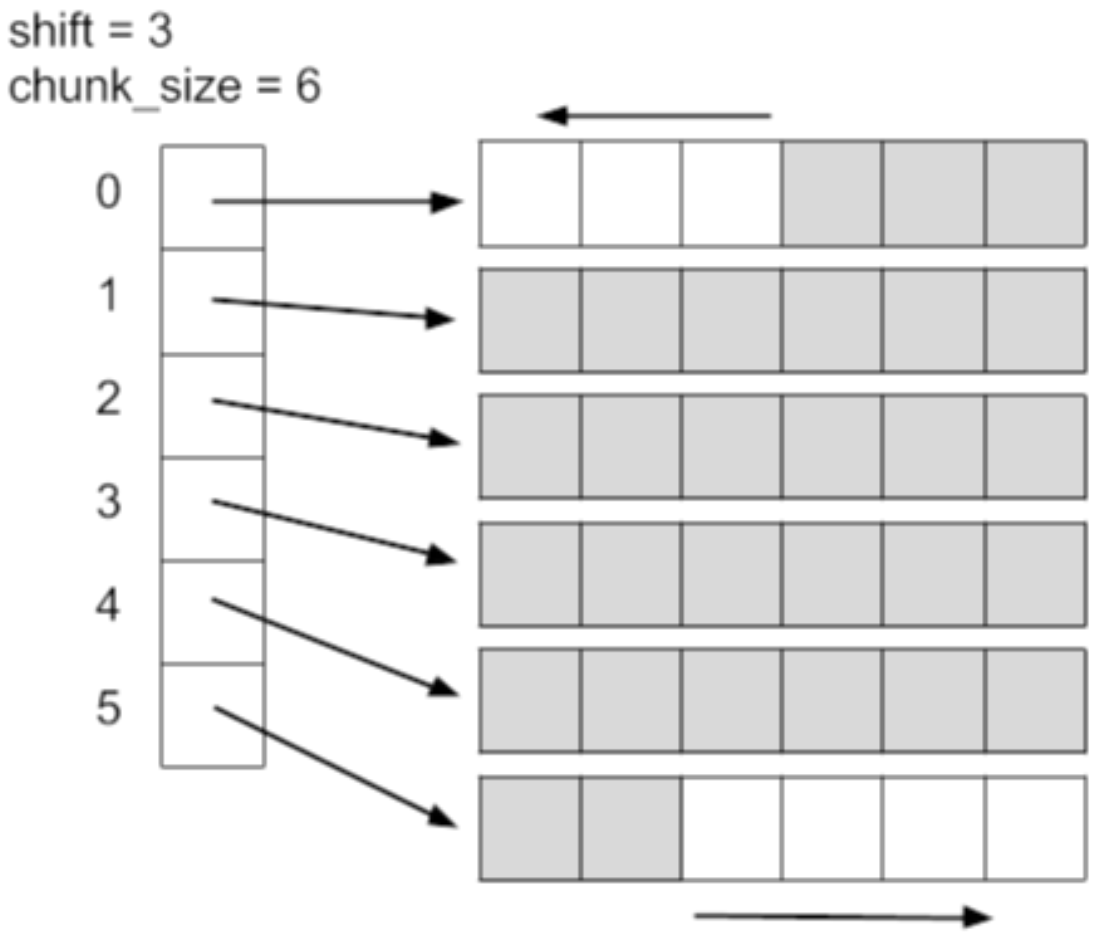
Standard Template Library

ЛЕКЦИЯ №10

std::array
`array.cpp`

1. замена массиву в стиле Си, выделенному на стеке
2. данные лежат в непрерывной области памяти (на стеке)
3. фиксированная длина
4. столь же эффективен
5. удобен для временных буферов
6. стек не бесконечен



std::deque
deque.cpp

1. добавление с обеих сторон эффективно
2. данные хранятся кусками фиксированного размера
3. произвольный доступ (но чуть медленнее, чем std::vector)
4. вставка в середину неэффективна
5. инвалидация итераторов иногда

std::stack
`stack.cpp`

1. не контейнер вовсе, а адаптор
2. обычно над std::deque
3. итераторы отсутствуют вовсе
4. очень ограниченный интерфейс

std::queue
queue.cpp

1. не контейнер вовсе, а адаптор
2. обычно над std::deque
3. итераторы отсутствуют вовсе
4. очень ограниченный интерфейс

