ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

MEMBER INITIALIZER LIST

```
//complex.h
class Complex{
    double re, im;
public:
    Complex();
    Complex(double re, double im);
    ...
};

//complex.cpp
Complex::Complex(){
    re = 0.;
    im = 0.;
}

Complex::Complex(double re, double im){
    this->re = re;
    this->im = im;
}
```

copy initialization

MEMBER INITIALIZER LIST

```
//complex.h
class Complex{
    double re, im;
public:
    Complex();
    Complex(double re, double im);
    ...
};
```

initializer list

```
//complex.cpp
Complex::Complex() : re(0.), im(0.){
}

Complex::Complex(double re, double im)
    : re(re), im(im) {
}
```

direct initialization

MEMBER INITIALIZER LIST

```
//complex.h
class Complex{
    double re, im;
public:
    Complex();
    Complex(double re, double im);
    ...
};
```

initializer list

```
//complex.cpp
Complex::Complex() : re{ 0. }, im{ 0. }{
}

Complex::Complex(double re, double im)
    : re{ re }, im{ im } {
}
```

uniform initialization
(since C++11)

DELEGATING CONSTRUCTOR

```
//complex.h
class Complex{
    double re, im;
public:
    Complex();
    Complex(double re, double im);
    ...
};
```

initializer list

```
//complex.cpp
Complex::Complex() : Complex(0., 0.){
}

Complex::Complex(double re, double im)
    : re{ re }, im{ im } {
}
```

Complex() delegates to Complex(double, double)

MEMBER INITIALIZER LIST

```
//complex.cpp
Complex::Complex() : im(0.), re(im) {
}
```

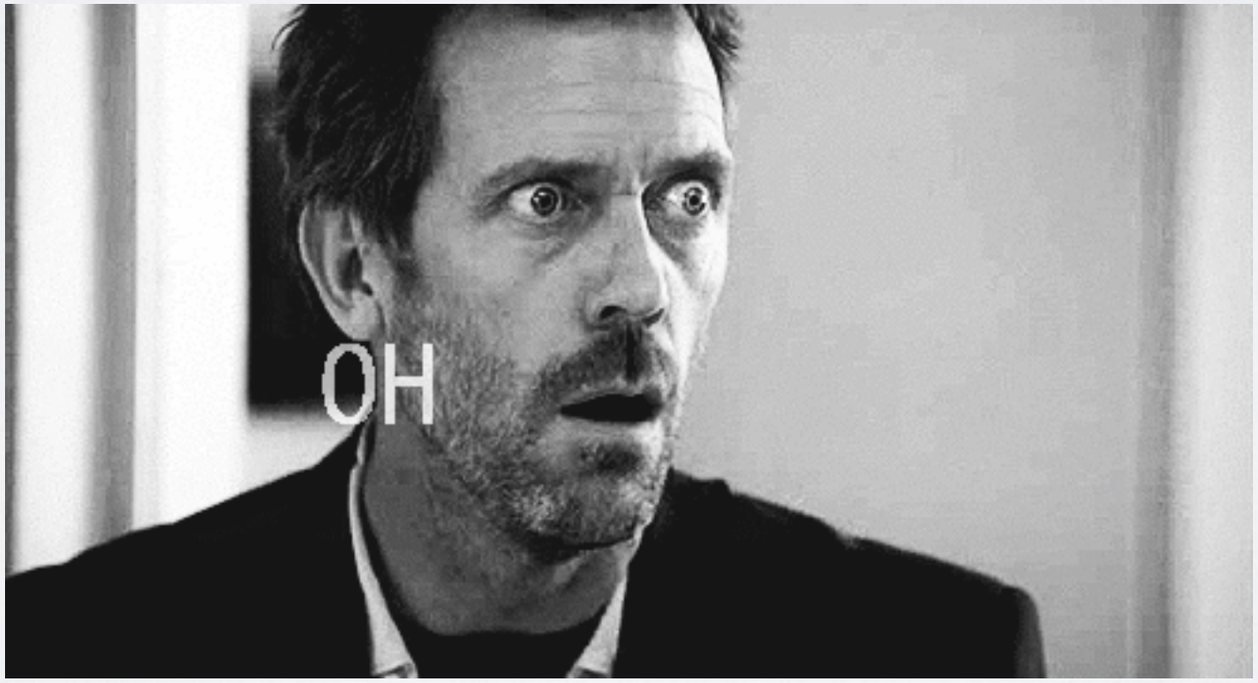
```
re = ???;
im = ???;
```

MEMBER INITIALIZER LIST

```
//complex.cpp
Complex::Complex() : im(0.), re(im) {
}
```

re = -9.25596e+61;

im = 0;



INITIALIZATION ORDER

1. Инициализация виртуальных базовых классов.
2. Инициализация прямых базовых классов.

3. Инициализация нестатических полей в порядке их объявления в классе.
4. Выполнение тела конструктора.

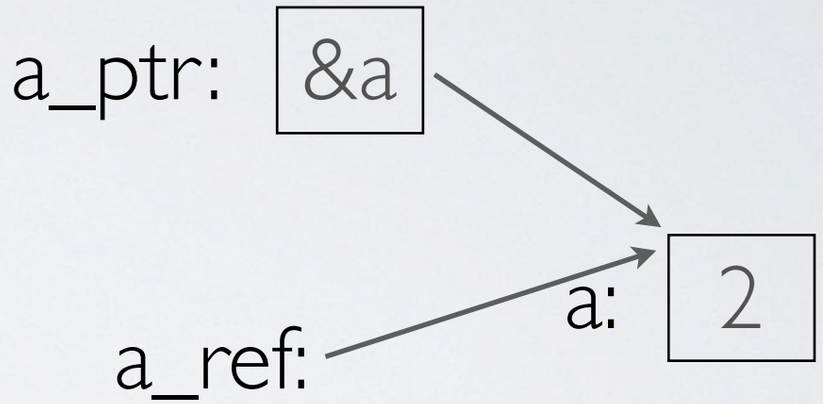
REFERENCE

```
//reference declaration
Type& nameRef = some_object;
const Type& nameRef = some_object;
```

Ссылка - это псевдоним уже существующего объекта (альтернативное имя объекта).

REFERENCE

```
int a = 2;
int &a_ref = a;
int *a_ref = &a;
```




```
a_ref = 3;
printf("%d\n", a);      // 3
```

Ссылку можно воспринимать как константный указатель, который разыменуется (неявно) при использовании.

REFERENCE

```
void swap(int p, int q) {
    int r = p;
    p = q;
    q = r;
}
```

Idiot-style

```
void swap(int *p, int *q) {
    int r = *p;
    *p = *q;
    *q = r;
}
```

C-style

```
void swap(int &p, int &q) {
    int r = p;
    p = q;
    q = r;
}
```

C++-style

REFERENCE

```
//stack.h
class Stack{
    size_t size;
    /*...*/
public:
    Stack();
    ~Stack();

    void clear();
    void push(int node);
    void pop();

    int& top();
    const int& top() const;

    ...
};
```

REFERENCE

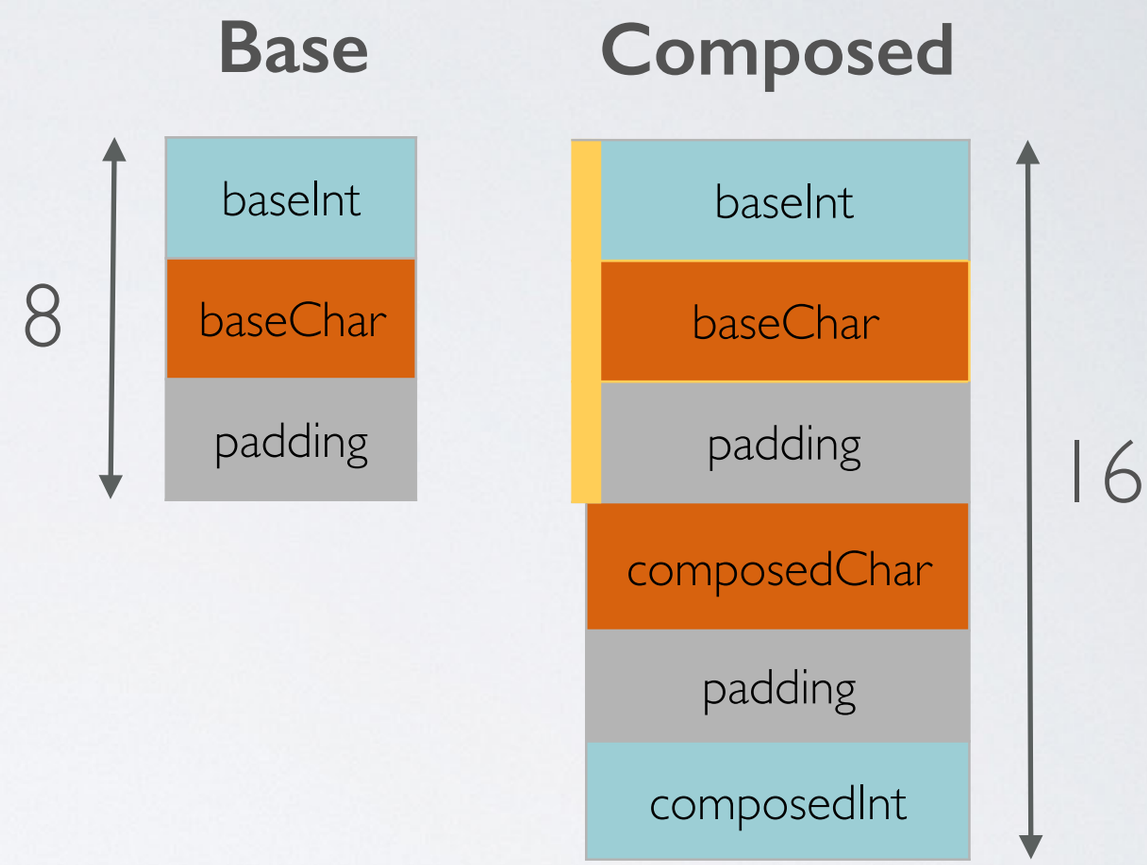
```
int& ref;       //error: ссылка должна быть
                //       проинициализирована

int& *ref;      //error: нельзя создать указатель
                //       на ссылку.

int& &ref;      //error: нельзя создать ссылку
                //       на ссылку.

int& ref = 1; //error: нельзя создать не константную
              //       lvalue-ссылку на временный объект.

int a;
int* &ref = &a;      //ok: ссылка на указатель

const int& ref = 1; //ok: константная ссылка
                    //    на временный объект.
```

REFERENCE VS POINTER

1. Нельзя иметь пустые (NULL) ссылки. Указатели можно.
2. Нельзя переинициализировать ссылку. Указатели могут указывать на другие объекты.
3. Ссылка обязательно должная быть инициализирована.

