

делаем обёртку с семантикой значения для полиморфных типов

Fun with type erasure:

implementing a value wrapper for polymorphic types

Павел Новиков

X @cpp_are

- a *very* brief refresher on polymorphism and pointer semantics
- most basic value wrapper using `std::any`
 - and its drawbacks for this application
- minimal implementation of value wrapper
 - *very* economical (*best value for your bytes!*) (get it?)
 - with small object optimization
- what else can we do?
 - saving pointer to interface
 - supporting move, copy assign and move assign
- adding other features
 - emplace, and assignment from values

We'll discuss

- a *very* brief refresher on polymorphism and pointer semantics
- most basic value wrapper using `std::any`
 - and its drawbacks for this application
- minimal implementation of value wrapper
 - *very* economical (best *value* for your bytes!) (get it?)
 - with small object optimization
- what else can we do?
 - saving pointer to interface
 - supporting move, copy assign and move assign
- adding other features
 - emplace, and assignment from values

We'll discuss

- a *very* brief refresher on polymorphism and pointer semantics
- most basic value wrapper using `std::any`
 - and its drawbacks for this application
- minimal implementation of value wrapper
 - *very* economical (best *value* for your bytes!) (get it?)
 - with small object optimization
- what else can we do?
 - saving pointer to interface
 - supporting move, copy assign and move assign
- adding other features
 - emplace, and assignment from values

We'll discuss

- a *very* brief refresher on polymorphism and pointer semantics
- most basic value wrapper using `std::any`
 - and its drawbacks for this application
- minimal implementation of value wrapper
 - *very* economical (*best value for your bytes!*) (get it?)
 - with small object optimization
- what else can we do?
 - saving pointer to interface
 - supporting move, copy assign and move assign
- adding other features
 - emplace, and assignment from values

We'll discuss

- a *very* brief refresher on polymorphism and pointer semantics
- most basic value wrapper using `std::any`
 - and its drawbacks for this application
- minimal implementation of value wrapper
 - *very* economical (*best value for your bytes!*) (get it?)
 - with small object optimization
- what else can we do?
 - saving pointer to interface
 - supporting move, copy assign and move assign
- adding other features
 - emplace, and assignment from values

We'll discuss

https://www.youtube.com/watch?v=mU_n_ohHQk

Meeting C++ 2020

META POLYMORPHISM ?



Jonathan Boccara - Meta Polymorphism



The idea

```
struct Shape {  
    virtual ~Shape() = default;  
    virtual void draw() = 0;  
    virtual void changeSize(float sx, float sy) = 0;  
};
```



Pointer semantics

```
struct Ellipse : Shape {  
    void draw() override;  
    void changeSize(float sx, float sy) override;  
};
```

```
struct Rectangle : Shape {  
    void draw() override;  
    void changeSize(float sx, float sy) override;  
};
```

```
struct Star : Shape {  
    void draw() override;  
    void changeSize(float sx, float sy) override;  
};
```

Pointer semantics

```
std::unique_ptr<Shape> s = std::make_unique<Star>();
```



type erasure happens at this point

Pointer semantics

```
std::unique_ptr<Shape> s = std::make_unique<Star>();
```

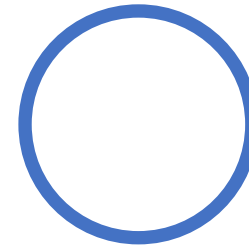
- pointer/reference semantics
- generally **can not** copy/clone values
- generally has to use memory allocation

Pointer semantics

```
std::vector<std::unique_ptr<Shape>> shapes;
```

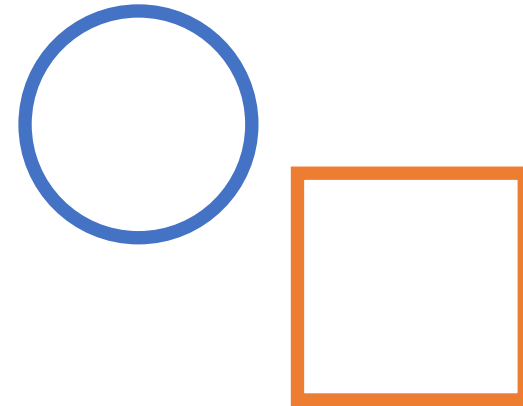
Pointer semantics


```
std::vector<std::unique_ptr<Shape>> shapes;  
shapes.push_back(std::make_unique<Ellipse>());
```



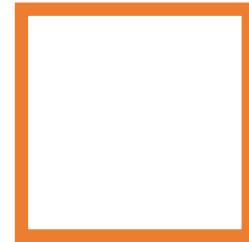
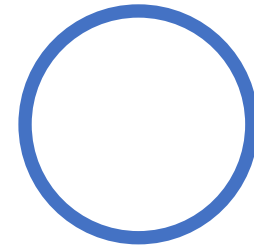
Pointer semantics

```
std::vector<std::unique_ptr<Shape>> shapes;  
shapes.push_back(std::make_unique<Ellipse>());  
shapes.push_back(std::make_unique<Rectangle>());
```



Pointer semantics

```
std::vector<std::unique_ptr<Shape>> shapes;  
shapes.push_back(std::make_unique<Ellipse>());  
shapes.push_back(std::make_unique<Rectangle>());  
shapes.push_back(std::make_unique<Star>());
```



Pointer semantics

```
std::vector<std::unique_ptr<Shape>> shapes;  
shapes.push_back(std::make_unique<Ellipse>());  
shapes.push_back(std::make_unique<Rectangle>());  
shapes.push_back(std::make_unique<Star>());
```

```
auto &shape1 = shapes[1];  
shape1->changeSize(3.f, 5.f);
```

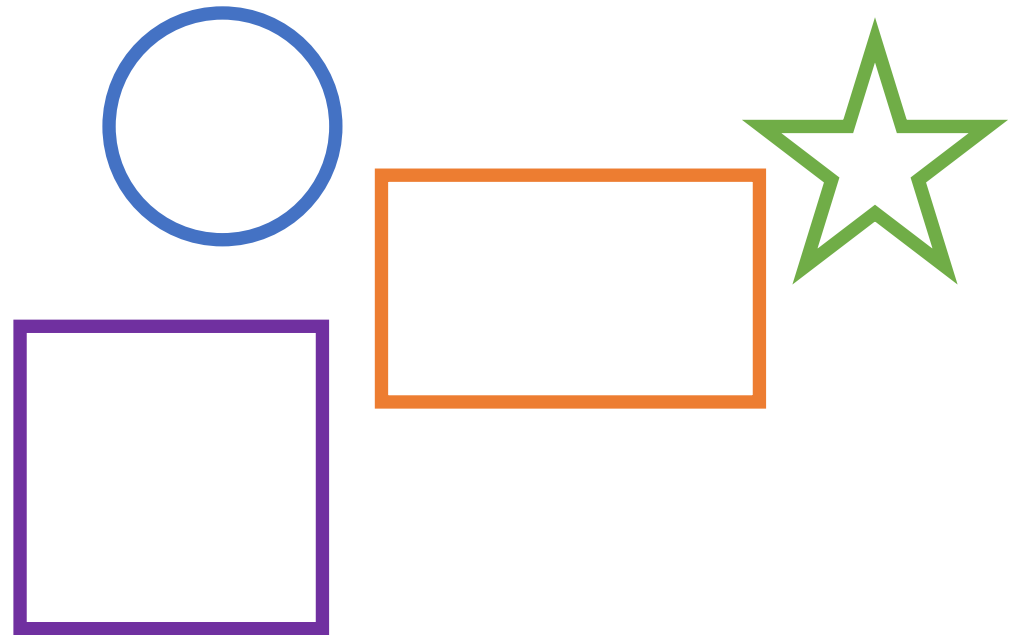


Pointer semantics

```
std::vector<std::unique_ptr<Shape>> shapes;  
shapes.push_back(std::make_unique<Ellipse>());  
shapes.push_back(std::make_unique<Rectangle>());  
shapes.push_back(std::make_unique<Star>());
```

```
auto &shape1 = shapes[1];  
shape1->changeSize(3.f, 5.f);
```

```
auto clone = ???  
clone.changeSize(4.f, 4.f);
```

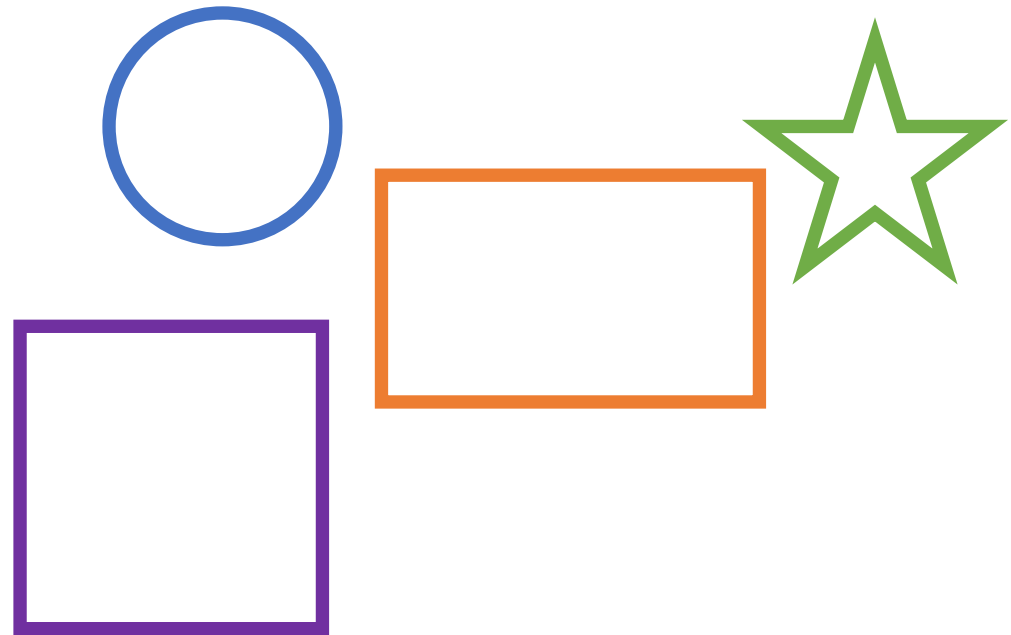


Pointer semantics

```
std::vector<ValueWrapper<Shape>> shapes;  
shapes.push_back(ValueWrapper<Shape>{ Ellipse{} });  
shapes.push_back(ValueWrapper<Shape>{ Rectangle{} });  
shapes.push_back(ValueWrapper<Shape>{ Star{} });
```

```
auto &shape1 = shapes[1];  
shape1.value().changeSize(3.f, 5.f);
```

```
auto clone = shape1;  
clone.value().changeSize(4.f, 4.f);
```

