1. Пары

Тип `pair` позволяет работать с двумя величинами как с единым целым. Конструктор по умолчанию создает пару значений, инициализируемых конструкторами по умолчанию соответствующих типов. Например, `pair<int, float>a` инициализирует оба значения нулями.

Обращение к элементам пары подобно структуре, поля называются `first` и `second` . В случае статического описания используется ., в случае указателя или итератора — ->. Например, `a.first, a.second` или `a->first, a->second`.

Создать пару можно либо используя оператор присвоения: `a.first = 5, a.second = 7.6`, либо используя специальную функцию `make_pair(5, 7.6)`. Явное присвоение лучше в случае перегруженных функций, так как 7.6 в качестве второго параметра функции интерпретируется как `double` и не может быть использован в перегруженной функции с параметром типа `float`

Сравнение двух переменных типа `pair` происходит по следующему алгоритму:

- две пары равны, если равны их соответствующие элементы.
- одна пара меньше второй, если выполняются следующие условия:

  `x.first < y.first || (x.first == y.first && x.second < y.second)`

Листинг 1. Пример работы с вектором

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      pair<int, char> a; //пустая пара
7
8      a. first  = 1; //заполнили
9      a.second = 'C';
10
11     cout << a.first  << "  " << a.second << endl;
12
13     pair<int, char> b(2, 'B');//инициализируем сразу
14
```

```
    cout << b.first << " " << b.second << endl;

    pair<int, char> c;
    c = make_pair(1, 'A');//заполняем

    cout << c.first << " " << c.second << endl;

    if(a < c) cout << "a < c\n";//сравнение
    else cout << "c < a\n";

    system("pause");
    return 0;
}
```

В результате `c` меньше чем `a`.

2. Контейнеры

Контейнеры делятся на две больших группы: *последовательные* и *ассоциативные*.

Последовательные контейнеры представляют собой набор элементов, расположение которых в контейнеры зависит от порядка поступления в контейнер: добавление элемента происходит либо в конец контейнера, либо в начало. К последовательным контейнерам относятся `vector, list, deque.`

Ассоциативные контейнеры представляют собой отсортированную последовательность, расположение элемента зависит от его значения. К ассоциативным контейнерам относятся `set, multiset, map, multimap.`

Для контейнеров должны выполняться три основных требования:

1. Поддерживается семантика значений вместо ссылочной семантики. При вставке элемента контейнер создает его внутреннюю копию, а не сохраняет ссылку на объект.

2. Элементы в контейнере располагаются в определенном порядке. При повторном переборе порядок должен остаться прежним. Для этого определены возвращающие итераторы для каждого контейнера.

3. В общем случае операции с контейнерами небезопасны. Необходимо следить за выполнением операций.

Для использования контейнеров необходимо подключать соответствующие библиотеки. Например, для использования списка подключается библиотека `#include<list>` и т. д.

Описание любого контейнера: `cont<type> x;`, где `cont` — наименование контейнера, `type` — тип элементов (может быть любым, включая контейнеры). Например, `vector<int> c`.

Основные методы, определенные для всех контейнеров:

- `x.size()` — возвращает размер контейнера.
- `x.empty()` — возвращает `true`, если контейнер пустой.
- `x.insert(pos, elem)` — вставляет копию `elem` в позицию `pos`. Возвращаемое значение зависит от контейнера.
- `x.erase(beg,end)` — удаляет все элементы из диапазона $[beg, end)$.
- `x.clear()` — удаляет из контейнера все элементы.

Для каждого контейнера определен свой итератор. Итератор — это «умный» указатель, который содержит данные о расположении элементов. Для каждого контейнера запись `p++` означает переход к следующему элементу, который определен для каждого контейнера по-разному.

Описание итератора `vector<int>::iterator iter`. Обращение к элементу, на который указывает итератор: `*iter`.

Для каждого контейнера определены два итератора `x.begin()`, определяющий положение начального элемента контейнера и `x.end()`, определяющий адрес следующей ячейки после последнего элемента контейнера. Многие алгоритмы, возвращают `x.end()` в качестве отсутствующего результата (например, при поиске элемента в контейнере возвращается либо итератор на этот элемент, либо `x.end()`, если искомого элемента в контейнере не существует).

2.1. Вектора

Вектор — контейнер, по своей структуре представляющий динамический массив.

Достоинства — возможность произвольного доступа к элементам, возможность резервирования памяти, вставка и удаление в конец вектора за константное время ($O(N)$.)

Недостатки — вставка и удаление в середину списка за линейное время, потеря всех итераторов при вставке и удаление, часть алгоритмов может выполняться с ошибками.

Основные методы вектора:

- `x.push_back(el)` — добавление элемента в конец вектора, `x.pop_back(el)` — удаление элемента из конца вектора.
- `x.at(num)`, `x[num]` — возвращает элемент с индексом `num`
- `x.resize(num)`, `x.resize(num, elem)` — контейнер увеличивается до размера `num`, при этом в первом случае элементы создаются своим контейнером по умолчанию, во втором — элементы создаются как копии `elem`.
- `x.front()`, `x.back()` — возвращает значения первого и последнего элементов вектора.

Для векторов обычно выделяется памяти больше, чем необходимо для контейнера. Существует функция `x.capacity()`, которая возвращает объем выделенной памяти. Пока памяти хватает не происходит перераспределение памяти и итераторов при действиях с элементами.

Можно заранее выделить память под вектора с помощью операции `x.reserve(N)`. Тогда можно вставлять и удалять элементы в вектор без перераспределения памяти.

Листинг 2. Пример работы с вектором

