13 Nisan 2022 (6.Ders)

endl neden kullanmamalıyız ??
standart çıkış akımının buffer'ini flash etmek (fiziksel olarak dosyaya yazılmamış bellek alanına -buffer yazılmış byte'ların fiilen dosyaya yazılması) istemiyorum. Çünkü bunun bir maliyeti var.

implicit (örtülü) dönüşüm: derleyicinin durumdan vazife çıkartarak yaptığı dönüşüm

explicit dönüşüm: kod yazarak yapılan dönüşüm

type-cast operators (tür değiştirme operatörleri)

a) static cast
b) const cast
c) reinterpret cast
d) dynamic cast → runtime polymorphism ile ilgili

a) static cast: does implicit conversions between types (such as int to float, pointer to void*)

```
char* strchr(const char* p, int c)
{

}
```

char* → Sahibi ben değilim Sahibi karar versin.

const char* → mutator değilim sözü, aldığım adresteki değişkeni değiştirmem

<> : angular bracket
static_cast <int> (dval)

```
int x = 10;
int y = 20;
double d1 = x/y;     // bölme işlemi int olarak yapılır, d1'e 0 ile ilk değer vermiş olunur
double d2 = static_cast<double> x/y;    // bölme işlemi double olarak yapılır, d2 = 0.5 olur.

int ival = dval;  // sentaks hatası değil, double'ın ondalık kısmı kaybolur, tanımsız davranış olabilir, int sayı sınırlarını aşıyorsa tanımsız olur.
int ival{dval} ; // C++ 'da narrowing conversion, sentax hatası olacaktı.
```

b) const cast : const'luğu kaldırmak

```
char* Strchr(const char* p, int c)
}
    return (char *) p;  // const_cast
}
```

c) reinterpret cast:

It is used to convert a pointer of some data type into a pointer of another data type, even if the data types before and after conversion are different.

```
int main(){
    int x = 235455;
    char* p = &x; // C++'da error
    char* p =(char*)&x;   // isteyerek kullanım
}
```

Syntax :
data_type *var_name = reinterpret_cast <data_type *>(pointer_variable);

```
int ival{};
const  int* iptr = &ival;
char* cp = reinterpret_cast<char* > (iptr);  // yapmazsınız, tek bir dönüşüm yok,
int*'dan char*'a ve const dönüşümü var.

*-----------------------------------------------------------------------------------------------

int ival{};
const  int* iptr = &ival;
char* cp = reinterpret_cast<char* > (const_cast<int*>  (iptr));
veya
char* cp = const_cast<char*> (reinterpret_cast <const char*> (iptr));
```

[[nodiscard]] // attribute

derleyici bir fonksiyonun geri dönüş değeri kullanılmadıysa uyarı vermesi sağlanabilir

```
[[nodiscard]]
int foo();

int main(){
    foo();  // derleyici fonksiyonun geri dönüş değeri kullanılmazsa hata verir
}
```

fonksiyonun delete edilmesi :

new-delete ile karıştırma !!

```
void func(int) = delete;
// bu fonksiyon var, derleyici ricamız eğer bu fonksiyona çağrı yapılırsa
bunu sentax hatası olarak bildir


void func(double);
void func(int );
void func(long ) = delete;

func(1.2); // double çağırılır
func(1.2f ); // double çağrılır
func(10L);  // sentax hatası verir
```

mülakat sorusu : öyle bir foo fonsiyonu olsun sadece legal olarak int ile çağrılabilsin ?

```
void func(int);
void func(double) = delete;
```

bunu bütün türler için yapmak zor

```
void func(int);
template <typename T>
void func(T) = delete;
```

C++ 23'de
"really" anahtar sözcüğü var

```
void func(really int);  // sadece int argüman kabul eder !!
```

ENUM:

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

Color myColor{blue};
myColor = 1; // hatalı, aritmetik türlerden enum türüne dönüşüm yok


Color myColor{blue};
int ival{2}
myColor = static_cast<Color> (ival);
```

*----------

