АЛЛОКАТОРЫ

Управление памятью и тонкая настройка контейнеров

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

- Плохая репутация аллокаторов
- Локальный (arena-based) подход
- Полиморфные аллокаторы
- Собственный pmr контейнер

Исторический вопрос

- Зачем вообще в C++98 были введены аллокаторы?
- От самых первых реализаций, Степанов спланировал вектор таким образом:

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

- Но это **странно**. Зачем вектору аллокатор в 1992-м году?
- В те времена распределение памяти считалось прерогативой операционной системы
- В языке C++ оно было инкапсулировано в operator new.

Исторический вопрос

- Зачем вообще в C++98 были введены аллокаторы?
- Исходно Степанов планировал аллокаторы для абстракции различий между near и far pointers. Вендоры предоставляли расширения, но стандарт языка понимал только T*, а не T __huge*.
- Идея в следующем: каждый раз когда контейнеру нужна память он пользуется функцией allocate своего аллокатора
- Аллокатор, в свою очередь, знает две вещи:
  - Откуда взять память
  - Как преобразовать её к T*
- Таким образом, аллокаторы никогда не планировались как механизм выделения памяти, а только как адаптер к выделителю.

Как это было

- Представим, что у вас в программе есть особый распределитель памяти s_malloc и вы пишете аллокатор с функциями allocate и deallocate:

```
template<typename T> struct s_alloc {
  typedef T value_type;
  typedef T* pointer;
  pointer allocate (size_t n) {
    return static_cast<pointer>(s_malloc(n * sizeof(T)));
  }
  void deallocate(pointer p, size_t n) { s_free(p); }
};
```

- Теперь это, наверное, можно использовать, размещая нечто в этой памяти?

```
template <typename T> using s_vector = vector<T, s_alloc<T>>;
```

О нет, секундочку…

- У нас две проблемы, которые в целом стали понятны к 1998 году.
- Первая проблема: взаимозаменяемость аллокаторов

```
vector<int, s_alloc<int> > v1, v2;
// тут много кода
v1 = v2; // что должно произойти тут с v1.alloc и v2.alloc?
```

- Вторая проблема: приведение аллокаторов
  - Возмём `std::list<T, s_alloc<T> >`
  - Внутри себя список будет создавать не T, а `__list_node<T>`.
  - То есть ему нужно иметь возможность получить `s_alloc<__list_node<T> >` который является совершенно отдельным типом

Взаимозаменяемость аллокаторов

- Стандарт 98 года накладывал ограничение: все конкретные экземпляры любого типа аллокаторов должны быть эквивалентными

```
template <typename T, typename U>
bool operator== (const s_alloc<T>&, const s_alloc<U>&) {
  return true;
}

template <typename T, typename U>
bool operator!= (const s_alloc<T>&, const s_alloc<U>&) {
  return false;
}
```

- Это снимало проблему взаимозаменяемости при операциях вида `v1 = v2`

Приведение и rebind

- Для приведения аллокаторов одного к другому и чтобы контейнер мог узнать тип аллокатора для чего-то кроме своего типа

```
template<typename T> struct s_alloc {
  // тут всё как было

  template<typename U> s_alloc(const s_alloc<U>&) {}

  template<typename U>
  struct rebind { typedef s_alloc<U> other; };

};
```

- Это снимает проблему приведения из allocator<T> в allocator<__list_node<T>>
- Может быть это всё?

Некоторые дополнительные функции

- Нет в 98-м году это ещё не всё. Вы также обязаны были предоставить функции construct и destroy, выполняющие функции размещающего new и явного вызова деструктора

```
void construct(pointer p, const T& t) {  new(p) T(t); }

void destroy(pointer p) { p->~T(); }
```

- И функцию max_size для ограничения общего размера (а также кучу typedefs для reference, const_pointer и всего такого)

```
size_type max_size() const {
  return numeric_limits<size_type>::max() / sizeof(T);
}
```

- Всё это было deprecated в 2017 году, многое было перенесено в traits в 2011

Case study

- Логирующий аллокатор в реалиях 1998 года.

```
vector<int, logging_alloc<int> > v;

for (int i = 0; i < 16; ++i)
  v.push_back(i);

vector<int, logging_alloc<int> > v2 = v;

v2.push_back(16);

v = v2;

list<int, logging_alloc<int> > l(v.begin(), v.end());
```

Weasel words

- Две фразы из C++03 (20.1.5.4), являющиеся по словам Алисдара Мередита причиной почему компания Блумберг присоединилась к комитету.

- `Implementations of containers described in this International Standard are permitted to assume that`

- **`All instances of a given allocator type are required to be interchangeable and always compare equal to each other.`**

- **`The typedef members pointer, const_pointer, size_type, and difference_type are required to be T*, T const*, std::size_t, and std::ptrdiff_t, respectively`**

- Запрещено состояние и умные указатели

Обсуждение

- Много ли полезных применений аллокаторов, не обладающих состоянием и возвращающих голые указатели вы можете придумать?

C++11 спешит на помощь!

- В стандарте C++11 и в следующем C++14 аллокаторам разрешили обладать состоянием
- Первое, что было добавлено это переопределение равенства для аллокаторов. Теперь два аллокатора считаются равными если один может освободить то, что аллоцировал другой и наоборот.
- Кроме того, аллокаторам разрешили чтобы pointer был не равен T*
- И, наконец, аллокаторам сделали класс allocator_traits куда собрали все редко переопределяемые вещи в качестве разумных значений по умолчанию.
- Дополнительно сделали класс pointer_traits

Характеристики аллокаторов

- Класс allocator_traits содержит:
  - Переопределения типов: value_type, pointer, const_pointer и прочие
  - Селекторы propagate_on_xxx
  - Шаблоны rebind_alloc и rebind_traits
  - Функции allocate/deallocate, а также construct и destroy
- Определение для собственного класса тривиально (и обычно не нужно)

```
namespace std {
  template <> struct allocator_traits<s_alloc> {
    // тут нужно определить все traits
  };
}
```

Характеристики аллокаторов

- Обычно достаточно определить что-либо в собственном аллокаторе и это будет "подхвачено" характеристикой
- На псевдо-коде (в реальности там SFINAE) это можно написать так:

```
template <typename Alloc> class allocator_traits {
  using value_type = typename Alloc::value_type;
  using pointer = typename Alloc::pointer | value_type*
  using const_pointer = typename Alloc::const_pointer |
      pointer_traits<pointer>::rebind<const value_type>::type;
  ... и так далее ...
```

- В итоге реальной необходимости специализировать traits нет

Характеристики аллокаторов

- Особенно интересно была решена проблема allocate vs construct

```
template <typename Alloc> class allocator_traits {
  // тут всё остальное
  static pointer allocate(Alloc &a, size_type n)
    { return a.allocate(n); }
  template <typename T, typename ... Args>
  static void construct(Alloc &a, T *p, Args&& ... args)
    { a.construct(p, forward<Args>(args)...) ||
      new (static_cast<void*>(p)) T(forward<Args>(args)...); }
```

- allocate работает с `pointer`, а construct – с типом `T*`.

