OTUS

# C++ developer. Professional
# Идея аллокаторов

otus.ru


**Проверить, идет ли запись**

# Меня хорошо видно && слышно?

Ставим “+”, если все хорошо
“-”, если есть проблемы

Тема вебинара

# Идея аллокаторов

**Арсений Черенков**

**senyacherenkov@yandex.ru**

# Правила вебинара

Активно
участвуем

Off-topic обсуждаем
в telegram

Задаем вопрос
в чат

Вопросы вижу в чате,
могу ответить не сразу

# Агенда

1. Понятие аллокатора
2. Контейнеры и аллокаторы
3. Реализация аллокатора (от C++98 до C++17)
4. Домашняя работа

# Умные указатели

... это про то, как управлять временем жизни объекта:

- передача владения
- подсчёт ссылок

```
auto customDeleter = [](MyStruct *s) { ... };

auto sharedSmartPtr = std::shared_ptr<MyStruct>(new MyStruct(10), customDeleter);
```

# Снова про heap

```
struct MyStruct
{
    MyStruct()
    {
        std::cout << "hello from MyStruct ctor" << std::endl;
    }
};
```

Можно так

```
MyStruct* arr = new MyStruct[NumberOfStructs];
```

А можно сяк

```
MyStruct* mem = ::operator new(NumberOfStructs * sizeof(MyStruct));
```

# Снова про heap

Можно так

```
MyStruct* arr = new MyStruct[NumberOfStructs];
```

А можно сяк

```
MyStruct* mem = ::operator new(NumberOfStructs * sizeof(MyStruct));

MyStruct* arr2 = new(mem) MyStruct[NumberOfStructs];
```

# Снова про heap

Можно так

```
MyStruct* arr = new MyStruct[NumberOfStructs];
```

new-expression

А можно сяк

```
MyStruct* mem = ::operator new(NumberOfStructs * sizeof(MyStruct));
```

operator new

```
MyStruct* arr2 = new(mem) MyStruct[NumberOfStructs];
```

placement operator new

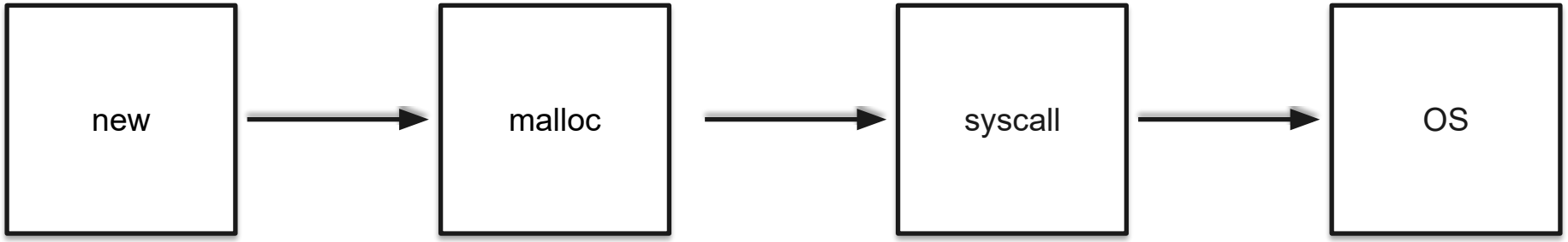
# Снова про heap

```
  void *   operator new (std::size_t sz)
{
    void *p;

    if((p = malloc (sz)) == 0)
    {
      new_handler handler = std::get_new_handler();
      if (! handler)
          throw bad_alloc();
      handler ();
    }

  return p;
}
```

# Снова про heap



# Контейнеры

... это в том числе про то, как управлять временем жизни объектов
А еще про то, как хранить объекты

- Линейно (std::vector)

```
std::vector<int> values;
for(int i=0; i < NumberOfInts; ++i)
   values.push_back(i);
```

Где будут храниться элементы

Насколько далеко друг от друга

# Интуитивное предположение

```
std::vector<int> values;

for(int i=0; i < NumberOfInts; ++i)
   values.push_back(i);
```

