# Структуры
# Классы

Лекция #8

Пустовалова О.Г.
доцент. каф. мат.мод.
ИММиКН ЮФУ

# Содержание


- Структуры
- Классы
- Конструкторы и деструкторы
- Перегрузка операторов
- Дружественные функции и классы

# Структуры

# Структуры

- Структура — это объединение различных переменных с разными типами данных, которому можно присвоить имя.

- Структуры обычно используют, когда необходимо сгруппировать некоторые данные, например, запись из базы данных или контакт из книги адресов.

**Дом**:
- город
- улица,
- номер дома
- стоимость

**Студент**:
- фамилия
- имя
- курс
- группа

# Структуры

```
struct имя_структуры
{
        компоненты_структуры
};
```

```
struct user
{
        std::string login;
        std::string password;
        int id;
};
```

```
struct baggage
{
        int code;
        std::string name;
        double weight;
};
```

# Структуры

```
#include <iostream>
using namespace std;

struct point {
        int x;
        int y;
};

void main() {
        point p1 = { 4, 6 };// p1.x=4 p1.y=6

        point p2;
        p2.x = 3; p2.y = 5;

        p1 = p2;// p1.x=3 p1.y=5
        cout << p1.x << " " << p1.y << endl;
}
```

# Структуры

```
#include <iostream>
using namespace std;

struct point {
	int x;
	int y;
};
void main() {

	point pp[5];   // массив

	for (int i = 0; i < 5; i++)
	{
		pp[i].x = i;
		pp[i].y = i*i;

		cout << pp[i].x << " " << pp[i].y << endl;
	}
}
```

# Структуры

```
#include "stdafx.h"
#include <iostream>
using namespace std;

struct point {
        int x;
        int y;
};

typedef struct point Point; //Определяем новый тип

void main() {

   point p1 = { 3, 5 }; //Обращение через имя структуры
   cout << p1.x << " " << p1.y << endl;

   Point p2 = { 4, 6 }; //Обращение через новый тип
   cout << p2.x << " " << p2.y << endl;

}
```

# Классы

# Классы

- Класс определяет **новый тип данных**, который соединяет в себе код и данные.
- По синтаксису класс аналогичен **структуре**.
- Однако класс также может включать **функции**, а не только данные.



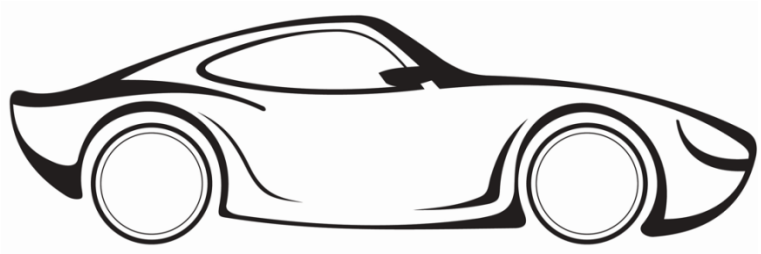
Класс тарелка



Класс кубик

# Классы

- **Классы в C++** — это абстракция описывающая методы, свойства, ещё не существующих объектов.
- **Объекты** — конкретное представление абстракции, имеющее свои свойства и методы.
- Созданные объекты на основе одного класса называются **экземплярами** этого класса.
- Эти объекты могут иметь различное поведение, свойства, но все равно будут являться **объектами одного класса**.



Класс автомобиль



Объекты класса

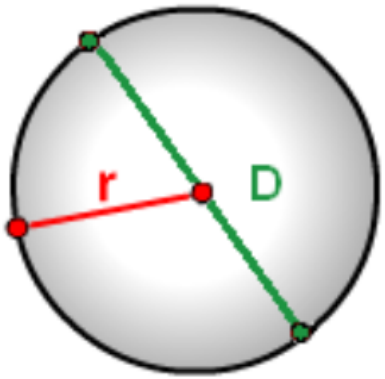
# Классы

- Классы в C++ предназначены для описания **объектов**.
- Их применение позволяет группировать для каждого объекта **данные** и **методы**, которые оперируют этими данными, в одной переменной.

```
class имя_класса { описание свойств и методов класса }
```

