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

```
template <bool IsConst>
   class myiterator {
   public:
       Node* iter = NULL;
       using difference_type = std::ptrdiff_t;
       using value_type = std::conditional_t<IsConst, const T, T>;
       using pointer = std::conditional_t<IsConst, const T*, T*>;
       using reference = std::conditional_t<IsConst, const T&, T&>;
       using iterator_category = std::bidirectional_iterator_tag;


       myiterator() {}
       myiterator(Node* iter) : iter(iter) {}
       reference operator*() {
           return (iter->data);
       }

       pointer operator->() { return &(iter->data); }

       myiterator& operator ++() {
           Node* ptr = iter->next;
           iter = ptr;
           return *this;
       }

       myiterator operator ++(int) {
           myiterator copy = *this;
           Node* ptr = iter->next;
           iter = ptr;
           return copy;
       }

       myiterator& operator --() {
           iter = iter->prev;
           return *this;
       }

       bool operator !=(const myiterator& other) {
           return !(iter == other.iter);
       }
       bool operator ==(const myiterator& other) {
           return (iter == other.iter);
       }
```

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

Реализация std::conditional

```
template<bool condition, typename T, typename F>
struct conditional{
    using type = F;
}
template<bool condition,typename T, typename F>
struct conditional<true, F, T>{
    using type = T;
}
template<bool condition, typename T, typename F>
using conditional_t = typename conditional<condition,T,F>::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

https://en.cppreference.com/w/cpp/iterator/iterator_traits

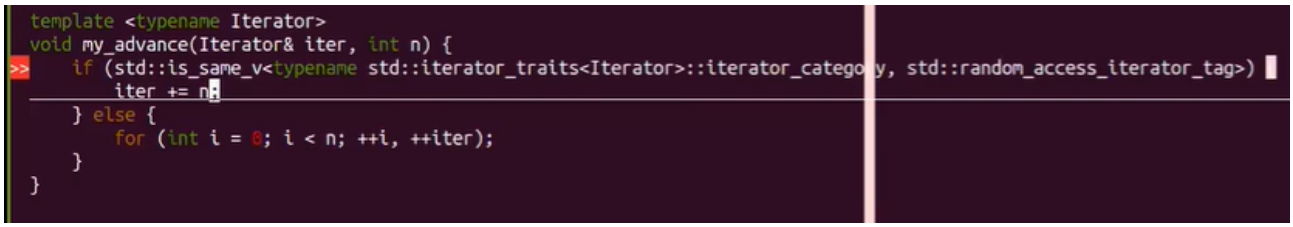
Среди прочего, в нем определены

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

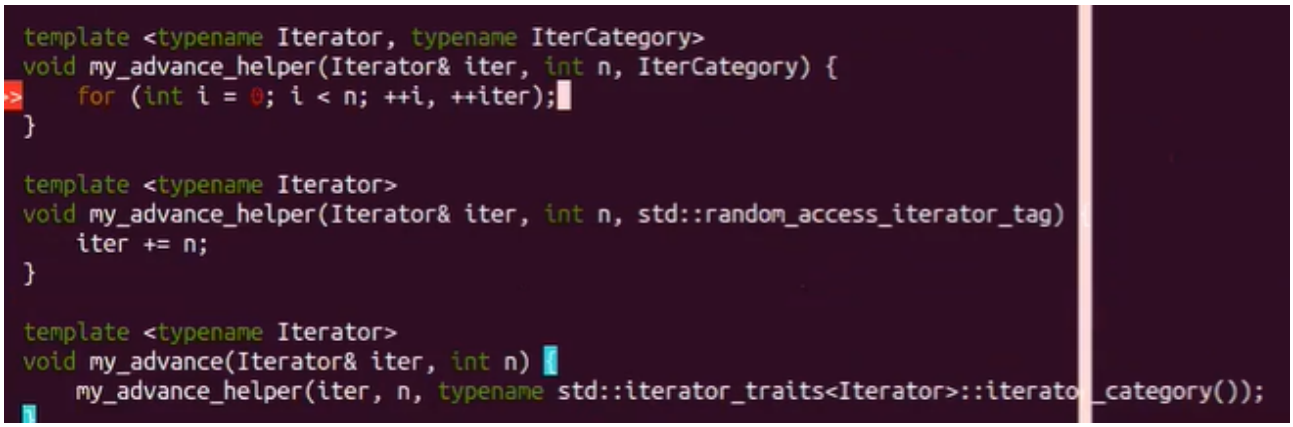
Попробуем реаллизовать нашу функцию



Это не сработает!

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)



Как нужно делать (с C++17)

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

constexp 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 справо налево

```
using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

     reverse_iterator rbegin() {
        return reverse_iterator(end());
    }
    reverse_iterator rend() {
        return reverse_iterator(begin());
    }

    const_reverse_iterator rbegin() const {
        return const_reverse_iterator(end());
    }
    const_reverse_iterator rend() const {
        return const_reverse_iterator(begin());
    }

    const_reverse_iterator crbegin() const {
        return const_reverse_iterator(end());
    }
    const_reverse_iterator crend() const {
        return const_reverse_iterator(begin());
    }
```

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

back_inserter

```
template <typename Container>
class back_insert_iterator{
    Container& container;
public:
    back_insert_iterator(Container& container): container(container){}

    back_insert_iterator<Container>& operator++(){
        return *this;
    }

    back_insert_iterator<Container>& operator*(){
        return *this;
    }
    back_insert_iterator<Container>& operator=(const typename Container::
   value_type& value){
        container.puch_back(value);
        return *this;
    }
}

template <typename Container>
back_insert_iterator<Container> back_inserter(Container& container){
    return back_insert_iterator(container);
}
```

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);
```