ПОСЛЕДОВАТЕЛЬНЫЕ КОНТЕЙНЕРЫ

Использование последовательных контейнеров стандартной библиотеки языка C++

К. Владимиров, Intel, 2017

- Непрерывные контейнеры
- Узловые контейнеры
- Адаптеры
- Полезные классы
- Строки

Виды последовательных контейнеров

- Контейнеры
  - array — массив с фиксированным размером, известным в момент компиляции
  - vector — массив с переменным размером и гарантией непрерывности памяти
  - deque — массив с переменным размером без гарантий по памяти
  - list — двусвязный список
  - forward_list — односвязный список
- Адапторы
  - stack — FIFO контейнер, чаще всего на базе deque
  - queue — LIFO контейнер, чаще всего на базе deque
  - priority_queue — очередь с приоритетами, чаще всего на базе vector

Общая информация о контейнерах

- Общие для всех контейнеров методы
  - empty — проверка пустоты контейнера
  - max_size — максимальный размер контейнера, доступный в данной реализации
  - swap — обмен контейнерных переменных содержимым
  - size (кроме array) — действительный размер контейнера
  - clear (кроме array) — очистка контейнера
  - front — первый элемент (также back для всех кроме forward_list)
  - begin, end, cbegin, cend — получение итераторов (см. далее)
- Требования к элементам контейнеров
  - Копируемость — у элемента должен быть разрешенный конструктор копирования
  - Изменяемость — элемент должен быть lvalue (т.к. все контейнеры неинтрузивные)
  - Конструируемость — требование к наличию конструктора по умолчанию

От ручного выделения к векторам

```
int *n = new int[10];
n[5] = 5;
// тут много кода
// какой сейчас размер у n?
// стереть крайний элемент?
// пуст ли теперь n?
// не забыть delete[]
```

```
vector<int> v(10);
v[5] = 5;
// тут много кода
size_t vsize = v.size();
v.pop_back();
if (v.empty()) { что-то }
// ресурсы будут освобождены
```

Первое представление об итераторах

```
vector<int> v(10);

// pi это указатель
auto pi = &v[0];
pi += 3;
assert (*pi == v[4])

// как узнать, что pi в
конце?
```

```
vector<int> v(10);

// vi это итератор
auto vi = v.begin();
vi += 3;
assert (*vi == v[4]);

if (vi == v.end()) { что-то }
```

Гарантии непрерывности памяти

```
// функция init написана в старом стиле

template <typename T> void init (T* arr, size_t size) {
  // тут используем arr[i] или *(arr + i)
}

// но её можно использовать с векторами

std::vector<T> t(n);
T *start = &t[0];
init_t (start, n);
assert (t[1] == start[1]);
```

Неприятное исключение: vector<bool>

```
std::vector<bool> t(n);

bool *start = &t[0]; // t[0] это vector<bool>::reference

assert (t[1] == start[1]); // oops!
```

Важно запомнить две вещи

- vector<bool> не удовлетворяет соглашениям контейнера vector
- vector<bool> не содержит элементов типа bool

Лучше использовать std::bitset, который официально не является контейнером в смысле STL (см. далее в разделе «полезные классы»)

Особые возможности vector

- Управление памятью
  - reserve — выделение неинициализированной памяти
  - capacity — возвращает размер памяти, зарезервированной под вектор
  - resize — изменение размера вектора (в том числе уменьшение)
  - shrink_to_fit (C++11) — срезка памяти вектора до реально используемой
- Добавление и удаление элементов
  - push_back — вставка в конец вектора, может приводить к реаллокациям
  - pop_back — удаление последнего элемента (не меняет резерв памяти)
- Доступ к элементам
  - operator[] — доступ по индексу без проверки
  - at — доступ по индексу с проверкой

Задача: что неправильно в этом коде?

```
vector<int> v;

for (int i = 0; i != N; ++i)
  v.push_back(i);
```

Ответ: вектор не терпит халатности

```
vector<int> v;
v.reserve(N);

for (int i = 0; i != N; ++i)
  v.push_back(i); // теперь здесь не будет перевыделений
```

- Вставка в конец вектора имеет всего лишь амортизированную константную сложность O(1)+. В этом плюсе кроются все минусы.
- Это означает, что всегда полезно думать о памяти вектора не меньше, чем о памяти динамического массива.