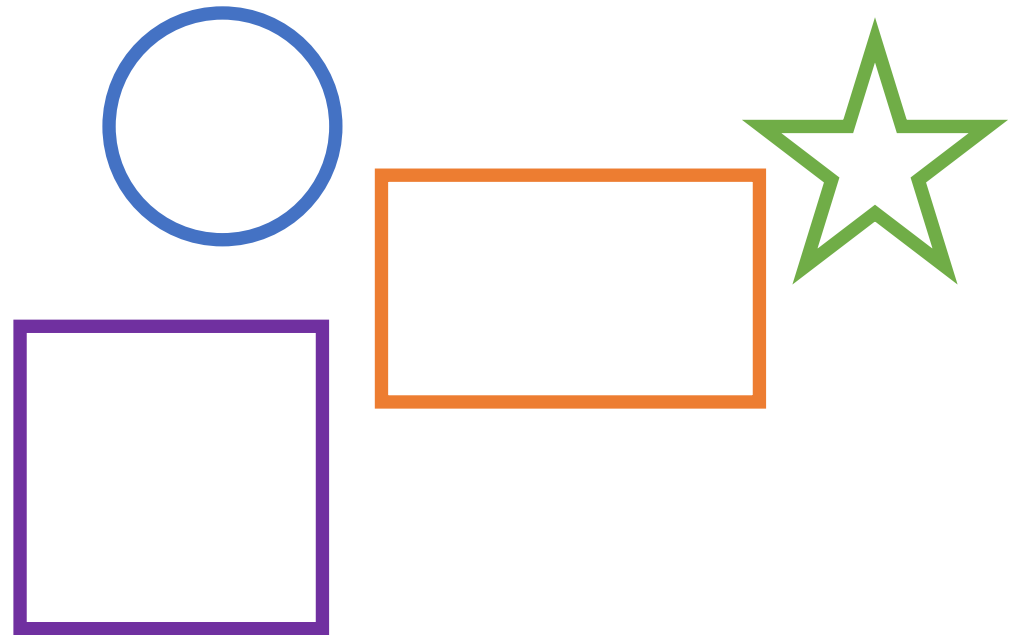


Value semantics?

```
std::vector<ValueWrapper<Shape>> shapes;  
shapes.emplace_back(Ellipse{});  
shapes.emplace_back(Rectangle{});  
shapes.emplace_back(Star{});
```

```
auto &shape1 = shapes[1];  
shape1.value().changeSize(3.f, 5.f);
```

```
auto clone = shape1;  
clone.value().changeSize(4.f, 4.f);
```



Value semantics?


```
template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T>
        requires (std::is_base_of_v<Interface, std::decay_t<T>> &&
                 !std::is_same_v<std::decay_t<T>, ValueWrapper>)
    explicit ValueWrapper(T &&object) //...
    //...
    Interface &value() { /*...*/ }
    const Interface &value() const { /*...*/ }
    bool hasValue() const { /*...*/ }
private:
    std::any data;
    const Interface &(*getter)(const std::any&) = nullptr;
};
```

Basic solution


```

template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T>
        requires (std::is_base_of_v<Interface, std::decay_t<T>> &&
                 !std::is_same_v<std::decay_t<T>, ValueWrapper>)
    explicit ValueWrapper(T &&object) //...
    //...
    Interface &value() { /*...*/ }
    const Interface &value() const { /*...*/ }
    bool hasValue() const { /*...*/ }
private:
    std::any data;
    const Interface &(*getter)(const std::any&) = nullptr;
};

```



should be `std::remove_cvref_t<T>`

Basic solution

```
template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T>
        requires (std::is_base_of_v<Interface, std::decay_t<T>> &&
                 !std::is_same_v<std::decay_t<T>, ValueWrapper>)
    explicit ValueWrapper(T &&object) //...
    //...
    Interface &value() { /*...*/ }
    const Interface &value() const { /*...*/ }
    bool hasValue() const { /*...*/ }
private:
    std::any data;
    const Interface &(*getter)(const std::any&) = nullptr;
};
```

Basic solution

```

template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T,
        std::enable_if_t<std::is_base_of_v<Interface, std::decay_t<T>> &&
            !std::is_same_v<std::decay_t<T>, ValueWrapper>>* = nullptr>
    explicit ValueWrapper(T &&object) //...
    //...
    Interface &value() { /*...*/ }
    const Interface &value() const { /*...*/ }
    bool hasValue() const { /*...*/ }
private:
    std::any data;
    const Interface &(*getter)(const std::any&) = nullptr;
};

```

Basic solution

```
template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T>
        requires (std::is_base_of_v<Interface, std::decay_t<T>> &&
                 !std::is_same_v<std::decay_t<T>, ValueWrapper>)
    explicit ValueWrapper(T &&object) //...
    //...
    Interface &value() { /*...*/ }
    const Interface &value() const { /*...*/ }
    bool hasValue() const { /*...*/ }
private:
    std::any data;
    const Interface &(*getter)(const std::any&) = nullptr;
};
```

Basic solution

```
template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T>
        requires (std::is_base_of_v<Interface, std::decay_t<T>> &&
                 !std::is_same_v<std::decay_t<T>, ValueWrapper>)
    explicit ValueWrapper(T &&object) //...
    //...
    Interface &value() { /*...*/ }
    const Interface &value() const { /*...*/ }
    bool hasValue() const { /*...*/ }
private:
    std::any data;
    const Interface &(*getter)(const std::any&) = nullptr;
};
```

Basic solution

```
template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T>
        requires (std::is_base_of_v<Interface, std::decay_t<T>> &&
                 !std::is_same_v<std::decay_t<T>, ValueWrapper>)
    explicit ValueWrapper(T &&object) //...
    //...
    Interface &value() { /*...*/ }
    const Interface &value() const { /*...*/ }
    bool hasValue() const { /*...*/ }
private:
    std::any data;
    const Interface &(*getter)(const std::any&) = nullptr;
};
```

Basic solution

```
//...
template<typename T>
    requires (std::is_base_of_v<Interface, std::decay_t<T>> &&
              !std::is_same_v<std::decay_t<T>, ValueWrapper>)
explicit ValueWrapper(T &&object) :
    data{ std::forward<T>(object) },
    getter{
        [](const std::any &a)->const Interface& {
            return std::any_cast<const std::decay_t<T>&>(a);
        } }
{}
//...
```

Basic solution

```
//...
template<typename T>
    requires (std::is_base_of_v<Interface, std::decay_t<T>> &&
              !std::is_same_v<std::decay_t<T>, ValueWrapper>)
explicit ValueWrapper(T &&object) :
    data{ std::forward<T>(object) },
    getter{
        [](const std::any &a)->const Interface& {
            return std::any_cast<const std::decay_t<T>&>(a);
        } }
{}
//...
```

Basic solution


```
//...  
Interface &value() {  
    return const_cast<Interface&>(getter(data));  
}  
const Interface &value() const {  
    return getter(data);  
}  
  
bool hasValue() const {  
    return data.has_value();  
}  
//...
```

Basic solution

```
//...  
Interface &value() {  
    return const_cast<Interface&>(getter(data));  
}  
const Interface &value() const {  
    return getter(data);  
}  
  
bool hasValue() const {  
    return data.has_value();  
}  
//...
```

Basic solution

init

```
auto v = ValueWrapper<Shape>{ Star{} };
```

copy/clone

```
auto x = v;
```

move

```
auto y = std::move(v);
```

copy assign

```
x = y;
```

move assign

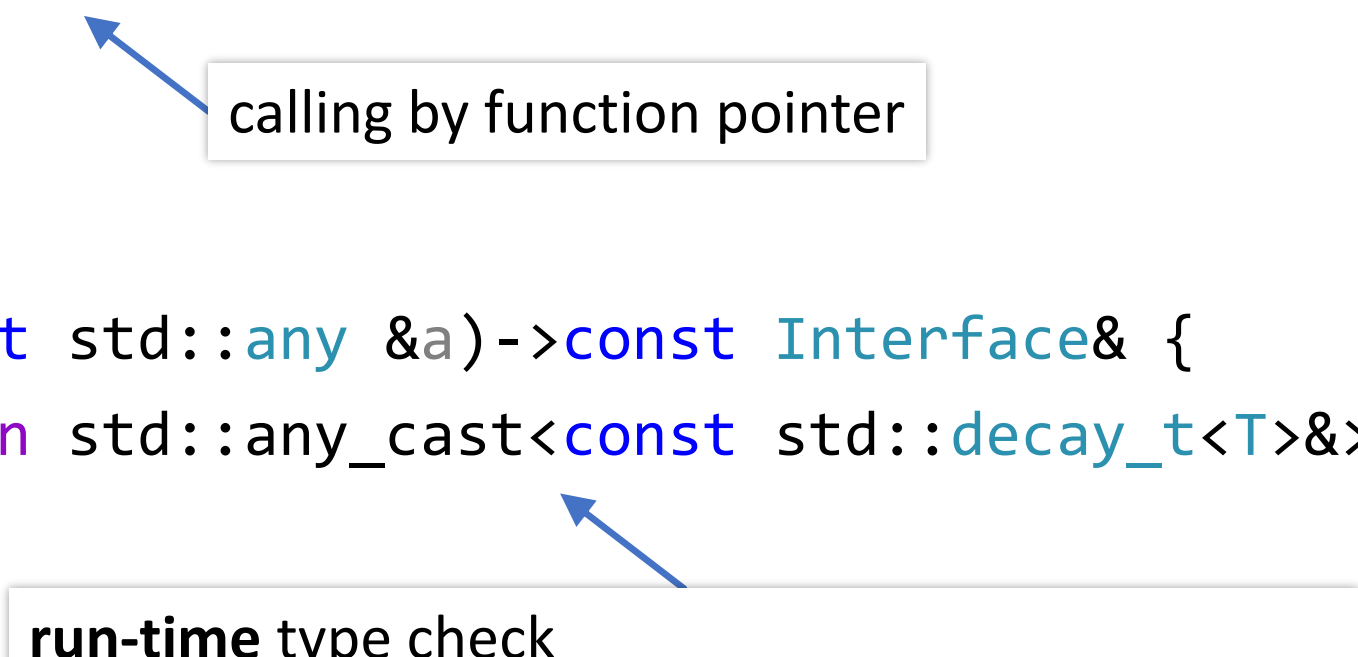
```
x = std::move(y);
```

using underlying value

```
x.value().changeSize(3.f, 3.f);
```

Basic solution

```
const Interface &operator->() const {  
    return getter(data);  
}  
//...  
getter{  
    [](const std::any &a)->const Interface& {  
        return std::any_cast<const std::decay_t<T>&>(a);  
    } }  
//...
```



The diagram consists of two callout boxes with arrows pointing to specific parts of the code. The first box, labeled "calling by function pointer", has an arrow pointing to the `getter(data)` call in the `operator->` function. The second box, labeled "run-time type check + another type 'un-erasure' call to get a reference", has an arrow pointing to the `std::any_cast` function call in the `getter` lambda.

run-time type check
+ another type "un-erasure" call to get a reference

Basic solution: drawbacks

std::any small object optimization:

std lib	sizeof(std::any)	small object capacity
MSVC	64	$((6 + 16 / \text{sizeof}(\text{void}*)) - 2) * \text{sizeof}(\text{void}*) = 48$ (56 for trivial types)
libstdc++	16	$\text{sizeof}(\text{void}*) = 8$
libc++	32	$3 * \text{sizeof}(\text{void}*) = 24$

* for x64

+ sizeof(getter) for ValueWrapper

Basic solution: drawbacks

std::any small object optimization:

std lib	sizeof(std::any)	small object capacity
MSVC	64	$((6 + 16 / \text{sizeof}(\text{void}*)) - 2) * \text{sizeof}(\text{void}*) = 48$ (56 for trivial types)
libstdc++	16	$\text{sizeof}(\text{void}*) = 8$
libc++	32	$3 * \text{sizeof}(\text{void}*) = 24$

* for x64

+ sizeof(getter) for ValueWrapper

Basic solution: drawbacks

std::any small object optimization:

std lib

MSVC

libstdc++

libc++

* for x64

+ sizeof



small object capacity

$$((6 + 16 / \text{sizeof}(\text{void}*)) - 2) * \text{sizeof}(\text{void}*) = 48$$

