Standard Template Library

ЛЕКЦИЯ №9

STL

1. Входит в поставку стандартных C++ компиляторов
2. Содержит контейнеры, структуры данных, итераторы и алгоритмы
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.

Виды контейнеров

Контейнер	Описание	Требуемый заголовок
deque	Двунаправленная очередь	<deque>
list	Линейный список	<list>
map	Хранит пары «ключ/значение», где каждый ключ ассоциирован только с одним значением	<map>
bitset	Битовое поле — это массив битов фиксированного размера	<bitset>
multiset	Множество, в котором каждый элемент не обязательно уникален	<set>
priority_queue	Очередь с приоритетами	<deque>
queue	Очередь	<deque>
set	Множество уникальных элементов	<set>
stack	Стек	<stack>
vector	Динамический массив	<vector>

Операции, поддерживаемые контейнерами

Все контейнеры должны поддерживать операцию присваивания. Они должны также поддерживать все логические операции. Другими словами, все контейнеры должны поддерживать следующие операции:

=,==,<,<=,!=,>,>=

Все контейнеры должны иметь конструктор, создающий пустой контейнер, и конструктор копирования. Они должны предусматривать деструктор, который освобождает всю память, использованную контейнером, и вызывать деструктор каждого элемента в контейнере.

Итераторы

Одной из основных парадигм данной библиотеки было разделение двух сущностей: контейнеров и алгоритмов. Но для непосредственного воздействия алгоритмом на данные контейнера пришлось ввести промежуточную сущность — итератор.

Итераторы позволили алгоритмам получать доступ к данным, содержащимся в контейнере, независимо от типа контейнера. Но для этого в каждом контейнере потребовалось определить класс итератора. Таким образом алгоритмы воздействуют на данные через итераторы, которые знают о внутреннем представлении контейнера.

Что нового в C++: range-based циклы

Example64_RangeFor

В C++11 была добавлена поддержка парадигмы foreach для итерации по набору. В новой форме возможно выполнять итерации в случае, если для объекта итерации перегружены методы begin() и end().

Данный циклы не работают с массивами, аллоцируемыми динамически (т.к. про них компилятор не знает, какой у них будет размер). А вот с такими работает:

```
int arr[] = { 1, 2, 3, 4, 5 };
for (auto e : arr)
std::cout << e << std::endl;
```

Простейший итератор для RangeFor
Example65_Iterator

```
// for работает с итератором как с указателем!
1.class IntIterator{
2.int operator*() ;
3.int operator->();
4.bool operator!=(IntIterator const& other)
 const;
5.IntIterator & operator++() ;
6.}
```

std::initializer_list<T>
списки инициализации

```
#include <initializer_list>

struct myclass {
  myclass (int,int);
  myclass (initializer_list<int>);
  /* definitions ... */
};

myclass foo {10,20};  // calls initializer_list ctor
myclass bar (10,20);  // calls first constructor
```

Методы контейнеров

Все контейнеры должны предоставлять перечисленные ниже функции.

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, сле- дующую за последним элементом контейнера.
const_iterator end() const	Возвращает константный итератор, указывающий на по- зицию, 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.

Требования к последовательному контейнеру

void clear()	Удаляет все элементы из контейнера.
iterator erase(iterator *i*)	Удаляет элемент, на который указывает *i*. Возвращает итератор, указывающий на элемент, находящийся после удаленного.
iterator erase(iterator *start*, iterator *end*)	Удаляет элементы в диапазоне, указанном *start* и *end*. Возвращает итератор, указывающий на элемент, нахо- дящийся после последнего удаленного.
iterator insert(iterator *i*, const T &*val*)	Вставляет *val* непосредственно перед элементом, спе- цифицированным *i*. Возвращает итератор, указываю- щий на вставленный элемент.
void insert(iterator *i*, size_type *num*, const T &*val*)	Вставляет *num* копий *val* непосредственно перед эле- ментом, специфицированным *i*.
template <class InIter> void insert(iterator *i*, InIter *start*, InIter *end*)	Вставляет последовательность, определенную *start* и *end*, непосредственно перед элементом, специфициро- ванным *i*.

std::vector
#include <vector>

1. Эквивалент массива с динамическим размером
2. Параметры шаблона:
   `template < class T, class Alloc = allocator<T> > class vector;`
3. Итераторы:
   1. begin, end – обычные (cbegin, cend – константные)
   2. rbegin, rend – revers итераторы (crbegin, crend – константные)
4. Доступ к элементам: [size_t] , at(size_t)
5. Модификаторы:
   1. push_back, pop_back –добавление/удаление элемента в конец
   2. insert – добавление элемента в произвольную точку

