# 

[**Стандартная библиотека С++. Назначение, структура и основные принципы организации**](#_ryc7bht3yv73) **3**

[**Контейнер vector. Назначение и основные принципы устройства контейнера**](#_akascv3hawkq) **4**

[**Контейнер vector. Типы данных, итераторы, доступ к элементам, конструирование. Размер и емкость**](#_u74t50hs2wi9) **4**

[**Контейнер vector. Стековые и списочные операции**](#_25o1zj1z1i50) **7**

[**Специализация vector<bool>**](#_oyh9on9kg749) **8**

[**Контейнеры стандартной библиотеки. Классификация и перечень**](#_hbrkwfrytzxf) **9**

[**Контейнеры стандартной библиотеки. Требования к элементам контейнеров**](#_go6bokn0btkk) **10**

[**Контейнер list. Назначение, принципы устройства, основные операции**](#_se1hkndn0hyv) **10**

[**Контейнер deque. Назначение, принципы устройства, основные операции**](#_y5mklh3bszim) **12**

[**Адаптеры стандартных контейнеров. Стек**](#_xaynad713oqr) **13**

[**Адаптеры стандартных контейнеров. Очередь**](#_678rc6x72ko) **14**

[**Адаптеры стандартных контейнеров. Очередь с приоритетом**](#_85x1seev7hhu) **15**

[**Ассоциативный контейнер map. Назначение, принципы устройства, основные операции**](#_m9xl5nynieyh) **17**

[**Ассоциативный контейнер set. Назначение, принципы устройства, основные операции**](#_6k1083d8v66z) **18**

[**Ассоциативные контейнеры multimap и multiset. Назначение, принципы устройства и основные отличия от контейнеров map и set**](#_cgmr65puy9bc) **19**

[**Понятие итератора. Основные принципы**](#_l6i5yvcr6iz7) **20**

[**Категории итераторов**](#_7pbtjxy6ddrc) **20**

[**Обратные итераторы**](#_cay5xa8ei5d6) **21**

[**Потоковые итераторы (ввод и вывод)**](#_n8qyfxf494d4) **21**

[**Понятие аллокатора. Основные принципы**](#_l53ut6f69z6t) **22**

[**Алгоритмы стандартной библиотеки. Основные принципы и классификация**](#_y9lm3zggbbf) **23**

[**Функциональные объекты (функторы). Предикаты**](#_b3r9iflb4ren) **24**

[**Немодифицирующие алгоритмы (for\_each, count, equal, mismatch)**](#_u3hfa29ucjzu) **25**

[**Немодифицирующие поисковые алгоритмы**](#_d7f1up6uqyaw) **26**

[**Модифицирующие алгоритмы (копирующие алгоритмы, transform, unique)**](#_5dchq7f97beo) **27**

[**Модифицирующие алгоритмы (замена и удаление элементов)**](#_qj2s11a85dep) **29**

[**Модифицирующие алгоритмы (fill, generate, reverse, rotate)**](#_3k8bng3jmm1j) **29**

[**Алгоритмы сортировки. Бинарный поиск**](#_j4avtcvtzm0) **30**

[**Алгоритмы слияния и разбиения (merge, partition)**](#_o190ebc8njys) **31**

[**Алгоритмы, реализующие операции над множествами (includes, set\_union, set\_intersection, set\_difference, set\_symmetric\_difference)**](#_aujz7ffg74oa) **32**

# Стандартная библиотека С++. Назначение, структура и основные принципы организации

Библиотека языка С – libc  
Библиотека языка С++ - std, она богаче, включает библиотеку языка С, контейнеры, потоки.  
Boost – библиотека шаблонов, библиотека мусорка, полезного также много, как и мусора  
STL – Standart Template library  
STL != Стандартная библиотека  
std – пространство имен стандартной библиотеки  
std::cout, std::vector – пример  
using namespace std; - плохо

Контейнеры Итераторы Алгоритмы  
(Хранят данные) Обеспечивают доступ к данным Обрабатывают данные

Т[] – массив T\* - указатель

1. Библиотека С++ библиотека С  
#include <x.h> #include<cx>  
#include<stdio> #include<cstdio>

2. Контейнеры  
#include<имя\_к-ра>  
#include<vector>

3. Средства поддержки итераторов  
#include<iterator>

4. Алгоритмы  
#include<algorithm>  
#include<stdlib> ß bsearch, qsort

5. Ввод.вывод  
#include<iostream> - объекты в/вы и операций  
#include<iosfwd> - forward declaration для iostream  
#include<iomanip> - манипулятор для в/вы  
#include<fstream> - файловый в/в  
#include<sstream> - строковый в/в (в память)  
#include<cstdio> - в/в языка C

6. Ошибки  
#include<exception>  
#include<cassert> - assert - проверка

7. Строки  
#include<string>  
#include<cstring> - библиотека С

8. Общие средства  
#include<memory> - работа с памятью  
#include<functional> - объекты – функции

Контейнеры  
MFC Интрузивные std Неинтрузивные  
Хранимые элементы должны не требуют этого условия  
иметь общего предка или содержать  
элементы для связи  
например ссылка на предыдущие и следующие элементы

# Контейнер vector. Назначение и основные принципы устройства контейнера

Контейнер vector

— последовательный контейнер, инкапсулирующий массивы переменного размера;

*(cppreference.com)*

— это массив, размер которого можно менять на этапе выполнения. (динамический массив по умолч.)

*(из лекций)*

Элементы хранятся непрерывно, а значит доступны не только через итераторы, но и через смещения, добавляемые к указателям на элементы. То есть указатель на элемент вектора может передаваться в любую функцию, ожидающую указатель на элемент массива.

Хранилище вектора обрабатывается автоматически, расширяясь и сужаясь по мере необходимости. Векторы обычно занимают больше места, чем статические массивы, поскольку некоторое количество памяти выделяется про запас на обработку будущего роста. Таким образом, память для вектора требуется выделять не при каждой вставке элемента, а только после исчерпания резервов.

Перераспределения обычно являются дорогостоящими операциями в плане производительности. Функция reserve() может использоваться для предварительного выделения памяти и устранения перераспределений, если заранее известно количество элементов.

// Отсюда можно не писать:

Vector считается самым оптимальным в использовании, когда количество элементов не превышает 1000.

Сложность (эффективность) обычных операций над векторами следующая:

* Произвольный доступ — постоянная O(1)
* Вставка и удаление элементов в конце — амортизированная постоянная O(1)+
* Вставка и удаление элементов — линейная по расстоянию до конца вектора O(n)+

# Контейнер vector. Типы данных, итераторы, доступ к элементам, конструирование. Размер и емкость

Объявление

|  |
| --- |
| template <class T, class A = allocator<T>> //размещает объекты в памяти class Vector {  … } vector<int> - аллокатор по умолчанию (в памяти) |

“Типы”

В классе vector объявлены typedefs

|  |
| --- |
| **typedef** T value\_type; **typedef** A allocator\_type; **typedef** **typename** A::size\_type size\_type; **typedef** **typename** A::difference\_type difference\_type; |

прямые итераторы:

|  |
| --- |
| **typedef** <зависит от реализации> iterator;  **typedef** <зависит от реализации> const\_iterator; //только для чтения |

и два обратных итератора (от конца к началу):

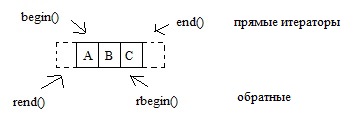
|  |
| --- |
| **typedef** reverse\_iterator<iterator> reverse\_iterator; **typedef** reverse\_iterator<iterator> const\_reverse\_iterator; |

|  |
| --- |
| **typedef** **typename** A::pointer pointer; **typedef** **typename** A::reference reference; |

+ 2 константных аналога

|  |
| --- |
| **template** <**class** C> **typename** C::value\_type **sum**(**const** C &c) {  **typename** C::value\_type s = 0;  **typename** C::const\_iterator p = c.begin();  **while** (p != c.end()) // элемент за концом массива  {  s += \*p;  ++p;  }  **return** s; } |

Итератор - способ получения доступа



Переход от обратного (ri) к прямому (i) итератору: ri.base()

Итератор - не указатель, а указатели являются итераторами.

