делаем обёртку с семантикой значения для полиморфных типов

Fun with type erasure:

implementing a value wrapper for polymorphic types

Павел Новиков

𝕏 @cpp_ape

- 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_ohIHQk

The idea

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


Pointer semantics

```cpp
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

```cpp
std::unique_ptr<Shape> s = std::make_unique<Star>();
```
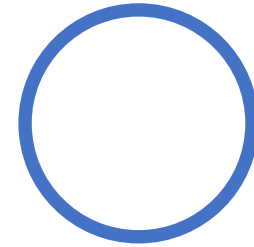
- pointer/reference semantics
- generally **can not** copy/clone values
- generally has to use memory allocation

Pointer semantics

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

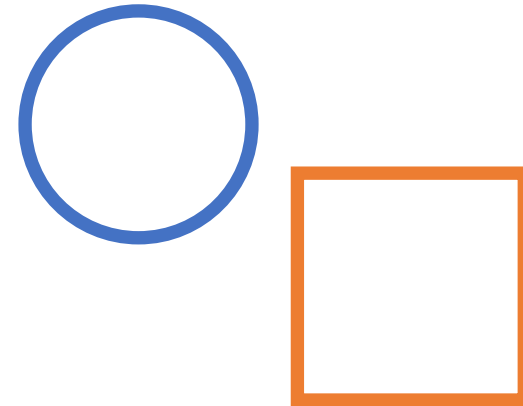# Pointer semantics

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

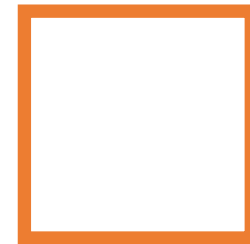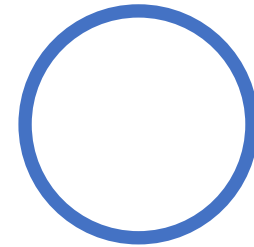Pointer semantics

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

Pointer semantics

```cpp
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

```cpp
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

```cpp
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

```cpp
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(2.f, 3.f);

auto clone = shape1;
clone.value().changeSize(3.f, 3.f);
```

Value semantics?

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


auto &shape1 = shapes[1];
shape1.value().changeSize(2.f, 3.f);


auto clone = shape1;
clone.value().changeSize(3.f, 3.f);
```

Value semantics?

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
//...
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

```cpp
//...
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

```cpp
//...
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

```cpp
//...
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

```cpp
//...
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

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

calling by function pointer

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 / sizeof(void*)) - 2) * sizeof(void*) = 48 (56 for trivial types)
libstdc++	16	sizeof(void*) = 8
libc++	32	3 * sizeof(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 / sizeof(void*)) - 2) * sizeof(void*) = 48 (56 for trivial types)
libstdc++	16	sizeof(void*) = 8
libc++	32	3 * sizeof(void*) = 24

* for x64

+ sizeof(getter) for ValueWrapper

Basic solution: drawbacks

`std::any` small object optimization:

std lib	small object capacity
MSVC	`((6 + 16 / sizeof(void*)) - 2) * sizeof(void*) = 48` (56 for trivial types)
libstdc++	`sizeof(void*) = 8`
libc++	`3 * sizeof(void*) = 24`

* for x64

+ `sizeof` ...apper

Basic solution: drawbacks

`std::any` small object optimization:

small buffer
+ pointer to function table
+ pointer to type info

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

small buffer
+ pointer to function

* 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 / sizeof(void*)) - 2) * sizeof(void*) = 48` (56 for trivial types)
libstdc++	16	`sizeof(void*) = 8`
libc++	32	`3 * sizeof(void*) = 24`

* for x64

+ `sizeof(getter)` for `ValueWrapper`

Basic solution: drawbacks

```cpp
template<typename Interface>
struct ValueWrapper {
  ValueWrapper() noexcept = default;
  template<typename T>
    requires (std::is_base_of_v<Interface, std::decay_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

```cpp
template<typename Interface>
struct ValueWrapper {
  ValueWrapper() noexcept = default;
  template<typename T>
    requires (std::is_base_of_v<Interface, std::decay_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

```cpp
template<typename Interface>
struct ValueWrapper {
  ValueWrapper() noexcept = default;
  template<typename T>
    requires (std::is_base_of_v<Interface, std::decay_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

```cpp
template<typename Interface>
struct ValueWrapper {
  ValueWrapper() noexcept = default;
  template<typename T>
    requires (std::is_base_of_v<Interface, std::decay_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

