

Севастопольский государственный университет
Кафедра «Информационные системы»

Курс лекций по дисциплине
**«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ
ПРОГРАММИРОВАНИЕ»**
(ООП)

Лектора: Пелипас Всеволод Олегович
Сметанина Татьяна Ивановна

Лекция 3

Описание класса.

Описание объекта.

Конструкторы и деструкторы классов.

Статические элементы классов.

Объектно-ориентированная технология (парадигма) программирования наиболее распространена и востребована в настоящее время.

При объектно-ориентированном подходе к программированию программа представляет собой совокупность взаимодействующих между собой объектов. Функциональную возможность и структуру объектов задают классы — типы данных, определенные пользователем.

Изучение ООП мы будем проводить на примере объектно-ориентированного языка программирования C++.

Язык C++ был создан Б. Страуструпом на базе синтаксиса языка C, он вобрал в себя некоторые концепции из других языков. Например, концепция классов взята из языка Simula, а концепция наследования — из языка Smalltalk.

Основные механизмы ООП

Во всех объектно-ориентированных языках программирования реализованы следующие основные механизмы ООП:

- инкапсуляция;
- наследование;
- полиморфизм.

1) **Инкапсуляция** — механизм, связывающий вместе код и данные, которыми он манипулирует, и одновременно защищающий их от произвольного доступа со стороны другого кода, внешнего по отношению к рассматриваемому.

2) **Наследование** — механизм, с помощью которого один объект (производного класса) приобретает свойства другого объекта (родительского, базового класса).

3) **Полиморфизм** — механизм, позволяющий использовать один и тот же метод для разных типов данных.

Описание класса

Класс является абстрактным типом данных, определяемым пользователем, и представляет собой модель реального объекта в виде данных и функций для работы с ними.

Данные класса называются **полями** (по аналогии с полями структуры), а функции класса – **методами**. Поля и методы называются **элементами** класса. Формат описания класса:

```
class <Имя класса> {  
    [private:] <описание скрытых элементов>; //по умолчанию  
    [protected:] <описание защищенных элементов>;  
    [public:]    <описание доступных элементов>;  
};           // Описание заканчивается точкой с запятой
```

private элементы класса (поля и методы) доступны только методам класса.

protected – доступны методам класса и классов-наследников.

public – доступны в пределах видимости (в том числе из методов других классов).

Описание класса

Пример:

```
class student
{
    // поля
    char name[30];           // имя
    int number;              // номер зачетки
    int year_birth, year_start; // год рождения и поступления
    int mark_m, mark_inf, mark_rus; // оценки при поступлении
    bool need_hostel;        // нуждается ли в общежитии
    public:                  // спецификатор доступа
        // методы
        void inp_data();      // заполнение полей
        void out_data();      // просмотр
    } объявление

    // вычисление среднего балла
    float average_mark() {
        return (mark_m + mark_inf + mark_rus)/3;}
    // вычисление возраста поступившего
    int age() {return (year_birth - year_start);}
};

} описание
```

Описание класса

Поля класса:

- могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс);
- могут быть описаны с модификатором **const**, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;
- могут быть описаны с модификатором **static**, но не как **auto**, **extern** и **register**.

Инициализация полей при описании не допускается.

Классы могут быть глобальными (объявленными вне любого блока) и локальными (объявленными внутри блока, например, функции или другого класса).

Описание класса

Все методы класса имеют непосредственный доступ к его скрытым полям (тела функций класса входят в область видимости **private** элементов класса).

В приведенном классе содержится два определения методов и два объявления (методы **average_mark()** и **age()**). Если тело метода определено внутри класса, он является встроенным (**inline**). Как правило, встроенными делают короткие методы. Если внутри класса записано только объявление (заголовок) метода, сам метод должен быть определен в другом месте программы с помощью операции доступа к области видимости (::):

Описание класса

```
//-----
```

```
void student::out_data ()
```

```
{
```

```
    cout<<"-----student-----"<<endl;
```

```
    cout<<"name ="<<name<<endl;
```

```
    cout<<"number ="<<number<<endl;
```

```
    cout<<"year_birth and year_start:"<<year_birth<<' '<< year_start  
<<endl;
```

```
    cout<<" mark_m, mark_inf, mark_rus:"<< mark_m<<' '<< mark_inf  
        <<' '<< mark_rus<<endl;
```

```
    cout<<" need_hostel - "<< need_hostel <<endl;
```

```
}
```

