# 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

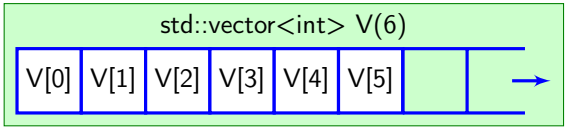
- Standard Library in C++ reference

## 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*

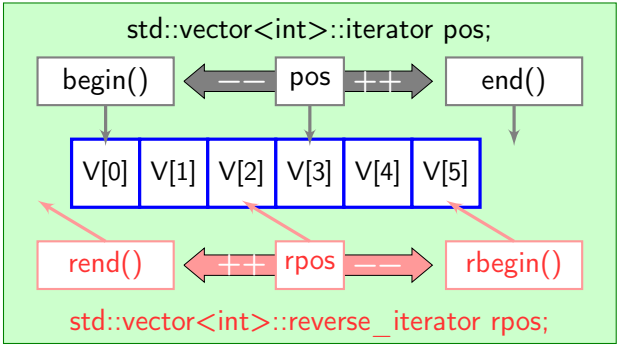



## Вектор как динамический массив

- Массив в котором можно легко менять количество элементов
- Элементы вектора хранятся «плотно» и упорядочено:
  быстрый доступ к *i*-у элементу: `v[i] = *(&v[0]+i)`
  ☞ `&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, а `*pos` – ссылку на его элемент



## Методы std::vector<> возвращающие итераторы

| | |
|---|---|
| `begin()` | итератор на начало контейнера |
| `end()` | итератор на конец контейнера |
| `rbegin()` | обратный итератор начала контейнера |
| `rend()` | обратный итератор конца контейнера |
| `cbegin()` | константные итераторы на начало |
| `cend()` | и конец контейнера |
| `crbegin()` | константные обратный итераторы |
| `crend()` | начала и конца контейнера |

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

- ☞ константный итератор не позволяет менять элемент на которые он указывает
- ☞ набор функций с такими же именами есть у всех контейнеров

# Перебор всех элементов вектора

**1 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
}
```

**2 итераторы (C++98 стиль)**

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

**3 итераторы (C++11 стиль)**

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

## ❹ 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

- ✓ 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 T {
   public:
      T(char x) : A(x)              {prt("ctor");}
      T(const T& x) : A(x.A)        {prt("copy");}
      T(T&& x) : A(std::move(x.A)) {prt("move");}
      ~T()                          {prt("dtor");}
   private:
      char A;
      void prt(std::string&& msg) {
         cout << msg << ": " << A << endl;
      }
};
```

## Тестовая программа

```
int main() {
  T a('A'),b('B')
  std::vector<T> vec;
  cout<<"+++capacity= "<<vec.capacity()<<endl;
  vec.push_back(a);
  cout<<"+++capacity= "<<vec.capacity()<<endl;
  vec.emplace_back(b);
  cout<<"+++capacity= "<<vec.capacity()<<endl;
  vec.emplace_back(T('C'));
  cout<<"+++capacity= "<<vec.capacity()<<endl;
}
```

в конце работы

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

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

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

## Выводы

1. В контейнере хранятся копии объектов; при получении объекта из контейнера вы так же получаете копию
   - ☞ Move семантика в `emplace` функциях позволяет создавать объекты «по месту»
2. 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 «умных» указателей

```
std::vector<std::unique_ptr<T>> vu;
vu.push_back(std::make_unique<T>('a'));
auto upb = std::make_unique<T>('b');
vu.push_back(std::move(upb));
cout << upb.get() << endl; // 0x0
vu.emplace_back(std::make_unique<T>('c'));
```

```
ctor: a
ctor: b
ctor: c
dtor: c
dtor: b
dtor: a
```

- Нет ненужного копирования объектов
- Удаление «автоматическое»

# STL: array<> (C++11)

## 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<>

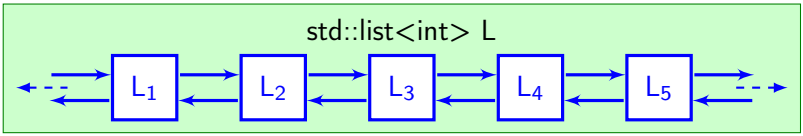
```
std::array ar {1,5,3,7};
std::sort( begin(ar), end(ar) ); // глобальная функция сортировки
cout << "sorted a= ";
for ( auto x : a ) {cout << x << " ";}
cout << endl; // sorted a= 1 3 5 7
```

## 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<>

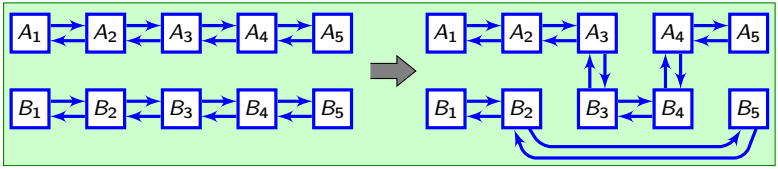
Список (`list`) – *двусвязный список*




- 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
  - `std::forward_list` – односвязный список (C++11)

# 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<>: основное

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

| *конструкторы* | |
|---|---|
| `std::list<T> c` | пустой список |
| `std::list<T> c(n,el)` | список из `n` элементов `el` |
| `std::list<T> c {e1,e2,e3}` | инициализация списком |
| `std::list<T> c1(c2)` | копирующий конструктор |
| *размер* | |
| `c.size()` | количество элементов в списке |
| `c.empty()` | `true` для пустого списка |
| *доступ к элементам* | |
| `c.front()` | первый элемент |
| `c.back()` | последний элемент |

