SCENT OF C++

Проблемы C. Переход к C++. Механизмы объединения и обобщения.
Демонстрация основных возможностей языка и библиотеки

К. Владимиров, Intel, 2020
mail-to: konstantin.vladimirov@gmail.com

Как зовут преподавателя?

- Владимиров Константин Игоревич
- Телефон: +7-903-842-27-55
- Email: konstantin.vladimirov@gmail.com
- Слайды на sourceforge
- Короткие примеры к слайдам на github
- Основная коммуникация через email

- Немного о кэшах
- Реализация на C и её проблемы
- Реализация на C++
- Бонус: больше кэшей

Проблема из реального мира

- Вы пишете программу, которая оперирует миллионами страниц разного размера, различающихся уникальным номером

```
struct page {
  int index;  // page index: 1, 2, ... n
  int sz;     // page size
  char *data; // page data
};
```

- Единственный способ получить страницу с номером n это получить её по сети довольно медленной функцией

```
void slow_get_page(int n, struct page *p);
```

- Локально у вас есть место всего на несколько тысяч страниц. Что делать?

Решение: кэш

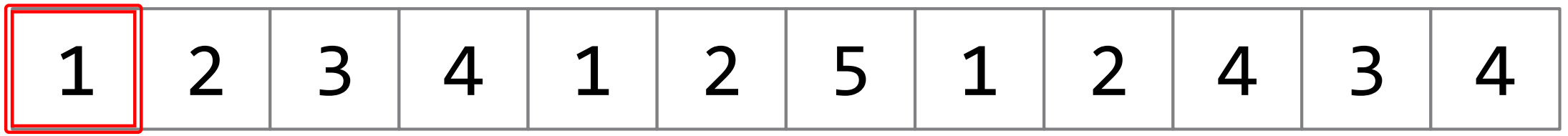
- Если у нас есть немного места, его хотелось бы использовать
- Мы хотим сохранять некоторые страницы, раз уж мы не можем сохранить все
- Конечно мы хотели бы сохранять те страницы, которые чаще используем
- Небольшая область памяти называется кэш, туда можно поместить страницу или оттуда можно вытеснить страницу
- Например у нас есть место на 4 страницы и к нам поступают запросы:

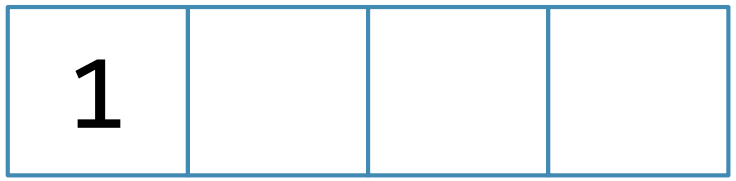
`1, 2, 3, 4, 1, 2, 5, 1, 2, 4, 3, 4`

- Как будет изменяться кэш? Какую **стратегию** кэширования тут можно выбрать?

Стратегия LRU (least recently used)

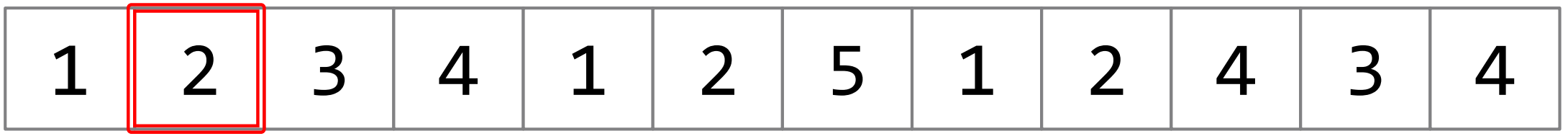
- Если запрошенный элемент найден, он перемещается вперёд
- Если нет, он помещается вперёд а последний вытесняется





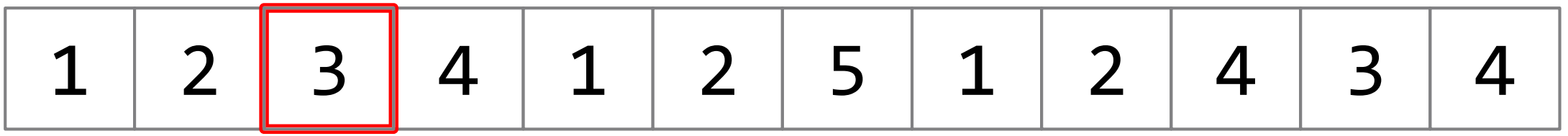
Стратегия LRU (least recently used)

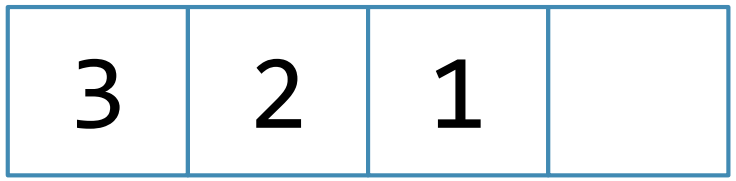
- Если запрошенный элемент найден, он перемещается вперёд
- Если нет, он помещается вперёд а последний вытесняется



Стратегия LRU (least recently used)

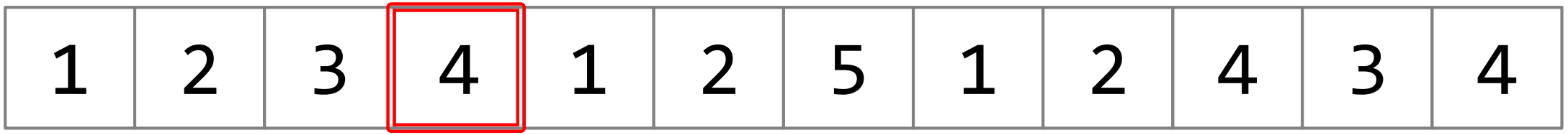
- Если запрошенный элемент найден, он перемещается вперёд
- Если нет, он помещается вперёд а последний вытесняется