Описание объектов

Конкретные переменные типа «класс» называются **экземплярами класса**, или **объектами**. Время жизни и видимость объектов зависит от вида и места их описания и подчиняется общим правилам C++.

```
student ivanov; // объект класса student  
student *st1= new student; // указатель  
student &st2=ivanov;      // ссылка
```

Описание объектов

При создании каждого объекта выделяется память, достаточная для хранения всех полей. Доступ к элементам объекта осуществляется через операцию `.` (точка) при обращении к элементу через имя объекта и операцию `->` при обращении через указатель, например:

```
float sr=ivanov.average_mark();  
    // cout<<ivanov.average_mark()<<endl;  
cout<<st1->average_mark ()<<endl;  
cout<<st2.average_mark()<<endl;
```

Обратиться таким образом можно только к элементам со спецификатором **public**.

Примеры описания классов и объектов

Рассмотрим класс «автомобиль».



Примеры описания классов и объектов

Класс «автомобиль».

Поля: (свойства) масса, количество дверей, размеры, название

Методы: (действия) вычисление каких-то характеристик

Объекты: (конкретные экземпляры)

Феррари, Ламборджини, Ягуар, BMW и т.д.

Примеры описания классов и объектов

Рассмотрим класс «планета».



Примеры описания классов и объектов

Класс «планета».

Поля: размер орбиты, диаметр планеты, масса и т.д.

Методы: вычисление положения относительно чего-нибудь

Объекты: Марс, Венера, Земля, Сатурн

Примеры описания классов и объектов

Рассмотрим класс «стол».



Примеры описания классов и объектов

Рассмотрим класс «самолет».



Описание объектов

Получить или изменить значения элементов со спецификатором **private** можно только через обращение к соответствующим методам.

Пример1: создадим класс **class_a**:

```
class class_a {                                //по умолчанию private
    int i;
    void f(int _i) {i=_i;} //!!!!!!
};
main() {
    class_a ob1; // создадим объект
    ob1.i=10; // нельзя, т.к. i - private
    ob1.f(10); // нельзя, т.к. f() - private
}
```

Исправим:

Описание объектов

```
class class_a {  
public:    int i;    void f(int _i) {i=_i;}  
};  
main() {  
    class_a ob1;  
    ob1.i=10;  
    ob1.f(10);}
```

Не стоит поля делать общедоступными, поэтому правильный вариант:

```
class class_a {  
private: int i;  
public: void f(int _i) {i=_i;}  
};
```

Обращение к данным через методы позволяет достичь полного контроля (проверка допустимых значений, проверка возможности присваивания).

Описание объектов

Метод класса может быть описан **private**, но тогда он доступен для вызова только из методов этого же класса.

```
class class_a{
    int i;                // по умолчанию private
    void f1(){cout<<i;}
    public:
    void f2(int _i) {
        i=_i;
        f1(); // вызов защищенного метода f1()
    }
};

main() {
    class_a ob1;
    ob1.f2(12); // допустимо
    ob1.f1();   // недопустимо
}
```

Указатель `this`

Каждый объект содержит свой экземпляр полей класса. Методы класса находятся в памяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов с полями именно того объекта, для которого они были вызваны. Это обеспечивается передачей в функцию скрытого параметра `this`, в котором хранится константный указатель на вызвавший функцию объект. Указатель `this` неявно используется внутри метода для ссылок на элементы объекта. В явном виде этот указатель применяется в основном для возвращения из метода указателя (`return this;`) или ссылки (`return *this;`) на вызвавший объект.

Работу с `this` рассмотрим более подробно позже.

Конструкторы

В каждом классе есть хотя бы один метод, имя которого совпадает с именем класса. Он называется **конструктором** и вызывается автоматически при создании объекта класса.

Конструктор предназначен для инициализации объекта и выделения памяти. Рассмотрим на примере описания класса **book**:

```
class book {  
    char name[30];  
    int pages;  
public: //конструктор без параметров  
    book() {strcpy(name, "\\0" ); pages=0;  
           cout<< "конструктор без параметров "};  
    book(char *_name, int _pages=100); //конструктор с параметрами  
};  
book::book(char *_name, int _pages) {  
    pages=_pages;  
    strcpy(name, _name); cout<< "конструктор с параметрами";  
}
```

Свойства конструкторов

- Конструктор **не возвращает** значение, даже типа **void**. Нельзя получить указатель на конструктор.
- Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).
- Конструктор, вызываемый без параметров, называется **конструктором по умолчанию**.
- Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только **один** из конструкторов.

