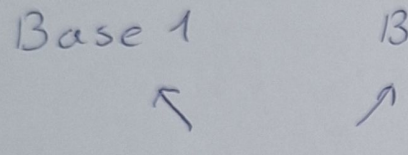
12. Множествено наследяване. Диамантен "проблем". Колекция от обекти в полиморфна йерархия. Копиране и триене. Visitor Pattern.

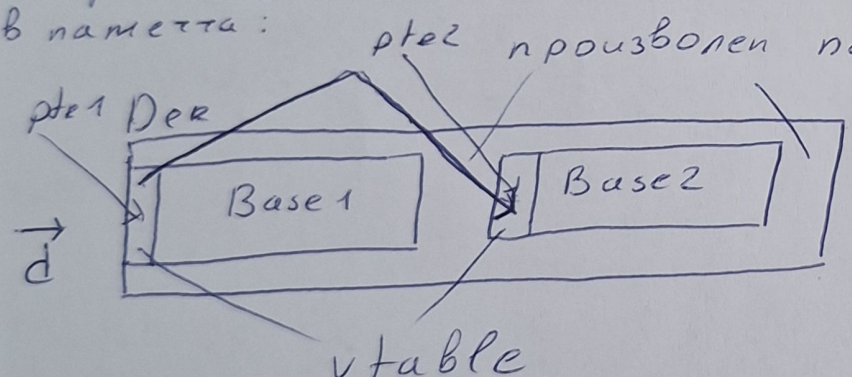
Множествено наследяване

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

Пр:



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(), obj2() {...}

Тук редът винаги е такъв

- ОП =

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

⚠ Проблем на предефинирането на ф-ции

Ако Base1 и Base2

virtual f(); virtual f(); virtual g();

Der

f() override;

// не работи

Ако няма override при извикване на f() ще има Compile Time ERROR.

Решение

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

vtable Base1/Der

Der::f()	0

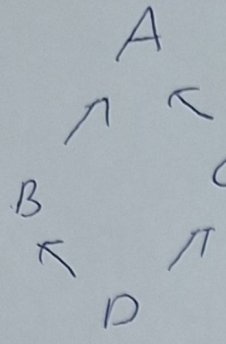
vtable Base2

Der::f()	-Δ(Base2)
Base2::g()	0

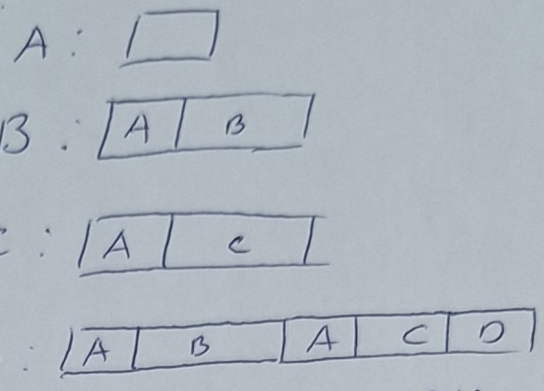
Така ще изглеждат vtables при Der → f() override

Диамантен "проблем"

Искаме да има следната структура:



Диаг:



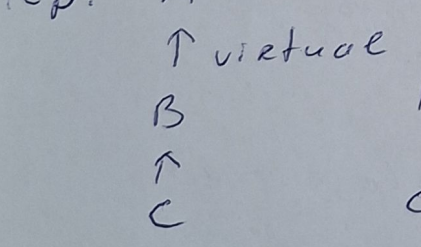
Проблемът идва от това, че трябва да пазим обект A - 2 пъти в паметта.

⚠ Трябва да D() : B(), C(), не можем ~~A()~~

Решение:

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

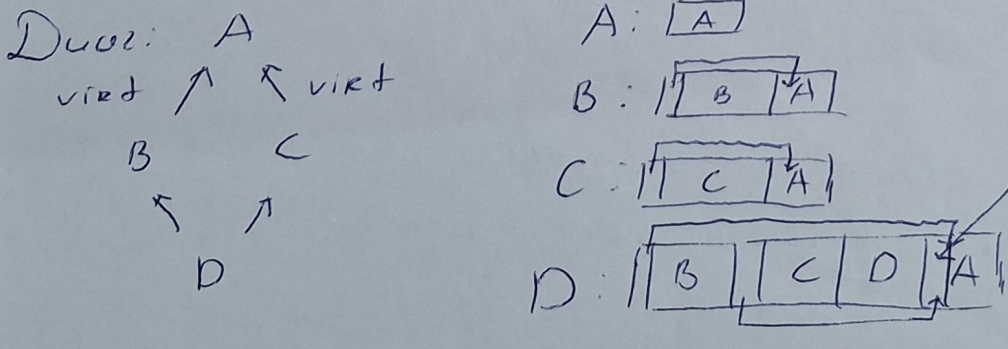
Пр:



B: A(3,7) &...;

C: ~~AB~~ B(), A(3,7) Ако се извика не се извика def A().

Диаг:

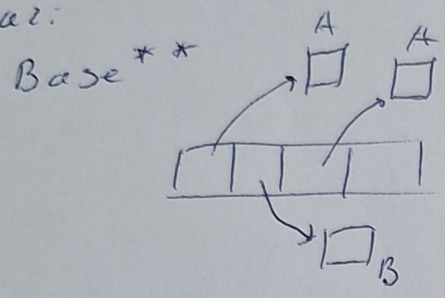


Колекция от обекти в полиморфна йерархия

Използва се <u>хетерогенен контейнер</u>:

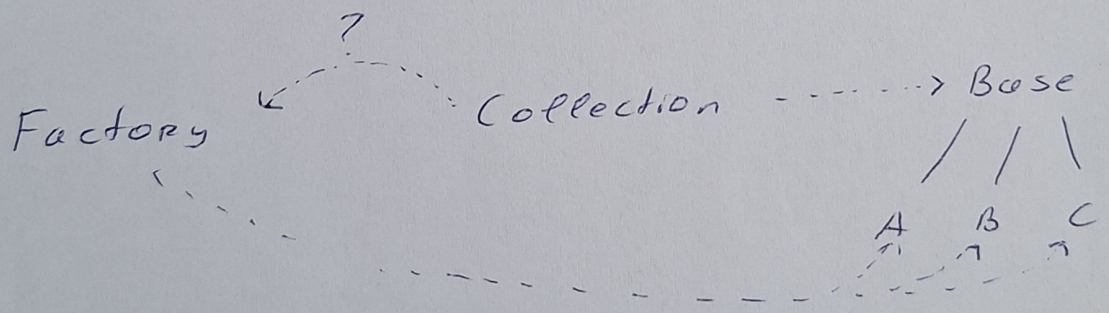
- обект отговорен за менеджиране (създаване, копиране, местене, трие не) на обекти от полиморфна йерархия.
- Base** _data;

Диаг:



- За създаването на тези обекти използване <u>Factory</u>
  - централизирано място за създаване на обекти
  - смислово разделен от хетер. конт.

Диаг:



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

- За копиране

  използва се clone() ф-ията, на всеки елемент, така не се налага да се разбира конкретния тип

Пр:

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

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

- изтриване
  имаме virt dest. в базовия клас на йерархията и се възползваме от него

Пр:

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

Visitor Pattern.
- Behavioral Desig Pattern
- разделяне на алгоритмите от обекти, върху които те работят

```
Base
virt Handle(Base* ptr)
virt HandleA(ptr)
virt HandleB(ptr)
virt HandleC(ptr)
```

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

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

Всеки нов наследник трябва да имплементира всички интеракции.