Case study: free list allocator

- Идея free list аллокатора: освобождать блоки не в глобальный аллокатор а во free list

```
list<int, freelist_alloc<int>> l(v.begin(), v.end());

l.remove(2); // тут никакого настоящего удаления
l.remove(6);

l.insert(l.begin(), -1); // тут никакого настоящего выделения
l.insert(l.begin(), -3);
```

- В программах с частой вставкой и удалением, это может много сэкономить
- Case study: `02-freelist.cc`

Обсуждение

- Может ли присваивание менять состояние аллокатора?
- В нашем примере мы видим

```
freelist_alloc& operator= (const freelist_alloc&) noexcept {
  return *this; // "оставить старый"
}
```

Но при этом

```
freelist_alloc& operator= (freelist_alloc&& other) noexcept {
  // тут сложная очистка и замена
}
```

- Обязательная ли это программа или может быть иначе?

Копирование аллокаторов

- Есть выбор из двух вариантов
  - Либо все аллокаторы разделяют какой-нибудь ресурс
  - Либо они не копируются (говорят также "не пропагируются на копировании")
- И действительно, рассмотрим freelist example
- Если он будет копироваться, то либо мы попадаем на двойное владение, либо нам придётся произвести массу ненужных удалений и выделений памяти
- В общем случае, запрещено может быть любое присваивание

Способ указать поведение

- Как именно аллокатор будет вести себя на копировании или перемещении определяется начиная с C++11 следующими typedefs

```
using propagate_on_container_copy_assignment = false_type;
using propagate_on_container_move_assignment = false_type;
using propagate_on_container_swap            = false_type;
using is_always_equal                        = false_type;
```

- Да, они специально так уродливы.
- Я буду их называть *POCCA*, *POCMA*, *POCS*, *IAE*
- Переопределяя любой из них в `true_type`, вы, скорее всего, знаете, что делаете.

- Плохая репутация аллокаторов
  - Локальный (arena-based) подход
- Полиморфные аллокаторы
- Собственный pmr контейнер

Постановка задачи: small vector

- Полный запрет копирования и перемещения создаёт локальные аллокаторы. Они привязываются к конкретному месту (вектору, строке, отображению, ….)
- Вектор, оптимизированный на небольшое количество элементов (так называемый small vector) это типичный пример полезного использования аллокаторов

```
template <class T, size_t BufSize = 200>
using SmallVector = vector<T, short_alloc<T, BufSize, alignof(T)>>;
```

- Такой вектор располагается на стеке пока в нём меньше чем BufSize байт и перелоцируется в кучу, когда места на стеке не хватает
- Аллокатор short_alloc, рассматриваемый в этом разделе, предложен Говардом Хинантом (см. список литературы)

Арена

- Ареной называется класс, управляющий локальным ресурсом

```
template <size_t N, size_t alignment = alignof(max_align_t)>
class arena {
  char buf_[N] alignas(alignment);
  char* ptr_;

public:
  arena() noexcept : ptr_(buf_) {}
  arena(const arena&) = delete;
  arena& operator=(const arena&) = delete;

  template <size_t ReqAlign> char* allocate(size_t n);
  void deallocate(char* p, size_t n) noexcept;
```

Стратегия аллокации и деаллокации

- При аллокации используется буфер либо глобальный new

```
template <size_t N, size_t alignment> template <size_t ReqAlign>
char *arena<N, alignment>::allocate(size_t n) {
 auto const aligned_n = align_up(n);
 auto bsz = static_cast<decltype(aligned_n)>(buf_ + N - ptr_);
 if (bsz < aligned_n)
   return static_cast<char*>(::operator new(n));
 char* tmp = ptr_;
 ptr_ += aligned_n;
 return tmp;
}
```

- При деаллокации выделенное глобальным new возвращается через delete

Аллокатор short_alloc

- Связывает конкретную арену с интерфейсом аллокатора

```
template <class T, size_t N, size_t A = alignof(max_align_t)>
class short_alloc {
 arena<N, A>& a_;

public:
  short_alloc(arena_type& a) noexcept : a_(a) {}

  T* allocate(size_t n) {
    char *res = a_.allocate<alignof(T)>(n * sizeof(T));
    return reinterpret_cast<T*>(res);
  }

// и так далее
```

Использование

- Выделение в явном виде арены

```
SmallVector<int>::allocator_type::arena_type a;
```

- В ней вектор на стеке

```
SmallVector<int> v{a};
```

- Далее использование как обычного вектора

Обсуждение

- Классическое SSO обходится вовсе без явной арены
- С другой стороны, явная арена это удобно, так как размер контейнера отвязан от его места для аллокации
- Что вы считаете лучшей идеей?

Экземпляры аллокаторов

- Коль скоро у аллокаторов есть состояние, одного их типа более не достаточно, важен конкретный экземпляр.

```
using CustomStr =
  string <char, char_traits<char>, CustomAlloc<char>>;

CustomAlloc<char> alloc1(SYSTEM), alloc2(LOCAL), alloc3;
```

- Здесь `alloc1`, `alloc2` и `alloc3` имеют принципиально разное состояние.

```
CustomStr x1(alloc1), x2(alloc2), x3(alloc3);
```

- Строки `x1`, `x2` и `x3` аллоцируются разными аллокаторами одного типа.
- И это порождает проблемы.

Проблемы scope у аллокаторов

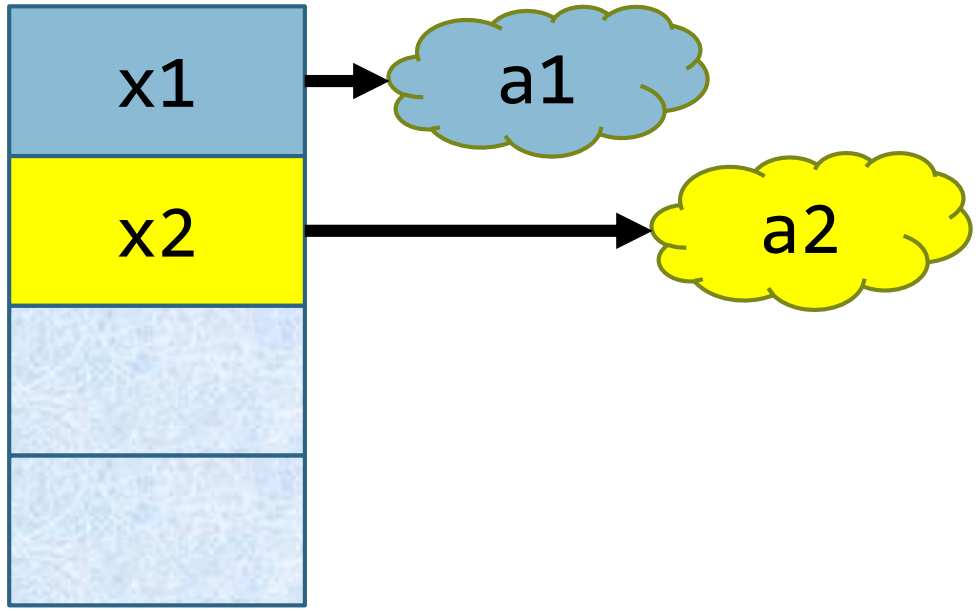
- Останемся в рамках прошлого слайда

```
CustomStr x1(alloc1), x2(alloc2), x3(alloc3);
```

