

6 Nisan 2022

scope leakage neden kötü ?

- 1) hata yapma riskiniz artar
- 2) kodu okuyan yanılır
- 3) kaynak tutuyor olabilir,

```
if (int x=3){  
    // x sadece burada geçerlidir  
}
```

if with mitralizer (scope leakage'ı engeller)

```
if ( int x=foo(); x>10)  
{  
    //
```

geçici nesne:

```
int main()  
{  
    double dval = 3.4;  
  
    const int& r1= 10; // doğru !!!  
    // derleyici durumdan vazife çıkartarak  
    /* pseudo code  
    int gn = 10;  
    const int &r1 = gn;  
    */  
    // referans geçici nesneye bağlanmış durumda  
    // geçici nesnenin hayatı ref.'in scope sonuna kadar hayatı kalır
```

void func(T &tr); // lvalue ref. alan fonksiyonu sadece lvalue expres. ile çağrılabılır,
rvalue ile çağrılamaz

void func(const T &tr); // const olduğundan lvalue ref. alan fonksiyonu hem lvalue hem
rvalue expr. ile çağrılabilsin

R value reference: (modern C++ ile dile eklendi)

neden böyle bir araç eklendi ?

dilde modern C++ öncesi problem yaratınca

- 1) move semantics (taşıma semantiği) yoktu
- derleyicinin daha efficient bir kod üretme olanağı
- copy yapılan yerde move yapılınca iyi bir verim elde edildi.

Hayati bitecek bir nesnenin kaynaklarını başka bir nesneye aktarmaya move semantics deniyor

- 2) generic programlamada sorun vardı.
perfect forwarding (mükemmel gönderim)

```
int main()  
{  
    int x;  
  
    int &&r = x; // okunuşu: int ref ref  
    // sağ taraf referansı sol taraf değerine  
    // bağlanamaz  
}
```

lvalue = lvalue or rvalue
const lvalue = lvalue or rvalue

lvalue_ref (&) = lvalue // lvalue bind to lvalue ref
rvalue_ref (&&) = rvalue // rvalue bind to rvalue ref
const_lvalue_ref = rvalue or lvalue // istisna !!

```
int foo();  
int& bar(); // referans döndürür adres değil !!
```

```
int main()  
{  
    int &r1 = foo(); // geçersiz, rvalue expr. to lvalue ref  
    int &&r2 = foo(); // ok, rvalue ref'e rvalue ile ilk değer  
  
    int &r3 = bar(); // ok, lvalue expr. to lvalue ref  
    int &&r4 = bar(); // sentax hatası  
}
```

&d adres ise rvalue, referans ise lvalue ref. olur.

int& bar(); \Rightarrow referans döndürür, adres değil

int* foo(); \Rightarrow adres döndürür.

type deduction (tür akarımı): a) auto
b) decltype
c) decltype (auto)

auto x = expr \Rightarrow x'in türünü anlatır, auto'ya karşılık gelen türler
olar.

auto x(expr)

auto x{expr}

auto x = expr

auto &x = expr

auto &&x = expr

} hepsinde tür akarımı x'i
kuralları farklıdır.

* Neden auto var? 1) auto ile bir değişken tanımlanırken
ilk değer vermek zorlu
2) yazım kolaylığı

AAA : almost always auto

1) auto ile yapılan tür çıkarımında const'luk düşer (normal değişkenlerde)
referanslık (pointer'da) ve top level const'luk düşer.

const int* const ptr
top level
low level

auto ile yapılan tür çıkarımında const'luk düşer (normal değişkenlerde)
auto ile yapılan tür çıkarımında top level const'luk ve referanslık düşer (pointer'da,
referansta)
dizilerde ise const'luk düşmez çünkü bu top level const. değildir

```
int main()
{
    int a[3] = {1, 2, 3};

    auto p = a;
    // array decay, p'nin türü int*, auto = int*
}
```

```
int main()
{
    const int a[3] = {1, 2, 3};

    auto p = a;
    // const'luk düşmez bu top level const. değil, array decay'de elde edilen tür
    const int*,
    auto = const int*
}
```

```
int main()
{
    auto p = "arda"; // p'nin türü const char*
}
```

