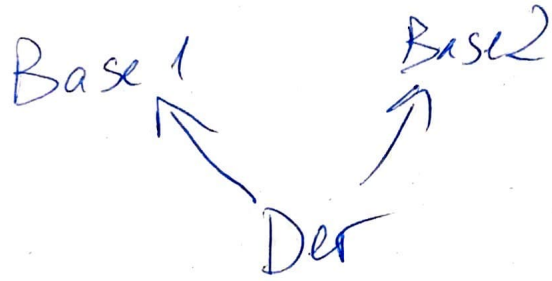
12. Тема | Множествено наследяване. Диамантен "проблем". Колекция от обекти в полиморфна йерархия. Копиране и триене. Visitor Pattern.

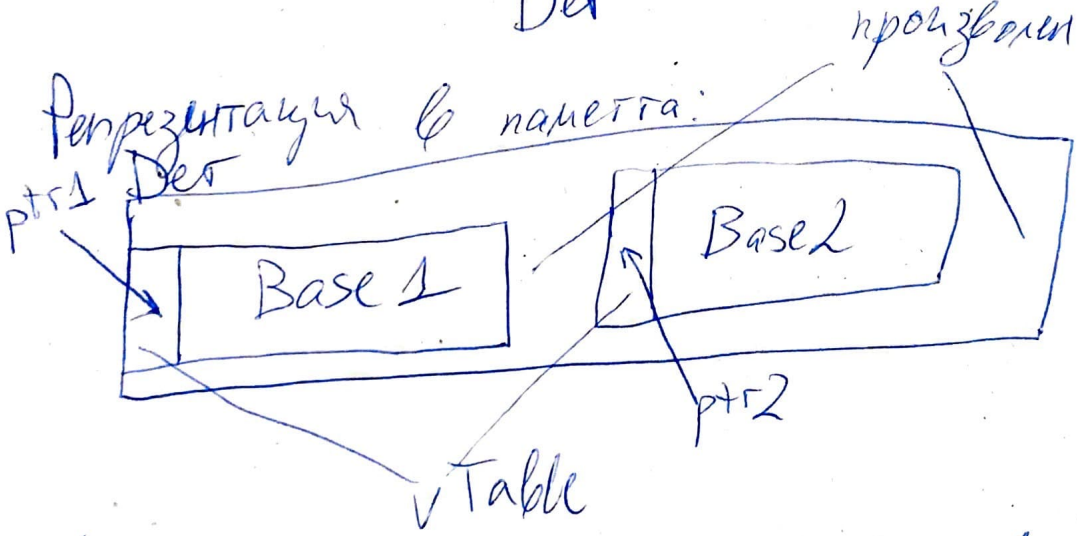
Множествено наследяване

В C++, за разлика от други езици, можем да имаме множествено наследяване.

Пример:



Репрезентация в паметта:



⚠ Адресите на d и ptr1 съвпадат.

Base2* ptr2 = &d; // Специален механизъм

- Инициализация

Der: Base1, Base2

A obj1;

B obj2;

Der(): Base1(), Base2(), ..., obj1(), ob2() {...}; Тук редът вътре е такъв

- ОП =

```
if(this != &other)
{
    Base1::operator=(other);
    Base2::operator=(other);
    free();
    copyFrom(other);
}
```

⚠ Проблем на предефинирането на ф-ции

Ако Base1 virtual f() и Base2 virtual f(); virtual g();

Der f() override;

// не работи

Ако няма override, при извикване на f() не има Compile Time error.

Решение:

vtable на Base1 и Base2, в която има Δ (отместване)

vtable Base1/Der

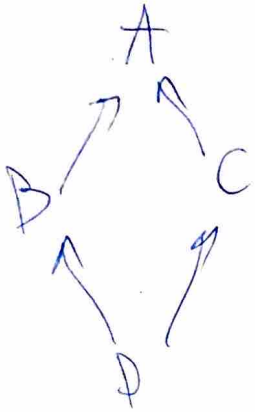
	Δ
Der::f()	0

vtable Base2

Der::f()	-Δ[Base2]
Base2::g()	0

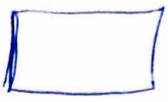
Така ще изглежда vtable при Der → f() override

Диамантен "проблем"

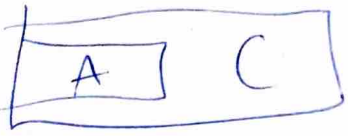


Искаме да има следната структура:

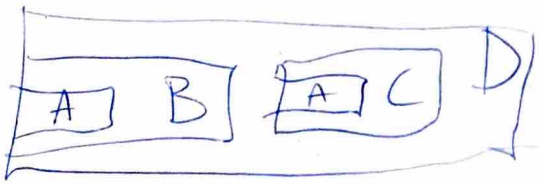
Диаг:

A: 

B: 

C: 

Проблемът идва от това, че трябва да пази обект A 2 пъти в паметта.

