



Объектно-ориентированное программирование

2023



STL

1. Входит в поставку стандартных C++ компиляторов
2. Содержит
 - алгоритмы (`std::rotate`, `std::find_if` и т.д.)
 - **контейнеры** (`std::vector<T>`, `std::list<T>` и т.д.)
 - функциональные объекты (`std::greater<T>`, `std::logical_and<T>` и т.д.)
 - **итераторы** (`std::iterator`, `std::back_inserter` и т.д.)
3. Спецификация находится тут:
<http://www.cplusplus.com/reference>

Контейнеры

Контейнер — это объект, который может содержать в себе другие объекты. Существует несколько разных типов контейнеров. Например, класс `vector` определяет динамический массив, `deque` создает двунаправленную очередь, а `list` представляет связный список. Эти контейнеры называются **последовательными контейнерами** (sequence containers), потому что в терминологии STL последовательность — это линейный список.

STL также определяет **ассоциативные контейнеры** (associative containers), которые обеспечивают эффективное извлечение значений на основе ключей. Таким образом, ассоциативные контейнеры хранят пары “ключ/значение”. Примером может служить `map`. Этот контейнер хранит пары “ключ/значение”, в которых каждый ключ является уникальным. Это облегчает извлечение значения по заданному ключу.

Итераторы

Предоставляет способ последовательного доступа ко всем элементам контейнера, не раскрывая его внутреннего представления.

Зачем

- Составной объект, скажем список, должен предоставлять способ доступа к своим элементам, не раскрывая их внутреннюю структуру.
- Иногда требуется обходить список по-разному, в зависимости от решаемой задачи.
- Нужно, чтобы в один и тот же момент было определено несколько активных обходов списка.

Идея

Основная его идея в том, чтобы за доступ к элементам и способ обхода отвечал не сам список, а отдельный объект - итератор. В классе `Iterator` определен интерфейс для доступа к элементам списка. Объект этого класса отслеживает текущий элемент, то есть он располагает информацией, какие элементы уже посещались.

Требования к последовательному контейнеру

iterator begin()	Возвращает итератор, указывающий на первый элемент контейнера.
const_iterator begin() const	Возвращает константный итератор, указывающий на первый элемент контейнера.
iterator end()	Возвращает итератор, указывающий на позицию, следующую за последним элементом контейнера.
const_iterator end() const	Возвращает константный итератор, указывающий на позицию, следующую за последним элементом контейнера.
bool empty() const	Возвращает true, если контейнер пуст.
size_type size() const	Возвращает количество элементов, в текущий момент хранящихся в контейнере.
void swap (ContainerType c)	Обменивает между собой содержимое двух контейнеров.

<code>void clear()</code>	Удаляет все элементы из контейнера.
<code>iterator erase(iterator i)</code>	Удаляет элемент, на который указывает <code>i</code> . Возвращает итератор, указывающий на элемент, находящийся после удаленного.
<code>iterator erase(iterator start, iterator end)</code>	Удаляет элементы в диапазоне, указанном <code>start</code> и <code>end</code> . Возвращает итератор, указывающий на элемент, находящийся после последнего удаленного.
<code>iterator insert(iterator i, const T &val)</code>	Вставляет <code>val</code> непосредственно перед элементом, специфицированным <code>i</code> . Возвращает итератор, указывающий на вставленный элемент.
<code>void insert(iterator i, size_type num, const T &val)</code>	Вставляет <code>num</code> копий <code>val</code> непосредственно перед элементом, специфицированным <code>i</code> .
<code>template <class InIter> void insert(iterator i, InIter start, InIter end)</code>	Вставляет последовательность, определенную <code>start</code> и <code>end</code> , непосредственно перед элементом, специфицированным <code>i</code> .

