КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

Лекция № 19.5 — Контейнеры и алгоритмы.

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2021

Оглавление

Библиотека контейнеров

Набор (set)

Библиотека содержит шаблоны двух классов:
set — набор (множество) и **multiset** — набор с несколькими ключами (мультимножество).

Наборы — это контейнеры, в которых хранятся *уникальные* элементы в определенном порядке.
Никакие два элемента в контейнере set не могут быть эквивалентными.

```
template < class T,                          // set::key_type/value_type
           class Compare = less<T>,          // set::key_compare/value_compare
           class Alloc = allocator<T>        // set::allocator_type
           > class set;
```

T — Тип элементов. Каждый элемент в контейнере набора однозначно идентифицируется этим значением — каждое значение само по себе также является ключом элемента.

Значение элементов в наборе не может быть изменено — элементы всегда являются константными, но они могут быть вставлены или удалены из контейнера.

Внутренне элементы в наборе всегда сортируются в соответствии с определенным строгим критерием слабого порядка, указанным его внутренним объектом сравнения (типа **Compare**).

Compare — Двоичный предикат, который принимает два аргумента того же типа, что и элементы, и возвращает логическое значение. Выражение **comp(a, b)**, где **comp** — объект типа **Compare**, а **a** и **b** — ключевые значения, должно возвращать истину, если считается, что **a** идет перед **b** в строгом слабом порядке, определяемом данной функцией.

Общедоступные функции-члены

(**constructor**)	создает набор
(**destructor**)	разрушает набор
operator=	присваивает содержимое контейнеру
begin()/end()	возвращает итератор на начало/конец набора
rbegin()/rend()	возвращает реверсивный итератор на начало/конец набора
cbegin()/cend()/crbegin()/crend()	— константные версии
empty()	проверка на пустоту
size()	возвращает количество элементов в контейнере
max_size()	возвращает максимально допустимое количество элементов в контейнере
insert()/erase()	вставляет/удаляет элемент
swap()	обменивает содержимым два контейнера
clear()	очищает содержимое контейнера
emplace()	создает и вставляет элемент
emplace_hint()	создает и вставляет элемент с подсказкой
key_comp()	возвращает объект, использующийся для сравнения
value_comp()	возвращает объект, использующийся для сравнения

`find()`	получить итератор на элемент
`count()`	количество элементов с указанным значением (1 или 0)
`lower_bound()`	возвращает итератор на нижнюю границу
`upper_bound()`	возвращает итератор на верхнюю границу
`equal_range()`	получить диапазон одинаковых элементов
`get_allocator()`	получить аллокатор

multiset — набор с несколькими ключами (мультимножество)

Мультимножества — это контейнеры, в которых элементы хранятся в определенном порядке, и в которых *несколько элементов могут иметь эквивалентные значения*.

В мультимножестве, как и в наборе, значение элемента идентифицирует сам элемент (значение само является ключом типа **T**).

Значение элементов в мультимножестве нельзя изменить за одну операцию (элементы всегда являются константными), но они могут быть удалены из контейнера или вставлены.

Внутренне элементы мультимножества всегда сортируются в соответствии с определенным строгим критерием слабого упорядочения, указанным его внутренним объектом сравнения (типа **`Compare`**), аналогично **`set`**. Мультимножества обычно реализуются как деревья двоичного поиска.

Свойства

1) На элементы в ассоциативных контейнерах ссылаются по их ключу, а не по их абсолютному положению в контейнере.

2) Элементы в контейнере всегда следуют строгому порядку. Всем вставленным элементам в этом порядке присваиваются позиции.

3) Значение элемента также является ключом, используемым для его идентификации.

4) Несколько элементов в контейнере могут иметь эквивалентные ключи.

5) Контейнер использует объект-аллокатор для динамической обработки своих потребностей в памяти.

```
template < class T,                          // multiset::key_type/value_type
           class Compare = less<T>,          //
multiset::key_compare/value_compare
           class Alloc = allocator<T> >      // multiset::allocator_type
           > class multiset;
```

T — Тип элементов.

Каждый элемент в контейнере набора однозначно идентифицируется этим значением — каждое значение само по себе также является ключом элемента.

Псевдонимы в качестве типа элементов **multiset::key_type** и **multiset::value_type**.

Compare — Двоичный предикат, который принимает два аргумента того же типа, что и элементы, и возвращает логическое значение.

Выражение **comp(a, b)**, где **comp** — объект типа **Compare**, а **a** и **b** — ключевые значения, должно возвращать истину, если считается, что **a** идет перед **b** в строгом слабом порядке, определяемом данной функцией.

Объект мультинабора (мультимножества) использует это выражение, чтобы определить как порядок следования элементов в контейнере, так и эквивалентность двух ключей элементов (путем рефлексивного сравнения — они эквивалентны, если **!Comp(a, b) && !Comp(b, a))**.

Выражение **comp(a, b)** может быть указателем на функцию или функциональный объект.

По умолчанию используется **less<T>**, которое возвращает то же самое, что и применение оператора «меньше» (**a < b**).

Псевдонимы для типа элементов **multiset::key_compare** и **multiset::value_compare**.

`Alloc` — Тип объекта аллокатора, используемого для выделения памяти. По умолчанию используется шаблон класса **`allocator`**, который определяет простейшую модель распределения памяти. Псевдоним в качестве типа элемента **`multiset::allocator_type`**.

Типы членов — Те же самые, что и для **`std::set`** (смотрим туда)

Общедоступные функции-члены

(**constructor**)	создает многоключевой набор
(**destructor**)	разрушает многоключевой набор
operator=	присваивает содержимое контейнеру
begin()/end()	возвращает итератор на начало/конец набора
rbegin()/rend()	возвращает реверсивный итератор на начало/конец набора
cbegin()/cend()/crbegin()/crend()	— константные версии
empty()	проверка на пустоту
size()	возвращает количество элементов в контейнере
max_size()	возвращает максимально допустимое количество элементов в контейнере
insert()/erase()	вставляет/удаляет элемент
swap()	обменивает содержимым два контейнера
clear()	очищает содержимое контейнера
emplace()	создает и вставляет элемент
emplace_hint()	создает и вставляет элемент с подсказкой
key_comp()	возвращает объект, использующийся для сравнения
value_comp()	возвращает объект, использующийся для сравнения

`find()`	получить итератор на элемент
`count()`	количество элементов с указанным значением (1 или 0)
`lower_bound()`	возвращает итератор на нижнюю границу
`upper_bound()`	возвращает итератор на верхнюю границу
`equal_range()`	получить диапазон одинаковых элементов
`get_allocator()`	получить аллокатор

Конструкторы std::multiset

(1) Пустой

```
explicit multiset(const key_compare& comp = key_compare( ),
                  const allocator_type& alloc = allocator_type( ));
explicit multiset(const allocator_type& alloc);
```

Конструктор пустого контейнера. Создает пустой контейнер без элементов.

comp — двоичный предикат, который, принимая два значения того же типа, что и те, которые содержатся в многоключевом наборе, возвращает истину, если первый аргумент идет перед вторым аргументом в строгом слабом порядке (который он определяет) и ложь в противном случае.

comp должен быть указателем на функцию или функциональный объект.

Тип элемента key_compare — это тип внутреннего объекта сравнения, используемый контейнером. Определен в **multiset** как псевдоним его второго параметра шаблона (**Compare**).

alloc — объект аллокатора

Контейнер хранит и использует его внутреннюю копию.

Тип элемента allocator_type — это тип внутреннего аллокатора, используемого контейнером. Определен в **set** как псевдоним его третьего параметра шаблона (**Alloc**).

