

### 3.21. Понятие полиморфизма. Понятие виртуальных функций. Разница в поведении виртуальных и неvirtуальных функций при наследовании. Предназначение и использование ключевых слов `override` и `final`.

**Идея:** Пусть у нас есть геометрические фигуры. Площадь квадрата человек будет считать как произведение сторон, площадь прямоугольного треугольника - половина произведения катетов, площадь произвольного многоугольника - будет триангулировать и считать площадь. Цель методов одна и та же (найти площадь), реализация - разная.

**Полиморфизм** - один из главных столпов объектно-ориентированного программирования. Его суть заключается в том, что один фрагмент кода может работать с разными типами данных. Свойство, которое позволяет использовать одно и тоже имя функции для решения двух и более схожих, но технически разных задач.

```
1 // ===== Версии без полиморфизма =====
2
3 struct Base_NoPolymorphism {
4     void f() { cout << 1; }
5 };
6 struct Derived_NoPolymorphism: public Base_NoPolymorphism {
7     void f() { cout << 2; }
8 };
9
10 // ===== Версии с полиморфизмом =====
11
12 struct Base {
13     virtual void f() { cout << 1; } // (1)
14 };
15
16 struct Derived: public Base {
17     // void f() const cout << 2; - не виртуальный, т.к. не совпадает с (1)
18     // virtual void f() const cout << 2; - виртуальный, но не переписывает (1)
19     virtual void f() {cout << 2; }
20     int check = 5;
21 };
22
23 int main () {
24     // До полиморфизма:
25     Base_NoPolymorphism b;
26     b.f(); //1
27     Derived_NoPolymorphism d;
28     d.f(); //2
29     Base_NoPolymorphism bb = d;
30     bb.f(); //1
31     Base_NoPolymorphism& bbb = d;
32     bbb.f(); //(*) , печатается 1, взяли версию родителя
33
34     // С полиморфизмом:
35     Base b1;
36     b1.f(); //1
37     Derived d1;
38     d1.f(); //2
39     Base bb1 = d1;
40     bb1.f(); // 1 - мы считаем этот объект родителем
41     Base& bbb1 = d1;
42     bbb1.f(); // 2 - он "помнит каким ребёнком был d1
43 }
```

**Виртуальная функция** - такая функция, что если обратиться к ней вне зависимости от того, это объект общего типа или частного типа, вы предпочитаете реализацию в объекте частного типа.

Для виртуальных функций выбор происходит в **Runtime**, для не виртуальных в **Compile time**. Именно поэтому в примерах выше если мы делаем копию, то выбирается родительский метод (всё определилось в **Compile Time**), а для ссылок/указателей всё выбирается в **Runtime**, мы выбрали дочерний метод.

Виртуальная функция дочернего класса является переопределением только если совпадают её сигнатура и тип возврата с сигнатурой и типом возврата виртуальной функции родительского класса. Для проверки того, что данная функция является переопределением, добавили модификатор **override** в C++11, из-за которого компилятор выдаёт **CE**, если метод не переопределяет виртуальную функцию родительского класса.

```
1 struct A {
2     virtual const char* getName(int x) { return "A"; }
3 };
4
5 struct B : public A {
6     // virtual const char* getName(short int x) override return "B"; - CE
7     // virtual const char* getName2(int x) const override return "B"; - CE
8     virtual const char* getName3(int x) override { return "B"; } // OK
9 };
```

Модификатор **final** используется, если вы не хотите, чтобы кто-то мог переопределить виртуальную функцию или наследовать определенный класс. Если пользователь пытается переопределить метод или наследовать класс с модификатором **final**, то компилятор выдаст ошибку.

```
1 struct A {
2     virtual const char *getName() { return "A"; }
3 };
4
5 struct B : public A {
6     virtual const char *getName() final { return "B"; } // OK
7 };
8
9 struct C : public B {
10     virtual const char * getName() { return "C"; } // CE
11 };
```