ЛЕКЦИЯ 10
КОНТЕЙНЕРЫ. ЧАСТЬ 2

АЛГОРИТМИЗАЦИЯ И
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

СМЫСЛ АССОЦИАТИВНОСТИ

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

```
std::vector<T> v; // int -> T
```

- Как сделать правильное отображение $T \to U$?

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
  - hash функция
- При этом ключи уникальны и мы можем менять значения, но не ключи

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:
  - определить равенство (все ли помнят как это сделать?)
  - определить хэш. Есть ли идеи, как именно это можно сделать?

- Обратите внимание: мы можем **добавлять в стандартную библиотеку свои специализации, но не перегрузки**

СОБСТВЕННЫЙ ХЭШ

- 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 случае это всегда угадайка

СОБСТВЕННЫЙ ХЭШ

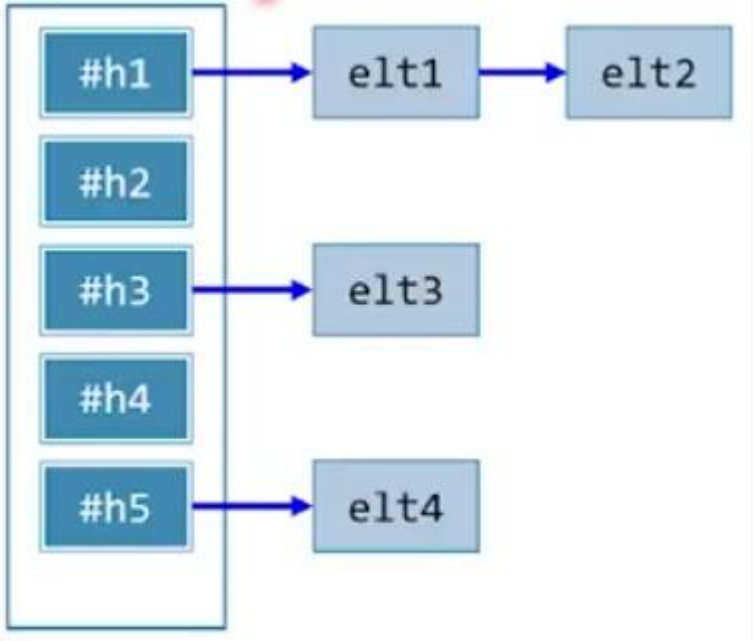
- Если угадайка не привлекает, то есть boost

```
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 и его всегда придется затаскивать в проект

ПРЕДСТАВЛЕНИЕ В ПАМЯТИ*

- О хэш-таблицах можно думать, как о массиве корзин (buckets), каждая из которых содержит элементы с одинаковым хэшом.



Здесь выполняются условия

```
hash(elt1) == hash(elt2);
elt1 != elt2;
hash(elt1) != hash(elt3);
```

НИЗКОУРОВНЕВАЯ ИНФОРМАЦИЯ

- Дополнительно каждый неупорядоченный контейнер дает возможность смотреть его статистику
- `bucket_count()` – количество корзин
- `max_bucket_count()` – максимальное количество корзин без реаллокации
- `bucket_size(n)` – размер корзины с номером `n`
- `bucket(Key)` – номер корзины для ключа `Key`
- `load_factor()` – среднее количество ключей в корзине
- `max_load_factor()` – максимальное количество ключей в корзине

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
```