(56 for trivial types)

$$\text{sizeof}(\text{void}*) = 8$$

$$3 * \text{sizeof}(\text{void}*) = 24$$

Basic solution: drawbacks

std::any small object optimization:

small buffer

+ pointer to function table

+ pointer to type info

std lib	<code>sizeof(std::any)</code>	small object capacity
MSVC	64	$((6 + 16 / \text{sizeof}(\text{void}^*)) - 2) * \text{sizeof}(\text{void}^*) = 48$ (56 for trivial types)
libstdc++	16	$\text{sizeof}(\text{void}^*) = 8$
libc++	32	$3 * \text{sizeof}(\text{void}^*) = 24$

* for x64

+ `sizeof(getter)` for `ValueWrapper`


Basic solution: drawbacks

std::any small object optimization:

std lib	<code>sizeof(std::any)</code>	small object capacity
MSVC	64	$((6 + 16 / \text{sizeof}(\text{void}^*)) - 2) * \text{sizeof}(\text{void}^*) = 48$ (56 for trivial types)
libstdc++	16	$\text{sizeof}(\text{void}^*) = 8$
libc++	32	$3 * \text{sizeof}(\text{void}^*) = 24$

* for x64

small buffer
+ pointer to function



+ `sizeof(getter)` for `ValueWrapper`

Basic solution: drawbacks

std::any small object optimization:

std lib	sizeof(std::any)	small object capacity
MSVC	64	$((6 + 16 / \text{sizeof}(\text{void}^*)) - 2) * \text{sizeof}(\text{void}^*) = 48$ (56 for trivial types)
libstdc++	16	$\text{sizeof}(\text{void}^*) = 8$
libc++	32	$3 * \text{sizeof}(\text{void}^*) = 24$

* for x64

+ sizeof(getter) for ValueWrapper

Basic solution: drawbacks

```
template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T>
        requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
    explicit ValueWrapper(T &&object);
    ValueWrapper(const ValueWrapper &other);
    ~ValueWrapper();
    ValueWrapper &operator=(const ValueWrapper &other);
    Interface &value();
    const Interface &value() const;
    bool hasValue() const;
private:
    //...
};
```

Minimal implementation

```
template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T>
        requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
    explicit ValueWrapper(T &&object);
    ValueWrapper(const ValueWrapper &other);
    ~ValueWrapper();
    ValueWrapper &operator=(const ValueWrapper &other);
    Interface &value();
    const Interface &value() const;
    bool hasValue() const;
private:
    //...
};
```

Minimal implementation

```
template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T>
        requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
    explicit ValueWrapper(T &&object);
    ValueWrapper(const ValueWrapper &other);
    ~ValueWrapper();
    ValueWrapper &operator=(const ValueWrapper &other);
    Interface &value();
    const Interface &value() const;
    bool hasValue() const;
private:
    //...
};
```

Minimal implementation

```
template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T>
        requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
    explicit ValueWrapper(T &&object);
    ValueWrapper(const ValueWrapper &other);
    ~ValueWrapper();
    ValueWrapper &operator=(const ValueWrapper &other);
    Interface &value();
    const Interface &value() const;
    bool hasValue() const;
private:
    //...
};
```

Minimal implementation

```
template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T>
        requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
    explicit ValueWrapper(T &&object);
    ValueWrapper(const ValueWrapper &other);
    ~ValueWrapper();
    ValueWrapper &operator=(const ValueWrapper &other);
    Interface &value();
    const Interface &value() const;
    bool hasValue() const;
private:
    //...
};
```

Minimal implementation

```
template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T>
        requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
    explicit ValueWrapper(T &&object);
    ValueWrapper(const ValueWrapper &other);
    ~ValueWrapper();
    ValueWrapper &operator=(const ValueWrapper &other);
    Interface &value();
    const Interface &value() const;
    bool hasValue() const;
private:
    //...
};
```

Minimal implementation


```
template<typename Interface>
struct ValueWrapper {
    ValueWrapper() noexcept = default;
    template<typename T>
        requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
    explicit ValueWrapper(T &&object);
    ValueWrapper(const ValueWrapper &other);
    ~ValueWrapper();
    ValueWrapper &operator=(const ValueWrapper &other);
    Interface &value();
    const Interface &value() const;
    bool hasValue() const;
private:
    //...
};
```

Minimal implementation

```
template<typename Interface>
struct ValueWrapper {
    //...
private:
    static constexpr size_t smallCapacity = 24;
    //...
    union {
        std::byte small[smallCapacity] = {};
        void *big;
    };
    struct Methods;
    const Methods *methods = nullptr;
    //...
};
```

small buffer + pointer

pointer to function table

Minimal implementation

```
//...
private:
    static constexpr size_t smallCapacity = 24;

template<typename T>
static constexpr bool isSmall =
    sizeof(T) <= smallCapacity &&
    alignof(T) <= alignof(void*);

union {
    std::byte small[smallCapacity] = {};
    void *big;
};
//...
```

```
//...
private:
    static constexpr size_t smallCapacity = 24;

template<typename T>
static constexpr bool isSmall =
    sizeof(T) <= smallCapacity &&
    alignof(T) <= alignof(void*);

union {
    std::byte small[smallCapacity] = {};
    void *big;
};
//...
```

```
//...
private:
    static constexpr size_t smallCapacity = 24;

template<typename T>
static constexpr bool isSmall =
    sizeof(T) <= smallCapacity &&
    alignof(T) <= alignof(void*);

union {
    std::byte small[smallCapacity] = {};
    void *big;
};
//...
```

```
//...
private:
//...
struct Methods {
    void (*destroy)(ValueWrapper &data) noexcept;
    void (*copyConstruct)(ValueWrapper &dst, const ValueWrapper &src);
    const Interface &(*get)(const ValueWrapper &data) noexcept;
};
const Methods *methods = nullptr;

template<typename T>
static constexpr Methods getTableForSmallObject();

template<typename T>
static constexpr Methods getTableForBigObject();

template<typename T>
static constexpr Methods table =
    isSmall<T> ? getTableForSmallObject<T>() : getTableForBigObject<T>();
};
```

```
//...
private:
//...
struct Methods {
    void (*destroy)(ValueWrapper &data) noexcept;
    void (*copyConstruct)(ValueWrapper &dst, const ValueWrapper &src);
    const Interface &(*get)(const ValueWrapper &data) noexcept;
};
const Methods *methods = nullptr;

template<typename T>
static constexpr Methods getTableForSmallObject();

template<typename T>
static constexpr Methods getTableForBigObject();

template<typename T>
static constexpr Methods table =
    isSmall<T> ? getTableForSmallObject<T>() : getTableForBigObject<T>();
};
```

```
//...
private:
//...
struct Methods {
    void (*destroy)(ValueWrapper &data) noexcept;
    void (*copyConstruct)(ValueWrapper &dst, const ValueWrapper &src);
    const Interface &(*get)(const ValueWrapper &data) noexcept;
};
const Methods *methods = nullptr;

template<typename T>
static constexpr Methods getTableForSmallObject();

template<typename T>
static constexpr Methods getTableForBigObject();

template<typename T>
static constexpr Methods table =
    isSmall<T> ? getTableForSmallObject<T>() : getTableForBigObject<T>();
};
```



```
//...
private:
//...
struct Methods {
    void (*destroy)(ValueWrapper &data) noexcept;
    void (*copyConstruct)(ValueWrapper &dst, const ValueWrapper &src);
    const Interface &(*get)(const ValueWrapper &data) noexcept;
};
const Methods *methods = nullptr;

template<typename T>
static constexpr Methods getTableForSmallObject();

template<typename T>
static constexpr Methods getTableForBigObject();

template<typename T>
static constexpr Methods table =
    isSmall<T> ? getTableForSmallObject<T>() : getTableForBigObject<T>();
};
```

```
template<typename T>
static constexpr Methods getTableForSmallObject() {
    return {
        [](ValueWrapper &v) noexcept { //destroy
            reinterpret_cast<T*>(&v.small)->~T();
        },
        [](ValueWrapper &dst, const ValueWrapper &src) { //copyConstruct
            ::new(dst.small) T{*reinterpret_cast<const T*>(&src.small)};
        },
        [](const ValueWrapper &v) noexcept ->const Interface& { //get
            return *reinterpret_cast<const T*>(&v.small);
        }
    };
}
```

Minimal implementation

```
template<typename T>
static constexpr Methods getTableForSmallObject() {
    return {
        [](ValueWrapper &v) noexcept { //destroy
            reinterpret_cast<T*>(&v.small)->~T();
        },
        [](ValueWrapper &dst, const ValueWrapper &src) { //copyConstruct
            ::new(dst.small) T{*reinterpret_cast<const T*>(&src.small)};
        },
        [](const ValueWrapper &v) noexcept ->const Interface& { //get
            return *reinterpret_cast<const T*>(&v.small);
        }
    };
}
```

Minimal implementation

```
template<typename T>
static constexpr Methods getTableForSmallObject() {
    return {
        [](ValueWrapper &v) noexcept { //destroy
            reinterpret_cast<T*>(&v.small)->~T();
        },
        [](ValueWrapper &dst, const ValueWrapper &src) { //copyConstruct
            ::new(dst.small) T{*reinterpret_cast<const T*>(&src.small)};
        },
        [](const ValueWrapper &v) noexcept ->const Interface& { //get
            return *reinterpret_cast<const T*>(&v.small);
        }
    };
}
```

Minimal implementation

```
template<typename T>
static constexpr Methods getTableForSmallObject() {
    return {
        [](ValueWrapper &v) noexcept { //destroy
            reinterpret_cast<T*>(&v.small)->~T();
        },
        [](ValueWrapper &dst, const ValueWrapper &src) { //copyConstruct
            ::new(dst.small) T{*reinterpret_cast<const T*>(&src.small)};
        },
        [](const ValueWrapper &v) noexcept ->const Interface& { //get
            return *reinterpret_cast<const T*>(&v.small);
        }
    };
}
```

Minimal implementation

```

template<typename T>
static constexpr Methods getTableForSmallObject() {
    return {
        [](ValueWrapper &v) noexcept { //destroy
            reinterpret_cast<T*>(&v.small)->~T();
        },
        [](ValueWrapper &dst, const ValueWrapper &src) { //copy construct
            ::new(dst.small) T{*reinterpret_cast<const T*>(&src.small)};
        },
        [](const ValueWrapper &v) noexcept ->const Interface& { //get
            return *reinterpret_cast<const T*>(&v.small);
        }
    };
}

```

we must std::launder() pointers

The diagram consists of a white rectangular box with a thin black border containing the text "we must std::launder() pointers". From the bottom-left corner of this box, a blue arrow points to the first `reinterpret_cast` call in the first lambda function. From the bottom-right corner of the box, another blue arrow points to the `reinterpret_cast` call in the second lambda function. A third blue arrow points from the top-left corner of the box to the `reinterpret_cast` call in the third lambda function.