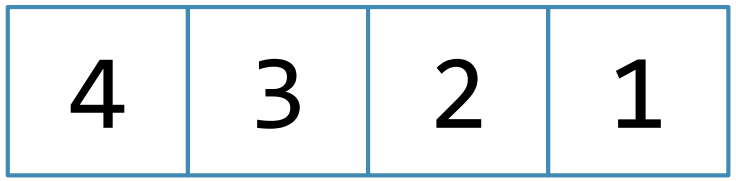




Стратегия LRU (least recently used)

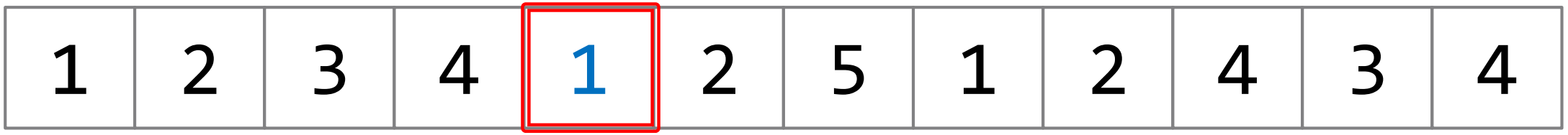
- Если запрошенный элемент найден, он перемещается вперёд
- Если нет, он помещается вперёд а последний вытесняется

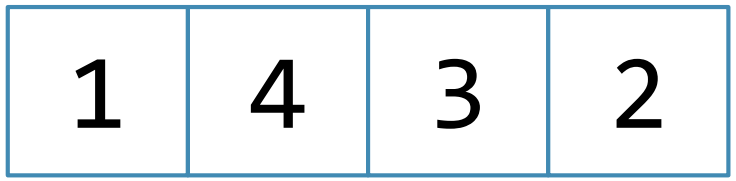




Стратегия LRU (least recently used)

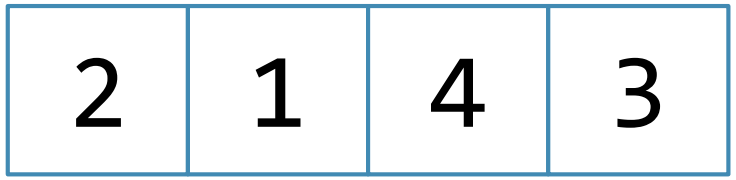
- Если запрошенный элемент найден, он перемещается вперёд
- Если нет, он помещается вперёд а последний вытесняется





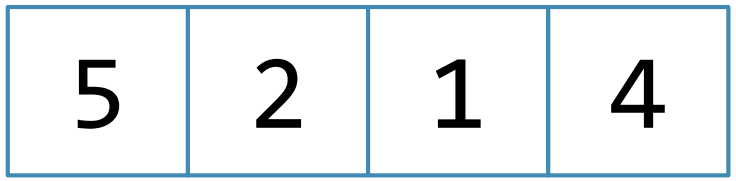
Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд
- Если нет, он помещается вперёд а последний вытесняется



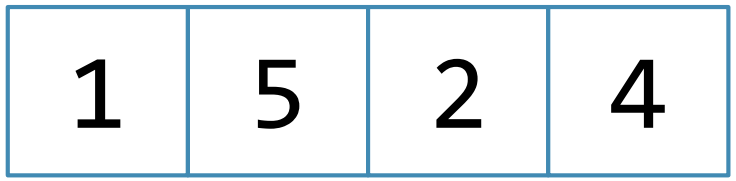
Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд
- Если нет, он помещается вперёд а последний вытесняется



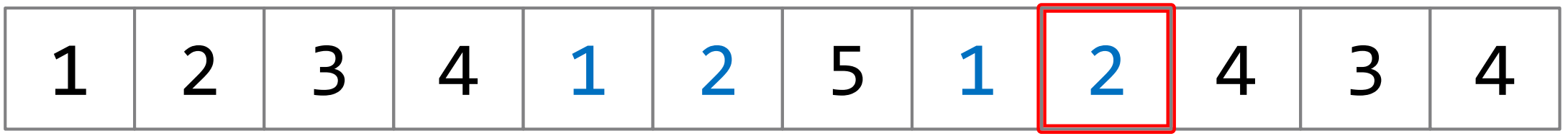
Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд
- Если нет, он помещается вперёд а последний вытесняется



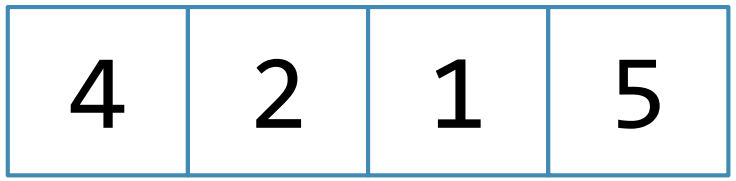
Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд
- Если нет, он помещается вперёд а последний вытесняется



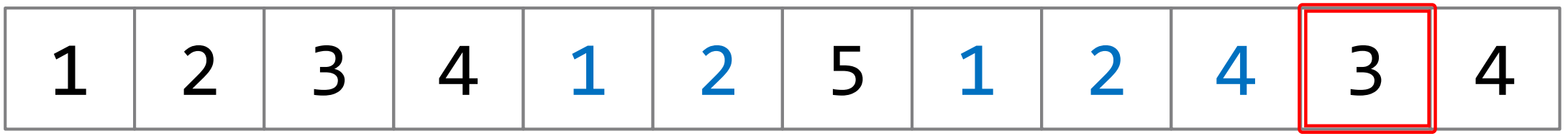
Стратегия LRU (least recently used)

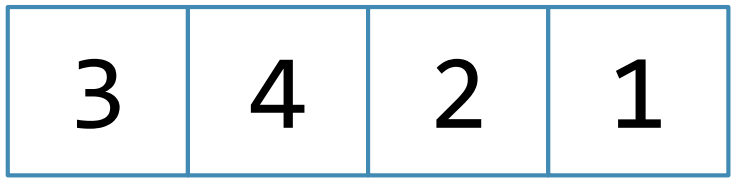
- Если запрошенный элемент найден, он перемещается вперёд
- Если нет, он помещается вперёд а последний вытесняется