ДВА ВИДА ИТЕРАЦИИ

- По хеш-таблице можно итерировать как по единому целому

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

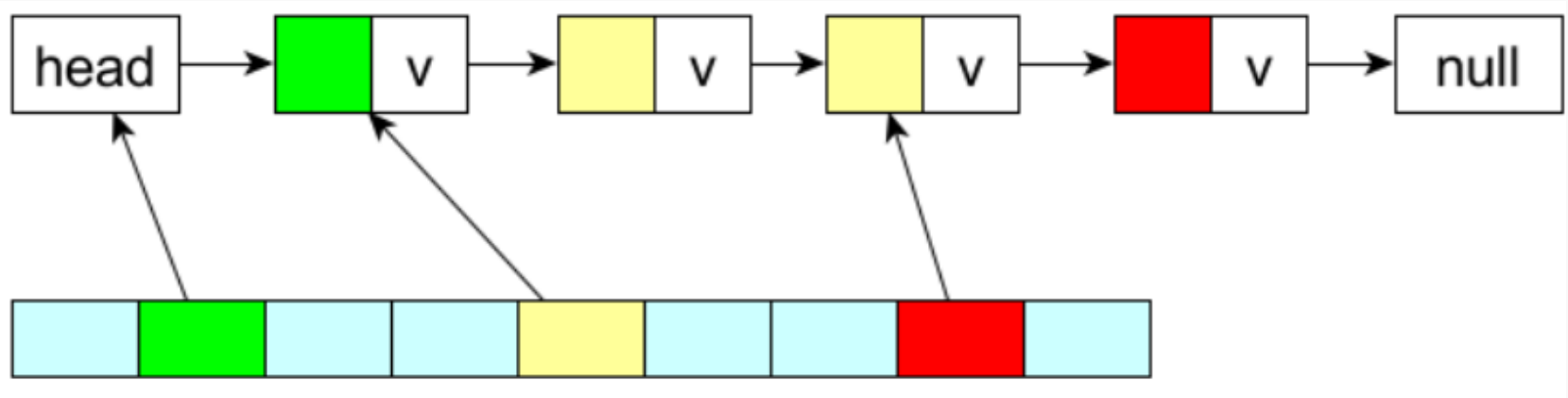
- Можно итерироваться внутри бакета, указав его номер

```
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 адаптер чтобы позволить второй вариант через range based for?

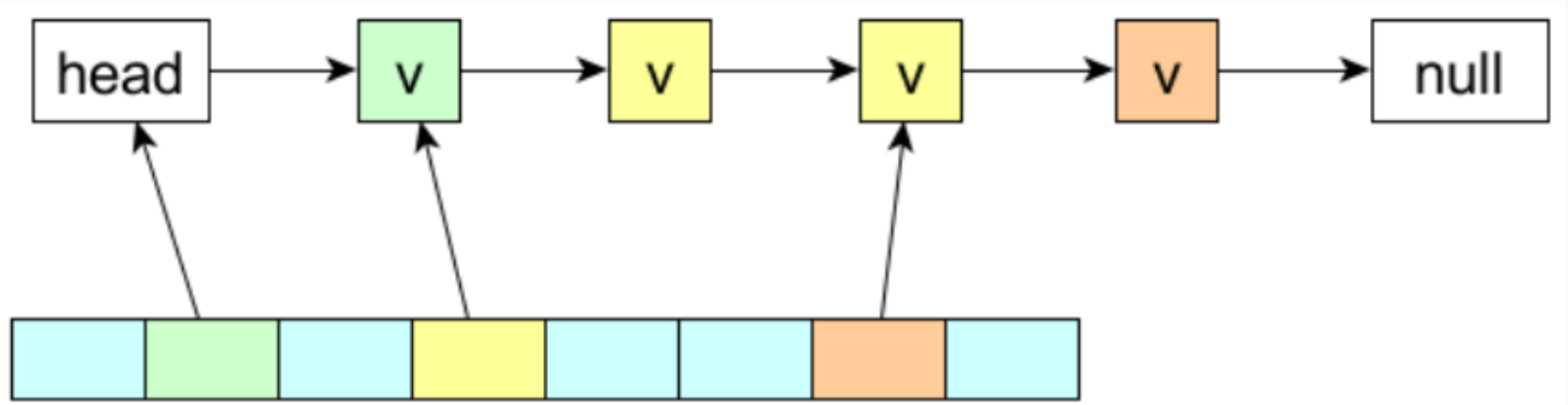
ПРЕДСТАВЛЕНИЕ В ПАМЯТИ

- 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



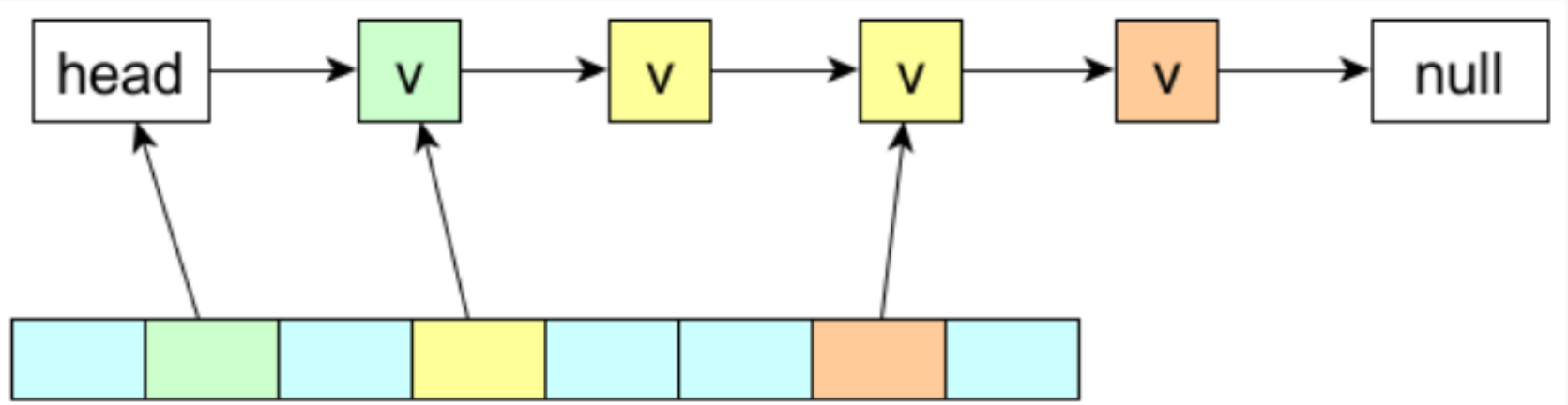
ОБСУЖДЕНИЕ: ОТКАЗ ОТ ХРАНЕНИЯ

- 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?



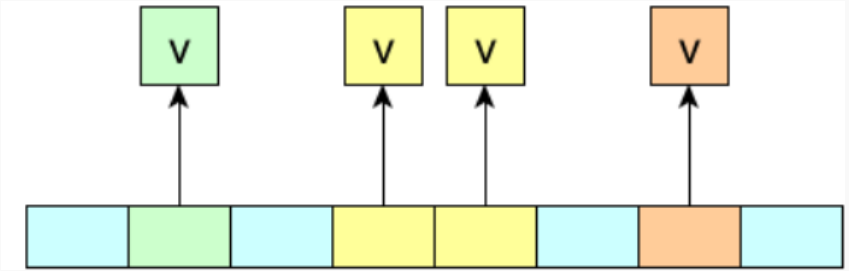
ГАРАНТИИ ПО ИТЕРАТОРАМ

- 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.



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.

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

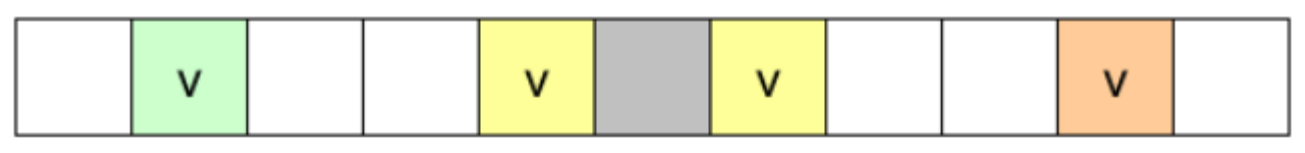
- Здесь функция может быть линейной по i, квадратичной или даже более сложной (см. двойное хеширование).

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

- В принципе именно открытая адресация подсказывает нам следующую идею.

ВТОРАЯ ИДЕЯ: FLAT MAP

