Nesne Yönelimli Programlama

C++ Fonksiyonlar

- Fonksiyonlar, bir programın içerisinde belirli bir görevi yerine getiren ve tekrar kullanılabilen kod bloklarıdır.
- **Kodun yeniden kullanılabilirliği:** Aynı işlemi farklı yerlerde tekrar tekrar yazmak yerine bir kez fonksiyon olarak tanımlayıp her yerden çağırabiliriz.
- **Kodun okunabilirliği:** Büyük programları daha küçük, daha yönetilebilir parçalara ayırır.
- **Hata ayıklamanın kolaylaşması:** Hata genellikle bir fonksiyonda meydana gelir ve bu sayede hata ayıklama daha kolay hale gelir.
- **Modüler programlama:** Programı bağımsız fonksiyonlara bölerek daha esnek hale getirir.
- C++'da bir fonksiyon, belirli bir ad, parametreler ve bir geri dönüş türü ile tanımlanır.
- Genel yapı:

```
1   geriDonusTuru fonksiyonAdi(parametreListesi) {
2         // Fonksiyon gövdesi
```

Fonksiyon Türleri:

- **Geri Dönüş Tipine Göre:**
  - **void Fonksiyonlar:** Hiçbir değer döndürmezler. Genellikle ekrana çıktı vermek veya bir işlemi gerçekleştirmek için kullanılırlar.
  - **Değer Döndüren Fonksiyonlar:** Belirli bir veri tipinde değer döndürürler. Örneğin, bir toplama işlemi yapan fonksiyon, toplam değeri döndürebilir.
- **Parametre Sayısına Göre:**
  - **Parametre Almayan Fonksiyonlar:** Hiçbir parametre almazlar.
  - **Parametre Alan Fonksiyonlar:** Fonksiyonun çalışması için gerekli olan değerleri dışarıdan alır.
  - **Değişken Sayıda Parametre Alan Fonksiyonlar:** Parametre sayısı önceden belirlenmemiş fonksiyonlardır.

- **Çağrılma Şekline Göre:**
  - **Normal Fonksiyonlar:** Standart şekilde çağrılan fonksiyonlardır.
  - **İnline Fonksiyonlar:** Derleyici tarafından çağrı yerine fonksiyon gövdesi yerleştirilen fonksiyonlardır.
  - **Recursive Fonksiyonlar:** Kendini çağıran fonksiyonlardır.
- **Görevi Göre:**
  - **Matematiksel İşlemler:** Toplama, çıkarma, çarpma gibi matematiksel işlemler yapan fonksiyonlar.
  - **Karar Verme:** Eğer-değilse gibi karar verme mekanizmalarını içeren fonksiyonlar.
  - **Döngüler:** Belirli bir koşul sağlandığı sürece tekrarlanan işlemleri yapan fonksiyonlar.
  - **Veri İşleme:** Verileri sıralama, arama gibi işlemleri yapan fonksiyonlar.
  - **Giriş/Çıkış:** Ekrandan veri alıp ekrana yazdırma işlemlerini yapan fonksiyonlar.

C++ Fonksiyon Parametreleri

- Fonksiyonlar, dışarıdan veri alabilen parametrelerle çalışabilir. Parametreler, fonksiyon çağrılırken girilen argümanlarla değiştirilir.

```
void fonksiyonAdi(int parametre1, double parametre2) {
    // Kodlar burada çalışır
}
```

Parametreler ve Argümanlar

- **Parametreler**: Fonksiyon tanımında yer alan değişkenlerdir.
- **Argümanlar**: Fonksiyon çağrıldığında kullanılan gerçek veriler ya da değişkenlerdir.

```
void topla(int a, int b) {
    cout << a + b;
}

int main() {
    topla(5, 10);  // 5 ve 10 argümanları gönderiliyor
}
```

Varsayılan Parametre

- Fonksiyon parametrelerine varsayılan değerler atanabilir. Eğer bir argüman sağlanmazsa, varsayılan değer kullanılır.

```
void topla(int a, int b = 5) {
    cout << a + b;
}

int main() {
    topla(10);  // 10 ve varsayılan 5 ile çağrılır
    topla(10, 20);  // 10 ve 20 ile çağrılır
}
```

Birden Fazla Parametre (Multiple Parameters)

- C++ fonksiyonları, birden fazla parametre alabilir. Parametreler virgül ile ayrılır.

```
void carp(int a, int b, int c) {
    cout << a * b * c;
}

int main() {
    carp(2, 3, 4);  // Sonuç: 24
}
```