Пример
Example70_vector

```
1.// vector размера 5 заполненыый int со значением 100
2. std::vector<int> v(5, 100);

3.// печать всех элементов
4. for (int i : v) std::cout << i << std::endl;

5.// итератор для вектора
6.std::vector<int>::iterator it;

7.// удаляем элементы равные 100
8. for (it = v.begin(); it != v.end();) {
9.         if (*it == 100) // значение элемента на итераторе
10.                  v.erase(it); // удаление элемента
11.         else it++; // сдвиг итератора
12.    };
```

std::map
#include <map>

1. Используются для создания «словарей», т.е. пар ключ-значение.

```
template < class Key, // map::key_type
      class T, // map::mapped_type
      class Compare = less<Key>, // map::key_compare
      class Alloc = allocator<pair<const Key,T> > // map::allocator_type
      > class map;
```

2. Основное назначение – поиск значений по ключу. Делается это с помощью
   `std::pair< map::key_type, map::mapped_type> operator[](map::key_type)`
3. В pair есть два атрибута– first и second
4. pair описан в #include<utility>

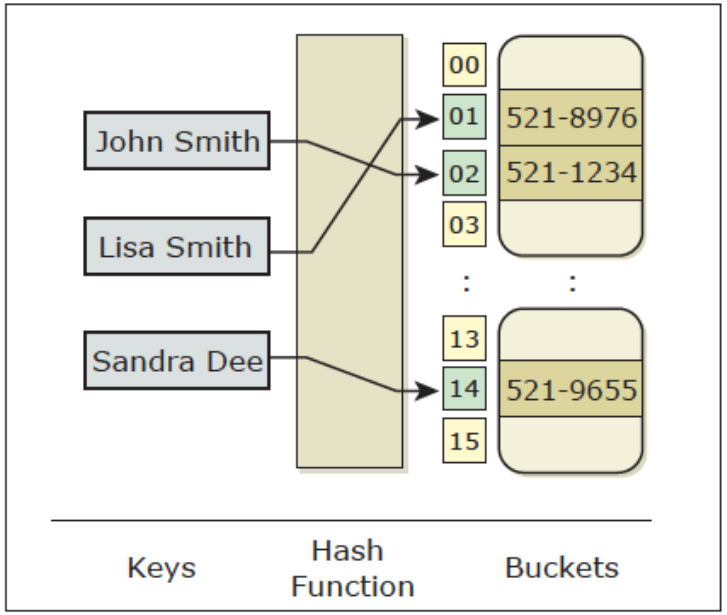


Image by MIT OpenCourseWare.

Пример
Example71_map

```
1.// пара строка – целое число
2.std::map<const char*,int> age;

3.// добавляем или меняем значение
4.age["Иванов"] = 18;

5.// явное добавление
6.age.insert(std::pair<const char*,int>("Петров",21);

7.// удаляем значение
8.age.erase("Петров");

9.// печатаем на экран с помощью итератора
10.for(auto value:age)
11.std::cout << "Age of " <<value.first << "=" << value.second << std::endl;
```

std::deque
Example79_Deque

```
1.std::deque<std::string> myDeque; // создаем пустой дек типа string
2.myDeque.push_back(std::string("World ")); // добавили в дек один элемент типа
  string
3.std::cout << "Количество элементов в деке: " << myDeque.size() << std::endl;
4.myDeque.push_front("Hello "); // добавили в начало дека еще один элемент
5.myDeque.push_back(" !!!"); // добавили в конец дека элемент
6.std::cout << "Количество элементов в деке изменилось, теперь оно = " <<
7. myDeque.size() << std::endl;
8.std::cout << "\nВведенный дек: ";
9. for(auto i:myDeque) std::cout << i << " ";
10.myDeque.pop_front(); // удалили первый элемент
11.myDeque.pop_back(); // удалили второй элемент
```

std::set
Example80_Set

```
1.#include <iostream>
2.#include <set>  // заголовочный файл множеств и мультимножеств
3.#include <iterator>
4.int main(){
5.std::set<char> mySet; // объявили пустое множество

6.// добавляем элементы в множество
7.mySet.insert('I’);
8.// печатаем
9.for( auto i: mySet)  std::cout << i << " ";
10.std::cout << std::endl;
11.// удаляем i
12.mySet.erase('i');
13.// ищем I
14. if(mySet.find('I')!=mySet.end()) std::cout << "Found I" << std::endl;
15.return 0;
16.}
```

