

[Sign up](#)[Code](#)[Pull requests](#) 0[Projects](#) 0[Actions](#)[Security](#)[Pulse](#)

Branch: master

[Find file](#)[Copy path](#)[Cplusplus\\_Ders\\_Notlari](#) / [delegating\\_constructor.md](#)

Fetching contributors...



161 lines (135 sloc) | 6.49 KB

[Raw](#)[Blame](#)[History](#)

## Delege Eden Kurucu İşlev (Delegating Constructor)

Bir sınıfın birden fazla kurucu işlevinin olması çok doğal ve çoğu zaman da gerekli. Bu durumda çoğunlukla bu kurucu işlevlerin paylaştığı ortak bir kod söz konusu oluyor. Yine tipik olarak bu ortak kodun bir kısmı sınıfın veri öğelerini ilk değer verici liste ile (`constructor initializer list`) başlatıyor. Ortak kodun farklı noktalarda yeniden yazılmasının kodun bakımını zorlaştırdığını dahası kodlama hatalarına davetiye çıkarttığını biliyorsunuz. C++11 standartlarından önce bu sorunla başa çıkmak için dil tarafından doğrudan desteklenen bir araç yoktu. C++11 ile dile eklenen delege eden kurucu işlev (`delegating constructor`) kurucu işlevlerde karşılaşılan bu tipik sorunla başa çıkmak için önemli bir destek sağlıyor.

C++11 öncesinde kurucu işlevlerin ortak bir koda sahip olması durumunda programcıların tipik başvurduğu yol, ortak kodu sınıfın bir `private` işlevinde toplamak ve kurucu işlevler içinde de bu işlevi çağırması. Standartlara bu konuda verilen SC22/WG21/N1986 No'lu öneri belgesindeki örneği inceleyelim:

```

class X {
    void commonInit();
    Y y_;
    Z z_;
public:
    X(); //1
    X(int); //2
    X(W); //3
};

X::X() : y_(42), z_(3.14)
{
    commonInit();
}

X::X(int i) : y_(i), z_(3.14)
{
    commonInit();
}

X::X(W e) : y_(53), z_( e )
{
    commonInit();
}

```

X sınıfının Y ve Z türlerinden veri ögeleri var. Sınıfın üç ayrı kurucu işlevinin olduğunu görüyorsunuz. Bu kurucu işlevlerin ortak olan kodu sınıfın `commonInit` isimli `private` işlevinde toplanmış. Kurucu işlevlerin ana bloğunun içinde `commonInit` işlevi çağrılıyor. Yine kurucu işlevler içinde, öge ilk değer verme listesi ile sınıfın `y_` ve `z_` isimli ögelerine ilk değer veriliyor. Peki bu kodlarda bizi rahatsız edecek noktalar var mı?

X sınıfının kurucu işlevlerin ortak kodunun bir parçası da ögelere ilk değer vermek. Ancak bu işi bir başka işleve delege edemiyoruz. Yalnızca kurucu işlevler veri ögelerine ilk değer verebilirler, değil mi? Örnekte yer alan `commonInit` işlevi çağrıldığında zaten ögeler hayata gelmiş olacak. Bu yapıda kurucu işlevlerin ana bloğunu boş bırakamıyoruz. Örneğin `commonInit` işlevinin kodundan bir hata nesnesi (`exception`) gönderilebilir. `commonInit` üye işlevi sınıfı kodları tarafından (yanlışlıkla) çağrılabilir. Bu işlevin sınıfı diğer işlevleri tarafından çağrılmasını engelleyen bir mekanizma yok. C++11 standartlarıyla gelen eklemeye artık bir kurucu işlev başka bir kurucu işlevin kodunu çalıştırabiliyor. Şimdi kodu yeniden düzenleyerek delege eden kurucu işlevler oluşturuyoruz:

```

class X {
    X(int, W&);
    Y y_;
    Z z_;
public:
    X(); //1
    X(int); //2
    X(W&); //3
};

X::X(int i, W &e) : y_(i), z_(e)
{
    /*ortak kod */
}

X::X() : X(42, 3.14) {}
X::X(int i) : X(i, 3.14) {}
X::X(W &w) : X(53, w) {}

