

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):** средство для доступа к элементам контейнера, обобщает понятие указатель
- **Алгоритм (Algorithm):** набор «функций» работающих с различных контейнерами

STL: Документация

Книги

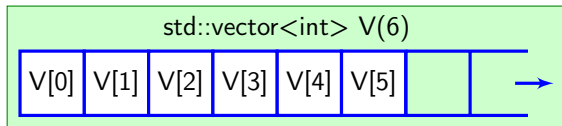
- Николай Джосаттис «C++. Стандартная библиотека»
(*Nicolai M. Josuttis "C++ Standard Library"*)
 - ✓ второе издание содержит C++11
- Мейерс Скотт «Эффективное использование STL»
 - ✓ углубленное описание, подразумевает знакомство с основами STL

Онлайн документация

- C++ reference

STL: vector<>

Вектор `vector<>` – *последовательный контейнер или последовательность*



Вектор как динамический массив

- Элементы вектора имеют определённый порядок, доступ к i -му элементу: `v[i] == *(&v[0]+i)`
 - Обычно вектор удерживает больше памяти чем нужно для хранения элементов, а при необходимости происходит перераспределение памяти
- 🔔 `&v[0]` не является константой и может измениться при выполнении операций с вектором

STL vector: краткий список функций

доступ к элементам вектора V

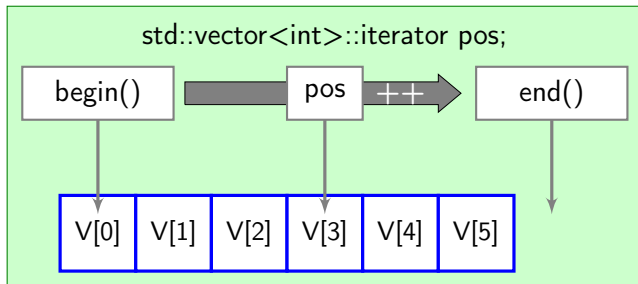
<code>V[idx]</code>	элемент с индексом <code>idx</code> , как в обычном массиве
<code>V.at(idx)</code>	<code>== V[idx]</code> с проверкой правильности индекса exception
<code>V.front()</code>	первый элемент
<code>V.back()</code>	последний элемент


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> (C++11)

STL: Итераторы


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



 `std::vector<int>::const_iterator` – итератор «только для чтения»

Функции `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	(C++11)
<code>cend()</code>	и конец контейнера	(C++11)
<code>crbegin()</code>	константные обратный итераторы	(C++11)
<code>crend()</code>	начала и конца контейнера	(C++11)

 набор функций с такими же именами есть у всех контейнеров

```
vector<int>::iterator pos = V.begin();  
vector<int>::const_iterator first = V.cbegin();
```

Перебор всех элементов вектора

❶ по индексам

```
for ( int i=0; i < V.size(); ++i ) { cout << V[i] << endl; }
```

❷ итераторы (C++98 стиль)

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

❸ итераторы в цикле while (C++98 стиль)

```
vector<int>::const_iterator first = V.begin();  
vector<int>::const_iterator last  = V.end();  
while ( first != last ) {  
    cout << *first++ << endl; // or *--last to move backward  
}
```

4 итераторы (C++11 стиль)

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

4 «автономные» (free) функции begin() & end() (C++11 стиль)

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

5 for-range (C++11 стиль)

```
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. ref.
```


Автономные функции получения итераторов (C++11)

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

Работают и с обычными массивами

```
int arr[] = {1,2,5,6,8,9};  
for ( auto pos = begin(arr); pos != end(arr); ++pos ) {  
    cout << *pos << " ";  
}  
cout << endl; // 1 2 5 6 8 9
```

- 👉 В обобщенном программировании предпочтительней использовать автономные функции: `begin(c)`, `end(c)`, `size(c)`
- 👉 Реализованы как лямбда функции, 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');  
    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

```
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

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

Пример использования array<>:

