

STL – Standard Template Library

STL опирается на нескольких ключевых концепций:

- **Контейнер (Container):** хранилище для объектов и методы для работы с контейнерами – добавление, удаление, сортировка...
- **Итератор (Iterator):** предоставляет единый интерфейс для доступа к элементам контейнеров, обобщает понятие указателя
- **Алгоритм (Algorithm):** набор «функций» работающих с элементами различных контейнеров; для настройки под конкретные задачи вы должны передать алгоритму вспомогательную функцию

STL: Документация

Онлайн документация

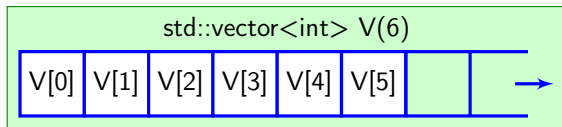
- [Standard Library in C++ reference](#)

Книги

- Николаи Джосаттис «C++. Стандартная библиотека»
(*Nicolai M. Josuttis "C++ Standard Library"*)
 - ✓ второе издание содержит C++11
- Мейерс Скотт «Эффективное использование STL»
 - ✓ углубленное описание, подразумевает знакомство с основами STL

STL: vector<>

Вектор `vector<>` – последовательный контейнер или последовательность

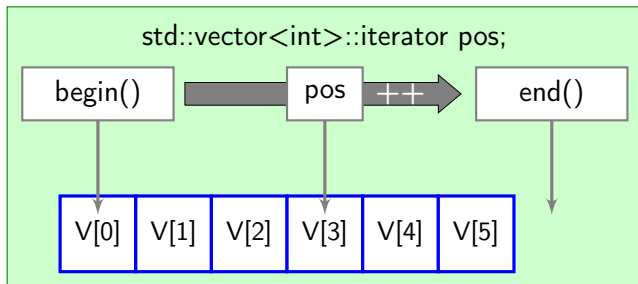


Вектор как динамический массив

- Массив в котором можно легко менять количество элементов
- Элементы вектора хранятся «плотно» и упорядочено:
быстрый доступ к i -му элементу: `v[i] = *(&v[0]+i)`
☞ однако `&v[0]` не является константой и может измениться при выполнении операций с вектором

STL: Итераторы

- С помощью итераторов осуществляется «навигация» и управление содержимым контейнеров
- Методы `V.begin()/V.end()` возвращают итераторы на начало и конец контейнера



Методы `std::vector<>` возвращающие итераторы

<code>begin()</code>	итератор на начало контейнера
<code>end()</code>	итератор на конец контейнера
<code>rbegin()</code>	обратный итератор начала контейнера
<code>rend()</code>	обратный итератор конца контейнера
<code>cbegin()</code>	константные итераторы на начало
<code>cend()</code>	и конец контейнера
<code>crbegin()</code>	константные обратный итераторы
<code>crend()</code>	начала и конца контейнера

- Константный итератор не позволяет менять элемент на которые он указывает:

```
std::vector<int>::iterator pos = V.begin(); // обычный итератор
std::vector<int>::const_iterator first =
    V.cbegin(); // итератор «только для чтения»
```

- набор функций с такими же именами есть у всех контейнеров

Перебор всех элементов вектора

❶ по индексам

```
std::vector<int> V {1,2,3};  
for(size_t i = 0; i < V.size(); ++i) {  
    cout << V[i] << endl; // 1 2 3  
}
```

❷ итераторы (C++98 стиль)

```
for(std::vector<int>::iterator pos=V.begin(); pos!=V.end(); ++pos) {  
    cout << *pos << endl;  
}
```

❸ итераторы (C++11 стиль)

```
for(auto pos=V.begin(); pos!=V.end(); ++pos) {  
    cout << *pos << endl;  
}
```

4 for-range C++11

```
for( auto x : V )           {cout << x << endl;} // local copy
for( auto& x : V )          {cout << x << endl;} // reference
for( const auto& x : V )    {cout << x << endl;} // const reference
```

