THE SITUATION

```
void sort(range, comp_func)
```

templates allow for an elegant ZOA

```
template<typename Func> void sort(range, Func comp_func)
```

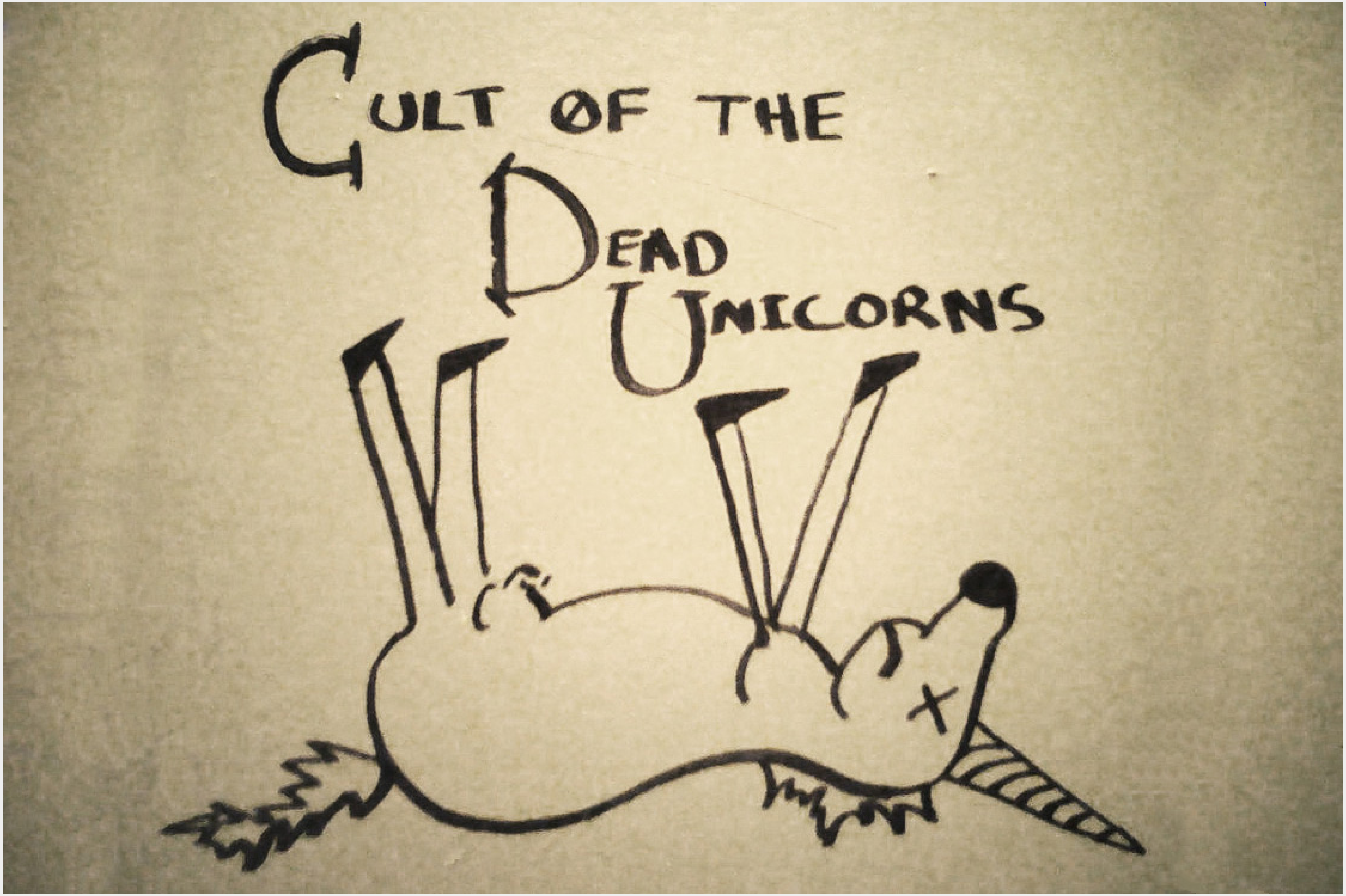
But what if want to implement something like a task queue?

```
template<typename Func> std::vector<Func> queue;
```

`std::tuple`

Enter `std::function`, the *magic* solution.

So how do you implement *magic*?


Cult of the
Dead
Unicorns

MAYBE IT'S A BAD PATTERN?

