

Классы и объекты C++

Основной способ определения
пользовательского типа

Основа ООП

Это произведение доступно по лицензии
"Attribution-ShareAlike" ("Атрибуция — На тех же условиях") 4.0 Всемирная (CC BY-SA 4.0)
<http://creativecommons.org/licenses/by-sa/4.0/deed.ru>



March 23, 2021

Моделируемые *сущности* при программировании на C++ чаще всего представляются в виде **классов**.

Для решения сложных задач необходимы способы разделения частей программы (кода и данных) на отдельные части, каждая из которых соответствует отдельному понятию.

Описание класса — “чертёж”, по которому определяется содержимое и поведение объектов (экземпляров класса).

Пример постановки задачи

Создать программу для деканата, предназначенную для учета студентов.

Каждый студент имеет:

ФИО, номер зачетки, текущий курс обучения, кол-во сданных экзаменов в текущем году.

Действия:

Создать студента, удалить, получить информацию о студенте, зарегистрировать сдачу экзамена (если сдал необходимое кол-во — перевести на следующий курс).

Подход 70-х (Fortran) “Массивы значений”

```
INTEGER NZ(200)
INTEGER YEAR(200)
INTEGER NEX(200)
CHARACTER*32 NAME(20)
```

...

```
C --- и функции для обработки,
C --- получающие номер студента и
C --- другие параметры
```

Данные о каждом студенте разрознены. Любая функция может из-за ошибки в индексе изменить данные не того студента. Изменение и отладка программы — очень сложная задача: любой участок кода может изменить любые данные.

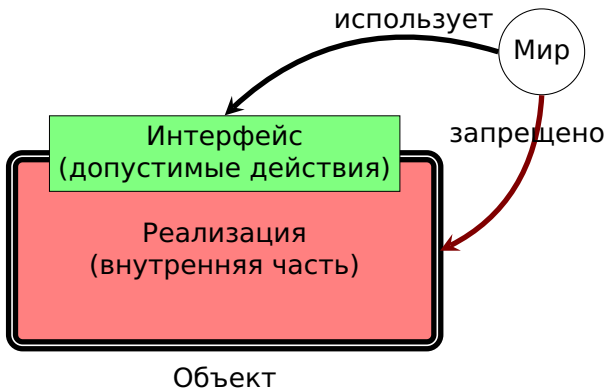
Подход 80-х (C, Pascal) “Структуры”

```
struct Stud {  
    char name[80];  
    int  nz, year, nex;  
};  
  
void printName( struct Stud *s )  
{  
    printf( "%s", s->name );  
    // ++year;  
}
```

Хорошо: Данные о каждом студенте сосредоточены в структуре. Каждая функция работает только с тем студентом, указатель на которого ей передали.

Плохо: Любая часть программы, получив указатель на студента, может делать с ним все, что захочет.

Методы ООП – “Инкапсуляция”



Воплощение инкапсуляции на C++

```
class Stud {  
    public:  
        void print() const;  
        void exam();  
        int a; // так плохо, но можно  
    private:  
        char *name;  
        int nz;  
        int someAction();  
};  
void f() {  
    Stud t; // Создали объект "t" типа "Stud" (подозрительно)  
    t.a = 12; // можно, но плохо  
    t.nz = 15; // нельзя (private)  
    t.print(); // можно  
}
```

Имя класса (имя типа)

Открытая часть (интерфейс)

Функция-член класса

Закрытая часть (реализация)

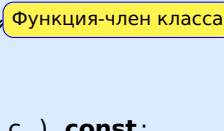
Член данных класса

К членам класса из открытой части (**public**) имеет доступ любая функция программы. К закрытым членам класса (**private**) имеют доступ только функции, описанные в самом классе.

Функции-члены класса (2)

Если функция описана в классе **без** ключевых слов “static” или “friend” — то это **функция-член класса** (всё равно, public или private).

```
class Stud {  
    public:  
        void print() const;  
        void exam();  
    private:  
        int someAction();  
        ptrdiff_t* unknownFunction( char c ) const;  
        friend void ff(); // не член - друг  
        static void gg(); // статический член класса  
};
```



У неё есть 3 важных отличия от просто функции.

Функции-члены класса (2)

Первое отличие — она принадлежит пространству имен данного класса.

```
class Stud {  
    public:  
        void print() const;  
    private:  
        int someAction() { nex += 2; };  
};  
  
void Stud::print() const  
{  
    cout << name << endl;  
}
```

Неявный inline

имя класса::имя функции

При определении функции снаружи класса необходимо указать, к какому классу она относится. Для этого используется оператор “::”. У разных классов могут быть функции-члены с одинаковыми именами и аргументами.

