→ C ve C++ arasındaki syntax farkından devam.

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

* Fakat tersi, `void * vptr = dx` / `int * iptr = (int*) vptr` → C++'da syntax hatası / → C de hata de olsa hata değil.

→ User Defined Type'lardaki 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 daha çok.

• C dilinde struct'ın bir elemanı olmak zorunda C++'da ise zorunlu değil → Adı 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 değişken kullanılamaz. Tagler tür belirlemediği için.
(typedef callback)

- union Data mydata
  struct Data mydata
  ⋮

gibi kullanılır.

• Bunlar olmadan kullanmak için typedef kullanımı gerekir.

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

→ ENUM: • C dilinde

```
enum Color { Red, Black, Green };
```

↳ underlying type'ı int olarak garantiye alınmıştır.

• C++'da
↳ underlying type'ı biz belirleyebiliriz.

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

• C dilinde farklı enum türleri birbirine dönüşebilir.

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

enum Color myColor = OFF → C'de mümkün
→ C++'da hata

int level = OFF → C++'da mümkün ama yapmayız

• 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 deyimlerden bile geliyor olsalar tanımında da RED bulunması hataya sebep olur. Aynı scope'ta birlikte bulunamazlar

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

→ Scoped Enum / Enum Class kullanımı

int main() {

ScreenColor myColor = ScreenColor :: Red → şeklinde kullanılır.

colon colon
scope resolution operator.

ScreenColor myColor = Red X → syntax error

}

*Type Deduction (Tür Çıkarımı):

→ Derleyici biz belirtmesek bile bir tür çıkarımında bulunabilir. Aslında int yapmadık.

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

* int main() {

int x = 10; → initialization

operatör değil

x = 35 → assignment

operatör

}

→ Default Initialization:

```
int main(){

    int x; // ilk değeri yok.

}
```

→ const lar default init edilemez.

→ Zero Initialization:

```
int x; ) 0 ve tanımlar

int main(){

}
```

→ Copy Initialization:

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

→ Direct Initialization:

() direct

```
int main(){

    int x(20); ) ilk değer böyle verilebilir.

    int y(); ) Bu initialization değil. Function declaration

}
```

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

```
int main(){

    int z{40}; ) ilk değer küşeli parantez içinde verilebilir.

}
```

⇒ Bunun faydası narrowing conversion'ının önüne geçer.

Örneğin double değer int bir değişkeni initleyebilir. Kayıplı olarak.

⇒ Bir diğer faydası most vexing parse durumunu ortadan kaldırır.

bazı kodlar hem fonksiyon bildirimi hem de nesne tanımı olabilir. Derin kuralına göre fonksiyon bildirimi önceliklidir. Bu duruma sadece most vexing parse denmiştir.

⇒ Uniformdur. Her yerde geçerlidir.

→ Value Initialization:

```
int main(){

    int x; → çöpleşe / indetermined init
    int y{}; → value init → değeri 0
                → pointerlar nullptr
                → boollar false

}
```

→ Namespace: • C'de tüm 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 gitti → namespace scope geldi
→ block scope
→ function scope
→ function prototype scope
→ Class scope eklendi.

Örn:
```
namespace nec {
    int x = 10;
}

int main() {
    nec::x = 20;
}
```
nec namespace'indeki x'i arıyor derleyici

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

using namespace std

→ Qualified Name: Ya çözünürlük (::) operandı, ya (.) operandı, ya da → operandı olmalı

std::size_t → Qualified name çünkü resolution operandı var.

→ Function ve Operator Overloading:

```
int main() {

    int tval = 10;
    std::cout << "tval = " << tval << "\n";

}
```
cout: sınıf değişkeni
<<: Operator overloading
"tval = ": string
Bu string, fonksiyona argüman olarak iletiliyor.

• Fonksiyon çağırmanın bir diğer yöntemi de operator overloading.

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

• cout çalışırken kendini return ediyor. Chaining deniyor

• Java method → C++ member function.

• cout << "mert\n" → mert yazar
cout.operator<<("mert"); → stringin adresini yazar.
(const char* / void*)

* using namespace std
→ namespace isim çakışmasına engel olur.

* Default Function Declaration / Implicit Function Dec:

* Yalnızca C' de var.

* Derleyici name lookup da bir fonksiyonu bulamazsa int returning türünde vardır diyip yaratıyor.

* C++' da syntax error.

* Fonksiyonlarda Varsayılan Argüman: (Default Argument)

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

```
int main() {

  func (3) → (3, 10)
  func (3,6) → (3,6)
}
```

→ Varsayılan Argüman belirlemek için, O argümanın sağındakilerin hepsi varsayılan argüman olmalı

* Derleme zamanında yapılır. Runtime'a etkisi yok.

* Bildirimde yapıldıysa, tanımda yapılmaz.

* Redeclaration yapılabilir: foo (int, int, int = 5) → sırası değişmek, _, 20, 5 değerlerini alır.
foo (int, int = 20, int)
↳ normalde bu syntax hatası