

Zero Overhead Pass By Value Through Invocable Abstractions

Engineering

Bloomberg

using std::cpp 2024

30 June, 2024

Filipp Gelman, PE

fgelman1@bloomberg.net

TechAtBloomberg.com

Outline

- Outline
- Zero Overhead Pass By Value
- Why Invocable Abstractions Add Overhead
- How To Avoid Overhead
- Handling Free Functions
- Handling Member Functions
- Handling Function Objects

Zero Overhead Pass By Value

```
struct Mountain {  
    explicit Mountain(char const* name);  
  
    void climb();  
  
};
```

Zero Overhead Pass By Value

```
struct Mountain {  
    explicit Mountain(char const* name);  
  
    void climb();  
  
    Mountain(Mountain const&) = delete;  
    Mountain(Mountaion&&) = delete;  
  
    ~Mountain();  
};
```

Not copyable or movable!

Zero Overhead Pass By Value

```
void normal_function(Mountain m) {  
    m.climb();  
}
```

Zero Overhead Pass By Value

```
void normal_function(Mountain m) {  
    m.climb();  
}
```

```
int main(int, char**) {  
  
}
```

Zero Overhead Pass By Value

```
void normal_function(Mountain m) {  
    m.climb();  
}  
  
int main(int, char**) {  
    normal_function(Mountain("Everest"));  
}
```

Zero Overhead Pass By Value

```
void normal_function(Mountain m) {  
    m.climb();  
}  
  
int main(int, char**) {  
    normal_function(Mountain("Everest"));  
}
```

No copy or move!

Constructed directly in argument slot.

Zero Overhead Pass By Value

```
struct Base {  
  
};
```

Zero Overhead Pass By Value

```
struct Base {  
    virtual void virtual_function(Mountain) = 0;  
};
```

By Value

Zero Overhead Pass By Value

```
struct Base {  
    virtual void virtual_function(Mountain) = 0;  
};
```

```
struct Derived : Base {
```

```
};
```

Zero Overhead Pass By Value

```
struct Base {  
    virtual void virtual_function(Mountain) = 0;  
};
```

```
struct Derived : Base {  
    void virtual_function(Mountain m) override {  
        m.climb();  
    }  
};
```

By Value

Zero Overhead Pass By Value

```
int main(int, char**) {  
    Derived d;  
  
}
```

Zero Overhead Pass By Value

```
int main(int, char**) {  
    Derived d;  
    Base& b = d;  
  
}
```

Zero Overhead Pass By Value

```
int main(int, char**) {  
    Derived d;  
    Base& b = d;  
  
    b.virtual_function(Mountain("Matterhorn"));  
}
```

No copy or move!

Zero Overhead Pass By Value

```
int main(int, char**) {  
    auto lambda = [](Mountain m) { m.climb(); }  
      
    By Value  
  
}
```


Zero Overhead Pass By Value

```
int main(int, char**) {  
    auto lambda = [](Mountain m) { m.climb(); };  
    lambda(Mountain("Denali"));  
}
```

No copy or move!

Zero Overhead Pass By Value

```
int main(int, char**) {  
    auto lambda = [](Mountain m) { m.climb(); };  
  
}
```

Zero Overhead Pass By Value

```
int main(int, char**) {  
    auto lambda = [](Mountain m) { m.climb(); };  
    std::function<void(Mountain)> fun(lambda);  
}
```

Zero Overhead Pass By Value

```
int main(int, char**) {  
    auto lambda = [](Mountain m) { m.climb(); };  
  
    std::function<void(Mountain)> fun(lambda);  
  
    fun(Mountain("Fuji"));  
}
```

Zero Overhead Pass By Value

```
int main(int, char**) {  
    auto lambda = [](Mountain m) { m.climb(); };  
std::function<void(Mountain)> fun(lambda);  
  
    fun(Mountain("Fuji"));  
}
```

Does not compile.

The Problem

- `std::function` Does Not Support Immovable Arguments
- `std::function` Always Moves Argument Values

The Cause

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

```
};
```

The Cause

```
template <typename R, typename... Args>
class function<R(Args...)> {
    template <typename Func>
    function(Func&&);
};
```


The Cause

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

```
    template <typename Func>
```

```
        requires std::is_invocable_v<Func, Args...>
```

```
        function(Func&&);
```

```
};
```

Can I call Func with Args&&...?

The Cause

```
template <typename Func, typename... Args>  
constexpr bool is_invocable_v =
```

The Cause

```
template <typename Func, typename... Args>  
constexpr bool is_invocable_v = requires(Func f) {  
  
};
```

The Cause

```
template <typename Func, typename... Args>  
constexpr bool is invocable v = requires(Func f) {  
    f(  
    });  
};
```

The Cause

```
template <typename Func, typename... Args>
constexpr bool is_invocable_v = requires(Func f) {
    f(std::declval<Args>()...);
};
```

The Cause

```
template <typename Func, typename... Args>
constexpr bool is_invocable_v = requires(Func f) {
    f(std::declval<Args>()...);
};
```

`std::declval` creates a value of the specified type.

The Cause

```
template <typename Func, typename... Args>
constexpr bool is_invocable_v = requires(Func f) {
    f(std::declval<Args>()...);
};
```

`std::declval` creates a value of the specified type.

```
std::declval<T& >() -> T&;
```

The Cause

```
template <typename Func, typename... Args>
constexpr bool is_invocable_v = requires(Func f) {
    f(std::declval<Args>()...);
};
```

`std::declval` creates a value of the specified type.

```
std::declval<T&>() -> T&;
std::declval<T&&>() -> T&&;
```


The Cause

```
template <typename Func, typename... Args>  
constexpr bool is_invocable_v = requires(Func f) {  
    f(std::declval<Args>()...);  
};
```

std::declval creates a value of the specified type.

```
std::declval<T&>() -> T&;  
std::declval<T&&>() -> T&&;  
std::declval<T>() -> T&&;
```

The Cause

```
template <typename Func, typename... Args>
constexpr bool is_invocable_v = requires(Func f) {
    f(std::declval<Args>()...);
};
```

std::declval creates an **rvalue reference** of the specified type.

```
std::declval<T& >() -> T&;
std::declval<T&&>() -> T&&;
std::declval<T >() -> T&&;
```