std::forward_list
`forward_list.cpp`

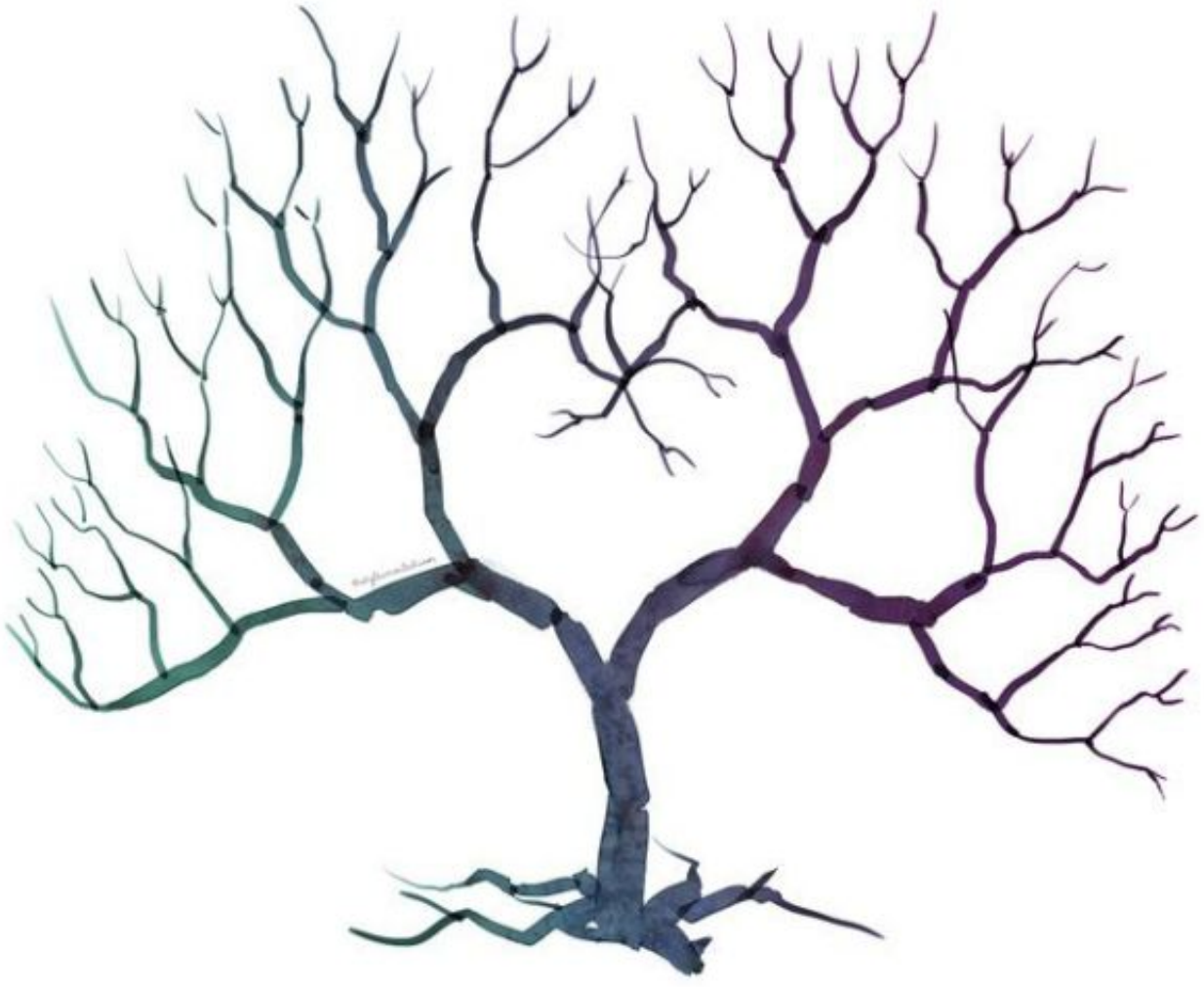
1. односвязный список
2. быстрая вставка и удаление (при наличии итератора)
3. однонаправленные (и только вперёд) итераторы
4. инвалидация итераторов – почти никогда
5. можно получить итератор на -1 элемент

std::list
`list.cpp`

1. двусвязный список
2. быстрая вставка и удаление (при наличии итератора)
3. двунаправленные итераторы
4. инвалидация итераторов – почти никогда

