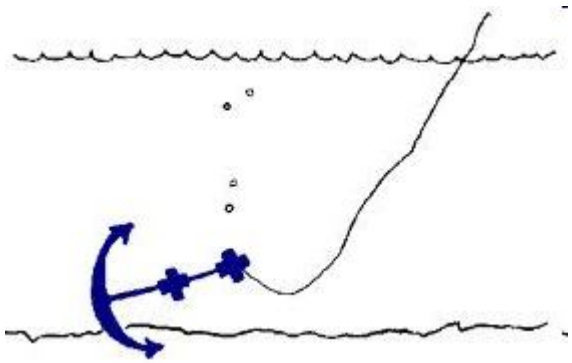
Карпов В.Э.

Объектно-ориентированное программирование

C++. Лекция 7



Дружественные функции и дружественные классы

Иногда желательно дать функциям, не являющимися элементами, доступ к личной части объекта класса. Для этого в декларации класса помещается декларация функции, перед которой стоит ключевое слово ***friend***.

Особенности дружественных функций:

- дружественная функция может быть другом двух и более классов;
- декларация friend - это только декларация, вводящая имя функции в самую широкую область действия в программе; декларацию можно поместить как в private-, так и в public- части описания класса – компилятору это безразлично;
- дружественная функция лишь имеет права доступа к приватной части объекта: если она не является полноправной функцией-элементом, то у нее нет и указателя **this**.

Иными словами, встретив ключевое слово ***friend***, компилятор просто отключает проверку прав доступа.

Замечание: Друзья не наследуются

Пример 1

Функция - элемент одного класса может быть дружественной иному классу.

```
class x
{   ...
  void f();
};
class y
{   ...
  friend void x::f(); //функция f() из класса x объявляется дружественной
};
class x
{   ...
  friend class y; //все функции класса y являются дружественными классу x
}
```

Пример 2

```
class X
{
private: int val;
public: X(int v=0) {val = v;};
        friend int f(X&);
        friend int g(X);
        int h(X) { cout << "\nfunc h:(" << x.val << ")\n"; return x.val; };
};
int f(X &x) { cout << "\nfunc f:(" << x.val << ")\n"; return x.val; }
int g(X x) { cout << "\nfunc g:(" << x.val << ")\n"; return x.val; }

void main()
{     X w;
      w = 199;
      X x(2), y(33);
      f(x);
      f(1);
      g(y);
      g(89);
      x.h(x);
      x.h(y);
      y.h(107);
}
```

Переопределение операторов

```
struct Complex
{
  float re, im;
  Complex(float r=0, float i=0) { re = r; im = i; }
  Complex Plus(Complex &arg) { return Complex(re+arg.re, im+arg.im); }
  Complex Minus(Complex &arg) { return Complex(re-arg.re, im-arg.im); }
};
…
Complex a(1,1), b(2,3), c;
c = a.Plus(b);
c = a.Minus(b);
```

Однако гораздо приятнее было бы использовать такие вызовы:

```
c = a+b;
c = a-b;
```

Механизм переопределения

- Разрешены имена функций вида "**operator@**", где **@** - некоторый зарезервированный в C++ оператор.
- Компилятор, встретив в программе выражение, в котором присутствует оператор **@**, проверяет тип аргументов. Если аргументом является объект, то проверяется, была ли определена функция "**operator@**" (в списке методов или еще где). Если да, то указание оператора **@** компилятор интерпретирует как вызов этой функции "operator@".

Использование оператора - это лишь сокращенная запись явного вызова функции оператора. Иными словами, имя функции – это "**operator@**".

- При переопределении операторов нельзя менять старшинство операций и их синтаксис.

Бинарные и унарные операторы

Бинарные операторы:

- как функция-элемент с одним аргументом: a.operator@(b);
- как внешняя функция с двумя аргументами: operator@(a,b).

Унарные операторы:

- как функция-элемент без аргументов: a.operator@();
- как внешняя функция с одним аргументом: operator@(a).

Обычно в качестве внешних функций используют дружественные данному классу функции. По крайней мере, это бывает удобно.

Пример

```
class Compl
{   private:        double re, im;
    public:

    void show() {cout<<"\n(" <<re<< ","<<im<<")";};

    Compl(double r = 0, double i = 0) {re = r; im = i;};
    friend Compl operator - (Compl); // Унарный минус
    friend Compl operator+(Compl, Compl);
    friend Compl operator*(Compl c1,CCompl c2)
      {return Compl(c1.re*c2.re-c1.im*c2.im, c1.im*c2.re+c1.re*c2.im);};
    Compl operator % (Compl);
    Compl operator ++ () { re++, im++; return (Compl(re,im)); };
    int operator<(Compl);
    Compl operator |(Compl);
    int operator[](int);
};
```

Продолжение

```
// Реализация методов. На «смысл» реализации внимания обращать не надо
int Compl::operator[](int a) { cout<<"\nOperator []:" << a << "\n"; return 0; };
Compl Compl::operator % (Compl c) { this->show(); c.show(); return 1; }
int Compl::operator<(Compl c) { return (this->re<c.re?1:0); }
Compl Compl::operator |(Compl c) { return c*c; };

Compl operator % (Compl c1, Compl c2) { c1.show(); c2.show(); return 1; }
Compl operator +(Compl c1, Compl c2) { return Compl(c1.re+c2.re, c1.im+c2.im); };
Compl operator - (Compl c) {return Compl(-c.re, c.im);};

void main(void)
{     Compl a, b(10), c(20,30), d;
      d = a+b+c+556;
      d = -d;
      d = a|d;
      Compl e = a+b+c+d;
      e = ++e;
      ++e; // или: e.operator++();
      a = b%c;
      a = c%1; // Пройдет всегда
      a = 1%c;          // Пройдет только в том случае, если
                        // оператор % переопределен как внешняя функция
}
```

Скобочные операторы

- При переопределении квадратных скобок (**[]**) компилятор требует наличия одного аргумента (любого типа).
- При переопределении **круглых** скобок количество аргументов произвольно. В частности, можно переопределять круглые скобки, используя неопределенное количество аргументов.

```
class A
{
…
    int operator[](A x) { … }
    A operator()(int i, int j, int k, A x) { … }
    A operator()(...);
};
```

Постфиксные и префиксные операторы

При переопределении операторов вида '++' или '--' следует иметь в виду: эти операторы утрачивают свою порядковую значимость:

Результат вычисления выражения

```
b = a++;
```

будет таким же, как и

```
b = ++a;
```

Т.е. сначала будет отработан оператор '++', а затем произойдет присваивание. Это связано с механизмом подстановки, когда вместо b = a++ и b = ++a компилятор сформирует вызов

```
b = a.operator++();
```

