Numaralandırma Sınıfları

`C++11` ile gelen araçlardan biri de numaralandırma sınıfları. Standartların kullandığı ingilizce resmi terim `"strongly typed enums"`. "Güçlü tür özelliği kazandırılmış numaralandırma türleri" olarak Türkçeye çevirebiliriz. Numaralandırma sınıflarının dile eklenmesiyle birlikte artık `C++`'da iki ayrı numaralandırma aracı var. Geçmişten gelen ve bazı eklemelerle halen varlığını koruyan numaralandırma türlerine bundan sonra "düz numaralandırma türleri" `(plain enum)` diyeceğiz. Numaralandırma sınıflarının dile neden eklendiğini anlayabilmek için, düz numaralandırma türlerinin `C++11` öncesinde yol açtığı tipik sorunları anlayabilmemiz gerekiyor:

numaralandırma sabitlerinin bilinirlik alanları

Düz numaralandırma türlerinde, tanıtılan numaralandırma sabitlerinin bilinirlik alanı numaralandırma türünün bilinirlik alanıdır. Bu durum numaralandırma sabitleri olan isimlerin aynı bilinirlik alanında bulunan diğer isimlerle çakışma riskini arttırır. Çok sayıda kütüphanenin bir arada kullanıldığı projelerde numaralandırma sabitleri olarak kullanılan isimlerin çakışması çok sık karşılaşılan bir durumdur:

```
//traffic.h
enum TrafficLight {Yellow, Green, Red};

//screen.h
enum ScreenColor {White, Gray, Green, Red, Magenta, Brown, Black};

//simulation.cpp
//#include "traffic.h"
//#include "screen.h"
```

Yukarıdaki kodda `traffic.h` başlık dosyasında tanıtılan enumTrafficLight isimli bir tür var. Bu başlık dosyası ile doğrudan bir ilişkisi olmayan `screen.h` isimli başlık dosyası ise enum ScreenColor isimli bir türü tanıtmış. `simulation.cpp` isimli kod dosyası her iki modülün sağladığı hizmetlerden faydalanabilmek için iki başlık dosyasını da #include önişlemci komutlarıyla kendi kaynak dosyasına eklemiş. Bu durumu derleyici bir sentaks hatası olarak işaretleyecek. `Green` ve `Red` isimleri çakışıyor. Aynı bilinirlik alanında birden fazla farklı varlık aynı ismi taşıyamaz, değil mi? Numaralandırma sınıflarının sağladığı faydalardan ilki burada. Numaralandırma sınıfı

türlerinin kendi bilinirlik alanları var ve tanıtılan numaralandırma sabitleri, numaralandırma sınıfının kendi bilinirlik alanında yer alıyor. Yukarıdaki kodda şimdi numaralandırma sınıflarını kullanıyoruz:

```
//traffic.h
enum class TrafficLight {Yellow, Green, Red};

//screen.h
enum class ScreenColor {White, Gray, Green, Red, Magenta, Brown, Black};

//simulation.h
//#include "traffic.h"
//#include "screen.h"

void func()
{
 auto traffic_light = TrafficLight::Green;
 auto screen_color = ScreenColor::Green;
 //
}
```

Numaralandırma sınıflarına ilişkin sabit isimleri numaralandırma sınıf ismiyle nitelenmek zorunda. Aşağıdaki gibi bir kullanım geçerli değil:

```
ScreenColor scr_color = Magenta; //geçersiz
```

Düz numaralandırma türleri ile tanıtılan numaralandırma sabitlerinin bir bilinirlik alanı olmamasına karşın `C++11` artık onları da numaralandırma tür ismiyle niteleyerek kullanabiliyoruz:

```
enum Suit {Club, Diamond, Heart, Spade};

Suit s = Suit::Diamond;
```

`C++11` öncesi numaralandırma sabitlerinin isim çakışmalarından korunması için bir yapı ile sarmalanması sık başvurulan bir yoldu:

```
struct SuitWrapper {
 enum Suit { Club, Diamond, Heart, Spade };
};

SuitWrapper::Suit s = SuitWrapper::Diamond;
```

Böyle bir sarmalamanın görece bir avantajı, sarmalayan sınıftan türetme yapılabilmesi. Ancak numaralandırma sınıflarından türetme yapılması olanağı yok. İsim çakışmasından korunmak için kullanılan bir başka yöntem de numaralandırma türünün tanımını bir isim alanı içine almaktı:

```
namespace Neco {
    enum Suit {Club, Diamond, Heart, Spade};
}

Neco::Suit s = Neco::Diamond;
```

düz numaralandırma türlerine ilişkin sorunlu tür dönüşümleri

Düz numaralandırma türlerine yalnızca aynı türden değerler atanabilir, yani diğer türlerden düz numaralandırma türlerine otomatik `(implicit)` tür dönüşümü yoktur:

```
enum Color { White, Yellow, Gray, Green, Brown, Black };
enum Font { Arial, Verdana, TimesNewRoman, Courier, Helvetica };

int main()
{
 Color c1 = Yellow; //geçerli
 Color c2 = 3; //geçersiz
 Font f1 = Verdana; //geçerli
 Font f2 = Black; //geçersiz
 c1 = f1; //geçersiz
}
```