Ассоциативные контейнеры в STL

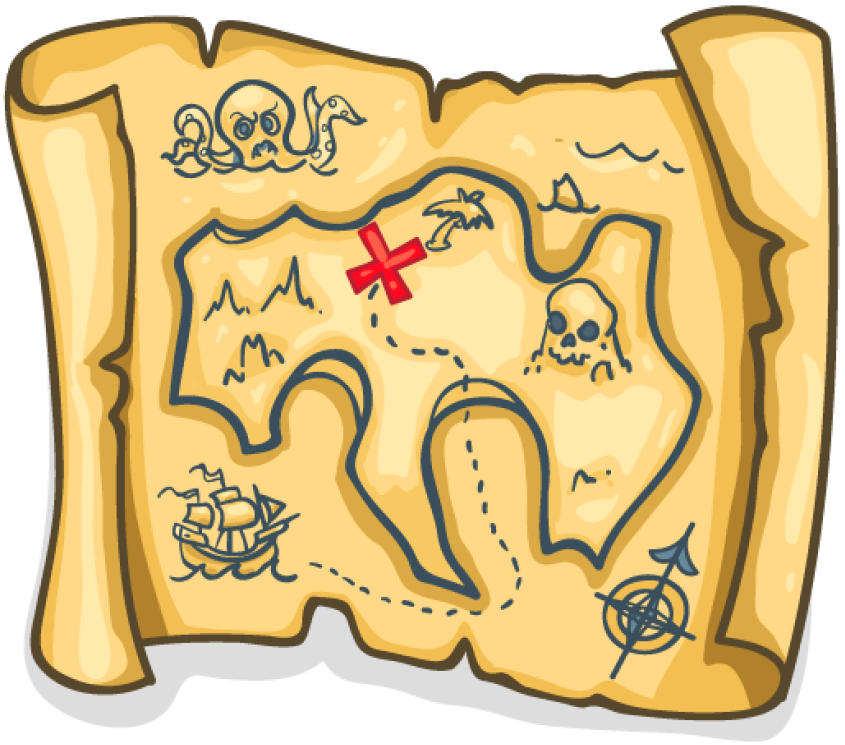
1. std::set
2. std::map
3. std::multiset
4. std::multimap



std::set
set.cpp

1. бинарное дерево (часто красно-чёрное*)
2. элементы уникальны
3. сравнение компаратором (по умолчанию operator<)
4. отношение эквивалентности
5. итерирование в порядке сравнения элементов
6. итераторы двунаправленны

* https://algs4.cs.princeton.edu/33balanced/

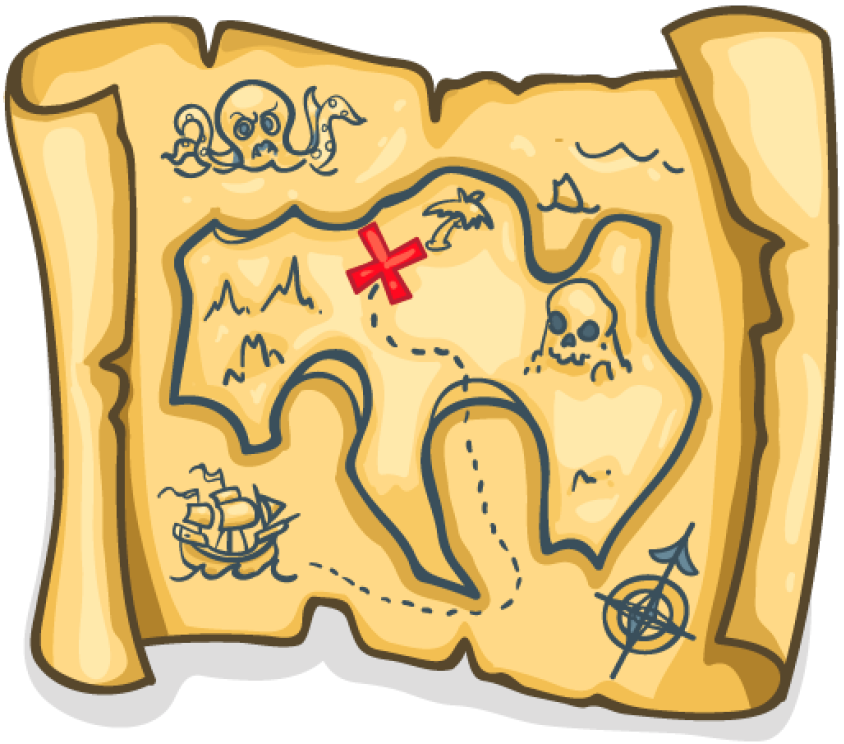


std::map
map.cpp

1. бинарное дерево (часто красно-чёрное)
2. ключи уникальны
3. сравнение ключа компаратором (по умолчанию operator<)
4. отношение эквивалентности
5. итерирование в порядке сравнения ключей элементов
6. итераторы двунаправленные

std::multiset
multiset.cpp

1. бинарное дерево (часто красно-чёрное)
2. ключи не уникальны
3. порядок элементов с одним ключём – в порядке вставки

