Nesne Yönelimli Programlamaya Giriş

Bu hafta

- Soyut Sınıflar ve Arayüzler (Abstract Classes & Interfaces)

Saf Sanal Fonksiyonlar (Pure Virtual Functions)

1. Saf Sanal Fonksiyon Nedir?

Saf sanal fonksiyon (Pure Virtual Function), bir sınıf içinde yalnızca tanımı yapılmış ancak herhangi bir gövdesi (implementation) olmayan fonksiyondur. Bu tür fonksiyonlar, türetilmiş sınıflarda yeniden tanımlanmalıdır.

Tanım

Saf sanal fonksiyonlar, C++ dilinde bir sınıfı soyut sınıf haline getirir. Gövdesiz olarak tanımlanır ve `= 0` ile işaretlenir.

Özellikler

- Sadece bir prototip içerir, gövdesi yoktur.
- `= 0` ile belirtilir.
- Bir sınıf en az bir saf sanal fonksiyon içeriyorsa, bu sınıf soyut sınıf (abstract class) olarak kabul edilir.
- Saf sanal fonksiyonlar, türetilmiş sınıflarda **override edilmek zorundadır**.

Soyut Sınıflar ve Arayüzler (Abstract Classes & Interfaces)

1. Soyut Sınıflar (Abstract Classes)

Tanım

Soyut sınıflar, en az bir **saf sanal fonksiyon** (pure virtual function) içeren sınıflardır. Bu sınıflar, doğrudan bir nesne olarak oluşturulamaz ve türetilmiş sınıflar tarafından genişletilir. Soyut sınıflar, türetilmiş sınıflar için bir şablon görevi görür ve ortak davranışları zorunlu hale getirir.

Özellikler

- Bir soyut sınıf en az bir saf sanal fonksiyon içerir.
- Saf sanal fonksiyonlar `= 0` ile tanımlanır.
- Soyut sınıflar, genellikle türetilmiş sınıflar için bir temel sınıf olarak kullanılır.
- Türetilmiş sınıflar, soyut sınıftaki saf sanal fonksiyonları **uygulamak (override etmek)** zorundadır.
- Soyut sınıflar, hem saf sanal fonksiyonlar hem de normal fonksiyonlar içerebilir. Bu, türetilmiş sınıflara bazı ortak işlevler sağlar.

Neden Kullanılır?

- Ortak bir davranışı farklı sınıflara zorunlu kılmak için kullanılır.
- Türetilmiş sınıfların belirli bir yapıyı veya işlevselliği takip etmesini sağlar.
- Polimorfizmi destekler.

Örnek

```cpp
#include <iostream>
using namespace std;

// Soyut sınıf
class Sekil {
public:
    virtual void ciz() const = 0; // Saf sanal fonksiyon

    virtual ~Sekil() {} // Sanal yıkıcı

    void bilgi() const {
        cout << "Bu bir şekildir." << endl;
    }
};
```

```cpp
class Daire : public Sekil {
public:
    void ciz() const override {
        cout << "Bir daire çiziliyor." << endl;
    }
};

class Kare : public Sekil {
public:
    void ciz() const override {
        cout << "Bir kare çiziliyor." << endl;
    }
};
```

```
int main() {
    Sekil* sekiller[] = { new Daire(), new Kare() };

// for (int i = 0; i < sizeof(sekiller) / sizeof(sekiller\[0]); i++) {
// sekiller\[i]->ciz();
// sekiller\[i]->bilgi();
// delete sekiller\[i]; }


    for (Sekil* sekil : sekiller) {
        sekil->ciz();
        sekil->bilgi();
        delete sekil;
    }

    return 0;
}
```

Çıktı:

```
Bir daire çiziliyor.
Bu bir şekildir.
Bir kare çiziliyor.
Bu bir şekildir.
```

2. Arayüzler (Interfaces)

Tanım

C++ dilinde, saf sanal fonksiyonlardan oluşan ve genellikle başka bir işlevsellik içermeyen soyut sınıflar "arayüz" (interface) olarak kabul edilir. Arayüzler, bir sınıfın belirli bir davranışı gerçekleştirebilmesi için gereken bir şablon sağlar.

Özellikler

- Arayüzler yalnızca saf sanal fonksiyonlar içerir.
- Çoklu kalıtımı desteklemek için kullanılabilir.
- Diğer sınıflara yalnızca bir şablon veya protokol sunar.
- Bir sınıf birden fazla arayüzü uygulayabilir.
- Arayüzler genellikle sınıflar arası bağımlılığı azaltmak ve esneklik sağlamak için kullanılır.

Neden Kullanılır?

- Farklı sınıfların aynı davranışı desteklemesini sağlamak.
- Kodun genişletilebilirliğini artırmak.
- Çoklu kalıtım durumlarında sınıf yapısını netleştirmek.

