Наследование

Полиморфизм подтипов. Динамическое связывание.

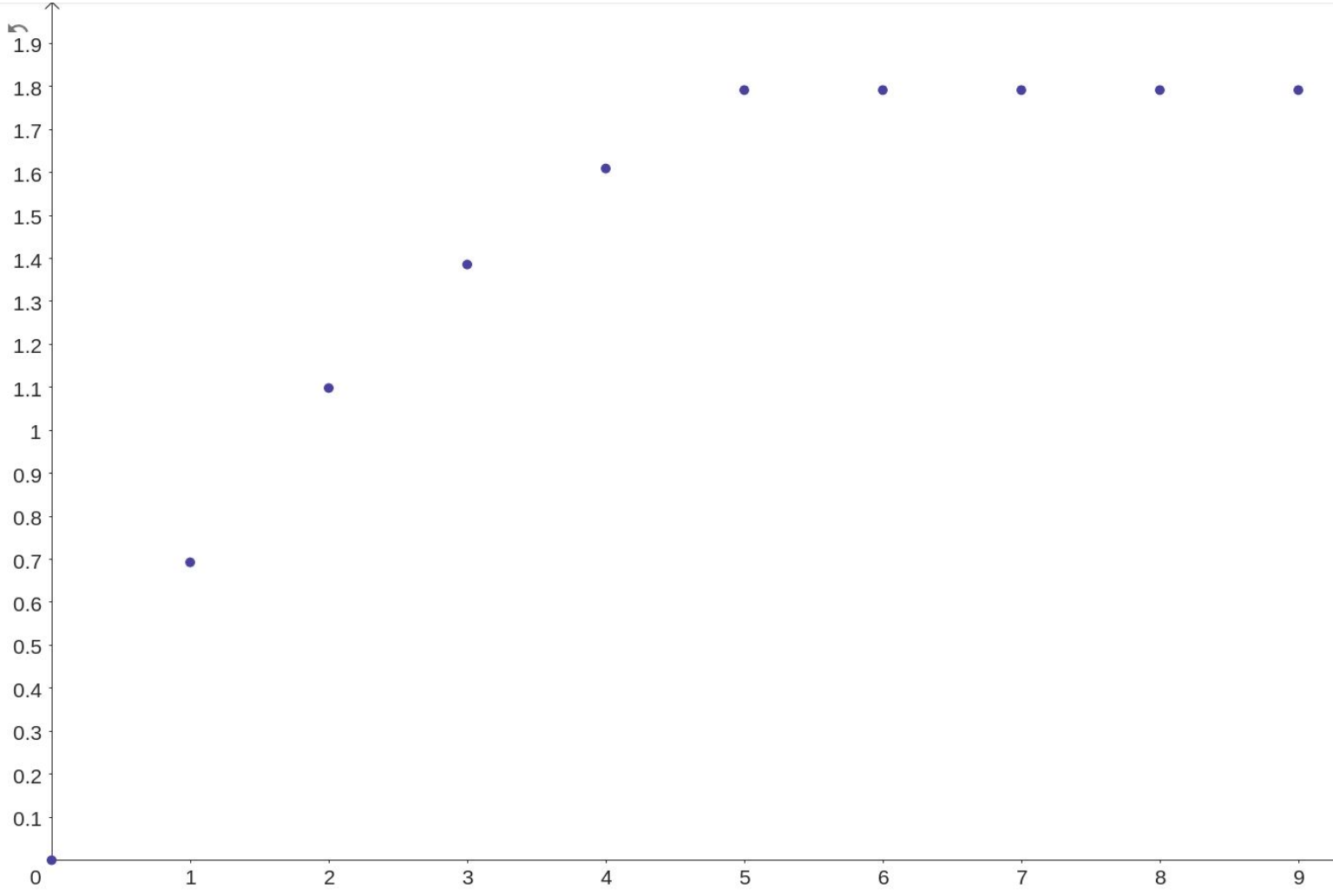
Задача

Пусть задана последовательность значений v = [a, b, c, ...].
Реализовать функцию:

```
f(x) = v[x],          x < len(v)
       v[len(v)-1], x >= len(v)
```

$x \in N$, $f(x) \in R$
v определяется на этапе выполнения программы.

x	y_1
0	0
1	0.6931
2	1.0986
3	1.386
4	1.6094
5	1.7918
6	1.7918
7	1.7918
8	1.7918
9	1.7918
10	1.7918