- **свойства** класса - это переменные
- **методы** - функции

## Свойства




**r** - радиус окружности

**D** - диаметр окружности

$\pi \approx 3.14$

## Методы

$$L = \pi D = 2\pi r$$

$$S = \pi r^2 = \frac{\pi}{4} D^2$$

# Объявление классов

```
class /*имя класса*/
{
      private:
/* список свойств и методов для использования внутри класса */

      public:
/* список методов доступных другим функциям и объектам программы */

      protected:
/*список средств, доступных при наследовании*/

};
```

# Классы. Пример 1. Класс ТОЧКА. Часть 1

```
#include <iostream>
#include <string>
using namespace std;

class point{
        private: int id ;
        public: int x, y;};

int main() {
        point p;
        p.x = 3; p.y = 5;
        cout << p.x << " " << p.y << endl;

        //p.id = 123;
return 0;
}
```

Нельзя обращаться к приватному полю за пределами класса

# Классы. Пример 1. Класс ТОЧКА. Часть 2

```
#include <iostream>
#include <string>
using namespace std;

class point{
        int id ;

        public: int x, y;};

int main() {
        point p;
        p.x = 3; p.y = 5;
        cout << p.x << " " << p.y << endl;

        //p.id = 123;
return 0;
}
```



По умолчанию поля **private**

Нельзя обращаться к приватному полю за пределами класса

# Классы. Модификаторы доступа

- **private** - свойства и методы используются только внутри класса;
- **public** - доступны другим функциям и объектам программы;
- **protected -** доступны при наследовании.

# Классы. Пример 2. Класс КРУГ. Часть 1

```
#include <iostream>
using namespace std;

const double PI = 3.14159265358979238463;

class circle // класс круг
{
	public:

	double x, y; //координаты  центра
	double r; //радиус

	double square() { return PI * r*r; }
	double length() { return 2 * PI*r; }
};
```

# Классы. Пример 2. Класс КРУГ. Часть 2

```
int main()
{
	circle A; // объект класса circle

	A.x = 0; // координаты центра и радиус
	A.y = 0;
	A.r = 5;

	cout << A.square() << " " << A.length();

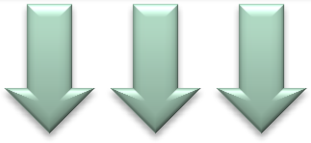
return 0;
}
```

# Классы. Не public, а private

Приватные поля повышают безопасность

```
class point
{
        private:
        int id ;
        int x, y;
};
```

Как инициализировать поля?



Использовать конструкторы

```
int main() {
    point p;
    p.x = 3; p.y = 5;
    cou
    p.
    ret
}
```

int point::x

член "point::x" (объявлено в строке 12) недоступно

# Классы. Конструктор для инициализации полей

```
...
class point {
        private:
                int id;
                int x, y;
        public:
        point(int xx, int yy) { x = xx; y = yy; };
```

Конструктор для задания значений полей
Всегда имеет тоже имя, что и класс
Всегда расположен в public

```
        // Функция вывода на экран
        void printXY() {
        cout << " x = " << x<<endl;
        cout << " y = " << y << endl;};
        };

int main() {
        point p(3,5); // Инициализируем x=3 и y=5
        p.printXY();
return 0;
}
```

# Классы. Конструктор класса

- **Конструктор класса** – это специальный метод (функция) класса. Конструктор вызывается при создании объекта класса.

Как правило, конструктор используется для:

- **выделения памяти** под объект класса;
- **начальной инициализации** внутренних данных класса.

**Имя конструктора класса совпадает с именем класса**

# Классы. Конструктор класса

- Вызов конструктора осуществляется при **создании** объекта класса.
- Конструктор класса вызывается компилятором.
- Конструктор может иметь **любое количество параметров**.
- Также конструктор может быть **без параметров** (конструктор по умолчанию).

**Имя конструктора класса совпадает с именем класса**

# Классы. Конструктор по умолчанию

```
...
class point {
        private:
                int id;
                int x, y;
        public:
        point() { x = 0; y = 0; };

        // Функция вывода на экран
        void printXY() {
        cout << " x = " << x<<endl;
        cout << " y = " << y << endl;};
        };

int main() {
        point p; // По умолчанию x=0 и y=0
        p.printXY();
return 0;
}
```

Конструктор без параметров
или **конструктор по умолчанию**
Всегда расположен в public

# Классы. Деструктор класса

- **Деструктор** — специальный метод класса, который служит для уничтожения элементов класса.
- Чаще всего его используют тогда, когда в конструкторе, при создании объекта класса, **динамически был выделен участок памяти и необходимо эту память очистить**, если эти значения уже не нужны для дальнейшей работы программы.
- Для простых классов (тех, которые только инициализируют значения обычных переменных-членов) деструктор не нужен, так как C++ автоматически очистит память сам.