Örnek

```cpp
#include <iostream>
using namespace std;

// Arayüz olarak kullanılan saf soyut sınıf
class Cizilebilir {
public:
    virtual void ciz() const = 0; // Saf sanal fonksiyon
    virtual ~Cizilebilir() {} // Sanal yıkıcı
};

class Dikdortgen : public Cizilebilir {
public:
    void ciz() const override {
        cout << "Bir dikdörtgen çiziliyor." << endl;
    }
};

class Ucgen : public Cizilebilir {
public:
    void ciz() const override {
        cout << "Bir üçgen çiziliyor." << endl;
    }
};
```

```
int main() {
    Cizilebilir* sekiller[] = { new Dikdortgen(), new Ucgen() };

    for (Cizilebilir* sekil : sekiller) {
        sekil->ciz();
        delete sekil;
    }

    return 0;
}
```

Çıktı:

```
Bir dikdörtgen çiziliyor.
Bir üçgen çiziliyor.
```

3. Soyut Sınıflar ve Arayüzler Arasındaki Farklar

Özellik	**Soyut Sınıf**	**Arayüz**
Tanım	En az bir saf sanal fonksiyon içeren sınıf.	Tümü saf sanal fonksiyonlardan oluşan sınıf.
Üyeler	Hem normal hem saf sanal fonksiyonlar olabilir.	Sadece saf sanal fonksiyonlar içerir.
Kalıtım	Tekli veya çoklu kalıtım yapılabilir.	Genellikle çoklu kalıtımda kullanılır.
Kullanım Amacı	Temel sınıf olarak işlev görür.	Şablon veya protokol sağlar.

4. Kullanım Alanları

Soyut Sınıflar

- Ortak özellikler ve davranışlar için temel sınıf sağlar.
- Bazı işlemleri türetilmiş sınıflarda zorunlu hale getirir.

Arayüzler

- Çoklu kalıtımı destekler.
- Sınıfların belirli bir davranışa sahip olmasını garanti eder.

5. Karşılaştırmalı Kod Örneği

Soyut Sınıf Örneği

```
class Arac {
public:
    virtual void hareket() const = 0; // Saf sanal fonksiyon
    virtual ~Arac() {}

    void bilgi() const {
        cout << "Bu bir araçtır." << endl;
    }
};

class Araba : public Arac {
public:
    void hareket() const override {
        cout << "Araba hareket ediyor." << endl;
    }
};
```

Arayüz Örneği

```
class Surulebilir {
public:
    virtual void sur() const = 0; // Saf sanal fonksiyon
    virtual ~Surulebilir() {}
};

class Bisiklet : public Surulebilir {
public:
    void sur() const override {
        cout << "Bisiklet sürülüyor." << endl;
    }
};
```

Senaryo:

Bir robotik sistemde farklı motor türleri bulunmaktadır (örneğin, **DC motor**, **Step motor**, ve **Servo motor**). Tüm motorların ortak davranışları vardır, ancak bu davranışlar her motor türü için farklı şekillerde uygulanır. Örneğin, tüm motorlar:

1. **Çalıştırma (start)**,
2. **Durdurma (stop)** ve
3. **Hız ayarı (setSpeed)** işlevlerine sahiptir.

Bu davranışları modellemek için bir soyut sınıf (`Motor`) kullanılır ve her motor türü türetilmiş sınıflar olarak tanımlanır.

```cpp
#include <iostream>
#include <string>
using namespace std;

// Soyut sınıf (Abstract class)
class Motor {
public:
    virtual void start() = 0;    // Saf sanal fonksiyon
    virtual void stop() = 0;     // Saf sanal fonksiyon
    virtual void setSpeed(int speed) = 0; // Saf sanal fonksiyon

    virtual ~Motor() {}
};
```

```cpp
// Türetilmiş sınıf: DC Motor
class DCMotor : public Motor {
public:
    void start() override {
        cout << "DC Motor çalıştırılıyor." << endl;
    }

    void stop() override {
        cout << "DC Motor durduruluyor." << endl;
    }

    void setSpeed(int speed) override {
        cout << "DC Motor hızı " << speed << " RPM olarak ayarlanıyor." << endl;
    }
};
```

```cpp
// Türetilmiş sınıf: Step Motor
class StepMotor : public Motor {
public:
    void start() override {
        cout << "Step Motor çalıştırılıyor." << endl;
    }

    void stop() override {
        cout << "Step Motor durduruluyor." << endl;
    }

    void setSpeed(int speed) override {
        cout << "Step Motor hızı " << speed << " adım/saniye olarak ayarlanıyor." << endl;
    }
};
```

```cpp
// Türetilmiş sınıf: Servo Motor
class ServoMotor : public Motor {
public:
    void start() override {
        cout << "Servo Motor çalıştırılıyor." << endl;
    }

    void stop() override {
        cout << "Servo Motor durduruluyor." << endl;
    }

    void setSpeed(int speed) override {
        cout << "Servo Motor pozisyonu " << speed << " derece olarak ayarlanıyor." << endl;
    }
};
```

```cpp
int main() {
    // Motor nesnelerini soyut sınıf işaretçileri ile kontrol etme
    Motor* motor1 = new DCMotor();
    Motor* motor2 = new StepMotor();
    Motor* motor3 = new ServoMotor();

    // Polimorfizm ile motorları çalıştırma
    motor1->start();
    motor1->setSpeed(1500);
    motor1->stop();

    motor2->start();
    motor2->setSpeed(200);
    motor2->stop();

    motor3->start();
    motor3->setSpeed(90);
    motor3->stop();

    // Bellek temizliği
    delete motor1;
    delete motor2;
    delete motor3;

    return 0;
}
```