Minimal implementation

```
template<typename T>
static constexpr Methods getTableForBigObject() {
    return {
        [](ValueWrapper &v) noexcept { //destroy
            delete static_cast<T*>(v.big);
        },
        [](ValueWrapper &dst, const ValueWrapper &src) { //copyConstruct
            dst.big = new T{ *static_cast<const T*>(src.big) };
        },
        [](const ValueWrapper &v) noexcept ->const Interface& { //get
            return *static_cast<const T*>(v.big);
        }
    };
}
```

Minimal implementation

```
template<typename T>
static constexpr Methods getTableForBigObject() {
    return {
        [](ValueWrapper &v) noexcept { //destroy
            delete static_cast<T*>(v.big);
        },
        [](ValueWrapper &dst, const ValueWrapper &src) { //copyConstruct
            dst.big = new T{ *static_cast<const T*>(src.big) };
        },
        [](const ValueWrapper &v) noexcept ->const Interface& { //get
            return *static_cast<const T*>(v.big);
        }
    };
}
```

Minimal implementation


```
template<typename T>
static constexpr Methods getTableForBigObject() {
    return {
        [](ValueWrapper &v) noexcept { //destroy
            delete static_cast<T*>(v.big);
        },
        [](ValueWrapper &dst, const ValueWrapper &src) { //copyConstruct
            dst.big = new T{ *static_cast<const T*>(src.big) };
        },
        [](const ValueWrapper &v) noexcept ->const Interface& { //get
            return *static_cast<const T*>(v.big);
        }
    };
}
```

Minimal implementation

```
template<typename T>
static constexpr Methods getTableForBigObject() {
    return {
        [](ValueWrapper &v) noexcept { //destroy
            delete static_cast<T*>(v.big);
        },
        [](ValueWrapper &dst, const ValueWrapper &src) { //copyConstruct
            dst.big = new T{ *static_cast<const T*>(src.big) };
        },
        [](const ValueWrapper &v) noexcept ->const Interface& { //get
            return *static_cast<const T*>(v.big);
        }
    };
}
```

Minimal implementation

```
ValueWrapper() noexcept = default;
```

```
template<typename T>
```

```
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
```

```
explicit ValueWrapper(T &&object) :
```

```
    methods{ &table<std::remove_cvref_t_t<T>> }
```

```
{
```

```
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
```

```
        ::new(small) std::remove_cvref_t<T>{ std::forward<T>(object) };
```

```
    }
```

```
    else {
```

```
        big = new std::remove_cvref_t<T>{ std::forward<T>(object) };
```

```
    }
```

```
}
```

Minimal implementation

```
ValueWrapper() noexcept = default;
```

```
template<typename T>
```

```
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
```

```
explicit ValueWrapper(T &&object) :
```

```
    methods{ &table<std::remove_cvref_t_t<T>> }  
{
```

```
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
```

```
        ::new(small) std::remove_cvref_t<T>{ std::forward<T>(object) };
```

```
    }
```

```
    else {
```

```
        big = new std::remove_cvref_t<T>{ std::forward<T>(object) };
```

```
    }
```

```
}
```

Minimal implementation

```
ValueWrapper() noexcept = default;
```

```
template<typename T>
```

```
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
```

```
explicit ValueWrapper(T &&object) :
```

```
    methods{ &table<std::remove_cvref_t_t<T>> }
```

```
{
```

```
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
```

```
        ::new(small) std::remove_cvref_t<T>{ std::forward<T>(object) };
```

```
    }
```

```
    else {
```

```
        big = new std::remove_cvref_t<T>{ std::forward<T>(object) };
```

```
    }
```

```
}
```

Minimal implementation

```
ValueWrapper() noexcept = default;
```

```
template<typename T>
```

```
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
```

```
explicit ValueWrapper(T &&object) :
```

```
    methods{ &table<std::remove_cvref_t_t<T>> }  
{
```

```
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
```

```
        ::new(small) std::remove_cvref_t<T>{ std::forward<T>(object) };
```

```
    }
```

```
    else {
```

```
        big = new std::remove_cvref_t<T>{ std::forward<T>(object) };
```

```
    }
```

```
}
```

Minimal implementation

```
ValueWrapper() noexcept = default;
```

```
template<typename T>
```

```
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
```

```
explicit ValueWrapper(T &&object) :
```

```
    methods{ &table<std::remove_cvref_t_t<T>> }  
{
```

```
{
```

```
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
```

```
        ::new(small) std::remove_cvref_t<T>{ std::forward<T>(object) };
```

```
    }
```

```
    else {
```

```
        big = new std::remove_cvref_t<T>{ std::forward<T>(object) };
```

```
    }
```

```
}
```

Minimal implementation

```
ValueWrapper() noexcept = default;
```

```
template<typename T>
```

```
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
```

```
explicit ValueWrapper(T &&object) :
```

```
    methods{ &table<std::remove_cvref_t_t<T>> }  
{
```

```
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
```

```
        ::new(small) std::remove_cvref_t<T>{ std::forward<T>(object) };  
    }
```

```
    else {
```

```
        big = new std::remove_cvref_t<T>{ std::forward<T>(object) };  
    }
```

```
}
```

Minimal implementation


```
ValueWrapper(const ValueWrapper &other) :  
    methods{ other.methods }  
{  
    if (methods)  
        methods->copyConstruct(*this, other);  
}
```

```
~ValueWrapper() {  
    if (hasValue())  
        methods->destroy(*this);  
}
```

Minimal implementation

```
ValueWrapper(const ValueWrapper &other) :  
    methods{ other.methods }  
{  
    if (methods)  
        methods->copyConstruct(*this, other);  
}
```

```
~ValueWrapper() {  
    if (hasValue())  
        methods->destroy(*this);  
}
```

Minimal implementation

```
ValueWrapper &operator=(const ValueWrapper &other) {  
    if (&other == this)  
        return *this;  
  
    if (hasValue()) {  
        methods->destroy(*this);  
        methods = nullptr;  
    }  
    if (other.methods) {  
        other.methods->copyConstruct(*this, other);  
        methods = other.methods;  
    }  
    return *this;  
}
```

needed to support self assignment

needed for weak exception guarantee

the order of assignment matters here

Minimal implementation

```
ValueWrapper &operator=(const ValueWrapper &other) {  
    if (&other == this)  
        return *this;  
  
    if (hasValue()) {  
        methods->destroy(*this);  
        methods = nullptr;  
    }  
    if (other.methods) {  
        other.methods->copyConstruct(*this, other);  
        methods = other.methods;  
    }  
    return *this;  
}
```

needed to support self assignment

needed for weak exception guarantee

the order of assignment matters here

Minimal implementation

```
ValueWrapper &operator=(const ValueWrapper &other) {  
    if (&other == this)  
        return *this;  
  
    if (hasValue()) {  
        methods->destroy(*this);  
        methods = nullptr;  
    }  
    if (other.methods) {  
        other.methods->copyConstruct(*this, other);  
        methods = other.methods;  
    }  
    return *this;  
}
```

needed to support self assignment

needed for weak exception guarantee

the order of assignment matters here

Minimal implementation

```
Interface &value() {  
    return const_cast<Interface&>(methods->get(*this));  
}  
const Interface &value() const {  
    return methods->get(*this);  
}  
  
bool hasValue() const {  
    return methods != nullptr;  
}
```

Minimal implementation

```
Interface &value() {  
    return const_cast<Interface&>(methods->get(*this));  
}  
  
const Interface &value() const {  
    return methods->get(*this);  
}  
  
bool hasValue() const {  
    return methods != nullptr;  
}
```

Minimal implementation

```
Interface &value() {  
    return const_cast<Interface&>(methods->get(*this));  
}  
const Interface &value() const {  
    return methods->get(*this);  
}  
  
bool hasValue() const {  
    return methods != nullptr;  
}
```

Minimal implementation


```
Interface &value() {  
    return const_cast<Interface&>(methods->get(*this));  
}  
const Interface &value() const {  
    return methods->get(*this);  
}  
  
bool hasValue() const {  
    return methods != nullptr;  
}
```

Minimal implementation

```
struct Impl : Iface {
    Impl() { std::cout << "Impl::ctor\n"; }
    Impl(const Impl &) { std::cout << "Impl::copy ctor\n"; }
    Impl(Impl &&) noexcept { std::cout << "Impl::move ctor\n"; }
    ~Impl() { std::cout << "Impl::dtor\n"; }
    Impl &operator=(const Impl &) { std::cout << "Impl::copy =\n"; return *this; }
    Impl &operator=(Impl &&) noexcept { std::cout << "Impl::move =\n"; return *this; }

    void foo() override {
        std::cout << "Impl::foo(): " << data << '\n';
    }

    char data[4] = "abc";
};
```

Minimal implementation

```

auto v = ValueWrapper<Iface>{ Impl{} };
Impl::ctor
Impl::move ctor
Impl::dtor

```

```

auto x = v;
Impl::copy ctor

```

```

auto y = std::move(v);
Impl::copy ctor

```

```

x = y;
Impl::dtor
Impl::copy ctor

```

```

x = std::move(y);
Impl::dtor
Impl::copy ctor

```

```

x->foo();
Impl::foo(): abc

```

```

Impl::dtor
Impl::dtor
Impl::dtor

```

<code>auto v = ValueWrapper<Iface>{ Impl{} };</code>	<code>Impl::ctor</code>
	<code>Impl::move ctor</code>
	<code>Impl::dtor</code>
<code>auto x = v;</code>	<code>Impl::copy ctor</code>
<code>auto y = std::move(v);</code>	<u><code>Impl::copy ctor</code></u>
<code>x = y;</code>	<code>Impl::dtor</code>
	<code>Impl::copy ctor</code>
<code>x = std::move(y);</code>	<code>Impl::dtor</code>
	<code>Impl::copy ctor</code>
<code>x->foo();</code>	<code>Impl::foo(): abc</code>
	<code>Impl::dtor</code>
	<code>Impl::dtor</code>
	<code>Impl::dtor</code>

```
auto v = ValueWrapper<Iface>{ Impl{} };
```

Impl::ctor
Impl::move ctor
Impl::dtor

```
auto x = v;
```

Impl::copy ctor

```
auto y = std::move(v);
```

Impl::copy ctor

```
x = y;
```

Impl::dtor
Impl::copy ctor

```
x = std::move(y);
```

Impl::dtor
Impl::copy ctor

```
x->foo();
```

Impl::foo(): abc

Impl::dtor
Impl::dtor
Impl::dtor

```
const Interface &value() const {  
    return methods->get(*this);  
}
```



meh...

