std::byte Türü

`C++17` standartları ile dile eklenen yeni türlerden biri std::byte. Yazdığımız programlar bellekteki `byte`'ları kullanıyor. `std::byte` türünün amacı bu bellek `byte`'larını oldukları gibi temsil etmek. Burada amaç `byte`'lardaki tutulan değerlere ne karakter kodu olarak bakmak (ki bunun için zaten `char` türleri var) ne de onları doğrudan tam sayı değerleri olarak kullanmak (bunun için de tam sayı türlerimiz var). Amaçlanan ham bellek byte'larını temsil etmek. Sayısal bir hesaplamanın söz konusu olmadığı durumlarda, `std::byte` türünün kullanılması ile derleme zamanına yönelik katı bir tür kontrolü hedefleniyor.

`std::byte` türüne ilişkin sentaks kurallarını ve bu türün kullanım olanaklarını tam olarak kavrayabilmek için öncelikle `C++17` standartları ile `enum` türleri konusunda temel sentaksa yapılan bir eklentiyi anlamamız gerekiyor: Artık baz türü `(underlying type)` belirli olan bir `enum` nesnesine küme parantezi içinde doğrudan enumarator olmayan bir tam sayı ile ilk değer verilebiliyor. Aşağıdaki koda bakalım:

```
enum Color : unsigned char {red, green, yellow};
enum class Pos {off, on, hold};

int main()
{
 Color cx{ 5 }; //geçerli
 Pos mypos{20}; //geçerli
 //...
}
```

Bu şekilde ilk değer vermenin geçerli olabilmesi için

Baz türün belirlenmiş olması gerekiyor. Geleneksel `enum`'lar söz konusu olduğunda baz tür bildirimde belirtilmeli. `enum class`'lar söz konusu olduğunda ise baz tür belirtilmese de varsayılan baz türün `int` kabul edildiğini hatırlayalım. Bildirimde ilk değer küme parantezi içinde ve doğrudan verilmeli. Küme parantezi içinde ilk değer verme `(brace initialization)` kuralları gereği daraltıcı bir tür dönüşümü `(narrowing conversion)` söz konusu olmamalı. Aşağıdaki kodu inceleyelim:

```
enum Nec : char { };
Nec i1{ 42 }; // C++17'de gecerli (daha önce geçersizdi)
Nec i2 = 42; // geçersiz
Nec i3(42); // geçersiz
Nec i4 = { 42 }; // geçersiz

enum class Weekday { mon, tue, wed, thu, fri, sat, sun };
Weekday s1{ 0 }; // C++17'de gecerli (daha önce geçersizdi)
Weekday s2 = 0; // geçersiz
Weekday s3(0); // geçersiz
Weekday s4 = {0}; // geçersiz

enum class Pos : char {on, off, hold, standby};
Pos pos1{ 0 }; //C++17'de gecerli (daha önce geçersizdi)
Pos pos2 = 0; // geçersiz
Pos pos3 = {0}; // geçersiz

enum BitFlag { bit1 = 1, bit2 = 2, bit3 = 4 };
BitFlag flag{ 0 }; // halen geçersiz (baz tür belli degil)

enum Erg : char { };
Erg i5{ 42.2 }; //geçersiz (daraltıcı dönüşüm)
```

Yukarıdaki kodda enum nesnelerine ilk değer vermede `(initialization)` yeni kurallar, geçerli ve geçersiz bildirimlerle örnekleriyle gösteriliyor. Başlangıçta en azından tuhaf olarak karşılanacak bu yeni sentaksın ana amacı `std::byte` türünün (ya da ileride oluşturulacak özel bazı türlerin) gerçekleştirimine olanak sağlamak.