Function pointer

```
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    void (*fun_ptr)(int) = fun; // & removed

    fun_ptr(10); // * removed

    return 0;
}
```

```
int func(int);
// fonksiyonun türü int(int)
// fonksiyonun adresinin türü int(*)(int)

int main()
{
    auto p = func; // function to pointer
    conversion
    // auto : int (*) (int)

    int (*p)(int) = func; // auto ile aynı
}
```

2) auto &x=expr 'nın kuralları
const'luk düşmez

```
int main()
{
    const int x{10};

    auto& y = x; // const'luk düşmez, auto'ya karşı gelen const int türü,
}
```

```
int main()
{
    int a[3] = {1, 2, 3};

    auto& x = a; // auto'ya karşı gelen tür int[3]
    // auto olmadan bildirim
    int(&x)[3] = a;
}
```

⇒ diziyi referans
1. döşte bahsedilmemişti.

```

int main()
{
    auto& x = "eray";
    // auto = const char[5];
    // auto olmasa :
    const char(&x)[5] = "eray";
}

```

?

std::string is a container class
 char* is just a pointer to a character sequence
 char array?

* In C string literals are arrays of char but
 in C++ they are constant array of char.

* there are some cases where you might prefer char* over
 std::string

int (*ptr)(int)

int foo(int);

```

int main()
{
    auto x = foo; // auto : decay, function to pointer conversion,
    // function pointer

    auto& x = foo; // auto: int(int)
    // auto olmadan:
    int (&x)(int) = foo;
}

```

\Rightarrow auto : int(*)(int)
 ? or
 int (\$)(int)

```

int main()
{
    int x = 10;
    int* ptr = &x;

    auto &p = ptr;
    // auto olmadan
    int*& p = ptr;           => pointer'a referans
                                1. derste islemisti
}

```

3) auto && = expr 'in kuralları, ikinci görülecektir!

```

int main()
{
    int ival = 10;

    auto&& x = ival;
    // bu forwarding reference, bu rvalue
    değil
}

```

reference collapsing rules (reference bozulma kuralları): ????????

<https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers>

T&	& :	T& (lvalue ref to lvalue ref: lvalue ref.)
T&	&& :	T& (lvalue ref. to rvalue ref: lvalue ref.)
T&&	& :	T& (rvalue ref. to lvalue ref : lvalue ref.)
T&&	&& :	T&& (rvalue ref. to rvalue ref: rvalue ref.)

```

int main()
{
    int ival = 10;

    auto&& x = ival;
    // int& x = ival; // üstteki ile karşılaştırınca, auto'ya karşılık gelen tür lvalue ref.
    // bu durumda rvalue ref.'e lvalue ref. oluştu. Reference collapsing'den lvalue ref. oluşur.
}

```

```

int main()
{
    auto&& x = expr; // iki ihtimal var, x ya lvalue ref veya rvalue ref olacak. Başka ihtimal yok.
    ilk değer veren ifadenin türüne bağlı
    expr. lvalue ise referans lvalue ref. olur
    expr. rvalue exp. ise referansı rvalue ref. olur.
}

```

C'de bir türe eş-isim vermek için typedef anahtar sözcüğü kullanılır.
C++'da using anahtar sözcüğü ile de yapılır.
typedef de geçerli !!

```

typedef int WORD;
using WORD = int;

```

```

typedef int *iptr;
using iptr = int*;

```

```

typedef int inta[5];
using inta = int[5];

```

```

typedef int (*fptr)(int);
using fptr = int (*)(int);

```

reference collapsing rules geçerli

```

int main()
{
    using lref = int&; // lref : int&'in türeş ismi

    int ival = 5;
    lref& r = ival; // r 'nin türü int&, lref'de int&, dolayısıyla lvalue ref'e lvalue ref.,

    int ival{2};
    lref &&r = ival; // sağ taraf ref. sol taraf ref. oluştu,
    r halen rvalue ref.
}

