



Объектно-ориентированное программирование

Петрусович Денис Андреевич

Доцент кафедры Проблем управления

petrusevich@mirea.ru



Тема 2. Структура STL

Контейнеры

Адаптеры

Итераторы

Алгоритмы

Функторы (и лямбда-выражения)



Тема 2. Контейнеры в STL

STL содержит классы, позволяющие хранить данные

Т.к. логика работы не зависит от природы хранимых объектов, это шаблоны классов

Для подключения контейнера `container` нужно включить библиотечный файл в код:

```
#include<container>
```



Контейнеры в STL

- Последовательные контейнеры (порядок добавленных элементов сохранен): `vector`, `array`, `list`, `deque`,...
- Ассоциативные контейнеры: хранятся пары «ключ-значение». Ключом может быть объект «сложной природы» (в отличие от индекса в массиве)



Класс `vector<>`

- Массив – непрерывный участок памяти, состоящий из одинаковых по структуре элементов
- `vector< >` - замена массива
- Есть эффективная индексация `[i]`
- Автоматизирована работа с памятью




Класс `vector<>`

10	1	-3	6	5	7	...		
----	---	----	---	---	---	-----	--	--



Класс `vector<>`

10	1	-3	6	5	7	...		
----	---	----	---	---	---	-----	--	--



```
vector<int> v;
```

```
//заполнение
```

```
cout<<v[5];
```



Класс `array<>`

Обладает схожими свойствами с классом `vector<>`

Менее гибкая работа с памятью

```
#include <array>
#include <iostream>
int main()
{
    array<int, 4> c0 = { 0, 1, 2, 3 };...
}
```




Работа с массивом

Поиск в неотсортированном массиве —
линейный поиск

Поиск элемента *elem_to_find* в векторе *vec*

```
for (i=0; i<vec.size(); i++)  
    if (vec[i]==elem_to_find)
```

...



Работа с массивом

Поиск в отсортированном массиве – бинарный поиск. Поиск элемента «3»

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----

1:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----



2:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----



3:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----



Размер рассматриваемой части массива
уменьшается в два раза на каждой итерации



Работа с массивом

Бинарный поиск. Поиск элемента «12»

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----

1:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----

2:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----

3:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----

4:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----



Адаптеры

Основа – существующие контейнеры

Сокращение операций так, чтобы получить интерфейс соответствующих структур данных

Стек stack

Очередь queue

Очередь с приоритетами (куча, heap) priority_queue

```
#include <queue>
priority_queue<int> q;
q.push(...);
int t = q.top();
q.pop();
```



Работа с массивом

Индексация [i] $O(1)$

Поиск в

отсортированном массиве $O(\log n)$

Поиск в

неотсортированном массиве $O(n)$

Вставка/удаление $O(n)$

n — число элементов в массиве



Контейнер deque< >

Аналог класса `vector< >`, «обертка» для массива

Добавлять/удалять элементы можно не только с конца (`push_back()`, `pop_back()`), но и в начало (`push_front()`, `pop_front()`)



СВЯЗНЫЕ СПИСКИ

Участок памяти не обязан быть непрерывным



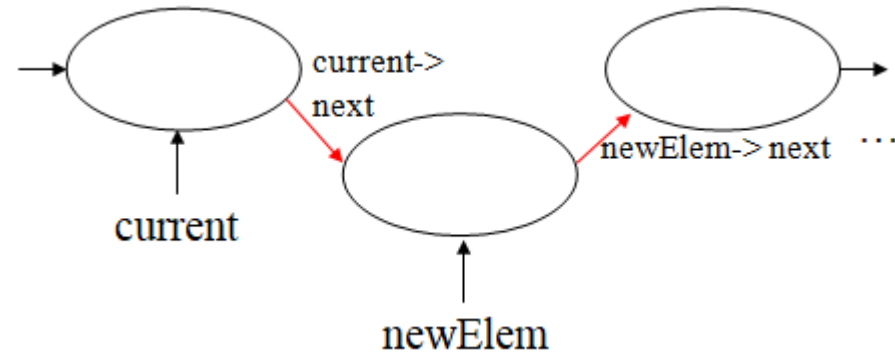
Односвязный список: в каждом элементе есть указатель на следующий next

Двусвязный список: вдобавок, есть указатель на предыдущий элемент previous



СВЯЗНЫЕ СПИСКИ

В такой структуре удобно вставлять/удалять элементы:



