

# Объектно- ориентированное программирование

*Введение*



# Парадигмы программирования

- Структурное программирование
- Функциональное программирование
- Логическое программирование
- Автоматное программирование
- Объектно-ориентированное программирование
- Событийно-ориентированное программирование
- Агентно-ориентированное программирование



# Объекто-ориентированное программирование

1. Использует в качестве основных логических конструктивных элементов объекты, а не алгоритмы
2. Каждый объект является экземпляром (instance) определенного класса (class);
3. Классы образуют иерархии

# Элементы объектной модели

- Абстракция
- Инкапсуляция
- Модульность
- Иерархия
- Контроль типов
- Параллелизм
- Персистентность





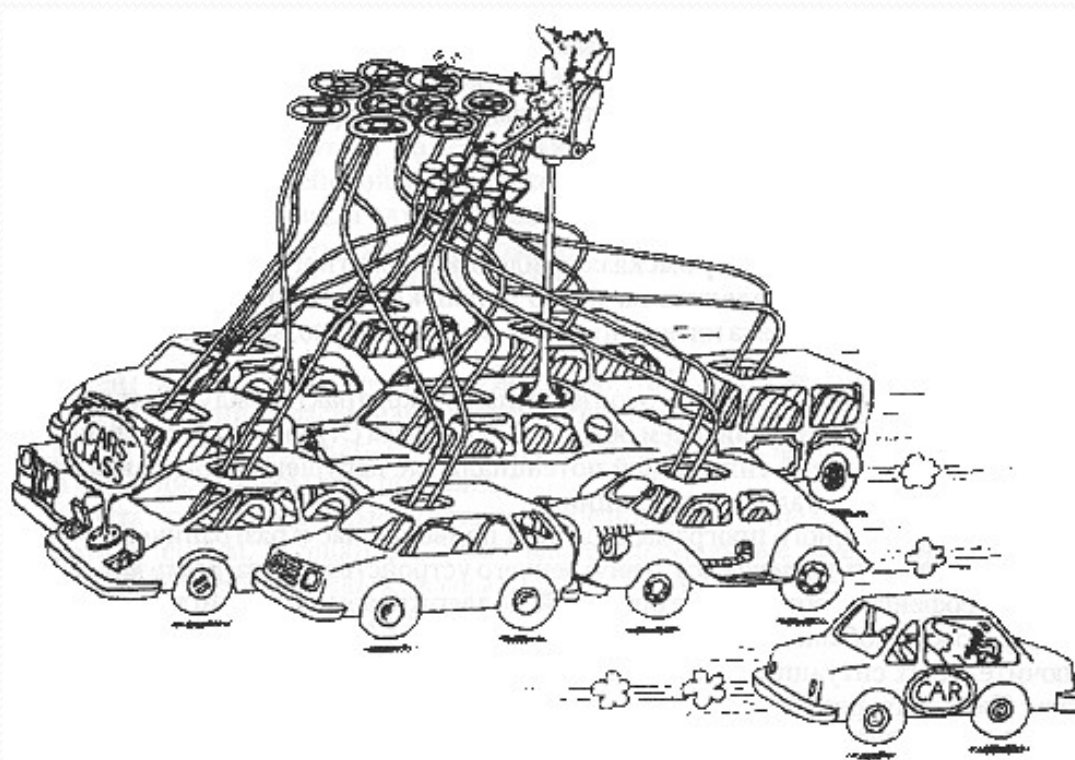
# Преимущества объектной модели

- стимулирует повторное использование не только кода, но и проектных решений
- приводит к созданию систем с устойчивыми промежуточными формами, что упрощает их изменение.
- уменьшает риски, связанные с проектированием сложных систем.
- учитывает особенности процесса познания.





Класс - это множество объектов, имеющих общую структуру и общее поведение.

