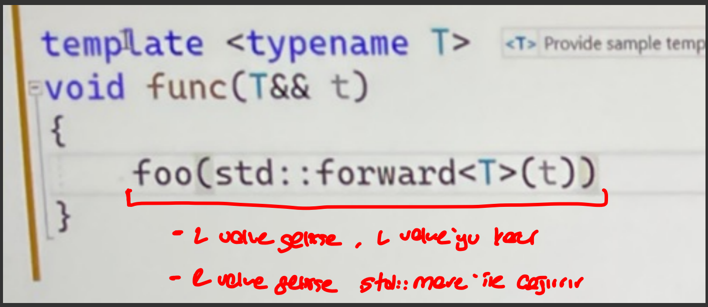
* Forwarding Referanstan devam:



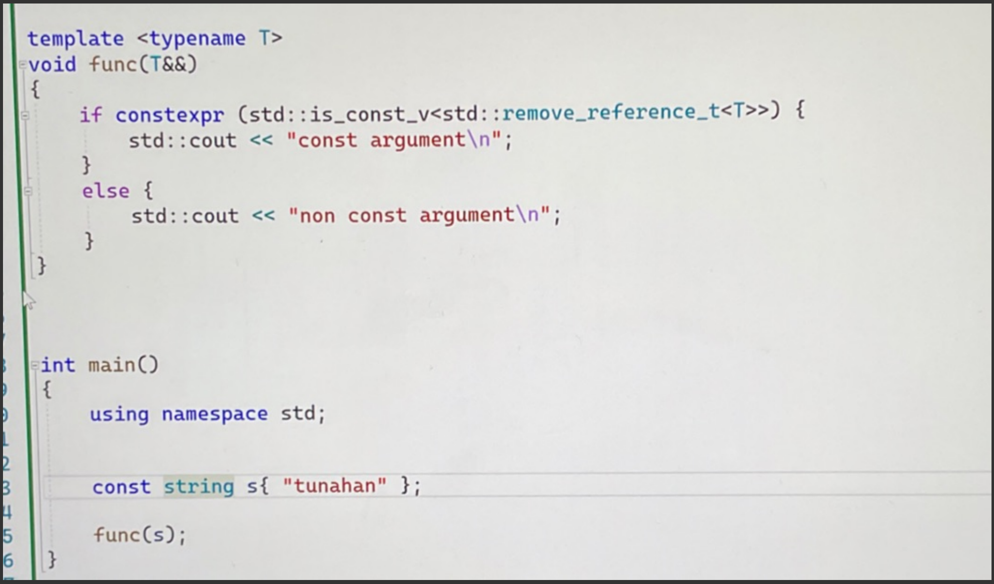
```cpp
template <typename T>
void func(T&& t)
{
    foo(std::forward<T>(t))
}
```

- L value gelirse , L value'yu korur
- R value gelirse std::move ile çağırır

→ Fakat Universal / Forwarding referanse, sadece perfect forwarding için kullanılmıyor!!

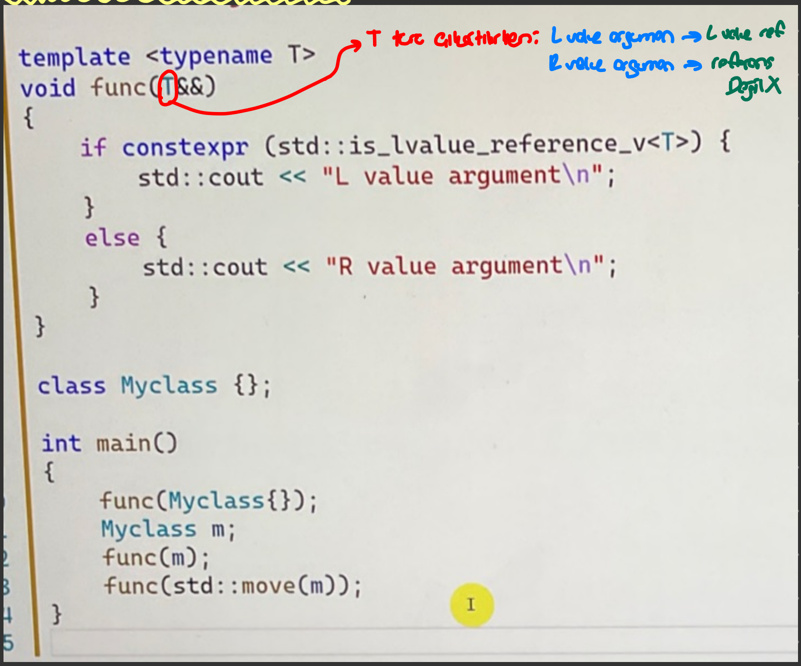
→ CONST / NON-CONST Parameter Deduction:

- universal referanse, const / non const parametre çıkarımı için kullanılabilir.



```cpp
template <typename T>
void func(T&&)
{
    if constexpr (std::is_const_v<std::remove_reference_t<T>>) {
        std::cout << "const argument\n";
    }
    else {
        std::cout << "non const argument\n";
    }
}


int main()
{
    using namespace std;

    const string s{ "tunahan" };

    func(s);
}
```

→ Value Category Dependent Code:

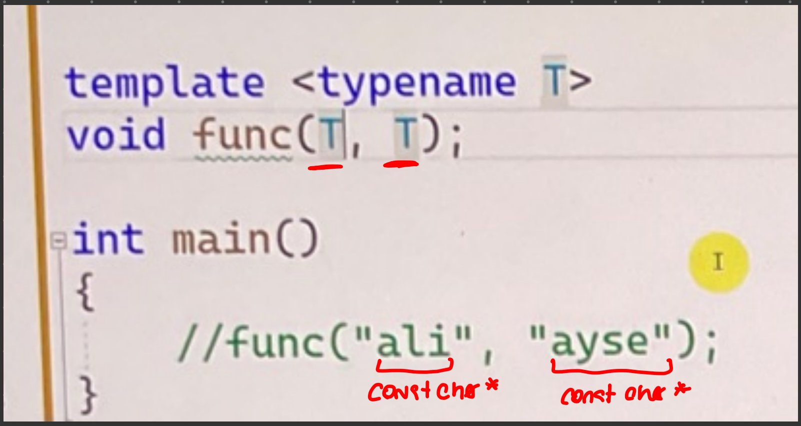


T tür çıkarılırken: L value argüman → L value ref
R value argüman → referans değil X

```cpp
template <typename T>
void func(T&&)
{
    if constexpr (std::is_lvalue_reference_v<T>) {
        std::cout << "L value argument\n";
    }
    else {
        std::cout << "R value argument\n";
    }
}

class Myclass {};

int main()
{
    func(Myclass{});
    Myclass m;
    func(m);
    func(std::move(m));
}
```

- T ya L value reference, ya da non-reference

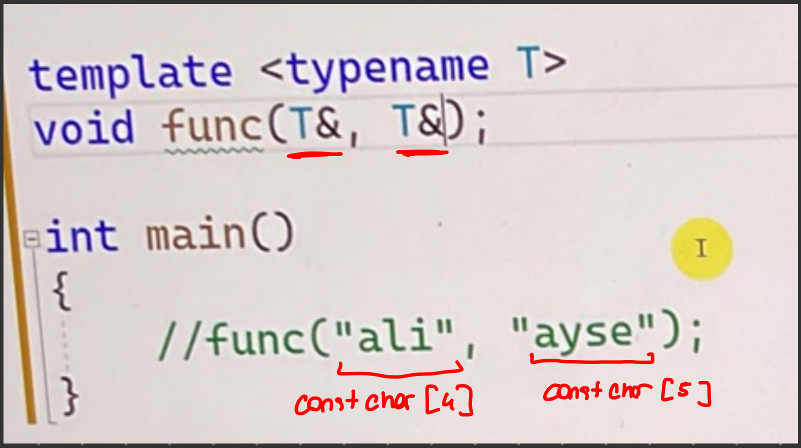
* Hatırlatma / Yorı:



```cpp
template <typename T>
void func(T, T);

int main()
{
    //func("ali", "ayse");
}
```

→ Syntax Hatası Yok ✓

- T için Çıkartılen tür: const char *

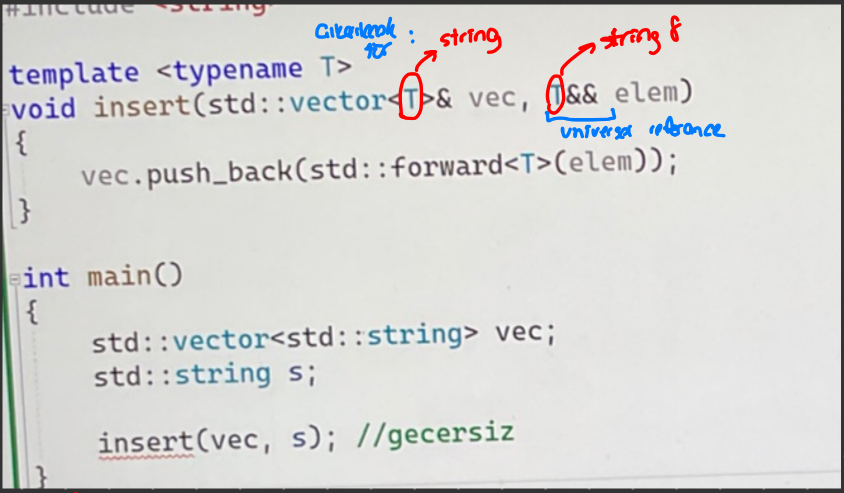


```cpp
template <typename T>
void func(T&, T&);

int main()
{
    //func("ali", "ayse");
}
```

→ Syntax Hatası ✗

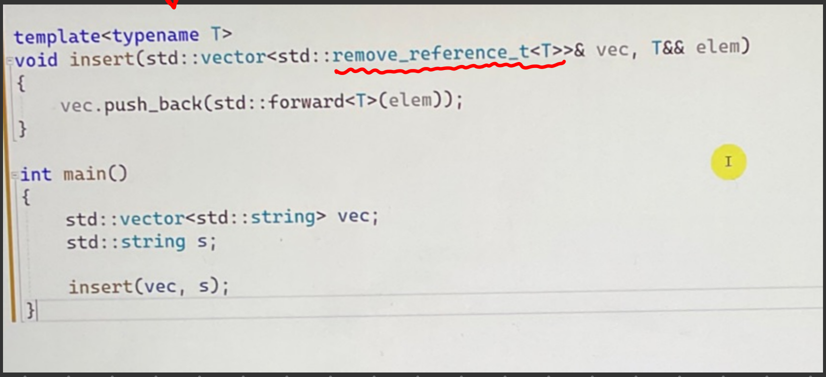
→ Çünkü, array decay olmayacak!