(2) Диапазонный

```
template <class InputIterator>
multiset(InputIterator first, InputIterator last,
         const key_compare& comp = key_compare( ),
         const allocator_type& = allocator_type( ));
```

Создает контейнер с таким количеством элементов, которое содержится в диапазоне **[first,last)**, причем каждый элемент создается на месте (empalce) из соответствующего элемента этого диапазона.

first, **last** — итератор ввода в начальную и конечную позиции из диапазона.

Используемый диапазон **[first,last)** включает все элементы между **first** и **last**, включая элемент, на который указывает **first**, но не включая элемент, на который указывает **last**.

Аргумент шаблона функции **InputIterator** должен быть типом итератора ввода, который указывает на элементы типа, из которого могут быть построены объекты **value_type**.

(3) Конструктор копирования (и копирования с аллокатором)

```
multiset(const multiset& x);
multiset(const multiset& x, const allocator_type& alloc);
```

Создает контейнер с копией каждого элемента из **x**.

x — другой объект набора того же типа (с теми же аргументами шаблона класса **T**, **Compare** и **Alloc**), содержимое которого либо копируется, либо принимается набором назначения.

(4) Конструктор перемещения (с перемещением аллокатора)

```
multiset(set&& x);
multiset(set&& x, const allocator_type& alloc);
```

Создает контейнер, в который перемещаются элементы из **x**.

Если указано значение **alloc** и оно отличается от аллокатора, который использовался в **x**, элементы перемещаются реально. В противном случае никакие элементы не создаются (их принадлежность контейнеру передается напрямую). Следует заметить, что **x** остается в неопределенном, но допустимом состоянии.

Данный конструктор портит все итераторы, указатели и ссылки на **x** в случае реального перемещения элементов.

(5) Из списка инициализации

```
multiset(initializer_list<value_type> il,
         const key_compare& comp = key_compare( ),
         const allocator_type& alloc = allocator_type( ));
```

Создает контейнер с копией каждого из элементов **il**.

il — объект типа `initializer_list`.

Эти объекты автоматически создаются из деклараторов списка инициализаторов.

Тип элемента **value_type** — это тип элементов в контейнере, определенный в **multiset** как псевдоним его первого параметра шаблона (**T**).

Контейнер хранит внутреннюю копию **alloc**, которая используется для выделения и освобождения памяти для его элементов, а также для их создания и уничтожения (как указано в его allocator_traits).

Если конструктору не передается аргумент **alloc**, используется распределитель, созданный по умолчанию, за исключением следующих случаев:

1) Конструктор копирования (3) создает контейнер, который хранит и использует копию аллокатора, возвращаемую вызовом типажа **selected_on_container_copy_construction()** для аллокатора из **x**.

2) Конструктор перемещения (4) получает аллокатор объекта **x**.

Контейнер также хранит внутреннюю копию **comp** (или объекта сравнения объекта **x**), которая используется для определения порядка элементов в контейнере и для проверки эквивалентности элементов.

Все элементы копируются, перемещаются или иным образом создаются путем вызова **allocator_traits::construct** с соответствующими аргументами.

Пример создания многоключевого набора

```
#include <iostream>
#include <set>

bool fncomp(int lhs, int rhs) {return lhs < rhs;}      // функция сравнения

struct classcomp {                                     // функциональный объект
  bool operator() (const int& lhs, const int& rhs) const {return lhs < rhs;}
};

int main () {

  std::multiset<int> first;                    // пустой набор из int

  int myints[] = {10, 20, 30, 20, 20};         // массив из списка инициализации
  std::multiset<int> second(myints, myints + 5); // указатель как итератор

  std::multiset<int> third(second);            // копия second
  std::multiset<int> fourth(second.begin(), second.end()); // с итераторами
  std::multiset<int, classcomp> fifth;         // класс в качестве Compare

  bool(*fn_pt)(int, int) = fncomp;             // указатель на функцию fncomp()
  std::multiset<int, bool(*)(int, int)>        // указатель на функцию
  sixth(fn_pt);                                // в качестве Compare
 }
```

`begin()` — возвращает итератор на начало

Поскольку в контейнерах набора элементы всегда упорядочены, **`begin()`** указывает на элемент, который идет первым в соответствии с критерием сортировки контейнера.

Если контейнер пуст, возвращаемое значение итератора не может быть разыменовано.

Если объект набора квалифицируется как **`const`**, функция возвращает **`const_iterator`**. В противном случае он возвращает **`iterator`**.

Типы элементов **`iterator`** и **`const_iterator`** - это двунаправленные типы итераторов, указывающие на элементы.

`end()` — возвращает итератор на конец

Итератор указывает на теоретический элемент, который будет следовать за последним элементом в этом контейнере.

Он не указывает на какой-либо элемент, и поэтому не может быть разыменован.

Если контейнер пуст, эта функция возвращает то же, что и **`multiset::begin()`**.

Поскольку диапазоны, используемые функциями стандартной библиотеки, не включают элемент, на который указывает их закрывающий итератор, эта функция часто используется в сочетании с **`multiset::begin()`** для указания диапазона, включающего все элементы в контейнере.

Пример

```
#include <iostream>
#include <set>

int main () {

  int myints[] = {42, 71, 71, 71, 12};
  std::multiset<int> mymultiset(myints, myints + 5);

  std::multiset<int>::iterator it;

  std::cout << "mymultiset содержит:";

  for (auto it = mymultiset.begin(); it != mymultiset.end(); ++it)
    std::cout << ' ' << *it;

  std::cout << '\n';

  return 0;
}
```

Вывод

```
mymultiset содержит: 15 39 77 77 98
```

insert() — вставить элементы

Функция либо копирует, либо перемещает в контейнер уже существующие объекты.

Замечания:

C++98: Нет никаких гарантий в отношении относительного порядка эквивалентных элементов.

C++11: Относительный порядок эквивалентных элементов сохраняется, и вновь вставленные элементы следуют за своими эквивалентами, уже находящимися в контейнере.

(1) Один элемент

```
iterator insert(const value_type& val);
iterator insert(value_type&& val);
```

`val` — значение, которое нужно скопировать (или переместить) во вставленные элементы.

Тип элемента **`value_type`** — это тип элементов в контейнере, определенный в **`multiset`** как псевдоним его первого параметра шаблона (**`T`**).

(2) С указанием куда вставлять

```
iterator insert(const_iterator position, const value_type& val);
iterator insert(const_iterator position, value_type&& val);
```

position — подсказка насчет места, куда должен вставляться элемент.

Замечания

C++98: Функция оптимизирует время вставки, если позиция указывает на элемент, который будет предшествовать вставленному элементу.

C++11: Функция оптимизирует время вставки, если позиция указывает на элемент, который будет следовать за вставленным элементом (или на конец, если он будет последним).

Это всего лишь подсказка и она не заставляет вставлять новый элемент в указанную позицию внутри набора (элементы в наборе всегда следуют определенному порядку).

Типы элементов **iterator** и **const_iterator** определены (в **multimap**) как двунаправленный тип итератора, указывающий на элементы.

(3) Из диапазона

```
template <class InputIterator>
void insert(InputIterator first, InputIterator last);
```

Аргумент шаблона функции **InputIterator** должен быть типом итератора ввода, который указывает на элементы типа, из которого могут быть построены объекты **value_type**.

(4) Из списка инициализации

```
void insert(initializer_list<value_type> il);
```

initializer_list — вставляются копии элементов из этого списка

Эти объекты автоматически создаются из деклараторов списка инициализации.

Тип элемента **value_type** — это тип элементов в контейнере, определенный в **multiset** как псевдоним его первого параметра шаблона (**T**).

Возвращаемое значение

В версиях, возвращающих значение, возвращается двунаправленный итератор, указывающий на вновь вставленный элемент в многоключевой набор.