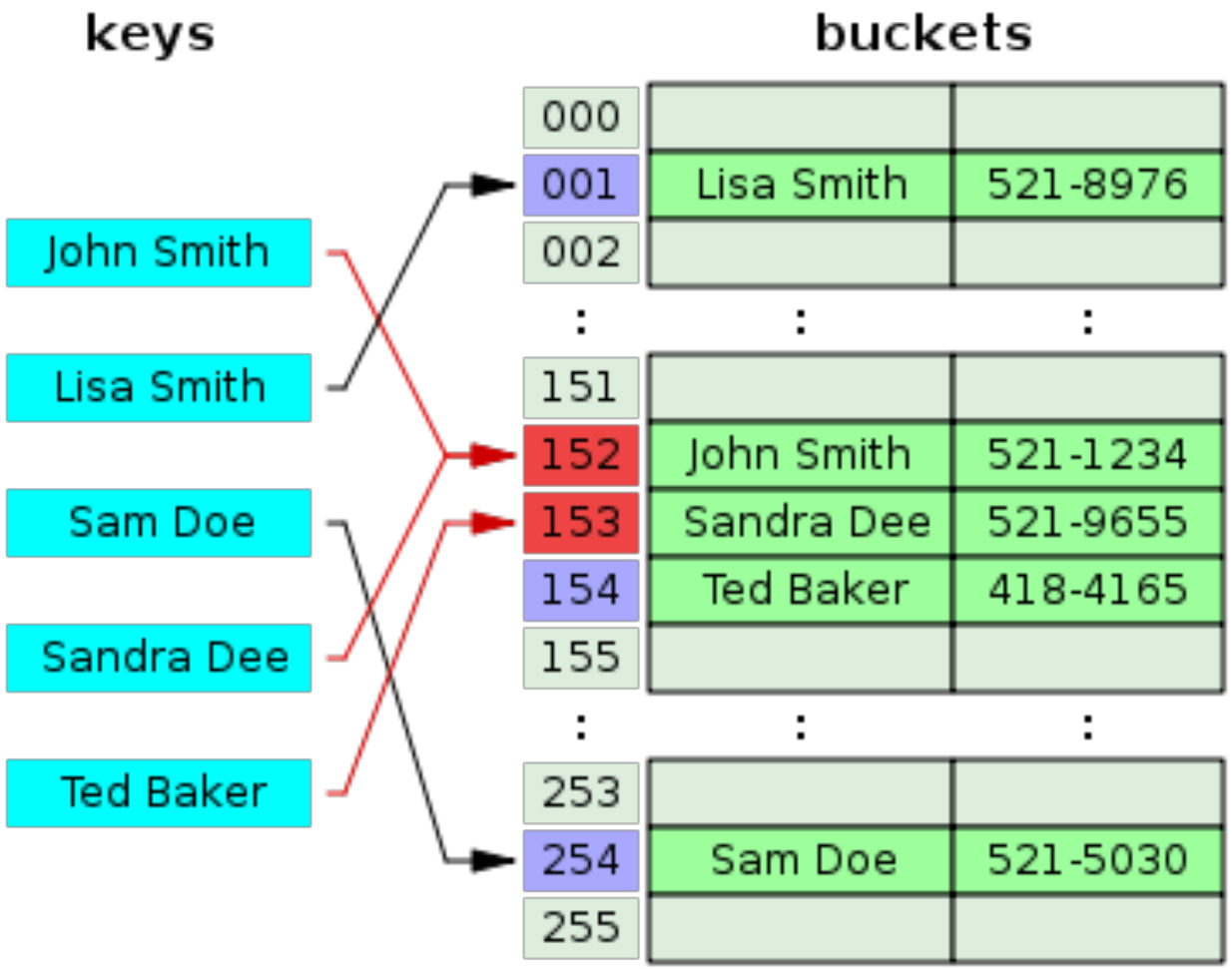


std::multimap

- бинарное дерево (часто красно-чёрное)
- ключи не уникальны
- порядок элементов с одним ключём – в порядке вставки