Вставка и удаление элементов

```
int data[8] = {2, 3, 5, 7, 9, 11, 13, 17};
vector<int> v;
v.insert (v.begin(), data, data + 8);
vector<int> v2;
v2.assign (v.begin() + v.size() / 2, v.end());
v.erase (v.begin() + v.size() / 2, v.end());
// сейчас v == {2, 3, 5, 7}; v2 == {9, 11, 13, 17};
```

Задача: как уменьшить capacity в C++98?

```
vector<int> v(10000);

// тут много всякого произошло

v.erase(v.begin() + 100, v.end());

assert (v.size() == 100 && v.capacity() == 10000);

// вектор занимает в памяти больше 30K, используя меньше 1K.
Как реально уменьшить capacity?

Подсказка: resize не работает, от имеет дело с размером
```

Решение: а вот и своп

```
vector<int> v(10000);

// тут много всякого произошло

v.erase(v.begin() + 100, v.end());

assert (v.size() == 100 && v.capacity() == 10000);

vector<int>(v).swap(v);
```

Это довольно сложный и в общем гениально красивый ход. Его надо разобрать на лекции.

Обсуждение

Какие способы инциализации вы бы добавили в вектор?

Пока что были рассмотрены:

- Value-инициализация по размеру через первый параметр конструктора
- Заполнение элементами через push_back
- Создание из встроенного массива или другого вектор через assign или insert

Хватит ли этого?

Списочная инициализация

```
int b[7] = {2, 3, 5, 7, 9, 11, 13};
vector<int> v = // хм... в C++98 тут ничего не напишешь
v.push_back(2);
v.push_back(3);
v.push_back(5); // хватит, я уже устал
```

Списочная инициализация

```
int b[7] = {2, 3, 5, 7, 9, 11, 13};

vector<int> v {2, 3, 5, 7, 9, 11, 13}; // C++11

vector<int> v = {2, 3, 5, 7, 9, 11, 13}; // C++11
```

- Списочная инициализация доступна для всех стандартных контейнеров
- Проблемой могут быть её механизмы

Списочная инициализация

```
class B {
  int a_, b_;
public:
  B (int a, int b) : a_(a), b_(b) {}
};

B b = {1, 2}; // C++11
```

- Эта разновидность называется расширенным синтаксисом
- Она не имеет отношения к списочной инициализации векторов

Расширенный синтаксис

Защищает от неявных преобразований

```
class B {
  int a_;
public:
  B (int a) : a_(a) {} // нет маркировки explicit
};

B b(3.14); // всё хорошо, работает double -> int приведение

B b{3}, c(3); // вызывается один и тот же конструктор

B b{3.14}; // ошибка
```

Два механизма инициализации

- Расширенный синтаксис
- Явный конструктор из списка инициализации

```
class B {
  int a_;
public:
  B (int a) : a_(a) {}
  B (std::initializer_list<int> il);
};

B b(1), c{1}; // теперь они вызывают разные конструкторы
```

Списочная инициализация: вектора

```
// это вектор [14, 14, 14]
vector<int> v1 (3, 14);
// а это вектор [3, 14]
vector<int> v2 {3, 14};
```

Это связано с наличием у вектора **нескольких** конструкторов

- `v(10); // размер 10, инициализация по умолчанию`
- `v(10, 1); // размер 10, инициализировать единицами`
- `v {10, 1}; // размер = размеру списка, инициализация списком`

Списочная инициализация для ваших контейнеров

- Хорошая новость: initializer_list это тоже разновидность последовательного контейнера и его можно обходить итераторами

```
template <typename T>
class Tree {
// тут какая-то специфика дерева
bool add_node (T& data);

public:
Tree(initializer_list<T> il) {
  for (auto ili = il.begin(); ili != il.end(); ++ili)
    add_node(*ili);
};
```

Обсуждение

Список инициализации, как и вектор, непрерывен в памяти. Преимуществом является то, что begin() возвращает просто указатель, а инкремент итератора это инкремент указателя.

Но именно для списка инициализации, нет ли в этом решении каких-то, иногда блокирующих его использование, недостатков?

Подсказка:

```
vector <int> v1 = {1, 2, 3};
// тут много кода
vector <int> v2 = {v1[2], v1[0], v1[1]};
```

Обсуждение

Какие объективные проблемы вы видите в классе vector по сравнению со встроенными массивами?