↳ Universal Referance' da Benzeri bir Conflict'e neden olabilir!



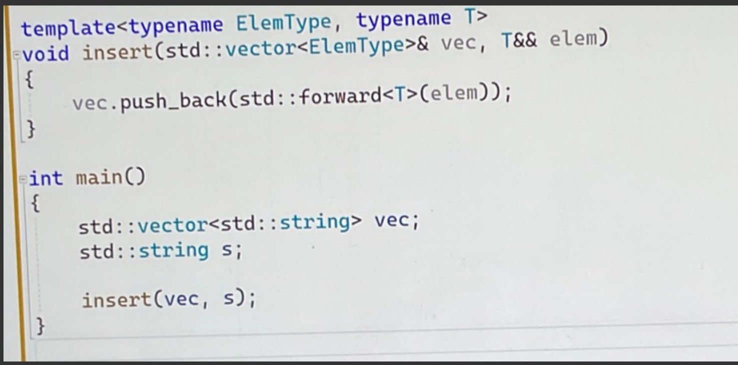
```cpp
#include <string>

template <typename T>
void insert(std::vector<T>& vec, T&& elem)
{
    vec.push_back(std::forward<T>(elem));
}

int main()
{
    std::vector<std::string> vec;
    std::string s;

    insert(vec, s); //gecersiz
}
```

Çözüm alternatifi.



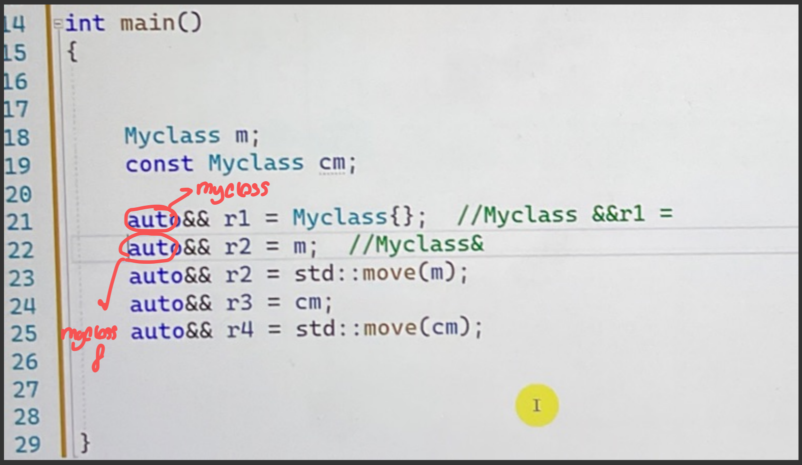
```cpp
template<typename T>
void insert(std::vector<std::remove_reference_t<T>>& vec, T&& elem)
{
    vec.push_back(std::forward<T>(elem));
}

int main()
{
    std::vector<std::string> vec;
    std::string s;

    insert(vec, s);
}
```

Çözüm Alternatifi



```cpp
template<typename ElemType, typename T>
void insert(std::vector<ElemType>& vec, T&& elem)
{
    vec.push_back(std::forward<T>(elem));
}

int main()
{
    std::vector<std::string> vec;
    std::string s;

    insert(vec, s);
}
```

* auto &&:



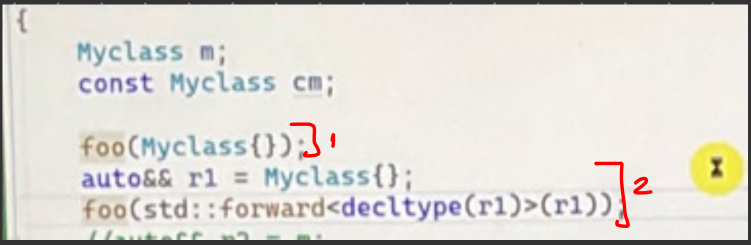
```cpp
int main()
{


    Myclass m;
    const Myclass cm;

    auto&& r1 = Myclass{};  //Myclass &&r1 =
    auto&& r2 = m;  //Myclass&
    auto&& r2 = std::move(m);
    auto&& r3 = cm;
    auto&& r4 = std::move(cm);



}
```

→ Template kurallarıyla aynı!

