Lecture 20

Iterators

Итерационные функции

```
std::distance(Iter fst, int n); // snd - fst, либо цикл
std::advance(Iter sat, int n); // fst + n, либо цикл
```

У таких функций, в отличие от интерфейса итераторов, неопределенная асимптотическия сложность.

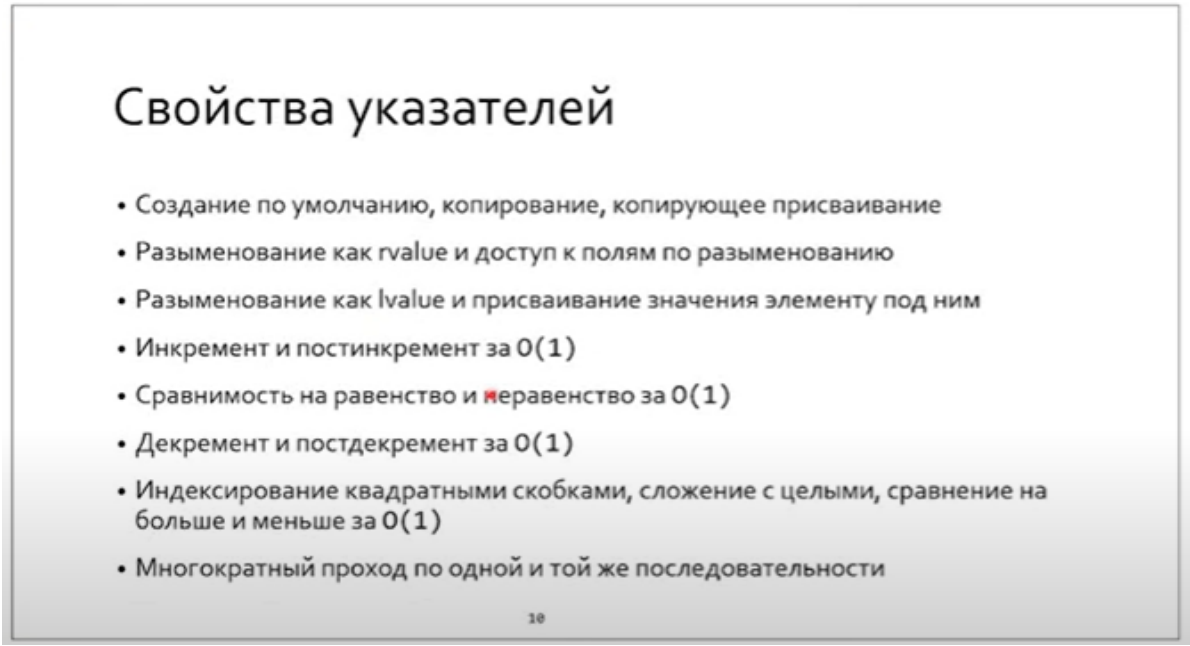
`prev = std::exchange(cur, cur + prev)` - записать в `cur` новое значение, а старое `cur` выдать в `prev`.

`std::find_if_not`

> В каких случаях будет вызвана внешняя begin, или `std::begin`?
>
> Если у нас не встроенный массив (наивысший приоритет, в котором начало - адрес первого, а конец - адрес за последним) и не что-то, у чего есть методы begin/end (догадывается о наличии через SFINAE) и есть внешняя функция.

ADL (argument depended lookup)

Если функция не может быть найдена в текущем namespace и в охватывающих - она будет искаться в namespace аргументов.



Output итераторы

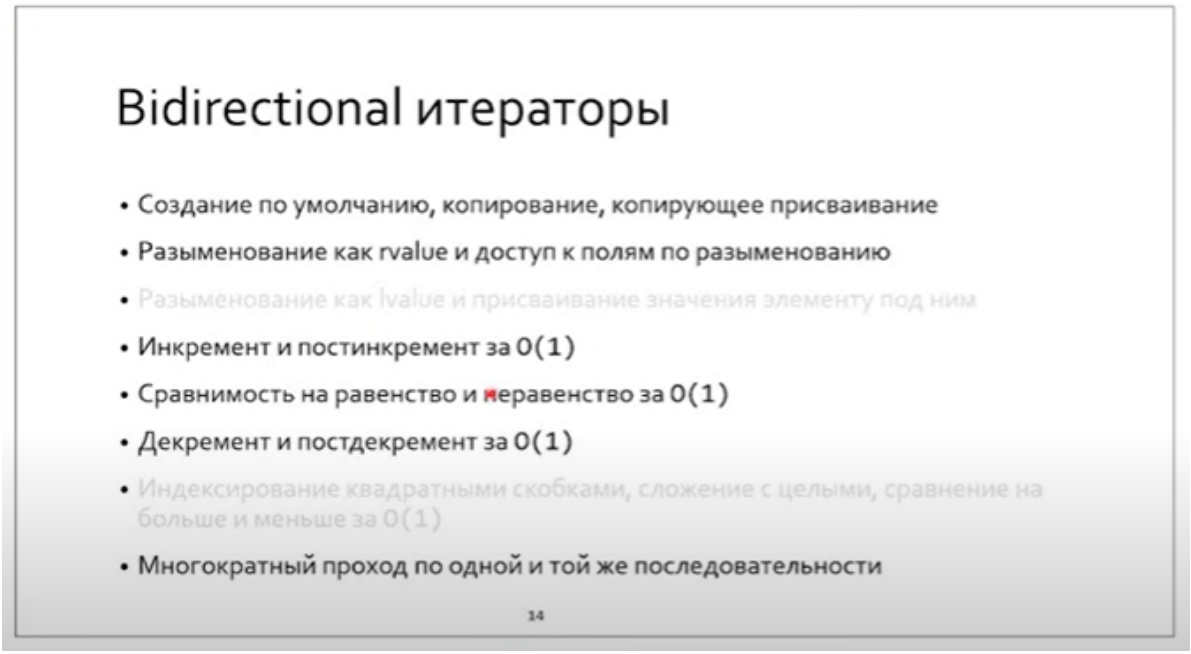
- Создание по умолчанию, копирование, копирующее присваивание
- Разыменование как rvalue и доступ к полям по разыменованию
- Разыменование как lvalue и присваивание значения элементу под ним
- Инкремент и постинкремент за O(1)
- Сравнимость на равенство и неравенство за O(1)
- Декремент и постдекремент за O(1)
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за O(1)
- Многократный проход по одной и той же последовательности

