ЛЕКЦИЯ 09
КОНТЕЙНЕРЫ

АЛГОРИТМИЗАЦИЯ И
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

ПОСЛЕДОВАТЕЛЬНЫЕ КОНТЕЙНЕРЫ

Контейнеры

- `vector` — массив с переменным размером и гарантией непрерывности памяти*
- `array` — массив с фиксированным размером, известным в момент компиляции
- `deque` — массив с переменным размером без гарантий по памяти
- `list` — двусвязный список
- `forward_list` — односвязный список

Адаптеры

- `stack` — LIFO контейнер, чаще всего на базе `deque`
- `queue` — FIFO контейнер, чаще всего на базе `deque`
- `priority_queue` — очередь с приоритетами, чаще всего на базе `vector`

*When choosing a container, remember vector is best.
Leave a comment to explain if you choose from the rest
(c) Tony van Eerd

ТРЕБОВАНИЯ К КОНТЕЙНЕРАМ

```
// C is vector, deque, list, array or forward_list
template <typename C> void forall(C &c) {
  // ???
}
```

Что я тут могу сделать независимо от того, что это за контейнер?

ТРЕБОВАНИЯ К КОНТЕЙНЕРАМ

```
// C is vector, deque, list, array or forward_list
template <typename C> void forall(C &c) {
  if (c.empty()) return;
  if (c.begin() == c.end()) return;
  C t = c;
  c.swap(t);
}
```

- Базовая общая функциональность невелика
- Но если вычеркнуть более экзотические `forward_list` и `array`, ситуация станет чуть лучше.

ТРЕБОВАНИЯ К КОНТЕЙНЕРАМ

```
// C is vector, deque, list, array or forward_list
template <typename C> bool forall_nofl(C &c) {
  return (c.size() == 3);
}
// C1 and C2 are vector, deque, list or forward_list
template <typename C1, typename C2>
void forall_noarr(C1 &c1, C2 &c2) {
  c2.clear();
  c2.assign(c1.begin(), c1.end());
}
```

- В целом вы почти всегда используете `vector`, `deque` или `list`

ПОСМОТРИМ НА НЕЗАКРАШЕННЫЕ ТОЧКИ

Контейнеры

- `vector` — массив с переменным размером и гарантией непрерывности памяти*
- `array` — массив с фиксированным размером, известным в момент компиляции
- `deque` — массив с переменным размером без гарантий по памяти
- `list` — двусвязный список
- `forward_list` — односвязный список

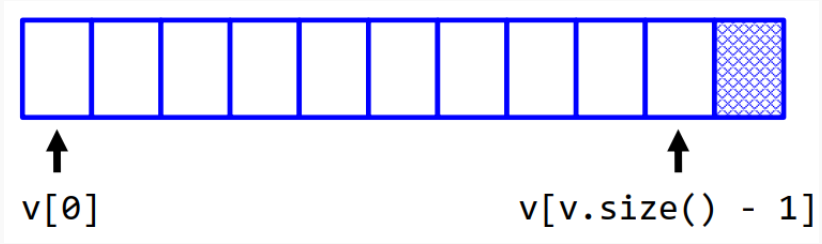
Адаптеры

- `stack` — LIFO контейнер, чаще всего на базе `deque`
- `queue` — FIFO контейнер, чаще всего на базе `deque`
- `priority_queue` — очередь с приоритетами, чаще всего на базе `vector`

*When choosing a container, remember vector is best.
Leave a comment to explain if you choose from the rest
(c) Tony van Eerd

ГАРАНТИИ НЕПРЕРЫВНОСТИ ПАМЯТИ

```
// функция init написана в старом стиле
template <typename T> void init (T* arr, size_t size);
// но её можно использовать с векторами
vector<T> t(n);
T *start = &t[0];
init_t(start, n);
assert(t[1] == start[1]);
```



*When choosing a container, remember vector is best.
Leave a comment to explain if you choose from the rest
(c) Tony van Eerd