D: 

 Трябва да, не можем A()

Решение

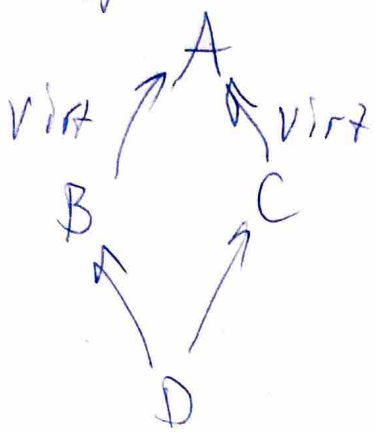
- virtual наследяване
- class X: virtual Y
- Всеки наследник на X е длъжен да каже как да се създаде Y. Предоверя се отговорността за Y надолу.

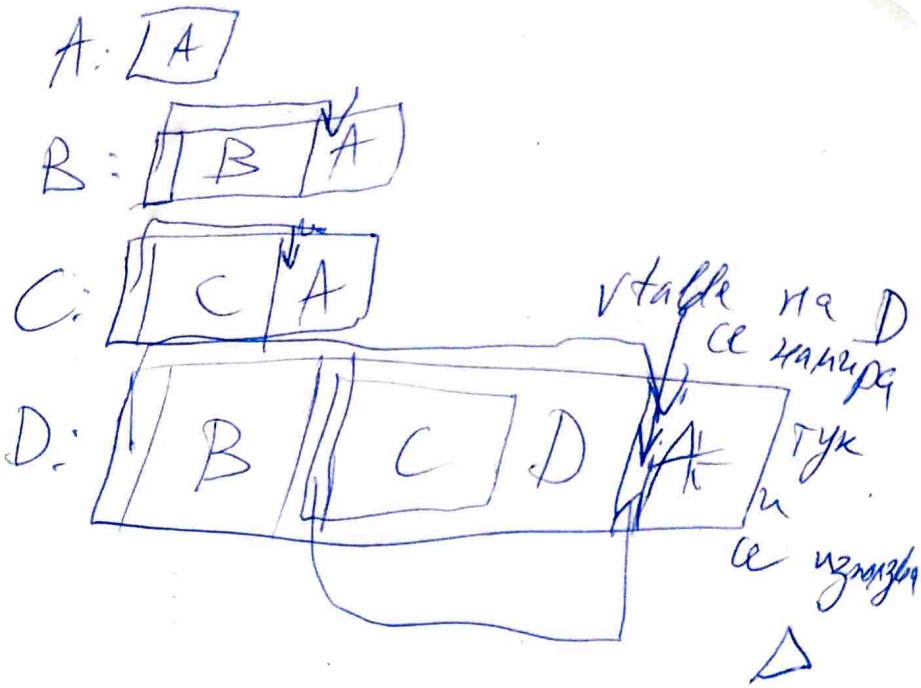
Пример: A
↑ virtual
B
↑
C

B: A(3,7) ;-3

C: B(), A(3,7) Ако се използе ще използва default на A().

Диаграма:



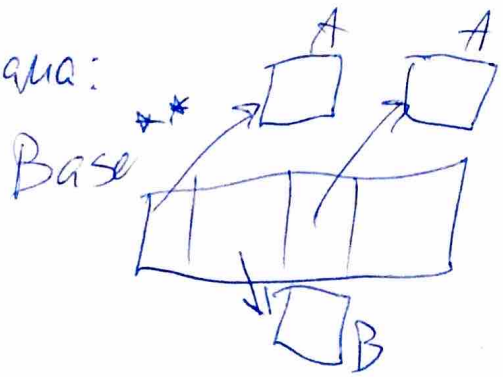


Колекция от обекти в полиморфна йерархия

Използва се хетерогенен контейнер

- обект отговорен за менежиране (създаване, копиране, местене, триене) на обекти от полиморфна йерархия.
- Base** _data;
- За създаването на тези обекти използване factory
- централизирано място за създаване на обекти
- смислово разделен от хетер. контейнер

Диаграма:



Диаграма:

при collection.add(factory::create(str))
няма зависимост между двете

- За копиране
  Използва се clone() ф-цията, на всеки елемент, така не се налага да се разбира конкретния тип.

