Алгоритмы и структуры данных. «Стандартные средства» языков и библиотек программирования

Примеры стандартных средств

- стандартная библиотека C
- «стандартные библиотеки» C++
- стандартные средства C#
- стандартные средства Common Lisp

Стандартная библиотека C

(примеры)

- qsort
- bsearch
- strstr

qsort

```
void qsort(
   void *base,
   size_t num,
   size_t width,
   int (__cdecl *compare ) (
         const void *,
         const void *)
);
```

bsearch

```
void *bsearch(
    const void *key,
    const void *base,
    size_t num,
    size_t width,
    int ( __cdecl *compare ) (
        const void *,
        const void *)
);
```

strstr

```
char *strstr(
    const char *str,
    const char *strSearch
);
```

Библиотека стандартных шаблонов

Standard Template Library - (STL)

Или «стандартная библиотека шаблонов»

- STL входит в стандарт языка
- STL содержит множество шаблонов функций и классов для хранения, поиска и выполнения других алгоритмов на структурах данных
- При использовании STL реализовывать свои структуры имеет смысл в том случае, если их нет в STL или их свойства не соответствуют требованиям
- STL включает
  - контейнерные классы
  - итераторы
  - алгоритмы
  - функциональные объекты

Шаблоны

Иногда – «параметризованные функции/классы»:

специальные синтаксические конструкции языка C++, предназначенные для кодирования обобщённых классов и алгоритмов, позволяющие абстрагироваться от некоторых конкретных параметров (свойств) реализуемых моделей

Примеры шаблонов. Функции

```
template <class T>
void PrintTwice(T data) {
    cout << "Twice: " << data * 2 << endl;
}
```

```
PrintTwice(2);    //void … (int data) {…}
PrintTwice(M_PI);//void … (double data) {…}
```

Примеры шаблонов. Классы

```
template <class T>
class Item {
    T Data;
public:
    Item() : Data( T() )  {}
    void Set(T src) {Data = src;}
    T GetData() const {return Data;}
    void PrintData() {cout << Data;}
};
```

```
Item<int> item1;
item1.SetData(2);
item1.PrintData();
```

```
Item<double> item2;
item2.SetData(2.0);
item1.PrintData();
```

template in template

```
template <class T> class Out {
template <class X> struct In {X x1; X x2};
public:
T x1;
In<T> p;
};

int main() {
 Out<int> test;
 test.p.x1 = 1;
 …
```

template in template

```
#include <iostream>
#include <vector>
#include <deque>
#include <list>
template<typename T, template<class, class...> class C, class... Args>
std::ostream& operator <<(std::ostream& os, const C<T, Args...>& objs) {
    os << __PRETTY_FUNCTION__ << '\n';
    for (auto const& obj : objs)
        os << obj << ' ';
    return os;
}
int main() {
    std::vector<float> vf { 1.1, 2.2, 3.3, 4.4 };
    std::cout << vf << '\n';

    std::list<char> lc { 'a', 'b', 'c', 'd' };
    std::cout << lc << '\n';

    std::deque<int> di { 1, 2, 3, 4 };
    std::cout << di << '\n';

    return 0; //https://www.onlinegdb.com/online_c++_compiler
}
```

Контейнеры

- Последовательные контейнеры (кроме обычных массивов)
  - вектор (vector)
  - список (list)
  - двунаправленная очередь (deque)
- Ассоциативные контейнеры
  - множество (set) и мультимножество (multiset)
  - словарь (map) и multimap (мультисловарь)

+ не упорядоченные варианты реализации с версии C++11

- Адаптеры
  - стек (stack)
  - очередь (queue)
  - очередь с приоритетами (priority_queue)
- Специальные контейнеры
  - битовое множество (bitset)
  - массив значений (valarray)
  - строка (basic_string)

Обычные массивы

- Имеют фиксированный размер (после создания)
- Быстрый доступ к произвольному элементу с использованием операции индексирования []
- Вставка и/или удаление элементов в/из середины осуществляется медленно
- Контроль границ массивов не предусмотрен

