КОНТЕЙНЕРЫ

Последовательные и ассоциативные контейнеры, адаптеры и вспомогательные классы

К. Владимиров, Intel, 2022
mail-to: konstantin.vladimirov@gmail.com

- Последовательные контейнеры
- Контейнеро-подобные классы
- Ассоциативные контейнеры
- Упорядоченные контейнеры

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

Что может смущать в этом коде?

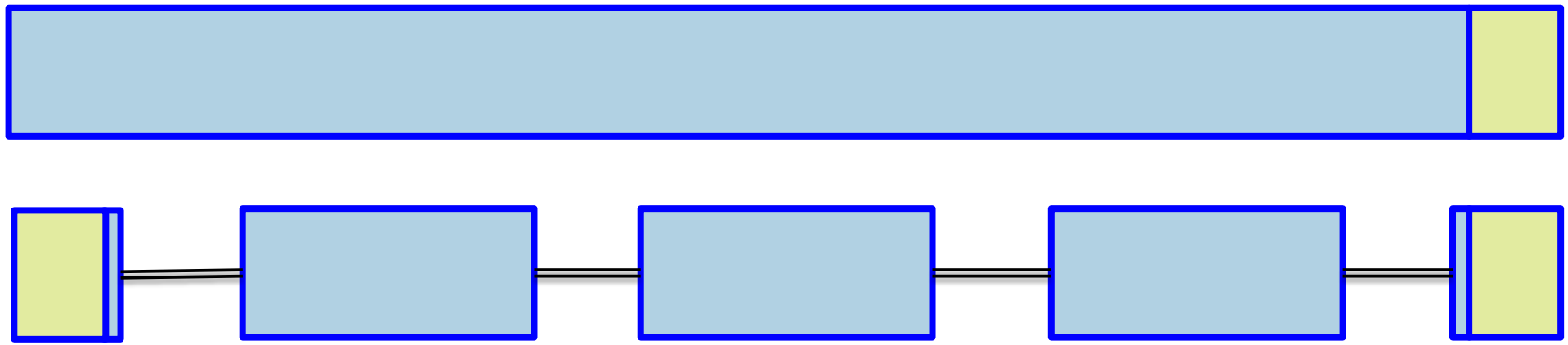
```
std::deque<int> d; // подумайте если бы это был vector?

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 $O(1)$.

Fill bench

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

Access bench

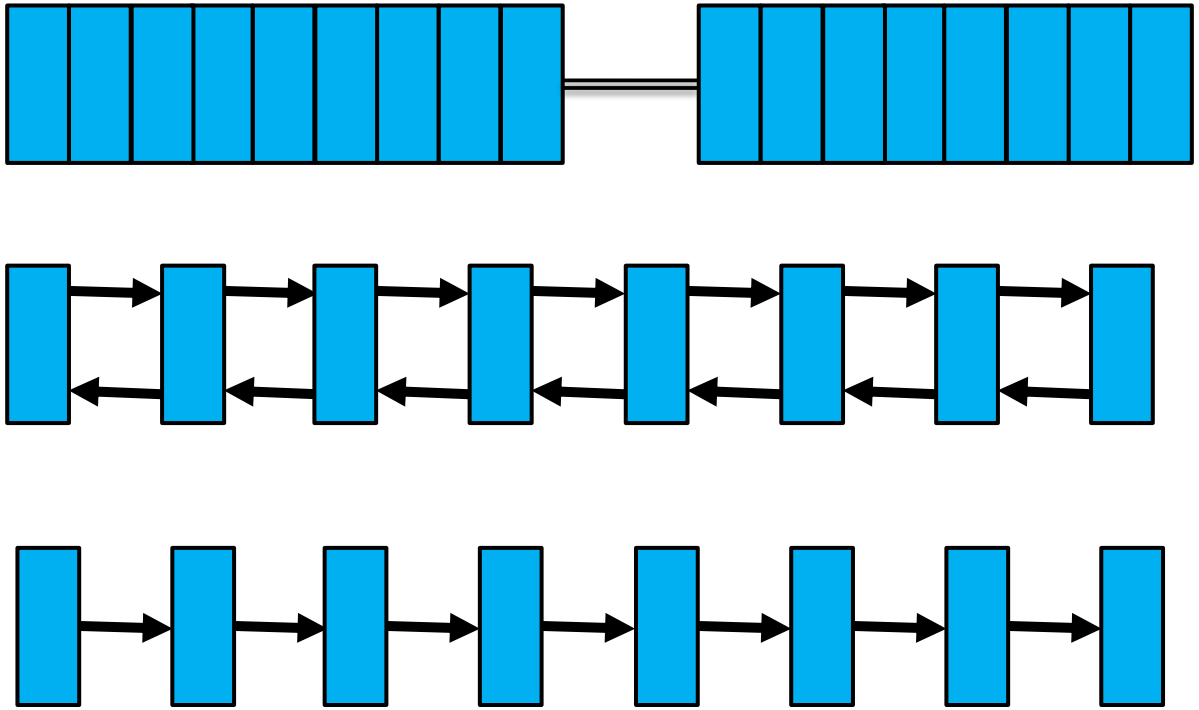
Обсуждение

- "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?

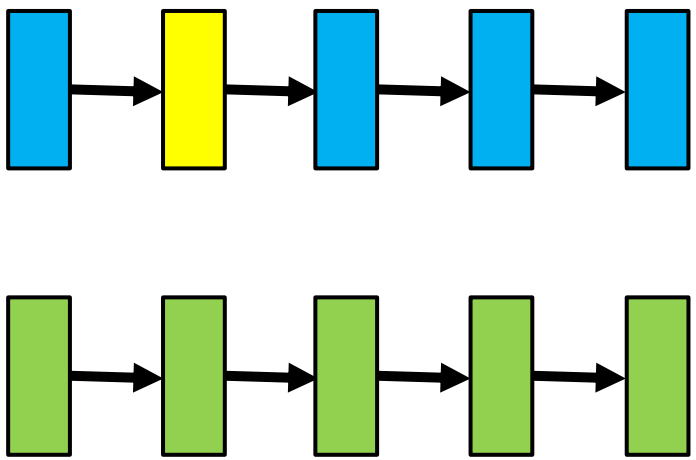
An In-Depth Study of the STL Deque Container

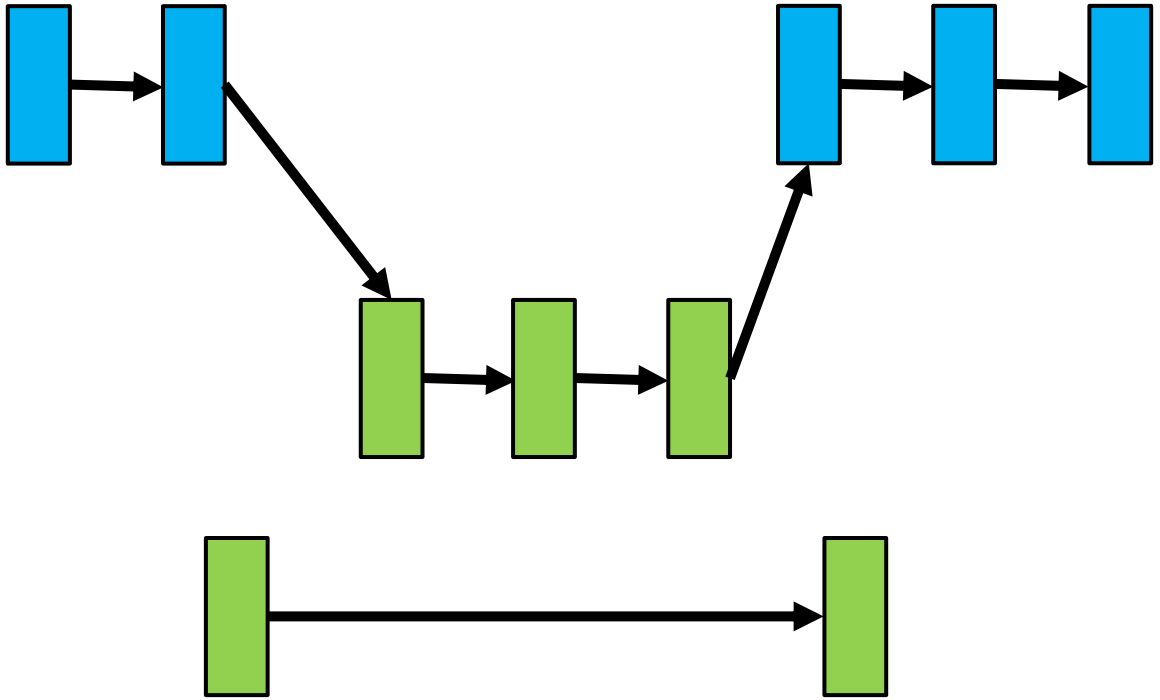
Узловые контейнеры

- **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.



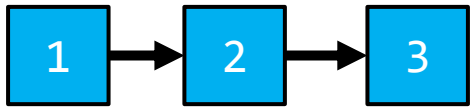
Особая возможность списков: сплайс

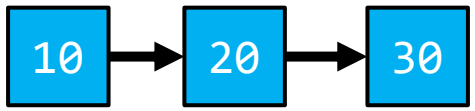


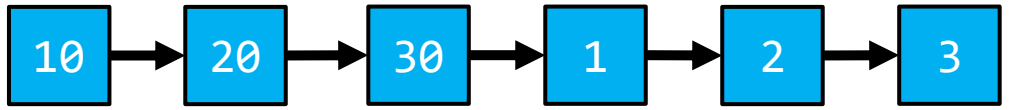


Сплайс для списков: простая форма