Операторы new и delete

Ограничения на применение переопределения этих операторов достаточно естественны и касаются типов аргументов:

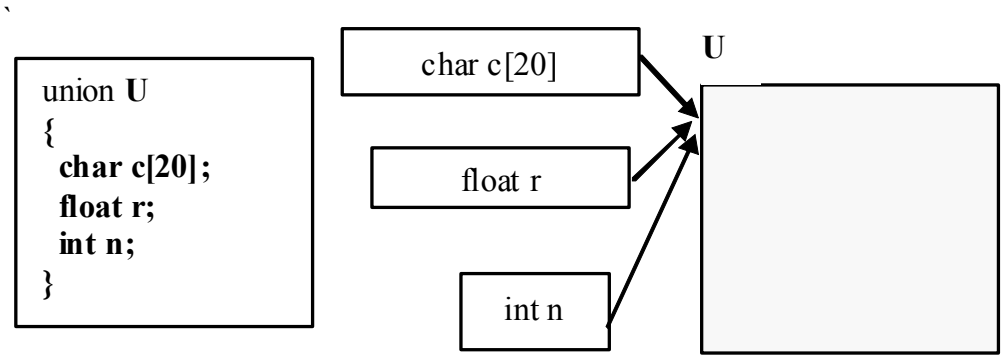
- void operator delete(**void *x**) { ... }
- void *operator new(**size_t x**) { ... }

Объединения

- Объединение – это структура, в которой все элементы имеют одинаковый адрес. Обычно объединения применяются в целях экономии памяти. Например, если в каждый момент времени значимым является только один элемент структуры, то целесообразно применять именно объединение.

- Исходя из определения объединения как структуры (и, следовательно, класса), справедливыми должны быть все соглашения, касающиеся работы с объектами (определения методов, конструкторов, деструкторов и т.п.)

- Использование объединений имеет некоторые особенности и ограничения. В частности, объединение не может использоваться как родительский класс или иметь родителей.



Пример

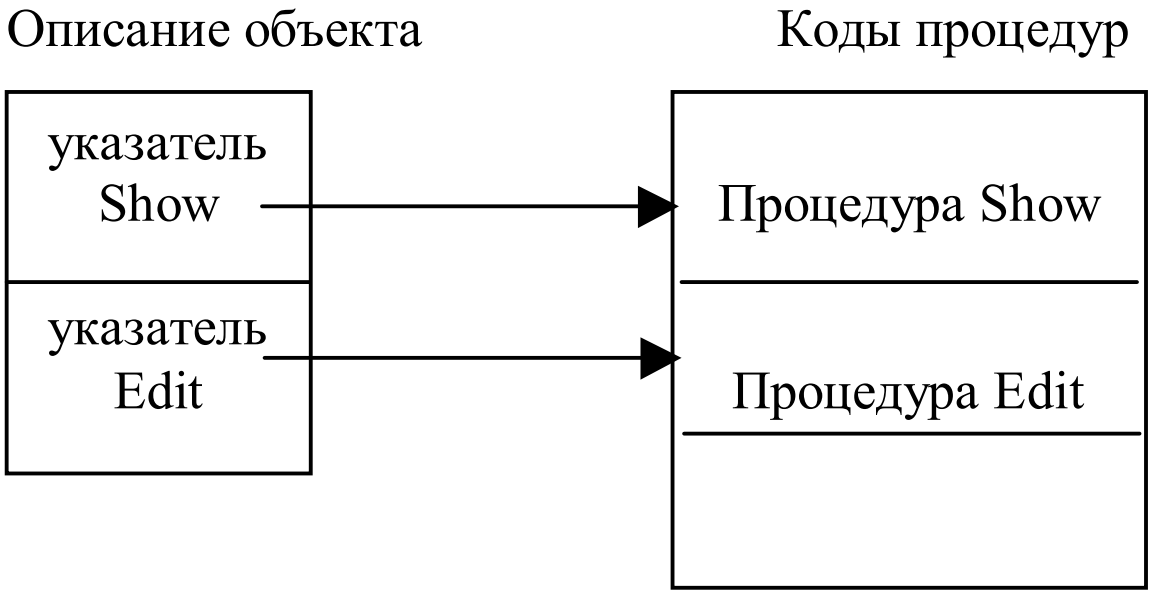
```
union U
{
    int i;
    char c;
    U(int n) { i=n; }
    U(char n) { c=n; }
    void show() { printf("i=%d, c='%c'\n",i,c+'0');};
};

void main(void)
{
    U a(1);
    a.show();
    a='2';
    a.show();
}
```

ВИРТУАЛЬНЫЕ ФУНКЦИИ И ВИРТУАЛЬНЫЕ КЛАССЫ

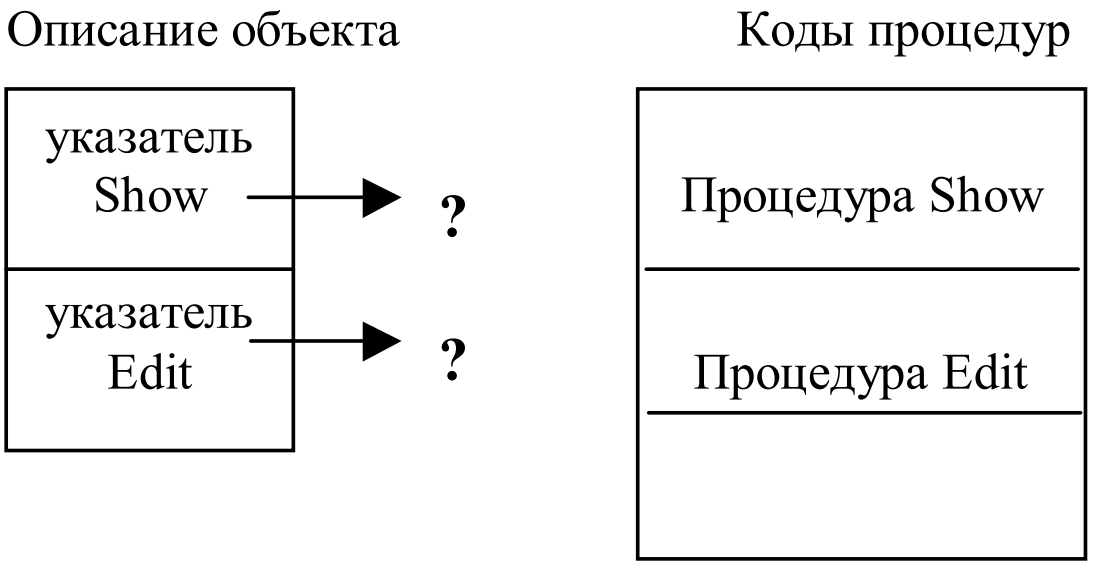
Раннее и позднее связывание

- Те правила (или методы) объектов, которые рассматривались нами выше, являлись *статическими*, т.е. во время компиляции программы каждый метод объекта однозначно связывался с реализующей его процедурой.
- Это – т.н. *раннее связывание* (т.е. связывание на этапе компиляции).