Geri Dönüş Değerleri

- Fonksiyonlar, belirli bir işlemi gerçekleştirdikten sonra bir değer döndürebilir.

```
int topla(int a, int b) {
    return a + b;
}

int main() {
    int sonuc = topla(10, 20);  // Sonuç: 30
}
```

Referans ile Parametre Geçirme (Pass By Reference)

- **Pass By Reference**, C++'da bir fonksiyona parametre olarak değişkenin kendisini (bellekteki adresini) göndermeyi sağlayan bir yöntemdir. Bu sayede, fonksiyon içinde yapılan değişiklikler doğrudan orijinal değişkeni etkiler.
- Değişkenlerin referansları kullanılarak, fonksiyon içinde yapılan değişiklikler ana programdaki değişkenlere de yansır.

Temel Özellikler:

- **Referans Operatörü (&)** kullanılarak gerçekleştirilir.
- Fonksiyona büyük veri yapıları gönderirken bellek kullanımını azaltır.
- Fonksiyon içindeki değişiklikler, çağrıldığı yerdeki değişkeni etkiler.

Avantajları:

- Daha hızlı ve verimli bellek kullanımı.
- Fonksiyonlar arasında büyük veri yapılarının kopyalanmasını önler.
- Orijinal değişkenin değerini fonksiyon içinde değiştirme imkanı sağlar.

Dezavantajları:

- Fonksiyon içinde yapılan değişiklikler, orijinal değişkeni etkileyebilir.
- Yan etkiler (side effects) oluşturabilir, bu da hata yapma riskini artırır.

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

// Fonksiyon Tanımı: sayiyi degistir fonksiyonu, sayi'nin referansını alır
void degistir(int &a) {
    a = 20; // Referans ile geçirilen değişkenin değeri değiştirilir
}

int main() {
    int sayi = 10; // Değişken Tanımlama
    cout << "Fonksiyon Çağrılmadan Önce sayi = " << sayi << endl;

    degistir(&sayi); // Fonksiyon Çağrısı

    cout << "Fonksiyon Çağrıldıktan Sonra sayi = " << sayi << endl; // Değerin Güncellendiğini Gösterir
    return 0;
}
```

Dizilerin Parametre Olarak Geçirilmesi

- Diziler fonksiyonlara referans olarak geçirilir. Bu nedenle dizi elemanlarındaki değişiklikler, orijinal diziye yansır.

```
void yazdir(int arr[], int boyut) {
    for (int i = 0; i < boyut; i++) {
        cout << arr[i] << " ";
    }
}

int main() {
    int sayilar[] = {1, 2, 3, 4, 5};
    yazdir(sayilar, 5);
}
```

Fonksiyon Prototipleri

Fonksiyon prototipleri, C++ programlama dilinde fonksiyonların nasıl kullanılacağını önceden tanımlamak için kullanılan bildirgelerdir. Bu prototipler, bir fonksiyonun adı, parametreleri ve dönüş tipi hakkında bilgi sağlar. Fonksiyon prototipleri, fonksiyonların tanımlanmasından önce çağrılmasına izin verir ve programın derleyici tarafından daha iyi anlaşılmasını sağlar.

Fonksiyon Prototipleri Nedir?

Fonksiyon prototipi, bir fonksiyonun tanımını içermeyen bir bildirimdir. Prototip, derleyiciye fonksiyonun ne zaman ve nasıl çağrılacağını bildirir. Prototip, aşağıdaki bilgileri içerir:

1. **Dönüş Tipi:** Fonksiyonun geri döndüreceği veri türü.
2. **Fonksiyon Adı:** Fonksiyonun adı, hangi işlemi gerçekleştireceğini tanımlar.
3. **Parametreler:** Fonksiyona geçilecek değişkenlerin türleri ve isimleri. Parametreler isteğe bağlıdır; bazı fonksiyonlar parametre almaz.

Gerçek Hayat Örneği

- **Ürün İndirimi Hesaplama**: Bir ürünün fiyatına göre indirim hesaplayan fonksiyon.