**Методы доступа к элементам**

|  |
| --- |
| reference **operator**[] (size\_type n); //без проверки const\_reference **operator**[] (size\_type n) **const**; //только у константных объектов |

|  |
| --- |
| **void** **f** (vector<**int**> &v) {  v[0] = 10; //доступ на запись  **int** a = v[0]; //на чтение } **void** **g** (**const** vector<**int**> &v) {  v[0] = 10; //ошибка  **int** a = v[0];  } |

(И у последующих методов есть константные аналоги (“const” в начале и в конце))

|  |
| --- |
| reference **at**(size\_type n); //как operator[], но с проверкой out\_of\_range reference **front**(); //первый элемент, ссылка reference **back**(); //последний элемент, ссылка |

**Конструктор контейнера**

|  |
| --- |
| **explicit** **vector**(**const** A& = A()); //конструктор по умолчанию (explicit чтобы “=” нельзя было писать) **explicit** **vector**(size\_type n, **const** T &val = T(), **const** A& = A()); //создает n копий val (T() - значение типа по умолчанию) vector(In first, In last, **const** A& = A()); //создает вектор из другого контейнера, In - итератор ввода, требует разыменования на чтение vector <**string**> v = {“один”,”два”,”три”}; //initializer\_list vector <**string**> v1 (v.begin(), v.begin()+1); //v1 = {“один”} vector <**string**> v2 (v.end(), v.begin()); //неопределенное поведение |

|  |
| --- |
| vector (In first, In last) {  **while** (first++ != last)  … } vector <**int**> v2(1000,1); //тысяча единиц **template** <**class** In> **void** **assign**(In first, In last); **void** **assign**(size\_type n, **const** T &val); |

**Размер и емкость**

|  |
| --- |
| size() - размер массива capacity() - емкость массива **void** **resize**(size\_type sz, T val = T()); // новый размер **void** **reserve**(size\_type n); // выделяет память под n элементов size\_type **max\_size**() **const**; // сколько max элементов может быть на этой платформе vector<**bool**> - битовый вектор, “недешево” |

# 

# Контейнер vector. Стековые и списочные операции

### Стековые операции

|  |
| --- |
| **void** **push\_back**(**const** T &val); //добавление элемента в конец **void** **pop\_back**(); //удаление последнего элемента |

\*Примечание(не факт, что относятся к стековым):

reference front() - Предоставляет доступ к первому элементу

reference back() - Предоставляет доступ к последнему элементу

Существуют также их константные аналоги.

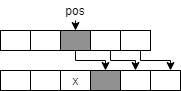
### 

### Списочные операции

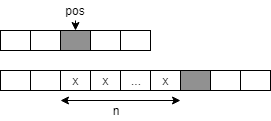
insert - вставка в вектор

erase - удаление из вектора

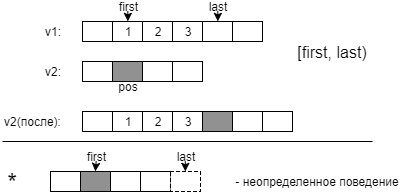
clear - очистка вектора

iterator **insert**(iterator pos, **const** T &x); //Возвращается итератор на элемент x

|  |
| --- |
| iterator **insert**(iterator pos, size\_type n, **const** T &x); //Возвращается итератор на первый x |



|  |
| --- |
| **template**<**class** In> **void** **insert**(iterator pos, In first, In last); |

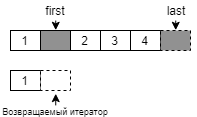


\*Примечание: данное неопределенное поведение возникает тогда, когда берем и вставляем из одного и того же места.

|  |
| --- |
| iterator **erase**(iterator pos); |



|  |
| --- |
| iterator **erase**(iterator first, iterator last); |

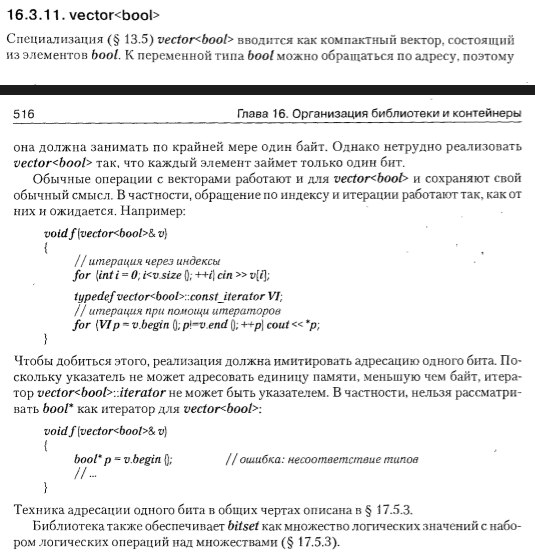


|  |
| --- |
| clear() ~ erase(begin(), end()); |

# 

# Специализация vector<bool>

Специализация vector с типом bool представляет собой битовый вектор, то есть обеспечивает оптимизацию пространства путем сохранения одного значения на бит, а не байт.



# Контейнеры стандартной библиотеки. Классификация и перечень

Контейнеры стандартной библиотеки:

*Последовательные:*

|  |
| --- |
| vector, list, deque |

*Ассоциативные:*

|  |
| --- |
| map, set, multimap, multiset |

Классификация:

* интрузивные

хранимые элементы должны иметь общего предка или содержать эл-ы для связи с др.

* неинтрузивные

не требуют общего предка и эл-та для связи

Пример интрузивного контейнера:

struct Node

{

Node \*prev, next;

};

struct List

{

Node \*head;

};

# Контейнеры стандартной библиотеки. Требования к элементам контейнеров

Контейнеры стандартной библиотеки:

*Последовательные:*

|  |
| --- |
| vector, list, deque |

*Ассоциативные:*

|  |
| --- |
| map, set, multimap, multiset |

Требования к элементам:

1. Элемент должен уметь копироваться (**operator**= конструктор копирования)
2. Элементы должны поддерживать сравнение (**operator**<), причем **operator**< удовлетворяет требованиям строгого слабого упорядочения (cmp):

а) cmp(x,x) == false

б) если cmp(x,y) == true и cmp(y,z) == true, то cmp(x,z) == true

в) equiv(x,y) определяется как !(cmp(x,y) || cmp(y,x))

1. Элементы контейнера должны иметь конструктор по умолчанию

Const T& = T()

# Контейнер list. Назначение, принципы устройства, основные операции

1. Последовательный контейнер;
2. Представляет собой двусвязный список;
3. Оптимизирован под операции вставки и удаления из середины.

List поддерживает почти весь набор операций характерных для деков и векторов, но List **не имеет**

* reserve;
* capacity;
* at и [ ];

Сортировка по неубыванию

|  |
| --- |
| **template** <**class** Cmp> **void** **sort**(); |

Cортировка по Cmp

|  |
| --- |
| **template** <**class** Cmp> **void** **sort**(Cmp);//Cmp-компаратор |

Перемещение элементов списка (без копирования)

|  |
| --- |
| **void** **splice** (iterator pos, list &x);//pos-указатель на место вставки **void** **splice** (iterator pos, list &x,iterator p); **void** **splice** (iterator pos, list &x, iterator first, iterator last);//[first;last) |

Cлияние

|  |
| --- |
| **void** **merge**(list & x);//сливает \*this и x |

Пример

|  |
| --- |
| lis1.sort(); lis2.sort(); lis1.merge(lis2);//lis2.empty()==true |

|  |
| --- |
| back();//ссылка на последний элемент push\_back();//Добавление элемента в конец front();//ссылка на первый элемент push\_front();//вставка в начало |

