

Лекция 10. Объекты и new/delete. Умные указатели. Наследование.

Евгений Линский

```
class Matrix {
private:
    int** mtx;
    ...
public:
    Matrix(size_t n) {
        mtx = new int* [n];
        int * tmp = new int [n * n];
        ...
    }
    ~Matrix() {
        delete [] mtx[0];
        delete [] mtx;
    }
};
```

- ▶ функции malloc/free не “не знают” про конструктор и деструктор
- ▶ операторы new/delete “знают”

Один объект:

```
Matrix* m = new Matrix(4);  
...  
delete m;
```

Много объектов:

```
Matrix* m = new Matrix [100];  
...  
delete [] m;
```

`delete []` — надо пройти по памяти и у каждого объекта вызвать деструктор (просто `delete` этого не сделает).

```
Matrix* m = new Matrix [100];
```

- ▶ Почему это, на самом деле, не скомпилируется?

```
Matrix* m = new Matrix [100];
```

- ▶ Почему это, на самом деле, не скомпилируется?
- ▶ Компилятор не знает, как создать объект Matrix, поскольку у него конструктор с параметром. Нужен default конструктор.

```
class Matrix {  
    ...  
public:  
    Matrix() { mtx = new int* [100]; ...}  
};
```

Можно и так:

```
Matrix ** mtxs = new Matrix* [2];  
mtxs [0] = new Matrix(3);  
mtxs [1] = new Matrix();  
  
delete mtxs [0];  
delete mtxs [1];  
delete [] mtxs;
```

Object — некоторый класс.

```
void f() {  
    Object *p = new Object(...);  
    p->save_to_file("out.txt");  
    delete p;  
}
```

Хотим, сделать “обертку” (MyArray — “обертка” над динамическим массивом) над указателем.

```
void f() {  
    scoped_ptr p(new Object(...));  
    p->save_to_file("out.txt");  
}
```

Не нужно думать про освобождение памяти! Об этом позаботится деструктор `scoped_ptr`.

```
class scoped_ptr {
private:
    Object* myPointer;
public:
    scoped_ptr(Object* ptr);
    ~scoped_ptr();
public:
    Object* ptr();
    Object& operator *() const;
    Object* operator ->() const;
    bool isNull() const;

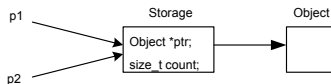
private:
    scoped_ptr(const scoped_ptr& p);
    const scoped_ptr& operator=(const scoped_ptr& p);
}
```



```
scoped_ptr::scoped_ptr(Object* ptr) {
    myPointer = ptr;
}
scoped_ptr::~~scoped_ptr() {
    if (myPointer != 0) {
        delete myPointer;
    }
}
Object& scoped_ptr::operator*() const {
    return *myPointer;
}
Object* scoped_ptr::operator->() const {
    return myPointer;
}
bool scoped_ptr::isNull() const {
    return myPointer == 0;
}
```

- ▶ `scoped_ptr` — нельзя копировать (`copy`) и передавать в функции
- ▶ `unique_ptr` — можно перемещать (`move`) и передать в функцию один раз
- ▶ `shared_ptr` — можно все! (почти)

```
unique_ptr::unique_ptr(unique_ptr& p) {
    myPointer = p.myPointer;
    p.myPointer = 0;
}
unique_ptr& unique_ptr::operator=(unique_ptr& p) {
    if (this != &p) {
        if (myPointer != 0) {
            delete myPointer;
        }
        myPointer = p.myPointer;
        p.myPointer = 0;
    }
    return *this;
}
```



- 1 Создаём вспомогательный объект Storage, в котором есть счётчик ссылок (и кроме того указатель на Object), устанавливаем его на 1.
`shared_ptr p1 = new Object;`
- 2 Пусть теперь создали другой указатель и присвоили ему значение 1ого, тогда увеличиваем счётчик на 1.
`shared_ptr p2 = p1;`
- 3 Теперь область видимости заканчивается и вызывается деструктор p2. Уменьшаем счётчик на 1, т.к. значение счётчика не 0, то больше ничего не делаем.
- 4 Вызывается деструктор p1, уменьшаем счётчик на 1, т.к. он стал равен нулю, вызываем деструктор для Object, а затем для Storage.

Когда такой подход не сработает? Подсказка: циклические ссылки.