```
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);
```

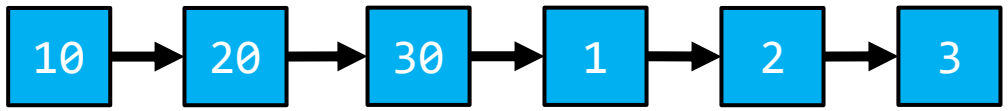


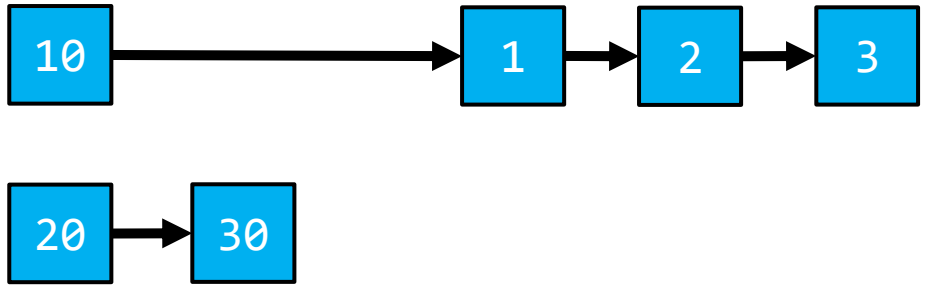




Сплайс для списков: сложная форма

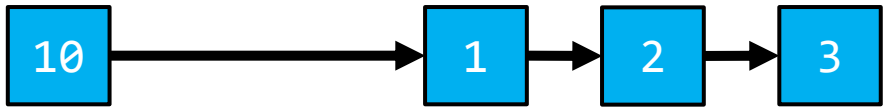
```
// forward_list<int> fst = {10, 20, 30, 1, 2, 3 };
// forward_list<int> snd = {};
// it указывает на 1