Неупорядоченные контейнеры в STL

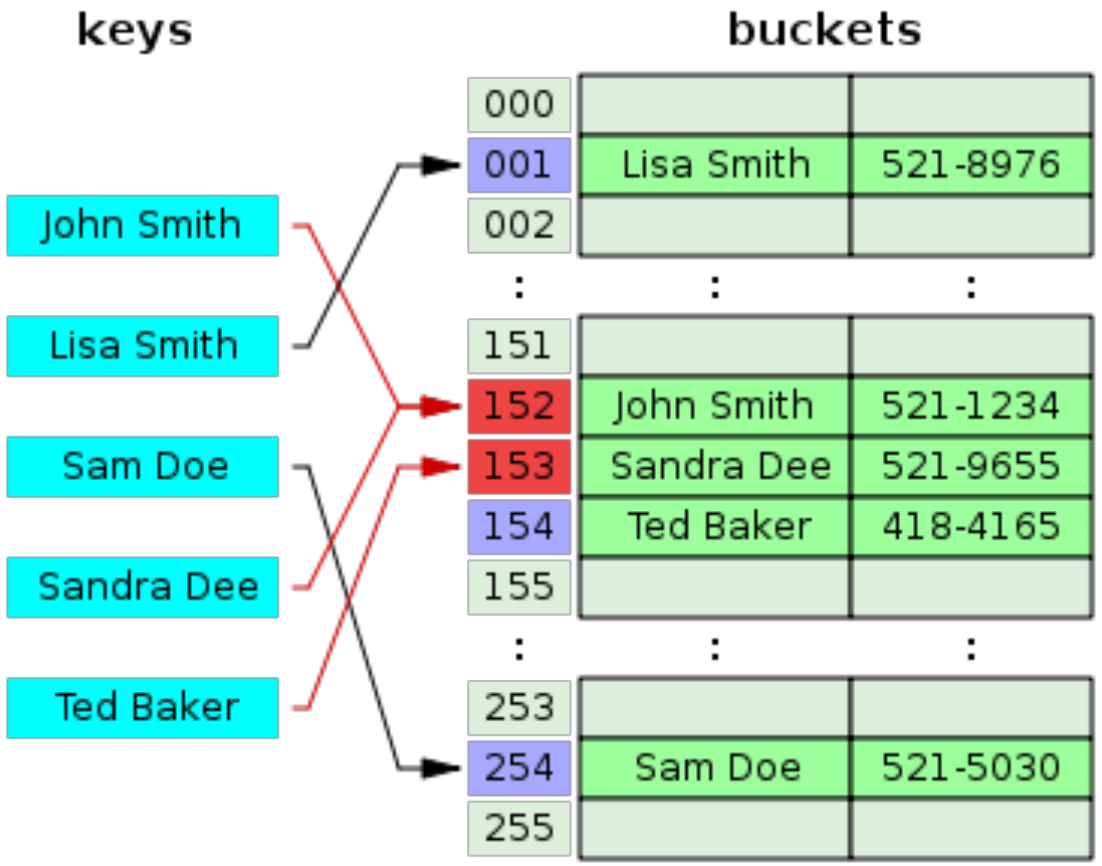
1. std::unordered_set
2. std::unordered_map
3. std::unordered_multiset
4. std::unordered_multimap



std::unordered_set

unordered_set.cpp

1. хеш-таблица
2. элементы уникальны
3. сравнение по хеш-функции
4. итерирование в порядке хешей
5. итераторы однонаправленные



std::unordered_map

1. хеш-таблица
2. элементы уникальны по ключу
3. сравнение по хеш-функции ключей
4. итерирование в порядке хешей
5. итераторы однонаправленные

