# make_unique ve hata güvenliği

Aşağıdaki deyimlerde `exp1` ve `exp2` ifadelerinin ele alınma sırası ve `f`, `g` ve `h` isimli işlevlerinin çağrılma sırası hakkında ne söylenebilir? Şimdilik `exp1` ve `exp2` ifadelerinin işlev çağrısı içermediğini düşünelim.

Örnek 1(a)

```
f(exp1, exp2);
```

Örnek 1(b)

```
f(g(exp1), h(exp2));
```

Sorunun yanıtını verebilmemiz için bazı temel kuralları çok iyi anlamış olmamız gerekiyor.

- Bir işlev çağrısının fiilen gerçekleştirilmesinden önce, işleve gönderilen tüm argüman ifadelerin tamamen değerlendirilmiş olması gerekiyor. Eğer işlev çağrısında kullanılan argüman olan ifadeler yan etki(ler) `(side effect)` içeriyor ise bu yan etkiler işlev çağrısından önce gerçekleşmiş olmak zorunda.
- Çağrılmış olan bir işlevin kodunun yürütülmesinin başlamasından sonra, çağıran işlevin içinde yer alan hiç bir ifade ele alınmaz ya da bir ifadenin ele alınması sürdürülmez. İşlev kodlarının yürütülmesi kesiksizdir. Yani bir işlev kodu yürütülürken programın akışı aynı zamanda başka bir işlev koduna girmez.
- İşlevlere gönderilen argüman olarak kullanılan ifadeler (başka kurallar tarafından bir belirleyicilik oluşturulmamışsa) herhangi bir sırada ve geçişli `(interleaved)` olarak yürütülebilir. Burada geçişli sözcüğünü şu anlamda kullanıyoruz: Programın çalışması sırasında ifadelerden biri kısmen yapılabilir. Daha sonra bir başka ifade kısmen yapılabilir. Sonra daha önce kısmen yapılan ifade yeniden ele alınabilir.

```
f(exp1, exp2);
```

Bu deyim için tek söyleyebileceğimiz hem `exp1` hem de `exp2` ifadelerinin `f` işlevi çağrılmadan yürütülmüş olacaklarıdır. Derleyici exp1 ifadesini, exp2 ifadesinden önce ya da sonra yürütebilecek bir kod oluşturabilir. Hatta derleyici `exp1` ve `exp2` ifadelerini geçişli olarak yürütecek bir kod da oluşturabilir. Örneğin üretilen kodun çalışma

zamanındaki akışı şöyle olabilir:

1. exp1 kısmen yürütüldü
2. exp2 kısmen yürütüldü
3. exp1 tamamen yürütülmüş oldu
4. exp2 tamamen yürütülmüş oldu
5. Şimdi de diğer deyime bakalım:

```
f(g(exp1), h(exp2));
```

Yukarıda açıkladığımız kurallardan aşağıdaki sonuçları çıkartabiliriz:

- `exp1` ifadesi programın akışı `g` işlevine girmeden önce yürütülmüş olmalı.
- `exp2` ifadesi programın akışı `h` işlevine girmeden önce yürütülmüş olmalı.
- `g` ve `h` işlevlerinin çalışması programın akışı `f` işlevine girmeden önce bitmiş olmalı.
- `exp1` ve `exp2` ifadeleri birbiriyle geçişli olarak yürütülebilir. Fakat bu ifadelerin birinin yürütülmesi sırasında bir işlev çağrısı gerçekleşirse çağrılan işlevin kodu kesiksiz ve geçişsiz yürütülür. Örneğin `g` işlevinin kodu çalışmakta iken `exp2` ifadesi kısmen yürütülemez. Ancak `g` ya da `h` çağrısının hangisinin daha önce gerçekleşeceği konusunda bir güvence söz konusu değildir.

Peki şimdi buradan hareketle konuyu hata güvenliğine `(exception safety)` getirelim. Bir başlık dosyasından aşağıdaki gibi bir işlev bildirimi gelmiş olsun. Bildirimdeki `T1` ve `T2`'nin sınıf türleri olduğunu düşünelim:

Örnek - 2

```
void f(T1 *, T2 *);
```

Bildirilen bu işlevin kodda herhangi bir yerde aşağıdaki gibi çağrıldığını düşünelim:

```
f(new T1, new T2);
```

Acaba yukarıdaki gibi bir çağrı hata güvenliği açısından bir soruna yol açar mı? Evet, işlev çağrısında hata güvenliğine ilişkin birden fazla sorun var:

```
new T1
```

gibi bir ifadeye teknik olarak `"new ifadesi"` deniyor. Bir `new` ifadesi karşılığında ne yapıldığını bir hatırlayalım. (Şimdilik `new` ifadesinin dizi ya da diğer biçimlerinin söz konusu olmadığını düşünüyoruz):

- Bir bellek alanı elde edilir.
- Bu bellek alanında yeni bir nesne hayata getirilir. Yani ilgili sınıfın kurucu işlevi `(constructor)` çağrılır.
- Eğer hayata gelen nesne için çağrılan kurucu işlev bir hata nesnesi gönderirse `(exception throw ederse)` nesne için elde edilen bellek alanı geri verilir.