Пример:

```
copyFrom(const Collection& other)
    _data = new Base*[_size];

    for(0 to _size){
        _data[i] = other._data[i]->clone();
    }
//clone() да има (std::nothrow)
```

- Изтриване
  - Имаме virt destr. в базовия клас на йерархията и се възползваме от него

Пример:

```
for(0 to _size){
    delete _data[i];
}
delete[] _data;
```

Visitor Pattern:

- Behavioural Design Pattern
- разделяне на алгоритмите от обектите върху, които те работят

Base
virt Handle(Base* ptr)
virt Handle A(A* ptr)
virt Handle B(B* ptr)
virt Handle C(C* ptr)

```
f() {
    ptr1 -> Handle(ptr2)
}

A::handle(Base* ptr) {
    ptr -> HandleA(this);
}
```

Всеки нов наследник трябва да имплементира всички интерфейси

Animal.h

```
#pragma once
#include <iostream>

enum class AnimalType
{
    Dog,
    Cat,
    Cow
};
class Animal
{
    AnimalType type;
public:
    Animal(AnimalType type) : type(type){};
    virtual void roar() const = 0;
    virtual Animal* clone() const = 0; //създава копие на текущия об-т
    virtual ~Animal() = default;

    AnimalType getType() const
    {
        return type;
    }

};
```

Cat.h

```
#pragma once
#include "Animal.h"

class Cat : public Animal
{
  public:
      Cat();
      void roar() const override;
      Animal* clone() const override;

};
Cat::Cat(): Animal(AnimalType::cat) {}

void Cat::roar() const
{
   std::cout << "Meow!" << std::endl;
}

Animal* Cat::clone() const
{
   Animal* newObj = new (std::nothrow) Cat(*this); //copy cons
   return newObj;
}
```

Dog.h

```
#pragma once
#include "Animal.h"

class Dog:public Animal
{
  public:
    Dog();
    void roar() const override;
    Animal * clone() const override;
};

Dog::Dog(): Animal(AnimalType::Dog) {}

void Dog::roar() const
{
  std::cout << "Wof!" << std::endl;
}

Animal* Dog::clone() const
{
  return new (std::nothrow) Dog(*this);
}
```

Cow.h

```
#pragma once
#include "Animal.h"

class Cow:public Animal
{
  public:
      Cow();
      void roar() const override;
      Animal* clone() const override;

};
Cow::Cow() : Animal(AnimalType::cow) {}

void  Cow::roar() const
{ std::cout << "Mu !" <<std::endl;

}
Animal* Cow::clone() const
{
   Animal* newObj = new(std::nothrow) Cow(*this);
   return newObj;
}
```

AnimalFactory.h

```
#include Animal.h
#include "Cat.h"   #includ"Dog.h"
  #include "Cow.h"
Animal* animalFactory(AnimalType type);

Animal* animalFactory(AnimalType type)
{
    swith(type)
    {
    case Dog:
        return new Dog();
    case Cow:
        return new Cow();
    case Cat:
        return new Cat();

    }
    return nullptr;
}
```

Farm.h

```
#pragma once
#include "Animal.h"

class Farm
{
private:
    static const size_t DEFAULT_CAP = 8;
    Animal** animals = nullptr;

    size_t animalsCount = 0;
    size_t cap = DEFAULT_CAP;

    void free();
    void copyFrom(const Farm& other);
    void moveFrom(Farm&& other);
    void resize();

public:
    Farm();
    Farm(const Farm& other);
    Farm(Farm&& other) noexcept;

    Farm& operator=(const Farm& other);
    Farm& operator=(Farm&& other) noexcept;

    ~Farm();
    void addAnimal(AnimalType animalType);
    void addAnimal(const Animal& animal);
    void roarAll() const;
    AnimalType getTypeByIndex(unsigned index) const;
};
```

Farm.cpp

```
#include "Farm.h"
#include "AnimalFactory.h"

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

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

void Farm::moveFrom(Farm&& other) noexcept
{
    animals = other.animals;
    cap = other.cap;
    size = other.size;

    other.animals = nullptr;
    other.cap = other.size = 0;
}
```

```
void Farm::resize()
{
    Animal** newAnimals = new Animal*[cap*=2]{nullptr};
    for(size_t i=0; i<size; i++)
    {
        newAnimals[i] = animals[i];
    }
    delete[] animals;
    animals = newAnimals;
}
void Farm::addAnimal(AnimalType animalType)
{
    if(animalsCount==cap)
        resize();
    animals[animalsCount++] = AnimalFactory(animalType);
}
void Farm::addAnimal(const Animal& animal)
{
    if(animalsCount==cap)
        resize();
    animals[animalsCount++] = animal.clone();
}
Farm::Farm()
{
    animals = new Animal*[cap]{nullptr};
}
```

Farm.cpp
N≥2

```
Farm::Farm(const Farm& other)
{
    copyFrom(other);
}

Farm::Farm(Farm&& other) noexcept
{
    moveFrom(std::move(other));
}

Farm& Farm::operator=(const Farm& other)
{
    if(this != other)
    {
        free();
        copyFrom(other);
    }
    return *this;
}

Farm& Farm::op=(Farm&& other) noexcept
{
    if(this != other)
    {
        free()
        moveFrom(std::move(other));
    }
    return *this;
}

Farm::~Farm() {
    free();
}
```

```
void Farm::roarAll() const
{
    for(size_t i=0; i<size; i++)
        animals[i]->roar();
}

AnimalType getTypeByIndex(unsigned index)const
{ if(index >= animalsCount)
        throw std::out_of_range("Index is invalid");
    return animals[i]->getType();
}
```