Указатели и ссылки

Разбор задачи

В предыдущей лекции

1. Массивы и адресная арифметика ➡ **STL**
2. Передача тяжелых значений в функции ➡ **const&**
3. Out и in-out параметры функций ➡ **return, &**
4. Семантика отсутствия объекта ➡ **std::optional**
5. **Создание динамических структур данных**

Задача

Дан контейнер строк. Сгруппировать строки по длине.

Пример.

Вход:

```
['b', 'abcd', 'abc', 'cba', 'a']
```

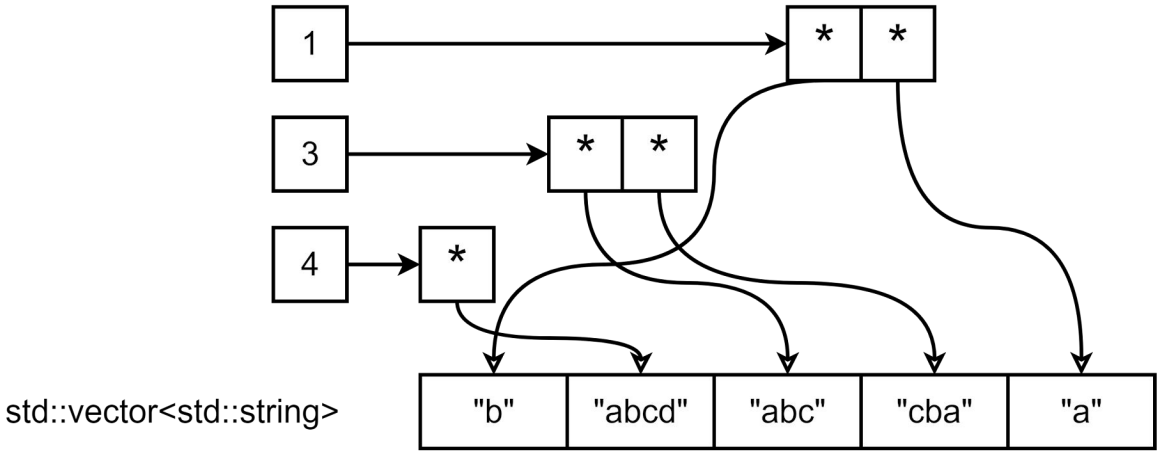
Выход:

```
1: ['b', 'a']
3: ['abc', 'cba']
4: ['abcd']
```

