



## **Лекция 3. Создание и разрушение объектов**



## Конструкторы

Специальные функции, объявляемые в классе.

Имя функции совпадает с именем класса.

Не имеют возвращаемого значения.

Предназначены для инициализации создаваемых объектов класса.

```
class Date {  
    int year;  
    int month;  
    int day;  
public:  
    void init(int day, int month, int year);  
  
    void add_year(int year);  
    void add_month(int month);  
    void add_day(int day);  
}
```



## Перегрузка конструкторов

В классе может присутствовать несколько конструкторов с разным количеством или типом параметров.

```
class Date {  
    int year;  
    int month;  
    int day;  
public:  
    Date(int day, int month, int year);  
    Date(int day, int month);  
    Date(int day);  
    Date();  
    Date(const char *date) const; <- не может быть константой.  
}
```



## Списки инициализации

Позволяют проинициализировать поля до входа в конструктор.

```
class Date {  
    int year;  
    int month;  
    int day;  
  
public:  
    Date(int day, int month, int year) : year(year), month(month), day(day)  
    {}  
}
```

Инициализация полей в списке происходит в *порядке объявления полей в классе*, а не в порядке их следования в списке инициализации.



## Значения по умолчанию

Конструкторы (как и функции) могут иметь значения по умолчанию.

Значения параметров по умолчанию нужно указывать в *объявлении функции*.

```
class Date {  
    int year;  
    int month;  
    int day;  
  
public:  
    Date(int day = 0, int month = 0, int year = 0)  
        : year(year), month(month), day(day) {}  
}
```

```
Date zero;  
Date days (10);  
Date daysAndMonths (10, 2);
```



## Ключевое слово `explicit`

Позволяет запретить *неявное* пользовательское преобразование.

```
class Segment {  
    Point first;  
    Point second;  
  
public:  
    Segment() {}  
    explicit Segment(double length) : second(length, 0) {}  
}
```

```
Segment first;  
Segment second(10);  
Segment third = 20; <- ошибка компиляции при наличии explicit.
```



## Конструктор по умолчанию

Если не объявлено ни одного конструктора – компилятором будет создан конструктор по умолчанию.

```
class Segment {  
    Point first;  
    Point second;  
  
public:  
    Segment(Point first, Point second)  
        : first(first), second(second) {}  
}
```

```
Segment first; <- ошибка компиляции – отсутствие конструктора без параметров.  
Segment second(Point(), Point(1, 2));
```



## Особенности синтаксиса C++

«Если что-то похоже на объявление функции, то это и есть объявление функции.»

```
class Point {  
    int x;  
    int y;  
  
public:  
    explicit Point (int x = 0, int y = 0) : x(x), y(y) {}  
};
```

```
Point first;  
Point second();  
  
double value = 5.1;  
Point third(int(value));  
Point fourth((int)value);
```





## Деструкторы

Специальные функции, объявляемые в классе.

Имя функции совпадает с именем класса, плюс знак ~ в начале.

Не имеют возвращаемого значения и параметров.

Вызывается автоматически при удалении экземпляра структуры.

Предназначены для освобождения используемых ресурсов.

```
class IntArray {
    size_t size;
    int *data;

public:
    explicit IntArray(size_t size) : size(size), data(new int[size]) {}
    ~IntArray() {
        delete []data;
    }
}
```



## Время жизни объекта

*Время жизни* – временной интервал между вызовами конструктора и деструктора.

```
void foo() {
    Point first();           // Вызов конструктора first.
    Point second(20, 20);   // Вызов конструктора second.
    for (int i = 0; i < 10; ++i) {
        Point third(30, 30); // Вызов конструктора third.
    } // Вызов деструктора third.
} // Вызов деструкторов second => first.
```

Деструкторы переменных на стеке вызываются в обратном порядке (по отношению к порядку вызова конструкторов).

В программе не должно быть обращения к переменным *ДО* начала их времени жизни или *ПОСЛЕ*. В противном случае это ведет к неопределенному поведению.