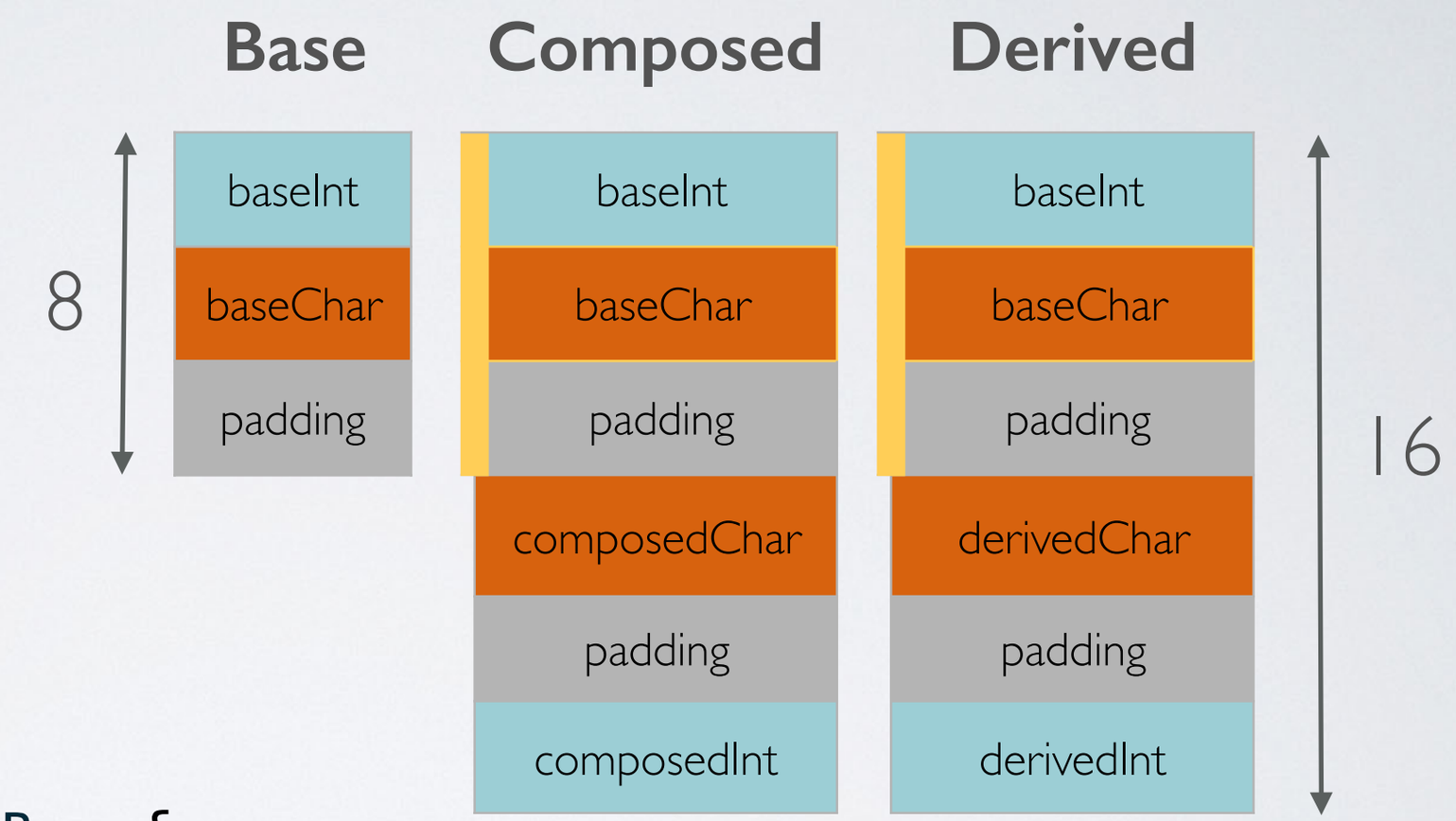
COMPOSITION

```
class Base{
    int baseInt;
    char baseChar;
};

class Composed{
    Base base;
    char composedChar;
    int composedInt;
};
```



INHERITANCE

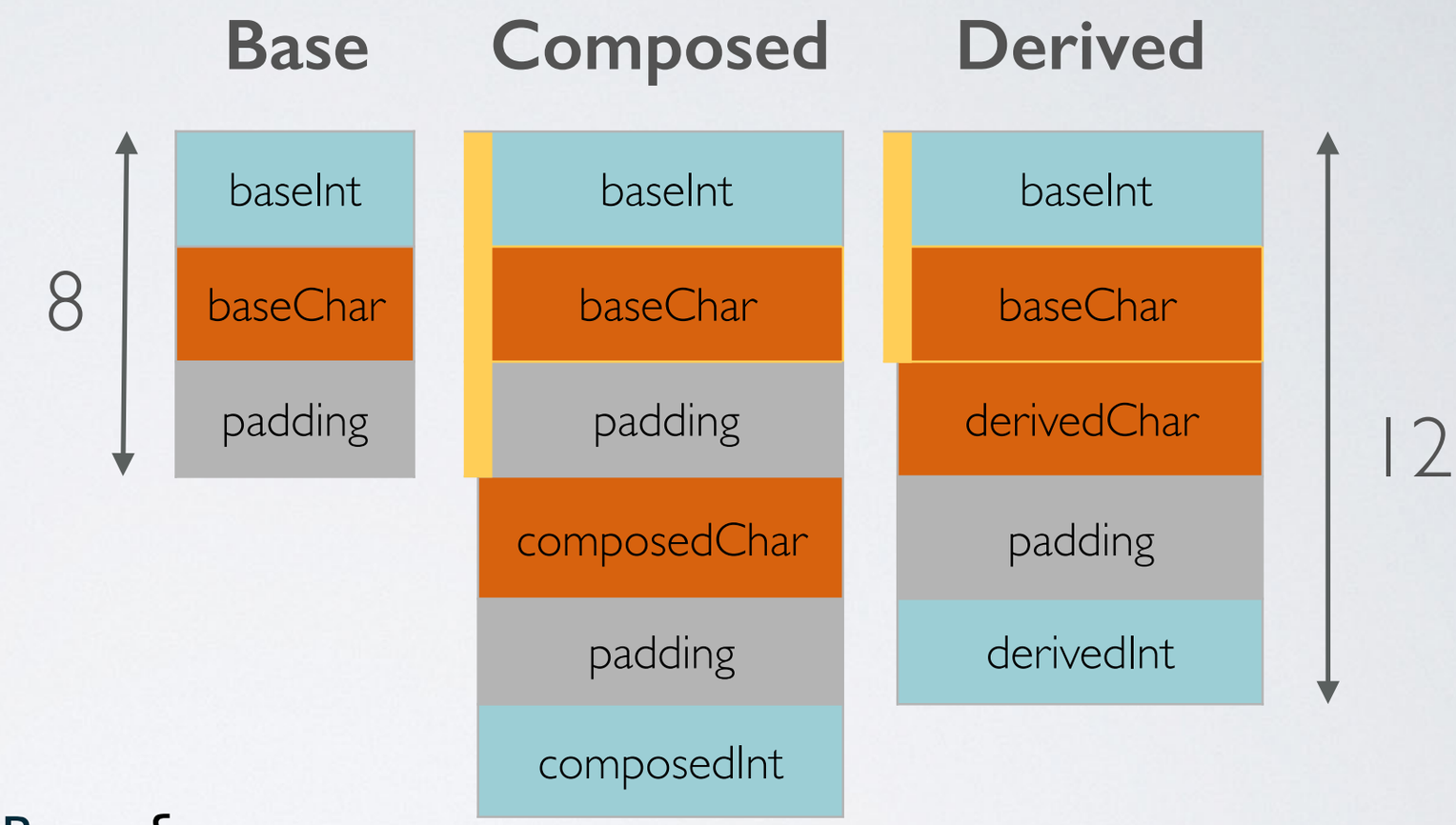
```
class Base{
    int baseInt;
    char baseChar;
};

class Composed{
    Base base;
    char composedChar;
    int composedInt;
};

class Derived: public Base{
    char derivedChar;
    int derivedInt;
};
```