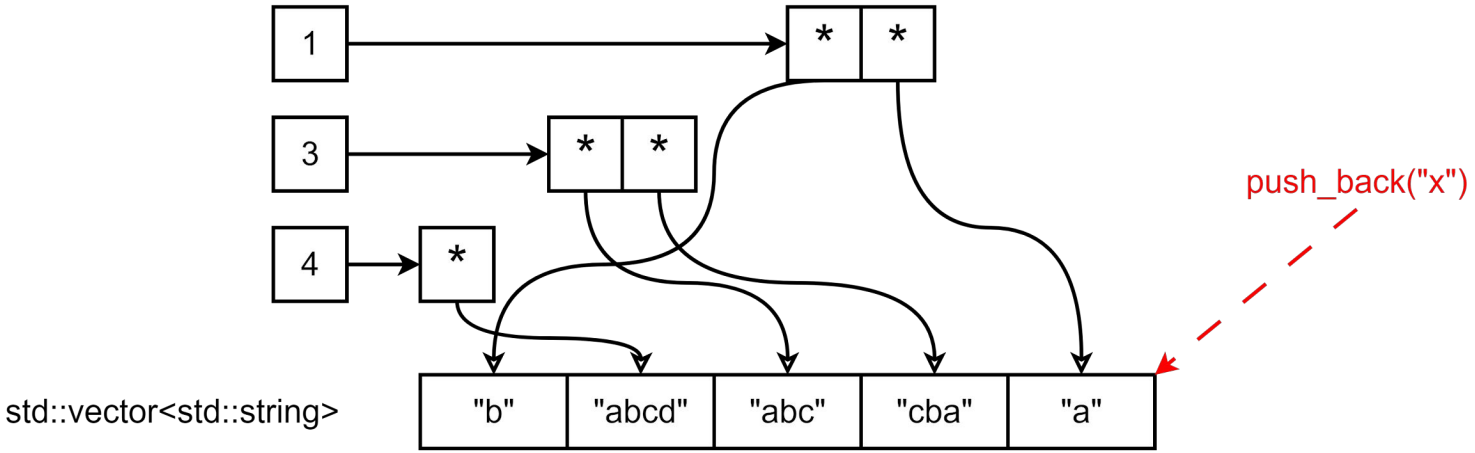
Замечания к реализации

- Сохранить исходный контейнер строк
- Избежать создания копий строк

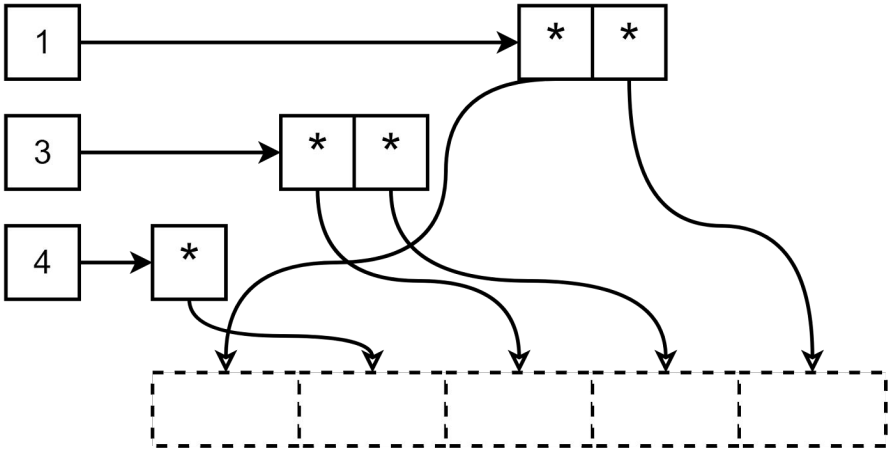
std::unordered_map<std::size_t, std::vector<const std::string*>>



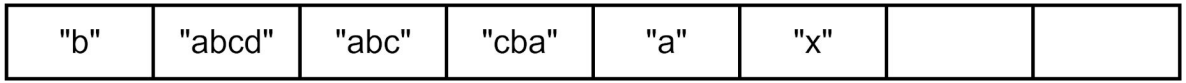
std::unordered_map<std::size_t, std::vector<const std::string*>>



std::unordered_map<std::size_t, std::vector<const std::string*>>



std::vector<std::string>



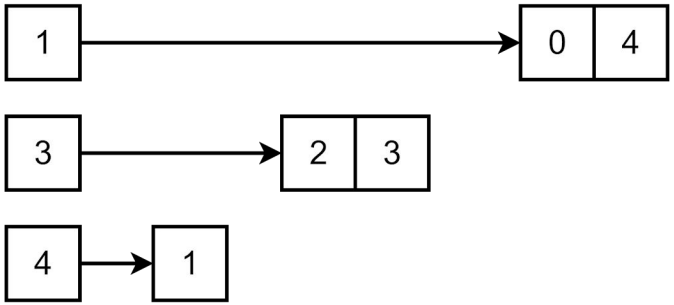
https://godbolt.org/z/dT4s18zsP

Недостатки решения

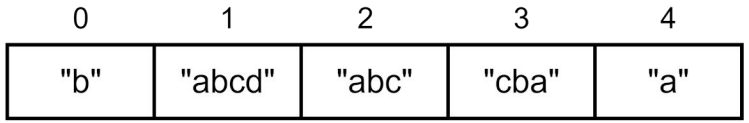
```
std::vector<std::string>
+
std::unordered_map<
    std::size_t,
    std::vector<const std::string*>>
```