## Специализированные функции для list

| | |
|---|---|
| `c.remove(val)` | удаление элементов со значением `val` |
| `c.remove_if(op)` | удаление элементов `el` если `op(el)==true` |
| `c.unique([op])` | удаляет «равные» соседние элементы |
| `c.reverse()` | изменение порядка элементов на обратный |
| `c1.splice(pos,c2)` | перемещение всех элементов из `c2` в `c1` перед `pos` |
| | сортировка для `list` |
| `c.sort([op])` | сортировка с помощью `<` или `op()` |
| `c1.merge(c2)` | 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>
std::ostream& operator << (std::ostream& out, const std::list<T> & L) {
  for(const auto& l : L) {out << l << " ";} // for-range
  return out;
}
```

- создаем два списка и заполняем их

```
std::list<int> list1 {1, 2, 3, 4, 5, 6, 7, 8, 9};
std::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();
std::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 ellements 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 сортированных 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

```
std::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 вызов `++it` после `erase(it)` неопределен

# STL: string

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

1. C-стринг: `char*` и `const char*`
2. класс `std::string` и подобные: `wstring`, `u16string`,`u32string` C++11
3. класс `std::string_view`: дает «ссылку» на существующий текст без возможности его модифицировать, только для чтения 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 std::cout, std::endl; // перечисление через запятую C++17
```

**пример**

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

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

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

// String by repeating one character  1st arg - number characters,
std::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";
std::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
std::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
std::string s6(s5,6,4);                // 3d arg - number of characters
cout << "s6= " << s6 << endl;          // s6= line

// copy substring using iterators         1st arg - start iterator
std::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**

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

**Raw string literals**

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

| | |
|---|---|
| `size(), length()` | размер, длина стринга |
| `empty()` | `true` для пустого стринга |
| `capacity()` | размер зарезервированной памяти |
| `reserve(num)` | запрос на резервирование памяти |
| `resize(...)` | удаляет или добавляет символы в конец |
| `clear()` | удаляет все элементы |
| `shrink_to_fit()` | уменьшает `capacity` до `size()` |

## Доступ к отдельным символам текста

| | |
|---|---|
| `[i], at(i)` | доступ к *i*-у символу |
| `front(), back()` | первый и последний символы |

## Лексикографическое сравнение

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

## Модификация стринга

| | |
|---|---|
| `+` | соединение, конкатенация |
| `+=,append(),push_back()` | добавление в конец |
| `insert()` | вставка символов в середину |
| `erase(),pop_back()` | удаление символов |
| `replace()` | замена части стринга |

## Полезные функции

| | |
|---|---|
| `c_str(),data()` | возвращает C-string |
| `substr(pos,len)` | возвращает часть стринга |
| `find(str)` | поиск в стринге |
| `», «, getline()` | 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`

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

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

# Преобразования стринг $\leftrightarrow$ число

## функции преобразования в C++11

| | |
|---|---|
| `stoi(),stol(),stoll()` | к знаковому целому |
| `stoul(),stoull` | к без-знаковому целому |
| `stof(),stod(),stold()` | к числу с плавающей точкой |
| `to_string(),to_wstring()` | преобразует `Int` или `Float` к стрингу |

- `int` $\rightarrow$ `string` **и преобразует как** `%d` **в** `printf`

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

- `double` $\rightarrow$ `string` **и преобразует как** `%f` **в** `printf`

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

# 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**

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

- **«чтение» string → int**

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

# STL string_view (C++17)

## Основное

- `std::string_view` это пара `{begin(),size()}` «ссылающаяся» на непрерывную последовательность символов которая хранится где-то еще: например в C-стринге `(char [])` или в `std::string`
  - ☞ если объект с текстом разрушается то `string_view` становится недействительным
- в основном предназначен для передачи в функцию read-only текста
- неявное преобразование `std::string_view` $\rightarrow$ `std::string` запрещено
- `std::string_view` не обязан быть нуль-терминированным
- `std::string_view` можно использовать в `constexpr`

```
#include <string_view>  // C++17
```

## инициализация

```
constexpr std::string_view sv {"Hello"}; // <- C-string Hello
std::string s = "World";
std::string_view sv2 {s};       // <- std::string World
std::string_view sv3 {sv};      // <- string_view Hello
std::string_view sv4 {&s[2],2}; // <- pair: begin(),size() rl
```

## string_view как параметр функции

```
auto First = [](std::string_view s1, std::string_view s2) {
   return ( s1 < s2) ? s1 : s2;
};
cout << First(sv2,sv3) << endl;  // string_view, string_view : Hello
cout << First(sv,"ABC") << endl; // string_view, const char* : ABC
cout << First(s,sv) << endl;     // string, string_view : Hello
```

## преобразование string_view → string

```
std::string s1 = sv; // ERROR: no implicit conversion
std::string s1 {sv}; // OK: Hello
```

# Регулярные выражения, regex (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 std::string& str) {
   std::regex dig_re("[0-9]");  // шаблон для любой цифры
   return std::regex_replace(str, dig_re, ""); // замена шаблона на ""
};
std::string test="D5_Kp6_ct";
std::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 std::string& str)-> std::string {
   // шаблон для цифры взятой один или более раз
   static std::regex dig_re("[0-9]+");
   // замена с подстановкой: $& - то что найденно по шаблону
   return std::regex_replace(str, dig_re, "[$&]");
};

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

☞ регулярные выражения не работают со `string_view`

# Дополнительные слайды

# Вектор указателей

```
std::vector<T*> vp;
vp.push_back( new T('A') );
vp.push_back( new T('B') );
vp.push_back( new T('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
```

## функции быстрого преобразования: from_chars() и to_chars() (C++17)

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

### std::from_chars() string → int

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

### std::from_chars() string → double

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

## to_chars() int → string

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

## to_chars() double → string

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

☞ преобразование с double поддерживается не всеми компиляторами