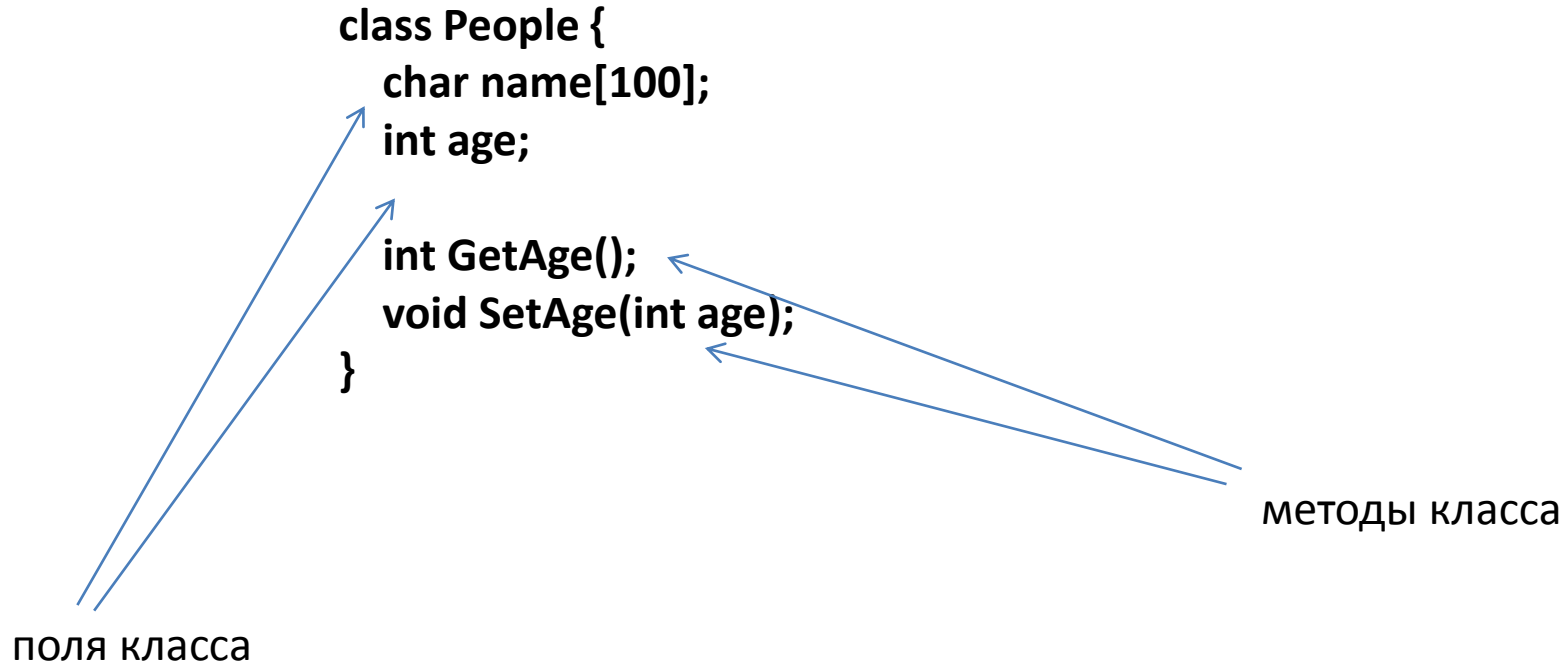


Для создания класса в языке C++ используется следующая синтаксическая конструкция:



Создание экземпляров класса:

```
People people;
```

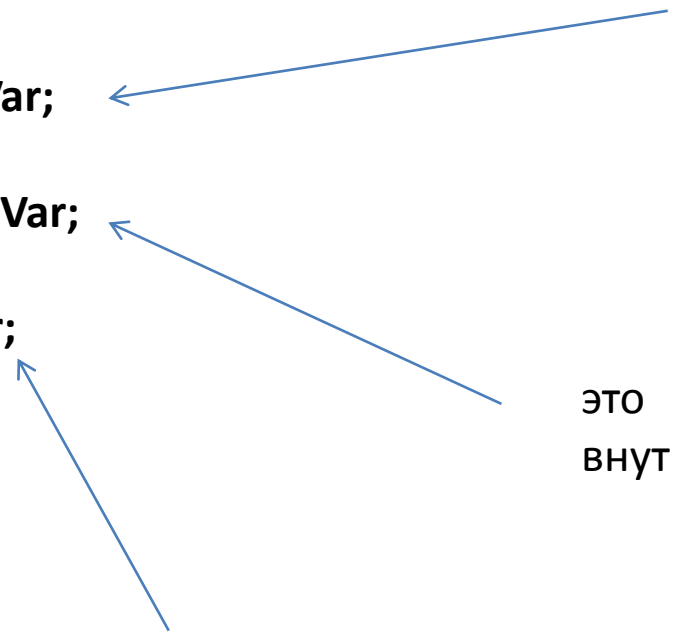
```
People *p = new People;
```

```
People all[20];
```