Свойства конструкторов

- Если программист не указал ни одного конструктора, компилятор создает его **автоматически**. Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов. В случае, когда класс содержит константы или ссылки, при попытке создания объекта класса будет выдана ошибка, поскольку их необходимо инициализировать конкретными значениями, а конструктор по умолчанию этого делать не умеет.
 - Конструкторы **не наследуются**.
 - Конструкторы нельзя описывать с модификаторами **const**, **virtual** и **static**.

Свойства конструкторов

- Конструкторы глобальных объектов вызываются до вызова функции **main**. Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).

- Конструктор вызывается, если в программе встретилась какая-либо из синтаксических конструкций:

имя_класса имя_объекта [(список параметров)];

// Список параметров не должен быть пустым

имя_класса (список параметров);

// Создается объект без имени (список может быть пустым)

имя_класса имя_объекта = выражение;

// Создается объект без имени и копируется

Примеры конструкторов

```
main(){  
    book b1; // вызывается конструктор без параметров  
    book b2("Garri Potter",1000); // вызывается конструктор с парам.  
    book b3("Kolobok"); // вызывается конструктор с парам.  
    // значения не указанного параметра устанавливаются по  
    // умолчанию =100  
    book ("____", 2000); // создается безымянный объект  
    book b4=book("Potter", 250); // выделяется память под объект b4  
    // в который копируется безымянный объект.  
    //...  
}
```

Примеры конструкторов

Первый из приведенных выше конструкторов является конструктором по умолчанию, поскольку его можно вызвать без параметров.

Объекты класса **book** теперь можно инициализировать различными способами, требуемый конструктор будет вызван в зависимости от списка значений в скобках.

При задании нескольких конструкторов следует дать возможность компилятору распознать нужный вариант (списки параметров должны быть различными).

Программист обязательно должен создавать конструктор при работе с динамической памятью.

Деструкторы

Деструктор — это метод класса, служащий для выполнения **завершающих действий** с объектом (освобождения памяти занимаемой объектом).

Имя деструктора начинается с тильды (~), непосредственно за которой следует имя класса.

Деструктор вызывается **автоматически**, когда объект выходит из области видимости:

- для локальных объектов — при выходе из блока, в котором они объявлены;
- для глобальных — перед выходом из **main**;
- для объектов, заданных через указатели — деструктор вызывается неявно при использовании операции **delete**.

Свойства деструкторов

- **не имеет** аргументов и возвращаемого значения (!!!);
- **не может** быть объявлен как **const** или **static**;
- **не наследуется**;
- **может** быть виртуальным.

Указатель на деструктор определить **нельзя**.

Если деструктор явным образом не определен, компилятор автоматически создает пустой деструктор.

У одного класса может быть только один деструктор.

Описывать в классе деструктор явным образом требуется в случае, когда объект содержит указатели на память, выделяемую динамически – иначе при уничтожении объекта память, на которую ссылались его поля-указатели, не будет помечена как свободная.

Пример описания класса book

```
#include<iostream>
#include<string.h>
using namespace std;
class book {
    char *name; // указатель на динамическую переменную
    int pages;
public:
    book();      // объявление конструктора
    ~book();     // объявление деструктора
};
book::book(){   // описание конструктора
    name= new char[30]; cout<<"конструктор"<<endl;
    strcpy(name, "NEW BOOK/0"); pages=100;
}
book::~~book(){ // описание деструктора
    delete []name; cout<<"деструктор"<<endl;
}
main(){ book ob1; book *ob2;
    ob2=new book;
    delete ob2;
}
```

Деструкторы

Деструктор *можно вызвать явным образом* путем указания полностью уточненного имени, например:

```
class B { //...  
    B();    // конструктор  
    ~B();   // деструктор  
};  
main(){ //...  
B *ob= new B; // создаем указатель  
    //...  
ob -> ~B(); // вызов деструктора  
    //...  
}
```

Это может понадобиться для объектов, которым с помощью перегруженной операции **new** выделялся конкретный адрес памяти. Без необходимости явно вызывать деструктор объекта **не рекомендуется**.