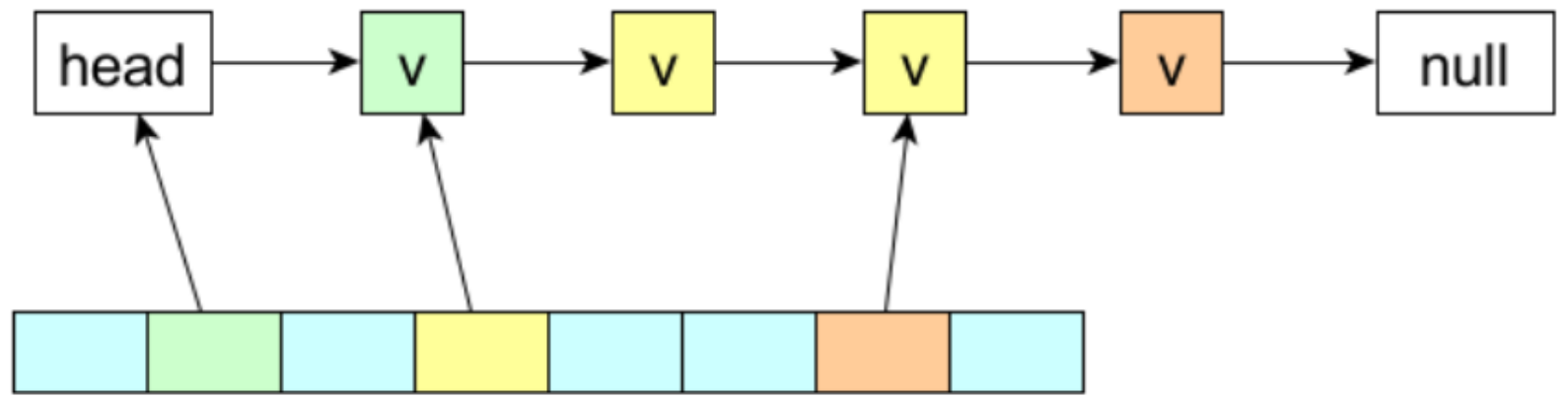
- Мы можем в принципе хранить всё как один вектор



- 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.

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.

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…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.

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`?

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?

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;
```

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.

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?

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);
}
```

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Бьерн Страуструп, Язык программирования C++/ ред. А. Боборыкин. – 4-е изд. - Москва: Издательство БИНОМ, 2023. – 1213 с.
2. 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
3. Nicolai M. Josuttis, The C++ Standard Library - A Tutorial and Reference, 2nd Edition , Addison-Wesley, 2012
4. Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step", CppCon'2017
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.