Let's save pointer to interface

```
private:
    //...
    union {
        std::byte small[smallCapacity] = {};
        void *big;
    };

    struct Methods { /*...*/ };
    const Methods *methods = nullptr;

    Interface *interface = nullptr;
    //
```

Let's save pointer to interface

```
Interface &value() {  
    return *interface;  
}  
const Interface &value() const {  
    return *interface;  
}
```

Let's save pointer to interface


```
template<typename T>
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
explicit ValueWrapper(T &&object) :
    methods{ &table<std::remove_cvref_t<T>> }
{
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
        interface =
            ::new(small) std::remove_cvref_t<T>{ std::forward<T>(object) };
    }
    else {
        auto *instance = new std::remove_cvref_t<T>{ std::forward<T>(object) };
        big = instance;
        interface = instance;
    }
}
```

Let's save pointer to interface

```
template<typename T>
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
explicit ValueWrapper(T &&object) :
    methods{ &table<std::remove_cvref_t<T>> }
{
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
        interface =
            ::new(small) std::remove_cvref_t<T>{ std::forward<T>(object) };
    }
    else {
        auto *instance = new std::remove_cvref_t<T>{ std::forward<T>(object) };
        big = instance;
        interface = instance;
    }
}
```

Let's save pointer to interface

```
template<typename T>
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
explicit ValueWrapper(T &&object) :
    methods{ &table<std::remove_cvref_t<T>> }
{
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
        interface =
            ::new(small) std::remove_cvref_t<T>{ std::forward<T>(object) };
    }
    else {
        auto *instance = new std::remove_cvref_t<T>{ std::forward<T>(object) };
        big = instance;
        interface = instance;
    }
}
```

Let's save pointer to interface

```
struct Methods {  
    void (*destroy)(ValueWrapper &data) noexcept;  
    void (*copyConstruct)(ValueWrapper &dst, const ValueWrapper &src);  
};
```



removed get

Let's save pointer to interface

```
template<typename T>
static constexpr Methods getTableForSmallObject() {
    return {
        [](ValueWrapper &v) noexcept { //destroy
            reinterpret_cast<T*>(&v.small)->~T();
        },
        [](ValueWrapper &dst, const ValueWrapper &src) { //copyConstruct
            dst.interface =
                ::new(dst.small) T{*reinterpret_cast<const T*>(&src.small)};
        }
    };
}
```

Let's save pointer to interface

```
template<typename T>
static constexpr Methods getTableForBigObject() {
    return {
        [](ValueWrapper &v) noexcept { //destroy
            delete static_cast<T*>(v.big);
        },
        [](ValueWrapper &dst, const ValueWrapper &src) { //copyConstruct
            auto *instance = new T{ *static_cast<const T*>(src.big) };
            dst.big = instance;
            dst.interface = instance;
        }
    };
}
```

Let's save pointer to interface

MSVC for x64

```
union {  
    unsigned char _TrivialData[56];  
    struct {  
        union {  
            unsigned char _Data[48];  
            void* _Ptr;  
        };  
        const _Any___RTTI* _RTTI;  
    };  
};  
uintptr_t _TypeData;
```

small buffer for trivial types

small buffer or pointer for other types

pointer to function table

pointer to type info packed with type category flag in the lower 2 bits

Comparison to `std::any`

MSVC for x64

```
union {  
    unsigned char _TrivialData[56];  
    struct {  
        union {  
            unsigned char _Data[48];  
            void* _Ptr;  
        };  
        const _Any___RTTI* _RTTI;  
    };  
};  
uintptr_t _TypeData;
```

small buffer for trivial types

small buffer or pointer for other types

pointer to function table

pointer to type info packed with type category flag in the lower 2 bits

Comparison to `std::any`

MSVC for x64

```

union {
    unsigned char _TrivialData[56];
    struct {
        union {
            unsigned char _Data[48];
            void* _Ptr;
        };
        const _Any__RTTI* _RTTI;
    };
};
uintptr_t _TypeData;

```

small buffer for trivial types

small buffer or pointer for other types

pointer to function table

pointer to type info packed with type category flag in the lower 2 bits

Comparison to `std::any`

libstdc++

```
union _Arg {  
    void *_M_obj;  
    const std::type_info *_M_typeinfo;  
    any *_M_any;  
};  
void (*_M_manager)(_Op, const any*, _Arg*);  
union {  
    void *_M_ptr;  
    aligned_storage<sizeof(_M_ptr), alignof(void*)>::type _M_buffer;  
};
```

pointer to "do-all" function

small buffer or pointer

Comparison to `std::any`

libstdc++

```
union _Arg {  
    void *_M_obj;  
    const std::type_info *_M_typeinfo;  
    any *_M_any;  
};  
void (*_M_manager)(_Op, const any*, _Arg*);  
union {  
    void *_M_ptr;  
    aligned_storage<sizeof(_M_ptr), alignof(void*)>::type _M_buffer;  
};
```

pointer to "do-all" function

small buffer or pointer

Comparison to `std::any`

libc++

```
void *(*__h)(_Action,
             any const*,
             any*,
             const type_info*,
             const void *__fallback_info);

union {
    void *__ptr;
    aligned_storage_t<3 * sizeof(void*),
                    alignment_of<void*>::value> __buf;
};
```

pointer to "do-all" function

small buffer or pointer

Comparison to `std::any`

libc++

```
void *(*__h)(_Action,
             any const*,
             any*,
             const type_info*,
             const void *__fallback_info);

union {
    void *__ptr;
    aligned_storage_t<3 * sizeof(void*),
                    alignment_of<void*>::value> __buf;
};
```

pointer to "do-all" function

small buffer or pointer

Comparison to `std::any`

ValueWrapper

```
union {  
    std::byte small[smallCapacity];  
    void *big;  
};
```

small buffer or pointer



```
const Methods *methods;
```

pointer to function table



```
Interface *interface;
```

pointer to interface



Comparison to `std::any`

ValueWrapper

```
union {  
    std::byte small[smallCapacity];  
    void *big;  
};
```

small buffer or pointer



```
const Methods *methods;
```

pointer to function table



```
Interface *interface;
```

pointer to interface



Comparison to `std::any`

ValueWrapper

```
union {  
    std::byte small[smallCapacity];  
    void *big;  
};
```

small buffer or pointer



```
const Methods *methods;
```

pointer to function table



```
Interface *interface;
```

pointer to interface



Comparison to `std::any`


```
auto v = ValueWrapper<Iface>{ Impl{} };
```

Impl::ctor
Impl::move ctor
Impl::dtor

```
auto x = v;
```

Impl::copy ctor

```
auto y = std::move(v);
```

Impl::copy ctor

```
x = y;
```

Impl::dtor
Impl::copy ctor

```
x = std::move(y);
```

Impl::dtor
Impl::copy ctor

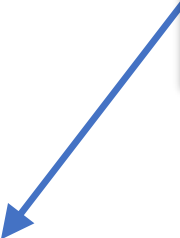
```
x->foo();
```

Impl::foo(): abc

Impl::dtor
Impl::dtor
Impl::dtor

```
template<typename T>
static constexpr bool isSmall =
    sizeof(T) <= smallCapacity &&
    alignof(T) <= alignof(void*) &&
    std::is_nothrow_move_constructible_v<T>;
```

support noexcept move
for small objects



```
struct Methods {
    void (*destroy)(ValueWrapper &data) noexcept;
    void (*copyConstruct)(ValueWrapper &dst, const ValueWrapper &src);
    void (*moveConstruct)(ValueWrapper &dst, ValueWrapper &src) noexcept;
};
```

Let's move!

```
template<typename T>
static constexpr Methods getTableForSmallObject() {
    return {
        [](ValueWrapper &v) noexcept { //destroy
            reinterpret_cast<T*>(&v.small)->~T();
        },
        [](ValueWrapper &dst, const ValueWrapper &src) { //copyConstruct
            dst.interface =
                ::new(dst.small) T{*reinterpret_cast<const T*>(&src.small)};
        },
        [](ValueWrapper &dst, ValueWrapper &src) noexcept { //moveConstruct
            dst.interface =
                ::new (dst.small) T{std::move(*reinterpret_cast<T*>(&src.small))};
        }
    };
}
```

```
template<typename T>
static constexpr Methods getTableForBigObject() {
    return {
        [](ValueWrapper &v) noexcept { //destroy
            delete static_cast<T*>(v.big);
        },
        [](ValueWrapper &dst, const ValueWrapper &src) { //copyConstruct
            auto *instance = new T{ *static_cast<const T*>(src.big) };
            dst.big = instance;
            dst.interface = instance;
        },
        [](ValueWrapper &dst, ValueWrapper &src) noexcept { //moveConstruct
            dst.big = std::exchange(src.big, nullptr);
            dst.interface = std::exchange(src.interface, nullptr);
            src.methods = nullptr;
        }
    };
}
```

```
ValueWrapper(ValueWrapper &&other) noexcept : methods{ other.methods } {  
    if (methods)  
        methods->moveConstruct(*this, other);  
}
```

```
ValueWrapper &operator=(ValueWrapper &&other) noexcept {  
    if (&other == this)  
        return *this;  
  
    if (hasValue())  
        methods->destroy(*this);  
  
    methods = other.methods;  
    if (methods)  
        methods->moveConstruct(*this, other);  
  
    return *this;  
}
```

```
ValueWrapper(ValueWrapper &&other) noexcept : methods{ other.methods } {  
    if (methods)  
        methods->moveConstruct(*this, other);  
}
```

```
ValueWrapper &operator=(ValueWrapper &&other) noexcept {  
    if (&other == this)  
        return *this;  
  
    if (hasValue())  
        methods->destroy(*this);  
  
    methods = other.methods;  
    if (methods)  
        methods->moveConstruct(*this, other);  
  
    return *this;  
}
```

```
ValueWrapper(ValueWrapper &&other) noexcept : methods{ other.methods } {  
    if (methods)  
        methods->moveConstruct(*this, other);  
}
```

```
ValueWrapper &operator=(ValueWrapper &&other) noexcept {  
    if (&other == this)  
        return *this;  
  
    if (hasValue())  
        methods->destroy(*this);  
  
    methods = other.methods;  
    if (methods)  
        methods->moveConstruct(*this, other);  
  
    return *this;  
}
```

```
ValueWrapper(ValueWrapper &&other) noexcept : methods{ other.methods } {  
    if (methods)  
        methods->moveConstruct(*this, other);  
}
```

```
ValueWrapper &operator=(ValueWrapper &&other) noexcept {  
    if (&other == this)  
        return *this;  
  
    if (hasValue())  
        methods->destroy(*this);  
  
    methods = other.methods;  
    if (methods)  
        methods->moveConstruct(*this, other);  
  
    return *this;  
}
```



```
ValueWrapper(ValueWrapper &&other) noexcept : methods{ other.methods } {  
    if (methods)  
        methods->moveConstruct(*this, other);  
}
```

```
ValueWrapper &operator=(ValueWrapper &&other) noexcept {  
    if (&other == this)  
        return *this;  
  
    if (hasValue())  
        methods->destroy(*this);  
  
    methods = other.methods;  
    if (methods)  
        methods->moveConstruct(*this, other);  
  
    return *this;  
}
```

```

auto v = ValueWrapper<Iface>{ Impl{} };
Impl::ctor
Impl::move ctor
Impl::dtor

```

```

auto x = v;
Impl::copy ctor

```

```

auto y = std::move(v);
Impl::move ctor

```

```

x = y;
Impl::dtor
Impl::copy ctor

```

```

x = std::move(y);
Impl::dtor
Impl::move ctor

```

```

x->foo();
Impl::foo(): abc

```

```

Impl::dtor
Impl::dtor
Impl::dtor

```

Small object move construction:

```
auto y = std::move(v);
```

our `ValueWrapper<Iface>` and
MSVC `std::any`

just moves

```
Impl::move ctor
```

libstdc++ and **libc++** `std::any`

moves and destroys the initial object

```
Impl::move ctor
```

```
Impl::dtor
```

Differences to `std::any`

Small object move assignment:

```
x = std::move(y);
```

`ValueWrapper<Iface>` and
MSVC `std::any`

destroys this object and
moves

```
Impl::dtor  
Impl::move ctor
```

Differences to `std::any`

libstdc++ `std::any`

destroys this object,
moves, and destroys
the initial object

```
Impl::dtor  
Impl::move ctor  
Impl::dtor
```

libc++ `std::any`

destroys this object,
moves, and destroys
the initial object,
but in a funny way

```
Impl::move ctor  
Impl::dtor  
Impl::move ctor  
Impl::dtor  
Impl::move ctor  
Impl::dtor  
Impl::move ctor
```

Small object move assignment:

```
x = std::move(y);
```

`ValueWrapper<Iface>` and
MSVC `std::any`

destroys this object and
moves

libstdc++ `std::any`

destroys this object,
moves, and destroys
the initial object

libc++ `std::any`

destroys this object,
moves, and destroys
the initial object,
but in a funny way



```
Impl::move ctor  
Impl::dtor  
Impl::move ctor  
Impl::dtor  
Impl::move ctor  
Impl::dtor  
Impl::move ctor
```