От встроенных массивов к array

```
int s_array[10]; // на стеке, фиксированный размер
int s_varray[n]; // ошибка если n не константа (VLA запрещены)
int *d_array = new int[n]; // в куче, произвольный размер
vector<int> vec(n); // в куче, произвольный размер
array<int, 10> arr; // на стеке, фиксированный размер
```

Использование array так же эффективно как использование встроенного массива. В то же время vector — плохая замена встроенному массиву, так как требует работы с динамической памятью.

Отличия array от встроенных массивов

- Индекс это часть типа
  - Массивы деградируют к указателям, которые не помнят свой размер
  - Для std::array размер является частью типа

```
void trap (Animal* animals, size_t size);

Animal four_animals[4];

Animal five_animals[5];

trap (four_animals, 4);

trap (five_animals, 5); // Это два вызова одной функции
```

Отличия array от встроенных массивов

- Индекс это часть типа
  - Массивы деградируют к указателям, которые не помнят свой размер
  - Для std::array размер является частью типа

```
template <size_t sz> void trap (array<Animal, sz> animals);

array<Animal, 4> four_animals;

array<Animal, 5> five_animals;

trap (four_animals);

trap (five_animals); // Это две совсем разных функции
```

Отличия array от встроенных массивов

- Инвариантность
  - Встроенные массивы деградируют к указателям, которые **ковариантны:** если A обобщает B, то A* обобщает B*
  - std::array ни к чему не деградируют и поэтому **инвариантны**

```
class Dog : public Animal { тут много собачьей специфики };

void trap (Animal* animals, size_t size);

Dog dogs[5];

trap (dogs, 5); // ok, Dog* is Animal*
```

Отличия array от встроенных массивов

- Инвариантность
  - Встроенные массивы деградируют к указателям, которые **ковариантны:** если A обобщает B, то A* обобщает B*
  - std::array ни к чему не деградируют и поэтому **инвариантны**

```
class Dog : public Animal { тут много собачьей специфики };

template <size_t sz> void trap (array<Animal, sz> animals);

array<Dog, 5> dogs;

trap<5> (dogs); // ошибка, array<Dog> это не array<Animal>
```

Почему контейнеры не ковариантны?

Ответ: простой контрпример

```
vector<Cat*> v1;

vector<Animal*>& v2 = v1; // ok, если контейнеры ковариантны

v2.push_back(new Dog); // приехали
```

Можно поставить обратный вопрос: а почему, собственно, указатели не инвариантны? Предлагается над ним подумать дома.

Подсказка #1: ковариантны только одинарные указатели.

Подсказка #2: для ответа недостаточно логики, понадобится также исторический контекст.

Обсуждение

Вам предлагают обертку для указателя

```
template <typename T> class WrapPtr {
  T *ptr_;
public:
  WrapPtr (T* ptr) : ptr_(ptr) {}
  T* get() { return ptr_; }
};
```

Является ли она ковариантной или инвариантной относительно генерализации?

Обсуждение

Ковариантность указателей не работает когда они участвуют в аргументах функций

```
template <typename T> struct Base {
  virtual Base* foo(Base *ptr);
};

template <typename T> struct Derived {
  Derived* foo(Derived *ptr) override; // fail
};
```

Это полезное или вредное свойство языка?

Подсказка: подумайте о вызове по указателю на базовый класс.

Интересный факт: ковариантные возвращаемые типы поддерживаются!

- Непрерывные контейнеры
  - Узловые контейнеры
- Адаптеры
- Полезные классы
- Строки

Рассмотрите deque вместо vector в качестве своего основного контейнера

- Эффективно растёт в обоих направлениях
- Не требует больших реаллокаций с перемещениями, так как разбит на блоки
- Гораздо меньше фрагментирует кучу

Задача: что неправильно в этом коде?

```
deque<int> v;

for (int i = 0; i != N; ++i)
  v.push_back(i);
```

Ответ: всё хорошо

```
deque<int> v;

for (int i = 0; i != N; ++i)
  v.push_back(i);
```

- Вставка в конец дека имеет всегда честную константную сложность O(1).
- Это означает, что думать о памяти дека вам вообще не нужно.