пустой вектор

динамическое выделение памяти

# Контейнеры

```
std::vector<int> values;


values.reserve(NumberOfInts);

for(int i=0; i < NumberOfInts; ++i)
   values.push_back(i);
```

# Контейнеры

```
std::vector<int> values;
```

пустой вектор

```
values.reserve(NumberOfInts);
```

Выделяется память под 10 элементов

```
for(int i=0; i < NumberOfInts; ++i)
   values.push_back(i);
```

Элементы конструируются в выделенной памяти

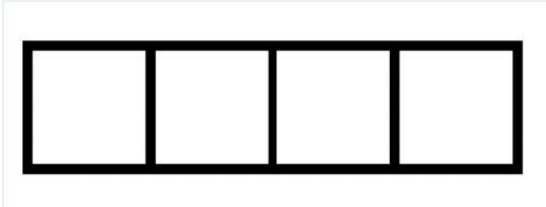
# Вопрос

Что если мы хотим:

- Трассировать работу с памятью
- Хотим выделять память в GPU
- Изменить стратегию выделения памяти

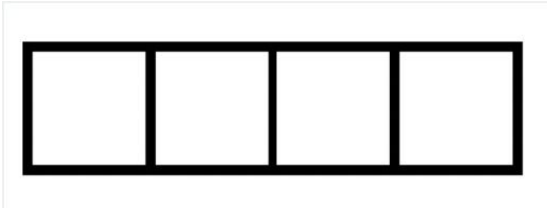
# Фрагментация памяти

1. Представим, что у нас всего 4 байта памяти.



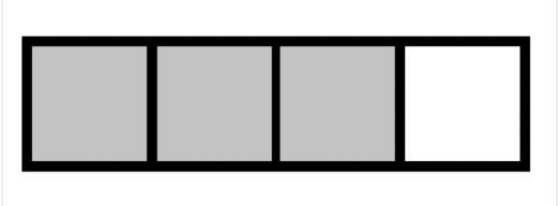
# Фрагментация памяти

1. Представим, что у нас всего 4 байта памяти.



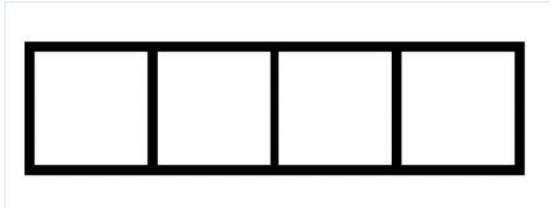
```
auto value1 = new uint8_t{42};
auto value2 = new uint8_t{42};
auto value3 = new uint8_t{42};
```

2. Заняли 3 байта из 4-ех. Свободен 1 байт.



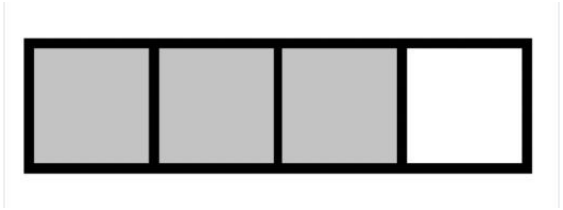
# Фрагментация памяти

1. Представим, что у нас всего 4 байта памяти.



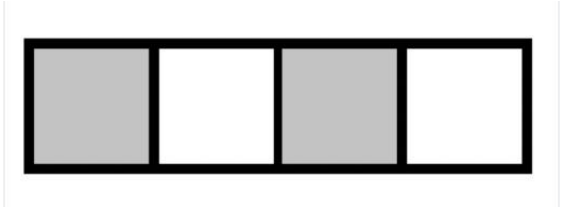
```
auto value1 = new uint8_t{42};
auto value2 = new uint8_t{42};
auto value3 = new uint8_t{42};
```

2. Заняли 3 байта из 4-ех. Свободен 1 байт.



