  movq (%rdi), %rax
  leaq 8(%rdi), %r8
  movl $42, 12(%rsp)
  leaq 12(%rsp), %rsi
  movq %r8, %rdi
  call *16(%rax)  // <-- indirect call
  addq $24, %rsp
  ret
```

Disassembly

```
long test(any_invocable<int(int), long(long)> fun) {
    return fun(42l);
}
```

Disassembly

```
test(any_invocable<int (int), long (long)>):
  subq $24, %rsp
  movq (%rdi), %rax
  leaq 8(%rdi), %r8
  movq $42, 8(%rsp)
  leaq 8(%rsp), %rsi
  movq %r8, %rdi
  call *24(%rax)  // <-- indirect call
  addq $24, %rsp
  ret
```

Disassembly

```
any_invocable<int(int)> test() {
    return [](int x) { return x; };
}
```

Disassembly

```
test():
  movq $vtable_for<lambda>, (%rdi)
  movq %rdi, %rax
  ret
```

Disassembly

```
vtable_for<lambda>:
  .quad buffer::destroy<lambda>
  .quad buffer::relocate<lambda>
  .quad buffer::invoke<lambda>(buffer&, int, int&&)
```

Disassembly

```
int buffer::invoke<lambda, int&&>(buffer&, int, int&&):
  movl (%rsi), %eax
  ret
```

Disassembly

```
any_invocable<int(int), long(long)> test() {
    return [](auto x) { return x; };
}
```

Disassembly

```
test():
  movq $vtable_for<lambda>, (%rdi)
  movq %rdi, %rax
  ret
```

Disassembly

```
vtable_for<lambda>:
  .quad buffer::destroy<lambda>
  .quad buffer::relocate<lambda>
  .quad buffer::invoke<lambda>(buffer&, int, int&&)
  .quad buffer::invoke<lambda>(buffer&, long, long&&)
```

Disassembly

```
int buffer::invoke<lambda, int&&>(buffer&, int, int&&):
  movl (%rsi), %eax
  ret

long buffer::invoke<lambda, long&&>(buffer&, long, long&&):
  movq (%rsi), %rax
  ret
```

Disassembly

```
struct NonTrivial {
    NonTrivial() noexcept;
    NonTrivial(NonTrivial&&) noexcept;
    ~NonTrivial();
    template <typename T>
    T operator()(T x) { return x; }
};
```

Disassembly

```
any_invocable<int(int), long(long)> test() {
    return NonTrivial();
}
```

Disassembly

```
test():
  pushq %r12
  movq %rdi, %r12
  subq $16, %rsp
  leaq 15(%rsp), %rdi
  call NonTrivial() [complete object constructor]
  leaq 8(%r12), %rdi
  leaq 15(%rsp), %rsi
  movq $vtable_for<NonTrivial, int (int), long (long)>, (%r12)
  call NonTrivial(NonTrivial&&) [complete object constructor]
  leaq 15(%rsp), %rdi
  call ~NonTrivial() [complete object destructor]
  addq $16, %rsp
  movq %r12, %rax
  popq %r12
  ret
```

Disassembly

```
int buffer::invoke<NonTrivial, int, int&&>(buffer&, int&&):
  movl (%rsi), %eax
  ret

long buffer::invoke<NonTrivial, long, long&&>(buffer&, long&&):
  movq (%rsi), %rax
  ret

void buffer::destroy<NonTrivial>(buffer&):
  jmp ~NonTrivial() [complete object destructor]
```

Disassembly

```
void buffer::relocate<NonTrivial>(buffer&, buffer&):
  pushq %rbp
  movq %rdi, %rbp
  movq %rsi, %rdi
  movq %rbp, %rsi
  call NonTrivial(NonTrivial&&) [complete object constructor]
  movq %rbp, %rdi
  popq %rbp
  jmp ~NonTrivial() [complete object destructor]
```

Disassembly

```
struct NonTrivial {
    // ...
    char data[64];
};

any_invocable<int(int), long(long)> test() {
    return NonTrivial();
}
```

Disassembly

```
test():
  pushq %r12
  movq %rdi, %r12
  pushq %rbx
  subq $72, %rsp
  movq %rsp, %rdi
  call NonTrivial() [complete object constructor]
  movq $vtable_for<NonTrivial, int (int), long (long)>, (%r12)
  movl $64, %edi
  call operator new(unsigned long)
  movq %rax, %rbx
  movq %rax, %rdi
  movq %rsp, %rsi
  call NonTrivial(NonTrivial&&) [complete object constructor]
  movq %rbx, 8(%r12)
  movq %rsp, %rdi
  call ~NonTrivial() [complete object destructor]
  addq $72, %rsp
  movq %r12, %rax
  popq %rbx
  popq %r12
  ret
  movq %rax, %r12
  jmp .L9
test() [clone .cold]:
.L9:
  movq %rsp, %rdi
  call NonTrivial::~NonTrivial() [complete object destructor]
  movq %r12, %rdi
  call _Unwind_Resume
```

Disassembly

Discard:

```
any_invocable<int(int), long(long)> make();

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

Disassembly

```
test():
  subq $56, %rsp
  movq %rsp, %rdi
  call make()
  movq (%rsp), %rax
  testq %rax, %rax
  je .L1
  leaq 8(%rsp), %rdi
  call *(%rax)  // <-- indirect call
.L1:
  addq $56, %rsp
  ret
```

Disassembly

Move:

```
auto test(any_invocable<int(int), long(long)> fun) {
    return fun;
}
```

Disassembly

```
test(any_invocable<int (int), long (long)>):
  pushq %r12
  movq (%rsi), %rax
  movq %rdi, %r12
  movq $0, (%rsi)
  movq %rax, (%rdi)
  testq %rax, %rax
  je .L1
  leaq 8(%rdi), %r8
  leaq 8(%rsi), %rdi
  movq %r8, %rsi
  call *8(%rax)  // <-- indirect call
.L1:
  movq %r12, %rax
  popq %r12
  ret
```

Benchmarks

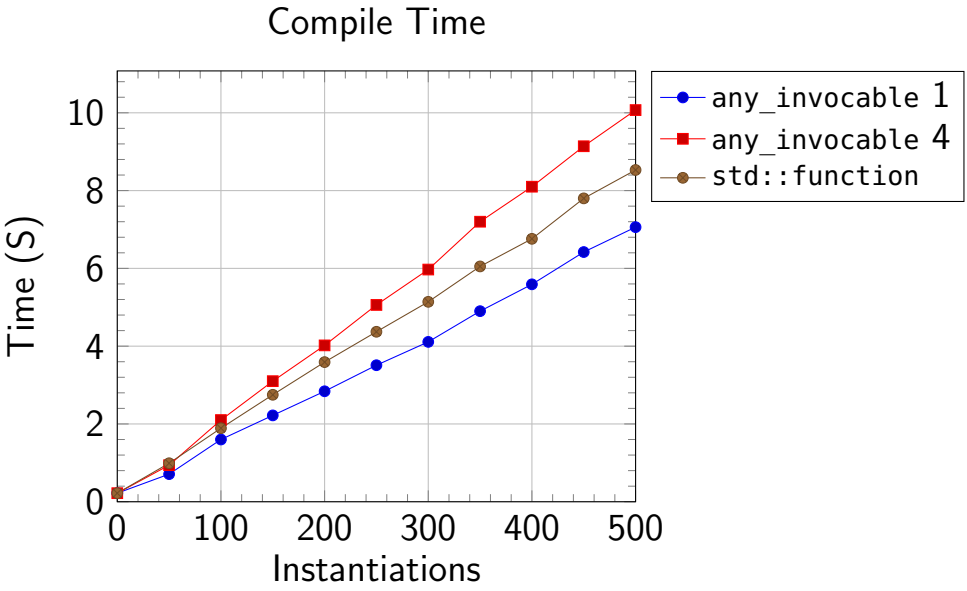
```
template <int N>
struct Callable {
    int operator()(...) { return N; };
};
```