The Cause

```
void normal_function(Mountain m);
```

The Cause

```
void normal_function(Mountain m);
```

```
Mountain make_value();
```

The Cause

```
void normal_function(Mountain m);
```

```
Mountain make_value();
```

```
int main(int, char**) {  
    normal_function(make_value());  
}
```

The Cause

```
void normal_function(Mountain m);
```

```
Mountain make_value();
```

```
Mountain&& make_rvalue_ref();
```

```
int main(int, char**) {  
    normal_function(make_value());  
  
}
```

The Cause

```
void normal_function(Mountain m);
```

```
Mountain make_value();
```

```
Mountain&& make_rvalue_ref();
```

```
int main(int, char**) {  
    normal_function(make_value());  
    normal_function(make_rvalue_ref());  
}
```

The Cause

```
void normal_function(Mountain m);
```

```
Mountain make_value();
```

```
Mountain&& make_rvalue_ref();
```

```
int main(int, char**) {  
    normal_function(make_value());  
    normal_function(make_rvalue_ref());  
}
```

Passing a prvalue is different than passing an rvalue reference.

The Cause

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

```
};
```

The Cause

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

Potentially values

```
};
```

The Cause

```
template <typename R, typename... Args>  
class function<R(Args...)> {  
    void* m_bound;
```

Points to the target object.

```
};
```

The Cause

```
template <typename R, typename... Args>  
class function<R(Args...)> {  
    void* m_bound;  
    R (*m_thunk)(void*, Args&&...);  
};
```

Forwards arguments to the target object.

```
};
```

The Cause

```
template <typename R, typename... Args>
class function<R(Args...)> {
    void* m_bound;
    R (*m_thunk)(void*, Args&&...);

    R operator()(Args... args) {
        return m_thunk(
            m_bound, std::forward<Args>(args)...);
    }
};
```

The Cause

```
R operator() (Args... args) {  
    return m_thunk(  
        m_bound, std::forward<Args>(args)...);  
}
```

The Cause

```
R operator()(Args... args) {  
    return m_thunk( Points to thunk<T>.  
        m_bound, std::forward<Args>(args)... );  
}
```

```
template <typename T>  
static R thunk(void* bound, Args&& args...) {  
  
}
```

The Cause

```
R operator()(Args... args) {  
    return m_thunk(  
        m_bound, std::forward<Args>(args)...);  
}
```

```
template <typename T>  
static R thunk(void* bound, Args&& args...) {  
    return (*static_cast<T*>(bound))(  
        std::forward<Args>(args)...);  
}
```


The Cause

```
R operator() (Args... args) { Values.  
    return m_thunk(  
        m_bound, std::forward<Args>(args)...);  
}
```

```
template <typename T>  
static R thunk(void* bound, Args&& args...) {  
    return (*static_cast<T*>(bound))(  
        std::forward<Args>(args)...);  
}
```

The Cause

```
R operator() (Args... args) {  
    return m_thunk(  
        m_bound, std::forward<Args>(args)...);  
}
```

```
template <typename T>  
static R thunk(void* bound, Args&& args...) {  
    return (*static_cast<T*>(bound))(  
        std::forward<Args>(args)...);  
}
```

References, not values.

The Clue

```
struct Boulder {  
    // rule of five  
};  
  
void target(Boulder );  
  
void test() {  
    target(Boulder());  
}
```

The Clue

```
struct Boulder {  
    // rule of five  
};  
  
void target(Boulder );  
  
void test() {  
    target(Boulder());  
}  
  
test():  
    subq $24, %rsp  
    leaq 15(%rsp), %rdi  
    call Boulder::Boulder() [complete object constructor]  
    leaq 15(%rsp), %rdi  
    call target(Boulder)  
    leaq 15(%rsp), %rdi  
    call Boulder::~~Boulder() [complete object destructor]  
    addq $24, %rsp  
    ret
```

The Clue

```
struct Boulder {  
    // rule of five  
};
```

```
void target(Boulder );
```

```
void test() {  
    target(Boulder());  
}
```

```
test():  
    subq $24, %rsp  
    leaq 15(%rsp), %rdi  
    call Boulder::Boulder() [complete object constructor]  
    leaq 15(%rsp), %rdi  
    call target(Boulder)  
    leaq 15(%rsp), %rdi  
    call Boulder::~~Boulder() [complete object destructor]  
    addq $24, %rsp  
    ret
```

The Clue

```
struct Boulder {  
    // rule of five  
};  
  
void target(Boulder&&);  
  
void test() {  
    target(Boulder());  
}
```

```
test():  
    subq $24, %rsp  
    leaq 15(%rsp), %rdi  
    call Boulder::Boulder() [complete object constructor]  
    leaq 15(%rsp), %rdi  
    call target(Boulder&&)  
    leaq 15(%rsp), %rdi  
    call Boulder::~~Boulder() [complete object destructor]  
    addq $24, %rsp  
    ret
```

Same instructions!

The Rule

<https://itanium-cxx-abi.github.io/cxx-abi/abi.html#non-trivial-parameters>

If a parameter type is a class type that is **non-trivial for the purposes of calls**, the caller must allocate space for a temporary and pass that temporary by reference.

The Rule

<https://itanium-cxx-abi.github.io/cxx-abi/abi.html#non-trivial>

non-trivial for the purpose of calls

A type is considered non-trivial for the purposes of calls if:

- it has a non-trivial copy constructor, move constructor, or destructor, or
- all of its copy and move constructors are deleted

The Rule

https://en.cppreference.com/w/cpp/types/is_trivially_copyable

```
template <typename T>
struct is_trivially_copyable;

template <typename T>
inline constexpr bool is_trivially_copyable_v =
    is_trivially_copyable<T>::value;
```

The Rule

https://en.cppreference.com/w/cpp/language/classes#Trivially_copyable_class

A *trivially copyable class* is a class that

- has at least one eligible ... constructor or ... assignment operator
- each eligible ... constructor is trivial
- each eligible ... assignment operator is trivial
- has a non-deleted trivial destructor.

