5.8. Особенности перегрузки операторов new и delete. Разница между оператором new и функцией operator new. Разновидности оператора new, placement new, nothrow new, их правильная перегрузка. Особенности вызова нестандартного operator delete. В каком случае компилятор способен вызвать нестандартный operator delete самостоятельно?

Оператор **new** помимо выделения памяти под какой-то тип, еще и направляет конструкторы для него в каждую ячейку выделенной памяти. Этим оператор new отличается от сишного malloc (та только выделяет столько байт сколько попросили, конструкторы элементов приходилось вызывать вручную).

Действия оператора new можно перегрузить, но не целиком - только ту часть, которая отвечает выделению памяти. Конструкторы будут неизбежно вызваны после выделения памяти.

Отсюда следует **разница между оператор new и функцией оператор new**: при создании нового объекта память выделяется с помощью функции operator new, а затем вызывается конструктор для инициализации памяти. Здесь оператор new выполняет и выделение, и инициализацию, в то время как функция operator new выполняет только выделение.

Пример: у структуры S конструктор сделан приватным, тогда new S (или new S()) не скомпилируется, а operator new(sizeof(S)) (но нужно сделать reinterpret_cast от void* к S*, но вообще-то так делать не стоит) скомпилируется, и выделит память с помощью глобального new, без вызова конструктора

Заметка: для operator delete reinterpret_cast можно не делать

По станадрту оператор new принимает число - сколько нужно выделить байт

Оператор delete сначала вызовет деструкторы, потом очистит память, перегрузке аналогично подлежит только очищение памяти

Выделяем память с помощью сишной функции malloc(n) - запрашивает у ядра операционной системы память в байтах и возвращает указатель на выделенную память, если не получилось то nullptr.

Оператор **delete** принрмает в качестве аргумента указатель. Си-шная функция освобождает память - free.

Операторы new и delete для массивов отличаются от обычных

```
1   void* operator new(size_t n){
2       // return malloc(n);
3       void p* = malloc(n);
4       if (!p) throw std::bad_alloc();
5       return p;
6   }
7   void* operator new[](size_t n){
8       void p* = malloc(n);
9       if (!p) throw std::bad_alloc();
10      return p;
11  }
12  void operator delete(void* ptr){
13      free(ptr)
14  }
15  void operator delete[](void* ptr){
16      free(ptr)
```

```
    }
```

Если память не удалось выделить, то в стандартой реализации new вызывается функция **new_handler**, которая может другим способом попробовать выделить память (возможно с диска). Кроме того, можно самим переопределить функцию new_handler с помощью **set_new_handler**. После того как он ее вызывает, он снова пытается сделать malloc. Если попросим выделить 0 байт, оператор new выделит все равно 1 байт, так как в Си могут быть указатели на один и тот же адрес, а в C++ это запрещено.

Можно перегрузить оператор new для конркретного типа - для этого пишем оператор new в теле класса. Так как перегрузка оператора общая для всех объектов класса, то функция перегрузки оператора должна быть static. (Правило “частное лучше общего” при выборе new так же актуально); P.S. это можно не писать и оно и так будет считаться статик А еще можно делать операторы с кастомными параметрами - тогда при вызове new с такими параметрами, будет вызываться перегруженный оператор, но стандартный при этом так же будет работать.

Placement new

Напоминание: Если выделен какой-то кусок памяти под S, но конструктор на нем еще не было вызван, то есть синтаксис чтобы направить конструктор на уже выделенную память по указателю

```
    S* p = reinterpret_cast<S*>(operator new(sizeof(S)));
    S* p1 = reinterpret_cast<S*>(operator new(sizeof(S)));
    new(p) S(); //default construct will be called
    new (p1) S(value); //construct from value
```

Заметка: если в структуре переопределен оператор new, то placement new для нее работать не будет

Можно перегрузить оператор **new** именно для **placement new**. По умолчанию оператору new передается сколько памяти нужно выделить (компилятор подставлят это сам в обычном new), однако здесь память уже выделена, поэтому мы не пользуемся этим. Кроме того, как уже говорилось выше, в операторе new можно перегрузить только часть с выделением памяти, но в нашем случае память уже выделена, поэтому данный оператор ничего по сути не делает.

```
    void* operator new (size_t, S* p){
        return p;
    }
```

