

# Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 6  
13.10.2017

# Операторы new и delete

Зачем они нужны?

```
#include <iostream>
class C {
public:
    int arr[100];
    C(int a) { /*...*/ }
};

int main() {
    C * c = new C(123);
    // ...
    delete c;
};
```

- Создание и удаление динамических объектов

**Проблема:** Временем жизни таких объектов приходится управлять вручную.

# Проблемы new и delete

1. Можно забыть написать delete.
2. Можно написать лишний delete.
3. Утечки памяти при исключениях и т.п.
4. delete / delete[].

Как решать?

- ▶ Оставить delete умным указателям.
- ▶ Оставить new make-функциям.

# Но всё-таки про new/delete

Оператор **new** состоит из двух частей:

1. Выделение сырой (свободный кусок динамической) памяти. Может возникнуть исключение.
2. Конструирование объекта в сырой памяти.

Оператор **new** гарантирует, что если в конструкторе произошло исключение, то выделенная динамическая память автоматически очистится.

Оператор **delete** делает все наоборот:

1. Вызывается деструктор.
2. Освобождается память.

# Размещающий оператор new

Как было раньше:

```
int * p = (int*)(malloc(sizeof(int)));  
// ...  
free(p);
```

Два способа нельзя смешивать (malloc + delete, new + free)

Способ с new предпочтительнее, и его реализацию можно перегружать:

```
void *p = malloc(sizeof(C));  
C * c = new (p) C(123); // placement new;  
// ...  
c -> ~C();  
free(p);
```

# Размещающий оператор new

Как-то надо бороться с исключениями:

```
void * p = malloc(sizeof(C));  
if (!p) return 1;  
C * c;  
try {  
    c = new (p) C(123);  
} catch (...) {  
    free(p);  
    throw;  
}  
try {  
    // ...  
} catch (...) {  
    c -> ~C();  
    free(p);  
    throw;  
}  
c -> ~C();  
free(p);
```

# operator new, operator delete

```
//new C(x)  
void *p = operator new(sizeof(C));  
C * c;  
try {  
    c = new(p) C(x);  
} catch (...) {  
    operator delete(p);  
    throw;  
}  
//delete p  
if (p!=NULL) {  
    p->~C();  
    operator delete(p);  
}
```

# Перегрузка

```
void * operator new (size_t sz) {  
    std::cout << "operator new with " << sz << std::endl;  
    void * p = malloc(sz);  
    if (!p) throw std::bad_alloc();  
    return p;  
}
```

```
void operator delete(void * p) {  
    std::cout << "operator delete" << std::endl;  
    free(p);  
}
```



# Перегрузка

Можно перегрузить так:

```
void*operator new (size_t sz, double a, int x) {  
    // ...  
    return ::operator new(sz);  
}
```

Но тогда нужно написать парный ему

```
void operator delete(void * p, double a, int x) {  
    // ...  
    ::operator delete(p);  
}
```

Как тогда их вызвать?

```
C* p = new(1.23, 123) C(111);  
delete p;
```

# Перегрузка

Оператор new/delete внутри класса обязан быть статическим.  
static можно не писать.

```
class A {  
    int param;  
public:  
    A(int a): param(a) {  
        cout << "A::A(" << a << ")" << std::endl;  
    }  
    virtual ~A() { cout << "A::~A()" << std::endl; }  
    static void* operator new(size_t sz) {  
        cout << "A::operator new" << std::endl;  
        return ::operator new(sz);  
    }  
    static void operator delete(void* ptr) {  
        cout << "A::operator delete" << std::endl;  
        ::operator delete(ptr);  
    }  
};
```

# Аллокаторы

Класс, который выделяет память каким-то специальным образом.

```
map<int, string> m;  
// map<int, string, less<int>, allocator<pair<int, string>> > m;
```

Обязательно определяются:

- ▶ `value_type` — тип, для которого работает аллокатор
- ▶ `allocate` — выделение памяти под  $n$  объектов, но не конструирование
- ▶ `deallocate` — освобождение памяти

Необязательно:

- ▶ `construct` — инициализация заданной памяти заданным значением
- ▶ `destroy` — уничтожение памяти без освобождения

