



## Тема II

# Основы объектно-ориентированного программирования

# Определение ООП

**Объектно-ориентированное программирование (ООП)** - методология программирования, основанная на представлении программы в виде совокупности **объектов**, каждый из которых является экземпляром определенного **класса**, а классы могут образовывать **иерархию** наследования.

# Преимущества использования ООП

- уменьшение сложности программного обеспечения;
- повышение надежности программного обеспечения;
- обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов;
- обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

# Классы и объекты

- **Класс** – определенный пользователем проблемно-ориентированный тип данных, описывающий внутреннюю структуру объектов, которые являются его экземплярами.
- **Объект (экземпляр класса)** находится в таком же отношении к своему классу, в каком переменная находится по отношению к своему типу.

## Категории программистов при объектно-ориентированном подходе

- **Разработчики класса** определяют назначение класса, его интерфейс, реализуют интерфейс класса в рамках предоставленных им средств
- **Пользователи класса** создают объекты предоставленного в их распоряжение класса и работают с этими объектами, используя интерфейс, предоставленный разработчиками класса

Детали реализации класса обычно скрываются разработчиками от пользователей этого класса.

# Разграничения для различных категорий программистов

Разработчики класса	Пользователи класса
Описание класса (интерфейс)	Описание класса (интерфейс) в режиме чтения
Реализация класса	Использование класса (создание объектов, работа с объектами, их корректное уничтожение)

# Состав класса

В состав класса входят данные и функции. В совокупности они называются **членами класса**.

- Данные, входящие в класс, называются **данными-членами** или **полями**.
- Функции, принадлежащие классу, называют **функциями-членами** или **методами**.

Объединение в одной конструкции полей и методов называется **инкапсуляцией**

# Описание класса в языке C++ (в простейшем случае)

```
class имя_класса {  
    спецификаторы_доступа  
    описания_полей  
    описания_методов  
};
```

**Описания полей** по формату совпадают с описаниями переменных .

**Описания методов** представляют собой заголовки функций.

**Спецификаторы доступа** служат для разграничения полномочий между разработчиками и пользователями класса



# Внутренние и внешние методы

Методы класса делятся на **внутренние** и **внешние**.

- Внутренние методы реализованы в рамках описания класса; их текст доступен для чтения пользователям класса.
- Внешние методы реализованы как отдельные функции. Для того, чтобы показать, что это – не обычная функция, а метод конкретного класса, к имени метода добавляется имя класса с помощью операции **::** (расширение области видимости). Текст внешних методов может быть скрыт от пользователей класса.

# Пример описания и реализации класса (точка на плоскости)

## I) Реализация с использованием внутренних методов

```
class Point2D {  
    double x, y; // поля класса  
    double Module () { // внутренний метод  
        return sqrt(x*x+y*y);  
    }  
};
```

# Пример описания и реализации класса (точка на плоскости)

## 2) Реализация с использованием внешних методов

```
class Point2D {  
    double x, y; // поля класса  
    double Module (); // внешний метод  
};
```

```
// внешний метод класса Point2D  
double Point2D::Module () {  
    return sqrt(x*x+y*y);  
}
```

# Использование класса Point2D

```
// ВНИМАНИЕ! Приведенный текст не работает  
// (ошибки компиляции)  
  
Point2D p; // создание объекта  
p.x = 12; // обращение к полям объекта  
p.y = -3;  
cout << p.Module();  
    // вызов метода в применении к объекту p
```

# Недостатки приведенной реализации

- После создания объекта значения его полей не определены (*решается написанием конструкторов*)
- Пользователь класса не имеет доступа к элементам (полям и методам) созданного объекта (*решается использованием спецификаторов доступа*)

# Спецификаторы доступа

- **private:** - члены класса, доступные только разработчикам класса (т.е. только при реализации методов этого класса)
- **public:** - члены класса, доступные как разработчикам, так и пользователям класса
- **protected:** - члены класса, доступные разработчикам класса и разработчикам классов-потомков

Спецификатор по умолчанию – **private:**

# Точка на плоскости (вариант 2)

```
class Point2D {  
public:  
    double x, y; // поля класса  
    double Module (); // внешний метод  
};
```

```
// ошибки компиляции исчезли!  
  
Point2D p; // создание объекта  
p.x = 12; // обращение к полям объекта  
p.y = -3;  
cout << p.Module();  
    // вызов метода в применении к объекту p
```

