Умные указатели

Практическое объектно-ориентированное программирование

30.10.2024

План

- Проблема владения
- Уникальное владение
- Совместное владение
- Закольцованный мир

Владение ресурсом

- Памятью владеет тот, кто её выделяет и освобождает.

```
void foo(S*);
S *presource = new S;
foo(p);
delete p;
```

- Что может пойти не так в этом коде?

RAII: resource acquisition is initialization

• Стандарт управления ресурсом это RAII классы.

```
template <typename S> int foo(int n) {
  RAIIPointer<S> p = make_raiiptr<S>(n); // RAII
  // .... some code ....
  if (condition)
    return FAILURE; // dtor called
  // .... some code ....
  return SUCCESS; // dtor called
}
```

Нулевой вариант

• Как вы думаете почему нет такого стандартного класса?

```
template <typename T> class CopyingPointer {
  T *value;
public:
  CopyingPointer(T *value) : value(value) {}
CopyingPointer(const CopyingPointer& rhs) :
    value(new T*rhs.value)
  CopyingPointer(CopyingPointer&& rhs) : value(rhs.value) {
    rhs.value = nullptr;
}
```

Список разумных альтернатив

- Семантика значения (уникальное владение).
- Запрет копирования и перемещения (уникальное владение).
- Запрет копирования (уникальное владение).
- Подсчёт ссылок в контрольном блоке (совместное владение).
- Интрузивный подсчёт ссылок (совместное владение).

Немного о некопируемых объектах

- Допустим у нас есть объект с запрещенным копированием и присваиванием.

```
template <typename T> class ScopedPointer {
  T *value;
public:
  ScopedPointer(T *value) : value(value) {}
  ScopedPointer(const ScopedPointer& rhs) = delete;
  ScopedPointer(ScopedPointer&& rhs) = delete;
```

- В каких случаях можно его передать и вернуть по значению?

PRVALUE elision

```
void foo(ScopedPointer<int>);
foo(ScopedPointer<int>new int(42)); // OK
auto bar() {
  return ScopedPointer<int>new int(42); // OK
}
auto n = bar(); // OK
auto buz() {
  ScopedPointer<int> t{new int(42)};
  return t; // FAIL, NRVO case
}
```

Идея для unqiue_ptr

```
template <typename T> class UniquePointer {
  T *value;
public:
UniquePointer(T *value) : value(value) {}
UniquePointer(UniquePointer& rhs) = delete;
UniquePointer& operator= (const UniquePointer& rhs) = delete;
UniquePointer(UniquePointer&& rhs) : ptr(rhs.ptr) {
  rhs.ptr = nullptr;
}
UniquePointer& operator= (UniquePointer&& rhs) {
  std::swap(*this, rhs); return *this;
}
```

Немного о перемещаемых объектах

• Для перемещаемых именованных объектов возврат по значению работает как перемещение.

```
decltype(auto) buz() {
  UniquePointer<int> t{new int(42)};
  return t; // → UniquePointer<int>
}
auto n = buz(); // → UniquePointer<int>, ok
```

• Не надо лишний раз писать std::move на результат.

```
return std::move(t); // → UniquePointer<int>&&
```

Вариант проектирования

- Но что если отделить удаление в отдельный параметр шаблона?

```
template <typename T, typename Deleter = default_delete<T> >
class unique_ptr {
T *ptr_;
Deleter del_;
public:
unique_ptr(T *ptr = nullptr, Deleter del = Deleter()) :
ptr_(ptr), del_(del) {}
 unique_ptr() { del_(ptr_); }
// и так далее
```

- Как мог бы выглядеть default_delete?

Проблема unique_ptr to array

- Хотелось бы чтобы правильно отрабатывали массиво-подобные типы.

```
unique_ptr<int[]> ui (new int[1000]());
```

- Что в этом случае можно сделать?

```
template <typename T,typename Deleter = default_delete<T>>
class unique_ptr;
template <typename T> struct default_delete {
  void operator() (T *ptr) { delete ptr; }
};
```

Влияние на размеры

- Как вы думаете, влияет ли на размер необходимость хранить удалитель?
- Можете ли вы предложить вариант реализации чтобы не было проблем?