Удаляет элемент по значению

|  |
| --- |
| **void** **remove** (**const** T& val);// ==val |

Специальный шаблонный метод удаляет по предикату(не по значению)

|  |
| --- |
| **template** <**class** Pred> **void** **remove\_if**(Pred p); |

Пример

|  |
| --- |
| **if** (p(elem) {};//удаляет elem |

Удаляет смежные дубликаты

|  |
| --- |
| **void** **unique**(); |

Переворот списка

|  |
| --- |
| **void** **reverse**(); |

!!!Доп инфа не из лекций

**Список отличий списка(list) от вектора и дека:**

* Отсутствие произвольного доступа. Быстрый доступ существует только к первому и последнему элементу списка. Отталкиваясь от первого или последнего элемента списка, можно получить доступ к любому элементу внутри. Такой обход выполняется по существующей цепочке связей внутри списка. Обход можно делать и с первого и с последнего элемента
* Вставка и удаление элементов выполняется одинаково быстро (при условии, что итератор уже установлен в эту позицию). И вставка и удаление выполняются за константное время. При вставке или удалении не требуется смещать элементы, поэтому и вставка и удаление происходит очень быстро, а все операции сводятся к манипулированию указателями.
* Вставка в список никогда не делает недействительными никакие итераторы, а операция удаления из списка делает невалидными только итераторы, указывающие на удаленный элемент.
* Список поддерживает обработку исключений таким образом, что почти каждая операция либо выполняется успешно либо не выполняется вообще. Таким образом список не может оказаться в промежуточном положении из-за незавершенной операции.
* Списки содержат такие функции, как front(),push\_front(),pop\_front() и back(),push\_back(),pop\_back()
* У списков нельзя получить доступ к элементу подобно массиву, используя [] и нельзя с помощью .at() потому как списки не поддерживают произвольный доступ.
* Списки не поддерживают операции изменения емкости и перераспределения памяти, потому как они спискам не требуются. Каждый объект внутри списка занимает свою собственную ячейку в памяти и эта ячейка будет оставаться корректной вплоть до того момента пока объект не будет удален.
* Списки содержат много функций для перемещения и удаления элементов. Эти функции выполняются быстрее чем аналогичные функции из общих алгоритмов, потому что в случае со списками, функции списков всего-лишь перенаправляют указатели и никак не затрагивают сами объекты
* vector делает упор на быстрый произвольный доступ, a list — на быструю вставку и удаление элементов.

# Контейнер deque. Назначение, принципы устройства, основные операции

deque = vector +list

оптимизирован на вставку/удаление в начало и в конец строки  
deque не имеет capacity и reverse

представляет операцию индексирования

не из лекций

Определено в заголовочном файле <deque>

template< class T, class Allocator = [std::allocator](http://ru.cppreference.com/w/cpp/memory/allocator)<T>> class deque;

std::deque (двусторонняя очередь) представляет собой последовательный индексированный контейнер, который позволяет быстро вставлять и удалять элементы с начала и с конца. Кроме того, вставка и удаление с обоих концов двусторонней очереди оставляет действительными указатели и ссылки на остальные элементы.

В отличие от [std::vector](http://ru.cppreference.com/w/cpp/container/vector), элементы дека не хранятся непрерывно: обычно реализован с помощью набора выделенных массивов фиксированного размера.

Хранилище дека обрабатывается автоматически, расширяясь и сужаясь по мере необходимости. Расширение дека дешевле, чем расширение [std::vector](http://ru.cppreference.com/w/cpp/container/vector), потому что оно не требует копирования существующих элементов в новый участок памяти.

Сложность (производительность) стандартных операций над двусторонней очередью следующая:

* Произвольный доступ - постоянная O(1)
* Вставка и удаление элементов с начала и с конца - амортизированная постоянная O(1)
* Вставка и удаление элементов - линейная O(n)

std::deque отвечает требованиям [Container](http://ru.cppreference.com/w/cpp/concept/Container), [AllocatorAwareContainer](http://ru.cppreference.com/mwiki/index.php?title=cpp/concept/AllocatorAwareContainer&action=edit&redlink=1), [SequenceContainer](http://ru.cppreference.com/w/cpp/concept/SequenceContainer) и [ReversibleContainer](http://ru.cppreference.com/mwiki/index.php?title=cpp/concept/ReversibleContainer&action=edit&redlink=1).

|  |  |
| --- | --- |
| Доступ к элементам | |
| at | Предоставляет доступ к указанному элементу с проверкой индекса |
| operator[] | Предоставляет доступ к указанному элементу |
| front | Предоставляет доступ к первому элементу |
| back | Предоставляет доступ к последнему элементу |
| Итераторы | |
| begin | Возвращает итератор на первый элемент |
| end | Возвращает итератор на элемент, следующий за последним |
| rbegin | Возвращает обратный итератор на первый элемент |
| rend | Возвращает обратный итератор на элемент, следующий за последним |
| Вместимость | |
| empty | Проверяет отсутствие элементов в контейнере |
| size | Возвращает количество элементов в контейнере |
| max\_size | Возвращает максимально допустимое количество элементов в контейнере |
| Модификаторы | |
| clear | Очищает контейнер |
| insert | Вставляет элементы |
| erase | Удаляет элементы |
| push\_back | Добавляет элемент в конец |
| pop\_back | Удаляет последний элемент |
| push\_front | вставляет элементы в начало списка |
| pop\_front | удаляет первый элемент |
| resize | Изменяет количество хранимых элементов |
| swap | Обменивает содержимое |

# Адаптеры стандартных контейнеров. Стек

Адаптеры - интерфейс к некоторому контейнеру:

* stack - стек
* queue - очередь
* priority queue - очередь с приоритетом

|  |
| --- |
| **template**<**class** T, **class** C = deque<T>> **class** stack {  **protected**:  C c;  **public**:  value\_type &**top**() { **return** c.back(); }  **void** **pop**() { c.pop\_back(); }  **void** **push** (**const** value\_type &x) { c. push\_back(x); }  **explicit** **stack** (**const** C &a = C()): **C**(a) {} }; |

# Адаптеры стандартных контейнеров. Очередь

Адаптеры - интерфейс к некоторому контейнеру:

* stack - стек
* queue - очередь
* priority queue - очередь с приоритетом

queue - адаптер “очередь” (основана на деке)

Определенный в заголовочном файле <queue> контейнер queue - это интерфейс для контейнера, позволяющего вставлять элементы в конец ( back() ) и извлекать элементы из начала (front() );

Код на след. странице

|  |
| --- |
| template <**class** **T**, **class** **C** = deque<T>> class std::queue{  **protected**:   C c; **public**:  typedef typename C::value\_type value\_type;  typedef typename C::size\_type size\_type;  typedef C container\_type;  explicit queue(const C&a=C()):c(a){}  bool empty() const{return c.empty();}  size\_type size() const{return c.size();}  value\_type& front(){return c.front();}  const value\_type& front() const{return c.front();}  value\_type& back(){return c.back();}  const value\_type& back() const{return c.back();}  void push(const value\_type& x){c.push\_back(x);}  void pop(){c.pop\_front();} }; |

По умолчанию очередь queue для хранения своих элементов использует deque, но для этой цели может применяться и любая другая последовательность, предоставляющая операции front(), back(), push\_back() и pop\_front(). Поскольку вектор не предоставляет операцию pop\_front(), он не может использоваться в качестве базового контейнера для очередей.

Очереди, кажется встречаются во всех системах. Для простой ситемы, основанной на передаче сообщений, можно определить сервер( программу, обрабатывающую сообщения) например так:

|  |
| --- |
| **struct** Message { //сообщение  //... };  void server {queue<Message>& q}  {  while (! q.empty()) {  Massage& m = q.front(); //получили сообщение  m.service(); //вызвали функцию обслуживания запроса  q.pop(); //уничтожили сообщение  }  } |

Сообщения помещаются в очередь при помощи push().

Дальнейший пример следует переписывать в ответ только если есть четкое понимание каждой лексемы нижеприведенного кода.