Требования к последовательному контейнеру



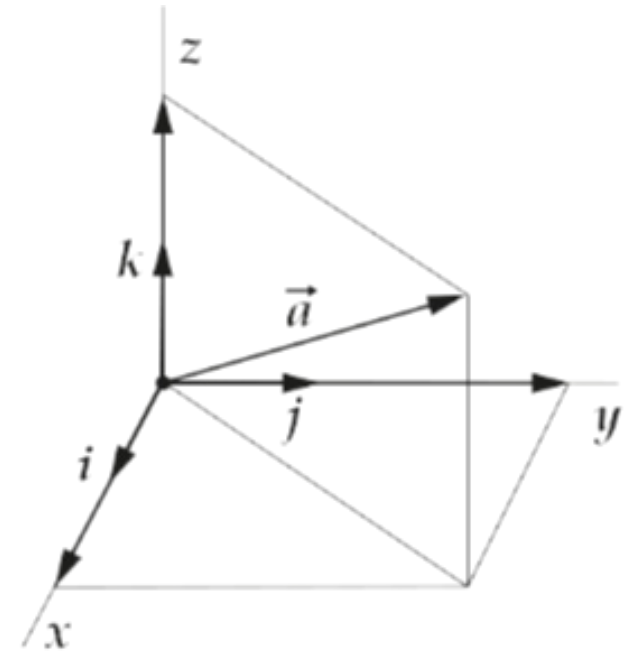
Последовательные контейнеры в STL

1. `std::array`
2. `std::vector`
3. `std::deque`
4. `std::stack`
5. `std::queue`
6. `std::priority_queue`
7. `std::forward_list`
8. `std::list`

std::vector

Vector

1. аналог динамическому массиву Си
2. эмулирует расширяемость
3. добавление в начало неэффективно
4. данные лежат в непрерывной области памяти (в куче)
5. итераторы произвольного доступа
6. инвалидация итераторов почти всегда



Простейший итератор для RangeFor Iterator

```
// for работает с итератором как с указателем!  
class IntIterator{  
    int operator*() ;  
    int operator->() ;  
    bool operator!=(IntIterator const& other) const;  
    IntIterator & operator++() ;  
}
```

Итератор

Контейнер может иметь произвольную структуру и различные методы доступа:



Атрибут	Тип
difference_type	Тип для хранения значения результата вычитания двух итераторов (<u>ptrdiff_t</u>)
value_type	Тип, на который указывает итератор (обычно T – параметр шаблона)
pointer	Тип указателя на элемент контейнера (обычно T*)
reference	Тип ссылки на элемент шаблона (обычно T&)
iterator_category	Категория итератора: <u>input iterator tag</u> <u>output iterator tag</u> <u>forward iterator tag</u> <u>bidirectional iterator tag</u> <u>random access iterator tag</u>

iterator_traits

http://www.cplusplus.com/reference/iterator/iterator_traits/



`curr=head;`



`curr=curr.next;`

Однонаправленный СПИСОК

сложный пример с итератором

Uniqueliterator

```
class ListIterator{
private:
    List&    list;
    size_t   index;
    friend class List;
public:
    using difference_type = int ;
    using value_type = List::value_type;
    using reference = List::value_type& ;
    using pointer = List::value_type*;
    using iterator_category = std::forward_iterator_tag;

    ListIterator(List &l,int i) : list(l), index(i){}

    ListIterator& operator++(){
        ++index;
        return *this;
    }

    reference operator*(){
        return list[index];
    }

    pointer operator->(){
        return &list[index];
    }

    bool operator!=(const ListIterator& other){
        if(index!=other.index) return true;
        if(&list!=&(other.list)) return true;
        return false;
    }
};
```



Предопределенные
итераторы

<http://www.cplusplus.com/reference/iterator/>

reverse_iterator

move_iterator

back_insert_iterator

front_insert_iterator

insert_iterator

istream_iterator

ostream_iterator

istreambuf_iterator

ostreambuf_iterator

std::back_insert_iterator

BackInsert

```
std::vector<int> foo;
std::vector<int> bar;
for (int i = 1; i <= 5; i++) {
    bar.push_back(i * 10);
}

std::back_insert_iterator< std::vector<int> > back_it(foo);
copy(bar.begin(), bar.end(), back_it);
```

Как устроен back_insert_iterator?