No-throw оператор new - не бросающий оператор new; он возвращает nullptr, если не удалось выделить память.

```
    // Перегрузка:
    void* operator new(size_t n, std::nothrow_t) {
        return malloc(n);
    }

    // Пример использования:
    int main() {
        int* ptr = new(std::nothrow) int(5);
    }
```

Placement delete не существует. Если хотим вызывать delete с кастомными параметрами, нужно будет вызывать функцию оператор delete явно: operator delete(ptr, mystruct).В таком случае нужно будет еще отдельно вызвать деструктор.

```
    S* ptr = new(mystruct) S();
    operator delete(ptr, mystruct);
    p->~S()
```

На самом деле компилятор иногда умеет вызывать кастомный оператор delete самостоятельно - только в случае если конструктор бросил исключение, так как компилятор должен подчистить выделенную память (сам вызывает кастомный delete) если он может это сделать (если кастомного delete нет - то ничего не происходит).

5.9. Реализуйте стековый аллокатор. Это такой аллокатор, который заводит большой массив в стековой памяти и всю память берет из него, ни разу не обращаясь к new (тем самым давая выигрыш во времени для контейнера, построенного на нем). Он делает это в предположении, что количество запрошенной памяти никогда не превзойдет размер этого массива.

PoolAllocator/StackAllocator - когда аллокатор конструируется в нем выделятся сразу большой пул огромного размера, а дальше при его allocate этот аллокатор хранит в себе одно число - указатель на первый незанятый байт (кратный 4 или 8), сдвигает этот указатель на соответсвующее число и вовзрвщает кусочек на котором он стоял до этого. При deallocate он не делает ничего. При вызове деструктора, удаляется весь пул

Когда это нужно: когда знаем что памяти много и ее заведомо хватит чтобы весь контейнер поместится. Такой аллокатор дает существенный выигрыш по времени в тех контейнерах, в которых каждое добавление - это вызов new(list, map, unordered_map)

Вообще, при таком аллокаторе даже не всегда приходится выделять динамическую память. Просто создает на стеке создает массив, а дальше ведет себя как этот аллокатор (если не больше 100 тысяч элементов, то такое сработает)

```
#pragma once
#include <vector>
#include <ctime>

#define tchS template<size_t chunkSize>
#define tT template<typename T>
#define tU template<typename U>
#define tTU template<typename T, typename U>
const int kAllocSz = 1000;

tchS
class FixedAllocator {
 private:
  std::vector<void*> pool_; // храним вектор свободных значений
 public:
  FixedAllocator() = default;
  ~FixedAllocator() = default;

  static FixedAllocator<chunkSize>& get_alloc() {
    static FixedAllocator<chunkSize> a;
    return a;
  }

  void* allocate();
  void deallocate(void* el, size_t t) { if (t == chunkSize) pool_.push_back(
   el); }
```

```
};


// ==================== Реализация FixedAllocator ==============================
tchS
void* FixedAllocator<chunkSize>::allocate() {
  if (pool_.size() == 0) {
    try {
      char* ptr = static_cast<char*>(::operator new(kAllocSz * chunkSize));
      for (int i = 0; i < kAllocSz; ++i)
        pool_.push_back(static_cast<void*>(ptr + i * chunkSize));
    } catch (...) {
      throw;
    }
  }
  void* ptr = pool_.back();
  pool_.pop_back();
  return ptr;
}
```

5.10. Понятие allocator-aware контейнеров. Параметры аллокаторов propagate_on_container_copy_assignment /move_assignment / swap, их предназначение и использование. Функция select_on_container_copy_construction, ее предназначение и использование. Реализация allocator-awareness на примере контейнера vector: правильная работа с аллокатором при копировании/перемещении/присваивании контейнера.

AllocatorAwareContainer - это контейнер, который содержит экземпляр аллокатора и использует этот экземпляр во всех своих функциях-членах для выделения и деаллокации памяти, а также для создания и уничтожения объектов в этой памяти (такими объектами могут быть элементы контейнера, узлы или, для неупорядоченных контейнеров, массивы ведер).