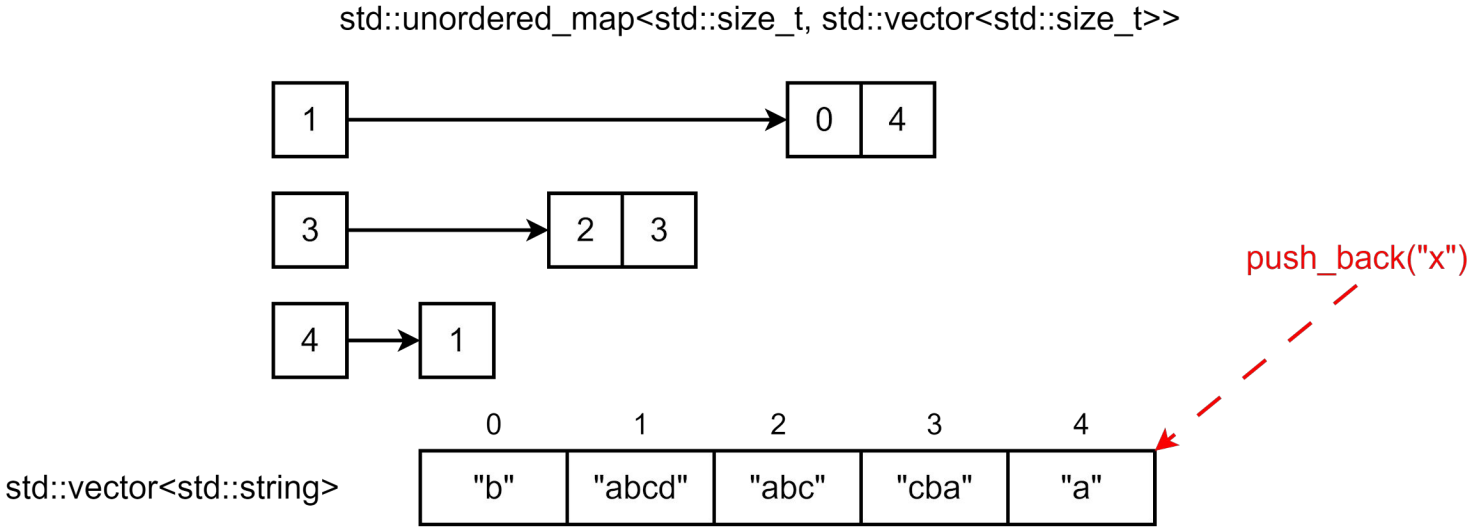
- Указатели могут провиснуть
- Легко нарушить согласованность структур

std::unordered_map<std::size_t, std::vector<std::size_t>>

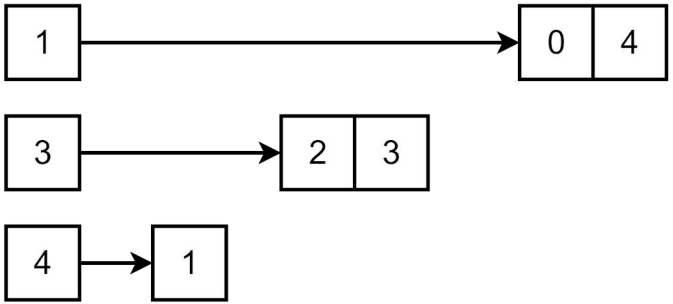


std::vector<std::string>

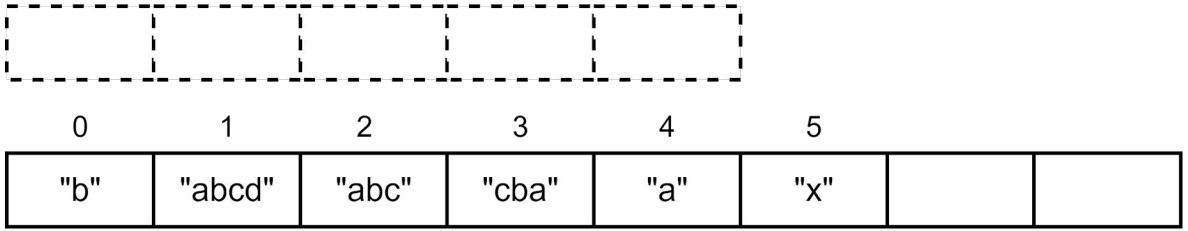



std::unordered_map<std::size_t, std::vector<std::size_t>>
1
0
4
3
2
3
4
1
push_back("x")
0
1
2
3
4
std::vector<std::string>
"b"
"abcd"
"abc"
"cba"
"a"

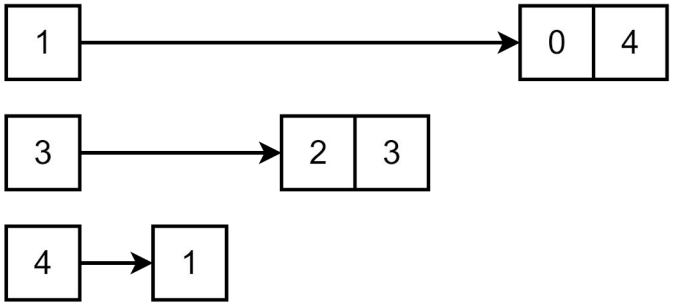
std::unordered_map<std::size_t, std::vector<std::size_t>>