STL: vector

- Динамический массив
  - `#include <vector>`
  - `vector<int> vec;`
- Эффективный доступ к элементам
  - `operator [], at, front, back`
- «Медленные» вставка/удаление в середине
  - `insert, erase`
- «Быстрые» вставка/удаление в конце
  - `push_back, pop_back`
- Дополнительные ~~опции~~ операции
  - `size, empty, clear, …`

STL: deque

- Аналогичен вектору с быстрым доступом как в начале, так и в конце контейнера
- Динамический массив
  - `#include <deque>`
  - `deque<string> dq;`
- Эффективный доступ к элементам
  - `operator [], at, front, back`
- «Медленные» вставка/удаление в середине
  - `insert, erase`
- «Быстрые» вставка/удаление с двух сторон
  - `push_front, pop_front, push_back, pop_back`
- Другие операции
  - `size, empty, clear, …`

STL: list

- Двусвязный список
  - `#include <list>`
  - `list<float> ls;`
- Быстрые вставка/удаление в произвольной позиции
  - `insert, erase, push_front, pop_front, push_back, pop_back`
- В том числе в начало и конец
  - `front, back`
- Медленный доступ к произвольной позиции
  - нет операции [], перебор с использованием итератора
- Дополнительные операции
  - `size, empty, clear`
  - `reverse, sort, unique, merge, splice, …`

STL: set

- Хранит множество значений («ключей»)
- Значения уникальны
- В основе сбалансированное бинарное дерево поиска
  - `#include <set>`
  - `set<string> s;`
- Быстрые вставка/удаление (O - ?)
  - `insert, erase`
- Быстрый поиск
  - `find`
- Дополнительные операции
  - `size, empty, clear, …`

STL: multiset

- Хранит множество значений («ключей»)
- Значения **не обязательно** уникальны
- В основе сбалансированное бинарное дерево поиска
  - `#include <set>`
  - `multiset<string> ms;`
- Быстрые вставка/удаление
  - `insert, erase`
- Быстрый поиск
  - `find`
- Дополнительные операции
  - `size, empty, clear, …`

STL: map

- Хранит множество пар {ключ, значение} pairs
- Каждый ключ связан ровно с одним значением
- В основе сбалансированное дерево
  - `#include <map>`
  - `map<string, int> m;`
- Быстрые добавление/удаление
  - `m["fred"] = 99;`
  - `insert, erase`
- Быстрый поиск
  - `int x = m["fred"];`
  - `find`
- Дополнительные операции
  - `size, empty, clear, …`

STL: multimap

- Хранит множество пар {ключ, значение} pairs
- Каждый ключ необязательно связан ровно с одним значением
- В основе сбалансированное дерево
  - `#include <map>`
  - `multimap<string, int> mm;`
- Быстрые добавление/удаление
  - `insert, erase`
- Быстрый поиск
  - `find`
- Дополнительные операции
  - `size, empty, clear, …`

Ассоциативные контейнеры: упорядоченность

- Ассоциативные контейнеры STL как правило реализованы с использованием сбалансированных бинарных деревьев поиска
  - Классы (типы данных), должны реализовывать операцию сравнения

    ```
    bool operator <(T other) const
    ```

  - Если в классе это не предусмотрено, то можно передавать в конструктор «класс-компаратор», т.е. класс в котором перегружена операция (), которая умеет сравнивать целевые объекты

    ```
    bool operator()(T a, T b)
    ```

Ассоциативные контейнеры: пример «класса-компаратора»

```
set<Student *> students;
/* Сравниваются указатели! */

class StudentComparator {
public:
   bool operator() (const Student * a, const
  Student * b) {
      return (a->GetID() < b->GetID());
   }
};

set<Student *, StudentComparator> students;
/*  При построении функции используется
  сравнение на основе класса */
```