```
double indirimHesapla(double fiyat, double indirimYuzdesi = 0.10) {
    return fiyat - (fiyat * indirimYuzdesi);
}

int main() {
    double fiyat = 100.0;
    cout << "İndirimli Fiyat: " << indirimHesapla(fiyat);
}
```

C++ Fonksiyon Aşırı Yükleme (Function Overloading)

C++ programlama dilinde, **fonksiyon aşırı yükleme** (function overloading), aynı ada sahip birden fazla fonksiyon tanımlayabilme özelliğidir. Bu fonksiyonlar farklı parametre listelerine sahip olmalıdır. Yani, aynı işi yapan ancak farklı veri tiplerinde veya farklı sayıda parametre alan fonksiyonları aynı isimle tanımlayabiliriz.

Neden Fonksiyon Aşırı Yükleme Kullanılır?

- **Kodun Okunabilirliği:** Aynı işlemi farklı veri tiplerinde yapan fonksiyonlar için aynı isim kullanarak kodun okunabilirliğini artırır.
- **Esneklik:** Aynı işlemi farklı durumlarda farklı parametrelerle gerçekleştirebilme imkanı sunar.
- **Kullanım Kolaylığı:** Programcı, parametrelerin tipine göre hangi fonksiyonun çağrılacağını düşünmek zorunda kalmaz.

- Aynı isimle birden fazla fonksiyon tanımlanabilir, ancak parametre sayısı ya da türü farklı olmalıdır.

```
int topla(int a, int b) {
    return a + b;
}

double topla(double a, double b) {
    return a + b;
}

int main() {
    cout << topla(5, 10);        // int versiyonu çağrılır
    cout << topla(5.5, 2.3);     // double versiyonu çağrılır
}
```

Önemli Notlar:

- **Derleyici Nasıl Karar Verir:** Derleyici, fonksiyon çağrısı sırasında verilen parametrelerin tiplerine bakarak hangi fonksiyonun çağrılacağını belirler. En uygun eşleşmeyi bulmaya çalışır.
- **Ambiguite:** Eğer derleyici, hangi fonksiyonun çağrılacağına karar veremezse, "ambiguous call" hatası verir.
- **Default Parametreler:** Fonksiyon aşırı yükleme ile default parametreler bir arada kullanılamaz.

C++ Kapsam (Scope)

- **Yerel Değişkenler**: Fonksiyon içinde tanımlanan değişkenler, sadece o fonksiyonun içinde erişilebilirdir.
- **Global Değişkenler**: Tüm fonksiyonlar tarafından erişilebilen, fonksiyonların dışında tanımlanan değişkenlerdir.

```cpp
int globalVar = 100;

void fonksiyon() {
    int localVar = 10;
    cout << globalVar;  // Erişilebilir
    cout << localVar;   // Erişilebilir
}

int main() {
    fonksiyon();
    cout << globalVar;  // Erişilebilir
    // cout << localVar;  // Erişilemez (Hata)
}
```

C++ Özyineleme (Recursion)

Özyineleme veya **rekürsiyon**, bir fonksiyonun kendi içinde kendini çağırması sürecidir. Bu, bir problemi daha küçük, benzer alt problemlere bölerek çözmeye benzer. Özyineleme, matematiksel ve bilgisayar bilimlerinde birçok problem için güçlü bir çözüm yöntemidir.

Nasıl Çalışır?

Özyinelemeli bir fonksiyon genellikle iki ana bölümden oluşur:

1. **Base Case (Temel Durum):** Bu, özyinelemenin durdurulması gereken koşuldur. Örneğin, faktöriyel hesaplayan bir fonksiyon için base case, sayının 0 veya 1 olmasıdır.
2. **Recursive Case (Özyinelemeli Durum):** Bu durumda, fonksiyon kendisini daha küçük bir girdi ile tekrar çağırır.

- Örnek: Faktöriyel Hesaplama

```
int faktoriyel(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * faktoriyel(n - 1);
    }
}