The Trick

```
void target(Boulder);
```

```
void test() {  
    void (* const by_val_ptr)(Boulder) = &target;
```

```
    void (* const by_ref_ptr)(Boulder&&) =  
        reinterpret_cast<void(*) (Boulder&&)>(by_val_ptr);
```

```
    by_ref_ptr(Boulder());
```

```
}
```

The Trick

```
void target(Boulder);
```

```
void test() {
```

```
    // ...
```

```
    by_ref_ptr(Boulder());
```

```
}
```

```
test():
```

```
    subq $24, %rsp
```

```
    leaq 15(%rsp), %rdi
```

```
    call Boulder::Boulder() [complete object constructor]
```

```
    leaq 15(%rsp), %rdi
```

```
    call target(Boulder)
```

```
    leaq 15(%rsp), %rdi
```

```
    call Boulder::~~Boulder() [complete object destructor]
```

```
    addq $24, %rsp
```

```
    ret
```

The Trick

```
void target(Boulder);  
  
void test(Boulder&& boulder) {  
    // ...  
    by_ref_ptr(std::move(boulder));  
}
```

Pass reference instead of value!

The Abstraction

https://en.cppreference.com/w/cpp/utility/functional/function_ref

The Abstraction

https://en.cppreference.com/w/cpp/utility/functional/function_ref

```
template <typename>  
class function_ref;
```

The Abstraction

https://en.cppreference.com/w/cpp/utility/functional/function_ref

```
template <typename>  
class function_ref;
```

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


The Abstraction

https://en.cppreference.com/w/cpp/utility/functional/function_ref

```
template <typename>  
class function_ref;
```

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

The Abstraction

https://en.cppreference.com/w/cpp/utility/functional/function_ref

```
template <typename>  
class function_ref;
```

```
template <typename R, typename... Args>  
class function_ref<R(Args...)>;  
template <typename R, typename... Args>  
class function_ref<R(Args...) noexcept>;  
template <typename R, typename... Args>  
class function_ref<R(Args...) const>;
```

The Abstraction

https://en.cppreference.com/w/cpp/utility/functional/function_ref

```
template <typename>  
class function_ref;
```

```
template <typename R, typename... Args>  
class function_ref<R(Args...)>;  
template <typename R, typename... Args>  
class function_ref<R(Args...) noexcept>;  
template <typename R, typename... Args>  
class function_ref<R(Args...) const>;  
template <typename R, typename... Args>  
class function_ref<R(Args...) const noexcept>;
```

The Abstraction

https://en.cppreference.com/w/cpp/utility/functional/function_ref

```
template <typename>  
class function_ref;
```

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

Let's implement it.

The Abstraction

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

```
public:
```

```
}
```


The Abstraction

```
template <typename R, typename... Args>
class function_ref<R(Args...)>{
    void* m_bound;
    thunk_t m_thunk;           Forwards arguments.

public:

}
}
```

The Abstraction

```
template <typename R, typename... Args>
class function_ref<R(Args...)>{
    void* m_bound;
    thunk_t m_thunk;

public:
    function_ref(Func); Constructors.

}
```


The Abstraction

```
template <typename R, typename... Args>
class function_ref<R(Args...)>{
    void* m_bound;
    thunk_t m_thunk;

public:
    function_ref(Func);

    R operator()(Args... args) const

        Call the target object.
}
}
```

The Abstraction

```
template <typename R, typename... Args>
class function_ref<R(Args...)>{
    void* m_bound;
    thunk_t m_thunk;

public:
    function_ref(Func);

    R operator()(Args... args) const {
        m_thunk(m_bound, std::forward<Args>(args)...);
    }
}
```

The Abstraction

```
template <typename R, typename... Args>
class function_ref<R(Args...)>{
    void* m_bound;
    thunk_t m_thunk;      Pointer to function.

public:
    function_ref(Func);

    R operator()(Args... args) const {
        m_thunk(m_bound, std::forward<Args>(args)...);
    }
}
```

The Abstraction

// function ref // thunk t

void(int&) -> void(void*, int&)

The Abstraction

```
// function_ref    // thunk_t  
void(int&          ) -> void(void*, int&          )  
void(int const&   ) -> void(void*, int const&   )
```

The Abstraction

```
// function_ref    // thunk_t  
void(int&          ) -> void(void*, int&          )  
void(int const&   ) -> void(void*, int const&   )  
void(int&&        ) -> void(void*, int&&        )
```

The Abstraction

```
// function_ref    // thunk_t  
void(int&          ) -> void(void*, int&          )  
void(int const&   ) -> void(void*, int const&   )  
void(int&&        ) -> void(void*, int&&        )
```

References unchanged.

The Abstraction

```
// function_ref    // thunk_t  
void(int&          ) -> void(void*, int&          )  
void(int const&   ) -> void(void*, int const&       )  
void(int&&        ) -> void(void*, int&&          )
```

```
void(int          ) -> void(void*, int          )
```


The Abstraction

```
// function_ref    // thunk_t  
void(int&          ) -> void(void*, int&          )  
void(int const&   ) -> void(void*, int const&       )  
void(int&&         ) -> void(void*, int&&          )  
  
void(int           ) -> void(void*, int           )  
void(Trivial      ) -> void(void*, Trivial      )
```

The Abstraction

```
// function_ref    // thunk_t  
void(int&          ) -> void(void*, int&          )  
void(int const&    ) -> void(void*, int const&    )  
void(int&&         ) -> void(void*, int&&         )  
  
void(int           ) -> void(void*, int           )  
void(Trivial        ) -> void(void*, Trivial        )
```

Trivial types by value.

The Abstraction

```
// function_ref    // thunk_t  
void(int&          ) -> void(void*, int&          )  
void(int const&    ) -> void(void*, int const&    )  
void(int&&         ) -> void(void*, int&&         )  
  
void(int           ) -> void(void*, int           )  
void(Trivial       ) -> void(void*, Trivial       )  
  
void(NonTrivial) -> void(void*, NonTrivial&&)
```

The Abstraction

```
// function_ref    // thunk_t  
void(int&          ) -> void(void*, int&          )  
void(int const&    ) -> void(void*, int const&    )  
void(int&&         ) -> void(void*, int&&         )  
  
void(int           ) -> void(void*, int           )  
void(Trivial        ) -> void(void*, Trivial        )  
  
void(NonTrivial) -> void(void*, NonTrivial&&)
```

Non-trivial types by rvalue reference.