BackInsertIterator

```
template <class Container>
    class back_insert_iterator
    {
protected:
    Container* container;

public:
    typedef Container container_type;
    explicit back_insert_iterator (Container& x) : container(&x) {}

    // копирование значения
    back_insert_iterator<Container>&
    operator= (const typename Container::value_type& value)
    { container->push_back(value); return *this; }

    // перемещение значения
    back_insert_iterator<Container>&
    operator= (typename Container::value_type&& value)
    { container->push_back(std::move(value)); return *this; }

    // стандартный набор операторов
    back_insert_iterator<Container>& operator* ()
    { return *this; }
    back_insert_iterator<Container>& operator++ ()
    { return *this; }
    back_insert_iterator<Container> operator++ (int)
    { return *this; }
    };
```


istream_iterator,
insert_iterator,
ostream_iterator

ostream

```
std::istream_iterator<double> eos; // end-of-stream iterator
std::istream_iterator<double> iit(std::cin); // stdin
iterator
if (iit != eos) value1 = *iit;
```

```
std::vector<double> vec;
std::insert_iterator<std::vector<double>>
insert_it(vec, vec.begin());
std::copy(iit, eos, insert_it);
```

```
std::ostream_iterator<double> out(std::cout, " ");
std::copy(vec.begin(), vec.end(), out);
```

Итераторы – итого

1. Итераторы – хранят ссылку на контейнер и даже на определенный элемент в контейнере
2. Итераторы – знают о структуре контейнера
3. Итераторы – предоставляют однотипный интерфейс по доступу к любому контейнеру
4. Итераторы – могут не только получать данные из контейнера, но могут записывать данные в контейнер (зависит от структуры контейнера)
5. Итераторы не могут менять размер контейнера



std::array

Array

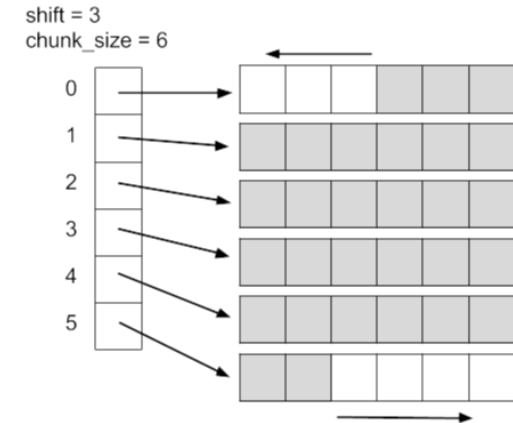
- замена массиву в стиле Си, выделенному на стеке
- данные лежат в непрерывной области памяти (на стеке)
- фиксированная длина
- столь же эффективен
- удобен для временных буферов
- стек не бесконечен