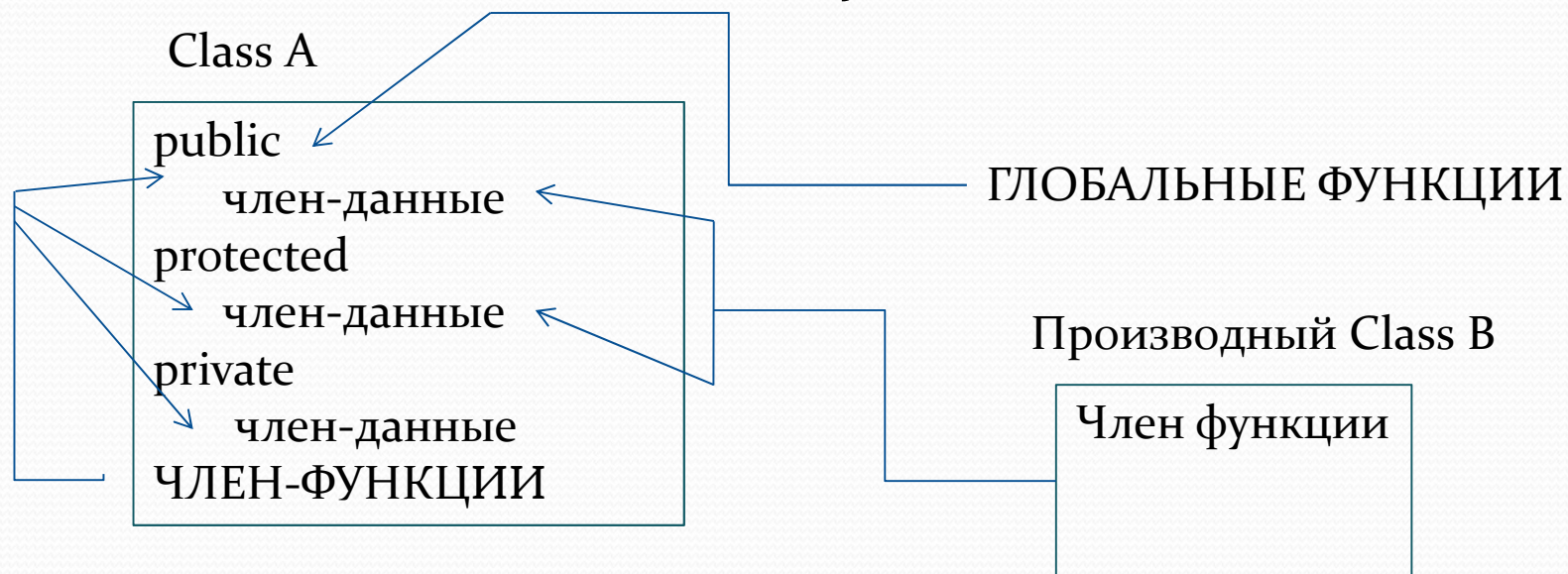


struct Person

```
{char *Fam_name[25];  
int age;  
void set_name(char*);  
void set_age(int);  
} person1, person2;
```

class PERSON

```
{ private:  
char *Fam_name[25];  
int age;  
public:  
void set_name(char*);  
void set_age(int);  
};
```





## Ограничения доступа

Private	Public	Protected
Член-данные и член-функции доступны только через член-функции данного класса.	Член-данные и член-функции доступны из любого места программы, где имеется представитель класса.	Член-данные и член-функции доступны только через член-функции данного класса и его потомков.

## Пример использования класса

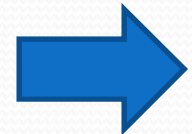
```
class date
{private:
int month, day, year;
public:
void set(int, int, int);
void out();
date(int, int, int);
~date()
} today;
date::date(int m=0, int d=1, int
{month = m;
day = d;
year = y;
}
```



```
data::set(int m, int d, int y)
{
    month=m;
    day=d;
    year=y;
}
data::out()
{cout << day << " " << month
    << " " << year;
}
```

```
y=1998)
```

```
today.set(4,4,2011);
today.out();
```





# Основные свойства и правила использования конструкторов:

- конструктор имеет то же имя, что и класс, в котором он объявляется;
- конструктор не возвращает значения (даже типа `void`);
- конструктор не наследуется в производных классах.
- конструктор может иметь параметры, заданные по умолчанию;
- конструктор - это функция, но его нельзя объявить с ключевым словом `virtual`;
- невозможно получить в программе адрес конструктора;
- если конструктор не задан в программе, то он будет автоматически сгенерирован;
- конструктор вызывается автоматически только при описании объекта;
- объект, содержащий конструктор, нельзя включить в виде компонента в объединение;
- конструктор класса `X` не может иметь параметр типа `X`, может иметь параметр ссылку на объект типа `X`, в этом случае он называется конструктором для копирования (`copy constructor`) класса `X`.