The Abstraction

```
template <typename T>  
struct thunk_arg<T> {  
  
};
```

The Abstraction

```
template <typename T>  
requires std::is_reference_v<T>  
struct thunk_arg<T> {  
  
};
```

The Abstraction

```
template <typename T>  
requires std::is_reference_v<T>  
struct thunk_arg<T> {  
    using type = T;           References unchanged.  
};
```

The Abstraction

```
template <typename T>  
requires std::is_trivially_copyable<T>  
struct thunk_arg<T> {  
  
};
```


The Abstraction

```
template <typename T>  
requires std::is_trivially_copyable<T>  
struct thunk_arg<T> {  
    using type = T;           Trivial types by value.  
};
```

The Abstraction

```
template <typename T>  
struct thunk_arg {  
    using type = T&&;  
};
```

Non-trivial types by rvalue reference.

The Abstraction

```
template <typename T>  
struct thunk_arg {  
    using type = T&&;  
};
```

```
template <typename T>  
using thunk_arg_t = typename thunk_arg<T>::type;
```

The Abstraction

```
template <typename T>  
struct thunk_arg {  
    using type = T&&;  
};
```

```
template <typename T>  
using thunk_arg_t = typename thunk_arg<T>::type;  
  
using thunk_t = R(*) (void*, thunk_arg_t<Args>...)
```

The Abstraction

```
template <typename T>  
struct thunk_arg {  
    using type = T&&;  
};
```

```
template <typename T>  
using thunk_arg_t = typename thunk_arg<T>::type;  
  
using thunk_t = R(*) (void*, thunk_arg_t<Args>...)
```

The Abstraction

Implement constructors for:

- Free Functions
- Member Functions
- Function Objects

Free Functions

```
void normal_function(Mountain m) {  
    m.climb();  
}
```

Free Functions

```
void normal_function(Mountain m) {  
    m.climb();  
}
```

```
int main(int, char**) {  
    function_ref<void(Mountain> fun(&normal_function));  
  
}
```


Free Functions

```
void normal_function(Mountain m) {  
    m.climb();  
}  
  
int main(int, char**) {  
    function_ref<void(Mountain)> fun(&normal_function);  
    fun(Mountain("Kilimanjaro"));  
}
```

Free Functions

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

Free Functions

```
template <typename R, typename... Args>
class function_ref<R(Args...)> {
    template <typename R2, typename... Args2>
        function_ref(R2 (*ptr)(Args2...)) ;
};
```

ptr Points to some free function.

Free Functions

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

    template <typename R2, typename... Args2>
    requires (convertible_from<Args2, Args>)

    function_ref(R2 (*ptr)(Args2...));

};
```

Convert from each Arg to corresponding Arg2.

Free Functions

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

    template <typename R2, typename... Args2>
    requires (convertible from<Args2, Args> && ...
             && convertible_from<R, R2>)
    function_ref(R2 (*ptr)(Args2...));

};
```

Convert from each Arg to corresponding Arg2.

Convert from R2 to R.

Free Functions

```
template <typename To, typename From>  
concept convertible_from =
```

Can I create a To value using a From value?

Free Functions

```
template <typename To, typename From>  
concept convertible from =  
    std::same_as<To, From> ||
```

Are To and From the same?

Free Functions

```
template <typename To, typename From>  
concept convertible_from =  
    std::same_as<To, From> ||  
    requires (From (&start)())
```

start creates a From value.

Free Functions

```
template <typename To, typename From>  
concept convertible_from =  
    std::same_as<To, From> ||  
    requires (From (&start)(), void (&finish)(To))
```

start creates a From value.
finish takes a To value.

Free Functions

```
template <typename To, typename From>
concept convertible_from =
    std::same_as<To, From> ||
    requires (From (&start)(), void (&finish)(To)) {
        finish(start());
    };
```

start creates a From value.
finish takes a To value.

Free Functions

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

    template // ...
    requires // ...
    function_ref(R2 (*ptr)(Args2...))

};
```

Free Functions

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

```
    template // ...
```

```
    requires // ...
```

```
    function_ref(R2 (*ptr)(Args2...))
```

```
        : m_bound(reinterpret_cast<void*>(ptr))
```

```
};
```

Free Functions

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

    template // ...
    requires // ...
    function_ref(R2 (*ptr)(Args2...))
        : m_bound(reinterpret_cast<void*>(ptr))
        , m_thunk(&free_fun<R2(Args2...)>::thunk) {}

};
```

Free Functions

```
template <typename R2, typename... Args2>  
struct free_fun<R2(Args2...)> {
```

```
    static R thunk
```

Forwards arguments to function.

```
};
```

Free Functions

```
template <typename R2, typename... Args2>  
struct free_fun<R2(Args2...)> {
```

```
    static R thunk(void* bound,
```

Points to original function.

```
};
```

Free Functions

```
template <typename R2, typename... Args2>
struct free_fun<R2(Args2...)> {

    static R thunk(void* bound,
                   thunk_arg_t<Args>... args) {

    }

};
```


Free Functions

```
template <typename R2, typename... Args2>
struct free_fun<R2(Args2...)> {
    static R thunk(void* bound,
                   thunk_arg_t<Args>... args) {
        The type of the original function.
    }
};
```

Free Functions

```
template <typename R2, typename... Args2>  
struct free_fun<R2(Args2...)> {
```

```
    static R thunk(void* bound,  
                  thunk_arg_t<Args>... args) {
```

```
        using fun_t = R2(*) (thunk_arg_t<Args2>...);
```

```
        fun_t is not the exact original function type.
```

```
    }
```

```
};
```

Free Functions

```
template <typename R2, typename... Args2>
struct free_fun<R2(Args2...)> {

    static R thunk(void* bound,
                   thunk_arg_t<Args>... args) {
        using fun_t = R2(*) (thunk_arg_t<Args2>...);

    }

};
```

The trick: convert non-trivial values into rvalue references.