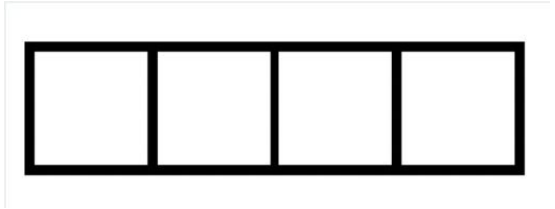
```
delete value2;
```

3. Освободили 1 байт. Свободно 2 байта



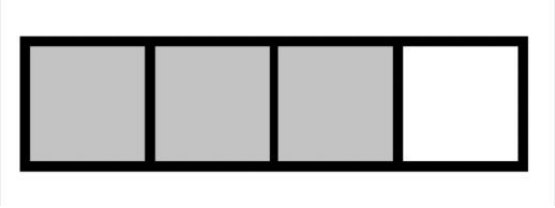
# Фрагментация памяти

1. Представим, что у нас всего 4 байта памяти.



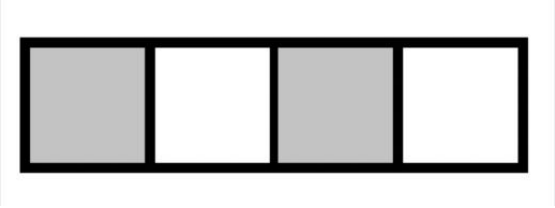
```
auto value1 = new uint8_t{42};
auto value2 = new uint8_t{42};
auto value3 = new uint8_t{42};
```

