

Курс «С++. Программирование на языке высокого уровня»

Павловская Т.А.

Лекция 5. Классы

Логика объектно-ориентированного подхода. Описание классов и объектов. Основные элементы класса: поля, методы, указатель `this`, конструкторы, деструкторы, операции. Дружественные функции и классы. Указатели на элементы классов.

ООП

"Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning."

Rich Cook

Преимущества и недостатки ООП

Преимущества (при создании больших программ):

- использование при программировании понятий, более близких к предметной области;
- локализация свойств и поведения объекта о одном месте, позволяющая лучше структурировать и, следовательно, отлаживать программу;
- возможность создания библиотеки объектов и создания программы из готовых частей;
- исключение избыточного кода за счет того, что можно многократно не описывать повторяющиеся действия;
- сравнительно простая возможность внесения изменений в программу без изменения уже написанных частей, а в ряде случаев и без их перекомпиляции.

Недостатки ООП:

- некоторое снижение быстродействия программы, связанное с использованием виртуальных методов;
- идеи ООП не просты для понимания и в особенности для практического использования;
- для эффективного использования существующих ОО систем требуется большая объем первоначальных знаний.

Свойства ООП

- **Инкапсуляция** - скрывание деталей реализации; объединение данных и действий над ними.
- **Наследование** позволяет создавать иерархию объектов, в которой объекты-потомки наследуют все свойства своих предков. Свойства при наследовании повторно не описываются. Кроме унаследованных, потомок обладает собственными свойствами. Объект в C++ может иметь сколько угодно потомков и предков.
- **Полиморфизм** - возможность определения единого по имени действия, применимого ко всем объектам иерархии, причем каждый объект реализует это действие собственным способом.

Класс (объект) – инкапсулированная абстракция с четким протоколом доступа

Технология разработки ОО программ

В процесс проектирования перед всеми остальными добавляется еще один этап - разработка иерархии классов.

1. В предметной области **выделяются понятия**, которые можно использовать как классы. Кроме классов из прикладной области, обязательно появляются классы, связанные с аппаратной частью и реализацией.
2. **Определяются операции** над классами, которые впоследствии станут методами класса. Их можно разбить на группы:
 - связанные с конструированием и копированием объектов;
 - для поддержки связей между классами, которые существуют в прикладной области;
 - позволяющие представить работу с объектами в удобном виде.
3. **Определяются функции**, которые будут виртуальными.
4. **Определяются зависимости** между классами.

Процесс создания иерархии классов - итерационный. Например, можно в двух классах выделить общую часть в базовый класс и сделать их производными.

Классы должны как можно ближе соответствовать моделируемым объектам из предметной области.

Описание класса

```
class <имя>{  
  [ private: ]  
  <описание скрытых элементов>  
  public:  
  <описание доступных элементов>  
};
```

Поля класса:

- могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс);
- могут быть описаны с модификатором `const`;
- могут быть описаны с модификатором `static`, но не как `auto`, `extern` и `register`.

Инициализация полей при описании не допускается.

©Павловская Т.А. могут быть глобальными и локальными.

(СПбГУ ИТМО)

Локальные классы

- внутри локального класса запрещается использовать автоматические переменные из области, в которой он описан;
- локальный класс не может иметь статических элементов;
- методы этого класса могут быть описаны только внутри класса;
- если один класс вложен в другой класс, они не имеют каких-либо особых прав доступа к элементам друг друга

Пример описания класса

```
class monstr{
    int health, ammo;
public:
    monstr(int he = 100, int am = 10)
        { health = he; ammo = am;}
    void draw(int x, int y, int scale, int position);
    int get_health(){return health;}
    int get_ammo(){return ammo;}
};
```

```
void monstr::draw(int x, int y, int scale, int position)
{ /* тело метода */ }
```

```
inline int monstr::get_ammo(){return ammo;}
```

Описание объектов

```
monstr Vasia;  
monstr Super(200, 300);  
monstr stado[100];  
monstr *beavis = new monstr (10);  
monstr &butthead = Vasia;
```

Доступ к элементам объекта

```
int n = Vasia.get_ammo();  
stado[5].draw;  
cout << beavis->get_health();
```

Константные объекты и методы

Константный объект:

```
const monstr Dead (0,0);
```

Константный метод:

```
int get_health() const {return health;}
```

Константный метод:

- объявляется с ключевым словом `const` после списка параметров;
- не может изменять значения полей класса;
- может вызывать только константные методы;
- может вызываться для любых (не только константных)

Указатель this

```
monstr & the_best(monstr &M){
    if( health > M.health()) return *this;
    return M;
}
void cure(int health, int ammo){
    this -> health += health;
    monstr:: ammo += ammo;
}
...
monstr Vasia(50), Super(200);
monstr Best = Vasia.the_best(Super);
```

Конструкторы

- ◆ Конструктор не возвращает значение, даже типа `void`. Нельзя получить указатель на конструктор.
- ◆ Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).
- ◆ Конструктор, вызываемый без параметров, называется конструктором по умолчанию.
- ◆ Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.

Конструкторы - продолжение

- ◆ Если программист не указал ни одного конструктора, компилятор создает его *автоматически*. Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов.
- ◆ *Конструкторы не наследуются.*
- ◆ Конструкторы нельзя описывать как `const`, `virtual` и `static`.
- ◆ Конструкторы глобальных объектов вызываются до вызова функции `main`.
- ◆ Локальные объекты создаются, как только становится активной область их действия.
- ◆ Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).

Вызов конструктора

Вызов конструктора выполняется, если в программе встретилась одна из конструкций:

имя_класса имя_объекта [(список параметров)];

имя_класса (список параметров);

имя_класса имя_объекта = выражение;

```
monstr Super(200, 300), Vasia(50), Z;
```

```
monstr X = monstr(1000);
```

```
monstr Y = 500;
```

Несколько конструкторов

```
enum color {red, green, blue};  
class monstr{  
    int health, ammo;  
    color skin;  
    char *name;  
public:  
    monstr(int he = 100, int am = 10);  
    monstr(color sk);  
    monstr(char * nam);  
    int get_health(){return health;}  
    int get_ammo(){return ammo;}  
  
};
```


Реализация конструкторов

```
monstr::monstr(int he, int am)
    {health = he; ammo = am; skin = red; name = 0;}
monstr::monstr(color sk){
    switch (sk){
        case red : health = 100; ammo = 10; skin = red; name = 0; break;
        case green: health = 100; ammo = 20; skin = green; name = 0; break;
        case blue : health = 100; ammo = 40; skin = blue; name = 0; break;
    }
}
monstr::monstr(char * nam){
    name = new char [strlen(nam) + 1];
    strcpy(name, nam);
    health = 100; ammo = 10; skin = red;
```

Список инициализаторов конструктора

```
monstr::monstr(int he, int am):
```

```
    health (he), ammo (am), skin (red), name (0){ }
```

Конструктор копирования

```
T::T(const T&) { /* Тело конструктора */ }
```

- при описании нового объекта с инициализацией другим объектом;
- при передаче объекта в функцию по значению;
- при возврате объекта из функции.

• при обработке исключений.

©Павловская Т.А.

(СПбГУ ИТМО)

Пример конструктора копирования

```
monstr::monstr(const monstr &M){  
    if (M.name){  
        name = new char [strlen(M.name) + 1];  
        strcpy(name, M.name);}  
    else name = 0;  
    health = M.health; ammo = M.ammo;  
    skin = M.skin;  
}
```

```
monstr Vasia (blue);  
monstr Super = Vasia;  
monstr *m = new monstr ("Ork");  
monstr Green = *m;
```

Статические поля

- Память под статическое поле выделяется один раз

```
class A {  
    public: static int count; /* Объявление */  
};  
int A::count; // Определение  
// int A::count = 10; // Вариант определения
```

- поля доступны через имя класса и через имя объекта:

```
A *a, b; cout << A::count << a->count << b.count;
```

- На статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как `private`, можно изменить только с помощью статических методов.
- Память, занимаемая статическим полем, не учитывается при определении размера объекта с помощью операции `sizeof`.

Статические методы

```
class A{  
    static int count;  
    public:  
        static void inc_count(){ count++; }  
};
```

```
A::int count;  
void f(){  
    A a;  
    // a.count++ — нельзя  
    a.inc_count();    // или A::inc_count();
```

Дружественные функции и классы

- ◆ Дружественная функция объявляется внутри класса, к элементам которого ей нужен доступ, с ключевым словом `friend`.
- ◆ Дружественная функция может быть обычной функцией или методом другого ранее определенного класса.
- ◆ Одна функция может быть дружественной сразу несколькими классами.