Free Functions

```
template <typename R2, typename... Args2>
struct free_fun<R2(Args2...)> {

    static R thunk(void* bound,
                  thunk_arg_t<Args>... args) {
        using fun_t = R2(*) (thunk_arg_t<Args2>...);

        return reinterpret_cast<fun_t>(bound)
            (std::forward<Args>(args)...);
    }
};
```

Free Functions

```
void normal_function(Mountain m) {  
    m.climb();  
}
```

By value.

Free Functions

```
void normal_function(Mountain m) {  
    m.climb();  
}
```

```
int main(int, char**) {  
    function_ref<void(Mountain)> fun(&normal_function);  
  
}
```

Free Functions

```
void normal_function(Mountain m) {  
    m.climb();  
}  
  
int main(int, char**) {  
    function_ref<void(Mountain)> fun(&normal_function);  
    fun(Mountain("Kilimanjaro"));  
}
```

Member Functions

```
int main(int, char**) {  
    Class object;
```

```
}
```


Member Functions

```
int main(int, char**) {  
    Class object;
```

```
    function_ref<void(Mountain)> fun
```

```
}
```

Member Functions

```
int main(int, char**) {  
    Class object;  
  
    function_ref<void(Mountain)> fun(  
        nontype<&Class::member_function>,  
        object);  
  
}
```

Member Functions

```
int main(int, char**) {  
    Class object;  
  
    function_ref<void(Mountain)> fun(  
        nontype<&Class::member_function>,  
        object);  
  
    fun(Mountain("Blanc"));  
}
```

Member Functions

```
function_ref(nontype_t<MemFun>, Class& obj);
```

Member Functions

```
template <
    typename R2,           The member function return type
    typename Class,
    typename... Args2,

function_ref(nontype_t<MemFun>, Class& obj);
```

Member Functions

```
template <  
    typename R2,           The member function return type  
    typename Class,      The class type  
    typename... Args2,
```

```
function_ref(nontype_t<MemFun>, Class& obj);
```

Member Functions

```
template <  
    typename R2,           The member function return type  
    typename Class,       The class type  
    typename... Args2,    The member function argument types
```

```
function_ref(nontype_t<MemFun>, Class& obj);
```

Member Functions

```
template <  
    typename R2,  
    typename Class,  
    typename... Args2,  
    R2 (Class::*MemFun) (Args2...) > The member function
```

```
function_ref(nontype_t<MemFun>, Class& obj);
```


Member Functions

```
template <  
    typename R2,  
    typename Class,  
    typename... Args2,  
    R2 (Class::*MemFun)(Args2...)>  
requires (convertible_from<Args2, Args>  
function_ref(nontype_t<MemFun>, Class& obj));
```

Convert from each Arg to corresponding Arg2.

Member Functions

```
template <
    typename R2,
    typename Class,
    typename... Args2,
    R2 (Class::*MemFun)(Args2...)>
requires (convertible from<Args2, Args> && ...
    && convertible_from<R, R2>)
function_ref(nontype_t<MemFun>, Class& obj);
```

Convert from each Arg to corresponding Arg2.
Convert from R2 to R.

Member Functions

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

    template // ...
    requires // ...
    function_ref(nontype_t<MemFun>, Class& obj)

};
```

Member Functions

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

    template // ...
    requires // ...
    function_ref(nontype t<MemFun>, Class& obj)
        : m_bound(std::addressof(obj))

};
```

Member Functions

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

    template // ...
    requires // ...
    function_ref(nontype_t<MemFun>, Class& obj)
        : m_bound(std::addressof(obj))
        , m_thunk(&mem_fun<MemFun>::thunk) {}

};
```

Member Functions

```
template // ...  
struct mem_fun<MemFun> {  
    static R thunk
```

Forwards arguments to member function.

```
};
```

Member Functions

```
template // ...  
struct mem_fun<MemFun> {  
    static R thunk(void* bound
```

Points to object.

```
};
```

Member Functions

```
template // ...
struct mem_fun<MemFun> {
    static R thunk(void* bound,
                   thunk_arg_t<Args>... args) {

    }
};
```


Member Functions

```
template // ...
struct mem_fun<MemFun> {
    static R thunk(void* bound,
                  thunk_arg_t<Args>... args) {
```

The original member function.

```
};
```

Member Functions

```
template // ...
struct mem_fun<MemFun> {
    static R thunk(void* bound,
                   thunk_arg_t<Args>... args) {
        using fun_t = R2(Class::*)(thunk_arg_t<Args2>...);

        fun_t is not the same type as MemFun.

    }
};
```

Member Functions

```
template // ...
struct mem_fun<MemFun> {
    static R thunk(void* bound,
                  thunk_arg_t<Args>... args) {
        using fun_t = R2(Class::*)(thunk_arg_t<Args2>...);

```

The trick: convert non-trivial values into rvalue references.

```
};
```

Member Functions

```
template // ...
struct mem_fun<MemFun> {
    static R thunk(void* bound,
                   thunk_arg_t<Args>... args) {
        using fun_t = R2(Class::*)(thunk_arg_t<Args2>...);

        return (static_cast<Class*>(bound) ->*
                reinterpret_cast<fun_t>(MemFun))
                (std::forward<Args>(args)...);
    }
};
```

Member Functions

```
struct Class {  
    void member_function(Mountain m);  
};
```

By value.

Member Functions

```
struct Class {  
    void member_function(Mountain m);  
};
```

```
int main(int, char**) {  
    Class object;
```

```
}
```

Member Functions

```
struct Class {  
    void member_function(Mountain m);  
};
```

```
int main(int, char**) {  
    Class object;
```

```
    function_ref<void(Mountain)> fun(  
        nontype<&Class::member_function>,  
        object);
```

```
}
```

Member Functions

```
struct Class {  
    void member_function(Mountain m);  
};  
  
int main(int, char**) {  
    Class object;  
  
    function_ref<void(Mountain)> fun(  
        nontype<&Class::member_function>,  
        object);  
  
    fun(Mountain("Toubkal"));  
}
```


Function Objects

```
struct Class {  
    void operator()(Mountain m);  
};
```

By value.

Function Objects

```
struct Class {  
    void operator()(Mountain m);  
};
```

```
int main(int, char**) {  
    Class object;
```

```
    function_ref<void(Mountain)> fun(object);
```

```
}
```

Function Objects

```
struct Class {  
    void operator()(Mountain m);  
};  
  
int main(int, char**) {  
    Class object;  
  
    function_ref<void(Mountain)> fun(object);  
  
    fun(Mountain("Elbrus"));  
}
```

Calls `Class::operator()`

Function Objects

```
struct Class {
```

```
    void operator()(Mountain m);
```

Call this member function.