Функции-члены класса (3)

Второе отличие — она имеет право доступа к закрытой части класса (и к членам данных, и к функциям-членам).

```
class Stud {
public:
    void print() const;
private:
    int nex;
    char *name;
    int someAction() { nex += 2; }; // Ok
};

void Stud::print() const
{
    cout << name << endl; // Ok
    someAction();         // Ok
}
```

Функции-члены класса (4)

Третье отличие — её можно вызвать **только** для объектов, которые **являются** экземплярами данного класса.


```
class Stud {
    public:
        void print() const;
    private:
        int someAction() { nex += 2; };
};
// in some function ...
Stud t; Stud *pt = &t;
t.print();
pt->print();
```

Для этого при вызове функций-членов снаружи используются операторы “.” и “->” Слева от точки — имя объекта, слева от “->” — указатель на объект. Для доступа к членам объекта, для которого вызвана функция-член, не требуется использовать операторы.

Функции-члены класса (5)

Указатель “this” (следствие третьего отличия)

При вызове функции-члена ей неявно передается специальный аргумент — указатель на объект, для которого она вызвана. Внутри функции-члена этот указатель обозначается ключевым словом **this**. И при доступе к членам объекта он неявно используется.

```
class Stud {  
    public:  
        void print() const;  Запрещает изменять *this  
    private:  
        int someAction() { nex += 2; }; // this->nex += 2  
};  
void Stud::print() const  
{  
    cout << name << endl; // то же что и  
    cout << this->name << endl;  
}  
Stud f;  
f.print(); // передаётся указатель на f
```

Функции-члены класса (6)

Функции-члены также могут быть перегружены.

```
class Stud {  
    public:  
        void f1( int a );  
        void f1( char a );  
        void f1( const Stud *a );  
        void f1( const Stud &a );  
        void f1( Stud &&a );  
        void f1( const Prep &a );  
  
        void f2() const;  
        void f2();  
  
        void f3() &;  
        void f3() &&;  
};
```

Преобразования для неявного аргумента (**this**)
не производятся.

Среди функций-членов есть специальные:

- **конструкторы,**
- **деструктор,**
- **функции-операторы.**

Конструкторы

Всегда вызываются при создании объекта.

Имя функции-конструктора совпадает с именем класса.

```
class Stud {
public:
    Stud();
    Stud( const char *aname, int anz);
    Stud( const Stud &r );
    Stud( int id );
    // .....
};
Stud::Stud()
{ /* ..... */}
```

Конструкторы могут быть перегружены. Тип возвращаемого значения не указывается — всегда возвращает объект. Инструкцию **return** применять **нельзя**.

C++11: Один конструктор можно вызывать из другого.

Деструктор

Вызываются при уничтожении объекта (почти всегда автоматически).

Имя функции-деструктора — знак тильды спереди, а затем имя класса.

```
class Stud {  
    public:  
        // .....  
        ~Stud(); // - деструктор (описание)  
        // .....  
};  
Stud::~~Stud()    // - определение деструктора  
{ /* ..... */}  
  
void g()  
{  
    Stud t("Vasya", 543 );  
    // ...  
} // здесь будет вызван деструктор для объекта "t"
```

Деструктор всегда без аргументов, без возвращаемого значения, и всегда один.

Конструктор по умолчанию (1)

Конструктор по умолчанию — это тот конструктор, который можно вызвать без аргументов. Создает объект по умолчанию.

```
class Stud {  
    public:  
        Stud(); // Конструктор по умолчанию  
        // .....  
};  
class Dog {  
    public:  
        Dog( double rating = 0 ); // И обычный конструктор,  
                                     // и конструктор по умолчанию  
        // .....  
};  
Stud a;  
Dog gafgaf; ← Вызовы конструкторов по умолчанию
```

Конструктор по умолчанию (2)

Если в классе нет ни одного конструктора, компилятор сам создаст конструктор по умолчанию (вызвав конструкторы по умолчанию всех вложенных объектов).

```
class XStud {  
    private:  
        Stud s;  
        int a, b;  
};  
XStud z;
```

Созданный компилятором конструктор по умолчанию вызовет `Stud::Stud()` (если он существует), и конструкторы по умолчанию для `a` и `b`. Для фундаментальных типов конструкторы по умолчанию ничего не делают (переменные имеют неопределенное значение). Если в классе есть хотя бы один конструктор, написанный пользователем, то компилятор не создает конструктор по умолчанию. Если понятие “объект по умолчанию” не имеет смысла, то не следует создавать конструктор по умолчанию.

