*** Tekrar:** - Override Specifier

```cpp
class Base {
public:
    virtual void foo(int);
};

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

→ Der, foo'nun override'ı → peki neden override specifier'ı kullanıyoruz?

1.

```cpp
public:
    virtual void foo(unsigned int);
};

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

→ return değeri ve parametreleri aynı olmalı!
→ Burada virtual dispatch devreye girmez X
↳ override specifier kullansaydık, bu hatayı bize söylerdi!

2.

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

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

→ Fonksiyon virtual değil!, inheritance zaten yok!

3.

Taban sınıfın virtual fonksiyonu değişti. Takip edilmesi zor durumlar olabilir. Ayrıca information hiding de olacak

*** move ve noexcept'ten duvam:**

- noexcept = nothrow garantisi. Fakat nothrow garantisi verilemeyen koda, noexcept demek hatalı bir durum } → Eğer exception throw edilirse terminate çağrılır!
- eğer move noexcept olmazsa, strong guarantee gerektiren fonksiyonlarda (vektör push back gibi..) seçilemez X Kopyalama seçilir anlık bazı verim farkı olacak!

*** moved From State:**

- L value referanslar move edildikten sonra bu state'e gelir! Derleyici bu state için bir takım garantiler sunar
  1. Destructable olması
  2. unspecified but valid olması → değerini bilmiyoruz ama hala eskisi gibi kullanabiliriz.

*** Invariant:**

- Nesnenin kullanılabilir durumda olması, sınıfın invariantlarını tutuyor olması demek
  (invariant: her zaman doğru olması gereken durumlar!)

→ Çünkü standart kütüphane öğelerinin bu garantisi olsa da, derleyicinin yazdığı move memberlar, her zaman bizim nesnemiz için invariantlığını koruyamayabilir!
  ↳ bizim nesnemizde data memberlar birbirine bağımlı olabilir.
  - Default constructable değildir
  - Pointers / smart pointers ....

```cpp
class SharedInt {
public:
    explicit SharedInt(int val) : m_sp{ std::make_shared<int>(val) } {  }

    std::string as_string() const
    {
        return std::to_string(*m_sp);
    }
private:
    std::shared_ptr<int> m_sp;
};


int main()
{
    SharedInt x{ 20 };
    SharedInt y{ x }; //

    std::cout << x.as_string() << '\n';
    std::cout << y.as_string() << '\n';

    SharedInt a{ 20 };
    SharedInt b{ std::move(a) };

    std::cout << a.as_string() << '\n';

}
```

```cpp
class Card {
public:
    Card(const std::string& val) : m_val{ val }
    {
        //assertValidCard(value); // ensure the value is always valid
    }

    std::string get_value() const
    {
        return m_val;
    }
private:
    std::string m_val;
};

int main()
{
    Card c1{ "king_of_hearts" };

    auto c2 = std::move(c1);

    std::cout << c1.get_value() << '\n';
}
```

bu bir invariant

→ Bir herhangi bir test nesnesinin ←
taşınmış olsa bile,
bir değerinin olmasını
istiyoruz!

ya da 2. alternatif
bu fonksiyonu modifiye
edebilirdik!
- string yerine optional döndürürdü

Fakat derleyicinin yazdığı move ctor bizim invariantımıza
uygun değil!

Boş string döndürdü!

* Move Semantics in Generic Programming:

→ template code larda, çağrı zaman fonksiyon tipi run time da belli olurken, move yapılıp yapılmayacağını kestirmek zor!

* Universal Reference / Perfect Forwarding:

- Modern Cpp de 3 çeşit referans bulunur: - L value ref
  - R value ref
  - Universal / Forwarding ref

```
6  template <typename T>
7  void func(T&&);
8
9  int main()
10 {
11
12 }
```

→ const ve volatile olmadan!

```
int main()
{
    auto &&r =
}
```

```
void foo(auto&& x);
```

→ Cpp 20 : Abbreviated Template Syntax

→ Universal reference'a non-const / const her nesne bağlanabilir.

```
template <typename T>
void foo(T&&);

template <typename T>
void func(const T&);
```

Bu iki fonksiyon şablonunu istediğimiz her nesneyi gönderebiliriz. Tuhaf fark!