# Классы. Деструктор

```
...
class point {
private:
        int x, y;

public:
point() { x = 0; y = 0; }; // Конструктор без параметров
point(int xx, int yy) { x =xx; y = yy; }; // Конструктор

// Деструктор
~point() { cout << "destructor works"<<endl; };
};

int main() {
point p1;// работает конструктор без параметров
point p2(3,5);// работает конструктор без параметров

return 0;
}
```

```
Destructor
Destructor
```

# Классы. Деструктор. Динамическая память. Часть 1

```
...
class MyArr {
private:
	int *arr;
	int len;
public:
	MyArr(int length) // конструктор
	{
		assert(length > 0);
		arr = new int[length];
		len = length;
	}

	~MyArr() // деструктор
	{
		delete[] arr;
		cout << "Destructor" << endl;
	}
```

# Классы. Деструктор. Динамическая память. Часть 2

```
void setValue(int index, int value) { arr[index] = value; }

int getValue(int index) { return arr[index]; }

int getLength() { return len; }
}; // описание класса закончилось

int main() {

        int n = 10;
        MyArr arr(n); // выделяем 10 целочисленных значений
        for (int count = 0; count < n; ++count)
                arr.setValue(count, count + 1);

        cout << "The value of element "<< n/2<<" is "
        << arr.getValue(n/2)<<endl;

return 0;}
```

# Классы. Конструкторы и деструкторы

1. **конструктор и деструктор, всегда объявляют в разделе public**;
2. при объявлении конструктора, **тип данных возвращаемого значения не указывается**;
3. **у деструктора также нет типа данных для возвращаемого значения**, к тому же деструктору нельзя передавать никаких параметров;
4. **имя класса и конструктора должно быть идентично**;
5. **имя деструктора идентично имени конструктора, но с приставкой ~** ;

# Классы. Конструкторы и деструкторы

6. В классе **допустимо создавать несколько конструкторов**, если это необходимо. Имена будут одинаковыми. Компилятор будет их различать по передаваемым параметрам (как при перегрузке функций).

7. Если мы не передаем в конструктор параметры, он считается **конструктором по умолчанию**;

8. В классе может быть объявлен **только один деструктор**;

9. Деструктор срабатывает в тот момент, когда завершается работа программы и уничтожаются все данные.

# Классы. Пример 3. Класс КРУГ. Часть 1

```
#include <iostream>
using namespace std;
const double PI = 3.14159265358979238463;

class circle //определяем класс круг
{
	private:
		double x, y; //координаты  центра
		double r; //радиус

	public:
		circle()//это конструктор по умолчанию:
		{
			r = 1; x = 0; y = 0;
		}
```

# Классы. Пример 3. Класс КРУГ. Часть 2

```
public:     //это конструктор по умолчанию:
            circle()
            {
                  r = 1; x = 0; y = 0;
            }
            //это конструктор с параметрами:
            circle(int X, int Y,int R)
            {
                  r = R; x = X; y = Y;
            }

            void printCircle()
            {
                  cout << "r = " << r << endl;
                  cout << "x = " << x << endl;
                  cout << "y = " << y << endl << endl;
            }
```

# Классы. Пример 3. Класс КРУГ. Часть 3

```
// метод для ввода значений с клавиатуры

void setCircle() {

        cout << " r: "; cin >> r;
        cout << " x: "; cin >> x;
        cout << " y: "; cin >> y;
}

double square() { return PI * r*r; }

double length() { return 2 * PI*r; }
```

# Классы. Пример 3. Класс КРУГ. Часть 3

```
        ~circle() // это деструктор
        {
                cout << "Destructor" << endl;
        }
}; // описание класса закончилось

int main()
{
        circle A; // работает конструктор по умолчанию

        A.printCircle();

        cout <<" L = " <<A.length()<<endl;
```

# Классы. Пример 3. Класс КРУГ. Часть 4

```
        // работает конструктор с параметрами
        circle B(1,2,3);

        // вывод на экран
        B.printCircle();

       // ввод новых значений
        B.setCircle();

        B.printCircle();


return 0; // работает деструктор

}
```