```cpp
template<typename Interface>
struct ValueWrapper {
  ValueWrapper() noexcept = default;
  template<typename T>
    requires (std::is_base_of_v<Interface, std::decay_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

```cpp
template<typename Interface>
struct ValueWrapper {
  ValueWrapper() noexcept = default;
  template<typename T>
    requires (std::is_base_of_v<Interface, std::decay_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

```cpp
template<typename Interface>
struct ValueWrapper {
  ValueWrapper() noexcept = default;
  template<typename T>
    requires (std::is_base_of_v<Interface, std::decay_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

```cpp
template<typename Interface>
struct ValueWrapper {
  ValueWrapper() noexcept = default;
  template<typename T>
    requires (std::is_base_of_v<Interface, std::decay_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

```cpp
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

```cpp
    //...
  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;
    };
    //...
```

```cpp
    //...
  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;
    };
    //...
```

```cpp
    //...
  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;
    };
    //...
```

```cpp
    //...
  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;
  };
  //...
```

```cpp
    //...
  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>();
};
```

```cpp
    //...
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>();
};
```

```cpp
  //...
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>();
};
```

```cpp
    //...
  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>();
};
```

```cpp
  //...
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>();
};
```

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
ValueWrapper() noexcept = default;

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

Minimal implementation

```cpp
ValueWrapper() noexcept = default;


template<typename T>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
explicit ValueWrapper(T &&object) :
  methods{ &table<std::decay_t<T>> }
{

  if constexpr (isSmall<std::decay_t<T>>) {
    ::new(small) std::decay_t<T>{ std::forward<T>(object) };
  }

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

Minimal implementation

```cpp
ValueWrapper() noexcept = default;

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

Minimal implementation

```cpp
ValueWrapper() noexcept = default;

template<typename T>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
explicit ValueWrapper(T &&object) :
  methods{ &table<std::decay_t<T>> }
{

  if constexpr (isSmall<std::decay_t<T>>) {
    ::new(small) std::decay_t<T>{ std::forward<T>(object) };
  }

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

Minimal implementation

```cpp
ValueWrapper() noexcept = default;

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

Minimal implementation

```cpp
ValueWrapper() noexcept = default;

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

Minimal implementation

```cpp
ValueWrapper() noexcept = default;

template<typename T>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
explicit ValueWrapper(T &&object) :
  methods{ &table<std::decay_t<T>> }
{
  if constexpr (isSmall<std::decay_t<T>>) {
    ::new(small) std::decay_t<T>{ std::forward<T>(object) };
  }

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

Minimal implementation

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

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

Minimal implementation

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

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

Minimal implementation

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


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

Minimal implementation

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

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

```cpp
auto x = v;
```
```
Impl::copy ctor
```

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

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

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

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

```
Impl::dtor
Impl::dtor
Impl::dtor
```

```
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
```

```
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
```

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

meh…

Let's save pointer to interface

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

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

  Interface *interface = nullptr;
  //
```

Let's save pointer to interface

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

Let's save pointer to interface

```cpp
template<typename T>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
explicit ValueWrapper(T &&object) :
  methods{ &table<std::decay_t<T>> }
{

  if constexpr (isSmall<std::decay_t<T>>) {
    interface =
      ::new(small) std::decay_t<T>{ std::forward<T>(object) };
  }
  else {
    auto *instance = new std::decay_t<T>{ std::forward<T>(object) };
    big = instance;
    interface = instance;
  }
}
```

Let's save pointer to interface

```cpp
template<typename T>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
explicit ValueWrapper(T &&object) :
  methods{ &table<std::decay_t<T>> }
{

  if constexpr (isSmall<std::decay_t<T>>) {
    interface =
      ::new(small) std::decay_t<T>{ std::forward<T>(object) };
  }
  else {
    auto *instance = new std::decay_t<T>{ std::forward<T>(object) };
    big = instance;
    interface = instance;
  }
}
```

Let's save pointer to interface

```cpp
template<typename T>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
explicit ValueWrapper(T &&object) :
  methods{ &table<std::decay_t<T>> }
{

  if constexpr (isSmall<std::decay_t<T>>) {
    interface =
      ::new(small) std::decay_t<T>{ std::forward<T>(object) };
  }
  else {
    auto *instance = new std::decay_t<T>{ std::forward<T>(object) };
    big = instance;
    interface = instance;
  }
}
```

Let's save pointer to interface