```
#include <iostream>
#include<vector>
using namespace std;

int main()
{
  int n = 10;
  vector<int> a; //описали вектор
  for(int i = 0; i < n; i++)//заполнили
    a.push_back(i);

  for(int i = 0; i < a.size(); i++)//вывод с помощью индекса
    cout << a[i] << " ";
  cout << endl;
```

```
15
16    int x = 11;
17    for(int i = 0; i < a.size(); i++)//вставляем новый элемент после кратных трем
18      if (a[i] % 3 == 0)
19        a.insert(a.begin() + i + 1, x);
20
21    for(vector<int>::iterator it = a.begin(); it != a.end(); it++)//вывод с итератором
22      cout << *it << " ";
23    cout << endl;
24
25    a.erase(a.begin() + 2, a.begin() + 5);//удалили элементы [2, 5)
26
27    for(vector<int>::iterator it = a.begin(); it != a.end(); it++)//вывод с итератором
28      cout << *it << " ";
29    cout << endl;
30
31    system("pause");
32    return 0;
33 }
```

Сначала заполняем вектор числами от 0 до 9. Потом после всех элементов, кратных 3, вставляем 11. Потом удаляем элементы со второго по четвертый. Результат работы программы:

0 1 2 3 4 5 6 7 8 9

0 1 11 1 2 3 11 4 5 6 11 7 8 9 11

0 1 11 11 4 5 6 11 7 8 9 11

2.2. Деки

Дек (англ. deque) — переводится как двусторонняя очередь. Это динамический массив, открытый с обоих концов. Вставка и удаление как с конца, так и с начала выполняется за константное время. Поддерживается произвольный доступ.

Обычно реализуется в виде набора блоков, первый и последний блоки «растут» в противоположных направлениях.

Для работы с деком необходимо подключить библиотеку `#include<deque>`.

Заранее память не выделяется. Из-за необходимости перехода между блоками обращение к элементам и перемещение итераторов происходит медленнее, чем у векторов.

Определены операции, совпадающие с операциями над векторами, кроме `x.capacity()` и `x.reserve(N)`. Дополнительно существует операция добавления и извлечения из начала дека: `x.push_front(el)` и `pop_front()`.

Листинг 3. Пример работы с деком

```
#include <iostream>
#include<deque>
using namespace std;

int main()
{
  int n = 10;
  deque<int> a; //описали вектор
  for(int i = 0; i < n; i+= 2){//заполнили 1 с конца, 2 с начала
    a.push_back(i);
    a.push_front(i + 1);
  }

  for(int i = 0; i < a.size(); i++)//вывод с помощью индекса
    cout << a[i] << " ";
  cout << endl;

  int x = 11;
  for(int i = 0; i < a.size(); i++)//вставляем новый элемент после кратных трем
    if (a[i] % 3 == 0)
      a.insert(a.begin() + i + 1, x);

   for(deque<int>::iterator it = a.begin(); it != a.end(); it++)//вывод с помощью
       итератора
    cout << *it << " ";
  cout << endl;

