

• Tekrar: Run time'da belli olan program **late binding**

**dynamic binding**

Taban sınıfın pointeri / referansı ile **virtual dispatch** diyeceğiz.

• Modern C++'da **contextual keyword** olan **override** eklendi

```
20
21
22 class BMW : public Car {
23 public:
24     void start() override {
25         std::cout << "BMW has just started!\n";
26     }
27     void run() {
28         std::cout << "BMW is running now!\n";
29     }
30     void stop() {
31         std::cout << "BMW has stopped!\n";
32     }
33 }
34
```

```
class Base {
public:
    //...
    void func(int);
};
```

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

• **Logic hata:** virtual olmayan bir fonksiyon, override edilmeye çalışılmış. Taban sınıf fonksiyonu çağırılır.

```
class Base {
public:
    virtual void func(unsigned int);
};
```

```
class Der : public Base {
public:
    void func(int) override;
};
```

• **Logic hata:** İmzası farklı bir override yapılmış. Eğer **override** keyword'ü olmazdı **syntax hata**sı olmazdı.

→ **Override keyword** eklendiği için: Artık **syntax hata**sı override compiler'a kontrol ettiriyor. Bu override legal mi diye.

• RTTI (Run Time Type Information / Identification):

- Run time'da oluşacak sınıfın hangi derived class olduğunu anlamaya yarayan araç

\* Her kavramına sahip dillerde **static typing** olmak üzere iki adet tür vardır. **dynamic typing**

**static typing** → verinin türüne göre bir süreçte  
**dynamic typing** → " " " " run time'da

`void car_game(Car& cr)` → Compiler açısından cr, car türünden

```
{
    cr.start();
    cr.run();
    cr.stop();
    std::cout << "-----"
```

→ Ama run time'da neye göre  
biri orada olabilir

→ Dynamic type olması için polymorphic class  
olmalıdır.

- Dynamic: dinamik
- static: derlemeden, tanımlanmış, kontrolüne geçiş

Virtual Dispatchin Çalışması: Parametre:

\* Konsistans:

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

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

→ Private, Override edilebilir.

• Aynı şekilde derived class, private'da da Override eder.

→ Override'in Access Control'e tabi olması gerekir. Access Control compile time'tan

→ Verilen Argüman Compile Time'a ilişkin bir metakomutur. Bu yüzden

```
class Base {
public:
    virtual void vfunc(int x = 10)
    {
        std::cout << "Base::vfunc(int x) x = " << x << "\n";
    }
};

class Der : public Base {
public:
    void vfunc(int x = 77) override
    {
        std::cout << "Der::vfunc(int x) x = " << x << "\n";
    }
};
```

→ Der vfunc, 10 değeriyle çağırılır.

→ Compiler, default argümanı 10 olarak alır.

Biz Override edilmis rolüne ihtiyarsak:

```
int main()
{
    Der myder;
    myder.vfunc();
}
```

→ myder.vfunc' olarak çağırılmalı  
bu statik tir.

- \* Name lookup → Compile time / static type
- \* Access Control → Compile time / static type
- \* Default Argument → Compile time / static type

```

1 class Base {
2 public:
3     Base()
4     {
5         vfunc();
6     }
7 private:
8     virtual void vfunc()
9     {
10         std::cout << "Base::vfunc()\n";
11     }
12 };
13
14 class Der : public Base {
15 public:
16     void vfunc() override
17     {
18         std::cout << "Der::vfunc()\n";
19     }
20 };
21
22 int main()
23 {
24     Der myder;
25 }

```

→ Override edilmeyen fonksiyonlar, Override edilebilen fonksiyonları çağırır.  
 → Aneke constructor'da durum farklı

• Fakt: Virtual dispatch devreye girmez.

→ Çeşitli hayata gelme sırasında önce base class constructor çağırılır. Henüz derived class hayata gelmedi.

→ Destructor'da da virtual dispatch devreye girmez.

• Neden böyle? Aneke içinde virtual çağırılır

```

class Base {
public:
    void foo(Base *p)
    {
        vfunc();
    }
}

```

- Çünkü assembly düzeyinde fonksiyonun parametresi olmase bile aslında Base \* p var.

→ Özetle: \* base object (pointer veya ref degi) direkt kendi adıyla fonksiyon çağırır

\* base class in içindeki referanslı fonksiyonların çağırılmasında

```

class Base {
public:
    void foo()
    {
        Base::vfunc();
        vfunc();
    }
}

```

→ önce base  
 → sonra virtual

= Virtual dispatch devreye girmez !!

- \* base class constructor da,
- \* base class destructor da,

• Sınıf nesnesinin içinde sonal olmayan fonksiyonlar var. kaplamaz !!

sonal fonksiyonlar ise = Burada durum farklı. Sonal fonksiyonlar → class'ı polymorphic class yapar. Polymorphic classlar da (temel veri elemanları + 1 adet pointer) sahiptir.

- Bu pointer vptr / sonal fonksiyon tablosu gösterici



## • Virtual Function Table :

```
0
1  &Toyota::start
2  &Toyota::run
3  &Toyota::stop
```

virtual function table for class Dacia

```
0
1  &Dacia::start
2  &Dacia::run
3  &Dacia::stop
```

- Aynı hiyerarşideki sanal fonksiyonlar için, compile time da oluşan tablo

- Bu indexler, her class için aynı.

```
Car *carptr;

carptr->run();

carptr->vptr[1]();
```

derleyicinin oluşturduğu kod nit. eder.

- Overload edilmişse, tabloya base class'ın ki eklenir.

- maaliyet: vptr'ye erişmek ve vptr[?] gibi dereferans yapması

=====

her polimorfik nesne için vptr'nin initialize edilmesi  
her sanal işlev çağrısı için (derleyici optimizasyon yapmıyor ise)  
2 x dereferencing

her polimorfik sınıf için sanal fonksiyon tablosu veri yapısı oluşturuluyor

her polimorfik nesne için bir pointer denen birle olacaktı. VE, bellekte yer kaplayan fonksiyon tablosu

## - Pure virtual Function Tanımı:

```
class Car {  
public:  
    virtual void start() = 0;  
  
    virtual void run() = 0;  
  
    virtual void stop() = 0;  
};
```

- Car mx; gibi **instanste edilemez X**

- Derived class **concrete class** olmak için **hepsini implement etmelidir**. Yoksa, o da **abstract class** olur.

\* Constructor lar **virtual olamaz X**. Syntax hatası. Onun yerine **virtual constructor idiom / clone constructor idiom** var

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

→ Clone constructor idiom:

```
class Toyota : public Car {  
public:  
    Car* clone() override  
    {  
        return new Toyota(*this);  
    }  
};
```

```
void car_game(Car* p)  
{  
    Car* carptr = p->clone();  
    carptr->start();  
    p->start();  
    p->run();  
    p->stop();  
}
```

→ Clone ile kendinden bir adet daha yaratır

\* Destructor **virtual olabilir ✓**. Çoğu zaman olmak zorunda. Yoksa, taban sınıf nesnesinin **pointer** ile, **taban sınıf** nesnesi **delete** edilirken → **undefined behavior**.

→ ya public virtual  
ya da non-virtual protected tanımlı **polimorfik sınıflar** } → **nesnenin yaşamını ok**