Пример `multiset::insert` (C++98)

```
#include <iostream>
#include <set>

int main () {

  std::multiset<int> mymultiset;
  std::multiset<int>::iterator it;

  // что-нибудь вставим:
  for (int i = 1; i <= 5; i++) mymultiset.insert(i*10);  // 10 20 30 40 50
  it = mymultiset.insert(25);         // it <- 25
  it = mymultiset.insert(it, 27);     // максимальная эффективность 27 > it
  it = mymultiset.insert(it, 29);     // максимальная эффективность 29 > it
  it = mymultiset.insert(it, 24);     // а тут не эффективно        24 < 29

  int myints[] = {5, 10, 15};
  mymultiset.insert(myints, myints + 3);

  std::cout << "mymultiset содержит:";
  for (it = mymultiset.begin(); it != mymultiset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

}
```

myset содержит: 5 10 10 15 20 24 25 27 29 30 40 50

erase() — удалить элементы

Удаляет элементы из контейнера **multiset**.

Операция уменьшает размер контейнера на количество удаленных элементов, которые разрушаются.

```
(1)       void erase(const_iterator position);                          // C++98
(1)   iterator erase(const_iterator position);                          // C++11
(2)  size_type erase(const value_type& val);
(3)       void erase(const_iterator first, const_iterator last); // C++98
(3)   iterator erase(const_iterator first, const_iterator last); // C++11
```

position — итератор, указывающий на единственный элемент, который нужно удалить из мультимножества.

val — значение, которое нужно удалить из мультимножества. Все элементы со значением, эквивалентным этому, удаляются из контейнера.

Тип элемента **value_type** — это тип элементов в контейнере, определенный в мультимножестве как псевдоним его первого параметра шаблона (**T**).

first, last — итераторы, определяющие диапазон в удаляемом контейнере мультимножества — **[first, last)**, т.е. диапазон включает все элементы между первым и последним, включая элемент, на который указывает **first**, и не включая тот, на который указывает **last**.

Типы элементов **iterator** и **const_iterator** — это типы двунаправленных итераторов, которые указывают на элементы.

Возвращаемое значение

Для версии на основе значения (2) функция возвращает количество удаленных элементов.

C ++ 98: версии (1) и (3) не возвращают никаких значений.

C ++ 11: версии (1) и (3) возвращают итератор на элемент, который следует за последним удаленным элементом, или **`multiset::end()`**, если был удален такой последний элемент.

Срок действия итератора

Итераторы, указатели и ссылки, относящиеся к элементам, удаленным функцией, недействительны. Все остальные итераторы, указатели и ссылки сохраняют свою действительность.

find() — получить итератор для элемента

```
const_iterator find(const value_type& val) const;
iterator       find(const value_type& val);
```

Ищет в контейнере элемент, эквивалентный **`val`**, и, если тот найден, возвращает на него итератор. В противном случае он возвращает итератор **`multiset::end()`**.

Важно: — эта функция возвращает итератор для одного элемента из, возможно, нескольких эквивалентных. Чтобы получить весь диапазон эквивалентных элементов, следует использовать **`multiset::equal_range`**.

Два элемента набора считаются эквивалентными, если объект сравнения контейнера ре

Пример удаления из мультимножества

```
#include <iostream>
#include <set>

int main () {

  std::multiset<int> mymultiset;
  std::multiset<int>::iterator it;

  // Вставим чего-нибудь:
  mymultiset.insert(40);                                  // 40
  for (int i = 1; i < 7; i++) mymultiset.insert(i*10); // 10 20 30 40 40 50 60
  it = mymultiset.begin();                                //  ^
  it++;                                                   //     ^
  mymultiset.erase(it);                                   // 10 30 40 40 50 60
  mymultiset.erase(40);                                   // 10 30 50 60
  it = mymultiset.find(50);                               //        ^
  mymultiset.erase(it, mymultiset.end());                 // 10 30

  std::cout << "mymultiset содержит:";
  for (it = mymultiset.begin(); it != mymultiset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

equal_range() — получить диапазон одинаковых элементов

```
pair<const_iterator, const_iterator> equal_range(const value_type& val) const;
pair<iterator, iterator>             equal_range(const value_type& val);
```

Возвращает границы диапазона, который включает все элементы в контейнере, эквивалентные **val**.

Если совпадений не найдено, длина возвращаемого диапазона будет равна нулю, причем оба итератора будут указывать на первый элемент, который считается идущим после **val** в соответствии с внутренним объектом сравнения контейнера (**key_comp**).

Два элемента мультимножества считаются эквивалентными, если объект сравнения **comp(a, b)** для контейнера рефлексивно возвращает **false**, т.е. независимо от порядка, в котором элементы **a** и **b** передаются в качестве аргументов).

Возвращаемое значение

Функция возвращает **pair**, член которой **pair::first** является нижней границей диапазона (так же, как **lower_bound( )**), а **pair::second** — верхней границей (так же, как **upper_bound( )**).

Пример удаления нескольких эквивалентных элементов, используя `multiset::equal_range()`

```
#include <iostream>
#include <set>

typedef std::multiset<int>::iterator It;  // aliasing the iterator type used

int main () {

  int myints[] = {77, 30, 16, 2, 30, 30};
  std::multiset<int> mymultiset(myints, myints + 6);  // 2 16 30 30 30 77
  std::pair<It, It> ret = mymultiset.equal_range(30); //      ^        ^

  mymultiset.erase(ret.first, ret.second);

  std::cout << "mymultiset содержит:";
  for (It it = mymultiset.begin(); it != mymultiset.end(); ++it) // auto it
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Вывод

```
multiset содержит: 2 16 77
```

emplace() — создать и вставить элемент

Вставляет новый элемент в мультиключевой набор.

```
template <class... Args>
iterator emplace(Args&&... args); // аргументы для конструктора элемента
```

Новый элемент создается на месте с использованием аргументов **args** в качестве аргументов его конструктора. Размер контейнера пр этом увеличивается на единицу.

Внутренне контейнеры **multiset** сохраняют все свои элементы отсортированными в соответствии с критерием, указанным в его объекте сравнения. Элемент всегда вставляется в соответствующую позицию именно в этом порядке.

Элемент создается на месте путем вызова **allocator_traits::construct** с переадресацией аргументов **args** конструктору элемента.

Аналогичная функция-член **insert()** либо копирует, либо перемещает в контейнер уже существующие объекты.

Замечания:

C++98: Нет никаких гарантий в отношении относительного порядка эквивалентных элементов.

C++11: Относительный порядок эквивалентных элементов сохраняется, и вновь вставленные элементы следуют за своими эквивалентами, уже находящимися в контейнере.

Возвращаемое значение

Функция возвращает итератор на вставленный элемент.

Тип-члена **iterator** — это двунаправленный итератор, указывающий на элемент.

Пример

```
#include <iostream>
#include <set>
#include <string>

int main () {

  std::multiset<std::string> mymultiset;

  mymultiset.emplace("foo");
  mymultiset.emplace("bar");
  mymultiset.emplace("foo");

  std::cout << "mymultiset содержит:";
  for (const std::string& x: mymultiset)
    std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Вывод

```
mymultiset содержит: bar foo foo
```

emplace_hint() — создать и вставить элемент с подсказкой

Вставляет новый элемент в мультиключевой набор.