НЕПРИЯТНОЕ ИСКЛЮЧЕНИЕ: VECTOR<BOOL>

```
vector<bool> t(n);
bool *start = &t[0]; // это не скомпилируется, но представим
assert (t[1] == start[1]); // oops!
```

Важно запомнить две вещи

- `vector<bool>` не удовлетворяет соглашениям контейнера `vector`
- `vector<bool>` не содержит элементов типа `bool`
- Не используйте `vector<bool>` для обобщённого программирования

```
using vector_bool = vector<bool>;
vector_bool x(10); // условно ок, но тут лучше std::bitset
```

ЗАДАЧА: ЧТО МОЖНО ЗДЕСЬ УЛУЧШИТЬ?

```
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
```

- При вставке в конец вектору могут потребоваться реаллокации памяти
- Это означает, что всегда полезно думать о памяти вектора не меньше, чем о памяти динамического массива

ЕЩЁ ПРО SIZE И CAPACITY

- size это сколько элементов у вектора уже есть
- capacity это сколько элементов в нём может быть до первого перевыделения

```
vector<int> v(10000);
assert (v.size() == 10000);
assert (v.capacity() >= 10000);
```

- Размер это что-то, чем можно в явном виде управлять в отличии от ёмкости

```
v.resize(100);
assert (v.size() == 100);
assert (v.capacity() >= 10000);
```

АМОРТИЗАЦИЯ

- При написании метода push, вам предлагалось оценить его алгоритмическую сложность
- Проблема в том, что она очевидно O(1) если не надо реаллоцировать и O(n) если надо
- То есть мы платим иногда. Это примерно как купить машину и платить только за бензин пока машина не износится, а потом купить новую
- В экономике распределение стоимости товара по стоимости его периода эксплуатации называется амортизацией товара
- Амортизированное O(n) обозначается O(n)+

АМОРТИЗИРОВАННАЯ СТОИМОСТЬ

- По определению амортизированная стоимость операции это стоимость N операций, отнесённая к N
- Для динамического массива $c_i = 1 + [realloc] \cdot (i - 1)$
- Амортизированная стоимость одной вставки будет $\frac{\sum_i c_i}{N}$ для N вставок
- Допустим, мы, если реаллокация нужна, растим массив на 10 элементов $\sum_i c_i = ?$

АМОРТИЗИРОВАННАЯ СТОИМОСТЬ

- По определению амортизированная стоимость операции это стоимость N операций, отнесённая к N
- Для динамического массива $c_i = 1 + [realloc] \cdot (i - 1)$
- Амортизированная стоимость одной вставки будет $\frac{\sum_i c_i}{N}$ для N вставок
- Допустим, мы, если реаллокация нужна, растим массив на 10 элементов $\sum_i c_i = N + \sum_{k=1}^{N/10} 10 \cdot k = O(N^2)$
- Заметим, что это очень плохая стратегия. Амортизированная сложность push будет $\frac{O(N^2)}{N} = O(N) +$. Можем ли мы придумать и доказать нечто лучшее?

ЛУЧШАЯ СТРАТЕГИЯ

- Прирост вдвое

$$\frac{\sum c_i}{N} = \frac{N + \sum_{j=1}^{\lg N} 2^j}{N} = \frac{O(N)}{N} = O(1) +$$

- Видно, что разница есть: при одной стратегии у нас в среднем линейное а при другой в среднем постоянное время вставки
- Увы, взять сумму $\sum_{j=1}^{\lg N} 2^j$ в общем уже не так просто, а при более сложных стратегиях, это становится мучительно
- Можем ли мы упростить себе жизнь?