Позднее связывание

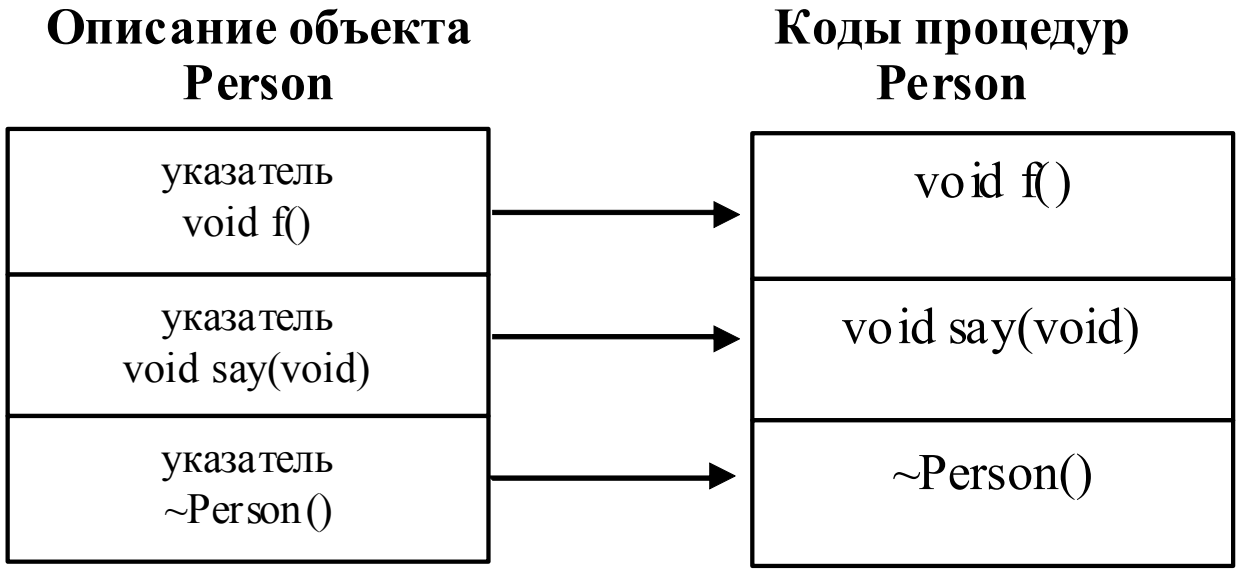
- Несмотря на сгенерированный компилятором код процедур методов, объекту **не известны их адреса**.
- Связь метода с реализующей процедурой определяется **на этапе выполнения программы** (позднее связывание).



•Указание компилятору того, что следует использовать именно позднее связывание, осуществляется введением ключевого слова ***virtual*** при описании метода. Такой метод называется *виртуальным*.

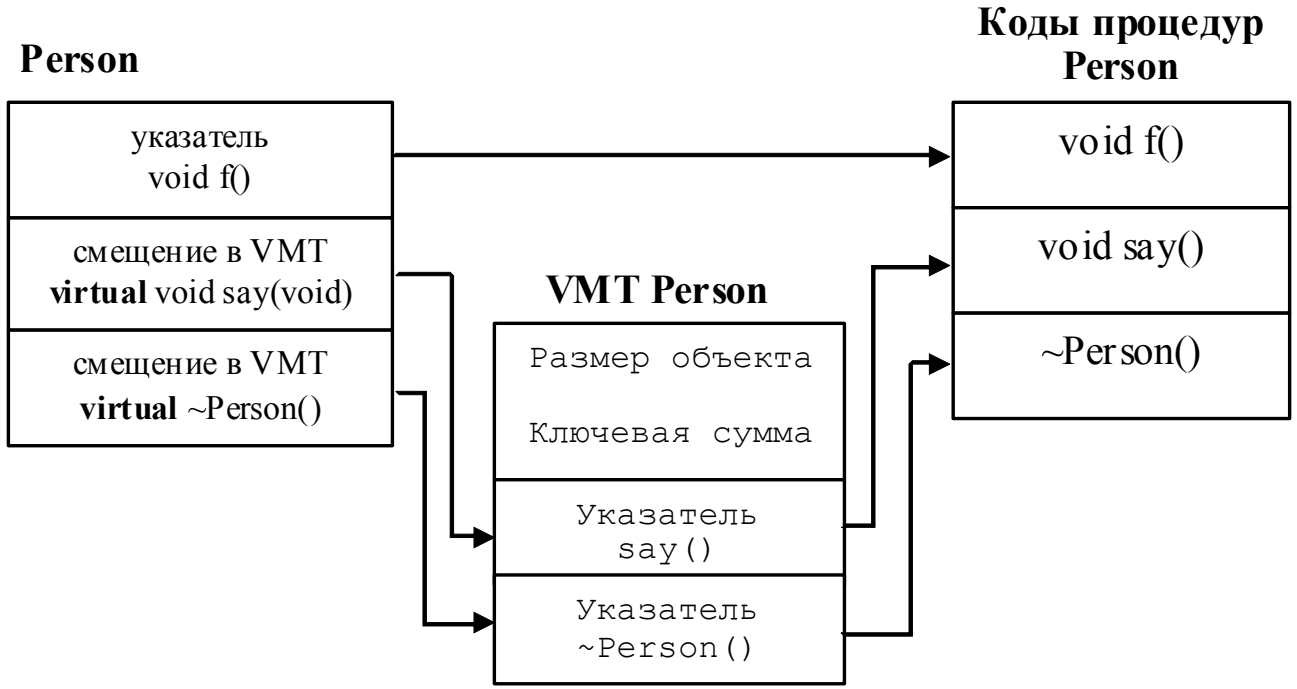
"Обычный" объект

```
struct Person
{
  void f(void) { printf("A::f()\n"); }
  void say(void) { printf("Person::say()\n"); }
  ~Person() { printf("~Person.\n"); }
};
```