Bu şu anlama geliyor: Yukarıdaki çağrıdaki her iki new ifadesi de aslında iki ayrı işlev çağrısının yapılmasını sağlıyor:

1. `operator new` işlevine yapılan çağrı. (Çağrılan `operator new` işlevi global olabilir ya da sınıf tarafından sağlanmış `operator new` işlevi olabilir).
2. Sınıfın kurucu işlevine yapılan çağrı.

Peki, derleyici aşağıdaki sırayla yürütülecek bir kod oluşturursa neler olabilir?

1. `T1` türünden nesne için bellek alanı elde edilir.

2. `T1` nesnesi için kurucu işlev çağrılır.
3. `T2` türünden nesne için bellek alanı elde edilir.
4. `T2` nesnesi için kurucu işlev çağrılır.

Böyle bir işlem sıralamasında sorun şu: Bir hata nesnesi `(exception)` gönderilmesi nedeniyle `3.` adım ya da `4.` adım başarısız olursa, `C++` standartları hayata getirilmiş olan `T1` nesnesi için sonlandırıcı işlevin çağrılmasını ve nesne için edinilmiş bellek alanının geri verilmesini zorunlu kılmıyor. Bu da hem kaynak sızıntısı `(resource leak)` hem de bellek sızıntısı anlamına geliyor.

Aşağıdaki gibi bir işlem sırası da söz konusu olabilir:

1. `T1` türünden nesne için bellek alanı elde edilir.
2. `T1` türünden nesne için bellek alanı elde edilir.
3. `T2` nesnesi için kurucu işlev çağrılır.
4. `T2` nesnesi için kurucu işlev çağrılır.

Böyle bir sıralamada bir değil farklı etkilere neden olabilecek iki ayrı hata güvenliği sorunu var:

Eğer bir hata nesnesi gönderilmesi nedeniyle `3.` adım başarısız olursa, `T1` nesnesi için edinilen bellek alanı geri verilir. Ancak standartlar `T2` nesnesi için edinilen bellek alanın geri verilmesi konusunda bir güvence vermiyor. Yani bu durumda bir bellek sızıntısı olacak. Eğer bir hata nesnesi gönderilmesi nedeniyle `4.` adım başarısız olursa, `T1` nesnesi için yer edinilmiş ve bu yerde `T1` nesnesi kurulmuş demektir. Fakat standartlar bu nesne için sonlandırıcı işlevin çağrılmasını ve nesne için ayrılan bellek alanının geri verilmesini zorunlu kılmıyor. Bu da `T1` nesnesi için hem kaynak sızıntısı hem de bellek sızıntısı demek.

Burada akıllara şöyle bir soru gelebilir? Neden derleyici ne yapılması gerekiyorsa bunun yapılmasını sağlayan bir kod üretmiyor? `C++` verimliliği esas alan bir dil. Derleyicinin işlem sırasını istediği gibi belirleyebilmesi daha iyi bir optimizasyon yapabilmesini ve daha verimli bir kod üretebilmesini sağlıyor. Derleyicinin hata güvenliği açısından ne gerekiyorsa o şekilde kod üretmesi, oluşturulacak kodun verimliliğini büyük ölçüde düşürürdü. Belki `unique_ptr` gibi bir akıllı gösterici sınıfı bize yardımcı olabilir, değil mi?

Şimdi `Örnek-2'` deki kodun aşağıdaki şekilde değiştirildiğini düşünelim: Yukarıdaki işlevin şu şekilde değiştirildiğini düşünelim:

**Örnek-3**

```
void f(std::unique_ptr<T1>, std::unique_ptr<T2>);
```

İşlev kaynak dosyada bir yerde aşağıdaki gibi çağrılmış olsun:

```
f(std::unique_ptr<T1>{new T1}, std::unique_ptr<T2>{new T2});
```

Acaba böyle bir çağrının `Örnek 2'` deki çağrıya göre bir avantaj sağlıyor mu? Hata güvenliği açısından bir sorun var mı?

Birçok programcı akıllı gösterici kullanmanın hata güvenliğine ilişkin tüm problemleri çözdüğünü sansa da gerçekte bu doğru değil.  Evet bir `unique_ptr` nesnesine bağlanan kaynaklar sızıntıya karşı korunmuş oluyor ama problem daha programın akışı daha `unique_ptr` nesnesinn kurucu işlevine girmeden gerçekleşiyor. Derleyicinin oluşturduğu kodda işlem sırasının şöyle olduğunu düşünelim:

1. `T1` türünden nesne için bellek alanı elde edilir.
2. `T1` nesnesi için kurucu işlev çağrılır.
3. `T2` türünden nesne için bellek alanı elde edilir.
4. `T2` nesnesi için kurucu işlev çağrılır.
5. `unique_ptr<T1>` nesnesi için kurucu işlev çağrılır.