# Списки инициализации членов класса

# Классы. Список инициализации членов класса. Часть 1

```
#include <string>
#include <iostream>
using namespace std;

class Employee
{
private:
	int id;
	string name;

public:
Employee(int id, string name) :id(id), name(name) {};
```

# Классы. Список инициализации членов класса. Часть 2

```
void printEmployee() {
cout << " id: " << id << " " << " name: " << name << endl;}

}; // описание класса закончено

int main()
{

        Employee a(7,"Tom");
        a.printEmployee();

return 0;
}
```

# Делегирующий конструктор

# Классы. Делегирующий конструктор

- ✓ Начиная с C++11, **конструкторам разрешено вызывать другие конструкторы**.
- ✓ Этот процесс называется **делегированием конструкторов** (или еще цепочкой конструкторов).
- ✓ Чтобы один конструктор вызывал другого, нужно просто сделать **вызов этого конструктора в списке инициализации членов**.

# Классы. Делегирующий конструктор

```
class Boo
{
private:

public:
        Boo() // конструктор по умолчанию
        {
                // часть кода X
        }

// используем конструктор по умолчанию Boo()
// для выполнения части кода X
        Boo(int value) : Boo()
        {
                // часть кода Y
        }
};
```

# Классы. Делегирующий конструктор. Часть 1

```
#include <string>
#include <iostream>
using namespace std;

class Employee
{
private:
        int id;
        string name;

public:
        Employee(int id = 0, const string &name = "") :
        id(id), name(name)
        {
                cout << "Employee " << name << " created.\n";
        }
```

# Классы. Делегирующий конструктор. Часть 2

```
// продолжение описания класса

// Используем делегирующие конструкторы
// для минимизации дублированного кода
Employee(const string &name) : Employee(0, name) { }

void printEmployee() {
cout << " id: " << id << " " << " name: " << name << endl;
}

~Employee()
{
        cout << " Destructor " <<endl;
}
}; // описание класса закончено
```

# Классы. Делегирующий конструктор. Часть 3

```
int main()
{

        // конструктор с параметрами по умолчанию id=0, name=""
        Employee a;
        a.printEmployee();

        // задали name= "Tom"
        Employee b("Tom");
        b.printEmployee();

        // id=7, name=“Mike"
        Employee c(7,"Mike");
        c.printEmployee();

return 0;
}
```

# Конструктор копирования

# Классы. Конструктор копирования. Пример 1. Часть 1

Копирование объекта

```
example_constructor
~example_destructor
~example_destructor
```

```
#include <iostream>
using namespace std;

class example {
public:
        example()
        {
                cout << " example_constructor"<<endl;
        }

        ~example() {
                cout << " ~example_destructor"<<endl;
        }
};
int main() {  // работает конструктор для ex1
                example ex1;
                example ex2 = ex1;
return 0; // работает деструктор для ex1 и ex2

}
```

# Классы. Конструктор копирования. Пример 1. Часть 2

- Конструктор в этом пример вызовется всего лишь однажды при создании объекта ex1.
- Деструктор вызовется для обоих объектов во время завершения работы программы.
- Если бы объекты были указателями и память выделялась динамически, то это привело бы к ошибке во время выполнения программы.
- Аналогичные ошибки могут происходить при передаче объекта в качестве аргумента в функцию и при возврате объекта из функции.

# Классы. Конструктор копирования. Пример 2. Часть 1

Передача объекта в функцию

```
...
class example {
public:
        example() {
                cout << "  example_constructor"<<endl;
        }

        ~example() {
                cout << "  ~example_destructor"<<endl;
        }
};

void function(example ex) { cout << "  function" << endl; }

int main() {
        // работает конструктор для ex1
        example ex1;
        // работает деструктор для ex1
        // и для побитовой копии ex1
        function(ex1);
return 0;}
```

```
example_constructor
function
~example_destructor
~example_destructor
```

# Классы. Конструктор копирования. Пример 2. Часть 2

- Конструктор вызывается лишь **однажды** при создании объекта, тогда как деструктор вызывается дважды — при удалении копии и при удалении самого объекта.
- При **передаче объекта в функцию по значению** создается его временная побитовая копия.
- Если исходный объект имеет поле с указателем, под который выделяется необходимый объем динамической памяти, тогда в его побитовой временной копии также окажется клон-указатель, указывающий на ту же область памяти.
- При выходе из функции копия объекта уничтожается с высвобождением участка памяти, на который указывает указатель.
- Но деструктору также приходится в завершение программы уничтожать и сам объект, который мы передаем в функцию, и второй раз высвобождать уже удаленную память.

