

## \* Teremis Sınıflarla Using Birlikte:

```
class Base {
public:
    virtual ~Base() = default;
    void func(int);
    void func(long);
    void func(char);
};

class Der : public Base {
public:
    using Base::func; ]> base'in tümメンバーを引き継ぐ。
    void func(int, int);
};

int main()
{
    Der myder;

    myder.func(12); ]> From Base
    myder.func(12L); I
    myder.func('A');
    myder.func(12, 34); ]> From Der
}
```

→ Ayni şekilde, protected memberler da using ile derived class'a erişebilir.

```
#include <iostream>

class Base {
protected:
    using value_type = int;
}; ]> type alias
                    olusturmak icin

class Der : public Base {
public:
    using Base::value_type;
}; ]> isim erisite etmek icin

int main()
{
    Der myder;

    Der::value_type x;
}
```

## \* Inherited Constructor:

→ Constructor initialization first rule, base class'a boglayabilmeli.

```
Der.h
{
    std::cout << "Base(int x) x = " << x << "\n";
}

Base(int x, int y)
{
    std::cout << "Base(int x, int y) x = " << x << " y = " << y << "\n";
}

void func(int);
void foo(int);
void g();
void bar(double);
};

class Der : public Base {
public:
    Der(int x) : Base(x) {}
    Der(int a, int b) : Base(a, b)
    {
    }
    void baz();
};
```

Fakat modern C++ da yani = inherited constructor.  
bir tane olurdu

```
Der.h
{
    void func(int);
    void foo(int);
    void g();
    void bar(double);
};

class Der : public Base {
public:
    using Base::Base;
    void baz();
};

int main()
{
    Der myder1(12);
    Der myder2(1, 2);
}
```

→ Using Base::Base, derived tem constructorun insert etti.

• Fakat. 4 olmazsa Public bloklarla constructorlarda, odur. ✓  
Protected daysa collision olurdu X

## \* Final Keyword:

- contextual keyword
- fonksiyonun sonra koltum yoplumasını engeller.

```

#include <iostream>

class Base {
};

class Der final : public Base {
};

class SDer : public Der {
};

```

→ Bu da final class olmaz.

→ Birer seerde final override da ver:

```

void start()final override // override final da yarlısolur.
{
    std::cout << "Volvo has started!\n";
}
void run()override
{
    std::cout << "Volvo is running now!\n";
}
void stop()override
{
    std::cout << "Volvo has stopped!\n";
}

```

→ Artırıcı classları terefen classlar, bu fonksiyon override edemez X  
Base class'ın kullanılır ✓

→ Const işaretleri sadece herhangi  
birde yapılmamalı

- const final override
- const override final ...

- Deçleyiciler, final override garantisi, virtual dispatch yapmak yerine, de-virtualization (compile time da hangi funkcı olursun bilmek) ve basımlı.

\* Kullanılma: Polymorphic sınıfların abstractoları → public virtual  
olmalı. Genel tohumlu sınıf nesneleri, tozun sınıf  
→ protected non virtual  
pointer'i ne yönetir ve aynı şekilde abstract edilir. → Taban sınıf pointer'i ile derived class deletion, derived  
dövremi ist.

\* Double Dispatch:

```

void collide(Car* p1, Car* p2)
{
}

```

→ hangi kodun çalışacağı  
artırıcı 1 değil, 2 adet car  
bağımı başlı.

\* Global bir fonksiyon nasıl işaretlendirir?:

```
class Car {  
public:  
    virtual void print(std::ostream&) const = 0;  
    virtual Car* clone() = 0;  
    virtual void start() = 0;  
    virtual void run() = 0;  
    virtual void stop() = 0;  
    virtual ~Car() = default;  
};  
  
class Bmw : public Car {  
public:  
    virtual void print(std::ostream& os) const override  
    {  
        return os << "I am a BMW\n";  
    }  
    ~Bmw() override  
};
```

→ virtual bir fonksiyon, global bir fonksiyon oluşturmak, global fonksiyon override etmeli gibi sonuc elde edebilir.

```
inline std::ostream& operator<<(std::ostream& os, const Car& other)  
{  
    other.print(os);  
    return os;  
}
```

\* Covariance / Variant Return Type:

- Override olması için, fonksiyon imzası ve return değeri tari → aynı olmalı.

- taban sınıfın fonksiyonunu override eden türün sınıfının fonksiyonunun return type'i, base ile aynı olmalı. istisna olarak covariant

- Base 1 returns Base 2 class \*      /      Derived 1 returns Derived 2 class \*  
Base 2 class 8                          |      Derived 2 class 8

```
class B {  
};  
  
class D : public B {  
};  
  
class Base {  
public:  
    virtual B* foo();  
    virtual B& bar();  
};  
  
class Der : public Base {  
public:  
    D* foo() override;  
    D& bar() override;  
};
```

→ Fakat bu yollarda pointer ve referans tari geçerli.

```
int main()  
{  
    Der myder;  
  
    D* derptr = static_cast<D*>(myder.foo());
```

Der'in, foo'su eğer Base\* olsaydı  
hata, type-cost uygulanır.

→ auto & l value ref defl. Forwarding referansı / universal reference

### \* Polymorphic Listeler:

- Pointer senesinde ile okutulabilir → **onectives X**
- Reference wrapper
- Smart pointer

```
int main()
{
    Car* a[size];

    for (auto &p : a) {
        p = create_random_car();
    }

    //

    for (auto p : a) {
        std::cout << *p;
        p->start();
        p->run();
        p->stop();
        delete p;
    }
}
```

→ outside base class tener bit orada söyleyebiliriz artı.

### \* RTTI:

→ Derived Class, Bağışının daha azını sağlamamalı → **promise no-less**

→ Tersenis sınıfları overrideları, token sınıfları limitesini kullanamalı → **require no-more**

```
void car_game(Car* p)
{
    std::cout << *p;
    p->start();
    p->run();
    //eger araba volvo ise cam tavani acilsin
    p->stop();
}

int main()
{
    for (int i = 0; i < 100; ++i) {
        Car* p = create_random_car();
        car_game(p);
        delete p;
    }
}
```

→ Bu dynamic cast ile yapabiliriz.

**Notlar:** Upcasting: Tersenis sınıftan → Base Class'a örtük  
Downcasting: Base Class'tan → Derived Class'a

## \* Dynamic Cast:

- Run-time polymorfizmini ve dynamic cast yapılıp, yapılmayacağını iki kere bir arada.

```
3
4
5
6
7
8
9 void car_game(Car* p)
10 {
11     std::cout << *p;
12     p->start();
13     p->run();
14     //Volvo* vp = dynamic_cast<Volvo*>(p); ]> Nur sevde de tenitilabilir.
15     auto vp = dynamic_cast<Volvo*>(p);
16     if (vp) { → Dynamic cast sonucu: Castlenen class adıhar
17         vp->open_sunroof(); Nullpointer.
18         getchar();
19
20     }
21     p->stop();
22 }
```

Foket Bu Scope Local'de olsa da okunabilir

```
if (auto vp = dynamic_cast<Volvo*>(p); vp) { → Daha kolonlu: ✓
    vp->open_sunroof();
    (void)getchar();
}
```

\* Dikkat: Dynamic Cast yalnızca Polymorphic Sınıfları kullanılır.