КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

Лекция № 19.4 — 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

Ассоциативные контейнеры

Ассоциативные контейнеры — это такие коллекции, в которых позиция элемента зависит от его значения, то есть после занесения элементов в коллекцию порядок их следования будет задаваться их значениями. К ассоциативным контейнерам относятся:

-набор (set);

- словарь (map).

Набор (set)

Библиотека содержит шаблоны двух классов:
set — набор (множество) и **multiset** — набор с несколькими ключами (мультимножество).
а также функции:
begin — возвращает итератор на первый элемент в последовательности
end — возвращает итератор на последний элемент в последовательности

Наборы — это контейнеры, в которых хранятся *уникальные* элементы в определенном порядке.

В наборе элемент идентифицируется его значением — значение само по себе является ключом типа **T**, и каждое значение должно быть уникальным.

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[1], указанным его внутренним объектом сравнения (типа **Compare**).

---

1 **Строгий слабый порядок** — это двоичный предикат, сравнивающий два объекта и возвращающий истину, если первый предшествует второму. Этот предикат должен удовлетворять стандартному математическому определению строгого слабого порядка. Точные требования можно найти в википедии (Отношение порядка), но примерно они означают, что строгий слабый порядок должен вести себя так, как ведет себя «меньше чем» — если a меньше b, то b не меньше a, если a меньше b и b меньше c, тогда a меньше c.
Двоичный предикат — это двоичная функция, результат которой представляет истинность или ложность некоторого условия. Двоичный предикат может, например, быть функцией, которая принимает два аргумента и проверяет, равны ли они.
Двоичная функция — это тип функционального объекта — объекта, который может быть вызван, как если бы это была обычная функция C++. Двоичная функция вызывается с двумя аргументами.

```
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.

Псевдонимы в качестве типа элементов **set::key_type** и **set::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 **set** использует это выражение, чтобы определить как порядок следования элементов в контейнере, так и эквивалентность двух ключей элементов (путем рефлексивного сравнения — они эквивалентны, если истинно **!Comp(a, b) && !Comp(b, a)**).

Никакие два элемента в контейнере set не могут быть эквивалентными.

Это выражение 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 **set::key_compare** и **set::value_compare**.

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

Типы членов

Тип члена	Определение	Примечания
`key_type`	Первый параметр шаблона (**`T`**)	
`value_type`	Первый параметр шаблона (**`T`**)	
`key_compare`	Второй параметр шаблона (**`Compare`**)	по умолчанию `less<key_type>`
`value_compare`	Второй параметр шаблона (**`Compare`**)	по умолчанию `less<key_type>`
`allocator_type`	Третий параметр шаблона (**`Alloc`**)	по умолчанию `allocator<value_type>`
`reference`	`value_type&`	
`const_reference`	`const value_type&`	
`pointer`	`allocator_traits<allocator_type>::pointer`	для аллокатора по умолчанию: `value_type*`
`const_pointer`	`allocator_traits<allocator_type>::const_pointer`	для аллокатора по умолчанию: `const_value_type*`
`iterator`	двунаправленный итератор на `const value_type`	(*) может быть преобразован в `const_iterator`
`const_iterator`	двунаправленный итератор на `const value_type`	(*)

`reverse_iterator`	`reverse_iterator<iterator>`	(*)
`const_reverse_iterator`	`reverse_iterator<const_iterator>`	(*)
`difference_type`	знаковый целочисленный тип, идентичный `iterator_traits<iterator>::difference_type`	обычно `ptrdiff_t`
`size_type`	беззнаковый целочисленный тип, который может представлять любое неотрицательное значение `difference_type`	обычно `size_t`

(*) Примечание. Все итераторы задают значение для константных элементов. Является ли тип члена **`const_`** тем же типом, что и его аналог, отличный от **`const_`**, зависит от конкретной реализации библиотеки, но программы не должны полагаться на то, что они отличаются.

При доступе к отдельным элементам по их ключу контейнеры **`set`** обычно медленнее, чем контейнеры **`unordered_set`**, но они допускают прямую итерацию по подмножествам в зависимости от их порядка. Контейнеры **`set`** обычно реализуются как деревья двоичного поиска.

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

Конструкторы std::set

(1) Пустой

```
explicit set(const key_compare& comp = key_compare( ),
             const allocator_type& alloc = allocator_type( ));
explicit set(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 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 **`set`** как псевдоним его второго параметра шаблона (**`Compare`**).

alloc — объект аллокатора

Контейнер хранит и использует внутреннюю копию этого распределителя.

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

Если `allocator_type` является экземпляром аллокатора по умолчанию (у которого нет состояния), этот параметр не имеет значения.

