

# Время жизни

## Задача: реализовать класс Optional<T>

```
template <typename T>
class Optional {
public:
    Optional();
    Optional(const T& value);

    T& value();

    bool has_value() const;

private: // ???
};
```

*Предложите реализацию*

## Попытка 1: наивное произведение типов

```
template <typename T>
class Optional {
public:
    Optional() = default;
    Optional(const T& value)
        : value_(value), has_value_(true)

    ...
private:
    T value_;
    bool has_value_ = false;
};
```

*Недостатки решения?*

## Напоминание: наш отладочный Tracer

```
struct Tracer {  
    Tracer() { std::cout << "Tracer::Tracer()\n"; }  
    ~Tracer() { std::cout << "Tracer::~~Tracer()\n"; }  
  
    Tracer(const Tracer&) { ... }  
    Tracer(Tracer&&) { ... }  
    Tracer& operator=(const Tracer&) { ... }  
    Tracer& operator=(Tracer&&) { ... }  
};
```

## Иллюстрация проблемы

```
Optional<Tracer> tracer_opt;  
std::cout << std::boolalpha  
          << tracer_opt.has_value() << '\n';
```

*Какой будет вывод?*

# Иллюстрация проблемы

```
Optional<Tracer> tracer_opt;  
std::cout  
    << std::boolalpha  
    << "has_value: " << tracer_opt.has_value() << '\n';
```

Вывод:

```
Tracer::Tracer()  
has_value: false  
Tracer::~Tracer()
```

# Проблемы наивного решения

1. Для `T value_` безусловно вызывается конструктор по умолчанию.
2. При этом `has_value_ = false`.

=> Нарушен инвариант класса.

## Попытка 2: pointer-like

```
template <typename T>
class Optional {
public:
    Optional() = default;
    Optional(const T& value) : value_(new T(value)) {}

    T& value() { return *value_; }
    bool has_value() const { return value_ != nullptr; }

private:
    T* value_ = nullptr;
};
```

*TODO: Rule of 5*



## Попытка 2: pointer-like

```
template <typename T>
class Optional {
public:
    Optional() = default;
    Optional(const T& value) : value_(new T(value)) {}

    T& value() { return *value_; }
    bool has_value() const { return value_ != nullptr; }

private:
    T* value_ = nullptr;
};
```

*Недостатки решения?*


## Попытка 2: pointer-like

```
template <typename T>
class Optional {
public:
    Optional() = default;
    Optional(const T& value) : value_(new T(value)) {}

    T& value() { return *value_; }
    bool has_value() const { return value_ != nullptr; }

private:
    T* value_ = nullptr;
};
```

*Аллокации*



# Промежуточный итог: недостатки решений

Произведение типов:

- Нарушен инвариант: объект существует при `has_value_ == false`
- Для живого объекта будут вызываться все специальные методы

Pointer-like:

- Аллокации

## `std::optional` лишен ЭТИХ недостатков

- `optional` handles expensive-to-construct objects
- no dynamic memory allocation ever takes place

# Время жизни объекта

Начинается когда:

- Выделена память и
- Выполнена инициализация

*Крайне упрощено, до уровня обмана.  
См. стандарт.*

Завершается когда:

- Начинается вызов деструктора или
- Память освобождена

# new

- operator new выделяет память
- placement new expression конструирует объект в выделенной памяти
- new expression выделяет память и вызывает конструктор(ы)

## Попытка 3: raw bytes

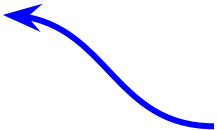
```
template <typename T>
class Optional {
public:
    ...
private:
    char data[sizeof(T)];
    bool has_value_ = false;
};
```

*Нет динамического выделения памяти*

## Попытка 3: raw bytes

```
template <typename T>
class Optional {
public:
    Optional(const T& value) : has_value_(true) {
        new (data_) T(value);
    }

private:
    char data_[sizeof(T)];
    bool has_value_ = false;
};
```



*Placement new*



## Попытка 3: raw bytes

```
template <typename T>
```

```
class Optional {
```

```
public:
```

```
    Optional(const T& value) : has_value_(true) {
```

```
        new (data_) T(value);
```

```
    }
```

*Явный вызов деструктора*

```
    ~Optional() { if (has_value_) value().~T(); }
```

```
};
```

*TODO: Rule of 5*

## Попытка 3: raw bytes

```
template <typename T>
class Optional {
public:
    ~Optional() { if (has_value_) value().~T(); }

    T& value() { return ??? data_; }

private:
    char data_[sizeof(T)];
};
```

## Попытка 3: raw bytes

```
template <typename T>
class Optional {
public:
    ~Optional() { if (has_value_) value().~T(); }

    T& value() { return *reinterpret_cast<T*>(data_); }

private:
    char data_[sizeof(T)];
};
```