Добавление виртуальности

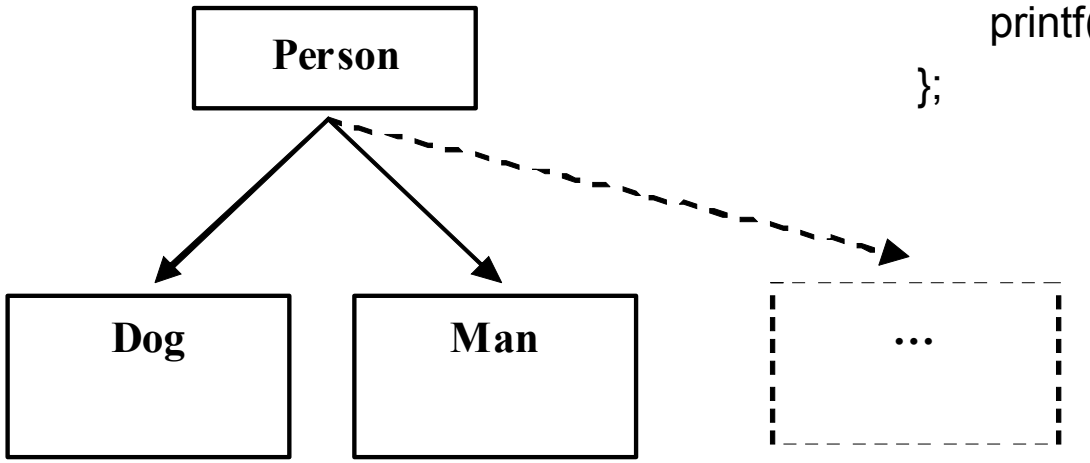
```
struct Person
{
  void f(void) { printf("A::f()\n"); }
  virtual void say(void) { printf("Person::say()\n"); }
  virtual ~Person() { printf("~Person.\n"); }
};
```



Структура классов задачи

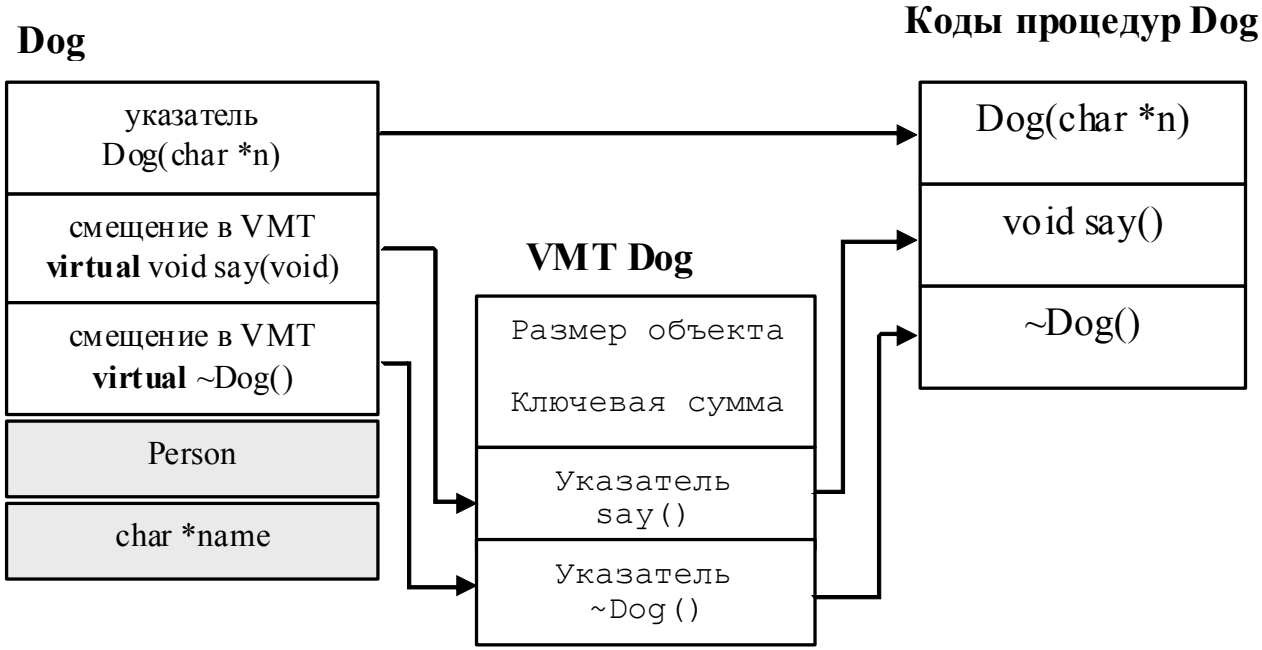
- Задача – создать гибкую (удобную) систему управления объектми.
- Производные классы Dog и Man

```
struct Person
{
  void f(void) { printf("A::f()\n"); }
  virtual void say(void) {
      printf("Person::say()\n"); }
  virtual ~Person() {
      printf("~Person.\n"); }
};
```



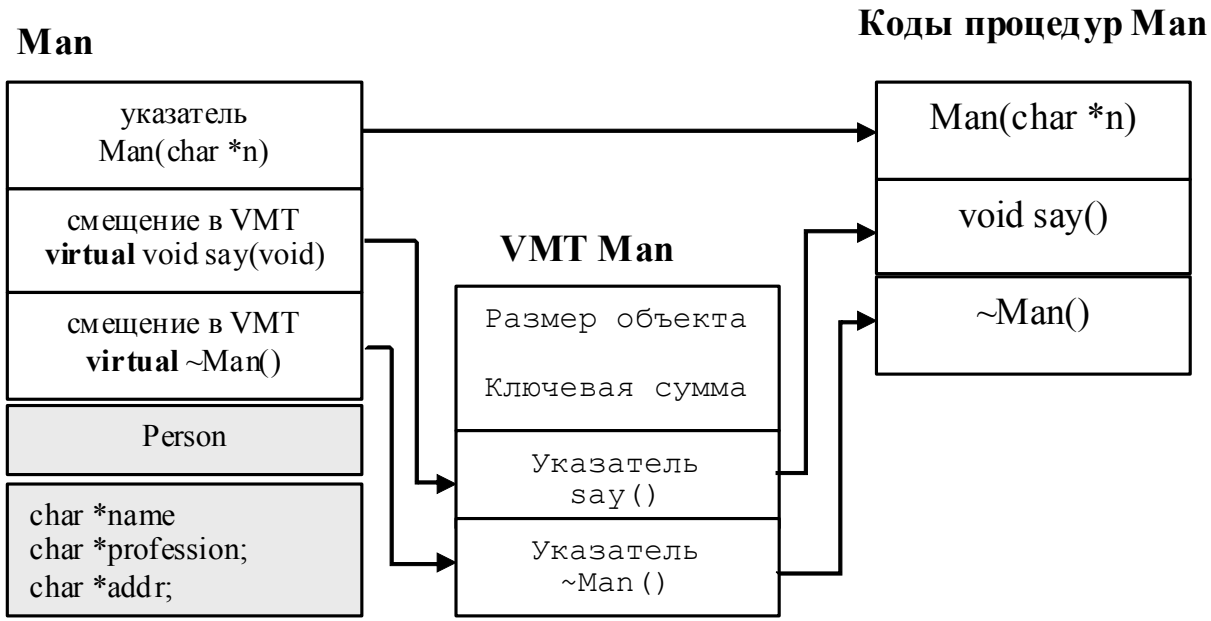
Дочерний класс Dog

```
struct Dog: public Person
{
  char *name;
  Dog(char *n) { name = n; }
  virtual void say(void) { printf("Dog: my name is %s\n", name); }
  virtual ~Dog() { printf("~Dog.\n"); }
};
```