Small object move assignment

```
x = std::move(y);
```

ValueWrapper<Iface> and
MSVC std::any

destroys this object and
moves

```
Impl::dtor  
Impl::move ctor
```

```
any & operator=(any && __rhs) _NOEXCEPT {  
    any(_VSTD::move(__rhs)).swap(*this);  
    return *this;  
}
```

```
lil Impl::move ctor  
de Impl::dtor  
m Impl::move ctor  
th Impl::dtor  
Impl::move ctor  
Impl::dtor  
I Impl::move ctor  
I Impl::dtor  
I Impl::dtor
```

Differences to std::any

```
Impl::dtor  
Impl::move ctor  
Impl::dtor  
Impl::move ctor
```

Small object copy assignment:

```
x = y;
```

	std::any	
ValueWrapper<Iface>	MSVC and libstdc++	libc++
destroys this object and copies	constructs a copy, then moves into this object	constructs a copy, then moves into this object, but in a funny way
Impl::dtor Impl::copy ctor	Impl::copy ctor Impl::dtor Impl::move ctor Impl::dtor	Impl::copy ctor Impl::move ctor Impl::dtor Impl::move ctor Impl::dtor Impl::move ctor Impl::dtor
Differences to std::any		

Small object copy assignment:

```
x = y;
```

needed for
strong exception guarantee

`std::any`

`ValueWrapper<Iface>`

MSVC and libstdc++

libc++

destroys this object and
copies

constructs a copy, then
moves into this object

constructs a copy, then
moves into this object,
but in a funny way

```
Impl::~dtor  
Impl::copy ctor
```

```
Impl::copy ctor  
Impl::~dtor  
Impl::move ctor  
Impl::~dtor
```

```
Impl::copy ctor  
Impl::move ctor  
Impl::~dtor  
Impl::move ctor  
Impl::~dtor  
Impl::move ctor  
Impl::~dtor
```

Differences to `std::any`

Small object copy assignment:

```
x = y;
```

```
any & operator=(any const & __rhs) {  
    any(__rhs).swap(*this);  
    return *this;  
}
```

`ValueWrapper<Iface>`

destroys this object and
copies

`Impl::dctor`

`Impl::copy ctor`

MSVC and

constructs
moves into

`Impl::copy`

`Impl::dctor`

`Impl::move`

`Impl::dctor`

`Impl::copy ctor`

`Impl::move ctor`

`Impl::dctor`

`Impl::move ctor`

`Impl::dctor`

`Impl::move ctor`

`Impl::dctor`

`Impl::dctor`

`Impl::move ctor`

`Impl::dctor`

`Impl::move ctor`

`Impl::dctor`

Differences to `std::any`

```

auto v = ValueWrapper<Iface>{ Impl{} };
Impl::ctor
Impl::move ctor
Impl::dtor

```

```

auto x = v;
Impl::copy ctor

```

```

auto y = std::move(v);
Impl::move ctor

```

```

x = y;
Impl::dtor
Impl::copy ctor

```

```

x = std::move(y);
Impl::dtor
Impl::move ctor

```

```

x->foo();
Impl::foo(): abc

```

```

Impl::dtor
Impl::dtor
Impl::dtor

```

```
struct Methods {  
    void (*destroy)(ValueWrapper &data) noexcept;  
    void (*copyConstruct)(ValueWrapper &dst, const ValueWrapper &src);  
    void (*moveConstruct)(ValueWrapper &dst, ValueWrapper &src) noexcept;  
    void (*assign)(ValueWrapper &dst, const ValueWrapper &src);  
    void (*moveAssign)(ValueWrapper &dst, ValueWrapper &src) noexcept;  
};
```

Assignment support (see bonus slides)

- `emplace` (see bonus slides)
- `value assignment` (see bonus slides)
- enforce everywhere that the value type must be copy-constructible
 - to be modeled after interface of `std::any`
 - add to all functions where value type is concerned:
`requires (std::is_copy_constructible_v<std::remove_cvref_t<T>>)`
- in-place construction?
- `reset`?
`x = {};`
- `swap`?
- experiment with function tables vs. "do-all" functions?
 - performance?
 - binary size?

What else?

- `emplace` (see bonus slides)
- value assignment (see bonus slides)
- enforce everywhere that the value type must be copy-constructible
 - to be modeled after interface of `std::any`
 - add to all functions where value type is concerned:
`requires (std::is_copy_constructible_v<std::remove_cvref_t<T>>)`
- in-place construction?
- reset?
`x = {};`
- swap?
- experiment with function tables vs. "do-all" functions?
 - performance?
 - binary size?

What else?

- `emplace` (see bonus slides)
- value assignment (see bonus slides)
- enforce everywhere that the value type must be copy-constructible
 - to be modeled after interface of `std::any`
 - add to all functions where value type is concerned:
`requires (std::is_copy_constructible_v<std::remove_cvref_t<T>>)`
- in-place construction?
- reset?
`x = {};`
- swap?
- experiment with function tables vs. "do-all" functions?
 - performance?
 - binary size?

What else?

- `emplace` (see bonus slides)
- value assignment (see bonus slides)
- enforce everywhere that the value type must be copy-constructible
 - to be modeled after interface of `std::any`
 - add to all functions where value type is concerned:
`requires (std::is_copy_constructible_v<std::remove_cvref_t<T>>)`
- in-place construction?
- reset?
`x = {};`
- swap?
- experiment with function tables vs. "do-all" functions?
 - performance?
 - binary size?

What else?

- `emplace` (see bonus slides)
- value assignment (see bonus slides)
- enforce everywhere that the value type must be copy-constructible
 - to be modeled after interface of `std::any`
 - add to all functions where value type is concerned:
`requires (std::is_copy_constructible_v<std::remove_cvref_t<T>>)`
- in-place construction?
- reset?
`x = {};`
- swap?
- experiment with function tables vs. "do-all" functions?
 - performance?
 - binary size?

What else?

- `emplace` (see bonus slides)
- value assignment (see bonus slides)
- enforce everywhere that the value type must be copy-constructible
 - to be modeled after interface of `std::any`
 - add to all functions where value type is concerned:
`requires (std::is_copy_constructible_v<std::remove_cvref_t<T>>)`
- in-place construction?
- `reset`?
`x = {};`
- `swap`?
- experiment with function tables vs. "do-all" functions?
 - performance?
 - binary size?

What else?

P3019: indirect and polymorphic: Vocabulary Types for Composite Class Design

- `std::indirect` — value wrapper for non-polymorphic types
(*kinda* similar to `std::unique_ptr`)
- `std::polymorphic` — value wrapper for polymorphic types
similar to our `ValueWrapper`

C++26?

https://www.youtube.com/watch?v=u4_JAn83FJQ



online

Павел Новиков

Align Technology

Веселье с type erasure +
раскладываем данные из
абстрактной структуры в
параметры функции



See also

Thanks for listening!



Fun with type erasure: implementing a value wrapper for polymorphic types

Павел Новиков

X @crr_are

Slides: bit.ly/4kWGfXF



Bonus slides

Assignment support

```
struct Methods {  
    void (*destroy)(ValueWrapper &data) noexcept;  
    void (*copyConstruct)(ValueWrapper &dst, const ValueWrapper &src);  
    void (*moveConstruct)(ValueWrapper &dst, ValueWrapper &src) noexcept;  
    void (*assign)(ValueWrapper &dst, const ValueWrapper &src);  
    void (*moveAssign)(ValueWrapper &dst, ValueWrapper &src) noexcept;  
};
```

```

template<typename T>
static constexpr Methods getTableForSmallObject() {
    return {
        [](ValueWrapper &v) noexcept { /*destroy ...*/},
        [](ValueWrapper &dst, const ValueWrapper &src) { /*copyConstruct ...*/},
        [](ValueWrapper &dst, ValueWrapper &src) noexcept { /*moveConstruct ...*/},
        [](ValueWrapper &dst, const ValueWrapper &src) { //assign
            if constexpr (std::is_copy_assignable_v<T>) {
                *reinterpret_cast<T*>(&dst.small) =
                    *reinterpret_cast<const T*>(&src.small);
            }
            else {
                reinterpret_cast<T*>(&dst.small)->~T(); // destroy
                dst.interface =
                    ::new(dst.small) T{*reinterpret_cast<const T*>(&src.small)}; // copy construct
            }
        },
        [](ValueWrapper &dst, ValueWrapper &src) noexcept { //moveAssign
            if constexpr (std::is_nothrow_move_assignable_v<T>) {
                *reinterpret_cast<T*>(&dst.small) =
                    std::move(*reinterpret_cast<T*>(&src.small));
            }
            else {
                reinterpret_cast<T*>(&dst.small)->~T(); // destroy
                dst.interface =
                    ::new (dst.small) T{std::move(*reinterpret_cast<T*>(&src.small))}; // move construct
            }
        }
    };
}

```



```
static constexpr Methods getTableForSmallObject() {
```

```

[](ValueWrapper &dst, const ValueWrapper &src) { //assign
    if constexpr (std::is_copy_assignable_v<T>) {
        *reinterpret_cast<T*>(&dst.small) =
            *reinterpret_cast<const T*>(&src.small);
    }
    else {
        reinterpret_cast<T*>(&dst.small)->~T(); // destroy
        dst.interface =
            // copy construct
            ::new(dst.small) T{*reinterpret_cast<const T*>(&src.small)};
    }
},

```

```

else {
    reinterpret_cast<T*>(&dst.small)->~T(); // destroy
    dst.interface =
        ::new (dst.small) T{std::move(*reinterpret_cast<T*>(&src.small))}; // move construct
    }
}
};
}

```

```
static constexpr Methods getTableForSmallObject() {
```

```

[](ValueWrapper &dst, const ValueWrapper &src) { // assign
    if constexpr (std::is_copy_assignable_v<T>) {
        *reinterpret_cast<T*>(&dst.small) =
            *reinterpret_cast<const T*>(&src.small);
    }
    else {
        reinterpret_cast<T*>(&dst.small)->~T(); // destroy
        dst.interface =
            // copy construct
            ::new(dst.small) T{*reinterpret_cast<const T*>(&src.small)};
    }
},

```

```

else {
    reinterpret_cast<T*>(&dst.small)->~T(); // destroy
    dst.interface =
        ::new (dst.small) T{std::move(*reinterpret_cast<T*>(&src.small))}; // move construct
    }
}
};
}

```

```
static constexpr Methods getTableForSmallObject() {
```

```

[] (ValueWrapper &dst, const ValueWrapper &src) { // assign
    if constexpr (std::is_copy_assignable_v<T>) {
        *reinterpret_cast<T*>(&dst.small) =
            *reinterpret_cast<const T*>(&src.small);
    }
    else {
        reinterpret_cast<T*>(&dst.small)->~T(); // destroy
        dst.interface =
            // copy construct
            ::new(dst.small) T{*reinterpret_cast<const T*>(&src.small)};
    }
},

```

```

else {
    reinterpret_cast<T*>(&dst.small)->~T(); // destroy
    dst.interface =
        ::new (dst.small) T{std::move(*reinterpret_cast<T*>(&src.small))}; // move construct
    }
}
};
}