Алокаторы памяти

ОПТИМИЗИРУЕМ ОПЕРАЦИЮ ДОСТУПА К ПАМЯТИ

Аллокаторы

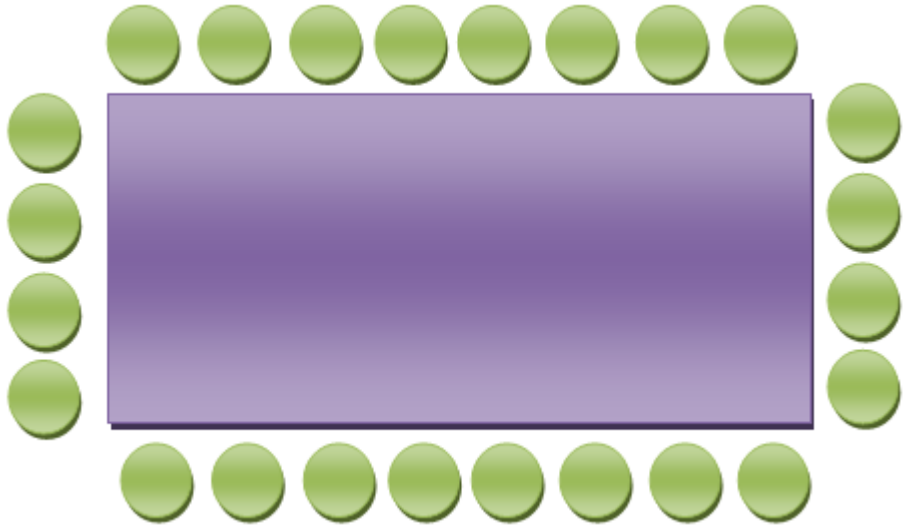
Каждый контейнер имеет определенный для него *аллокатор* (allocator). Аллокатор управляет выделением памяти для контейнера. Аллокатором по умолчанию является объект класса allocator, но вы можете определять свои собственные аллокаторы, если это необходимо для специализированных приложений. Для большинства применений аллокатора по умолчанию вполне достаточно.

Классическая история- ресторан

Предположим у нас есть суши ресторан.

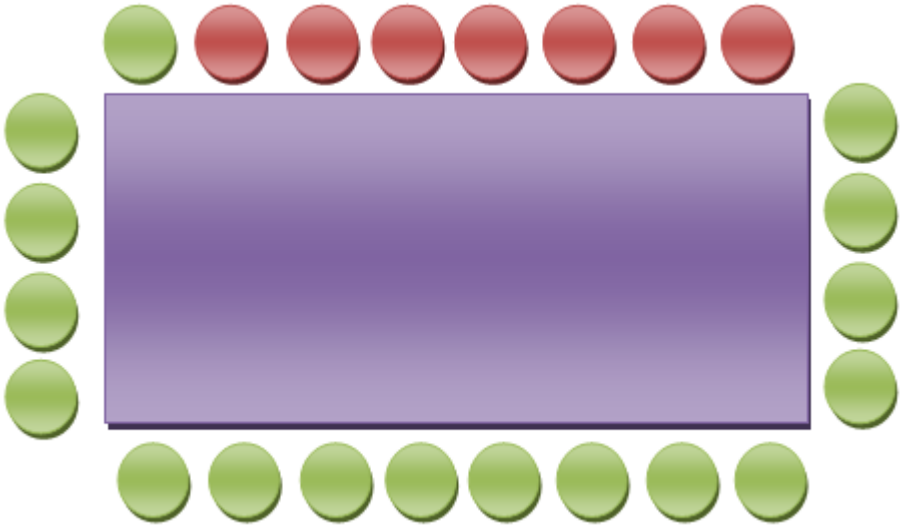
У нас есть стол и места, размещенные вокруг.

В наши обязанности входит размещение посетителей за столом.



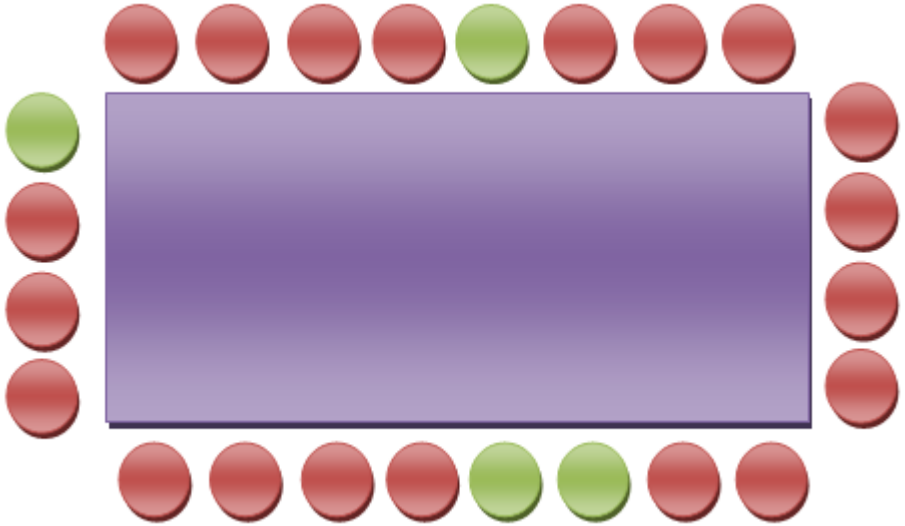
Назначаем места посетителям

Заходит компания их трёх человек и просит показать им места. Довольный приходом посетителей, ты любезно их усаживаешь. Не успевают они присесть и положить себе немного суши, как дверь снова открывается, и заходят ещё четверо! Ресторан теперь выглядит так…