(2) Диапазонный

```
template <class InputIterator>
set(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 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)

```
set(const set& x);
set(const set& 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 (и перемещения с аллокатором)

```
set(set&& x);
set(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.

(5) Из списка инициализации

```
set(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 **set** 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 — создание набора (constructing sets)

Никакого вывода, только демонстрация некоторых способов создания контейнера типа **set**.

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

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

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

int main () {

  std::set<int> first;                          // пустой набор int'ов
  int myints[] = {10, 20, 30, 40, 50};    // массив из списка инициализации
  std::set<int> second(myints, myints+5); // диапазонный конструктор из массива
  std::set<int> third(second);                  // копия second'a
  std::set<int> fourth(second.begin(), second.end()); // с итераторами
  std::set<int, classcomp> fifth;               // класс в качестве Compare
  bool (*fn_pt)(int, int) = fncomp;             // указатель на функцию fncomp()
  std::set<int, bool(*)(int, int)> sixth(fn_pt); // function pointer as Compare

  return 0;
}
```

operator= — копирование содержимого контейнера

Присваивает контейнеру новое содержимое, заменяя текущее.

(1) Копирование

```
set& operator= (const set& x);
```

Присваивание копированием (1) копирует все элементы из **x** в контейнер, при этом **x** сохраняет свое содержимое.

(2) Перемещение

```
set& operator= (set&& x);
```

Присваивание перемещением (2) перемещает элементы **x** в контейнер, при этом **x** остается в неопределенном, но допустимом состоянии.

(3) Из списка инициализации

```
set& operator= (initializer_list<value_type> il);
```

Присваивание списка инициализатора (3) копирует элементы из **il** в контейнер.
Новый размер контейнера такой же, как размер **x** (или **il**) до вызова.
Элементы, хранящиеся в контейнере до вызова, либо назначаются, либо уничтожаются.

Пример

```
// assignment operator with sets
#include <iostream>
#include <set>

int main () {

  int intarray[] = {12, 82, 37, 64, 15};
  std::set<int> first(intarray, intarray + 5);   // набор из 5 int
  std::set<int> second;                          // пустой набор

  second = first;                            // теперь second содержит 5 int
  first = std::set<int>();                   // а first теперь пуст

  std::cout << "Размер first: "  << int (first.size())  << '\n';
  std::cout << "Размер second: " << int (second.size()) << '\n';
  return 0;
}
```

Вывод

```
Размер first: 0
Размер second: 5
```

`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 **`set::begin()`**.

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

insert() — вставить элементы

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

(1) Один элемент

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

Функция оптимизирует время вставки, если **position** указывает на элемент, который будет следовать[2] 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 **map**) как двунаправленный тип итератора, указывающий на элементы.

---

2 указание на следующий элемент — это для C++11, для C++98 — указание на тот, который будет предшествовать

(3) Из диапазона

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

(4) Из списка инициализации

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

Поскольку элементы в наборе уникальны, операция вставки проверяет, эквивалентен ли каждый вставленный элемент элементу, уже находящемуся в контейнере. Если эквивалентный элемент существует, новый элемент не вставляется, при этом возвращается итератор на существующий.

Внутренне контейнеры набора сохраняют все свои элементы отсортированными в соответствии с критерием, указанным в его объекте сравнения. Элементы всегда вставляются в соответствующие позиции в соответствии с этим порядком.

Возвращаемое значение

Версии с одним элементом (1) возвращают **`pair`**, причем ее член **`pair::first`** устанавливается на итератор, указывающий либо на вновь вставленный элемент, либо на эквивалентный элемент, уже находящийся в наборе. Для элемента **`pair::second`** устанавливается значение **`true`**, если новый элемент был вставлен, и **`false`**, если эквивалентный элемент уже существует.

Версии с подсказкой (2) возвращают итератор, указывающий либо на вновь вставленный элемент, либо на элемент, который уже имел такое же значение в наборе.

Итераторы двунаправленные. **`pair`** — это шаблон класса, объявленный в **`<utility>`**.

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

int main () {

  std::set<int> myset;
  std::set<int>::iterator it;
  std::pair<std::set<int>::iterator, bool> ret;
  // set some initial values:
  for (int i = 1; i <= 5; ++i) myset.insert(i*10);   // set: 10 20 30 40 50
  ret = myset.insert(20);                            // вставлен не будет
  if (ret.second == false) it = ret.first;           // "it" теперь -> на 20
  myset.insert(it, 25);                 // вставка с максимальной эффективностью
  myset.insert(it, 24);                 // вставка с максимальной эффективностью
  myset.insert(it, 26);                 // мимо
  int myints[]= {5, 10, 15};            // 10 уже в контейнере, вставки не будет
  myset.insert(myints, myints + 3);

  std::cout << "myset содержит:";
  for (it = myset.begin(); it != myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```

Вывод