MSVC compiler

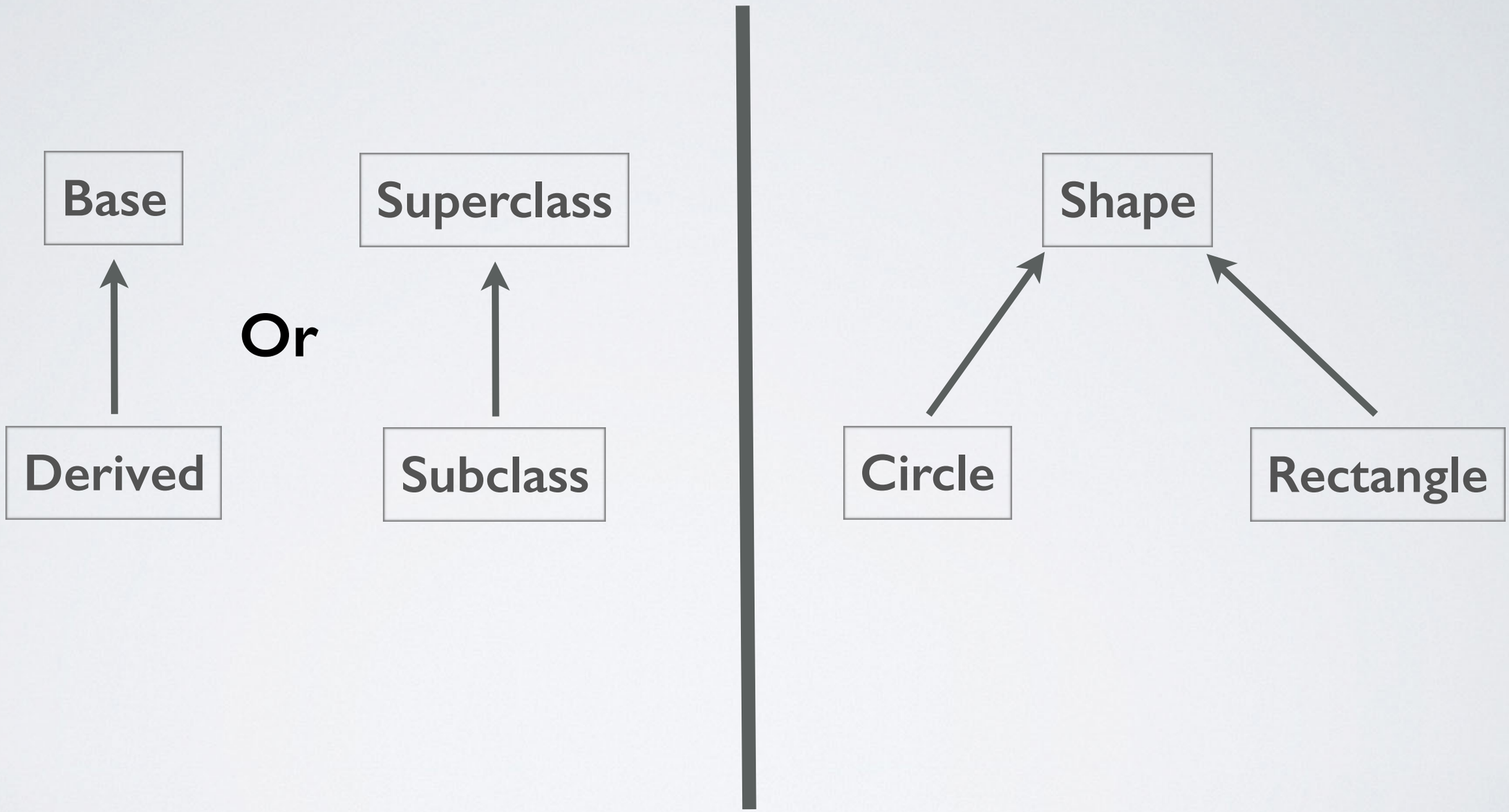
INHERITANCE

```
class Base{
    int baseInt;
    char baseChar;
};

class Composed{
    Base base;
    char composedChar;
    int composedInt;
};

class Derived: public Base{
    char derivedChar;
    int derivedInt;
};
```



Clang compiler

INHERITANCE



INHERITANCE

```
class Base{
    int baseInt;
    char baseChar;
public:
    Base(): baseInt(0), baseChar('0'){
    }
    ...
};
```

Автоматически вызовется Base()

```
class Derived: public Base{
    char derivedChar;
    int derivedInt;
public:
    Derived(char a, int b): derivedChar(a), derivedInt(b){
    }
    ...
};
```

INHERITANCE

```
class Base{
    int baseInt;
    char baseChar;
public:
    Base(): baseInt(0), baseChar('0'){
    }
    ...
};
```

Можно и так.

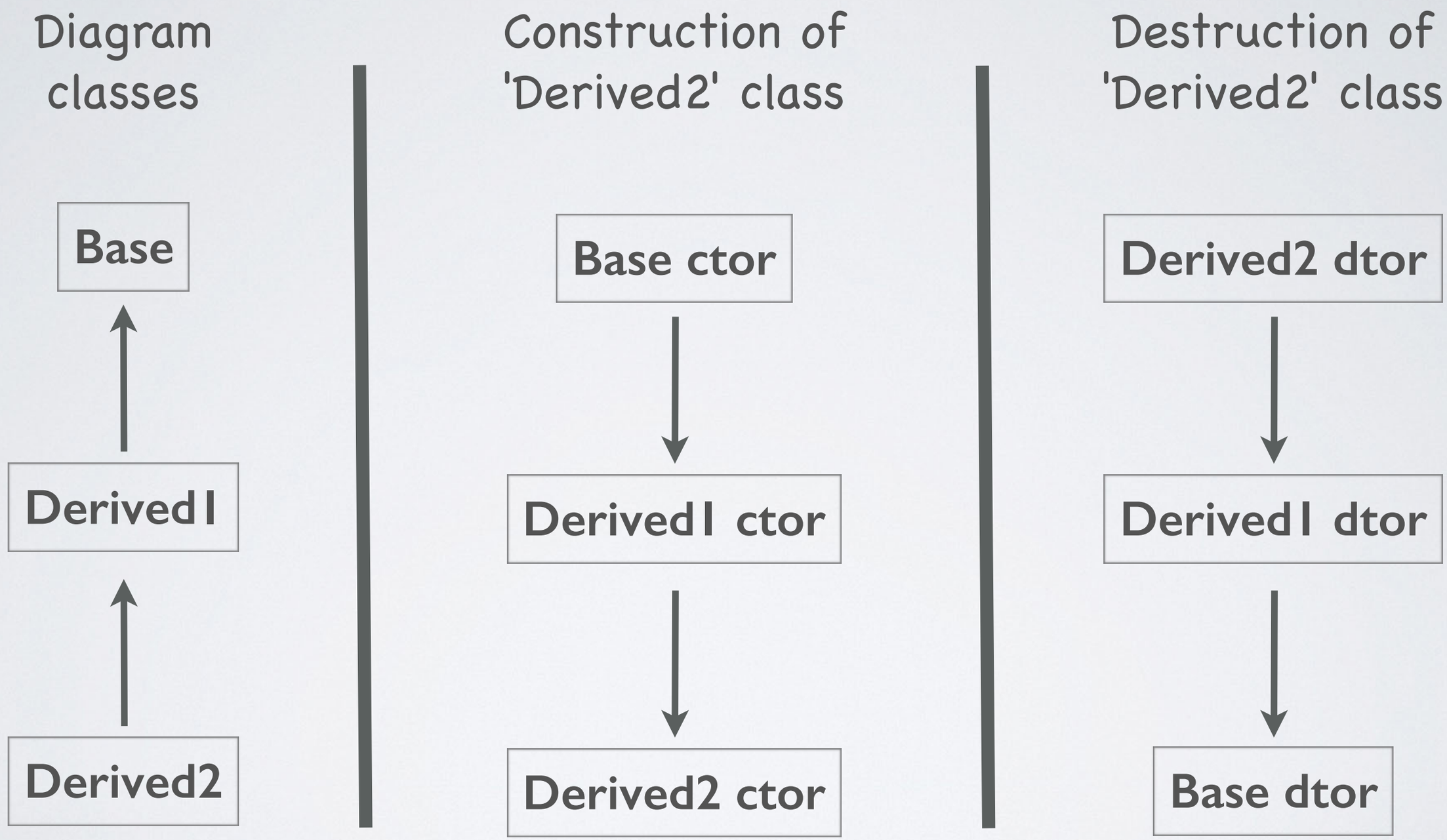
```
class Derived: public Base{
    char derivedChar;
    int derivedInt;
public:
    Derived(char a, int b): Base(), derivedChar(a), derivedInt(b){
    }
    ...
};
```

INHERITANCE

```
class Base{
    int baseInt;
    char baseChar;
public:
    Base(int baseA, char baseB): baseInt(baseA), baseChar(baseB){
    }
    ...
};

class Derived: public Base{
    char derivedChar;
    int derivedInt;
public:
    Derived(int baseA, char baseB, char a, int b)
        : Base(baseA, baseB), derivedChar(a), derivedInt(b){
    }
    ...
};
```