Yukarıdaki kodda geçersiz atamaları görüyorsunuz. Numaralandırma türlerine diğer türlerden otomatik tür dönüşümü olmaması yanlış yazımlara karşı önemli bir koruma sağlar. Eğer, küçük bir olasılıkla da olsa, geçersiz olan ilk değer verme ya da atama işlemleri istenerek yapılıyorsa `static_cast` tür dönüştürme işleci kullanılabilir:

```
enum Color { White, Yellow, Gray, Green, Brown, Black };
enum Font { Arial, Verdana, TimesNewRoman, Courier, Helvetica };

int main()
{
 Color c1 = Yellow;
 Color c2 = static_cast<Color>(3);
 Font f1 = Verdana; //geçerli
 Font f2 = static_cast<Font>(Black);
 c1 = static_cast<Color>(f1);
 //
}
```

Ancak düz numaralandırma türlerinden tamsayı ve gerçek sayı türlerine otomatik `(implicit)` tür dönüşümü var. Aşağıdaki koda bakalım:

```
enum Color { White, Yellow, Gray, Green, Brown, Black };

int main()
{
 Color c = Green;
 int cnt = 0;
 //
 int ival = c;
 //
}
```

Yukarıdaki kodda `int` türden `ival` değişkenine `Color` türünden `c` değişkeninin değeriyle ilk değer verilmiş. Dilin kurallarına göre kod geçerli. `Color` türünden `c` derleyici tarafından `int` türden `3` değerine dönüştürülecek. Belki de kodlayıcımız ilk değer verme işleminde `cnt` ismini kullanmak yerine yanlışlıkla `c` ismini yazmıştı. Numaralandırma sınıflarında artık diğer türlere otomatik tür dönüşümü geçerli değil:

```
enum class Color { White, Yellow, Gray, Green, Brown, Black };

int main()
{
 Color c = Color::Green;
 int cnt = 0;
 //
 //int i = c; //geçersiz
 int j = static_cast<int>(c); //geçerli
 //
}
```

Yukarıdaki kodda `i` değişkenine verilen ilk değer geçerli değil. Çünkü `Color` türünden int türüne otomatik tür dönüşümü yok. Ancak `j` değişkenine ilk değer verilirken `Color` türünden değer `static_cast` işleciyle int türüne dönüştürülüyor. Kod geçerli.

numaralandırma türlerinin baz türleri

Hem `C` 'de hem de `C++` 'da da numaralandırma türleri için derleyici arka planda bir tamsayı türü kullanır. Bir numaralandırma türüne ilişkin derleyici tarafından arka planda kullanılan tamsayı türüne ingilizcede `"underlying type"` deniyor. Biz bu anlamda "baz tür" terimini kullanacağız. `C` 'de numaralandırma türlerinin baz türü her zaman `int` türüdür. Yani bir `C` kodunda `enum Data` bir tür olmak üzere

```
assert(sizeof(enum Data) == sizeof(int))
```

ifadesi her zaman doğrulanır.

Ancak `C++03` 'de bir numaralandırma türüne ilişkin baz türün ne olacağına karar veren derleyicidir. Derleyici numaralandırma türü karşılığında işaretli ya da işaretsiz `char, short, int, long` türlerinden birini baz tür olarak seçebilir. Yine derleyici `C++03` 'de farklı numaralandırma türleri için farklı baz türler seçebilir. Baz türün seçiminde derleyici yalnızca şu kurallara uymak zorundadır: Tüm numaralandırma sabitleri seçilen baz türde temsil edilebilmelidir. Tüm numaralandırma sabitleri eğer `int` ya da `unsigned int` türünde ifade edilebiliyor ise baz tür olarak daha büyük bir tür seçilmeyecektir. `int` ürünün 32 bit olduğu bir sistem için kodların derlendiğini düşünelim:

```
enum Pos {Off, On};
```

Kullanılan numaralandırma sabitleri `char, signed char, unsigned char` türlerine sığıyor. `C++03` 'te derleyici baz tür olarak bu türlerden birini seçebileceği gibi doğrudan int türünü de baz tür olarak kullanılabilir.

```
enum BufferSize{DefaultSize = 10000000, LargeSize = 20000000};
```

Yukarıdaki bildirimde ise BufferSize türü için derleyici baz tür olarak `C++03` 'te `int` türünü seçecektir. Numaralandırma türlerine ilişkin baz türlerinin derleyiciden derleyiciye değişebilmesi hem taşınabilirlik sorunları oluşturuyor hem de bir numaralandırma türünün ön bildiriminin `(forward declaration)` yapılmasını engelliyor.