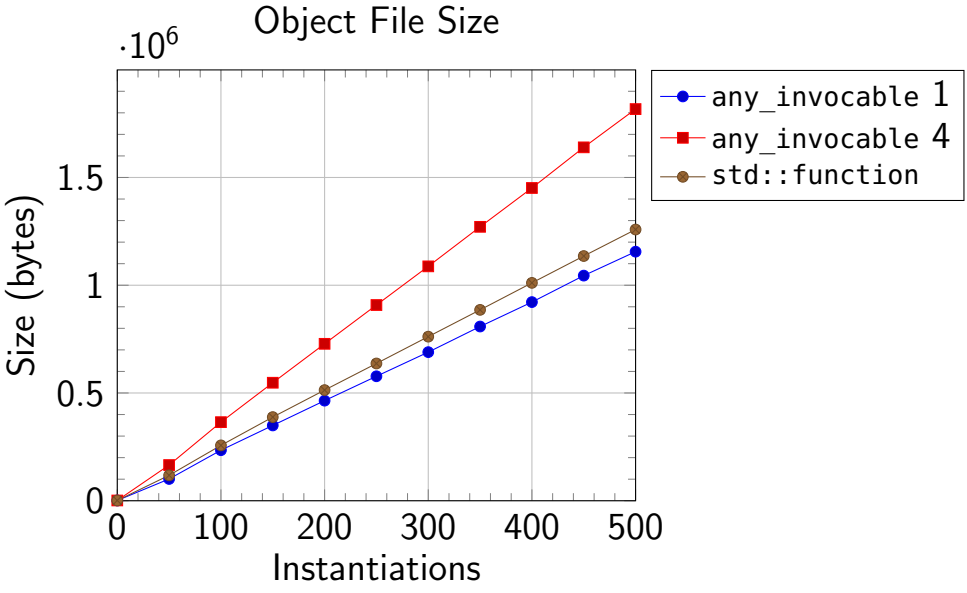
Benchmarks

```
using fun_t =
    std::function<int()>;
    // any_invocable<int()>;
    // any_invocable<int(), int(int)>;
    // any_invocable<int(), int(int), int(int, int)>;
```

Benchmarks

```
void test() {
    std::vector<fun_t> vec;
    vec.reserve(N);
    vec.emplace_back(Callable<0>());
    vec.emplace_back(Callable<1>());
    // ...
    vec.emplace_back(Callable<N - 1>());
}
```





Benchmarks

```
static auto const callable =
    [](auto... args) {
        return (0 + ...  + args);
    };
```

Benchmarks

```
template <typename fun_t>
void bm_creation(benchmark::State& state) {
    for (auto _ : state) {
        fun_t fun(callable);
        benchmark::DoNotOptimize(fun);
    }
}

BENCHMARK_TEMPLATE(bm_creation, std::function<int()>);
BENCHMARK_TEMPLATE(bm_creation, any_invocable<int()>);
BENCHMARK_TEMPLATE(bm_creation, any_invocable<int(), int(int)>);
// ...
```

Benchmarks

Benchmark	Time	CPU	Iterations
`std::function`	2.06 ns	2.06 ns	343402757
`any_invocable 1`	1.47 ns	1.47 ns	447794717
`any_invocable 2`	1.44 ns	1.45 ns	477966584
`any_invocable 3`	1.45 ns	1.45 ns	479729562
`any_invocable 4`	1.45 ns	1.45 ns	476625632

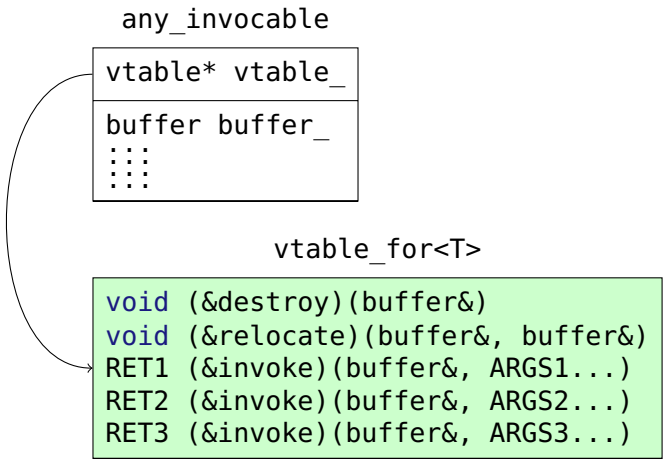
Benchmarks

```
template <typename fun_t>
void bm_invoke(benchmark::State& state) {
    fun_t fun(callable);
    benchmark::DoNotOptimize(fun);
    for (auto _ : state) {
        benchmark::DoNotOptimize(fun());
    }
}

BENCHMARK_TEMPLATE(bm_invoke, std::function<int()>);
BENCHMARK_TEMPLATE(bm_invoke, any_invocable<int()>);
BENCHMARK_TEMPLATE(bm_invoke, any_invocable<int(), int(int)>);
// ...
```

Benchmarks

Benchmark	Time	CPU	Iterations
`std::function`	1.45 ns	1.45 ns	475683992
`any_invocable 1`	1.44 ns	1.44 ns	496247430
`any_invocable 2`	1.45 ns	1.45 ns	481218079
`any_invocable 3`	1.45 ns	1.45 ns	479078764
`any_invocable 4`	1.45 ns	1.45 ns	486214222

Alternative Implementations



Alternative Implementations

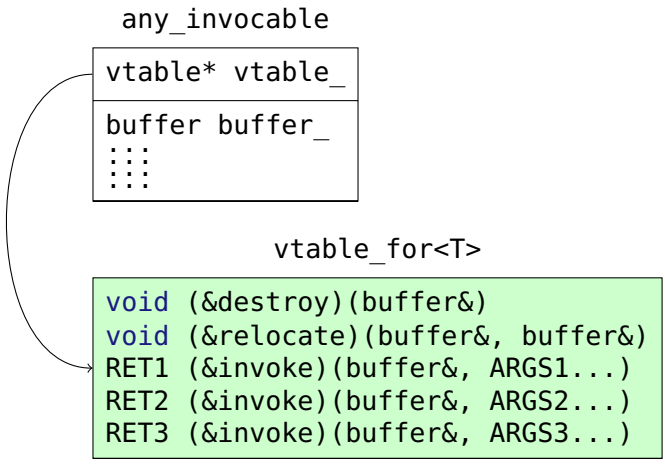
Function Storage:

1. Pointer to static struct with references to functions.
2. In-place pointers to non-member functions.
3. In-place pointers to member functions.