C++

YOU ONLY PAY FOR WHAT YOU USE

CLOSURE

```cpp
int val = 3;
auto lam = [val](int arg) -> int { return arg + val; };
```

```cpp
struct closure
{
    int val;
    explicit closure(int v) : val(v) {}

    int operator() (int arg)
    {
        return arg + val;
    }
};

int val = 3;
closure old{ val };
```

```
for (int i = 1; i <= 10; ++i)
    std::cout << i << '\n';
```

SIZE OF CALLABLE *THINGS*

Function pointers

```
void foo(int arg);

void bar()
{
    auto f = foo;
    std::cout << sizeof(f);
}
```

8 or 4 byte

Lambdas

```
auto f = [](char arg) {};
std::cout << sizeof(f);
```

1 byte

```
int val = 3;
auto f = [val](int arg) {};
std::cout << sizeof(f);
```

sizeof(int) bytes

```
auto f = [](char arg[10]) {};
std::cout << sizeof(f);
```

1 byte

FUNCTION POINTER TYPES

```
void foo(int arg);
auto f = foo;
```

```
decltype(f) == void(*)(int);
```

MEMBER FUNCTION POINTERS

```
class foo
{
    void bar(int arg);
};

auto f = &foo::bar;
```

```cpp
class foo
{
public:
    void bar(int arg);
};

auto f = &foo::bar;
```

```cpp
decltype(f) == void(foo::* bar)(int)
```

```cpp
f(3);
```

```cpp
foo obj;
(obj.*f)(3);
```

```
auto f = [](int arg) {};
```

```
decltype(f) == decltype(f)
```

ALTERNATIVES

COROUTINES

THE BASICS

```
void foo() {}
```

subroutine

```
coro_return_type<int> test()
{
    co_await coro_awaitable_type{};
}
```

coroutine

```cpp
util::coro_task task_;

template<typename F> explicit delegate(F&& func)
{
    task_ = [](auto func) -> util::coro_task
        {
            co_await std::experimental::suspend_always{};
            func();
        }(std::forward<F>(func))
    );
}
```

THAT'S IT?

FUNCTION POINTERS

```
int foo() { return 3; }

void bar()
{
    int(*f)() = foo;
    std::cout << (**********/*INLINE COMMENTS YAY*/***********f)(
}
```

```cpp
void(*f)(int) = [](int arg) { std::cout << arg; };
f(3);
```

```cpp
class delegate
{
public:
    using invoke_ptr_t = void(*)(int);

    explicit delegate(invoke_ptr_t f) : invoke_ptr_(f) {}

    void operator() (int arg)
    {
        invoke_ptr_(arg);
    }
private:
    invoke_ptr_t invoke_ptr_;
};
```

```
delegate del{ [](int arg) { std::cout << arg; } };
del(3);
```

GENERIC

```cpp
int arr[10];
arr[0] = 2;

delegate<void(int)> del{
    [arr](int arg)
    { std::cout << arg + arr[0]; }
};

del(1);
```

Only captureless lambdas are convertible to function pointers

CONVERT TO FUNCTIONAL PROGRAMMING

STATIC

```cpp
static int val = 4;
auto pure = [](int arg)
{
    return arg + val;
};
std::cout << pure(-1);
```

```cpp
int(*f)(int) = pure;
```

```cpp
thread_local static T cap{ std::forward<T>(closure) };

invoke_ptr_ = static_cast<invoke_ptr_t>([](Args... args) -> R
{
    return cap(std::forward<Args>(args)...);
});
```

```cpp
thread_local static T cap{ std::forward<T>(closure) };

invoke_ptr_ = static_cast<invoke_ptr_t>([](Args&&... args) -> R
{
    return cap(std::forward<Args>(args)...);
});
```

```cpp
using del_t = delegate<int(void)>;

std::vector<del_t> vec;

for (int i = 0; i <= 3; ++i)
    vec.emplace_back([i]() { return i; });

std::cout << vec.back()();
```

```cpp
thread_local static T cap{ std::forward<T>(closure) };
```

Easy to use correctly

Easy to use incorrectly

Easy to use correctly

Hard to use incorrectly

HOW ABOUT WE STORE THE CLOSURE INPLACE

```cpp
template<typename R, typename... Args> class delegate<R(Args...)>
{
public:
    using invoke_ptr_t = R(*)(Args...);
    using storage_t = ???
}
```

void*

`std::aligned_storage`

```
{
    std::aligned_storage<sizeof(int), alignof(int)> storage;
    new(&storage)int{ 3 };

    std::cout << reinterpret_cast<int&>(storage);
}
std::cout << 6;
```

UB

```
{
    std::aligned_storage<sizeof(int), alignof(int)> storage;
    new(&storage)int{ 3 };

    std::cout << reinterpret_cast<int&>(storage);
}
std::cout << 6;
```

```cpp
{
    std::aligned_storage<sizeof(int), alignof(int)>::type storage
    new(&storage)int{ 3 };

    std::cout << reinterpret_cast<int&>(storage);
}
std::cout << 6;
```

```cpp
template<typename T> explicit delegate(T&& closure)
{
    new(&storage_)T{ std::forward<T>(closure) };

    invoke_ptr_ = static_cast<invoke_ptr_t>(
        [](storage_t& storage, Args&&... args) -> R
        {
            return reinterpret_cast<T&>(storage)(
                std::forward<Args>(args)...
            );
        }
    );
}
```