  a.erase(a.begin() + 2, a.begin() + 5);//удалили элементы [2, 5)

  for(deque<int>::iterator it = a.begin(); it != a.end(); it++)//вывод с помощью
      итератора
    cout << *it << " ";
  cout << endl;

  system("pause");
```

```
34    return 0;
35 }
```

Сначала заполняем дек числами от 0 до 9 (0 добавляем в конец, 1 — в начало, 2 — в конец, 3 — в начало и т. д.). Потом после всех элементов, кратных 3, вставляем 11. Потом удаляем элементы со второго по четвертый. Результат работы программы:

9 7 5 3 1 0 2 4 6 8

9 1 1 7 5 3 1 1 1 0 1 1 2 4 6 1 1 8

9 1 1 1 1 1 0 1 1 2 4 6 1 1 8

2.3. Списки

`list` — контейнер, представляющий из себя двусвязный список. В отличии от векторов и деков не поддерживает произвольный доступ. Следовательно, обращение к элементу контейнера происходит медленнее. Но существуют преимущества относительно векторов и деков.

- Вставка и удаление в любое место контейнера происходит за константное время, так как происходит просто перераспределение указателей, а не сдвиг элементов, как в случае векторов и деков.
- Все итераторы после операций над элементами остаются действительными.
- Практически каждая операция со списками завершается успешно.

Из вышесказанного следует, что для списков не определены операции над памятью, `x.capacity()` и `x.reserve(N)`, также не определена операция доступа к элементу `x.at(num)`.

Для списков часть определена часть операций, являющихся оптимизациями алгоритмов. Как метод класса такие операции работают существенно быстрее.

- `x.remove(val)`, `x.remove_if(func)` — удаляет все элементы в первом случае со значением `val`, во втором — все элементы, для которых `func(elem)` возвращает `true`.
- `x.sort()` — сортирует все элементы оператором $<$
- `x.merge(y)` — перемещаются все элементы `y` в контейнер `x` с сохранением сортировки (списки `x` и `y` должны быть предварительно отсортированы).

- `x.unique()` — удаляет дубликаты (элементы с одинаковыми значениями, расположенные рядом).
- `x.reverse()` — переставляет все элементы в обратном порядке.

Листинг 4. Пример работы со списком

```
#include <iostream>
#include<list>
using namespace std;

int main()
{
   int n = 10;
    list <int> a; //описали список
   for(int i = 0; i < n; i+= 2){//заполнили 1 с конца, 2 с начала
     a.push_back(rand()%5);
     a.push_front(rand()%5);
   }

   for(list<int>::iterator it = a.begin(); it != a.end(); it++)//вывод
     cout << *it << " ";
   cout << endl;

   int x = 11;
   for(list<int>::iterator it = a.begin(); it != a.end(); it++)//вставляем новый элемент
        перед кратными трем
     if (*it % 3 == 0)
       a.insert(it, x);

   for(list<int>::iterator it = a.begin(); it != a.end(); it++)//вывод с помощью итератора
     cout << *it << " ";
   cout << endl;

   a.sort(); //сортируем

   for(list<int>::iterator it = a.begin(); it != a.end(); it++)//вывод с помощью итератора
     cout << *it << " ";
   cout << endl;