Стратегия LRU (least recently used)

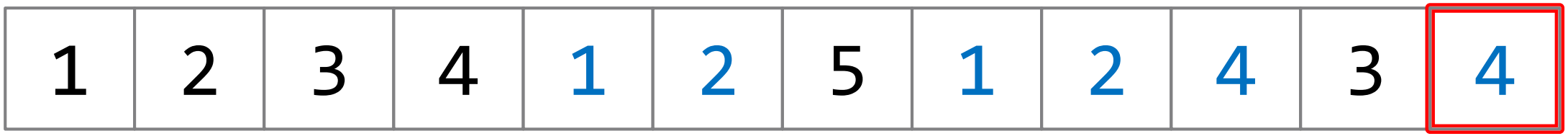
- Если запрошенный элемент найден, он перемещается вперёд
- Если нет, он помещается вперёд а последний вытесняется

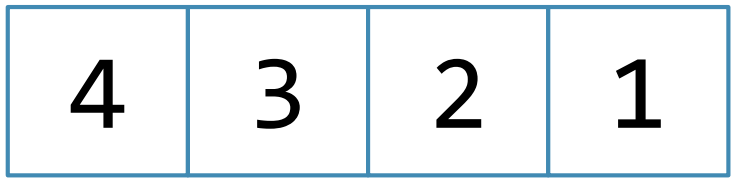




Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд
- Если нет, он помещается вперёд а последний вытесняется



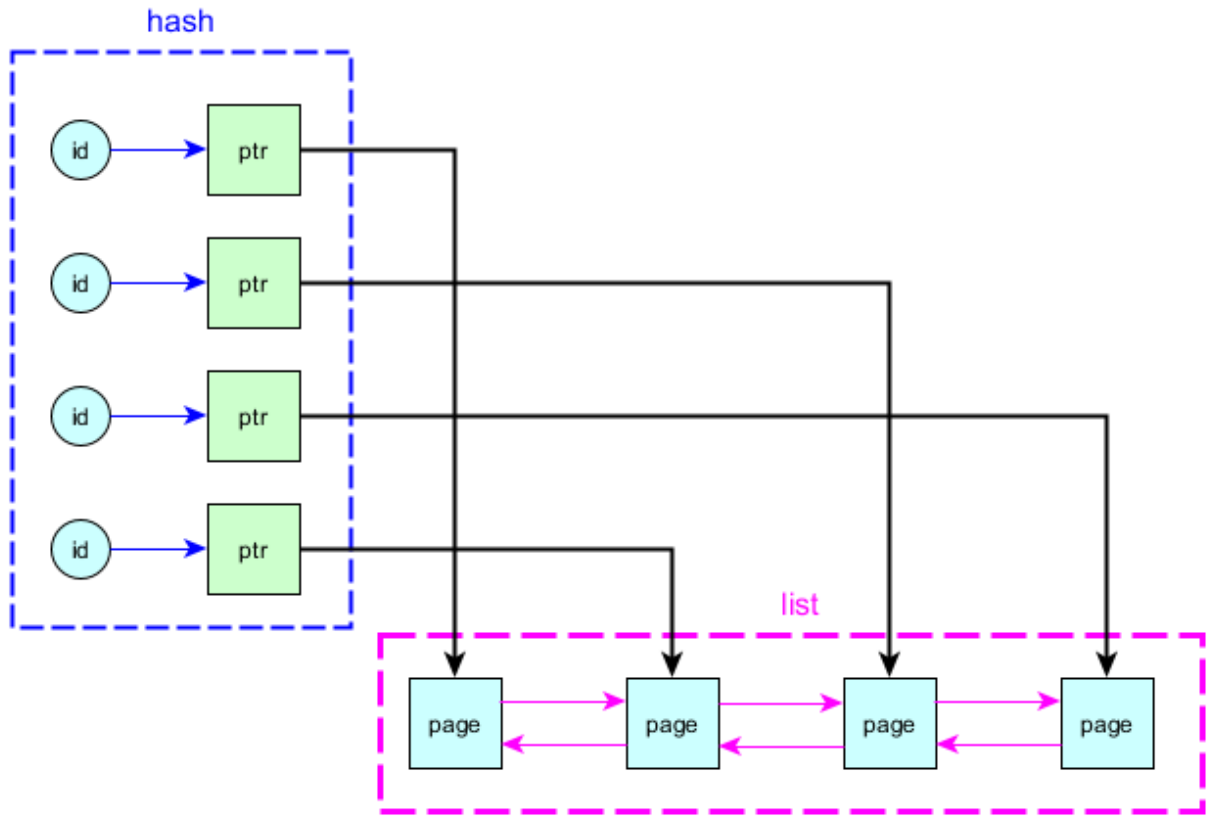


Обсуждение

- Какие структуры данных нам понадобятся чтобы сделать LRU cache?