Дочерний класс Man

```
struct Man: public Person
{
  char *name;
  char *profession;
  char *addr;
  Man(char *n) { name = n; }
  virtual void say(void) { printf("Man: my name is %s\n", name); }
  virtual ~Man() { printf("~Man.\n"); }
};
```

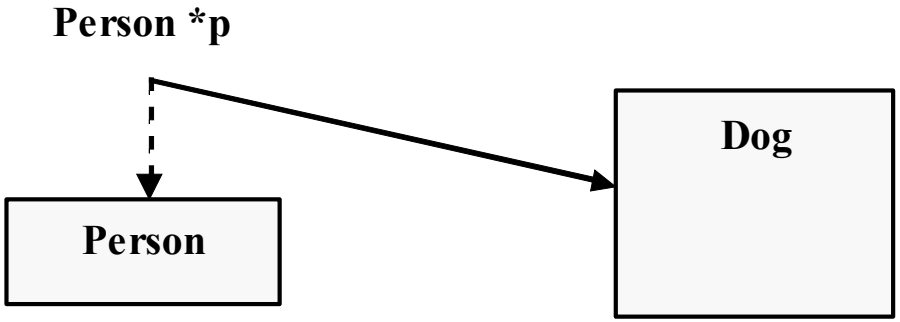


"Маленькие" и "большие" объекты

- Пока мы получили механизм **косвенного** вызова метода, который, очевидно, работает медленнее метода непосредственной адресации.

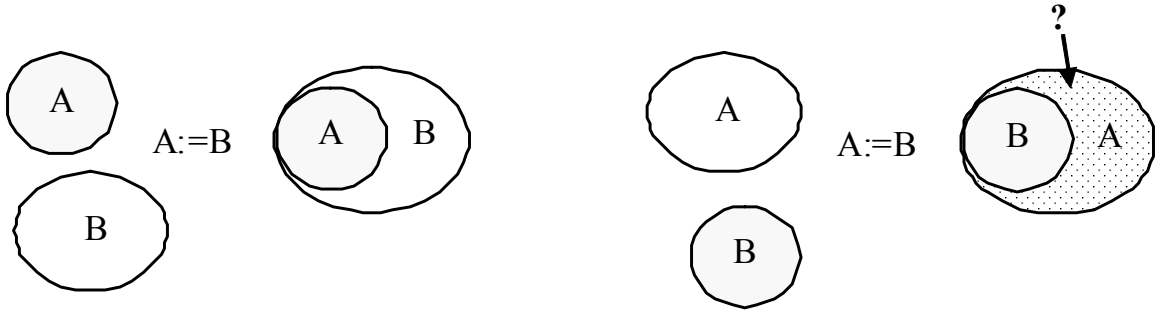
- Попробуем выделить память следующим образом:

```
Person *p;
p = new Dog("Pif");
```



Общее правило:

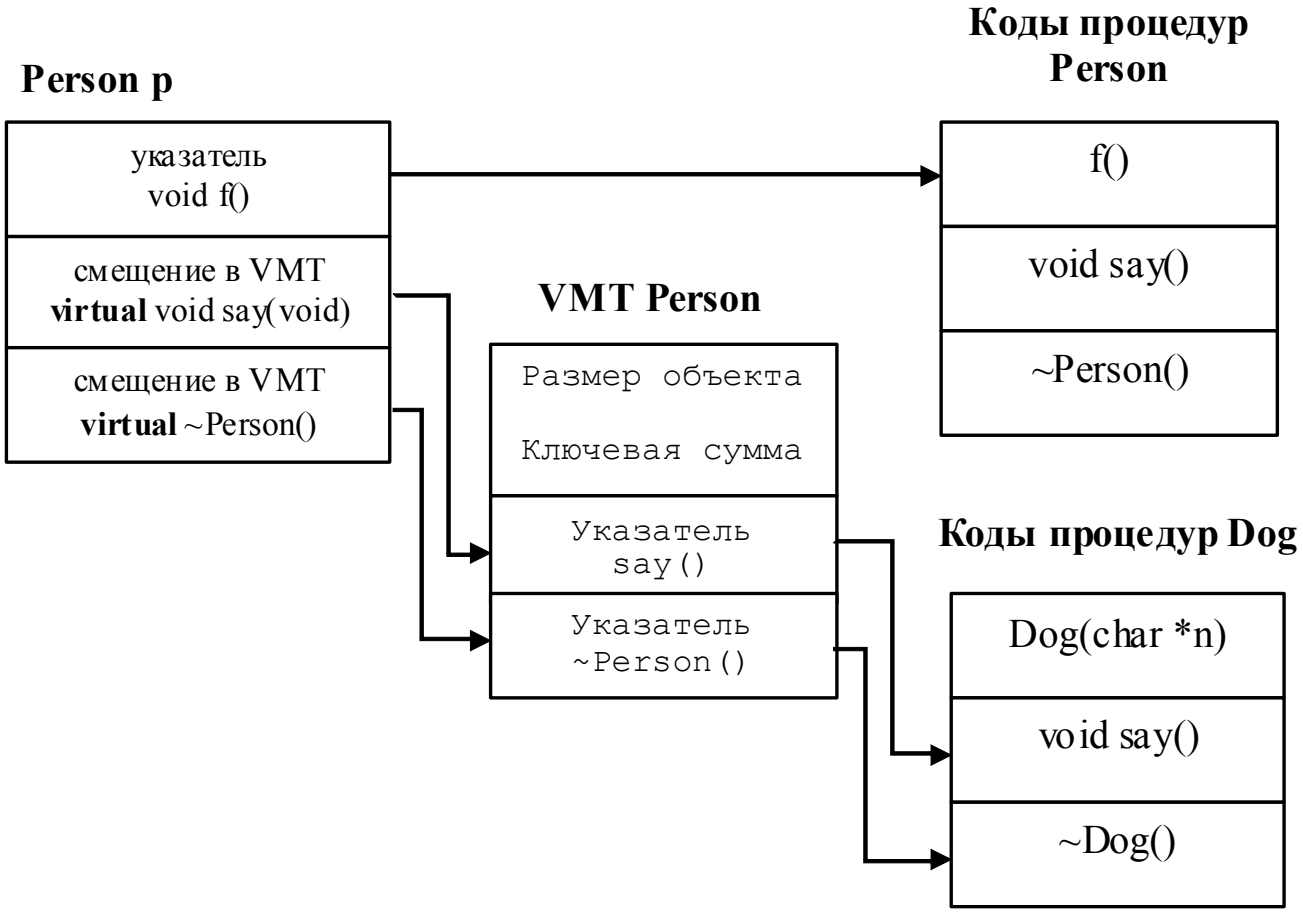
- **Меньшему можно присвоить большее, но не наоборот**.
- Обоснование: природа (компилятор) не терпит неопределенности:



"Маленькие" и "большие" объекты

Person *p;

p = new Dog("Pif");



Виртуальность

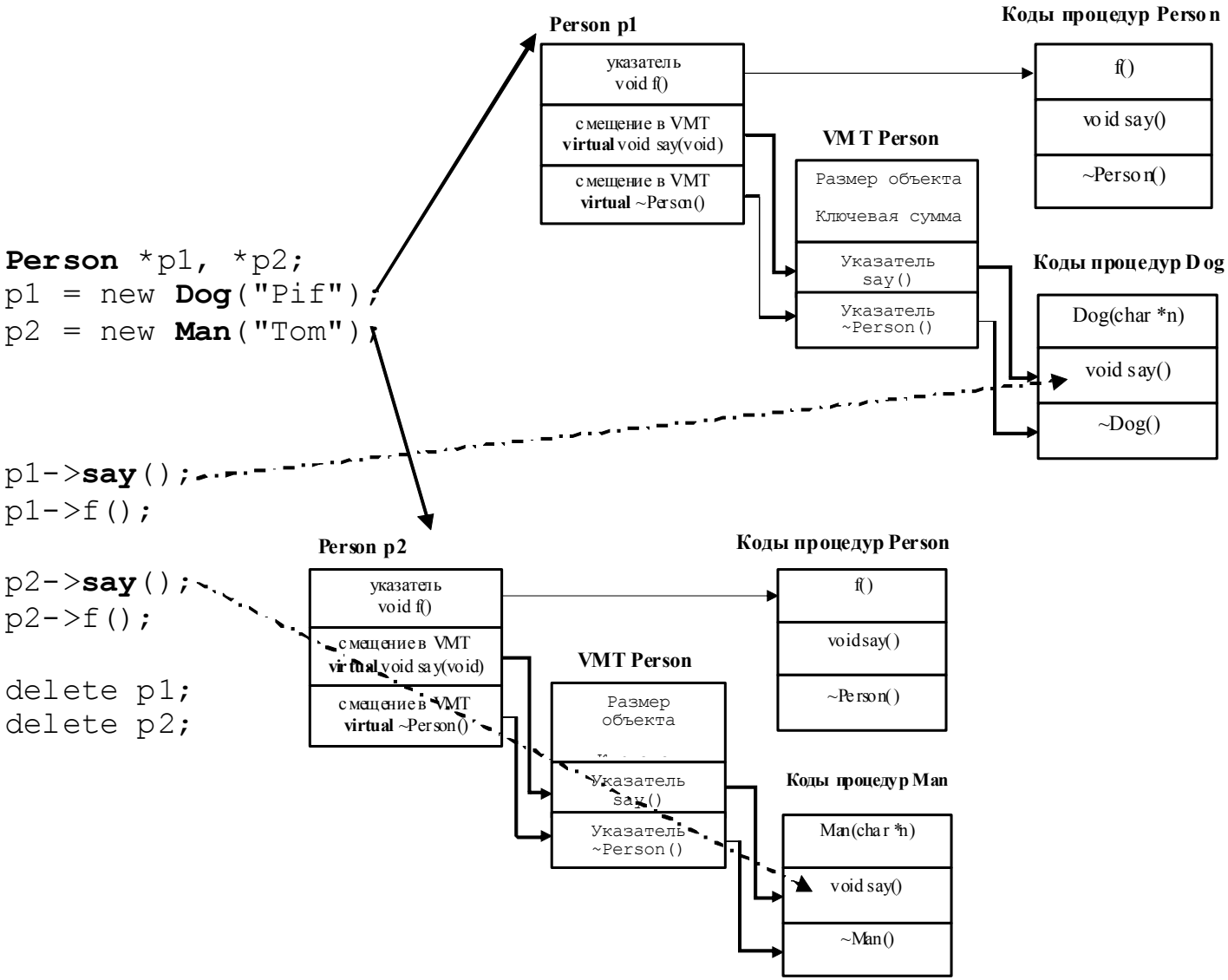
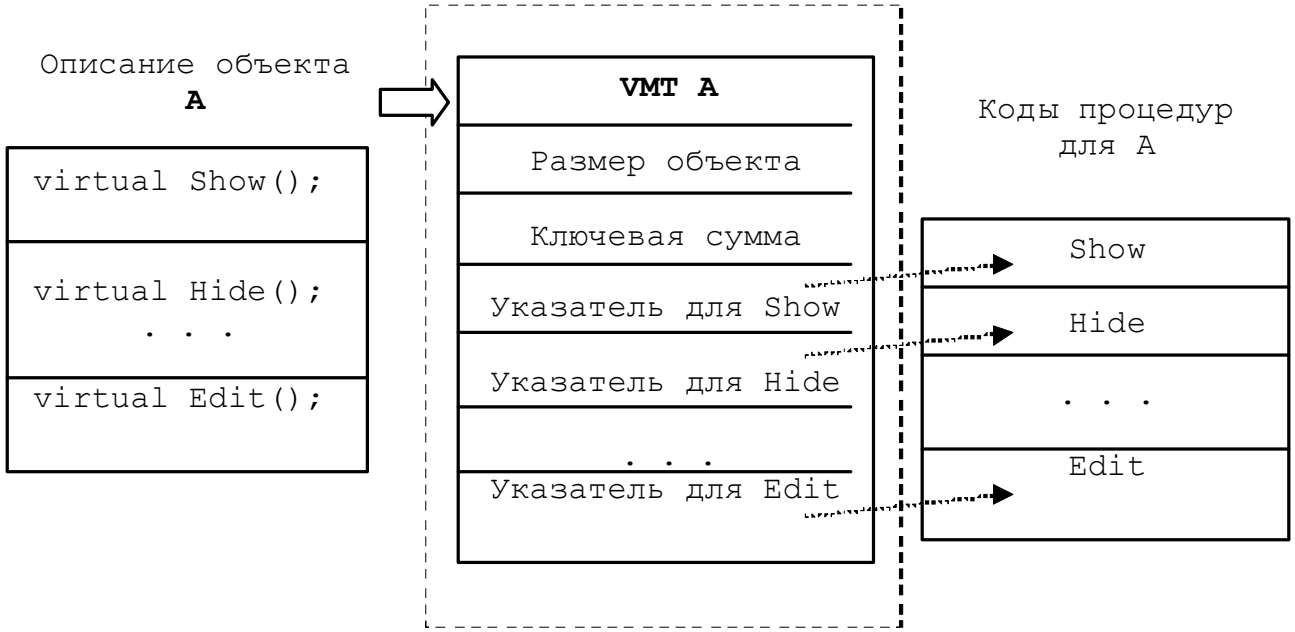


