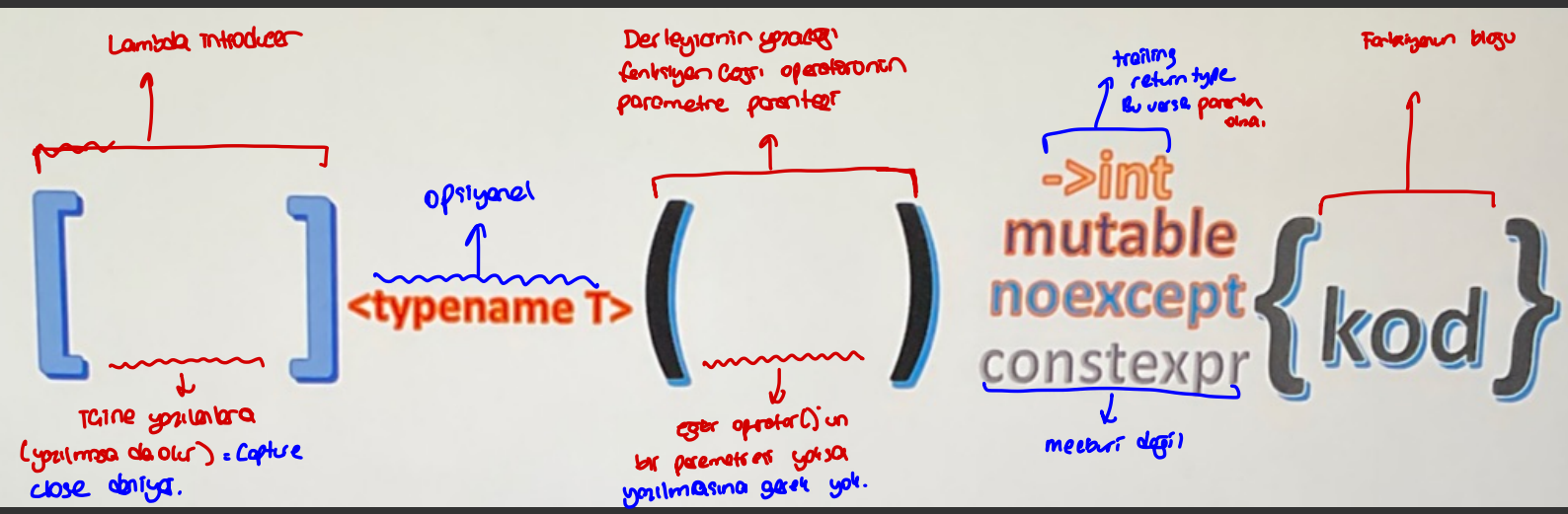
* Lambda Expressions: → geçen dersler devam (Sonraya ertelendi)

- C++ 11 ile eklendi.
- Derleyiciye sınıf tanımı yaptıran bir ifade → isimlendirmesiz kendi yapar
  - → tanımladığı sınıfa member olarak "operator()" tanımlar
  - → Yaddaki lambda ifadesi oluşturduğu sınıf tipinden bir, geçici nesneye dönüştürür. (pr value)
- Derleyicinin oluşturduğu bu sınıfa → closure type denir.
  - ↳ closure type'tan da closure object denir.




```
int main()
{
    using namespace std;

    []() {cout << "ali"; } ();
}
```

operator()

⇒ fonksiyon çağrısı bu

lambda ifadesi.

↳ Compiler bu lambda ifadesinden bir class yardı, sonra onu geçici nesne yapar.

```
class xyzm13t_ {
public:
    int operator()(int a)
    {
        return a * a;
    }
};

int main()
{
    using namespace std;

    auto f = [](int a) {return a * a; };

    std::cout << f(12) << "\n";
}
```

→ normalde böyle bir class bu

parametresi

return type'a göre deduction yapılabilir.

geçici nesne

function call → bu default olarak const member function.

const olmasını istemiyorsak, mutable keyword. (mutable variable dan farklı)

↳ non-static veri elemanını değiştirebilir artık.