STL: stack

- Обеспечивает интерфейс «стек» по отношению к другим контейнерам
  - `#include <stack>`
  - `stack<string> st;//deque по умолчанию`
- Операции
  - `push, pop, top`
  - `size, empty, …`
- Может быть использован с вектором и списком
  - `stack<string, vector<string> > st;`
  - `stack<string, list<string> > st;`

STL: stack

- Обеспечивает интерфейс «стек» по отношению к другим контейнерам
  - `#include <stack>`
  - `stack<string> st;//deque по умолчанию`
- Операции
  - `push, pop, top`
  - `size, empty, …`
- Может быть использован с вектором и списком
  - `stack<string, vector<string> > st;`
  - `stack<string, list<string> > st;`

STL: queue

- Обеспечивает интерфейс «очередь» по отношению к другим контейнерам
  - `#include <queue>`
  - `queue<string> q;//deque по умолчанию`
- Операции
  - `push, pop, top`
  - `size, empty, …`
- Может быть использован на основе списка
  - `queue< string, list<string> > q;`

STL: priority_queue

- Интерфейс очередь с приоритетами
  - `#include <queue>`
  - `priority_queue<int> pq;//по умолчанию вектор`
- Основные операции
  - `push, pop, top`
  - `size, empty, …`
- Можно использовать очередь в качестве основы
  - `priority_queue< int, deque<int> > pq;`

Итераторы

- Необходимо иметь возможность перебирать элементы в контейнере
- Итерации в массивах C++ :

```
const int SIZE = 10;
string names[SIZE];

for (int x = 0; x < SIZE; ++x) {
    cout << names[x] << endl;
}
```

или

```
string * end = names + SIZE;
for (string * cur = names; cur < end; ++cur) {
    cout << *cur << endl;
}
```

Итераторы

- Как осуществляется перебор в контейнерах STL?
- В векторах и двунаправленных очередях можно так (но только в этих контейнерах):

```
vector<string> animals;
animals.push_back("pig");
animals.push_back("cat");
animals.push_back("dog");
animals.push_back("bear");
for (int x=0; x < animals.size(); ++x) {
    cout << animals[x] << endl;
}
```

Итераторы

- Обобщенный подход к перебору элементов в STL основан на итераторах – специальных объектах, похожих на указатели, с помощью которых осуществляется доступ к значениям.
- Все контейнеры предоставляют метод `begin`, которые предоставляет итератор, указывающий на первое значение в контейнере
- Для итераторов перегружены операции, свойственные указателям

++, --	перейти к следующему (предыдущему) элементу
*, ->	доступ к значению, на которое указывает итератор
==, !=	сравнение итераторов на равенство

Итераторы

- Как определить конец контейнера?
- Все контейнеры предоставляют метод `end`, возвращающий специальный итератор, представляющий конец контейнера

```
set<string> animals;
animals.push_back("pig");
animals.push_back("cat");
animals.push_back("dog");
animals.push_back("bear");
set<string>::iterator it;
for (it = names.begin(); it != names.end(); ++it) {
   cout << *it << endl;
}
```

Итераторы

- В какой последовательности осуществляется перебор элементов в контейнерах?
  - для последовательных контейнеров – в естественном порядке их хранения – от первого до последнего
  - для множеств и словарей значения возвращаются в порядке, определяемом обходом сбалансированного бинарного дерева, лежащего в основе контейнера

Итераторы

- Пример обхода

```
set<int> s;
s.insert(10);
s.insert(20);
s.insert(15);
s.insert(12);
set<int>::iterator it = s.begin();
while(it != s.end()) {
  cout << *it << " ";
  it++;
}
system("pause");
```

Итераторы

- Можно также использовать обратные итераторы (reverse iterators) и соответственно методы `rbegin` и `rend` контейнеров

```
set<string> names;

names.insert("sasha");
names.insert("ilma");
names.insert("petr");
names.insert("denis");

set<string>::reverse_iterator rit;
for (rit = names.rbegin(); rit != names.rend(); ++rit) {
   cout << *rit << endl;
}
```