# Основные свойства и правила использования деструкторов:

- деструктор имеет то же самое имя, что и класс, в котором он объявляется, с префиксом ~ (тильдой);
- деструктор не возвращает значения ;
- деструктор не наследуется в производных классах;
- производный класс может вызвать деструкторы для его базовых классов;
- деструктор не имеет параметров;
- класс может иметь только один деструктор;
- деструктор - это функция, и он может быть виртуальным;
- невозможно получить в программе адрес деструктор);
- если деструктор не задан в программе, то он будет автоматически сгенерирован компилятором;
- деструктор можно вызвать так же, как обычную функцию, например:  

```
date *my_day;  
my_day->date::~~date().
```
- деструктор вызывается автоматически при разрушении объекта.





# Области видимости для классов

```
int x = 2;  
class Example  
{  
    void f () {x = 0;}  
        short x  
};  
void f () {::x = 0;}
```

# Спецификатор памяти static

```
class Example
{   public:
    static int x ;
} p1, p2;
int Example::x=67;
```



# Спецификатор const

```
class Stack {  
    char s[MaxSize];  
    int top;  
public:  
    Stack () {top = 0;}  
    void Look_Top() const {cout << s[top];}  
    void push() {top++;}  
};  
const Stack s1;  
Stack s2;  
  
s2.Look_Top();  
s1.push();  
s1.Look_Top();
```



# Указатель this

```
date today, my_day;  
today.out();  
my_day.out();  
cout << day << " " << month;
```

```
Date *const this;  
cout << this>day << " " << this>month;
```



## Организация списка

```
class spisok {  
    int value;  
    char * str;  
    spisok * previous;  
    spisok * next;  
public:  
    static spisok * first;  
    spisok(int, char*);  
    void Look();  
    spisok * Get_Next();  
};  
spisok * spisok::first = 0;
```

```
spisok::spisok(int val, char *  
strok) : str(strok)  
{  
    value=val;  
    if (first == 0)  
    { first = this;  
      previous = 0;  
      next = 0;  
    }  
    else  
    { first->previous = this;  
      this->next = first;  
      previous = 0;  
      first = this;  
    }  
}
```

```
void spisok::Look()
{
    cout << value << "\n" << str << "\n";
}
spisok * spisok::Get_Next()
{ return next;
}
main()
{
    spisok * p;
    p = new spisok(1,"stroka1");
    p = new spisok(2,"stroka2");
    p = new spisok(3,"stroka3");
    p=spisok::first;
    while (p!=0)
    {
        p->Look();
        p=p->Get_Next();
    }
}
```



# Основные свойства и правила использования указателя `this`:

- каждый новый объект имеет свой скрытый указатель `this`;
- `this` указывает на начало своего объекта в памяти;
- `this` не надо дополнительно объявлять;
- `this` передается как скрытый параметр во все нестатические член-функции своего объекта;
- `this` - это локальная переменная, которая недоступна за пределами объекта.

# Дружественные функции

```
class rectangle  
{int color, x, y;  
};
```

```
class circle
```

```
{int color, x, y, radius;
```

```
public:
```

```
friend bool equal-color(circle c, rectangle r);
```

```
};
```

```
bool equal-color (circle c, rectangle r)
```

```
{ if(c.color == r.color) return true;
```

```
else return false;
```

```
}
```

Circle

Private  
color

Rectangle

Private  
color

Equal-color

```
graph TD; Circle[Circle] --> Equal-color[Equal-color]; Rectangle[Rectangle] --> Equal-color;
```



Член-функция одного класса может быть объявлена со спецификатором friend для другого класса.

```
class X {  
    int function_of_X(...);  
};  
class Y {  
    friend int X::function_of_X(...);  
};  
class Z {  
    friend class Y;  
};
```