Деки против векторов

Вектора

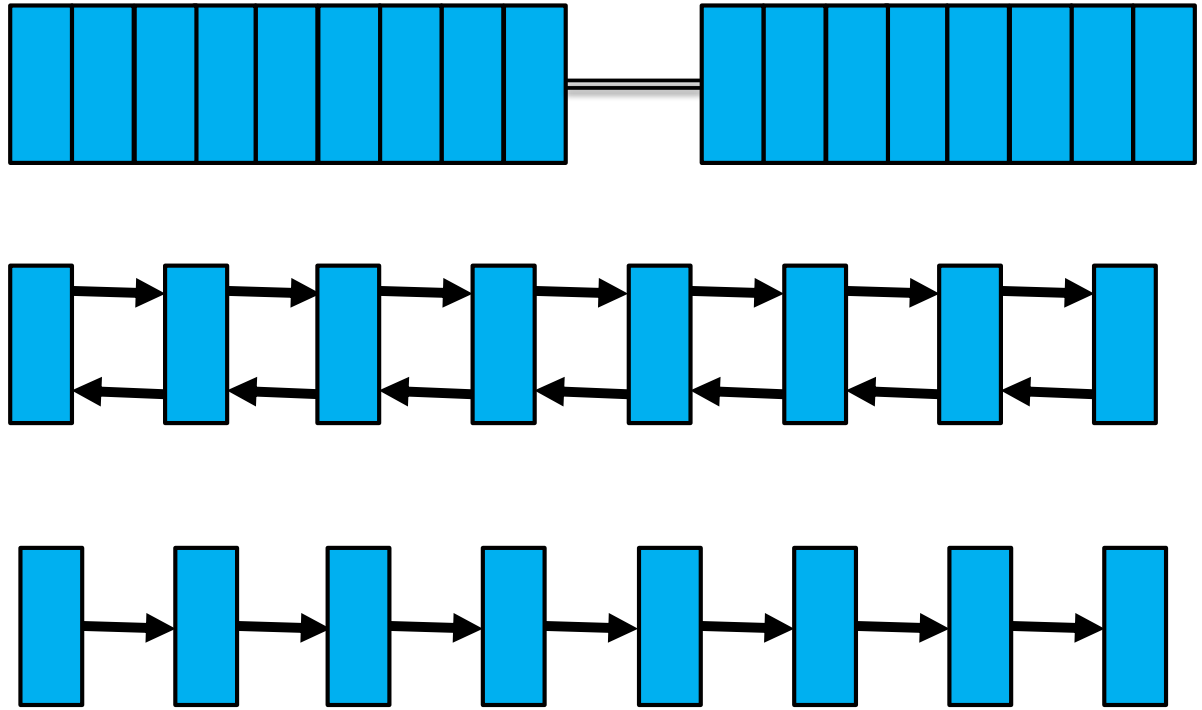
- Доступ к элементу O(1)
- Вставка в конец аморт. O(1)+
- Вставка в начало O(N)
- Вставка в середину O(N)
- Вычисление размера O(1)
- Есть гарантии по памяти
- Есть reserve / capacity

Деки

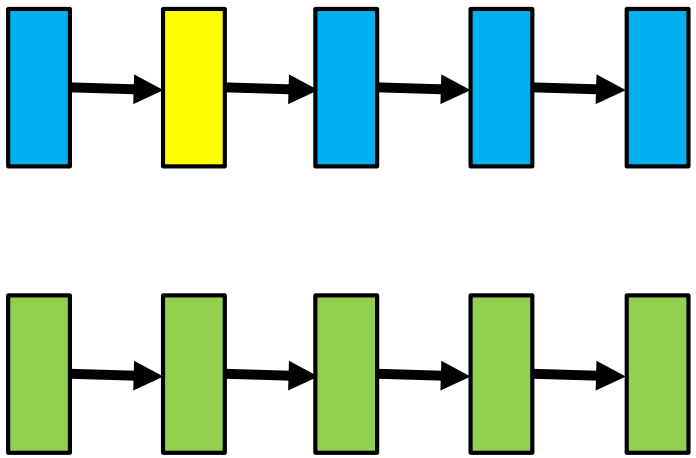
- Доступ к элементу O(1)
- Вставка в конец O(1)
- Вставка в начало O(1)
- Вставка в середину O(N)
- Вычисление размера O(1)
- Нет гарантий по памяти
- Нет необходимости в reserve/capacity