std::bitset
Example81_bitset

```
1.#include <iostream>
2.#include <bitset>   // заголовочный файл битовых полей
3.int main( ) {
4.     int number;
5.     std::cout << "Введите целое число от 1 до 255: ";
6.     std::cin >> number;
7.     std::bitset<8> message(number);
8.     std::cout << number << " = " << message << std::endl;

9.      std::bitset<8> bit2 = message;
10.     message = message.flip();
11. // поменять все биты на противоположные
12.     std::cout << "Инвертированное число: " << message << std::endl;
13.     std::bitset<8> bit3 = bit2 | message;
14.}
```

Итераторы

ОДНОТИПНАЯ РАБОТА С РАЗНОТИПНЫМИ КОНТЕЙНЕРАМИ

Итераторы в <vector>

1. Это объект, который указывает на элемент в структуре данных и позволяет перебирать элементы этой структуры данных. Фактически он действует как указатель на элемент массива.
2. Для вектора он выглядел так:
   `std::vector<int>::iterator it;`
3. Доступ к элементам мы осуществляли:`*it;`
4. Смещение по итератору производили `it++, it--;`

Итератор

Контейнер может иметь произвольную структуру и различные методы доступа:

Итератор указывает на элемент контейнера и знает как перейти к следующему элементу

Программе работает только с итератором и его интерфейсом (++)

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
Example72_back_insert_iterator

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

6.// итератор добавляющй элементы в конец foo
7.std::back_insert_iterator< std::vector<int> > back_it(foo);

8.// копируем из bar в back_it
9.std::copy(bar.begin(), bar.end(), back_it);
```

Как устроен back_insert_iterator?
Example76_BackInsertIteratorInside

```
1.template <class Container>  class back_insert_iterator {
2.protected:
3.  Container* container;
4.public:
5.typedef Container container_type;
6.explicit back_insert_iterator (Container& x) : container(&x) {}
7.// копирование значения
8.back_insert_iterator<Container>& operator= (const typename Container::value_type& value)
  {
9. container->push_back(value); return *this; }
10.// перемещение значения
11.back_insert_iterator<Container>& operator= (typename Container::value_type&& value)    {
12.container->push_back(std::move(value)); return *this; }
13.// стандартный набор операторов
14.back_insert_iterator<Container>& operator* ()    { return *this; }
15.back_insert_iterator<Container>& operator++ ()    { return *this; }
16.back_insert_iterator<Container> operator++ (int)    { return *this; }};
```

Семантика перемещения (std::move)
Example77_move

Основная идея в том, что если вы получается в конструкторе копирования или в операторе копирования Rvalue ссылку, то мы должны переместить ее содержимое внутрь нашего объекта. Это и есть семантика перемещения.

Если получаем Lvalue ссылку,то мы копируем значение.

Поддержка семантики перемещения – обязанность самого объекта!

Упрощенно std::move выглядит так:

```
template <class T> T&& move(T& value) {    return
static_cast<T&&> (value);}
```

istream_iterator, insert_iterator, ostream_iterator Example73_Iterators

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

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



7. std::ostream_iterator<double> out(std::cout," ");
8. 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)

Интересные алгоритмы

1. `#include <algorithm>`
2. Sort a vector:
   `std::sort(vec.begin(), vec.end());`
3. Reverse a vector:
   `std::reverse(vec.begin(), vec.end());`
4. Min/Max:
   `std::min(3,1) == 1 ; std::max(3,1) == 3`
5. // много всего на http://cplusplus.com/

Алокаторы памяти

ОПТИМИЗИРУЕМ ОПЕРАЦИЮ ДОСТУПА К ПАМЯТИ

Аллокаторы

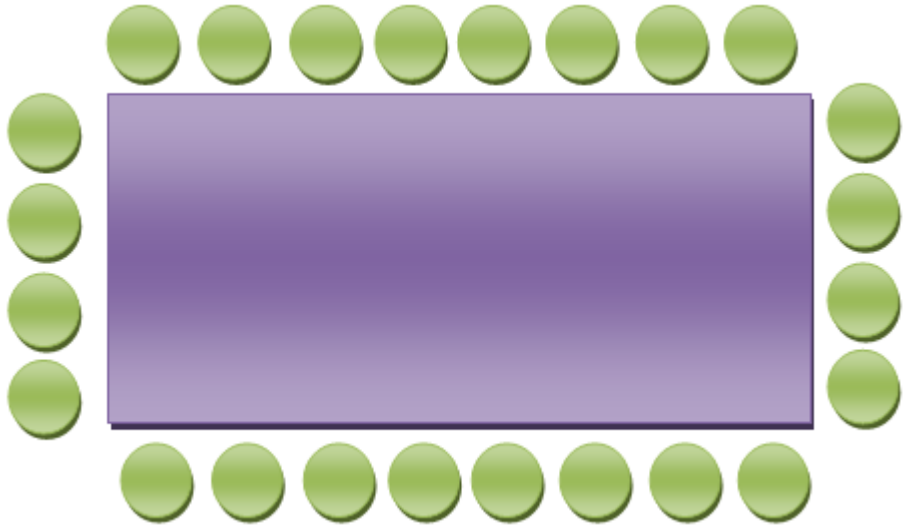
Каждый контейнер имеет определенный для него *аллокатор* (allocator). Аллокатор управляет выделением памяти для контейнера. Аллокатором по умолчанию является объект класса allocator, но вы можете определять свои собственные аллокаторы, если это необходимо для специализированных приложений. Для большинства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

Классическая история- ресторан

Предположим у нас есть суши ресторан.

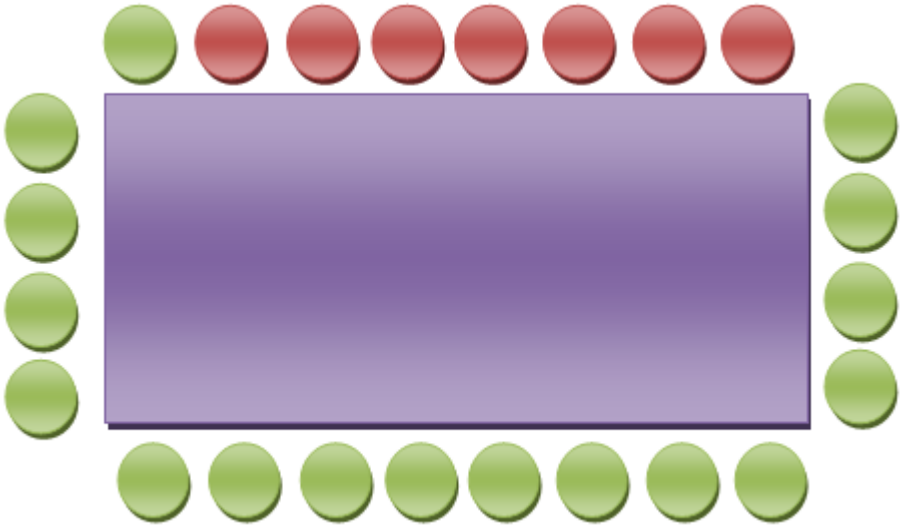
У нас есть стол и места, размещенные вокруг.

В наши обязанности входит размещение посетителей за столом.



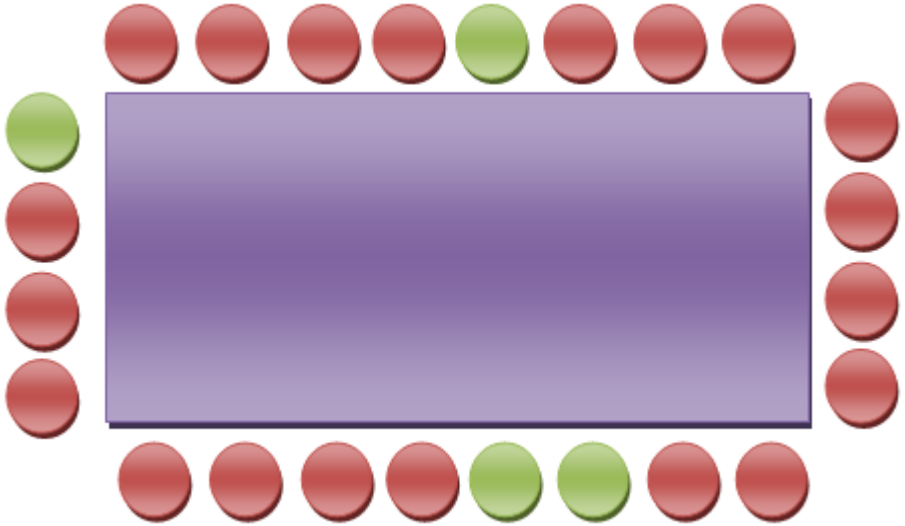
Назначаем места посетителям

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

Example74_OperatorNew

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:

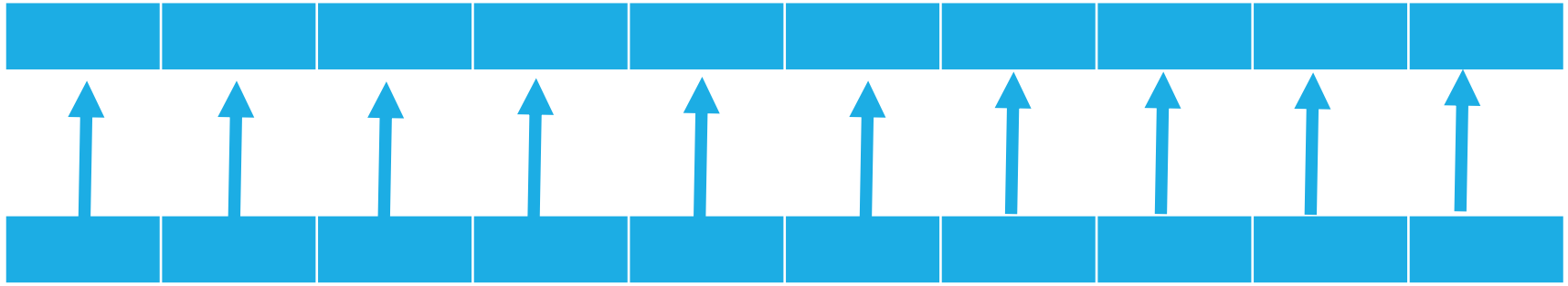
```
void *operator new(size_t size);