std::deque

Deque

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



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



`std::queue`

Queue

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

Добавляемый
узел



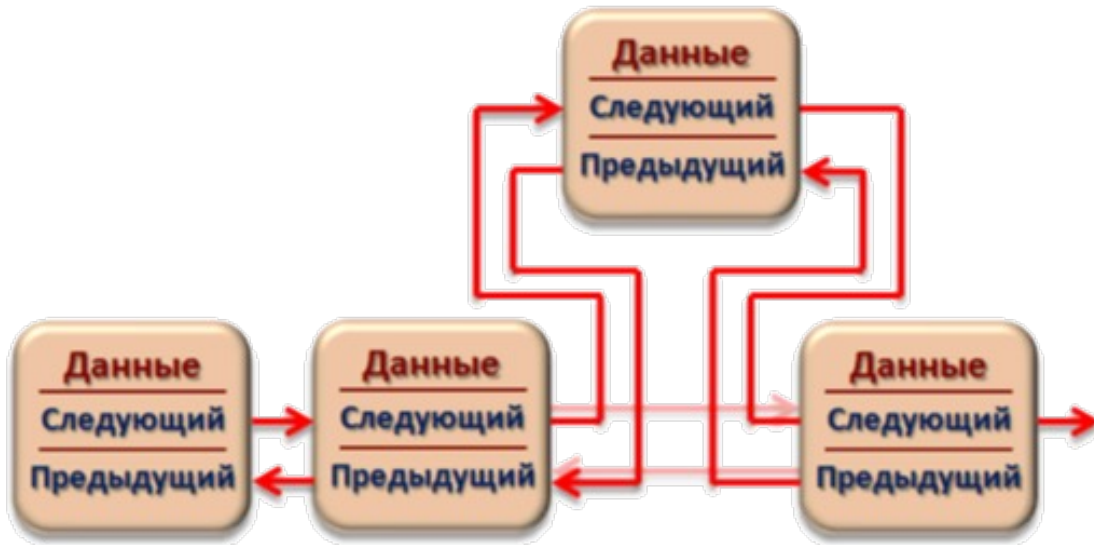
`std::forward_list` **Forward_list**

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

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

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

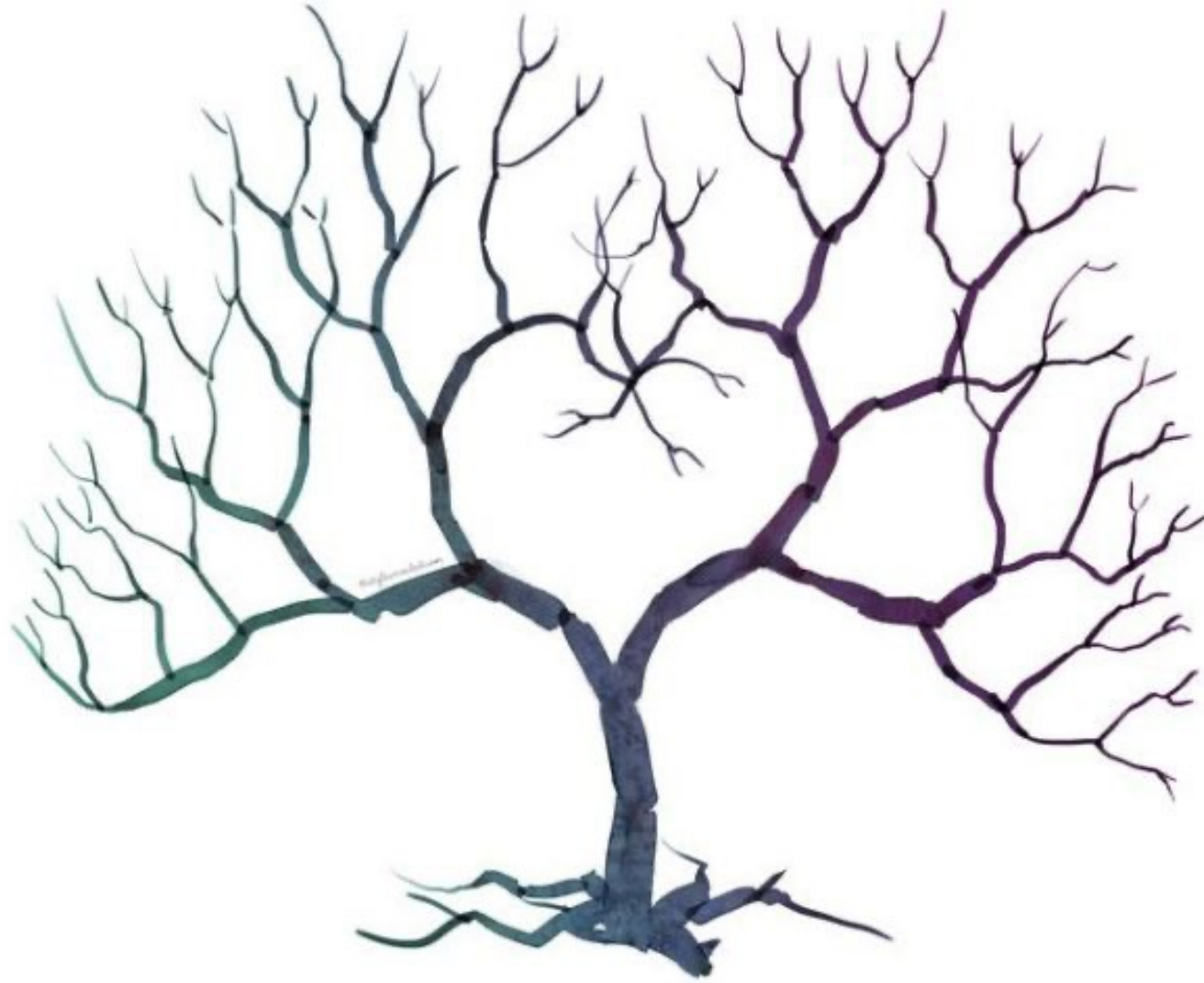
* <https://algs4.cs.princeton.edu/33balanced/>