Виды итераторов

- входные
- выходные
- однонаправленные
- двунаправленные
- произвольного доступа

Алгоритмы

- STL предлагает функции, которые применимы к контейнерам – алгоритмы
- Некоторые алгоритмы STL работают только с определенными контейнерами

```
#include <algorithm>
vector<string> animals;
animals.push_back("pig");
animals.push_back("cat");
animals.push_back("dog");
animals.push_back("pig");
unique(animals.begin(), animals.end());
sort(animal.begin(), animals.end());

vector<string>::iterator it = animals.begin()
for (; it != animals.end(); ++it) {
   cout << *it << endl;
}
```

Алгоритмы

```
class StringPrinter {
public:
   void operator()(const string & s) {
      cout << s << endl;
   }
};

vector<string> animals;
animals.push_back("pig");
animals.push_back("cat");
animals.push_back("dog");
animals.push_back("pig");

unique(animals.begin(), animals.end());
sort(animals.begin(), animals.end());

StringPrinter printer;
for_each(animals.begin(), animals.end(), printer);
```

Алгоритмы. Обзор

- binary_search
- copy
- count
- count_if
- fill
- reverse
- for_each
- generate
- find
- set_difference
- set_intersection
- set_symmetric_difference
- sort
- stable_sort
- rotate
- make_heap
- max, min

Требования к хранимым классам

- Хранимые в контейнерах STL классы должны явно определять:
  - конструктор по умолчанию
  - конструктор копирования
  - деструктор
  - операции =, ==, <
- Не все эти операции необходимы всегда, но проще их реализовать, чем заботиться об актуальности при использовании
- Многие ошибки возникают из-за неправильной реализации этих методов

Объекты против указателей на объекты

- Контейнеры STL могут хранить и объекты и указатели на объекты
- vector<Student> или vector<Student *>
- В первом случае память выделяется и освобождается автоматически, во втором – ответственность на программисте, который должен корректно создавать и удалять объекты в необходимые моменты времени

Функторы (функциональные классы)

- Вспомогательные классы, для которых перегружают операцию ()
- В дополнение к приведенным ранее примерам:

```
#include <list>
#include <algorithm>
#include <iostream>

class greater10 {
public: bool operator()(int v) {return v>10; }
};

int main( ) {
list <int> L; list <int>::iterator It;
list <int>::iterator result;
L.push_back( 5 );       L.push_back( 10 );
L.push_back( 15 );      L.push_back( 20 );
L.push_back( 10 );
result = find_if( L.begin( ), L.end( ), greater10() );
cout << *result;
}
```

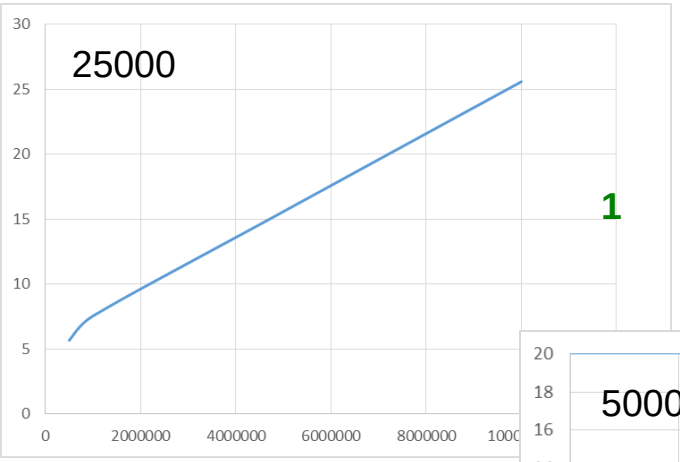
Другие библиотеки C++

- boost
  - многопоточное программирование
  - контейнеры
  - юнит-тестирование
  - структуры данных
  - функциональные объекты
  - обобщённое программирование
  - графы
  - работа с геометрическими данными
  - ввод-вывод
  - межъязыковая поддержка
  - итераторы
  - математические и числовые алгоритмы
  - работа с памятью
  - синтаксический и лексический разбор

