

# Введение в ООП

классы и наследование

# ООП

- Определяются сущности, связанные с задачей
  - сущности - объекты
- Функционал системы рассматривается как действия и взаимодействия объектов
- Действия над объектами осуществляются через их интерфейсы
- Детали реализации, не существенные для действий, скрываются

# Объект - Структура Вектор

```
struct Vector3  
{  
    double x, y, z;  
}
```

//объявления функций оперирующих данными структуры

```
double length(const Vector3& v);  
void scale(Vector3& v, double s);  
void add(Vector3& v1, const Vector3& v2);
```

//определение функции

```
double length(const Vector3& v)  
{  
    return sqrt(v.x*v.x + v.y*v.y + v.z*v.z);  
}
```

# В C++ структура может быть с функциями

```
//vector3.h
struct Vector3
{ //члены - данные
    double x, y, z;
    //объявление функций членов
    double length()
    {
        return sqrt(x*x + y*y + z*z);
    }
    void scale(double s);
    void add(const Vector3& v);
}
```

```
//vector3.cpp
void Vector3::scale(double s)
{
    x = x*s;
    y = y*s;
    z = z*s;
}

void Vector3::add(const Vector3& v)
{
    x += v.x;
    y += v.y;
    z += v.z;
}
```

# Обращение к функциям-членам и данным структур

```
Vector3 V;  
V.x = 10.0; V.y=2.0; V.z=5.0; //оператор доступа .  
V.scale(0.4);  
Vector3* ptrV = &V;  
cout << ptrV->length(); //оператор доступа ->  
ptrV = new Vector3;  
ptrV->add(V);  
V = *ptrV;  
ptrV->x=1.0  
  
delete ptrV;
```

# Контроль доступа к функциям и данным в C++

- Три уровня доступа
  - **private** - могут быть доступны только из функций членов
  - **public** - могут быть доступны в любом месте программы
  - **protected** - могут быть доступны в функциях членах и при наследовании
- Структуры по умолчанию имеют доступ public к своим членам
- Регулирование доступа требуется для сокрытия и защиты данных (принцип наименьших привилегий)
- Как правило данные имеют доступ private (protected), а функции - public.
- public-функции определяют интерфейс работы с объектами класса

# Классы в C++

- Класс(class) - аналогичен структуре(struct), но все члены по умолчанию имеют уровень доступа private
- В программах C++ чаще используются классы
- классы в ROS

# Класс Vector3

```
class Vector3
{
private: //секция приватных данных класса
    double x, y, z;
public: //секция публичных данных класса
    //объявление функций класса
    double length();
    void scale(double s);
    void add(const Vector3& v);
}
```



# Влияние уровня доступа

```
Vector3 V;
```

```
V.x = 10.0; V.y=2.0; V.y=5.0; ⇒ ошибка компиляции - нарушение уровня  
доступа к private данным
```

```
V.scale(0.4);
```

```
Vector3* ptrV = &V;
```

```
cout<<ptrV->length();
```

```
cout<<ptrV->x; ⇒ ошибка компиляции - нарушение уровня доступа к private  
данным
```

```
ptrV = new Vector3;
```

```
ptrV->add(V);
```

```
V = *ptrV;
```

```
ptrV->x=1.0 ⇒ ошибка компиляции - нарушение уровня доступа к private  
данным
```

```
delete ptrV;
```

# Зачем скрывать данные

- Модульность программного кода
- Соккрытие внутренней реализации
- Возможность изменения реализации
- Должны ли все данные быть скрытыми ?
  - не обязательно, но это снижает инкапсуляцию - влияет на качество и модульность кода

# Как читать/изменять private данные класса?

Функции доступа (getters & setters)

```
class Vector3
```

```
{  
private:  
    double x, y, z;  
public:  
    double getX(){return x;}  
    void setX(double x){ this->x = x; }  
    void setXYZ(double x, double y, double z)  
    {  
        this->x = x;  
        this->y = y;  
        this->z = z;  
    }  
}
```

# Как читать/изменять private данные класса?

Изменение внутреннего представления

```
class Vector3
```

```
{  
private:  
    double ro, fi, z;  
public:  
    double getX(){return ro*cos(fi);}  
    void setXYZ(double x, double y, double z)  
    {  
        this->ro = sqrt(x*x + y*y);  
        this->fi = atan2(y, x);  
        this->z = z;  
    }  
}
```