К полям и методам класса можно настроить свои уровни доступа, это является реализацией принципа инкапсуляции :

```
class Test {  
  public:  
    int openVar;  
  private:  
    int closedVar;  
  protected:  
    int inhVar;  
}
```

данное поле доступно везде



The diagram consists of three blue arrows pointing from text annotations to fields in the class definition. The first arrow points from 'данное поле доступно везде' to 'int openVar;'. The second arrow points from 'это поле доступно только внутри класса' to 'int closedVar;'. The third arrow points from 'к этому полю можно обратиться внутри класса и в его классах наследниках' to 'int inhVar;'.

это поле доступно только
внутри класса

к этому полю можно
обратиться внутри класса и в
его классах наследниках

```
class People {  
    public:  
        int age;
```

public - данное поле доступно везде

```
public:  
    void Show(void) {  
        cout << age;  
    }  
}
```

РАЗРЕШЕНО

```
class Worker : public People {  
    public:  
        void ShowInfo(void) {  
            cout << age;  
        }  
}
```

РАЗРЕШЕНО

```
People p;  
p.Age = 25;
```

РАЗРЕШЕНО

```
class People {  
protected:  
    int age;
```

```
public:  
    void Show(void) {  
        cout << age;  
    }  
}
```

```
class Worker : public People {  
public:  
    void ShowInfo(void) {  
        cout << age;  
    }  
}
```

```
People p;  
p.Age = 25;
```

protected

к этому полю можно обратиться внутри
класса и в его классах наследниках

РАЗРЕШЕНО

РАЗРЕШЕНО

ЗАПРЕЩЕНО

```
class People {  
    private:  
        int age;
```

```
public:  
    void Show(void) {  
        cout << age;  
    }  
}
```

```
class Worker : public People {  
    public:  
        void ShowInfo(void) {  
            cout << age;  
        }  
}
```

```
People p;  
p.Age = 25;
```

private

это поле доступно только внутри класса

РАЗРЕШЕНО

ЗАПРЕЩЕНО

ЗАПРЕЩЕНО

```
class People {  
public:  
    int age;  
}
```

```
void Show(People& value) {  
    cout << value.age;  
}
```

```
People p;  
Show(p);
```

РАЗРЕШЕНО

```
class People {
```

```
private:
```

```
    int age;
```

```
}
```

```
void Show(People& value) {
```

```
    cout << value.age;
```

```
}
```

ЗАПРЕЩЕНО

```
People p;
```

```
Show(p);
```

```
class People {  
private:  
    int age;
```

```
    friend void Show(People& value);  
}
```

```
void Show(People& value) {  
    cout << value.age;  
}
```

```
People p;  
Show(p);
```

friend функция может
получить доступ к **protected**
и **private** полям класса

РАЗРЕШЕНО

Для наследования классов в языке C++ применяется следующая конструкция:

```
class People {  
public:  
    char name[100];  
    int age;  
}
```

```
class Student : public People {  
public:  
    char group[32];  
}
```

В этом случае класс **Student** получит всю функциональность класса **People** и дополнит ее своей функциональностью.

Модификатор доступа **public** в строке наследования влияет на права доступа к полям базового класса из класса наследника. В данном случае они не меняются.

Одной из реализаций полиморфизма являются виртуальные функции. Они позволяют вызвать функцию в том классе, каким объект является на самом деле.

Пусть есть следующие классы:

```
class Parent {  
public:  
    void nonvirt_func()    {cout << 'non virtual in parent';};  
    virtual void virt_func() {cout << 'virtual in parent';};  
}
```

```
class Child : public Parent {  
public:  
    void nonvirt_func() {cout << 'non virtual in child';};  
    void virt_func()    {cout << 'virtual in child';};  
}
```

Parent *parent = new Parent();

parent->nonvirt_func(); //на экране появится '**non virtual in parent**' – мы
//сделали вызов в экземпляре класса **Parent**

parent->virt_func(); //на экране появится '**virtual in parent**' – мы
//сделали вызов в экземпляре класса **Parent**

Child *child = new Child();

child->nonvirt_func(); //на экране появится '**non virtual in child**' – мы
//сделали вызов в экземпляре класса **Child**

child->virt_func(); //на экране появится '**virtual in child**' – мы
//сделали вызов в экземпляре класса **Child**

// Любого наследника мы можем привести к классу предка

Parent *parent = new Child();

parent->nonvirt_func(); //на экране появится **'non virtual in parent'**

// не виртуальная функция вызовется в том классе,

// **экземпляром которого наш объект объявлен**

parent->virt_func(); //на экране появится **'virtual in child'**

// виртуальная функция вызовется в том классе,

// **кем наш объект является на самом деле**