   a.unique(); //"удаляем"дубликаты
```

```
34
35    for(list<int>::iterator it = a.begin(); it != a.end(); it++)//вывод с помощью итератора
36      cout << *it << " ";
37    cout << endl;
38
39    system("pause");
40    return 0;
41 }
```

Сначала заполняем вектор случайными числами от 0 до 4. Потом перед всеми элементами, кратными 3, вставляем 11. Потом сортируем список. Потом удаляем дубликаты.

Результат работы программы:

4 3 4 0 2 1 4 4 3 2

4 11 3 4 11 0 2 1 4 4 11 3 2

0 1 2 2 3 3 4 4 4 4 11 11 11

0 1 2 3 4 11

2.4. Множества и мультимножества

Множества (англ. set) — это ассоциативный контейнер, устроенный по типу сбалансированного дерева бинарного поиска. Множество (set) содержит отсортированные по возрастанию элементы (по умолчанию) и не содержит дубликатов. Мультимножество (multyset) содержит дубликаты.

Множества используются для ситуаций, когда необходимо вести поиск по элементам, так как поиск по дереву бинарного поиска (элементы которого определены следующим образом: значения всех элементов слева от корня меньше корня, справа — больше корня) занимает логарифмическое время ($O(\log N)$).

Вставка и удаление элемента требуют существенных временных затрат, так необходимо сохранить структуру дерева. Функция `x.insert(el)` для мультимножества возвращает позицию нового элемента, для множества — пару `pair<iterator, bool>`: позицию нового элемента и `true`, если вставка успешна и `<x.end(), false>` — если в множестве уже существует элемент с таким же значением.

В множестве и мультимножестве существуют специальные функции поиска. Это оптимизированные функции одноименных алгоритмов. Эти функции обеспечивают логарифмическую сложность вместо линейной сложности для универсальных алгоритмов.

- `x.count(elem)` — возвращает количество элементов со значением `elem`
- `x.find(elem)` — возвращает позицию первого элемента со значение `elem` или `x.end()`, если элемента не существует
- `x.lower_bound(elem)` — возвращает первую позицию, в которую может быть вставлен элемент `elem` ($\geqslant$ `elem`).
- `x.upper_bound(elem)` — возвращает последнюю позицию, в которую может быть вставлен элемент `elem` ($>$ `elem`).
- `x.equal_range(elem)` — возвращает пару, содержащую первую и последнюю позиции, в которых может быть вставлен элемент `elem` (интервал, в котором элементы $=$ `elem`).

Например, есть множество $\{1, 2, 4, 5, 6\}$.

Результатом `x.lower_bound(3)`, `x.upper_bound(3)` будет итератор, соответствующий элементу со значением 4. Результатом `x.equal_range(3)` будет пара итераторов, соответствующих элементу со значением 4. Подобный результат говорит о том, что элемента 3 во множестве нет.

Результатом `x.lower_bound(5)` будет итератор, соответствующий элементу со значением 5, результатом `x.upper_bound(5)` — итератор, соответствующий элементу со значением 6. Результатом `x.equal_range(5)` будет пара итераторов, соответствующих элементам со значениями 5 и 6.

Листинг 5. Пример работы с множеством

```
#include <iostream>
#include<set>
using namespace std;

int main()
{
  int n = 10;
  set<int> a; //описали множество
  for(int i = 0; i < n; i++)//заполнили
    a.insert(rand()%10);

  for(set<int>::iterator it = a.begin(); it != a.end(); it++)//вывод
```

```
13      cout << *it << " ";
14    cout << endl;
15
16    //вывод значений итераторов, куда можно вставить 3 и 7
17    cout << *a.lower_bound(3) << " " << *a.upper_bound(3) << " ";
18    cout << *a.equal_range(3).first << " - " << *a.equal_range(3).second << endl;
19
20    cout << *a.lower_bound(7) << " " << *a.upper_bound(7) << " ";
21    cout << *a.equal_range(7).first << " - " << *a.equal_range(7).second << endl;
22
23    system("pause");
24    return 0;
25 }
```

Сначала заполняем вектор случайными числами от 0 до 10. Используем `set`, поэтому дубликаты не вставляются, следовательно, размер контейнера может быть меньше 10. Потом определяем итераторы после выполнения операций `x.lower_bound()`, `x.upper_bound()` и `x.equal_range()` для значений 3 и 7. Итератор вывести на экран нельзя, поэтому выводим только значения элементов, на которые указывает итератор.

Результат работы программы:

0 1 2 4 7 8 9

4 4 4 – 4

7 8 7 – 8

2.5. Отображения и мультиотображения

Отображение (англ. map) — ассоциативный контейнер, содержащий пары «ключ/значение». Сортировка элементов производится по ключу. В отображении ключи уникальны, в мультиотображении могут быть дубликаты.

По своим свойствам соответствуют множеству по ключу. Обладает аналогичными функциями. Можно сказать, что множество — это отображение, у элементов которых совпадают ключ и значение.

Описание отображения `map <int, string> x`.

Отображение можно рассматривать как ассоциативный массив. Возможен прямой доступ к элементу. В качестве «индекса» используется ключ. Операция `m[key]` возвращает ссылку на значение элемента с ключом `key`. Основной недостаток — если элемента

с ключом `key` нет, то этот элемент будет вставлен в контейнер, вместо ожидаемого результата, что такого элемента нет. Следовательно, с ассоциативными контейнерами надо обращаться осторожно.

Листинг 6. Пример работы с множеством