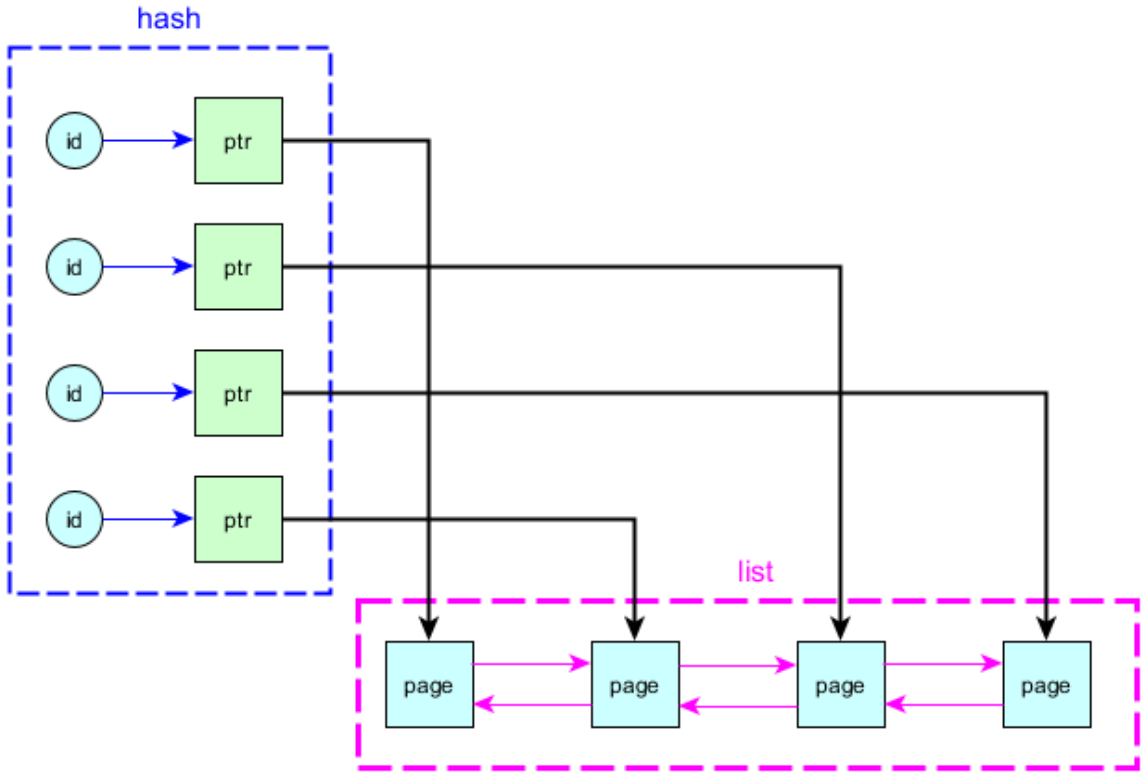
Обсуждение

- Какие структуры данных нам понадобятся чтобы сделать LRU cache?
- Двусвязный список для собственно кэша
- Хеш-таблица для того, чтобы быстро определять кэширован ли элемент



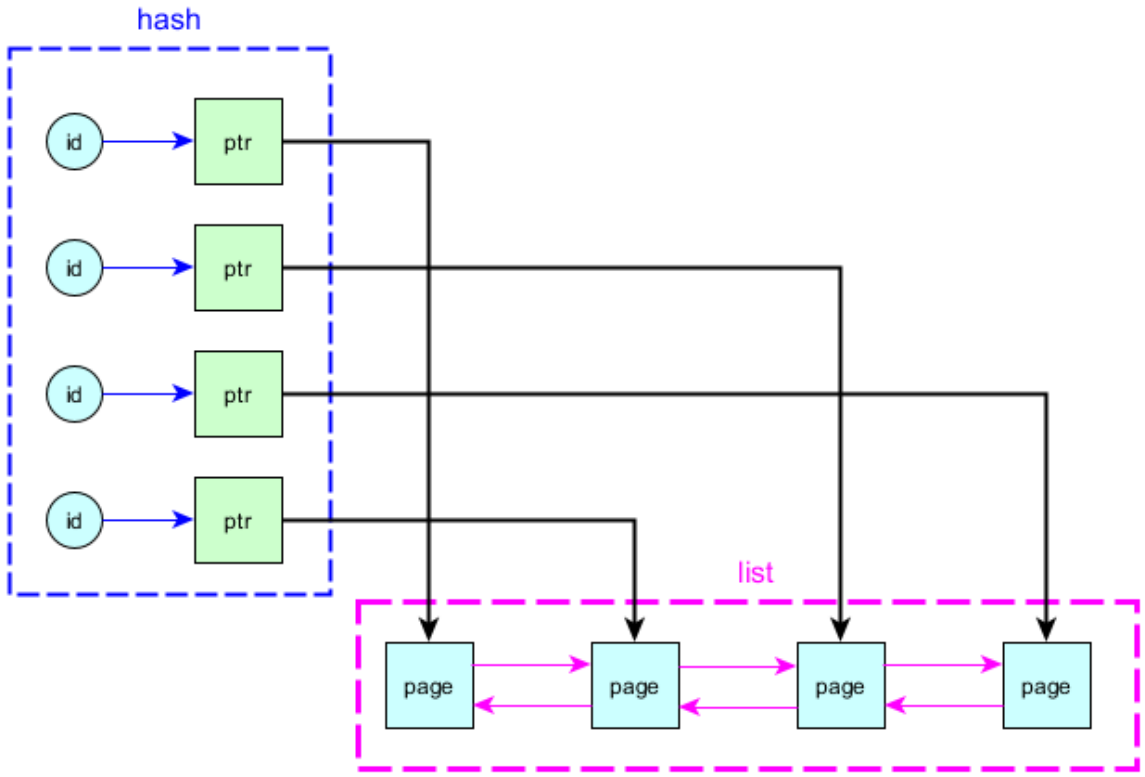
Алгоритм LRU – текстовое описание

1. По запросу на очередную страницу, проверить кэширована ли она. Если да, перейти к пункту 3. Если запросов нет, завершить работу
2. Использовать медленный способ получить страницу. Удалить из списка и стереть из хеш-таблицы последнюю страницу в списке. Добавить в голову списка и в хеш таблицу новую страницу и перейти к пункту 4
3. Переместить узел соответствующий кэшированной странице в голову списка
4. Обработать страницу в голове списка и вернуться к пункту 1



Алгоритм LRU – псевдокод

```
next_page = get_page_number()
while next_page != NO_PAGE do
  page_node = is_cached(next_page)
  if page_node != nil
    list_move(page_node, list_top)
  else
    list_delete(list_bottom)
    page_node = slow_get_page()
    list_add(page_node, list_top)
  end if
  process_page(page_node)
  next_page = get_page_number()
end while
```



Обсуждение

- Вам больше нравится текст или псевдокод?