Внутренний tuple

- Интересный вариант решения это внутренний tuple.

```
template <typename T, typename Deleter = default_delete<T> >
class unique_ptr {
  std::tuple<T *, Deleter> content_;
```

- Теперь благодаря EBCO мы тратим не больше места, чем нужно

Уникальные указатели и аллокаторы

- Рассмотрим следующую интересную схему.

```
struct Bar { pmr::string data{"data"}; };
struct Foo {
  std::unique_ptr<Bar> bar_{ std::make_unique<Bar>() };
};
```

- Использование

```
pmr::vector<Foo> foos; // по умолчанию стоит test,esource
foos.emplace_back();
foos.emplace_back();
```

- Что на экране?

Давайте немного усложним Foo

- Поскольку в стандарте нет pmr::unique_ptr, сделаем это руками

```
class Foo {
  unique_ptr<Bar, polymorphic_allocator_delete> d_bar;
public:
  Foo(polymorphic_allocator<byte> alloc) : d_bar(nullptr, alloc) {
// тут выделение ресурса в терминах аллокатора
  }
};
```

- Как будем реализовать пользовательский удалитель?

Пользовательский удалитель

• Примерный вид удалителя

```
class polymorphic_allocator_delete {
  polymorphic_allocator<byte> alloc;
public:
  polymorphic_allocator_delete(polymorphic_allocator<byte> alloc):
    alloc(alloc) {}
  template <typename T> void operator()(T *ptr) {
    alloc.destroy(ptr);
    alloc.deallocate(ptr, 1);
  }
};
```

Конструктор для Foo

• Поскольку в стандарте нет pmr::unique_ptr, сделаем это руками

```
class Foo {
  unique_ptr<Bar, polymorphic_allocator_delete> d_bar;
public:
  Foo(polymorphic_allocator<byte> alloc) : d_bar(nullptr, alloc) {
    Bar *const bar = alloc.allocate(1);
    alloc.construct(bar);
    d_bar.reset(bar);
  }
};
```

• И что теперь на экране?

Обсуждение

• Кажется ли вам хорошей идеей дерево из unique_ptr?

```
template <typename Data> class Tree {
  struct Node {
    unique_ptr<Node> left, right;
    Data d; // предполагаем некопируемые данные
  };
  unique_ptr<Node> top_;
public:
??? find (int inorder_pos);
```

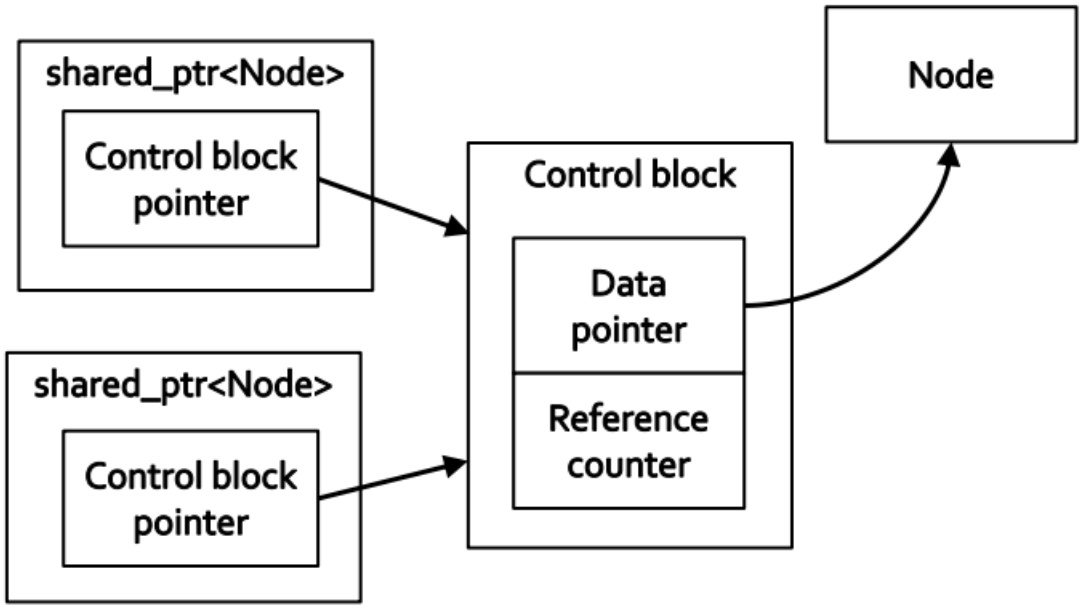
Идея для метода find в дереве

- Вариант с совместным владением требует перестройки дерева под себя

```
template <typename Data> class Tree {
  struct Node {
    shared_ptr<Node> left, right;
    Data d;
  };
  shared_ptr<Node> top_;
public:
  shared_ptr<Node> find (int inorder_pos);
```

- Здесь есть существенная проблема. Какая?