Argument Forwarding:

1. Forwarding references.
2. Forwarding values.

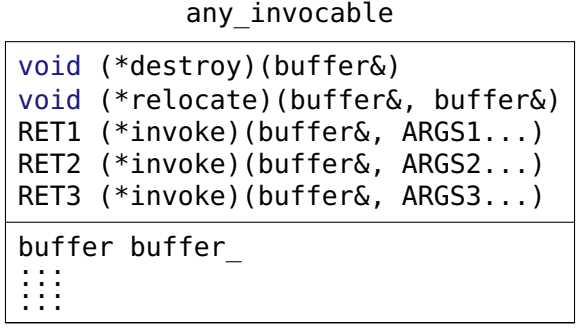
Alternative Function Storage



Alternative Function Storage

```
static_assert(sizeof(std::function<int()>) == 32);
static_assert(sizeof(any_invocable<int()>) == 40);
static_assert(
    sizeof(any_invocable<int(), int(int)>) == 40);
static_assert(
    sizeof(any_invocable<int(), int(int), int(int, int)>) == 40);
```

Alternative Function Storage



Alternative Function Storage

```
struct storage {
    buffer buffer_;
    void (*destroy_)(buffer&) noexcept;
    void (*relocate_)(buffer&, buffer&) noexcept;

    storage() noexcept;
    storage(storage&& other) noexcept;
    storage& operator=(storage&& other) noexcept;
    ~storage();

    // converting constructor
};
```

Alternative Function Storage

```
storage() noexcept :
    destroy_(nullptr),
    relocate_(nullptr) {}
```

Alternative Function Storage

```
storage(storage&& other) noexcept :
    destroy_(std::exchange(other.destroy_, nullptr)),
    relocate_(std::exchange(other.nullptr_)) {
    if (relocate_) relocate_(other.buffer_, buffer_);
}
```

Alternative Function Storage

```
storage& operator=(storage&& other) noexcept {
    if (&other != this) {
        if (destroy_) destroy_(buffer_);
        destroy_ = std::exchange(other.destroy_, nullptr);
        relocate_ = std::exchange(other.relocate_, nullptr);
        if (relocate_) relocate_(other.buffer_, buffer_);
    }
    return *this;
}
```

Alternative Function Storage

```
~storage() {
    if (destroy_) destroy_(buffer_);
}
```

Alternative Function Storage

```
template <typename T, typename... ARGS>
explicit storage(std::in_place_type<T>, ARGS&&... args) :
    destroy_(&buffer::destroy<T>),
    relocate_(&buffer::relocate<T>) {
    buffer_.construct<T>(std::forward<ARGS>(args)...);
}
```

Alternative Function Storage

```
template <typename RET, typename... ARGS, typename... FNS>
struct invocable_interface<RET(ARGS...), any_invocable<FNS...>> {
    RET (*invoke_)(buffer&, ARGS&&...);

    template <typename T>
    explicit invocable_interface(std::in_place_type_t<T>) noexcept;

    invocable_interface() noexcept = default;

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

Alternative Function Storage

```
template <typename T>
invocable_interface(std::in_place_type_t<T>) noexcept :
    invoke_(&buffer::invoke<T>) {}
```

Alternative Function Storage

```
RET operator()(ARGS... args) {
    any_invocable<FNS...>& self =
        static_cast<any_invocable<FNS...>&>(*this);

    return invoke_(self.buffer_, std::forward<ARGS>(args)...);
}
```

Alternative Function Storage

```
template <typename... FNS>
class any_invocable :
    storage,
    invocable_interface<
        FNS,
        any_invocable<FNS...>... {
public:
    // ...
};
```

Alternative Function Storage

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

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

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

Alternative Function Storage

```
template <typename T>
any_invocable(T&& object)
    requires (!std::is_same_v<std::decay_t<T> &&
        (is_invocable_v<std::decay_t<T>, FNS> && ...)) :
    storage(
        std::in_place_type<std::decay_t<T>>,
        std::forward<T>(object)),
    invocable_interface<FNS, any_invocable<FNS...>>(
        std::in_place_type<std::decay_t<T>>)... {}
```

Alternative Function Storage

```
static_assert(sizeof(std::function<int()>) == 32);

// Size was 40 always.

static_assert(sizeof(any_invocable<int()>) == 56);
static_assert(
    sizeof(any_invocable<int(), int(int)>) == 64);
static_assert(
    sizeof(any_invocable<int(), int(int), int(int, int)>) == 72);
```

Alternative Function Storage

```
int test(any_invocable<int(int)> fun) {
    return fun(42);
}
```

Alternative Function Storage

```
test(any_invocable<int (int)>):
  subq $24, %rsp
  movl $42, 12(%rsp)
  leaq 12(%rsp), %rsi
  call *48(%rdi)  // <-- indirect call
  addq $24, %rsp
  ret
```

Alternative Function Storage

```
long test(any_invocable<int(int), long(long)> fun) {
    return fun(42l);
}
```

Alternative Function Storage

```
test(any_invocable<int (int), long (long)>):
  subq $24, %rsp
  movq $42, 8(%rsp)
  leaq 8(%rsp), %rsi
  call *56(%rdi)  // <-- indirect call
  addq $24, %rsp
  ret
```

Alternative Function Storage

```
any_invocable<int(int)> test() {
    return [](int x) { return x; };
}
```

Alternative Function Storage

```
test():
  movq $buffer::destroy<lambda>(buffer&), 32(%rdi)
  movq %rdi, %rax
  movq $buffer::relocate<lambda>(buffer&, buffer&), 40(%rdi)
  movq $buffer::invoke<lambda, int, int&&>(buffer&, int&&), 48(%rdi)
  ret
```

Alternative Function Storage

```
any_invocable<int(int), long(long)> test() {
    return [](auto x) { return x; };
}
```

Alternative Function Storage

```
test():
  movq $buffer::destroy<lambda>(buffer&), 32(%rdi)
  movq %rdi, %rax
  movq $buffer::relocate<lambda>(buffer&, buffer&), 40(%rdi)
  movq $buffer::invoke<lambda, int, int&&>(buffer&, int&&), 48(%rdi)
  movq $buffer::invoke<lambda, long, long&&>(buffer&, long&&), 56(%rdi)
  ret
```

Alternative Function Storage

```
struct NonTrivial { /* ... */ };

static_assert(buffer::fits<NonTrivial>());

any_invocable<int(int), long(long)> test() {
    return NonTrivial();
}
```

Alternative Function Storage

```
test():
  pushq %r12
  movq %rdi, %r12
  subq $16, %rsp
  leaq 15(%rsp), %rdi
  call NonTrivial::NonTrivial() [complete object constructor]
  leaq 15(%rsp), %rsi
  movq %r12, %rdi
  movq $void buffer::destroy<NonTrivial>(buffer&), 32(%r12)
  movq $void buffer::relocate<NonTrivial>(buffer&, buffer&), 40(%r12)
  call NonTrivial::NonTrivial(NonTrivial&&) [complete object constructor]
  leaq 15(%rsp), %rdi
  movq $int buffer::invoke<NonTrivial, int, int&&>(buffer&, int&&), 48(%r12)
  movq $long buffer::invoke<NonTrivial, long, long&&>(buffer&, long&&), 56(%r12)
  call NonTrivial::~NonTrivial() [complete object destructor]
  addq $16, %rsp
  movq %r12, %rax
  popq %r12
  ret
```

Alternative Function Storage

```
test():
  // ...
  call NonTrivial() [complete object constructor]
  // ...
  movq $void buffer::destroy<NonTrivial>, 32(%r12)
  movq $void buffer::relocate<NonTrivial>, 40(%r12)
  call NonTrivial(NonTrivial&&) [complete object constructor]
  // ...
  movq $buffer::invoke<NonTrivial, int, int&&>, 48(%r12)
  movq $buffer::invoke<NonTrivial, long, long&&>, 56(%r12)
  call ~NonTrivial() [complete object destructor]
  // ...
  ret
```

Alternative Function Storage

```
struct NonTrivial {
    // ...
    char data[64];
};

static_assert(!buffer::fits<NonTrivial>());

any_invocable<int(int), long(long)> test() {
    return NonTrivial();
}
```

Alternative Function Storage