```cpp
struct Methods {
  void (*destroy)(ValueWrapper &data) noexcept;

  void (*copyConstruct)(ValueWrapper &dst, const ValueWrapper &src);
};
```

removed get

Let's save pointer to interface

```cpp
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

```cpp
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 + 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 + 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 + 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 + pointer

Comparison to `std::any`

libstdc++

```cpp
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 + 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 + pointer

Comparison to `std::any`

libc++

```cpp
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 + pointer

Comparison to `std::any`

ValueWrapper

```
union {
    std::byte small[smallCapacity];
    void *big;
};
const Methods *methods;
Interface *interface;
```

small buffer + pointer

pointer to function table

pointer to interface

Comparison to std::any

ValueWrapper

```cpp
union {
  std::byte small[smallCapacity];
  void *big;
};
const Methods *methods;
Interface *interface;
```

small buffer + pointer

pointer to function table

pointer to interface

Comparison to `std::any`

ValueWrapper

```
union {
  std::byte small[smallCapacity];
  void *big;
};
const Methods *methods;
Interface *interface;
```

small buffer + pointer

pointer to function table

pointer to interface

Comparison to `std::any`

ValueWrapper

```cpp
union {
  std::byte small[smallCapacity];
  void *big;
};
const Methods *methods;
Interface *interface;
```

small buffer + pointer

pointer to function table

pointer to interface

Comparison to `std::any`

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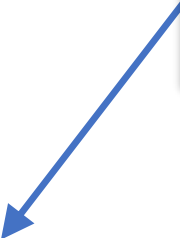
```cpp
template<typename T>
static constexpr bool isSmall =
    sizeof(T) <= smallCapacity &&
    alignof(T) <= alignof(void*) &&
    std::is_nothrow_move_constructible_v<T>;

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

support noexcept move
for small objects

Let's move!

```cpp
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))};
    }
  };
}
```

```cpp
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;
    }
  };
}
```

```cpp
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;
}
```

```cpp
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;
}
```

```cpp
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;
}
```

```cpp
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;
}
```

```cpp
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;
}
```

```cpp
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;
}
```

```cpp
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`	**libstdc++** and **libc++** `std::any`
just moves	moves and destroys the initial object
`Impl::move ctor`	`Impl::move ctor` `Impl::dtor`

Differences to `std::any`

Small object move assignment:

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

ValueWrapper<Iface> and MSVC std::any	libstdc++ std::any	libc++ std::any
destroys this object and moves	destroys this object, moves, and destroys the initial object	destroys this object, moves, and destroys the initial object, but in a funny way

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

Differences to `std::any`

Small object move assignment:

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

ValueWrapper<Iface> and MSVC std::any	libstdc++ std::any	libc++ std::any
destroys `this` object and moves	destroys `this` object, moves, and destroys the initial object	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`

we're gonna need a bigger slide

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;
}
```

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

Differences to std::any

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

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;`

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

ValueWrapper<Iface>	MSVC and l	Impl::copy ctor
destroys this object and copies	constructs a ... moves into ...	Impl::move ctor
		Impl::dtor
		Impl::move ctor
		Impl::dtor
		Impl::move ctor
		Impl::dtor
		Impl::dtor

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

```
Impl::co
Impl::dt
Impl::mo
Impl::dtor
```

Differences to `std::any`

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

```
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::decay_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::decay_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::decay_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::decay_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::decay_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::decay_t<T>>)`
- in-place construction?
- reset?
  `x = {};`
- swap?
- experiment with function tables vs. "do-all" functions?
  - performance?
  - binary size?

What else?

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

See also

Thanks for listening!

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

Павел Новиков

 @cpp_ape

Slides: bit.ly/3ErIwXa

Bonus slides

Assignment support

```cpp
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;
};
```

```cpp
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
      }
    }
  };
}
```

57

```cpp
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)};
        }
    },
```

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

58

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

58

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

58

```cpp
template<typename T>
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))};
    }
  }
}

  };
}
```

59

```cpp
template<typename T>
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))};
        }
    }
}
    };
}
```

59

```cpp
template<typename T>

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))};
    }
  }

};
}
```

```cpp
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;

      }

    }

  };

}
```

60

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

61

```
template<typename T>

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;
    }
  }
};
}
```

61

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

61

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

```
template<typename T>
```

```cpp
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 {
```

```cpp
    [](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;
        }
    }
}
```