```
template <class... Args>
iterator emplace_hint(const_iterator position, Args&&... args);
```

Вставляет новый элемент в мультимножество с подсказкой позиции вставки. Этот новый элемент создается на месте с использованием аргументов **args** в качестве аргументов его конструктора.

Значение **position** используется как подсказка для точки вставки. Тем не менее, элемент будет вставлен в соответствующую позицию в соответствии с порядком, который устанавливается его внутренним объектом сравнения, но эта подсказка используется функцией для начала поиска точки вставки, что значительно ускоряет процесс в том случае, когда фактическая точка вставки находится в указанной позиции или близко к ней.

Пример `multiset::emplace_hint()`

```
#include <iostream>
#include <set>
#include <string>

int main () {

  std::multiset<std::string> mymultiset;
  auto it = mymultiset.cbegin();

  mymultiset.emplace_hint(it,"яблоко");
  it = mymultiset.emplace_hint(mymultiset.cend(), "апельсин");
  it = mymultiset.emplace_hint(it, "дыня");
  mymultiset.emplace_hint(it, "melon");

  std::cout << "mymultiset содержит:";
  for (const std::string& x: mymultiset)
    std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Вывод

```
mymultiset содержит: яблоко дыня дыня  апельсин
```

Неупорядоченный набор (unordered set)

Неупорядоченные наборы — это контейнеры, в которых уникальные элементы хранятся в произвольном порядке и которые позволяют быстро извлекать отдельные элементы на основе их значения.

Библиотека содержит шаблоны двух классов:

unordered_set — неупорядоченный набор (неупорядоченное множество);

unordered_multiset — неупорядоченный набор с несколькими ключами (неупорядоченное мультимножество).

а также функции:

begin — возвращает итератор на первый элемент в последовательности

end — возвращает итератор на последний элемент в последовательности

В **unordered_set** значение элемента одновременно является его ключом, который однозначно его идентифицирует. Ключи неизменяемы, поэтому элементы **unordered_set** не могут быть изменены в контейнере одномоментно, однако их можно удалять и вставлять.

Внутри элементы **unordered_set** не сортируются в каком-либо конкретном порядке, а организованы в корзины в зависимости от их хэш-значений, чтобы обеспечить быстрый доступ к отдельным элементам напрямую по их значениям (с постоянной средней временной сложностью в среднем).

Контейнеры **unordered_set** быстрее, чем контейнеры **set**, для доступа к отдельным элементам по их ключу, хотя, как правило, они менее эффективны для итерирования по диапазону.

Итераторы в контейнере — это как минимум однонаправленные итераторы (forward iterator).

```
template < class Key,                        // unordered_set::key_type/value_type
           class Hash = hash<Key>,           // unordered_set::hasher
           class Pred = equal_to<Key>,       // unordered_set::key_equal
           class Alloc = allocator<Key>      // unordered_set::allocator_type
           > class unordered_set;
```

Парметры шаблона

Key — тип элементов.

Каждый элемент в **unordered_set** уникально идентифицируется этим значением.

Псевдонимы типа этого параметра **unordered_set::key_type** и **unordered_set::value_type**.

Hash — хеш-функция

Унарный функциональный объект, который принимает в качестве аргумента объект того же типа, что и элементы, и возвращает на его основе уникальное значение типа **size_t**.

Это может быть либо класс, реализующий оператор вызова функции, либо указатель на функцию. По умолчанию используется **hash<Key>**, который возвращает хеш-значение с вероятностью коллизии, приближающейся к **1.0/std::numeric_limits<size_t>::max()**.

Объект **unordered_set** использует хеш-значения, возвращаемые этой функцией, для внутренней организации своих элементов, ускоряя процесс поиска отдельных элементов.

Псевдоним типа этого параметра **unordered_set::hasher**.

Pred — бинарный предикат.

Принимает два аргумента того же типа, что и сами элементы, и возвращает логическое значение. Выражение **pred(a, b)**, где **pred** — объект этого типа, а **a** и **b** — значения ключей, должно возвращать истину, если **a** должно считаться эквивалентным **b**.

Это может быть либо класс, реализующий оператор вызова функции, либо указатель на функцию.

По умолчанию используется **equal_to<Key>**, который возвращает то же самое, что и применение оператора равенства (**a == b**).

Объект **unordered_set** использует это выражение, чтобы определить, эквивалентны ли два ключа элемента. Никакие два элемента в контейнере **unordered_set** не могут иметь ключи, для которых этот предикат возвратит значение **true**.

Псевдоним типа этого параметра **unordered_set::key_equal**.

Alloc — тип объекта аллокатора.

Тип объекта аллокатора, который используется для распределения памяти. По умолчанию используется шаблон класса **allocator**, который определяет простейшую модель распределения памяти. Псевдоним типа этого параметра **unordered_set::allocator_type**.

При ссылках на функции-члены шаблона **unordered_set** эти же имена (**Key**, **Hash**, **Pred** и **Alloc**) используются в качестве параметров шаблона.

Общедоступные функции-члены

(**constructor**)	создает набор
(**destructor**)	разрушает набор
operator=	присваивает содержимое контейнеру
empty()	проверка на пустоту
size()	возвращает количество элементов в контейнере
max_size()	возвращает максимально допустимое количество элементов в контейнере
begin()/end()	возвращает итератор на начало/конец набора
cbegin()/cend()	константные версии
insert()/erase()	вставляет/удаляет элемент
swap()	обменивает содержимым два контейнера
clear()	очищает содержимое контейнера
emplace()	создает и вставляет элемент
emplace_hint()	создает и вставляет элемент с подсказкой

Поиск элементов:

`find()`	получить итератор на элемент
`count()`	количество элементов с указанным значением (1 или 0)
`equal_range()`	получить диапазон одинаковых элементов
`get_allocator()`	получить аллокатор

Корзины[1]

`bucket_count()`	возвращает количество корзин
`max_bucket_count()`	возвращает максимально возможное количество корзин
`bucket_size()`	возвращает размер корзины
`bucket()`	найти корзину для элемента

Политики хеширования

`load_factor()`	возвращает коэффициент нагрузки
`max_load_factor()`	возвращает максимально возможный коэффициент нагрузки
`rehash()`	установить количество корзин
`reserve()`	запросить изменение емкости

1 Корзины, гнезда, сегменты, слоты. При вычислении, например, гистограммы мы выполняем «биннинг» данных или группируем несколько более или менее непрерывных (смежных) значений в меньшее количество «бинов (ящиков)».
При сортировке по «корзинам» мы используем «корзины» и назначаем «корзину» каждому значению некоторой коллекции.

Конструкторы std::unordered_set

(1) Пустой

```
explicit unordered_set(size_type n = ,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& alloc = allocator_type());

explicit unordered_set(const allocator_type& alloc);
```

Создает пустой объект **unordered_set**, не содержащий элементов и имеющий нулевой размер. Этот конструктор может создавать контейнер с конкретными объектами хеширования, **key_equal** и **allocator**, а также с минимальным количеством хэш-корзин .

n — минимальное количество начальных корзин.

Это не количество элементов в контейнере, а минимальное количество слотов, требуемых для внутренней хеш-таблицы при построении контейнера.

Если этот аргумент не указан, конструктор определяет **n** автоматически способом, который зависит от конкретной реализации библиотеки.

size_type — это целочисленный тип без знака.

hf — функциональный объект, выполняющий хэширование.

hasher — это функция, которая возвращает целое значение на основе ключа объекта контейнера, переданного ему в качестве аргумента.

Тип члена **hasher** определяется в **unordered_set** как псевдоним второго параметра его шаблона (**Hash**).

eql — функциональный объект, выполняющий сравнение Он возвращает истину, если два ключа объекта-контейнера, переданные ему в качестве аргументов, должны считаться равными.

Тип члена **key_equal** определяется в **unordered_set** как псевдоним третьего параметра его шаблона (**Pred**).

Alloc — объект-аллокатор, который будет использоваться вместо того, чтобы создать новый.

Для экземпляров класса, использующих свою версию шаблона класса аллокатора по умолчанию, этот параметр не имеет значения.