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

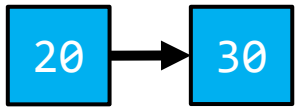


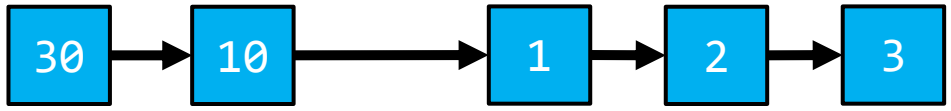


Сплайс для списков: средняя форма

```
// 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());
```







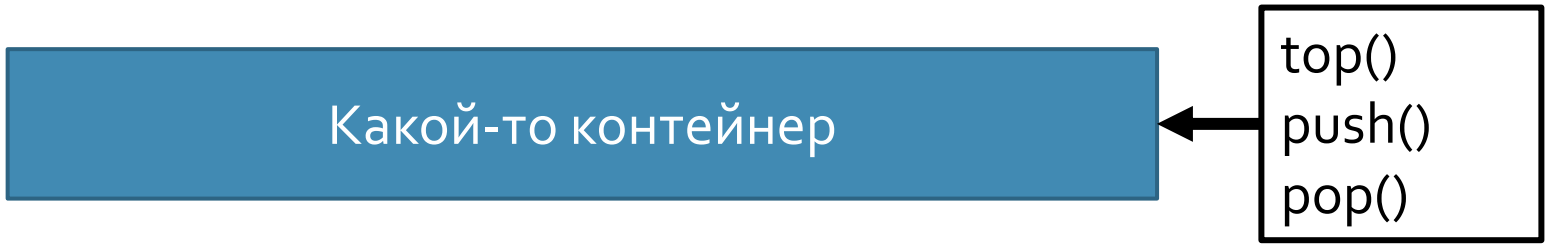
https://godbolt.org/z/P9M9arjTd

Обсуждение

- Какие вы видите применения спискам?

Идея контейнерных адаптеров

"Стек"



"Очередь"

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 Container::value_type>>
class priority_queue;
```

Case study: 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));
    }
  }
}
```


https://godbolt.org/z/K9Tbbeeqv

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 вернёт s2.top()?
```

- К счастью всё это безобразие перекрыто static asserts

https://godbolt.org/z/4Yr4EE7Ye

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 требует) вполне определённый интерфейс

https://godbolt.org/z/8vzhx1r59

Обсуждение

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

- Последовательные контейнеры
  - Контейнеро-подобные классы
- Ассоциативные контейнеры
- Упорядоченные контейнеры

Коротко о битовых масках

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

```
// 24-bit number
bitset<24> s1 = 0x7ff000;
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.

https://godbolt.org/z/5hs789KP7

Обсуждение: поговорим о строках

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

Строки: 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 (это не так) что строка была бы устроена вот 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

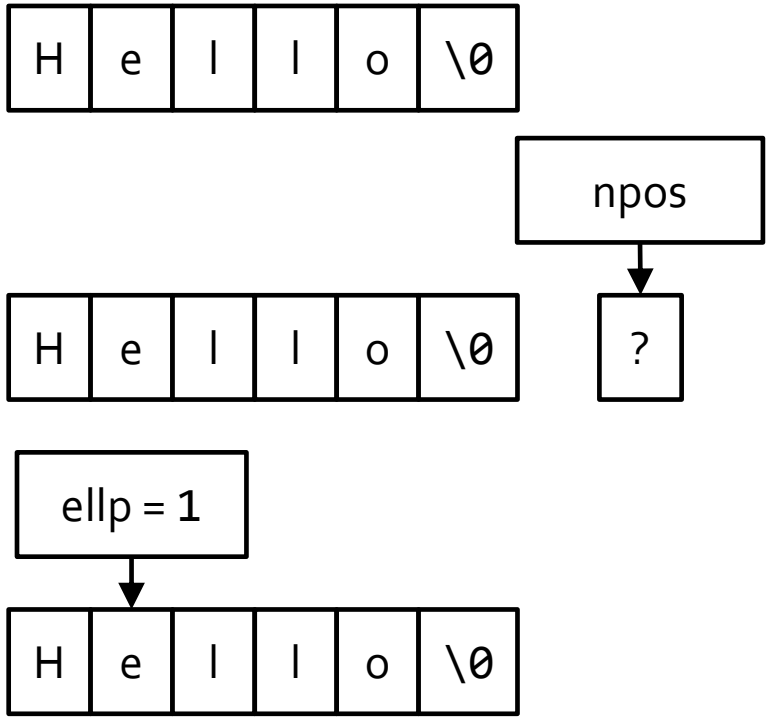
- Следующие вопросы не слишком логически связаны
- Как по вашему выглядит аллокатор для `std::list`?
- Как вы думаете, строка должна иметь методы вроде `reserve` и `capacity`?
- Ну и раз уж мы вынесли строку в отдельный класс, что вы думаете о специальных интерфейсах для неё?

