Лекция 2. Классы объектов

**Класс** - пользовательский тип данных, объединяющий данные и функции для отображения моделей реального мира.

Данные класса называют **полями**, функции класса - **методами**. Поля и методы называют **элементами класса**.

class имя\_класса

{

private://По умолчанию, может отсутствовать

//Описание скрытых элементов

public:

//Описание общедоступных элементов

protected:

//Список средств, доступных при наследовании

};

Спецификаторы доступа определяют область видимости элементов.

**private (локальный)**//Элементы, расположенные в этой области, доступны только внутри класса. Получить значение или изменить эти элементы можно только через обращение к соответствующим методам класса.

**protected (защищённый)**//Разрешён доступ внутри текущего и производных классов.

**public (глобальный)**//Разрешён доступ из любых функций.

Поля класса:

1) могут иметь любой тип, кроме типа своего класса; разрешены ссылки на этот класс - указатели.

2) могут быть описаны с модификатором const; при этом они инициализируются 1 раз с помощью конструктора.

3) могут использовать модификатор static.

Инициализация полей при описании не допускается.

Пример. Реализация простейшего класса.h

#include <iostream.h>

class Cls

{

private:

int x;

public:

setd (int a)

{

x=a;

}

shows()

{

cout<<x<<endl;

}

};

int main()

{

Cls d, s;

d.setd (1234);

d.shows();

s.setd(543);

s.shows();

return 0;

}

2 способ (класс с невстроенными методами)

class Cls

{

private:

int x;

public:

void setd (int);

void shows();

};

void Cls::setd (int a)

{

x=a;

}

void Cls::shows ()

{

cout<<x<<endl;

}

…

**::** - знак операции глобального разрешения, который устанавливает взаимосвязь функций и класса, к которому относится эта функция.

**Конструктор и деструктор класса**

**Конструктор** – это специальный метод класса, который предназначен для инициализации элементов класса некоторыми начальными значениями.

Имя конструктора должно совпадать с именем класса. Конструктор отличается от других функций следующим:

1) Конструктор не имеет возвращаемого значения, так как автоматически вызывается системой и не имеет вызывающей функции, которой он мог бы вернуть значение.

2) Класс может иметь несколько конструкторов, имеющих различный набор параметров.

3) Параметры конструктора могут иметь любой тип, а также 1 из конструкторов может содержать параметры по умолчанию.

4) Конструктор без параметров называется **конструктором по умолчанию**.

5) При отсутствии конструктора он создаётся компилятором автоматически. В случае наличия неинициализированных констант и ссылок будет сгенерирована ошибка.

6) Конструкторы не наследуются.

7) Конструктор не возвращает сообщение об ошибке инициализации (следует использовать механизмы обработки исключительных ситуаций).

Пример. Класс с конструктором.

#include <iostream.h>

class Cls

{

private:

int x;

public:

Cls():x(0)

{

}

void count()

{

x++;

}

void shows()

{

cout<<x<<endl;

}

};

int main()

{

Cls d;

d.count ();

d.shows();//Выводится 1

d.count ();

d.shows();//Выводится 2

return 0;

}

Инициализация полей в конструкторе располагается между заголовком и телом конструктора и отделяется от заголовка двоеточием. Инициализирующее значение помещается в скобки после имени инициализируемой переменной. При инициализации нескольких полей они отделяются друг от друга запятыми. Инициализация полей в конструкторе с помощью списка инициализации происходит до начала выполнения тела конструктора. Список инициализации конструктора используется для задания начальных значений констант и ссылок.

Пример.

private:

сonst b;

public:

Cls():b(111)

{//b=111; - ошибка!

}

Пример. Использование классов с несколькими конструкторами.

#include <iostream.h>

class Cls

{

private:

int x;

public:

Cls():x(1)

{

}

Cls(int w):x(w)

{

}

void shows()

{

cout<<x<<endl;

}

};

int main()

{

Cls d, g(25), f, s(44);

d.shows();//Выводится 1

g.shows ();//Выводится 25

f=s;

f.shows();//Выводится 44

return 0;

}

**Деструктор** - особый метод класса, который предназначен для освобождения памяти, занимаемой объектом, и для других действий в момент уничтожения класса. Имя деструктора начинается с тильды (~), и далее следует имя класса.