6. `unique_ptr<T2>` nesnesi için kurucu işlev çağrılır
7. `f` işlevi çağrılır.

Yukardaki senaryoda eğer `3.` ya da `4.` adımda bir hata nesnesi gönderilirse aynı problemler ortaya çıkar. Ya da şu senaryoya bakalım:

1. `T1` türünden nesne için bellek alanı elde edilir.
2. `T2` türünden nesne için bellek alanı elde edilir.
3. `T1` nesnesi için kurucu işlev çağrılır.
4. `T2` nesnesi için kurucu işlev çağrılır.
5. `unique_ptr<T1>` nesnesi için kurucu işlev çağrılır.
6. `unique_ptr<T2>` nesnesi için kurucu işlev çağrılır
7. `f` işlevi çağrılır.

`3.` ya da `4.` adımlarda hata nesnesi gönderilirse yine aynı sorunlar çıkar, değil mi?

Burada sorun `unique_ptr`'nin kullanılması değil yanlış bir şekilde kullanılması. Şimdi daha iyi bir kullanımın nasıl olabileceğini inceleyelim:

**Örnek - 4**

Bir başlık dosyasında

```
void f(std::unique_ptr<T1>, std::unique_ptr<T2>);
// çağrının yapıldığı yer

f(make_unique<T1>(), make_unique<T2>());
```

Burada temel fikir, aynı akış içinde çağrılan işlevler kesiksiz çalışmasından faydalanmak. `new` ifadesiyle hayata getirilecek nesnemizin bellek alanını elde ederek bu bellek alanında nensnemizi oluşturacak aynı zamanda `unique_ptr` akıllı gösterici nesnesini oluşturacak bir işlev kullanmak istiyoruz. Böyle bir işlevin her tür için çalışması gerekeceğinden işlevi bir şablon biçiminde ifade etmemiz gerekiyor. İşlevi çağıran kod kurucu işleve argümanları dışarıdan `make_unique` işlevine geçmek zorunda olacağından `new` ifadesinde kullanacağımız kurucu işlev argumanlarını `make_unique` işlevine argüman olarak geçeceğiz. make_unique işlevi aldığı argümanları mükemmel gönderim `(perfect forwarding)` mekanizması ile sınıfın kurucu işlevine gönderecek. Çok gerekmesine karşın `C++11` standartlarında unutulan `make_unique` işlevi `C++14` standartlarıyla dile eklendi. `make_unique` şablonunun kodunun aşağıdaki gibi olduğunu düşünebiliriz:

```
template<typename T, typename ...Args>
std::unique_ptr<T> make_unique(Args&& ...args)
{
 return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

Bu hata güvenliği sorunlarını çözüyor. Kodumuzda yalnızca `3` işlev çağrısı var. Derleyici hangi sırayla çalışacak kod üretirse üretsin bir sorun olmayacak. İşlem sırasının aşağıdaki gibi olduğunu düşünelim:

1. `make_unique<T1>` işlevi çağrılır.
2. `make_unique<T2>` işlevi çağrılır.
3. `f` işlevi çağrılır.

`1.` adımda hata nesnesinin gönderilmesi durumunda, `make_unique` işlevinin kendisi hatalara karşı güvenli olduğu için bir sızıntı olmaz.

Peki, ikinci adımda bir hata nesnesinin gönderilmesi durumunda, birinci adımda oluşturulan `unique_ptr` nesnesi için temizlik işlemleri yapılacak mı? Evet yapılacak. Şimdi aklımıza şu soru gelebilir. Örnek 2 `'de de bir` T1

`nesnesi yaratılmıştı ve gerekli temizlik işlemleri yapılmadığı için sızıntı olmuştu. Ne fark etti? Bu kez durum aynı değil. Burada oluşturulan` unique_ptr `nesnesi geçici bir nesne` (temporary object)`. Geçici nesneler için temizlik işlemlerinin nasıl yapılacağı standartlarda açıkça belirtilmiş: Standartlar` 12.2/3`'de şöyle diyor:

Geçici nesnelerin hayatı, bu nesnelerin oluşturulmasını içeren ifadenin ele alınması bitince, sona erer. Bu kural yürütülen koddan bir hata nesnesi gönderilmesi nedeniyle çıkılması durumunda da geçerlidir.

**Temel İlkeler:**

`shared_ptr` ile yönetilecek dinamik sınıf nesnelerini `make_shared` işlevi ile, `unique_ptr` ile yönetilecek dinamik sınıf nesnelerini `make_unique` işleviyle oluşturun. `new` işlecini doğrudan kullanmaktan kaçının. Bunun yerine ham bellek alanının edinilmesini sarmalayan ve başka bir nesneye aktarılmasını sağlayan, yani bir fabrika gibi çalışan bir modeli `make_uniqe` işleviyle gerçekleştirin.

Not: Bu yazı `Herb Sutter`'ın `(exception safety - GotW102)` makalesinin serbest çevirisidir.