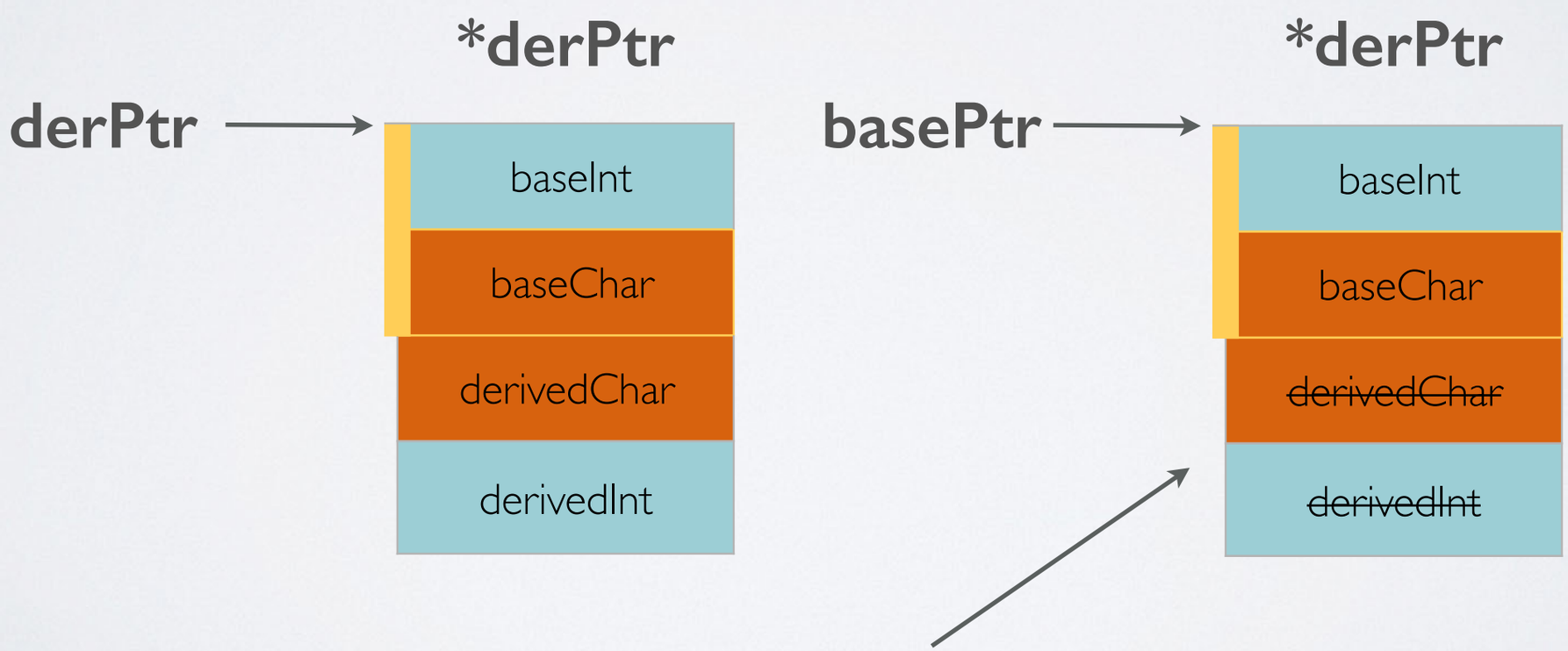
Обязательно вызывать в явном виде

CONSTRUCTION & DESTRUCTION ORDER



USING INHERITANCE

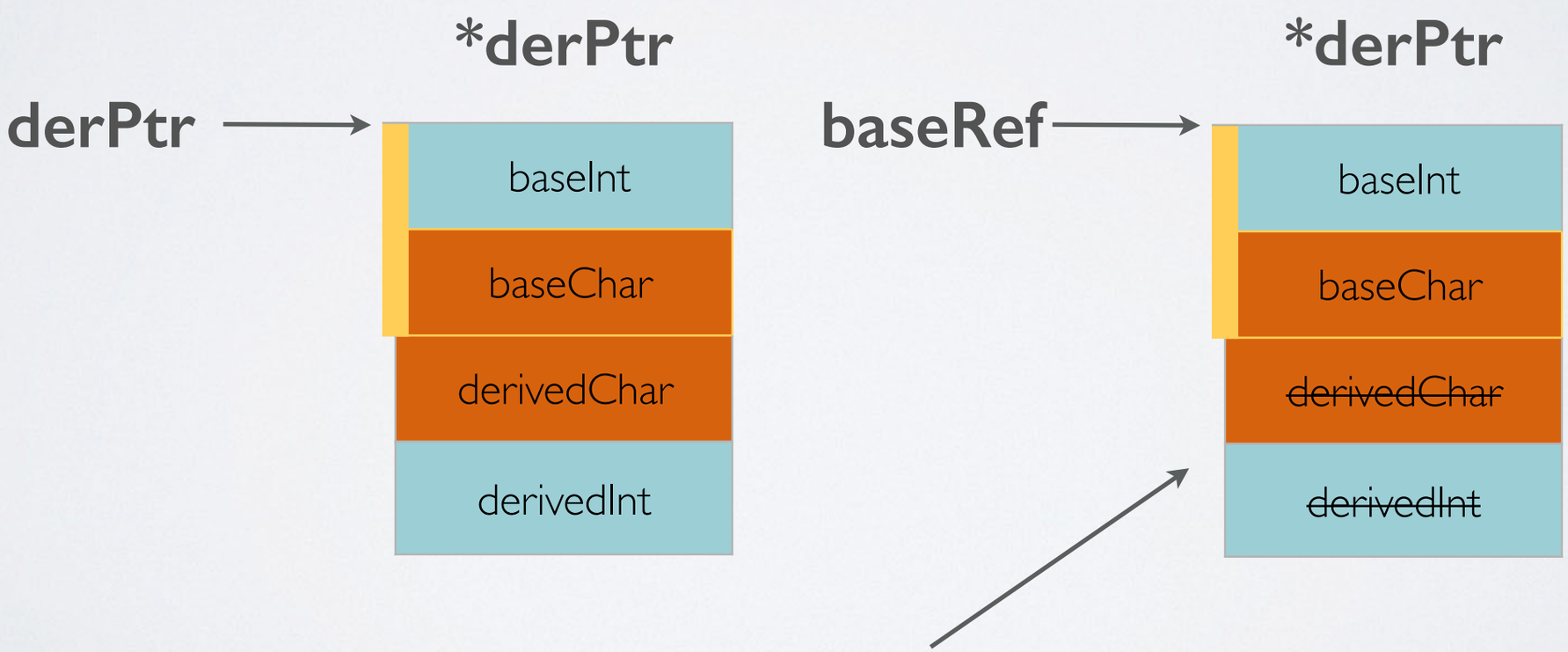
```
int main(){
    Derived* derPtr = new Derived;

    Base* basePtr = derPtr;

    return 0;
};
```



поля 'Derived' класса не доступны для basePtr

USING INHERITANCE

```
int main(){
    Derived* derPtr = new Derived;

    Base& baseRef = *derPtr;

    return 0;
};
```

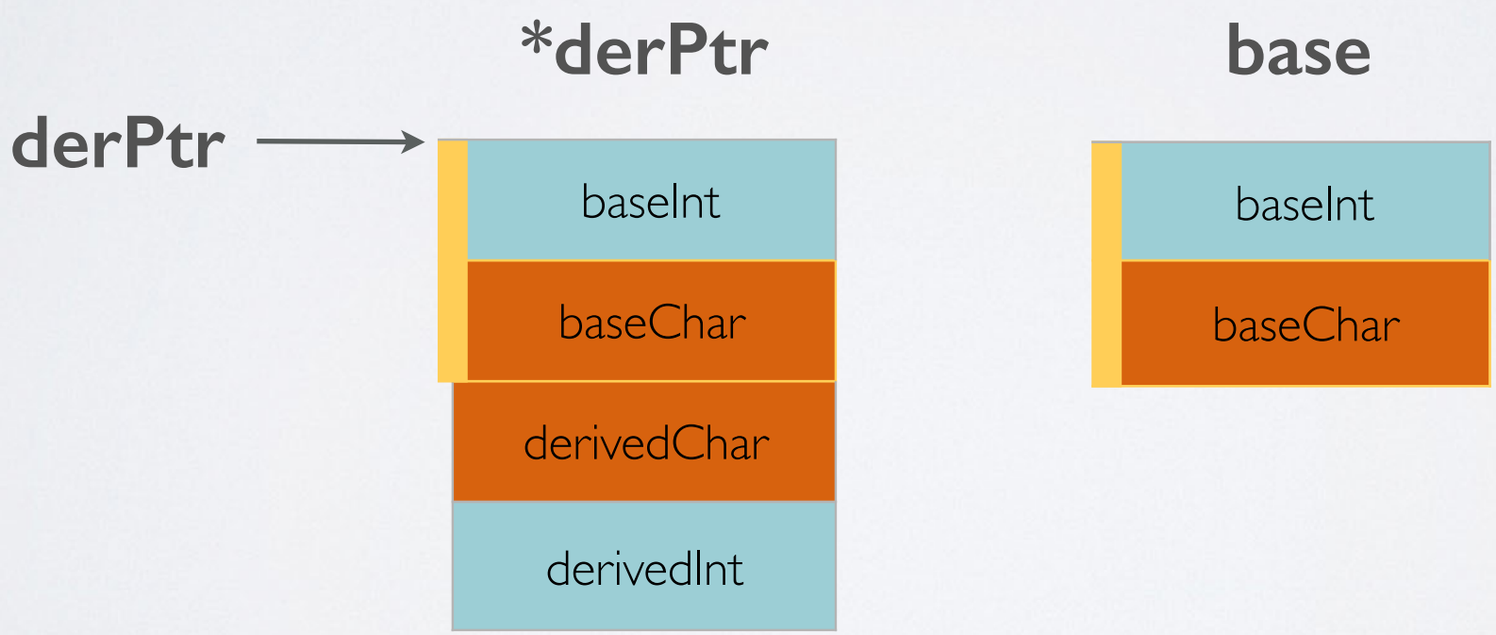


поля 'Derived' класса не доступны для baseRef

USING INHERITANCE

```
int main(){
    Derived* derPtr = new Derived;

    Base base = *derPtr;

    return 0;
};
```

copying operation

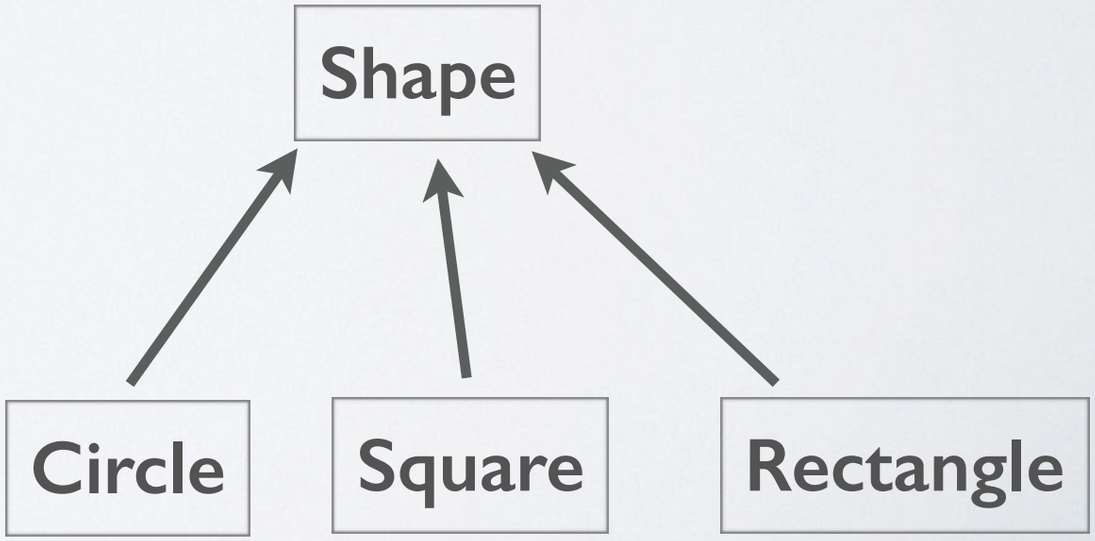


It's new object.