Через некоторое время

И тут приходят четыре человека и просят усадить их. Прагматичный по натуре, ты тщательно отслеживал, сколько осталось свободных мест, и смотри, сегодня твой день, четыре места есть! Есть одно «но»: эти четверо жутко социальные и ходят сидеть рядом. Ты отчаянно оглядываешься, но хоть у тебя и есть четыре свободных места, усадить эту компанию рядом ты не можешь! Просить уже имеющихся посетителей подвинуться посреди обеда было бы грубовато, поэтому, к сожалению, у тебя нет другого выбора, кроме как дать новым от ворот поворот, и они, возможно, никогда не вернутся. Всё это ужасно печально.



Динамическое выделение памяти

1. **malloc и new пытаются быть всем в одном флаконе для всех программистов...**
   Они выделят вам несколько байтов ровно тем же способом, что и несколько мегабайтов. Уних нет той более широкой картины, которая есть у программистов.

2. **Относительно плохая производительность...**
   Стоимость операций free или delete в некоторых схемах выделения памяти также может быть высокой, так как во многих случаях делается много дополнительной работы для того, чтобы попытаться улучшить состояние кучи перед последующими размещениями. «Дополнительная работа» является довольно расплывчатым термином, но она может означать объединение блоков памяти, а в некоторых случаях может означать проход всего списка областей памяти, выделенной вашему приложению!

3. **Они являются причиной фрагментации кучи...**

4. **Плохая локальность ссылок...**
   В сущности, нет никакого способа узнать, где та память, которую вернёт вам malloc или new, будет находиться по отношению к другим областям памяти в вашем приложении. Это может привести к тому, что у нас будет больше дорогостоящих промахов в кеше, чем нам нужно, и мы в концов будем танцевать в памяти как на углях.

Перегрузка операторов new и delete

operator_new.cpp

Операторы new и delete можно перегрузить. Для этого есть несколько причин:

- Можно увеличить производительность за счёт кеширования: при удалении объекта не освобождать память, а сохранять указатели на свободные блоки, используя их для вновь конструируемых объектов.
- Можно выделять память сразу под несколько объектов.
- Можно реализовать собственный "сборщик мусора" (garbage collector).
- Можно вести лог выделения/освобождения памяти.

Операторы new и delete имеют следующие сигнатуры:

```
void *operator new(size_t size);

void operator delete(void *p);
```

Оператор new принимает размер памяти, которую необходимо выделить, и возвращает указатель на выделенную память.

Оператор delete принимает указатель на память, которую нужно освободить.

allocator_traits

http://www.cplusplus.com/reference/memory/allocator_traits/

```
template <class T>
struct custom_allocator {
  typedef T value_type;
  custom_allocator() noexcept {}
  template <class U> custom_allocator (const custom_allocator<U>&) noexcept {}
  T* allocate (std::size_t n) { return static_cast<T*>(::operator new(n*sizeof(T))); }
  void deallocate (T* p, std::size_t n) { ::delete(p); }
};

template <class T, class U>
constexpr bool operator== (const custom_allocator<T>&, const custom_allocator<U>&) noexcept
{return true;}

template <class T, class U>
constexpr bool operator!= (const custom_allocator<T>&, const custom_allocator<U>&) noexcept
{return false;}

int main () {
  std::vector<int,custom_allocator<int>> foo = {10,20,30};
  for (auto x: foo) std::cout << x << " ";
  std::cout << '\n';
  return 0;
}
```