# Рекомендации по использованию разграничений доступа

- Поля класса следует максимально защитить от пользователей класса, объявив их со спецификатором **private**:
- Для работы с объектами пользователь может пользоваться методами этих объектов, объявленными со спецификатором **public**:
- Для получения информации о характеристиках объекта (в том числе о защищённых полях) должны быть написаны т.н. **get**-методы
- Для изменения характеристик объекта (в том числе защищённых полей) должны быть написаны т.н. **set**-методы



# Точка на плоскости (вариант 3)

## Описание класса

```
class Point2D {  
private:  
    double x, y; // защищенные поля класса  
public:  
    double GetX(); // get-методы для полей  
    double GetY();  
    void SetX(double ax); // set-методы для полей  
    void SetY(double ay);  
    double Module ();  
};
```

# Точка на плоскости (вариант 3)

## Реализация класса

```
double Point2D::GetX() {  
    return x;  
}  
double Point2D::GetY() {  
    return y;  
}  
void Point2D::SetX(double ax) {  
    x = ax;  
}  
void Point2D::SetY(double ay) {  
    y = ay;  
}  
double Point2D::Module () {  
    return sqrt(x*x+y*y);  
}
```

# Точка на плоскости (вариант 3)

## Использование класса

```
Point2D p; // создание объекта

p.x = 12; // обращение к защищённым полям объекта
p.y = -3; // приведёт к ошибкам компиляции

p.SetX(12); // надо использовать set-методы
p.SetY(-3);

cout << p.Module();
```

# Константные методы

Методы, не изменяющие значения полей объекта, для которого эти методы применяются, называются **КОНСТАНТНЫМИ**.

Для объявления константного метода необходимо записать слово **const** в конце заголовка такого метода (как в описании класса, так и в реализации)

*Преимущества константных методов:*

- дополнительный контроль компилятора за правильностью написания;
- в функциях, в которые объект передаётся по константной ссылке, для этого объекта можно вызывать только константные методы.

# Точка на плоскости (вариант 4)

## Описание и реализация класса

```
class Point2D {  
private:  
    double x, y; // защищенные поля класса  
public:  
    double GetX() const; // get-методы для полей  
    double GetY() const; // объявляются константными  
    void SetX(double ax); // set-методы для полей  
    void SetY(double ay);  
    double Module () const; // константный метод  
};
```

```
double Point2D::GetX() const{  
    return x;  
}
```

```
// другие методы реализованы аналогично
```

# Точка на плоскости (вариант 4)

## Использование класса

```
void PrintPoint (const Point2D& p) {  
    cout << p.GetX() << " " << p.GetY() << endl;  
    cout << p.Module() << endl;  
}
```

Если методы класса `Point2D`, вызываемые в функции `PrintPoint`, не объявлены константными, возникнет ошибка компиляции этого кода!

# Конструкторы

**Конструктор** – специальный метод, который неявно вызывается при создании нового объекта.

Назначение конструктора – выполнение дополнительных действий по инициализации объекта (например, задание начальных значений полей создаваемого объекта).

```
Point2D p; // для объекта p неявно вызывается
           // конструктор

Point2D* t = new Point2D;
           // после выделения динамической памяти
           // для вновь созданного объекта также
           // неявно вызывается конструктор
```

# Описание конструктора

- Конструктор может быть как внутренним, так и внешним методом;
- Имя конструктора совпадает с именем класса;
- Конструктор не возвращает никакого значения (даже **void**);
- За счет механизма перегрузки может быть создано несколько конструкторов, различающихся набором параметров;
- Если ни одного конструктора не написано, реализуется т.н. **конструктор по умолчанию** без параметров. Этот конструктор не выполняет никаких дополнительных действий.



# Точка на плоскости (вариант 5)

## Описание класса

```
class Point2D {  
private:  
    double x, y; // защищенные поля класса  
public:  
    Point2D(); // конструктор без параметров  
                // строит точку в начале координат  
    Point2D(double, double);  
        // конструктор, в который в качестве  
        // параметров передаются координаты точки  
    double GetX() const; // get-методы для полей  
    double GetY() const; // объявляются константными  
    void SetX(double ax); // set-методы для полей  
    void SetY(double ay);  
    double Module () const; // константный метод  
};
```

# Точка на плоскости (вариант 5)

## Реализация конструкторов

```
Point2D::Point2D() {  
    x = 0; y = 0;  
}  
Point2D::Point2D(double ax, double ay) {  
    x = ax; y = ay;  
}
```

За счет использования параметров по умолчанию можно обойтись одним конструктором:

```
class Point2D {  
...  
    Point2D(double = 0, double = 0);  
...  
};
```

# Вызов нужного конструктора

```
Point2D p1; // для объекта p1 вызывается
            // конструктор без параметров, который
            // строит точку в начале координат

Point2D p2(5); // строится точка p2 с координатами
              // (5; 0)

Point2D* t = new Point2D(10, -3);
            // после выделения динамической памяти
            // для вновь созданного объекта
            // вызывается конструктор с двумя параметрами
```

# Список инициализаторов в конструкторе

В реализации конструктора может быть задан **список инициализаторов**, который записывается после заголовка конструктора через знак `:`. Элементы списка разделяются запятыми.