Вставка/удаление $O(1)$

Индексация $O(n)$

Линейный поиск $O(n)$

Бинарный поиск хуже линейного



Класс `list< >`

Присутствуют операции добавления/удаления с начала и конца списка:

`push_back()`, `push_front()`, `pop_back()`, `pop_front()`

Нет операции индексации `[i]`

Для перечисления элементов в списке применяются итераторы

Функция `erase()` используется для удаления элемента, адресуемого итератором



Ассоциативные контейнеры

Содержат пары «ключ - значение»

При запросе по ключу нужно осуществить поиск элемента в контейнере или определить место в памяти, где элемент должен быть в контейнере

По умолчанию элементы сортируются при добавлении (требуется операция <)

Например, контейнер с оценками студентов:

```
...marks;
```

```
marks["Petrov"] = 5;
```

```
marks["Sidorov"] = 4;
```

```
cout<<marks["Sidorov"];
```



Ассоциативные контейнеры

Основные виды контейнеров:

- `set` (только уникальные элементы);
- `multiset` (возможны повторяющиеся элементы);
- `map` (`set`, который содержит элементы «ключ-значение», ключ уникален);
- `multimap` (возможно дублирование ключей)



Ассоциативный список `map`

В основе сбалансированное дерево с упорядочиванием по ключу

Поиск по ключу / вставка / удаление элемента «стоит» $O(\log n)$

Можно перечислить элементы по возрастанию ключей

```
map<string, int> marks;  
marks.insert(std::pair<string,  
    int>("peter", 4));  
...  
marks["peter"] = 3;  
cout<<marks["peter"];
```



Ассоциативный список `map`

```
map<string, int>::iterator it_f =
    marks.find("peter");
cout<<"\nFound: " << (*it_f).first <<
    ": " << (*it_f).second << endl;

map<string, int>::iterator it =
    marks.begin();
for (; it != marks.end(); it++)
{
    cout << (*it).first << ": " <<
        (*it).second << endl;
}
```



Множество set

Позволяет перечислить все уникальные элементы,
добавленные в множество

Основано на одной из разновидностей
сбалансированного дерева

В отличие от map работа ведётся с ключами, а не с
парами ключ – значение

Быстрая проверка: есть ли элемент в множестве?
 $O(\log n)$

Поиск, вставка, удаление – $O(\log n)$



Основа set/map

Реализация может быть любой, но операции должны быть по производительности не хуже некоторых границ

Поиск/вставка элемента в set/map работает за $O(\log n)$: реализация основана на сбалансированных деревьях (АВЛ-деревья, красно-черные деревья, декартовы деревья и т.п.)

Задачу можно решить за $O(1)$ с помощью хеширования



Хеширование

Задачу можно решить за $O(1)$ с помощью хеширования: классы `unordered_map`, `unordered_set`