→ L value için, L value ref çıkarımı

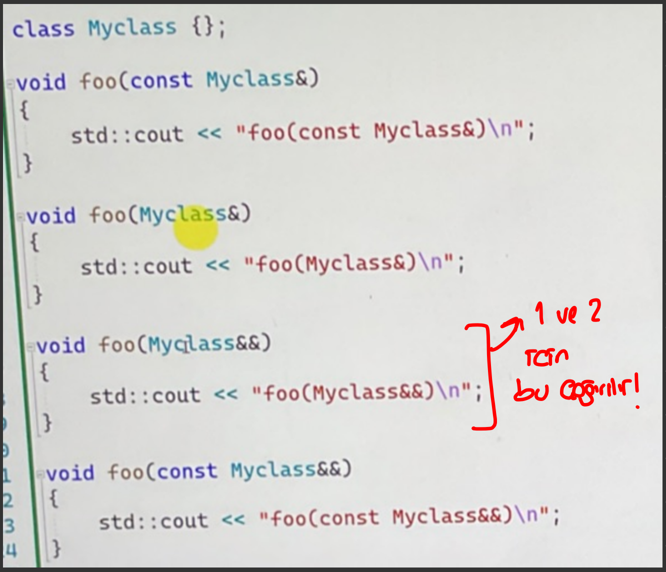
R value için, non-reference çıkarımı

* auto && ve Perfect Forwarding:



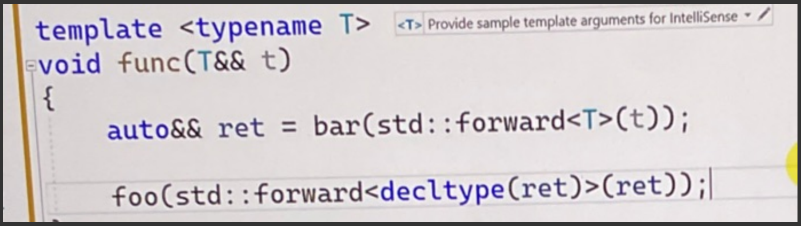
```cpp
{
    Myclass m;
    const Myclass cm;

    foo(Myclass{});
    auto&& r1 = Myclass{};
    foo(std::forward<decltype(r1)>(r1));
```

1 ve 2 aynı anlama geliyor



```cpp
class Myclass {};

void foo(const Myclass&)
{
    std::cout << "foo(const Myclass&)\n";
}

void foo(Myclass&)
{
    std::cout << "foo(Myclass&)\n";
}

void foo(Myclass&&)
{
    std::cout << "foo(Myclass&&)\n";
}

void foo(const Myclass&&)
{
    std::cout << "foo(const Myclass&&)\n";
}
```

* auto &&'in kullanım alanları: - generic bir functionda, onun return değerini bir parametrede tutup, o parametreyi perfect passing yapabiliriz!

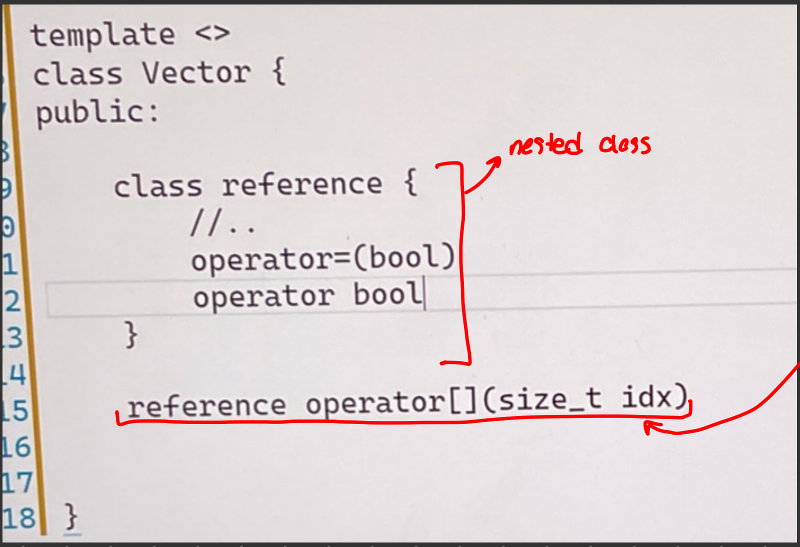


```cpp
template <typename T>
void func(T&& t)
{
    auto&& ret = bar(std::forward<T>(t));

    foo(std::forward<decltype(ret)>(ret));
```

*** Vector < bool > :** Bu bir container değil!

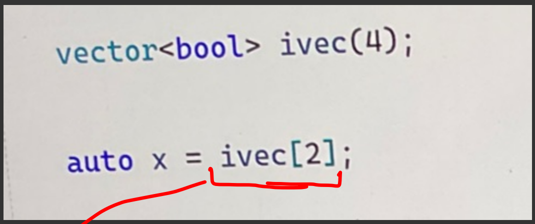
Bunda boolean türden nesne tutulmuyor! wtF!

→ Vector boolda aslında bit tutuluyor Ama bit'e referans döndürülemez!



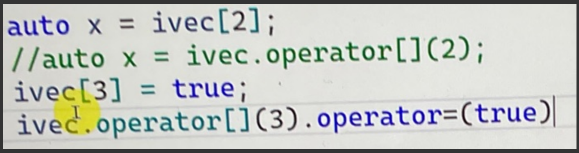
```cpp
template <>
class Vector {
public:

    class reference {
        //..
        operator=(bool)
        operator bool
    }

    reference operator[](size_t idx)
}
```