Каждый элемент списка содержит имя поля и способ его инициализации в круглых скобках.

Для обычных полей использование списка инициализации допустимо, но не обязательно.

*Список инициализации обязателен:*

- если в качестве поля задаётся объект другого класса, для которого должен быть запущен конструктор;
- для вызова конструктора предка при использовании механизма наследования

# Пример конструктора со списком инициализаторов

```
Point2D::Point2D(double ax, double ay):  
    x(ax), y(ay) { }
```

Обратите внимание на то, что тело конструктора стало пустым!

# Деструкторы

**Деструктор** – специальный метод, который неявно вызывается при корректном уничтожении объекта.

Назначение деструктора – выполнение дополнительных действий по освобождению ресурсов, захваченных при создании объекта или в процессе работы с ним.

Если деструктор не написан, вызывается т.н. **деструктор по умолчанию**, который не выполняет никаких дополнительных действий.

# Вызовы деструктора

```
void my_func() {  
    Point2D p1;  
    Point2D* t1 = new Point2D(10, -3);  
    Point2D* t2 = new Point2D(0, 16);  
  
    ...  
  
    delete t1;  
        // вызывается деструктор для  
        // объекта, адрес которого хранится в t1  
  
    return;  
        // вызывается деструктор для локального  
        // объекта p1, время жизни которого  
        // истекает  
}
```

**Для объекта, адрес которого хранится в t2,  
деструктор НЕ ВЫЗЫВАЕТСЯ!**

# Описание деструктора

- Деструктор может быть как внутренним, так и внешним методом;
- Имя деструктора совпадает с именем класса, перед которым стоит знак ~;
- Деструктор не имеет параметров.



# Пример 2: класс «человек»

## Описание класса

```
class Person {  
private:  
    char* name;  
        // единственное поле - имя человека  
public:  
    Person(const char*);  
        // конструктор, в который передается имя  
    ~Person(); // деструктор  
    const char* GetName() const;  
};
```

# Пример 2: класс «человек»

## Реализация конструктора и деструктора

```
Person::Person(const char* aname) {  
    name = new char [strlen(aaname)+1];  
    strcpy(name, aaname);  
}  
Person::~~Person() {  
    delete [] name;  
}
```

# Конструктор копирования

Конструктор копирования – специальный конструктор, который получает в качестве параметра константную ссылку на объект этого же типа:

```
class Person {  
...  
public:  
    Person(const Person&);  
        // конструктор копирования  
...  
};
```

# Вызовы конструктора копирования

Конструктор копирования вызывается:

1. при создании нового объекта с инициализацией существующим объектом:

```
Person n1("Serge Kashkevich");  
Person n2(n1);  
    // запуск конструктора копирования  
Person n3 = n1;  
    // также запуск конструктора копирования
```

# Вызовы конструктора копирования

Конструктор копирования вызывается:

2. при передаче в функцию объекта по значению:

```
void my_func(Person t) {  
...  
}  
...  
Person n1("Serge Kashkevich");  
...  
my_func(n1);  
// формальный параметр t создается из объекта n1  
// конструктором копирования
```

# Вызовы конструктора копирования

Конструктор копирования вызывается:

3. при выходе из функции, возвращающей объект:

```
Person my_func() {  
    char s[100];  
    // формирование имени  
    Person t(s);  
    ...  
    return t;  
    // объект t будет уничтожен после выхода из  
    // функции, но перед этим на его основе  
    // конструктором копирования будет сформирован  
    // результат  
}  
...  
cout << my_func().GetName();
```

# Когда нужно писать конструктор копирования?

Если конструктор копирования не написан, работает т.н. **стандартный конструктор копирования**

Стандартный конструктор копирует значения всех полей источника в создаваемый объект. Если этого достаточно, писать собственный конструктор копирования не нужно.

Для класса `Point2D` можно воспользоваться стандартным конструктором.

Класс `Person` требует написания собственного конструктора копирования, т.к. необходимо выделить память для хранения новой строки с именем.

