Тема 12

Ключови думи - override, final. Виртуални таблици. Колекции от обекти в полиморфна йерархия. Копиране и трие. Factory pattern.

Когато пишем виртуални функции можем да използваме ключовите думи override и final. Override указва, че дадена функция презаписва функция от базовия клас. Ако в базовия клас няма такава функция, тогава няма да се компилира. Final указва, че дадена функция не може да се презаписва надолу по йерархията. Final може да се използва и за класове, където знаем, че даден клас не може да се наследява.

Пример:

A virtual void f() const;

↑

B void f() const override;

↑

C void f() const override final;

Забележка: Не е позволено да използваме final за чисто виртуалните функции, защото те са предназначени да бъдат презаписвани във всеки наследник, а с final предпазваме това.

Виртуални таблици - това са таблици с указатели към функции
Имаме следния пример:

A virtual void f(); virtual void g();

↑

B void f() override; void g() override; void h();

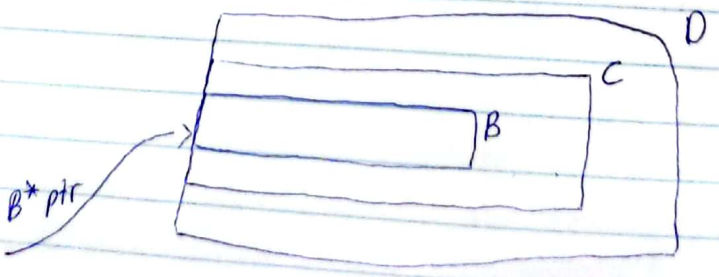
↑

C void f() override;

↑

D void g() override

```
D obj;
A *ptr = &obj;
ptr->f(); // C::f()
ptr->g() // B::g()
ptr->h() // D::h()
```



Виртуалните таблици изглеждат по следния начин:

A
A::f()
A::g()

B	
B::f()	- презаписана
B::g()	- презаписана
B::h()	- появява се за пръв път

C	
C::f()	- презаписва
B::g()	} наследени
B::h()	} от B

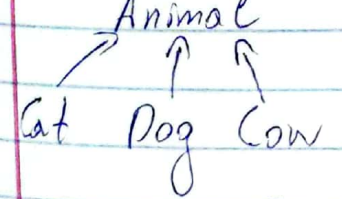
D	
D::f()	- презаписана
B::g()	- наследена
D::h()	- презаписана

Всеки път, когато създаваме обект от даден тип, в него "живее" указател, сочещ към съответната виртуална таблица. В примера създаваме указател от тип A и го насочваме към обект от тип D. Указателят намира необходимата виртуална ~~функция~~ таблица и избира правилната функция.

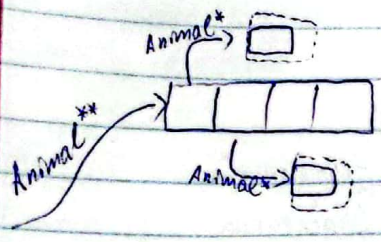
Забележка: Това е бавен механизъм, защото веднъж дереферираме указателя, за да намерим обект от тип A, после втори път дереферираме указателя, за да намерим виртуалната таблица и после отново за функцията f().

Можем да реализираме колекция от различни типове (с общ базов клас) чрез масив от указатели, така наречения хетерогенен контейнер. Указателите трябва да са от типа на базовия клас.

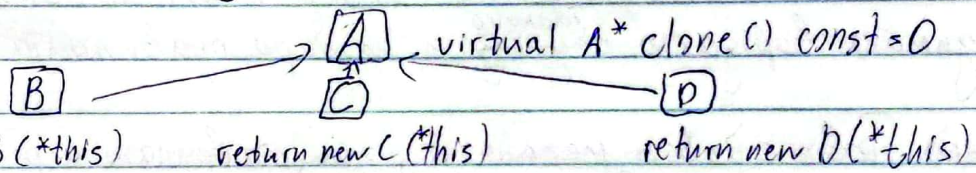
Пример:



Можем да обградим Animal** в клас (Farm), който да има самостоятелно копиране и трие. В Animal** не се интересуваме от вида на обектите.



Ако имаме необработена динамична памет, трябва да разпишем голямата петворка.
Искаме да реализираме копиране на колекцията и това трябва да стане без да нарушаваме абстракцията - искаме обектите да се копират без да ни налага да запитваме за техния тип. Затова дефинираме виртуална функция Clone(), която ще връща копие на обекта. Тази функция е чисто виртуална в базовия клас и трябва да я разпишем във всеки един от наследниците.



От тук копирането става тривиално

```
void Farm::copyFrom(const Farm& other)
{
  animals = new Animal*[other.capacity];
  animalsCount = other.animalsCount;
  capacity = other.capacity;
  for(size_t i=0; i<animalsCount; i++)
      animals[i] = other.animals[i]->clone();
}
```

Триене: Понеже имаме виртуален деструктор в базовия клас, не се интересуваме в колекцията какви са обектите, които трием

```
void Farm::free()
{
  for(size_t i=0;i<animalsCount;i++)
  {  delete animals[i];
  delete[] animals;
}
```

Design patterns - решение на проблем, който възниква често. Важно е решението да може да се преизползва

Пример: Factory pattern - подходът за решаване на този проблем се осъществява като създадем отделен клас за създаването на обектите

```
class AnimalFactory
{
 public:
  virtual Animal* createAnimal(Animal::AnimalType type);
};
```

В този клас има самостоятелна функция, която се грижи за създаването на нивотни по определен идентификатор (в случая според даден тип). Най-удачно е този тип да е enum.

Пример:

```
enum AnimalType
{
  Dog=0,
  Cat=1,
  Cow=2
};
AnimalType getType() const
{
  return myType;
}
```