```
};
```

```
int main(int, char**) {
```

```
    function_ref<void(Mountain)> fun(
```

```
        nontype<&Class::operator()>,
```

```
        object);
```

```
}
```

Function Objects

```
struct Class {  
    void operator()(Mountain m);  
    void operator()(int);
```

Ambiguous.

```
};
```

```
int main(int, char**) {  
    function_ref<void(Mountain)> fun(  
        nontype<&Class::operator()>,  
        object);  
}
```

Function Objects

```
struct Class {
```

```
    template <typename T>
```

```
    void operator()(T);
```

Cannot deduce T.

```
};
```

```
int main(int, char**) {
```

```
    function_ref<void(Mountain)> fun(
```

```
        nontype<&Class::operator()>,
```

```
        object);
```

```
}
```

Function Objects

```
struct Class {
```

Need pointer to member function that would be invoked.

```
};
```

```
int main(int, char**) {  
    function_ref<void(Mountain)> fun(  
        nontype<&Class::operator()>,  
        object);  
}
```

Matching Function Signatures

```
template <typename T>
```

```
void high_order_function(void (*fun)(T));
```


Matching Function Signatures

```
template <typename T>  
void high_order_function(void (*fun)(T));
```

```
void normal_function(int);
```

Matching Function Signatures

```
template <typename T>  
void high_order_function(void (*fun)(T));
```

```
void normal_function(int);
```

```
int main(int, char**) {  
    high_order_function(&normal_function);  
}
```

Unambiguous. Deduces $T = \text{int}$.

Matching Function Signatures

```
template <typename T>  
void high_order_function(void (*fun)(T));
```

```
void normal_function(int);
```

```
void normal_function(long);
```

```
int main(int, char**) {  
    high_order_function(&normal_function);  
}
```

Matching Function Signatures

```
template <typename T>  
void high_order_function(void (*fun)(T));
```

```
void normal_function(int);  
void normal_function(long);
```

```
int main(int, char**) {  
high_order_function(&normal_function);  
}
```

Ambiguous. Cannot deduce T.

Matching Function Signatures

Not a template.

```
void high_order_function(void (*fun)(int));
```

```
void normal_function(int);
```

```
void normal_function(long);
```

```
int main(int, char**) {  
    high_order_function(&normal_function);  
}
```

Matching Function Signatures

```
void high_order_function(void (*fun)(int));
```

```
void normal_function(int);
```

```
void normal_function(long);
```

```
int main(int, char**) {  
    high_order_function(&normal_function);  
}
```

Selects matching overload.

Matching Function Signatures

```
struct Class {  
    void operator()(Mountain);  
    void operator()(int);
```

Not viable.

```
};
```

```
int main(int, char**) {  
    Class object;  
  
    function_ref<void(Mountain)> fun(object);  
}
```

Matching Function Signatures

```
struct Class {
```

```
    void operator()(Mountain &&);
```

Best match.

```
    void operator()(Mountain const&);
```

Viable, not best match.

```
};
```

```
int main(int, char**) {
```

```
    Class object;
```

```
    function_ref<void(Mountain)> fun(object);
```

```
}
```


Matching Function Signatures

```
// function_ref    // Class::operator()  
void(Arg)         -> void(Arg)
```

Accept this signature.

Matching Function Signatures

```
// function_ref    // Class::operator()  
void(Arg)         -> void(Arg)
```

```
void(Arg const&)
```

Accept this signature.

Matching Function Signatures

```
// function_ref      // Class::operator()  
void(Arg)           -> void(Arg)           // 1   Prefer this.
```

```
void(Arg const&)   // 2
```

Accept this signature.

Matching Function Signatures

```
// function_ref      // Class::operator()  
void(Arg)           -> void(Arg)           // 1   Same rank.  
                    void(Arg&&)           // 1   Same rank.  
                    void(Arg const&)     // 2
```

Matching Function Signatures

```
// function_ref    // Class::operator()  
void(Arg)          -> void(Arg)           // 1  
                   void(Arg&&)         // 1  
                   void(Arg const&)    // 2  
  
void(Arg&&)        -> void(Arg&&)           // 1
```

Matching Function Signatures

```
// function_ref    // Class::operator()  
void(Arg)          -> void(Arg)           // 1  
                   void(Arg&&)          // 1  
                   void(Arg const&)    // 2  
  
void(Arg&&)         -> void(Arg&&)           // 1  
                   void(Arg const&)    // 2
```

Matching Function Signatures

```
// function_ref    // Class::operator()  
void(Arg)         -> void(Arg)           // 1  
                  void(Arg&&)           // 1  
                  void(Arg const&)     // 2  
  
void(Arg&&)       -> void(Arg&&)           // 1  
                  void(Arg)           // 1  
If move constructible. void(Arg const&) // 2
```

Matching Function Signatures

```
// function_ref    // Class::operator()  
void(Arg)         -> void(Arg)           // 1  
                  void(Arg&&)          // 1  
                  void(Arg const&)    // 2  
  
void(Arg&&)        -> void(Arg&&)          // 1  
                  void(Arg)           // 1  
                  void(Arg const&)    // 2  
  
void(Arg const&) -> void(Arg const&)    // 1
```


Matching Function Signatures

```
// function_ref    // Class::operator()  
void(Arg)         -> void(Arg)           // 1  
                  void(Arg&&)         // 1  
                  void(Arg const&)    // 2  
  
void(Arg&&)        -> void(Arg&&)         // 1  
                  void(Arg)         // 1  
                  void(Arg const&)    // 2  
  
void(Arg const&) -> void(Arg const&)    // 1  
                  void(Arg)         // 2
```

If copy constructible.

Matching Function Signatures

```
void overloaded1(Mountain&& );  
void overloaded1(Mountain const&);
```

Best match for Mountain value?

Matching Function Signatures

```
void overloaded1(Mountain&& );  
void overloaded1(Mountain const&);
```

Tag<true> inherits from Tag<false>.