Вариант 1: внешняя функция

```
double get_or_last(const std::vector<double>& v,
                   std::size_t i) {
  return i < v.size() ? v[i] : v.back();
}
```

✓ Просто

✗ Слабая связность интерфейса

https://godbolt.org/z/rhsMf5x3P

Вариант 2: композиция

```
struct VectorWrapper {
  double get_or_last(std::size_t i) const {
    return i < v_.size() ? v_[i] : v_.back();
  }
  std::vector<double> v_;
};
```

✓ Более сильная связность интерфейса

✗ Усложняется доступ к вектору

https://godbolt.org/z/b8sjYjj4G

Вариант 3: наследование

```
class ExtendedVector : public std::vector<double> {
 public:

};
```

Благодаря публичному наследованию
через класс ExtendedVector
доступны все публичные методы базового класса

Вариант 3: наследование

```
class ExtendedVector : public std::vector<double> {
 public:
  using std::vector<double>::vector;



};
```

Inheriting constructor

Вариант 3: наследование

```
class ExtendedVector : public std::vector<double> {
 public:


  double operator[](std::size_t i) const {
```

Методы класса-наследника скрывают методы базового класса с теми же именами

```
};
```

Вариант 3: наследование

```
class ExtendedVector : public std::vector<double> {
 public:
  using std::vector<double>::vector;

  double operator[](std::size_t i) const {
    return i < size()
        ? std::vector<double>::operator[](i)
        : back();
  }
};
```


https://godbolt.org/z/a76GTr7W5

Резюме

- ✓ Сильная связность интерфейса
- ✓ Не требуется специальный доступ к вектору
- ✗ Есть множество неочевиных проблем, о них позже

Какое решение лучше — зависит от ситуации

Смысл публичного наследования

Наследование класса B от класса A корректно, если оно выражает одновременно два отношения классов:

- Класс B расширяет класс A (**extends**)
- Класс B является частным случаем класса A (**is a**)

Подводные грабли такого приема

- Upcast
- Сокрытие методов базового класса

Upcast

```
void f(const std::vector<double>& v) {
  std::cout << v[1000] << '\n';
}

ExtendedVector v = {...};
f(v);
```

Оператор [] какого класса будет вызван в функции f?

https://godbolt.org/z/b79ardezn

Сокрытие методов базового класса

```
class Gadget : public Widget {
 public:
  // using Widget::f;

  // Скрывает все Widget::f, если нет using Widget::f
  void f(int) {
    std::cout << "Gadget::f(int)\n";
  }
};
```


https://godbolt.org/z/YhE3ra5Ka

Порядок конструирования?

```
struct A {
  A();
  T t_;
  U u_;
};

struct B : A {
  B();
  V v_;
  W w_;
};
```

```
B obj;
```

Порядок конструирования?

```
struct A {
  A();
  T t_;
  U u_;
};

struct B : A {
  B();
  V v_;
  W w_;
};
```

```
B obj;

Порядок вызова:
T
```

Порядок конструирования?

```
struct A {
  A();
  T t_;
  U u_;
};

struct B : A {
  B();
  V v_;
  W w_;
};
```

```
B obj;

Порядок вызова:
T
U
```

Порядок конструирования?

```
struct A {
  A();
  T t_;
  U u_;
};

struct B : A {
  B();
  V v_;
  W w_;
};
```

```
B obj;

Порядок вызова:
T
U
A
```

Порядок конструирования?

```
struct A {
  A();
  T t_;
  U u_;
};

struct B : A {
  B();
  V v_;
  W w_;
};
```

```
B obj;

Порядок вызова:
T
U
A
V
```

Порядок конструирования?

```
struct A {
  A();
  T t_;
  U u_;
};

struct B : A {
  B();
  V v_;
  W w_;
};
```

```
B obj;

Порядок вызова:
T
U
A
V
W
```

Порядок конструирования?

```
struct A {
  A();
  T t_;
  U u_;
};

struct B : A {
  B();
  V v_;
  W w_;
};
```

```
B obj;

Порядок вызова:
T
U
A
V
W
B
```

Резюме

- Мы применили наследование как механизм повторного использования кода.
- Гораздо чаще вы будете встречать другое применение.

Постановка задачи

Даны типы геометрических фигур (Circle, Triangle, Rectangle).