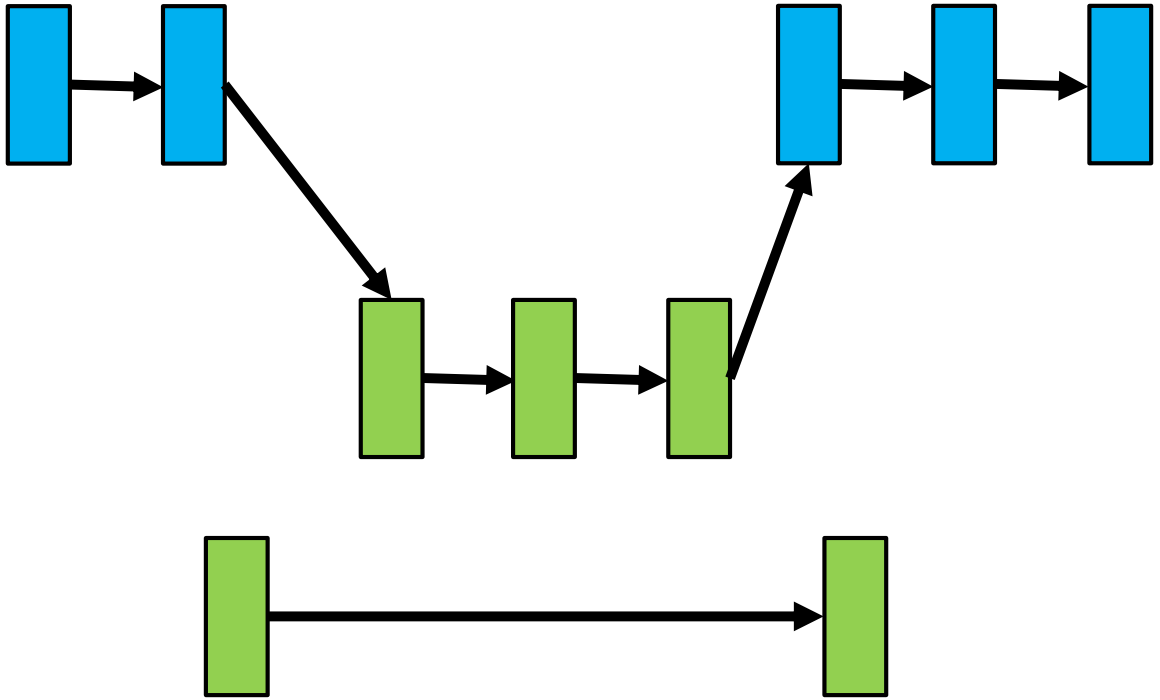
Другие узловые контейнеры

- **deque**
  - контейнер с произвольным доступом
- **list**
  - контейнер с последовательным двусторонним доступом
- **forward_list**
  - контейнер с последовательным односторонним доступом



Особая возможность списков: сплайс

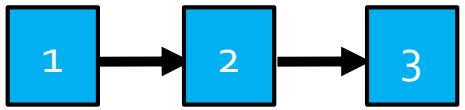


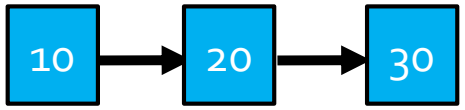


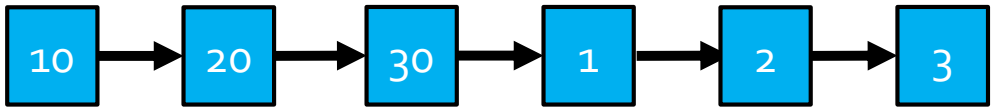
Сплайс для списков

```
forward_list<int> first = { 1, 2, 3 };
forward_list<int> second = { 10, 20, 30 };
auto it = first.begin(); // указывает на 1

// перекидываем весь список second в начало first, it указывает на 1
first.splice_after (first.before_begin (), second);
```



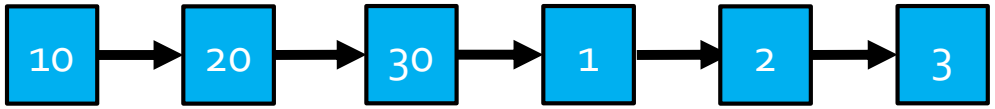


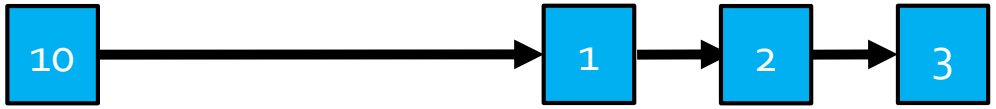


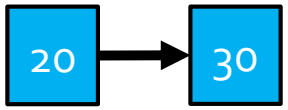
Сплайс для списков

```
// forward_list<int> first = {10, 20, 30, 1, 2, 3 };
// forward_list<int> second = {};
// it указывает на 1

// перекидываем элементы со второго по it в список second
second.splice_after (second.before_begin(), first, first.begin(), it);
```