```
test():
  pushq %r12
  movq %rdi, %r12
  pushq %rbx
  subq $72, %rsp
  movq %rsp, %rdi
  call NonTrivial::NonTrivial() [complete object constructor]
  movl $64, %edi
  movq $void buffer::destroy<NonTrivial>(buffer&), 32(%r12)
  movq $void buffer::relocate<NonTrivial>(buffer&, buffer&), 40(%r12)
  call operator new(unsigned long)
  movq %rax, %rbx
  movq %rax, %rdi
  movq %rsp, %rsi
  call NonTrivial::NonTrivial(NonTrivial&&) [complete object constructor]
  movq %rbx, (%r12)
  movq %rsp, %rdi
  movq $int buffer::invoke<NonTrivial, int, int&&>(buffer&, int&&), 48(%r12)
  movq $long buffer::invoke<NonTrivial, long, long&&>(buffer&, long&&), 56(%r12)
  call NonTrivial::~NonTrivial() [complete object destructor]
  addq $72, %rsp
  movq %r12, %rax
  popq %rbx
  popq %r12
  ret
  movq %rax, %r12
  jmp .L9
test() [clone .cold]:
.L9:
  movq %rsp, %rdi
  call NonTrivial::~NonTrivial() [complete object destructor]
  movq %r12, %rdi
  call _Unwind_Resume
```

Alternative Function Storage

Discard:

```
any_invocable<int(int), long(long)> make();

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

Alternative Function Storage

```
test():
  subq $72, %rsp
  movq %rsp, %rdi
  call make()
  movq 32(%rsp), %rax
  testq %rax, %rax
  je .L1
  movq %rsp, %rdi
  call *%rax  // <-- indirect call
.L1:
  addq $72, %rsp
  ret
```

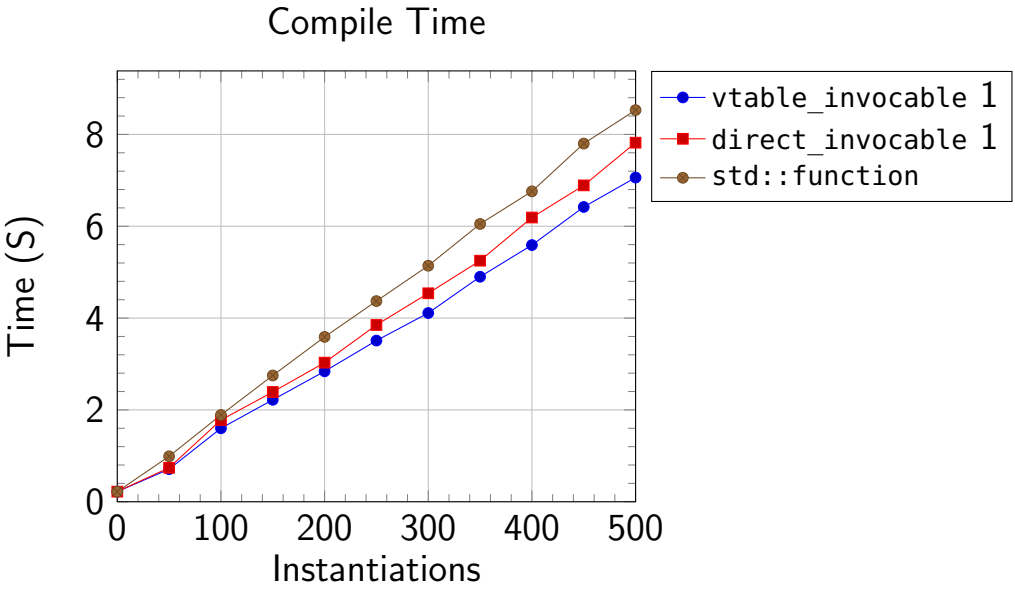
Alternative Function Storage

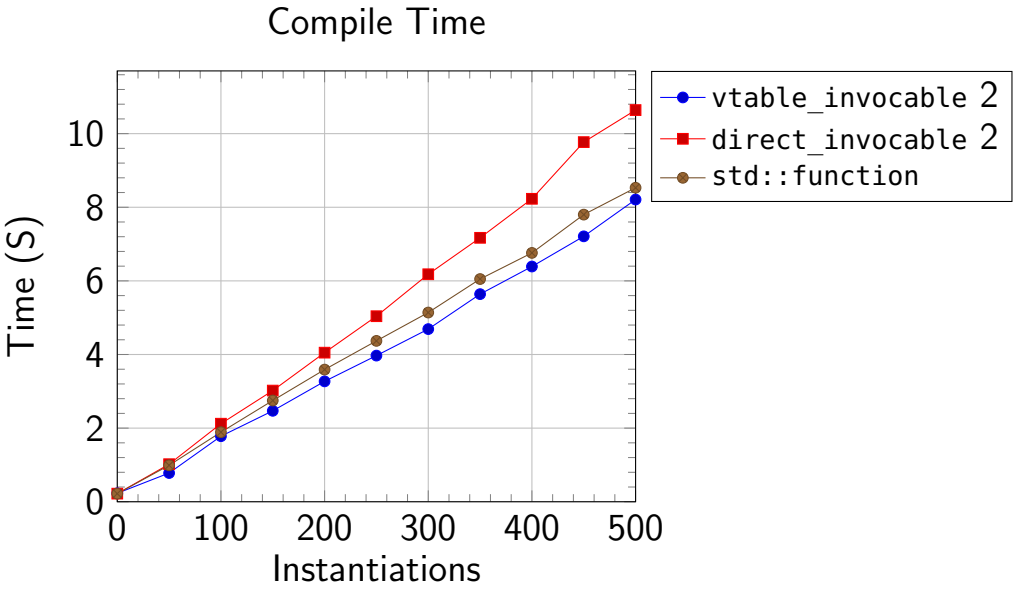
Move:

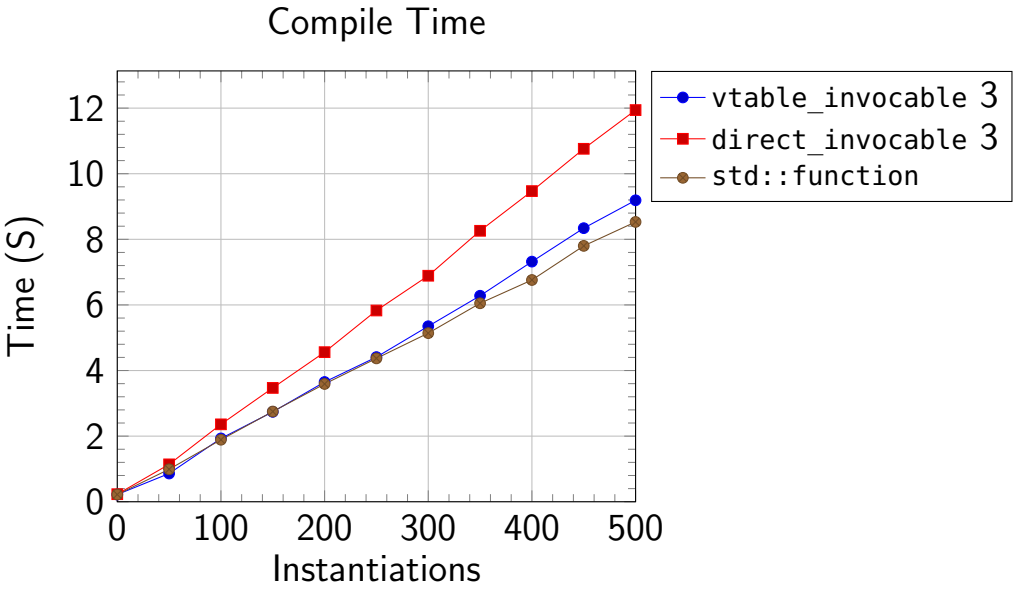
```
auto test(any_invocable<int(int), long(long)> fun) {
    return fun;
}
```