Обсуждение

- Вам больше нравится текст или псевдокод?
- В разных книгах по разному. Кнут больше любит текст, Кормен больше любит псевдокод
- Я лично больше люблю просто код на C или на C++. Увы, чтобы он был достаточно хорош и читаем, нужно неплохо знать язык поэтому на первых порах и текст и псевдокод почти неизбежны
- В этом курсе алгоритмы будут излагаться по разному: иногда текстом, иногда псевдокодом, иногда просто кодом

- Немного о кэшах
- Реализация на C и её проблемы
- Реализация на C++
- О чем этот курс

Скетч: LRU на языке C

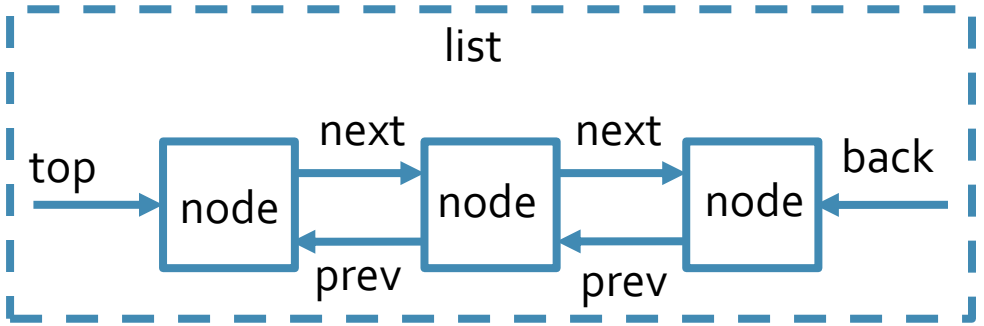
- Вы все можете потренироваться на Problem LC*
- Но это нелёгкая проблема. На языке C вам нужно как минимум:
  - Написать свой двусвязный список
  - Написать к нему тесты
  - Написать свою хеш-таблицу
  - Написать к ней тесты
  - Написать уровень абстракции LRU кэша
  - Написать к нему тесты
- Это три модуля и три заголовочных файла, не считая тестов и в общем работа на троих на сутки

* http://olymp1.vdi.mipt.ru/

LRU на языке C: двусвязный список

```
struct list_node_t {
  struct list_node_t *next;
  struct list_node_t *prev;
  struct page_t data;
};

struct list_t {
  struct list_node_t *top;
  struct list_node_t *back;
};
```



- Какие интерфейсные функции нам могут понадобиться?

LRU на языке C: двусвязный список

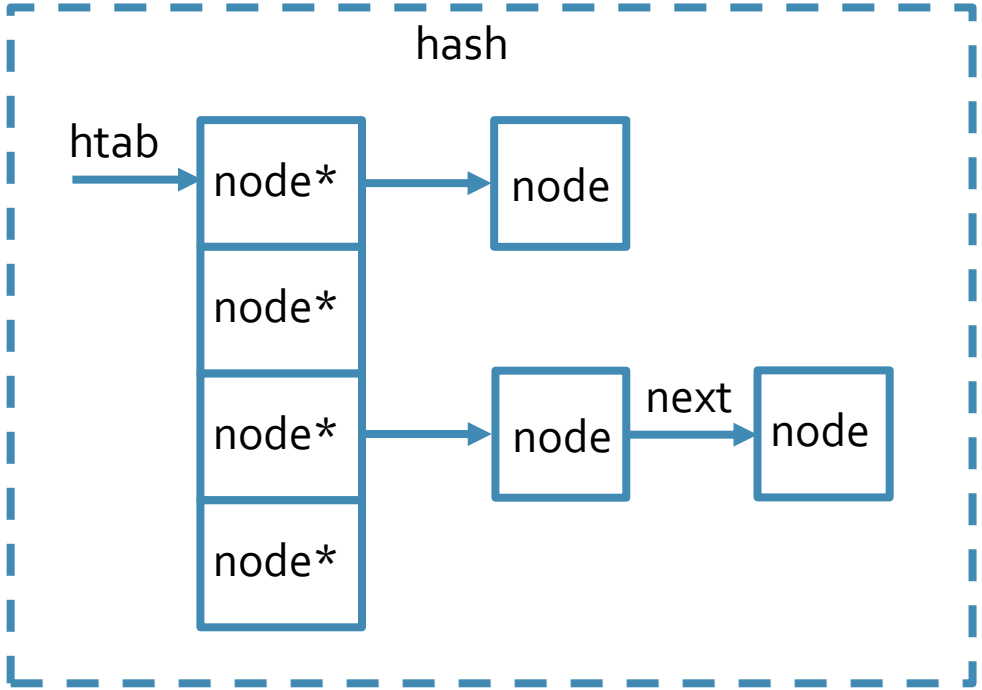
```
struct list_t list_create();
int list_size(const struct list_t *lst);
struct page_t *list_back(struct list_t *lst);
void list_pop_back(struct list_t *lst);
void list_push_front(struct list_t *lst, struct page_t *q);
void list_move_upfront(struct list_t *lst, struct list_node_t *p);
void list_free(struct list_t *lst);
```

LRU на языке C: хеш-таблица

```
struct hashmap_entry_t {
  int key;
  list_node_t *node;
};

struct hashmap_node_t {
  struct hashmap_node_t *next;
  struct hashmap_entry_t entry;
};

struct hash_t {
  struct hashmap_node_t **htab;
  int len;
};
```



LRU на языке C: хеш-таблица

```
// создаёт пустой хеш с заданным количеством бакетов
struct hash_t htable_create(int len);

// ищет узел по ключу
list_node_t *htable_find(struct hash_t *h, int key);

// вставляет пару ключ+значение в хеш
void htable_insert(struct hash_t *h, struct hashmap_entry_t ent);

// стирает пару ключ+значение из хеша по ключу
void htable_erase(struct hash_t *h, int key);

// освобождает хеш и всю использованную память
void htable_free(struct hash_t *h);
```