● «автономные» функции std::begin() и std::end() C++11

```
#include <iterators> // for functions begin() & end()
for(auto pos=begin(V); pos!=end(V); ++pos) { cout << *pos << endl; }
```

● итераторы в цикле while

```
std::vector v {1,2,3}; // C++17 type deduction
auto first = begin(v); // от первого к последнему
while (first != end(v)) {cout << *first++ << endl;} // 1 2 3
auto last = end(v);    // от последнего к первому
while (begin(v) != last) {cout << *--last << endl;} // 3 2 1
```

Автономные функции получения итераторов C++11

- ✓ Для контейнера `C` функции `std::begin(C)` и `std::end(C)` вызывают `C.begin()` и `C.end()`
- ✓ Начиная с C++17 имеется автономная функция `std::size(C)`

Работают и с обычными массивами

```
double arr[] = {0.5,1.2,3.};  
cout << "size(arr)=" << std::size(arr) << endl; // 3  
for( auto pos = std::begin(arr); pos != std::end(arr); ++pos ) {  
    cout << *pos << endl; // 0.5 1.2 3  
}
```

- 👉 В обобщенном программировании предпочтительней использовать автономные функции: `std::begin(C)`, `std::end(C)`, `std::size(C)`; функции реализованы как лямбда, а в C++17 как `constexpr` лямбда

Вектор содержащий объекты класса

✿ Как хранить объекты «своего класса» в векторе?

Класс для тестирования

```
class my_class {
public:
    my_class(char x) {A=x; print("ctor");}
    my_class(const my_class& a) {A=a.A;print("copy ctor");}
    ~my_class() {print("dtor");}
    void print(string msg) {
        cout << msg << ": " << A << endl;
    }
private:
    char A;
};
```

Тестовая программа

```
int main() {  
    my_class a('A'), b('B'), c('C');  
    std::vector<my_class> vec;  
    cout<<"+++capacity= "<<vec.capacity()<<endl;  
    vec.push_back(a);  
    cout<<"+++capacity= "<<vec.capacity()<<endl;  
    vec.push_back(b);  
    cout<<"+++capacity= "<<vec.capacity()<<endl;  
    vec.push_back(c);  
    cout<<"+++capacity= "<<vec.capacity()<<endl;  
}
```

после окончания main()

dtor: C dtor: B dtor: A dtor: C dtor: B dtor: A

ctor: A
ctor: B
ctor: C
+++capacity= 0

copy ctor: A
+++capacity= 1
copy ctor: B
copy ctor: A
dtor: A
+++capacity= 2
copy ctor: C
copy ctor: B
copy ctor: A
dtor: B
dtor: A
+++capacity= 4

Выводы

- ❶ В контейнере хранятся **копии** объектов; при получении объекта из контейнера вы так же получаете **копию**
- ❷ В процессе «хранения» объект может многократно копироваться: копирование производится вызовом копирующего конструктора

- 👉 Помните, что копирование объектов — основа STL
- 👉 Предусмотрите эффективные копирующие конструкторы и операторы присваивания
- 👉 Подумайте о возможной выгоде хранения указателей, а не самих объектов

Вектор указателей

```
std::vector<my_class*> vp;  
vp.push_back( new my_class('A') );  
vp.push_back( new my_class('B') );  
vp.push_back( new my_class('C') );
```

ctor: A
ctor: B
ctor: C

👉 Нет ненужного копирования!
✗ Нет вызова деструкторов!