Дружественные функции - пример


```
class monstr;  
class hero{  
    public:  
        void kill(monstr &);  
};
```

```
class monstr{  
    friend int steal_ammo(monstr &);  
    friend void hero::kill(monstr &);  
};
```

```
int steal_ammo(monstr &M){return --M.ammo;}  
void hero::kill(monstr &M){  
    M.health = 0; M.ammo = 0;  
}
```

Дружественные классы - пример

```
class hero{  
    ...  
    friend class mistress;  
}  
class mistress{  
    ...  
    void f1();  
    void f2();  
}
```



Деструкторы

Деструктор вызывается автоматически, когда объект выходит из области видимости:

- для *локальных* объектов — при выходе из блока, в котором они объявлены;
- для *глобальных* — как часть процедуры выхода из main;
- для *объектов, заданных через указатели*, деструктор вызывается неявно при использовании операции delete.

Пример деструктора:

```
monstr::~~monstr() {delete [] name;}
```

Деструктор можно вызвать явным образом путем указания полностью уточненного имени, например:

```
    monstr *m; ...  
m -> ~monstr();
```

Деструктор:

- не имеет аргументов и возвращаемого значения;
- не может быть объявлен как `const` или `static`;
- не наследуется;
- может быть виртуальным
- Если деструктор явным образом не определен, компилятор автоматически создает пустой деструктор.

Перегрузка операций

. * ?: :: # ## sizeof

- при перегрузке операций сохраняются количество аргументов, приоритеты операций и правила ассоциации (справа налево или слева направо), используемые в стандартных типах данных;
- для стандартных типов данных переопределять операции нельзя;
- функции-операции не могут иметь аргументов по умолчанию;
- функции-операции наследуются (за исключением =);
- функции-операции не могут определяться как static.

Функции-операции

Формат:

```
тип operator операция ( список параметров) {  
    тело функции  
}
```

Функцию-операцию можно определить:

- как метод класса
- как дружественную функцию класса
- как обычную функцию

Перегрузка унарных операций

1. Внутри класса:

```
class monstr{  
    ...  
    monstr & operator ++()  
        {++health; return *this;}  
}
```

```
monstr Vasia;  
cout << (++Vasia).get_health();
```

Перегрузка унарных операций

2. Как дружественную функцию:

```
class monstr{  
    ...  
    friend monstr & operator ++( monstr &M);  
};  
monstr& operator ++(monstr &M) {++M.health; return M;}
```

3. Вне класса:

```
void change_health(int he){ health = he;}  
...  
monstr& operator ++(monstr &M){  
    int h = M.get_health(); h++;  
    M.change_health(h);  
    return M;}
```

Перегрузка постфиксного инкремента

```
class monstr{
    ...
    monstr operator ++(int){
        monstr M(*this); health++;
        return M;
    }
};
monstr Vasia;
cout << (Vasia++).get_health();
```

Перегрузка бинарных операций

1. Внутри класса:

```
class monstr{  
    ...  
    bool operator >(const monstr &M){  
        if( health > M.get_health())  
            return true;  
        return false; }  
};
```

2. Вне класса:

```
bool operator >(const monstr &M1, const monstr &M2){  
    if( M1.get_health() > M2.get_health())  
        return true;  
    return false;
```


Перегрузка операции присваивания

операция-функция должна возвращать ссылку на объект, для которого она вызвана, и принимать в качестве параметра единственный аргумент — ссылку на присваиваемый объект

```
const monstr& operator = (const monstr &M){  
    // Проверка на самоприсваивание:  
    if (&M == this) return *this;  
    if (name) delete [] name;  
    if (M.name){name = new char [strlen(M.name) + 1];  
                strcpy(name, M.name);}  
    else        name = 0;  
    health = M.health; ammo = M.ammo; skin = M.skin;  
    return *this;  
}
```

```
monstr A(10), B, C;  
C = B = A;
```

Перегрузка операций new и delete

- им не требуется передавать параметр типа класса;
- первым параметром функциям new и new[] должен передаваться размер объекта типа size_t (это тип, возвращаемый операцией sizeof, он определяется в заголовочном файле <stddef.h>); при вызове он передается в функции неявным образом;
- они должны определяться с типом возвращаемого значения void*, даже если return возвращает указатель на другие типы (чаще всего на класс);
- операция delete должна иметь тип возврата void и первый аргумент типа void*;

Операции выделения и освобождения памяти являются

© Павловская Т.А. элементами класса.