Если запрашивающая программа и сервер выполнятся в разных процессах или потоках, для доступа к очереди необходима синхронизация. Например:

|  |
| --- |
| void server2 {queue<Message>& q, Lock& lck}  {  while (! q.empty()) {  Massage& m;  { LockPtr h(lck); //блокируем только на время  //извлечение сообщения  if (q.empty())return; //сообщение получил кто-то другой  m = q.front();  q.pop();  }  m.service();  }  } |

# Адаптеры стандартных контейнеров. Очередь с приоритетом

Адаптеры - интерфейс к некоторому контейнеру:

* stack - стек
* queue - очередь
* priority queue - очередь с приоритетом

Лекция: Элементы выходят из этой очереди согласно приоритету, определенному компаратором.

Очередь с приоритетом (priority queue) это такая очередь, где каждому элементу назначен приоритет определяющий порядок, котором элемент достигают выхода из очереди (top ()):

|  |
| --- |
| template <**class** **T**, **class** **C** = vector<T>, **class** **Cmp** = less<typename C::value\_type>> **class** **priority\_queue**{ **protected**:   C c;  Cmp cmp; **public**:  typedef typename C::value\_type value\_type;  typedef typename C::size\_type size\_type;  typedef C container\_type;   explicit priority\_queue [**const** Cmp&a1-Cmp(),**const** Cmp&a2-C()]:   c(a2), cmp(a1){make\_**heap** (с.begin(), с.**end**(), cmp);}  template <**class** **In**>  **priority**\_**queue** (**In** **first**, **In** **last**, **const** **Cmp**& = Cmp(), **const** C&-C());  bool **empty**() **const**{**return** c.empty();}   size\_type **size**() **const** {**return** с.size();}  **const** value\_type& top() **const** {**return** с.front();}  **void** **push** (**const** value\_type&);   **void** **pop**(); }; |

Объявление priority\_queue находится в заголовочном файле <queue>.

По умолчанию priority\_value просто сравнивает элементы при помощи оператора <, и top() возвращает наибольший элемент:

|  |
| --- |
| **struct** Message {  **int** priority;  **bool** **operator**<(**const** Message& х) **const** {**return** priority <x priority;} }; |

|  |
| --- |
| **void** server (priority\_queue<Message> q, Lock &lck) {  **while** (!q.empty()){  Message m;   {LockРtr **h** (lck); //блокирует только на время  //извлечение сообщения  **if** (q.empty()) **return**; //сообщение получил кто-то другой  m = q.top();   q.pop();  }  m.service(); //вызвали функцию обслуживания запроса  } } |

Этот пример отличается от примера с очередью тем, что сообщения с более высоким приоритетом будут обслуживаться первыми. Порядок, в котором элементы с равными приоритетами достигнут выхода из очереди не определен. Считается, что два элемента имеют одинаковый приоритет, если ни у одного из них приоритет не выше, чем у другого.

Альтернативу оператору < для сравнения элементов можно указать в аргументе шаблона. Например, мы могли бы отсортировать строки без учета регистра, поместив их в

|  |
| --- |
| priority\_queue <string, vector<string>, Nocase> pq; //Используем Nocase для сравнения |

при помощи pq.push(), а затем получив обратно при помощи pq.top() и pq.pop();

Объекты, определенные шаблонами с разными аргументами, относятся к разным типам, например:

|  |
| --- |
| priority\_queue<string>& pq1=pq; //ошибка, типы не совпадают |

Мы можем создать критерий сравнения, не затрагивая типа priority\_queue, задав объект сравнения подходящего типа в качестве аргумента конструктора. Например:

|  |
| --- |
| **struct** String\_cmp{//тип, используемый для выражения критерия сравнения во время выполнения String **cmp**(**int** n = 0);//использовать критерий сравнения n //… };  typedef priority\_queue<**string**,vector<**string**>,String\_cmp>Pqueue;  **void** **g**(Pqueue& pq) //pq использует String\_cmp() для сравнений { Pqueue **pq2**(String\_cmp(nocase)); pq = pq2; // pq и pq2 одинаковых типов } //pq теперь так же использует String\_cmp() |

Поддержание порядка элементов не бесплатно, но не очень дорого. Один полезный способ реализации priority\_queue заключается в использовании дерева, чтобы отслеживать относительное положение элементов. Это приводит к тому что обе операции push() и pop() обходятся как О(log(n)).

По умолчанию очередь с приоритетом для хранения своих элементов использует вектор, но в этом качестве может использоваться и любая другая последовательность, использующая операции front(), push\_back(), pop\_back() и итераторы с произвольным доступом. Очередь с приоритетом скорее всего реализуется при помощи кучи.

# Ассоциативный контейнер map. Назначение, принципы устройства, основные операции

Key -> value  
В map все элементы отсортированы по возрастанию ключа  
Key – НЕ обязательно целое целое число  
Не хранит значения, а хранит пары

|  |
| --- |
| **template** <**class** **Key**, **class** **T**, **class** **Cmp** = **less**<**key**> (**less**<**T**> - это **T**.operator<), **class** **A** = **allocator**<**pair**<**const** **Key**, **T**>>> **class** **map**; |

map хранит pair<const Key, T>

[] – оператор индексирования  
T& **operator**[] (**const** Key &k);  
Lvalue(запись) = rvalue(чтение)  
если k не найдено/нет, то \*this[k] = T();  
Сложность доступа по индексу O(log2N), N – количество пар 2, так как дерево сбалансированное бинарное

find - поиск элемента по ключу  
возвращает итератор на э-т, если не нашел, то на конец map  
iterator **find**(**const** Key &k)

Count – количество эл-тов с заданным ключом  
size\_type **count**(**const** Key &k) **const**;

Возвращает итератор на первый элемент не менее, чем заданное:  
Iterator **lower\_bound**(**const** Key &k);

Возвращает итератор на первый элемент больше, чем заданное:  
Iterator **upper\_bound**(**const** Key &k);

Возвращает набор элементов для конкретного ключа  
pair <iterator,iterator> equal\_range(**const** Key &k);

# Ассоциативный контейнер set. Назначение, принципы устройства, основные операции

Set — это ассоциативный контейнер, который содержит упорядоченный набор уникальных объектов типа Key(key == value — сам ключ является значением). Сортировка элементов осуществляется применением функции Compare к ключам множества, сортировка по возрастанию. Операции поиск, удаление и вставка имеют логарифмическую сложность O(log n)+. Наборы обычно реализуется как красно-черные деревья. Поддерживает итераторы обратного доступа.

Ассоциативные контейнеры set (множества) можно рассматривать как ассоциативные массивы map, для которых значения не важны и отслеживаются лишь ключи. Это приводит к незначительным изменениям в пользовательском интерфейсе:

|  |
| --- |
| **template<class Key, class Cmp = less<Key>, class A = allocator<Key> >**  **class std::set**  **{**  **public:**  // как map, за исключением:  **typedef Key value\_type;** // сам ключ является значением  **typedef Cmp value\_compare;**  // нет операции индексирования []  **};** |

Множества полагаются на операцию сравнения (по умолчанию <), а не на операцию равенства ==. Это означает, что эквивалентность элементов определяется через неравенство, и что итерирование по контейнеру set выполняется строго в определенном порядке. Как и mар, контейнер set предоставляет операции ==, ! =, <,>,<=,>= и swap ().

# Ассоциативные контейнеры multimap и multiset. Назначение, принципы устройства и основные отличия от контейнеров map и set

**std::multimap**

|  |
| --- |
| **template**<  **class** Key,  **class** T,  **class** Compare = std::less<Key>,  **class** Allocator = std::allocator<std::pair<**const** Key, T> > > **class** multimap; |

Multimap это ассоциативный контейнер, который содержит упорядоченный список пар "ключ-значение". В отличие от map, в multimap допускаются ключи с одинаковыми значениями. Сортировка выполняется по ключам в соответствии с функцией сравнения Compare. Операции поиска, и удаления имеют логарифмическую сложность.