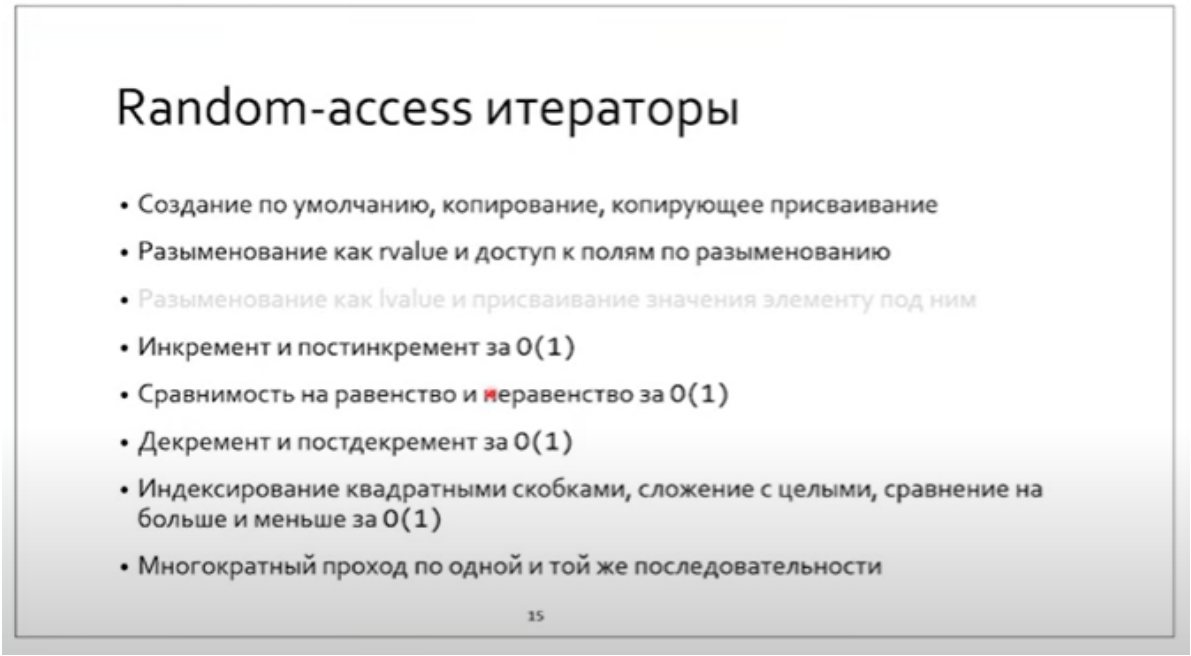
Input итераторы

- Создание по умолчанию, копирование, копирующее присваивание
- Разыменование как rvalue и доступ к полям по разыменованию
- Разыменование как lvalue и присваивание значения элементу под ним
- Инкремент и постинкремент за O(1)
- Сравнимость на равенство и неравенство за O(1)
- Декремент и постдекремент за O(1)
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за O(1)
- Многократный проход по одной и той же последовательности