std::vector<std::string>

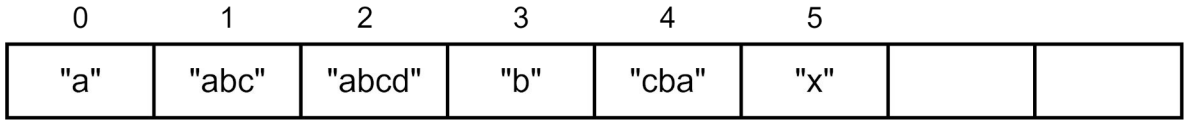


std::unordered_map<std::size_t, std::vector<std::size_t>>



std::sort

std::vector<std::string>



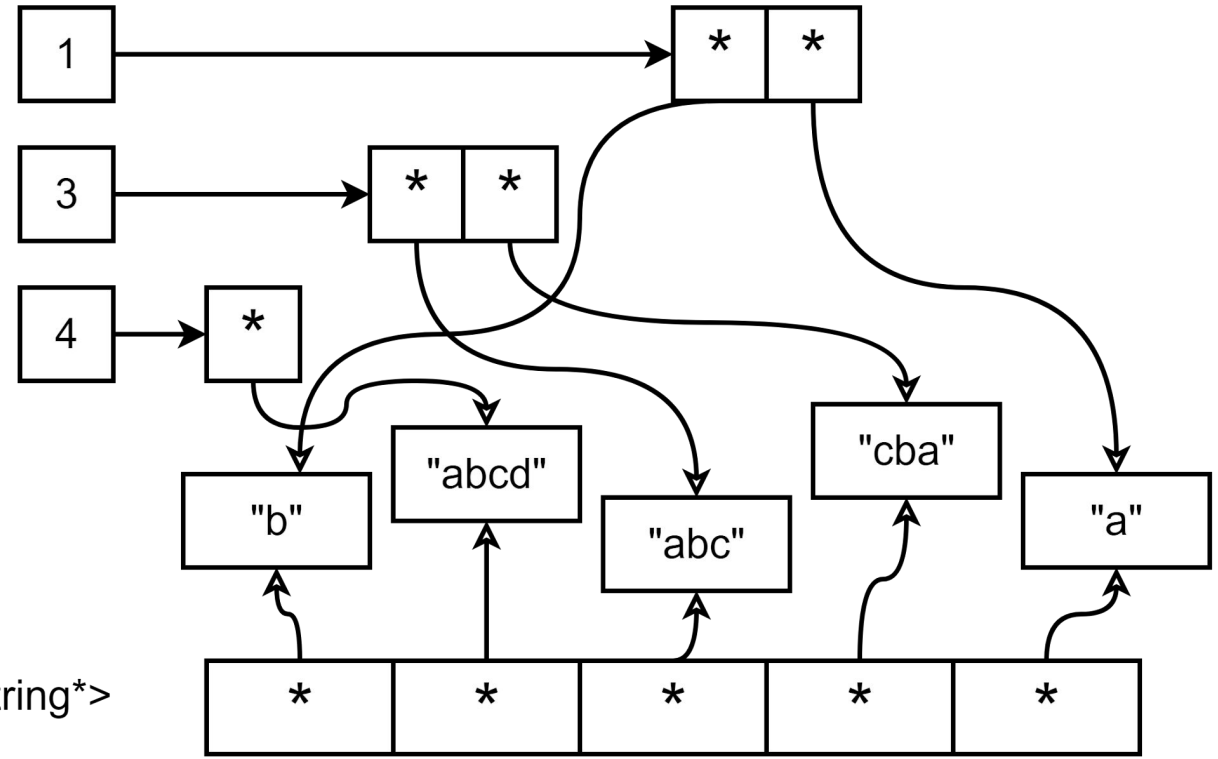
https://godbolt.org/z/YE9oW9ETo

Недостатки решения

```
std::vector<std::string>
+
std::unordered_map<
    std::size_t,
    std::vector<std::size_t>>
```

- Индексы могут инвалидироваться
- Легко нарушить согласованность структур

std::unordered_map<std::size_t, std::vector<const std::string*>>



https://godbolt.org/z/7erxe68r8

Недостатки решения

```
std::vector<std::unique_ptr<std::string>>
+
std::unordered_map<
    std::size_t,
    std::vector<const std::string*>>
```

Аллокации на каждый объект => низкая производительность.