Хеш-функция: отображение ключа на пространство адресов

Коллизия: два ключа дают один адрес

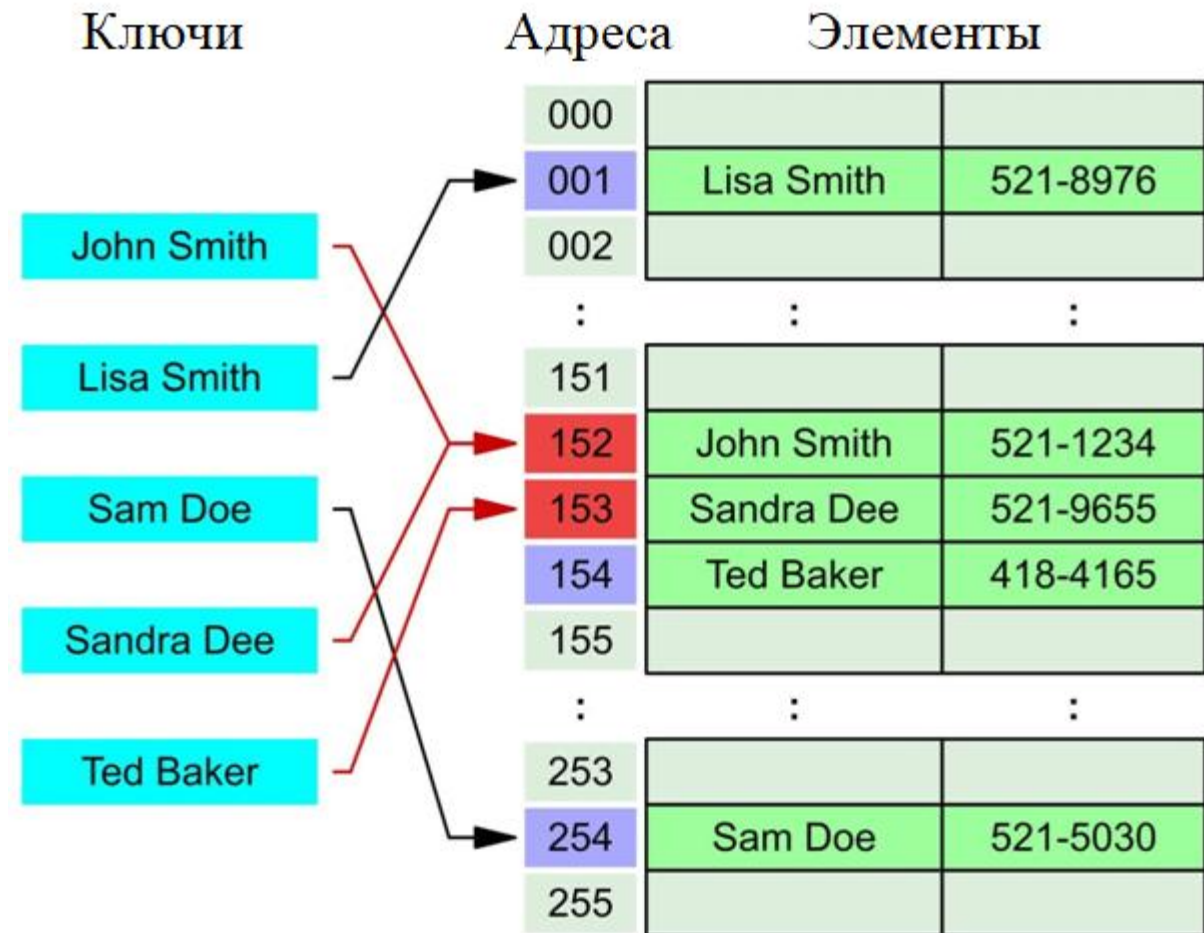
За производительность «расплачиваемся» доп.расходами на память

Хранимые объекты нельзя перечислить с помощью итератора



Хеширование

Первый подход (открытая адресация): хранение записи по адресу, определенному по хэш-функции