- Попробуем разместить их в вектор

```
vector<CustomStr> vec;

vec.push_back(x1);
vec.push_back(x2);
vec.reserve(4);
```



Проблемы scope у аллокаторов

- Останемся в рамках прошлого слайда

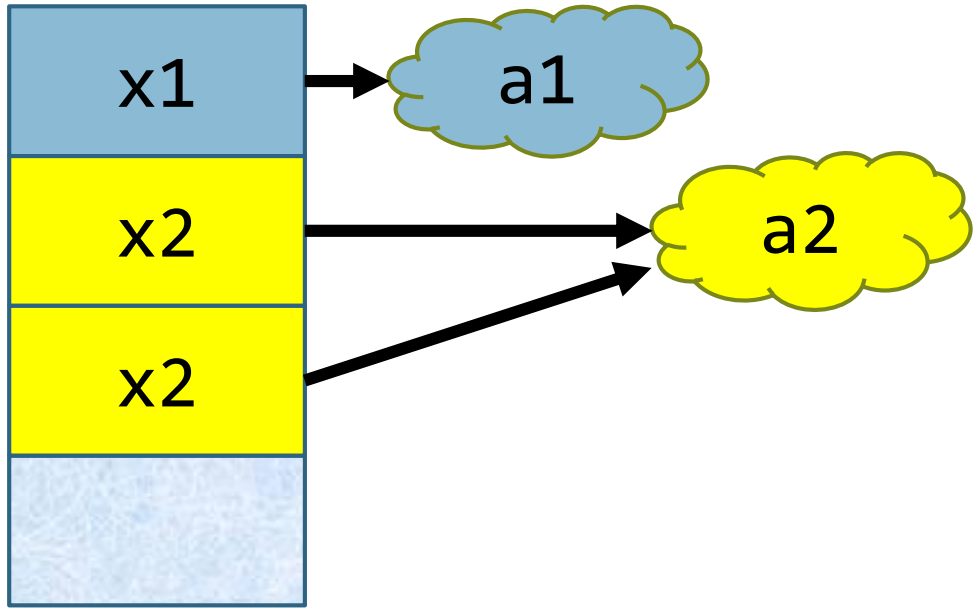
```
CustomStr x1(alloc1), x2(alloc2), x3(alloc3);
```

- Попробуем разместить их в вектор

```
vector<CustomStr> vec;

vec.push_back(x1);
vec.push_back(x2);
vec.reserve(4);

vec.insert(vec.begin(), x3);
```



Проблемы scope у аллокаторов

- Останемся в рамках прошлого слайда

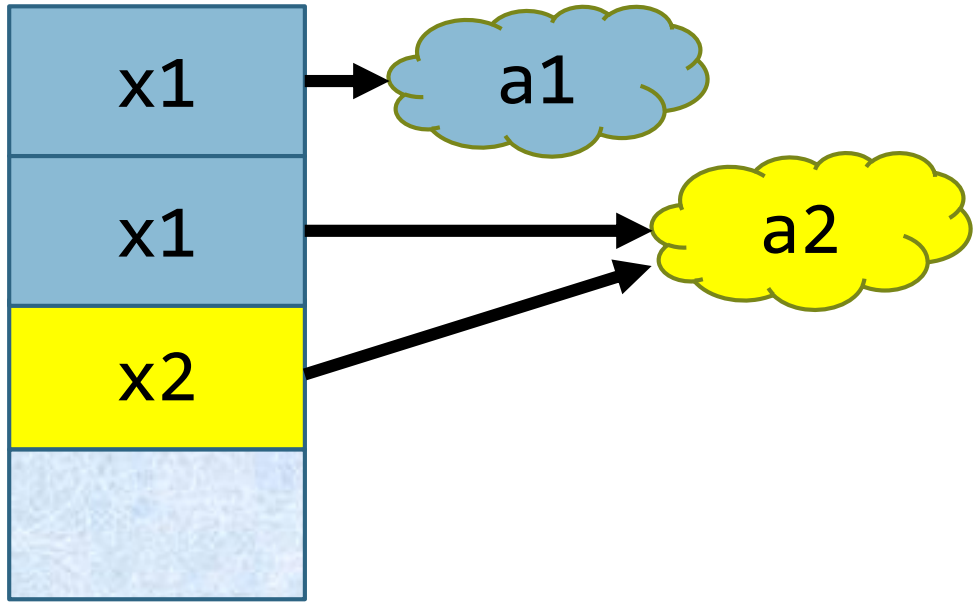
```
CustomStr x1(alloc1), x2(alloc2), x3(alloc3);
```

- Попробуем разместить их в вектор

```
vector<CustomStr> vec;

vec.push_back(x1);
vec.push_back(x2);
vec.reserve(4);

vec.insert(vec.begin(), x3);
```



Проблемы scope у аллокаторов

- Останемся в рамках прошлого слайда

```
CustomStr x1(alloc1), x2(alloc2), x3(alloc3);
```