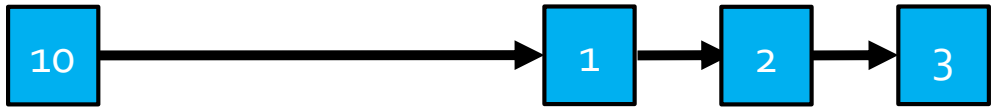


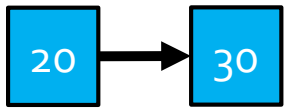


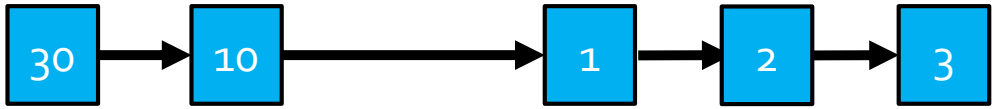
Сплайс для списков: упражнение

```
// forward_list<int> first = { 10, 1, 2, 3 };
// forward_list<int> second = { 20, 30 };
// it указывает на 1

// перекидываем все элементы второго списка начиная со второго в первый
// ???
```



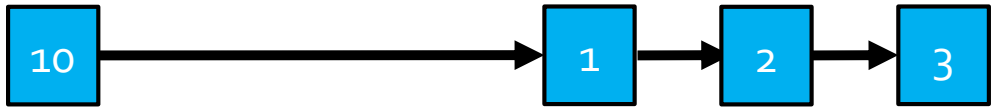


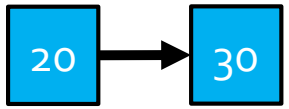


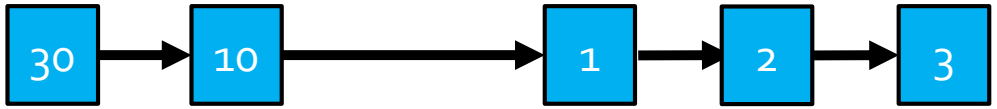
Сплайс для списков: решение

```
// forward_list<int> first = { 10, 1, 2, 3 };
// forward_list<int> second = { 20, 30 };
// it указывает на 1

// перекидываем все элементы второго списка начиная со второго в первый
first.splice_after (first.before_begin(), second, second.begin());
```







Задача: что не так в этом коде?

```
template <typename Container> void foo (Container &c)
{
  if (c.size() == 0)
    {
      // особая обработка
    }
  // обычное тело функции
}
```

Решение: использован не тот метод

```
template <typename Container> void foo (Container &c)
{
  if (c.empty())
    {
      // особая обработка
    }
  // обычное тело функции
}
```

Дело в том, что у списков size имеет сложность O(N) и это связано с возможностью делать splice.

Балансировка size/splice

Две опции:

1. размер списка хранится и обновляется при вставках, но тогда splice должна проверить размер вставляемой последовательность
2. splice работает перевязкой указателей, но тогда размер списка вычисляется

По стандарту выбрана опция (2)

size у списков O(N), splice у списков O(1)