ДОПОЛНЕНИЕ: МЕТОД ПОТЕНЦИАЛА

- Выберем функцию потенциала $\Phi(n)$ так, чтобы $\Phi(0) = 0, \Phi(n) \geq 0$
- Здесь n это номер шага
- Амортизированная стоимость это стоимость плюс изменение потенциальной функции $c_n + \Phi(n) - \Phi(n-1)$
- Выбор потенциальной функции облегчает вычисления потому что

$\sum_i (c_i + \Phi(i) - \Phi(i-1)) = \Phi(n) - \Phi(0) + \sum_i c_i \geq \sum_i c_i$

- Удачный выбор сделает выражение $\sum_i (c_i + \Phi(i) - \Phi(i-1))$ проще, чем $\sum_i c_i$
- Обсуждение: как выбрать для массива?

ДОПОЛНЕНИЕ: МЕТОД ПОТЕНЦИАЛА

- Для массива поскольку при реаллокации вдвое $2 \cdot s_n \geq c_n$

$\Phi(n) = 2 \cdot s_n - c_n$

- Без реаллокации

$c_i + \Phi(i) - \Phi(i-1) = 1 + (2 \cdot s_i - C) - (2 \cdot s_{i-1} - C) = 1 + 2(s_i - s_{i-1}) = 3$

- С реаллокацией $\Phi(i-1) = 2k - k = k, \Phi(i) = 2(k+1) - 2k = 2$

$c_i + \Phi(t_i) - \Phi(t_{i-1}) = (k+1) + 2 - k = 3$

- В итоге в любом случае $\sum c_i \leq 3N$ и мы доказали асимптотику $O(1)$
- В качестве упражнения на дом проанализируйте стратегию роста в $\log(N)$ раз

ОБСУЖДЕНИЕ

- Выбор простого роста вдвое не всегда лучшая стратегия
- Реальная стратегия из libstdc++ несколько сложнее и обладает рядом приятных теоретических свойств

```
const size_type __len = size() + std::max(size(), __n);
```

- Попробуйте дома проанализировать эту стратегию и обосновать почему она выбрана в качестве основной

ЧТО МОЖЕТ СМУЩАТЬ В ЭТОМ КОДЕ?

```
std::deque<int> d; // подумайте если бы это был vector?
for (int i = 0; i != N; ++i) {
  d.push_front(i);
  d.push_back(i);
}
```

- deque — массив с переменным размером без гарантий по памяти

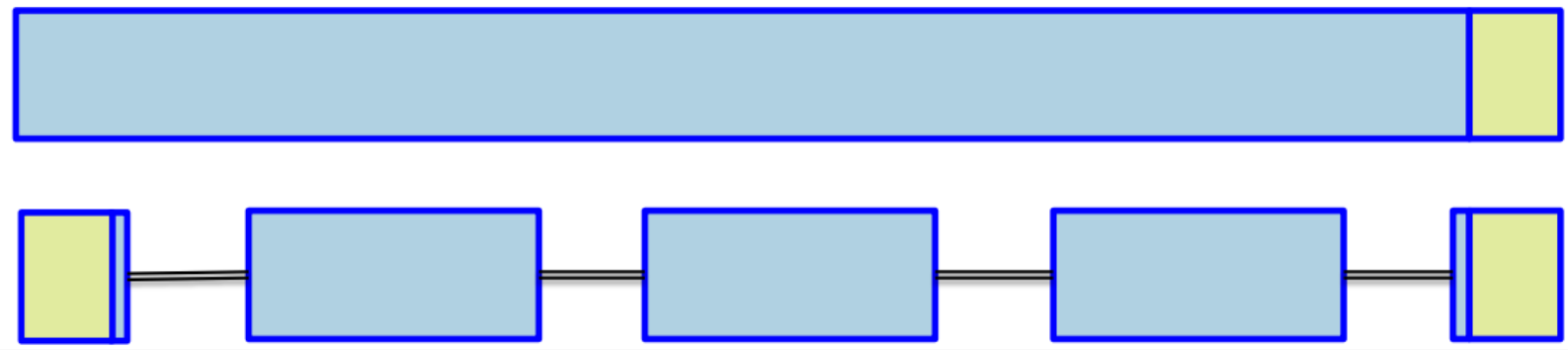
ЧТО МОЖЕТ СМУЩАТЬ В ЭТОМ КОДЕ?

```
std::deque<int> d; // подумайте если бы это был vector?
for (int i = 0; i != N; ++i) {
  d.push_front(i);
  d.push_back(i);
}
```