Порядок пар ключ-значение с одинаковыми ключами соответствует порядку вставки пар и не меняется. (начиная с C++11)

Печать всех значений с ключом “Stroustrup” (C++11)

|  |
| --- |
| **void** **print\_member**(**const** multimap <string, **int**> &pb) {  **auto** it = pb.equal\_range(“Stroustrup”);  **for**(**auto** i = it.first; i != it.second; ++i)  cout << i -> second << endl; } |

Вставка значений:

|  |
| --- |
| multimap <**char**\*, **int**> m; m.insert(make\_pair(“x”, 4)); m.insert(make\_pair(“x”, 5)); |

**std::multiset**

|  |
| --- |
| **template**<  **class** Key,  **class** Compare = std::less<Key>,  **class** Allocator = std::allocator<Key> > **class** multiset; |

Multiset это ассоциативный контейнер, который содержит упорядоченный набор объектов типа Key. В отличие от set, в multiset допускаются ключи с одинаковыми значениями. Сортировка производится с помощью функции сравнения ключей Compare. Операции поиска, вставки и удаления имеют логарифмическую сложность.

Multimap и multiset обычно реализуются как бинарные деревья поиска.

# Понятие итератора. Основные принципы

То, что выглядит и ведет себя как итератор, итератором и является.

Итератор - шаблонный класс. Он знает как устроен конкретный контейнер внутри, но наружу выдает тот интерфейс, который дозволен (оговоренный интерфейс)

Пример.

|  |
| --- |
| **template** <**class** C> **typename** C::value\_type **sum**(**const** C &c) {  **typename** C::value\_type s = 0;  **typename** C::const\_iterator p = c.begin();  **while** (p != c.end()) // элемент за концом массива  {  s += \*p;  ++p;  }  **return** s; } |

Итератор - способ получения доступа, он должен уметь:

\*р - разыменование (получение значения)

++р - инкремент

Итератор - не указатель, а указатели являются итераторами.

|  |
| --- |
| X **b**(a); или b = a; //копирование ++а или а++ или \*а++ //инкремент |

# Категории итераторов



Итератор ввода позволяет читать значение из контейнера.

Итератор вывода – записывать в контейнер.

Итератор Forward предназначен для прохода по контейнеру от начала к концу.

Итератор Bidirectional – то же, что и Forward + от конца к началу.

Итератор Random позволяет обратиться к любому элементу контейнера.

1. Все категории итераторов могут:

|  |
| --- |
| X **b**(a); или b = a; //копирование ++а или а++ или \*а++ //инкремент |

1. Input:

|  |
| --- |
| a == b или a != b / /сравнение  =+a или =a-> / /разыменование на чтение rvalue |

1. Output:

|  |
| --- |
| \*a= или a->= / /разыменование на запись lvalue |

не поддерживает сравнения

1. Forward (0+1+2):

|  |
| --- |
| X a(b); ++a == ++b; / /сохранение обхода при копировании |

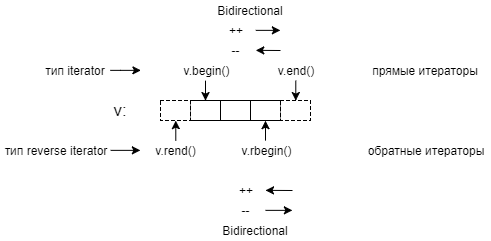
1. Bidirectional (0+1+2+3):

|  |
| --- |
| - -a или a- - или \*a- - / /обход в обратном порядке |

1. Random (0+1+2+3+4):

|  |
| --- |
| a+n или n+a или a-n / /смещение на n элементов  a-b / /разность итераторов  a<b или a>b или a<=b или a>=b / /сравнения  a[n] / /доступ по индексу  a[n] эквивалентно a+n |

# Обратные итераторы



# Потоковые итераторы (ввод и вывод)

## Итератор входного потока

istream\_iterator<T> читает (используя operator>>) последовательные элементы из входного потока, для которого создан. После своего создания итератор каждый раз при использовании операции ++ читает значение типа T. Если достигнут конец потока, итератор становится равным значению end-of-stream. Конструктор без параметров istream\_iterator() всегда создает итераторный объект конца потокового ввода, являющийся единственным законным итератором, который следует использовать для конечного условия.

Основная особенность итераторов входного потока - не гарантируется сохранение равенства при инкременте. Также не допускается доступ к предыдущему элементу.

## Итератор выходного потока

ostream\_iterator<T> записывает (operator<<) последовательные элементы в выходной поток, из которого был создан. Возможно указание дополнительного параметра типа char\*, который будет записываться после каждого значения типа T. С помощью выходного итератора невозможно получить значение.

## Пример

using namespace std;   
   
int main() {   
 istream\_iterator<int> is (cin);   
 ostream\_iterator<int> os (cout, " — new data");   
   
 int input;   
   
 while ((input = \*is) != 33) {   
 \*os++ = input;   
 is++ ;   
 }   
}

# Понятие аллокатора. Основные принципы

### Аллокаторы

Аллокаторы – шаблоны классов, инкапсулирующие стратегию распределения памяти. Это позволяет обобщенным контейнерам отделить управление памятью от фактических данных.

|  |  |
| --- | --- |
| Заголовочный файл <memory> | |
|  | |
| **allocator** | аллокатор по умолчанию |
| **allocator\_traits**  (C++11) | предоставляет информацию о распределитель типа []](http://ru.cppreference.com/mwiki/index.php?title=%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD:cpp/memory/dsc_allocator_traits&action=edit) |
| **allocator\_arg\_t**  (C++11) | тег типа используются для выбора распределитель-Aware перегрузок конструктора |
| **allocator\_arg**  (C++11) | Объект типа std::allocator\_arg\_t используется для выбора распределитель-Aware конструкторов |
| **uses\_allocator**  (C++11) | проверяет, является ли указанный тип поддерживает использование-распределитель строительства |
|  | |
| **scoped\_allocator\_adaptor**  (C++11) | реализует многоуровневую распределитель для многоуровневых контейнеров |

Аллокатор умеет выделять и освобождать память в требуемых количествах определённым образом. std::allocator -- пример реализации аллокатора из стандартной библиотеки, просто использует new и delete, которые обычно обращаются к системным вызовам malloc и free.

Более сложный пример -- pool allocator. Раз системные вызовы дороги, почему бы на них не сэкономить? Выделим сразу 1 гигабайт памяти (к примеру), а дальше в аллокаторе будем выдавать память из этого пула и увеличивать указатель head (опять же, упрощенно). Реальное выделение памяти только одно, системных вызовов почти нет, ура-ура, программа ускорилась.

Собственно, для этого и нужны аллокаторы, чтобы вставлять свое, крутое, нестандартное, выделение памяти в любое место. Большинство стандартных контейнеров их принимают.

Требования к аллокаторам, и их значения по-умолчанию приведены в главе *"Allocator requirements [allocator.requirements]"*. В этой же главе приводится пример минимального аллокатора:

|  |
| --- |
| **template** <**class** Tp>  **struct** SimpleAllocator {  **typedef** Tp value\_type;   SimpleAllocator(аргументы конструктора);  **template** <**class** T> SimpleAllocator(**const** SimpleAllocator<T>& other);  Tp\* **allocate**(std::**size\_t** n);   **void** **deallocate**(Tp\* p, std::**size\_t** n);  };  **template** <**class** T, **class** U>  **bool** **operator**==(**const** SimpleAllocator<T>&, **const** SimpleAllocator<U>&);  **template** <**class** T, **class** U>  **bool** **operator**!=(**const** SimpleAllocator<T>&, **const** SimpleAllocator<U>&); |

Предполагается, что контейнеры обязаны обращаться к аллокатору не напрямую, а через шаблон std::allocator\_traits, который предоставляет значения по-умолчанию, такие как typedef T\* pointer;, и т.п.

# Алгоритмы стандартной библиотеки. Основные принципы и классификация

Все стандартные алгоритмы находятся в пространстве имен std а их объявления - в заголовочном файле <algorithm>.