Общая проблема

- Две структуры данных, которые должны быть согласованы
- Нет механизма контроля согласованности

Конкретные классы

Пару лекций назад

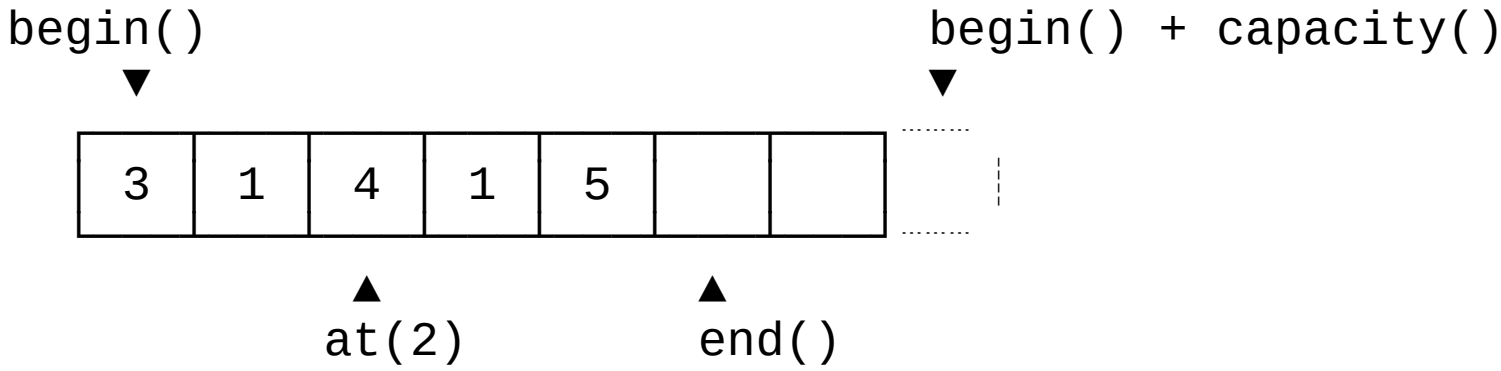
```
struct Point {
  double x_, y_;
  void move(Vec2d vec);
};
```

```
void point_move(struct Point* point, Vec2d vec);
```

Блиц-опрос

1. Зачем нужны классы?
   (Какой критерий выбора между классом и структурой?)
2. Зачем нужен конструктор?
3. Зачем нужна инкапсуляция?

Устройство std::vector



Покритикуйте этот код

```
struct IntVector {
  int* begin_;
  int* end_;
  int* capacity_;
};

size_t vector_size(const IntVector* v) {
  return ???;
}
```

Покритикуйте этот код

```
struct IntVector {
  int* begin_;
  int* end_;
  int* capacity_;
};

size_t vector_size(const IntVector* v) {
  return v->end_ - v->begin_;
}
```

Иллюстрация проблемы

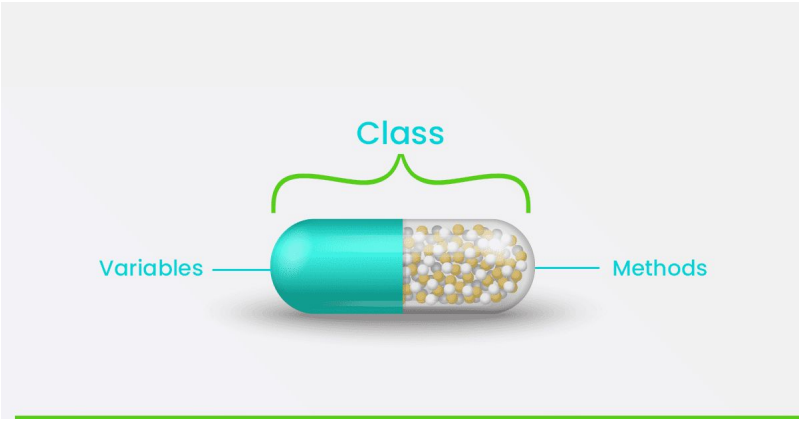
Можно ли защититься от написания такого кода?
Как это возможно в языке C?

```
IntVector* iv = vector_new(/*size=*/42);
iv->end_ = iv->begin_ - 1;
```

Инкапсуляция —

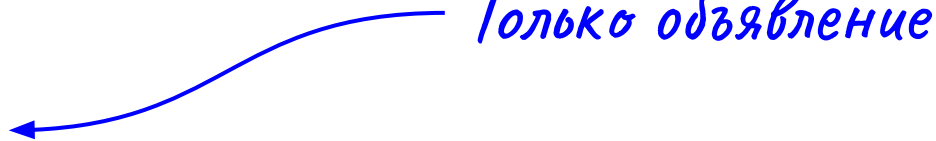
механизм языка, который:

1. Связывает данные и методы их обработки
2. Позволяет ограничивать доступ к компонентам класса