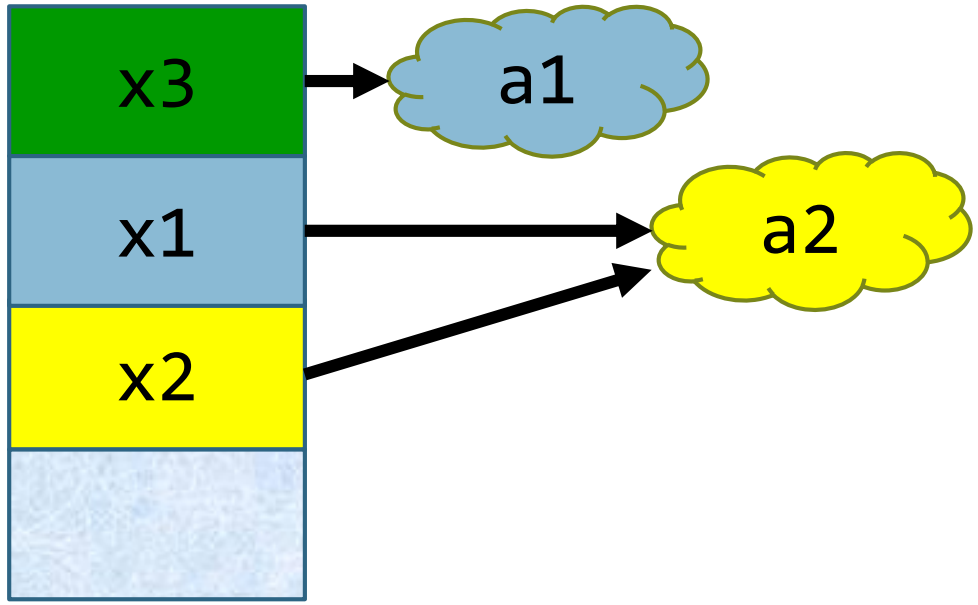
- Попробуем разместить их в вектор

```
vector<CustomStr> vec;

vec.push_back(x1);
vec.push_back(x2);
vec.reserve(4);

vec.insert(vec.begin(), x3);
```

- При этом память на сам вектор добывается по старинке из std::allocator



Решение в C++11

- Хотелось бы чтобы контейнер при создании элементов спрашивал "ты используешь аллокатор?" и если да, то передавал свой.

```
namespace cust {
  template <typename T> CustomAlloc;

  template <typename T> using alloc =
     ::std::scoped_allocator_adapter<CustomAlloc<T>>;
 template <typename T> using vector = ::std::vector<T, alloc<T>>;
 template <typename T> using string =
     ::std::basic_string<char, char_traits<char>, alloc<char>>;
};

cust::vector<cust::string> vs(&alloc1); // propagated ok
```

Обсуждение

- Какие проблемы бросаются в глаза?

Обсуждение

- Какие проблемы бросаются в глаза?
- Код загрязняется вирусными шаблонами даже для обычных аллокаторов.

```
map <int, float, less<int>, s_alloc<pair<const int, float>>> m;
basic_string<char, char_traits<char>, s_alloc<char>> s;
```

- Scoped адаптер это классно, но это может очень серьёзно увеличить писанину шаблонов: в более-менее серьёзном проекте туда придётся руками затаскивать не только string и vector, а вообще всё.
- Кроме того, излишняя ортогональность пропагирования затрудняет написание контейнеров (см. следующий слайд)

Аргумент от контейнера

- Написать даже обычный move-assignment в этих условиях нелегко

```
container& operator= (constainer &&rhs) {
  if (alloc_traits::POCMA) {
    this->clear_and_deallocate_memory();
    this->alloc_ = move(rhs.alloc_);
    this->impl_ = move(rhs.impl_);
  }
  else if (alloc_traits::IAE || alloc_ == rhs.alloc_) {
    this->clear_and_deallocate_memory();
    this->impl_ = move(rhs.impl_);
  }
  else {
    this->assign(move_iterator(rhs.begin()), move_iterator(rhs.end());
  }
  return *this
}
```

Рациональная идея

- Вполне возможно, следует сделать ещё один шаг: аллокатор вообще не должен быть частью типа контейнера.
- Он должен быть всегда scoped
- Он никогда не должен копироваться и перемещаться

- Плохая репутация аллокаторов
- Локальный (arena-based) подход
- Полиморфные аллокаторы
- Собственный pmr контейнер

Вернёмся к пройденному

- Когда мы проектировали наивный аллокатор мы спроектировали его просто

```
template<typename T> struct memory_resource {
  T* allocate (size_t n);
  void deallocate(T* p, size_t n);
};
```

- Время показало, что полезна также функция is_equal особенно если мы разрешаем состояние
- Что если отсюда удалить типы?

Абстракция ресурса в памяти

- Если убрать типы, то мы должны вручную передать alignment и сделать этот класс виртуальной базой

```
struct memory_resource {
  virtual void* allocate (size_t n,
    size_t align = alignof(std::max_align_t)) = 0;
  virtual void deallocate(void* p, size_t n) = 0;
  virtual bool is_equal(const memory_resource&) const = 0;
};
```

- Это очень плохой дизайн. Что вам не нравится здесь?

Абстракция ресурса в памяти

- Если убрать типы, то мы должны вручную передать alignment и сделать этот класс виртуальной базой

```
struct memory_resource {
  virtual void* allocate (size_t n,
    size_t align = alignof(std::max_align_t)) = 0;
  virtual void deallocate(void* p, size_t n) = 0;
  virtual bool is_equal(const memory_resource&) const = 0;
};
```

- Это очень плохой дизайн. Что вам не нравится здесь?
- Параметр по умолчанию в виртуальной функции это очень нехорошо. Он свяжется статически, а не динамически.

Абстракция ресурса в памяти

- Чтобы решить эту проблему, давайте сделаем ресурс в памяти с использованием идиомы NVI

```
struct memory_resource {
  void* allocate(size_t n, size_t align = alignof(max_align_t));
  void deallocate(void* p, size_t n);
  bool is_equal(const memory_resource&) const noexcept;

protected:
  virtual void* do_allocate(size_t n, size_t align) = 0;
  // и остальные две так же
};
```

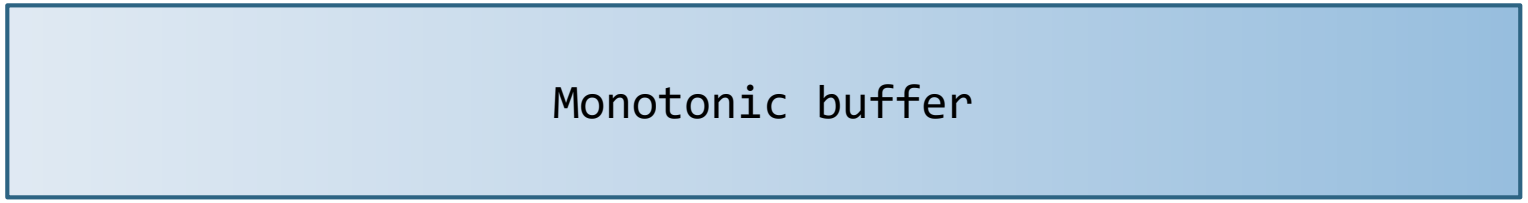
- Теперь всё почти готово к использованию