- 👉 При очистке контейнера уничтожатся указатели, а не сами объекты: помните о возможной утечке памяти
- 👉 Удаление объектов надо делать «вручную»

Вызов delete и очистка вектора

```
for ( auto p : vp ) { delete p; }  
vp.clear();
```

dtor: A
dtor: B
dtor: C

STL: array<>

Модель статического массива: array<Type, Size_t>

- 👉 Массив фиксированного размера с интерфейсом STL контейнера и с *производительностью не хуже чем для обычного C-массива*
- 👉 Параметр `Size_t` в шаблоне задает число элементов на всё время жизни и это **константа времени компиляции**
- 👉 нет операций добавления, удаления изменения размера

Инициализация

```
#include <array>
std::array<int,4> a1;           // elements of a1 undefined
std::array<int,4> a2 {};       // all elements are 0
std::array<int,4> a4 {1,2,3,4}; // explicite values
std::array<int,3> a3 {1,2,3,4}; // ERROR: too many values
std::array a5 {1,2,3,4,5};     // type deduction C++17
```

Пример использования array<>

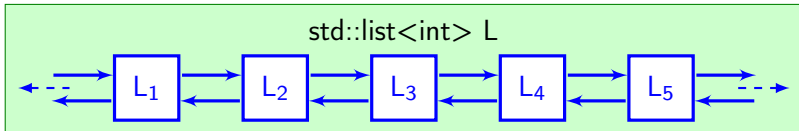
```
std::array ar {1,5,3,7};  
std::sort( begin(ar), end(ar) ); // глобальная функция сортировки  
cout << "sorted a= ";  
for ( auto x : a ) {cout << x << " "};  
cout << endl; // sorted a= 1 3 5 7
```

Передача array<> в функцию

```
template<typename T, size_t S>  
T sum_els(const std::array<T,S>& arr) { // size is a part of the type  
    T s {0};  
    for( const auto& a : arr ) {s += a;}  
    return s;  
}  
  
std::array da {1.1, 2.2, 3.3, 4.4, 5.5};  
auto sum = sum_els(da); // 16.5
```

STL: list<>

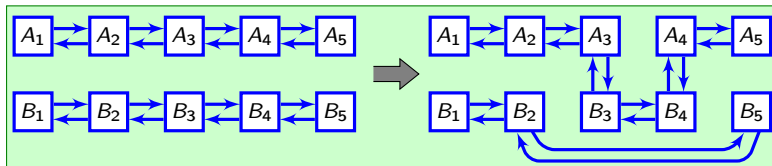
Список (`list`) – *последовательный контейнер*



- Каждый элемент хранится в своей области памяти независимо от других
- Последовательный доступ к элементам списка:
 - **нет доступа по индексу:** $L[i]$ **невозможен**, чтобы получить доступ к i -му элементу, надо последовательно «пройти» от первого элемента
 - медленный доступ к произвольному элементу, быстрый к следующему или предыдущему
 - функция `size()` «перелистывает» весь список

STL list<>: характерные особенности

- ✓ Легко осуществляется добавление и удаление элементов:



- ✓ операция не затрагивает никакие другие элементы, кроме тех с которыми она производится
- ✓ меняются только ссылки на следующий и предыдущий элементы
- ✓ при добавлении нового элемента происходит выделение памяти для одного элемента, при удалении она освобождается
- ✓ Множество специализированных функций для «перемещения» элементов

STL list<>: список методов

```
#include <list>           // заголовочный файл
```

конструкторы

<code>std::list<T> c</code>	пустой список
<code>std::list<T> c(n,el)</code>	список из <code>n</code> элементов <code>el</code>
<code>std::list<T> c {e1,e2,e3}</code>	инициализация списком
<code>std::list<T> c1(c2)</code>	копирующий конструктор

размер

<code>c.size()</code>	количество элементов в списке
<code>c.empty()</code>	<code>true</code> для пустого списка

доступ к элементам


<code>c.front()</code>	первый элемент
<code>c.back()</code>	последний элемент

Специализированные функции для list

<code>c.remove(val)</code>	удаление элементов со значением <code>val</code>
<code>c.remove_if(op)</code>	удаление элементов <code>el</code> если <code>op(el)==true</code>
<code>c.unique([op])</code>	удаляет «равные» соседние элементы
<code>c.reverse()</code>	изменение порядка элементов на обратный
<code>c1.splice(pos,c2)</code>	перемещение всех элементов из <code>c2</code> в <code>c1</code> перед <code>pos</code>