2. Заняли 3 байта из 4-ех. Свободен 1 байт.



```
delete value2;
```

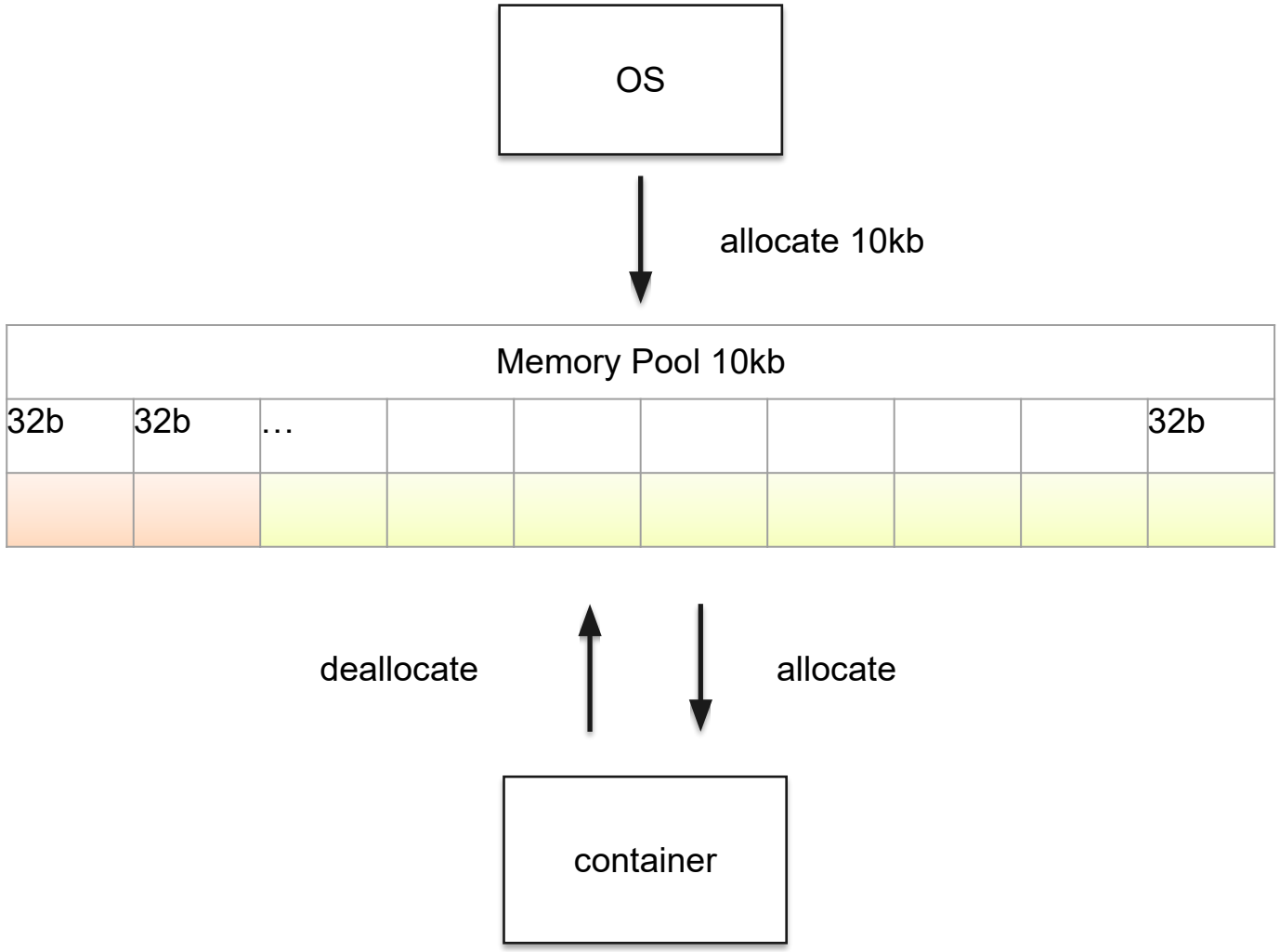
3. Освободили 1 байт. Свободно 2 байта



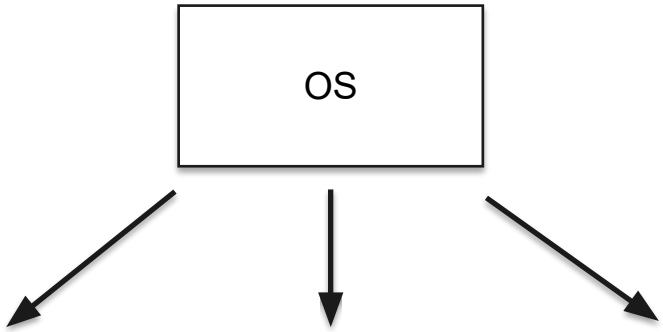
```
auto value4 = new uint16_t{42};
```

**4. Памяти не хватает!**

# Memory Pool


OS
allocate 10kb
Memory Pool 10kb
32b
32b
…
32b
deallocate
allocate
container

# Memory Pool




| Memory Pool 1kb | | | |
|---|---|---|---|
| 1b | 1b | … | 1b |
| | | | |

| Memory Pool 1kb | | | |
|---|---|---|---|
| 2b | 2b | … | 2b |
| | | | |

| Memory Pool 1kb | | | |
|---|---|---|---|
| 4b | 4b | … | 4b |
| | | | |

# Как быть?

- Переопределить глобально operator new

```
void* operator new(std::size_t sz)
{
   static constexpr std::size_t poolSize = 1000;
   static int allocatedSize = 0;
   static void* pool = std::malloc(poolSize);

   if (pool != nullptr && sz <= (poolSize - allocatedSize))
   {
      auto ptr = (char*)pool + allocatedSize;
      allocatedSize += sz;
      return ptr;
   }

   throw std::bad_alloc{};
}
```

# Google TCMalloc

https://github.com/google/tcmalloc

# Аллокатор

... это про то, как выделять память под объекты (а также освобождать)

- в какой памяти
- по каким адресам
- …

```
template<class T, class Allocator = std::allocator<T>>
class vector;
```

# Аллокатор

... это про то, как выделять память под объекты (а также освобождать)
в какой памяти
по каким адресам
…

```
template<class T, class Allocator = std::allocator<T>>
class vector;

std::vector<int, std::allocator<int>> v;
```