```cpp
    };
}
```

62

```cpp
template<typename T>
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;
        }
    }
}
    };
}
```
62

Assignment support

```cpp
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;
}
```

63

Assignment support

```cpp
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;
}
```

```
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::copy =


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


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


                                            Impl::dtor
                                            Impl::dtor
                                            Impl::dtor
```

Other features

- emplace
- value assignment

```cpp
template<typename T, typename... Args>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
std::decay_t<T> &emplace(Args&&...args) {
  if (hasValue()) {
    methods->destroy(*this);
    methods = nullptr;
  }

  T *instance = nullptr;
  if constexpr (isSmall<std::decay_t<T>>) {
    instance = ::new(small) std::decay_t<T>{ std::forward<Args>(args)... };
  }
  else {
    instance = new std::decay_t<T>{ std::forward<Args>(args)... };
    big = instance;
  }
  methods = &table<std::decay_t<T>>;
  interface = instance;
  return *instance;
}
```

```cpp
template<typename T, typename... Args>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
std::decay_t<T> &emplace(Args&&...args) {
  if (hasValue()) {
    methods->destroy(*this);
    methods = nullptr;
  }

  T *instance = nullptr;
  if constexpr (isSmall<std::decay_t<T>>) {
    instance = ::new(small) std::decay_t<T>{ std::forward<Args>(args)... };
  }
  else {
    instance = new std::decay_t<T>{ std::forward<Args>(args)... };
    big = instance;
  }
  methods = &table<std::decay_t<T>>;
  interface = instance;
  return *instance;
}
```

```cpp
template<typename T, typename... Args>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
std::decay_t<T> &emplace(Args&&...args) {
  if (hasValue()) {
    methods->destroy(*this);
    methods = nullptr;
  }

  T *instance = nullptr;
  if constexpr (isSmall<std::decay_t<T>>) {
    instance = ::new(small) std::decay_t<T>{ std::forward<Args>(args)... };
  }
  else {
    instance = new std::decay_t<T>{ std::forward<Args>(args)... };
    big = instance;
  }
  methods = &table<std::decay_t<T>>;
  interface = instance;
  return *instance;
}
```

```cpp
template<typename T, typename... Args>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
std::decay_t<T> &emplace(Args&&...args) {
  if (hasValue()) {
    methods->destroy(*this);
    methods = nullptr;
  }

  T *instance = nullptr;
  if constexpr (isSmall<std::decay_t<T>>) {
    instance = ::new(small) std::decay_t<T>{ std::forward<Args>(args)... };
  }
  else {
    instance = new std::decay_t<T>{ std::forward<Args>(args)... };
    big = instance;
  }
  methods = &table<std::decay_t<T>>;
  interface = instance;
  return *instance;
}
```

```cpp
template<typename T, typename... Args>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
std::decay_t<T> &emplace(Args&&...args) {
  if (hasValue()) {
    methods->destroy(*this);
    methods = nullptr;
  }

  T *instance = nullptr;
  if constexpr (isSmall<std::decay_t<T>>) {
    instance = ::new(small) std::decay_t<T>{ std::forward<Args>(args)... };
  }
  else {
    instance = new std::decay_t<T>{ std::forward<Args>(args)... };
    big = instance;
  }
  methods = &table<std::decay_t<T>>;
  interface = instance;
  return *instance;
}
```

```cpp
template<typename T, typename... Args>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
std::decay_t<T> &emplace(Args&&...args) {
  if (hasValue()) {
    methods->destroy(*this);
    methods = nullptr;
  }

  T *instance = nullptr;
  if constexpr (isSmall<std::decay_t<T>>) {
    instance = ::new(small) std::decay_t<T>{ std::forward<Args>(args)... };
  }
  else {
    instance = new std::decay_t<T>{ std::forward<Args>(args)... };
    big = instance;
  }
  methods = &table<std::decay_t<T>>;
  interface = instance;
  return *instance;
}
```
67

```cpp
template<typename T, typename... Args>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
std::decay_t<T> &emplace(Args&&...args) {
  if (hasValue()) {
    methods->destroy(*this);
    methods = nullptr;
  }

  T *instance = nullptr;
  if constexpr (isSmall<std::decay_t<T>>) {
    instance = ::new(small) std::decay_t<T>{ std::forward<Args>(args)... };
  }
  else {
    instance = new std::decay_t<T>{ std::forward<Args>(args)... };
    big = instance;
  }
  methods = &table<std::decay_t<T>>;
  interface = instance;
  return *instance;
}
```