- `deque` — массив с переменным размером без гарантий по памяти
- Поэтому ответ: всё хорошо.
- Вставка в начало и в конец дека имеет всегда честную константную сложность $O(1)$

РАССМОТРИТЕ DEQUE ВМЕСТО VECTOR*

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



**Но оставьте комментарий в коде если вы его действительно выберете*

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

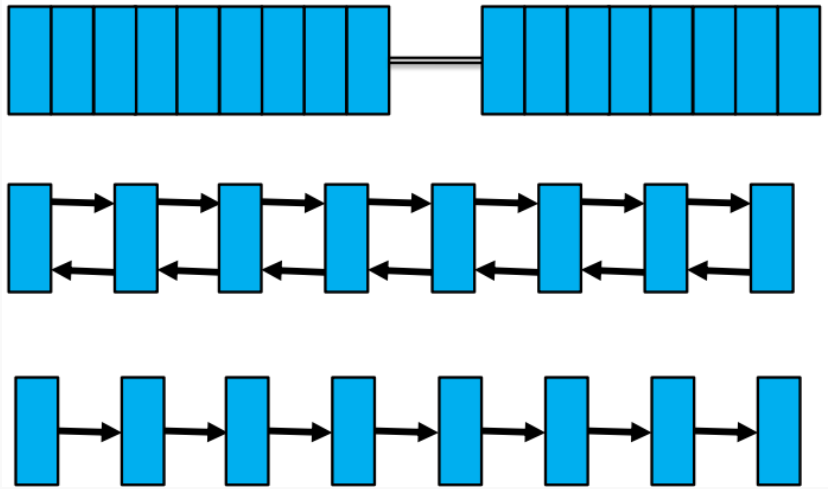
- Доступ к элементу O(1)
- Вставка в конец O(1)
- Вставка в начало O(1)
- Вставка в середину O(N)
- Вычисление размера O(1)
- Нет гарантий по памяти
- Нет необходимости в reserve/capacity

ОБСУЖДЕНИЕ

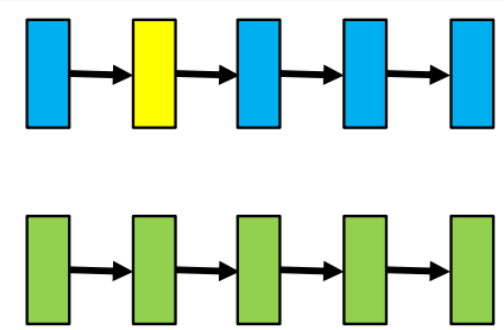
- *"deque is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence"*
- А как бы вы реализовали `deque`?

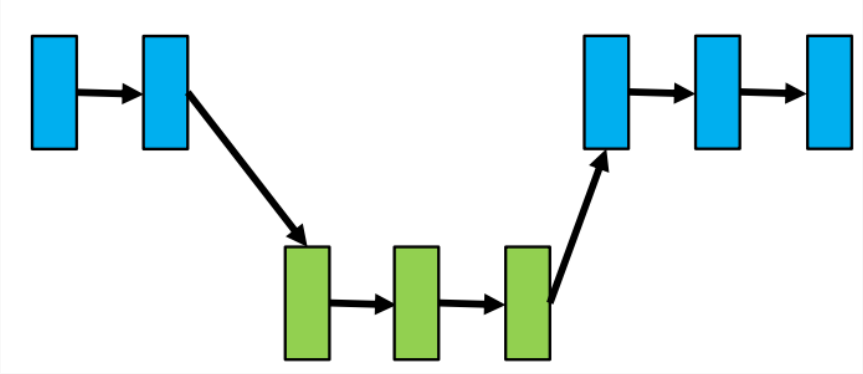
УЗЛОВЫЕ КОНТЕЙНЕРЫ

- **deque** произвольный доступ, быстрая вставка в начало и в конец.
- **forward_list** последовательный доступ, быстрая вставка в любое место.
- **list** последовательный доступ, быстрая вставка в любое место, итерация в обе стороны.