# Аллокатор

... это про то, как выделять память под объекты (а также освобождать)
в какой памяти
по каким адресам
…

```
template<class Key,
        class T,
        class Compare = std::less<Key>,
        class Allocator = std::allocator<std::pair<const Key, T>>
class map;
```

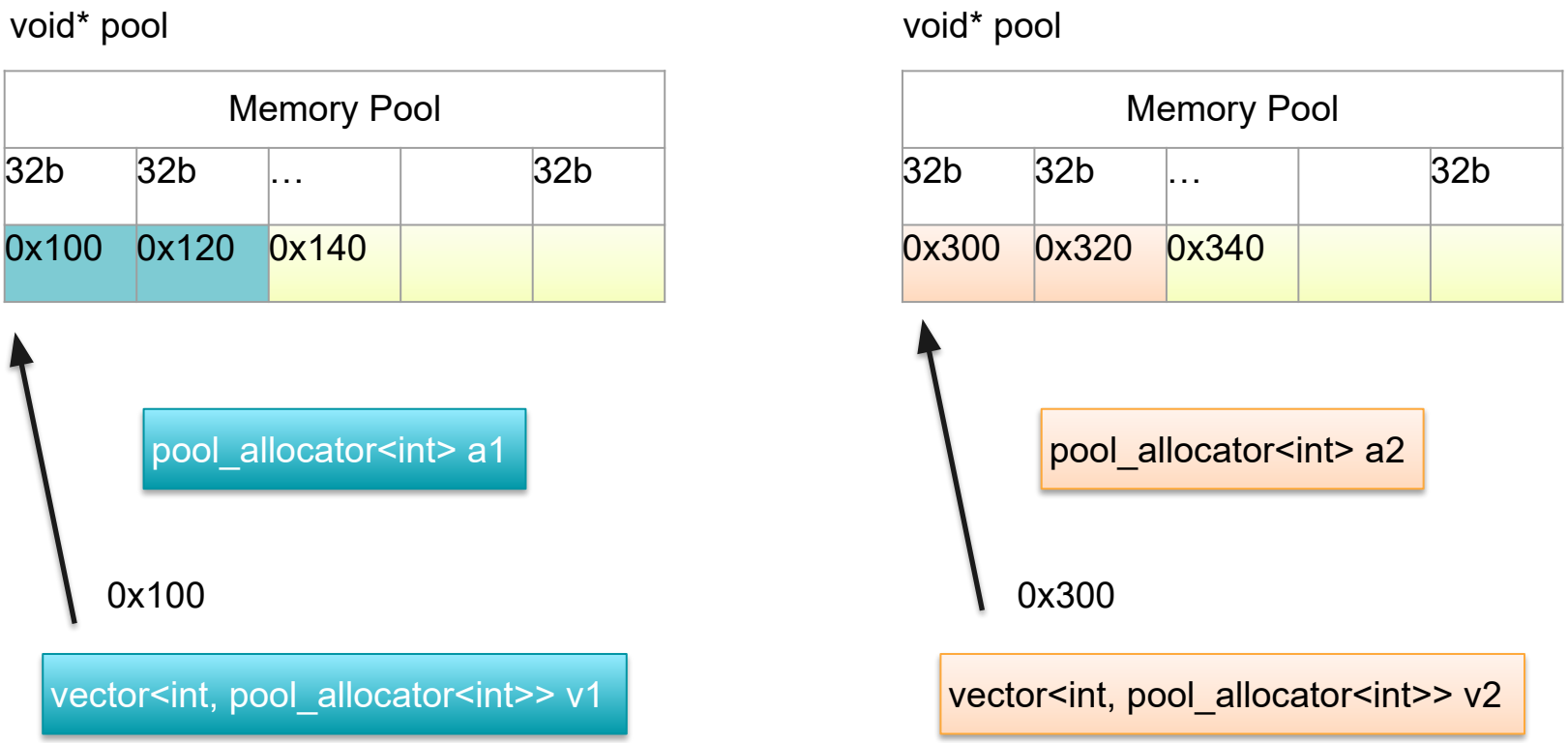
# Зачем?