Задача:

- Реализовать унифицированную обработку их экземпляров:
  - Хранение
  - Общие операции

Фигуры

```
struct Triangle { Point a_, b_, c_; };

struct Circle {
  Point center_;
  double radius_;
};

struct Rectangle {
  Point top_left_;
  Point bottom_right_;
};
```

Хотим создать общий массив фигур

Попытка 1: Union

```
union Shape {
  Circle circle_;
  Triangle triangle_;
  Rectangle rectangle_;
};
```

Попытка 1: Union

```
union Shape {
  Circle circle_;
  Triangle triangle_;
  Rectangle rectangle_;
};
```

Как узнать, какой элемент активен?

Попытка 1.2: Tagged Union

```
enum class ShapeType {Circle, Triangle, Rectangle};

struct Shape {
  union Data {
    Circle circle_;
    Triangle triangle_;
    Rectangle rectangle_;
  } data_;
  ShapeType shape_type_;
};
```

Попытка 1.3: Tagged Union (union-like class)

```
enum class ShapeType {Circle, Triangle, Rectangle};

struct Shape {
  union {
    Circle circle_;
    Triangle triangle_;
    Rectangle rectangle_;
  };
  ShapeType shape_type_;
};
```

Попытка 1.3: реализация полиморфизма

```
void print_shape(const Shape& s) {
  switch (s.shape_type_) {
  case ShapeType::Circle:
    return print_circle(s.circle_);

  case ShapeType::Triangle:
    return print_triangle(s.triangle_);
  }
}
```

Проблемы решения

1. Ветвления по типу в каждом методе
   Компилятор может проверить полное покрытие enum'а switch'ем, но не может проанализировать if'ы.
2. Как следствие, сложно добавить новый тип внутри библиотеки.
3. Невозможно добавить новый тип на стороне клиента.
4. Union в C++ гораздо сложнее union в C. Об этом в другой раз.

Если очень хочется…

Если нужен полиморфизм по фиксированному множеству типов, воспользуйтесь std::variant:

```
std::variant<Circle, Triangle, Rectangle> shape_;
```

Виртуальный метод

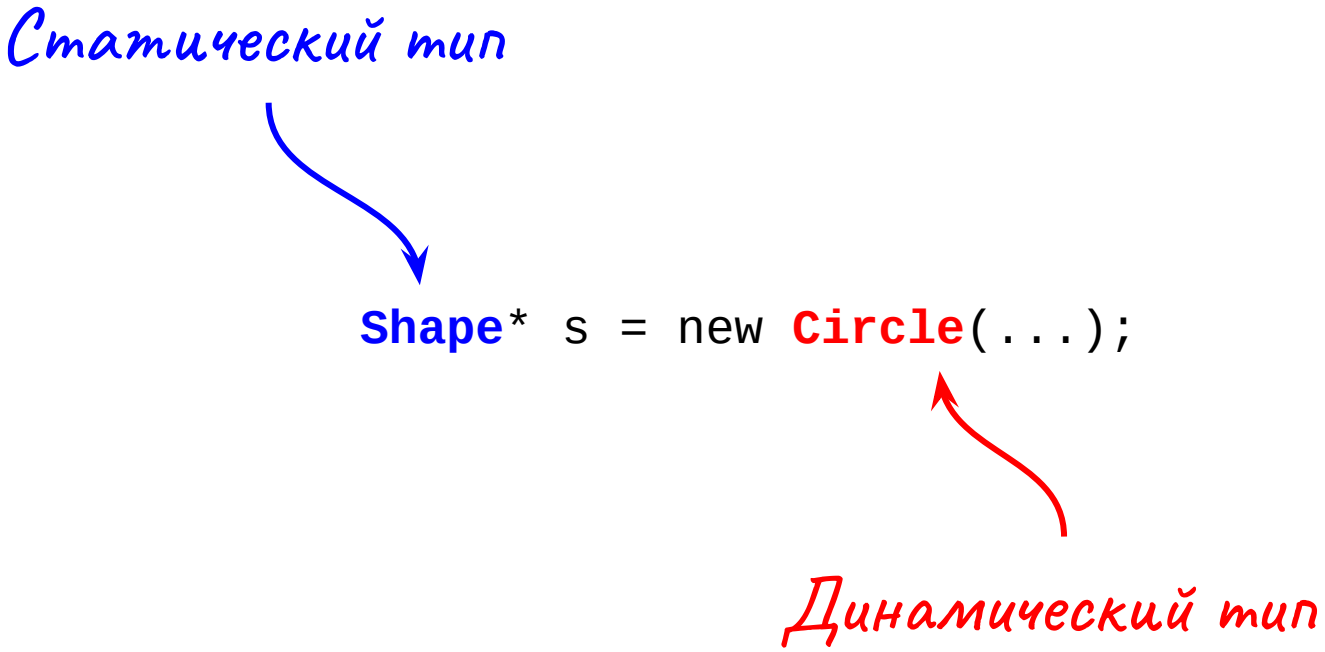
```
class Shape {
 public:
  virtual ~Shape() = default;
  virtual void print() const = 0;
};

class Circle : public Shape {
 public:
  void print() const override;
  ...
};
```

