Cplusplus_Ders_Notlari / **private_kalitimi.md**

`Java` , `C#` gibi dillerden biraz farklı olarak `C++` dilinde 3 ayrı kalıtım `(inheritance)` biçimi var: `public` , `private` ve `protected` kalıtımları. Aslında bunlardan yalnızca `public` kalıtımı, `Nesne Yönelimli Programlama` 'daki kalıtım kavramına karşı geliyor. `public` kalıtımı ile ingilizcede " `is a"` ilişkisi denilen modeli gerçekliyoruz. `private` ve `protected` kalıtımları ise tamamen farklı amaçlarla kullanılıyorlar. Daha sonra bu konuya geri dönmek üzere önce `private` kalıtımına ilişkin kuralları bir gözden geçirelim:

`C++` 'da kalıtımın hiçbir biçiminde taban sınıfın `private` bölümüne türemiş sınıfların erişim hakkı yok. Yani taban sınıfın `private` bölümü hem taban sınıfın `(parent class)` kendi müşterilerine `(clients)` hem de taban sınıftan kalıtım yoluyla elde edilecek sınıflara `(child classes)` kapalı. `private` kalıtımında taban sınıfın `public` ve `protected` bölümleri türemiş sınıfın `private` bölümü gibi ele alınıyor. Taban sınıfın `public` ya da `protected` bölümüne türemiş sınıf müşterilerinin erişim hakkı yok. Aşağıdaki kodu inceleyelim:

```
class Base {
public:
 void pub_func();
protected:
 void pro_func();
private:
 void pri_func();
};

class Der : private Base {
public:
 void derfunc()
 {
  pub_func(); //gecerli
  pro_func(); //geçerli
  pri_func(); //geçersiz
 }
};

int main()
{
 Der myder;

 myder.pri_func();  //geçersiz
 myder.pro_func();  //geçersiz
 myder.pub_func();  //geçersiz
}
```

`Der` sınıfı `Base` sınıfından `private` kalıtımı yoluyla oluşturulmuş. `':'` atomundan sonra `private` anahtar sözcüğü kullanılmasaydı da yine kod geçerli olacak ancak `private` kalıtımı anlamına gelecekti. Yani sınıflar söz konusu olduğunda varsayılan kalıtım biçimi `private`:

```
class Base {
//
};

class Der : Base {  //private kalıtımı
//
};

Yapılar için ise varsayılan kalıtım biçimi public:

class Base {
//
};

struct Der : Base {  //public kalıtımı
//
};
```

`public` kalıtımında türemiş sınıf türünden bir nesne aynı zamanda taban sınıf türünden bir nesne kabul edildiğinden türemiş sınıftan taban sınıfa `(upcasting)` dönüşüme izin veriliyor:

```
class Base {
 //
};

struct Der : public Base {
public:

};

int main()
{
 Der myder;
 Base *base_ptr = &myder;  //geçerli
 Base &base_ref = myder;  //geçerli
}
```

Ancak `private` kalıtımında bu tür dönüşümler yalnızca taban sınıfın arkadaşları olan ya da taban sınıftan türeyen sınıflar için geçerli:

```
class Base {
 //
};

struct Der : private Base {
public:
 void func()
 {
  Der myder;
  Base *base_ptr = &myder;  //geçerli
  Base &base_ref = myder;  //geçerli
 }
};

int main()
{
 Der myder;
 Base *base_ptr = &myder;  //geçersiz
 Base &base_ref = myder;  //geçersiz
}
```

Kalıtım biçiminin `public`, `protected` ya da `private` olması taban sınıfın `private` olmayan sanal işlevlerinin türemiş sınıflar tarafından ezilmesine `(override)` engel bir durum değil:

```
class Base {
public:
 virtual void func();
 //
};

struct Der : private Base {
        void func()override;
 //
};
```

Yukarıdaki kodda `Base` sınıfından `private` kalıtımı yoluyla elde edilen `Der` sınıfı `Base` sınıfının `public` sanal işlevi olan `func` işlevini ezmiş `(override etmiş)`.

private kalıtımı neden kullanılır?

`private` kalıtımına ilişkin kuralları gözden geçirdiğimize göre artık bu kalıtım biçiminin ne işe yaradığını ya da ne

fayda sağladığını incelemeye başlayabiliriz. `public` kalıtımı ile bir taban sınıfın `(parent class)` `public` arayüzünü devralan yeni bir sınıf `(child class)` oluşturuyoruz. Böyle iki sınıf arasındaki ilişkiye ingilizcede popüler olarak `"is a" ilişkisi deniyor.

Eğer `X` sınıfı `Y` sınıfından kalıtım yoluyla elde edildi ise
Her bir `X` aynı zamanda bir `Y`'dir. Yani `Y` nesnesi gereken her yerde bir `X` nesnesi de kullanılabilir:

- Her satış görevlisi bir çalışandır
- Her aslan bir hayvandır.
- Her politikacı bir yalancıdır.