Const L value reference

→ Aralarındaki fark: Const L value referans'a argüman gönderdiğimizde, gönderdiğimiz argümanın, hem value category'sini hem de constluğunu fonksiyon bizzat içinde bozabiliriz

→ Universal reference, o bilgiyi tutar.

```
template <typename T>
void foo(T&& arg)
{

}
```

Perfect forwarding'de type deduction

→ Effective modern Cpp kitabında detaylı açıklama var!

```
Myclass m;
foo(m)
      T          arg
   Myclass&   Myclass&

foo(Myclass{})
      Myclass  Myclass&&
```

referans collapsing

→ non const L value

→ Non-const R value

```
const Myclass m;
      T                arg
   const Myclass&   const Myclass&
foo(m)
```

→ Const L value

* Sadece T'nin türüne / constuctora bakarak değil, arg'a bakarak da bu bilgiye ulaşabiliriz. Bunu anlamak neler yapabiliriz?

```
template <typename T>
void foo(T&& arg)
{
    // eğer T türü referans türü ise
    bar(arg)
    // eğer T türü referans türü değil ise
    bar(std::move(arg))
}
```

Bunu yapmayı kolaylaştıran: std::forward:

```
template <typename T>
void foo(T&& arg)
{
    std::forward<T>(arg)
}
```

arg eğer, L value referans ise → L value expression
" " R value referans ise → R value expression

```
template <typename T>
void foo(T&& arg)
{
    bar(std::forward<T>(arg))
}
```

perfect forwarding!!

→ Perfect Forwarding'e nerelerde ihtiyaç duyarız:

- Containerların "emplace" fonksiyonları

Çünkü emplace'e containerda saklanacak objenin constructor'ına geçilmesi gereken argümanları veririz!

→ varsa (val cat, const gibi) perfect forward ederek doğru seçimi yaptırırız!

- "make_unique" fonksiyonu

emplacedeki gibi make_unique'in new ile hayata getireceği nesnenin constructorına argüman yollarız.

→ Fonksiyonun argüman sayısına göre ilgili olacak yazdığımız overload alternatifleri

```
foo(x, y)

call_foo(x, y)


template<typename T, typename U>
void call_foo(T &&t, U &&u)
{
    foo(std::forward<T>(t), std::forward<U>(u));
}
```

böyle bir call-foo ile istediğimiz foo overloadına, istediğimiz argümanı gönderebiliriz

```
template<typename ...Args>
void call_foo(Args&& ...args)
{
    foo(std::forward<Args>(args)...);
}
```

variadic template alternative

```
template <typename T>
void call_func(T&& arg)
{
    func(std::forward<decltype(arg)>(t));
}
```

burayı T yerine decltype(arg) ✓

```
auto fn = [](auto&& r) {
    func(std::forward<decltype(r)>(r));
};
```

→ Lambda expression alternative

→ önemli ←

Call	f(X&)	f(const X&)	f(X&&)	f(const X&&)	template<typename T> f(T&&)
f(v)	1	3	no	no	2
f(c)	no	1	no	no	2
f(X{})	no	4	1	3	2
f(move(v))	no	4	1	3	2
f(move(c))	no	3	no	1	2

v → non const; c → const; X{} → temp (r val)

→ referansa göre seçim sırası tablosu Josuttis kitabından

tam uyum yoksa, fallback her zaman universal ref → ★

★ Universal Constructor:

```cpp
class Myclass {
public:
    Myclass(const Myclass&);
    Myclass(Myclass&&);

    template <typename T>
    Myclass(T&&);
};

int main()
{
    Myclass m(12);
    Myclass m2(3.4);
}
```

Universal Constructor:

- Her türlü argüman geçebiliriz.
- Ancak, geçilen argüman, üst tablodaki seçim kriterine göre, copy veya move ctor'un önüne geçebilir!

```cpp
class Nec {
public:
    Nec() = default;

    Nec(const Nec&)
    {
        std::cout << "copy constructor\n";
    }

    Nec(Nec&&)
    {
        std::cout << "move constructor\n";
    }

    template<typename T>
    Nec(T&&)
    {
        std::cout << "universal constructor\n";
    }
};

int main()
{
    Nec nec;
    const Nec xnec;
    Nec a{ xnec }; // copy ctor
    Nec b{ nec };  // universal constructor
    Nec c{ std::move(nec) }; // move ctor
    Nec d{ std::move(xnec) }; // universal constructor
}
```

1. seçenek Nec& olm
2. seçenek universal ctor
3. seçenek const Nec&