LRU на языке C: собственно кэш

```
struct cache_t {
  int sz;
  struct hash_t hash;
  struct list_t lst;
};

// создать пустой кэш
struct cache_t cache_create(int cache_size);

// поискать и при необходимости вставить страницу
int cache_lookup(struct cache_t *c, const struct page_t *q);

// освободить всю память
void cache_free(struct cache_t *c);
```

LRU на языке C: собственно кэш

```
int cache_lookup(struct cache_t *c, const struct page_t *query) {
  struct list_node_t *pnode = htable_find(c->hash, query->id);

  if (pnode != nullptr) {
    struct hashmap_entry_t newent;
    if (list_size(c->lst) == c->sz) {
      int backid = list_back(c->lst)->id;
      htable_erase(c->hash, backid);
      list_pop_back(c->lst);
    }
    list_push_front(c->lst, query);
    newent.key = query->id;
    newent.node = list_begin(&c->lst);
    htable_insert(&c->hash, newent);
    return false;
  }
  list_move_upfront(&c->lst, pnode);
  return true;
}
```

Программа-драйвер для Problem LC

```
cache = cache_create(cache_size);

for (int i = 0; i < num_queries; ++i) {
  struct page_t query;
  res = scanf("%d", &query.id);

  if (cache_lookup(&cache, &query))
    hits += 1;
}

cache_free(&cache);
```

Обсуждение

- Хороша ли получившаяся программа?

Обсуждение

- Хороша ли получившаяся программа?
- Нет, она довольно плоха
- Она слишком сложна и явно недостаточно оттестирована
- В ней есть тяжёлые и неприятные части (например хеш-таблица довольно наивная и скорее всего не потянет серьёзные испытания на прочность)
- В ней приходится постоянно следить за памятью, например простой способ организовать утечку это забыть вызвать одну функцию
- Ну и наконец… а что если нам также надо кешировать **другой тип страниц**?

Обсуждение

- Язык C является мощным и совершенным инструментом
- Но по своей природе он слишком низкоуровневый
- Он исключительно хорош, чтобы писать код в пределах нескольких функций
- Но он практически непригоден, чтобы строить надёжные переиспользуемые абстракции
- Он для этого непригоден до такой степени, что у него даже нет нормальных структур данных в стандартной библиотеке. Каждая большая программа на C реализует свой динамический массив (часто не один)

- Немного о кэшах
- Реализация на C и её проблемы
- Реализация на C++
- Бонус: больше кэшей

C++: inception

- *Bjarne Stroustrup* разработал C++ в 1983 году во время работы в AT&T Bell Labs для облегчения работы над крупномасштабными задачами
- Изначально C++ компилировался в язык C и первый компилятор назывался Cfront
- До сих пор общее подмножество C и C++ это почти весь C
- Прежде, чем написать LRU cache на C++ нам потребуется ввести несколько новых концепций

Объединение данных и методов

- Одной из главных особенностей C++ является объединение данных и методов их обработки

```
struct Point { double x, y; };

struct Triangle { struct Point pts[3]; }; // C way

double square(const struct Triangle *pt) {
  double sq = pt->pts[0].x * (pt->pts[1].y - pt->pts[2].y) +
              pt->pts[1].x * (pt->pts[2].y - pt->pts[0].y) +
              pt->pts[2].x * (pt->pts[0].y - pt->pts[1].y);
  return abs(sq) / 2.0;
}
```

Объединение данных и методов

- Одной из главных особенностей C++ является объединение данных и методов их обработки

```
struct Point { double x, y; };

struct Triangle { // C++ way
  struct Point pts[3];

  double square() {
    double sq = pts[0].x * (pts[1].y - pts[2].y) +
                pts[1].x * (pts[2].y - pts[0].y) +
                pts[2].x * (pts[0].y - pts[1].y);
    return abs(sq) / 2.0;
  }
};
```


01-basics/triangle.cc

Объединение данных и методов

- Не обязательно определять функцию внутри

```
struct Triangle {
  Point pts[3];
  double square() const; // const так как мы не меняем полей класса
};

double Triangle::square() const {
  double sq = pts[0].x * (pts[1].y - pts[2].y) +
              pts[1].x * (pts[2].y - pts[0].y) +
              pts[2].x * (pts[0].y - pts[1].y);
  return abs(sq) / 2.0;
}
```

Объединение данных и методов

- Одной из главных особенностей C++ является объединение данных и методов их обработки
- Использование получившегося кода

```
Triangle t; // объект типа Triangle

t.pts[0] = Point {1.0, 1.0}; // запись поля
t.pts[1] = Point {3.0, 3.0};
t.pts[2] = Point {1.0, 2.0};

double a = t.square(); // вызов метода
```

Таким образом с методами занесёнными внутрь структуры, мы работаем так же как и с полями через точку либо через стрелочку.

Обсуждение: this

- Хорошо спроектированная структура данных на C часто также берёт "указатель на себя" первым параметром
- Делая его неявным, мы как бы говорим "сделай для себя"

```
double square(const struct Triangle *pt) {
  double sq = pt->pts[0].x * (pt->pts[1].y - pt->pts[2].y) +
              pt->pts[1].x * (pt->pts[2].y - pt->pts[0].y) +
              pt->pts[2].x * (pt->pts[0].y - pt->pts[1].y);
  return abs(sq) / 2.0;
}
```

Обсуждение: this

- Хорошо спроектированная структура данных на C часто также берёт "указатель на себя" первым параметром
- Делая его неявным, мы как бы говорим "сделай для себя"

```
double Triangle::square() const {
  double sq = this->pts[0].x * (this->pts[1].y - this->pts[2].y) +
              this->pts[1].x * (this->pts[2].y - this->pts[0].y) +
              this->pts[2].x * (this->pts[0].y - this->pts[1].y);
  return abs(sq) / 2.0;
}
```

- Указывать явный this иногда необходимо. Но здесь это сделано без необходимости и это дурной тон