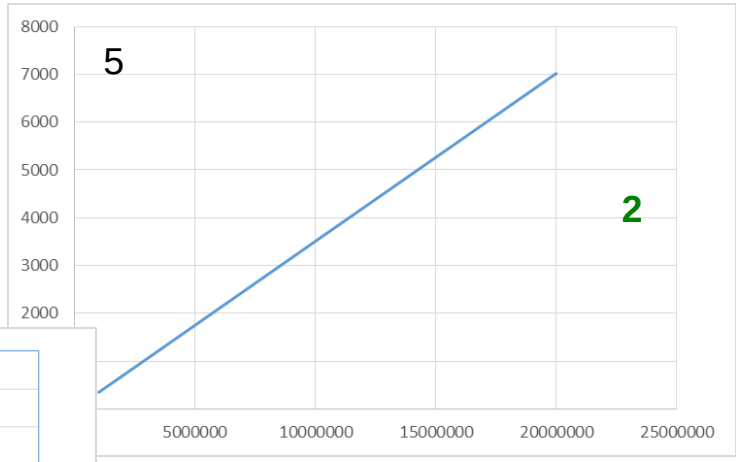
Карта C++ (2012) (Алёна Сагалаева)

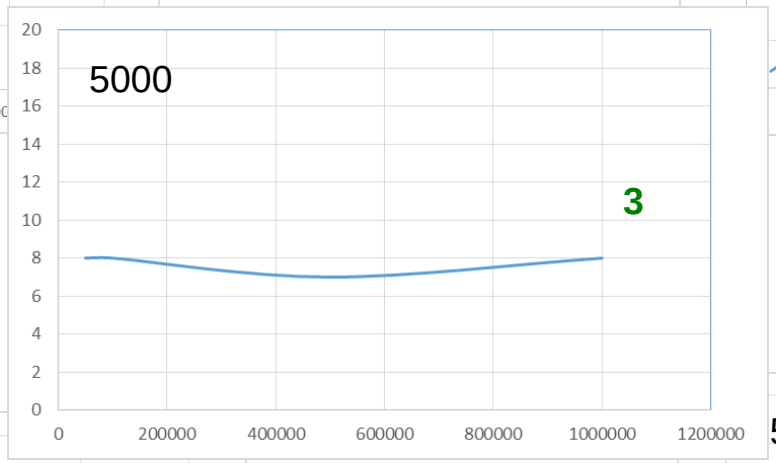
Карта C++ (2017) (Алёна Сагалаева)

Производительность операций в контейнерах



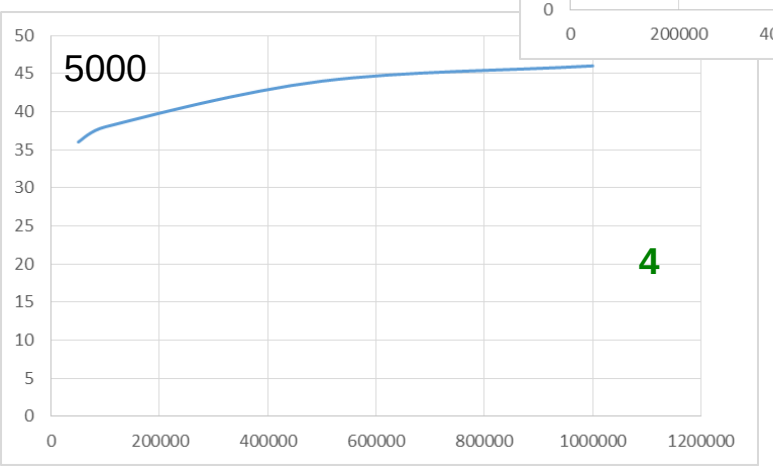
A vector.insert (at begin)