Пример с использованием динамических переменных

Опишем класс – динамический массив из целых чисел

```
#include<iostream.h>
```

```
class mas {
```

```
    int *x; // массив
```

```
    int size; // размер
```

```
public:
```

```
    input_x(); // объявление функции заполнения
```

```
    int max(); // объявление функции поиска максимального
```

```
mas(); // объявление конструктора по умолчанию
```

```
mas(int k); // объявление конструктора с параметром
```

```
~mas(); // объявление деструктора
```

```
};
```

Пример с использованием динамических переменных

// описание конструктора по умолчанию без параметров

```
mas::mas() {      cout<<"конструктор по умолчанию"<<endl;
cout<<"введите размер массива";
cin>>size; //заполнение поля size
x=new int[size]; //выделение памяти
}
```

//описание конструктора с параметрами

```
mas::mas(int k) {  cout<<"конструктор с параметром"<<endl;
size=k; // заполнение поля size
x=new int[size]; //выделение памяти
}
```

// описание деструктора

```
mas::~~mas() {      cout<<"деструктор"<<endl;
delete []x; // освобождение памяти
}
```

Пример с использованием динамических переменных

//описание методов класса **mas**

```
mas::input_x() { // заполнение  
cout<<"введите элементы массива"<<endl;  
for(int i=0; i<size; i++)  
    cin>>x[i];  
}
```

```
int mas::max() { // поиск максимального  
    int max=x[0];  
for(int i=0; i<size; i++)  
    if (x[i]>max) max=x[i];  
    return max;  
}
```

Пример с использованием динамических переменных

```
//-----  
main() {  
    // создание объектов класса mass  
    mas ob1;      // вызов конструктора по умолчанию  
    mas ob2(10);  // вызов конструктора с параметрами  
  
    // вызов методов  
    ob1.input_x(); // заполнение поле  
    ob2.input_x(); // заполнение поле  
  
    cout<<"max="<<ob1.max()<<endl  
    cout<<"max="<<ob2.max()<<endl  
  
    // неявный вызов деструктора  
}
```

```
конструктор  
введите размер массива 3  
конструктор с параметрами  
введите элементы массива  
3 2 1  
введите элементы массива  
1 2 3 4 -45 45 56 2 -10 19  
max=3  
max=56  
деструктор  
деструктор
```

Пример с использованием динамических переменных

```
//----- 2 версия -----  
main() {                               // создание указателя  
    mas *ob;  
    ob=new mas();                      // вызов конструктора по умолчанию  
    ob->input_x();                     // заполнение полей  
    cout<<"max="<<ob->max()<<endl;  
    delete ob; // неявный вызов деструктора  
}
```

```
конструктор по умолчанию  
введите размер массива3  
введите элементы массива  
89  
7  
-87  
max=89  
деструктор
```

// ob=new mas(4);

```
конструктор с параметром  
введите элементы массива  
1  
-23  
34  
3  
max=34  
деструктор
```

Конструктор копирования

Конструктор копирования — это специальный вид конструктора, получающий в качестве единственного параметра указатель на объект этого же класса:

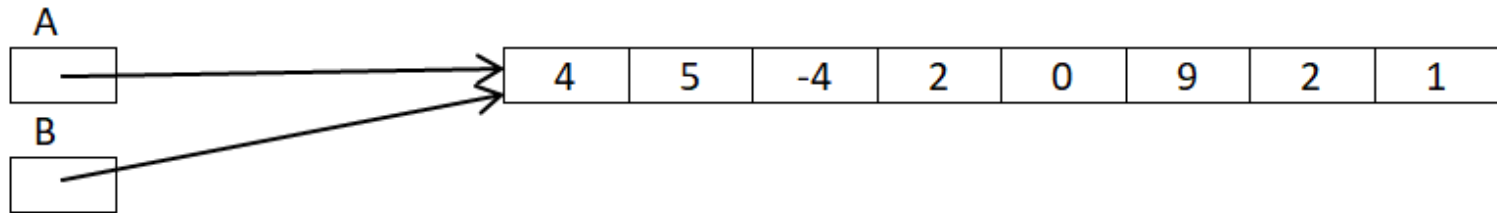
`T::T(const T& ob) { ... /* Тело конструктора класса T */ }`
где **T** - имя класса.

Этот конструктор вызывается в тех случаях, когда новый объект создается путем копирования существующего:

- при описании нового объекта с инициализацией другим объектом
- при передаче объекта в функцию по значению;
- при возврате объекта из функции.