## Сравнить компоненты объектов разных классов, имеющие атрибут private

```
class my_class2;
class my_class1
{   int a;
    friend void fun(my_class1&,my_class2&);
public:
    my_class1(int A) : a(A) {};
};
class my_class2
{   int a;
    friend void fun(my_class1&,my_class2&);
public:
    my_class2(int A) : a(A) {};
};
void fun(my_class1& M1,my_class2& M2)
{   if (M1.a == M2.a) cout << "equal\n";
    else cout << "not equal\n";
}
void main(void)
{   my_class1 mc1(100);
    my_class2 mc2( 100);
    fun(mc1,mc2);}
```



## Функция одного класса со спецификатором friend для другого класса.

```
class Y;
class X
{   int a;
    void Y(int c): a(c){};
public:
    void display(X* pX);
};
class Y
{   int a;
    void X(int C): a(C){};
public:
    friend void X::display(X*);
};
void X::display(X* pX)
{ cout<< pX->a << '\t' << a << endl; }
void main(void)
{ X my_X(100);
  Y my_Y(200);
  my_Y.display(&my_X);    // Результат: 100 200
}
```

## Основные свойства и правила использования спецификатора friend:

- friend функции не являются компонентами класса, но получают доступ ко всем его компонентам;
- если friend функции одного класса не являются компонентами другого класса, то они вызываются так же, как и обычные глобальные функции (без операторов . и ->);
- если friend функции одного класса не являются компонентами другого класса, то они не имеют указателя this;
- friend функции не наследуются в производных классах;
- отношение friend не является транзитивным.



## Объявление и разрушение глобальных объектов:

```
class A
{ int i;
public:
    A(int I) : i(I)
        { cout << "class A" << i << " constructor\n"; }
    ~A()
        { cout << "class A" << i << " destructor\n"; }
};
A a1(1), a2(2);
void main(void)
{ getch(); }
```

Результаты выполнения этой программы:

class A 1 constructor

class A2 constructor

< здесь можно нажать любую клавишу >

class A2 destructor

class A 1 destructor

# Объявление и разрушение локальных объектов.

```
class A
{ int i;
  public:
  A(int I) : i(I)  { cout << "class A" << i << " constructor\n"; }
  ~A()            { cout << "class A" << i << " destructor\n"; }
};

void function(void)
{ cout<< "begin\n";
  A a1(1),a2(2);
  cout <<"end\n"; }

void main(void)
{ cout<< "before\n";
  function();
  cout << "after\n"; } }
```

Результаты:

```
before
begin
class A 1 constructor
class A2 constructor
end
class A2 destructor
class A 1 destructor
after
```



# Создание объектов в динамически выделяемой памяти

```
class A
{ int i;
public:
  A(int I) : i(I) { cout << "class A" << i << " constructor\n"; }
  ~A () { cout << "class A" << i << " destructor\n"; }
};

void main(void)
{ A *p1 == new A(1);
  A *p2 == new A(2);
  delete p1;
  delete p2; }
```

Результаты :

```
class A 1 constructor
class A2 constructor
class A 1 destructor
class A2 destructor
```

## Объявление объектов в виде компонентов в других классах.

```
class a
{
    int j;
public:
    a(int J) : j(J) { cout << "class a" << j << " constructor\n"; }
    ~a() { cout << "class a" << j << " destructor\n"; }
};

class b
{
    int i;
    a a1;
public:
    b(int I,int J) : a1 (J),i(I) { cout << "class b" << i << "constructor\n"; }
    ~b() { cout << "class b" << i << "destructor\n"; }
};

void main(void)
{
    b a1(1, 1);
    b a2(2,1);}
}
```

