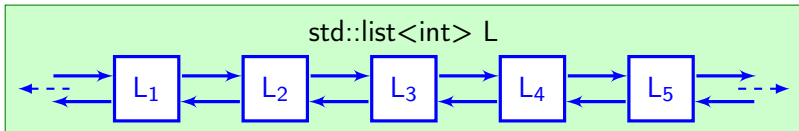


STL: list<>

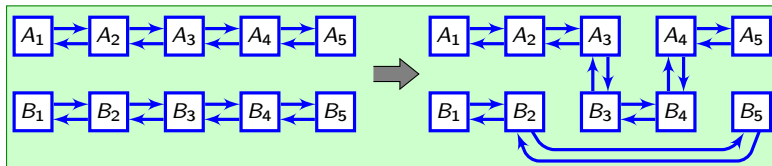
Список (`list`) – *последовательный контейнер*, но он имеет внутреннее устройство совершенно другого типа чем вектор



- 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, 5,5,5 };  
list1.unique();           // remove consecutive duplicates  
cout << " list1: " << list2 << endl; // list1: 1 2 3 4 5
```

● `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

STL: string

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

- 1 С-стринг: `char*` и `const char*`
- 2 классы: `std::string` и `std::wstring` (а так же `u16string` и `u32string` в C++11)
- 3 `std::string_view` – неизменяемая (read-only) строка C++17

Мотивация введения string

- 👉 Класс поведение которого подобно поведению «встроенных» типов:
- ✓ присваивание с помощью `=`
 - ✓ сравнение с помощью `==`, `<`, `>` ...
 - ✓ слияние с помощью `+`, `+=`
 - ✓ выделение части текста, поиск, замена и другое

STL: string. Элементарные операции.

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

● создание и копирование

```
string s0;                                // with no arguments: empty string
cout << "s0 is: " << s0 << endl; // s0 is:

string s1 ("Initial string");             // one argument: text in quotes
cout << "s1 is: " << s1 << endl; // s1 is: Initial string

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

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

● создание string из уже имеющегося текста

```
const char* line = "short line for testing";
string s4(line);                // full copy: ctor
cout << "s4 is: " << s4 << endl; // s4 is: 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 is: " << s5 << endl; // s5 is: short line

// take only the first characters: 1st arg.- stl-string
                                // 2nd arg.- start position,
string s6 (s5,6,4);            // 3d arg.- number of characters
cout << "s6 is: " << s6 << endl; // s6 is: line

// take only the first characters: 1st arg: start iterator
string s7 (s4.begin(),s4.end()-5); // 2nd arg: end iterator
cout << "s7 is: " << s7 << endl;  // s7 is: short line for te
```

Функции для работы с STL string

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>	(C++11)

Доступ к отдельным символам текста

<code>[i], at(i)</code>	доступ к i -му символу	
<code>front(), back()</code>	первый и последний символы	(C++11)

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>	удаляет символы (pop_back C++11)
<code>replace()</code>	замена части строки
<code>»,«, getline()</code>	операции ввода/вывода в поток

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

● функции `find()`, `substr()`

```
string tst("K+ pi- K- pi+ ");
auto p = tst.find(string("pi"));
if ( p != string::npos ) {           // string::npos - не найдено
    string sub = tst.substr(p,3);    // 3 символа начиная с p
    cout << " find: " << sub << endl; // find: pi-
}
```

📖 `static const size_type npos = -1;` – это специальное значение, используется как индикатор «неуспеха»: не найдена позиция символа, или ошибки в функции

● функции `c_str()`: преобразование к C-string

```
string sts("Hello wold!\n");
printf("%s",sts.c_str()); // Hello wold!
```

📖 вместо `c_str()` можно использовать функцию `data()` (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 числа (<code>int/float</code>) к строке

● `int` \rightarrow `string`

```
int i = 12345;
string si = "\"" + to_string(i) + "\"";
cout << " converting " << i << " with to_string(): " << si << endl;
// converting 12345 with to_string(): "12345"

cout << to_string(M_PI) << endl; // 3.141593 - 7 значащих цифр
```

Raw string literals

(C++11)

👉 В C++11 появился удобный способ задания текста содержащего специальные символы: обратную косую черту `'\'`, переход на новую строку `\n` и др.

старый способ C++98

```
string test1="C:\\A\\B\\file.txt";  
string test2="First\nSecond\nThird";  
cout << test1 << endl << test2 << endl;
```

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

НОВЫЙ в C++11

```
string rt1=R"(C:\A\B\file.txt)";  
string rt2=R"(First\nSecond\nThird)";  
string rt3=R"(First  
Second  
Third)";  
cout<<rt1<<endl<<rt2<<endl<<rt3<<endl;
```

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

STL: ввод/вывод в string

string stream: `#include <sstream>`

👉 позволяет использовать `string` в качестве потока и затем использовать имеющиеся функции ввода-вывода

● «запись» `Rational` → `string`

```
stringstream ss;                // поток-строинга
ss << Rational(13,17);          // запись
string mystr = ss.str();        // итоговый стринг
cout << " mystr: " << mystr << endl; // mystr: 13/17
```

● «чтение» `string` → `Rational`

```
stringstream tt(mystr);         // поток на основе стринга
Rational myr; tt >> myr;        // чтение из потока
cout << " myr= " << myr << endl; // myr= 13/17
```

Регулярное выражение: «шаблон» для поиска в тексте

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

● удаление всех цифр из текста

```
#include <regex> // regular expression
auto RmDigits = [](const string& str)-> string {
    regex dig_re("[0-9]"); // любая цифра
    return regex_replace(str, dig_re, ""); // замена на пустой стринг
};
string test="D5_Kp6_ct";
string out = RmDigits(test);
cout << test << " -> " << out << endl; // D5_Kp6_ct -> D_Kp_ct
```

● взятие чисел в квадратные скобки

```
#include <regex> // regular expression
auto BrDig = [](const string& str)-> string {
    // шаблон для цифры взятой один или более раз
    static regex dig_re("[[:digit:]]+");
    // замена с подстановкой: $& - то что найдено по шаблону
    return regex_replace(str, dig_re, "$&");
};
string test2="a1 a2 a3 a44";
out = BrDig(test2);
cout << test2 << " -> " << out << endl;
a1 a2 a3 a44 -> a[1] a[2] a[3] a[44]
```