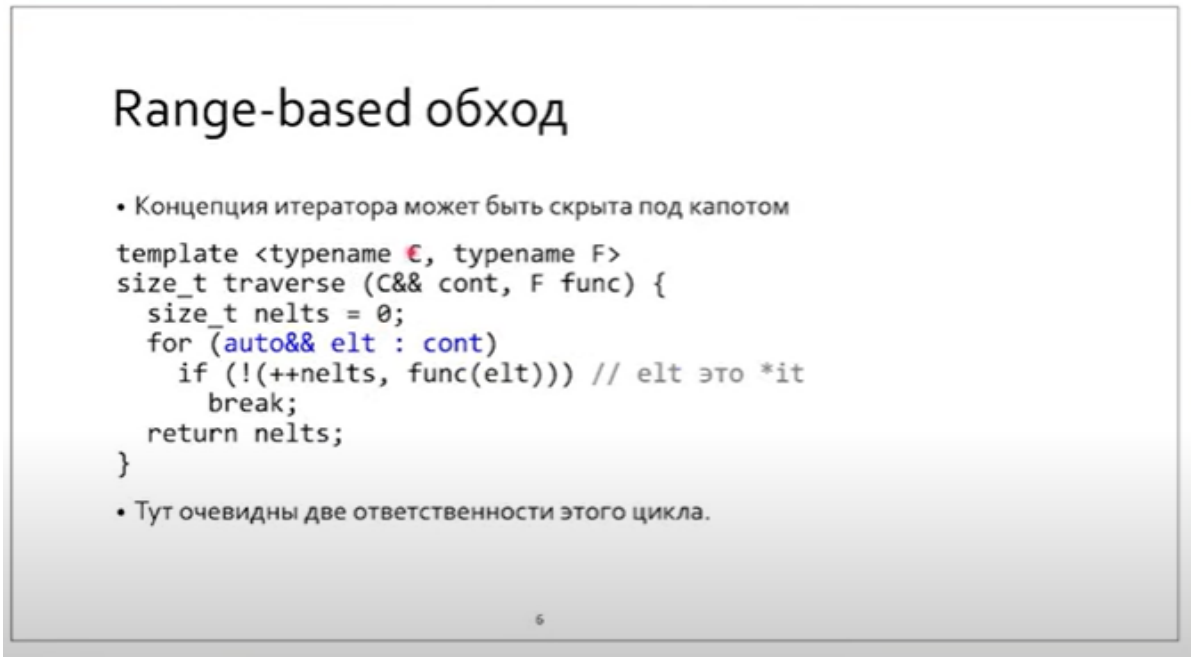
Forward итераторы

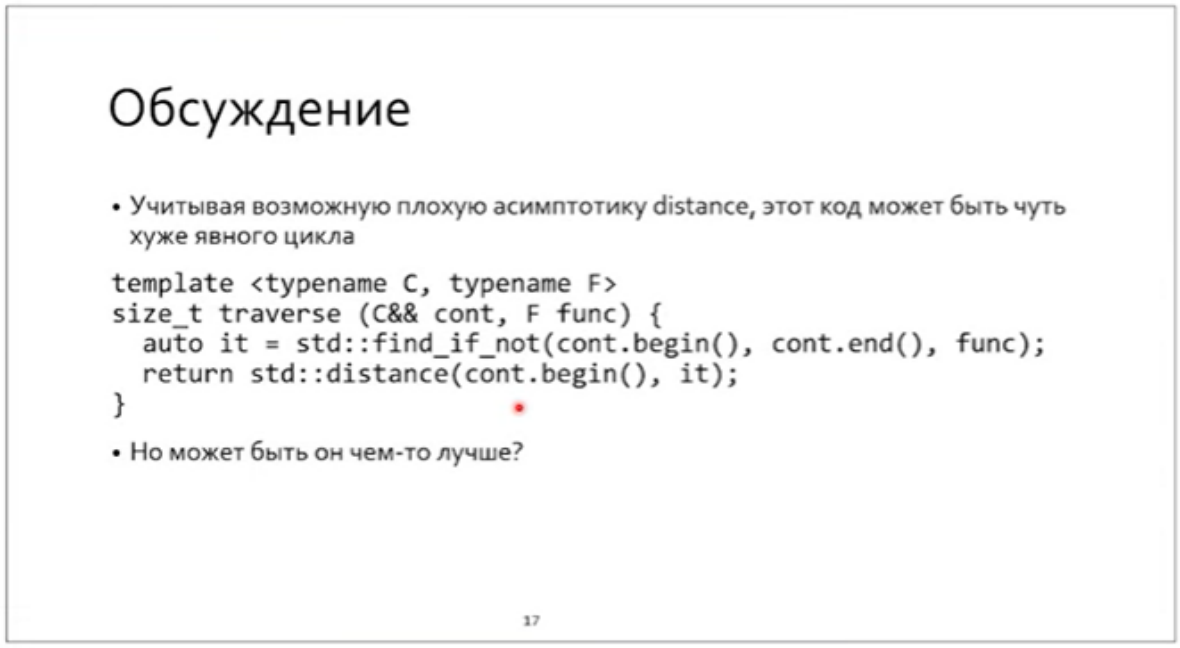
- Создание по умолчанию, копирование, копирующее присваивание
- Разыменование как rvalue и доступ к полям по разыменованию
- Разыменование как lvalue и присваивание значения элементу под ним
- Инкремент и постинкремент за O(1)
- Сравнимость на равенство и неравенство за O(1)
- Декремент и постдекремент за O(1)
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за O(1)
- Многократный проход по одной и той же последовательности





Итератор - копируемый объект!