Bu ne anlama geliyor? Çalışan gereken her yerde bir satış görevlisi de kullanılabilir. Bir hayvan gereken her yerde bir aslan kullanılabilir. Yalancı gereken her yerde bir politikacı bu işi görebilir. Ancak `private` ve `protected` kalıtımları aslında bambaşka bir amaçla kullanılıyorlar. Yani artık `"is a"` ilişkisi söz konusu değil.

Bir nesnenin başka bir nesneyi onun sahibi olarak kullanmasına ingilizcede `"composition"` deniyor. `Composition`, nesne yönelimli programlamanın en önemli araçlarından biri. Sınıflar arasında composition gösteren bir ilişkiye ingilizcede popüler olarak `"has a"` ilişkisi deniyor:

Eğer her `X`'in bir `Y` türünden bir öğesi var ise, bir `X` nesnesi belirli hizmetleri kendi müşterilerine sağlamak için sahibi olduğu `Y` nesnesini kullanabilir:

Bilgisayarın ana kartı var.
Savaşçının silahları var.
Arabanın motoru var.

`C++` gibi bir dilde `composition` ilişkisini kodlamanın en basit ve en sık tercih edilen yolu bir sınıfın başka bir sınıf türünden veri öğesi ya da öğelerine sahip olması. Gelin bu yola "içerme" `(containment)` diyelim. Her arabanın bir motoru var, değil mi?

```
class Engine {
public:
 void start();
 ////
};

class Car {
 Engine its_engine;
public:
 void start()
 {
  its_engine.start();
 }
 ///
};
```

Yukarıdaki kodda `Car` sınıfının `Engine` sınıfı türünden bir öğesi var. `Car` sınıfı kendi müşterilerine hizmet veririken bu iş için `Engine` sınıfının `public` arayüzünü kullanarak `Engine` sınıfının kodlarından faydalanabilir. Eğer `Car` sınıfını `Engine` sınıfından `private` kalıtımı ile elde etsek de sonuç benzer olacak. Yani duruma `Car` sınıfının işlevselliği açısından içerme ile `private` kalıtımı arasında bir fark yok.

```
class Engine {
public:
 void start();
 ////
};

class Car : private Engine{
 Engine itsEngine;
public:
 using Engine::start;
 ///
};
```

Şimdi bu iki yapıyı, yani içerme ile `private` kalıtımını birbiriyle karşılaştıralım. Önce ortak noktalara değinelim:

1. İki yapıda da her `Car` nesnesinin içinde bir `Engine` nesnesi var ve `Car` nesnesi bu engine nesnesini kullanabiliyor.
2. İki yapıda da `Car` sınıfının müşterilerine için `Car *` türünden `Engine *` türüne dönüşüm izini verilmiyor. (Çünkü her araba aynı zamanda bir motor değildir).
3. İki yapıda da `Car` sınıfı `Engine` sınıfının `public` arayüzünü kendi arayüzüne eklemiyor.
4. İki yapıda da `Car` sınıfı `Engine` sınıfının `public` arayüzünün istediği kısım ya da kısımlarını kendi `public` arayüzüne seçerek katabilir.

Şimdi de farklılıklara bakalım:

1. Eğer bir arabanın birden fazla motoru olacak ise tercihimiz içerme olurdu. Bu durumda `private` kalıtımının kullanılması çoklu kalıtım gerektirecekti.
2. `private` kalıtımında `Car` sınıfının kendi kodlarına ve arkadaşlarına `Car *` türünden `Engine *` türüne dönüşüm izni veriliyor. Ancak "içerme" durumunda böyle bir izin söz konusu değil. 3. `private` türetmesinde `Car` sınıfı Engine sınıfının `protected` bölümüne erişebiliyor. Ancak "içerme" durumunda `Car` sınıfının `Engine` sınıfın `protected` bölümüne erişim hakkı yok.
3. İçerme durumunda `Engine` sınıfının `public` arayüzündeki bir işlevi `Car` sınıfının public arayüzüne katmak için bu işlevi çağıracak yeni bir işlev `(forwarding function)` oluşturmak gerekiyor:

```
void Car::start()
{
        itsEngine.start();
}
```

`private` kalıtımında ise bu işi bir sınıf içi `using` bildirimiyle gerçekleştirebiliyoruz:

`using Engine::start;`

5. `private` türetmesinde `Car` sınıfı `Engine` sınıfının sanal işlevlerini ezebiliyor ama içerme durumunda bu doğrudan mümkün değil. Bu dolaylı olarak şöyle gerçekleştirebilir:

```
class Engine {
public:
 virtual void maintain();
};

class Car {
 class SpecialEngine : public Engine{
  void maintain()override;
 };
 SpecialEngine m_se;
 //
};
```