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);
```

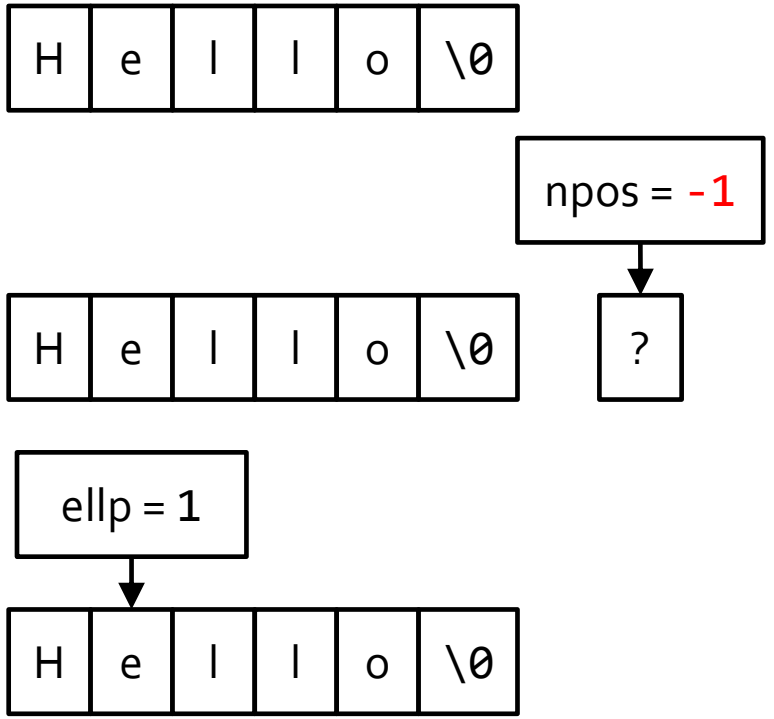
- Кто видит возможную проблему в этом коде?



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 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 &arg);
// .....
foo(kName);
```

Решение: string_view (C++17)

- string_view это невладеющий указатель на строку

```
static std::string_view kName = "oh literal, my literal";
// .....
int foo(std::string_view arg);
// .....
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);
```


https://godbolt.org/z/sff3E8cGs

Views: идея для span (C++20)

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

- 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 string view.

Обсуждение

- Хватит ли нам последовательных контейнеров?

- Последовательные контейнеры
- Контейнеро-подобные классы
- Ассоциативные контейнеры
- Упорядоченные контейнеры

Смысл ассоциативности

- Вектора индексированы целыми числами и позволяют сопоставить целое число хранимому значению

```
vector<T> v; // int → T
```

- Как сделать произвольное отображение T → U?

Ассоциативный массив

- Основная идея ассоциативного массива это контейнер unordered map

```
template<
  typename Key, typename T,
  typename Hash = std::hash<Key>,
  typename KeyEqual = std::equal_to<Key>,
  typename Allocator = std::allocator<std::pair<const Key, T>>
> class unordered_map;
```

- Здесь важными являются два отношения: отношение equals и собственно hash функция.
- При этом ключи уникальны и мы можем менять значения но не ключи.

https://godbolt.org/z/4sx38b4dW

Обсуждение: собственный ключ

- Допустим у нас есть пользовательская структура из двух строк

```
struct S { std::string first_name, last_name; };

std::unordered_map<S, std::string> Ump; // error
```

- Для неё нужно сделать две вещи
  - Определить равенство (все ли помнят как)
  - Определить хеш. Есть ли тут у вас идеи как именно? Хорош ли вариант по ссылке?
- Обратите внимание: мы можем добавлять в стандартную библиотеку специализации.

https://godbolt.org/z/4E4W38WjM

Собственный hash

- Простейший способ это сделать что-нибудь исходя из фантазии

```
size_t operator()(const S& s) const noexcept {
  std::hash<std::string> h;
  auto h1 = h(s.first_name), h2 = h(s.last_name);
  return h1 ^ (h2 << 1);
}
```

- Этот способ привлекателен, так как мы же программисты
- Часто (например в этом случае) он даже работает
- Но в общем это всегда угадайка

Собственный hash

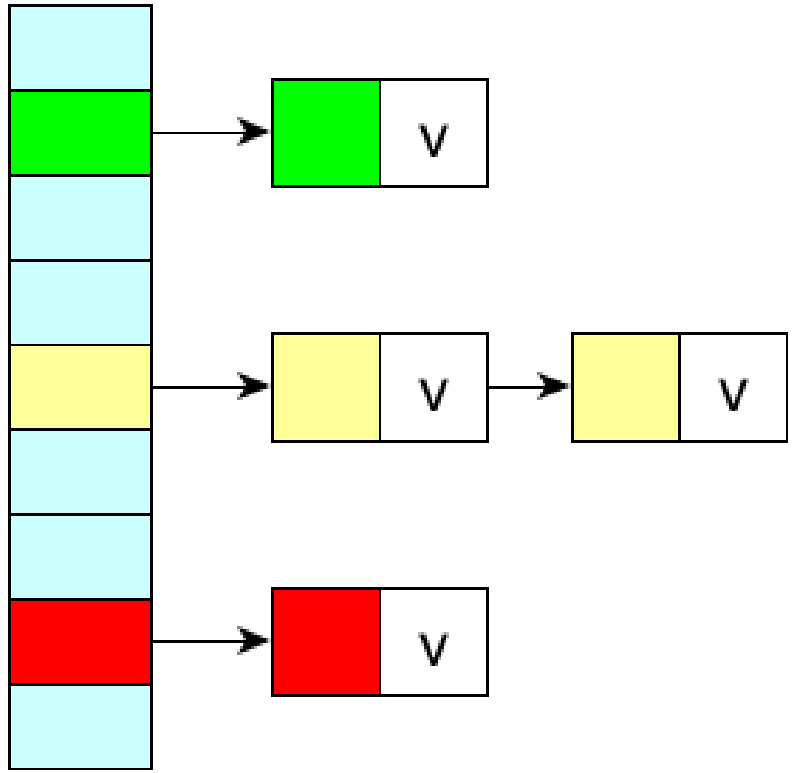
- Если угадайка не привлекает, есть boost

```
size_t operator()(const S& s) const noexcept {
  std::hash<std::string> h;
  auto h1 = h(s.first_name), h2 = h(s.last_name);
  size_t seed = 0;
  boost::hash_combine(seed, h1);
  boost::hash_combine(seed, h2);
  return seed;
}
```

- Это работает всегда. Но это boost, его надо затаскивать в проект.