# Конструктор класса (структуры)

- Специальная функция, которая вызывается автоматически при создании объекта класса
- Функция имеет имя класса и не имеет возвращаемого значения
- Обычный способ для инициализации данных объекта класса
- Может быть несколько, различающихся по передаваемым параметрам, конструкторов
- Конструктор по умолчанию
- Конструктор простых переменных
- Динамическое создание объекта с помощью оператора new

# Деструктор класса (структуры)

- Специальная функция-член класса, которая вызывается при уничтожении класса
- Выполняет специальные функции освобождения данных
- Имеет имя равное имени класса с символом ~
- Не имеет параметров и возвращаемого значения
- Класс может иметь только один деструктор
- Деструктор по умолчанию
- Деструктор вызывается, когда объект, созданный на стеке выходит из области видимости
- Деструктор вызывается, когда объект созданный с помощью new уничтожается с помощью delete

```
class Vector3
```

```
{
```

```
private:
```

```
double x, y, z;
```

```
char* name;
```

```
public:
```

```
Vector3(double x, double y, double z)
```

```
{
```

```
    this->x = x;
```

```
    this->y = y;
```

```
    this->z = z;
```

```
    mem = new char[100];
```

```
}
```

```
Vector3():
```

```
    x(0.0), y(0.0), z(0.0), mem(new char[100])
```

```
{
```

```
}
```

```
~Vector3()
```

```
{
```

```
    delete mem[];
```

```
}
```

```
}
```

список  
инициализации  
членов класса

```
void func()
```

```
{
```

```
    Vector3 a (1.0, 2.0, 3.0); ⇒ конструктор для a
```

```
{
```

```
    Vector3 b; ⇒ конструктор для b
```

```
    a = b;
```

```
}
```

```
⇒ деструктор для b
```

```
Vector3 *p;
```

```
p = new Vector3 (1.0, 1.0, 1.0); ⇒ конструктор для *p
```

```
...
```

```
delete p; ⇒ деструктор для *p
```

```
}
```

```
⇒ деструктор для a
```

```
class Vector3
```

```
{
```

```
private:
```

```
double x = 0;
```

```
double y = 0;
```

```
double z = 0;
```

```
char* name = new char[100];
```

```
public:
```

```
~Vector3()
```

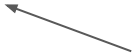
```
{
```

```
delete mem[];
```

```
}
```

```
}
```

Инициализация при  
объявлении



```
void func()
```

```
{
```

```
Vector3 a (1.0, 2.0, 3.0); ⇒ конструктор для a
```

```
{
```

```
Vector3 b; ⇒ конструктор для b
```

```
a = b;
```

```
}
```

```
⇒ деструктор для b
```

```
Vector3 *p;
```

```
p = new Vector3 (1.0, 1.0, 1.0); ⇒ конструктор для *p
```

```
...
```

```
delete p; ⇒ деструктор для *p
```

```
}
```

```
⇒ деструктор для a
```



# Включение в класс членов-объектов

```
class Triangle
```

```
{
```

```
private:
```

```
    Vector3 A, B, C;
```

```
public:
```

```
    Triangle(double x1, double y1, double z1,
```

```
            double x2, double y2, double z2,
```

```
            double x3, double y3, double z3):
```

```
        A(x1, y1, z1), // вызов конструкторов для членов объектов
```

```
        B(x2, y2, z2),
```

```
        C(x3, y3, z3)
```

```
    {}
```

```
}
```

# Включение в класс членов-объектов

```
class Triangle
{
    private:
        Vector3 A, B, C;
    public:
        Triangle() {}
}
```

----->

ошибка если у  
Vector3 нет  
тривиального  
конструктора

# copy constructor

```
class Vector3
{
private:
    double x, y, z;
public:
    Vector3(){ x = y = z = 0;}
    Vector3(double x, double y, double z):x(x),y(y),z(z){ }
    Vector3(const Vector3& left)
    {
        this->x = left->x; this->y = left->y; this->z = left->z;
    }
};
```