`std::map`

Map

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

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

std::multimap

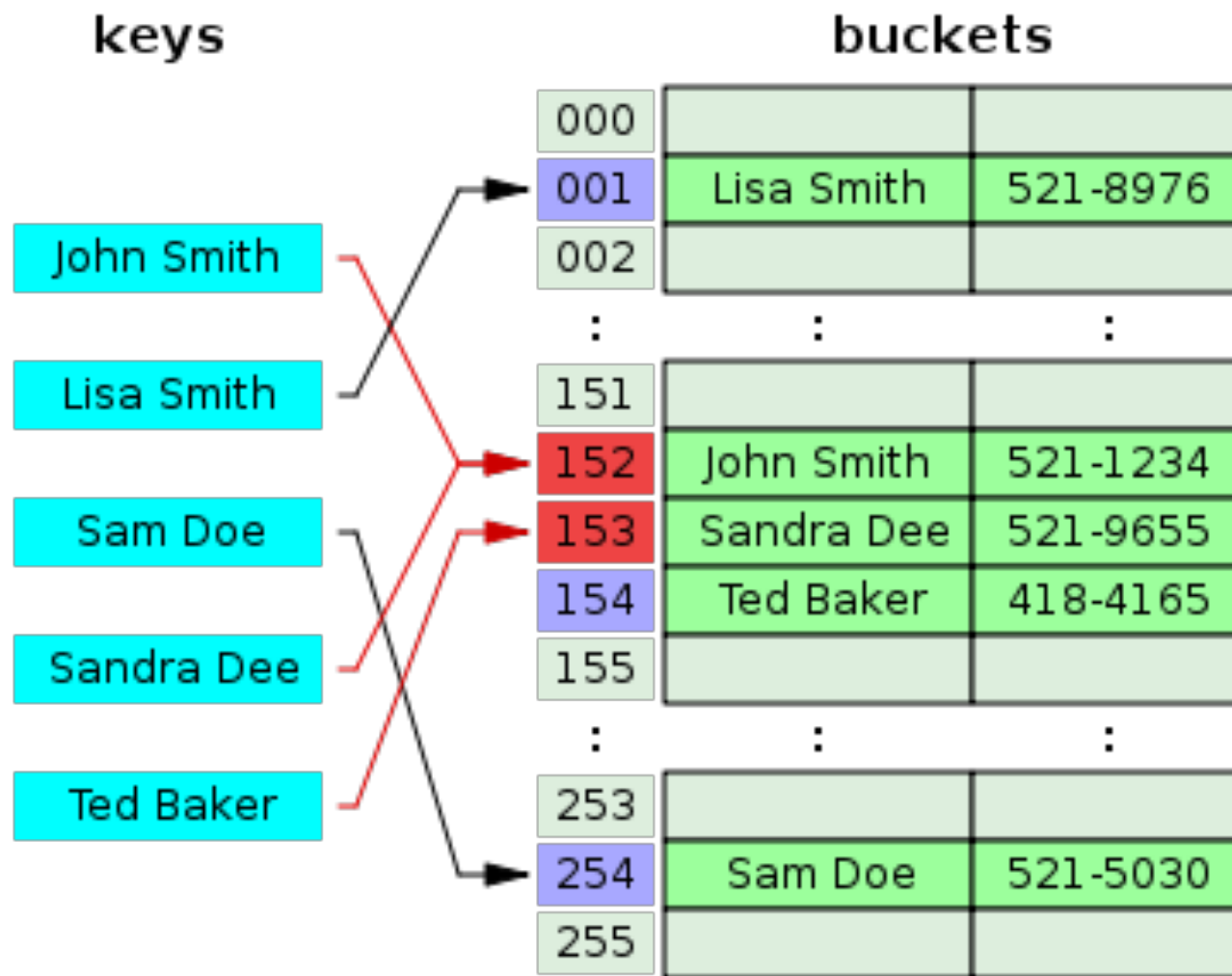
- бинарное дерево (часто красно-чёрное)
- ключи не уникальны
- порядок элементов с одним ключём — в порядке вставки

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

Аллокаторы памяти

ОПТИМИЗИРУЕМ
ОПЕРАЦИЮ ДОСТУПА К
ПАМЯТИ



Аллокаторы

Каждый контейнер имеет определенный для него *аллокатор* (allocator).

Аллокатор управляет выделением памяти для контейнера.

Аллокаторм по умолчанию является объект класса `allocator`, но вы можете определять свои собственные аллокаторы, если это необходимо для специализированных приложений.