```
void match_signature(void (*) (Mountain), Tag<true>);  
void match_signature(void (*) (Mountain&&), Tag<true>);  
void match_signature(void (*) (Mountain const&), Tag<false>);
```

Matching Function Signatures

```
void overloaded1(Mountain&& );  
void overloaded1(Mountain const&);
```

```
void match_signature(void (*) (Mountain), Tag<true>);  
void match_signature(void (*) (Mountain&&), Tag<true>);  
void match_signature(void (*) (Mountain const&), Tag<false>);
```

```
int main(int, char** ) {  
    match_signature(&overloaded1, Tag<true>{});  
  
}
```

Matching Function Signatures

```
void overloaded1(Mountain&& );  
void overloaded1(Mountain const&);
```

match_signature corresponds to best overloaded1.

```
void match_signature(void (*)(Mountain), Tag<true>);  
void match_signature(void (*)(Mountain&&), Tag<true>);  
void match_signature(void (*)(Mountain const&), Tag<false>);
```

```
int main(int, char**) {  
    match_signature(&overloaded1, Tag<true>{});  
  
}
```

Matching Function Signatures

```
void overloaded2(Mountain);
```

```
void overloaded2(int);
```

```
void match_signature(void (*) (Mountain), Tag<true>);
```

```
void match_signature(void (*) (Mountain&&), Tag<true>);
```

```
void match_signature(void (*) (Mountain const&), Tag<false>);
```

```
int main(int, char** ) {
```

```
}
```

Matching Function Signatures

```
void overloaded2(Mountain);
```

```
void overloaded2(int);
```

```
void match_signature(void (*) (Mountain), Tag<true>);
```

```
void match_signature(void (*) (Mountain&&), Tag<true>);
```

```
void match_signature(void (*) (Mountain const&), Tag<false>);
```

```
int main(int, char** ) {
```

```
    match_signature(&overloaded2, Tag<true>{});
```

```
}
```

Matching Function Signatures

```
void overloaded2(Mountain);  
void overloaded2(int);
```

```
void match_signature(void (*)(Mountain), Tag<true>);  
void match_signature(void (*)(Mountain&&), Tag<true>);  
void match_signature(void (*)(Mountain const&), Tag<false>);
```

```
int main(int, char**) {  
  
    match_signature(&overloaded2, Tag<true>{});  
}
```


Matching Function Signatures

```
auto ptr_to_mem_fun =
```

Matching Function Signatures

```
auto ptr_to_mem_fun =  
    MatchSignature<Args...>
```

Matching Function Signatures

```
auto ptr_to_mem_fun =  
    MatchSignature<Args...>::template match<R, Class>
```

Matching Function Signatures

```
auto ptr_to_mem_fun =  
    MatchSignature<Args...>::template match<R, Class>(  
        &Class::operator(), Tag_v<Args>...);
```

Selects best overload.

Matching Function Signatures

```
template <typename T>  
inline constexpr Tag<true> Tag_v{};  
  
auto ptr_to_mem_fun =  
    MatchSignature<Args...>::template match<R, Class>(  
        &Class::operator(), Tag_v<Args>...);
```

Prefer better match.

Matching Function Signatures

```
template <typename T>
inline constexpr Tag<true> Tag_v{};

auto ptr_to_mem_fun =
    MatchSignature<Args...>::template match<R, Class>(
        &Class::operator(), Tag_v<Args>...);

template <typename... Args>
using MatchSignature =
    MatchSignatureImpl<TypeList<Args...>>;
```

Matching Function Signatures

```
struct MatchSignatureImpl
```

Matching Function Signatures

```
template <  
    typename First,  
    typename... Rest
```

```
struct MatchSignatureImpl<  
    TypeList<First, Rest...>
```

Examine arguments one at a time.

Matching Function Signatures

```
template <  
    typename First,  
    typename... Rest,  
    typename... Accum>  
struct MatchSignatureImpl<  
    TypeList<First, Rest...>, TypeList<Accum...>>
```

Accumulate results of examining each argument.

Matching Function Signatures

```
template <
    typename First,
    typename... Rest,
    typename... Accum>
struct MatchSignatureImpl<
    TypeList<First, Rest...>, TypeList<Accum...>>
: MatchSignatureImpl< /* ... */>
, MatchSignatureImpl< /* ... */>
, MatchSignatureImpl< /* ... */> {};
```

Recurse until all arguments examined.

Matching Function Signatures

```
template // ...  
struct MatchSignatureImpl<  
    TypeList<First const&, Rest...>, TypeList<Accum...>>
```

Specialize for const&.

Matching Function Signatures

```
template // ...  
struct MatchSignatureImpl<  
    TypeList<First const&, Rest...>, TypeList<Accum...>>  
: MatchSignatureImpl<  
    TypeList<Rest...>
```

Remove first argument.

Matching Function Signatures

```
template // ...
struct MatchSignatureImpl<
    TypeList<First const&, Rest...>, TypeList<Accum...>>
: MatchSignatureImpl<
    TypeList<Rest...>,
    TypeList<
        Accum...,
        Tagged<First const&, true>>> {};
```

Corresponding argument in match.

Matching Function Signatures

```
template // ...  
requires convertible_from<First, First const&>  
struct MatchSignatureImpl<  
    TypeList<First const&, Rest...>, TypeList<Accum...>>
```

Specialize when First is copyable.

Matching Function Signatures

```
template // ...
requires convertible_from<First, First const&>
struct MatchSignatureImpl<
    TypeList<First const&, Rest...>, TypeList<Accum...>>
: MatchSignatureImpl<
    TypeList<Rest...>,
    TypeList<Accum..., Tagged<First const&, true>>>
```

Matching Function Signatures

```
template // ...
requires convertible_from<First, First const&>
struct MatchSignatureImpl<
    TypeList<First const&, Rest...>, TypeList<Accum...>>
    : MatchSignatureImpl<
        TypeList<Rest...>,
        TypeList<Accum..., Tagged<First const&, true>>>
    , MatchSignatureImpl<
        TypeList<Rest...>,
        TypeList<Accum..., Tagged<First, false>>> {};
```

Corresponding argument can be a value.
Worse match than const&.

Matching Function Signatures

```
template // ...  
struct MatchSignatureImpl<  
    TypeList<>, TypeList<Tagged<Args, Tags>...> {
```

All arguments examined.