Что тут можно сделать?



- Благодаря созданию с алиасингом, данные могут ссылаться не на все, а на часть данных контрольного блока.
- При этом владение сохраняется за контрольным блоком.

Проблема: два контрольных блока

• Представим, что два разных разделяемых указателя были сделаны из одного.

```
Node *n = new Node();
shared_ptr<Node> spn1(n);
shared_ptr<Node> spn2(n);
```

• Это приводит к созданию двух контрольных блоков и той же проблеме двойного удаления, от которой мы исходно пытались уйти с помощью подсчёта ссылок.

• Это ещё один аргумент за make_shared, которая исключает такие проблемы

Пошарить самого себя

• Совместное владение предполагает иной подход к копированию.

```
struct Node
  shared_ptr<Node> getspn()
    return shared_ptr<Node>(this); // грубая ошибка


;
shared_ptr<Node> bp1 = make_shared<Node>();
shared_ptr<Node> bp2 = bp1->getspn();
```

• Все ли видят тут проблему?

Curiously recurring template parameter

- Идиома CRTP очень проста: мы используем одновременно шаблоны и наследование.

template <class T> class Base ....
class Derived : public Base <Derived> ....

- Мы наследуемся от базы, параметризованной самим наследником!
- Более подробно мы поговорим об этой технике на следующей лекции

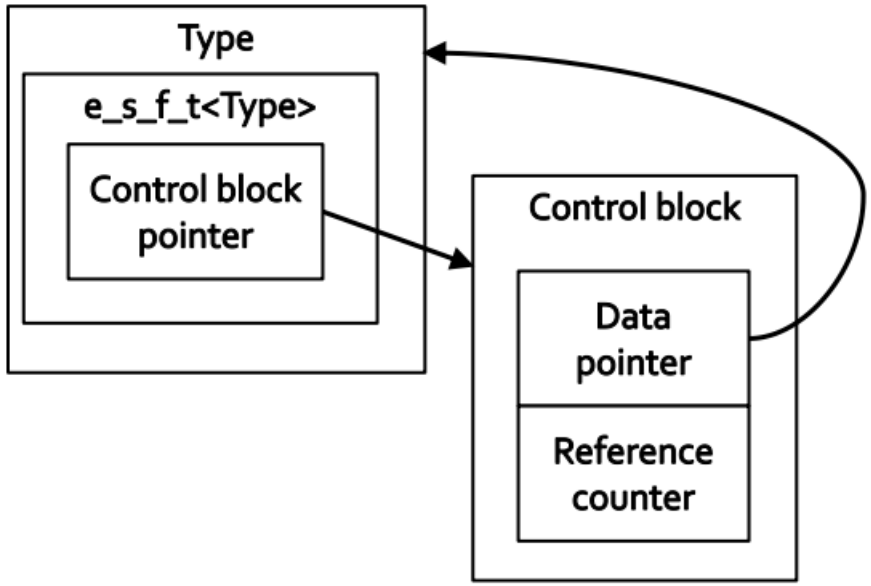
Правильный вариант

- Правильная стратегия действий.

```
struct Node: enable_shared_from_this<Node> {
  shared_ptr<Node> getspn() {
    return shared_from_this();
}
};
```

- Как бы вы реализовали enable_shared_from_this?

Как работает enable_shared_from_this



- Сохраняет некий указатель на контрольный блок в классе.
- Предоставляет для конструктора shared_ptr интерфейс по проверке, что такой указатель в классе есть.
- Обсуждение: может ли указатель на контрольный блок быть shared_ptr?

Правильное использование

```
struct Node: enable_shared_from_this<Node> {
  shared_ptr<Node> getspn() { return shared_from_this(); }
private:
  some_ptr control_block_pointer;
};
auto n = make_shared<Node>(); // контрольный блок создан
shared_ptr<Node> gp1 = n.getspn(); // OK
```

- Необходимость такой дисциплины удручает и частично мотивирует фабричные методы.