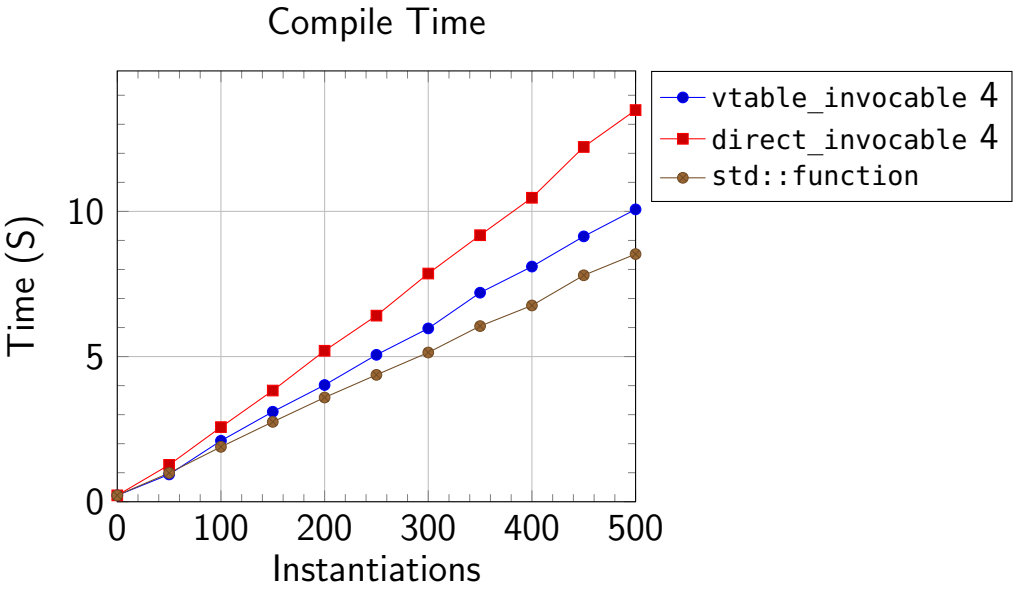
Alternative Function Storage

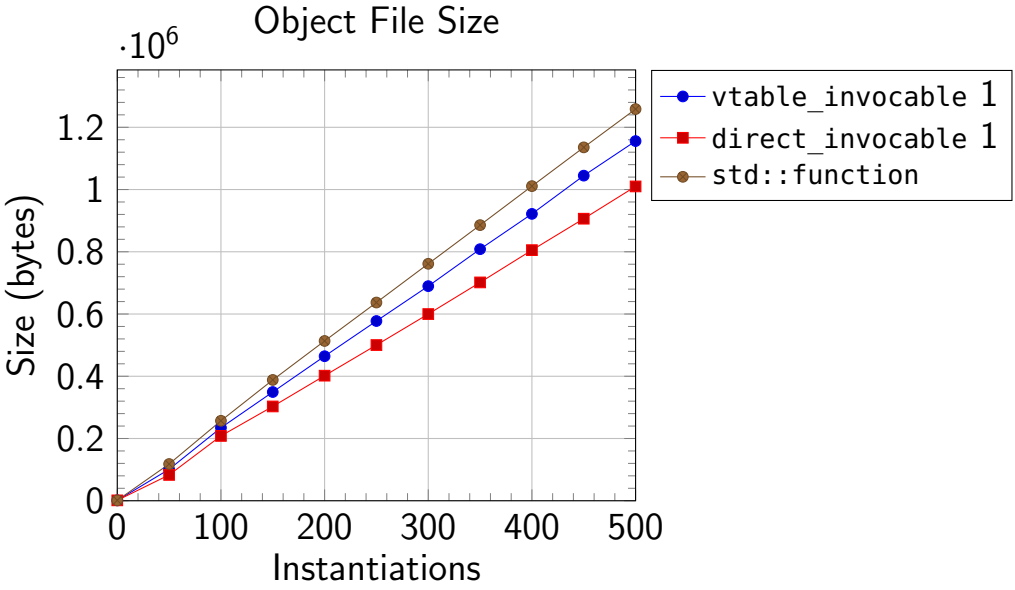
```
test(any_invocable<int (int), long (long)>):
  pushq %r12
  movq %rdi, %r12
  pushq %rbx
  movq %rsi, %rbx
  subq $8, %rsp
  movq 32(%rsi), %rax
  movq $0, 32(%rsi)
  movq %rax, 32(%rdi)
  movq 40(%rsi), %rax
  movq $0, 40(%rsi)
  movq %rax, 40(%rdi)
  testq %rax, %rax
  je .L2
  movq %rdi, %rsi
  movq %rbx, %rdi
  call *%rax // <-- indirect call
.L2:
  movq 48(%rbx), %rax
  movq %rax, 48(%r12)
  movq 56(%rbx), %rax
  movq %rax, 56(%r12)
  addq $8, %rsp
  movq %r12, %rax
  popq %rbx
  popq %r12
  ret
```