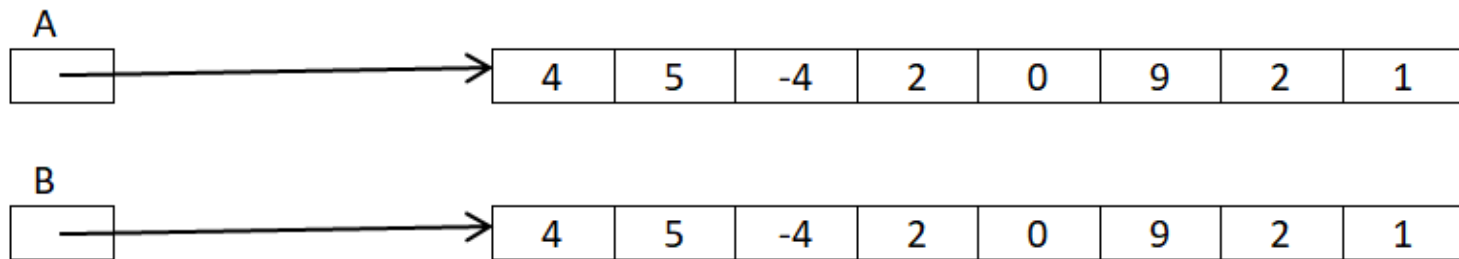
Если программист не указал ни одного конструктора копирования, компилятор создает его автоматически. Такой конструктор выполняет поэлементное копирование полей. Такое копирование называется **поверхностным** (*shallow copy*). Если класс содержит указатели или ссылки, это, скорее всего, будет неправильным, поскольку и копия, и оригинал будут указывать на одну и ту же область памяти.

Глубокое и поверхностное копирование



```
int *B;  
B=A;  
delete [] A;
```

Глубокое и поверхностное копирование



```
int *B= new int [8];  
for(int i=0; i<8;i++) B[i]=A[i];  
delete [] A;
```


Конструктор копирования

Если один из объектов прекратит свое существование, а второй объект будет указывать на область памяти уже освобожденную, то возникнет паразитный указатель, а работа программы **не будет** стабильной и предсказуемой.

Во избежание подобных проблем, необходимо вместо стандартного конструктора копирования использовать собственный, который будет осуществлять *глубокое копирование* с перемещением значений полей, находящихся в динамической памяти.

Конструктор копирования

Запишем конструктор копирования для класса **mas**:

```
mas::mas(mas &ob){ cout<<"конструктор копирования"<<endl;  
    size=ob.size; // заполнение поля size  
    x=new int[size]; //выделение памяти  
    for(int i=0; i<size; i++) //копирование элементов массива  
        x[i]=ob.x[i];  
}
```

```
main() {  
    mas ob1; ob1.input_x();  
//...  
    mas ob3(ob1); //вызов конструктора копирования  
//...  
}
```

Статические элементы класса

С помощью модификатора **static** можно описать статические поля и методы класса.

Статические поля применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемый всеми объектами ресурс. Эти поля существуют для всех объектов класса в **единственном** экземпляре, то есть **не дублируются**.

Память под статическое поле выделяется **один раз** при его инициализации независимо от числа созданных объектов (и даже при их отсутствии) и инициализируется с помощью операции доступа к области действия, а не операции выбора:

Статические элементы класса

```
class A {  
public: static int count;  
};  
int A::count=10; //инициализация произвольным значением  
main(){  
    A *a, b;  
    cout<<" " <<A::count;  
    cout<<" "<< b.count;  
    cout<<" "<<a->count; } // будет выведено 10 10 10
```

Статические поля доступны как через имя класса, так и через имя объекта. На статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как **private**, нельзя инициализировать с помощью операции доступа к области видимости. Им можно присвоить значения только с помощью статических методов.

Память, занимаемая статическим полем, не учитывается при определении размера объекта операцией **sizeof**.

Статические элементы класса

Статические поля **нельзя** инициализировать в конструкторе, так как они создаются до создания любого объекта.

Классическое применение статических полей – подсчет объектов. Для этого в классе объявляется целочисленное поле, которое увеличивается в конструкторе и уменьшается в деструкторе.

Статические методы могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса, поскольку им не передается скрытый указатель `this`. Обращение к статическим методам производится так же, как к статическим полям – либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта.

Статические методы **не могут** быть константными (**const**) и виртуальными (**virtual**).

Пример с использованием статических методов

```
class A {  
    static int count; // по умолчанию private  
    public: static void f1(){count++;}  
};
```

```
int A::count=10; // инициализация произвольным значением
```

```
main(){  
    A *a, b;  
    // изменение поля с помощью статического метода  
    A::f1();  
    b.f1();  
    a->f1();  
}
```

Рекомендации по составу класса

Как правило, класс как тип, определенный пользователем, должен содержать скрытые (**private**) поля и следующие функции:

- конструкторы, определяющие, как инициализируются объекты класса;
- набор методов, реализующих свойства класса;
- деструктор класса;
- конструктор копирования, реализованный программистом (при работе с динамической памятью).