3.21. Понятие полиморфизма. Понятие виртуальных функций. Разница в поведении виртуальных и невиртуальных 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.

```
// =============== Версии без полиморфизма ===================

struct Base_NoPolymorphism {
  void f() { cout << 1; }
};
struct Derived_NoPolymorphism: public Base_NoPolymorphism {
  void f() { cout << 2; }
};

// =============== Версии с полиморфизмом ===================

struct Base {
  virtual void f() { cout << 1; } // (1)
};

struct Derived: public Base {
  // void f() const  cout « 2;  - не виртуальный, т.к. не совпадает с (1)
  // virtual void f() const  cout « 2;  - виртуальный, но не переписывает (1)
  virtual void f() {cout << 2; }
  int check = 5;
};

int main () {
  // До полиморфизма:
  Base_NoPolymorphism b;
  b.f(); //1
  Derived_NoPolymorphism d;
  d.f(); //2
  Base_NoPolymorphism bb = d;
  bb.f(); //1
  Base_NoPolymorphism& bbb = d;
  bbb.f(); //(*), печатается 1, взяли версию родителя

  // С полиморфизмом:
  Base b1;
  b1.f(); //1
  Derived d1;
  d1.f(); //2
  Base bb1 = d1;
  bb1.f(); // 1 - мы считаем этот объект родителем
  Base& bbb1 = d1;
  bbb1.f(); // 2 - он "помнит каким ребёнком был d1
}
```

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.

```
struct A {
  virtual const char* getName(int x) { return "A"; }
};

struct B : public A {
  // virtual const char* getName(short int x) override  return "B";  - CE
  // virtual const char* getName2(int x) const override  return "B";  - CE
  virtual const char* getName3(int x) override { return "B"; } // OK
};
```

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.

```
struct A {
  virtual const char *getName() { return "A"; }
};

struct B : public A {
  virtual const char *getName() final { return "B"; } // OK
};

struct C : public B {
  virtual const char * getName() { return "C"; } // CE
};
```

NOTE: для любой функции, которая является "дочерней"от какой-то родительской (помеченной virtual) формально можно не писать virtual, если у неё нет своих "детей". Однако на практике часто проще ориентироваться в виртуальных функциях, если это подписано всегда.