- Абстрагирование логики контейнера от модели памяти
- Разделение процессов управления памятью и инициализации объектов
- Возможность по реализации пользовательских стратегий управления памятью

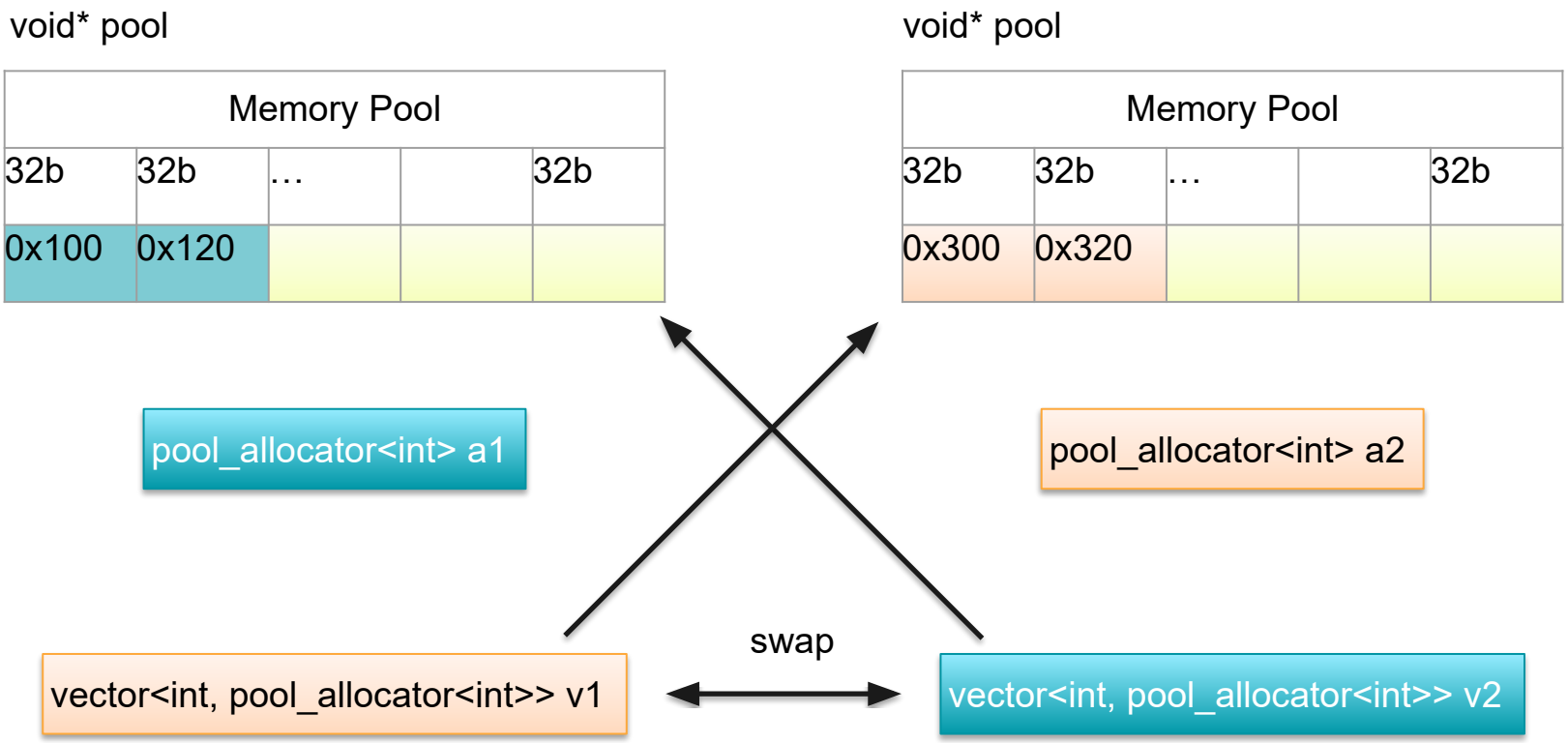
# C++03x std::allocator

- stateless
- Реализовывает
    - Allocate - вызов operator new
    - Construct - вызов placement operator new и как результат конструктора
    - Deallocate - вызов delete
    - Destroy - вызов деструктора