```
#include <array>
array<int,4> a {1,5,3,7};
sort(a.begin(), a.end()); // функция сортировки для контейнеров
cout << " sorted a= ";
for ( auto x : a ) cout << x << " ";
cout << endl; // sorted a= 1 3 5 7
```

Передача array<> в функцию:

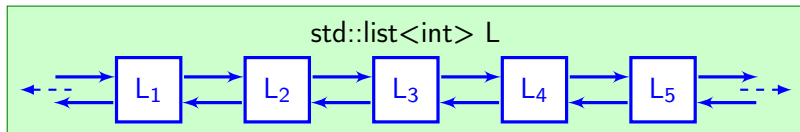
```
template<typename T, size_t S>
T sum_elements(const array<T,S>& arr) { // size is a part of the type
    T s;
    for(const auto& a : arr) { s += a; }
    return s;
}

array<double,5> da {1.1, 2.2, 3.3, 4.4, 5.5};
auto sum = sum_elements(da); // 16.5
```

STL: list<>

Список (list) – последовательный контейнер

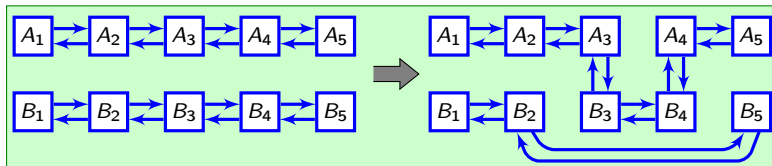
☞ имеет внутреннее устройство совершенно другого типа чем вектор



- Каждый элемент хранится в своей области памяти независимо от других
- Последовательный доступ к элементам списка:
 - нет доступа по индексу: `List[i]` невозможен, чтобы получить доступ к `i`-му элементу, надо последовательно «пройти» предыдущие элементы
 - 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: элементарные операции

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

Конструкторы, копирование, размер списков

<code>list<TYPE> c</code>	пустой список	
<code>list<TYPE> c(n,el)</code>	список из <code>n</code> элементов <code>el</code>	
<code>list<TYPE> c {e1,e2,e3}</code>	инициализация списком	(C++11)
<code>list<TYPE> c1(c2)</code>	копирующий конструктор	
<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

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

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(): a) перемещение всего списка

```
list1.splice(list1.end(),           // destination position
             list2);                // source list
                                   // list2 is empty now

cout << " list1: " << list1 << endl;
    list1: 1 2 3 4 5 6 7 8 9 9 8 7 6 5 4 3 2 1
cout << " list2: " << list2 << endl;
    list2:
```

● splice(): б) перемещение части списка

```
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: " << list1 << endl;    // list1: 1 2 3 4 5 6 7 8 9 9 8
cout << " list2: " << list2 << endl;    // list2: 7 6 5 4 3 2 1
```

● функция `remove`

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

● функция `remove_if`

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


● сортировка `sort()` для `list`

```
cout << " befor sort: " << list1 << endl; // befor sort: 2 4 6 8
auto gt = [](int e1,int e2)->bool{return e1>e2;}; //lambdas for sorting
list1.sort(gt);
cout << " after sort: " << list1 << endl; // after sort: 8 6 4 2
```

● слияние отсортированных контейнеров

```
list2 = {21,5,-1};
list1.merge(list2,gt); // must use the same sort function gt
                        // list2 is empty now
cout << " list1: " << list1 << endl; // list1: 21 8 6 5 4 2 -1
cout << " list2: " << list2 << endl; // list2:
```

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

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

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

```
list1 = {1,3,2,4,6,7,8,9};  
list1.unique( [](int e1, int e2) -> bool{return e1%2==e2%2;} );  
cout << " list1: " << list1 << endl; // list1: 1 2 7 8 9
```

● функция `reverse()`

```
list2 = {2,3,5,7};  
list2.reverse();           // reverses the order of the elements  
cout << " list2: " << list2 << endl; // list2: 7 5 3 2
```

● Удаление элементов в цикле

```
list l {1,2,3,4,5,6,7,8,9};  
// remove all items 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` уже не работает!