Принципы:

* Алгоритмы ничего не знают о контейнере, с которым они работают, работают с итераторами.
* Алгоритмы не выполняют проверки диапазона на их входе и выходе. Ошибку выхода за пределы диапазона нужно стараться предотвратить другими средствами.
* Если алгоритм возвращает итератор, этот итератор того же типа, что и был на входе.
* Любой алгоритм является шаблонной функцией.
* Алгоритмы просты.

Алгоритмы получают на вход:

1. Диапазон итераторов [first, last).
2. Определенный итератор pos.
3. Cmp - способ сравнения (компаратор).
4. Предикаты: унарный UnPred, бинарный BinPred.
5. Операции: унарная UnOp, бинарная BinOp.
6. Значение val.

Cmp, UnPred, BinPred, UnOp, BinOp могут быть:

1. обычными функциями;
2. функциональными объектами (функторами); //если нужно хранить состояние
3. Лямбда-функции (начиная с C++11).

Алгоритмы бывают модифицирующие и немодифицирующие.

# Функциональные объекты (функторы). Предикаты

Функциональные объекты

|  |
| --- |
| #**include** <functional> plus ~ a+b minus ~ a-b multiplies ~ a\*b divides ~ a/b modulus ~ a%b negate ~ -a |

|  |
| --- |
| **void** **f**(vector<**double**> &a, vector<**double**> &b, vector<**double**> &res) {  Transform(a.begin(), a.end(), b.begin(), back\_inserter(res), multiplies<**double**>()); } |

Все контейнерные методы работают быстро. Если есть выбор, исп. контейнерную операцию или алгоритм для одной и той же цели, надо выбирать конт. операцию.

Предикаты

Предикат – ф-ия, возвращающая логическое значение

*Унарные* (1 арг.) :

logical\_not ~ !arg

*Бинарные* (2 арг.):

|  |
| --- |
| equal\_to ~ a == b not\_equal\_to ~ a != b greater ~ a > b less ~ a < b greater\_equal ~ a >= b lesser\_equal ~ a <= b logical\_and ~ a && b logical\_or ~ a || b |

|  |
| --- |
| //сравнивает, находя первое расхождение  pair<In1, In2> mismatch (In1 first, In1 last,  In2 first2, BinPred p = equal\_to<**typename** In::value\_type>()) {  while (first!=last && p(\*first, \*first2))  {  ++first;  ++first2;  }  return pair<In1,In2>(first,first2); } mismatch(li.begin(), li.end(), vi.begin(), less\_equal<**int**>()); |

# Немодифицирующие алгоритмы (for\_each, count, equal, mismatch)

в учебнике стр 587 на всякий

Немодифицированные алгоритмы являются основным средством для поиска чего либо в последовательности без написания цикла. эти алгоритмы могут приниматьв качестве аргументов константные итераторы и (за исключением for-each() ) не должны вызывать операции, изменяющие элементы последовательности.

FOR\_EACH

|  |
| --- |
| template<class In, class Op> Op for\_each(In first, In last, Op f)  {  while (first != last)f(\*first++);  return f;  } |

COUNT

|  |
| --- |
| template<class In, class T>  typename iterator\_traits<In>::difference\_type count(In first, In last, const T& val);  template<class In, class Pred>  typename iterator\_traits<In>::difference\_type count\_if(In first, In last, Pred p); |

простая версия count();

|  |
| --- |
| template<class In, class T> int count(In first, In last, const T& val);  {  int res = 0;  while (first != last) if (\*first ++== val) ++ res;  return res;  } |

EQUAL()(равенство) и MISMATCH()(несовпадение) сравнивают две последовательности

equal() проверяет, равны ли соответствующие пары элементов в последовательностях

mismatch() ищет первую пару элементов, которые при сравнении не совпадают, и возвращает итераторы на эти элементы

|  |
| --- |
| template<class In, class In2> bool equal(In first, In last, In2 first2);  template<class In, class In2, class BinPred>  bool equal(In first, In last, In2 first2, BinPred p);  template class In, class In2>  pair<In, In2> mismatch(In first, In last, In2 first2);  template<class In, class In2, class BinPred>  pair(In, In2)mismatch(In first, In last, In2 first2, BinPred p); |

# Немодифицирующие поисковые алгоритмы

template

|  |
| --- |
| In **find**(In first, In last, **const** T& val); |

в интервале ищет значение

|  |
| --- |
| In **find\_if**(In first, In last, UnPred fl); |

ищет элемент удовлетвор. предикату

возвращают итератор на найденный элемент или предикат

xxxxxx\_if - работает с UnPred

поиск последовательности в последовательности => треб. итераторы доступа

|  |
| --- |
| For **find\_first\_of** ( For first, For last, For2 first2, For2 last2); |

1 и 3, 2 и 4 не обязат одинаковы => мб вхождение очереди в список

|  |
| --- |
| For **find\_first\_of**( For first, For last, For2 first2, For last2, BinPred f); |

|  |
| --- |
| **int** x[]={1, 3, 4}, y[]={0, 2, 3, 4 ,5}; **void** **f**() {  **int** \*p = find\_first\_of(x, x+3, y, y+5); //правильно сдвигает итераторы пока не найдет равных  //p=&x[1]; **int** \*q=find\_first\_of(p+1, x+3, y, y+5); //q=&x[2]; } |

Находит пары смежных одинаковых значений

|  |
| --- |
| For **adjacent\_find**(For first, For last); |

+версия с BinPred

перемещается по даиг берет соседние элементы сравнивает

возвращает итератор на 1-й совпад. элемент

Считает количество элементов с заданным значением

|  |
| --- |
| **template**<**class** In, **class** T> **typename** iterator\_treits<In>::difference\_type **count**(In first, In last, **const** T &val); |

+count\_if c … UnPred, f вместо val

количество элементов удовлетворяющих предикату

|  |
| --- |
| **void** **f**(**char** \*p, **int** size) {  **int** n = count(p, p+size, ‘e’); // кол-во e в строке } с пред. : count\_if(p, p+size, [](**char** c)->**bool** {  **return** c=’e’; } count\_if(p, p+size, EPred); **class** **CHarPred** {  **public**:**explicit** **CharPred**(**char** c): **findChar**(c){}  **bool** **operator**()(**char** c) **const**  {  **return** c == findChar; } }  **private**:   **char** findChar; } |

Как использовать:

|  |
| --- |
| **void** **f**(**char** \*p, **int** size) {  **int** n = count\_if(p, p+size, CharPred(‘e’));  **int** m = count\_if(p, p+size, CharPred(‘m’)); } |