## Выделение динамической памяти (Си)

`void*` `malloc(size_t sizemem)` – выделяет блок памяти, размером `sizemem` байт и возвращает указатель на начало блока. Содержание выделенного блока памяти *не инициализируется* (остается с неопределенными значениями).

`void*` `calloc(size_t nmemb, size_t size)` – выделяет память для массива размером `nmemb`, каждый элемент которого равен `size` байт и возвращает указатель на выделенную память. Память при этом *очищается* (зануляется).

`void*` `realloc(void *ptr, size_t size)` – меняет размер блока памяти, на который указывает `ptr`, на размер, равный `size` байт. Содержание будет неизменным в пределах наименьшего из старых и новых размеров, а новая распределенная память будет *не инициализирована*.



## Освобождение динамической памяти (Си)

`void free(void *ptr)` – освобождает место в памяти, на который указывает `ptr`, полученный динамическим выделением памяти. Иначе (если функция `free` уже вызывалась для этого участка памяти, дальнейший ход событий непредсказуем – `undefined behavior`). Если `ptr == NULL`, то не выполняется никаких действий.

Вызов функции `realloc` с параметром `size` равным нулю эквивалентен `free(ptr)`.



## Выделение динамической памяти (C++)

Для создания объекта в динамической памяти используется оператор `new`, он отвечает за вызов конструктора.

```
class IntArray {
    size_t size;
    int *data;

public:
    explicit IntArray(size_t size) : size(size), data(new int[size]) {}
    ~IntArray() { delete []data; }
}
```

```
// Только выделение памяти.
IntArray *oldStyle = (IntArray*) malloc(sizeof(IntArray));

// Выделение памяти и создание объекта.
IntArray *newStyle = new IntArray(5);
```



## Освобождение динамической памяти (C++)

При вызове оператора `delete` вызывается деструктор объекта.

```
// Выделение памяти и создание объекта.  
IntArray *newStyle = new IntArray(5);  
  
// Вызов деструктора и освобождение памяти.  
delete newStyle;
```

Операторы `new[]` и `delete[]` работают аналогично:

```
// Выделение памяти и создание 5 объектов.  
// При создании объектов вызывается конструктор по умолчанию.  
IntArray *array = new IntArray[5];  
  
// Вызов деструкторов и освобождение памяти.  
delete[] newStyle;
```



## Оператор new с размещением

```
// Выделение памяти.  
void *pointer = malloc(sizeof(IntArray));  
  
// Создание объекта по адресу pointer.  
IntArray *array = new (pointer) IntArray(10);  
  
// Явный вызов деструктора.  
array->~IntArray();  
  
// Освобождение памяти.  
myfree(pointer);
```

Проблемы с выравниванием:

```
char buffer[sizeof(IntArray)];  
new (buffer) IntArray(20); // Потенциальная проблема.
```



## Идиома программирования

Устойчивый способ выражения некоторой составной конструкции в одном или нескольких языках программирования.

Является шаблоном решения задачи, записи алгоритма или структуры данных путем комбинирования встроенных элементов языка.

Можно считать самым низкоуровневым шаблоном проектирования, применяемым на стыке проектирования и кодирования.

Одна и та же идиома может выглядеть по-разному в разных языках, либо в ней может не быть необходимости в некоторых из них.





## RAII

Resource Acquisition Is Initialization (получение ресурса есть инициализация).

*Программная идиома* объектно-ориентированного программирования.

*Основная идея* – с помощью тех или иных программных механизмов получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение – с уничтожением объекта.

*Типичный способ реализации* – организация получения доступа к ресурсу в конструкторе, а освобождения – в деструкторе.

Применяется для:

- Выделения памяти
- Открытия файлов/устройств/каналов
- Мьютексов/критических секций/других механизмов блокировки



## Пример RAII на C++

```
class File {
    const std::FILE *file;

public:
    File(const char *filename) : file(std::fopen(filename, "w+")) {
        if (!file) {
            throw std::runtime_error("file open failure");
        }
    }
    ~File() {
        std::fclose(file);
    }
    void write(const char *data) {
        if (std::fputs(data, file) == EOF) {
            throw std::runtime_error("file write failure");
        }
    }
}
```



**Конец лекции**