```
#include <iostream>
#include<set>
using namespace std;

int main()
{
  int n = 10;
  set<int> a; //описали множество
  for(int i = 0; i < n; i++)//заполнили
    a.insert(rand()%10);

  for(set<int>::iterator it = a.begin(); it != a.end(); it++)//вывод
    cout << *it << " ";
  cout << endl;

  //вывод значений итераторов, куда можно вставить 3 и 7
  cout << *a.lower_bound(3) << " " << *a.upper_bound(3) << " ";
  cout << *a.equal_range(3).first << " - " << *a.equal_range(3).second << endl;

  cout << *a.lower_bound(7) << " " << *a.upper_bound(7) << " ";
  cout << *a.equal_range(7).first << " - " << *a.equal_range(7).second << endl;

  system("pause");
  return 0;
}
```

Сначала заполняем вектор случайными числами от 0 до 10. Используем `set`, поэтому дубликаты не вставляются, следовательно, размер контейнера может быть меньше 10. Потом определяем итераторы после выполнения операций `x.lower_bound()`, `x.upper_bound()` и `x.equal_range()` для значений 3 и 7. Итератор вывести на экран нельзя, поэтому выводим только значения элементов, на которые указывает итератор. Потом заменяем значения, используя ассоциативный массив (например, если ключа со значением 3 нет, то добавится новый элемент).

Результат работы программы:

$\{0, Q\}\,\{1, D\}\,\{2, F\}\,\{4, H\}\,\{5, L\}\,\{6, R\}\,\{7, P\}, \{8, M\}$

$4\,4\,4 - 4$

$7\,8\,7 - 8$

$\{0, Q\}\,\{1, D\}\,\{2, C\}\,\{3, A\}\,\{4, H\}\,\{5, L\}\,\{6, R\}\,\{7, P\}, \{8, M\}$

2.6. Создание контейнеров

Существует несколько возможностей создания контейнеров.

- Создание пустого контейнера.

  Например, `vector<int> c`

- Создание контейнера размеров n, заполненного элементами по умолчанию (0 — для целых чисел, 0.0 — для вещественных и т. д.).

  Например, `vector<int> c(n)`

- Создание копии контейнера того же типа.

  Например, `vector<int> c(c1)`

- Создание контейнера и инициализация его копиями всех элементов в интервале $[beg, end)$.

  Этот способ позволяет переписывать элементы из одного контейнера в другой. Например, сначала создать список, заполнить его элементами, потом создать копию этого списка в виде множества и использовать поиск уже в множестве.

  ```
  list<int>a(n);
  ```

  Заполнили список. Создали множество, заполнив его копиями этого списка.

  ```
  set<int>b(a.begin(), a.end());
  ```

Листинг 7. Пример различных вариантов создания контейнеров

```
1 #include <iostream>
2 #include <vector>
3 #include <set>
4 #include <list>
5 using namespace std;
6
7 int main()
```

```
{
    int n = 10;
    vector<int> a;//пустой вектор

    for(int i = 0; i < n; i++)//заполнили
      a.push_back(rand()%5);

    for(vector<int>::iterator it = a.begin(); it != a.end(); it++)
      cout << *it << " ";
    cout << endl;

    vector<int>b(n, 5);//вектор из 10 элементов, к-ые = 5

    for(vector<int>::iterator it = b.begin(); it != b.end(); it++)
      cout << *it << " ";
    cout << endl;

    vector<int>c(a); //копия вектора a

    for(vector<int>::iterator it = c.begin(); it != c.end(); it++)
      cout << *it << " ";
    cout << endl;

    set<int>d(a.begin(), a.end()); //множество, копия вектора a

    for(set<int>::iterator it = d.begin(); it != d.end(); it++)
      cout << *it << " ";
    cout << endl;

     list <int>e(a.begin() + 1, a.begin() + 5);//список, копия элементов вектора a с 1 по
         4

    for(list<int>::iterator it = e.begin(); it != e.end(); it++)
      cout << *it << " ";
    cout << endl;

