Язык C++

Мещерин Илья

Лекция 12

Аллокаторы

8.1) Placement new
Явный вызов конструктора и деструктора на месте

```
1  T* p = new T(...);
2  new (p) T(...);
3  p->~T();
```

8.2) Перегрузка операторов new, delete
n - кол-во байт

```
void* operator new(size_t n){}
```

При вызове оператора *new*

а) Выделяется память

б) Вызывается конструктор

Когда перегружаем *new* думаем только о первой части, вторую часть перегрузить нельзя
Оператор *new* и функция *new()* - разные вещи

Реализация перегрузки функции *new* (т.к. пункт б делается сам)

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

В оператор *new* можно передать дополнительные аргументы

```
new (p, ...) T (...);
void* operator new(size_t n, ...){}
```

При вызове оператора *delete*

а) Вызывается деструктор

б) Освобождается память

Только вторую часть можно перегрузить
Можно написать *delete(...)* но вызывать эту функцию нужно руками, то есть сначала вызвать деструктор *p->~T()* потом наш *delete(...)* (стандартный *delete p* переопределить с параметрами нельзя)

Пример

```
void* operator new(size_t n, int k){
    //code here...
}

int main(){
    int *p = new(42) int;
}
```

Все то же самое отдельно для операторов *new[]* и *delete[]*

Совет: если реализовали кастомный *new*, то стоит реализовать *delete* с такими же параметрами такой *delete* вызовется, если конструктор кинет исключение

Пример

```
1  class Animal {
2      public:
3      virtual void say() = 0;
4      virtual ~Animal() {}
5  };
6
7  class Sheep : public Animal {
8      public:
9      virtual void say() {
10         printf("Sheep says baaaaa\n");
11     }
12
13     virtual ~Sheep() {
14         printf("Sheep is dead\n");
15     }
16
17     void operator delete(void* p) {
18         printf("Reclaiming Sheep storage from %p\n", p);
19         ::operator delete(p);
20     }
21 };
22
23 int main(int argc, char** argv) {
24     Animal* ap = new Sheep;
25     ap->say();
26     delete ap;
27 }
```

Вызовется правильный *delete* не смотря на то, что *static operator delete*

8.3) Nothrow new

```
#include <new>
```

```
#include <memory>
struct nothrow_t {};
new(std::nothrow) T();
void* operator new (std::size_t size, const std::nothrow_t &nothrow_value) noexcept;
```

Такой вызов оператора *new* не бросает исключений

8.4) New_handler

Правильная реализация функции *new*: в бесконечном цикле пытаемся выделить память и если не смогли, то вызываем *new_handler()*
То есть эта функции по логике должна каким-то образом помогать функции *new* выделить память

```
std::new_handler std::set_new_handler(std::new_handler new_p)
std::new_handler std::get_new_handler()
```

В *std::set_new_handler()* можно передать свою функцию *new_handler()*, по умолчанию *nullptr*

8.5) Реализация аллокатора

```
1  template<class T>
2  struct std::allocator<T> {
3      typedef T value_type;
4      T* allocate(size_t n) const {
5          return (T*)(new char[n * sizeof(T)]);
6      }
7      void deallocate(T* p, size_t n) const {
8          delete[] (char*)p;
9      }
10     template<typename ... Args>
11     void* construct(T* p, const Args &... args) const {
12         return new(p) T(args...);
13     }
14     void destroy(T* p) const {
15         p->~T();
16     }
```

Аллокатор - обертка над *new* и *delete*

8.6) std::allocator_traits

```
template<class Alloc>
std::allocator_traits<Alloc>
```

Реализует за аллокатор те методы, которые аллокатор не реализовал
construct() и *destroy()* у всех аллокаторов реализуется одинаково, поэтому эту реализацию можно вынести в отдельный класс - *allocator_traits*
Вектор реализуется через обращения к *allocator_traits*

```
template<class Alloc>
std::allocator_traits<Alloc>{
    static Alloc::value_type* allocate(Alloc &a, size_t n){
```

```
            return a.allocate();
        }
    }
```

Аналогично реализуется *deallocate()*
Для *construct(), destroy()* класс умеет проверять реализованы ли уже эти функции в аллокаторе с нужной сигнатурой и иначе реализует сам