USING INHERITANCE

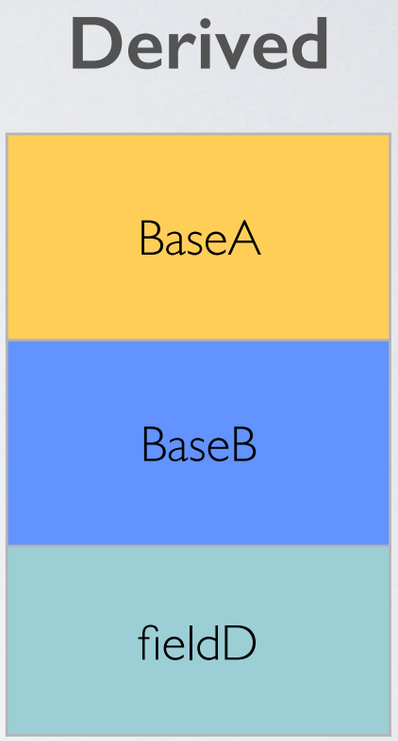
```
int main(){
    std::vector<Shape*> shapes;

    shapes.push_back(new Circle(1, 2, 2));
    shapes.push_back(new Rectangle(5, 5, 5, 5));
    shapes.push_back(new Square(10, 10, 2));

    for(int i = 0; shapes.size() > i; ++i){
        shapes[i]->draw();
        delete shapes[i];
    }

    return 0;
};
```

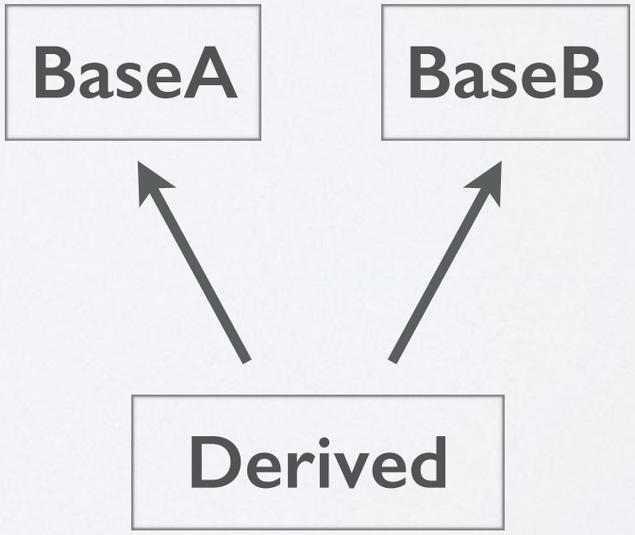
virtual function



MULTIPLE INHERITANCE

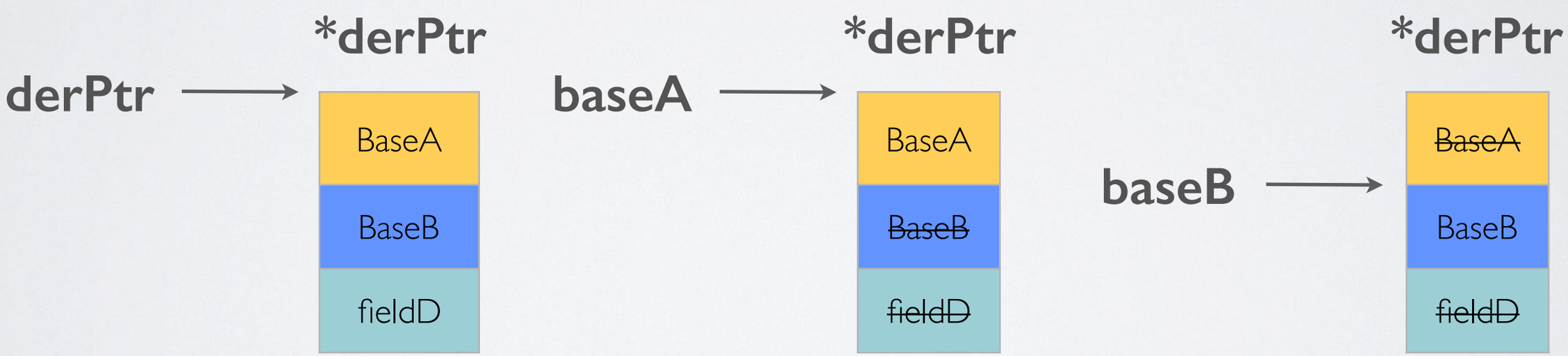
```
class BaseA{
    int fieldA;
};

class BaseB{
    int fieldB;
};

class Derived : public BaseA, public BaseB{
    int fieldD;
};
```





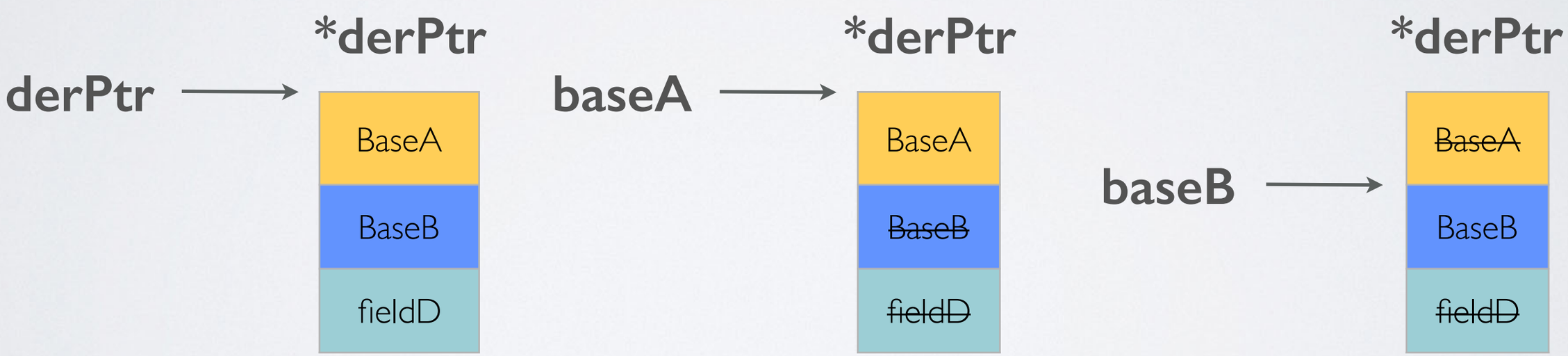
USING MULTIPLE INHERITANCE

```
int main(){
    Derived* derPtr = new Derived;

    BaseA* baseA = derPtr;
    BaseB* baseB = derPtr;

    return 0;
};
```



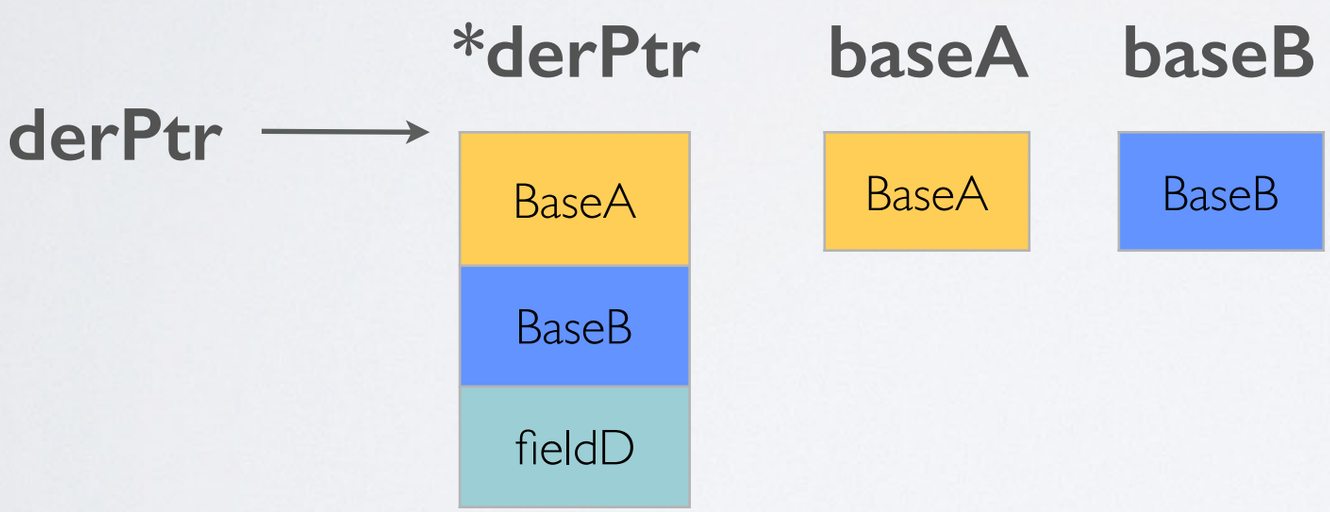
USING MULTIPLE INHERITANCE

```
int main(){
    Derived* derPtr = new Derived;

    BaseA& baseA = *derPtr;
    BaseB& baseB = *derPtr;

    return 0;
};
```