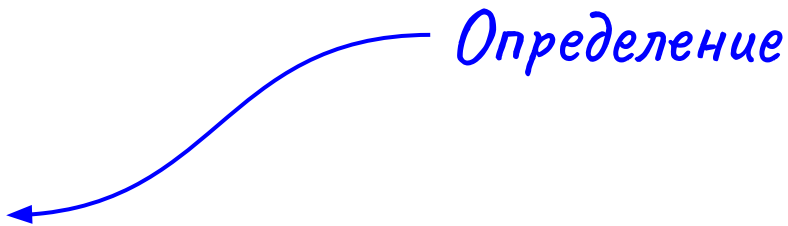
Инкапсуляция в языке C

```
// IntVector.h

struct IntVector;

IntVector* vector_new(size_t size);
void vector_push_back(IntVector* v);
size_t vector_size(const IntVector* v);
```



Инкапсуляция в языке C

```
// IntVector.c
#include “IntVector.h”

struct IntVector { ... };

size_t vector_size(const IntVector* v) {
  return v->end_ - v->begin_;
}
```



Инкапсуляция в языке C

```
// IntVector.h

struct IntVector;

IntVector* vector_new(size_t size);
void vector_push_back(IntVector* v);
size_t vector_size(const IntVector* v);
```

Покритикуйте этот подход

Инкапсуляция в языке C

```
// IntVector.c

struct IntVector { ... };

IntVector* vector_new(size_t size) {
  IntVector* iv = malloc(...);
  ...
  return iv;
}
```

Вынужденная аллокация

Инкапсуляция в C++

```
class IntVector {
 public:
  void push_back(int value);
  std::size_t size() const;

 private:
  int* begin_;
  int* end_;
  int* capacity_;
};
```

Инкапсуляция в C++

Что мешает написать такой код?

```
IntVector v(42);
std::memset(&v, 0, sizeof(v));
```


https://godbolt.org/z/fa3qTa9PT

Инкапсуляция в C++

Что мешает написать такой код?

```
IntVector v(42);
std::memset(&v, 0, sizeof(v));
```

Мешает совесть

https://godbolt.org/z/fa3qTa9PT

Инкапсуляция в C++

- В C++ линейная модель памяти,
  поэтому модификаторы доступа не могут защитить данные.
- Сокрытие в C++ защищает **имена**.

```
error: 'int* IntVector::begin_' is private
       within this context
```

Пример конструктора

```
class IntVector {
 public:
  IntVector(std::size_t size) {
    begin_ = new int[size]();
    end_ = begin_ + size;
    capacity_ = end_;
  }
  ...
};
```

Написано не очень хорошо, доработаем позже

Еще раз

Класс

Конструктор

Инкапсуляция

Инвариант

Инвариант класса — это множество условий, сохраняющих свою истинность на протяжении всего времени жизни экземпляра класса.

Инвариант определяет внутренне непротиворечивое (согласованное) состояние объекта.

Примеры инвариантов

- vector
  - begin ≤ end ≤ capacity
- string
  - begin ≤ end ≤ capacity
  - c_str() — null-terminated string
- Rational (простая дробь, например 3/4)
  - ???

Примеры инвариантов

- vector
  - begin ≤ end ≤ capacity
- string
  - begin ≤ end ≤ capacity
  - c_str() — null-terminated string
- Rational (простая дробь, например 3/4)
  - знаменатель != 0
  - Дробь несократима

Объединяем механизмы

Класс создает контекст для инварианта.

Конструктор устанавливает инвариант.

Инкапсуляция помогает поддерживать инвариант.

Рекомендации

C.ctor: Constructors

C.40: Define a constructor if a class has an invariant

C.41: A constructor should create a fully initialized object

Класс Point

```
struct Point {
  double x_;
  double y_;
};
```

Какой инвариант у этого типа?

Класс Point

```
struct Point {
  double x_;
  double y_;
};
```

Инварианта может не быть.

Тогда класс не нужен, достаточно структуры.

Константный интерфейс

```
class Point {
 public:
  Point(double x, double y);

  double x() const;
  double y() const;

 private:
  double x_, y_;
};
```

Поля объекта нельзя менять независимо

https://godbolt.org/z/z1T7sGK8K

В чем разница?

```
class Point {
 public:
  Point(double x, double y);

  double x() const;
  double y() const;

 private:
  double x_, y_;
};
```

```
struct ROPoint {
  const double x_ = 0;
  const double y_ = 0;
};
```


https://godbolt.org/z/rWEhcW8Wq

Константные поля удаляют operator=

```
Point p1(1, 2), p2(3, 4);
p1 = p2;  // ok

ROPoint rop1(1, 2), rop2(3, 4);
rop1 = rop2;  // error: use of deleted operator=
```

C.12: Don’t make data members const or references in a copyable or movable type

https://godbolt.org/z/8G6MqGfhd

Причины создания классов

- Моделирование объектов реального мира
- Моделирование абстрактных объектов
- Сокрытие деталей реализации
- Упрощение передачи параметров в методы
- **Упаковка родственных операций**
- и др.

