Обзор STL. Часть 2

Введение в объектно-ориентированное программирование

20.03.2025

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.

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?

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

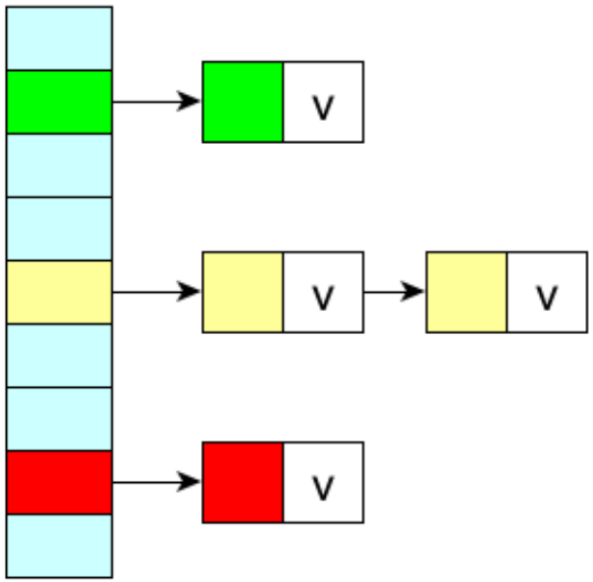
- Если угадайка не привлекает, есть boost

```
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 снизу это 1.33 и в общем это уже довольно плохо.



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

Два вида итерации

- По хеш-таблице можно итерировать как по единому целому

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

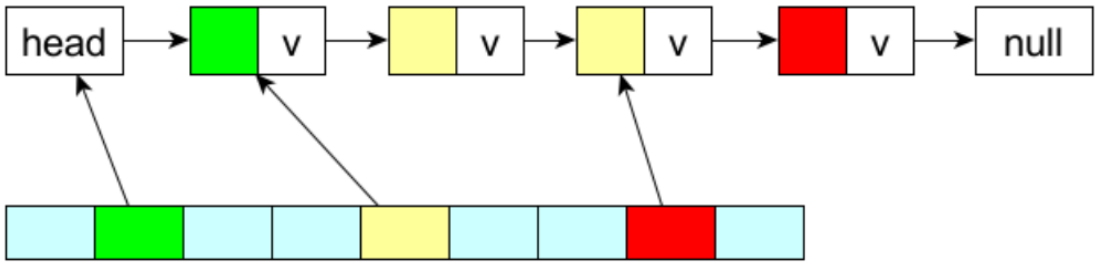
- Можно итерироваться внутри бакета, указав его номер

```
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.

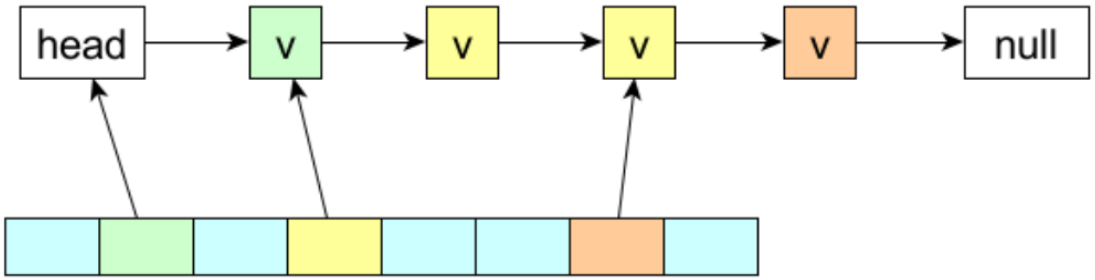
Представление в памяти

• 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



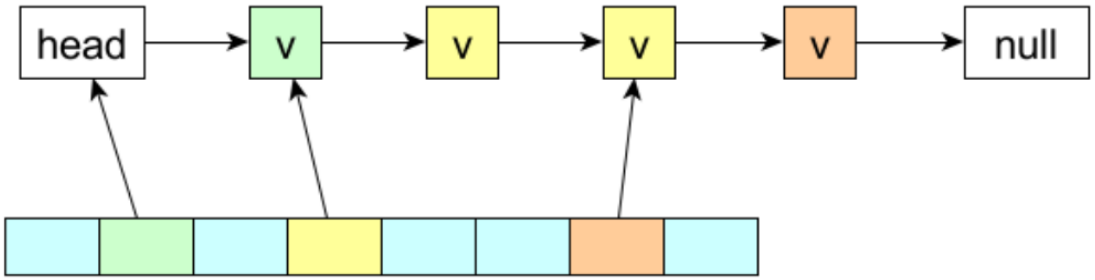
Обсуждение: отказ от хранения

- 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?



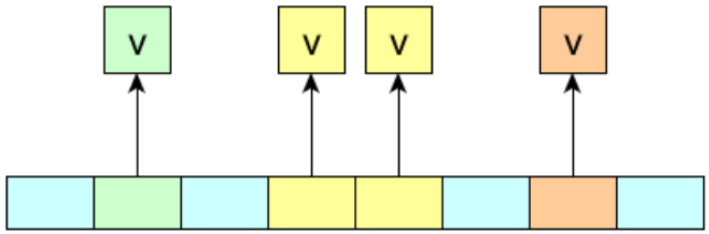
Гарантии по итераторам

• 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.



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.

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.
- 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;
```

- Это нарушает инвариант контейнера и последствия сложно предсказать

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 10 A
s.insert(10); // insert 10 B
• 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);
}
```