Существующие в стандарте ресурсы

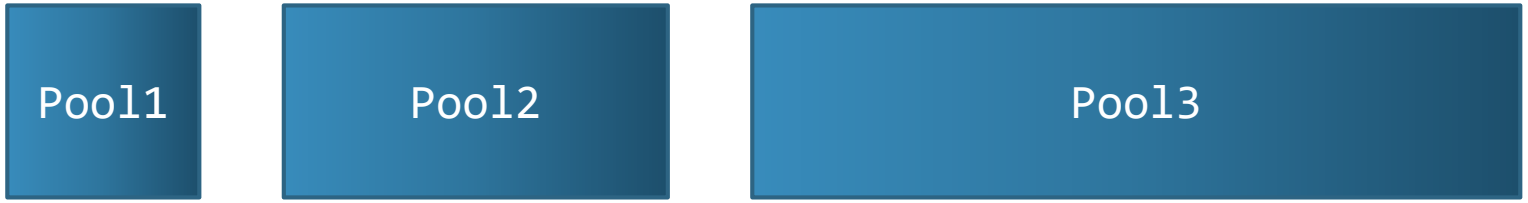
- Теперь, когда существует memory_resource, от него можно наследовать
  - `null_memory_resource` – самый интересный ресурс, всегда nullptr
  - `new_delete_resource` – стандартный ресурс с new/delete
  - `synchronize_pool_resource` – мультипул с многопоточной синхронизацией
  - `unsynchronize_pool_resource` – быстрый мультипул без синхронизации
  - `monotonic_buffer_resource` – монтонное выделение
- Тут встречаются два новых термина – мультипул (multipool) и монотонное (monotonic) выделение
- Это две стратегии работы с памятью, настолько себя зарекомендовавшие, что их предложили в стандарт

Monotonic & multipool

- Монотонный ресурс это ресурс, который монотонно выделяет память внутри некоего заранее выделенного буфера. Память не освобождается, в конце работы прибивается сам буфер (стоит памяти при частых аллокациях)



- Мультипул ресурс это несколько связанных пулов, в которые выделяется и освобождается память. Пулы преаллоцируются и при нехватке, выделяется больший и больший (ускоряет работу при аллокациях/деаллокациях)



Интригующий пример

- Следующий пример почти возможен в C++17

```
constexpr size_t sz = 1000 * sizeof(double);
char buffer[sz] alignas(double);

pmr::monotonic_buffer_resource alloc(buffer, sz);

double start = 0.0;
pmr::vector<double> v1(&alloc);
generate_n(back_inserter(v1), 100,
  [start] () mutable { return (start += 1.1); });
```

- В этом примере средне интригующее то, что generate_n не вызывает никаких аллокаций памяти, см. case study `04-memresource.cc`
- Но есть и ещё более интригующий вопрос: что такое pmr::vector?

Абстракция аллокатора

- К сожалению, `memory_resource`, каким он введён в C++17 не совпадает с интерфейсом аллокаторов, которые уже есть во всех контейнерах
- Идея сделать к нему адаптер, который является честным C++11 аллокатором

```
template<typename T> struct polymorphic_allocator {

  polymorphic_allocator();
  polymorphic_allocator(memory_resource *mr);

  T* allocate (size_t n);
  void deallocate(T* p, size_t n);
  // и так далее
};
```

Особенности polymorphic_allocator

- Всегда scoped

```
template <class Tp> class polymorphic_allocator:
    public scoped_allocator_adaptor<__details::polymorphic_allocator_imp<Tp>>
```

- Содержит указатель на memory_resource

```
private: memory_resource* memory_rsrc;
```

- При этом копирующий конструктор копирует этот указатель

```
polymorphic_allocator(const polymorphic_allocator& other) = default;
```

- Запрещает копирующее присваивание

```
polymorphic_allocator& operator=(const polymorphic_allocator& rhs) = delete;
```

Обсуждение

- Аллокаторы снова становятся тем, чем они всегда были: тонким адаптером.
- Настоящий источник памяти теперь это `memory_resource`

Пространство имён pmr

- Теперь посмотрим как будет выглядеть вектор с таким аллокатором

```
namespace pmr {
  template <class T> using vector =
    ::std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

- Начиная с C++17, в пространство имён std::pmr включена вся стандартная библиотека с полиморфной аллокацией
- Использование мы уже видели ранее

```
pmr::monotonic_buffer_resource resrc(buffer, sz);

pmr::vector<double> v1(&resrc);
```

Case study: тестовый memory resource

- Тестовый ресурс памяти проверяет что аллокация соответствует деаллокации и проверяет утечки

```
struct test_resource : public pmr::memory_resource {
  // тут всякий интерфейс
private:
  struct allocation_rec {
    void *ptr_;
    size_t nbytes_, nalign_;
  };

  pmr::memory_resource *parent_;
  pmr::vector<allocation_rec> blocks_;
};
```

Аллокация

- Переопределение `do_allocate`

```
void *test_resource::do_allocate(size_t bytes, size_t align) {
  void *ret = parent_->allocate(bytes, align);
  blocks_.emplace_back(ret, bytes, align);
  return ret;
}
```

- Видно, что тестовый ресурс просто сцепляется с тем, над которым он живёт
- Это обычная идея: теперь мы можем складывать `memory_resources` в иерархические стопки

Обсуждение: цепочки ресурсов

- Сейчас в C++17 комбинирование осуществляется за счёт одного параметра parent
- Много ли можно выиграть, если придумать иные стратегии объединения распределителей?

Обсуждение: цепочки ресурсов

- Сейчас в C++17 комбинирование осуществляется за счёт одного параметра parent
- Много ли можно выиграть, если придумать иные стратегии объединения распределителей?
- Например выше аллокатор Хиннанта:
  - Брал память из буфера на стеке
  - Если буфер закончился, брал память из new/delete.
- Мы можем устроить `memory_resource` получающий на выход два: основной и дополнительный ресурс

Ресурс по умолчанию

- Для тестирования удобно установить логгирующий ресурс по умолчанию

```
int main () {
  static test_resource newdefault{pmr::new_delete_resource()};
  pmr::set_default_resource(&newdefault);

  /* .... */

}
```

- Обратите внимание: ресурс обязательно `static`, ему ещё освобождать всё в деструкторах

Case study: тонкости ресурсов памяти*

- Рассмотрим следующую интересную схему

```
class Bar { string data{"data"}; };

class Foo { unique_ptr<Bar> bar_{make_unique<Bar>()}; };
```

- Использование

```
pmr::vector<Foo> foos; // по умолчанию стоит test_resource
foos.emplace_back();
foos.emplace_back();
```

- Что на экране?

* via David Sankel talk on C++Now'18

Case study: тонкости ресурсов памяти*

- Рассмотрим следующую интересную схему

```
class Bar { string data{"data"}; };

class Foo { unique_ptr<Bar> bar_{make_unique<Bar>()}; };
```

- Использование

```
pmr::vector<Foo> foos; // по умолчанию стоит test_resource
foos.emplace_back();
foos.emplace_back();
```

- Что на экране?
- Очевидно на экране `a:8 a:16` так как содержимое `string` вне pmr

* via David Sankel talk on C++Now'18

Давайте занесём в pmr

- Рассмотрим следующую интересную схему