nested class



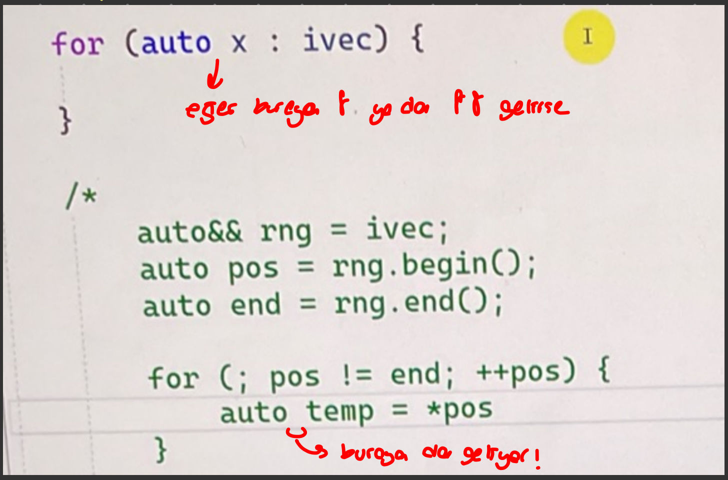
```cpp
vector<bool> ivec(4);

auto x = ivec[2];
```

Burada aslında reference adındaki bir proxy class üzerinde işlem yapıyoruz!



```cpp
auto x = ivec[2];
//auto x = ivec.operator[](2);
ivec[3] = true;
ivec.operator[](3).operator=(true)
```

*** Range Based For loop karşılığında yazılan kod:**



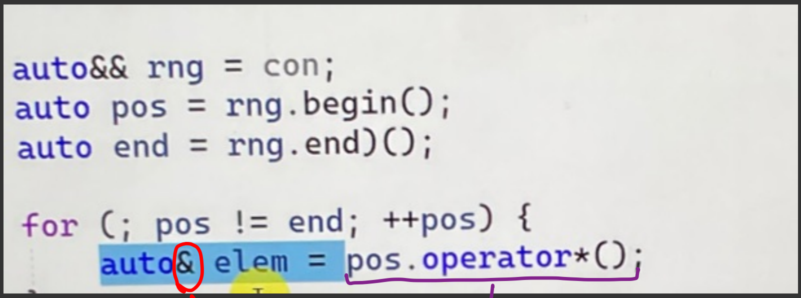
```cpp
for (auto x : ivec) {

}

/*
    auto&& rng = ivec;
    auto pos = rng.begin();
    auto end = rng.end();

    for (; pos != end; ++pos) {
        auto temp = *pos
    }
```

eğer buraya & ya da && gelirse

buraya da geliyor!

✱✱ Normalde range based for loopta atama yapılacaksa, for (auto & x : ivec) → şeklinde yazılır.

→ Ancak, vector < bool > gibi, proxy sınıf döndürülen durumlarda, bu şekilde auto & ile for loop yazarsak



```cpp
auto&& rng = con;
auto pos = rng.begin();
auto end = rng.end)();

for (; pos != end; ++pos) {
    auto& elem = pos.operator*();
```

Compiler bu şekilde kod üretir.

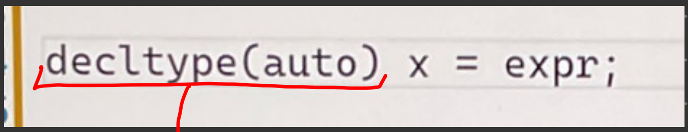
⇒ Const olmayan L value referansa, R value ref bağlıyoruz! ⇒ Syntax Hatası!

ref eklendi!

proxy sınıf türünden R value!

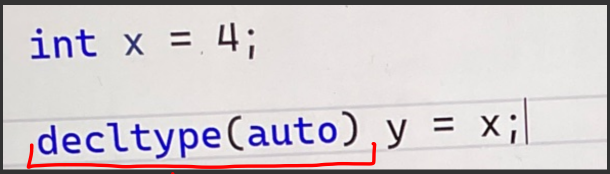
* Bu yüzden auto && kullanmak en mantıklısı!!!

*** Decltype (auto):**

- 

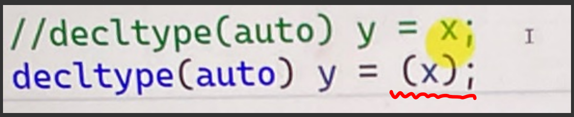
```cpp
decltype(auto) x = expr;
```

x'in türü, decltype kurallarına göre belirlenir !! / yani decltype (expr) ne ise, decltype (auto) için alınacak tür o

- 

```cpp
int x = 4;

decltype(auto) y = x;
```

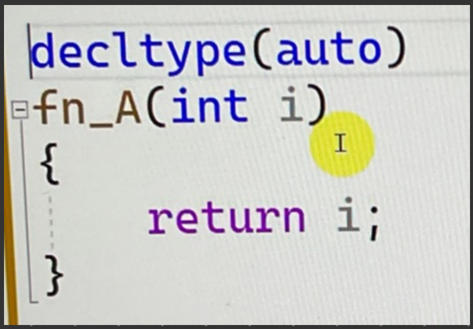
int ⇒ decltype (x) deseydik int elde ederdik!
⇒ y'nin türü (decltype(auto)) int olacak!

- 

```cpp
//decltype(auto) y = x;
decltype(auto) y = (x);
```