# Классы. Конструктор копирования. Пример 3. Часть 1

Возвращение объекта из функции

```
...
class example {
public:
        example() {
                cout << "  example_constructor"<<endl;
        }
        ~example() {
                cout << "  ~example_destructor"<<endl;
        }};

example function() {
                example ex; // работает конструктор
                cout << "  function" << endl;
return ex; // создается временная копия
}

int main() {  // работает конструктор
                example ex;
                function();
return 0;}
```

```
example_constructor
example_constructor
function
~example_destructor
~example_destructor
~example_destructor
```

# Классы. Конструктор копирования. Пример 3. Часть 2

- В данном случае конструктор срабатывает дважды — во время создания объекта ex во время создания ex1.
- Деструкторов оказалось три, а не два.
- Так случилось потому, что в **функции для возврата формируется временная копия возвращаемого объекта**.
- Именно она должна была бы присвоиться объекту, который принимал бы возвращаемое из функции значение.
- И именно эту копию уничтожает второй по счету деструктор.
- Таким образом, мы вновь получаем возможность лишнего указателя на одну и ту же область выделенной памяти, что также приведет к необходимости высвобождения одной и той же памяти дважды.

# Классы. Конструктор копирования

- ✓ Это специальный конструктор, который позволяет получить **идентичный к заданному объект**.
- ✓ С помощью конструктора копирования можно получить **копию уже существующего объекта**.
- ✓ Конструктор копирования так же называется **инициализатором копии** (copy initializer).
- ✓ Конструктор копирования должен получать входным параметром **ссылку (&) на такой же класс**.

# Классы. Когда нужен конструктор копирования

- ✓ **создать копию объекта** класса
- ✓ **передать объект класса в некоторую функцию** как параметр-значение
- ✓ если нужно **возвратить объект класса по значению**

# Классы. Конструктор копирования

- ✓ Изначально в любом классе присутствуют неявный конструктор копирования, конструктор по умолчанию и деструктор.
- ✓ Однако ценность неявного конструктора копирования невелика по причине его «поверхностной» работы.
- ✓ Чтобы осуществить «глубокое копирование», такой конструктор определяют явно, обозначая параметры копирования.
- ✓ Синтаксис такого конструктора выглядит следующим образом:

```
имя_класса (const имя_класса &объект_класса)
{
...
}
```

# Классы. Конструктор копирования. Пример 4

```
...
class example {
public:
example() {cout << "  example_constructor"<<endl;}
~example(){cout << "  ~example_destructor"<<endl;}

example(const example &ex) {// конструктор копирования
        cout << "  copy_constructor" << endl;
}
};

example function() {
        example ex;
        cout << "  function" << endl;
return ex;
}
int main() {
                example ex1;
                function();
return 0;}
```

```
example_constructor
example_constructor
function
copy_constructor
~example_destructor
~example_destructor
~example_destructor
```

# Перегрузка операторов

# Классы. Перегрузка операторов. Замечания

- ✓ с помощью перегрузки невозможно создавать **новые обозначения для операций**;
- ✓ перегрузка операторов **не изменяет порядок выполнения операций и их приоритет**;
- ✓ унарный оператор не может использоваться для переопределения бинарной операции так же, как и бинарный оператор не переопределит унарную операцию

Перегрузка – это не более чем иной способ вызова функций.

Задача перегрузки операторов часто заключается в улучшении понимания кода, нежели в обеспечении выигрыша в каких-то вопросах.

# Классы. Перегрузка операторов. Ограничения

**Перегрузить можно практически любой оператор, за исключением:**

. точка (выбор элемента класса);

* звездочка (определение или разыменование указателя);

:: двойное двоеточие (область видимости метода);

?: знак вопроса с двоеточием (тернарный оператор сравнения);

# диез (символ препроцессора);

## двойной диез (символ препроцессора);

**sizeof** оператор нахождения размера объекта в байтах;

# Классы. Перегрузка операторов. Рекомендации

- ✓ Выполняйте перегрузку операторов только для имитации привычной записи. Для того чтобы сделать **код более удобочитаемым**.
- ✓ Если код становится сложнее по структуре или читабельности, следует **отказаться от перегрузки операторов** и использовать функции.
- ✓ Для больших операндов с целью экономии места используйте для **передачи аргументы с типом константных ссылок**.
- ✓ Оптимизируйте возвращаемые значения.
- ✓ Не трогайте операцию копирования, если она подходит для вашего класса.