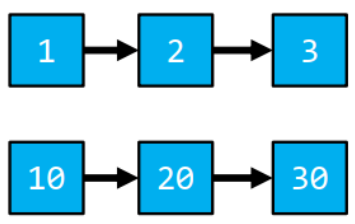
ОСОБАЯ ВОЗМОЖНОСТЬ СПИСКОВ: СПЛАЙС

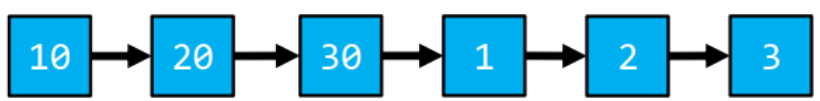




СПЛАЙС ДЛЯ СПИСКОВ: ПРОСТАЯ ФОРМА

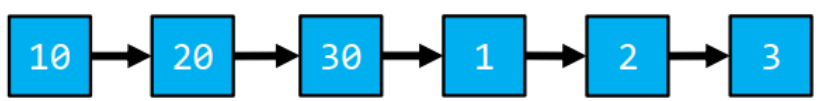
```
forward_list<int> fst = { 1, 2, 3 };
forward_list<int> snd = { 10, 20, 30 };
auto it = fst.begin(); // указывает на 1
// перемещаем second в начало first, it указывает на 1
fst.splice_after(fst.before_begin(), snd);
```

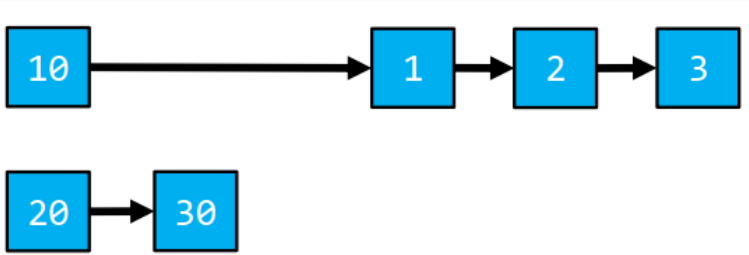




СПЛАЙС ДЛЯ СПИСКОВ: СЛОЖНАЯ ФОРМА

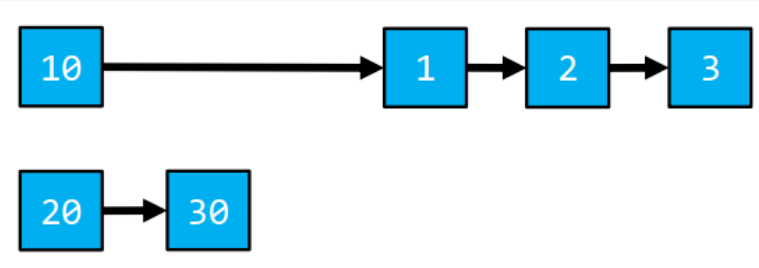
```
// forward_list<int> fst = {10, 20, 30, 1, 2, 3 };
// forward_list<int> snd = {};
// it указывает на 1
// перекидываем элементы со второго по it в список second
snd.splice_after(snd.before_begin(), fst, fst.begin(), it)
```