```
myset содержит: 5 10 15 20 24 25 26 30 40 50
```

erase() — удалить элементы

Удаляет из установленного контейнера либо один элемент, либо диапазон элементов.

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

```
(1)  iterator  erase(const_iterator position);
(2)  size_type erase(const value_type& val);
(3)  iterator  erase(const_iterator first, const_iterator last);
```

Пример удаления

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

int main () {

  std::set<int> myset;
  std::set<int>::iterator it;

  // что-нибудь вставим:
  for (int i = 1; i < 10; i++)
      myset.insert(i*10); // 10 20 30 40 50 60 70 80 90

  it = myset.begin();
  ++it;                   // "it" -> на 20
```

```
  myset.erase(it);                // 10 30 40 50 60 70 80 90
  myset.erase(40);                // 10 30 50 60 70 80 90
  it = myset.find (60);           //            ^
  myset.erase(it, myset.end());  // 10 30 50

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

  return 0;
}
```

Вывод

```
myset содержит:  10 30 50
```

emplace — создать и вставить элемент

Вставляет новый элемент в набор, если он уникален.

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

Новый элемент создается на месте с использованием аргументов **args** в качестве аргументов для его создания.

Вставка имеет место только в том случае, если никакой другой элемент в контейнере не эквивалентен размещаемому (элементы в контейнере **set** уникальны). Если таки вставлен, размер контейнера увеличивается на единицу.

Внутренне контейнеры **set** 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.

Возвращаемое значение

Если функция успешно вставляет элемент, функция возвращает **pair** с итератором на вновь вставленный элемент и значение **true**.

В противном случае возвращается итератор эквивалентного элемента и значение **false**.

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

pair — это шаблон класса, объявленный в **<utility>**.

std::allocator_traits::construct — создать элемент

Создает объект на месте в положении, на которое указывает **p**, перенаправляя **Args** конструктору класса объекта.

```
#include <memory>
template <class T, class... Args>
static void construct(allocator_type alloc, T* p, Args&&... args );
```

Объект создается на месте без выделения памяти для него. В неспециализированном определении шаблона **allocator_traits**[3] эта функция-член фактически вызывает

```
alloc.construct(p, forward<Args>(args)...)
```

если такой вызов правильно сформирован. В противном случае вызывается:

```
::new(static_cast<void*>(p)) T(forward<Args>(args)...)
```

Специализация для конкретного типа аллокатора может давать другое определение.

Параметры

alloc — объект аллокатора.

allocator_type — псевдоним параметра шаблона **allocator_traits**.

p — указатель на блок памяти.

args — аргументы, передаваемые в конструктор.

Args — список из нуля и большего количества типов.

3 Шаблон **allocator_traits** предоставляет однородный интерфейс для тип **allocator**. Неспециализированная версия шаблона предоставляет интерфейс, который позволяет использовать в качестве аллокатора любой класс, который предоставляет по крайней мере общедоступный член типа **value_type** и общедоступные функции-члены **allocate()** и **deallocate()**.

Пример std::set::emplace

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

int main () {

  std::set<std::string> myset;

  myset.emplace("foo");
  myset.emplace("bar");
  auto ret = myset.emplace("foo");

  if (!ret.second)
      std::cout << "foo уже тут\n";

  return 0;
}
```

Вывод

```
foo уже тут
```

emplace_hint() — создать и вставить элемент с подсказкой

Вставляет новый элемент в набор, если он уникален, используя подсказку о позиции вставки.

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

Новый элемент создается на месте с использованием аргументов **args** в качестве аргументов для его создания.

Вставка имеет место только в том случае, если никакой другой элемент в контейнере не эквивалентен размещаемому (элементы в контейнере **set** уникальны). Если таки вставлен, размер контейнера увеличивается на единицу.

Значение в позиции используется как подсказка для точки вставки. Тем не менее, элемент будет вставлен в соответствующую позицию в соответствии с порядком, описанным его внутренним объектом сравнения — подсказка используется функцией для начала поиска точки вставки, что значительно ускоряет процесс, когда фактическая точка вставки находится либо в правильной позиции, либо близко к ней.

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

Возвращаемое значение

Если функция успешно вставляет элемент, функция возвращает итератор на вновь вставленный элемент. В противном случае возвращается итератор эквивалентного элемента в контейнере.

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

Пример создания и вставки с указанием куда

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

int main () {

  std::set<std::string> myset;
  auto it = myset.cbegin();

  myset.emplace_hint(it,"alpha");                  // в начало
  it = myset.emplace_hint(myset.cend(),"omega");   // в конец
  it = myset.emplace_hint(it,"epsilon");           // эффективно
  it = myset.emplace_hint(it,"beta");              // эффективно

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

  return 0;
}
```

Вывод

```
myset содержит: alpha beta epsilon omega
```

key_comp() — дозвращает объект, используемый для сравнения

Возвращает копию объекта сравнения, используемого контейнером и определяющего порядок элементов.

```
key_compare key_comp( ) const;
```

По умолчанию это функциональный объект **less**, который возвращает то же, что и **operator<**.

