

STL – Standard Template Library

STL опирается на нескольких ключевых концепций:

- **Контейнер (Container):** содержит объекты и средства для работы с ними – добавление, удаление, сортировка и другие
- **Итератор (Iterator):** средство для доступа к элементам контейнера, обобщает понятие указатель
- **Алгоритм (Algorithm):** набор «функций» работающих с различных контейнерами
- **Объект-функция (Function Object):** служит для настройки работы алгоритмов под конкретные задачи

STL: Документация

Книги

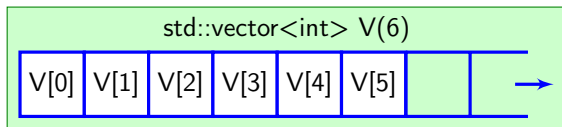
- Николай Джосаттис «C++. Стандартная библиотека»
(*Nicolai M. Josuttis "C++ Standard Library"*)
 - ✓ второе издание содержит C++11
- Мейерс Скотт «Эффективное использование STL»
 - ✓ углубленное описание, подразумевает знакомство с основами STL

Онлайн документация


- C++ reference

STL: vector<>

`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

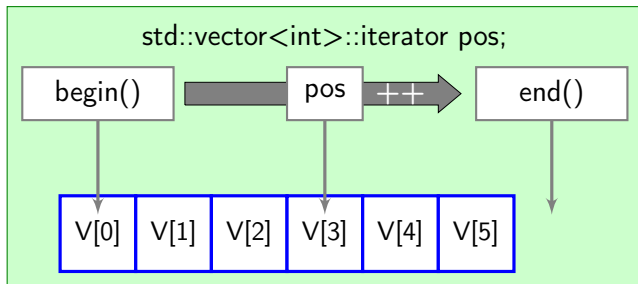
<code>V[idx]</code>	элемент с индексом <code>idx</code> , как в обычном массиве
<code>V.at(idx)</code>	<code>== V[idx]</code> с проверкой правильности индекса exception
<code>V.front()</code>	первый элемент
<code>V.back()</code>	последний элемент


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 на объекты типа `const`

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)` & `end(c)` вызывают `c.begin()` & `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 (с 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 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
```