```

```
static constexpr Methods getTableForSmallObject() {
```

```
    [](ValueWrapper &v) noexcept { /*destroy ...*/ },
    [](ValueWrapper &dst, const ValueWrapper &src) { /*copyConstruct ...*/ },
    [](ValueWrapper &dst, ValueWrapper &src) noexcept { /*moveConstruct ...*/ },
    [](ValueWrapper &dst, const ValueWrapper &src) { //assign
        if constexpr (std::is_copy_assignable_v<T>) {
            *reinterpret_cast<T*>(&dst.small) =
```

```

    [](ValueWrapper &dst, ValueWrapper &src) noexcept { //moveAssign
        if constexpr (std::is_nothrow_move_assignable_v<T>) {
            *reinterpret_cast<T*>(&dst.small) =
                std::move(*reinterpret_cast<T*>(&src.small));
        }
        else {
            reinterpret_cast<T*>(&dst.small)->~T(); // destroy
            dst.interface =
                // move construct
                ::new (dst.small) T{std::move(*reinterpret_cast<T*>(&src.small))};
        }
    }
}

```

};

}

static constexpr Methods getTableForSmallObject() {

```

[](ValueWrapper &v) noexcept { /*destroy ...*/},
[](ValueWrapper &dst, const ValueWrapper &src) { /*copyConstruct ...*/},
[](ValueWrapper &dst, ValueWrapper &src) noexcept { /*moveConstruct ...*/},
[](ValueWrapper &dst, const ValueWrapper &src) { //assign
    if constexpr (std::is_copy_assignable_v<T>) {
        *reinterpret_cast<T*>(&dst.small) =

```

```

[](ValueWrapper &dst, ValueWrapper &src) noexcept { //moveAssign
    if constexpr (std::is_nothrow_move_assignable_v<T>) {
        *reinterpret_cast<T*>(&dst.small) =
            std::move(*reinterpret_cast<T*>(&src.small));
    }
    else {
        reinterpret_cast<T*>(&dst.small)->~T(); // destroy
        dst.interface =
            // move construct
            ::new (dst.small) T{std::move(*reinterpret_cast<T*>(&src.small))};
    }
}

```

};

}

static constexpr Methods getTableForSmallObject() {

```

[](ValueWrapper &v) noexcept { /*destroy ...*/},
[](ValueWrapper &dst, const ValueWrapper &src) { /*copyConstruct ...*/},
[](ValueWrapper &dst, ValueWrapper &src) noexcept { /*moveConstruct ...*/},
[](ValueWrapper &dst, const ValueWrapper &src) { //assign
    if constexpr (std::is_copy_assignable_v<T>) {
        *reinterpret_cast<T*>(&dst.small) =

```

```

[](ValueWrapper &dst, ValueWrapper &src) noexcept { //moveAssign
    if constexpr (std::is_nothrow_move_assignable_v<T>) {
        *reinterpret_cast<T*>(&dst.small) =
            std::move(*reinterpret_cast<T*>(&src.small));
    }
    else {
        reinterpret_cast<T*>(&dst.small)->~T(); // destroy
        dst.interface =
            // move construct
            ::new (dst.small) T{std::move(*reinterpret_cast<T*>(&src.small))};
    }
}

```

};

}

```

template<typename T>
static constexpr Methods getTableForBigObject() {
    return {
        [](ValueWrapper &v) noexcept { /*destroy ...*/},
        [](ValueWrapper &dst, const ValueWrapper &src) { /*copyConstruct ...*/},
        [](ValueWrapper &dst, ValueWrapper &src) noexcept { /*moveConstruct ...*/},
        [](ValueWrapper &dst, const ValueWrapper &src) { //assign
            if constexpr (std::is_copy_assignable_v<T>) {
                *static_cast<T*>(dst.big) = *static_cast<const T*>(src.big);
            }
            else {
                delete static_cast<T*>(dst.big); // destroy
                auto *instance = new T{ *static_cast<const T*>(src.big) };
                dst.big = instance; // copy construct
                dst.interface = instance;
            }
        },
        [](ValueWrapper &dst, ValueWrapper &src) noexcept { //moveAssign
            if constexpr (std::is_nothrow_move_assignable_v<T>) {
                *static_cast<T *>(dst.big) = std::move(*static_cast<T *>(src.big));
            }
            else {
                delete static_cast<T *>(dst.big); // destroy
                dst.big = std::exchange(src.big, nullptr); // move
                dst.interface = std::exchange(src.interface, nullptr);
                src.methods = nullptr;
            }
        }
    };
}

```

```
static constexpr Methods getTableForBigObject() {
```

```
    [](ValueWrapper &dst, const ValueWrapper &src) { //assign
        if constexpr (std::is_copy_assignable_v<T>) {
            *static_cast<T*>(dst.big) = *static_cast<const T*>(src.big);
        }
        else {
            delete static_cast<T*>(dst.big); // destroy
            auto *instance = new T{ *static_cast<const T*>(src.big) };
            dst.big = instance; // copy construct
            dst.interface = instance;
        }
    },
}
else {
    delete static_cast<T*>(dst.big); // destroy
    dst.big = std::exchange(src.big, nullptr); // move
    dst.interface = std::exchange(src.interface, nullptr);
    src.methods = nullptr;
}
};
}
```



```
static constexpr Methods getTableForBigObject() {
```

```

[] (ValueWrapper &dst, const ValueWrapper &src) { // assign
    if constexpr (std::is_copy_assignable_v<T>) {
        *static_cast<T*>(dst.big) = *static_cast<const T*>(src.big);
    }
    else {
        delete static_cast<T*>(dst.big); // destroy
        auto *instance = new T{ *static_cast<const T*>(src.big) };
        dst.big = instance; // copy construct
        dst.interface = instance;
    }
},
}
else {
    delete static_cast<T*>(dst.big); // destroy
    dst.big = std::exchange(src.big, nullptr); // move
    dst.interface = std::exchange(src.interface, nullptr);
    src.methods = nullptr;
}
}
};
}

```

```
static constexpr Methods getTableForBigObject() {
```

```

[] (ValueWrapper &dst, const ValueWrapper &src) { // assign
    if constexpr (std::is_copy_assignable_v<T>) {
        *static_cast<T*>(dst.big) = *static_cast<const T*>(src.big);
    }
    else {
        delete static_cast<T*>(dst.big); // destroy
        auto *instance = new T{ *static_cast<const T*>(src.big) };
        dst.big = instance; // copy construct
        dst.interface = instance;
    }
},
}
else {
    delete static_cast<T*>(dst.big); // destroy
    dst.big = std::exchange(src.big, nullptr); // move
    dst.interface = std::exchange(src.interface, nullptr);
    src.methods = nullptr;
}
}
};
}

```

static constexpr Methods getTableForBigObject() {

```

[](ValueWrapper &v) noexcept { /*destroy ...*/},
[](ValueWrapper &dst, const ValueWrapper &src) { /*copyConstruct ...*/},
[](ValueWrapper &dst, ValueWrapper &src) noexcept { /*moveConstruct ...*/},
[](ValueWrapper &dst, const ValueWrapper &src) { //assign
    if constexpr (std::is_copy_assignable_v<T>) {
        *static_cast<T*>(dst.big) = *static_cast<const T*>(src.big);
    }
    else {

```

```

[](ValueWrapper &dst, ValueWrapper &src) noexcept { //moveAssign
    if constexpr (std::is_nothrow_move_assignable_v<T>) {
        *static_cast<T *>(dst.big) = std::move(*static_cast<T *>(src.big));
    }
    else {
        delete static_cast<T *>(dst.big); // destroy
        dst.big = std::exchange(src.big, nullptr); // move
        dst.interface = std::exchange(src.interface, nullptr);
        src.methods = nullptr;
    }
}

```

};

}

static constexpr Methods getTableForBigObject() {

```

[](ValueWrapper &v) noexcept { /*destroy ...*/},
[](ValueWrapper &dst, const ValueWrapper &src) { /*copyConstruct ...*/},
[](ValueWrapper &dst, ValueWrapper &src) noexcept { /*moveConstruct ...*/},
[](ValueWrapper &dst, const ValueWrapper &src) { //assign
    if constexpr (std::is_copy_assignable_v<T>) {
        *static_cast<T*>(dst.big) = *static_cast<const T*>(src.big);
    }
    else {

```

```

[](ValueWrapper &dst, ValueWrapper &src) noexcept { //moveAssign
    if constexpr (std::is_nothrow_move_assignable_v<T>) {
        *static_cast<T *>(dst.big) = std::move(*static_cast<T *>(src.big));
    }
    else {
        delete static_cast<T *>(dst.big); // destroy
        dst.big = std::exchange(src.big, nullptr); // move
        dst.interface = std::exchange(src.interface, nullptr);
        src.methods = nullptr;
    }
}
}

```

};

}

static constexpr Methods getTableForBigObject() {

```

[](ValueWrapper &v) noexcept { /*destroy ...*/},
[](ValueWrapper &dst, const ValueWrapper &src) { /*copyConstruct ...*/},
[](ValueWrapper &dst, ValueWrapper &src) noexcept { /*moveConstruct ...*/},
[](ValueWrapper &dst, const ValueWrapper &src) { //assign
    if constexpr (std::is_copy_assignable_v<T>) {
        *static_cast<T*>(dst.big) = *static_cast<const T*>(src.big);
    }
    else {

```

```

[](ValueWrapper &dst, ValueWrapper &src) noexcept { //moveAssign
    if constexpr (std::is_nothrow_move_assignable_v<T>) {
        *static_cast<T *>(dst.big) = std::move(*static_cast<T *>(src.big));
    }
    else {
        delete static_cast<T *>(dst.big); // destroy
        dst.big = std::exchange(src.big, nullptr); // move
        dst.interface = std::exchange(src.interface, nullptr);
        src.methods = nullptr;
    }
}
}

```

};

}

Assignment support

```
ValueWrapper &operator=(const ValueWrapper &other) {  
    if (hasValue()) {  
        if (methods == other.methods) {  
            methods->assign(*this, other);  
            return *this;  
        }  
        methods->destroy(*this);  
        methods = nullptr;  
    }  
  
    if (other.methods) {  
        other.methods->copyConstruct(*this, other);  
        methods = other.methods;  
    }  
    return *this;  
}
```

Assignment support

```
ValueWrapper &operator=(ValueWrapper &&other) noexcept {  
    if (hasValue()) {  
        if (methods == other.methods) {  
            methods->moveAssign(*this, other);  
            return *this;  
        }  
  
        methods->destroy(*this);  
    }  
  
    methods = other.methods;  
    if (methods)  
        methods->moveConstruct(*this, other);  
  
    return *this;  
}
```

<code>auto v = ValueWrapper<Iface>{ Impl{} };</code>	<code>Impl::ctor</code>
	<code>Impl::move ctor</code>
	<code>Impl::dtor</code>
<code>auto x = v;</code>	<code>Impl::copy ctor</code>
<code>auto y = std::move(v);</code>	<code>Impl::move ctor</code>
<code>x = y;</code>	<code>Impl::copy =</code>
<code>x = std::move(y);</code>	<code>Impl::move =</code>
<code>x->foo();</code>	<code>Impl::foo(): abc</code>
	<code>Impl::dtor</code>
	<code>Impl::dtor</code>
	<code>Impl::dtor</code>

Other features

- `emplace`
- `value assignment`

```

template<typename T, typename... Args>
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
std::remove_cvref_t<T> &emplace(Args&&...args) {
    if (hasValue()) {
        methods->destroy(*this);
        methods = nullptr;
    }
    T *instance = nullptr;
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
        instance =
            ::new(small) std::remove_cvref_t<T>{ std::forward<Args>(args)... };
    }
    else {
        instance = new std::remove_cvref_t<T>{ std::forward<Args>(args)... };
        big = instance;
    }
    methods = &table<std::remove_cvref_t<T>>;
    interface = instance;
    return *instance;
}