Конструктор по умолчанию (3)

В C++98, C++03 массивы пользовательских объектов можно было создать только по умолчанию. Поэтому для создания массивов объектов было необходимо было наличие конструктора по умолчанию.

```
Stud group[50];  
Stud *xg = new Stud[100];
```

В C++11 можно создавать массивы объектов, передавая аргументы для других конструкторов.

```
Stud gr[] { {"Vasia",1}, {"Olya",2}, {"Katya",5} };  
Stud *ngr = new Stud[2] { {"Pol",17}, {"Sol",21} };  
delete[] ngr;
```

Конструктор копирования

Конструктор копирования создает копию объекта, явно или неявно.

```
class Stud {  
    public:  
        Stud( const Stud &r ); // конструктор копирования  
};  
Stud fun1( Stud x )  
{  
    x.exam();  
    return x; // неявно вызывается к-тор копирования  
}  
void g()  
{  
    Stud a("Harry",721); Stud c("Unknown", 0 );  
    Stud b = a; // явно вызыв. к-тор копирования  
    c = fun1(a); // неявно вызыв. к-тор копирования  
}
```

C++11: Есть конструктор перемещения

```
Stud( Stud &&r ); // конструктор перемещения
```

Конструктор копирования (2)

Если в классе нет конструктора копирования, компилятор сам попытается создать конструктор копирования (вызвав конструкторы копирования всех вложенных объектов).

Для фундаментальных типов действие конструктора копирования заключается в побитовом копировании.

Если объекты классов сами захватывают ресурсы (память, файловые дескрипторы, разделяемые объекты, сокеты ...), то необходимо или написать свой конструктор копирования (может быть перемещения), или запретить копирование объекта (=delete).

Конструктор копирования (3)

В этом примере используется (правильно) конструктор копирования, созданный компилятором.

```
class D {  
  public:  
  D( int ax, int ay ) { x = ax; y = ay; }  
  // D( const &D ) = delete; // так можно запретить  
  private:  
  int x, y;  
};  
D f( 12, 5);  
D z = f; // Работает конструктор копирования,  
        // созданный компилятором  
        // (побитово копирует a и b)  
  
D y { f }; // Аналогично, но C++11
```

Конструктор копирования (4)

Пример недопустимого использования конструктора копирования, созданного компилятором.

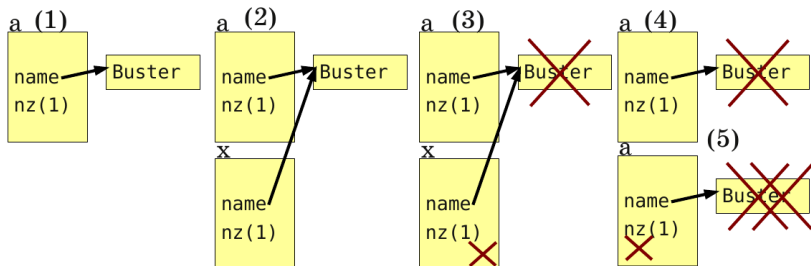
```
class Stud {
public:
    Stud( const char *aname, int anz);
    // ... здесь нет конструктора копирования
private:
    int nz;
    char *name; // - указатель на ресурс (память)
};
Stud::Stud(const char *aname, int anz)
{
    nz = anz; // лучше использовать список инициализации
    name = new char[ strlen(aname)+1 ]; //- Захват ресурса
    strcpy( name, aname );
}
Stud::~Stud()
{
    delete[] name; // - Освобождение ресурса
}
```

На первый взгляд – ничего страшного.

Конструктор копирования (5)

Продолжение примера.

```
void g( Stud x )  
{  
    x.print();  
}  
// (3)  
void f()  
{  
    Stud a("Buster", 1 ); // (1)  
    g( a ); // (2)  
    a.print(); // (4)  
} // (5)
```



Конструктор копирования (6)

Продолжение примера. Возможная реализация конструктора копирования.

```
class Stud {  
  public:  
    Stud( const Stud &rhs ); - описание  
    // Stud( Stud &&rhs ); - конструктор перемещения  
    // ....  
};  
  
Stud::Stud( const Stud &rhs )  
{  
  nz = rhs.anz; // или через список инициализации  
  name = new char[ strlen(rhs.name)+1 ]; // - Захват  
                                           // ресурса  
  strcpy( name, rhs.name );  
}
```

P.S. Более правильно использовать идиому “захват ресурса есть инициализация” – Resource Acquisition Is Initialization (**RAII**).