Отличия деструктора от других методов:

1) Не имеет параметров и возвращаемого значения.

2) При описании деструктора нельзя использовать модификаторы const и static.

3) Деструкторы не наследуются.

4) Нельзя получить указатель на деструктор.

5) Деструктор может быть виртуальным.

Деструктор вызывается автоматически в следующих случаях:

1) Для локальных объектов при выходе из области видимости.

2) Для глобальных объектов при завершении программы.

3) Для объектов, заданных через указатели при выполнении операции delete.

Если деструктор не описан в классе, то он создаётся автоматически. Явным образом деструктор следует описывать в случае необходимости освобождения динамически выделенной памяти.

Пример. Использование деструктора.

class Cls

{

private:

int \*a;

public:

Cls()

{

a= new int[10];

}

void add (int n)

{

for (int i=0; i<n; i++)

{

cout<<"Vvedite a["<<i<<"]:";

cin>>a[i];

}

}

void shows (int n)

{

for (int i=0; i<n; i++) cout<<a[i]<<" ";

cout<<endl;

}

~Cls()

{

delete []a;

cout<<"Destruct"<<endl;

}

};

int main()

{

Cls mas;

mas.add (5);

mas.shows (5)

return 0;

}

**Конструкторы вложенных классов**

Если объект является объектом другого класса, необходимо обеспечить правильную работу конструктора объекта включённого класса. Для этого в определении конструктора включающего класса после символа : записываются элементы включаемого конструктора с указанием его параметров (если есть). Конструктор включаемого класса будет вызван перед конструктором включающего класса.

Пример. Класс с вложенным конструктором

#include <iostream>

using namespace std;

class Cls01

{

int \*a;

double \*b;

public:

Cls01 (int p1, double p2)

{

a=new int (p1);

b=new double (p2);

}

~Cls01()

{

delete a; cout<<"Delete a"<<endl;

delete b; cout<<"Delete b"<<endl;

}

void out\_ab()

{

cout<<"a="<<\*a<<"\*b="<<\*b<<endl;

}

};

class Cls02

{

Cls01 x,y;

public:

Cls02 (int, double, int, double);

void out\_xy()

{

x.out\_ab();

y.out\_ab();

}

};

Cls02::Cls02 (int l1, double l2, int l3, double l4): x (l1, l2), y (l3,l4)

{

}

void main()

{

Cls02 s(10, 1.2, 20, 2.5);

s.out\_xy();

}

**Работа с объектами класса**

Объекты могут использоваться в качестве аргументов классов. Метод может возвращать объект в вызывающий метод.

Пример. Найти сумму и разность целочисленных полей двух объектов класса и занести их в аналогичные поля двух других объектов класса.

#include <iostream.h>

class Cls

{

private:

int k;

public:

Cls():k(0)

{

}

Cls(int n):k(n)

{

}

void shows()

{

cout<<"k="<<k<<endl;

}

void sum(Cls v1, Cls v2)

{

k=v1.k+v2.k;

}

Cls raz(Cls v2)

{

Cls tmp;

tmp.k=k-v2.k;

return tmp;

}

};

int main()

{

Cls c1(8), c2(7), c3, c4;

c1.shows();//8

c2.shows();//7

c3.sum(c1, c2);

c3.shows();//15

c4=c1.raz(c2);

c4.shows();//1

return 0;

}

Если поле класса описано с ключевым словом static, то его значение будет одинаково для всех объектов класса. Статическое поле аналогично статической переменной. Оно видно только внутри объекта класса, а время жизни совпадает с временем жизни программы (продолжает существовать даже в отсутствии объектов данного класса). Статическое поле в независимости от количества объектов размещается в единственном экземпляре и инициализируется один раз, по умолчанию заполняется нулями.