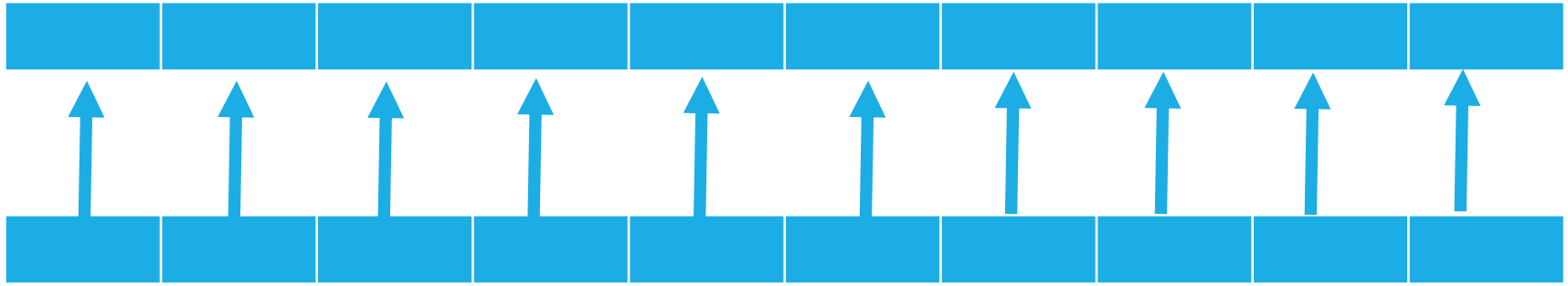
Более правильный пример allocator.cpp

```
template<class T,size_t BLOCK_SIZE>
 struct allocator {
        using value_type = T;
        using pointer = T * ;
        using const_pointer = const T*;
        using size_type = std::size_t;
        template<typename U>
        struct rebind {
            using other = allocator<U,BLOCK_SIZE>;
        };

        T* allocate(size_t n);
        void deallocate(T* ,  size_t );
        template<typename U, typename ...Args>
        void construct(U *p, Args &&...args) ;
        void destroy(pointer p);
 };
```

Simple Allocator [1/4]

Used_blocks: Память для стркутур Item

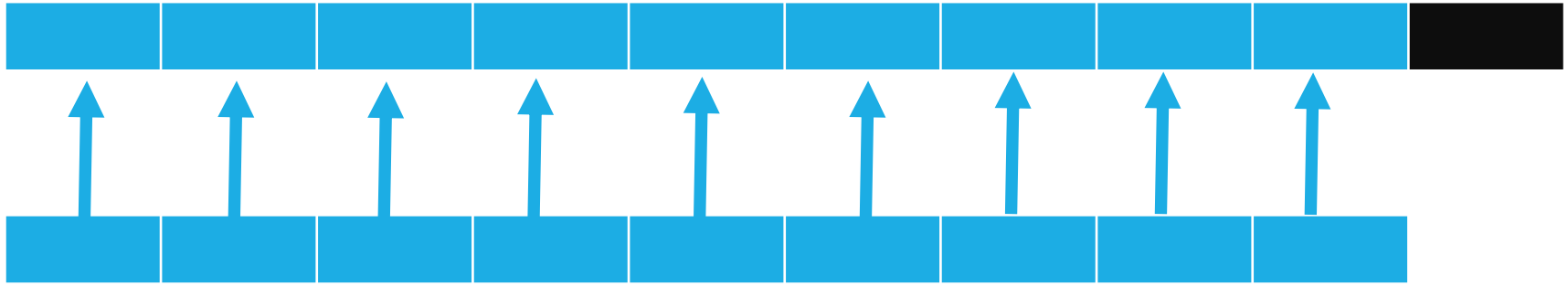


Free_blocks: Указатели на свободные блоки

Вначале все блоки свободные. Каждый указатель в структуре free_blocks указывает на некоторый адрес в структуре used_blocks.

Simple Allocator [2/4]

Used_blocks: Память для стркутур Item



Free_blocks: Указатели на свободные блоки

Когда выделяется память – то возвращается значение последнего указателя в структуре free_blocks на адрес
В used_blocks

Simple Allocator [3/4]

`Used_blocks: Память для стркутур Item`

`Free_blocks: Указатели на свободные блоки`

После того как выделенно 5 объектов – мы имеет вот такую ситуацию

Simple Allocator [4/4]

Used_blocks: Память для стркутур Item

Free_blocks: Указатели на свободные блоки

Это происходит когда мы взывали оператор delete. В конец массива free_blocks добавляется адрес Освободившегося блока

Простой Allocator фиксированной длинны `simple_allocator.cpp`

```
void test2(){
    auto begin = std::chrono::high_resolution_clock::now();
    std::list<SomeStruct,mai::allocator<SomeStruct,1000>> my_list;
    for(int i=0;i<1000;i++) my_list.push_back(SomeStruct());
    for(int i=0;i<1000;i++) my_list.erase(my_list.begin());
    auto end = std::chrono::high_resolution_clock::now();
    std::cout << "test2: " << std::chrono::duration_cast<std::chrono::mi
croseconds>(end - begin).count() << std::endl;

}
```

Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