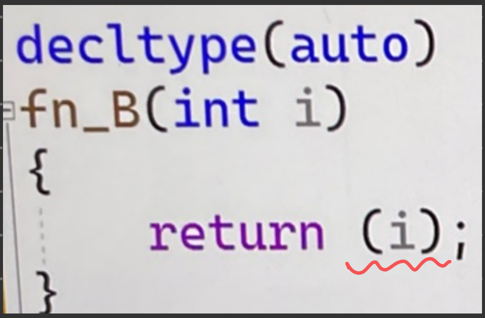
- decltype((x)) = int & olduğu için
- decltype (auto) = int&

(#)



```cpp
decltype(auto)
fn_A(int i)
{
    return i;
}
```

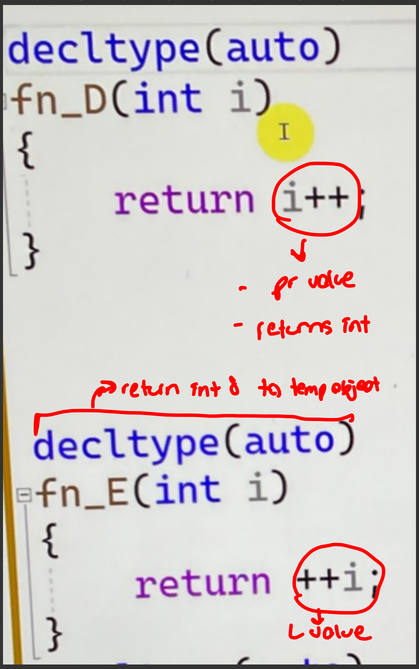
⇒ Bu kodda herhangi bir sorun yok!
⇒ return değer int ✓



```cpp
decltype(auto)
fn_B(int i)
{
    return (i);
}
```

⇒ Fakat bu kodda sorun var!
⇒ return değer int & ⇒ fakat otomatik ömürlü nesneye referans döndürüyor!

tanımsız davranış



```cpp
decltype(auto)
fn_D(int i)
{
    return i++;
}
```

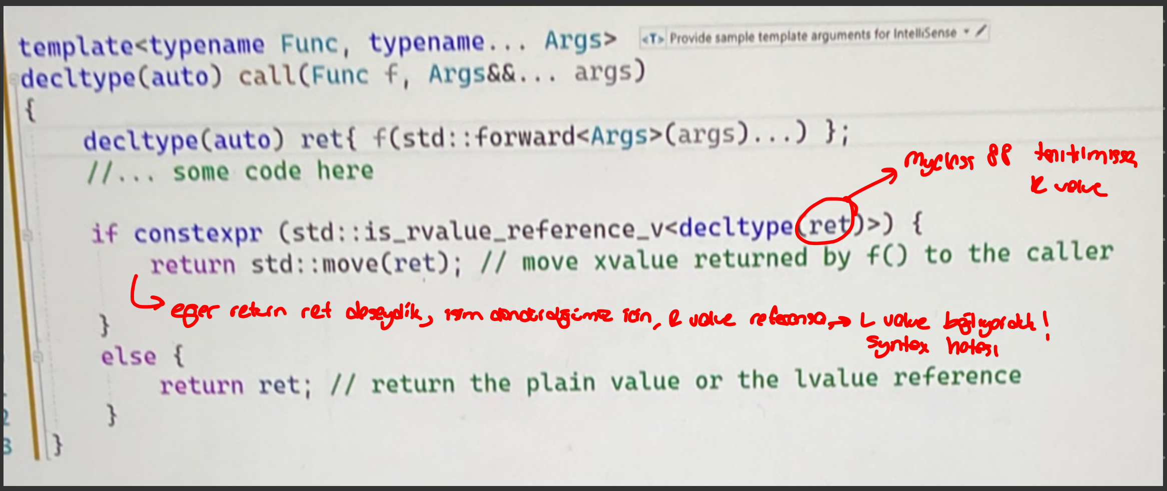
- pr value
- returns int

return int & to temp object

```cpp
decltype(auto)
fn_E(int i)
{
    return ++i;
}
```

L value

→ Decltype(auto) ile std::invoke:

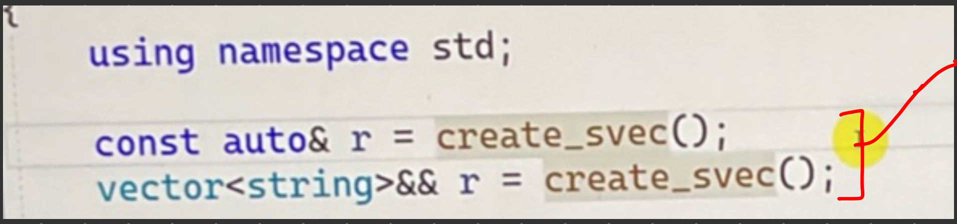


```
template<typename Func, typename... Args>   <T> Provide sample template arguments for IntelliSense
decltype(auto) call(Func f, Args&&... args)
{
    decltype(auto) ret{ f(std::forward<Args>(args)...) };
    //... some code here

    if constexpr (std::is_rvalue_reference_v<decltype(ret)>) {
        return std::move(ret); // move xvalue returned by f() to the caller
    }
    else {
        return ret; // return the plain value or the lvalue reference
    }
}
```