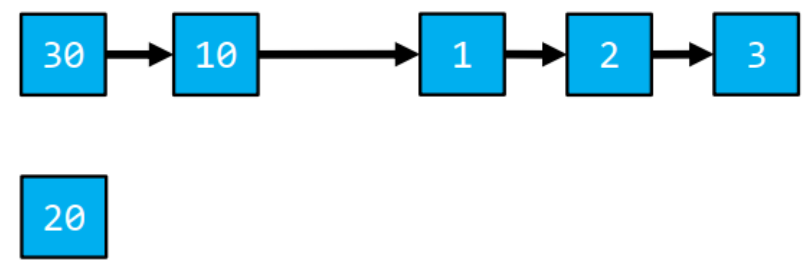




СПЛАЙС ДЛЯ СПИСКОВ: СРЕДНЯЯ ФОРМА

```
// forward_list<int> fst = { 10, 1, 2, 3 };
// forward_list<int> snd = { 20, 30 };
// it указывает на 1
// все элементы второго списка начиная со второго в первый
fst.splice_after(fst.before_begin(), snd, snd.begin());
```





ОБСУЖДЕНИЕ

- Какие вы видите применения спискам?

ИДЕЯ КОНТЕЙНЕРНЫХ АДАПТЕРОВ

ВИДЫ АДАПТЕРОВ

- **stack** – LIFO стек над последовательным контейнером

```
template <class T, class Container = deque<T>> class stack;
```

- **queue** – FIFO очередь над последовательным контейнером

```
template <class T, class Container = deque<T>> class queue;
```

- **priority_queue** – очередь с приоритетами (как binary heap) над последовательным контейнером

```
template <class T,
          class Container = vector<T>,
          class Compare = less<typename Conatiner::value_type>>
class priority_queue;
```

АЛГОРИТМ ПРИМА

```
pq.push(std::make_pair(first(G), src)); // first(G)
while(!pq.empty()) {
  auto elt = pq.top().second; pq.pop();
  if (mst[elt]) continue;              // mst[v]
  for (auto e: adjacent(G, elt)) {  // adjacent(G, v)
    w = weight(G, e); v = tip(G, e);// weight(G, e); tip(G, e)
    if (!mst[elt] && key[v] < w) {  // key[v]
      key[v] = w; parent[v] = u;    // parent[v]
      pq.push(std::make_pair(w, v));
    }
  }
}
```

ЗАЩИТА ОТ ОРТОГОНАЛЬНОСТИ

```
std::stack<int> s; // ok, это stack<int, deque<int>>
std::stack<int, std::vector<long>> s1; // сомнительно
std::stack<int, std::vector<char>> s2; // совсем плохо
s2.push(1000);
// что вернет s2.top()?
```

К счастью, это безобразие перекрыто `static asserts`

НЕДОСТАТОЧНАЯ ОРТОГОНАЛЬНОСТЬ

```
std::stack<int, std::forward_list<int>> s; // ok
s.push(100); // ошибка: нет push_back
s.pop();     // ошибка: нет pop_back
s.top();     // ошибка: нет back
```

- Эти ошибки неочевидны
- Стек вполне может быть сделан на односвязном списке
- Но адаптер `std::stack` требует (неявно) вполне определенный интерфейс

ОБСУЖДЕНИЕ

- Почему стек, очередь и очередь с приоритетами не отдельные контейнеры?
- И почему двухголовая очередь deque не адаптер?

О БИТОВЫХ МАСКАХ

- `bitset` — это альтернатива `array<bool>`, то есть у него фиксированный размер, являющийся параметров контейнера
- При этом он хранит данные более компактно (как vector<bool>)

```
// 24-bit number
bitset<24> s1 = 0x7ff00;
bitset<24> s2 = 0xff00;
s1[0] = 1; // или s1.set(0) или s1.set(0, 1)
auto s3 = s1 & s2; // s3 = 0xf000
```

- По сути он делает array<bool> не нужным

О СТРОКАХ. СНОВА

- Почему специальный `std::string`, а не `std::vector<char>`?
- Важная ремарка: формально `std::string` – это непрерывный контейнер, имеющий с вектором очень много общего

