Тема 10

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

Деф. наследяване

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

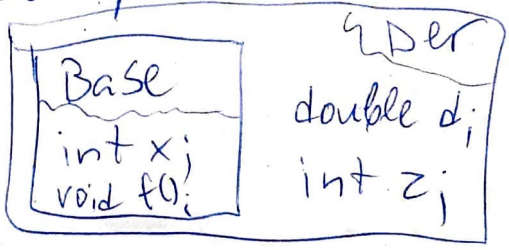
Пример:

```
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



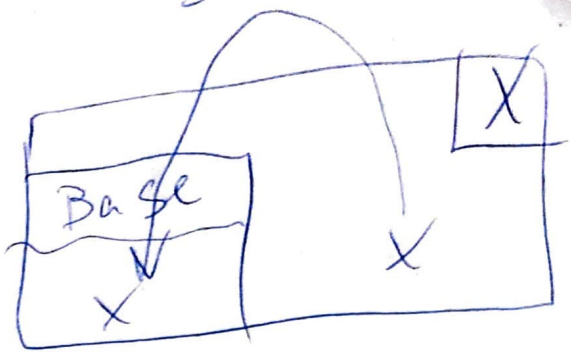
Разлики между Композиция и Наследяване:

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

Разликата идва смислово композицията has a, а наследяване is a

Пример: edge case

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



Видове наследяване
(става чрез модификаторите private, protected и public

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

и class Der: ___ Base

a) private	б) protected	в) public
p-не е достъпно	p-не е дост.	p -II-
pr-private	pr-protected	pr-protelted
pc-private	pc-protelted	pc-public

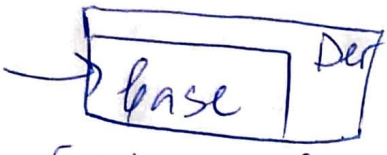
Параметри на ф-ции:
Можем да правим ref/pointer от базовия клас към наследник, но обратното дава Compile time

Пример: Base* b1 = new Base(); √ Der* d1 = new Base(); X
Base* b2 = new Der(); √ Der* d2 = new Der(); √

```
f(const Base &);
f(b1) √  f(d2) √  f(b2) √
```

```
f(const Der &);
f(d2) √
```

това се получава заради репрезентацията в паметта на Der

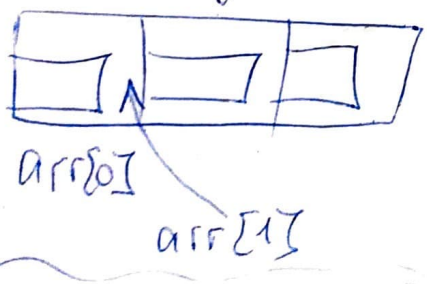


когато има указател от Base, той сочи само своята си част и използва vTable в последствие, за да използва и частта на Der.

Пример: edge case

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

Не можем да имаме статичен масив от ..., наследници, защото в паметта, arr[...], изисква паметта с определена дължина, обаче, ако Der има големина arr[1], уже се спи.



Конструктор

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

Ако няма опредлен конст. на Base се извиква def.

Пример: edge Case

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

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

Деструктори

Винаги се трие отвън навътре

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

← тук се извиква ~Der(), ~Base()

Копиращ конструктор

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

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

В Der имаме отговорност единствено за данните на Der.

1 copyFrom	2. copyFrom

Пример: edge case

```
Der(const Der&) {}
```

тук ще се извика Base() ако е дефиниран

4. Копиране с местене

```
Der(Der&& other):Base(std::move(other)){
    moveFrom(std::move(other));
}
```

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

5. ОП=

```
Der& operator=(const Der&)
{
    if(this != &other)
    {
        Base::operator=(other); ← инициализира Base
        free(); ← трие данните само за Der
        copyFrom(other);
    }
```

6. МОП=

Същото, но с std::move(other)

Person.h

```
#pragma once
#include <iostream>

class Person
{
    char* name = nullptr;
    int age = 0;

    void copyFrom(const Person& other);
    void free();
    void moveFrom(Person&& other);

public:
    Person() = default;
    Person(const char* name, int age);

    Person(const Person& other);
    Person(Person&& other) noexcept;

    Person& operator=(const Person& other);
    Person& operator=(Person&& other) noexcept;
    ~Person();

    const char* getName() const;
    int getAge() const;
    void print() const;

protected:
    void setName(const char* name);
    void setAge(int age);
};
```

Person.cpp