Представление в памяти*

- О хеш-таблицах можно думать как о массиве корзин (buckets), каждая из которых содержит элементы с одинаковым хешом.
- Это даёт асимптотически быстрый поиск (индексацию по массиву) если load factor хорош.
- load factor = size / bucket count
- На картинке слева это 0.75 и в общем это **уже** довольно плохо.



* это не настоящее представление в памяти

Низкоуровневая информация

- Дополнительно каждый неупорядоченный контейнер даёт возможность смотреть его статистику
- `bucket_count()` — количество бакетов
- `max_bucket_count()` — максимальное количество бакетов без реаллокаций
- `bucket_size(n)` — размер бакета с номером n
- `bucket(Key)` — номер бакета для ключа Key
- `load_factor()` — среднее количество ключей в бакете
- `max_load_factor()` — максимальное количество ключей в бакете

Обсуждение

- По сути неупорядоченный контейнер это что-то вроде гибрида непрерывного и узлового последовательного контейнера.
- Что это означает в практическом смысле в плане управления памятью?
- Напомню: в узловых контейнерах (list) управлять памятью не нужно кроме случаев особых аллокаторов. А в последовательных (vector) об этом нельзя забывать.

Рехэш

- Особая функция `rehash(count)` служит для того, чтобы изменить количество бакетов (установить в count) и перераспределить по ним элементы
- `reserve(count)` делает то же самое, что `rehash(ceil(count / max_load_factor()))`
- Особый случай `rehash(0)` позволяет безусловно (в автоматическом режиме) перехешировать контейнер

Резервирование памяти

- Следующий эксперимент показывает эффект резервирования

```
std::unordered_map<int, Foo> mapNoReserve, mapReserve;
// контрольная точка 1

mapReserve.reserve(1000);
// контрольная точка 2

for(int i = 0; i < 1000; ++i) {
  mapNoReserve.emplace(i, Foo());
  mapReserve.emplace(i, Foo());
}
// контрольная точка 3
```


https://godbolt.org/z/jWKdWxWx3

Два вида итерации

- По хеш-таблице можно итерировать как по единому целому

```
for (auto it = m.begin(); it != m.end(); ++it)
```

- Можно итерироваться внутри бакета, указав его номер

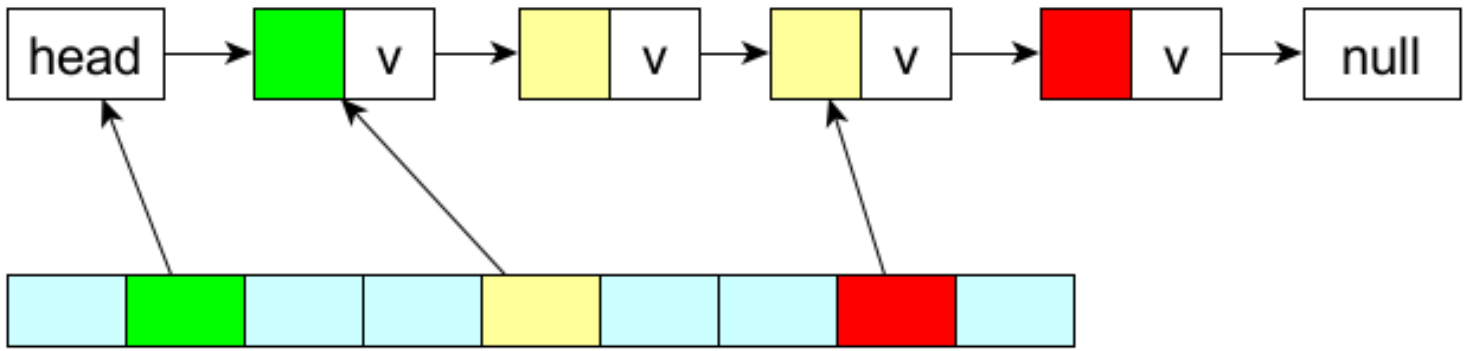
```
for (int i = 0; i < m.bucket_count(); ++i) {
  for (auto it = m.begin(i); it != m.end(i); ++it)
```

- В обоих случаях вам доступен только forward iterator.
- Как бы вы написали адаптор чтобы позволить второй вариант через range based for?

https://godbolt.org/z/3dT3scf35

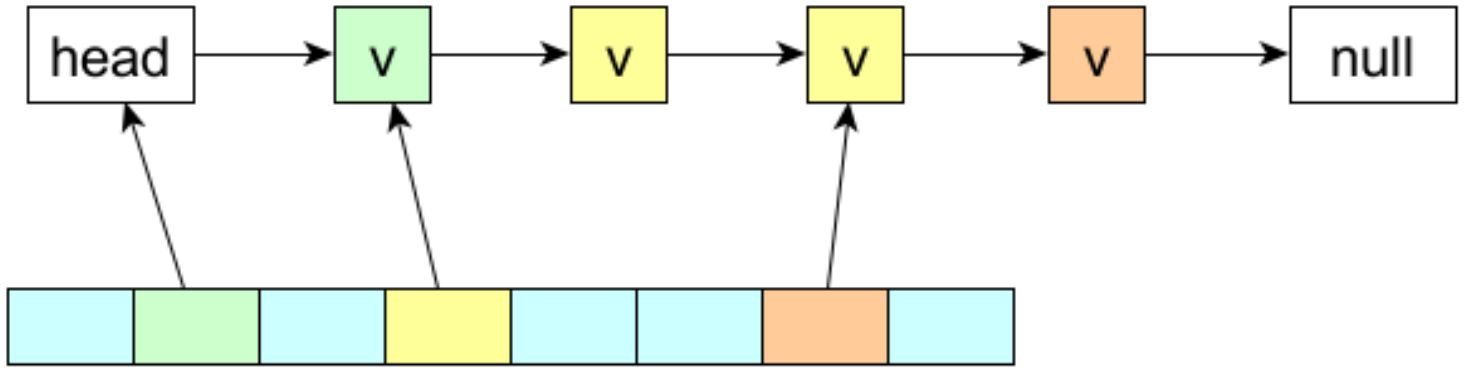
Представление в памяти

- На самом деле в распространённых реализациях (libstdc++, etc) таблица представлена списком элементов, каждый из которых хранит свой хеш и вектором указателей на начало блока
- Стандарт устроен так, что это практически единственный способ выполнить все его ограничения