Обсуждение: this

- Хорошо спроектированная структура данных на C часто также берёт "указатель на себя" первым параметром
- Делая его неявным, мы как бы говорим "сделай для себя"

```
double Triangle::square() const {
  double sq = pts[0].x * (pts[1].y - pts[2].y) +
              pts[1].x * (pts[2].y - pts[0].y) +
              pts[2].x * (pts[0].y - pts[1].y);
  return abs(sq) / 2.0;
}
```

- Здесь мы не пишем this, но подразумеваем его

```
Triangle t; t.square(); // this == &t
```


01-basics/this.cc

Обобщение данных и методов

- Ещё одна важная концепция это обобщение через механизм шаблонов
- Конкретный тип: точка из двух целых координат

```
struct Point_int { int x, y; };
Point_int p;
```

- Обобщённый тип: точка из двух любых координат

```
template <typename T> struct Point { T x, y; };
Point<int> pi;
Point<double> pd;
```

Обобщение данных и методов

- Тот же треугольник можно обобщить на любые типы точек

```
template <typename T> struct Point { T x, y; };

template <typename U> struct Triangle {
  Point<U> pts[3]; //  U будет подставлено как T в Point

  //  почему я изменил интерфейс на double_square?
  U double_square() {
    U sq = pts[0].x * (pts[1].y - pts[2].y) +
           pts[1].x * (pts[2].y - pts[0].y) +
           pts[2].x * (pts[0].y - pts[1].y);
    return (sq > 0) ? sq : -sq; //  почему больше не abs?
  }
};
```


01-basics/triangle-common.cc

Обсуждение

- С одной стороны обобщение создаёт возможности

```
Triangle<double> t;
Triangle<float> tf;
```

- С другой стороны оно создаёт проблемы
- Это очень часто ходит рука об руку

Обобщение функций

- Пожалуй едиснтвенным способом написать на C максимум двух чисел является макрос

```
#define MAX(x, y) (((x) > (y)) ? (x) : (y))
```

- Перечислите все проблемы в этом макросе

Обобщение функций

- Пожалуй едиснтвенным способом написать на C максимум двух чисел является макрос

```
#define MAX(x, y) (((x) > (y)) ? (x) : (y))
```

- Перечислите все проблемы в этом макросе
- На C++ шаблон функции лишён этих проблем

```
template <typename T> T max(T x, T y) {
  return (x > y) ? x : y;
}
```

Обобщение вместо `void*`

- Стандартная функция из библиотеки C

```
void qsort (void* base, size_t num, size_t size,
            int (*compar)(const void*,const void*));
```

- Что можно с ней сделать используя шаблоны?

Обобщение вместо `void*`

- Первая итерация

```
template <typename T, typename Comp>
void qsortpp (T* base, size_t num, Comp compare);
```

- Вместо передачи указателя и длины, можно передавать два указателя на начало и конец интервала

Обобщение вместо `void*`

- Вторая итерация

```
template <typename T, typename Comp>
void qsortpp (T* start, T* fin, Comp compare);
```

- Вместо указателей можно использовать указателе-подобные объекты, так называемые итераторы и получить

```
template <typename It, typename Comp>
void sort (It start, It fin, Comp compare);
```

Обсуждение

- Что будет работать быстрее?

```
void qsort (void* base, size_t num, size_t size,
            int (*compar)(const void*,const void*));
```

или

```
template <typename T, typename Comp>
void qsortpp (T* base, size_t num, Comp compare);
```

И почему?

- Есть ли проблемы у C++-style сортировки?

01-basics/qsortbench.c

01-basics/qsortbench.cc

Стандартная библиотека

- C++ имеет массу стандартных обобщённых контейнеров и обобщённых алгоритмов над ними
- Например `list` это стандартный двусвязный список
- Работа с ним не сложнее, чем с самописным треугольником

```
Triangle<double> t; // создать треугольник
t.pts = {{1, 0}, {2, 1}, {3, 2}}; // задать точки
double sq = t.double_square() / 2; // вычислить площадь
```

Стандартная библиотека

- C++ имеет массу стандартных обобщённых контейнеров и обобщённых алгоритмов над ними
- Например `list` это стандартный двусвязный список
- Работа с ним не сложнее, чем с самописным треугольником

```
list<int> lst; // создать список

lst.push_back(2); // добавить несколько узлов
lst.push_back(1);
lst.push_front(1); // {1, 2, 1}

lst.remove_if(1); // удалить все единицы
```


https://en.cppreference.com/w/cpp/container/list

Скетч: LRU на языке C++

- Снова можно потренироваться на Problem LC, там есть вариант для C++
- Теперь всё упрощается
  - Можно взять `list` в качестве двусвязного списка
  - Можно взять `unordered_map` в качестве хеш-таблицы
  - Написать уровень абстракции LRU кэша
  - Написать к нему тесты
- Это один модуль (или даже просто один заголовочный файл) и работы здесь одному человеку часа на два, если с тестами

`01-basics/cache.cc`

Скетч: LRU на языке C++

```
// assume T::id of type KeyT
template <typename T, typename KeyT = int>
struct cache_t {
  size_t sz_;
  std::list<T> cache_;

  typedef std::list<T>::iterator ListIt;
  std::unordered_map<int, ListIt> hash_;

  bool full() const;
  bool lookup(const T *elem);
};
```

Скетч: LRU на языке C++

```
bool lookup(const T *elem) {
  auto hit = hash_.find(elem->id);

  if (hit == hash_.end()) {
    if (full()) {
      hash_.erase(cache_.back().id);
      cache_.pop_back();
    }
    cache_.push_front(*elem);
    hash_[elem->id] = cache_.begin();
    return false;
  }

  auto eltit = hit->second;
  if (eltit != cache_.begin())
    cache_.splice(cache_.begin(), cache_, eltit, std::next(eltit));
  return true;
}
```