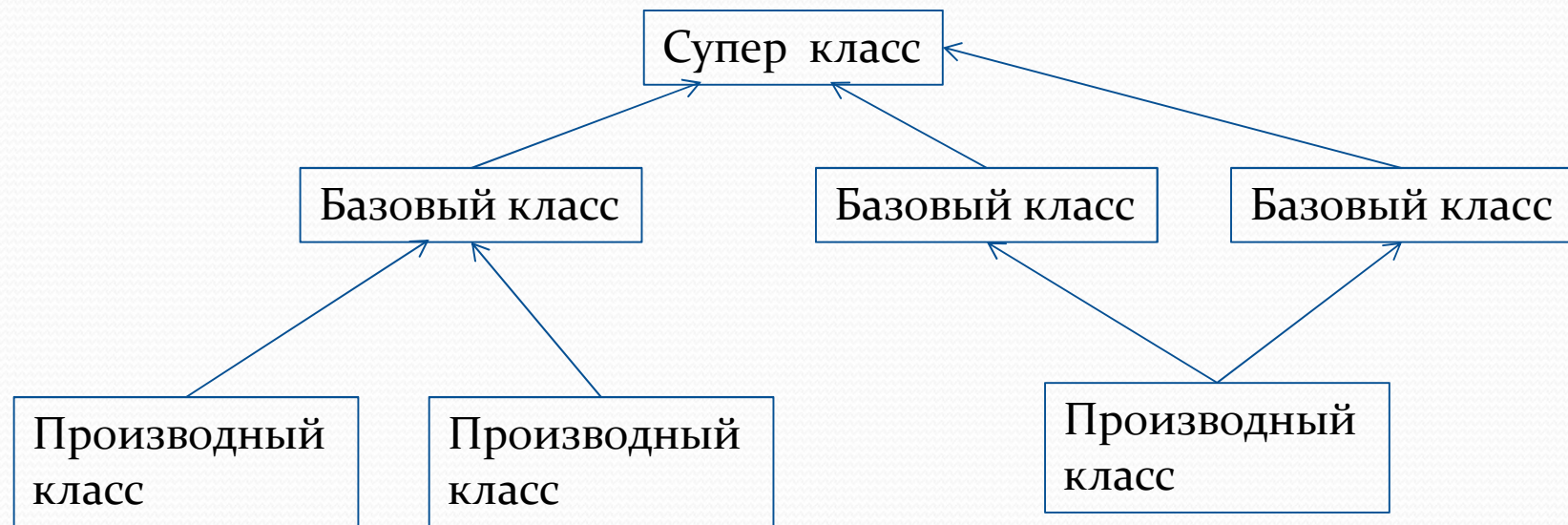
Результаты:

```
class a1 constructor
class b1 constructor
class a1 constructor
class b2 constructor
class b2 destructor
class a1 destructor
class b1 destructor
class a1 destructor
```



# Наследование

Организация связи между абстрактными типами данных, при которой имеется возможность на основании существующих типов данных порождать новые типы



Простое наследование

Множественное наследование



```
class employee
```

```
{ char * name;    // имя
```

```
  int income;     // доход
```

```
  employee * next; // следующий служащий
```

```
public:
```

```
  employee (char * n, int i); // конструктор
```

```
  void print() const; // вывод на экран
```

```
};
```

```
employee::employee(char * n, int i) : name(n), income(i)
```

```
{ next = 0;}
```

```
void employee::print() const
```

```
{ cout << name << "\n";}
```



```
class manager : public employee
{ int level;      // уровень
  employee * group; // подчиненные
public:
  manager(char *, int, int, employee *);
  void print() const;
};

void manager::print() const
{ employee::print();
  cout << "руководит : ";
  group->print();}

manager::manager(char* n, int i,int l, employee * g):
    employee(n,i), level(l), group(g)
```

```
void main()  
{ employee person (“ИВАНОВ”,20);  
  manager one_more(“Петров”,40,1,&person);  
  person.print();  
  one_more.print();  
}
```



# Основные правила использования базовых и производных классов:

Пусть функция  $F$  принадлежит базовому классу  $B$ . Тогда в производном классе  $P$  можно:

- 1) полностью заменить функцию  $F$  (старая  $B::F$  и новая  $P::F$ );
- 2) доопределить (частично изменить) функцию  $F$ ;
- 3) использовать функцию  $B::F$  без изменения.