# copy constructor

```
class MyString
```

```
{
```

```
private:
```

```
    char* str;
```

```
    int length
```

```
public:
```

```
    const char* getStr() { return str; }
```

```
    MyString(const char* str){ length = strlen(str); this->str = new char[length]; }
```

```
    ~MyString(){ delete[] str; }
```

```
};
```

```
MyString* S = new MyString("Petya");
```

```
MyString S2 = S;
```

```
delete S;
```

```
cout << S2.getStr() << "\n";
```

# copy constructor

```
class MyString
```

```
{
```

```
private:
```

```
    char* str;
```

```
    int length
```

```
public:
```

```
    MyString(const char* str){ length = strlen(str); this->str = new char[length]; }
```

```
    MyString(const MyString& left)
```

```
{
```

```
    length = left.length;
```

```
    str = new char[length];
```

```
    for(int i = 0; i < length; i++)
```

```
        str[i] = left.str[i];
```

```
}
```

```
};
```

# явный (explicit) constructor

```
class MyString
{
private:
    char* str = std::null_ptr;
    int length
public:
    void setStr(const char* x){
        if(str) delete str; length = strlen(str); this->str = new char[length]; }
    MyString(const char* str){ setStr(str);}
};
```

**MyString S = "Petya";** - неявное создание объекта

# явный (explicit) constructor

```
class MyString
{
private:
    char* str = std::null_ptr;
    int length
public:
    void setStr(const char* x){
        if(str) delete str; length = strlen(str); this->str = new char[length]; }
    explicit MyString(const char* str){ setStr(str);}
};
```

**MyString S = "Petya"; - неявное создание объекта - ошибка компиляции**

# Наследование (расширение и изменение базового класса)

**enum**

```
{  
    FIGURE,  
    CIRCLE,  
    POLY,  
    ANOTHER_FIGURE  
}
```

**class Figure**

```
{  
    protected:  
        Vector3 Center;  
        int figure;  
    public:  
        int type(){ return figure; }  
        double getBigRadius(){ return 0; }  
        double getSquare() { return 0.0; }  
        double getPerimeter() { return 0.0; }  
        void draw() { }  
        Figure(const Vector3& p. int t):  
            Center(p), figure(t){}  
}
```



# Наследование (расширение и изменение базового класса)

```
class Circle: public Figure
{
protected:
    //дополнительные данные
    double R;
public:
    //дополнительные функции
    void setRadius() ;
    //переопределяемые функции Figure
    double getBigRadius(){ return R; }
    double getSquare() { return 3.14 * R*R; }
    double getPerimeter() { return 2*3.14*R; }
    void draw() ;
    Circle(const Vector& p, rad):
        Figure(p, Circle),R(rad)
    {}
};
```

```
class Poly: public Figure
{
protected:
    Vector3* vertices;
    unsigned int size;
public:
    double getBigRadius();
    double getSquare();
    double getPerimeter() ;
    void draw() ;
    Poly(const Vector& p[], int size);
};
```

# Наследование

- Повторное использование кода
- Возможность изменить поведение функций (переопределение)

# Возможность приведения указателей к базовому типу

```
Circle C(point, rad);
```

```
Figure* fig = &C;
```

```
Figure* fig2 = new Poly(points, 10); //приведение указателя к указателю на базовый тип
```

```
Poly* po = fig2; //ошибка компиляции -- приведение возможно только в сторону предка
```

```
//если мы знаем что fig2 - это Poly - можем осуществить принудительное приведение
```

```
Poly* po = static_cast<Poly*>(fig2);
```

```
double p = fig->getPerimeter(); //Figure::Perimeter - вызов функции класса предка
```

```
double p2 = po->getPerimeter(); //Poly::Perimeter - вызов функции класса Poly
```

```
delete fig2; // деструктор базового класса Figure
```

# Обработка однотипных объектов

```
Figure* fig_arr[10];  
fig_arr[0] = new Circle(...);  
fig_arr[1] = new Poly(...);  
...  
fig_arr[9] = new Circle(...);
```

```
for( int i = 0; i < 10; i++)  
{  
    switch (fig_arr[i]->type())  
    {  
        case CIRCLE:  
            Circle* c = static_cast<Circle*>(fig_arr[i]);  
            c->draw();  
            break;  
        case POLY:  
            Poly* p = static_cast<Poly*>(fig_arr[i]);  
            p->draw();  
            break;  
        case ANOTHER_FIGURE:  
            ....  
    }  
}
```