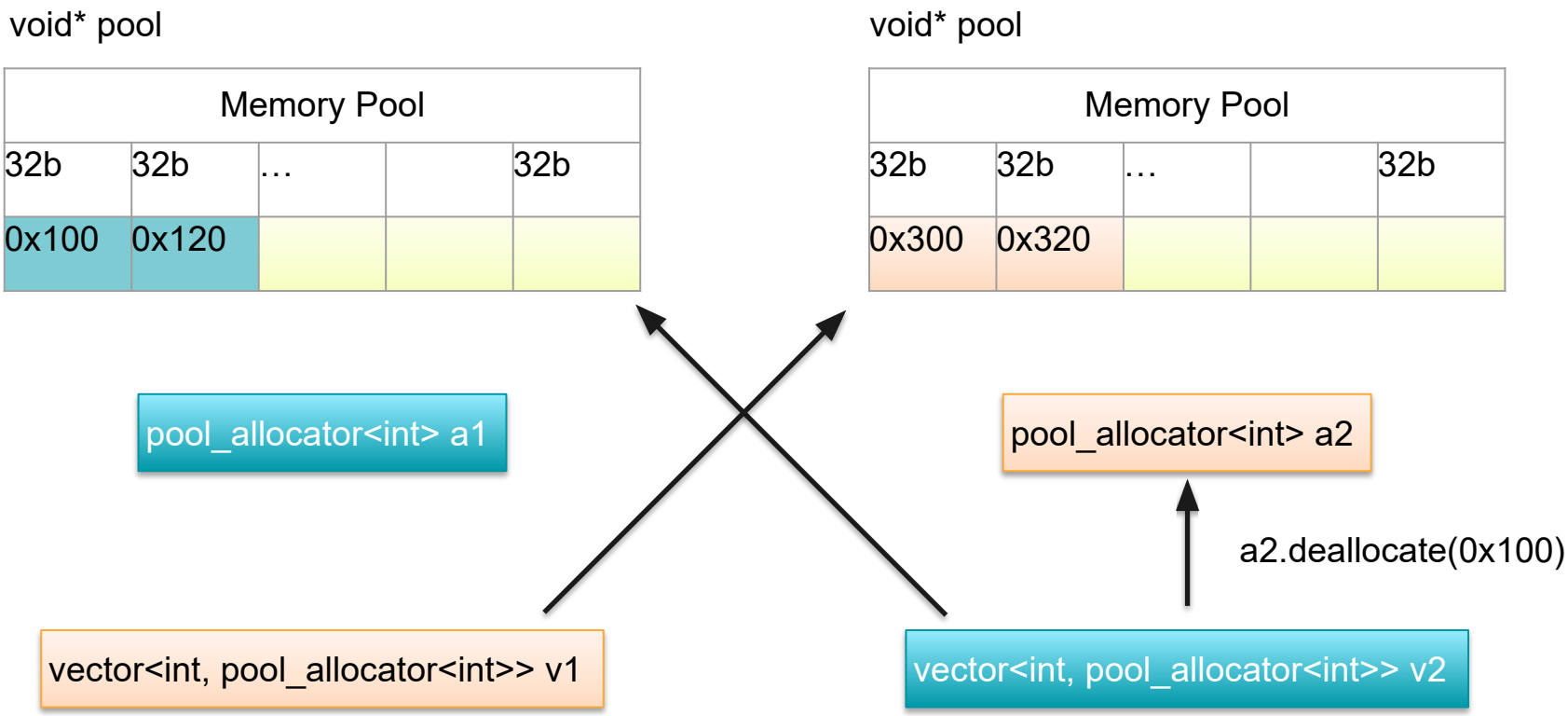
# Stateful allocator



# Stateful allocator



# Stateful allocator



# Краткая эволюция аллокаторов

## C++03

template<typename T> class allocator

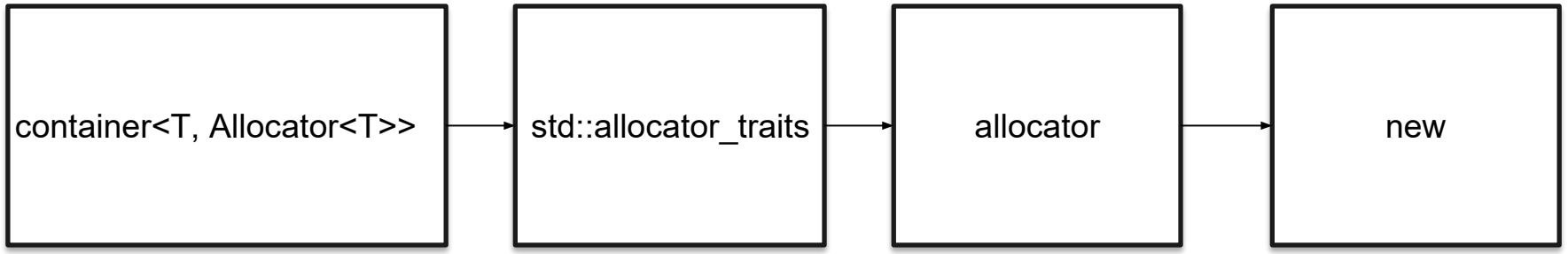
- 7 typedefs
- 1 вложенный шаблон
- 2 конструктора
- 7 функций
- 2 оператора

## C++11

template<typename T> class allocator

- 1 typedef
- 2 конструктора
- 2 функции
- 2 оператора

# C++11 std::allocator_traits



# C++11 AllocatorAwareContainer

https://en.cppreference.com/w/cpp/named_req/AllocatorAwareContainer

```
template <class Alloc> struct  allocator_traits
{
     static Alloc select_on_container_copy_construction(const Alloc& a)
     {
          return a.select_on_container_copy_construction(); | a;
     }
...
};


vector(const vector& other)
:alloc(std::allocator_traits:: select_on_copy_construction(other.get_allocator()))
{}
```

# C++11 AllocatorAwareContainer

https://en.cppreference.com/w/cpp/named_req/AllocatorAwareContainer

```
template <class Alloc> struct  allocator_traits
{
...
    using propagate_on_container_copy_assignment
    = Alloc::propagate_on_container_copy_assignment | std::false_type;

    using propagate_on_container_move_assignment
    = Alloc::propagate_on_container_move_assignment | std::false_type;

    using propagate_on_container_swap
    = Alloc::propagate_on_container_swap | std::false_type;
};
```

# C++11 Allocator

- Дефолтные реализации (e.g. construct, destroy) возьмутся из std::allocator_traits
- Можно реализовать как stateful

# C++17 polymorphic_allocator

- Использует динамический полиморфизм
- Реализован на основе имеющегося интерфейса аллокатора
- Переопределение стратегии управления памятью реализовывается при помощи интерфейса memory_resource

# Подводим итоги

**C++98/03**

stateless allocator

**C++11**

std::allocator_traits
allocator aware containers

**C++17**

std::pmr::polymorphic_allocator
std::pmr::memory_resource
std::pmr::vector<T>

# Спасибо за внимание!

# Вопросы?

Ставим “+”,
если вопросы есть

Ставим “–”,
если вопросов нет

# Не забудьте принять участие в опросе

# Следующий вебинар

22 сентября 2023

## Шаблонная магия

- Ссылка на вебинар будет в ЛК за 15 минут
- Материалы к занятию в ЛК — можно изучать
- Обязательный материал обозначен красной лентой