Таблица виртуальных методов

- На этапе компиляции программы в том случае, если в описании методов объекта встречается слово *virtual*, для данного объекта будет автоматически создана структура, называемая *Таблицей Виртуальных Методов* или VMT (Virtual Methods Table).
- Эта таблица размещается в инициализированной части сегмента данных и выглядит так:

```
class A
{
  virtual void Show();
  virtual void Hide();
  virtual void Edit();
};
```

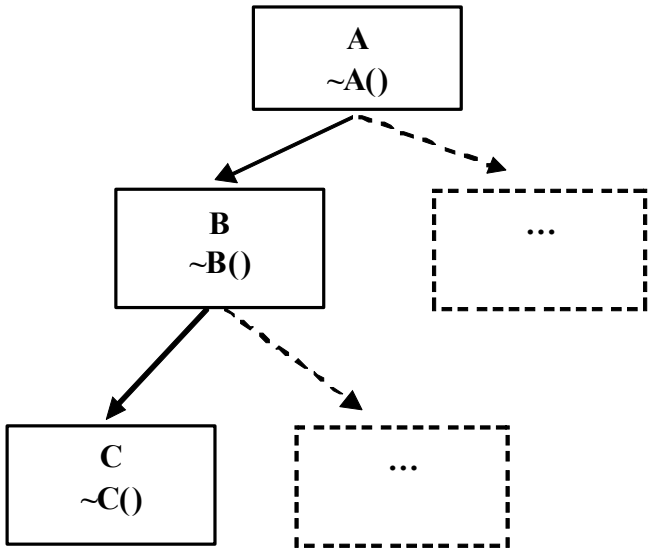


Конструктор

- Конструктор может иметь столько параметров, сколько необходимо.
- Конструктор может выполнять различные полезные действия, однако все это – побочный эффект.
- Главное предназначение конструктора заключается в том, что при его вызове происходит **инициализация адресов в VMT**, т.е. происходит связывание методов с реализующими процедурами (позднее связывание).
- В C++ конструктор есть всегда (конструктор по умолчанию)

Еще раз о деструкторах

```
struct Person
{
  …
  virtual ~Person() { printf("~Person.\n"); }
};
struct Dog: public Person
{
  …
  virtual ~Dog() { printf("~Dog.\n"); }
};
```




```
Person *p = new Dog("Pif");
…
delete p;
```

```
C *c;
…
delete c;
```
~C() → ~B()→ ~A()

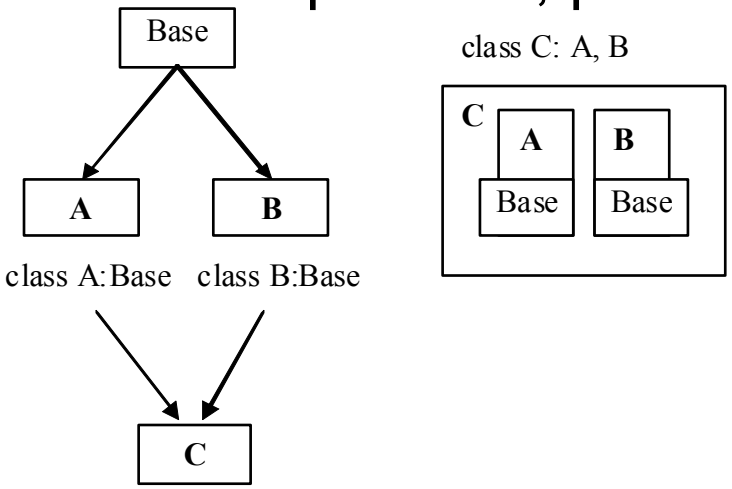
Сначала сработает деструктор ~Dog, а затем – деструктор ~Person.

Это – общее правило вызовов деструкторов в иерархии классов.

Виртуальные базовые классы

- Проблема множественного наследования - это дублирование наследуемых методов и переменных.

class C: public A, public B



Экземпляр C будет содержать 2 экземпляра класса Base, унаследованных от обоих родителей.

Виртуальные родители

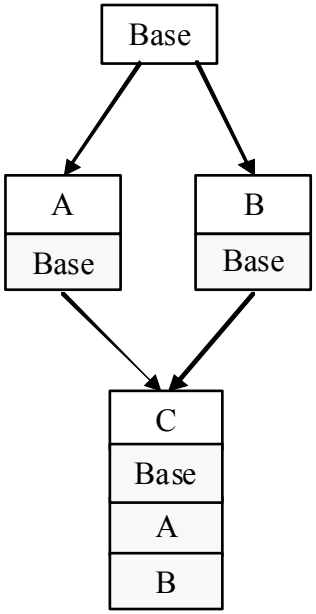
Чтобы избавиться от подобного дублирования, следует объявить классы A и B следующим образом:

Нахождение подобного рода неопределенностей – это задача компилятора.

class A: virtual public Base

class B: virtual public Base

Тогда в C будет содержаться только один экземпляр класса Base.



Class C: public A, public B {…}

Абстрактные классы

```
class A
{
    virtual void f() = 0;
}
```

"*virtual void f()=0;*" - это объявление **чисто виртуальной функции**. Это делается во избежание возможных ошибок, чтоб не забыть переопределить метод в дальнейшем (в Smalltalk'е описывается как «определено в подклассе»).

Задача компилятора - выдать сообщение об ошибке при обращении к чисто виртуальной функции.

```
A a;
a.f(); // Ошибка
```

Потоки

Типо-безопасная (сохраняющая тип) и единообразная обработка операций ввода–вывода может быть осуществлена с помощью одного имени совмещенной функции.