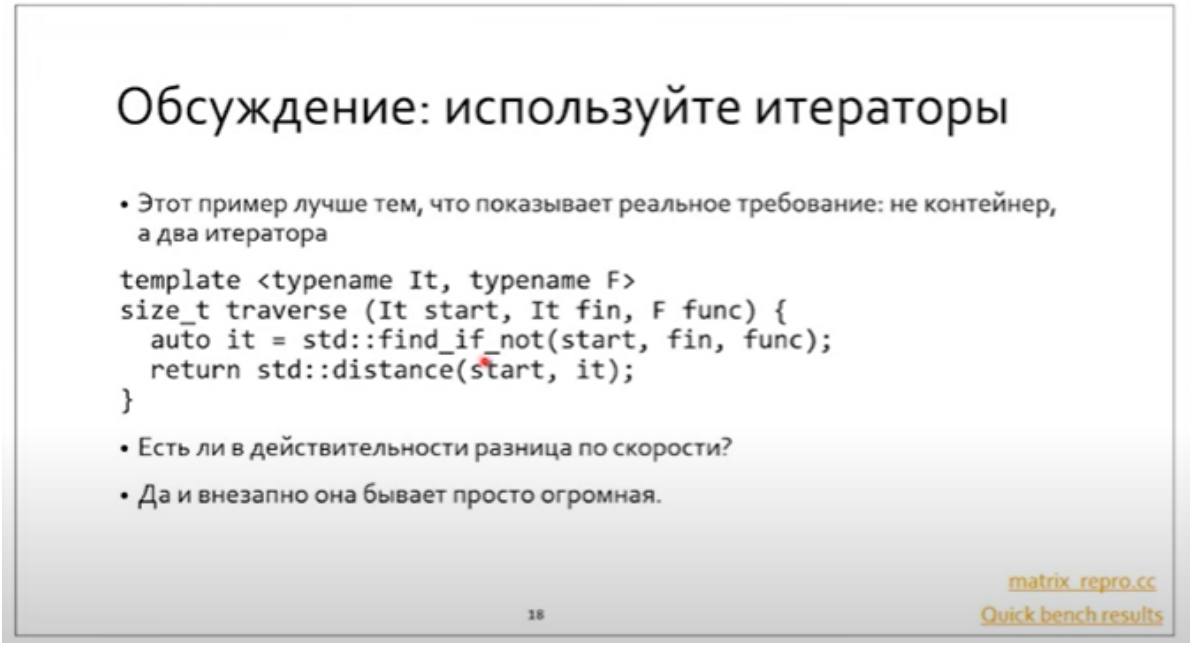




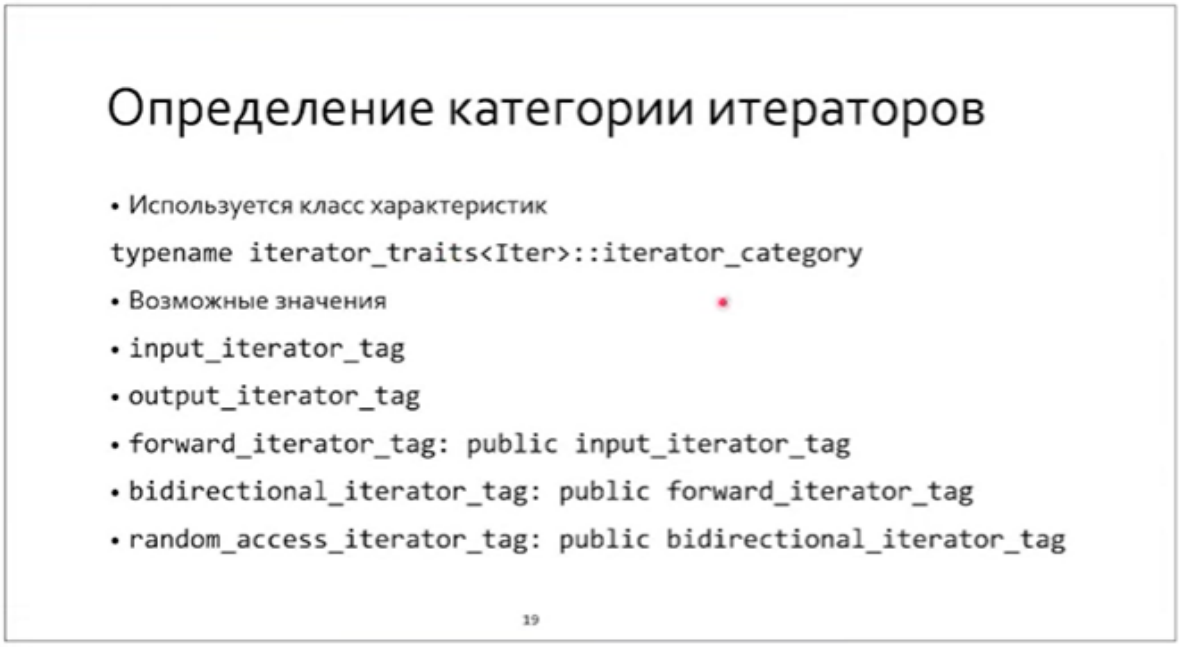
Тут вернется позиция, для которой `func` вернула `false`.

Если не лезть в контейнер, то из-за -1 уровня косвенности будет ускорение по производительности (в 3 раза) - см. `matrix_repro.cc`

При итерациях по чему-то - нужно передавать пару итераторов.



У итераторов синтаксически одинаковый интерфейс, за исключением класса характеристик:



Их можно вывести, перегрузив оператор <<

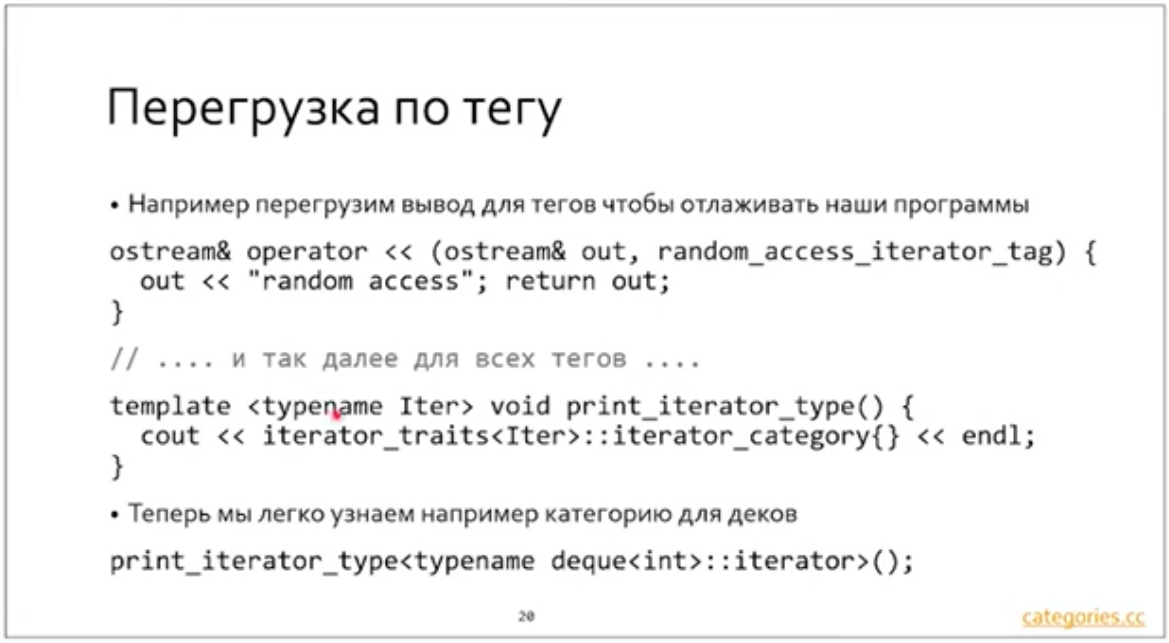
```
43  }
44
45  template <typename Iter> void print_iterator_type() {
46    std::cout << typename iterator_traits<Iter>::iterator_category{} << std::endl;
47  }
48
49  int main() {
50    print_iterator_type<typename std::deque<int>::iterator>();
51    print_iterator_type<typename std::forward_list<int>::iterator>();
52    print_iterator_type<typename std::list<int>::iterator>();
53    print_iterator_type<typename std::vector<int>::iterator>();
54    print_iterator_type<std::istream_iterator<int>>();
55    print_iterator_type<std::ostream_iterator<int>>();
56  }
```

