

4.18 Понятие итератора. Пример реализации итераторов для любого из стандартных контейнеров (на ваш выбор). Разница между константными и неконстантными итераторами. Реализация константных итераторов без дублирования кода относительно обычных итераторов, применение метафункции `std::conditional`.

Разница между константными и неконстантными итераторами заключается в том, что элементы, на которые указывает константный итератор, нельзя модифицировать. При разыменовании `const` итератора мы получаем не ссылку на `T`, а `const` ссылку на `T`.

Можно инициализировать константный итератор неконстантным итератором, но не наоборот.

Пример реализации для `List`

```
1  template <bool IsConst>
2      class myiterator {
3      public:
4          Node* iter = NULL;
5          using difference_type = std::ptrdiff_t;
6          using value_type = std::conditional_t<IsConst, const T, T>;
7          using pointer = std::conditional_t<IsConst, const T*, T*>;
8          using reference = std::conditional_t<IsConst, const T&, T&>;
9          using iterator_category = std::bidirectional_iterator_tag;
10
11
12         myiterator() {}
13         myiterator(Node* iter) : iter(iter) {}
14         reference operator*() {
15             return (iter->data);
16         }
17
18         pointer operator->() { return &(iter->data); }
19
20         myiterator& operator ++() {
21             Node* ptr = iter->next;
22             iter = ptr;
23             return *this;
24         }
25
26         myiterator operator ++(int) {
27             myiterator copy = *this;
28             Node* ptr = iter->next;
29             iter = ptr;
30             return copy;
31         }
32
33         myiterator& operator --() {
34             iter = iter->prev;
35             return *this;
36         }
37
38         bool operator !=(const myiterator& other) {
39             return !(iter == other.iter);
40         }
41         bool operator ==(const myiterator& other) {
42             return (iter == other.iter);
43         }
44     }
```

```

44     operator myiterator<true>() { return myiterator<true>{iter}; }
45 };
46
47 using const_iterator = myiterator<true>;
48 using iterator = myiterator<false>;

```

Реализация std::conditional

```

1  template<bool condition, typename T, typename F>
2  struct conditional{
3      using type = T;
4  }
5  template<bool condition, typename T, typename F>
6  struct conditional<true, F, T>{
7      using type = F;
8  }
9  template<bool condition, typename T, typename F>
10 using conditional_t = typename conditional<condition,F,T>::type;

```

4.19 Функции std::advance и std::distance, пример их применения. Реализация функции advance с правильной поддержкой разных видов итераторов, два способа такой реализации: с помощью перегрузки функций (старый способ до C++17) и с помощью if constexpr (новый способ, начиная с C++17)

В стандартной библиотеке есть функции std::advance и std::distance, которые позволяют сделать следующее:

- Функция std::advance берет итератор и продвигает его на указанное количество шагов.

```

std::list<int> v = {1, 2, 3, 4, 5};
std::list<int>::iterator it = v.begin();
std::advance(it, 3);
std::cout << *it;

```

Выведет 4

- Функция std::distance берет два итератора и считает расстояние между ними (сколько шагов нужно пройти, чтобы пройти от одного итератора до другого)

```

std::list<int>::iterator it2 = v.end();
std::cout << std::distance(it, it2);

```

Выведет 2

```

std::list<int>::iterator it2 = v.end();
std::cout << std::distance(it2, it);

```

А вот так UB

Как реализовать advance? Ведь если мы передаем ей forward/bidirectional итераторы, то она сможет отработать только за линейное время прибавлением по единичке, но если мы передаем ей random access iterator, то такое прибавление можно сделать за $O(1)$.

Возникает проблема, как понять, какой именно итератор передан в функцию на вход, чтобы сделать реализацию наиболее эффективной.

