Лекция 7. ООП: инкапсуляция.

Евгений Линский

Черный ящик для динамического массива

```
int *array = new int[size];
array[i] = ...
...
delete[] array;
```

Задачи программиста:

- не забыть создать
- не выйти случайно за границы
- не перепутать размер
- не забыть удалить

Черный ящик для динамического массива

array.h

```
class Array {
private:
    size_t size;
    int *data;
public:
    Array(int s); // конструктор
    ~Array(); // деструктор
};
```

array.cpp

```
#include "array.h"
Array :: Array(size_t s) {
  size = s;
  data = new int [size];
}

Array :: ~Array() {
  delete[] data;
}
```

Терминология

```
#include "array.h"
main() {
  Array a(100); // вызов конструктора Array(...)
                // direct initialization
  Array a(); // the most vexing parse??!
  a.data[34] = 3434; //compilation error, data - private
} //вызов деструктора ~Array()
```

К private можно получить доступ только из методов того же класса.

Терминология:

1. int a;
   - int — тип переменной
   - a — имя переменной
2. Array a(100);
   - Array — класс (class)
   - a — объект/экземпляр класса (object/instance)
   - a.data — поле (field)
   - a.get_size() — метод (method)

Доступ к полям

array.h

```
class Array {
private:
    size_t size;
    int *data;
public:
    Array(int s);
    int get(int i);
    void set(int i, int v);
    size_t get_size();
    ~Array();
};
```

Доступ к полям

array.cpp

```
int Array::get(int i) {
  if((i < 0) || (i >= size))
      return -1;
  return data[i];
}

void Array::set(int i, int val) {
  if ((i < 0) || (i >= size))
    return;
  data[i] =  val;
}

size_t Array::get_size() {
  return size;
}
```

1. Что не так?

Доступ к полям

array.cpp

```
int Array::get(int i) {
  if((i < 0) || (i >= size))
      return -1;
  return data[i];
}

void Array::set(int i, int val) {
  if ((i < 0) || (i >= size))
    return;
  data[i] =  val;
}

size_t Array::get_size() {
  return size;
}
```

1. Что не так?
2. Обработка ошибок: get — нельзя отличить ошибку от успеха; set — об ошибке не узнаем в вызывающем коде.

C style инкапсуляция

C style

```
struct array_s {
    size_t size;
    int *data;
};
typedef array_t struct array_s;
void init(array_t *a, size_t s) {
  a->size = s;
  a->data = malloc(s * sizeof(int));
}
void deinit(array_t *a) {
  free(a->data);
}
int main() {
  array_t a;
  init(&a, 100);
  deinit(&a);
}
```

Реализация инкапсуляции

После компиляции (в объектном файле) от класса ничего “инкапсуляционного” не остается. Можно считать что компилятор преобразует программу так:

```
struct Array {
    size_t size;
    int *data;
};
void Array::Array(Array *this, size_t s) {
  this->size = s;
  this->data = new int [s];
}
void Array::~Array(Array *this) {
  delete[] this->data;
}
int main() {
  Array a;
  //просто вызов обычной фунции с таким странным именем
  Array::Array(&a, 100);
  Array::~Array(&a);
}
```

Реализация инкапсуляции

```
class Array {
private:
...
};
//void Array::set(Array *this, int i, int val)  ...
void Array::set(int i, int val) { this->data[i] = val; }

int main() {
  Array a(100);
  a.set(3, 10);
  Array b(100);
  b.set(5, 20)
}
```

- Все проверки private/public выполняются только во время компиляции (в двоичном коде уже ничего приватного нет)
- Нулевой параметр this всем функциям добавляет для нас компилятор (this можно использовать)
- В программе выше в памяти будет хранится 2 набора переменных (a и b) и 1 код функции set

Перегрузка (overloading) в C++

В C++ можно так:

```
int max(int a, int b) { ... }
int max(int a, int b, int c) { ... }
```

- Name mangling (манглинг) — “преобразование имени функции компилятором для линкера”
- в C:
  - max(int a, int b) —> max
  - max(int a, int b, int c) –> max
- в C++ (как-то так):
  - max(int a, int b) –> max_int_int
  - max(int a, int b, int c) –> max_int_int_int
- То есть можно, например, сделать два конструктора

