

Lecture 20

Iterators

Итерационные функции

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

У таких функций, в отличие от интерфейса итераторов, неопределенная асимптотическая 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.

Свойства указателей

- 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 **н**еравенство 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

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 \neq равенство 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

11

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 \neq равенство 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

12

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 \neq равенство 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

13

Bidirectional 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 \neq равенство 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

14

Random-access 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 \neq равенство 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

15

Итератор - копируемый объект!

Range-based обход

- Концепция итератора может быть скрыта под капотом

```
template <typename C, typename F>
size_t traverse (C&& cont, F func) {
    size_t nelts = 0;
    for (auto&& elt : cont)
        if (!(++nelts, func(elt))) // elt это *it
            break;
    return nelts;
}
```

- Тут очевидны две ответственности этого цикла.

6

Обсуждение

- Учитывая возможную плохую асимптотику `distance`, этот код может быть чуть хуже явного цикла

```
template <typename C, typename F>
size_t traverse (C&& cont, F func) {
    auto it = std::find_if_not(cont.begin(), cont.end(), func);
    return std::distance(cont.begin(), it);
}
```

- Но может быть он чем-то лучше?

17

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.

Обсуждение: используйте итераторы

- Этот пример лучше тем, что показывает реальное требование: не контейнер, а два итератора

```
template <typename It, typename F>
size_t traverse (It start, It fin, F func) {
    auto it = std::find_if_not(start, fin, func);
    return std::distance(start, it);
}
```

- Есть ли в действительности разница по скорости?
- Да и внезапно она бывает просто огромная.

matrix.repro.cc
[Quick bench results](#)

18

У итераторов синтаксически одинаковый интерфейс, за исключением класса характеристик:

Определение категории итераторов

- Используется класс характеристик

```
typename iterator_traits<Iter>::iterator_category
```

- Возможные значения

- input_iterator_tag

- output_iterator_tag

- forward_iterator_tag: public input_iterator_tag

- bidirectional_iterator_tag: public forward_iterator_tag

- random_access_iterator_tag: public bidirectional_iterator_tag

19

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:

Перегрузка по тегу

- Например перегрузим вывод для тегов чтобы отлаживать наши программы

```
ostream& operator << (ostream& out, random_access_iterator_tag) {
    out << "random access"; return out;
}

// .... и так далее для всех тегов ....

template <typename Iter> void print_iterator_type() {
    cout << iterator_traits<Iter>::iterator_category{} << endl;
}
```

- Теперь мы легко узнаем например категорию для деков

```
print_iterator_type<typename deque<int>::iterator>();
```

20

categories.cc

получим вывод типов итераторов на экран.

Как навесить ограничение на передаваемый тип итератора?

Conditional type

```
1  template <bool B, typename T, typename F>
2  struct conditional { using type = T; }
3
4  template <typename T, typename F>
5  struct conditional<false, T, F> { using type = F; }
6
7  template <bool B, typename T, typename F>
8  using conditional_t = typename conditional<B, T, F>::type;
9
10 // если хотим сделать невалидным для F:
11
12 template <bool B, typename T = void>
13 struct conditional { using type = T; }
14
15 template <typename T = void>
16 struct conditional<false, T> {}
17
18 template <bool B, typename T = void>
19 using enable_if_t = typename enable_if_t<B, T>::type;
20
21 // 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

Пишем свой итератор: подготовка

- Создание `zip_range` очень просто

```
template<typename Keys, typename Values>
auto make_zip_range(Keys& K, Values &V) {
    return zip_range_t<Keys, Values>{K, V};
}
```

- И сам он очень прост, сложности только с типом итератора.
- Что должен внутри себя хранить `zip range`?

28

Ссылки, потому что массивы заданы извне.

Пишем свой итератор: тело

- Тело тоже не представляет проблем

```
template<typename Keys, typename Values>
class zip_range_t {
    Keys &K_; Values &V_;
public:
    zip_iterator_t<KIter, VIter> begin() {
        return make_zip_iterator(std::begin(K_), std::begin(V_));
    }
}
```

// тут должно быть что-то

- Что вы будете писать дальше?

29

В качестве категории по умолчанию нужно использовать `input_iterator`.

Порядок реализации:

Пишем свой итератор: первые шаги

- В нашем итераторе нам нужно определить пять фундаментальных подтипов
 - **iterator_category** – категория нашего итератора
 - **difference_type** – тип для хранения разности итераторов
 - **value_type** – тип значений, по которым мы итерируемся
 - **reference** – тип ссылки на значения, по которым мы итерируемся
 - **pointer** – тип указателя на значения, по которым мы итерируемся
- Как вы думаете как мы их определим в нашем случае?

30

Важно! Класс с полями-ссылками требует дефолтного конструктора.

Простые вещи

- Некоторые вещи действительно просты

```
// вспомогательные using для value_type составных частей
using KeyType = typename iterator_traits<KeyIt>::value_type;
using ValueType = typename iterator_traits<ValueIt>::value_type;

// наше value это пара values
using value_type = std::pair<KeyType, ValueType>;
```
- К сожалению так нельзя определить тип pointer, потому что мы **на самом деле** не итерируемся по контейнеру пар
- Мы вернёмся к этому довольно скоро

31

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

Базовый интерфейс

- Нет никаких проблем чтобы попарно увеличивать и уменьшать итераторы

```
zip_iterator_t(KeyIt Kit, ValueIt Vit) : Kit_(Kit), Vit_(Vit) {}  
zip_iterator_t &operator++() { ++Kit_; ++Vit_; return *this; }  
zip_iterator_t &operator++(int) { тоже ничего сложного }
```

- Первая засада ждёт на операторе разыменования

```
using reference = std::pair<KeyType&, ValueType&>;  
reference operator*() const { return {*Kit_, *Vit_}; }
```

- Будет ли это работать?

32

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:

Всегда пользуйтесь traits

- Очевидно:

```
using reference = std::pair<KeyType&, ValueType&>;
```

- Это ошибка если в контейнере `reference` отличается от `value&`, например для `vector<bool>` и многих других

- Корректно:

```
using KeyRef = typename iterator_traits<KeyIt>::reference;  
using ValueRef = typename iterator_traits<ValueIt>::reference;  
using reference = std::pair<KeyRef, ValueRef>;  
reference operator*() const { return {*Kit_, *Vit_}; // ok
```

33

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

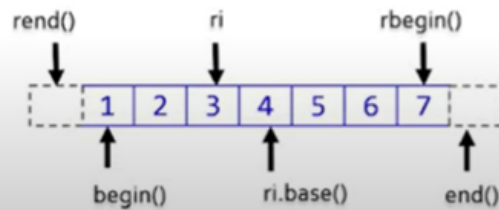
[ziprange.hpp](#)

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)
-