10. Наследяване. Видове наследяване. Парам на ф-ции (ref и *). Конст и дестр. при наследяване. Копиране при наследяване. Move семантика при наследяване

Деф. наследник

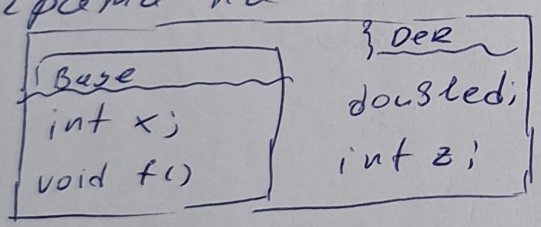
Der е наследник Base, ако разширява неговите данни и/или поведение.

Пр:
```
struct Base {
    int x;
    void f();
};

struct Der : Base {
protected:
    int z;
public:
    double d;
};
```

```
int main() {
    Der obj;
    obj.x;
    obj.f();
    obj.d;
```

Диаграма на паметта



Разлика м/у Композиция и Наследяване.

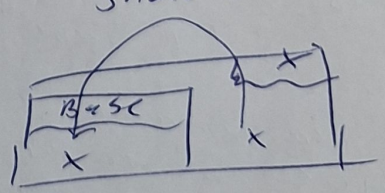
Теоретично няма разлика, идеята е Der да отговаря за частта на Base.

Разликата идва смислово композиция - has a и наследяване - is a

Пр: edge case

```
class X : public Base {
    int x;
    g() {
        x++ // X::x
        Base::x-- // Base::x
    }
};
```

shadow-ва



Видове наследяване

Става чрез модификаторите private, protected - public

```
class Base {
  private:
    int p;
  protected
    double pp;
  public
    short pc;
};
```

и class Der : ___ Base

а) private
p - не е достъпно
pp - private
pc - private

б) protected
p - -||-
pp - protected
pc - protected

в) public
p - -||-
pp - protected
pc - public

3

Параметри на ф-ции.

Можем да правим референтер от базовия клас към наследник, но обратното да за Compile time.

Пр: Base* b1 = new Base(); ✓
Base* b2 = new Der(); ✓

Der* d1 = new Base(); ✗
Der* d2 = new Der(); ✓

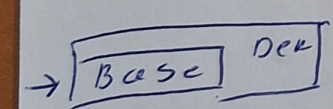
f(const Base&);

f(b1), f(b2), f(d2); ✓

f(const Der&);

f(d2) ✓

Това се получава заради репрезентацията в паметта на Der.

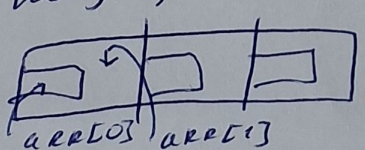


когато има указател от Base, той чете само своята ѝ част и използва vtable в последствие за да използва частта на Der.

Пр: edge case

```
Der arr[3];
f(arr, 3); → f(Base*, size)
```

Не можем да имаме статичен масив от наследници, защото S паметта, arr[..], измества в паметта с определена да кажа големина.



обаче, ако Der има големина, arr[1], ще се счупи.

Конструктор.

```
struct Der : Base {
    Der() {...}
}
```

Ако няма определен конс. на Base се извиква def, но ако Base() = delete; има Compile time.

Пр: edge case

```
class Der : Base
    A a;
    B b;
    C c;
```

Тук ще се извикат Base(), A(), B(), C(), Der();

Деструктори

Викат се тръе отвън навътре

```
struct Der : Base {
    ~Der {..}
}
```

← ще се извикат ~Der(), ~Base())

Копиращ конструктор

```
struct Der : Base {
    Der(const Der& oth) : Base(oth) {
        copyFrom(oth);
    }
}
```

=> : CC of Base(), CC of Der();

В Der имаме отговорност единствено за данните на Der.

1 copyFrom	2 copyFrom

Пр: edge case ← тук ще се извика Base() ако Der(const Der&) { е дефиниран

...

}

4. Конструктор с местене

```
Der( Der&& oth): Base( std::move(oth))?
    moveFrom( std::move(oth));
```

}

Ако Base няма конструктор с местене (Base(Base&&)) се извиква CC of Base();

5. ОП =

```
Der& operator=(const Der&)?
    if(this != &oth)?
        Base::operator=(oth); <- инициализира Base
        free(); <- трие данните само на Der
        copyFrom(oth);
    }
}
```

6. ОМОП =

~~Същото~~ същото е ...