Конструктор по умолчанию (default constructor)

```
class Array {
...
    Array(int s) { size = s; data = new int [size]; }
    Array() { size = 100; data = new int [size]; }
};
int main() {
  Array a; // вызов default конструктора
  Array b(100);
}
```

Если у класса нет конструктора или деструктора, то компилятор сгенерирует такие:

```
  Array() { }
  ~Array() { }
```

Конструктор копий (copy constructor)

```
int main() {
  int a = 3;
  int b = a;
}
```

Хочу пользоваться объектами как обычными переменными.

```
1 int main() {
2   Array a(200);
3   a.set(3, 42);
4   Array b(a); // хочу, чтобы b был копией a
5   b.set(3, 24);
6 }
```

По умолчанию у объектов (как и у структур) произойдет побайтовое копирование полей.

1. Ну и что?

Конструктор копий (copy constructor)

```
int main() {
  int a = 3;
  int b = a;
}
```

Хочу пользоваться объектами как обычными переменными.

```
1 int main() {
2   Array a(200);
3   a.set(3, 42);
4   Array b(a); // хочу, чтобы b был копией a
5   b.set(3, 24);
6 }
```

По умолчанию у объектов (как и у структур) произойдет побайтовое копирование полей.

1. Ну и что?
2. Поля data у a и у b будут указаывать на одно и то же место в памяти (b.set(...) поменяет a).

Конструктор копий (copy constructor)

```
int main() {
  int a = 3;
  int b = a;
}
```

Хочу пользоваться объектами как обычными переменными.

```
1 int main() {
2   Array a(200);
3   a.set(3, 42);
4   Array b(a); // хочу, чтобы b был копией a
5   b.set(3, 24);
6 }
```

По умолчанию у объектов (как и у структур) произойдет побайтовое копирование полей.

1. Ну и что?
2. Поля data у a и у b будут указаывать на одно и то же место в памяти (b.set(...) поменяет a).
3. В строке 6 произойдет двойной вызов delete на один и тот же адрес –> программа упадет.

Конструктор копий (copy constructor)

```
class Array {
...
  Array(const Array& a) {
    size = a.size;
    data = new int [size];
    for(size_t i = 0; i < size; i++) data[i] = a.data[i];
  }
};
```

Еще конструктор копий нужен при передаче объекта по значению в функцию:

```
void create_copy_and_fill_it(Array a) {
  ...
}
```

1. А почему Array(const Array& a), а не Array(Array a)?

Конструктор копий (copy constructor)

```
class Array {
...
  Array(const Array& a) {
    size = a.size;
    data = new int [size];
    for(size_t i = 0; i < size; i++) data[i] = a.data[i];
  }
};
```

Еще конструктор копий нужен при передаче объекта по значению в функцию:

```
void create_copy_and_fill_it(Array a) {
  ...
}
```

1. А почему Array(const Array& a), а не Array(Array a)?
2. Будет бесконечная рекурсия.

Конструктор копий (copy constructor)

```
void print(Array a) {
  for(size_t i = 0; i < a.get_size(); i++ ){
    print("%d ", a.get(i));
  }
}
```

1. Хорошая идея?

Конструктор копий (copy constructor)

```
void print(Array a) {
  for(size_t i = 0; i < a.get_size(); i++ ){
    print("%d ", a.get(i));
  }
}
```

1. Хорошая идея?
2. Нет. Лишние накладные расходы на вызов конструктора копий. Лучше передать по константной ссылке print(const Array& a).

Оператор присваивания (copy assignment)

```
int main() {
  int a = 3;
  int b = 5;
  b = a;
}
```

Хочу пользоваться объектами как обычными переменными.

```
int main() {
  Array a(200); a.set(3, 42);
  Array b(20); b.set(3, 24);
  b = a;
}
```

По умолчанию у объектов (как и у структур) произойдет побайтовое копирование полей.

1. Будут аналогичные проблемы, как и у конструктора копий.
2. В чем разница от предыдущего случая?