## Попытка 4: union-like класс

```
template <typename T> class Optional {  
    public:  
        Optional(const T& value)  
            : has_value_(true) { new (&value_) T(value); }  
  
        T& value() { return value_; }  
  
    private:  
        union { T value_; };  
        bool has_value_;  
};
```

# Ключевая идея

- Мы можем разделять выделение памяти и конструирование объекта в ней

## Вернемся к проблеме в сору ctor вектора

```
Vector(const Vector<T>& other)
    : items_(new T[other.size()]()),
      size_(other.size()) {
    for (std::size_t i = 0; i < size(); ++i) {
        items_[i] = other[i];
    }
}
```

## Вернемся к проблеме в сору ctor вектора

```
Vector(const Vector<T>& other)
    : items_(new T[other.size()]()),
      size_(other.size()) {
    for (std::size_t i = 0; i < size(); ++i) {
        items_[i] = other[i];
    }
}
```

*Вызов конструкторов  
по умолчанию*

*Вызов операторов присваивания*

## Используем placement new

```
Vector(const Vector<T>& other)
    : items_(
        static_cast<T*>(
            operator new(sizeof(T) * other.size()))),
        size_(other.size()) {

    for (std::size_t i = 0; i < size(); ++i) {
        new (items_ + i) T(other[i]);
    }
}
```



## Используем placement new

```
Vector(const Vector<T>& other)
    : items_(
        static_cast<T*>(
            operator new(sizeof(T) * other.size()))),
        size_(other.size()) {

    std::uninitialized_copy_n(
        other.items_, other.size(), items_);
}
```

## В деструкторе парные операции

```
~Vector() {  
    for (std::size_t i = 0; i < size(); ++i) {  
        items_[i].~T();  
    }  
    operator delete(items_);  
}
```

## В деструкторе парные операции

```
~Vector() {  
    std::destroy_n(items_, size_);  
    operator delete(items_);  
}
```

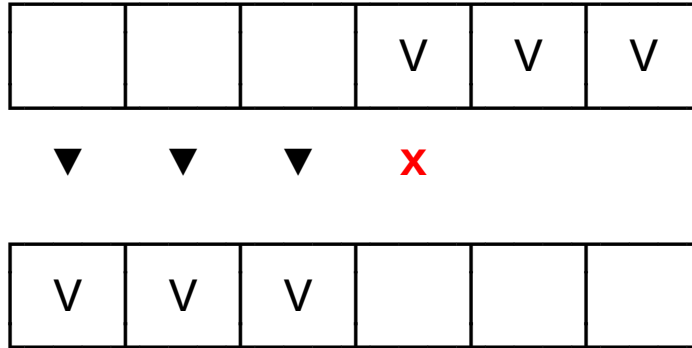
См. реализацию

[https://en.cppreference.com/w/cpp/memory/uninitialized\\_copy](https://en.cppreference.com/w/cpp/memory/uninitialized_copy)

# Как быть с исключениями в move-конструкторе?

```
Vector(Vector<T>&& other)
    : items_(
        static_cast<T*>(
            operator new(sizeof(T) * other.size()))),
      size_(other.size()) {
    for (std::size_t i = 0; i < size_; ++i) {
        new (items_ + i) T(std::move(other[i]));
    }
}
```

# Иллюстрация перемещения



*Смогли переместить  
только часть.  
Что делать?*

Продолжение следует

PS



## Вернемся к типу Optional

```
template <typename T> class Optional {  
    public:  
        // operator bool();  
        // operator*();  
        // operator->();  
};
```

## operator bool

```
template <typename T> class Optional {  
    public:
```

```
        ??? operator bool() { ??? }
```

```
    private:
```

```
        bool has_value_ = false;
```

```
        union { T value_; };
```

```
};
```

## operator\*

```
template <typename T> class Optional {  
    public:  
  
        ??? operator*() { ??? }  
  
    private:  
        bool has_value_ = false;  
        union { T value_; };  
};
```

## operator\*

```
template <typename T> class Optional {  
    public:  
  
        T& operator*() { return value_; }  
        const T& operator*() const { return value_; }  
  
    private:  
        bool has_value_ = false;  
        union { T value_; };  
};
```

## operator->

```
template <typename T> class Optional {  
    public:  
  
        ??? operator->() { ... }  
  
    private:  
        bool has_value_ = false;  
        union { T value_; };  
};
```

## operator->

```
template <typename T> class Optional {  
    public:  
  
        const T* operator->() const { return &value_; }  
        T* operator->() { return &value_; }  
  
    private:  
        bool has_value_ = false;  
        union { T value_; };  
};
```

# Drilldown

<https://godbolt.org/z/Y1P8TvhY6>

В лекциях везде обман

<https://en.cppreference.com/w/cpp/utility/optional/value>



Продолжение следует