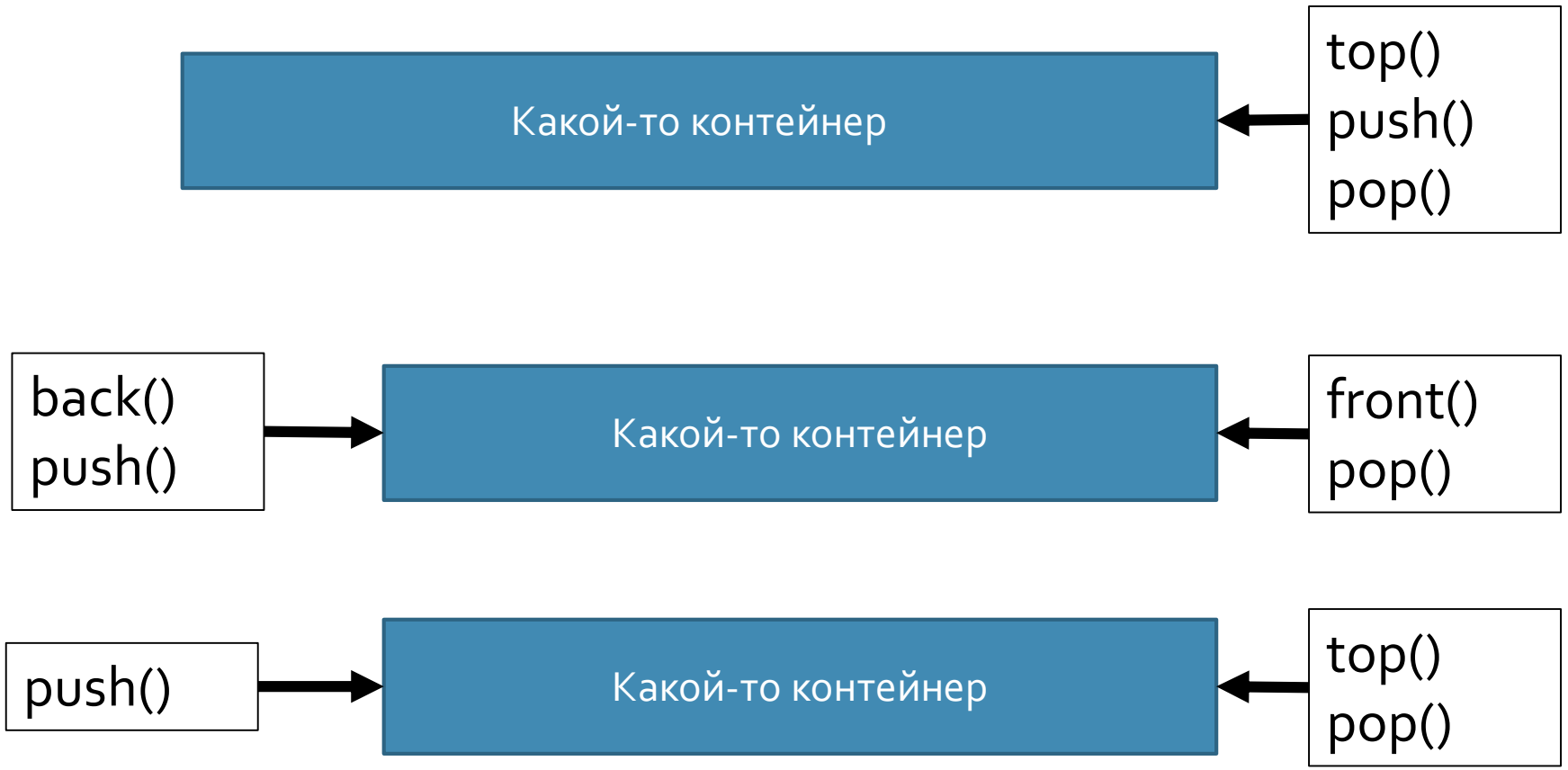
Но при этом empty у списков O(1)

Особые возможности списков

- Очистка
  - Remove
  - Unique
- Манипуляции списками
  - Splice
  - Reverse
  - Sort
  - Merge

- Непрерывные контейнеры
- Узловые контейнеры
- Адаптеры
- Полезные классы
- Строки

Идея контейнерных адаптеров



Излишняя ортогональность адаптеров

```
stack <int> s; // ok, это stack <int, deque<int>>
stack <int, vector<long>> s1; // сомнительно
stack <int, vector<char>> s2; // совсем плохо
s2.push(1000);
// Что вернёт s2.top()?
// Ещё хуже:
stack <int, forward_list<int>> s; // ошибка компиляции
// Но эта ошибка неочевидна. Стек же может быть сделан на односвязном списке. Но не в STL-uniform way.
```

Задача: борьба с интерфейсом

```
stack <T> s; // помним, что под ним дышит deque <T>

for (долгий-долгий цикл)
  s.push (сложное значение);

// тут много всего

// А вот тут надо очистить стек. Типа deque<T>::clear

// Но как?
```

Решение: и снова своп

```
stack <T> s; // помним, что под ним дышит deque <T>

for (долгий-долгий цикл)
  s.push (сложное значение);

// тут много всего

// А вот тут надо очистить стек. Типа deque<T>::clear

stack<T>().swap(s);
```

Обсуждение

- Почему стек, очередь и очередь с приоритетами не отдельные контейнеры?