Но:

- *Книга написана в контексте Java*
- *Содержит Java-специфичные идиомы*

«Совершенный код», 6.4

Java-specific пример

```
class Collections {
  public static
  int binarySearch(List<...> list, T key)

  public static
  void shuffle(List<...> list, Random rnd)

  public static
  void copy(List<...> dest, List<...> src)
}
```


java.util.Collections

Группировка строк

Создадим класс Index. Инварианты:

- Владеет контейнером строк и отображением len -> [word*].
- Контейнер и отображение неизменяемы.

Конструктор устанавливает инвариант

```
class Index {
 public:
  explicit Index(Words words)
      : words_(std::move(words)),
        len_to_word_ptrs_(group_by_len(words_)) {}
  ...
 private:
  Words words_;
  LenToWordPtrs len_to_word_ptrs_;
};
```

Инкапсуляция поддерживает инвариант

```
class Index {
 public:
  ...
  const Words& words() const { return words_; }

  const LenToWordPtrs& len_to_word_ptrs() const {
    return len_to_word_ptrs_;
  }
  ...
};
```


https://godbolt.org/z/s6oaYrhzK

Вспомним о синонимах типов

```
using DocumentId = std::size_t;
using TermPosition = std::size_t;

void f(DocumentId doc_id, TermPosition tp);

DocumentId doc_id = ...;
TermPosition tp = ...;

f(doc_id, tp); // OK
f(tp, doc_id); // OK
```

Идиома Strong Typedef

```
struct X {
  explicit X(std::size_t value)
    : value_(value) {}

  std::size_t value_;
};
```

Вместо X – имя вашего типа

Где определить метод find?

```
class Vector {
 public:
  T* find(T value) const;
};

T find(Iterator begin, Iterator end);
```

1

2

Подход STL

- InputIt find(InputIt first, InputIt last, const T& value)

Внешняя функция подходит для:

– Массивов

– std::vector, std::list

– std::string

Все просто, так?

Подход STL

- InputIt find(InputIt first, InputIt last, const T& value)

Да, но:

- std::map::find
- std::set::find
- std::unordered_map::find
- std::unordered_set::find

Подход STL

- InputIt find(InputIt first, InputIt last, const T& value)

Да, но:

- std::map::find
- std::set::find
- std::unordered_map::find
- std::unordered_set::find

Ладно, но:

- std::string::find

Пример «как не надо»

```
class QuadraticEquationSolver {
 public:
  void set_a(double a); // + set_b, set_c
  void solve();

  int root_count() const;
  double x1() const;
  double x2() const;
 ...
};
```


https://godbolt.org/z/fEYeEfzTn

Пример «как не надо»

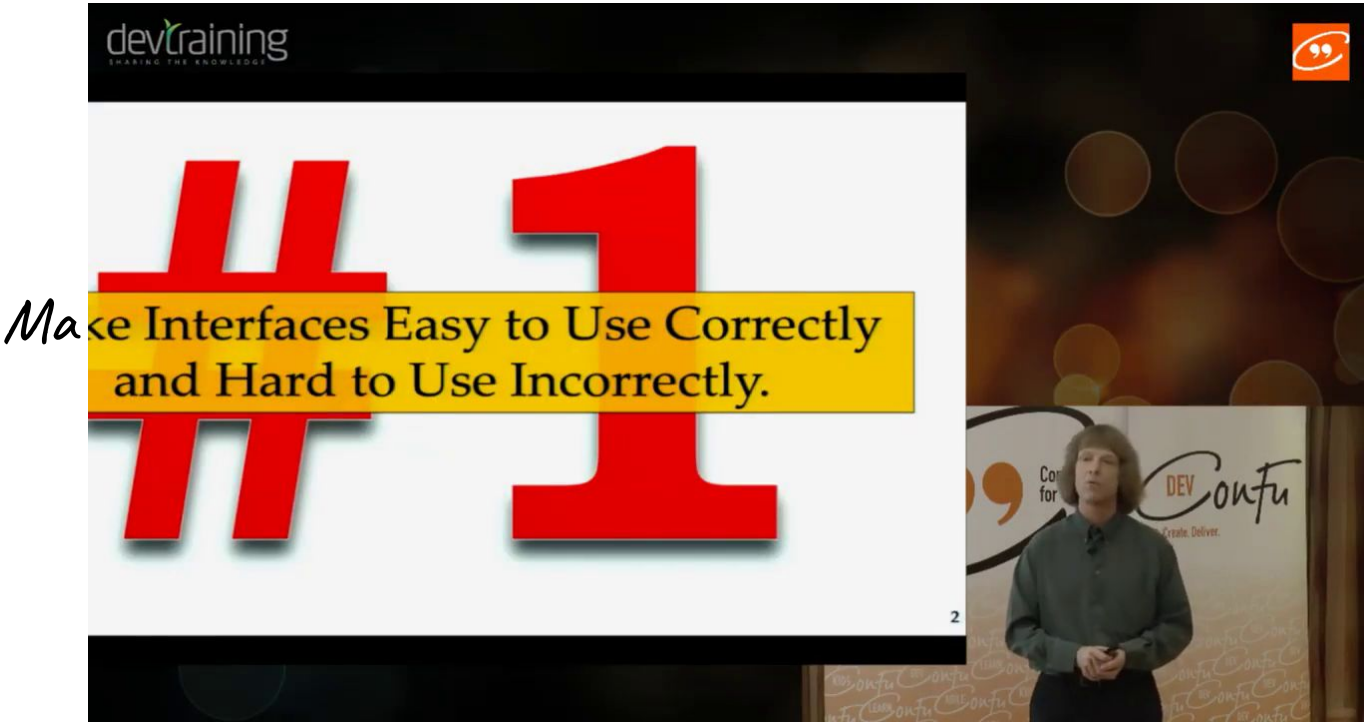
```
QuadraticEquationSolver solver;

solver.set_a(3);
solver.set_b(-14);
solver.set_c(-5);

solver.solve();

fmt::print("x1 = {} x2 = {}\n",
           solver.x1(), solver.x2());
```