# Виртуальные функции - полиморфизм

```
class Figure
```

```
{
```

```
protected:
```

```
    Vector3 Center;
```

```
public:
```

```
    virtual double getBigRadius(){ return 0; }
```

```
    virtual double getSquare() { return 0.0; }
```

```
    virtual double getPerimeter() { return 0.0; }
```

```
    virtual void draw() { }
```

```
    Figure(const Vector3& p):
```

```
        Center(p){}
```

```
    virtual ~Figure(){}  
}
```

# Виртуальные функции - полиморфизм

```
class Circle: public Figure
{
protected:
    double R;
public:
    double getBigRadius(){ return R; }
    double getSquare() { return 3.14 * R*R; }
    double getPerimeter() { return 2*3.14*R;}
    void draw() ;
    Circle(const Vector& p, rad):
        Figure(p),R(rad)
    {}
}
```

```
class Poly: public Figure
{
protected:
    Vector3* vertices;
    unsigned int size;
public:
    double getBigRadius();
    double getSquare();
    double getPerimeter() ;
    void draw() ;
    Poly(const Vector& p[], int size);
    ~Poly();
}
```

# Возможность использования указателей на базовый класс

```
Circle C(point, rad);
```

```
Figure* fig = &C;
```

```
Figure* fig2 = new Poly(points, 10);
```

```
double p = fig->getPerimeter(); //Circle::Perimeter - вызов функции класса Circle
```

```
double p2 = fig2->getPerimeter(); //Poly::Perimeter - вызов функции класса Poly
```

Если функция объекта полиморфная (virtual), то при вызове такой функции по указателю на базовый класс, будет вызвана функция того класса наследника, адрес объекта которого в данный момент хранит указатель..

# Обработка однотипных объектов

```
Figure* fig_arr[10];  
fig_arr[0] = new Circle(...);  
fig_arr[1] = new Poly(...);  
...  
fig_arr[9] = new Circle(...);  
  
for( int i = 0; i < 10; i++)  
{  
    fig_arr[i]->draw(); //вызываются функции отрисовки соответствующих классов  
}  
  
for( int i = 0; i < 10; i++)  
{  
    delete fig_arr[i]; //вызываются деструкторы соответствующих классов  
}
```



# Таблица виртуальных методов

**class Figure**

{

**protected:**

Vector3 Center;

**public:**

**virtual double getBigRadius(){ return 0; }**

**virtual double getSquare() { return 0.0; }**

**virtual double getPerimeter() { return 0.0; }**

Figure(**const** Vector3& p):

Center(p){}

virtual ~Figure(){}

}



vtable - скрытый член базового класса

getBigRadius

Figure::getBigRadius

getSquare

Figure::getBigSquare

getPerimeter

Figure::getPerimeter

~

Figure::~~Figure

# Виртуальные функции - полиморфизм

```
class Circle: public Figure
```

```
{
```

```
protected:
```

```
    double R;
```

```
public:
```

```
    double getBigRadius(){ return R; }
```

```
    double getSquare() { return 3.14 * R*R; }
```

```
    double getPerimeter() { return 2*3.14*R; }
```

```
    Circle(const Vector& p, rad):
```

```
        Figure(p),R(rad)
```

```
    {
```

```
}
```



vtable

getBigRadius	Circle::getBigRadius
getSquare	Circle::getBigSquare
getPerimeter	Circle::getPerimeter
~	Circle::~~Circle

# Профит

- Задание с помощью базового класса интерфейса
- Возможность обработки массивов объектов наследников базового класса не зная их конкретного класса
- Возможность построения обобщенных алгоритмов, использующих интерфейсные функции базового класса, и изменения их конкретных действий за счет изменений функций в наследниках

# Абстрактный класс - интерфейс

```
class Figure
{
public:
    virtual double getBigRadius() = 0; //абстрактные виртуальные функции
    virtual double getSquare() = 0;
    virtual double getPerimeter() = 0;
    virtual void draw() = 0;
    virtual ~Figure(){} //деструктор должен быть виртуальным, но не может быть абстрактным
}
```