```cpp
template<typename T, typename... Args>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
std::decay_t<T> &emplace(Args&&...args) {
  if (hasValue()) {
    methods->destroy(*this);
    methods = nullptr;
  }

  T *instance = nullptr;
  if constexpr (isSmall<std::decay_t<T>>) {
    instance = ::new(small) std::decay_t<T>{ std::forward<Args>(args)... };
  }
  else {
    instance = new std::decay_t<T>{ std::forward<Args>(args)... };
    big = instance;
  }
  methods = &table<std::decay_t<T>>;
  interface = instance;
  return *instance;
}
```

```cpp
template<typename T>
  requires (std::is_base_of_v<Interface, std::decay_t<T>>)
ValueWrapper &operator=(T &&object) {
  if constexpr (std::is_assignable_v<std::decay_t<T>, T&&>) {
    if (methods == &table<std::decay_t<T>>) {
      if constexpr (isSmall<std::decay_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::decay_t<T>>;

  return *this;
}
```

68

```cpp
template<typename T>
ValueWrapper &operator=(T &&object) {
  if constexpr (std::is_assignable_v<std::decay_t<T>, T&&>) {

if constexpr (std::is_assignable_v<std::decay_t<T>, T&&>) {
  if (methods == &table<std::decay_t<T>>) {
    if constexpr (isSmall<std::decay_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::decay_t<T>>;

  return *this;
}
```

```cpp
template<typename T>
ValueWrapper &operator=(T &&object) {
  if constexpr (std::is_assignable_v<std::decay_t<T>, T&&>) {
    if constexpr (std::is_assignable_v<std::decay_t<T>, T&&>) {
      if (methods == &table<std::decay_t<T>>) {
        if constexpr (isSmall<std::decay_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::decay_t<T>>;

  return *this;
}
```

69

```cpp
template<typename T>
ValueWrapper &operator=(T &&object) {
  if constexpr (std::is_assignable_v<std::decay_t<T>, T&&>) {

  if constexpr (std::is_assignable_v<std::decay_t<T>, T&&>) {
    if (methods == &table<std::decay_t<T>>) {
      if constexpr (isSmall<std::decay_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::decay_t<T>>;

  return *this;
}
```

69

```cpp
template<typename T>
ValueWrapper &operator=(T &&object) {
    if constexpr (std::is_assignable_v<std::decay_t<T>, T&&>) {

    if constexpr (std::is_assignable_v<std::decay_t<T>, T&&>) {
        if (methods == &table<std::decay_t<T>>) {
            if constexpr (isSmall<std::decay_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::decay_t<T>>;


    return *this;
}
```

```cpp
template<typename T>
ValueWrapper &operator=(T &&object) {
    if constexpr (std::is_assignable_v<std::decay_t<T>, T&&>) {
        if (methods == &table<std::decay_t<T>>) {
            if constexpr (isSmall<std::decay_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::decay_t<T>>;


return *this;

    return *this;
}
```
70

```cpp
template<typename T>
ValueWrapper &operator=(T &&object) {
    if constexpr (std::is_assignable_v<std::decay_t<T>, T&&>) {
        if (methods == &table<std::decay_t<T>>) {
            if constexpr (isSmall<std::decay_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::decay_t<T>>;


return *this;

    return *this;
}
```

70

```cpp
template<typename T>
ValueWrapper &operator=(T &&object) {
    if constexpr (std::is_assignable_v<std::decay_t<T>, T&&>) {
        if (methods == &table<std::decay_t<T>>) {
            if constexpr (isSmall<std::decay_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::decay_t<T>>;


return *this;

    return *this;
}
```

70

```cpp
template<typename T>
ValueWrapper &operator=(T &&object) {
    if constexpr (std::is_assignable_v<std::decay_t<T>, T&&>) {
        if (methods == &table<std::decay_t<T>>) {
            if constexpr (isSmall<std::decay_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::decay_t<T>>;


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

70

Value assignment

```cpp
private:
  //...
  template<typename T>
  void construct(T &&object) {
    if constexpr (isSmall<std::decay_t<T>>) {
      interface =
        ::new(small) std::decay_t<T>{ std::forward<T>(object) };
    }
    else {
      auto *instance = new std::decay_t<T>{ std::forward<T>(object) };
      big = instance;
      interface = instance;
    }
  }
```

Refactored constructor

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