random, forward, bidirect, random, input, output

Как отличить input_iterator от forward_iterator, если у них нет синтаксических различий (т.е. SFINAE не сработает), и есть лишь семантические?

Это решается с помощью тэгов (iterator_category). Они позволяют вынести семантические различия на синтаксический уровень.

Перегрузка по тэгам:



получим вывод типов итераторов на экран.

Как навесить ограничение на передаваемый тип итератора?

Conditional type

```
template <bool B, typename T, typename F>
struct conditional { using type = T; }

template <typename T, typename F>
struct conditional<false, T, F> { using type = F; }

template <bool B, typename T, typename F>
using conditional_t = typename conditional<B, T, F>::type;

// если хотим сделать невалидным для F:

template <bool B, typename T = vo id>
struct conditional { using type = T; }

template <typename T = void>
struct conditional<false, T> {}

template <bool B, typename T = void>
using enable_if_t = typename enable_if_t<B, T>::type;

// т.о. конструкция будет невалидной, если B = false, т.к. не будет
определен type.
```

Применительно к итераторам это выглядит так:

Проверка категории

- Иногда мы хотим обложить перегрузку SFINAE проверкой

```
template <typename It>
using iterator_category_t =
        typename std::iterator_traits<It>::iterator_category;

template <typename It, typename T = std::enable_if_t<
    std::is_base_of_v<input_iterator_tag,
                      iterator_category_t<It>>>>
void foo(It first, It last)
```

- Все ли понимают, почему base of, а не same?

25

https://godbolt.org/z/Ma7Pn8MrY

Итератор помимо iterator_category должен также определять следующие типы:

```
public:
    using value_type = int;
    using difference_type = ptrdiff_t;
    using pointer = int*;
    using reference = int&;
```

Например, напишем итератор по паре значений:

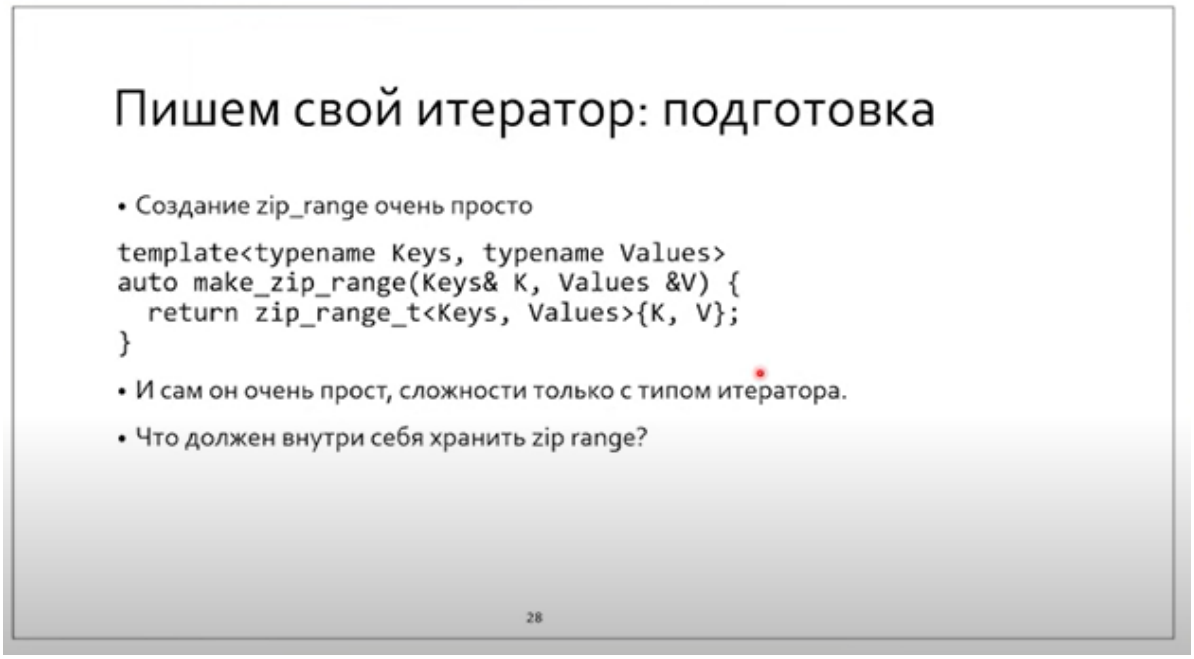
Case study: пишем свой итератор

- Постановка задачи: итерирование сразу по двум контейнерам

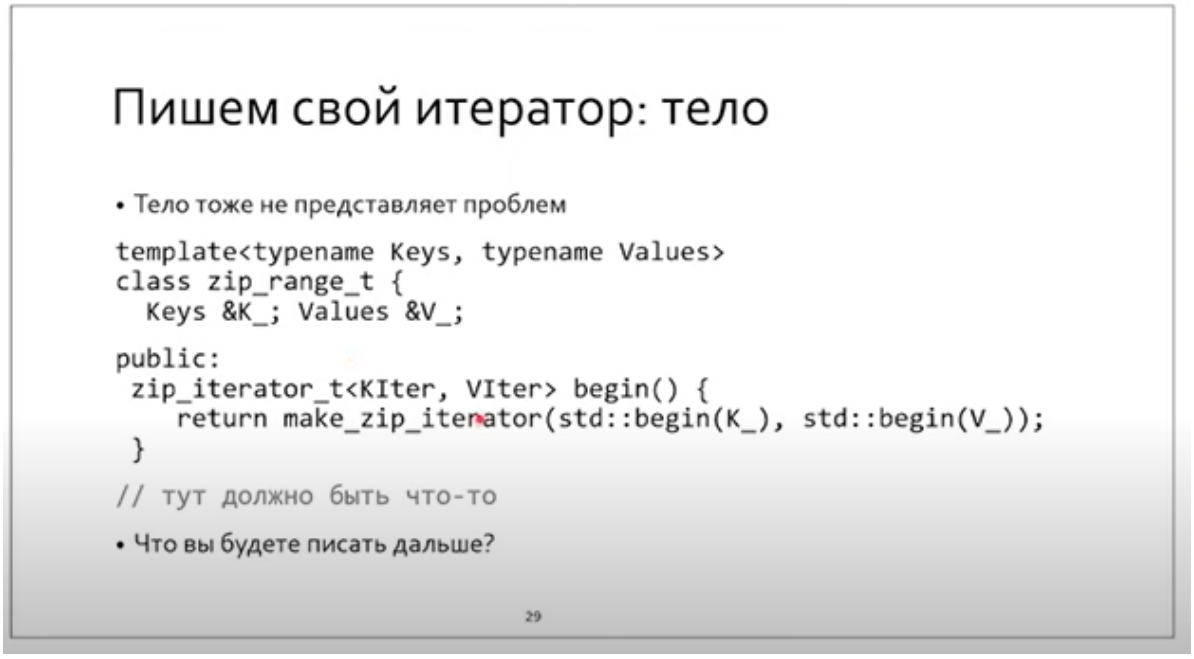
```
std::vector<int> keys = {1, 2, 3, 4};
std::vector<double> values = {4.0, 3.0, 2.0, 1.0};

for (auto &&both : make_zip_range(keys, values))
  std::cout << both.first << ", " << both.second << "; ";

// 1, 4.0; 2, 3.0; 3, 2.0; 4, 1.0
```

- Нужно придумать легковесную обёртку zip_range и возвращаемые ей итераторы (тип для них)

27

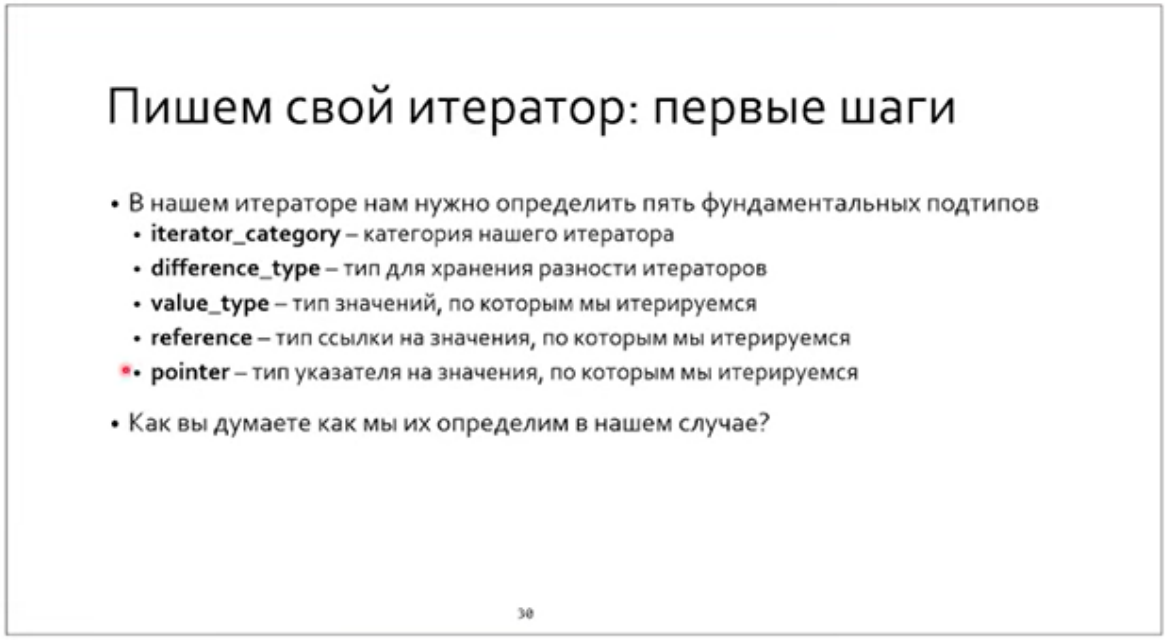


Ссылки, потому что массивы заданы извне.