`std::byte` aslında aşağıdaki gibi tanımlanmış bir `enum class` türü:

```
enum class byte : unsigned char {};
```

Tanımdan da görüldüğü gibi `std::byte` için baz tür olarak unsigned char türü seçilmiş. Bu durumda `td::byte` türünden bir nesneye ilk değer verebilmek için çok sınırlı sayıda seçenek söz konusu. Zaten `std::byte` türü ile hedeflenen bu şekilde katı bir tür denetimi sağlamak:

```
#include <cstddef>

int main()
{
 std::byte b1; //geçerli (belirlenmemiş değer)
 std::byte b2{ 0 }; //geçerli
 std::byte b3{}; //geçerli (b2 ile aynı)
 std::byte b4{ 12 }; //geçerli
 std::byte b5{ 0xFF }; //geçerli
 std::byte b6{ 0b1001'0101 }; //geçerli
 std::byte b7{ 256 }; //geçersiz (daraltıcı dönüşüm)
 std::byte b8(45); //geçersiz
 std::byte b9 = 68; //geçersiz
 std::byte b10 = { 45 }; //geçersiz
}
```

İlk değer verilmemiş otomatik ömürlü bir `std::byte` nesnesinin tanımlanması geçerli olsa da böyle bir nesnenin hayata belirlenmemiş bir değer ile `(indetermined value)` geldiğine dikkat etmemiz gerekiyor. Öğeleri `std::byte` türünden olan bir diziye de tam sayılarla ilk değer veremiyoruz:

```
#include <cstddef>

int main()
{
 std::byte a[] = { 2, 4, 5}; //geçersiz
 std::byte b[] = { std::byte{3}, std::byte{7}, std::byte{9} };
}
```

std::byte sizeof değeri

`std::byte` baz türü `unsigned char` olan bir enum türü olduğundan `sizeof` değerinin `1` olması garanti altında. `unsigned char` sistemlerin çoğunda `8` bitlik bir tam sayı türü. Ancak farklı sistemlerde `char` türlerinin bit sayısı farklı olabileceğinden bu değerin kullanılması gereken durumlarda taşınabilirlik sağlamak amacıyla standart `<limits>` başlık dosyasında tanımlanan

```
std::numeric_limits<unsigned char>::digits
```

`static constexpr` değeri kullanılmalı.

`to_integer<>` işlev şablonu

`std::byte` türünden tam sayı türlerine ve tam sayı türlerinden `std::byte` türüne otomatik tür dönüşümü `(implicit type conversion)` yok. Ancak bir `std::byte` nesnesini herhangi bir tam sayı türüne dönüştürmek için `to_integer` isimli işlev şablonunu kullanabiliyoruz:

```
#include <cstddef>
#include <iostream>

int main()
{
 std::byte b{ 0xAB };
 int ival = std::to_integer<int>(b);

 std::cout << "ival = " << ival << '\n';
}
```

`std::byte` türünden `bool` türüne de otomatik tür dönüşümü olmadığından lojik ifade beklenen yerlerde `std::byte` nesnelerini doğrudan kullanamıyoruz:

```
#include <cstddef>

int main()
{
 std::byte b{ 16 };
 //...
 //if (b) {} //gecersiz
 if (std::to_integer<bool>(b)) {}
 //...
 if (b != std::byte{ 0 }) {}
 //...
}
```

bitsel işlemler

Bir `std::byte` nesnesine yalnızca atama ya da kopyalama yapabilir ya da onu bitsel işlemlere sokabiliriz.

`std:byte` türü için bitsel işlemler, operatör yüklemesi `(operator overloading)` mekanizması ile mümkün kılınmış:

```
template<typename Integer>
constexpr byte operator<< (byte b, Integer shift) noexcept;

template<typename Integer>
constexpr byte& operator<<= (byte& b, Integer shift) noexcept;

template<typename Integer>
constexpr byte operator>> (byte b, Integer shift) noexcept;

template<typename Integer>
constexpr byte& operator>>= (byte& b, Integer shift) noexcept;

constexpr byte& operator|= (byte& left, byte right) noexcept;
constexpr byte& operator&= (byte& left, byte right) noexcept;
constexpr byte& operator^= (byte& left, byte right) noexcept;

constexpr byte operator| (byte left, byte right) noexcept;
constexpr byte operator& (byte left, byte right) noexcept;
constexpr byte operator^ (byte left, byte right) noexcept;
constexpr byte operator~ (byte b) noexcept;
```