```
Color myColor{blue};
int ival;
ival = myColor;  // legal, tehlikeli, bu yüzden enum class eklendi !!!
```

C ==> enum türünü int olarak kabul etmiş, C++ => int olma garantisi yok

```
enum ScreenColor{Red, Magenta, Brown};
enum TrafficLight{Red, Yellow, Green};  // enum scope'unda kaynaklı, Red isim çakışması
```

Çakışan enum'ları Namespace ile ayırmak

```
namespace Neco {
    enum ScreenColor{Red, Magenta, Brown};
}
```

*----------

scoped enum (kapsamlandırılmış enum) // class değil, C++ class'ın class olmayan tür nitelemesi var !!

```
enum class Color{Red, Yellow, Green};
enum class TrafficLight{Red, Yellow, Green};  // Red isim çakışması olmaz   // class değil

int main(){
    auto c =  red;  // sentax hatası
    auto c = Color::red;
    Color c = Color::green;
    TrafficLight tl = TrafficLight::green;
}
```

```
enum class Color{Red, Yellow, Green};

int main(){
    Color mycolor { Color::green};
    int ival = mycolor;  // sentax hatası, enum class hata verdirdi, sadece enum olsa derleyici hata vermez
}

int main(){
    Color mycolor { Color::green};
    int ival = static_cast<int> mycolor;  // bilerek yapıyorsak, derleyici hata üretmez
}
```

```
enum class Color{Red, Yellow, Green};

int main(){
    const char* const pcolors[] = {"red", "blue", "green"};

    Color myColor;

    pcolors[myColor]; // derleyici hata verir, burada int'e dönüşüm yok
    pcolors[static_cast<int> (myColor)];
}
```
// Necati hoca esas burada using gerekir diyor !!!!, C++20'de öneri kabul edildi.
// kontrollü gevşetme

```
enum class Color{Red, Yellow, Green};

void func(){
    using enum Color;  // dikkat ! using kullanılınca, enum class Color olmaz !!

    auto c1 = red;
    auto c2 = brown;
}
```

*----------

C++20'de

```
namespace neco{
    enum class Color{Red, Yellow, Green};
}

void func(){
    using enum neco::Color;  // dikkat ! using kullanılınca, enum class Color olmaz !!

    auto c1 = red;
    auto c2 = brown;
}
```

enum => aritmetik tür'e otomatik dönüşüm var // C++ gençlik hatası
enum class => aritmetik tür'e otomatik dönüşüm yok, bilerek dönüşüm yapmak istiyorsak static_cast'i kullanıyoruz

özet : enum class veya using kullanın, isim çakışması olmaz

Derleyici süreçleri (C++ mülakatlarında sorulur !!)
1) name lookup (hangi tokenların isim olduğunu anlaması)
2) context kontrol (fonksiyon olup olmadığı) → (function, variable)
3) access kontrol → C'de yok, sınıfın her elemana erişim kontrolü

namelookup: (çok iyi anlaşılmalı)

derleyicinin bir ismin hangi varlığa karşılık geldiğini anlama sürecidir.
Namelookup herşeyin başıdır. İlk önce yapılır.

C++ dilinde (istisna olmadan) bir ismin aranıp bulunamaması sentax hatasıdır.

name lookup:
1) isim arama süreci aranan ismin bulunmasıyla sona erer. (bir daha devam etmez)
2) isim arama dilin karmaşık kurallarınca belirlenen bir sırayla yapılır.

name lookup 2 kuralını bozan şeyler var
scope resolution operatör ::
::x => x'i global isim alanında arar.

```
int x = 10;  // dördüncü olarak global isim alanında aranacak

void func()
{
    // üçüncü olarak bu blok içerisinde arayacak
    // code
    if(1){  // ikinci olarak bu blok içerisinde arayacak
        //code
        if(1){
            x = 5;  // ilk önce bu blok için x'i arayacak
        }
    }
}
```