```

Decltype

decltype: // declaration type'dan uydurma

compile time'da bir tür elde edilir.
Tür kullanılan her yerde kullanılabilir.

```

int main()
{
    int x = 10;

    decltype(x); // int

    decltype(x) ival = 5;
}

```

```

struct Data{
    int a, b;
};

int main()
{
    Data mydata;
    decltype(mydata.a); // int
}

int main()
{
    int a[5] = {};
    decltype(a) b; // int[5]
}

```

expression'da tür çıkarımı nasıl olur ?

```

int main()
{
    int ival = 45;
    decltype((ival)); // iki parantez olunca isim kuralı
    biter ve expr. olur.
}

```

decltype operandı olan ifadenin value category'sı

- a) PR value expr. => T
- b) Lvalue expr. => T&
- c) X value expr. => T&&

```

int main()
{
    const int x = 20;
    decltype(x); // const int
}

```

```

int main()
{
    int x = 10;
    int& r = x;
    decltype(r) a; // hata çünkü
    referanslara ilk değer verilmeli
}

```

```

int main()
{
    int ival = 45;
    decltype(ival + 5); // int

    decltype((ival)); // int&
    decltype((ival)) x; // hata olur çünkü ref. ilk değer vermek gereklidir

    int *ptr = &ival;
    int a = 5;
}

```

Unintended side effects

- 1) scope linkage , if with initializer scope linkage'i engeller
- 2) r value reference'in direkt eklenme gereklisi
 - a) move semantics
 - b) perfect forwarding
- 3)

lvalue = lvalue or rvalue
const lvalue = lvalue or rvalue

lvalue_ref (&) = lvalue // lvalue bind to lvalue ref
rvalue_ref (&&) = rvalue // rvalue bind to rvalue ref
const_lvalue_ref = rvalue or lvalue // istisna !!

4) type deduction (tür çıkarımı)

- a) auto
- b) decltype
- c) decltype(auto)

- 5) auto x = expr
auto &x = expr
auto &&x = expr
- } kurallar farklı
} ikide püntülecek

acronym

AAA : almost
always
auto

- 6) auto ile yapılan tür çıkarımında const'luk düşer (normal değişkenlerde)
auto ile yapılan tür çıkarımında top level const'luk ve referanslık düşer (pointer'da,
referansta)
dizilerde ise const'luk düşmez çünkü bu top level const. değildir

7) auto &x = expr

Const'luk düşmez.

8) function pointer

```
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    void (*fun_ptr)(int) = fun; // & removed

    fun_ptr(10); // * removed

    return 0;
}
```

9) reference Collapsing rules

T&	& :	T& (lvalue ref to lvalue ref: lvalue ref.)
T&	&& :	T& (lvalue ref. to rvalue ref: lvalue ref.)
T&&	& :	T& (rvalue ref. to lvalue ref : lvalue ref.)
T&&	&& :	T&& (rvalue ref. to rvalue ref: rvalue ref.)

*-----

```
int main()
{
    int ival = 10;

    auto&& x = ival;
    // int& x = ival; // üstteki ile karşılaştırınca, auto'ya karşılık gelen tür lvalue ref.
    // bu durumda rvalue ref.'e lvalue ref. oluştu. Reference collapsing'den lvalue ref. oluşur.
}
```

10)

C'de bir türe eş-isim vermek için typedef anahtar sözcüğü kullanılır.

C++'da using anahtar sözcüğü ile de yapılır. typedef de geçerli !!

*-----

```
typedef int WORD;  
using WORD = int;
```

11) decltype 'da ifadenin tür sıktırı : decltype((x))

decltype operandı olan ifadenin value category'sı

- a) PR value expr. => T
- b) Lvalue expr. => T&
- c) X value expr. => T&&