Форсим правильное использование

```
struct Node: enable_shared_from_this<Node> {
  shared_ptr<Node> getspn() { return shared_from_this(); }
  template<typename ... T>
  static shared_ptr<Node> create(T&& ... t) {
    return shared_ptr<Node>(new Node(forward<T>(t)...));
  }
private:
  some_ptr control_block_pointer;
  Node(node args);
};
```

Приведение

- Говорят, что обычные указатели ковариантны
- Если B законно преобразуется к A, то B* законно преобразуется к A*

```
class A ;
class B : public A ;
B *b = new B(); A *a = static_cast<A*>(b);
```

- Но шаблонные классы (в том числе класс shared_ptr) инвариантны
- Отношения между A и B не распространяются на S<A> и S<B>

```
shared_ptr<B> b = make_shared<B>();
shared_ptr<A> a = ???
```

Приведение

• Есть способы сохранить нечто вроде свойств указателей.

```
class A {};
class B : public A {};
B *b = new B();
A *a = static_cast<A*>(b);
shared_ptr<B> b = make_shared<B>();
shared_ptr<A> a = static_pointer_cast<A>(b);
```

• Аналогично работают dynamic_pointer_cast и const_pointer_cast, имитируя известные приведения.

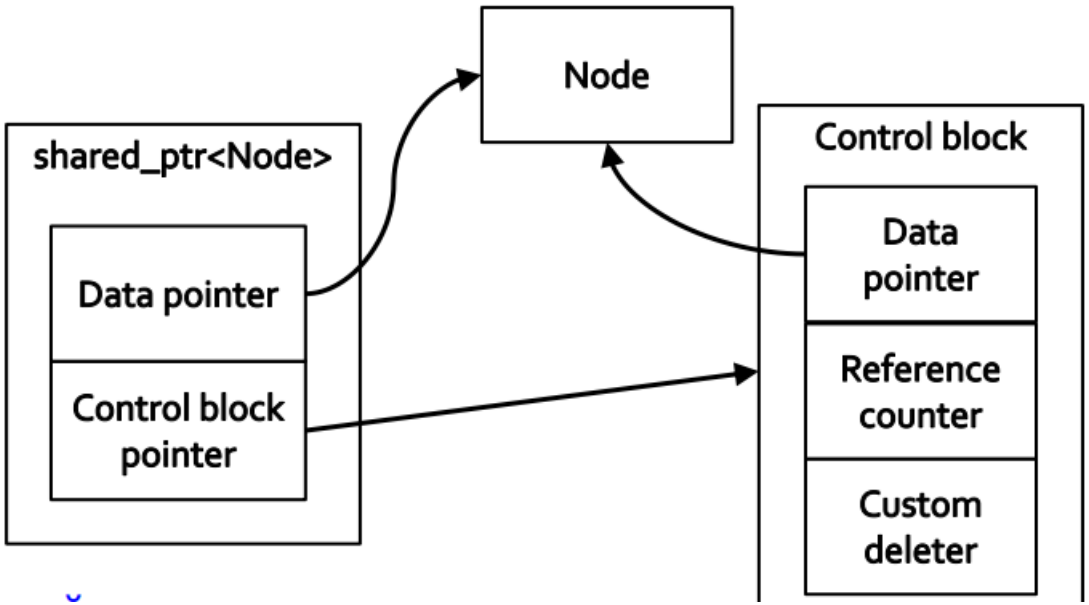
Обсуждение

- «Shared pointer is as good as global variable when it comes to being able to reason about code that uses one» © Sean Parent

Пользовательские удалители

- При разделяемом владении момент освобождения ресурса бывает вообще не очевиден. Тем важнее задать способ освобождения.
SDL_Surface* s = SDL_LoadBMP(....); // хочется shared_ptr
- Здесь явно не достаточно обычного delete для освобождения
shared_ptr<SDL_Surface>(SDL_LoadBMP(....), SDL_FreeSurface);
- Важно отметить: в отличии от unique_ptr, пользовательский удалитель не является частью типа. Он идёт не в шаблонный параметр, а в конструктор.
- Из-за этого надо его явно указывать даже для встроенных массивов.
- Кто-нибудь понимает почему это так?