→ köşeli parantezin içi boşsa, oluşan sınıfın data member'ı yok → stateless lambda denir.

```cpp
int main()
{
    using namespace std;

    auto x = [](int a) {return a * a; };
    auto y = [](int a) {return a * a; };
}
```

→ Bu ikisi aynı türden değil !!!

x ve y'nin türü farklı. Return değerimiz ise class türünden

```cpp
int main()
{
    auto f = [](int x)->double {
        return x * x;
    };
}
```

trailing return type ile '->double'

Derleyicinin çıkarım yapmasına engel olup, biz belirlemiş olduk.

```cpp
int main()
{
    auto f = [](int x)->double {
        return 10;

        return 4.5;
    };
}
```

eğer trailing return type olmasaydı

→ Çıkarım yapılamadı X

```cpp
class abcxyz {
public:
    template <typename T>
    int operator()(T x)const
    {
        return x + x;
    }
};

int main()
{
    using namespace std;

    auto f = [](auto x){
        return x + x;
    };

    cout << f(12) << "\n";
}
```

→ generalized lambda expression, bu şekilde bir member template oluşturur.

hatalı mı? → auto olmayaydı?

auto ile fonksiyon parametre bildirimi, Derleyici, fonksiyon çağrı operatör fonksiyonunu member template olarak yazar. → generalized lambda expression.

```cpp
int x = 5;
int y = 12;

int main()
{
    using namespace std;

    auto f = [](int a) {
        return a * (x + y);
    };
}
```

→ global expression

Fakat bunlar otomatik değişkit / fonksiyon içinde tanımlanmış olmaydı. lambda expressionda kullanımı derleyici hatası

global expressionlar lambda expression içinde kullanılabilir.

```cpp
int main()
{
    using namespace std;

    static int x = 5;
    static int y = 12;

    auto f = [](int a) {
        return a * (x + y);
    };
}
```

Burada hata olmamasının nedeni statik değişkenler olmaları.

```cpp
class abc_ytr_ {
public:
    abc_ytr_(int a, int b) : x(a), y(b) {}
    auto operator()(int a)const
    {
        return a * (x + y);
    }
private:
    int x, y;
};
```

non-static data members'ları init eden constructor

yapmasını istediğimiz fonksiyon.

non-static data members

bu şekilde (capture olmak); x veya y'yi bilmiyor.

```cpp
auto f = [x, y](int a) {
    return a * (x + y);
};
```

```cpp
int x = 5;

auto f = [x](int a) {
    ++x;
    return 1;
};
```

data member.

* Const üye fonksiyon, data member değiştiremez. bu yüzden syntax hatası.

```cpp
auto f = [x](int a)mutable {
    ++x;
    return 1;
};
```

artık "fonksiyon" const değil.

* Fakat Çok Önemli:

```cpp
int main()
{
    using namespace std;

    int x = 5;

    auto f = [x](int a)mutable {
        x *= a;
    };

    std::cout << "x = " << x << "\n";
    f(10);

    std::cout << "x = " << x << "\n";
}
```

copy capture

Fakat ne önceside ne sonrasında x'in değeri değişmedi.

```cpp
using namespace std;

int x = 5;

auto f = [&x](int a) {
    x *= a;
};

std::cout << "x = " << x << "\n";
f(10);

std::cout << "x = " << x << "\n";
```

reference capture

mutable olmak zorunda değil. (tekrar rule)

artık yerel değişken x değişti.

* Capture All by Copy

```cpp
int main()
{
    int a{}, b{}, c{}, d{};

    auto f = [=]() {
    };
}
```

"=" kopyasını tüm yerel değişkenleri kopyaladık.

