* Mini Tekrar:
- Hem composition, hem inheritance da nesne içinde başka bir nesne var
- Derived class nasıl yaratılır onun içindeki base class da nasıl yaratılır. Önce derived içindeki base
↓
Sonra derived içindeki member
↓
Sonra derived ctor çağırılır.

```
12  class Base
13  {
14  public:
15      Base()
16      {
17          std::cout<< "Base constructor\n";
18      }
19      ~Base()
20      {
21          std::cout<< "Base destructor\n";
22      }
23
24  };
25
26
27  class Der : public Base
28  {
29  public:
30      Der()
31      {
32          std::cout<< "Der constructor\n";
33      }
34      ~Der()
35      {
36          std::cout<< "Der destructor\n";
37      }
38
39  };
40
41
42  int main()
43  {
44      Der myDer;
45      return 0;
46  }
```

```
Base constructor
Der constructor
Der destructor
Base destructor
```

Sırası önemli, sonra gelen ilk çıkar.

- Sıra olarak 1. eğer varsa base member
2. base ctor
3. eğer varsa derived member
4. derived ctor
5. ve tam ters sırada destructors

```
class Base {
public:
    Base(int x)
    {
        std::cout << "Base(int x) x = " << x << "\n";
    }

};

class Der : public Base {
public:
    Der(int i) : Base(i)
    {

    }

};

int main()
{
    Der myder(12);
}
```

Burada Base default ctor değil

Base(i) → kime parametre ile olurdu Base'e

Der ctor böyle olmalı. yoksa syntax hatası

Böyle olmazsa Der'in default ctor'u girmez.

• Multi level Inheritance:

```
class Base {
public:
    Base(int);
};

class Der : public Base {
public:
};

class SDer : public Der {

public:
    SDer(): Base(0) {}
};
```

SDer, direkt olarak Base'i böyle init edemez

• Copy Constructor:

```
class Base {
public:
    Base()
    {
        std::cout << "Base default ctor\n";
    }
    Base(const Base&)
    {
        std::cout << "Base copy ctor\n";
    }
};

class Der : public Base {
};

int main()
{
    Der der1;

    Der der2 = der1;
}
```

→ Derived class'ın copy constructor'ını compiler default'etti.

```
Microsoft Visual Studio Debug Console
Base default ctor
Base copy ctor

D:\KURSLAR\MART2022\Release\MART2022.e
Press any key to close this window . .
```

•Fakat: Biz kendi elimizle Der için copy ctor yazsaydık işler değişirdi.

↓

Bu durumda der, copy construct edilse bile, Default ctor çağırılırdı!

• Eğer Copy Constructor'ı Derived için bir yazıyorsak:

```
class Der : public Base {
public:
    Der() = default;
    Der(const Der&other) : Base(other)
    {

    }

};


int main()
{
    Der der1;

    Der der2 = der1;
}
```

Bu şekilde eklemiş oluruz

• Move Constructor: → Copy ctor'daki durumun aynısı, move ctor için de geçerli

```
class Base {
public:

};


class Der : public Base {
public:
    Der() = default;
    Der(Der&&other) : Base(std::move(_Arg:other)) {
        //
    }
};


int main()
{
    Der der1;

    Der der2 = std::move(_Arg:der1);
}
```

bu yüzden olmak zorunda

• Copy Assignment:

```
class Base {
public:
    Base& operator=(const Base&)
    {
        std::cout << "Base copy assignment\n";
        return *this;
    }
};


class Der : public Base {
public:
    Der& operator=(const Der&other)
    {
        Base::operator=(other);
        return *this;
    }
};


int main()
{
    Der x, y;

    x = y;
}
```

bunu tercih et.

• Hiç bir şekilde Base belirtilmemesine rağmen, base copy assignment böyle değiştirilebilir.

↳ x=y işlemi de Der tercihi

• Alternatif olarak şöyle bir upcast yapılabilir.

```
class Der : public Base {
public:
    Der& operator=(const Der&other)
    {
        *(Base*)this = other;

        return *this;
    }
};
```

→ Çok genel bir özet yaptı yani ders

* Move Assignment:

```
public:
    Base& operator=(Base&&)
    {
        std::cout << "Base move assignment\n";
        return *this;
    }
};


class Der : public Base {
public:
    Der& operator=(Der&& other)
    {
        Base::operator=(std::move(other));
        return *this;
    }
```

→ Hatırlatma:

```
eğer şu 3 işlevden biri bildirilmiş ise derleyici sınıfın move member'larını yazmaz

copy ctor
move ctor
destructor

eğer şu 2 işlevden biri bildirilmiş ise derleyici sınıfın copyt member'larını delete eder

move ctor
move assignment
```

* Run-Time Polymorphism:

- concrete class → somut sınıf
- abstract class → soyut sınıf → base class interface verir. → Biz sürekli yazmak zorundayız.
- polymorphic class → base class interface verir. Biz istersek override ederiz, kod

```
class Airplane {
public:
    void takeoff();
    virtual void land();  //virtual functionb
    virtual void fly() = 0; //pure virtual function
};
```

böyle fonksiyonlardan 1 tane bile varsa bu polymorphic class

böyle fonc 1 tane bile varsa abstract class

• Function Overriding:

→ Eğer hangi fonksiyonun çağırıldığı run time'da belirlenirse late binding. Function override temelden late binding

```
class Base {
public:
    virtual void vfunc();
};


class Der : public Base {
public:
    void vfunc();
};
```

→ Ayrı parametre ve return değerinde olmalı

- const bildirim de imzanın parçası override eden de const bulunmalı

→ Derived class'ın vfunc'ını virtual olarak tanımak da tanımasak da override eder.

→ Belirtimle zorunda değil.

→ Virtual Dispatch:

- Taban sınıf pointer, türemiş sınıf nesnesini gösterebilir referans " " " bağlanabilir.

```
class Der : public Base {
public:
    void vfunc()
    {
        std::cout << "Der::vfunc()\n";
    }
};

int main()
{
    Der myder;

    Base* baseptr = &myder;

    baseptr->vfunc();
}
```

→ Bu derived class vfunc / Compiler gözetti çağırdı.

```
int main()
{
    Der myder;

    Base& baseref = myder;

    baseref.vfunc();
}
```

→ referans semantiği ile bu şekilde

- Fakat fonksiyon, base nesnesinin kendisiyle çağırılırsa virtual dispatch devreye girmez.
  → virtual dispatch sadece pointer ve referansla çağırılır.

→ bu olaya object slicing'e sebep verir.