`C++11` standartlarıyla bu konuda önemli değişiklikler yapıldı:

- Hem düz numaralandırma türleri hem de numaralandırma sınıfları için bildirimde ya da tanımda baz tür belirlenebiliyor. Aşağıdaki örneklere bakalım:

```
enum Suit : unsigned char {Club, Diamond, Heart, Spade};

enum class BufferSize : unsigned int {
 SmallSize   = 10000u,
 DefaultSize = 100000u,
 LargeSize   = 1000000u
};

enum Color : unsigned char;

enum Font; //Geçersiz

enum class ErrorCode;

enum class ImageWidth : unsigned long;
```

Yukarıdaki kodda tanımlanan düz `enum Suit` türünün baz türü olarak `unsigned char` türü seçilmiş. Baz türün `enum` etiketi olarak seçilen isimden sonra gelen `":"` atomunu izlediğini görüyorsunuz. Numaralandırma sınıfı olarak tanımlanan `BufferSize` türünde ise baz tür olarak `unsigned int` seçilmiş. Düz numaralandırma türü olan `Color` yalnızca bildirilmiş ancak tanımlanmamış. `C++11` öncesinde numaralandırma türlerine ilişkin ön bildirim yapılamıyordu. Bildirimde baz tür olarak `unsigned char` türü seçilmiş. Düz numaralandırma türü olan `Font` yalnızca bildirilmiş ancak tanımlanmamış. Bildirimde baz tür belirtilmediği için bildirim geçersiz. Numaralandırma sınıfı olarak bildirilen `ErrorCode` türünde baz tür belirtilmemiş. Baz tür varsayılan biçimde `int` türü kabul edilecek. Numaralandırma sınıfı olarak bildirilen `ImageWidth` türünde baz tür olarak `unsigned long` türünün seçildiği belirtilmiş.

ön bildirim neden önemli?

Kodla tanımlanan türler `(user defined types)` söz konusu olduğunda derleyici bir türün tanımını görmese de o türün bildirimine dayanarak belirli bağlamlarda kullanımını geçerli kabul eder. Derleyicinin varlığından haberdar olduğu ancak henüz tanımını görmediği bir türe "tamamlanmamış tür" `(incomplete type)` denir. Tamamlanmamış türlerden gösterici değişkenler tanımlanabilir ya da tamamlanmamış türler işlev bildirimlerinde kullanılabilir. Tamamlanmamış türleri belirli bağlamlarda kullanabilmek için derleyici bu türlerin ön bildirimini görmelidir. Bir türü ön bildirimle kullanabildiğimiz durumlarda bu türün tanımını içeren başlık dosyasını kendi kodumuza dahil etmemiz gerekmez. Bu durumda başlık dosyalarının birbirine bağımlılığı ortadan kaldırıldığı gibi derleme süreleri kısalır. Ayrıca başlık dosyasından gelecek isimlerin çakışma riski ortadan kalkmış olur. Aslında tıpki gösterici değişkenlerde olduğu gibi numaralandırma türlerinden değişkenleri de numaralandırma türünün tanımını derleyiciye göstermeden tanımlayabiliriz. Ancak bunun için derleyicinin söz konusu numaralandırma türü için bellekte kaç `byte` yer ayrılacağını bilmesi gerekir. Aşağıdaki koda bakalım:

```
//fileoperations.h

enum ErrorCode : unsigned int;

class File {
private:
 ErrorCode m_ec;
 ///
};
```

`fileoperations.h` isimli başlık dosyasında tanımlanan `File` isimli sınıfın `private` veri öğelerinden biri `ErrorCode` numaralandırma türünden. `ErrorCode` türünün ön bildirimi yapılmış ve bildirimde baz türün `unsigned int` türü olduğu belirtilmiş. `C++11` öncesinde `File` sınıfının tanımının geçerli olabilmesi için derleyicinin `ErrorCode` türünün de tanımını görmesi gerekirdi. `ErrorCode` türünün `error.h` isimli bir başlık dosyasında tanımlandığını kabul edelim. Bu durumda `fileoperations.h` başlık dosyası `error.h` başlık dosyasını dahil edecekti:

```
//fileoperations.h

#include "error.h"

class File {
private:
 ErrorCode m_ec;
 ///
};
```

Oysa `C++11` kurallarına göre artık `ErrorCode` türünün ön bildirimi bu türden bir veri öğesini kullanabilmemiz için yeterli. Şimdi kazandığımız avantajlara bir bakalım: `fileoperations.h` başlık dosyasını dahil eden müşteri kodları `ErrorCode` türünün tanımını içeren başlık dosyasını dahil etmeyecekler. Böylece

- Derleme süresi kısalacak.
- `error.h` başlık dosyasında tanımlanan numaralandırma sabitleri olan isimler müşteri kodlar tarafından görülmeyeceği için isim çakışması riski söz konusu olmayacak: `ErrorCode` türünde onlarca numaralandırma sabiti tanımlanmış olabilir, değil mi?
- `error.h` başlık dosyasındaki değişimler `fileoperations.h` dosyasını dahil eden kodların değiştirilmesini ya da bu kodların yeniden derlenmesini gerektirmeyecek.