Скетч: LRU на языке C++

```
cache_t<page_t> c;
std::cin >> m >> n; // scanf("%d%d", &n, &m);
c.sz_ = m;
for (int i = 0; i < n; ++i) {
  page_t p;
  std::cin >> p.id; // scanf("%d", &q);
  assert(std::cin.good());
  if (c.lookup(&p)) hits += 1;
}
std::cout << hits << "\n"; // printf("%d\n", n);
```

Обсуждение

- Получившаяся программа всё ещё плоха с точки зрения C++
  - Структура кэша не инкапсулирует контекст и не создаёт законченной абстракции
  - Мы не написали конструктор, а вместо этого присваиваем sz напрямую
- Но она уже гораздо лучше: короче, яснее, надёжней и вы можете использовать этот кеш чтобы кешировать в принципе любые данные
- К тому же в неё легко вносить изменения и оптимизации. Скажем добавить контроль размера кеша как поле в класс а не вычислять каждый раз размер списка

Идея конструктора

- Этот участок кода очень подозрительный

```
cache_t<page_t> c;
std::cin >> m >> n; // scanf("%d%d", &n, &m);
c.sz_ = m;
```

- Хочется сделать параметр размера обязательным при создании кэша

Идея конструктора

- Напишем простой конструктор

```
template <typename T, typename KeyT = int>
struct cache_t {
 size_t sz_;
 // ....
 cache_t(size_t sz) : sz_(sz) {} // ctor
```

- Теперь попытка создать кэш без параметра это ошибка

```
cache_t<page_t> c; // FAIL
```

Идея конструктора

- Напишем простой конструктор

```
template <typename T, typename KeyT = int>
struct cache_t {
 size_t sz_;
 // ....
 cache_t(size_t sz) : sz_(sz) {} // ctor
```

- Мы создаём кэш после того как у нас есть вся информация

```
std::cin >> m >> n;
cache_t<page_t> c{m}; // creating with ctor
```

Мы только потрогали воду лапкой

- Язык C++ богат возможностями
- Если вы просто знаете C, то многим вы можете начинать пользоваться прямо сейчас
- Но увы, язык C++ сложен и коварен
- Использовать его, не обладая глубоким пониманием происходящего часто бывает неприятно

C++ за 21 день?

- Надо осознавать, что мы находимся в начале **долгого** пути и настраиваться на марафонскую дистанцию
- Я надеюсь, на втором курсе мы успеем разобрать
  - Базовое объектно-ориентированное программирование
  - Простые шаблоны классов и функций
  - Основные механизмы стандартной библиотеки
  - Много интересных алгоритмов и структур данных
- Последний пункт особенно важен. Наверное можно механически выучить язык не научившись программированию на этом языке. Но это бессмысленно.

- Немного о кэшах
- Реализация на C и её проблемы
- Реализация на C++
- Бонус: больше кэшей

Оптимальное кэширование

- Представим, что мы заранее знаем все страницы которые потребуются в будущем
- Тогда оптимальная стратегия (названная именем Ласло Белади) который открыл её в 1966 году состоит в следующем: для каждой следующей страницы посмотреть в будущем когда она встретится. Выкинуть из кэша ту, которая встретится позже прочих
- Опишите алгоритм кэширования Белади
- Оцените его сложность
- Реализуйте его на C++

Тестирование кэшей

- Допустим мы хотим сравнить LRU с оптимальным кэшированием
- Как бы вы генерировали тестовые данные?
- Какие паттерны доступа вы бы использовали?
- Обобщается ли "паттерн доступа" как структура на C++?

Простейшее кэширование: RANDOM

- Intel i860 использовал случайное кэширование
- Случайное кэширование вытесняет из кэша случайную страницу. Оно хорошо своей простотой: по сути нам нужен только массив
- Не реализуйте такое кэширование. Подумайте над своими тестами. Стоит, пожалуй, добавить в них какой-то такой, который на случайном кэшировании будет исключительно плох. Как он может выглядеть?

Простейшее кэширование: FIFO

- Ещё один простейший алгоритм кэширования требует из всех структур данных только очередь (например на базе односвязного списка)
- Из кэша вытесняются страницы с хвоста очереди, новые добавляются в начало
- Увы, этот алгоритм имеет один серьёзный недостаток: увеличение размера кэша при определённых условиях может привести к ухудшению hit rate
- Найдите такой случай и добавьте его в тесты
- Как LRU ведёт себя на таких тестах? Можно ли найти такое для LRU?

https://en.wikipedia.org/wiki/Beladys_anomaly

Обобщение: LRU-K

- LRU-K убирает из кэша страницу, K-й доступ к которой дальше всего во времени
- Ясно, что LRU-1 это просто LRU
- Какие дополнительные структуры данных потребуются для LRU-K?
- Насколько сложно обобщить C++ реализацию LRU для LRU-K?
- Какой выигрыш она даст на ваших тестах?

Обсуждение

- Настало время разработать вашу собственную стратегию кэширования?

Домашняя работа HWC – кэши

- Выберите любой алгоритм кэширования из перечисленных ниже
  - ARC (adaptive replacement cache)
  - 2Q (алгоритм двух очередей, дальнейшее развитие LRU-2)
  - LFU (вытеснение наименее часто используемого)
  - LIRS (low-inference recency set)
- Реализуйте его на C++ и пришлите на email ссылку на ваш репозиторий
- Разработка хороших тестов – важная часть вашей задачи

Литература

- [CC11] ISO/IEC 14882 – "Information technology – Programming languages – C++", 2011
- [C11] ISO/IEC 9899 – "Information technology – Programming languages – C", 2011
- [BS] Bjarne Stroustrup, The C++ Programming Language (4th Edition) , 2013
- [$K\&R$] Brian W. Kernighan, Dennis Ritchie – The C programming language, 1988
- [$Tannen$] Andrew Tanenbaum – Modern Operating Systems (2nd ed), 2001
- [$Megiddo$] Megiddo, Modha – Outperforming LRU with an Adaptive Replacement Cache Algorithm, 2004