

* Namespace: • İsimler aynı kopyalanmaz scope

* C'de

File
Block
Function Prototype
Function

C++'da

Namespace Scope
Class Scope
Block Scope
Function Prototype Scope
Function Scope

→ Namespace oluşturma: namespace isim {

}

sonunda noktalı virgül olmaz
terminator

→ Nec namespace

```
namespace ali {  
    int x = 10;  
}  
  
namespace veli {  
    double x = 3.4;  
}
```

Farklı namespace
aynı isim olabilir

```
namespace ali {  
    int x = 10;  
}  
  
namespace veli {  
    double x = 3.4;  
}  
  
int main()  
{  
    ali::x  
}
```

İsimler farklıdır di → qualified

ali namespace

• Fonksiyon bloğu içinde namespace oluşturulmaz !! → Global namespace'e tanımlar.

• Başlık dosyasından gelenler hep std namespace içindedir.

• Namespace'in access kontrolü yok !! → private gibi
→ protected

```
namespace ali {  
    void foo(int);  
}  
  
namespace veli {  
    void foo(int, int);  
}
```

Farklı scope olduğu
için
Function
overload
değil.

```

namespace nec {
    namespace ali {
        namespace veli {
            int x;
        }
    }
}

int main()
{
    nec::ali::veli::x
}

```

→ nested namespace

```

namespace nec {
    int a, b, c;
}

namespace nec {
    int x, y;
}

```

x ve y'de karga dahi olur.

- Compiler bu iki tanımlı birleştiriyor → **kamusalite**
- Böylelikle birden fazla header dosyası tek bir namespace'e eklenir uygulanır

→ Qualified olarak bildirimin true style gelir:

- using declaration
- using namespace directive (declaration)
- ADL = Argument Dependent Look-up

*→ Using Declaration:

```

namespace ali {
    int x, y, z;
}

int main()
{
    using ali::x;

    x = 10;
    ali::x = 20;
}

```

→ Burada başka int x tanılamaz!

• Birden fazla tanımlı yapılan scope'a enjekte etmiş olduk



• namespace'te tanımlanmış sayıların.

```

namespace ali {
    int x y, z;
}

namespace veli {
    using ali::x;
}

int main()
{
    veli::x = 10;
}

```

Ar'deki x bu namespace'e enjekte edildi

Arke Vel'i'ni'de x'ivar.

```

int g = 10;

namespace nec {
    int x = 4;
}

namespace ali {
    using ::g;
    using nec::x;
    int y = 10;
}

int main()
{
    ali::g = 10;
    ali::x = 10;
    ali::y = 120;
}

```

→ statik ümörto olur.

globaldeki g

```
#include <iostream>
```

```

int func()
{
    std::cout << "func is called!\n";
    return 5;
}

namespace nec {
    int x = func();
}

int main()
{
    std::cout << "main starts\n";
}

```

→ global değıken

Aslında bu da global namespace içinde tanımlıdır bir değıken. 1 numara geygi geyrür. Sadece 2 olur.

1

2

*Fakat her zaman ilk fonksiyon tanımlıdır deklarasyon

→ Using Namespace Direktif:

- Bir namespace'te ne var ne yok kullanmak için. Bu bildirimin de kapsamı vardır.
- Bildirime konu isim alanı içindeki isimleri bildirildiği isim alanına enjekte ETMEZ!!

```
namespace ali {  
    int a, b, c;  
}  
  
int main()  
{  
    using namespace ali;  
  
    a = 5;  
    b = 7;  
    c = 10;  
}
```

Ayrıca bu fonksiyon scope'inde tanımlı.

Bunlar legal.

```
#include <iostream>  
  
namespace neco {  
    int x, y;  
}  
  
using namespace neco;  
  
int x;  
  
int main()  
{  
    //x = 5;  
}
```

→ bu bildirim legal. Çünkü
using namespace → enjekte etmez

Fakat bu ambiguity

```
namespace ali {  
    void foo();  
}  
  
namespace veli {  
    int foo = 5;  
}  
  
using namespace ali;  
using namespace veli;  
  
int main()  
{  
    foo();  
    foo = 5;  
    foo;  
}
```

here ambiguity

→ ADL: Argument Dependent Lookup: Yerli Lookup

```
namespace ali {  
    class A {  
    };  
    void func(A);  
}  
  
int main()  
{  
    ali::A ax;  
    func(ax);  
}
```

→ Fonksiyona, bir namespace içindeki bir file için
orjinal gönderiyorsa k, fonksiyonun ismini
nitelenmesek bile, ilgili namespace'te de arar
↓
buna yada de arar
ilgili namespace'te de

```
#include <iostream>
```

```
int main()  
{  
    //std::cout << "merhaba dünya";  
    operator<<(std::cout, "merhaba dünya");  
}
```

ADL olduğundan overload func'a
std::yineye göre yola

→ ADL olmasa
(std::cout) olduğu
yol overload olamazdı.

```
namespace nec {  
    class X {  
        //...  
    };  
    void func(X)  
    {  
        std::cout << "nec::func\n";  
    }  
}  
  
void func(nec::X)  
{  
    std::cout << "global func\n";  
}  
  
int main()  
{  
    nec::X nx;  
    func(nx);  
}
```

→ bu da ambiguity


```

namespace nec {
    class X {
    };

    void func(X);
}

```

```

int main()
{
    int func(int);
    nec::X nx;

    func(nx);
}

```

Information hiding
hatası

```

#include <iostream>
#include <vector>

```

```

class MyClass {
public:
    friend void func(Myclass);
};

```

Friend bildirimi olsa
bile,
func bulunamaz!!!

```

int main()
{
    MyClass m;
    func(m);
}

```

Fakat su an ADL deneye
girer ve buldu

→ Unnamed Namespace: - C'deki global static variable ile aynı.

```

namespace {
    int x = 10;
    void foo();
    class MyClass {
    };
}

```

→ C++ C'teki global static ile
gizlemek deprecated.

→ Namespace Alias:

- C'de **typedef** ile bir eş isim tanımlarken, C++'da **using** ile yapılır böyle gelir.

- Çünkü template kullanıma uygun

```
typedef int (*fcmp)(const void*, const void*);  
using fcmp = int (*)(const void*, const void*);
```

- Bir namespace nested ise, bunu yapmak zorunlu olabilir.

```
#include <iostream>  
  
namespace ali {  
    namespace veli {  
        namespace necati {  
            int x = 5;  
        }  
    }  
}  
  
namespace project = ali::veli::necati;  
  
int main()  
{  
    project::x = 5;  
}
```

namespace alias

- Version kontrolünde kullanılıyor.

- Fakat namespace isimleri de değiştirilebilir.