- Если объявить указатель  $pB$  на базовый класс, то ему можно присвоить значение указателя на объект производного класса;

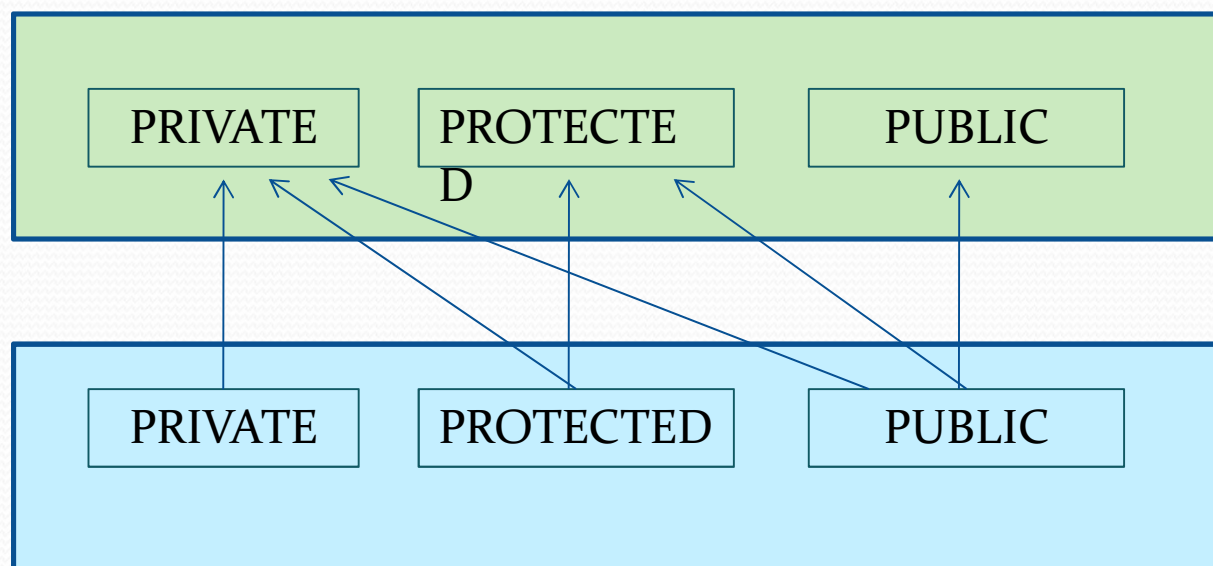
- указателю  $pP$  на производный класс нельзя присвоить значение указателя на объект базового класса;

- регулирование доступа к компонентам базового и производного классов осуществляется с помощью атрибутов `private`, `public` и `protected`;

- производный класс может быть в свою очередь базовым. Множество классов, связанных отношением наследования базовый - производный, называется иерархией классов.


# Наследование атрибутов компонентов базового класса:

наследник



родитель





```
class base
{protected:
    int x;
public:
    char * str;
    void f(int, char*);
};
```

```
class generate : private base
{
protected:
    int base::x;
public:
    char * base::str;
    base::f;
...}
```

# Множественное наследование

```
class base1
{public:
int field;
char * str;
};
class base2
{public:
int field;
int data;
};
class generate :public base1, public base2
{...};
```

```
Generate ex;
ex.field = 4;
ex.base1::field = 4;
```