(СПбГУ ИТМО)

```
class Obj { ... };  
class pObjj{  
    ...  
    private:  
        Obj *p;  
};
```

```
pObjj *p = new pObjj;
```

```
static void * operator new(size_t size);  
void operator delete(void * ObjToDie, size_t size);
```

```
#include <new.h>  
SomeClass a = new(buffer) SomeClass(his_size);
```

Перегрузка операции приведения типа

operator имя_нового_типа ();

```
monstr::operator int(){  
    return health;  
}  
...  
monstr Vasia; cout << int(Vasia);
```

Перегрузка операции вызова функции

```
class if_greater{  
    public:  
        int operator () (int a, int b) const {  
            return a > b;  
        }  
};  
  
if_greater x;  
cout << x(1, 5) << endl; // x.operator () (1, 5)  
cout << if_greater()(5, 1) << endl;
```

Перегрузка операции индексирования

```
class Vect{
public:
    explicit Vect(int n = 10);
    //инициализация массивом:
    Vect(const int a[], int n);
    ~Vect() { delete [] p; }
    int& operator [] (int i);
    void Print();
private:
    int* p;
    int size;
};
```

Перегрузка операции индексирования

```
Vect::Vect(int n) : size(n){ p = new int[size];}
```

```
Vect::Vect(const int a[], int n) : size(n){
```

```
    p = new int[size]; for (int i = 0; i < size; i++) p[i] = a[i]; }
```

```
int& Vect::operator [] (int i){
```

```
    if(i < 0 || i >= size){
```

```
        cout << "Неверный индекс (i = " << i << ")" << endl;
```

```
        cout << "Завершение программы" << endl;
```

```
        exit(0); }
```

```
    return p[i];
```

```
}
```

Перегрузка операции индексирования

```
void Vect::Print(){  
    for (int i = 0; i < size; i++) cout << p[i] << " ";  
    cout << endl; }
```

```
int main(){  
    int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
    Vect a(arr, 10);  
    a.Print();  
    cout << a[5] << endl;  
    cout << a[12] << endl;  
    return 0;  
}
```


Указатели на элементы классов

Указатель на метод класса:

```
возвр_тип (имя_класса::*имя_указателя)(параметры);
```

описание указателя на методы класса monstr

```
int get_health() {return health;}
```

```
int get_ammo() {return ammo;}
```

имеет вид:

```
int (monstr:: *pget)();
```

Указатель можно задавать в качестве параметра функции:

```
void fun(int (monstr:: *pget)()){
```

```
    (*this.*pget)(); // Вызов функции через операцию .*
```

```
    (this->*pget)(); // Вызов функции через операцию ->*
```

Присваивание значения указателю на метод класса:

```
pget = & monstr::get_health;
```

```
monstr Vasia, *p; p = new monstr;
```

Вызов через операцию .* :

```
int Vasin_health = (Vasia.*pget());
```

Вызов через операцию ->* :

```
int p_health = (p->*pget());
```

Правила использования указателей на методы классов:

- Указателю на метод можно присваивать только адреса методов, имеющих соответствующий заголовок.
- Нельзя определить указатель на статический метод класса.
- Нельзя преобразовать указатель на метод в указатель на обычную функцию, не являющуюся элементом класса

Указатель на поле класса

```
тип_данных(имя_класса::*имя_указателя);
```

В определение указателя можно включить его инициализацию:

```
&имя_класса:имя_поля; // Поле должно быть public
```

Если бы поле health было объявлено как public, определение указателя на него имело бы вид:

```
int (monstr::*phealth) = &monstr::health;  
cout << Vasia.*phealth; // Обращение через операцию .*  
cout << p->*phealth; // Обращение через операцию ->*
```

Рекомендации по составу класса

Как правило, класс как тип, определенный пользователем, должен содержать скрытые (private) поля и следующие функции:

- конструкторы, определяющие, как инициализируются объекты класса;
- набор методов, реализующих свойства класса (при этом методы, возвращающие значения скрытых полей класса, описываются с модификатором const, указывающим, что они не должны изменять значения полей);
- набор операций, позволяющих копировать, присваивать, сравнивать объекты и производить с ними другие действия, требующиеся по сути класса;
- класс исключений, используемый для сообщений об ошибках с помощью генерации исключительных ситуаций