Aşağıdaki kodu derleyip çalıştırın:

```
#include <cstddef>
#include <iostream>

int main()
{
 std::byte bb{0XAF};
 std::cout << std::hex << std::uppercase;

 std::cout << std::to_integer<int>(bb) << "\n";
 std::byte bl = (bb << 4) | (bb >> 4);
 std::cout << std::to_integer<int>(bl) << "\n";
}
```

std::byte türü ve formatlı giriş çıkış işlemleri

`std::byte` türü için standart kütüphanenin sağladığı formatlı giriş ve çıkış işlevleri `(inserter / extractor)` bulunmuyor:

```
#include <cstddef>
#include <iostream>

int main()
{
 std::byte b{ 0xA5 };

 std::cout << b; // geçersiz
 std::cin >> b; //geçersiz
 //...
}
```

Ancak bu işlevleri istersek kodlarını kendimiz dilediğimiz gibi yazabiliriz. Önce formatlı çıkış işlemi için bir işlev `(inserter)` tanımlayalım:

```
#include <cstddef>
#include <ostream>
#include <bitset>

std::ostream &operator<<(std::ostream &os, std::byte b)
{
 using btype = std::bitset<std::numeric_limits<unsigned char>::digits>;
 return os << btype(std::to_integer<unsigned char>(b));
}

#include <iostream>

int main()
{
 std::byte b1{ 0xB5 }, b2{ 0xE4 };
 std::cout << b1 << '\n' << b2 << '\n';
}
```

Şimdi de formatlı giriş işlemi için bir işlev `(extractor)` tanımlayalım:

```
#include <istream>
#include <bitset>
#include <cstddef>

std::istream &operator>>(std::istream &is, std::byte &b)
{
 using btype = std::bitset<std::numeric_limits<unsigned char>::digits>;
 btype bs;
 is >> bs;

 if (!is.fail())
  b = std::byte{ static_cast<unsigned char>(bs.to_ulong()) };

 return is;
}

#include <iostream>

int main()
{
 std::byte b;

 std::cout << "bir byte giriniz: ";
 std::cin >> b;

 std::cout << "deger : " << std::to_integer<int>(b);
}
```

`std::bitset` sınıfını kullanarak `std::byte` nesnesinin taşıdığı değeri `std::string` olarak da ifade edebiliriz:

```
#include <cstddef>
#include <string>
#include <bitset>
#include <limits>

std::byte b{ 198 };
using bitset_b = std::bitset<std::numeric_limits<unsigned char>::digits>;
auto str = bitset_b{ std::to_integer<unsigned char>(b) }.to_string();
```

Standart kütüphanenin std::byte türü için bir "sabit operatör işlevi" `(literal operator function)` tanımlaması

belki iyi bir fikir olabilirdi. Böyle bir ihtiyaç söz konusu olduğunda kendi std::byte sabitlerimizi oluşturabilmek için bir işlev tanımlayabiliriz. Aşağıdaki kodu inceleyelim:

```
#include <cstddef>
#include <iostream>

constexpr std::byte operator ""_b(unsigned long long val)
{
 return std::byte{ static_cast<unsigned char>(val) };
}

int main()
{
 constexpr std::byte b1 = 10_b;
 constexpr std::byte b2 = 48_b;
 std::cout << std::hex << std::uppercase;

 std::cout << std::to_integer<int>(b1 | b2) << "\n";
 std::cout << std::to_integer<int>(b1 & b2) << "\n";
 std::cout << std::to_integer<int>(b1 ^ 0xF_b) << "\n";
}
```