(2) Диапазонный

```
template <class InputIterator>
unordered_set(InputIterator first, InputIterator last,
              size_type n = /* see below */,
              const hasher& hf = hasher(),
              const key_equal& eql = key_equal(),
              const allocator_type& alloc = allocator_type());
```

(3) Копирования

```
unordered_set(const unordered_set& uset);
unordered_set(const unordered_set& uset, const allocator_type& alloc);
```

(4) Перемещения

```
unordered_set(unordered_set&& ust);
unordered_set(unordered_set&& ust, const allocator_type& alloc);
```

(5) Из списка инициализации

```
unordered_set(initializer_list<value_type> il,
              size_type n = ,
              const hasher& hf = hasher(),
              const key_equal& eql = key_equal(),
              const allocator_type& alloc = allocator_type());
```

il — объект типа **initializer_list**.

Создается автоматически из деклараторов списка инициализаторов.

Тип элемента **value_type** — это тип элементов в контейнере, определенный в **unordered_set** как псевдоним первого параметра шаблона (**Key**).

Пример создания контейнера unordered_set

```
#include <iostream>
#include <string>
#include <unordered_set>

template<class T> T cmerge(T a, T b) {  // слияние двух контейнеров
    T t(a);
    t.insert(b.begin(), b.end());
    return t;
}

int main () {

  std::unordered_set<std::string> first;                                  // пустой
  std::unordered_set<std::string> second({"red","green","blue"});     // ini lst
  std::unordered_set<std::string> third({"orange","pink","yellow"}); // ini lst
  std::unordered_set<std::string> fourth(second);                         // copy
  std::unordered_set<std::string> fifth(cmerge(third,fourth));        // move
  std::unordered_set<std::string> sixth(fifth.begin(), fifth.end()); // range

  std::cout << "sixth содержит: ";
  for (const std::string& x: sixth) std::cout << " " << x;
  std::cout << std::endl;
}
```

Вывод

```
sixth содержит: pink yellow red green orange blue
```

unordered_set::operator= ()

Присваивание копированием

```
unordered_set& operator= (const unordered_set& uset);
```

Присваивание перемещением

```
unordered_set& operator= (unordered_set&& uset);
```

Присваивание списка инициализации

```
unordered_set& operator= (intitializer_list<value_type> il);
```

Все итераторы, ссылки и указатели на элементы, которые были в контейнере до вызова, становятся недействительными.

begin() — возвращает итератор на начало

(1) Итератор на первый элемент в контейнере

```
      iterator begin( ) noexcept;
const_iterator begin( ) const noexcept;
```

(2) Итератор на первый элемент в корзине[2]

```
      local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
```

Возвращает итератор, указывающий на первый элемент в контейнере **unordered_set** (1) или в одной из его корзин (2).

Объект **unordered_set** не дает никаких гарантий, какой конкретный элемент считается его первым элементом. Но в любом случае диапазон от начала до конца охватывает все элементы в контейнере (или корзине) до тех пор, пока он не будет признан недействительным.

Все итераторы в **unordered_set** имеют константный доступ к элементам, даже тем, чей тип не имеет префикса **const_** — элементы могут быть вставлены или удалены, но не могут быть изменены в контейнере.

2 bucket — сегмент, корзина, ведро, гнездо, слот ...

end() — возвращает итератор на конец

(1) Итератор на элемент в контейнере, следующий за последним

```
      iterator end( ) noexcept;
const_iterator end( ) const noexcept;
```

(2) Итератор на элемент в корзине, следующий за последним

```
      local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
```

Возвращает итератор, указывающий на последний элемент в контейнере **unordered_set** (1) или в одной из его корзин (2).

Итератор, возвращаемый **end( )**, указывает не на какой-либо элемент, а на позицию, которая следует за последним элементом в контейнере **unordered_set** (позиция за концом). Таким образом, возвращаемое значение не должно разыменовываться — оно обычно используется для описания открытого конца диапазона, такого как **[begin, end)**.

unordered_set не дает никаких гарантий относительно порядка его элементов. Но в любом случае диапазон от **begin( )** до **end( )** охватывает все элементы в контейнере (или корзине) до тех пор, пока он не будет признан недействительным.

Все итераторы в **unordered_set** имеют константный доступ к элементам, даже тем, чей тип не имеет префикса **const_** — элементы могут быть вставлены или удалены, но не могут быть изменены в контейнере.

find() — получить итератор на элемент

```
      iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
```

Ищет в контейнере элемент со значением **k** в качестве значения и, если он найден, возвращает на него итератор, в противном случае он возвращает итератор на **unordered_set::end()**.

Чтобы просто проверить, существует ли конкретный элемент, может использоваться функция-член **unordered_set::count()**.

Все итераторы в **unordered_set** имеют константный доступ к элементам, даже тем, чей тип не имеет префикса **const_**.

Элементы могут быть вставлены или удалены, но не могут быть изменены.

k — ключ, который нужно искать. Тип элемента **key_type** — это тип элементов в контейнере.

В контейнерах **unordered_set** это то же самое, что и **value_type**, определенное как псевдоним первого параметра шаблона класса (**Key**).

Возвращаемое значение

Итератор элемента, если указанное значение найдено в контейнере, если же оно не найдено, то **unordered_set::end()**.

Типы членов **iterator** и **const_iterator** являются однонаправленными.

Пример unordered_set::find

```
#include <iostream>
#include <string>
#include <unordered_set>

int main () {

  std::unordered_set<std::string> myset = {"red", "green", "blue"};

  std::string input;
  std::cout << "color? ";
  getline(std::cin, input);

  std::unordered_set<std::string>::const_iterator got = myset.find(input);
  if (got == myset.end())
      std::cout << "not found in myset";
  else
      std::cout << *got << " is in myset";

  std::cout << std::endl;

  return 0;
}
```

Вывод

color? blue

blue is in myset

count() — количество элементов с определенным значением ключа

```
size_type count(const key_type& k) const;
```

Ищет в контейнере элементы со значением **k** и возвращает количество найденных элементов. Поскольку контейнеры типа **unordered_set** не допускают повторяющихся значений, это означает, что функция фактически возвращает 1, если элемент с таким значением существует в контейнере, и ноль в противном случае.

k —

Value of the elements to be counted.

Тип члена **key_type** — это тип элементов в контейнере. В контейнерах **unordered_set** это то же самое, что и **value_type**, определенное как псевдоним первого параметра шаблона класса (**Key**).

Пример `unordered_set::count`

```
#include <iostream>
#include <string>
#include <unordered_set>

int main () {

  std::unordered_set<std::string> myset = { "hat", "umbrella", "suit" };

  for (auto& x: {"hat","sunglasses","suit","t-shirt"}) {
    if (myset.count(x) > 0)
      std::cout << "myset has " << x << std::endl;
    else
      std::cout << "myset has no " << x << std::endl;
  }

  return 0;
}
```

Вывод

```
myset has hat
myset has no sunglasses
myset has suit
myset has no t-shirt
```

equal_range() — получить диапазон элементов с определенным значением ключа

```
pair<iterator, iterator> equal_range(const key_type& k);

pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
```

Возвращает границы диапазона, который включает в себя все элементы, равные **k**.

В контейнерах **unordered_set**, где ключи уникальны, диапазон будет включать не более одного элемента.

Если **k** не соответствует ни одному элементу в контейнере, возвращаемый диапазон имеет в качестве значения **end( )** как для нижнtq, так и для верхней границы диапазона.

Все итераторы в **unordered_set** имеют константный доступ к элементам. Т.е. элементы в контейнере могут быть вставлены или удалены, но не могут быть изменены.

Тип члена **key_type** — это тип элементов в контейнере. В контейнерах **unordered_set** это то же самое, что и **value_type**, определенное как псевдоним первого параметра шаблона класса (**Key**).