```
class Bar { std::pmr::string data{"data"}; };

class Foo { unique_ptr<Bar> bar_{make_unique<Bar>()}; };
```

- Использование

```
pmr::vector<Foo> foos; // по умолчанию стоит test_resource
foos.emplace_back();
foos.emplace_back();
```

- Что на экране?

Давайте занесём в pmr

- Рассмотрим следующую интересную схему

```
class Bar { std::pmr::string data{"data"}; };

class Foo { unique_ptr<Bar> bar_{make_unique<Bar>()}; };
```

- Использование

```
pmr::vector<Foo> foos; // по умолчанию стоит test_resource
foos.emplace_back();
foos.emplace_back();
```

- Что на экране?
- На экране снова `a:8` `a:16` так как SSO, а сам `unique_ptr` всё ещё вне pmr

Давайте немного усложним Foo

- Поскольку в стандарте нет `pmr::unique_ptr`, сделаем это руками

```
class Foo {
  unique_ptr<Bar, polymorphic_allocator_delete> d_bar;
public:
  Foo() : d_bar(nullptr, {{get_default_resource()}}) {
    polymorphic_allocator<Bar> alloc{get_default_resource()};
    Bar *const bar = alloc.allocate(1);
    alloc.construct(bar);
    d_bar.reset(bar);
  }
};
```

- Но что такое `polymorphic_allocator_delete` и зачем он нужен?

Пользовательский удалитель

- Разумеется это не может быть ничем кроме пользовательского удалителя

```
class polymorphic_allocator_delete {
  polymorphic_allocator<byte> alloc_;

public:
  polymorphic_allocator_delete(polymorphic_allocator<byte> alloc):
      alloc_(move(alloc)) {}

  template <typename T> void operator()(T *ptr) {
    polymorphic_allocator<T> talloc(alloc_);
    talloc.destroy(ptr);
    talloc.deallocate(ptr, 1);
  }
};
```

- Теперь всё срастается

Давайте немного усложним Foo

- Поскольку в стандарте нет `pmr::unique_ptr`, сделаем это руками

```
class Foo {
  unique_ptr<Bar, polymorphic_allocator_delete> bar_;

public:
  Foo() : bar_(nullptr, {{get_default_resource()}}) {
    polymorphic_allocator<Bar> alloc{get_default_resource()};
    Bar *const bar = alloc.allocate(1);
    alloc.construct(bar);
    bar_.reset(bar);
  }
};
```

- И что после этого на экране?

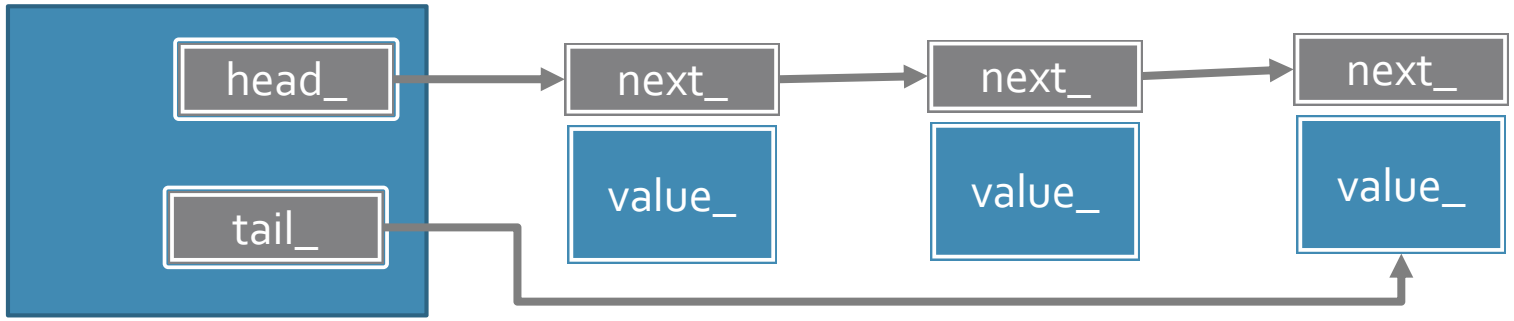
Обсуждение

- Разрабатывая собственные контейнеры и классы-обёртки, мы, конечно же, не хотим обрекать пользователя на **такое**?

- Плохая репутация аллокаторов
- Локальный (arena-based) подход
- Полиморфные аллокаторы
- Собственный pmr контейнер

Case study: slist*

- Основная идея: односвязный список с size за O(1) и splice за O(N) а также быстрой вставкой в начало и конец.



- Никакой rocket science, разумеется

* via Pablo Halpern talk on CppCon'17

Возможное устройство узла slist

- Устройство узла для такого контейнера

```
template <typename Tp> struct node;

template <typename Tp> struct node_base {
  node<Tp> *next_ = nullptr;
  // тут явно запрещены копирование и присваивание
};

template <typename Tp> struct node : node_base<Tp> {
  Tp value_;
};
```

- Увы довольно сложно угадать разумное значение по умолчанию для value_.

Трюк с union

- Устройство узла для такого контейнера

```
template <typename Tp> struct node;

template <typename Tp> struct node_base {
  node<Tp> *next_ = nullptr;
  // тут явно запрещены копирование и присваивание
};

template <typename Tp> struct node : node_base<Tp> {
  union { Tp value_; };
};
```

- Теперь это значение вообще не будет инициализировано при конструировании

Содержимое класса

- В общем это тоже тривиально

```
template <typename T> struct slist {
  using value_type = T;
  using iterator = <итератор>;
  // всё остальное
private:
  node_base head_; // сторожевой узел пустого списка
  node_base *ptail_;
  size_t size_;
  allocator_type alloc_; // храним аллокатор
};
```

- Но что такое allocator_type?

Инициализация аллокатора

- Тип `polymorphic_allocator<byte>` это как бы общий базовый класс для всех аллокаторов. Наличие в классе типа `allocator_type` и функции `get_allocator()`, сообщает всем, что вы *allocator-aware*

```
using allocator_type = pmr::polymorphic_allocator<byte>;
```

- Аргумент конструктора, а не шаблона

```
slist(allocator_type a = {}) :
  head_{}, ptail_{&head_}, size_{0}, alloc_{a} {};
```

- Обычные copy/move ctors

```
slist(const slist& rhs) : slist(rhs.get_allocator()) { .... }
slist(const slist&& rhs) : slist(rhs.get_allocator()) { .... }
```

Расширенное copy/move

- Мне не очень нравится идея, что

```
slist<int> s1(s2); // тут переходит тип аллокатора s2 на s1
```

- Хотелось бы иметь возможность задать аллокатор

```
slist(const slist& rhs, allocator_type a) : slist(a) { .... }
slist(const slist&& rhs, allocator_type a) : slist(a) { .... }
```

- Это называется расширенным синтаксисом copy/move