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.

Simple Allocator [1/4]

Used_blocks: Память для стркутур Item

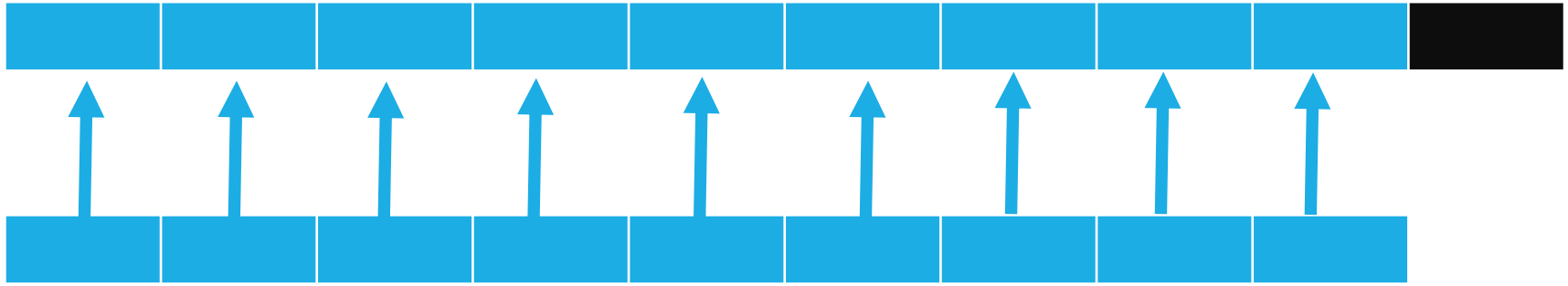


Free_blocks: Указатели на свободные блоки

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 Example75_SimpleAllocator

```
1.class TAllocationBlock {
2.public:
3.    TAllocationBlock(size_t size,size_t count);
4.    void *allocate();
5.    void deallocate(void *pointer);
6.    bool has_free_blocks();
7.    virtual ~TAllocationBlock();
8.private:
9.    size_t _size;
10.   size_t _count;
11.   char  *_used_blocks;
12.   void  **_free_blocks;
13.   size_t _free_count;
14. };
```

Использование аллокатора в vector
Example78_VectorAllocator

```
1.    typedef std::vector<Foo, Allocator < Foo>> vec;
2.    vec vector;
3.    // явное выделение памяти
4.    vector.reserve(10);
5.    std::istream_iterator<Foo> iter(std::cin);
6.    std::istream_iterator<Foo> eos;
7.    std::insert_iterator<vec> insert_iter(vector,vector.begin());
8.    std::copy(iter,eos,insert_iter);

9.     std::ostream_iterator<Foo> out(std::cout," ");
10.    std::copy(vector.begin(),vector.end(),out);
```

Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