Возвращаемое значение

Функция возвращает пару, где ее член **pair::first** является итератором для нижней границы диапазона, а **pair::second** — итератором для его верхней границы. Элементы в диапазоне — это элементы между этими двумя итераторами, включая **pair::first**, но не **pair::second**.

Типы **iterator** и **const_iterator** являются типами однонаправленного итератора.

insert() — вставить элементы

Функция вставляет новые элементы в **unordered_set**.

Каждый из элементов вставляется только в том случае, если он не эквивалентен какому-либо другому элементу, уже находящемуся в контейнере (элементы в **unordered_set** имеют уникальные значения). Размер контейнера при этом увеличивается на количество вставленных элементов.

Параметры определяют, сколько элементов вставлено и какими значениями они инициализируются:

(1) Один элемент копированием

```
pair<iterator, bool> insert(const value_type& val );
```

(2) Один элемент перемещением

```
pair<iterator, bool> insert(value_type&& val );
```

(3) Один элемент копированием с подсказкой

```
iterator insert(const_iterator hint, const value_type& val );
```

(4) Один элемент перемещением с подсказкой

```
iterator insert(const_iterator hint, value_type&& val );
```

val — объект, который будет скопирован (или перемещен как) значение нового элемента.

Версии (1) и (3) копируют элемент (т.е. **val** сохраняет его содержимое, а контейнер сохраняет копию).

Версии (2) и (4) перемещают элемент (т.е. **val** теряет свое содержимое, которое получает новый элемент в контейнере).

Тип элемента **value_type** — это тип элементов в контейнере, определенный в **unordered_set** как псевдоним первого параметра шаблона (**Key**).

hint — итератор позиции, предложенной как подсказка о том, откуда начать поиск подходящей точки вставки. Это значение может как использоваться контейнером для оптимизации его работы, так и не использоваться. Элемент будет сохранен в соответствующей корзине независимо от того, что передается в качестве подсказки.

Тип элемента **const_iterator** - это тип однонаправленного итератора.

(5) Вставка диапазона

```
template <class InputIterator>
void insert(InputIterator first, InputIterator last );
```

first, last — итераторы, определяющие диапазон элементов.

В контейнер **unordered_set** вставляются копии элементов из диапазона **[first, last)**

Диапазон включает все элементы между первым и последним, включая элемент, на который указывает **first**, но не тот, на который указывает **last**. Ни **first**, ни **last** не должны быть итераторами в целевом контейнере. Тип шаблона может быть любым типом итератора ввода.

(6) **Вставка диапазона из списка инициализации**

```
void insert( initializer_list<value_type> il );
```

il — объект типа **initializer_list**.

Из деклараторов списка инициализаторов компилятор будет строить такие объекты автоматически. Тип элемента **value_type** — это тип элементов, содержащихся в контейнере, определенный в **unordered_set** как псевдоним первого параметра шаблона (**Key**).

Возвращаемое значение

В версиях (1) и (2) функция возвращает объект пары.

Первым элементом пары является итератор, указывающий либо на вновь вставленный элемент в контейнер, либо на элемент с эквивалентным ключом.

Вторым элементом является значение типа **bool**, указывающее, был ли элемент успешно вставлен или нет.

В версиях (3) и (4) функция возвращает итератор, указывающий либо на вновь вставленный элемент в контейнер, либо на элемент с эквивалентным ключом.

Версии (5) и (6) не возвращают никакого значения.

Память для нового элемента выделяется с помощью функции **allocator_traits<allocator_type>::construct ()**, которая может генерировать исключения в случае сбоя (для аллокатора по умолчанию, если запрос на выделение не выполняется, генерируется **bad_alloc**).

Срок действия итератора

В большинстве случаев все итераторы в контейнере остаются действительными после вставки. Единственное исключение — когда рост контейнера вызывает повторное хеширование (rehash). В этом случае все итераторы в контейнере становятся недействительными.

Повторное хеширование выполняется принудительно, если новый размер контейнера после операции вставки превысит его пороговое значение (рассчитанное как **`bucket_count`** контейнера, умноженное на его **`max_load_factor`**).

Ссылки на элементы в контейнере **`unordered_set`** остаются действительными во всех случаях, даже после повторного хеширования.

Пример unordered_set::insert()

```
#include <iostream>
#include <string>
#include <array>
#include <unordered_set>

int main () {

  std::unordered_set<std::string> myset = {"yellow","green","blue"};
  std::array<std::string, 2> myarray = {"black","white"};
  std::string mystring = "red";

  myset.insert(mystring);                        // вставка копирвоанием
  myset.insert(mystring+"dish");                 // вставка перемещением
  myset.insert(myarray.begin(), myarray.end()); // вставка даипазона
  myset.insert({"purple","orange"});             // вставка списка инициализации

  std::cout << "myset содержит:";
  for (const std::string& x: myset) std::cout << " " << x;
  std::cout <<  std::endl;

  return 0;
}
```

Вывод

```
myset содержит: green blue reddish white yellow black red orange purple
```

erase() — удалить элемент

(1) удалить в позиции

```
iterator erase(const_iterator position);
```

(2) удалить по ключу

```
size_type erase(const key_type& k);
```

(3) удалить диапазон

```
iterator erase(const_iterator first, const_iterator last);
```

Удаляет из контейнера **unordered_set** либо один элемент, либо диапазон элементов **[first, last)**. Для каждого элемента вызывается деструктор, что уменьшает размер контейнера на количество удаленных элементов.

position — итератор, указывающий на единственный элемент, который нужно удалить из **unordered_set**.

k — значение удаляемого элемента.

Тип элемента **key_type** — это тип элементов в контейнере. В контейнерах **unordered_set** это то же самое, что и **value_type**, определенное как псевдоним первого параметра шаблона класса (**Key**).

first, last — итераторы, определяющие диапазон в удаляемом контейнере **unordered_set**.

Контейнеры **unordered_set** не следуют какому-либо определенному порядку для организации своих элементов, поэтому эффект удаления диапазона предсказать нелегко.

Тип члена **const_iterator** — тип однонаправленного итератора.

Возвращаемое значение

Версии (1) и (3) возвращают итератор, указывающий на позицию, следующую сразу за последним из удаленных элементов.

Версия (2) возвращает количество удаленных элементов, которое в контейнерах **unordered_set** (имеющих уникальные значения) равно 1, если элемент со значением **k** существовал (и, таким образом, был впоследствии удален), и ноль в противном случае.

Тип члена **iterator** — тип однонаправленного итератора.

Срок действия итератора

C++11 — недействительны только итераторы и ссылки на удаленные элементы.

Остальные не затронуты.

C++14 — недействительны только итераторы и ссылки на удаленные элементы.

Остальные не затронуты. Относительный порядок итерации элементов, не удаленных операцией, сохраняется.

Пример unordered_set::erase

```
#include <iostream>
#include <string>
#include <unordered_set>

int main () {

  std::unordered_set<std::string> myset =
  {"USA", "Canada", "France", "UK", "Japan", "Germany", "Italy"};

  myset.erase(myset.begin());                       // удаление из позиции
  myset.erase("France");                            // удаление по ключу
  myset.erase(myset.find("Japan"), myset.end() ); // удаление диапазона

  std::cout << "myset содержит:";
  for ( const std::string& x: myset ) std::cout << " " << x;
  std::cout << std::endl;

  return 0;
}
```

Вывод

```
myset содержит: Canada USA Italy
```

emplace() – создать и вставить элемент

```
template <class... Args>
pair <iterator,bool> emplace ( Args&&... args );
```

Вставляет новый элемент в **unordered_set**, если его значение уникально.

Этот новый элемент создается на месте с использованием **args** в качестве аргументов конструктора этого элемента.

