saf sanal işlevlerin tanımlanması

`C++` dilinde taban sınıfların saf sanal işlevleri `(pure virtual functions)` kalıtım yoluyla elde edilecek sınıflara bir arayüz `(interface)` sunan ancak bir kod sağlamayan operasyonları temsil ediyor. En az bir saf sanal işleve sahip olan sınıflar soyut `(abstract)` olarak nitelendiriliyor. Soyut sınıflar türünden nesneler oluşturamıyoruz. Bu sınıfları yalnızca gösterici ve referans semantiği ile kullanabiliyoruz. Bir sınıf türünden nesneler oluşturmamız için bir sınıfın somut `(concrete)` olması gerekiyor. Soyut bir sınıftan kalıtım yoluyla elde edilecek bir sınıfın somut olabilmesi için taban sınıfının tüm saf sanal işlevlerini ezmesi `(override etmesi)` gerekiyor. Eğer türemiş sınıf, taban sınıfının tek bir saf sanal işlevini bile ezmez ise kendisi de soyut bir sınıf oluyor. Bir başka deyişle, taban sınıfların saf sanal işlevleri, kalıtımla elde edilecek somut sınıfları kod temin etmeye zorlayan işlevler. Aşağıdaki koda bakalım:

```
class Abstract {
public:
 virtual void func() = 0; //saf sanal
};

class AbstractToo : public Abstract {

};

class Concrete : public AbstractToo {
public:
 void func()override;
};

int main()
{
 Abstract x; //gecersiz
 AbstractToo y; //gecersiz
 Concrete z;
}
```

`Abstract` isimli sınıf soyut, çünkü `func` isimli saf sanal bir işlev bildirmiş. `Abstract` sınıfından `public` kalıtımı yoluyla türetilen `AbstractToo` sınıfı da `Abstract` sınıfının saf sanal `func` işlevini ezmediği için soyut. `AbstractToo` sınıfından yine `public` kalıtımı ile türetilen `Concrete` sınıfının `func` işlevini ezdiğini

görüyorsunuz. `Concrete` bu durumda somut bir sınıf. `main` işlevi içinde `x` ve `y` isimli nesnelerin tanımı geçersiz. Soyut sınıflar türünden nesneler tanımlayamıyoruz. Ancak `Concrete` sınıfı türünden `z` isimli nesnenin tanımlanması geçerli.

Normal olarak saf sanal işlevleri tanımlamıyoruz. Amacımız türemiş sınıflara sadece bir arayüz sağlamak. Somut sınıflar saf sanal işlevleri ezerek yani bir kod sağlayarak söz konusu operasyonu somutlaştırmış oluyorlar. Ancak öyle durumlar var ki saf sanal bir işlevi bildirmekle kalmayıp aynı zamanda bu işlevi tanımlamamız gerekiyor. Bu yazının amacı saf sanal işlevlerin tanımlanmasını gerektiren senaryoları anlatmak.

saf sanal sonlandırıcı işlev `(pure virtual destructor)`

Tüm taban sınıfların sonlandırıcı işlevleri ya sanal ve `public` olmalı ya da eğer sanal olmayacak ise `protected` olmalı. Oluşturacağımız bir taban sınfın soyut olmasını istiyoruz ancak sınıfı soyut yapacak saf sanal bir işlev söz konusu değil. Bu durumda kullanılan tipik teknik sınıfın sonlandırıcı işlevinin saf sanal yapılması:

```
//abstractbase.h

class AbstractBase {
public:
 virtual ~AbstractBase() = 0;
};
```

`AbstractBase` sınıfından kalıtım yoluyla elde edilecek somut sınıfların sonlandırıcı işlevleri derleyicinin ekleyeceği kod ile `AbstractBase` sınıfının sonlandırıcı işlevine çağrı yapacaklar. Eğer `AbstractBase` sınıfının saf sanal sonlandırıcı işlevi tanımlanmaz ise bağlama zamanında `(link time)` hata oluşacak:

```
class ConcreteClass : public AbstractBase{
public:
 ~ConcreteClass()
 {
  //code
  //AbstractBase::~AbstractBase();
 }
};

int main()
{
 ConcreteClass x;  //bağlama zamanı hatası

}
```

Bu durumda `AbstractBase` sınıfının sonlandırıcı işlevinin bir koda sahip olmasa da tanımlanması gerekiyor:

```
//abstractbase.cpp

AbstractBase::~AbstractBase()
{

}
```

Bir taban sınıfın (saf olmayan) sanal işlevleri kalıtımla elde edilecek sınıflara hem bir arayüz hem de varsayılan bir kod sağlar. Kalıtımla elde edilecek sınıf, kendi tercihine göre taban sınıfın sanal işlevinin kodunu tercih edebileceği gibi, bu işlevi ezerek `(override)` kendi kodunu da sağlayabilir. Sık yapılan hatalardan biri, türemiş sınıfın böyle bir işlevi ezmeyi ihmal etmesi ve istemeden varsayılan koda razı olması. Oysa taban sınıfın bir saf sanal işlevinin somut olması gereken sınıflar tarafından ezilmesi zorunlu. Taban sınıf varsayılan kod sağlayacağı bir işlevi sanal yapmak yerine saf sanal yaparak varsayılan kod ile tanımlayabilir. Böylece somut olacak türemiş sınıflar bu işlevi ezmek zorunda kalacaklar. Yani türemiş sınıfların varsayılan kodu istemeleri durumunda taban sınıfın saf sanal işlevini

fiilen çağırmaları gerekecek.

```
//fighter.h
class Fighter {
public:
 virtual void sleep();
 virtual void move();
 virtual void attack();
        //
};

//ninjafigter.h
class NinjaFighter : public Fighter {
public:
 void sleep() override;
 virtual void move()override;
};
```