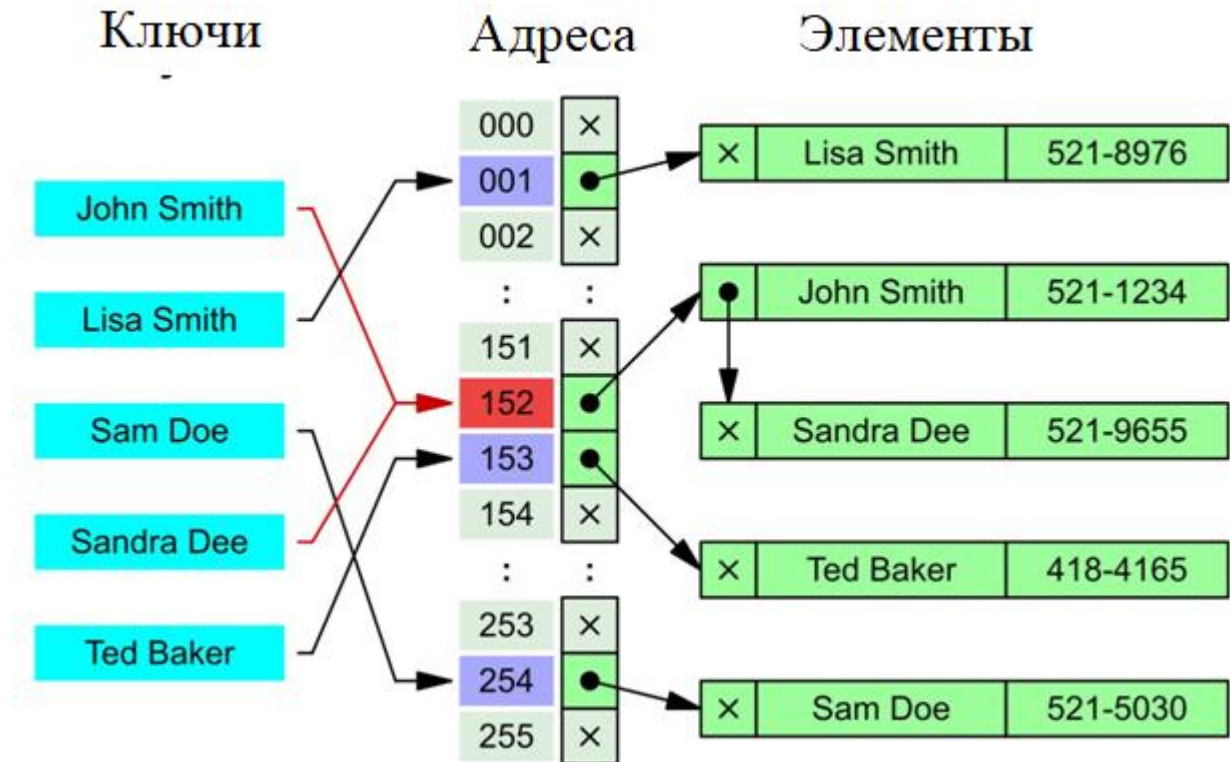




Хеширование

Второй подход (метод цепочек): по полученному адресу хранится ссылка на элемент

Коллизии: двум ключам соответствует один адрес





Итераторы

Средство для перечисления элементов в контейнере

Функции `begin()`, `end()` дают начало и конец контейнера (меньший и больший элемент в случае ассоциативного контейнера)

Итератор p перемещается по списку от начала к концу

```
list<char> lst;...
list<char>::iterator p;
p = lst.begin();
while (p!=lst.end())
{
    cout<<*p<<" ";
    p++;
}
```



Алгоритмы

Часто используемые функции, работающие с разными контейнерами

Поиск; сортировка; подсчёт количества элементов, удовлетворяющих условию; применение ко всем элементам контейнера функции; ...

Подключить библиотеку `#include<algorithm>`

Подсчёт количества букв 'a' или гласных в списке *lst*

```
int n1 = count(lst.begin(), lst.end(),  
              'a');
```

```
int n2 = count_if(lst.begin(),  
                  lst.end(), glas);
```



Алгоритмы

Полезные алгоритмы:

find, find_if

copy

sort

for_each

merge

count, count_if

swap

transform

...



Алгоритмы

Алгоритм `sort()` может работать с различными последовательными контейнерами

Объекты в контейнере должны иметь операции сравнения

```
vector<int> vec;
```

```
...
```

```
sort(vec.begin(), vec.end());
```



Алгоритмы

Алгоритмы `find`/`find_if` ищут определенный элемент или элемент, для которого истинен предикат

```
vector<int> v; ...  
if (find(v.begin(), v.end(), 25) !=  
v.end())  
{  
    // число 25 найдено  
}  
else  
{  
    // число 25 не найдено  
}
```



Алгоритмы

Алгоритм `for_each` обходит элементы контейнера

```
void show(int i) {...}
```

```
void update(int &i) {...}
```

```
int main()
```

```
{
```

```
    int M[]={1,2,3,4,5};
```

```
    int len = 5;
```

```
    for_each(M,M+len, show) ;
```

```
    for_each(M,M+len, update) ;
```

```
    return 0;
```

```
}
```




Алгоритмы

Алгоритм `for_each` обходит элементы контейнера

```
void show(int i) {...}
```

```
int main()
```

```
{
```

```
    vector <int> v; ...
```

```
    for_each(v.begin(), v.end(), show);
```

```
    return 0;
```

```
}
```



Литература

Шилдт. Самоучитель C++. Глава 14.

В любом учебнике по C++ есть глава, посвященная STL

Примеры работы итераторов:

<https://habr.com/ru/post/122283/>

<https://cpp.com.ru/stl/5.html>

<http://www.realcoding.net/articles/iteratory.html>

<http://www.realcoding.net/articles/iteratory-biblioteki-stl.html>

https://www.osp.ru/pcworld/1998/06/159178#part_2

<https://ci-plus-plus-snachala.ru/?p=298>

<https://purecodecpp.com/archives/3717>



Спасибо за внимание!

Петрусович Денис Андреевич

Доцент Кафедры Проблем управления

petrusevich@mirea.ru