Статический метод не привязан к объектам класса и имеет доступ только к статическим элементам класса. Такой метод может быть вызван при отсутствии объектов классов.

Пример. Работа со статическими полями и методами. Использовать 2 объекта для работы со статической переменной.

#include <iostream>

using namespace std;

class Cls

{

static int k;

public:

static void adds()

{

k++;

}

static void shows()

{

cout<<"k="<<k<<endl;

}

};

int Cls::k=0;

void main()

{

Cls::shows();//k=0.

Cls x,y;

x.adds();

y.adds();

x.shows();//k=2.

y.shows();//k=2.

Cls::shows();//k=2.

}

Определение статического поля, как правило, располагается вне класса и представляет собой объявление глобальной переменной. При объявлении полей и методов разрешено использовать модификатор const, который помещается перед объявлением поля или между заголовком и телом функции. Константные методы не могут изменять значения полей своего класса.

Пример.

void shows() const{

//k++;//Ошибка!

cout<<"k="<<k<<endl;

}

\*Если объявление и модификация метода разделены, то модификатор const указывается в любом случае. Если модификатор const применяется для объектов класса, то такие объекты не могут быть изменены и можно использовать только константные методы.

**Использование массивов объектов**

При работе с массивами объектов осуществляется различный вызов конструктора и деструктор каждого элемента массива. Конструктор не должен содержать список параметров.

Пример. Программа, работающая с массивом класса.

#include <iostream>

using namespace std;

const int n=30;

class Cls{

char \*str;

public:

Cls()

{

static int i=0;

str=new char[n];

char st[n];

itoa(++i, st, 10);

strcpy(str, st);

cout<<"Object"<<\*str<<"is created"<<endl;}

~Cls(){

cout<<\*str<<"object destruction"<<endl;

delete str;

}

void out\_obj(){

cout<<"Object"<<\*str<<"there"<<endl;

}

};

int main()

{

Cls mas1[3];//3 раза сработает конструктор

for (int i=0;i<3;i++)

mas1[i].out\_obj();

Cls \*mas2=new Cls[2];

for (int i =0; i<2; i++) mas2[i].out\_obj();

delete[2]mas2;

return 0;

}

**Дружественные функции класса**

**Дружественная функция** - это функция, не являющаяся элементом класса, но имеющая доступ к любым элементам класса. Прототип дружественной функции может находиться в любом месте описания класса и должен начинаться с ключевого слова **friend**. Тело дружественной функции может находиться в любом месте программы.

Особенности дружественных функции:

1) При описании им автоматически предписывается атрибут inline.

2) Функции не привязаны к объектам класса.

3) Функции не могут использовать указатель this.

4) Вызываются как обычные функции.

Дружественные функции могут быть связаны с разными элементами разных классов. Дружественными функциями не могут быть конструктор и деструктор.

Пример. Написать программу, подсчитывающую сумму значений полей двух классов.

#include <iostream>

using namespace std;

class Cl2;

class Cl1

{

int x;

public:

Cl1(int a):x(a){}

friend int sum(Cl1, Cl2);

};

class Cl2

{

int y, z;

friend int sum(Cl1, Cl2);

public:

Cl2(int b, int c):y(b),z(c){}

};

int sum(Cl1 s, Cl2 f)

{

return s.x+f.y+f.z;

}

int main()

{

Cl1 m1(10);

Cl2 m2(20, 30);

cout<<sum(m1, m2)<<endl;//60

return 0;

}

**Указатель** **this** является указателем на адрес объекта класса и используется для доступа к элементам класса. Указатель this является скрытым первым параметром любого метода класса (кроме статических объектов). Тип указателя - текущий класс.

Пример. Написать программу, использующую указатель this.

#include <iostream>

using namespace std;

class Cls

{

int k;

public:

Cls(int n):k(n){}

int sum(Cls a)

{

return this->k\*2+a.k;

}

};

int main()

{

Cls a=1, b=10;

cout<<a.sum(b)<<endl;//12

cout<<b.sum(a)<<endl;//21

return 0;

}