Разберем это решение

Разбор решения

1. **Виртуальный метод**
2. Чистый виртуальный метод
3. Переопределение (override) виртуального метода
4. Виртуальный деструктор

Статический и динамический тип



Статический и динамический тип

Статический тип — тип, известный на этапе компиляции из определения объекта или из выражения.

Динамический тип — тип объекта в памяти, на который ссылается ссылка или указывает указатель. В общем случае неизвестен на этапе компиляции.

Lippman, p. 650

Виртуальный метод

Виртуальный метод — это метод, адрес которого определяется на этапе выполнения программы в соответствии с динамическим типом объекта

Пример

```
struct Base {
  void f() const;
  virtual void g() const;
};

struct Derived : Base {
  void f() const;
  void g() const;
};
```

```
void test(const Base& obj) {
  obj.f();
  obj.g();
}

Derived d;
test(d);
```


https://godbolt.org/z/Yxo87PMnG

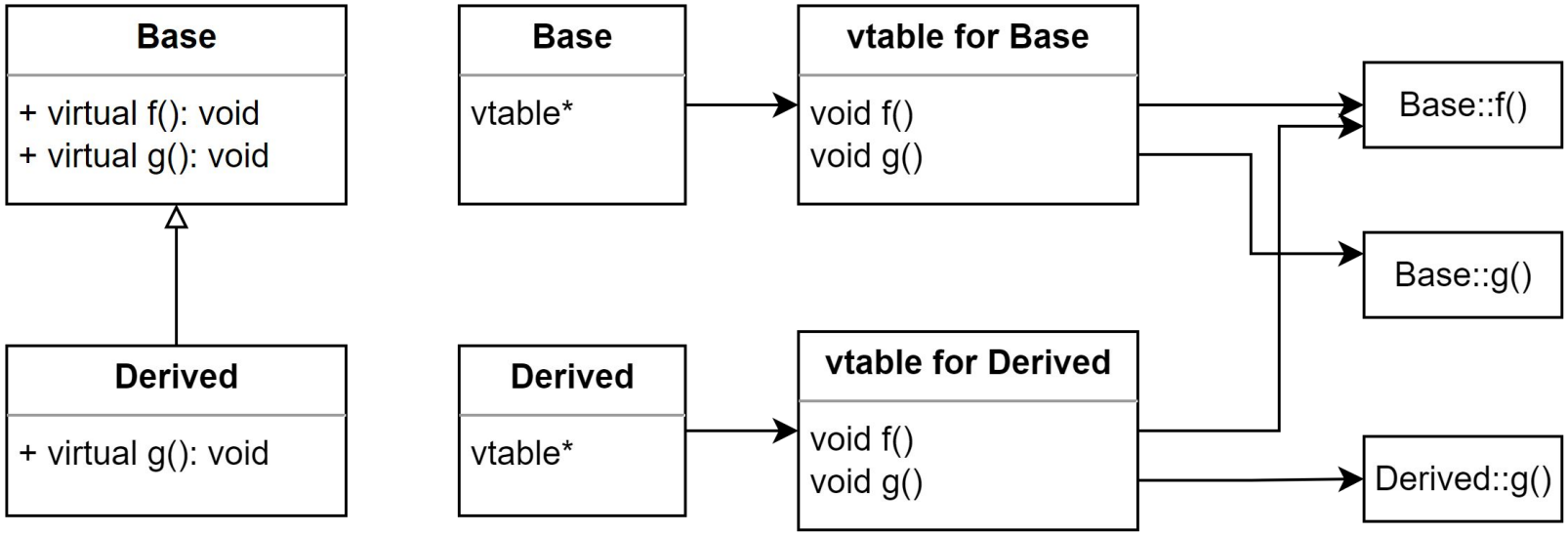
Пример

```
struct Base {
  virtual void f() const;
  virtual void g() const;
};

struct Derived : Base {
  void g() const;
};
```

```
void test(const Base& obj) {
  obj.f();
  obj.g();
}

Derived d;
test(d);
```