Вставка имеет место только в том случае, если ни один элемент в контейнере не имеет значения, эквивалентного тому, который был установлен (элементы в **unordered_set** имеют уникальные значения).

Если он вставлен, размер контейнера увеличивается на единицу.

Существует аналогичная функция-член **insert()**, которая либо копирует, либо перемещает существующие объекты в контейнер.

Возвращаемое значение

Если функция успешно вставляет элемент (в том случае, если не существует другого элемента с таким же значением), функция возвращает пару с итератором на вновь вставленный элемента и значение **true**. В противном случае он возвращает итератор на существующий эквивалентный элемент в контейнере, и значение **false**.

Срок действия итератора

В большинстве случаев все итераторы в контейнере остаются действительными после вставки. Единственное исключение — когда рост контейнера вызывает повторное хеширование (rehash). В этом случае все итераторы в контейнере становятся недействительными.

Повторное хеширование выполняется принудительно, если новый размер контейнера после операции вставки превысит его пороговое значение (рассчитанное как **`bucket_count`** контейнера, умноженное на его **`max_load_factor`**).

Ссылки на элементы в контейнере **`unordered_set`** остаются действительными во всех случаях, даже после повторного хеширования.

Пример unordered_set::emplace

```
#include <iostream>
#include <string>
#include <unordered_set>

int main () {

  std::unordered_set<std::string> myset;

  myset.emplace("potatoes");
  myset.emplace("milk");
  myset.emplace("flour");

  std::cout << "myset содержит:";
  for (const std::string& x: myset) std::cout << " " << x;

  std::cout << std::endl;
  return 0;
}
```

Вывод

```
myset содержит: potatoes flour milk
```

emplace_hint() — создать и вставить элемент с подсказкой

```
template <class... Args>
iterator emplace_hint(const_iterator position, Args&&... args );
```

Вставляет новый элемент в **unordered_set**, если его значение уникально.

Этот новый элемент создается на месте с использованием **args** в качестве аргументов конструктора данного элемента.

position указывает на место в контейнере, предложенное в качестве подсказки о том, где начать поиск точки вставки (контейнер может использовать или не использовать это предложение для оптимизации операции вставки).

Вставка имеет место только в том случае, если ни один элемент в контейнере не имеет значения, эквивалентного тому, который был установлен (элементы в **unordered_set** имеют уникальные значения). Если элемент вставлен, размер контейнера увеличивается на единицу.

Существует аналогичная функция-член **insert**, которая либо копирует, либо перемещает существующий объект в контейнер, а также может принимать позиционную подсказку.

Возвращаемое значение

Если функция успешно вставляет элемент (в том случае, когда не существует другого элемента с таким же значением), она возвращает итератор на вставленный элемент.

В противном случае возвращается итератор, указывающий на эквивалентный элемент в контейнере (он не заменяется).

Память для нового элемента выделяется с помощью **allocator_traits<allocator_type>::construct()**, который может генерировать исключения в случае сбоя (для аллокатора по умолчанию, если запрос на выделение не выполняется, выбрасывается **bad_alloc**).

В большинстве случаев все итераторы в контейнере остаются действительными после вставки. Единственное исключение — когда рост контейнера вызывает повторное хеширование. В этом случае все итераторы в контейнере становятся недействительными.

Повторное хеширование выполняется принудительно, если новый размер контейнера после операции вставки превысит его пороговое значение (рассчитанное как **bucket_count** контейнера, умноженное на его **max_load_factor**).

Ссылки остаются действительными во всех случаях, даже после повторного хеширования.

bucket_count() — возвратить количество корзин

```
size_type bucket_count() const noexcept;
```

Возвращается количество корзин (гнезд) в контейнере **unordered_set container**.

Корзина — это слот во внутренней хэш-таблице контейнера, которому присваиваются элементы на основе их хеш-значения.

Количество корзин напрямую влияет на коэффициент загрузки хеш-таблицы контейнера и, следовательно, на вероятность коллизии. Контейнер автоматически увеличивает количество корзин, чтобы поддерживать коэффициент загрузки ниже определенного порога (его **max_load_factor**), вызывая повторное хеширование каждый раз, когда количество корзин необходимо увеличить.

Пример unordered_set::bucket_count

```
#include <iostream>
#include <string>
#include <unordered_set>

int main () {

  std::unordered_set<std::string> myset = {
    "Mercury", "Venus", "Earth", "Mars",
    "Jupiter", "Saturn", "Uranus", "Neptune"
  };

  unsigned n = myset.bucket_count();

  std::cout << "myset имеет " << n << " корзин.\n";

  for (unsigned i = 0; i < n; ++i) {
    std::cout << "корзина #" << i << " содержит:";
    for (auto it = myset.begin(i); it != myset.end(i); ++it)
      std::cout << " " << *it;
    std::cout << "\n";
  }

  return 0;
}
```

Вывод

```
myset имеет 11 корзин.
корзина #0 содержит:
корзина #1 содержит: Venus
корзина #2 содержит: Jupiter
корзина #3 содержит:
корзина #4 содержит: Neptune Mercury
корзина #5 содержит:
корзина #6 содержит: Earth
корзина #7 содержит: Uranus Saturn
корзина #8 содержит: Mars
корзина #9 содержит:
корзина #10 содержит:
```

max_bucket_count() — вернуть максимальное количество корзин

```
size_type max_bucket_count() const noexcept;
```

Возвращает максимальное количество корзин, которое может иметь контейнер **unordered_set**.

Это максимальное потенциальное количество корзин, которое может иметь контейнер из-за системных ограничений или ограничений реализации его библиотеки.

bucket_size() — размер корзины хеша

```
size_type bucket_size(size_type n) const;
```

Возвращает количество элементов в корзине с номером **n**.

Корзина — это слот во внутренней хэш-таблице контейнера, которому присваиваются элементы на основе их хеш-значения.

Количество элементов в корзине влияет на время, необходимое для доступа к определенному элементу в корзине. Контейнер автоматически увеличивает количество сегментов, чтобы коэффициент загрузки (который является средним размером контейнера) был ниже его **max_load_factor**.

n — номер корзины.

номер корзины должен быть меньше **bucket_count**.

Тип элемента **size_type** — это целочисленный тип без знака.

Возвращаемое значение

Количество элементов в корзине с номером **n**.

Пример unordered_set::bucket_size

```
#include <iostream>
#include <string>
#include <unordered_set>

int main () {

  std::unordered_set<std::string> myset = {
      "red", "green", "blue", "yellow", "purple", "pink"
  };

  unsigned nbuckets = myset.bucket_count();

  std::cout << "myset содержит " << nbuckets << " корзин:\n";

  for (unsigned i = 0; i < nbuckets; ++i) {
    std::cout << "корзина #" << i
              << " содержит " << myset.bucket_size(i) << " элементов.\n";
  }

  return 0;
}
```

Вывод

```
myset содержит 7 корзин:
корзина #0 содержит 1 элементов.
корзина #1 содержит 1 элементов.
корзина #2 содержит 2 элементов.
корзина #3 содержит 0 элементов.
корзина #4 содержит 1 элементов.
корзина #5 содержит 1 элементов.
корзина #6 содержит 0 элементов.
```

bucket() — найти корзину элемента

```
size_type bucket ( const key_type& k ) const;
```

Возвращает номер корзины, в котором находится элемент со значением **k**.

Корзина — это слот во внутренней хэш-таблице контейнера, которому присваиваются элементы на основе их хеш-значения. Корзины нумеруются от 0 до (**bucket_count - 1**).

Доступ к отдельным элементам в корзине можно получить с помощью итераторов диапазона, возвращаемых **unordered_set::begin()** и **unordered_set::end()**.