БАЗОВАЯ ФУНКЦИОНАЛЬНОСТЬ

```
#include <cstring>
#include <cassert>

char astr[] = “hello”;
char bstr[15];
int alen = std::strlen(astr);
assert(alen == 5);
std::strcpy(bstr, astr);
std::strcat(bstr, “, world”);
res= std::strcmp(astr, bstr);
assert(res < 0);
foo(bstr);
```

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

string astr = “hello”;
string bstr;
int alen = astr.length();
assert(alen == 5);
bstr = astr;
bstr += “, world”;
res = astr.compare(bstr);
assert(res < 0);
foo(bstr.c_str());
```

ШАБЛОН КЛАССА СТРОКИ

- Представим (а это не так), что строка устроена так:

```
template <typename CharT> class basic_string { …. };
```

- Определения для удобства

```
typedef basic_string<char> string;
typedef basic_string<u16char_t> u16string;
typedef basic_string<u32char_t> u32string;
typedef basic_string<wchar_t> wstring;
```

- Что бросается в глаза?

ХАРАКТЕРИСТИКИ ТИПОВ

- Есть много вопросов, ответы на которые разные для разных строк с разными типами символов. Разумно свести всё это в класс

```
template <typename CharT> class char_traits { .... };
```

- Основные методы

```
assign, eq, lt, move, compare, find, eof, ....
template <typename CharT,
          typename Traits = std::char_traits<CharT>>
class basic_string;
```

- К слову, а является ли способ выделения памяти характеристикой символа?

АЛЛОКАТОРЫ

- Выделение памяти абстрагирует аллокатор. Стандартный аллокатор сводится к `malloc`.

```
template <typename CharT,
          typename Traits = std::char_traits<CharT>
          typename Allocator = std::allocator<CharT>>
class basic_string;
```

- К слову, полный шаблон вектора тоже выглядит не вполне очевидно

```
template <typename T,
          typename Allocator = std::allocator<T>>
class vector;
```

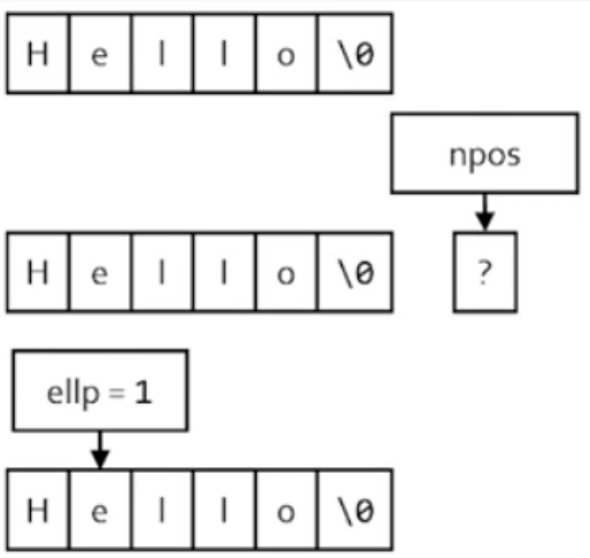
ОБСУЖДЕНИЕ

- Следующие вопросы не очень связаны логически
- Как по вашему выглядит аллокатор `std::list`?
- Как вы думаете, строка должна иметь методы вроде `capacity` и `reserve`?
- Ну и раз мы вынесли строку в отдельный класс, что вы думаете о специальных интерфейсах для нее?

ПОИСК В СТРОКАХ

- Строки предлагают эффективные специальные возможности поиска в них