Он представляет собой или указатель на функцию или функциональный объект, который принимает два аргумента того же типа, что и элементы контейнера, и возвращает **true**, если считается, что первый аргумент идет перед вторым в том строгом слабом порядке, который он определяет, и **false** в противном случае.

Два элемента набора считаются эквивалентными, если **key_comp** рефлексивно[4] возвращает **false** .

В наборах контейнеров ключами для сортировки элементов являются сами значения, поэтому **key_comp** и его родственная функция-член **value_comp** эквивалентны.

Возвращаемое значение

Объект, используемый для сравнения элементов.

Тип элемента **key_compare** — это тип объекта сравнения, связанный с контейнером, определенный в **set** как псевдоним его второго параметра шаблона (**Compare**).

4 независимо от порядка, в котором элементы передаются в качестве аргументов

Пример

```
// set::key_comp()
#include <iostream>
#include <set>

int main () {

  std::set<int> myset;
  int highest;

  std::set<int>::key_compare mycomp = myset.key_comp(); //
  for (int i = 0; i <= 5; i++) myset.insert(i); // 0 1 2 3 4 5

  std::cout << "myset содержит:";
  highest = *myset.rbegin();                    // 5
  std::set<int>::iterator it = myset.begin();   // auto it = myset.begin();
  do {
    std::cout << ' ' << *it;
  } while (mycomp(*(++it), highest));           // которые < 5
  std::cout << '\n';

  return 0;
}
```

Вывод

```
myset contains: 0 1 2 3 4
```

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 возвращает итератор на него, если тот найден, в противном случае он возвращает итератор **set::end()**.

Два элемента набора считаются эквивалентными, если объект сравнения контейнера рефлексивно возвращает **false**.

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

Пример использования

```
std::set<int> anyset;
...
anyset.erase (anyset.find(40));
```

lower_bound()/upper_bound() — вернуть итератор на нижнюю/верхнюю границу

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

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

Функция **lower_bound()** возвращает итератор, указывающий на первый элемент в контейнере, который либо идет за **val**, либо эквивалентный **val**. Для этого она использует внутренний объект сравнения (**key_comp**) и возвращает итератор на первый элемент, для которого **key_comp(element, val)** вернет **false**.

Если класс набора создается с типом сравнения по умолчанию (**less**), функция возвращает итератор для первого элемента, который не меньше **val**.

Функция **upper_bound()** возвращает итератор, указывающий на первый элемент в контейнере, который идет за **val**. Для этого она использует внутренний объект сравнения (**key_comp**) и возвращает итератор на первый элемент, для которого **key_comp(val, element)** вернет **true**.

Если класс набора создается с типом сравнения по умолчанию (**less**), функция возвращает итератор для первого элемента, большего **val**.

lower_bound() возвращает **set::end()**, если все элементы предшествуют **val**.

upper_bound() возвращает **set::end()**, если после **val** нет никаких элементов.

Пример использования lower_bound() и upper_bound()

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

int main () {

  std::set<int> myset;
  std::set<int>::iterator itlow, itup;

  for (int i = 1; i < 10; i++)
      myset.insert(i*10);             // 10 20 30 40 50 60 70 80 90
  itlow = myset.lower_bound(30);      //       ^                    не перед
  itup  = myset.upper_bound(60);      //                      ^      строго за
  myset.erase(itlow, itup);           // 10 20 70 80 90

  std::cout << "myset содержит: ";
  for (std::set<int>::iterator it = myset.begin(); it != myset.end(); ++it)
      std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Вывод

```
myset содержит: 10 20 70 80 90
```

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**. Поскольку все элементы в контейнере типа **set** уникальны, возвращаемый диапазон будет содержать не более одного элемента.

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 набора считаются эквивалентными, если объект сравнения контейнера рефлексивно возвращает false.

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 поиск эквивалентных элементов

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

int main () {

  std::set<int> myset;

  for (int i = 1; i <= 5; i++) myset.insert(i*10);   // myset: 10 20 30 40 50

  std::pair<std::set<int>::const_iterator, std::set<int>::const_iterator> ret;
  ret = myset.equal_range(30);
  std::cout << "the lower bound -> : " << *ret.first  << '\n';
  std::cout << "the upper bound -> : " << *ret.second << '\n';

  ret = myset.equal_range(35);
  std::cout << "the lower bound -> : " << *ret.first  << '\n';
  std::cout << "the upper bound -> : " << *ret.second << '\n';
}
```

Вывод

```
the lower bound -> : 30
the upper bound -> : 40
the lower bound -> : 40
the upper bound -> : 40
```