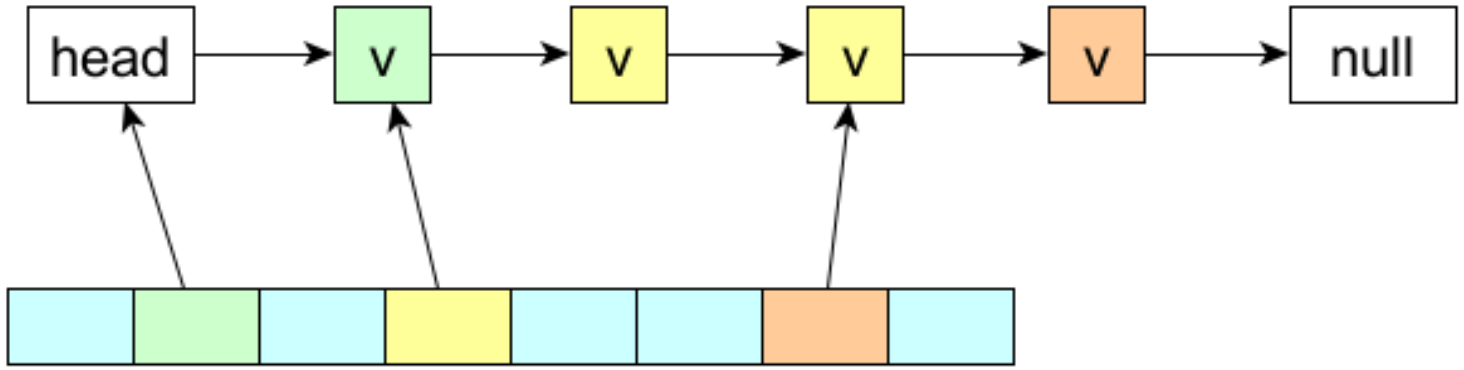
Обсуждение: отказ от хранения

- Идея для оптимизации это отказ от хранения.
- Вместо того, чтобы хранить хеш, мы вычисляем хеш каждый раз когда смотрим бакет.
- Что вы думаете про эту оптимизацию?



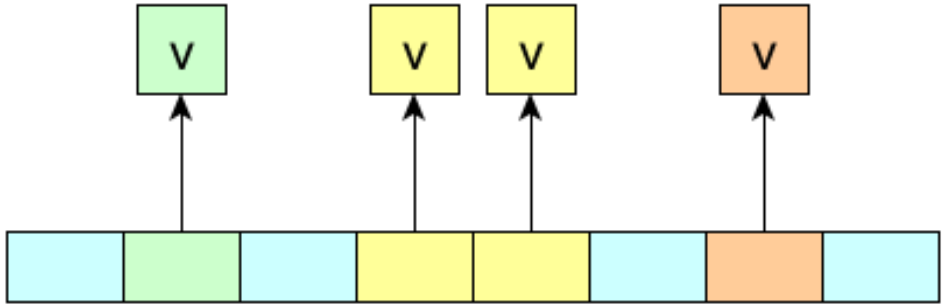
Гарантии по итераторам

- Так как unordered map это по сути список, гарантии по итераторам для него **как для списка**. И даже для рехеша.
- Не можем ли мы улучшить наше отображение, убрав строгие гарантии по итераторам?



Первая идея: node_map

- Мы можем отказаться от хранения указателя в списке бакетов.
- Это лишает нас гарантий по итераторам при рехеше и ставит нас перед лицом внезапных реаллокаций.
- Кроме того мы усложняем (фактически теряем) итерацию по бакетам.
- Кстати, как бы вы организовали быстрый переход к началу бакета при таком подходе?
- Этот контейнер довольно популярен в библиотеке Abseil от Google.



https://abseil.io/docs/cpp/guides/container

Интермедия: алгоритмический базис

- Таблицы в которых мы точно не знаем по хешу номер бакета называются таблицами с открытой адресацией (в противоположность прямой адресации)
- При открытой адресации используется probing (исследование) ячеек.

$$h(x) = (h'(x) + i) \; mod \; m$$

- Здесь функция может быть линейной по i, квадратичной или даже более сложной (см. двойное хеширование).

$$h(x) = \left(h'(x) + ih''(x)\right) mod \; m$$

- В принципе именно открытая адресация подсказывает нам следующую идею.