# Пример 2: класс «человек»

## Реализация конструктора копирования

```
Person::Person(const Person& d) {  
    name = new char [strlen(d.name)+1];  
    strcpy(name, d.name);  
}
```



# Указатель `this`

При работе методов класса специальный указатель **`this`** содержит адрес объекта, для которого **вызывается этот метод.**

```
Person::Person(const Person& d) {  
    this->name = new char [strlen(d.name)+1];  
    strcpy(this->name, d.name);  
} // здесь this можно и не писать...
```

```
Person Person::my_func() {  
    Person p(*this);  
    // работаем с объектом p  
    ...  
    return p;  
} // здесь без this не обойтись...
```

# Константные и статические поля

- Поле класса может быть объявлено константным с помощью модификатора **const**. Константные поля могут содержать различные значения для различных объектов. Эти значения задаются в конструкторе и в дальнейшем не могут быть изменены.
- Поле класса может быть объявлено статическим с помощью модификатора **static**. При реализации класса необходимо определить поле, общее для всех объектов этого класса. Если статическое поле объявлено как **public**, к нему можно обращаться и пользователям класса

# Пример работы с константными и статическими полями

```
class Person {  
    static int next_ID;  
    const int ID;  
...  
public:  
    int GetID() { return ID; }  
...  
};
```

```
int Person::next_ID = 0;  
  
Person::Person(const Person& d) : ID(++next_ID) {  
...  
}  
  
// такой же инициализатор пишем для других конструкторов
```

# Выброс исключений в методах класса

Методы класса могут выбрасывать исключения, информируя пользователей класса о неправильной работе с объектами класса.

Механизм выброса исключений является одним из допустимых способов передачи сообщений пользователям класса. Для конструкторов это – единственный способ информации пользователей о невозможности создать объект.

# Пример выброса исключений в конструкторе

Поле «имя» для каждого объекта класса Person не должно быть пустой строкой. Кроме того, недопустима передача в конструктор пустого указателя.

```
Person::Person(const char* aname) {  
    if (aname == NULL)  
        throw "Invalid parameter (NULL)";  
    if (aname[0] == '\\0')  
        throw "Invalid name length";  
    name = new char [strlen(aaname)+1];  
    strcpy(name, aaname);  
}
```

# Перехват выброшенных ИСКЛЮЧЕНИЙ

```
char s[200];

try {
    cout << "Введите имя персонажа: ";
    cin.getline(s, 200);
    Person t(s);
}
catch (...) {
    cout << "Имя персонажа задано неверно!" << endl;
}
```

# Перехват выброшенных исключений (2-й способ)

```
char s[200];

try {
    cout << "Введите имя персонажа: ";
    cin.getline(s, 200);
    Person t(s);
}
catch (const char * s) {
    cout << s << endl;
}
```

# Использование стандартного класса exception

Для эффективной обработки выбрасываемых исключений можно воспользоваться стандартным классом exception. В его состав, помимо других методов, входят:

- конструктор, получающий в качестве параметра строку символов – текст сообщения, связанного с исключением;
- метод what(), который возвращает строку, переданную в конструктор.



# Пример использования класса exception

```
Person::Person(const char* aname) {  
    if (aname == NULL)  
        throw exception("Invalid parameter (NULL)");  
    if (aname[0] == '\\0')  
        throw exception("Invalid parameter (empty)");  
    name = new char [strlen(aaname)+1];  
    strcpy(name, aaname);  
}
```

```
try {  
    Person t(s);  
}  
catch (const exception& e) {  
    cout << e.what() << endl;  
}
```

# Дружественные функции и классы

Иногда желательно иметь непосредственный доступ извне к скрытым полям класса. Это достигается за счет использования **дружественных** функций и классов.

Дружественная функция, которая может быть обычной функцией или методом другого класса, должна быть описана внутри класса, к скрытым полям которого она должна иметь доступ, с описателем **friend**, и получать в качестве параметра объект класса или ссылку на этот объект.

Если все методы какого-то класса должны иметь доступ к скрытым полям другого, то весь такой класс можно объявить дружественным.

# Примеры задания дружественных функций и классов

```
class Point2D {  
...  
    friend my_func(Point2D&);  
    friend another_class::method(Point2D, int);  
...  
    friend class some_class;  
};
```

# Использование стража включения

Для предотвращения ошибок, связанных с повторным описанием класса, логично обрамлять это описание стражем включения:

```
#ifndef __Point2D_defined__  
#define __Point2D_defined__  
class Point2D {  
  
...  
};  
#endif
```