```
string s = “Hello”;
unsigned long notfound = s.find(“bye”);
assert(notfound == std::string::npos);
unsigned long ellp = s.find(“ell”);
unsigned long hpos = s.find(“H”, ellp);
assert(hpos == std::string::npos);
```

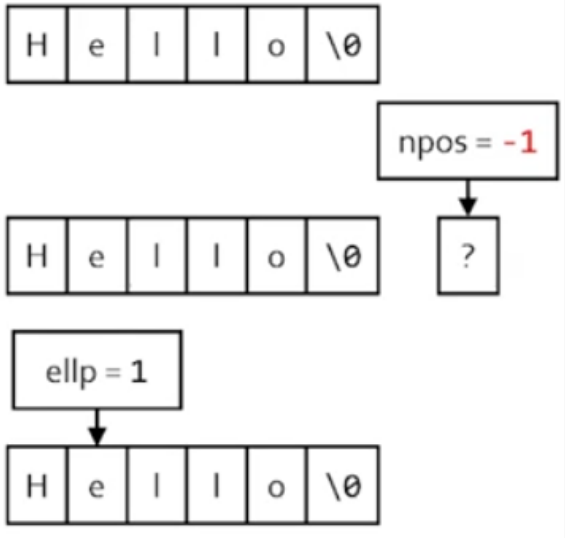


- Кто видит возможную проблему в этом коде?

ПОИСК В СТРОКАХ

- Но использование этих возможностей таит сюрпризы

```
using szt = std::string::size_type;
string s = “Hello”;
szt notfound = s.find(“bye”);
assert(notfound == std::string::npos);
szt ellp = s.find(“ell”);
szt long hpos = s.find(“H”, ellp);
assert(hpos == std::string::npos);
```



ПРОБЛЕМА СТАТИЧЕСКИХ СТРОК

- Что вы думаете об использовании константных статических строк?

```
static const std::string kName = “oh literal, my literal”;
// ....
int foo(const std::string &s);
// ....
foo(kName);
```

РЕШЕНИЕ: STRING_VIEW (SINCE C++17)

- `string_view` – это невладеющий указатель на строку

```
static std::string_view kName = “oh literal, my literal”;
// ....
int foo(std::string_view s);
// ....
foo(kName);
```

Здесь нет ни heap indirection, ни создания временного объекта

БАЗОВЫЕ ОПЕРАЦИИ НАД STRING_VIEW

- remove_prefix
- remove_suffix
- copy
- substr
- compare
- find
- data

```
std::string str = "    trim me  ";
std::string_view sv = str;
auto trimfst = sv.find_first_not_of(" ");
auto minsz = std::min(trimfst, sv.size());

sv.remove_prefix(minsz);

auto trimlst = sv.find_last_not_of(" ");
auto sz = sv.size() - 1;
minsz = std::min(trimlst, sz);

sv.remove_suffix(sz - minsz);
```

VIEWS: ИДЕЯ ДЛЯ SPAN (SINCE C++20)

- `std::span` для одномерных массивов то же, что `std::string_view` для строк

```
int arr[4] = {1, 2, 3, 4}; // просто данные
std::array<int, 4> arr = {1, 2, 3, 4}; // копирование до main
```

- `std::span` решает эту проблему

```
std::span<int, 4> arr = {1, 2, 3, 4}; // просто данные
```

- По умолчанию второй параметр N – это `std::dynamic_extent`

```
std::span<int> dynarr(arr); // неизвестный размер
```

- Разумеется, у него куда более простой интерфейс, чем у `std::string_view`

ОБСУЖДЕНИЕ

- Хватит ли нам последовательных контейнеров?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Бьерн Страуструп, Язык программирования C++/ ред. А. Боборыкин. – 4-е изд. - Москва: Издательство БИНОМ, 2023. – 1213 с.
2. Дональд Кнут Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol.1. Fundamental Algorithms. — 3-е изд. — М .: «Вильямс» , 2006. – 720 с.
3. Дональд Кнут Искусство программирования, том 2. Получисленные алгоритмы = The Art of Computer Programming, vol.2. Seminumerical Algorithms. — 3-е изд. — М .: «Вильямс» , 2007. — 832 с.
4. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦМНО, 1999. – 960 с.
5. Скотт Мейерс, Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2017. – 300 с.
6. Скотт Мейерс, Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2016. – 298 с.