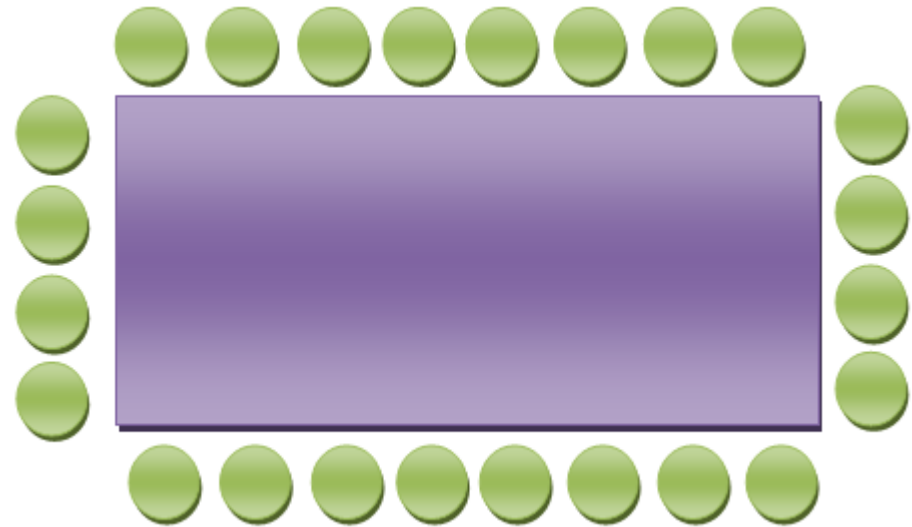
Для большинства применений аллокатора по умолчанию вполне достаточно.

Классическая история - ресторан

Предположим у нас есть суши ресторан.

У нас есть стол и места, размещенные вокруг.

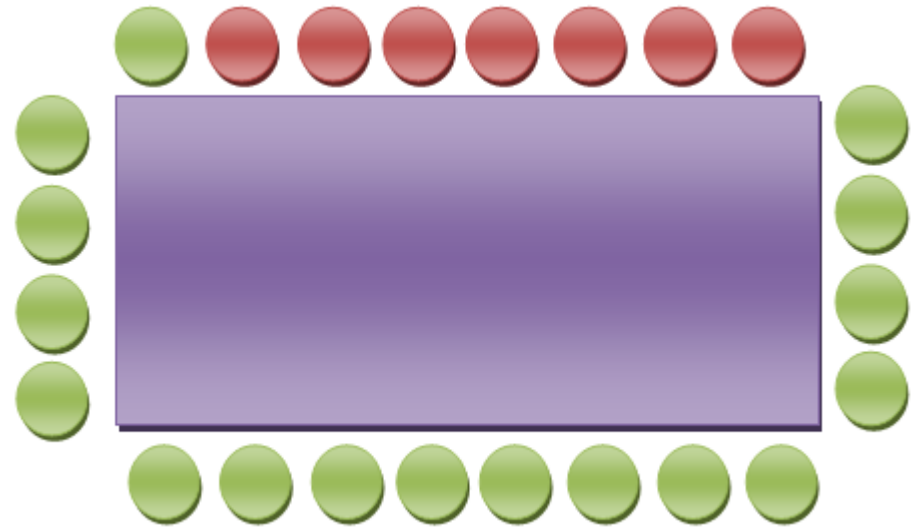
В наши обязанности входит размещение посетителей за столом.