# Аллокаторы

```
template <class T>
struct TMyAllocator {
    using value_type = T;
    TMyAllocator() = default;
    template <class U>
    constexpr TMyAllocator(const TMyAllocator<U>&) noexcept {}

    T* allocate(size_t n) {
        std::cout << "allocate n = " << n << std::endl;
        if (n > size_t(-1) / sizeof(T))
            throw std::bad_alloc();
        auto p = static_cast<T*>(malloc(n * sizeof(T)));
        if (!p)
            throw std::bad_alloc();
        return p;
    }
    void deallocate(T* p, std::size_t) noexcept {
        free(p);
    }
};
```

# Аллокаторы

```
int main() {  
    std::vector<int, TMyAllocator<int>> vec;  
    for (int i = 0; i < 6; ++i)  
        vec.push_back(0);  
}
```

allocate n = 1

allocate n = 2

allocate n = 4

allocate n = 8

Решение о том, сколько памяти запросить и в какой момент, целиком принимается контейнером, вне зависимости от аллокатора, который используется для поставки этой памяти. Стандартный аллокатор хорошо подходит для вектора, но часто не самое лучшее решение для списка.

# Smart pointers

Чем плохи обычные встроенные указатели?

- ▶ Указывают на массив или на объект?
- ▶ Владеет ли указатель тем, на что указывает?
- ▶ Трудно обеспечить уничтожение ровно один раз.
- ▶ Обычно сложно определить, является ли указатель висячим.
- ▶ Нельзя предоставить информацию компилятору о том, могут ли два указателя указывать на одну область памяти.

«Умный» («интеллектуальный») указатель притворяется обычным указателем с дополнительными функциями.

Обертка над обычными указателями.

# Smart pointers

Хочется что-то вроде такого

```
SmartPointer sp(new C);
```

и дальше пользоваться как обычным указателем, не задумываясь об удалении.

А что делать тут?

```
SmartPointer sp2 = sp;
```

Всегда можно обмануть умный указатель:

```
C * ptr = new C;
```

```
SmartPointer sp(ptr);
```

```
SmartPointer sp2(ptr);
```

# Smart pointers: стратегии

- ▶ Запрет копирования и присваивания.
- ▶ Глубокое копирование.
- ▶ Подсчет ссылок в специальных счетчиках.
- ▶ Список ссылок.
- ▶ Передача владения.



# Стратегия передачи владения

Если кто-то пытается скопировать указатель, то ему и передается владение, и исходный умный указатель не указывает больше на объект. Такой указатель не нужно класть в контейнер .

```
std::auto_ptr<int> p (new int(123));  
std::vector<std::auto_ptr<int> > v;  
v.push_back(p);
```

# Умные указатели в C++ 11 / 14

```
std::auto_ptr<>    // deprecated  
std::unique_ptr<>  
std::shared_ptr<>  
std::weak_ptr<>
```

## std::unique\_ptr

- ▶ Реализует семантику исключительного владения
- ▶ Перемещение передает владение от исходного указателя целевому, целевой при этом обнуляется.
- ▶ Копирование не разрешается.
- ▶ При деструкции освобождает ресурс, которым владеет.

Обычное использование – возвращаемый тип фабричных функций для объектов иерархии:

```
template <typename T>  
std::unique_ptr<Base> makeObject(T&& params);
```

# Некоторые методы `std::unique_ptr`

- ▶ `reset` — заменяет объект;
- ▶ `release` — освобождает владение;
- ▶ `get` — возвращает указатель на объект, которым владеет;

Запрещены неявные преобразования обычного указателя в умный:

```
std::unique_ptr<Base> p;  
p = new A();
```

# std::unique\_ptr

Две разновидности для индивидуальных объектов и для массивов:

```
std::unique_ptr<T> // *, ->  
std::unique_ptr<T[]> // []
```

std::unique\_ptr можно присваивать в std::shared\_ptr (можно не задумываться над тем, как будет использован возвращаемый указатель).

# Как избавиться от new?

Написать обертку!

```
template <typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params) {
    return std::unique_ptr<T>(
        new T(std::forward<Ts>(params)...)
    )
}
```

Чего не хватает? Массивов, пользовательских удалителей.  
Функция уже есть — `std::make_unique` в C++14.

```
std::unique_ptr<Base> p(new Base); // дважды пишем Base
auto p1(std::make_unique<Base>()); // make
```