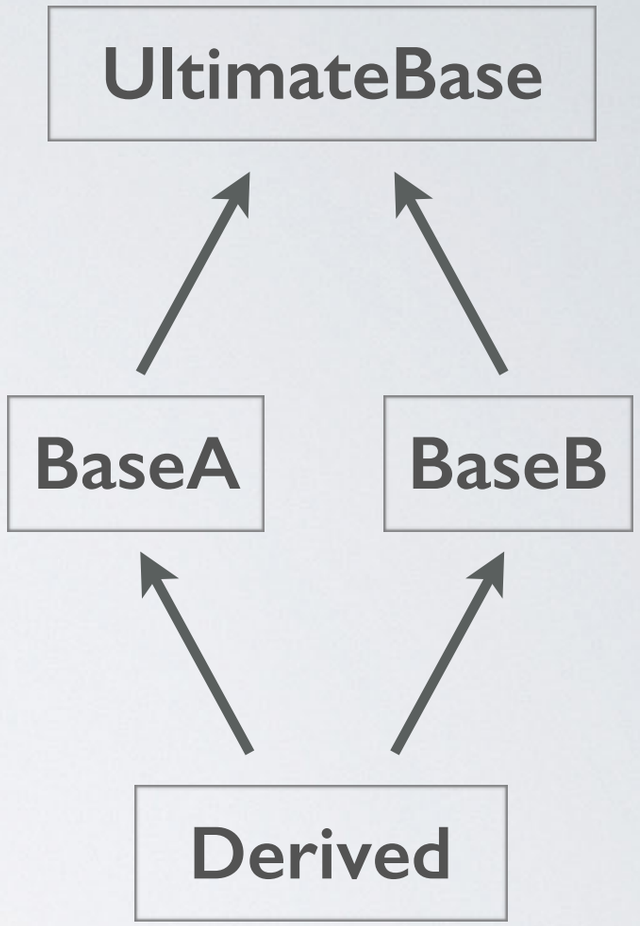
USING MULTIPLE INHERITANCE

```
int main(){
    Derived* derPtr = new Derived;

    BaseA baseA = *derPtr;
    BaseB baseB = *derPtr;

    return 0;
};
```



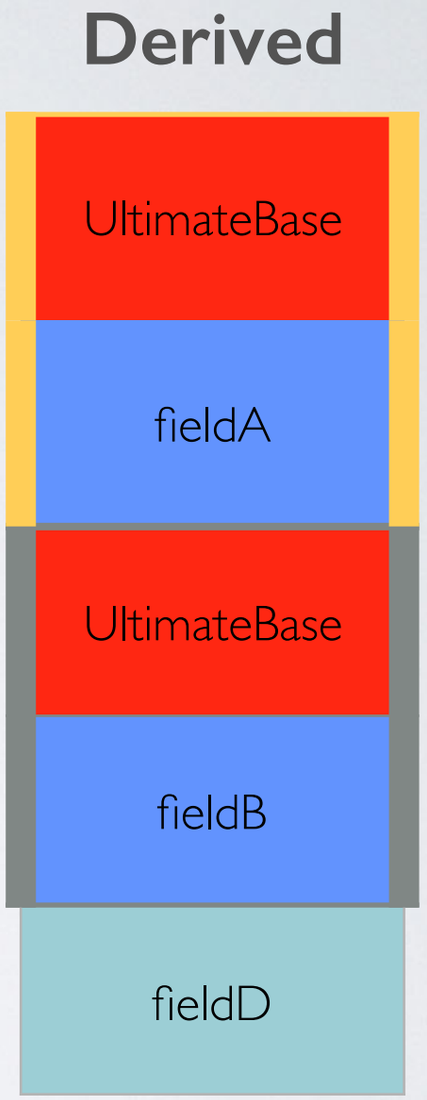
"DREADED DIAMOND"

```
class UltimateBase{
    int fieldU;
};

class BaseA : public UltimateBase{
    int fieldA;
};

class BaseB : public UltimateBase{
    int fieldB;
};

class Derived : public BaseA, public BaseB{
    int fieldD;
};
```



"DREADED DIAMOND"

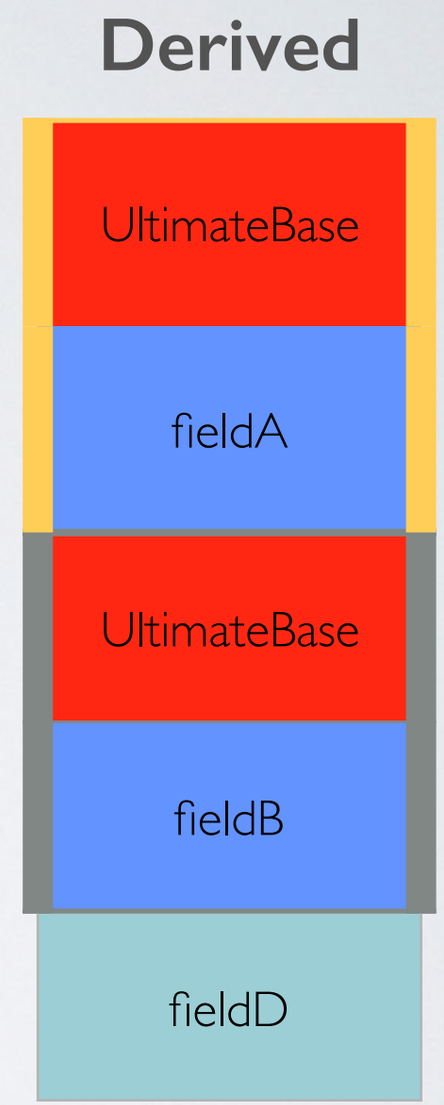
```
class UltimateBase{
    int fieldU;
};

class BaseA : public UltimateBase{
    int fieldA;
};

class BaseB : public UltimateBase{
    int fieldB;
};

class Derived : public BaseA, public BaseB{
    int fieldD;
};
```



"DREADED DIAMOND"

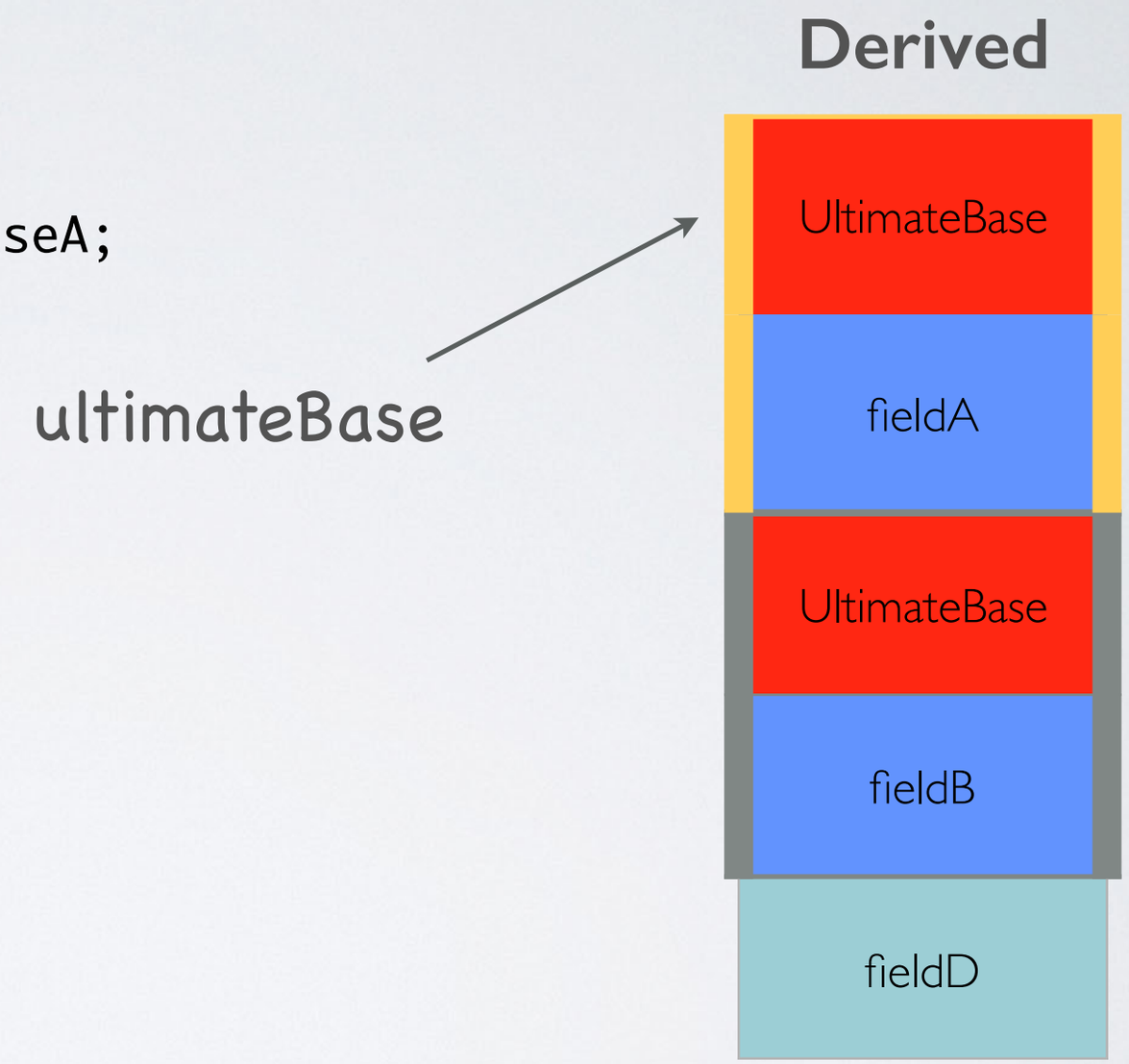
```
int main(){
    Derived* derPtr = new Derived;

    UltimateBase* ultimateBase = derPtr;

    return 0;
};
```

ultimateBase ???