сортировка для list

<code>c.sort([op])</code>	сортировка с помощью <code><</code> или <code>op()</code>
<code>c1.merge(c2)</code>	слияние сортированных списков так что объединение остаётся сортированным

 `[op]` – функция `[необязательная]` возвращающая `bool` (предикат)

STL list<>: Пример

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

● удобная печать коротких списков

```
template<class T>
ostream& operator << (ostream& out, const list<T> & L) {
    for(const auto& l : L) {out << l << " ";} // for-range
    return out;
}
```

● создаем два списка и заполняем их

```
list<int> list1 {1, 2, 3, 4, 5, 6, 7, 8, 9};
list<int> list2; // empty list
for(int i = 1; i < 10; i++) {list2.push_front(i);}
cout << list2 << endl; // 9 8 7 6 5 4 3 2 1
```

● обычные и обратные итераторы

```
// regular iterator
for(auto it = list1.begin(); it != list1.end(); ++it) {
    cout << *it << " ";
}
cout << endl; // 1 2 3 4 5 6 7 8 9

// reverse iterators
for(auto rit = list1.rbegin(); rit != list1.rend(); ++rit) {
    cout << *rit << " ";
}
cout << endl; // 9 8 7 6 5 4 3 2 1
```

● splice(): а) перемещение всего списка

```
// move list2 to the end of list1
list1.splice(list1.end(),           // destination position
             list2);               // source list
cout << list1 << endl; // 1 2 3 4 5 6 7 8 9 9 8 7 6 5 4 3 2 1
cout << list2 << endl; // list2 is empty now
```

● splice(): б) перемещение части списка

```
// move tail of list1 to begin of list2
auto it = list1.begin();
advance(it,11); // increments 'it' by 11
list2.splice( list2.begin(),           // destination position
             list1, it, list1.end() ); // source list in [it end)
cout << list1 << endl; // 1 2 3 4 5 6 7 8 9 9 8
cout << list2 << endl; // 7 6 5 4 3 2 1
```

● метод `remove`: удаление элементов

```
// removes all elements equal to 5
list1 = {1,2,5,3,5,6,7};
list1.remove(5);          // removes all 5
cout << list1 << endl; // 1 2 3 6 7
```

● `remove_if`: удаление элементов удовлетворяющих условию

```
// remove all odd elements
list1 = {1,2,3,4,5,6,7,8,9};
list1.remove_if( [](int el) -> bool {return el%2;} );
cout << list1 << endl; // 2 4 6 8
```

● сортировка

```
// lambdas for sorting list in descending order
auto gt = [](int e1,int e2) {return e1>e2;};
list1 = 4,2,8,5;
cout << "befor sort: " << list1 << endl; // 4 2 8 5
list1.sort(gt);
cout << "after sort: " << list1 << endl; // 8 5 4 2
```

● слияние отсортированных контейнеров

```
// merge sorted lists
list2 = {21,5,-1};
list1.merge(list2,gt); // we must use the same sort function 'gt'
cout << list1 << endl; // list1 remains sorted: 21 8 5 5 4 2 -1
cout << list2 << endl; // list2 is empty now
```

● функция `unique()`: удаляет повторы соседних элементов

```
// remove consecutive duplicates in list
list1 = { 1,1,1, 2,2, 3, 4,4, 1,1,1 };
list1.unique();
cout << list2 << endl; // 1 2 3 4 1
```

● `unique(op)`: удалить соседние элементы одинаковой четности

```
// remove consecutive elements of the same parity
list1 = {1,3,2,4,6,7,8,9};
list1.unique( [](int e1, int e2) {return e1%2==e2%2;} );
cout << list1 << endl; // 1 2 7 8 9
```

● метод `reverse()`

```
// reverses the order of elements in list
list1 = {2,3,5,7};
list1.reverse();
cout << list1 << endl; // 7 5 3 2
```