Тип элемента **key_type** — это тип элементов в контейнере. В контейнерах **unordered_set** это то же самое, что и **value_type**, определенное как псевдоним первого параметра шаблона класса (**Key**).

load_factor() — возвратить коффициент загрузки

```
float load_factor( ) const noexcept;
```

Возвращает текущий коэффициент загрузки в контейнере **unordered_set**.

Коэффициент загрузки — это соотношение между количеством элементов в контейнере (его размером) и количеством ведер (**bucket_count**):

```
load_factor = size / bucket_count
```

Коэффициент загрузки влияет на вероятность коллизий в хеш-таблице (то есть вероятность того, что два элемента находятся в одной корзине). Контейнер автоматически увеличивает количество сегментов, чтобы поддерживать коэффициент загрузки ниже определенного порога (его **max_load_factor**), вызывая повторное хеширование каждый раз, когда требуется расширение контейнера.

Чтобы получить или изменить этот порог, следует использовать функцию-член **max_load_factor( )**.

max_load_factor() — получить/установить максимальный коэффициент загрузки

(1) Получить

```
float max_load_factor( ) const noexcept;
```

(2) Установить

```
void max_load_factor(float z);
```

Получить или установить максимальный коэффициент загрузки.

Первая версия (1) возвращает текущий максимальный коэффициент загрузки для контейнера **unordered_set**.

Вторая версия (2) устанавливает **z** как новый максимальный коэффициент загрузки для контейнера **unordered_set**.

Коэффициент загрузки — это соотношение между количеством элементов в контейнере (его размером) и количеством ведер (**bucket_count**).

По умолчанию контейнеры **unordered_set** имеют **max_load_factor**, равный 1.0.

Коэффициент загрузки влияет на вероятность коллизий в хеш-таблице (то есть вероятность того, что два элемента находятся в одной корзине). Контейнер использует значение **max_load_factor** как порог, который вызывает увеличение количества сегментов (и, таким образом, вызывает повторное хеширование).

Срок действия итераторов

Никаких изменений, если только изменение этого значения не приводит к повторному хешированию. В этом случае все итераторы в контейнере становятся недействительными.

Если новый **`max_load_factor`** установлен ниже текущего **`load_factor`**, принудительно выполняется повторное хеширование.

Ссылки на элементы в контейнере **`unordered_set`** остаются действительными во всех случаях, даже после повторного хеширования.

Пример unordered_set::max_load_factor

```
#include <iostream>
#include <string>
#include <unordered_set>

int main () {

  std::unordered_set<std::string> myset = {
      "New York", "Paris", "London", "Hong Kong", "Bangalore", "Berlin"
  };

  std::cout << "текущий max_load_factor: " << myset.max_load_factor()
            << std::endl;
  std::cout << "текущий размер: " << myset.size() << std::endl;
  std::cout << "текущий bucket_count: " << myset.bucket_count() << std::endl;
  std::cout << "текущий load_factor: "  << myset.load_factor()  << std::endl;

  float z = myset.max_load_factor();
  myset.max_load_factor(z/2.0);
  std::cout << "[max_load_factor уполовинен]" << std::endl;

  std::cout << "new max_load_factor: " << myset.max_load_factor() << std::endl;
  std::cout << "new size: " << myset.size() << std::endl;
  std::cout << "new bucket_count: " << myset.bucket_count() << std::endl;
  std::cout << "new load_factor: "  << myset.load_factor()  << std::endl;

}
```

Вывод

```
текущий max_load_factor: 1
текущий размер: 6
текущий bucket_count: 7
текущий load_factor: 0.857143
[max_load_factor уполовинен]
new max_load_factor: 0.5
new size: 6
new bucket_count: 13
new load_factor: 0.461538
```

rehash() — установить количество корзин

```
void rehash(size_type n);
```

Устанавливает количество корзин в контейнере равным **n** или более.

Если **n** больше, чем текущее количество корзин в контейнере (**bucket_count**), принудительно выполняется повторное хеширование.

Новое количество корзин может быть равно или больше **n**.

Если **n** меньше текущего количества сегментов в контейнере (**bucket_count**), функция может не изменить количество корзин и не вызвать повторное хеширование.

Пример unordered_set::rehash

```
#include <iostream>
#include <string>
#include <unordered_set>

int main () {

  std::unordered_set<std::string> myset;

  myset.rehash(12);

  myset.insert("office");
  myset.insert("house");
  myset.insert("gym");
  myset.insert("parking");
  myset.insert("highway");

  std::cout << "current bucket_count: " << myset.bucket_count() << std::endl;

  return 0;
}
```

Вывод

```
current bucket_count: 13
```

reserve() — запросить изменение емкости

```
void reserve(size_type n);
```

Устанавливает количество корзин в контейнере (**bucket_count**) в наиболее подходящее, чтобы содержать не менее **n** элементов, значение.

Если **n** больше, чем текущее **bucket_count**, умноженное на **max_load_factor**, **bucket_count** контейнера увеличивается и выполняется принудительное повторное хеширование.

Если **n** меньше этого, функция может не иметь никакого эффекта.

n — количество запрошенных элементов в качестве минимальной емкости.

Срок действия итераторов

Если происходит повторное хеширование, все итераторы становятся недействительными, но ссылки и указатели на отдельные элементы остаются действительными.

Если фактического повторного хеширования не происходит, никаких изменений не будет.

hash_function() — получить хеш-функцию

```
hasher hash_function() const;
```

Возвращает функциональный объект, выполняющий хэширование, который используется контейнером.

Хеш-функция — это унарная функция, которая принимает объект типа **key_type** в качестве аргумента и возвращает на его основе уникальное значение типа **size_t**.

Она устанавливается контейнером при его создании (см. конструктор). По умолчанию это хеш-функция по умолчанию для соответствующего типа ключа: **hash<key_type>**.

Возвращаемое значение

Хеш-функция

Тип члена **hasher** — это тип хэш-функции, используемой контейнером, определенный в **unordered_set** как псевдоним его второго параметра шаблона (**Hash**).

Пример unordered_set::hash_function

```
#include <iostream>
#include <string>
#include <unordered_set>

typedef std::unordered_set<std::string> stringset;

int main ( ) {

  stringset myset;

  stringset::hasher fn = myset.hash_function( );

  std::cout << "that: " << fn("that") << std::endl;
  std::cout << "than: " << fn("than") << std::endl;

  return 0;
}
```

Вывод

```
that: 1046150783
than: 929786026
```

Две похожие строки дают совершенно разные хеш-значения.

key_eq() — получить предикат эквивалентности ключей

```
key_equal key_eq( ) const;
```

Возвращает предикат сравнения эквивалентности ключей, используемый контейнером **unordered_set**.

Сравнение эквивалентности ключей — это предикат, который принимает значение двух элементов в качестве аргументов и возвращает логическое значение, указывающее, должны ли они считаться эквивалентными. Он устанавливается контейнером при создании (см. конструктор).

По умолчанию это **equal_to<key_type>**, которое возвращает то же самое, что и применение оператора равенства (**==**) к аргументам.

Возвращаемое значение

Функциональный объект сравнения эквивалентности ключей (equality comparison object).

Тип элемента **key_equal** — это тип предиката сравнения эквивалентности ключей, определенный в **unordered_set** как псевдоним его третьего параметра шаблона (**Pred**).

Пример unordered_set::key_eq

```
#include <iostream>
#include <string>
#include <unordered_set>

int main () {

  std::unordered_set<std::string> myset;

  bool case_insensitive = myset.key_eq()("checking","CHECKING");

  std::cout << "myset.key_eq() is ";
  std::cout << (case_insensitive ? "case insensitive" : "case sensitive" );
  std::cout << std::endl;

  return 0;
}
```

Вывод

```
myset.key_eq() is case sensitive
```