Senaryo:

Robotik bir sistemde farklı sensörler bulunmaktadır. Bu sensörlerin:

1. **Veri okumak (readData)**,
2. **Kalibrasyon yapmak (calibrate)** gibi ortak işlevleri vardır.

Her sensör bu işlevleri farklı şekilde uygular. Soyut bir sınıf (`Sensor`) kullanılarak tüm sensörler (`basinc` , `sicaklik` , `ultrasonik`) için bir şablon tanımlayın ve her sensör bu sınıftan türetilsin.

```cpp
#include <iostream>
#include <string>
using namespace std;

// Soyut sınıf (Abstract class)
class Sensor {
public:
    // Saf sanal fonksiyonlar
    virtual void readData() = 0;
    virtual void calibrate() = 0;

    virtual ~Sensor() {}
};

// Türetilmiş sınıf: Sıcaklık Sensörü
class TemperatureSensor : public Sensor {
public:
    void readData() override {
        cout << "Sıcaklık Sensörü: 25°C okundu." << endl;
    }

    void calibrate() override {
        cout << "Sıcaklık Sensörü kalibre edildi." << endl;
    }
};
```

```cpp
// Türetilmiş sınıf: Basınç Sensörü
class PressureSensor : public Sensor {
public:
    void readData() override {
        cout << "Basınç Sensörü: 1013 hPa okundu." << endl;
    }

    void calibrate() override {
        cout << "Basınç Sensörü kalibre edildi." << endl;
    }
};

// Türetilmiş sınıf: Ultrasonik Sensör
class UltrasonicSensor : public Sensor {
public:
    void readData() override {
        cout << "Ultrasonik Sensör: Mesafe 120 cm ölçüldü." << endl;
    }

    void calibrate() override {
        cout << "Ultrasonik Sensör kalibre edildi." << endl;
    }
};
```

```cpp
int main() {
    // Sensör nesnelerini soyut sınıf işaretçileri ile kontrol etme
    Sensor* sensors[] = {
        new TemperatureSensor(),
        new PressureSensor(),
        new UltrasonicSensor()
    };

// for (int i = 0; i < sizeof(sensors) / sizeof(sensors[0]); i++) {
// sensors[i]->readData();
// sensors[i]->calibrate();
// delete sensors[i]; }

    // Sensörlerin verilerini okuma ve kalibrasyon yapma
    for (Sensor* sensor : sensors) {
        sensor->readData();
        sensor->calibrate();
        cout << "----------------------" << endl;
        delete sensor; // Bellek temizliği
    }

    return 0;
}
```

Mesajlaşma (Message Passing)

Mesajlaşma (Message Passing) Nedir?

Mesajlaşma, Nesne Yönelimli Programlama'da (OOP) nesnelerin birbiriyle iletişim kurmasını sağlayan temel bir kavramdır. C++ dilinde mesajlaşma, bir nesnenin başka bir nesne veya örneğin metodlarını (fonksiyonlarını) çağırması ile gerçekleştirilir. Bu mekanizma, kapsülleme (encapsulation) ve soyutlamayı (abstraction) destekleyerek programların modüler ve bakımı kolay hale gelmesini sağlar.

C++'ta Mesajlaşmanın Temel Özellikleri

a) Nesne Etkileşimi (Object Interaction)

Nesneler, sınıflarında tanımlanan üye fonksiyonları çağırarak mesaj alır ve gönderir.

```cpp
#include<iostream>
using namespace std;

class Araba {
public:
    void calistir() {
        cout << "Araba çalıştırılıyor." << endl;
    }
};

int main() {
    Araba araba1;
    araba1.calistir(); // araba1 nesnesi mesaj gönderiyor
    return 0;
}
```

Çıktı:

```
Araba çalıştırılıyor.
```

Çizim: Nesne Etkileşimi

b) Kapsülleme (Encapsulation)

Bir nesnenin iç durumu sadece public metodları aracılığıyla erişilir ve değiştirilir, bu da veri güvenliğini sağlar.