Например:

put(int)

put(char)

put(char)*

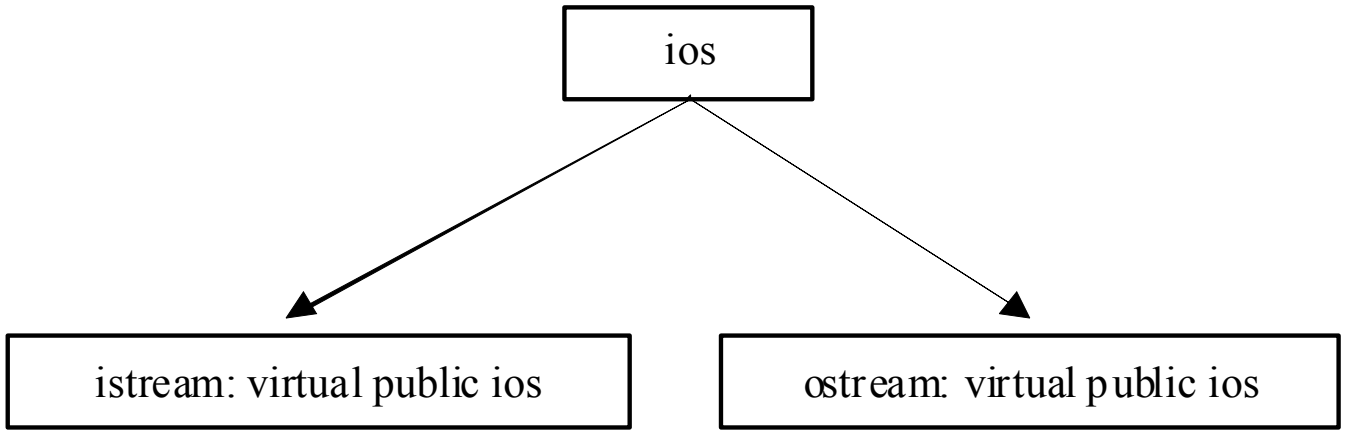
…

При этом полезно бывает использовать форматированный вывод, реализуемый, например, с помощью функции *form()*

*char *form(char *format,...)*

Ввод-вывод в C++

- Класс *ios* содержит базовые функции ввода–вывода низкого уровня.
- Класс *ios* большой, аппаратно–зависимый и потому некрасивый. С ним напрямую работать крайне тяжело.
- Класс *ios* является базовым классом для более удобных, интерфейсных классов – классов *istream* и *ostream*. Эти классы являются надстройкой класса *ios.*



istream

Служит для удобства ввода, реализует переопределение оператора '>>'. В нем можно найти функции почти "на все случаи жизни":

```
class istream : virtual public ios {
public:
    istream& operator>> (istream& (*_f)(istream&));
    istream& operator>> (ios& (*_f)(ios&) );
    istream& operator>> (unsigned char&);
    istream& operator>> (int&);
    istream& operator>> (long&);
    …
    istream& operator>> (float&);
    istream& operator>> (long double&);
}
```

ostream

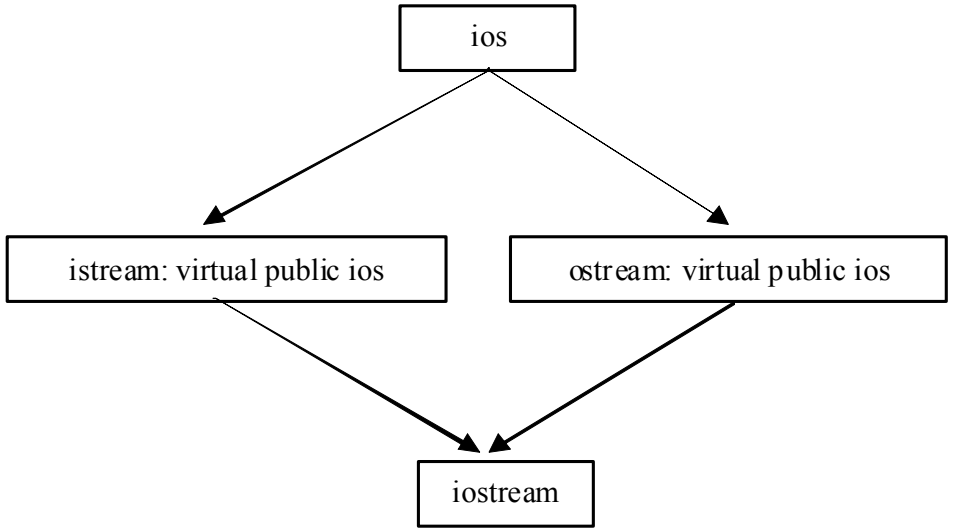
Устроен аналогично, реализует операции вывода:

```
class ostream : virtual public ios {
public:
   ostream& operator<< ( signed char);
   …
   ostream& operator<< (int);
   ostream& operator<< (double);
}
```

iostream

iostream - обобщенный класс ввода–вывода:

```
class iostream : public istream, public ostream {
    public:
        iostream(streambuf _FAR *);
        virtual ~iostream();
    protected:
        iostream();
}
```



istream_withassign и ostream_withassign

Эти классы вводят лишь переопределение оператора присваивания (=) для установки связей потоков с другими потоками и буферами:

```
class istream_withassign : public istream {
public:
    istream_withassign();
    virtual ~istream_withassign();
    // gets buffer from istream and does entire initialization
    istream_withassign& operator= (istream&);
    // associates streambuf with stream and does entire initialization
    istream_withassign& operator= (streambuf _FAR *);
}
class ostream_withassign : public ostream {
public:
    ostream_withassign();
    virtual ~ostream_withassign();
    // gets buffer from istream and does entire initialization
    ostream_withassign& operator= (ostream&);
    // associates streambuf with stream and does entire initialization
    ostream_withassign& operator= (streambuf _FAR *);
}
```

cin, cout и cerr

```
extern istream_withassign       cin;
extern ostream_withassign       cout;
extern ostream_withassign       cerr;
```

Для нового типа (например, *complex*), оператор << можно переопределить:

```
    ostream & operator<<(ostream &s, complex c)
    { return s<<"("<<c.re<<','<<c.im.<<")"; }
```

Инициализация и закрытие потоков.

```
cout.flush()
```

Состояние потока

```
enum stream_state {_good,_eof,_fail,_bad}
switch (cin.rdstate())
{   case _good:...
    case _eof:...
    case _fail:...
    case _bad:...
}
while (cin>>z) cout << z<<"\n";
```