Язык C++

Мещерин Илья

Лекция 15

9.4$\frac{1}{3}$) std::set

Ассоциативный контейнер, который содержит упорядоченный набор уникальных объектов типа *Key*. Сортировка элементов осуществляется применением функции *Compare* к ключам множества. Операции поиска, удаления и вставки имеют логарифмическую сложность. Данный тип обычно реализуется как красно-черные деревья.

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

Есть методы *insert(), erase(), begin(), end(), find()* (проверяет есть ли элемент в множестве, и если есть, то возвращает итератор на него, а иначе возвращает итератор на элемент, следующий за последним элементом контейнера), *size(), clear()*.

```
s.find(x) != s.end()
```

Так можно проверить есть ли элемент x в множестве s.

Метод *count(key)* возвращает количество элементов с ключом *key* (в *std::set* возвращает 0 или 1).

9.4$\frac{2}{3}$) std::multiset, std::multimap

В отличие от *std::set* (*set::map*) в *std::multiset* (*set::multimap*) допускаются ключи с одинаковыми значениями.

Метод *iterator lower_bound(key)* возвращает итератор, указывающий на первый элемент, который является не меньше, чем *key*. Если такой элемент не найден то возвращает итератор на элемент, следующий за последним элементом контейнера.

Метод *iterator upper_bound(key)* возвращает итератор, указывающий на первый элемент, больший, чем *key*. Аналогично поведение, если такой элемент не найден.

Метод *std::pair<iterator, iterator> equal_range(Key)* возвращает набор элементов для конкретного ключа.

Эти три методы есть и у обычных *std::set, std::map*.

9.5) std::unordered_map

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
```

Являются реализацией хеш-таблиц, поэтому операции поиска, вставки и удаления выполняются за константное время.

KeyEqual используется для проверки ключей на равенство.

Аналогично существуют *unordered_set, unordered_multimap, unordered_multiset*.

Так же есть методы позволяющие работать с бакетами и *load_factor*.

Итераторы

10.1) Общие слова об итераторах

Главное предназначение итераторов заключается в предоставлении возможности пользователю обращаться к любому элементу контейнера при сокрытии внутренней структуры контейнера от пользователя.

```
std::map<int, string> m;
std::map<int, string>::iterator it = m.begin();
```

Любой итератор можно инкрементировать и разыменовать.

10.2) Виды итераторов

InputIterator.

Нельзя делать декремент, нельзя создавать копию такого итератора, чтобы несколько раз пробежаться по данным (неопределенное поведение). Можно инкрементировать, разыменовать (но нельзя менять объект, полученный при разыменовании) и сравнить на равенство (с помощью *operator==, operator!=*).

ForwardIterator(однонаправленный итератор, частный случай InputIterator).

Главным отличием от InputIterator является то, что можно несколько раз пробегаться по содержимому, например, создавая копию. Используется в *forward_list* и *unordered_map*.

BidirectionalIterator(частный случай ForwardIterator).

Позволяет делать декремент. Используется в *std::set*, *std::map* и *std::list*.

RandomAccessIterator(частный случай BidirectionalIterator)

Можно сравнивать итераторы такого типа, прибавлять/вычитать число, вычитать итераторы друг из друга, использовать *operator[]*. Используется в *std::vector, std::deque*.

OutputIterator

Позволяет изменять объект, полученный после разыменования итератора. Можно делать инкремент, но нельзя создавать копию для нескольких проходов. Операция чтения полученного значения после разыменования не всегда является допустимой. Тот факт, что итератор является OutputIterator'ом никак не зависит от предыдущей классификации.

10.3) *const* и *reverse* итераторы

```
std::map<int, string> m;
for (std::map<int, string>::iterator it = m.begin(); it != m.end(); ++it)
```

Пример цикла по контейнеру.

```
for (std::map<int, string>::const_iterator it = m.cbegin(); it != m.cend(); ++it)
```

Чтобы запретить менять элементы при проходе по ним, можно завести константные итераторы.

```
for (std::map<int, string>::reverse_iterator it = m.rbegin(); it != m.rend(); ++it)
```

Можно обойти контейнер в обратном порядке (только для BidirectionalIterator).

```
for (std::map<int, string>::const_reverse_iterator it = m.crbegin(); it != m.crend(); ++it)
```

И их комбинация.

```
template<class BidirectionalIterator>
class my_reverse_iterator{
    BidirectionalIterator it;
public:
   my_reverse_iterator(const BidirectionalIterator it): it(it) {}
```

```
    my_reverse_iterator<BidirectionalIterator>& operator++(){
        --it;
        return *this;
    }
    typename BidirectionalIterator::value_type operator*() const{
        return *it;
    }
};
```

Пример реализации обертки над обычными итераторами для реализации *reverse* итераторов.

10.4) std::iterator_traits

Помогает автоматически определить *iterator_tag* (классификация итераторов в начале параграфа).

10.5) std::distance, std::advance

Функция std::distance(first, last) возвращает количество инкрементов итератора *first*, чтобы дойти до *last*.

Функция std::advance(it, k) *k* инкрементирует итератор *it*.

В зависимости от того, какие итераторы были переданы, эти функции должны по-разному реализовываться.

```
1  template<class Iterator, class tmp>
2  size_t distance_impl(Iterator first , Iterator second, tmp){
3      size_t ans = 0;
4      while(first != second) first++, ans++;
5      return ans;
6  }
7
8  template<class Iterator>
9  size_t distance_impl(Iterator first , Iterator second, random_access_iterator_tag){
10     return second - first;
11 }
12
13 template<class Iterator>
14 size_t dist(Iterator first , Iterator second){
15     return distance_impl(first, second, typename std::iterator_traits<Iterator>::iterator_category());
16 }
17
18 int main(){
19     list <int> a;
20     vector<int> b;
21     dist(a.begin(), a.end());
22     dist(b.begin(), b.end());
23 }
```

В данном случае для *std::list* вызовется первая версия функции *distance_impl()*, а для *std::vector* вторая. В функции *dist()* мы вызываем конструктор от типа *std::iterator_traits<Iterator>::iterator_category*.