Идея: не писать новый класс с нуля, а реализовать на основе существующего. Постановка задачи:

- ▶ 2010 год, решили создать класс “странный связный список”
 - Состоит из одного класса (обычно два: List и Node)
 - Не может быть пустым
 - Сложный деструктор
- ▶ 2017 год, решили создать класс “странный двусвязный список”

```
class List {  
private:  
    int val;  
    List *next;  
public:  
    List(int val);  
    ~List();  
    void push_back(int val);  
    size_t length() const;  
};
```

```
List l(5);  
l.push_back(7);  
l.push_back(8);
```

```
#include "List.h"
List::List(int val) {
    this->val = val; this->next = NULL;
}
~List::List() { /*do it yourself */ }

void List::push_back(int val) {
    List* cur = this;
    while (cur->next != NULL) cur = cur->next;
    cur->next = new List(val);
}

size_t List::length() const {
    // 'this' has type 'const List* const' in const method
    size_t count = 0; const List* cur = this;
    while (cur->next != NULL) {
        cur = cur->next;
        count++;
    }
    return count;
}
```

```
#include "List.h"
class DoubleList: public List {
private:
    DoubleList* prev;
public:
    DoubleList(int val);
    ~DoubleList();
    void push_back(int val);
    void pop_back(); //new method
};
```

- ▶ Можно считать, что поля и методы из класса List скопируются компилятором в новый класс DoubleList.
- ▶ Хотим добавить поле prev и изменить (перекрыть) код метода push_back.
- ▶ Перекрыть (override) — имя метода тоже, **параметры те же**, код другой.
- ▶ Перегрузить (overload) — имя метода тоже, **другие параметры**, код другой.
int max(int, int) и int max(int, int, int)

```
#include "DoubleList.h"
DoubleList::DoubleList(int val): List(val) {
    this->prev = NULL;
}

// Base class destructor will be called automatically
~DoubleList::DoubleList() { /* do it yourself */}

void DoubleList::push_back(int val) {
    DoubleList* cur = this;
    while (cur->next != NULL) cur = cur->next;
    cur->next = new DoubleList(val);
    cur->next->prev = cur;
}

void DoubleList::pop_back() { /* do it yourself */}
```


Термины:

- ▶ List — базовый класс (base class), предок, супер класс
- ▶ DoubleList — производный класс (derived class), потомок

```
DoubleList::DoubleList(int val): List(val) {...}
```

Если в базовом классе, есть конструктор по умолчанию, то компилятор подставит его вызов сам.

В примере есть три ошибки:

- 1 Модификатор доступа в List должен быть `protected`, а не `private`.
- 2 У метода `push_back` надо дописать `virtual`.

```
class List {  
protected:  
    int val;  
    List *next;  
public:  
    ...  
    virtual void push_back(int val);  
};
```

В примере есть три ошибки:

- 1 Модификатор доступа в List должен быть protected, а не private.
- 2 У метода push_back надо дописать virtual.
- 3 Приведения типов между указателями разных типов (DoubleList* и List*) в функции DoubleList::push_back.

```
class List {  
protected:  
    int val;  
    List *next;  
public:  
    ...  
    virtual void push_back(int val);  
};
```

```
cur->next = new DoubleList(val); // List* = DoubleList*  
cur = cur->next; // DoubleList* = List*  
(cur->next)->prev = cur // (List*)->DoubleList* =  
DoubleList*
```

DoubleList “умеет” все, что умеет List с точки зрения интерфейса, т.е. поддерживает те же публичные функции. Обратное неверно (у List нет pop_back).

Явное приведение типа = “компилятор, я не описался, так и задумано”

Безопасно (не нужно явное приведение типа):

```
DoubleList dl(5);  
List *l = &dl; // explicit cast is not required  
l->push_back(3);
```

Небезопасно (нужно явное приведение типа):

```
List l(5);  
DoubleList *dl = (DoubleList*)&l; // explicit cast is  
required  
dl->push_back(3); // all ok  
dl->pop_back(); // compilation ok, but run-time error
```

```
void DoubleList::push_back(int val) {  
    DoubleList* cur = this;  
    while (cur->next != NULL) cur = (DoubleList*)cur->next;  
    cur->next = new DoubleList(val);  
    ((DoubleList*)cur->next)->prev = cur;  
}
```

Функция создана в 2010 году (DoubleList еще не было даже в планах).

```
void fill(List *l, int val, size_t num) {  
    for(size_t i = 0; i < num; i++)  
        l->push_back(val);  
}
```

2017 год:

```
List l(5);  
DoubleList dl(6);  
  
fill(&l, 10, 42);  
fill(&dl, 100, 24);
```

Полиморфизм — можем написать код, который будет работать с разными типами данных. Полиморфизм в ООП подробно обсудим в следующий раз.