```

```

template<typename T, typename... Args>
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
std::remove_cvref_t<T> &emplace(Args&&...args) {
    if (hasValue()) {
        methods->destroy(*this);
        methods = nullptr;
    }
    T *instance = nullptr;
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
        instance =
            ::new(small) std::remove_cvref_t<T>{ std::forward<Args>(args)... };
    }
    else {
        instance = new std::remove_cvref_t<T>{ std::forward<Args>(args)... };
        big = instance;
    }
    methods = &table<std::remove_cvref_t<T>>;
    interface = instance;
    return *instance;
}

```

```

template<typename T, typename... Args>
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
std::remove_cvref_t<T> &emplace(Args&&...args) {
    if (hasValue()) {
        methods->destroy(*this);
        methods = nullptr;
    }
    T *instance = nullptr;
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
        instance =
            ::new(small) std::remove_cvref_t<T>{ std::forward<Args>(args)... };
    }
    else {
        instance = new std::remove_cvref_t<T>{ std::forward<Args>(args)... };
        big = instance;
    }
    methods = &table<std::remove_cvref_t<T>>;
    interface = instance;
    return *instance;
}

```

```

template<typename T, typename... Args>
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
std::remove_cvref_t<T> &emplace(Args&&...args) {
    if (hasValue()) {
        methods->destroy(*this);
        methods = nullptr;
    }
    T *instance = nullptr;
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
        instance =
            ::new(small) std::remove_cvref_t<T>{ std::forward<Args>(args)... };
    }
    else {
        instance = new std::remove_cvref_t<T>{ std::forward<Args>(args)... };
        big = instance;
    }
    methods = &table<std::remove_cvref_t<T>>;
    interface = instance;
    return *instance;
}

```

```

template<typename T, typename... Args>
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
std::remove_cvref_t<T> &emplace(Args&&...args) {
    if (hasValue()) {
        methods->destroy(*this);
        methods = nullptr;
    }
    T *instance = nullptr;
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
        instance =
            ::new(small) std::remove_cvref_t<T>{ std::forward<Args>(args)... };
    }
    else {
        instance = new std::remove_cvref_t<T>{ std::forward<Args>(args)... };
        big = instance;
    }
    methods = &table<std::remove_cvref_t<T>>;
    interface = instance;
    return *instance;
}

```

```

template<typename T, typename... Args>
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
std::remove_cvref_t<T> &emplace(Args&&...args) {
    if (hasValue()) {
        methods->destroy(*this);
        methods = nullptr;
    }
    T *instance = nullptr;
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
        instance =
            ::new(small) std::remove_cvref_t<T>{ std::forward<Args>(args)... };
    }
    else {
        instance = new std::remove_cvref_t<T>{ std::forward<Args>(args)... };
        big = instance;
    }
    methods = &table<std::remove_cvref_t<T>>;
    interface = instance;
    return *instance;
}

```

```

template<typename T, typename... Args>
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
std::remove_cvref_t<T> &emplace(Args&&...args) {
    if (hasValue()) {
        methods->destroy(*this);
        methods = nullptr;
    }
    T *instance = nullptr;
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
        instance =
            ::new(small) std::remove_cvref_t<T>{ std::forward<Args>(args)... };
    }
    else {
        instance = new std::remove_cvref_t<T>{ std::forward<Args>(args)... };
        big = instance;
    }
    methods = &table<std::remove_cvref_t<T>>;
    interface = instance;
    return *instance;
}

```



```

template<typename T, typename... Args>
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
std::remove_cvref_t<T> &emplace(Args&&...args) {
    if (hasValue()) {
        methods->destroy(*this);
        methods = nullptr;
    }
    T *instance = nullptr;
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
        instance =
            ::new(small) std::remove_cvref_t<T>{ std::forward<Args>(args)... };
    }
    else {
        instance = new std::remove_cvref_t<T>{ std::forward<Args>(args)... };
        big = instance;
    }
    methods = &table<std::remove_cvref_t<T>>;
    interface = instance;
    return *instance;
}

```

```

template<typename T>
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
ValueWrapper &operator=(T &&object) {
    if constexpr (std::is_assignable_v<std::remove_cvref_t<T>, T&&>) {
        if (methods == &table<std::remove_cvref_t<T>>) {
            if constexpr (isSmall<std::remove_cvref_t<T>>) {
                *reinterpret_cast<T*>(&small) = std::forward<T>(object);
            }
            else {
                *static_cast<T*>(big) = std::forward<T>(object);
            }
            return *this;
        }
    }

    if (hasValue()) {
        methods->destroy(*this);
        methods = nullptr;
    }

    construct(std::forward<T>(object));
    methods = &table<std::remove_cvref_t<T>>;

    return *this;
}

```

```
template<typename T>
```

```
ValueWrapper &operator=(T &&object) {
```

```
    if constexpr (std::is_assignable_v<std::remove_cvref_t<T>, T&&>) {
```

```
        if constexpr (std::is_assignable_v<std::remove_cvref_t<T>, T&&>) {  
            if (methods == &table<std::remove_cvref_t<T>>) {  
                if constexpr (isSmall<std::remove_cvref_t<T>>) {  
                    *reinterpret_cast<T*>(&small) = std::forward<T>(object);  
                }  
                else {  
                    *static_cast<T*>(big) = std::forward<T>(object);  
                }  
                return *this;  
            }  
        }  
    }
```

```
    construct(std::forward<T>(object));
```

```
    methods = &table<std::remove_cvref_t<T>>;
```

```
    return *this;
```

```
}
```

```
template<typename T>
```

```
ValueWrapper &operator=(T &&object) {
```

```
    if constexpr (std::is_assignable_v<std::remove_cvref_t<T>, T&&>) {
```

```
        if constexpr (std::is_assignable_v<std::remove_cvref_t<T>, T&&>) {  
            if (methods == &table<std::remove_cvref_t<T>>) {  
                if constexpr (isSmall<std::remove_cvref_t<T>>) {  
                    *reinterpret_cast<T*>(&small) = std::forward<T>(object);  
                }  
                else {  
                    *static_cast<T*>(big) = std::forward<T>(object);  
                }  
                return *this;  
            }  
        }  
    }  
}
```

```
    construct(std::forward<T>(object));
```

```
    methods = &table<std::remove_cvref_t<T>>;
```

```
    return *this;
```

```
}
```

```
template<typename T>
```

```
ValueWrapper &operator=(T &&object) {
```

```
    if constexpr (std::is_assignable_v<std::remove_cvref_t<T>, T&&>) {
```

```
        if constexpr (std::is_assignable_v<std::remove_cvref_t<T>, T&&>) {  
            if (methods == &table<std::remove_cvref_t<T>>) {  
                if constexpr (isSmall<std::remove_cvref_t<T>>) {  
                    *reinterpret_cast<T*>(&small) = std::forward<T>(object);  
                }  
                else {  
                    *static_cast<T*>(big) = std::forward<T>(object);  
                }  
                return *this;  
            }  
        }  
    }
```

```
    construct(std::forward<T>(object));
```

```
    methods = &table<std::remove_cvref_t<T>>;
```

```
    return *this;
```

```
}
```

```
template<typename T>
```

```
ValueWrapper &operator=(T &&object) {
```

```
    if constexpr (std::is_assignable_v<std::remove_cvref_t<T>, T&&>) {
```

```
        if constexpr (std::is_assignable_v<std::remove_cvref_t<T>, T&&>) {  
            if (methods == &table<std::remove_cvref_t<T>>) {  
                if constexpr (isSmall<std::remove_cvref_t<T>>) {  
                    *reinterpret_cast<T*>(&small) = std::forward<T>(object);  
                }  
                else {  
                    *static_cast<T*>(big) = std::forward<T>(object);  
                }  
                return *this;  
            }  
        }  
    }
```

```
        construct(std::forward<T>(object));
```

```
        methods = &table<std::remove_cvref_t<T>>;
```

```
        return *this;
```

```
    }
```

```
template<typename T>
```

```
ValueWrapper &operator=(T &&object) {
```

```
    if constexpr (std::is_assignable_v<std::remove_cvref_t<T>, T&&>) {  
        if (methods == &table<std::remove_cvref_t<T>>) {  
            if constexpr (isSmall<std::remove_cvref_t<T>>) {  
                *reinterpret_cast<T*>(&small) = std::forward<T>(object);  
            }  
            else {
```

```
                if (hasValue()) {  
                    methods->destroy(*this);  
                    methods = nullptr;  
                }  
            }  
        }
```

```
        construct(std::forward<T>(object));  
        methods = &table<std::remove_cvref_t<T>>;
```

```
        return *this;
```

```
        return *this;
```

```
    }
```

```
template<typename T>
```

```
ValueWrapper &operator=(T &&object) {
```

```
    if constexpr (std::is_assignable_v<std::remove_cvref_t<T>, T&&>) {  
        if (methods == &table<std::remove_cvref_t<T>>) {  
            if constexpr (isSmall<std::remove_cvref_t<T>>) {  
                *reinterpret_cast<T*>(&small) = std::forward<T>(object);  
            }  
            else {
```

```
                if (hasValue()) {  
                    methods->destroy(*this);  
                    methods = nullptr;  
                }  
            }  
        }
```

```
        construct(std::forward<T>(object));  
        methods = &table<std::remove_cvref_t<T>>;
```

```
        return *this;
```

```
        return *this;
```

```
    }
```



```
template<typename T>
```

```
ValueWrapper &operator=(T &&object) {
```

```
    if constexpr (std::is_assignable_v<std::remove_cvref_t<T>, T&&>) {  
        if (methods == &table<std::remove_cvref_t<T>>) {  
            if constexpr (isSmall<std::remove_cvref_t<T>>) {  
                *reinterpret_cast<T*>(&small) = std::forward<T>(object);  
            }  
            else {
```

```
                if (hasValue()) {  
                    methods->destroy(*this);  
                    methods = nullptr;  
                }  
            }  
        }
```

```
        construct(std::forward<T>(object));  
        methods = &table<std::remove_cvref_t<T>>;
```

```
        return *this;
```

```
    }  
    return *this;  
}
```

```
template<typename T>
```

```
ValueWrapper &operator=(T &&object) {
```

```
    if constexpr (std::is_assignable_v<std::remove_cvref_t<T>, T&&>) {  
        if (methods == &table<std::remove_cvref_t<T>>) {  
            if constexpr (isSmall<std::remove_cvref_t<T>>) {  
                *reinterpret_cast<T*>(&small) = std::forward<T>(object);  
            }  
            else {
```

```
                if (hasValue()) {  
                    methods->destroy(*this);  
                    methods = nullptr;  
                }  
            }  
        }
```

```
        construct(std::forward<T>(object));  
        methods = &table<std::remove_cvref_t<T>>;
```

```
        return *this;
```

```
        return *this;
```

```
    }
```

Value assignment

```
private:
```

```
//...
```

```
template<typename T>
```

```
void construct(T &&object) {
```

```
    if constexpr (isSmall<std::remove_cvref_t<T>>) {
```

```
        interface =
```

```
            ::new(small) std::remove_cvref_t<T>{ std::forward<T>(object) };
```

```
    }
```

```
    else {
```

```
        auto *instance = new std::remove_cvref_t<T>{ std::forward<T>(object) };
```

```
        big = instance;
```

```
        interface = instance;
```

```
    }
```

```
}
```

Refactored constructor

```
template<typename T>
    requires (std::is_base_of_v<Interface, std::remove_cvref_t<T>>)
explicit ValueWrapper(T &&object) :
    methods{ &table<std::remove_cvref_t<T>> }
{
    construct(std::forward<T>(object));
}
```