Где лежит deleter в shared_ptr



Для разделяемого указателя всё равно нужен контрольный блок.

- Поэтому нет никаких оснований усложнять типизацию.

Обсуждение

- Какие основные проблемы создаёт копирование и подсчёт ссылок при работе с разделяемыми указателями?

Закольцованные указатели

• Главная проблема указателей с совместным владением это возможность циркулярных ссылок.

```
struct Node {
  shared_ptr<Node> parent, left, right;
};
{
  shared_ptr<Node> master = make_shared<Node>(); // счётчик:1
  shared_ptr<Node> slave = make_shared<Node>(); // счётчик:1
  slave->parent = master; // счётчик:2
  master->left = slave; // счётчик:2
} // LEAK
```

Решение: слабые указатели

• Слабый указатель не владеет объектом, на который указывает.

```
struct Node {
  weak_ptr<Node> parent;
  shared_ptr<Node> left, right;
};
{   shared_ptr<Node> master = make_shared<Node>(); // счётчик:1
  shared_ptr<Node> slave = make_shared<Node>(); // счётчик:1
  slave->parent = master; // счётчик:1
  master->left = slave; // счётчик:2
} // OK, уничтожен master, после чего slave
```

Слабый указатель нельзя разыменовать

• Простое разыменование не сработает.

```
auto t = make_shared<int>(42);
weak_ptr<int> w = t;
int xt = *t; // ok
// сделать int xw = *w нельзя
```

• Зато его можно превратить в сильный указатель и потом разыменовать.

```
auto tprime = w.lock();
int xtp = *tprime;
```

• Защёлкивание не вовремя может создать те же проблемы циклической ссылки, но они более контролируемы.

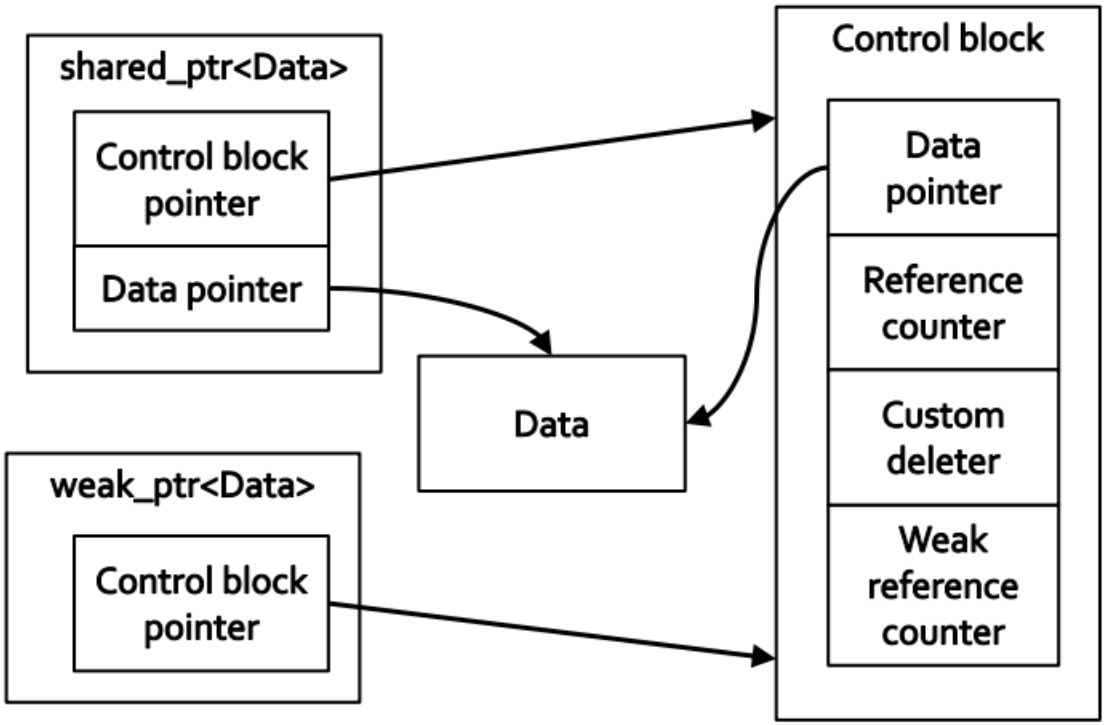
Слабый указатель не может повиснуть

Типичный случай, приводящий к подвисанию обычных указателей.

```
weak_ptr<int> foo () {
  auto res = make_shared<int>(42);
  return res;
} // в этот момент память была освобождена
  weak_ptr<int> result = foo();
assert (result.expired());
assert (result.lock() == nullptr);
```