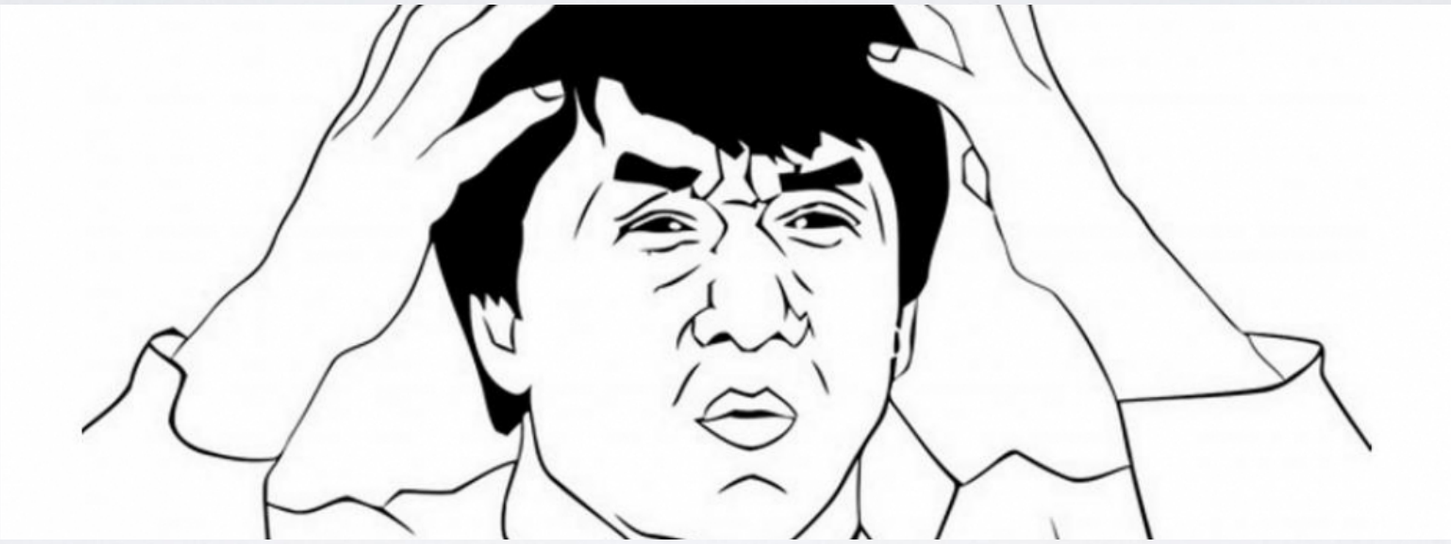


"DREADED DIAMOND"

```
int main(){
    Derived* derPtr = new Derived;
    BaseA* baseA = derPtr;
    UltimateBase* ultimateBase = baseA;
    delete devPtr;
    return 0;
};
```

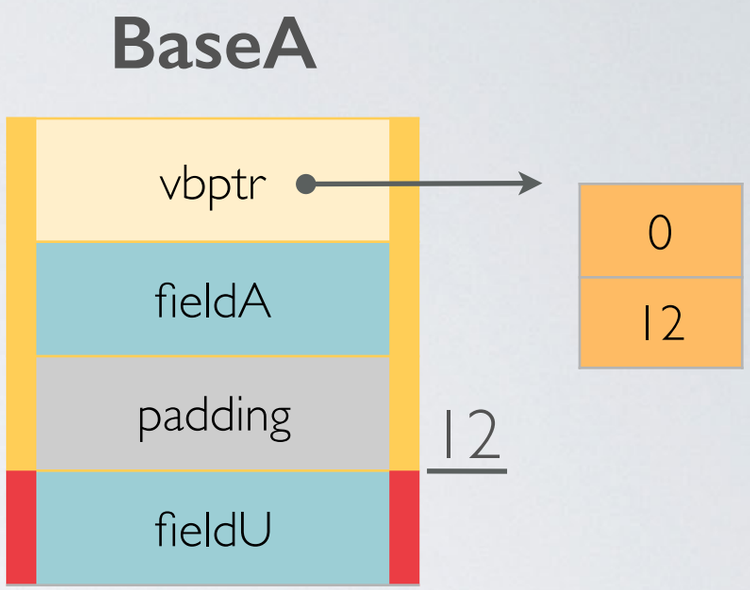


VIRTUAL INHERITANCE



VIRTUAL INHERITANCE

```
class UltimateBase{
    int fieldU;
};

class BaseA : virtual public UltimateBase{
    int fieldA;
};

class BaseB : virtual public UltimateBase{
    int fieldB;
};

class Derived : public BaseA, public BaseB{
    int fieldD;
};
```



vbptr - указатель на таблицу адресования виртуальных базовых классов.

VIRTUAL INHERITANCE

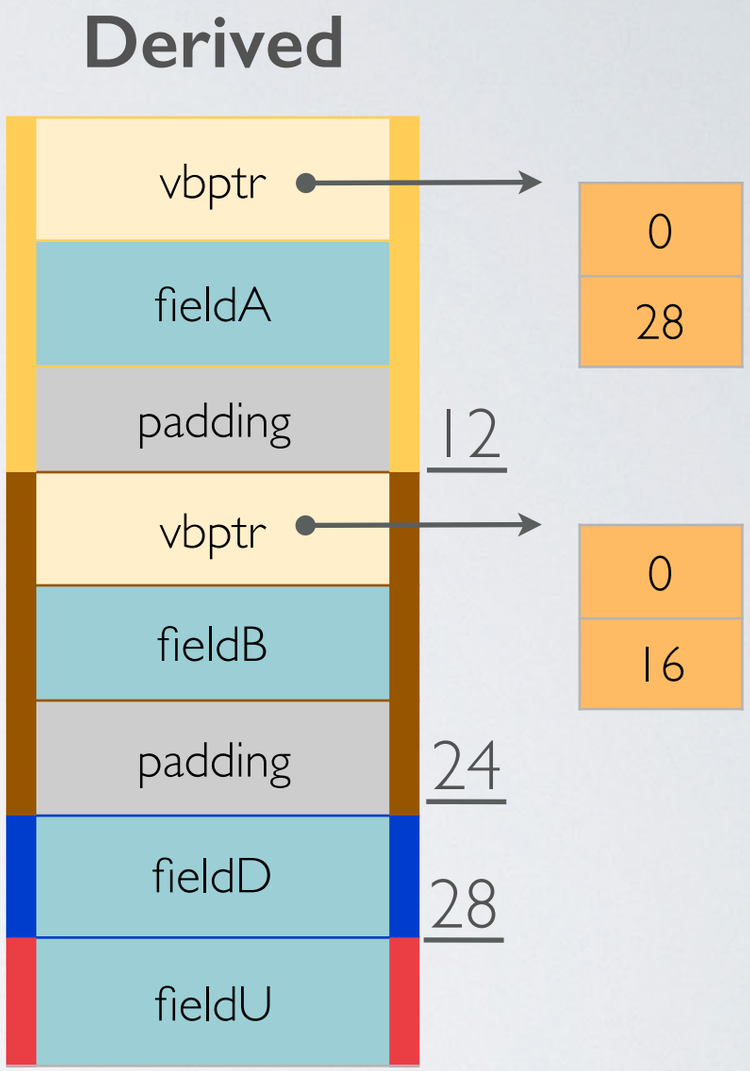
vbptr

Смещение от **vbptr** до начала класса
Смещение от **vbptr** до виртуального базового **класса 1**
Смещение от **vbptr** до виртуального базового **класса 2**
...
Смещение от **vbptr** до виртуального базового **класса N**

MSVC compiler

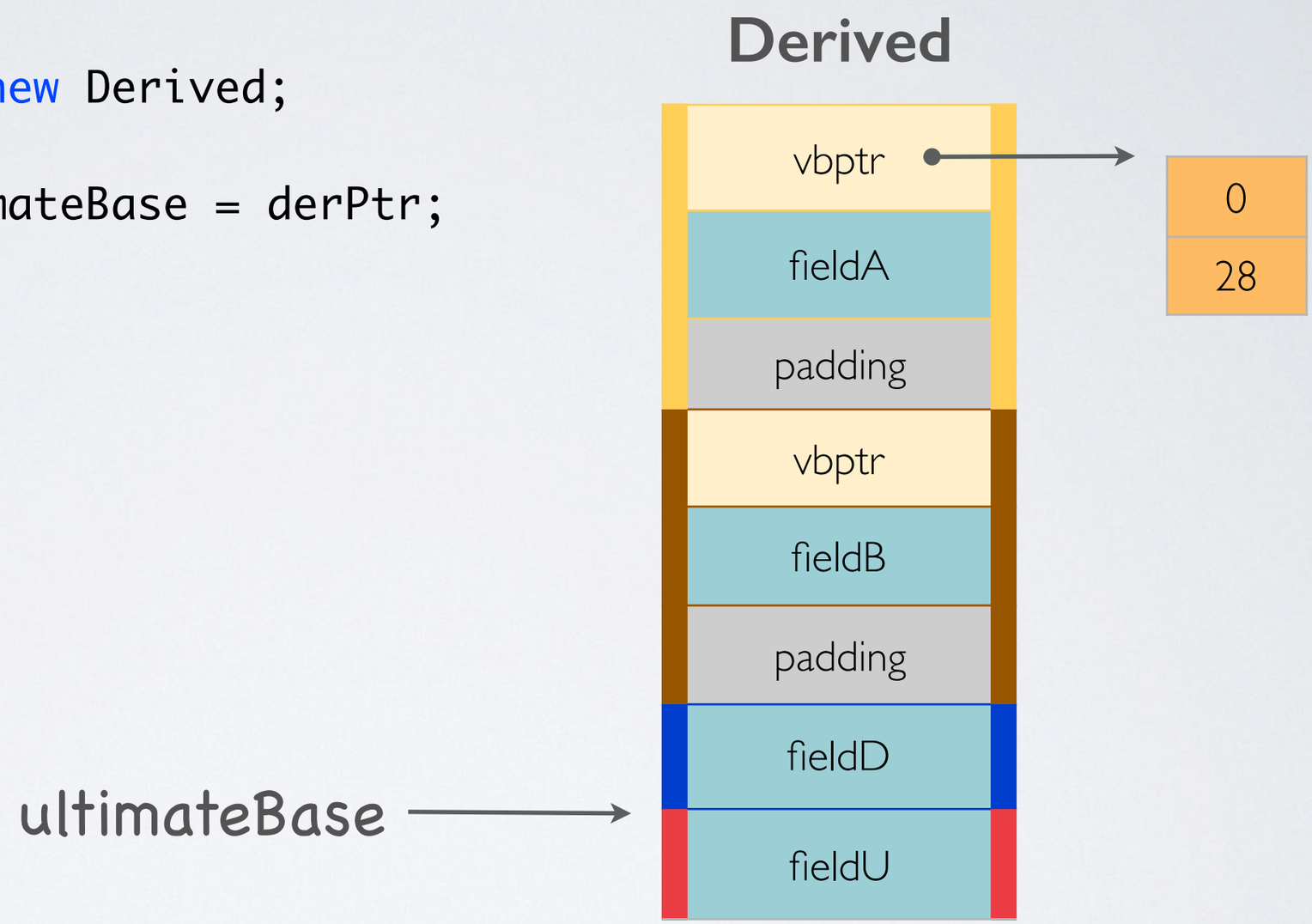
VIRTUAL INHERITANCE

```
class UltimateBase{
    int fieldU;
};

class BaseA : virtual public UltimateBase{
    int fieldA;
};

class BaseB : virtual public UltimateBase{
    int fieldB;
};

class Derived : public BaseA, public BaseB{
    int fieldD;
};
```