    system("pause");
    return 0;
}
```

Результат:

1 2 4 0 4 4 3 3 2 4

5 5 5 5 5 5 5 5 5 5

1 2 4 0 4 4 3 3 2 4

0 1 2 3 4

2 4 0 4

3. Итераторы

Итератор — это объект, предназначенный для перебора элементов контейнера STL. Итератор представляет некоторую позицию в контейнере.

Основные операторы:

- $*$ — получение значения элемента в текущей позиции итератора (`*iter`). Для `map` необходимо использовать $->$: `iter->first, iter->second`.

- $++$ — перемещение итератора к следующему элементу контейнера. Для разных контейнеров имеет разный смысл:

  Вектор, дек — переход к следующему элементу, увеличивая адрес на `sizeof()` байт.

  Список — переход к следующему элементу по полю `p->next`

  Множество и отображение — переход к следующему элементу, используя симметричный обход дерева.

- $==$ и $!=$ — проверка совпадений позиций, представленных двумя итераторами. Для вектора и дека определены операции сравнения $<$ и $>$.

4. Алгоритмы в библиотеке STL

Для работы с встроенными алгоритмами необходимо подключить библиотеку `<algorithm>`. Некоторые алгоритмы, необходимые для обработки числовых данных, определены в файла `<numeric>`.

Подробно описание алгоритмов рассматривать не будем. Остановимся только на общих принципах.

1. Большая часть алгоритмов возвращает итераторы. Большая часть алгоритмов в качестве параметров использует итераторы. Например, `vector<int>::iterator iter = min_element(x.begin(), x.end())`в качестве параметров использует итераторы. Результатом будет итератор, указывающий на элемент с минимальным значением.

   Очень опасно использовать, например, следующую запись: `remove(x.begin(), x.end(), *iter)`. Предполагается, что с помощью этого алгоритма удаляются все минимальные элементы. Но на самом деле удаляется только элемент, на который указывает `iter`.

   Для правильной работы необходимо выполнить:

   - `vector<int>::iterator iter = min_element(x.begin(), x.end())`
   - `int Min = *iter`
   - `remove(x.begin(), x.end(), Min)`

2. При работе с интервалами всегда подразумевается полуоткрытый интервал $[beg, end)$, т. е., включая beg и не включая end. Естественно, что $beg \leqslant end$.

   Если алгоритм использует несколько интервалов, то для первого задается начало и конец интервала, а для второго — только начало. Например, алгоритм `equal(x.begin(), x.end(), y.end())` сравнивает поэлементно содержимое коллекции x с содержимым коллекции y.

   ВАЖНОЕ ТРЕБОВАНИЕ для алгоритмов, осуществляющих запись в контейнеры или для алгоритмов, использующих несколько интервалов, необходимо заранее убедится, что размер контейнера достаточен для работы с алгоритмом.

3. Алгоритмы, предназначенные для «удаления» элементов (`remove` и `unique`), на самом деле просто переставляют элементы, «удаляя» нужные, но не изменяет размер контейнера. Однако эти алгоритмы возвращают итератор, указывающий на новый конец контейнера. И можно удалить все элементы, расположенные после этого итератора с помощью метода `x.erase()`:

   - `vector<int>::iterator iter = remove(x.begin(), x.end(), val)`

- `x.erase(iter, x.end())`

4. Модифицирующие алгоритмы (алгоритмы, удаляющие элементы, изменяющие порядок их следования или значения) не могут применяться для ассоциативных контейнеров.

Некоторые алгоритмы имеют могут быть представлены в нескольких видах: обычном, с суффиксом `_if` и суффиксом `_copy`.

- Обычный алгоритм используется при передаче значения. Например, `replace(x.begin(), x.end(), old, New)`

  заменяет все элементы со значением `old` значением `New`.

- Алгоритм с суффиксом `_if` используется при передаче функции. Например,

  `replace_if(x.begin(), x.end(), func, New)`

  заменяет все элементы, для которых `func(elem)` возвращает `true`, значением `New`.

- Алгоритм с суффиксом `_copy` используется в случае, когда результат записывается в новый контейнер (или интервал). Необходимо убедиться, что памяти в приемном контейнере хватает для копирования результата. Например,

  `replace_copy(x.begin(), x.end(), y.begin(), old, New)`

  заменяет все элементы со значением `old` значением `New` и копирует результат в контейнер `y`.

Полный список всех алгоритмов можно найти в справочной литературе по библиотеке STL.