```

Yukarıdaki kodu inceleyelim: Sınıfa daha önceki kodda yer almayan `private` bir kurucu işlev ekledik:

```
X::X(int i, W &e) : y_(i), z_(e)
{
    /*ortak kod */
}
```

Bu `private` işlev `y_` ve `z_` veri öğelerini üye ilk değer verme listesiyle hayata başlattığı gibi, kuruluş sürecinde yapılması gereken diğer işlemleri de ana bloğundaki kod ile gerçekleştiriyor. Yani bir önceki sürümdeki `init` işlevinin kodunun bu kurucu işlevin ana bloğu içine yerleştirildiğini düşenebilirsiniz. Diğer kurucu işlevler ise tüm işi `private` kurucu işleve yaptırıyorlar (delege ediyorlar).

SC22/WG21/N1986 No'lu öneride verilen güzel bir örnek de şöyle:

```
//fullname.h
#include <string>

class FullName {
    std::string firstName_;
    std::string middleName_;
    std::string lastName_;
public:
    FullName(const std::string &firstName, const std::string &middleName,
              const std::string &lastName);
    FullName(const std::string &firstName, const std::string &lastName);
    FullName(const FullName &name);
};

//fullname.cpp
#include "fullname.h"

using namespace std;

//delege edilen kurucu işlev
FullName::FullName(const std::string &firstName, const std::string &middleName,
                  const std::string &lastName)
: firstName_(firstName), middleName_(middleName), lastName_(lastName)
{
    // ...
}

// delege eden kopyalayan kurucu işlev
FullName::FullName(const FullName& name)
: FullName(name.firstName_, name.middleName_, name.lastName_)
{
    // ...
}

// delege eden kurucu işlev
FullName::FullName(const std::string &firstName, const std::string &lastName)
: FullName(firstName, " ", lastName)
{
    //
}
```

Yukarıdaki kodda `FullName` sınıfının hem iki parametrelili kurucu işlevi hem de kopyalayan kurucu işlevi, sınıfın üç parametrelili kurucu işlevine delege ediyorlar.

Delege eden kurucu işlev içinde, ilk değer verme listesi ile bir veri öğesine ilk değer veremiyoruz. Yani ilk değer verme listesinde bulunan tek öğe delege edilen kurucu işleve yapılan çağrı olmalı. Aşağıdaki koda bakalım:

```
class A {
    int mx;
    int my = 1;
public:
    A(int x) :mx{x}{}
    A() : A(0), my{0} {}; //geçersiz
};
```

A sınıfının kurucu işlevinin tanımı geçerli olsaydı sınıfın `my` isimli veri ögesine iki kez ilk değer verilmiş olurdu, değil mi?

Delege edilen kurucu işlev de aynı sentaksı kullanarak bir başka kurucu işleve delege edebilir. Ancak derleyici böyle bir durumda oluşacak sonsuz bir çevrimi kontrol etmekle yükümlü değil, böyle bir çevrim oluşmasından tamamen programcı sorumlu.

Eğer delege edilen bir kurucu işlev bir hata nesnesi gönderirse gönderilen hata nesnesini delege eden kurucu işlevde oluşturulan bir "işlev try bloğu" (`function try block`) ile yakalayabiliyoruz:

```
#include <iostream>

struct A {
    A()
    {
        std::cout << "A::A()\n";
        throw 1;
    }
    A(int) try : A(){
        std::cout << "A::A(int)\n";
    }
    catch (int) {
        std::cout << "hata A(int) islevinde yakalandi\n";
        throw;
    }
};

int main()
{
    try {
        A x(1);
    }
    catch (int) {
        std::cout << "hata main islevinde yakalandi\n";
    }
}
```

Yukarıdaki kodda A sınıfının varsayılan kurucu işlevinin bir hata nesnesi gönderdiğini görüyorsunuz. Sınıfın `int` parametrelili kurucu işlevi varsayılan kurucu işleve delege ediyor. Varsayılan kurucu işlevden gönderilen hata nesnesi `int` parametrelili kurucu işlevin oluşturduğu `işlev try bloğu` ile yakalanıyor.

Delege eden kurucu işlevler örneklerden de görüldüğü kodu karmaşıklıktan arındırıyor ve kodun bakımını kolaylaştırıyor.