The following rules apply to container construction

- Copy constructors of *AllocatorAwareContainer*s obtain their instances of the allocator by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction` on the allocator of the container being copied.
- Move constructors obtain their instances of allocators by move-constructing from the allocator belonging to the old container.
- All other constructors take a `const allocator_type&` parameter.

The only way to replace an allocator is copy-assignment, move-assignment, and swap:

- Copy-assignment will replace the allocator only if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value` is `true`
- Move-assignment will replace the allocator only if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment::value` is `true`
- Swap will replace the allocator only if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`. Specifically, it will exchange the allocator instances through an unqualified call to the non-member function swap, see *Swappable*.

Что вообще такое **копирование аллокатора** (что значит инициализировать один аллокатор другим аллокатором)?

Допустим у нас PoolAllocator. Если мы копируем аллокатор, то пул копировать не надо,

так как новый аллокатор должен уметь освобождать то, что выделил старый аллокатор.

```
alloc1 == alloc2
// this means
T* ptr = alloc1.allocate(1);
alloc2.deallocate(ptr, 1);
```

Чтобы несколько аллокаторов могли указывать на один и тот же пуул нужно использовать **shared_ptr<Pool>**, который принимает обычный C-style поинтер. И в деструкторе мы удаляем пуул, только тогда, когда наш указатель на пуул последний.

Поведение аллокатора при копировании и присваивании контейнера

Нужно ли нам в таком случае делать копию аллокатора или необходимо создать новый.

```
vector<int, PoolAlloc> v1;
// .... //
vector<int, PoolAlloc> v2 = v1;
```

1 вариант: мы хотим чтобы копия контейнера указывала на старый пулл

2 вариант: мы хотим чтобы у каждого контейнера был свой пуул.

В какой момент принимается решение копировать/не копировать аллокатор? Для этого в allocator_traits есть специальный метод **select_on_container_copy_construction**. Этот метод возвращает объект аллокатора, который будет использоваться в контейнере. По умолчанию вернётся копия аллокатора (два контейнера будут указвать на один и тот же пуул).

Что должен делать контейнер при копировании. Мы должны инициализировать аллокатор результатом выше упомянутого метода.

Также в allocator_traits есть using **propagate_on_container_copy_assignment**, который определяет нужно ли заниматься присваиванием аллокатора при присваивании контейнера. По умолчанию он равен std::false_type (в нем static bool_value = false). Но можно сделать его true. Аналогичная история со swap - **propagate_on_container_swap**

Оператор присваивания в этой реализации не безопасен относительно исключений!

```
template <typename T, typename Alloc = std::allocator<T>>
class Vector {
    T* arr;
    size_t sz, cap;
    Alloc alloc;

    using AllocTraits = std::allocator_traits<Alloc>;
public:
    Vector(size_t n, const T& val = T(), const Alloc& alloc = Alloc());

    Vector<T, Alloc>& operator=(const Vector<T, Alloc>& other) {
        if (this == &other) return *this;

        for (size_t i = 0; i < sz; ++i) {
            pop_back();
            //AllocTraits::destroy(alloc, arr + i);
        }
        AllocTraits::deallocate(arr, cap);

        //main decision
        if (AllocTraits::propagate_on_container_copy_assignment::value
&& alloc != other.alloc) {
            alloc = other.alloc;
            // what if the exception appear here??
        }

        sz = other.cap;
```

```
            cap = other.cap;

            AllocTraits::allocare(alloc, other.cap);
            for (size_t i = 0; i < sz; ++i) {
                push_back(other[i]);
            }
             return *this;
        }
    }
```

Отныне мы все чаще и чаще будем обращаться к разделу **named requirements** на cpp.reference

Например, наша реализация list должна быть написана согласно *AllocatorAwareContainer*.

```
\textbfПример{ select_on_container_copy_construction:}

template <typename T, typename Allocator>
List<T, Allocator>::List(const List<T, Allocator>& another):    allocator_(
    AllocatorTraits::select_on_container_copy_construction(another.allocator_
    )),
                                                                    head(
    AllocatorTraits::allocate(node_allocator_, 1)) {
    head->previous = head;
    head->next = head;
    for (auto it = another.begin(); it != another.end(); ++it)
        push_back(*it);
}
```