Контейнеры

Контейнер - holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.

- Последовательные
- Ассоциативные

Контейнер	push_back pop_back	push_front pop_back	[index] at	insert erase	find
vector	O*(1)/O(1)		O(1)	O(n)	
deque	O*(1)/O(1)	O*(1)	O(1)	O(n)	
list forward_list	O(1)	O(1)		O(1)	
map set			O(log n)	O(log n)	O(log n)
unordered_map unordered_set			O(1) expected	O(1) expected	O(1) expected

vector

Не надо заводить указатели и ссылки на вектор - инвалидация указателей и ссылок

Поля: array, size, capacity

reinterpret_cast for memory methods

```
T* newarr = reinterpret_cast<T*>(new char[count * sizeof(T)]);
```

placement new

Направляем на сырую память конструктор `T`. Не выделяет память.

```
new (newarr + i) T(arr[i]);
```

`memcpy` нельзя, так как в `T` могут быть указатели на свои поля.

Удаление элементов и присваивание нового массива

```
(arr + i) -> ~T();
delete[] reinterpret_cast<char*>(arr);
arr = newarr;
```

pop_back

Уменьшаем размер и вызываем деструктор.

vector<bool>

8 элементов - 1 байт.

```
std::vector<bool> b(5);
f(b[0]); // f: srting -> None
```

Получаем `CE` и видим, что тип `b[0] - std::vector<bool>::reference` .

```
template<>
class Vector<bool> {
  private:
    char* arr;
    size_t sz;
    size_t cap;

    struct BoolReference {
        char* cell;
        uint8_t index;

        BoolReference operator=(bool b) { // return copy
            if (b) {
                *cell |= (1 << index);
            } else {
                *cell &= ~(1 << index);
            }
            return *this;
        }

        operator bool() const {
            return *cell & (1 << index);
        }
    };

  public:
    BoolReference operator[](size_t index) {
        return BoolReference{arr + index / 8, index % 8};
```

```
    }
};
```

deque

Гарантирует отсутствие инвалидации указателей и ссылок, но не итераторов

Хранение: чанки (размер фиксированный) во внешнем массиве. При изменении размера изменяем внешний массив, но не внутренние.

`pop_...` не реаллоцирует память.

`std::deque<bool>` хранит обычные `bool`.

Контейнеры над deque

- stack, но можно `std::stack<int, std::vector<int>>`
- queue
- priority_queue

list

Гарантирует отсутствие инвалидации указателей, ссылок и итераторов

Строение: одна за другой ноды. Закольцованная фейк-нода (может лежать на стеке). Нода - наследник от база-ноды (добавляется значение в ноде).

Некоторые методы

- `insert(it)` - перед итератором
- `sort()` - mergesort, так как есть только ForwardIterator, для qsort нужен RandomAccessIterator
- `reverse()` тоже свой
- `splice()` - вклеиваем часть одного лиса в другой

`std::sort()`, `std::reverse()` не работают

forward_list

Только вперёд!

Нет `..._back`.

Нет фейковой вершины.

map

Гарантирует отсутствие инвалидации указателей, ссылок и итераторов (set тоже)

По ключам хранит значения. Красно-чёрное дерево.

```
template <typename Key, typename Value, typename Compare = std::less<Key>, typ
```

Поля: система нод, как в листе, однако храним в ноде флаг `red` и пару (это существенно для итератора) ключ[конст]-значение. Фиктивный корень, вся структура - левая вершина. Ещё есть компаратор в шаблонных параметрах.
`BaseNode* begin_` - указатель на самого левого сына
`BaseNode* end_` - указатель на фиктивный корень

Инкремент/декремент итератора - `O(log n)`, обход дерева - `O(n)`.

Методы

- `[] notconst` или `at` - если нет ключа, вызываем конструктор по умолчанию и добавляем элемент
- `find(key)` возвращает итератор или `end()`, если ключ не найден. Работает быстрее, чем `[]` или `at`.
- `insert({key, value})` возвращает пару: итератор и `bool`, удалась ли вставка. Можно ещё вставлять по итератору (вставит в ближайшую позицию к `*iter`).
- `erase(key/iter)` возвращает пару: следующий итератор и `bool`, удалось ли удаление
- `lower_bound/upper_bound(key)` возвращает итератор на наименьший элемент, больший или равный / больший данного

unordered_map

Гарантирует отсутствие инвалидации указателей, ссылок, но не итераторов

Умеет всё то же, что и map, только ключи не упорядочены.

```
template <typename Key, typename Value, typename Hash = std::hash<Key>, typena
```

Хэширует методом цепочек. Реализован через массив + forward_list с buckets. В массиве лежат указатели на вершины листа. В вершинах храним предвычисленный

хэш, чтобы понимать, где кончился бакет.

Как вставлять? Новый бакет в начале листа.

Как удалять? Указатели в массиве на концы бакетов.

Методы:

- `bucket_count()` - количество бакетов
- `reserve(size_t count)` - сделать размер таким, чтобы можно было хранить `count` элементов без коллизий с учётом `load factor`
- `load_factor() = количество элементов / количество бакетов` - характеристика числа коллизий
  - `max_load_factor(value)` - устанавливает максимальный, после которого будет рехэш.
  - Если хэш храним полностью, то во время рехэша не надо будет обращаться к `Hash`

Итераторы

Есть у любого контейнера, утиная типизация

```
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++i) {
    std::cout << *it << ' ';
}
```

Виды итераторов

```
Iterator ----------------------------------
|                                         |
InputIterator (==, *, ++): input stream   OutputIterator
|
ForwardIterator (can walk many times on sequence): forward_list/list, unordere
|
BidirectionalIterator (--): list, map/set
|
RandomAccessIterator (+=, <, -): deque
|
ContiguousIterator (&*it += n - ok: memory in a row): vector, c-pointers
```

copy

Вывод:

```
std::ostream_iterator ot(std::cout, ", ");
std::copy(v.begin(), v.end(), ot);
```

Копирование в контейнер:

```
std::copy(v.begin(), v.end(), std::back_inserter(v2));
std::copy(v.begin(), v.end(), std::front_inserter(d2));
std::copy(v.begin(), v.end(), std::inserter(v2, it));
```

ostream_iterator

```
template <typename T>
struct ostream_iterator {
    std::ostream& ostr;
    ostream_iterator(std::ostream& ostr): ostr(ostr) {}

    ostream_iterator& operator++() {
        return *this;
    }

    ostream_iterator& operator=(const T& value) {
        ostr << value;
        return *this;
    }

    ostream_iterator& operator*() {
        return *this;
    }
}
```

std::advance

Позволяет добавить число к итератору, даже если не RandomAccess.

```
std::advance(it, 4);
```

Реализация:

```
template <typename Iterator>
void advance(Iterator& iter, size_t n) {
    if constexpr (std::is_base_of_v<
```

```
    std::random_access_iterator_tag,
    typename std::iterator_traits<Iterator>::iterator_category>) { // return t
        iter += n;
    } else {
        for...
    }
}
```

const and reverse iterators

const_iterators don't allow you to change the values that they point to, regular iterators do.

reverse_iterators - mirror of usual.