Конструкторы преобразования

Этот конструктор можно вызвать с одним аргументом другого типа.

```
class Stud {  
    public:  
        Stud( int id ); // - Конструктор преобразования  
        Stud( const char *name, int id = 0 ); // - аналогично  
        // .....  
};
```

Используется для явного или неявного преобразования объекта другого типа в определяемый.

```
void f( Stud x ) { /* ..... */ }  
void g()  
{  
    Stud a( 12 ); // явный вызов к-тора преобразования  
    f( 54 );      // неявный вызов к-тора преобразования (+)  
                 // 4-тый уровень соответствия  
    f( "Vasya" );  
}
```

Запрет неявного преобразования

Конструктор, допускающий вызов с одним аргументом, можно запретить использовать для неявных преобразований, используя ключевое слово **explicit**.

```
class Stud {  
    public:  
        explicit Stud( int id );  
        // .....  
};  
  
void f( Stud x ) { /* ..... */ }  
void g()  
{  
    Stud a( 12 ); // явный вызов к-тора преобразования  
    f( 54 );      // ЗАПРЕЩЕН неявный вызов к-тора  
                  // преобразования - ошибка  
}
```

Список инициализации

Инициализацию вложенных объектов можно и нужно проводить, вызывая конструкторы вложенных объектов в **списке инициализации**.

```
class Stud {
public:
    Stud( const char *aname, int anz );
    // .....
private:
    int nz;
    int year = 1; // C++11
    char *name;
};
Stud::Stud( const char *aname, int anz )
    :nz( anz ), name( new char[ strlen(aname)+1 ] )
    // это - список инициализации
{
    strcpy( name, aname );
}
```

Это **обязательно** для ссылок, константных членов и объектов без конструктора по умолчанию. Порядок инициализации – как в описании класса:

Функции-друзья класса

Функции, помеченные в классе ключевым словом “**friend**” (дружественные функции), имеют доступ к закрытой части класса. При этом не получают “**this**”, не входят в пространство имён класса. Вызываются как просто функции.

```
class Stud {
public:
    // .....
    friend void swap( Stud &a, Stud &b); // дружественная
                                         // функция
private: // .....
};

void swap( Stud &a, Stud &b ) // определение
{                               // дружественной функции
    int t = a.nz; a.nz = b.nz; b.nz = t;
    char *nm = a.name; a.name = b.name; b.name = nm;
}
Stud x( "Vasya", 15 ), y( "Olya", 72 );
swap( x, y );
```

Статические члены данных класса: величины, общие для всех объектов данного класса. Например: количество объектов данного класса, для студентов – количество экзаменов в конце семестра, для генетической модели организмов – количество хромосом для данного вида, для модели механического взаимодействия тел – гравитационная постоянная.

Статические функции-члены класса: действия, которые применяются к классу целиком, а не к отдельным объектам. Например: узнать количество объектов данного типа, определить норму сбора огурцов для с/х рабочих, узнать или задать символ-разделитель для системы ввода вывода.

Описание и определение статических членов данных

```
class Stud {  
    private:  
    char *name;  
    int nz;  
    static int exam_per_year; // статический член данных  
    static const int zzz = 12; // описание и определение  
};  
  
int Stud::exam_per_year = 5; // определение
```

Для доступа к статическим членам данных используется оператор “::”. Статическая переменная должна быть инициализирована снаружи описания класса (обычно там же, где определяются функции-члены класса).
Исключение: константные интегральные типы.
Инициализация происходит до начала работы функции *main*, уничтожение – после ее завершения.

Описание и определение статических функций-членов

При описании используется ключевое слово **“static”**.

```
class Stud {
    public:
        static int get_epy();
        static void set_epy(int n);
        static int get_epy2()
            { return exam_per_year; } // .....
};
int Stud::get_epy()
{
    return exam_per_year;
}
void Stud::set_epy( int n )
{
    exam_per_year = n;
}
```

Определение не отличается от определения обычной функции-члена. Нет указателя **“this”**, и всего, что с ним связано.

Статические функции-члены

Статические функции-члены класса вызываются для класса целиком, а не для его объектов. Для этого используется оператор “::”.

```
void g()  
{  
    int e = Stud::get_epy();  
}
```

Статические функции-члены класса:

- имеют право доступа к закрытой части класса
- принадлежат пространству имен класса
- **не** вызываются для объекта класса – поэтому **не** имеют указателя **this**, **не** могут иметь слово **const** после списка аргументов, и могут использовать нестатические данные класса только в том случае, если явно передать объект класса.

Модификатор mutable.

В объекте могут быть члены данных, которые должны изменяться и для константных объектов. Для объявления таких данных используется модификатор **mutable**.

```
class Stud {
    public:
        void print() const;
    private:
        mutable int n_print;
};
void Stud::print() const
{
    cout << name << endl; // не изменяем name
    ++n_print;             // можно - mutable
    name = "Mister_X";    // нельзя, *this - const
}
```