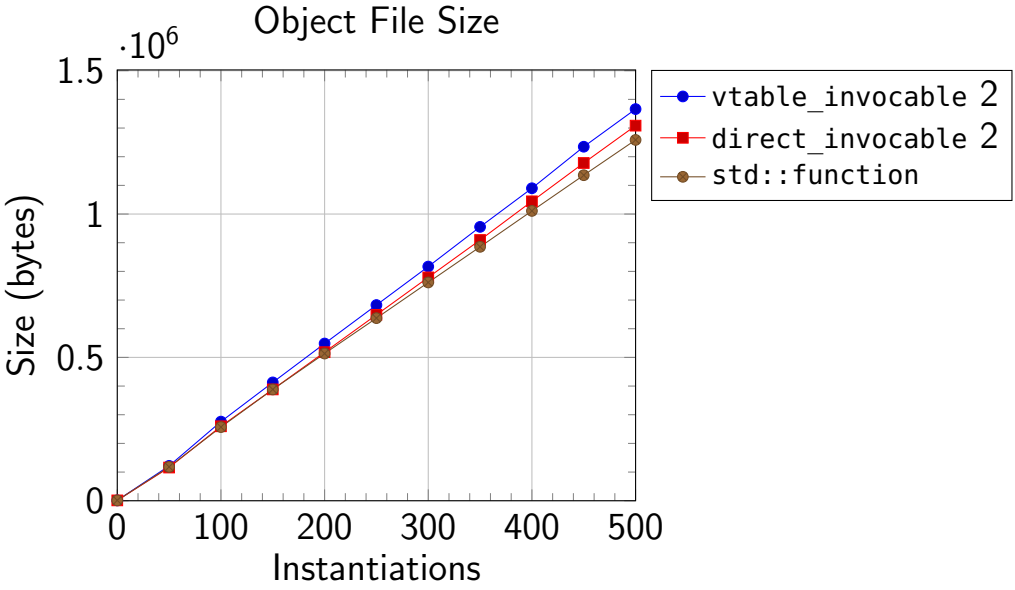


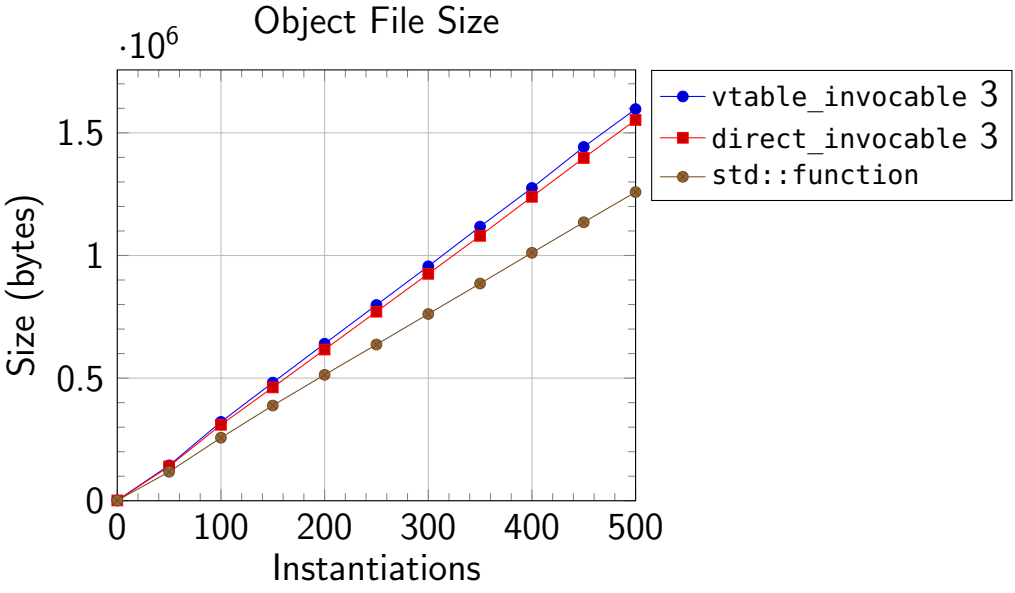


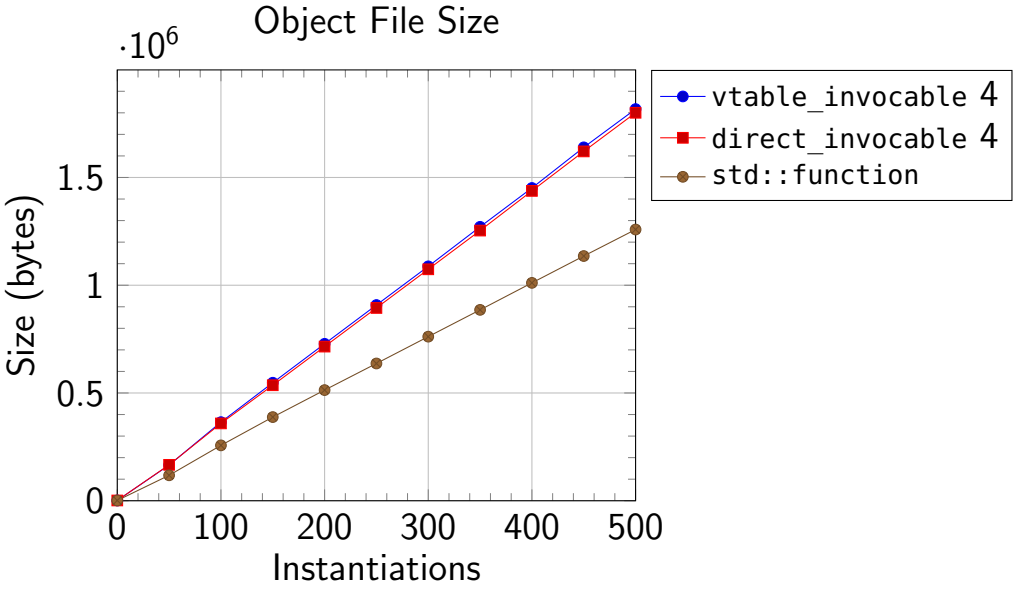












Alternative Function Storage

```
template <typename fun_t>
void bm_creation(benchmark::State& state) {
    for (auto _ : state) {
        fun_t fun(callable);
        benchmark::DoNotOptimize(fun);
    }
}
```

Alternative Function Storage

Benchmark	Time	CPU	Iterations
`std::function`	2.06 ns	2.06 ns	343402757
`vtable` 1	1.47 ns	1.47 ns	447794717
`direct` 1	1.46 ns	1.46 ns	475875162
`vtable` 4	1.44 ns	1.44 ns	472615864
`direct` 4	1.74 ns	1.74 ns	396204125

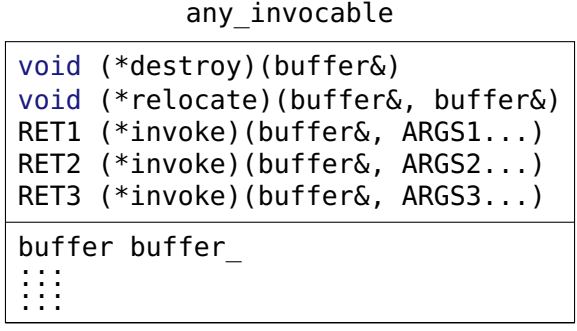
Alternative Function Storage

```
template <typename fun_t>
void bm_invoke(benchmark::State& state) {
    fun_t fun(callable);
    benchmark::DoNotOptimize(fun);
    for (auto _ : state) {
        benchmark::DoNotOptimize(fun());
    }
}
```

Alternative Function Storage

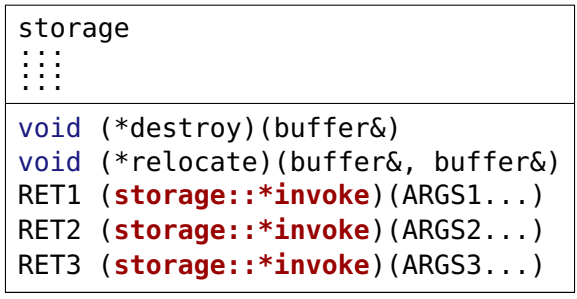
Benchmark	Time	CPU	Iterations
`std::function`	1.46 ns	1.46 ns	485641496
`vtable` 1	1.45 ns	1.45 ns	483319455
`direct` 1	1.45 ns	1.45 ns	479107098
`vtable` 4	1.45 ns	1.45 ns	478195449
`direct` 4	1.45 ns	1.45 ns	475809537