less<T> встроенный бин предикат для a<b

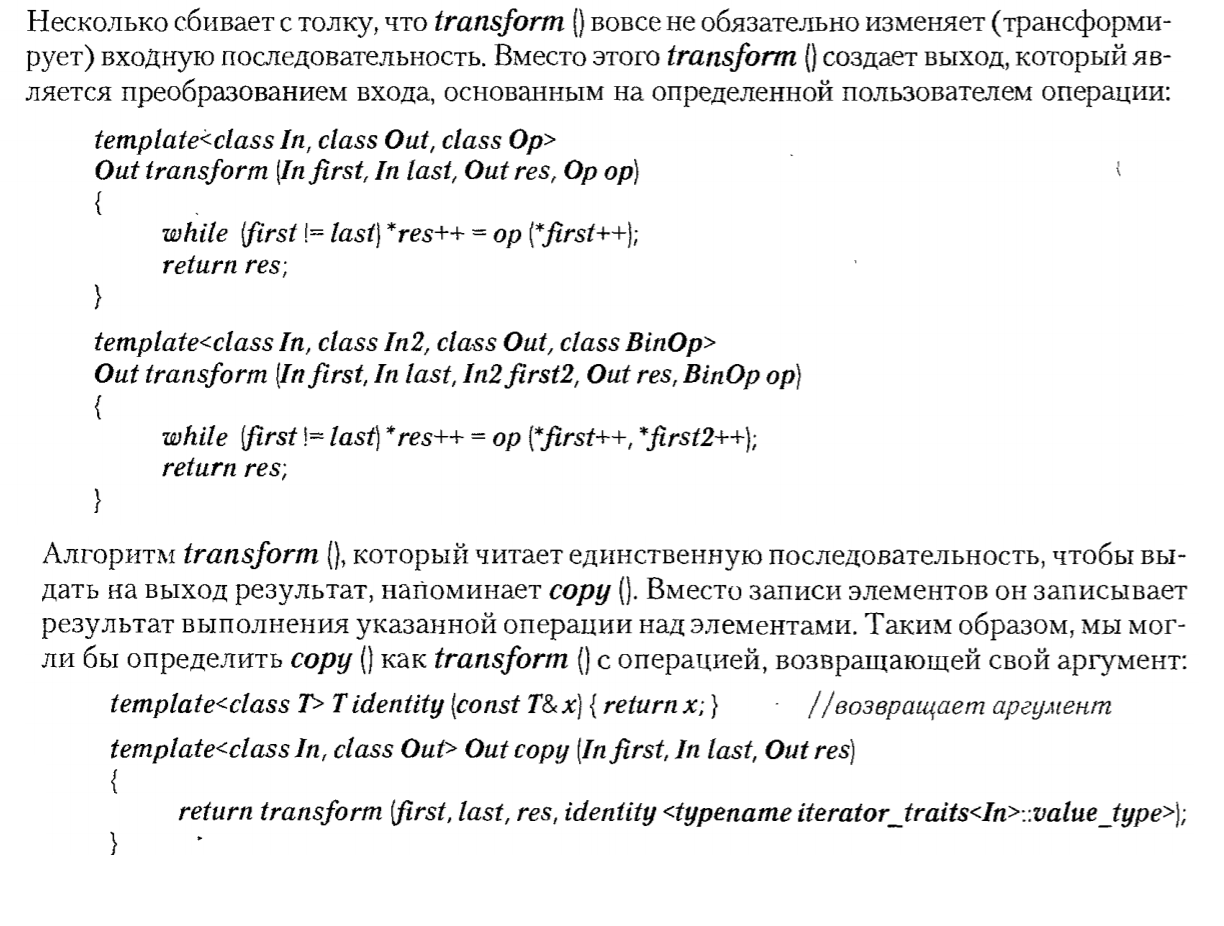
# Модифицирующие алгоритмы (копирующие алгоритмы, transform, unique)

**Копирующие алгоритмы**

|  |
| --- |
| **out** **copy**(**in** first, **in** last, **out** res) {  **while** (first != last)  \*res++ = \*first++;  **return** res; } |

|  |
| --- |
| //копирование наизнанку Bi2 copy\_backward(Bi first, Bi **last**, Bi2 res) {  **while** (first != **last**)  \*--res = \*--**last**;  **return** res; } |

|  |
| --- |
| //копируем то, что удовлетворяет условию  **Out** copy\_if(In first, In last, Out res, Pred p)  {  **while** (first != **last**){  if(p (\*(first)) \*res++ = \*first)  ++first;  }  **return** res;  } |



При сборе информации часто происходит дублирование данных. Алгоритм unique() - модифицирующий алгоритм, который удаляет смежные дубли(соседние одинаковые значения)

|  |
| --- |
| **For** unique(**For** first, **For** last); **For** unique(**For** first, **For** last, BinPred p);  **For** unique(**For** first, **For** last)  {  first = adjacent\_find(first, last);  return unicue\_copy(first, last, first);  } //на самом деле перемещает эл-ты в конец, а дальше нужен erase  Out unique\_copy(In first, In last, Out res); |

|  |
| --- |
| //пример нормального unique template <**class** **C**> **void** **true\_unique**(**C** &**c**) {  sort(c.begin(), c.end());  c.erase(unique(c.begin(), c.end()),c.end()); } |

# Модифицирующие алгоритмы (замена и удаление элементов)

**Алгоритмы замены**

Группа алгоритмов, проходят последовательность, заменяя некоторые значения на заданное значение.

|  |
| --- |
| **void** **replace**(For first, For last, **const** T& val, **const** T& new\_val); **void** **replace\_if**(For first, For last, Pred p, **const** T& new\_val); Out **replace\_copy**(In first, In last, Out res, **const** T& val, **const** T& new\_val); Out **replace\_copy\_if**(In first, In last, Out res, Pred p, **const** T& new\_val) {  **while** (first != last)  {  \*res++ = p(\*first) ? new\_val : \*first;  ++first;  }  **return** res; } |

Пример

|  |
| --- |
| replace(towns.begin(), towns.end(), ”Воронеж”, “Владивосток”); replace\_if(towns.begin(), towns.end(), [](string& v) { **return** strcmp(v.c\_str(), “Воронеж”) == 0;}); |

**Алгоритмы удаления**

Удаляют элементы последовательности, отталкиваясь от их значений или основываясь на предикате.

|  |
| --- |
| For **remove**(For first, For last, **const** T& val); For **remove\_if**(For first, For last, UnPred p); Out **remove\_copy**(In first, In last, Out res, **const** T& val); Out **remove\_copy\_if**(In first, In last, Out res, UnPred p); |

На самом деле remove (и remove\_if) перемещает удаленные элементы в конец и возвращает итератор, на котором начинаются удаленные элементы. Обычно используется вместе с методом erase.

# Модифицирующие алгоритмы (fill, generate, reverse, rotate)

Алгоритмы fill и generate предназначены для систематического присвоения последовательностям значений.

|  |
| --- |
| **void** **fill**(For first, For last, **const** T& val);// присваивает указанное значение **void** **generate**(For first, For last, Gen g); // G - объект, который можно вызвать. Присваивает значения, возвращаемые последовательными вызовами G **void** **fill\_n**(Out res, Size n, **const** T& val); //осуществляет присваивание первым n элементам  **void** **generate\_n**(Out res, Size n, Gen g); //осуществляет присваивание первым n элементам  //Примеры:  fill\_n(back\_inserter(v), 20, 90); srand(time()); generate\_n(back\_inserter(v), 100, []()->**int** {**return** rand();}); |

**void** **reverse**(Bi first, Bi last); - переворачивает последовательность

**void** **reverse\_copy**(Bi first, Bi last, Out res); - перенос в другой контейнер с переворотом

rotate - циклический сдвиг

rotate\_copy - перенос в другой контейнер со сдвигом

|  |
| --- |
| **void** **rotate**(For first, For middle, For last); //Было: [first, … , middle, … , last) //Стало: [middle, … , (last), first, … , last) - вставляет все, что было до middle, Out **rotate\_copy**(For first, For middle, For last, Out res); |

Еще есть такие штуки:

|  |
| --- |
| void random\_shuffle(Rand first, Rand last); - рандомно перемешивает void random\_shuffle(Rand first, Rand last, Gen g); - перегрузка с собственным генератором случайных равномерно распределенных чисел (или другое распределение). Генератор вызывается с аргументом, равным количеству элементов последовательности.  **template** <**class** T> **void** **swap**(T &a, T &b); //меняет a и b местами, можно специализировать |

# Алгоритмы сортировки. Бинарный поиск

|  |
| --- |
| **void** **sort**(Ran first, Ran last); //по умолчанию операция < **void** **sort**(Ran first, Ran last, Cmp cmp); //строгое слабое упорядочение O(N\*logN) ~ O() **void** **stable\_sort**(Ran first, Ran last); //гарантирует, что порядок одинаковых элементов не //меняется (относительный порядок не меняется) **void** **stable\_sort**(Ran first, Ran last, Cmp cmp); |

Алгоритмы частичной сортировки

|  |
| --- |
| **void** **partial\_sort**(Ran first, Ran middle, Ran last);  //+версия с Cmp |

[first, …, middle, …, last)

↓partial\_sort

[first, …, middle, … ,last) //first, …, - отсортировано; middle, …,last - не отсортировано

экономит время, когда не требуется сортировать всё

Сортировка с копированием

|  |
| --- |
| partial\_sort\_copy(In first, In last, Ran first2, Ran last2)  //+версия с Cmp |

Алгоритм n-ный элемент

|  |
| --- |
| **void** **nth\_element**(Ran first, Ran nth, Ran last); |

гарантирует, что на заданном месте будет стоять элемент, который стоял бы там при полной сортировке. (Пример использования: кто на 5 месте?)