https://godbolt.org/z/sn9E5fjoG

vtable


Base
+ virtual f(): void
+ virtual g(): void
Derived
+ virtual g(): void
Base
vtable*
Derived
vtable*
vtable for Base
void f()
void g()
vtable for Derived
void f()
void g()
Base::f()
Base::g()
Derived::g()

Разбор решения

1. Виртуальный метод
2. **Чистый виртуальный метод**
3. Переопределение (override) виртуального метода
4. Виртуальный деструктор

Что должны делать эти методы?

```
struct Shape {
  virtual double area() const {
    return ???;
  }

  virtual double perimeter() const {
    throw NotImplementedError();
  }

  virtual void print() const {
    std::cout << "???";
  }
};
```

Сделаем их чистыми виртуальными (pure virtual)

```
struct Shape {
  virtual double area() const = 0;

  virtual double perimeter() const = 0;

  virtual void print() const = 0;
};
```

Как вы думаете, что означает '= 0'?

Абстрактный класс

Класс называется абстрактным, если содержит хотя бы один pure virtual метод.

Экземпляр абстрактного класса нельзя создать.

Класс, содержащий только pure virtual методы, часто называют интерфейсом.

Разбор решения

1. Виртуальный метод
2. Чистый виртуальный метод
3. **Переопределение (override) виртуального метода**
4. Виртуальный деструктор

Какой метод будет вызван?

```
struct Base {
  virtual void print() const;
};

struct Derived : Base {
  void print();
};
```

```
Base* obj = new Derived();
obj->print();
```


https://godbolt.org/z/dc9PrGrEe

Какой метод будет вызван?

```
struct Base {
  virtual void print() const;
};

struct Derived : Base {
  void print() override;
};
```

error: 'void Derived::print()' marked 'override', but does not override

Какой метод будет вызван?

```
struct Base {
  virtual void print() const;
};

struct Derived : Base {
  void print() const override;
};

Base* obj = new Derived();
obj->print(); // Call Derived::print
```

Разбор решения

1. Виртуальный метод
2. Чистый виртуальный метод
3. Переопределение (override) виртуального метода
4. **Виртуальный деструктор**

Поведение программы?

```
struct Shape {
  virtual void print() = 0;
};

Shape* s = new Circle();
delete s;
```

Поведение программы?

```
struct Shape {
  virtual void print() = 0;
};

Shape* s = new Circle();
delete s;
```

Undefined

[expr.delete]

Поведение программы?

```
struct Shape {
  virtual ~Shape() = default;
  virtual void print() = 0;
};

Shape* s = new Circle();
delete s;
```

Ok

[expr.delete]

Use smart pointers, Luke!

```
struct Shape {
  virtual ~Shape() = default;
  virtual void print() = 0;
};

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

А лучше так

Виртуальный метод

```
class Shape {
 public:
  virtual ~Shape() = default;
  virtual void print() const = 0;
};

class Circle : public Shape {
 public:
  void print() const override;
  ...
};
```

Теперь все должно быть ясно

Когда создавать виртуальный деструктор?

- Пусть мы пишем класс Widget.
- Нужен ли ему виртуальный деструктор?

Когда создавать виртуальный деструктор?

- Пусть мы пишем класс Widget.
- Нужен ли ему виртуальный деструктор?

Конфликт:

- Создание вирт деструктора в абсолютно каждом классе расточительно.
- Отсутствие деструктора позволяет некорректное полиморфное использование.

Ограничение наследования

```
class Widget final {};
```

Упрощенное правило. Класс должен:

- Либо содержать виртуальный деструктор
- Либо содержать protected невиртуальный деструктор
- Либо быть final

Есть еще пара правил, но о них в другой раз.

Circle-ellipse problem

Пусть в нашей иерархии есть классы:

- Circle
- Ellipse

Какой должен быть базовым?

Circle <-- Ellipse

```
struct Circle {
  virtual double area();
  Point center_;
  double radius_;
};

struct Ellipse
   : public Circle {
  double area() override;
  double ry_;
};
```