Robin Hood hashing

Вторая идея: flat map

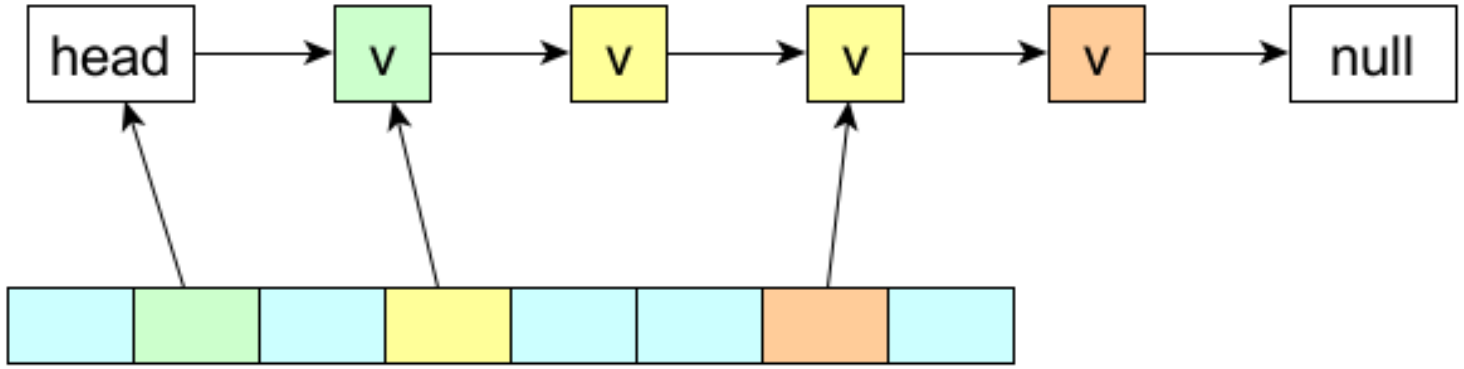
- Мы можем в принципе хранить всё как один вектор

- Да мы теряем все гарантии по итераторам и всё такое.
- Но мы приобретаем потрясающую локальность кешей и работать с этим практически также приятно, как с векторами.

benchmarks

Обсуждение

- Многие критикуют unordered контейнеры за то, что стандарт заперт ограничениями, позволяющими только неэффективную реализацию, максимум с пробингом.
- С другой стороны в **стандартной** библиотеке должно быть нечто, удобное всем. Для прочего есть abseil и folly.



Загадочные квадратные скобки

- Поскольку ассоциативный массив это массив, для него сделали удобное массиво-подобное обращение:

```
std::unordered_map<int, int> m = {{1, 20}, {100, 30}};
auto& x = m[100];
```

- Это эквивалентно вот чему:

```
auto p = m.emplace(100, int{});
auto it = p.first; auto b = p.second;
if (!b) it = m.find(100);
auto& x = it->second;
```

- Тут сразу видно два ограничения: оператор квадратные скобки не константный и у ключа должен быть конструктор по умолчанию.

Кстати о квадратных скобках

- Поскольку ассоциативный массив это массив, для него сделали удобное массиво-подобное обращение:

```
std::unordered_map<int, int> m = {{1, 20}, {100, 30}};
auto& x = m[100];
```

- Также можно использовать особый синтаксис auto, развязывающий пару

```
auto [it, b] = m.emplace(100, int{});
if (!b) it = m.find(100);
auto& x = it->second;
```

- Он называется structured binding.

https://godbolt.org/z/hfMe5b6rd

Неупорядоченные множества

- Особый вид `unordered_map` который хранит только ключи называется `unordered_set`.
- Вы можете рассматривать unordered_set как массив с дешевым поиском из уникальных элементов.

```
std::unordered_set s = {1, 2, 2, 2, 1}; // = {1, 2}
```

- Поддержка инварианта уникальности и поиска (в случае вектора нужна сортированность) дешевле, чем для вектора.

Case study: орбита в группе

- Группой называется множество элементов с групповой операцией над ними
- Например группа $\{Z_7, \times\}$ это числа $\{1 \dots 6\}$ с операцией умножения mod 7
- Зададимся генерирующими элементами группы, например $\{3, 5\}$
- Тогда у любого элемента будет **орбита**: все элементы которые можно получить умножая его на генераторы, умножая получившиеся результаты на генераторы и т.д.
- Естественный контейнер для хранения орбиты это `unordered_set` т.к. вектор при вставке придётся пересортировывать и удалять дубликаты.

https://godbolt.org/z/5W6dG564s

Обсуждение

- Чем `unordered_set` **хуже**, чем сортированный массив?

Обсуждение

- Чем `unordered_set` **хуже**, чем сортированный массив?
  - Оно не позволяет range-based queries
  - Оно не хранит повторные элементы
- Второе решается с помощью мультиконтейнера `unordered_multiset`
- К слову, видите ли вы применения для `unordered_multimap`?

multimap vs map of vector

- Последовательные контейнеры
- Контейнеро-подобные классы
- Ассоциативные контейнеры
- Упорядоченные контейнеры

Уникальность элементов

- Упорядоченное множество также хранит уникальные элементы.

```
std::set<int> s = {67, 42, 141, 23, 42, 106, 15, 50};
for (auto elt : s) cout << elt << endl;
```

- Ничего не сломается, но на экране будет.

```
15, 23, 42, 50, 67, 106, 141
```

- Главное отличие от unordered_set: оно хранит их именно что упорядоченно.
- Это позволяет range-based queries через upper и lower bound.