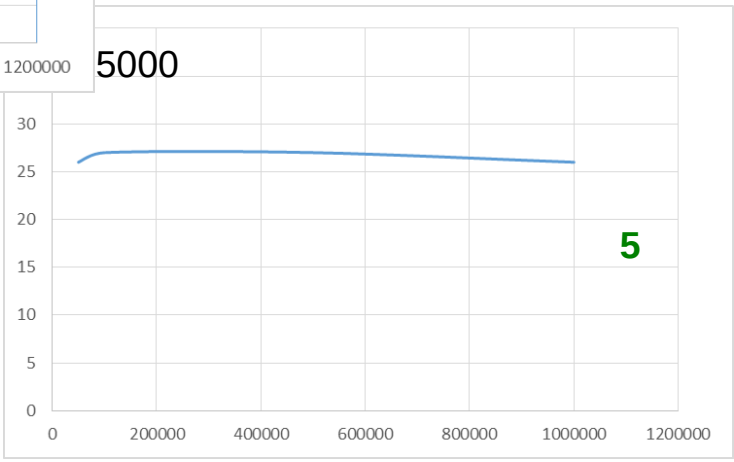
B set.insert

C unordered_set.insert



D list.push_back

E vector.push_back



C#

Класс	Описание
ArrayList	Реализует интерфейс IList с помощью массива с динамическим изменением размера по требованию.
BitArray	Битовый массив
CaseInsensitiveComparer	Проверяет равенство двух объектов без учета регистра строк.
CollectionBase	Базовый класс abstract для строго типизированной коллекции.
Comparer	Проверяет равенство двух объектов с учетом регистра строк.
DictionaryBase	Абстрактный базовый класс для строго типизированной коллекции пар ключ/значение.
Hashtable	Коллекция пар "ключ-значение", которые упорядочены по хэш-
Queue	Коллекция-очередь
ReadOnlyCollectionBase	Базовый абстрактный класс abstract для строго типизированный коллекции с доступом для чтения
SortedList	Коллекция пар "ключ-значение", упорядоченных по ключам. Доступ к парам можно получить по ключу и по индексу.
Stack	Стек
StructuralComparisons	Предоставляет объекты для выполнения структурного сравнения двух объектов коллекции.

C#

```
using System;
using System.Collections;
class Example {
 public static void Main()  {
 Hashtable t = new Hashtable();
 t.Add("txt", "notepad.exe");
 t.Add("bmp", "paint.exe");
 t.Add("dib", "paint.exe");
 t.Add("rtf", "wordpad.exe");
 try {t.Add("txt", "winword.exe"); }
 catch {Console.WriteLine("Key=\"txt\" already exists."); }
 Console.WriteLine("For key=\"rtf\", val={0}.", t["rtf"]);
 t["rtf"] = "winword.exe";
 Console.WriteLine("For key=\"rtf\", val={0}.", t["rtf"]);
 t["doc"] = "winword.exe";
 if (!t.ContainsKey("ht")) {
   t.Add("ht", "hypertrm.exe");
   Console.WriteLine("Value added for key=\"ht\": {0}", t["ht"]);
   }
 foreach(DictionaryEntry de in t) {
   Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
   }
  }
}
```

Common Lisp

- Структуры
  - списки (как же без них!)
  - массивы
  - хеш-таблицы
- Алгоритмы
  - `member`
  - `merge`
  - `remove, remove-if`
  - `count, count-if`
  - `set-difference, union, intersection`
  - `map, mapcar, …`
  - `sort, stable-sort`
  - …

Common Lisp

```
(let ((ls '((0) (1) (2) (0 1 5) (0 1 3) (0 1 5) (0 3 0)
  (0) (1) (2 7 19) (0 0 3 0))))
   (labels ((list< (a b)
    (cond ((null a) (not (null b)))
          ((null b) nil)
          ((=  (first  a)  (first  b))  (list<  (rest  a)
  (rest b)))
          (t (< (first a) (first b)))
     )))
        (sort ls #'list_compare)
    )
)



((0) (0) (0 0 3 0) (0 1 3) (0 1 5) (0 1 5) (0 3 0) (1)
  (1) (2) (2 7 19))
```