Pointers To Member Functions



Pointers To Member Functions

any_invocable

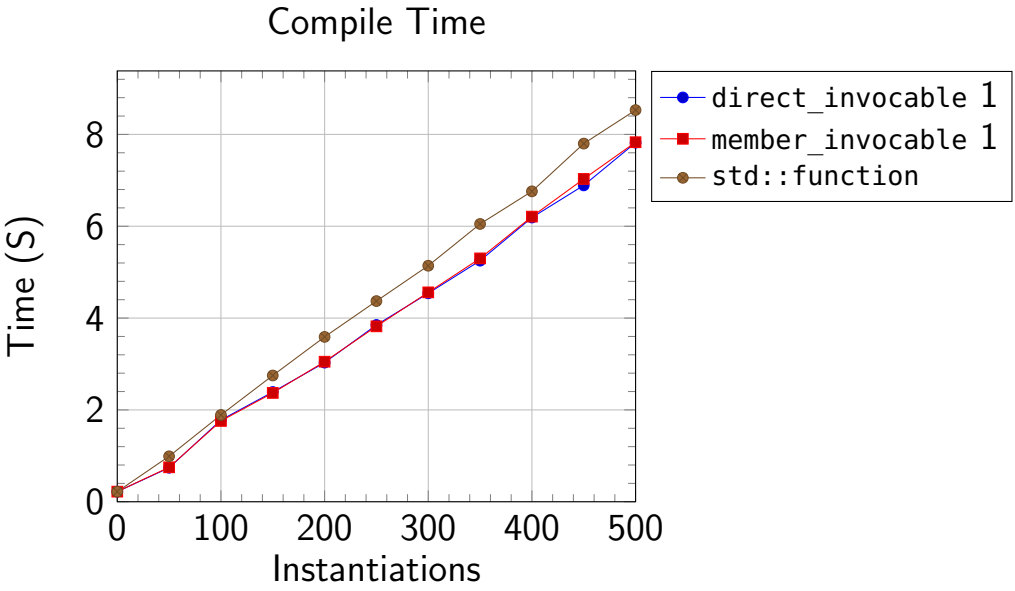


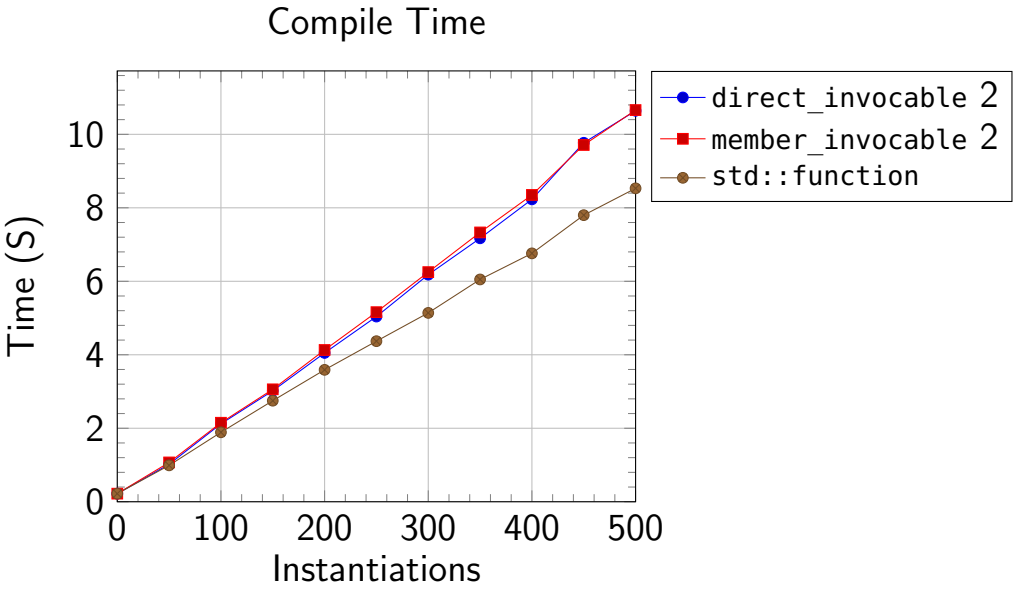
Pointers To Member Functions

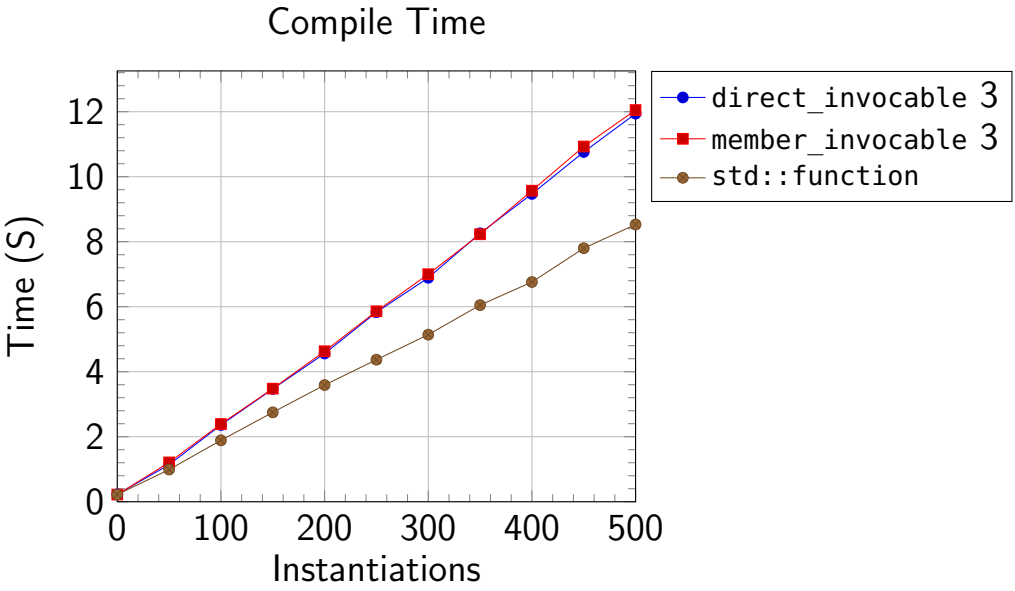
```
template <typename RET, typename... ARGS>
struct invocable_interface<RET(ARGS...)> {
    RET (invocable_interface::* invoke_)(buffer&, ARGS&&...);

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