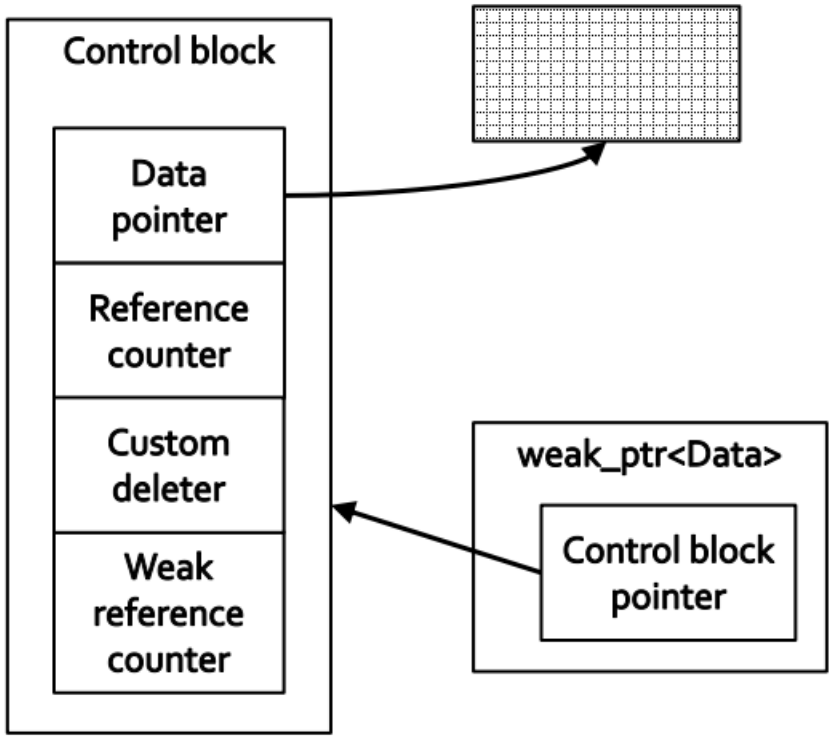
- Слабый указатель можно проверить на истечение срока жизни.
- Это даёт большие возможности для контроля валидности состояния.

Как может быть устроен weak_ptr



- Пока есть сильные ссылки, данные Data сохраняются в куче.
- Слабые указатели могут с помощью lock() получить к ним доступ, так как Data pointer не nullptr.

Как может быть устроен shared_ptr



- Если не остаётся ни одной сильной ссылки, то:
- control block живёт пока есть слабые ссылки.
- Data освобождается.
- Data pointer становится nullptr
- Если weak pointer указывает на control block без сильных ссылок, то он expired.

Обсуждение

- Интересно, что make_unique было почти стилистическим решением.
- При этом make_shared имеет весомые аргументы как за, так и против.
- В обоих случаях безопасность исключений это важный аргумент за и его не надо забывать.

```
foo(unique_ptr<T>(new T(1)), unique_ptr<T>(new T(2)));
foo(make_unique<T>(1), make_unique<T>(2));
```

- Первая строчка плоха (все ли видят почему?) вторая решает проблему

Обсуждение

- Интересно, что make_unique было почти чисто стилистическим решением.
- При этом make_shared имеет весомые аргументы как за, так и против.
- В обоих случаях безопасность исключений это важный аргумент за и его не надо забывать.

```
foo(shared_ptr<T>(new T(1)), shared_ptr<T>(new T(2)));
foo(make_shared<T>(1), make_shared<T>(2));
```

- Все те же самые соображения в силе.
- Есть ли специфичные аргументы?

Обсуждение

- Интересно, что make_unique было почти чисто стилистическим решением.
- При этом make_shared имеет весомые аргументы как за, так и против.
- В обоих случаях безопасность исключений это важный аргумент за и его не надо забывать.

```
foo(shared_ptr<T>(new T(1)));
foo(make_shared<T>(1));
```

- Все ли понимают почему здесь первая строчка не безопасна, а вторая безопасна?