Тема 11

* Полиморфизъм - едно име, много имплементации

1) Compile time polymorphism
   ⇒ при компилация на програмата е ясно коя ф-я ще се извика

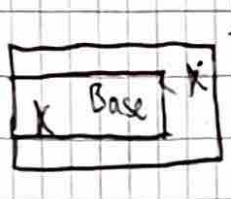
- Function overloading

```
void f();
void f(int);
void f(string);
```

- Operator overloading

```
3+4
"AB"+"CD"
3.2+5
```

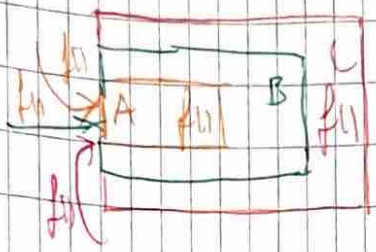
Base
↑ int x
Der
int x




```
Der d;
d.x; // Der::x  ⇒ винаги Base
Base* ptr = &d;
ptr->x // Base::x
```

```
Der d;
d.f1(); // Der::f1()
Base* ptr = &d;
ptr->f1() // Base::f1()
```

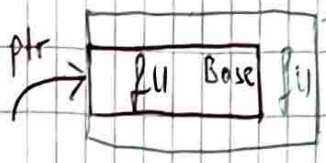
- Статично свързване
  → зависи на ф-ята, която да се извика, става по време на комп
  → изв. е указат./реф, от който се извиква ф-ята



- Динамично свързване
  → изв. на ф-я, става по време на изп. на програмата
  → Runtime polymorphism
  → Чрез вирт. ф-ции

```
class Base {
  virtual void f1() {...}
};   ↳ някой наследник ще я презапише
```

```
class Der : Base {
    void f() override {...};
```
↳ може да се изп., но не е коректно




```
Der d;
d.f(); // Der::f()
Base* ptr = &d;
ptr->f(); // Der::f()
```
↳ Винаги ще f() е вирт. и започва да търси "на дясно" за по-ключа имплем.

△ При създаването на Der, първо се извиква констр. на Base!

- Абстрактен клас
  - ⇒ клас предназначен за наследяване
  - ⇒ не можем да правим обект от него
  - ⇒ поне 1 абстр. ф-я

~ Абстрактна ф-я (Pure Virtual Func) - чисто виртуална

```
virtual void f() = 0
```
↳ ∀ наследник я разписва

- Ако някой не я разпише, става абстрактен

⚠ При полиморфна йерархия трябва винаги да имаме виртуален дестр.


```
Base * ptr = new Der();
delete ptr; // virtual ~Base()
                ↳ ~Der() {} → ~Base()
virtual ~Base = default;
```

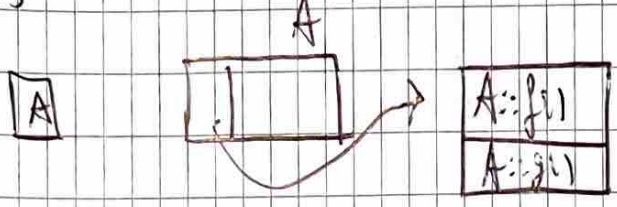
- Ключова дума final - никой наследник не може да презапише
  ↳ за класове - не можем да наследяваме

- Виртуални таблици

```
struct A {
  virt f();
  virt g();
}

struct B:A {
  f() override
}

struct C:B {
  f() override
  g() override
}
```



Минуси:
→ 2 дереферирания
→ много памет