```
void f(const Circle& c) {
  c.area(); // Oops…
}


f(Ellipse{{0, 0}, 1, 2});
```

Ellipse <-- Circle

```
struct Ellipse {
  void stretch_x(double k) { rx_ *= k; }
  Point center_;
  double rx_, ry_;
};

struct Circle : public Ellipse {
  Circle(Point center, double radius)
    : Ellipse{center, radius, radius} {}
};

Circle c({0, 0}, 1);
c.stretch_x(2); // Oops...
```

Circle-ellipse problem

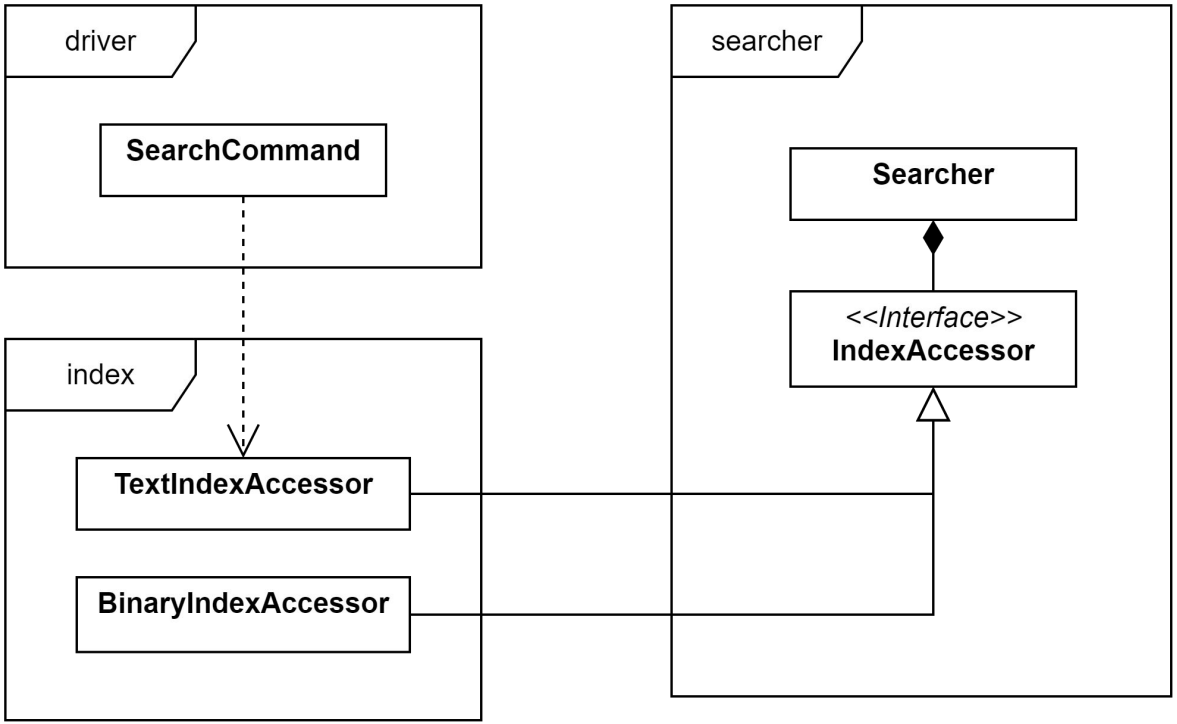
- Circle является частным случаем Ellipse
- Ellipse расширяет Circle

Вывод: они не должны быть объединены наследованием.

Пример 1

https://csc-cpp.readthedocs.io/ru/2023/s1/5-books.html

Пример 1: альтернатива



Пример 2

cmake presets:

https://cmake.org/cmake/help/latest/manual/cmake-presets.7.html

https://github.com/Kitware/CMake/blob/master/Source/cmCMakePresetsGraph.h#L61

Осталось за рамками

1. Срезка (slicing)
2. RTTI, dynamic_cast
3. Клонирование/полиморфное присваивание
4. Множественное наследование
5. Виртуальное наследование, ромбовидные иерархии
6. private наследование
7. Ковариантность по типу возвращаемого значения
8. Виртуальные методы и параметры по умолчанию
9. pure virtual function call
10. Вызов виртуальных функций из конструктора
11. А еще бы почитать ассемблер

Литература

Lippman, Chapter 15, Object-oriented programming

Q&A