```
};
```

Matching Function Signatures

```
template // ...
struct MatchSignatureImpl<
    TypeList<>, TypeList<Tagged<Args, Tags>...> {

    template <typename R, typename Class>
    static constexpr auto match

};
```

Matching Function Signatures

```
template // ...  
struct MatchSignatureImpl<  
    TypeList<>, TypeList<Tagged<Args, Tags>...> {
```

```
    template <typename R, typename Class>  
    static constexpr auto match(  
        R (Class::*mem_fun)(Args...)
```

Matched function pointer.

```
};
```

Matching Function Signatures

```
template // ...
struct MatchSignatureImpl<
    TypeList<>, TypeList<Tagged<Args, Tags>...> {

    template <typename R, typename Class>
    static constexpr auto match(
        R (Class::*mem_fun)(Args...),
        Tag<Tags>...)

};
```

Matching Function Signatures

```
template // ...
struct MatchSignatureImpl<
    TypeList<>, TypeList<Tagged<Args, Tags>...> {

    template <typename R, typename Class>
    static constexpr auto match(
        R (Class::*mem_fun)(Args...),
        Tag<Tags>...) {
        return mem_fun;
    }

};
```

We have the correct overload address.

Matching Function Signatures

```
template <typename Class, typename... Args>  
using invoke_result_t =
```

Customized `invoke_result_t`. Differs from `std::invoke_result_t`.

Matching Function Signatures

```
template <typename Class, typename... Args>  
using invoke result t = decltype(  
    std::declval<Class&>())
```

Invoke a reference to Class.

Matching Function Signatures

```
template <typename Class, typename... Args>  
using invoke_result_t = decltype(  
    std::declval<Class&>()  
    std::declval<Args(&)()>()...));
```

Pass it Args values.

Matching Function Signatures

```
template <typename Class, typename... Args>  
constexpr auto match_signature()
```

Returns pointer to best overload of operator().

Matching Function Signatures

```
template <typename Class, typename... Args>  
constexpr auto match_signature()  
    -> decltype(MatchSignature<Args...>::template match
```

Matching Function Signatures

```
template <typename Class, typename... Args>  
constexpr auto match_signature()  
-> decltype(MatchSignature<Args...>::template match<  
    invoke_result_t<Class, Args...>
```

Must explicitly specify return type.

Matching Function Signatures

```
template <typename Class, typename... Args>  
constexpr auto match_signature()  
-> decltype(MatchSignature<Args...>::template match<  
    invoke_result_t<Class, Args...>, Class>
```

Must explicitly specify class type.

Matching Function Signatures

```
template <typename Class, typename... Args>
constexpr auto match_signature()
-> decltype(MatchSignature<Args...>::template match<
    invoke_result_t<Class, Args...>, Class>(
        &Class::operator()))
```

Matching Function Signatures

```
template <typename Class, typename... Args>
constexpr auto match_signature()
-> decltype(MatchSignature<Args...>::template match<
    invoke_result_t<Class, Args...>, Class>(
        &Class::operator(),
        Tag_v<Args>...))
```

Matching Function Signatures

```
template <typename Class, typename... Args>
constexpr auto match_signature()
-> decltype(MatchSignature<Args...>::template match<
    invoke_result_t<Class, Args...>, Class>(
        &Class::operator(),
        Tag_v<Args>...)) {
return MatchSignature<Args...>::template match<
    invoke_result_t<Class, Args...>, Class>(
        &Class::operator(),
        Tag_v<Args>...);
}
```

Function Objects

```
template <typename R, typename... Args>
class function_ref<R(Args...)> {
    template <typename Class>
        function_ref(Class& obj);
};
```


Function Objects

```
template <typename R, typename... Args>  
class function_ref<R(Args...)> {  
    template <typename Class>  
    requires
```

```
    convertible_from<R, invoke_result_t<Class, Args...>>
```

Convert from whatever Class returns to R.

```
function_ref(Class& obj);  
};
```

Function Objects

```
template <typename R, typename... Args>
class function_ref<R(Args...)> {
    template <typename Class>
    requires
        convertible_from<R, invoke_result_t<Class, Args...>>
        && requires() {
            match_signature<Class, Args...>();
        }
    function_ref(Class& obj);
};
```

match_signature() can find the best overload.

Function Objects

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

    template // ...
    function_ref(Class& obj)

};
```

Function Objects

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

```
    template // ...  
    function_ref(Class& obj) : function_ref
```

Delegate to another constructor.

```
};
```

Function Objects

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

```
    template // ...
```

```
    function_ref(Class& obj) : function_ref(
```

```
        nontype<match_signature<Class, Args...>()>
```

Address of best overload.

```
};
```

Function Objects

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

    template // ...
    function_ref(Class& obj) : function_ref(
        nontype<match_signature<Class, Args...>()>,
        obj) {}

};
```

Function Objects

```
int main(int, char**) {  
    auto lambda = [] (Mountain m) { m.climb(); }  
  
}
```

Function Objects

```
int main(int, char**) {  
    auto lambda = [](Mountain m) { m.climb(); };  
  
      
  
}
```

By value.

Function Objects

```
int main(int, char**) {  
    auto lambda = [](Mountain m) { m.climb(); };  
  
    function_ref<void(Mountain)> fun(lambda);  
  
}
```

Function Objects

```
int main(int, char**) {  
    auto lambda = [](Mountain m) { m.climb(); };  
  
    function_ref<void(Mountain)> fun(lambda);  
  
    fun(Mountain("Aconcagua"));  
}
```

Thank You



<https://gitlab.com/bbfgelman11/pbv>

TechAtBloomberg.com/cplusplus

Engineering

Bloomberg