Жизненный цикл объекта

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()
```


https://godbolt.org/z/cYhzWY35n

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

std::optional

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

[basic.life]

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;
};
```

Попытка 3: raw bytes

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

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

};
```

TODO: Rule of 5

Явный вызов деструктора

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

Задача

Реализовать класс Vector.

Попытка 1

```
template <typename T>
struct Vector {
  Vector(std::size_t size)
    : items_(new T[size]()),
      size_(size) {
  }

  ~Vector() { delete[] items_; }

  T* items_;
  std::size_t size_;
};
```

Попытка 1

```
template <typename T>
struct Vector {
  Vector(std::size_t size)
    : items_(new T[size]()),
      size_(size) {
  }

  ~Vector() { delete[] items_; }

  T* items_;
  std::size_t size_;
};
```

Покритикуйте эту реализацию

https://godbolt.org/z/jsfx1Yh7s

Недостатки

1. Нет сокрытия данных, легко нарушить инвариант.
2. Сокрытие данных повлечет за собой реализацию интерфейса.
3. Нарушено Rule of 5.
4. Конструктор не explicit.

В процессе устранения этих проблем появятся новые.

Делаем explicit конструктор

```
template <typename T>
struct Vector {
  explicit Vector(std::size_t size)
    : items_(new T[size]()),
      size_(size) {
  }

  ~Vector() { delete[] items_; }

  T* items_;
  std::size_t size_;
};
```

Скрываем данные

```
template <typename T>
class Vector {
 public:
  explicit Vector(std::size_t size);
  ~Vector();

 private:
  T* items_;
  std::size_t size_;
};
```

Предложите интерфейс

Предложите интерфейс

- Получение количества элементов
- Обращение к элементу по индексу

Реализуем интерфейс

```
template <typename T>
class Vector {
 public:
  std::size_t size() { return size_; }

};
```

Реализуем интерфейс

```
template <typename T>
class Vector {
 public:
  std::size_t size() { return size_; }

};
```

Критика?

Реализуем интерфейс

```
template <typename T>
class Vector {
 public:
  std::size_t size() const { return size_; }

};
```

Критика?

Реализуем интерфейс

```
template <typename T>
class Vector {
 public:
  ??? operator[](std::size_t i) { return items_[i]; }

};
```

Реализуем интерфейс

```
template <typename T>
class Vector {
 public:
  T& operator[](std::size_t i) { return items_[i]; }

};
```

Реализуем интерфейс

```
template <typename T>
class Vector {
 public:
  T& operator[](std::size_t i) { return items_[i]; }

  const T& operator[](std::size_t i) const
  { return items_[i]; }

};
```

Промежуточный итог

- Этот вектор уже полезнее, чем C-style динамический массив
- Но все еще много проблем
- https://godbolt.org/z/6hhf49461

Copy Ctor

```
Vector<int> v1(10);

Vector<int> v2(v1);
```

https://godbolt.org/z/r67x7fnas

Как реализовать конструктор копирования?

Попытка 1: Copy Ctor

```
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];
  }
}
```

Попытка 1: Copy Ctor

```
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];
  }
}
```

Покритикуйте это решение

Попытка 1: Copy Ctor

```
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];
  }
}
```

Какие операции вызываются для T?

Попытка 1: Copy Ctor

```
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];
  }
}
```


https://godbolt.org/z/oh13PTsse

Проблемы реализации

```
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

https://godbolt.org/z/oh13PTsse

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]);
  }
}
```


https://godbolt.org/z/bKMd7P46Y

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_);
}
```


https://godbolt.org/z/r4hnGEKqz

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_);
}
```

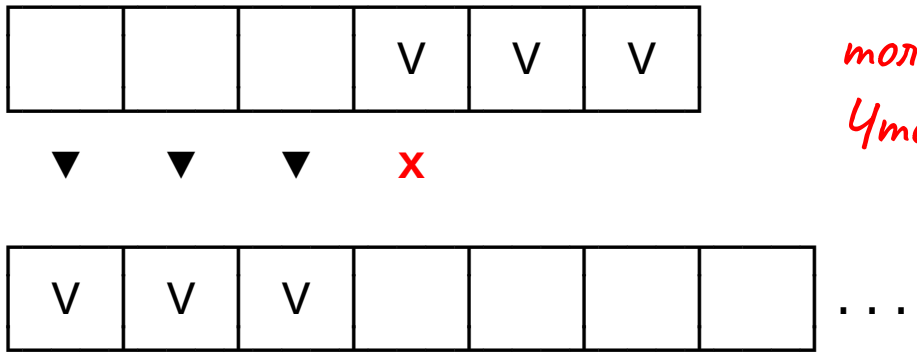
См. реализацию

https://en.cppreference.com/w/cpp/memory/uninitialized_copy

Как быть с исключениями в move-конструкторе?

```
Vector::push_back(const T& other) {
  if (size() == capacity()) {
    T* new_buf = operator new(sizeof(T) * size() * 2);
    for (std::size_t i = 0; i < size_; ++i) {
      new (new_buf + i) T(std::move(items_[i]));
    }
    ...
  }
}
```

Иллюстрация перемещения



Смогли переместить только часть. Что делать?

Q&A