Yukarıdaki kodda `Fighter` sınıfının `sleep`, `move` ve `attack` işlevleri sanal. Bu işlevler `Fighter` sınıfından kalıtım yoluyla elde edilecek somut sınıflara hem arayüz hem de varsayılan bir gerçekleştirim `(implementation)` sunuyor. Fighter sınıfından kalıtım yoluyla elde edilen `NinjaFighter` sınıfı sleep ve move işlevlerini ezerken attack isimli sanal işlevi ezmemiş. Bu durumda bir `Fighter` referansı ya da göstericisi ile attack işlevi çağrıldığında nesnenin dinamik türü `NinjaFighter` olsa da `Fighter` sınıfının kodu çalışacak. Şimdi de aşağıdaki koda bakalım:

```
//fighter.h
class Fighter {
protected:
 virtual void sleep() = 0;
 virtual void move() = 0;
 virtual void attack() = 0;
};

//fighter.cpp
void Fighter::sleep()
{
 //varsayılan kod
}

void Fighter::move()
{
 //varsayılan kod
}

void Fighter::attack()
{
 //varsayılan kod
}


//ninjafighter.h
class NinjaFighter : public Fighter {
public:
 void sleep() override;
 virtual void move()override;
 virtual void attack()override;
};

//ninjafighter.cpp
void NinjaFighter::sleep()
{
 //yeni kod
}

void NinjaFighter::move()
{
 //yeni kod
}

void NinjaFighter::attack()
{
 Fighter::attack();
}
```

Bu kez `Fighter` sınıfının işlevleri saf sanal yapılarak sınıfın protected bölümüne koyuldu. Bu işlevler muhtemelen "Sanal Olmayan Arayüz" `(Non Virtual Interface)` örüntüsüyle taban sınıfın sanal olmayan üye işlevleri tarafından çağrılacaklar. `NinjaFighter` sınının somut olabilmesi için yani bu sınıf türünden nesnelerin tanımlanabilmesi için `NinjaFighter` sınıfının `Fighter` sınıfının tüm saf sanal işlevlerini ezdiğini görüyorsunuz. `NinjaFighter` sınıfı `sleep` ve `move` işlevleri için kendi kodunu sağlarken attack işlevi için taban sınıfı olan `Fighter` sınıfının `attack` işlevini çağırıyor. Böylece varsayılan kodu kabullenmiş oluyor. Sınıfların üye işlevleri içinde `::` çözünürlük operatörü `(scope resolution)` ile yapılan çağrıların çalışma zamanı çok biçimliliğine tabi tutulmadığını anımsayın. Bu yüzden

```
Fighter::attack();
```

özyinelemeli bir çağrı değil.

Taban sınıf, kendisinden kalıtım yoluyla elde edilecek türemiş sınıflara bir operasyon için bir arayüz ve bu arayüze ilişkin türemiş sınıfların kullanacağı kodun bir kısmını vermek istiyor olabilir. Türemiş somut sınıfları kısmi bir kod kullanımına zorlamak için bir taban sınıfın ilgili işlevi saf sanal yapılabilir. Buradaki fikir, türemiş sınıfları, taban sınıfın saf sanal işlevini hem ezmeye zorlamak hem de bu işlevin koduna ilave olarak `(augmentation)` kendi kodlarını kullanmalarını sağlamak. Aşağıdaki koda bakalım:

```
//base.h
class Base {
public:
 virtual void vfunc() = 0;
 //
protected:
 void vfunc_impl();
 //
};

//base.cpp
void Base::vfunc()
{
 vfunc_impl();
}

void Base::vfunc_impl()
{
 //varsayılan kod
}
```

Yukarıdaki kodda `Base` sınıfının `vfunc` işlevi saf sanal yapılarak `Base` sınıfından kalıtım yoluyla elde edilecek somut sınıflar bu işlevi ezmeye zorlanmış. `vfunc` işlevinin `base.cpp` dosyasında tanımlandığını ve bu işlevin varsayılan kodu temsil eden `protected` olan `vfunc_impl` işlevini çağırdığını görüyorsunuz. Bu işlevi ezecek somut sınıflar taban sınıfın `vfunc` işlevini çağırabilecekleri ve buna kod ekleyebilecekleri gibi kendi kodlarının herhangi bir yerinde doğrudan taban sınıfın `vfunc_impl` işlevini de çağırabilirler:

```
//der.h
class Der : public Base {
public:
 void vfunc()override;
};

//der.cpp

void Der::vfunc()
{
 //Base::vfunc();
 vfunc_impl();
 //Der sınıfının ilave kodu
}
```