Порядок сравнения

- Множество создаёт упорядочение своих элементов

```
std::set<int> s = {67, 42, 141, 23, 42, 106, 15, 50};

auto itb = s.lower_bound(30);
auto ite = s.upper_bound(100);
```

- Теперь можно итерировать в интервале `[30, 100)` не зависимо от того есть ли в множестве в точности такие элементы

```
for (auto it = itb; it != ite; ++it)
  std::cout << *it << std::endl;
```

- Что на экране?

https://godbolt.org/z/cbr9ovavv

Порядок сравнения

- Можно задать любой предикат упорядочения

```
std::set<int, std::greater<int>> s = {
  67, 42, 141, 23, 42, 106, 15, 50};

auto itb = s.lower_bound(30);
auto ite = s.upper_bound(100);
```

- Задают ли итераторы `itb` и `ite` валидный интервал для итерирования?
- Что будет, например при таком цикле?

```
for (auto it = itb; it != ite; ++it)
  std::cout << *it << std::endl;
```


https://godbolt.org/z/coeMeaqnb

Порядок сравнения

- Можно задать любой предикат упорядочения

```
std::set<int, std::greater<int>> s = {
  67, 42, 141, 23, 42, 106, 15, 50};

auto itb = s.lower_bound(100);
auto ite = s.upper_bound(30);
```

- На прошлом слайде интервал был невалиден. Исправления подсвечены.
- Теперь всё хорошо, но это крайне контринтуитивно

```
for (auto it = itb; it != ite; ++it)
  std::cout << *it << std::endl;
```

Порядок сравнения

- Что если теперь упорядочить по (<=)

```
std::set<int, std::less_equal<int>> s = {
  67, 42, 141, 23, 42, 106, 15, 50};

auto itb = s.lower_bound(30);
auto ite = s.upper_bound(100);
```

- Тот же вопрос: валиден ли диапазон?

```
for (auto it = itb; it != ite; ++it)
  std::cout << *it << std::endl;
```

Порядок сравнения

- Что если теперь упорядочить по (<=)

```
std::set<int, std::less_equal<int>> s = {
  67, 42, 141, 23, 42, 106, 15, 50};

auto itb = s.lower_bound(30);
auto ite = s.upper_bound(100);
```

- Тот же вопрос: валиден ли диапазон?

```
for (auto it = itb; it != ite; ++it)
  std::cout << *it << std::endl;
```

- Это нарушает инвариант контейнера и последствия сложно предсказать.

https://godbolt.org/z/96dTnafT6

Требования к предикату сравнения

- Общая концепция называется strict weak ordering.
- Она включает:
  - Антисимметричность: $pred(x, y) \Rightarrow \neg pred(y, x)$
  - Транзитивность: $pred(x, y) \wedge pred(y, z) \Rightarrow pred(x, z)$
  - Иррефлексивность: $\neg pred(x, x)$
  - Транзитивность эквивалентности:
    $eq(x, y) \equiv \neg\, pred(x, y) \wedge \neg\, pred(y, x) \vdash eq(x, y) \wedge eq(y, z) \Rightarrow eq(x, z)$
- Она же распространяется на предикаты в алгоритмах сортировки и т.д.
- Математическая разминка: пусть $(a + ib < c + id) \Leftrightarrow (a < c) \wedge (b > d)$
  является ли это strict weak ordering для комплексных чисел?

https://godbolt.org/z/qhj8q7cMY

Обсуждение

- Наверное в multiset, где возможны одинаковые элементы такие же требования к предикату сравнения (а они там тоже действуют) введены зря?

Контрпример Майерса

- Наверное в multiset, где возможны одинаковые элементы такие же требования к предикату сравнения (а они там тоже действуют) введены зря?
- Нет не зря. Майерс сделал интересное наблюдение.

```
std::multiset<int, less_equal<int>> s;
s.insert(10); // insert 10A
s.insert(10); // insert 10B
```

- Теперь `equal_range` для `10` вернёт пустой интервал, что, очевидно, абсурдно.
- Общий вывод: strict weak ordering это очень важная концепция.

Обсуждение: удаление

- Контейнер std::map упорядочен по ключам, но не по значениям.
- Предположим мы хотим удалить из отображения все пары ключ-значение в некоем диапазоне значений.
- Мы вряд ли сможем сделать нечто лучше, чем нечто вроде:

```
for (auto it = s.begin(); it != s.end(); ++it)
   if (it->second < max && it->second > min)
     s.erase(it);
```

- Что тут не так?

Не стреляйте себе в ногу через erase

- Это очень плохая идея

```
for (auto it = s.begin(); it != s.end(); ++it)
  if (it->second < max && it->second > min)
    s.erase(it); // тут итератор стал невалидным
```

- В рамках C++98 это делалось вот так:

```
for (auto it = s.begin(); it != s.end();)
  if (it->second < max && it->second > min)
    s.erase(it++);
  else
    ++it;
```

Не стреляйте себе в ногу через erase

- Это очень плохая идея

```
for (auto it = s.begin(); it != s.end(); ++it)
  if (it->second < max && it->second > min)
    s.erase(it); // тут итератор стал невалидным
```

- В рамках C++11 это делается вот так:

```
for (auto it = s.begin(); it != s.end();)
  if (it->second < max && it->second > min)
    it = s.erase(it);
  else
    ++it;
```

Обсуждение

- Предложите решение для замены элемента в множестве

```
auto it = s.find(1);
if (it != s.end())
  *it = 3; // error: assignment of read-only location
```

- Пусть вам всё таки нужно заменить элемент 1 на 3. Что тогда?

Обсуждение

- Предложите решение для замены элемента в множестве

```
auto it = s.find(1);
if (it != s.end())
  *it = 3; // error: assignment of read-only location
```

- Пусть вам всё таки нужно заменить элемент 1 на 3. Что тогда?
- Теперь решение очевидно:

```
auto it = s.find(1);

if (it != s.end()) {
  s.erase(it); s.insert(3);
}
```

Литература

- ISO/IEC, "Information technology -- Programming languages – C++", ISO/IEC 14882: 2017
- Bjarne Stroustrup, 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
- Scott Meyers, Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14, 2012
- Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step", CppCon'2017
- Bindal A., Narang P., Indu S., Map vs. Unordered Map: An Analysis on Large Datasets, International Journal of Computer Applications, Volume 127, №2, oct'2015
- boost::flat_map and its performance compared to map and unordered_map, StackOverflow, https://stackoverflow.com/questions/21166675/boostflat-map-and-its-performance-compared-to-map-and-unordered-map