Бинарный поиск

применяется к упорядоченной последовательности.

|  |
| --- |
| **bool** **binary\_search**(For first, For last, **const** T &val); //O(logN) //+ версия с Cmp |

find универсальнее, binary\_search быстрее на упорядоченной последовательности

# Алгоритмы слияния и разбиения (merge, partition)

Алгоритмы слияния

|  |
| --- |
| **template** <**class** InputIterator1, **class** InputIterator2, **class** OutputIterator> OutputIterator **merge**(InputIterator1 first1,   InputIterator1 last1, InputIterator2 first2,   InputIterator2 last2, OutputIterator result);  **template** <**class** InputIterator1, **class** InputIterator2,   **class** OutputIterator, **class** Compare>  OutputIterator **merge**(InputIterator1 first1,   InputIterator1 last1, InputIterator2 first2,   InputIterator2 last2, OutputIterator result,   Compare comp); |

merge объединяет два сортированных диапазона [first1, last1) и [first2, last2) в диапазон [result, result + (last1 - first1) + (last2 - first2)). Объединение устойчиво, то есть для равных элементов в двух диапазонах элементы из первого диапазона всегда предшествуют элементам из второго. merge возвращает result + (last1 - first1) + (last2 - first2). Выполняется максимально (last1 - first1) + (last2 - first2) - 1 сравнений. Результат merge не определён, если возникающий в результате диапазон перекрывается с любым из первоначальных диапазонов.

|  |
| --- |
| **template** <**class** BidirectionalIterator> **void** **inplace\_merge**(BidirectionalIterator first,   BidirectionalIterator middle,   BidirectionalIterator last);  **template** <**class** BidirectionalIterator, **class** Compare> **void** **inplace\_merge**(BidirectionalIterator first,   BidirectionalIterator middle,   BidirectionalIterator last,   Compare comp); |

inplace\_merge объединяет два отсортированных последовательных диапазона [first, middle) и [middle, last), помещая результат объединения в диапазон [first, last). Объединение устойчиво, то есть для равных элементов в двух диапазонах элементы из первого диапазона всегда предшествуют элементам из второго. Когда доступно достаточно дополнительной памяти, выполняется максимально (last - first) - 1 сравнений. Если никакая дополнительная память не доступна, может использоваться алгоритм со сложностью O(N\*logN).

Алгоритмы разбиения

|  |
| --- |
| **template** <**class** BidirectionalIterator, **class** Predicate>  BidirectionalIterator **partition**(BidirectionalIterator first,   BidirectionalIterator last, Predicate pred); |

partition помещает все элементы в диапазоне [first, last), которые удовлетворяют pred, перед всеми элементами, которые не удовлетворяют. Возвращается итератор i такой, что для любого итератора j в диапазоне [first, i) будет pred (\*j) == true, а для любого итератора k в диапазоне [i, last) будет pred(\*k) == false. Делается максимально (last - first)/2 перестановок. Предикат применяется точно last - first раз.

|  |
| --- |
| **template** <**class** BidirectionalIterator, **class** Predicate> BidirectionalIterator **stable\_partition**(BidirectionalIterator first,   BidirectionalIterator last, Predicate pred); |

stable\_partition помещает все элементы в диапазоне [first, last), которые удовлетворяют pred, перед всеми элементами, которые не удовлетворяют. Возвращается итератор i такой, что для любого итератора j в диапазоне [first, i) будет pred(\*j) == true, а для любого итератора k в диапазоне [i, last) будет pred(\*k) == false. Относительный порядок элементов в обеих группах сохраняется. Делается максимально (last - first) \* log(last - first) перестановок, но только линейное число перестановок, если имеется достаточная дополнительная память. Предикат

применяется точно last - first раз.

# Алгоритмы, реализующие операции над множествами (includes, set\_union, set\_intersection, set\_difference, set\_symmetric\_difference)

Во всех алгоритмах первая версия ожидает, что оба диапазона должны быть отсортированы с оператором <, вторая версия ожидает, что они должны быть отсортированы с заданной функцией сравнения comp.

# std::includes

|  |
| --- |
| **template**< **class** InputIt1, **class** InputIt2 > **bool** **includes**( InputIt1 first1, InputIt1 last1,  InputIt2 first2, InputIt2 last2 ); |

|  |
| --- |
| **template**< **class** InputIt1, **class** InputIt2 > **bool** **includes**( InputIt1 first1, InputIt1 last1,  InputIt2 first2, InputIt2 last2, Compare comp ); |

Возвращает true, если каждый элемент из отсортированного диапазона [first2, last2) находится в отсортированном диапазоне [first, last). Также возвращает true если [first2, last2) пуст.

# std::set\_union

|  |
| --- |
| **template**< **class** InputIt1, **class** InputIt2, **class** OutputIt > OutputIt **set\_union**( InputIt1 first1, InputIt1 last1,  InputIt2 first2, InputIt2 last2,  OutputIt d\_first ); |

|  |
| --- |
| **template**< **class** InputIt1, **class** InputIt2,  **class** OutputIt, **class** Compare > OutputIt **set\_union**( InputIt1 first1, InputIt1 last1,  InputIt2 first2, InputIt2 last2,  OutputIt d\_first, Compare comp ); |

Создает отсортированный диапазон с началом в d\_first, состоящий из всех элементов, присутствующих в одном или обоих отсортированных диапазонах [first1, last1) и [first2, last2).

# std::set\_intersection

|  |
| --- |
| **template**< **class** InputIt1, **class** InputIt2, **class** OutputIt > OutputIt **set\_intersection**( InputIt1 first1, InputIt1 last1,  InputIt2 first2, InputIt2 last2,  OutputIt d\_first ); |

|  |
| --- |
| **template**< **class** InputIt1, **class** InputIt2,  **class** OutputIt, **class** Compare > OutputIt **set\_intersection**( InputIt1 first1, InputIt1 last1,  InputIt2 first2, InputIt2 last2,  OutputIt d\_first, Compare comp ); |

Создает отсортированный диапазон с началом в d\_first, состоящий из элементов, которые встречаются в обоих отсортированых диапазонах [first1, last1) и [first2, last2).

# std::set\_difference

|  |
| --- |
| **template**< **class** InputIt1, **class** InputIt2, **class** OutputIt > OutputIt **set\_difference**( InputIt1 first1, InputIt1 last1,  InputIt2 first2, InputIt2 last2,  OutputIt d\_first ); |

|  |
| --- |
| **template**< **class** InputIt1, **class** InputIt2,  **class** OutputIt, **class** Compare > OutputIt **set\_difference**( InputIt1 first1, InputIt1 last1,  InputIt2 first2, InputIt2 last2,  OutputIt d\_first, Compare comp ); |

Создает отсортированный диапазон с началом в d\_first, состоящий из элементов из отсортированного диапазона [first1, last1), которые не встречаются в отсортированном диапазоне [first2, last2).

# std::set\_symmetric\_difference



|  |
| --- |
| **template**< **class** InputIt1, **class** InputIt2, **class** OutputIt > OutputIt **set\_symmetric\_difference**( InputIt1 first1, InputIt1 last1,  InputIt2 first2, InputIt2 last2,  OutputIt d\_first ); |

|  |
| --- |
| **template**< **class** InputIt1, **class** InputIt2,  **class** OutputIt, **class** Compare > OutputIt **set\_symmetric\_difference**( InputIt1 first1, InputIt1 last1,  InputIt2 first2, InputIt2 last2,  OutputIt d\_first, Compare comp ); |

Создает отсортированный диапазон с началом в d\_first, состоящий из элементов из отсортированного диапазона [first1, last1), которые не встречаются в отсортированном диапазоне [first2, last2) и элементы из отсортированного диапазона [first2, last2), которые не встречаются в отсортированном диапазоне [first1, last1).