Все это время объект в неконсистентном состоянии

Архитектурные проблемы такого солвера

1. У класса нет четкого инварианта.
2. Класс предполагает определенный порядок вызова методов, но не форсирует его.
3. Согласованность внутреннего состояния легко нарушить (вызов любого set-метода после solve).
4. Поля класса выполняют функции глобальных переменных.

The Most Important Design Guideline



https://www.bilibili.com/video/BV15g4y1878e/

Организация исходников

В .hpp — объявления функций и определения классов.

В .cpp — определения функций и методов классов.

У сущности может быть:

- Сколько угодно объявлений
- Только одно* определение

** — все сложно*

Header

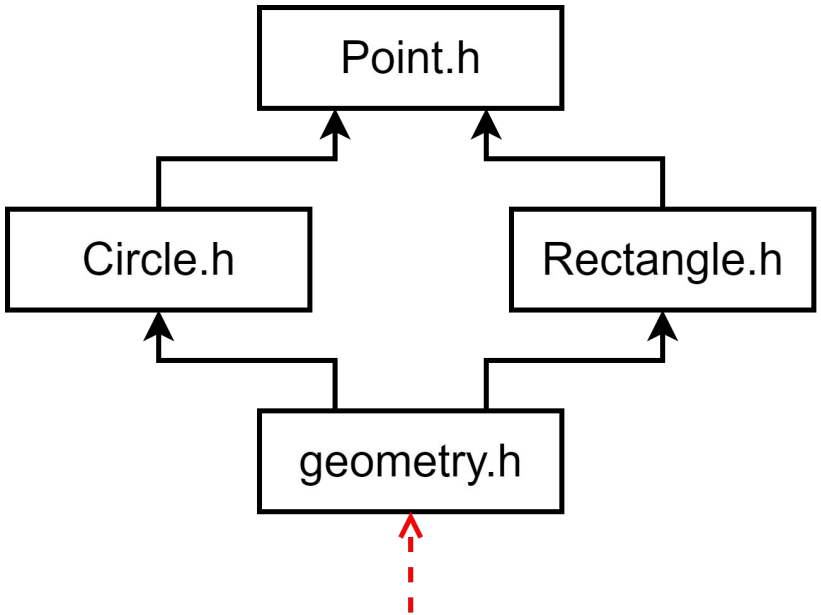
```
#pragma once

namespace geometry {

struct Point {
  double x_ = 0, y_ = 0;
  void move(Vec2d v);
};

}
```

#pragma once

Защищает от повторного включения.

Если не использовать, то:

```
#include <geometry/Point.h>
#include <geometry/Point.h> // Error: redefinition
```



Двойное включение Point.h

pragma once vs include guard

```
#ifndef GEOMETRY_POINT_H
#define GEOMETRY_POINT_H

#endif // GEOMETRY_POINT_H
```

pragma once — нет в стандарте, но поддерживается gcc/clang/MSVC

Implementation

```
#include <geometry/Point.h>

namespace geometry {

void Point::move(Vec2d v) {
  // …
}

}
```

Q&A