→ Function Overload Resolution ile devam:

→ İki durumda resolve edemez. 1) NO-match
2) Ambiguity

* Farklı kalite seviyesinde Dönüşüm: Varsayılan Conversion <
User Define Conversion <
Standard Conversion <

Exact match > Promotion > Conversion

* Type Cast Operators:

• implicit : örtük - gizli → talimat vermeden compiler yapar.

• explicit: fiilen typecast yapılması

* C dilinde 1 adet typecast var fakat C++'ın diğer typecast operatörleri var. : → static cast
→ const cast
→ reinterpret cast
→ dynamic cast

kullanılabilir ama kullanılmamalı.

1. static cast: Bizim kontrolümüzde olan, otomatik yapılabilen. + enumları C++'da int'e static cast edebil C'deki enum gibi kalır.

2. Const Cast: Const nesneden, low level const pointerdan + volatile ifadeleri düşürebilen.
→ program akışı kaynaklı değiştirmiyor.

3. Reinterpret: Nesnenin bitlerini, adresini farklı şekilde kullanmak için:
→ en iyi örnek, biz .bin (formatsız) dosyayı char* olarak okuyoruz. Bunu C++'da reinterpret_cast<char*> yap.

⇒ static_cast < int > (dval)
cast adı < hedef tür > (değişkenler)

* Enums:

→ Aritmetik türden enum'a otomatik dönüşüm yok. Ama enum türden aritmetik türe otomatik dönüşüm var. ← static_cast

* Modern C++'da: Scoped enums eklendi.

* enum Color : unsigned char {red, green, blue}
  ↓
  underlying type'ı belirtebiliriz.

* enum class Color {red, blue, green} → yine underlying type belirtilebilir.
  → Artık bu enumların kendi scopeları var.

```
enum class Color {red, blue, green};

int main()
{
    auto c = Color::red;
}
```

→ bu şekilde belirtilmezse red'i bulamaz.

```
enum class Color {red, blue, green};
enum class TrafficLight{red, yellow, green};
enum class TexBoxColor{red, yellow, green};
enum class WindowBaseColor{red, yellow, green};

int main()
{
    Color c = Color::red;
    TrafficLight tl = TrafficLight::red;
}
```

→ Scopeları farklı olduğu için aynı isimlendirme yapabiliriz.

→ Bu enumların sayı değerleri olarak kullanmak için

⇒ static_cast<int>(c)

```cpp
enum class Color {red, blue, green, brown, magenta, black};

void func()
{
    using enum Color;

    auto c1 = red;
    auto c2 = red;
}

int main()
{

}
```

using kullanarak bu scope'ları visible hale getirebiliriz.

* C tekrar:

- name lookup: Derleyicinin bir ismin, hangi varlığa ait olduğunu anlama sürecidir.
- context control: name lookup'tan sonra (isim bulunduktan sonra), name lookup bitti, sonra bağlamı kontrol edilir.

* Scope Resolution Operator:

```cpp
#include <cstdio>

int foo();

int main()
{
    int foo = 6;

    ::foo();
}
```

artık foo'yu global namespace'de ara. Eğer olmasaydı fonksiyon scope'unda farklı foo olduğu için hata verirdi.