Оператор присваивания (copy assignment)

```
int main() {
  int a = 3;
  int b = 5;
  b = a;
}
```

Хочу пользоваться объектами как обычными переменными.

```
int main() {
  Array a(200); a.set(3, 42);
  Array b(20); b.set(3, 24);
  b = a;
}
```

По умолчанию у объектов (как и у структур) произойдет побайтовое копирование полей.

1. Будут аналогичные проблемы, как и у конструктора копий.
2. В чем разница от предыдущего случая?
3. Случай 1: объекта b не было и его надо создать как копию a.
   Случай 2: объект b уже существует, его надо “обнулить” и потом создать как копию a.

Оператор присваивания (copy assignment). Версия 1.

```
void Array::operator=(const Array& a) {
  delete [] data;
  size = a.size;
  data = new int [size];
  for(int i = 0; i < size; i++) data[i] = a.data[i];
}
```

Оператор присваивания (copy assignment). Версия 2.

```
1 Array& Array::operator=(const Array &a){
2     if (&a == this){
3         return *this;
4     }
5     delete[] data;
6     size = a.size;
7     data = new int [size];
8     for (int i = 0; i < size; i++){
9         data[i] = a.data[i];
10    }
11    return *this;
12}
```

1. Зачем строки 2-4 и 11?

Оператор присваивания (copy assignment). Версия 2.

```
1 Array& Array::operator=(const Array &a){
2     if (&a == this){
3         return *this;
4     }
5     delete[] data;
6     size = a.size;
7     data = new int [size];
8     for (int i = 0; i < size; i++){
9         data[i] = a.data[i];
10    }
11    return *this;
12}
```

1. Зачем строки 2-4 и 11?
2. 2-4: a = a;

Оператор присваивания (copy assignment). Версия 2.

```
1 Array& Array::operator=(const Array &a){
2     if (&a == this){
3         return *this;
4     }
5     delete[] data;
6     size = a.size;
7     data = new int [size];
8     for (int i = 0; i < size; i++){
9         data[i] = a.data[i];
10    }
11    return *this;
12}
```

1. Зачем строки 2-4 и 11?
2. 2-4: a = a;
3. 11: a = b = c;

Копирующая инициализация (copy initialization)

= не всегда означает «оператор присваивания»:

```
int main() {
  Array a;
  // ...
  Array b = a; // Всегда вызывается Array(const Array&).
  print(b); // Обычно вызывается Array(const Array&) для
            // создания аргумента.
}
Array print(Array a) {
  // ...
  return a; // Copy init возвращаемого значения.
            // Array(const Array&).
}
```

Смотрим на конструкторы с одним параметром.

Неявные преобразования (implicit conversion)

Копирующая инициализация выглядит как преобразование типов:

```
int main() {
  Array a(10); // Array(10) ≈ Array(int).
  Array b = 10; // Array(10) ≈ Array(int).
  print(10); // Вызывается Array(int), чтобы создать аргумент.
}
void print(Array a) { ... }
// void print(int i) { ... } // Перегрузка добавит веселья.

Array singleCellArray() {
  return 10; // Array(10) ≈ Array(int).
}
```

Explicit-конструктор (C++11)

В C++11 можно отключить конструктор для копирующей инициализации:

```
class Array {
  ...
  explicit Array(int n) { ... }
  ...
};
int main() {
  Array a(10); // Компилируется.
  Array b = 10; // Не компилируется.
  print(10); // Не компилируется.
}
```

RVO, NRVO, copy elision

```
Array createArray() {
    // Создаётся временный объект, а потом копируется?
    // Return value optimization (обязательно с C++17).
    return Array(10);
}
Array createArray() {
    Array a(10); // Создаётся локальная переменная.
    // Переменная копируется в возвращаемое значение и
    // уничтожается?
    // Необязательный named return value optimization.
    return a;
}
```

Компилятор может выкинуть вызов, даже если в копирующем конструкторе происходит что-то, кроме копирования:

```
Array(const Array &a) { printf("copy!\n"); ... }
```

Можно выкинуть не только рядом с return:

```
print(Array(10)); // Можно создать сразу на месте параметра.
```