<https://habr.com/ru/post/274827/>

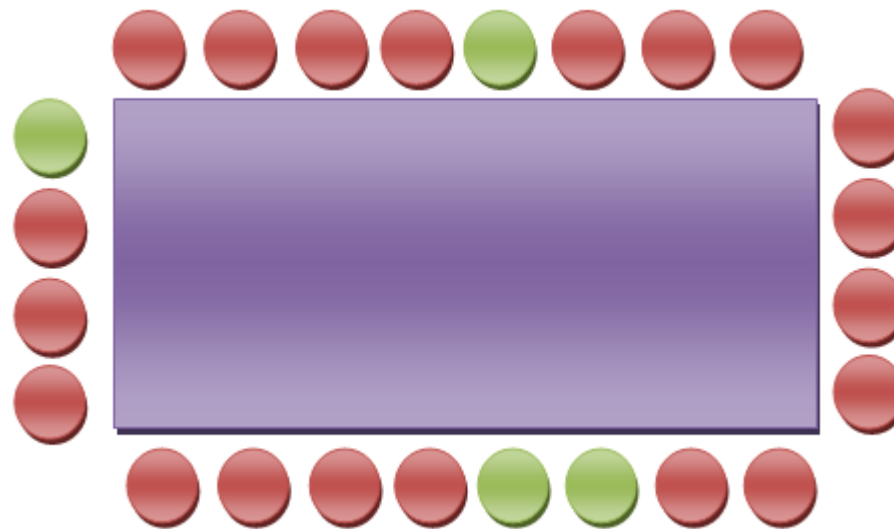
Назначаем места посетителям

Заходит компания из 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 среди 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 выделяют вам несколько байтов ровно тем же способом, что и несколько мегабайтов. У них 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;
}
```

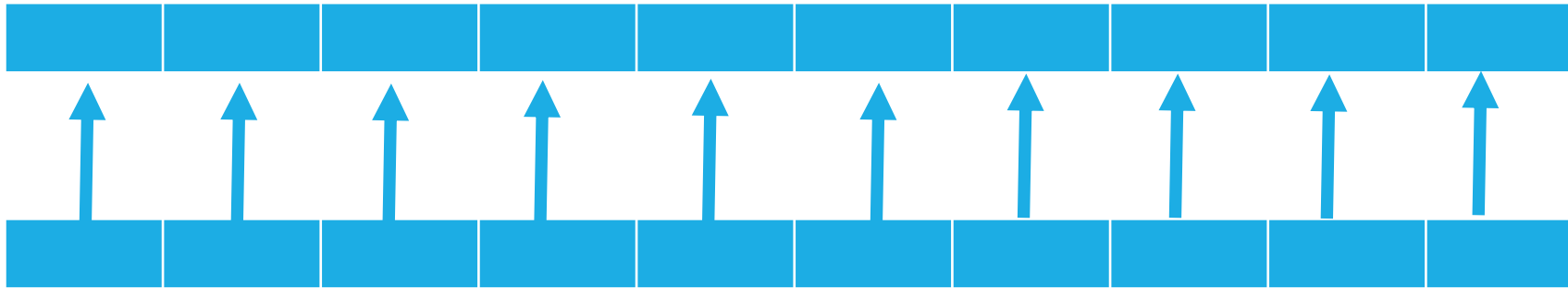

Более правильный пример Allocator

```
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 структур Item

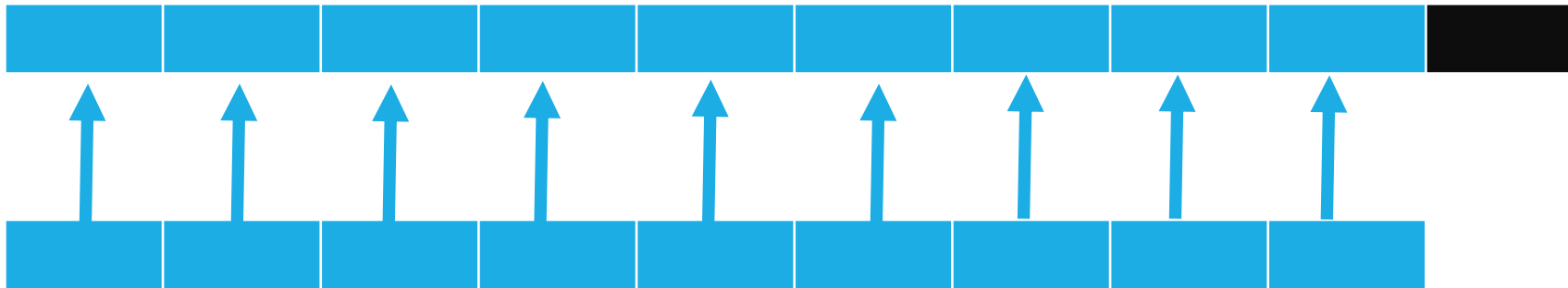


Free_blocks: Указатели на свободные блоки

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 структур 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 структур Item



Free_blocks: Указатели на свободные блоки

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

Simple Allocator [4/4]

Used_blocks: Память для структур Item



Free_blocks: Указатели на свободные блоки

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

Простой Allocator фиксированной длины

SimpleAllocator

```
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::microseconds>(end - begin).count() << std::endl;
}
```



Спасибо!

НА СЕГОДНЯ ВСЕ