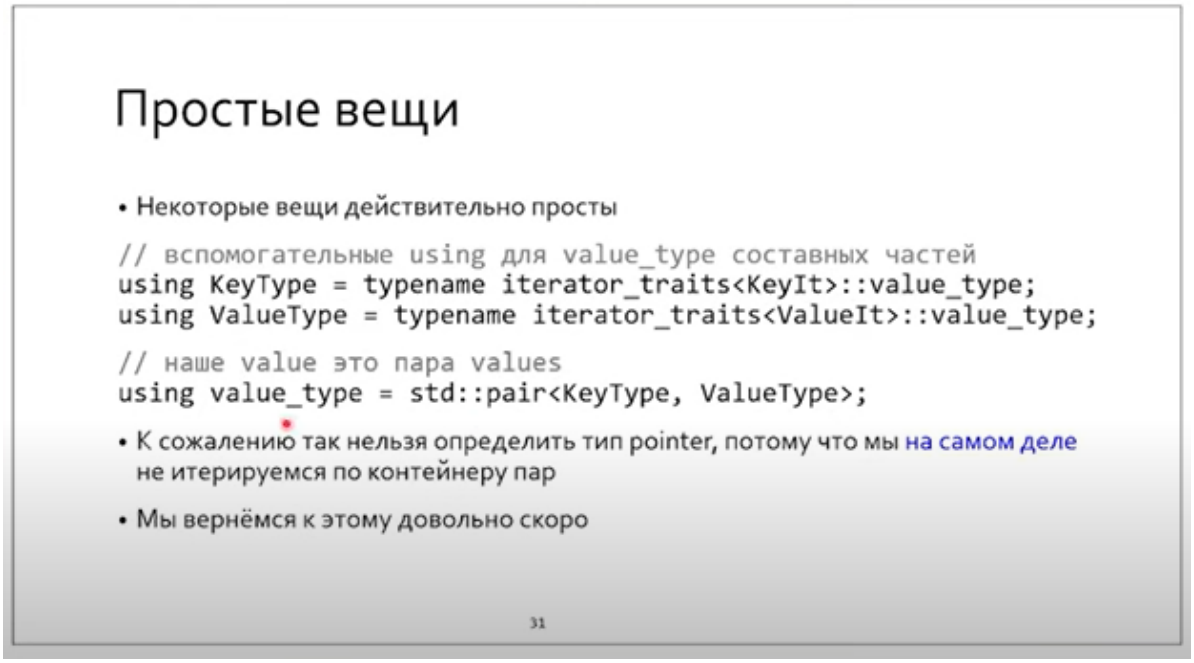


В качестве категории по умолчанию нужно использовать input_iterator.

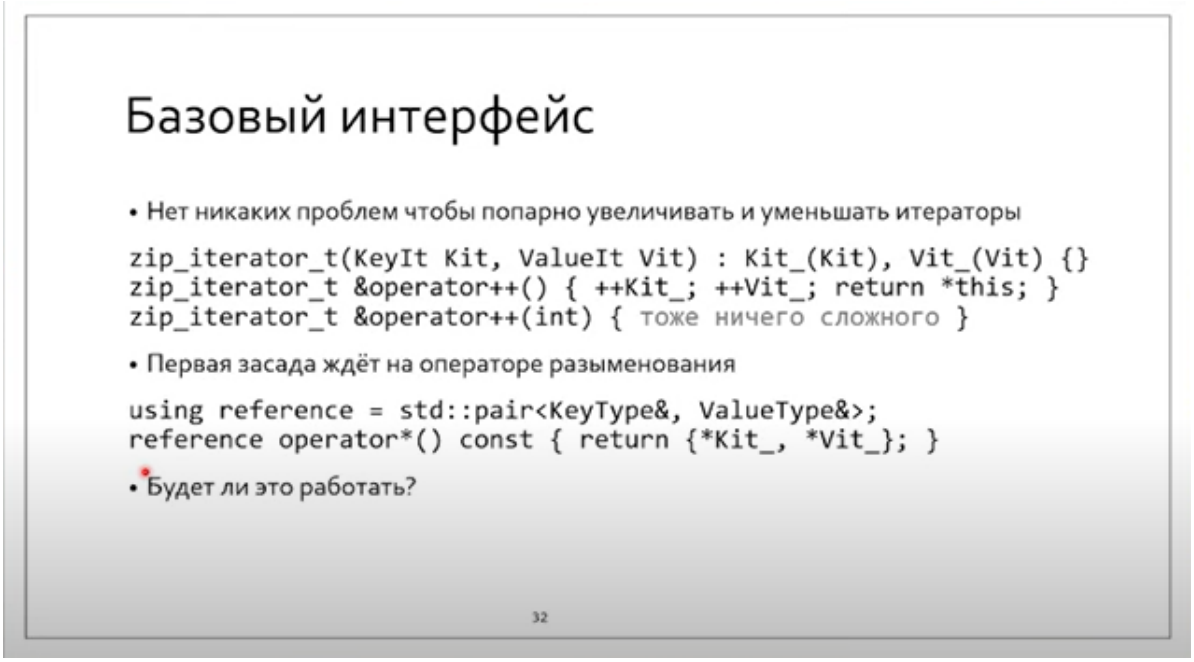
Порядок реализации:



Важно! Класс с полями-ссылками требует дефолтного конструктора.