Myclass && tanımlmışsa R value

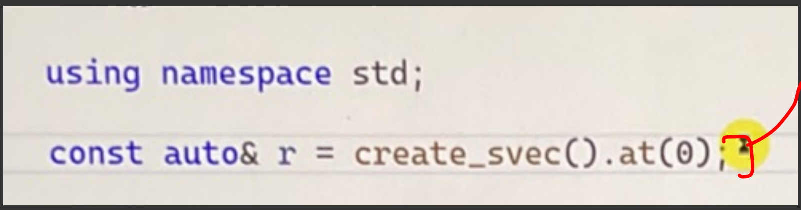
↳ eğer return ret olsaydık, isim döndürdüğümüz için, R value referansa → L value bağlıyordu! Syntax hatası

* Life time Extension:



```
{
    using namespace std;

    const auto& r = create_svec();
    vector<string>&& r = create_svec();
```

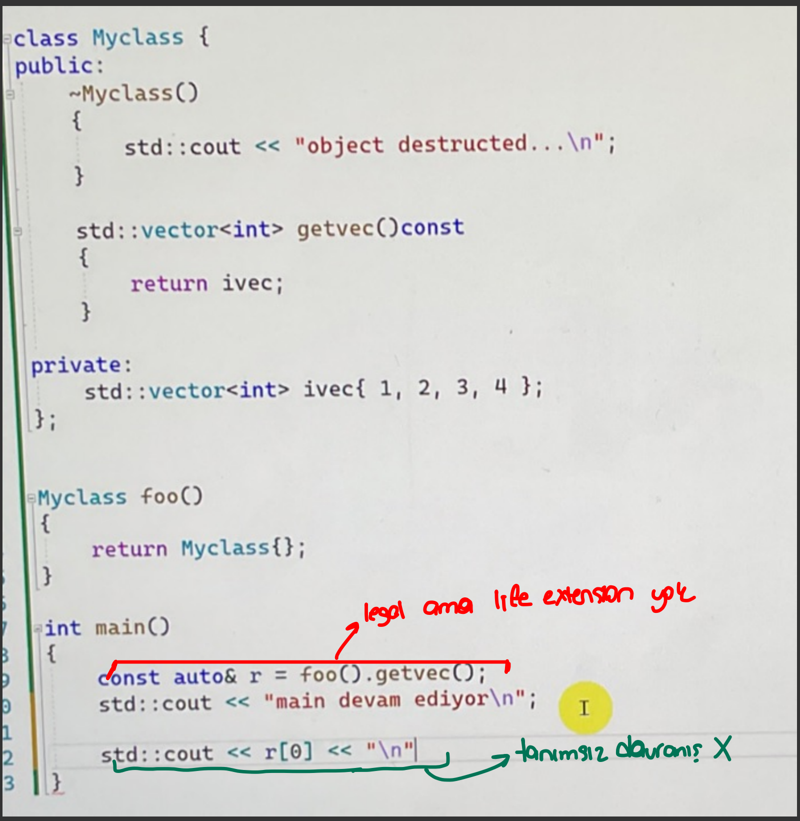
Burada life time extension var!



```
using namespace std;

const auto& r = create_svec().at(0);
```

Kod geçerli fakat, life extension yok ✗

⇒ Çünkü fonksiyonun geri dönüş değerine değil, fonksiyonun geri dönüş durduğu, geçici nesnenin üye fonksiyonun geri dönüş değerine referans oldu!!



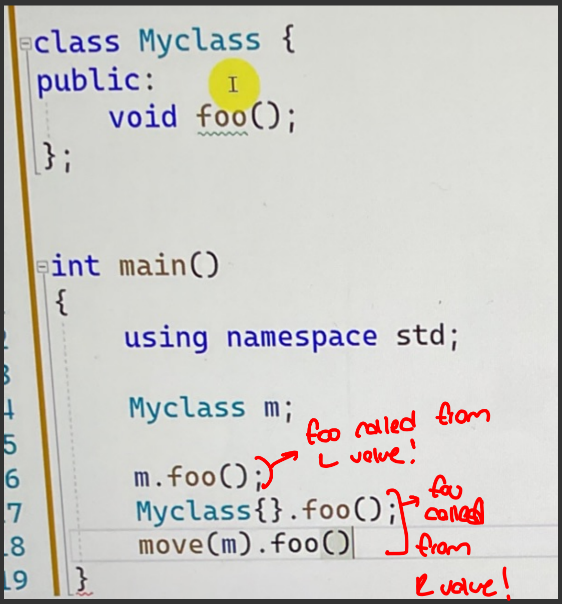
```
class Myclass {
public:
    ~Myclass()
    {
        std::cout << "object destructed...\n";
    }

    std::vector<int> getvec()const
    {
        return ivec;
    }

private:
    std::vector<int> ivec{ 1, 2, 3, 4 };
};


Myclass foo()
{
    return Myclass{};
}

int main()
{
    const auto& r = foo().getvec();
    std::cout << "main devam ediyor\n";

    std::cout << r[0] << "\n"
}
```