- Непрерывные контейнеры
- Узловые контейнеры
- Адаптеры
- Полезные классы
- Строки

Соблазн: operator+ для векторов

Как мог бы работать operator+ в случае векторов?

```
vector<int> v1 { 2, 3, 5, 7 };

vector<int> v2 { 20, 30, 50, 70 };

// и вот здесь очень хочется
// auto v = v1 + v2;
// assert(v == (vector<int>){ 22, 33, 55, 77 });
```

Обсуждение: так может быть и определим специализацию vector<int> (ну может ещё парочку) и для неё (них) сделаем operator+?

Соблазн: operator+ для векторов

Как мог бы работать operator+ в случае векторов?

```
vector<int> v1 { 2, 3, 5, 7 };

vector<int> v2 { 20, 30, 50, 70 };

// и вот здесь очень хочется
// auto v = v1 + v2;
// assert(v == (vector<int>){ 22, 33, 55, 77 });
```

Обсуждение: так может быть и определим специализацию vector<int> (ну может ещё парочку) и для неё (них) сделаем operator+?

Hint: нет, vector<bool> многому нас научил

Valarrays: вектора значений

```
valarray<int> v1 { 2, 3, 5, 7 };
valarray<int> v2 { 20, 30, 50, 70 };
valarray<int> v3 = v1 + v2; // { 22, 33, 55, 77 }
```

И даже вот так (умножение трактуется как dot product):

```
valarray<int> v4 = v1 * v2 + v1 + v2; // { 62, 123, 305, 567 }
valarray<int> v4 = pow (v1, 2); // 4, 9, 25, 29
```

Но настоящая мощь valarrays даже не в этом

Особая возможность: slicing

- Slice (не путать со splice!) это векторный указатель.
- Идею проще всего посмотреть на примере:

```
valarray<int> row(n);
slice red(0, n/3, 3);
row[red]=255; // установить каждую третью ячейку row
```

- slice имеет начало, конец и инкремент, он похож на запись цикла и действительно можно было бы записать (но слайс эффективней):

```
for (int i = 0, i != n/3, i += 3)
  row[i] = 255;
```

Коротко о битовых масках

- bitset это альтернатива array<bool> то есть у него фиксированный размер, являющийся параметром контейнера

```
// 24-bit number
bitset<24> s = 0x7ff000;
```

- увы, выпилить vector<bool> как того требуют добро и справедливость нереально
- с другой стороны так ли он нужен?

- Непрерывные контейнеры
- Узловые контейнеры
- Адаптеры
- Полезные классы
  - Строки

От C-строк к std::string

```
string s = "hello, "; // инициализировать C-строкой
s.append("Eric, the Bloody Axe");  // добавить в конец символов
s[0] = 'H'; // изменить первый символ на заглавный
s += '!'; // добавить через +=
const char *content = s.c_str(); // получить содержимое
char buf[20];
s.copy(buf, s, s.size()); // скопировать содержимое
// здесь содержимое будет освобождено
```

Основные возможности строк

- Гарантии непрерывности памяти (C++11)
- size / capacity и resize / reserve
- push_back, operator[], etc
- find / rfind / replace
- substr
- c_str / data
- Строки «понимают» завершающий нулевой символ

Строки очень удобны

```
// скольких трудов стоило бы написать такое на чистых char*?
void
replace_all (string& str, const string& from, const string& to) {
  size_t st_pos = 0;
  if (from.empty ()) return;
  while ((st_pos = str.find (from, st_pos)) != string::npos) {
    str.replace (st_pos, from.length(), to);
    // In case 'to' contains 'from', like 'x' vs 'yx'
    st_pos += to.length();
  }
}
```

Обсуждение

- Есть ли случаи когда vector<char> лучше string?

Литература

- ISO/IEC, "Information technology -- Programming languages – C++", ISO/IEC 14882:2014, 2014
- The C++ Programming Language (4th Edition)
- Nicolai M. Josuttis, The C++ Standard Library - A Tutorial and Reference, 2nd Edition , Addison-Wesley, 2012
- Scott Meyers, Effective STL, 50 specific ways to improve your use of the standard template library, Addison-Wesley, 2001