В стандартной библиотеке есть такая структура std::iterator_traits

Среди прочего, в нем определены

Member types	
Member type	Definition
difference_type	Iter::difference_type
value_type	Iter::value_type
pointer	Iter::pointer
reference	Iter::reference
iterator_category	Iter::iterator_category

- **value_type** - тот тип, на который ссылается итератор
- **pointer** - C-style указатель на объект под итератором
- **reference** - ссылка на объект под итератором
- **iterator_category** - это некоторый typedef, который позволяет понять, какой категории итератор нам дан

Попробуем реализовать нашу функцию

```
template <typename Iterator>
void my_advance(Iterator& iter, int n) {
    if (std::is_same_v<typename std::iterator_traits<Iterator>::iterator_category, std::random_access_iterator_tag>) {
        iter += n;
    } else {
        for (int i = 0; i < n; ++i, ++iter);
    }
}
```

Это не работает!

В чем, собственно, проблема? Проблема в том, что мы логически понимаем, что в ветку if мы не войдем, а компилятор не может это проверить на этапе компиляции и считает, что мы пытаемся сделать += к итератору, для которого данная функция не определена

Как нужно было делать (до C++17)

```
template <typename Iterator, typename IterCategory>
void my_advance_helper(Iterator& iter, int n, IterCategory) {
    for (int i = 0; i < n; ++i, ++iter);
}

template <typename Iterator>
void my_advance_helper(Iterator& iter, int n, std::random_access_iterator_tag) {
    iter += n;
}

template <typename Iterator>
void my_advance(Iterator& iter, int n) {
    my_advance_helper(iter, n, typename std::iterator_traits<Iterator>::iterator_category());
}
```

Как нужно делать (с C++17)

```
1 template <typename Iterator>
2 void my_advance (Iterator& it, int n) {
3     if constexpr ( std::is_same_v<typename std::iterator_traits<Iterator>::
4         iterator_category, std::random_access_iterator_tag>) {
5         it += n;
6     } else {
7         for (int i = 0; i < n; ++i; ++it);
8     }
9 }
```

constexpr означает не компилировать одну из веток if, если условие ложно, но условие должно проверяться в compile time

Extra

reverse_iterator

reverse_iterator - это такой итератор, который делает то же самое, что и обычный итератор, но в порядке не слева направо, а справа налево

```
1 using reverse_iterator = std::reverse_iterator<iterator>;
2 using const_reverse_iterator = std::reverse_iterator<const_iterator>;
3
4 reverse_iterator rbegin() {
5     return reverse_iterator(end());
6 }
7 reverse_iterator rend() {
8     return reverse_iterator(begin());
9 }
10
11 const_reverse_iterator rbegin() const {
12     return const_reverse_iterator(end());
13 }
14 const_reverse_iterator rend() const {
15     return const_reverse_iterator(begin());
16 }
17
18 const_reverse_iterator crbegin() const {
19     return const_reverse_iterator(end());
20 }
21 const_reverse_iterator crend() const {
22     return const_reverse_iterator(begin());
23 }
```

У reverse_iterator есть функция base(), возвращающая нормальный итератор, от которого взят reverse_iterator

back_inserter

```
1 template <typename Container>
2 class back_insert_iterator{
3     Container& container;
4 public:
5     back_insert_iterator(Container& container): container(container){}
6
7     back_insert_iterator<Container>& operator++(){
8         return *this;
9     }
10
11     back_insert_iterator<Container>& operator*(){
12         return *this;
13     }
14     back_insert_iterator<Container>& operator=(const typename Container::
15     value_type& value){
16         container.push_back(value);
17         return *this;
18     }
19 }
20
21 template <typename Container>
22 back_insert_iterator<Container> back_inserter(Container& container){
23     return back_insert_iterator(container);
24 }
```

istream_iterator

```
std::vector<int> v;

std::istream_iterator<int> it(std::cin);

for (int i = 0; i < 5; ++i, ++it) {
    v.push_back(*it);
}

for (int i = 0; i < 5; ++i) {
    std::cout << v[i] << ' ';
}
```

Зачем это надо? В стандартных алгоритмах, в которых требуются итераторы, не обязательно предоставлять итераторы на контейнеры, достаточно передать итератор на поток ввода

```
template <typename T>
class istream_iterator {
    std::istream& in;
    T value;
public:
    istream_iterator(std::istream& in): in(in) {
        in >> value;
    }

    istream_iterator<T>& operator++() {
        in >> value;
    }

    T& operator*() {
        return value;
    }
};

// std::ifstream in("input.txt");
// std::istringstream iss(s);
```