legal ama life extension yok

tanımsız davranış ✗

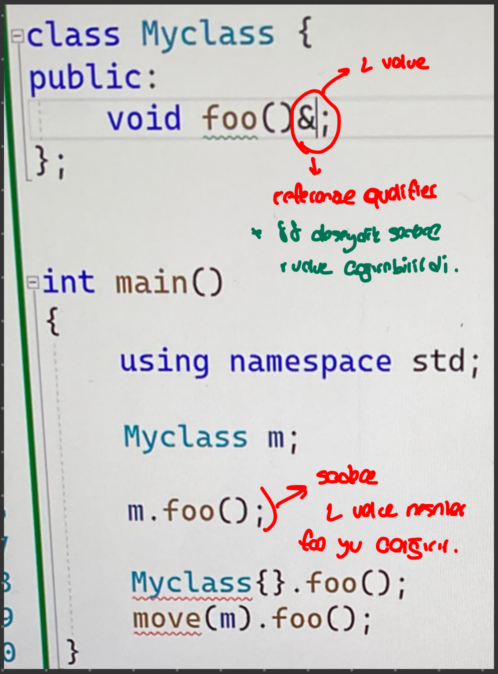
*** Reference Qualifier:** - C++'daki bir araç

- Bir üye fonksiyonun, hangi değer kategorisindeki nesnelerle çağırılabileceğini gösterir!
- Bir overload mekanizması oluşturur.



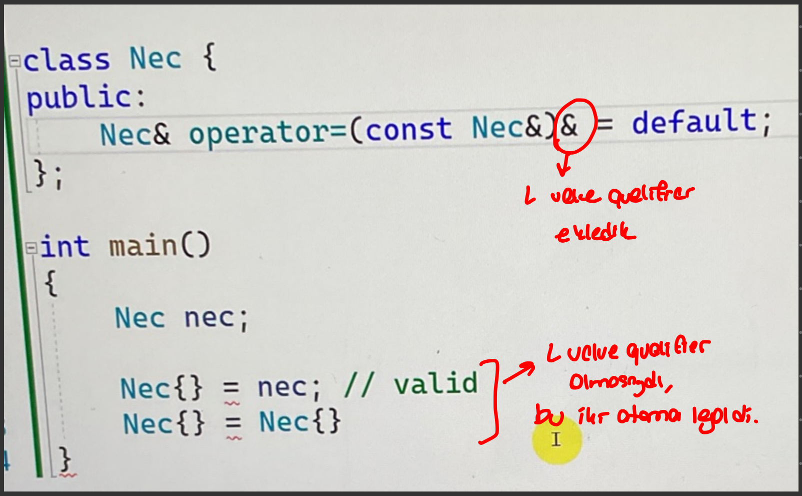
```cpp
class Myclass {
public:
    void foo();
};

int main()
{
    using namespace std;

    Myclass m;

    m.foo();
    Myclass{}.foo();
    move(m).foo()
}
```

→ Fakat, diyelim ki biz, R value nesnelerin foo'yu çağırmasının önüne geçmek istiyoruz



```cpp
class Myclass {
public:
    void foo()&;
};

int main()
{
    using namespace std;

    Myclass m;

    m.foo();
    Myclass{}.foo();
    move(m).foo();
}
```

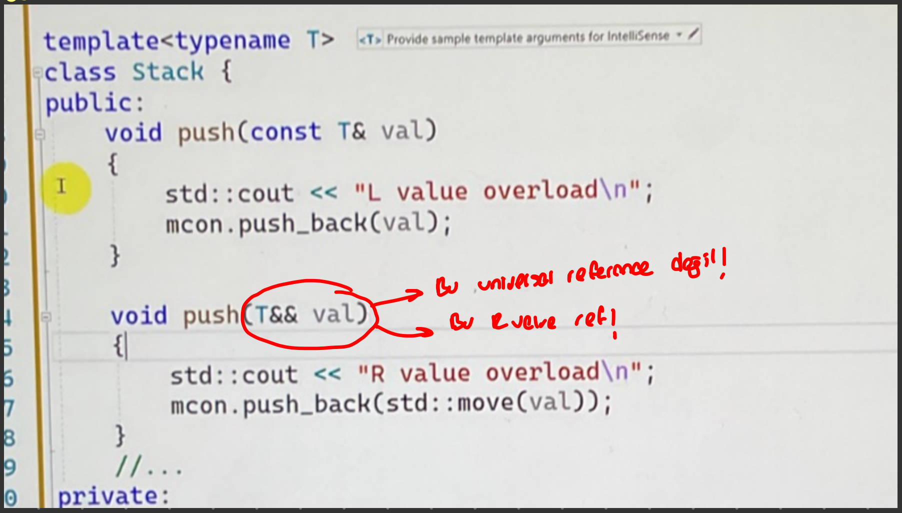
L value
reference qualifier
* && deseydik sadece r value çağırabilirdi.
sadece L value nesneler foo'yu çağırır.



```cpp
class Nec {
public:
    Nec& operator=(const Nec&)& = default;
};

int main()
{
    Nec nec;

    Nec{} = nec; // valid
    Nec{} = Nec{}
}
```

L value qualifier ekledik
L value qualifier olmasaydı, bu ikisi atama legaldi.

*** Mülakatlarda Sorulur:**



```cpp
template<typename T>
class Stack {
public:
    void push(const T& val)
    {
        std::cout << "L value overload\n";
        mcon.push_back(val);
    }

    void push(T&& val)
    {
        std::cout << "R value overload\n";
        mcon.push_back(std::move(val));
    }
    //...
private:
```

Provide sample template arguments for IntelliSense
Bu universal reference değil!
Bu R value ref!