```
slist<int> s1 (s2, a3);
```

- Очень часто при этом обычное копирование оставляют вообще дефолтным

```
slist(const slist& rhs) : slist() { .... }
```

Использование аллокатора

- Ключевой метод: emplace, так как insertion будет реализован через него

```
template <typename T>
template <typename ... Args> iterator
slist<T>::emplace(iterator i, Args&& ... args) {
  void *mem = alloc_.resource()->allocate(sizeof(node), alignof(node));
  node *ret = static_cast<node*>(mem);
  ret->next_ = i.prev->next_;
  alloc_.construct(addressof(ret->value_), forward<Args>(args)...);
  i.prev->next_ = ret;
  // какая-то обработка крайних случаев, инкремент размера, etc
}
```

- Здесь, конечно, хотелось бы использовать traits

Использование аллокатора

- Кажется, что использование traits с передачей аллокатора это лучший вариант

```
template <typename T>
template <typename ... Args> iterator
slist<T>::emplace(iterator i, Args&& ... args) {
  void *mem = traits_t::allocate(alloc_, sizeof(node), alignof(node));
  node *ret = static_cast<node*>(mem);
  ret->next_ = i.prev->next_;
  traits_t::construct(alloc_, addressof(ret->value_),
                      forward<Args>(args)...);
  i.prev->next_ = ret;
  // какая-то обработка крайних случаев, инкремент размера, etc
}
```

- Увы, это не работает из-за alignof: у traits просто не предусмотрено такого аргумента

Обсуждение

- Загадочное использование `std::addressof`

```
alloc_.construct(std::addressof(ret->value_),
                 std::forward<Args>(args)...);
```

- Кто-нибудь понимает что это вообще и зачем оно здесь?

Обсуждение

- Загадочное использование `std::addressof`

```
alloc_.construct(std::addressof(ret->value_),
                 std::forward<Args>(args)...);
```

- Кто-нибудь понимает что это вообще и зачем оно здесь?
- Отгадка простая: амперсанд может быть перегружен.
- Внутри `addressof(x)` делает шулерскую цепочку преобразований чтобы установить настоящий адрес.

Входим в тонкости

- Всё-таки здесь есть очевидная проблема

```
template <typename ... Args> iterator
emplace (iterator i, Args&& ... args) {
  void *mem = alloc_.resource()->allocate(sizeof(node), alignof(node));
  node *ret = static_cast<node*>(mem);
  ret->next_ = i.prev->next_;
  alloc_.construct(addressof(ret->value_), forward<Args>(args)...);
  i.prev->next_ = ret;
  // какая-то обработка крайних случаев, инкремент размера, etc
}
```

- Кто её уже увидел?
- Подсказка: надо смотреть чего в этом коде нет

Безопасность исключений!

- Всё-таки здесь есть очевидная проблема

```
template <typename ... Args> iterator
emplace (iterator i, Args&& ... args) {
  void *mem = alloc_.resource()->allocate(sizeof(node), alignof(node));
  node *ret = static_cast<node*>(mem);
  ret->next_ = i.prev->next_;
  alloc_.construct(addressof(ret->value_), forward<Args>(args)...);
  i.prev->next_ = ret;
  // какая-то обработка крайних случаев, инкремент размера, etc
}
```

- Увы, вызов конструктора имеет право бросить любое исключение
- Домашняя наработка: хочется взять в catch-all, не так ли?

Присваивание

- Обычное копирующее присваивание теперь устроено очень просто

```
slist& operator=(const slist& other) {
  if (&other == this)
    return *this;

  erase(begin(), end());
  this->assign(other.begin(), other.end());
  return *this;
}
```

- Что при этом происходит с аллокатором? Ничего.
- Полиморфный аллокатор никогда не копируется и никогда не должен.

Тонкость в реализации перемещения

- Стандартная реализация перемещения в данном случае будет очевидно плоха:

```
slist& operator=(slist&& rhs) noexcept {
  swap(rhs.head_, head_);
  swap(rhs.tail_p_, tail_p_);
  swap(rhs.alloc_, alloc_);
  return *this;
}
```

- Проблема в выделенной строчке, она требует перемещения аллокатора
- Полиморфный аллокатор никогда не перемещается и никогда не должен (для него перемещение равно копированию).

Тонкость в реализации перемещения

- Поэтому необходимо предусмотреть копирование если пришёл объект того же класса с другим аллокатором

```
slist& operator=(slist&& rhs) { // увы, никакого noexcept
  if (alloc_ == rhs.alloc_) {
    swap(rhs.head_, head_);
    swap(rhs.tail_p_, tail_p_);
  }
  else
    operator=(rhs); // копирование
  return *this;
}
```

- Вспомним перемещающее присваивание в C++11, всё стало куда проще.

Обсуждение

- Мы, кажется, идём к тому, что сейчас у нас и move-ctor станет не noexcept. Но никакого noexcept это очень плохо
- Почему?

Обсуждение

- Мы, кажется, идём к тому, что сейчас у нас и move-ctor станет не noexcept. Но никакого noexcept это очень плохо
- Почему?
- Например рассмотрим реаллокацию в `vector<slist>`. Для строгой гарантии исключений если move-ctor не noexcept, он будет вынужден создавать временный буфер.
- Итак, очень хотелось бы даже пожертвовав move-assignment сохранить noexcept в move-ctor.

Noexcept в конструкторе

- Два move ctor делаются как раз для того, чтобы сохранить noexcept в одном:

```
slist(slist&& rhs) noexcept: slist(rhs.get_allocator()) {
  swap(rhs.head_, head_);
  swap(rhs.tail_p_, tail_p_);
}

slist(const slist&& rhs, allocator_type a) : slist(a) { .... }
```

- Во втором случае внутри фигурных скобок будет нечто очень похожее на копирование (может быть даже явный вызов operator=)
- Зато первый случай может использоваться контейнерами не внося дополнительную цену

Обсуждение

- У pmr контейнеров move-assignment не может быть реализовано через move-ctor (а только через extended move ctor).
- Вы видите здесь нечто странное?

Обсуждение

- У pmr контейнеров move-assignment не может быть реализовано через move-ctor (а только через extended move ctor).
- Лично я, да, вижу

```
pmr::vector<int> foo() { return pmr::vector<int>{some_alloc}; }

pmr::vector<int> v = foo(); // 1

pmr::vector<int> w;
w = foo();                  // 2
```

- Здесь v использует `some_alloc`, w использует `default_alloc`
- Но в общем это довольно просто осознать: аллокаторы привязаны к месту

Освобождение памяти

- Никаких сюрпризов