● Удаление элементов в цикле

```
list l {1,2,3,4,5,6,7,8,9};  
// remove all elements that are divided by 3  
for(auto it=begin(l); it != end(l); ) {  
    if( (*it)%3 == 0 ) {  
        cout << " remove " << (*it) << endl;  
        it = l.erase(it); // 'next' position  
    } else {  
        ++it;  
    }  
}  
cout << " l= " << l << endl;
```

Output:

```
remove 3  
remove 6  
remove 9  
l= 1 2 4 5 7 8
```

Обратите внимание как работает функция `erase(it)`

- 🔗 удаляет элемент на позиции `it` и возвращает итератор следующего за ним элемента
- 🔗 удаление элемента «портит» итератор, который на него указывает, поэтому `++it` после `erase(it)` не работает!

STL: string

В C++ имеется несколько типов для «текстовых объектов»

- 1 С-строинг: `char*` и `const char*`
- 2 классы `string` и `wstring`, а начиная с C++11 `u16string`, `u32string`
- 3 `string_view`: по существу это пара (указатель, длина) на непрерывную последовательность символов, C++17

• `string` значительно облегчает работу с текстом

- ✓ присваивание с помощью `=`
- ✓ сравнение с помощью `==`, `<`, `>` ...
- ✓ слияние с помощью `+`, `+=`
- ✓ выделение части текста, поиск, замена и другое

STL string: создание и копирование

```
#include <string>
using namespace std;
```

● пример

```
string s0;                                // without arguments
cout << "s0= " << s0 << endl;           // s0=      empty string

string s1 ("Initial string");             // text in quotes
cout << "s1= " << s1 << endl;           // s1= Initial string

string s2 = "Second string";              // string with assignment
cout << "s2= " << s2 << endl;           // s2= Second string

// String by repeating one character 1st arg.- number characters,
string s3 (15, '*');                      // 2nd arg.- character itself
cout << "s3= " << s3 << endl;           // s3= *****
```

● создание string из уже имеющегося текста

```
// Constructing a string from existing text
const char* line = "short line for testing";
string s4 (line);           // full copy
cout << "s4= " << s4 << endl;    // s4= short line for testing

// take only the first characters    1st arg.- begin of c-string
string s5 (line,10);         // 2nd arg.- number of characters
cout << "s5= " << s5 << endl;    // s5= short line

// copy substring by index          1st arg.- stl-string
                                // 2nd arg.- start index position
string s6 (s5,6,4);          // 3d arg.- number of characters
cout << "s6= " << s6 << endl;    // s6= line

// copy substring using iterators    1st arg.- start iterator
string s7 (begin(s4),end(s4)-5);    // 2nd arg.- end iterator
cout << "s7= " << s7 << endl;    // s7= short line for te
```

STL string: raw string literals C++11

👉 В C++11 появился удобный способ задания текста содержащего символы обратной косой черты '\\' и новой строки '\n'

старый способ C++98

```
string t1="C:\\A\\B\\file.txt";  
string t2="First\\nSecond\\nThird";  
string t3="First\nSecond\nThird";
```

Raw string literals

```
string r1=R"(C:\A\B\file.txt)";  
string r2=R"(First\nSecond\nThird)";  
string r3=R"(First  
Second  
Third)";
```

печать

```
C:\A\B\file.txt  
First\nSecond\nThird  
First  
Second  
Third
```

STL string: элементарные операции

Размер и ёмкость

<code>size(), length()</code>	размер, длина строки
<code>empty()</code>	<code>true</code> для пустого строки
<code>capacity()</code>	размер зарезервированной памяти
<code>reserve(num)</code>	запрос на резервирование памяти
<code>resize(...)</code>	удаляет или добавляет символы в конец
<code>clear()</code>	удаляет все элементы
<code>shrink_to_fit()</code>	уменьшает <code>capacity</code> до <code>size()</code>