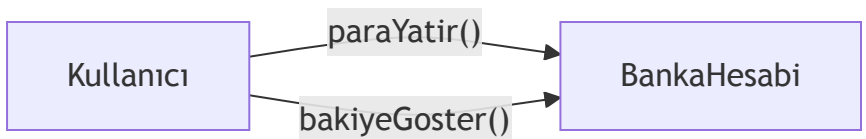
```cpp
#include<iostream>
using namespace std;

class BankaHesabi {
private:
    double bakiye;

public:
    BankaHesabi(double baslangicBakiye) : bakiye(baslangicBakiye) {}

    void paraYatir(double miktar) {
        if (miktar > 0) {
            bakiye += miktar;
            cout << miktar << " TL yatırıldı. Yeni bakiye: " << bakiye << " TL" << endl;
        }
    }

    void bakiyeGoster() {
        cout << "Mevcut bakiye: " << bakiye << " TL" << endl;
    }
};
```

```
int main() {
    BankaHesabi hesap(1000);
    hesap.bakiyeGoster();
    hesap.paraYatir(500);
    return 0;
}
```

Çıktı:

```
Mevcut bakiye: 1000 TL
500 TL yatırıldı. Yeni bakiye: 1500 TL
```

Çizim: Kapsülleme



c) Polimorfizm ve Override (Polymorphism and Overriding)

Türetilmiş sınıfların nesneleri, aynı mesaja farklı yanıtlar verebilir.

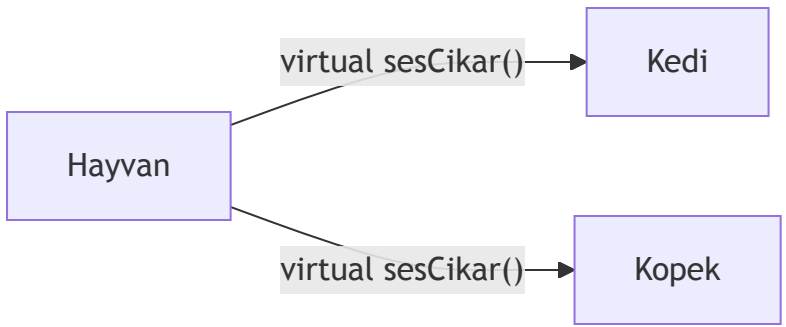
```cpp
#include<iostream>
using namespace std;

class Hayvan {
public:
    virtual void sesCikar() {
        cout << "Bir hayvan ses çıkarıyor." << endl;
    }
};

class Kedi : public Hayvan {
public:
    void sesCikar() override {
        cout << "Kedi miyavlıyor." << endl;
    }
};

class Kopek : public Hayvan {
public:
    void sesCikar() override {
        cout << "Köpek havlıyor." << endl;
```

```cpp
int main() {
    Hayvan* h1 = new Kedi();
    Hayvan* h2 = new Kopek();

    h1->sesCikar(); // Kedi yanıt veriyor
    h2->sesCikar(); // Köpek yanıt veriyor

    delete h1;
    delete h2;
    return 0;
}
```

Çıktı:

```
Kedi miyavlıyor.
Köpek havlıyor.
```

Çizim: Polimorfizm



d) Parametrelerin Mesaj Olarak Kullanımı

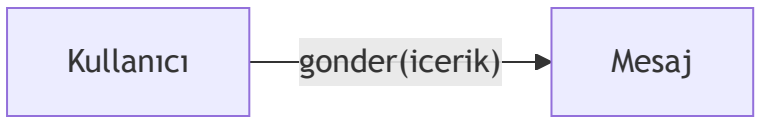
Veriler, üye fonksiyonlara parametre olarak iletilir ve bu parametreler "mesaj" içeriği gibi davranır.

```cpp
#include<iostream>
using namespace std;

class Mesaj {
public:
    void gonder(string icerik) {
        cout << "Mesaj gönderiliyor: " << icerik << endl;
    }
};

int main() {
    Mesaj mesaj;
    mesaj.gonder("Merhaba Dünya!"); // "Merhaba Dünya!" mesaj içeriği
    return 0;
}
```

Çıktı:

```
Mesaj gönderiliyor: Merhaba Dünya!
```

Çizim: Parametrelerin Mesajlaşmada Kullanımı



3. Mesajlaşmanın Avantajları

- **Modülerlik:** Nesneler birbiriyle kontrollü bir şekilde iletişim kurar.
- **Soyutlama ve Kapsülleme:** Nesneler yalnızca belirlenen metotlarla erişilir.
- **Bakım Kolaylığı:** Fonksiyonlar arası mesajlaşma sayesinde kod düzenli kalır.