# Классы. Перегрузка операторов. Рекомендации

- ✓ Если копирование по умолчанию не подходит, меняйте или явно запрещайте возможность копирования.
- ✓ Следует предпочитать функции-члены над функциями-нечленами в случаях, когда функции требуется доступ к представлению класса.
- ✓ Указывайте пространство имен и обозначайте связь функций с их классом.
- ✓ Используйте функции-нечлены для симметричных операторов.
- ✓ Используйте оператор () для индексов в многомерных массивах.
- ✓ С осторожностью используйте неявные преобразования.

# Классы. Перегрузка операторов. Пример 1

```
#include <iostream>
using namespace std;
class point { private: int x, y;
        public:
        point(int xx=0, int yy=0):x(xx), y(yy) {};

        void printXY() {
        cout << " x = " << x << " y = " << y << endl; };

point operator+ (point p) {
        point pp(p.x + this->x, p.y + this->y);
return pp;}
};
int main() {
        point p1(1, 2); point p2(3, 4);
        point p3 = p1 + p2;
        p3.printXY();
return 0;}
```

# Классы. Перегрузка операторов. Пример 2

```
. . .
class complex {
private:
        double re, im;
public:
        complex(double r = 0, double i = 0) : re(r), im(i) {}
        complex operator+(complex &other);
        void Display() { cout << re << ", " << im << endl; }
};

complex complex::operator+(complex &other) {
return complex(re + other.re, im + other.im);}

int main() {
        complex a(1.2, 3.4);
        complex b(5.6, 7.8);
        complex c=a+b;
c.Display();}
```

# Классы. Перегрузка операторов. Пример 3

. . .

```
class
```

# Дружественные функции

# Классы. Дружественные функции

- ✓ **Дружественная функция** — это функция, которая не является членом класса, но имеет доступ к членам класса, объявленным в полях **private** или **protected**.
- ✓ Для определения дружественных функций используется ключевое слово **friend.**

# Классы. Пример 1. Дружественные функции. Часть 1

```
#include <iostream>
using namespace std;

class Anything
{
private:
	int value;
public:
	Anything() { value = 0; }
	void add(int n) { value += n; }

// Делаем функцию reset() дружественной классу Anything
	friend void reset(Anything &anything);

	void show() { cout << value << endl; }
}; //описание класса закончилось
```

# Классы. Пример 1. Дружественные функции. Часть 2

```
// reset() теперь является другом класса Anything
void reset(Anything &anything)
{
// И мы имеем доступ к закрытым членам объектов
// класса Anything
	anything.value = 0;
}

int main()
{
	Anything one;
	one.add(4); // добавляем 4 к value
	one.show();
	reset(one); // сбрасываем value к 0
	one.show();
return 0;
}
```

# Классы. Пример 2. Дружественные функции. Часть 1

```
#include <iostream>
#include <string>
using namespace std;
// Начало описания класса
class Auto
{
	// Объявление дружественных функций
	friend void drive(Auto &);
	friend void setPrice(Auto &, int price);

	public:
	// Конструктор
	Auto(string autoName, int autoPrice)
	{
		name = autoName;
		price = autoPrice;
	}
```

# Классы. Пример 2. Дружественные функции. Часть 2

```
// Продолжение описания класса

        string getName() { return name; }
        int getPrice() { return price; }

// приватные свойства класса
        private:
                string name;
                int price;
};
//Конец описания класса
```

# Классы. Пример 2. Дружественные функции. Часть 3

```
// Реализация дружественных функций вне класса
// (их объявление было внутри класса)

void drive(Auto &a)
{       // функция вождения автомобиля
        cout << a.name << " is driven" << endl;
}

void setPrice(Auto &a, int price)
{// функция назначения цены
        if (price > 0)
        a.price = price;
}
```

# Классы. Пример 2. Дружественные функции. Часть 4

```
int main()
{
// tesla - экземпляр класса Auto
// Работает конструктор инициализации
Auto tesla("Tesla", 5000);

// drive - дружественная функция
// может обращаться к полям класса
drive(tesla);
cout << tesla.getName() << " : " << tesla.getPrice() << endl;

// setPrice - дружественная функция
// устанавливает новую цену авто
setPrice(tesla, 8000);
cout << tesla.getName() << " : " << tesla.getPrice() << endl;

return 0;
}
```

# Классы. Дружественные функции

- ✓ Когда мы объявляем дружественные функции, то фактически мы говорим компилятору, что это друзья класса и они имеют **доступ ко всем членам этого класса**, в том числе закрытым.