Şimdi önemli soru şu: `Composition` gereken bir durumda oluşturacağımız sınıfa istediğimiz işlevselliği hem "içerme" hem de `"private kalıtımı"` ile sağlayabiliyoruz. Bu durumda neden `private` kalıtımını tercih edelim? `Composition` açısından baktığımızda "içerme", `private` kalıtımının bir alt kümesi olarak görülebilir. `Composition` 'ı gerçeklerken `private` kalıtımı bize daha fazla araç sunuyor. İçerme yerine `private` kalıtımını tercih etmemizi gerektiren nedenler şunlar olabilir:

- Kullanılacak sınıfın `protected` kısmına (özellikle de `protected` kurucu işlevlere) erişmek istiyoruz.
- Kullanılacak sınıfın sanal işlev ya da işlevlerini işlevlerini ezmek `(override)` istiyoruz (ya da buna mecburuz). Eğer arayüzünü kullanacağımız sınıf soyut `(abstract)` ise bu sınıfın tüm saf sanal `(pure virtual)` işlevlerini ezmez isek bizim oluşturduğumuz sınıf da soyut olacaktı. Sınıfımız türünden nesneler oluşturabilmek `(instantiate)` için somut bir sınıf oluşturmak zorundayız.

Eğer bu iki olanaktan faydalanma gibi bir amaç söz konusu değilse tercih edilmesi gereken "içerme" yapısı. Kalıtıma göre sınıfların birbirine bağımılığı bu yapıda daha az. Diğer taraftan `private` kalıtım tek bir öğe sayısıyla sınırlı.

`private` kalıtımı `OOP` açısından bir kalıtım değil. Kalıtımdaki amaç taban sınıf olarak alınan sınıfın kodlarını kullanmak. `A` sınıfını `B` sınıfından `private` kalıtımıyla oluşturmak `A` 'yı `B` türünden yapmıyor ve `A` 'ya `B` 'nin arayüzünü katmıyor. Bu yüzden private inheritance tasarım ile değil gerçekleştirim `(implementasyon)` ile ilgili.

Eğer içerme ile `private` kalıtım arasında tereddütte kalıyorsanız şu ilkeye bağlı kalabilirsiniz: Kullanabildiğiniz her yerde içerme yapısını kullanın yalnızca zorunlu olduğunuz durumlarda `private` kalıtımı kullanın.

`private` kalıtımın içermeye tercih edileceği bir senaryo daha var:

C++'da statik olmayan `(non static)` bir veri öğesine sahip olmayan, yani boş sınıflar `(empty class)` olabiliyor. Standart kütüphane de bazı nedenlerden boş sınıfları kullanıyor. Boş bir sınıf türünden bir sınıf nesnesi tanımlandığında derleyici belirli işlemleri yapabilecek kodları üretebilmek için bu sınıf nesnesine bellekte bir yer ayırmak zorunda. Böylesi durumlarda derleyiciler boş sınıf nesneleri için tipik olarak `1` byte'lık bir yer ayırıyorlar. Ancak boş bir sınıf nesnesi başka nesnelerle birlikte aynı bellek bloğunda yer aldığında hizalama `(alignment)` nedeniyle daha fazla bir bellek alanı kullanılabiliyor. Aşağıdaki koda bakalım:

```
#include <iostream>

class A {

};

class B {
 A a;
 int x;
};

int main()
{
 std::cout << "sizeof(int) = " << sizeof(int) << "\n";
 std::cout << "sizeof(A)   = " << sizeof(A) << "\n";
 std::cout << "sizeof(B)   = " << sizeof(B) << "\n";
}
```

Benim çalıştığım sistemde yukarıdaki programın çıktısı şu şekilde oldu :

```
sizeof(int) = 4
sizeof(A)   = 1
sizeof(B)   = 8
```

Oysa derleyiciler kalıtımla boş bir sınıftan yeni bir sınıf oluşturulduğunda, popüler olarak "boş taban sınıf optimizasyonu" olarak bilinen `(EBO - empty base optimization)` bir tekniği uygulayarak boş taban sınıf nesnesi için bir yer ayırmıyorlar:

```
#include <iostream>

class A {

};

class B : private A{
 int x;
};

int main()
{
 std::cout << "sizeof(int) = " << sizeof(int) << "\n";
 std::cout << "sizeof(A)   = " << sizeof(A) << "\n";
 std::cout << "sizeof(B)   = " << sizeof(B) << "\n";
}
```

Benim çalıştığım sistemde yukarıdaki programın çıktısı şu şekilde oldu :

```
sizeof(int) = 4
sizeof(A)   = 1
sizeof(B)   = 4
```

Bu şu anlama geliyor. Eğer sınıfınız boş bir sınıf nesnesini kullanacak ise bu nesneyi sınıfınızın veri öğesi yapmak (içerme) yerine, sınıfınızı bu nesnenin ait olduğu boş sınıf türünden `private` kalıtımı ile oluşturmak, sınıf nesneleri için ihtiyaç duyulan bellek alanını azaltabilir.