VIRTUAL INHERITANCE

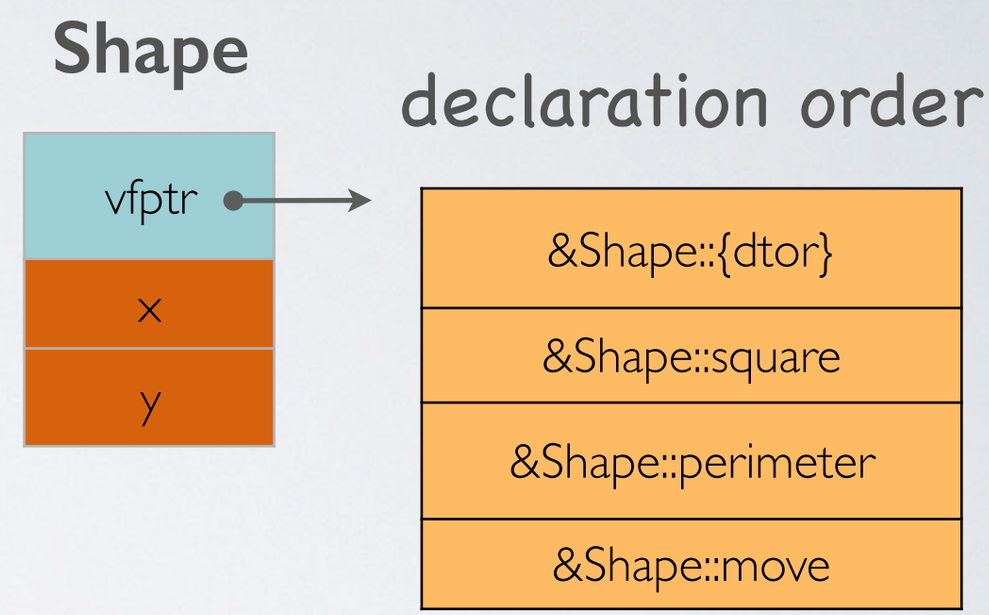
```
int main(){
    Derived* derPtr = new Derived;

    UltimateBase* ultimateBase = derPtr;

    return 0;
};
```



But Wait!!!---That's NOT All

VIRTUAL FUNCTIONS

```
class Shape{
    int x, y;
public:
    Shape(int x, int y);
    virtual ~Shape();

    virtual double square() const;
    virtual double perimeter() const;
    virtual double move(int x, int y);
    int getX() const;
    int getY() const;
};

class Circle: public Shape{
    int radius;
public:
    Circle(int x, int y, int radius);
    virtual ~Circle();

    virtual double square() const;
    virtual double perimeter() const;
};
```

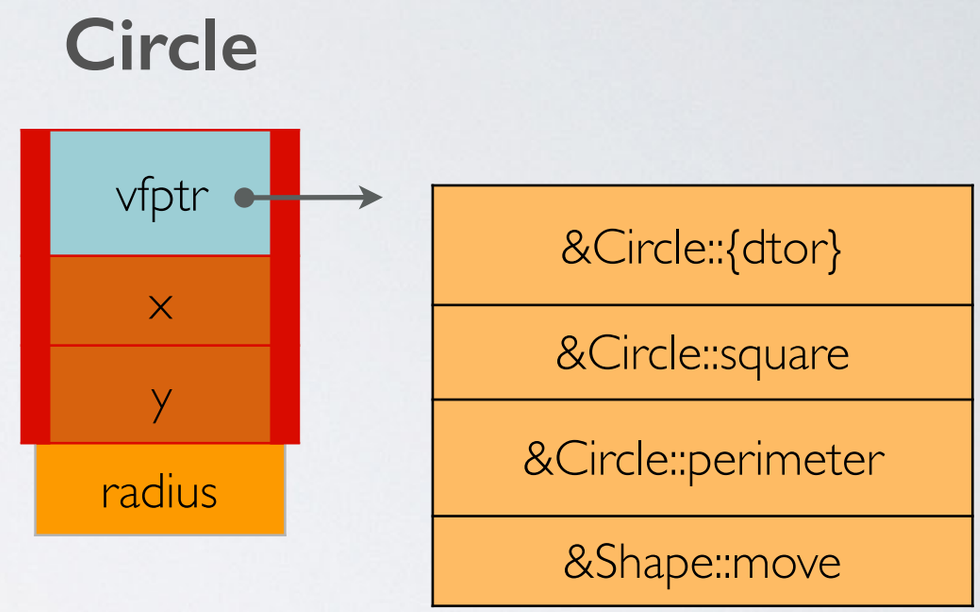


MSVC compiler

vfptr - указатель на таблицу виртуальных функций. У каждого класса своя таблица виртуальных функций.

VIRTUAL FUNCTIONS

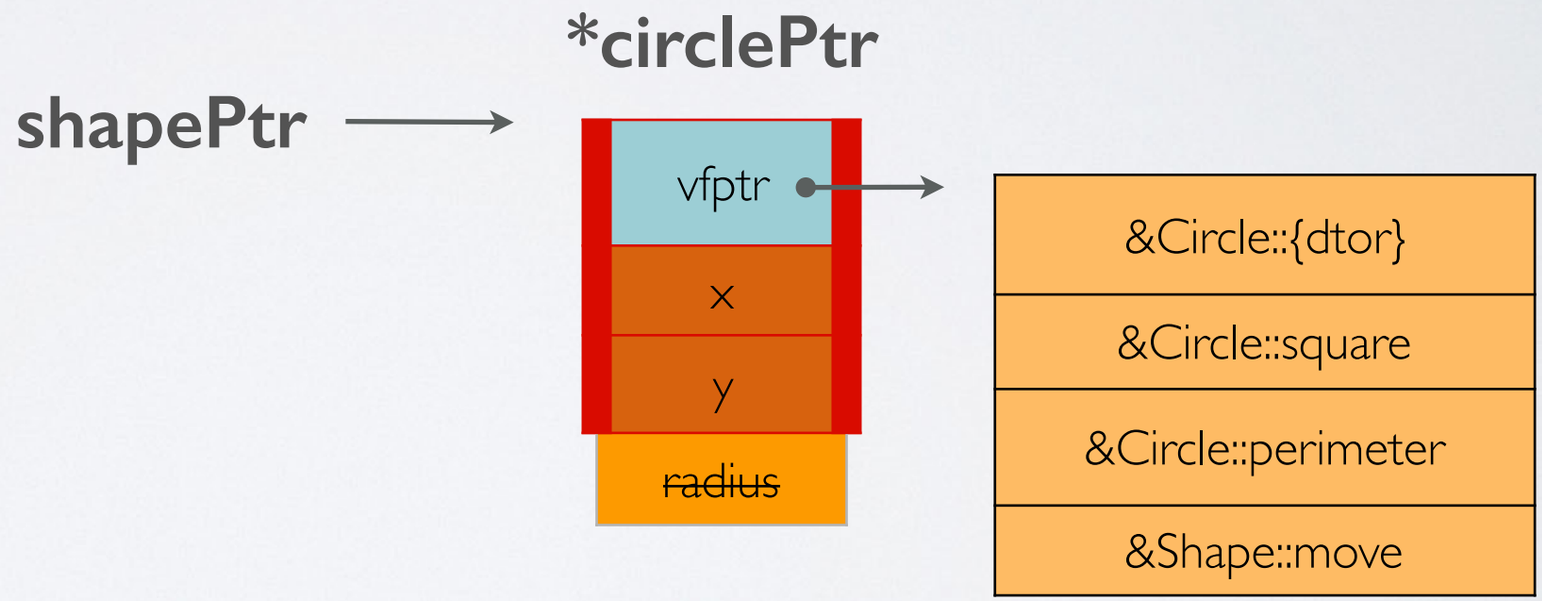
```
class Shape{
    int x, y;
public:
    Shape(int x, int y);
    virtual ~Shape();

    virtual double square() const;
    virtual double perimeter() const;
    virtual double move(int x, int y);
    int getX() const;
    int getY() const;
};

class Circle: public Shape{
    int radius;
public:
    Circle(int x, int y, int radius);
    virtual ~Circle();

    virtual double square() const;
    virtual double perimeter() const;
};
```



MSVC compiler

VIRTUAL FUNCTIONS

```
int main(){
    Circle* circlePtr = new Circle(1, 1, 1);

    Shape* shapePtr = circlePtr;
    shapePtr->move(1, 2);
    shapePtr->square();

    delete shapePtr;

    return 0;
};
```



КОНЕЦ ТРЕТЬЕЙ ЛЕКЦИИ

```
virtual ~Lection() { }
```