```
#include "Person.h"
#pragma warning(disable:4996)

void Person::copyFrom(const Person& other)
{
    age = other.age;
    name = new char[strlen(other.name)+1];
    strcpy(name, other.name);
}

void Person::free()
{
    delete[] name;
    name = nullptr;
    age=0;
}
void Person::moveFrom(Person&& other)
{
    name = other.name;
    age=other.age;
    other.name = nullptr;
    other.age=0;
}
Person::Person(const char* name, int age)
{
    setName(name);
    setAge(age);
}
Person::Person(const Person& other)
{
    copyFrom(other);
}
Person::Person(Person&& other)noexcept
{
    moveFrom(std::move(other));
}
Person& Person::op=(const Person& other)
{
    if(this!=&other)
    {
        free()
        copyFrom(other);
    }
    return *this;
}
```

```
Person& Person::op=(Person&& other) noexcept
{ if(this!=&other)
  { free();
    moveFrom(std::move(other));
  }
  return *this;
}
Person::~Person()
{ free();
}
const char* Person::getName() const
{ return name;
}
int Person::getAge() const
{ return age;
}
void Person::print() const
{ std::cout<<name<<" "<<age<<std::endl;
}
void Person::setName(const char* name)
{ if(!name)
    throw std::invalid_argument("Name is nullptr");
  if(this->name==name)
    return;
  delete[] this->name;
  this->name = new char[strlen(name)+1];
  strcpy(this->name, name);
}
void Person::setAge(int age)
{ this->age = age;
}
```

Teacher.h

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

class Teacher: public Person
{
    char** subjects = nullptr;
    size_t subjectsCount = 0;

    void free();
    void copyFrom(const Teacher& other);
    void moveFrom(Teacher&& other);

public:
    Teacher(const char* name, int age, const char* const*, size_t subjCount);
    Teacher(const Teacher& other);
    Teacher(Teacher&& other) noexcept;

    Teacher& op=(const Teacher&);
    Teacher& op=(Teacher&&) noexcept;

    ~Teacher();
};
```

Teacher.cpp

```
#include "Teacher.h"
#pragma warning(disable:4996)
static char** copyArrayOfStrings(const char* const strings, size_t size)
{
    char** result = new char*[size];
    for(size_t i=0; i<size; i++)
    {
        result[i] = new char[strlen(strings[i])+1];
        strcpy(result[i], strings[i]);
    }
    return result;
}
void Teacher::free()
{
    for(size_t i=0; i<subjectsCount; i++)
    {
        delete[] subjects[i];
        subjects[i] = nullptr;
    }
    delete[] subjects;
    subjects = nullptr;
    subjectsCount = 0;
}
void Teacher::copyFrom(const Teacher& other)
{
    this->subjects = copyArrayOfStrings(other.subjects, other.subjectsCount);
    this->subjectsCount = other.subjectsCount;
}
void Teacher::moveFrom(Teacher&& other) noexcept
{
    subjects = other.subjects;
    subjCount = other.subjCount;
    other.subjects = nullptr; other.subjCount = 0;
}
```

```
Teacher::Teacher(const char* name, int age, const char* const* subjects, size_t subjCount)
    : Person(name, age)
{
    if (!name || !subjects)
        throw std::invalid_argument("Invalid name or invalid subjects");
    this->subjects = copyArrOfStrings(subjects, subjCount);
    this->subjectsCount = subjCount;
}

Teacher::Teacher(const Teacher& other) : Person(other)
{
    copyFrom(other);
}

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

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

Teacher& Teacher::operator=(Teacher&& other)
{
    if (this != &other)
    {
        Person::operator=(std::move(other));
        free();
        moveFrom(std::move(other));
    }
    return *this;
}

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

Student.h

```cpp
#pragma once
#include <iostream>
#include "Person.h"
class Student:public Person
{
    size_t fn=0;
public:
    Student()=default;
    Student(const char* name,int age,size_t fn);

    size_t getFn() const;
    void setFn(size_t fn);
};
Student::Student(const char*...):Person(name,age)
{
    setFn(fn);
}
size_t Student::getFn() const
{ return fn;
}
void Student::setFn(size_t fn)
{
    if(fn<100)
        fn=100;
    this->fn=fn;
}
```