

→ C ve C++ arasındaki syntax farkından devam.

* Hem C'de hem C++'da void pointer'a herhangi bir türden değişkenin adresi ilk değer olarak verilir.

* Fokot tersi, / void * vptr = &x / → C++'da syntax hatası
int * iptr = (int*) vptr / → C'de hiç de olsa hata değil.

→ User Defined Type'lerdeki Farklar:

- C'de user defined type oluşturmak için:
 - structures kullanılır.
 - unions
 - enums

• C++'da bunlara ek olarak class gelir. Fakat C structure ile C++ structure aynı değil. C struct → C++ class gibi davranır.

• C dilinde struct'ın bir elemanı olmak zorunda C++'da ise zorunlu değil → Not empty class

struct Data {
int x → C++'da yazmak zorunda
} değiliz.

→ Typedef bildirimi: • C dilinde typedef bildirimi olmadan union, struct ve enum'ın tag'i ile çalıştırılmaz. Tagler tanımlanmadığı için.
(typedef alias)

- union Data mydata
struct Data mydata
:
:

gibi kullanılır.

• Benzer olmadan kullanmak için typedef kullanımı gerekir.

• C++'da typedef olmadan, direkt tag kullanılabilir.

→ ENUM: • C dilinde

enum Color { Red, Black, Green }

↳ underlying type:
int olarak garantiye
alınmıştır.

• C++'da

↳ underlying type: belz. belirtilebilir.

* enum Color :: unsigned long { R, G, B };
type u. long
yaptık

• C dilinde farklı enum türleri birbirine dönüştürülebilir.

→ enum Color { R, G, B }
enum Pos { ON, OFF }

enum Color myColor = OFF

→ C'de mümkün

→ C++'da hata

int val = OFF } → C++'da mümkün ama yapma

• C++'da 2 adet enum geldi

1) Scoped Enum / Enum class → enum class Color { R, G, B } gibi kullanırız.
2) Unscoped Enum

→ enum ScreenColor { Red, Black, White }
enum TrafficColor { Red, Yellow, Green }

Farklı da yapılabilmekteydi ama bu şekilde yazarak da RED kullanması hataya sebep olur. Aynı scope'a birlikte kullanımlar

→ enum class ScreenColor { Red, Black, White }
enum class TrafficColor { Red, Yellow, Green }

int main() {

ScreenColor myColor = ScreenColor::Red → seçilerek kullanılır.

ScreenColor myColor = Red X → syntax error

colon colon
scope resolution operator.

→ Scoped Enum / Enum class kullanımı

* Type Deduction (Tür Çıkarımı):

→ Değişkeni bir tür belirtmesek bile bir tür çıkarımında bulunabilir. Ancak int yazmadık.

örn: auto x = 10
→ int olarak kendi çıkarımında bulunur.

* int main() {
int x = 10; → initialization
x = 35 → assignment
}

→ Default Initialization:

```
int main() {  
    int x; // ilk deger yok.  
}
```

→ compiler default init edilemez.

→ Zero Initialization:

```
int x; } Ove trader  
int main() {  
}
```

→ Copy Initialization:

```
int main() {  
    int x = 10; } ilk deger verildi.
```

→ Direct Initialization:

```
int main() {  
    int x(20); } → ilk deger kaye verilebilir.  
    int y(); } → Bu initialization degil. Function declaration  
}
```

→ Uniform (Brace) Initialization (C++ 11'de geldi)

```
int main() {  
    int z{40}; } → ilk deger kışen parantez içinde verilebilir.
```

⇒ Birun faydası narrowing conversion'inin önüne geçer.

↓
örneğin double
değer int bir değeri
inttiğebilir. kayıplı olabilir.

⇒ Bir diğer faydası most vexing parse durumunu ortadan kaldırır.

↓
boş kodlar
hem fonksiyon bildirimi
hem de nesne tanıtımı
olabilir. Dikkat! kurallara göre
fonksiyon bildirimi önceliklidir.

Bu durumda şartı meşas most vexing parse
denmiştir.

⇒ Uniformdur. Her yerde geçerlidir.

→ Value Initialization:

```
int main() {  
    int x; → Garbage / indetermined init  
    int y{}; } → value init → deger 0  
                → pointer nullptr  
                → boolean false
```

→ Namespace:

- C'de her adet kod alanı var → global namespace
↳ local namespace → fonksiyonlar

(İsim alanı)

- C scope türleri → file scope
→ block scope
→ function prototype scope
→ function scope

- C++'da C'deki file scope gitir → namespace scope geldi
→ block scope
→ function scope
→ function prototype scope
→ Class scope eklendi.

Örn/: namespace nec {
int x = 10;
}

⇒ Standart kütüphanenin bütün isimleri namespace'de. std namespace.

using namespace std;

```
int main() {  
    nec::x = 20;  
}
```

nec namespace'indeki
x'te başka derleyici

→ Qualified Name: Ya c++'da (::) operandı, ya (.) operandı, ya da → operandı olmali

std::size_t → Qualified name c++'da resolution operandı var.

→ Function ve Operator Overloading:

```
int main() {  
    int ival = 10;  
    std::cout << "ival = " << ival << "\n";  
}
```

↓
string
değişkeni

↓
string

Operator overloading

Bu string, fonksiyona
argüman olarak kullanılır.

- Fonksiyon çağırmanın bir diğer yöntemi de operator overloading.

• String'i yandıran fonksiyon ile int/double... yandıran fonksiyonların yapısı farklı ama yine cout ile yandırabiliriz. Buna function overloading denir.

- cout kullanırken kendisi return ediyor. C++'da deniyor

• Java method → C++ member function.

• cout << "merhaba" → merhaba yazdır
cout.operator<<("merhaba"); → stringin adresini
yazdır.

* using namespace std
→ namespace ism adismana engel olur.

* Default Function Declaration / Implicit Function Dec:

* Yalnızca C'de var.

* Derleyici name lookup da bir fonksiyonu bulamazsa int return türünde varsayılan olarak değer atıyor.

* C++'de syntax error.

* Fonksiyonlarda Varsayılan Argüman: (Default Argument)

→ C'de buna izin verilmiyor. C++'de var. → void func (int, int = 10);

```
int main() {  
    func(3) → (3, 10)  
    func(3, 6) → (3, 6)  
}
```

→ Varsayılan Argüman belirtmek için, 0 argümanın sağlanabilen her şey varsayılan argüman dır.

* Derleme zamanında yapılır. Runtime'e etkisi yok.

* Bildirimi yapıldıysa, tanımlı yapılmaz.

* Re-declaration yapılabılır: `foo(int, int, int = 5)` } → ikinci defa int denir, —, 20, 5 değeri alır.
`foo(int, int = 20, int)`

↓
normalde bu syntax hatası