Доступ к отдельным символам текста

<code>[i], at(i)</code>	доступ к i -му символу
<code>front(), back()</code>	первый и последний символы

Лексикографическое сравнение

`==, !=, <, <=, >, >=, compare()`

Модификация строки

<code>+</code>	соединение, конкатенация
<code>+=, append(), push_back()</code>	добавление в конец
<code>insert()</code>	вставка символов в середину
<code>erase(), pop_back()</code>	удаление символов
<code>replace()</code>	замена части строки

Полезные функции

<code>c_str(), data()</code>	возвращает C-string
<code>substr(pos, len)</code>	возвращает часть строки
<code>find(str)</code>	поиск в строке
<code>», «, getline()</code>	операции ввода/вывода в поток

● Преобразование к C-string

- текст хранящийся в `string` можно передавать в «C-функции» с помощью вызова метода `c_str()` или начиная с C++11 `data()`
- обе функции возвращают указатель на массив символов с завершающим нулем, содержащий данные, эквивалентные хранящимся в строке

Пример: печать с помощью `printf`

```
string str("Hello world!\n");  
printf("%s",str.c_str()); // Hello world!  
printf("%s",str.data());  // Hello world!
```


● методы find(), substr()

```
// find 'pi' in a string "K+ pi- K- pi+"
string tst("K+ pi- K- pi+");
auto p = tst.find(string("pi")); // method of string
if ( p != string::npos ) {      // 'npos' means not found
    string sub = tst.substr(p,3); // three symbols starting from 'p'
    cout << "found " << quoted(sub) << endl; // "pi-"
}
```

- static const size_type npos = -1; специальное значение для «неуспеха»: не найдена позиция символа, или ошибки в функции
- std::quoted() заключает строку в кавычки (#include <iomanip>)

● функция sort() для string из #include <algorithm>

```
string ts("K+ K- eta");
sort(begin(ts),end(ts)); // global algorithm
cout << "after " << quoted(ts) << endl; // " +-Kkaet"
```

Преобразования строки ↔ число

функции преобразования в C++11

<code>stoi(), stol(), stoll()</code>	к знаковому целому
<code>stoul(), stoull</code>	к без-знаковому целому
<code>stof(), stod(), stold()</code>	к числу с плавающей точкой
<code>to_string(), to_wstring()</code>	преобразует <code>Int</code> или <code>Float</code> к строке

● `int` → `string` и преобразует как `%d` в `printf`

```
int i = 12345;  
string si = to_string(i);  
cout << "si= " << quoted(si) << endl; // "12345"
```

● `double` → `string` и преобразует как `%f` в `printf`

```
cout << quoted(to_string(M_PI)) << endl; // "3.141593"  
cout << quoted(to_string(1e-7)) << endl; // "0.000000"
```

новые функции в C++17: `from_chars()` и `to_chars()`

```
#include <charconv> // conversion functions in C++17
```

● `from_chars()` string → int

```
string Si = "12345 mm";
int ival = 0;
auto [ptr, ec] = from_chars( Si.data(), Si.data()+Si.size(), ival );
if ( ec == errc() ) {           // check error code
    cout << ival << endl;       // 12345
    cout << quoted(ptr) << endl; // " mm"
} else {
    cout << "Error: " << make_error_code(ec).message() << endl;
}
```

● `from_chars()` string → double

```
string Sd = "1.e-7";
double dval = 0;
from_chars( Sd.data(), Sd.data()+Sd.size(), dval ); // I skip error checking
cout << "dval= " << dval << endl; // dval= 1e-07
```

● `to_chars()` `int` → `string`

```
string St1(10, '_'); // ten underscores
auto [pt1, ec1] = to_chars( St1.data(), St1.data()+St1.size(), 12345 );
if ( ec1 == errc() ) { // check error code
    cout << quoted(St1) << endl; // "12345_ _ _ _ _"
} else {
    cout << "Error: " << make_error_code(ec1).message() << endl;
}
```