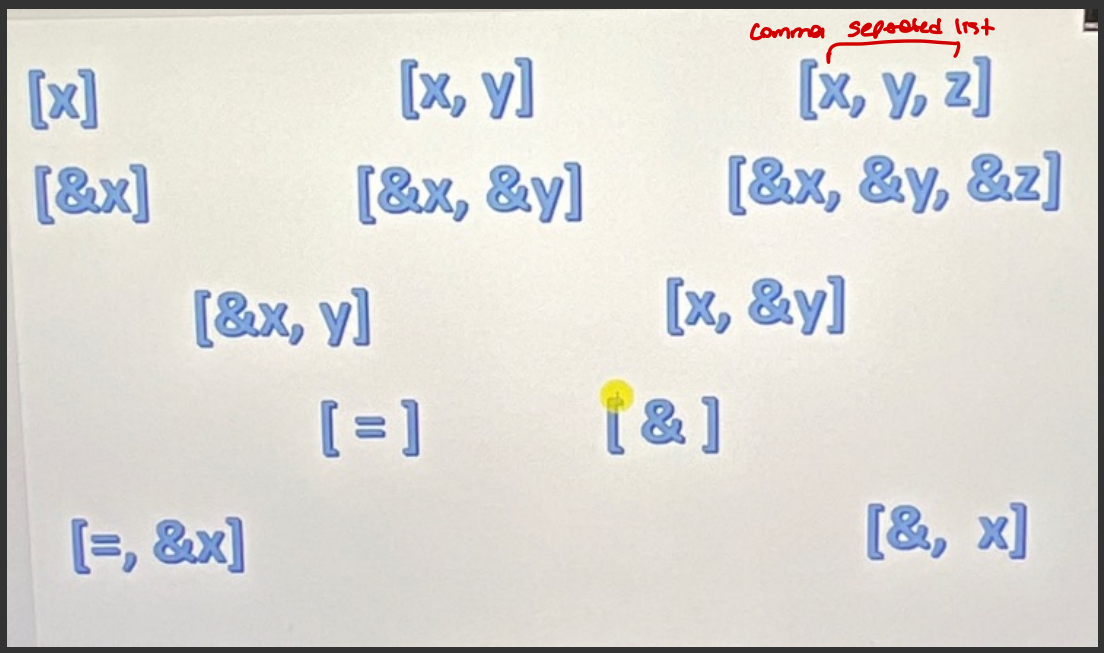
* Capture All by Reference:

```cpp
int main()
{
    int a{}, b{}, c{}, d{};

    auto f = [&](int x) {
        a *= x;
        b *= x;
        c *= x;
        d *= x;
    };
}
```

bunun

syntax hatası yok ✓



* Mülakatlarda Sorulur Dedi: Dangling Reference

```
auto func(int x)
{
    auto f = [&x]() {++x; };

    //..
    return f;
}


int main()
{
    auto f = func(10);

    f();
```

x burada yerel bir değişken.

lambda expression yine lambda expression döndürüyor.

Dangling reference oluşur.

→ Fakat reference ile capture edildiği için, dönüş expression ile sınırlı..

* Count / Count_if Algoritması

→ count, bir range'de verdiğimiz değere eşit olan öğelerin sayısı

→ count_if, range'deki öğeleri bir predicate'a gönderir, true değeri alır.

```
int main()
{
    using namespace std;

    vector<int> ivec;
    rfill(ivec, 1'000'000, Irand{0, 10'000});

    int low, high;

    cout << "sayilacak deger araligini giriniz: ";
    cin >> low >> high;

    cout << count_if(begin(ivec), end(ivec), [low, high](int i) {
        return i >= low && i <= high;
        }) << "\n";


}
```

Transform Algoritması:

```
int main()
{
    using namespace std;

    vector<string> svec;
    rfill(svec, 100, rname);
    print(svec);

    vector<string> destvec(100);

    string suffix;
    std::cout << "isimlere ne eklensin: ";
    cin >> suffix;
    transform(svec.begin(), svec.end(), destvec.begin(), [suffix](const string& s) {return s + suffix; });
    print( destvec)
}
```

→ input iteratorü aldık, bu callable'a verdik., return değeri destvec'e veriyoruz.

* Yani, bizim bir algoritmamız var, ve bu algoritma bizden bir callable istiyor.
  - → bir fonksiyon adresi gönderebiliriz
  - → functor class / function object gönderebiliriz. → Bunun için lambda expression kullanılır. → Daha kolay okunur / lokalize

```
int main()
{
    using namespace std;

    auto fsquare = [](int x) {return x * x; };
    int (*fptr)(int) = fsquare;

    cout << fptr(46) << "\n";
}
```

→ nothing bir capture yoksa stateless lambda

→ stateless lambda ifadeler implicit olarak function pointera dönüşür

*Immediately Invoked Function Expression:

→ Const nesneyi init etmek için kullandık.

```
const Myclass mx = [&]() {
    //..
}();
```

→ IIFE idiomu

*Cpp 20'ye kadar, lambdalar, stateless olsa dahi, default constructorları delete edilmiştir.

```
auto fsquare = [](int x) {return x * x; };

decltype(fsquare) fx;
```

Cpp 20'de legal