- ✓ При этом для дружественных функций **не важно, определяются они под спецификатором public или private**. Для них это не имеет значения.

- ✓ Дружественные функции **могут определяться в другом классе**

# Дружественные функции в другом классе

# Классы. Дружественные функции в другом классе. Часть 1

```
#include <iostream>
#include <string>
using namespace std;
class Auto;

class Person
{
public:
	Person(string s){name = s;}

// объявление функций дружественного класса
	void drive(Auto &a); // передача по ссылке
	void setPrice(Auto &a, int price);

private:
	string name;
};
```

# Классы. Дружественные функции в другом классе. Часть 2

```
class Auto
{
friend void Person::drive(Auto &);
friend void Person::setPrice(Auto &, int price);

public:
        // конструктор, устанавливающий значения
        Auto(string autoName, int autoPrice){
        name = autoName;
        price = autoPrice;}

        string getName() { return name; }
        int getPrice() { return price; }

private:
        string name;
        int price;
}; // описание класса закончилось
```

# Классы. Дружественные функции в другом классе. Часть 3

```
// дружественные функции для класса Person

void Person::drive(Auto &a)
{
	cout << name << " drives " << a.name << endl;
}


void Person::setPrice(Auto &a, int price)
{
	if (price > 0)
	a.price = price;
}
```

# Классы. Дружественные функции в другом классе. Часть 4

```
int main()
{       // создаем объект класса Auto
        // с именем Tesla и стоимостью 5000
        Auto tesla("Tesla", 5000);

        // создаем объект класса Person
        // с именем Tom
        Person tom("Tom");

// Работают дружественные функции класса Person
        tom.drive(tesla);
        tom.setPrice(tesla, 8000);

cout << tesla.getName() << " : " << tesla.getPrice() <<
endl;

return 0;}
```

```
Tom drives Tesla
Tesla : 8000
```

# Дружественные классы

# Классы. Дружественные классы

- ✓ Если класс содержит 5 — 10 методов и каждому из них необходим доступ к приватным элементам другого класса, то удобней не объявлять дружественные функции в теле другого класса, а объявить вместо них **дружественный класс**.

- ✓ Тогда методы этого дружественного класса, автоматически станут дружественными классу, который предоставляет дружбу.

# Классы. Пример. Дружественные классы. Часть 1

```
#include <iostream>
#include <string>
using namespace std;
class Auto; // заранее указать дружественный класс

class Person
{
	public:
		Person(string s){name = s;}

	// используем функции дружественного класса
	void drive(Auto &a);
	void setPrice(Auto &a, int price);

	private:
	string name; // приватное поле
};
```

# Классы. Пример. Дружественные классы. Часть 2

```
class Auto // описание класса Auto
{
friend class Person; // класс Person – дружественный к Auto
public:
	Auto(string autoName, int autoPrice)
	{
		name = autoName;
		price = autoPrice;
	}

	string getName() { return name; }
	int getPrice() { return price; }

private:
	string name;
	int price;
};
```

# Классы. Пример. Дружественные классы. Часть 3

```
// класс Person – дружественный к Auto
// используем функции
void Person::drive(Auto &a)
{
std::cout << name << " drives " << a.name << endl;
}

void Person::setPrice(Auto &a, int price)
{
	if (price > 0)
	a.price = price;
}
```

# Классы. Пример. Дружественные классы. Часть 4

```
int main()
{       // создаем объект класса Auto
        // с именем Tesla и стоимостью 5000
        Auto tesla("Tesla", 5000);

        // создаем объект класса Person
        // с именем Tom
        Person tom("Tom");

// Работают дружественные функции класса drive и setPrice
// класс Person использует класс Auto
        tom.drive(tesla);
        tom.setPrice(tesla, 8000);

cout << tesla.getName() << " : " << tesla.getPrice() <<
endl;

return 0;}
```

```
Tom drives Tesla
Tesla : 8000
```

# Спасибо за внимание!