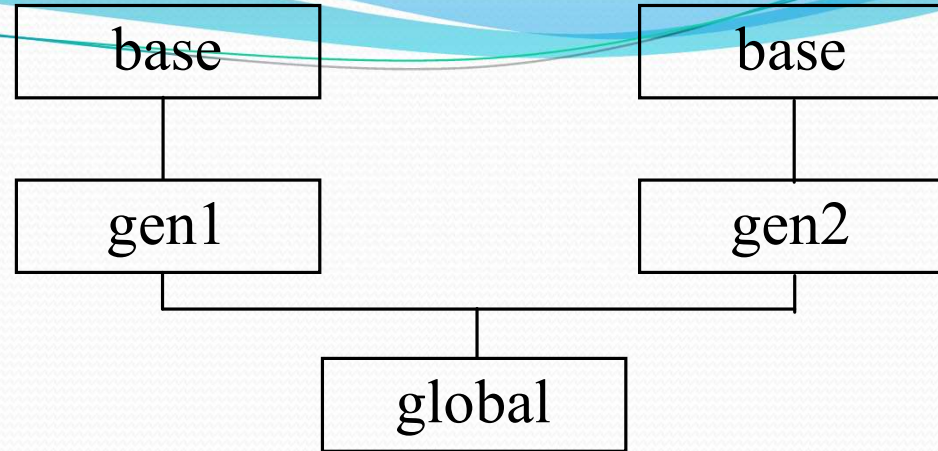
Часть унаследованная от класса base1
Часть унаследованная от класса base2
Собственная часть класса generate



```

class base
{public:
int field;
char * str;
};
class gen1 : public base
{public:
float s;
};
class gen2 : public base
{public:
char * name;
};
class global : public gen1, public gen2
{...} ex;

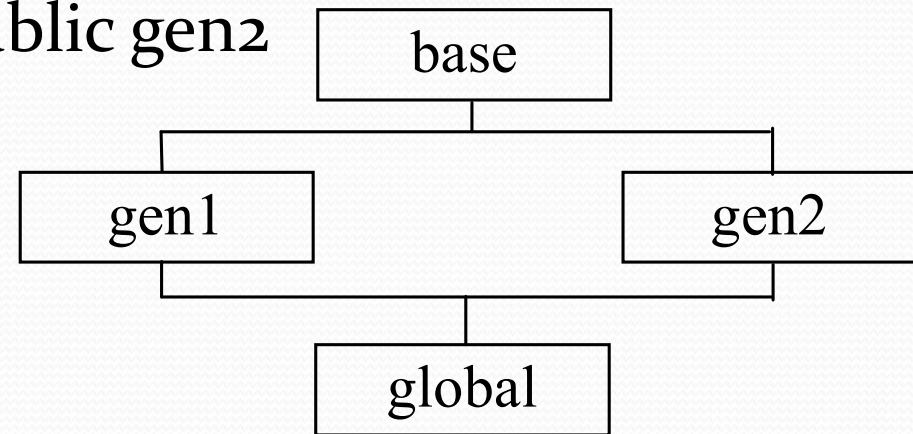
```



Часть унаследованная от класса base
Часть унаследованная от класса gen1
Часть унаследованная от класса base
Часть унаследованная от класса gen2
Собственная часть класса global

## Использование виртуального класса

```
class gen1 : virtual public base{...};  
class gen2 : virtual public base{...};  
class global : public gen1, public gen2  
{...} ex;
```



Формально конструктор класса global будет иметь вид:

```
global::global() : base(), gen1(), gen2() {...}
```



# Полиморфизм

```
class Base
{ public:
virtual int f(const int &d) ;
int CallFunction(const int &d)
    { return f(d)+1;    }
};
class Derived: public Base
{ public:
virtual int f(const int &d)
    { return d*d; }
};
int main()
{   Base a;
    cout << a.CallFunction(5)<< endl;
    Derived b;
    cout << b.CallFunction(5)<< endl;
    return();
}
```

```
class Clock
{ public:
    virtual void print() const { cout << "Clock!" << endl; }
};
class Alarm: public Clock
{ public:
    virtual void print() const { cout << "Alarm!" << endl; }
};
void settime(Clock &d)
{ d.print(); }
Clock W;
settime(W);
Alarm U;
settime(U);
Clock *c1 = &W;
c1->print();
c1 = &U;
c1->print();    }
```

```
((Alarm *)c1)->print();
```



## Правила описания и использования виртуальных функций:

1. Виртуальная функция может быть только методом класса.
2. Любую перегружаемую операцию-метод класса можно сделать виртуальной.
3. Виртуальная функция, как и сама виртуальность, наследуется.
4. Виртуальная функция может быть константной.
5. Если в базовом классе впервые объявлена виртуальная, то функция должна быть либо чистой (`virtual int f(void) = 0;`), либо для нее должно быть задано определение.
6. Если в базовом классе определена виртуальная функция, то метод производного класса с такими же именем и прототипом автоматически является виртуальным.
7. Конструкторы не могут быть виртуальными.
8. Статические методы не могут быть виртуальными.
9. Деструкторы могут (чаще — должны) быть.
10. Если некоторая функция вызывается с использованием ее полного имени, то виртуальный механизм игнорируется.

## Вызов виртуальной функции может не являться виртуальным в некоторых случаях:

- Вызывается не через указатель или ссылку:

global object;

object.f();

- Вызывается через указатель или ссылку, но с уточнением имени класса:

base \* p;

global object;

p = &object;

p->f();               // виртуальный вызов

p->global::f();       // не виртуальный вызов

- вызывается в конструкторе или деструкторе базового класса.