● `to_chars()` `double` → `string`

```
string St2(20, ' ');
to_chars( St2.data(), St2.data()+St2.size(), M_PI );
cout << quoted(St2) << endl; // "3.141592653589793 "
St2 = string(10, ' ');
to_chars( St2.data(), St2.data()+St2.size(), 1.2345e-7 );
cout << quoted(St2) << endl; // "1.2345e-07"
```

👉 преобразование с `double` поддерживается не всеми компиляторами

Регулярные выражения (regex) в C++11

Регулярное выражение: «шаблон» для поиска в тексте

- формальный язык для написания таких шаблонов
- функции для манипуляции текстом: поиск, замена, удаление

👉 `#include <regex> // header for regular expression`

● удаление всех цифр из текста

```
// remove all digits from text
auto RmDigits = [](const string& str) {
    regex dig_re("[0-9]"); // шаблон для любой цифры
    return regex_replace(str, dig_re, ""); // замена на пустой строинг
};
string test="D5_Kp6_ct";
string out = RmDigits(test);
cout << test << " -> " << out << endl; // D5_Kp6_ct -> D_Kp_ct
```

● взятие чисел в квадратные скобки

```
// enclose any number in square brackets
auto BrDig = [](const string& str)-> string {
    // шаблон для цифры взятой один или более раз
    static regex dig_re("[0-9]+");
    // замена с подстановкой: $& - то что найдено по шаблону
    return regex_replace(str, dig_re, "$&");
};

string t = "a b1 c22 d333";
out = BrDig(t);
cout << t << " -> " << out << endl;
// a b1 c22 d333 -> a b[1] c[22] d[333]
```

Дополнительные слайды

STL vector: список методов

доступ к элементам вектора V

<code>V[idx]</code>	элемент с индексом <code>idx</code> , как в обычном массиве
<code>V.at(idx)</code>	<code>== V[idx]</code> с проверкой правильности индекса <code>exception</code>
<code>V.front()</code>	первый элемент
<code>V.back()</code>	последний элемент

размер и ёмкость вектора

<code>size()</code>	количество элементов, размер, вектора
<code>empty()</code>	возвращает <code>true</code> для пустого вектора
<code>resize(num, [e])</code>	делает число элементов равным <code>num</code> , удаляя или добавляя <code>[e]</code> из/в конец вектора
<code>reserve(num)</code>	резервирование памяти для <code>num</code> элементов
<code>capacity()</code>	возвращает размер зарезервированной памяти
<code>shrink_to_fit()</code>	уменьшает <code>capacity</code> до <code>size()</code>

STL list<>: вставка и удаление

Эти функции имеются и в vector<>


<code>c.push_back(e1)</code>	добавляет элемент <code>e1</code> в конец списка
<code>c.pop_back()</code>	удаляет последний элемент
<code>c.insert(pos,e1)</code>	вставляет элемент перед итератором <code>pos</code> и возвращает итератор на него
<code>c.clear()</code>	удаляет все элементы
<code>c.erase(pos)</code>	удаляет элемент с итератором <code>pos</code> и возвращает итератор следующего элемента
<code>c.resize(num,[e1])</code>	удаляет/добавляет элементы в конец списка

«Новые функции»

<code>c.push_front(e1)</code>	добавляет элемент в начало списка
<code>c.pop_front()</code>	удаляет первый элемент

STL: ввод-вывод в string

string stream: `#include <sstream>`

 позволяет связать `string` с потоком ввода-вывода и затем использовать имеющиеся функции для записи-чтения в стринг

● «запись» `int` → `string`

```
stringstream ss;           // поток связан с пустым стрингом
int myint = 12345;
ss << myint;               // запись в поток
string mystr = ss.str();   // извлечение стринга
cout << quoted(mystr) << endl; // "12345"
```

● «чтение» `string` → `int`

```
stringstream tt(mystr); // поток связали со стрингом mystr
int tint;
tt >> tint;             // чтение из потока
cout << tint << endl;   // 12345
```