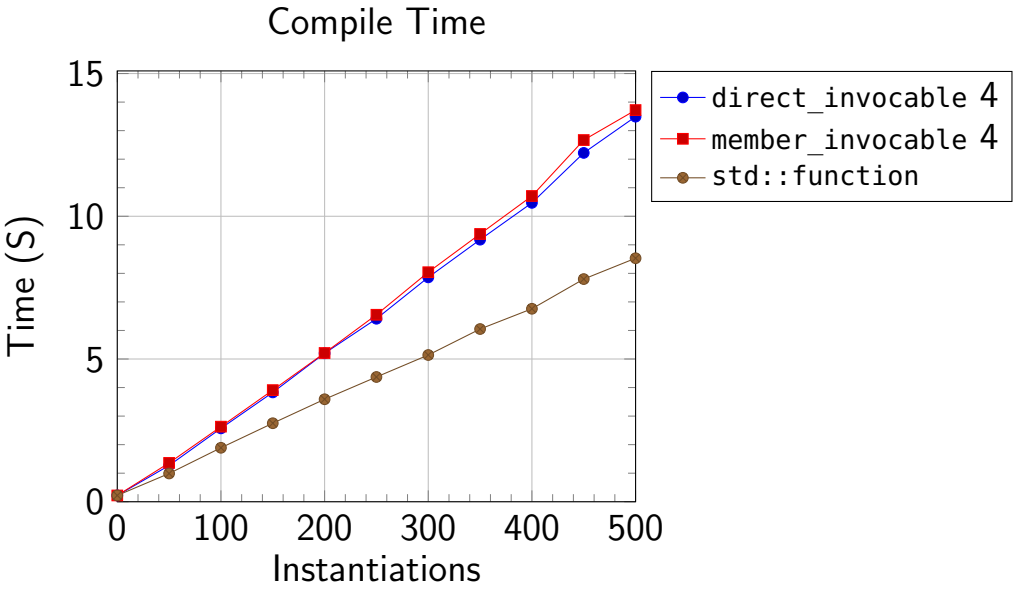
Pointers To Member Functions

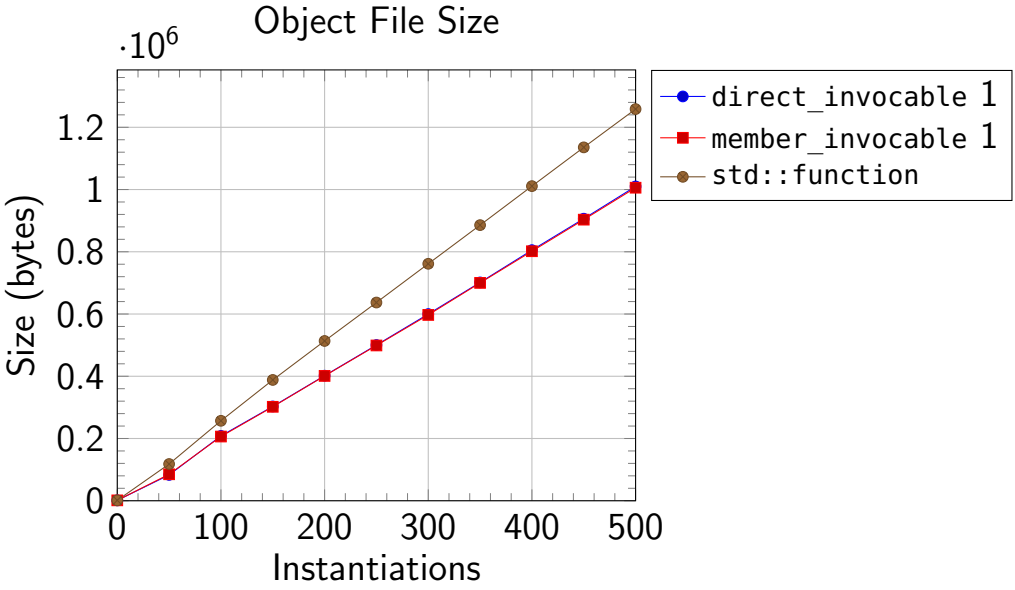
```
RET operator()(ARGS... args) {
    return (this->*invoke_)(std::forward<ARGS>(args)...);
}
```

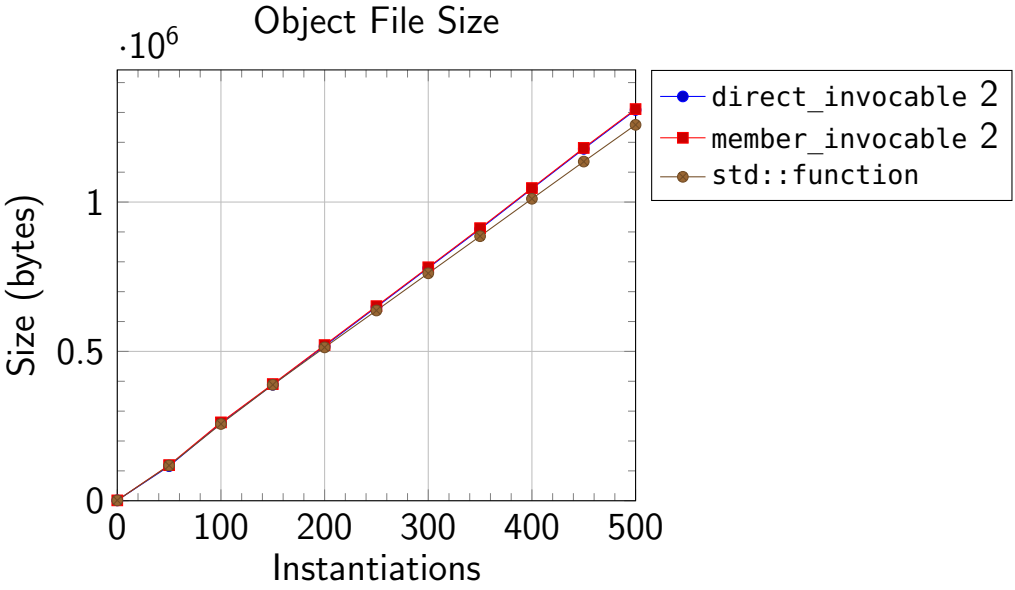


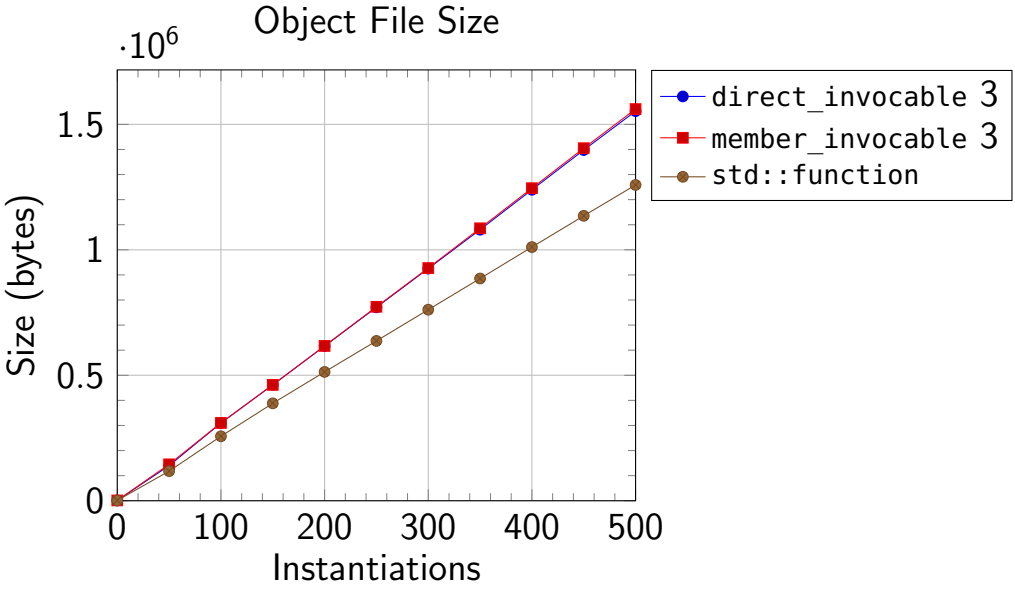


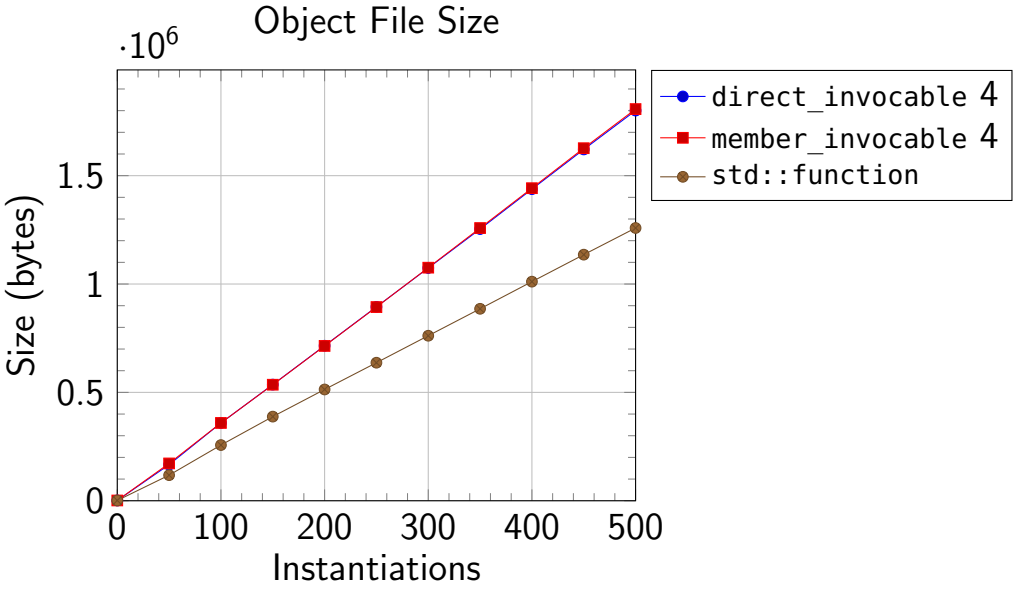












Thank You

Bloomberg

Engineering

TechAtBloomberg.com