```
while (erase_next != erase_past) {
  node* old_node = erase_next;
  erase_next = erase_next->next_;
  --size_;
  alloc_.destroy(addressof(old_node->value_));
  alloc_.resource()->deallocate(old_node,
    sizeof(node), alignof(node));
}
```

- Вызов `deallocate` вполне симметричен
- Также сложно использовать `traits`, так как нужен `alignof`

Обсуждение

```
alloc_.resource()->deallocate(аргументы);
```

- Вообще-то тут происходит виртуальный вызов
- Страшный виртуальный вызов
- По косвенности
- Через таблицу виртуальных функций
- Мы напуганы?

Обсуждение

```
alloc_.resource()->deallocate(аргументы);
```

- Вообще-то тут происходит виртуальный вызов
- Страшный виртуальный вызов
- По косвенности
- Через таблицу виртуальных функций
- Мы напуганы?
- Мы не напуганы. По сравнению со всем остальным при тысячах разных независимых замеров (см. статью Лакоса и прочие), оверхед на виртуальную функцию здесь пренебрежим.

Development cost

- Впрочем, есть вещи, которыми мы напуганы
- Простой пример

```
struct S {
  int n;
  vector<int> v;
};
```

- Сейчас, так как она написана, эта структура имеет по умолчанию всё необходимое: копирование, присваивание, etc.
- Сколько кода надо написать, чтобы завести в ней allocator-awareness?

Development cost

- Впрочем, есть вещи, которыми мы напуганы
- Простой пример

```
struct S {
  int n;
  vector<int> v;
};
```

- Сейчас, так как она написана, эта структура имеет по умолчанию всё необходимое: копирование, присваивание, etc.
- Сколько кода надо написать, чтобы завести в ней allocator-awareness?
- Даже для C++17 правильный ответ: **тонны** кода.

Таблица Халперна

Task	C++98/03	C++11/14	C++17
Использование аллокатора	MEDIUM	MEDIUM	EASY
Создание аллокатора	MEDIUM	EASY	EASY
Создание обладающего состоянием аллокатора	IMPOSSIBLE	EASY	EASY
Создание scoped аллокатора	IMPOSSIBLE	MEDIUM	EASY
Новый контейнер использующий аллокаторы	MEDIUM	HARD	MEDIUM

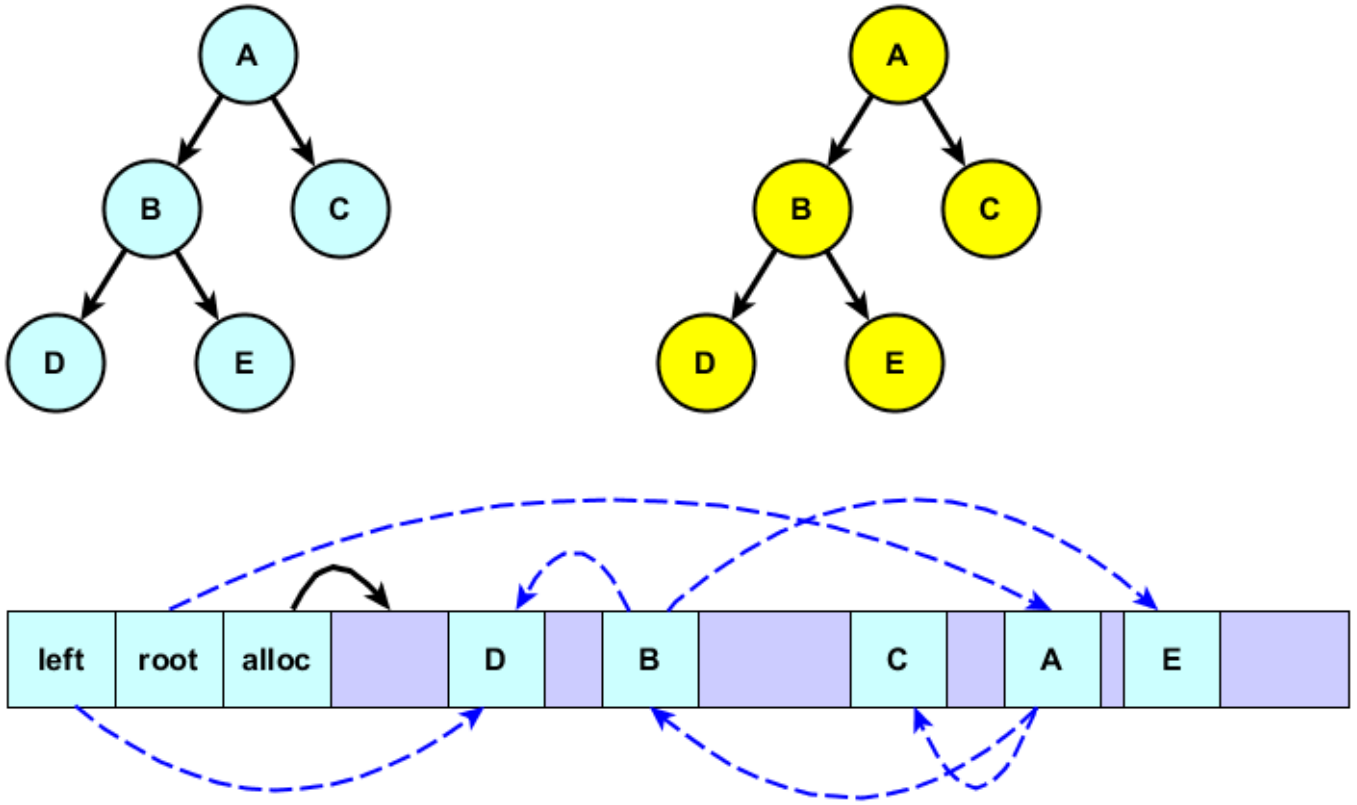
Литература

- ISO/IEC, "Information technology -- Programming languages – C++", ISO/IEC 14882: 2017
- Bjarne Stroustrup, The C++ Programming Language (4th Edition)
- Howard Hinnant, stack_alloc, howardhinnant.github.io/stack_alloc.html
- Alisdar Meredith, "Making allocators work", CppCon'14, parts 1 and 2
- Andrei Alexandrescu, "std::allocator Is to Allocation what std::vector Is to Vexation", CppCon'15
- Bob Steagall, "How to write a custom allocator", CppCon'17
- Pablo Halpern, N3916 proposal
- Pablo Halpern, "Allocators, the good parts", CppCon'17
- John Lakos, N4468, On Quantifying Memory-Allocation Strategies (paper with Meredith and others), 2015
- John Lakos, "Local (arena) memory allocators", CppCon'17 parts 1 and 2
- David Sankel, "C++17 std::pmr comes with a cost", C++Now'18

ДОМАШНЯЯ РАБОТА

Локально аллоцированные деревья

Проблема копирования для деревьев



Задание

- Разработать класс локально-аллоцированного дерева, поддерживающего простое копирование