ZOA

How about we split interface and implementation

- pure
- inplace_triv
- inplace
- dynamic

```
~inplace()
{
    // call closure destructor
}
```

MORE FUNCTION POINTERS

```cpp
template<typename T> explicit inplace(T&& closure)
{
    // ... same as before

    destructor_ptr_ = static_cast<destructor_ptr_t>(
        [](storage_t& storage) noexcept -> void
        { reinterpret_cast<T&>(storage).~T(); }
    );
}
```

```
~inplace()
{
    destructor_ptr_(storage_);
}
```

```cpp
template<typename T> explicit inplace(T&& closure)
{
    // ... same as before

    copy_ptr_ = copy_op<T, storage_t>();
}
```

```
template<
    typename T,
    typename S,
    typename std::enable_if_t<
    std::is_copy_constructible<T>::value, int
    > = 0
> copy_ptr_t copy_op()
{
    return [](S& dst, S& src) noexcept -> void
    {
        new(&dst)T{ reinterpret_cast<T&>(src) };
    };
}
```

```cpp
template<
    typename T,
    typename S,
    typename std::enable_if_t<
    !std::is_copy_constructible<T>::value, int
    > = 0
> copy_ptr_t copy_op()
{
    static_assert(std::is_copy_constructible<T>::value,
        "constructing delegate with move only type is invalid!");
}
```

```
del_t del_a = rand_bool ? copy_move_closure{} : move_only_closure

del_t del_b = del_a; // can we copy?
```

```cpp
inplace(const inplace& other) :
    invoke_ptr_{ other.invoke_ptr_ },
    copy_ptr_{ other.copy_ptr_ },
    destructor_ptr_{ other.destructor_ptr_ }
{
    copy_ptr_(storage_, other.storage_);
}
```

```
inplace& operator= (const inplace& other)
{
    if (this != std::addressof(other))
    {
        invoke_ptr_ = other.invoke_ptr_;
        copy_ptr_ = other.copy_ptr_;

        destructor_ptr_(storage_);
        copy_ptr_(storage_, other.storage_);
        destructor_ptr_ = other.destructor_ptr_;
    }
    return *this;
}
```

```cpp
R operator() (Args&&... args) const
{
    return invoke_ptr_(storage_, std::forward<Args>(args)...);
}
```

```
private:
    mutable storage_t storage_;

    invoke_ptr_t invoke_ptr_;
    copy_ptr_t copy_ptr_;
    destructor_ptr_t destructor_ptr_;
```

DESIGNING THE INTERFACE

Why did we even split interface and implementation?

How do we bring it all together?

VARIANT

Was all that for nothing?

Is there just no better way of solving this problem?

I WAS TRYING TO SOLVE THE WRONG PROBLEM

C++

YOU ONLY PAY FOR WHAT YOU USE

```cpp
template<
    typename T,
    template<size_t, typename, typename...>class Spec = spec::inp
    size_t size = detail::default_capacity
>
class delegate; // unspecified

template<
    typename R, typename... Args,
    template<size_t, typename, typename...>class Spec,
    size_t size
>
class delegate<R(Args...), Spec, size>;
```

Let's take a look a the result

```
R operator() (Args&&... args) const
{
    if (empty)
        throw std::bad_function_call();

    return invoke_ptr_(storage_, std::forward<Args>(args)...);
}
```

```cpp
template<
    typename R,
    typename S,
    typename... Args
> static R empty_inplace(S&, Args&&... args)
{
    throw std::bad_function_call();
}
```

```
explicit inplace() noexcept :
    invoke_ptr_{ empty_inplace<R, storage_t, Args...> },
    copy_ptr_{ copy_op<std::nullptr_t, storage_t>() },
    destructor_ptr_{ [](storage_t&) noexcept -> void {} }
{
    new(&storage_)std::nullptr_t{ nullptr };
}
```

```cpp
bool empty() const noexcept
{
    return reinterpret_cast<std::nullptr_t&>(storage_) == nullptr
}
```

BENCHMARKS

TEST DRIVEN DEVELOPMENT

TYPE ORIENTED DESIGN

LESSONS LEARNED

- Do not be afraid to challenge a status quo!
- The price of magic is runtime
- Be responsible for your state

QUESTIONS

How do you know it works?

LINKS:

- email: lukas.bergdoll@gmail.com
- github
- James McNellis - "my favorite C++ feature"
- David Sankel - "Variants: Past, Present, and Future"
- Full implementation

HIRING?