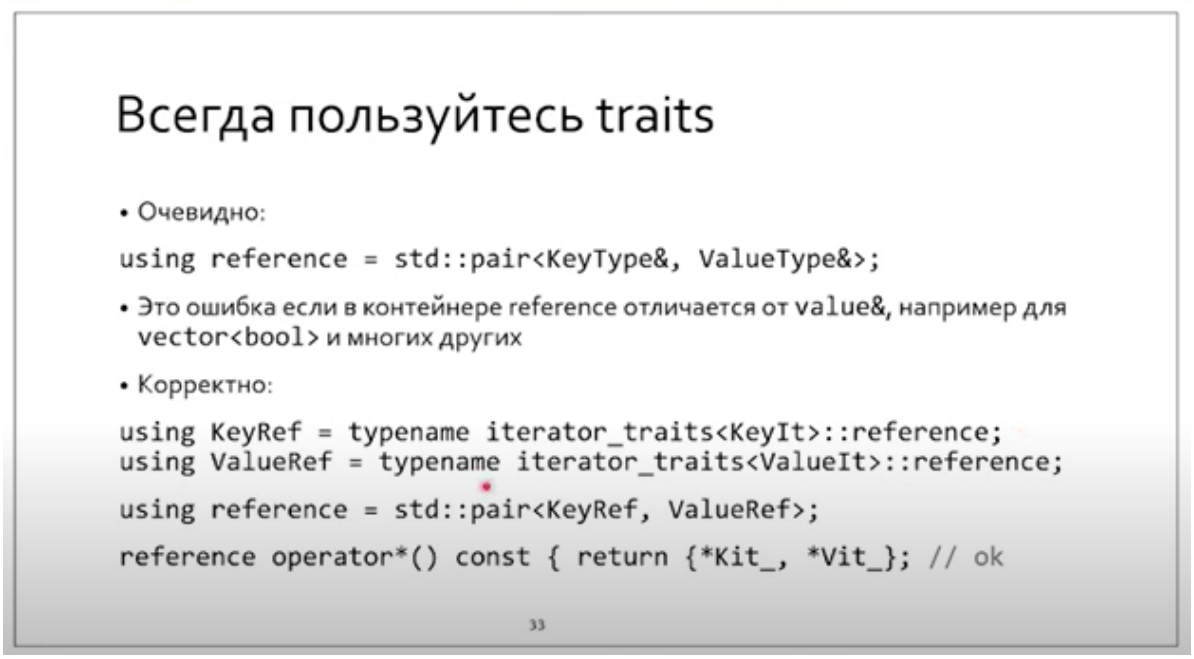


> Почему value_type - пара значений а не ссылок. Ну, есть тип reference + не всем свойствам значений удовлетворяют ссылки.



> Будет ли провисание ссылок при возврате из `operator*()`? Нет, т.к. ссылки ссылаются на внешние массивы. Обращаем внимание, что агрегат в этом операторе выводится к reference type.

Однако не все так просто с ссылками. Например, вектор булов так не сработает. Верное решение:



А что насчет указателей?

Можно определить вот так

```
1 void operator->() const {}
```

Тогда его компилятор рассматривает несуществующим. А если нужна реализация?

Настоящая проблема: стрелочка

- Как вообще должен выглядеть оператор разыменования?

```
auto zit = make_zip_iterator(k.begin(), b.begin());
assert (k.front() == zit->first);
// zit->first drills down to (zit.operator->())->first
```

- Это должен быть аналог разыменованию и обращению к полю

```
pointer operator->() const { return some pointer; }
```

- Но что такое pointer? Простое решение не подходит

```
using pointer = std::pair<KeyPtr, ValuePtr>; // нет p->first
```

34

Выход из положения - proxy class

Изящное решение: прокси класс

- На помощь приходит прокси-класс

```
template <typename Reference> struct arrow_proxy {
  Reference R;
  Reference *operator->() { return &R; } // non const
};
using pointer = arrow_proxy<reference>;
pointer operator->() const { return pointer{{*Kit_, *Vit_}}; }
```

- Есть некие опасения в том что прокси провиснет, но нам он нужен чтобы пережить drill-down, а его явно переживёт

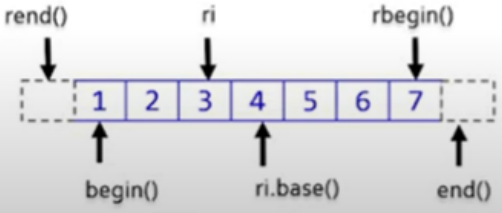
36 ziprange.hpp

Заметим, что класс невладеющий: он хранит ссылку на внешний объект. Естественно, он не провиснет во время создания / удаления прокси-класса.

Zip_range - типичный адаптер.

Переход к прямому итерированию

```
std::vector v {1, 2, 3, 4, 5, 6, 7};
auto ri = v.rbegin() + 4;
auto it = ri.base();
cout << *ri << " " << *it << endl; // 3 4
```



48

Как вывести reverse range based?

Реализация reverse_cont

```
template <typename T> struct reversion_wrapper {
  T& iterable;
};

template <typename T> auto begin(reversion_wrapper<T> w) {
  return rbegin(w.iterable);
}

template <typename T> auto end(reversion_wrapper<T> w) {
  return rend(w.iterable);
}

template <typename T>
reversion_wrapper<T> reverse_cont(T&& iterable) {
  return { iterable };
}
```

50 revcont.cc

Adapters

- inserters (back / front)
-