int main() {
    cout << "5! = " << faktoriyel(5);  // Sonuç: 120
}
```

- **Base case:** Eğer n 0 veya 1 ise, fonksiyon 1 döndürür.
- **Recursive case:** Eğer n 0 veya 1 değilse, fonksiyon n ile n-1'in faktöriyelinin çarpımını döndürür. Bu da fonksiyonu tekrar kendisi çağırarak bir alt seviyeye inmesine neden olur.

Özyinelemenin Avantajları

- **Bazı problemler için daha anlaşılır çözümler sunar:** Özellikle böl ve yönet yaklaşımına uygun problemler için özyineleme, daha doğal bir çözüm olabilir.
- **Matematiksel kavramları daha iyi ifade etmeye yardımcı olur:** Örneğin, matematiksel indüksiyon ile yakından ilişkilidir.

Özyinelemenin Dezavantajları

- **Performans:** Özyineleme, her çağrıda yeni bir stack frame oluşturduğu için, büyük girdiler için performans kaybına neden olabilir.
- **Anlaşılması zor olabilir:** Özellikle karmaşık özyinelemeli fonksiyonlar, anlaşılması zor olabilir.
- **Stack Overflow:** Çok derin özyinelemeler, stack overflow hatasına neden olabilir.

Özyinelemenin Kullanım Alanları

- **Sıralama algoritmaları:** Quick sort, merge sort gibi.
- **Ağaç ve grafik algoritmaları:** İn-order, pre-order, post-order traversal gibi.
- **Matematiksel hesaplamalar:** Faktöriyel, Fibonacci sayısı gibi.
- **Veri yapıları:** Bağlı listeler, ağaçlar gibi.

Önemli Noktalar

- **Base case:** Her özyinelemeli fonksiyonun bir base case'i olmalıdır, aksi takdirde sonsuz döngüye girer.
- **Recursive case:** Recursive case, problemi daha küçük alt problemlere böler.
- **Stack:** Özyineleme, stack (yığın) veri yapısını kullanır. Her fonksiyon çağrısı için bir stack frame oluşturulur.

Örnek : Fibonacci serisi nasıl özyineleme ile hesaplanır?

Fibonacci serisi, her bir sayının kendisinden önceki iki sayının toplamı olduğu bir sayı dizisidir. Özyineleme (recursion) kullanarak, Fibonacci serisini oldukça basit bir şekilde hesaplayabiliriz.

Bir Fibonacci serisinde şu kurallar geçerlidir:

- İlk sayı 0’dır.
- İkinci sayı 1’dir.
- Sonraki her sayı, kendisinden önceki iki sayının toplamıdır.

Matematiksel olarak:

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- Fibonacci(n) = $Fibonacci(n-1) + Fibonacci(n-2)$ (n ≥ 2)

Özyinelemeli Fonksiyon ile Fibonacci Hesaplama

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

// Fibonacci serisini hesaplayan özyinelemeli fonksiyon
int fibonacci(int n) {
    // Temel durumlar
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    }
    // Özyineleme
    return fibonacci(n-1) + fibonacci(n-2);
}

int main() {
    int n;
    cout << "Fibonacci serisinin kaçıncı terimini hesaplamak istiyorsunuz? ";
    cin >> n;

    cout << "Fibonacci(" << n << ") = " << fibonacci(n) << endl;

    return 0;
}
```

Açıklama:

1. **Temel durumlar**: `n == 0` ve `n == 1` için fonksiyon sabit değerleri döndürür: sırasıyla 0 ve 1.
2. **Özyinelemeli durum**: Eğer `n > 1` ise, Fibonacci fonksiyonu kendisini çağırarak `n-1` ve `n-2` terimlerini hesaplar. Bu hesaplamalar, son iki Fibonacci sayısının toplamını verir.

Örnek Çıktı:

Eğer `n = 5` girilirse, fonksiyon şu şekilde çalışacaktır:

- `fibonacci(5)` = `fibonacci(4)` + `fibonacci(3)`
- `fibonacci(4)` = `fibonacci(3)` + `fibonacci(2)`
- `fibonacci(3)` = `fibonacci(2)` + `fibonacci(1)`
- `fibonacci(2)` = `fibonacci(1)` + `fibonacci(0)`
- `fibonacci(1)` = 1 ve `fibonacci(0)` = 0 (temel durumlar)

Böylece hesaplama şu şekilde olur:

- `fibonacci(2)` = 1 + 0 = 1
- `fibonacci(3)` = 1 + 1 = 2

Baslik Dosya Kullanımı

Başlık dosyaları, C++ dilinde fonksiyonların, sınıfların ve değişkenlerin tanımlarını ve prototiplerini depolamak için kullanılan dosyalardır. Başlık dosyaları, kodun yeniden kullanılabilirliğini artırır ve projelerin düzenli ve anlaşılır olmasına yardımcı olur. Aşağıda başlık dosyalarının kullanımını fonksiyonlarla birlikte daha detaylı bir şekilde açıklayacağım.

Başlık Dosyası Nedir?

Başlık dosyası, genellikle `.h` veya `.hpp` uzantısına sahip bir dosyadır ve genellikle şu öğeleri içerir:

- Fonksiyon prototipleri
- Sınıf tanımları
- Makro tanımları
- Sabitler ve değişkenler

Başlık dosyaları, kodun birden fazla kaynak dosyası arasında paylaşılmasına olanak tanır.

Başlık Dosyası Oluşturma

Aşağıdaki adımlar, basit bir başlık dosyası oluşturma ve bu dosyayı bir C++ programında kullanma örneğini içermektedir.

1. Başlık Dosyasını Tanımlama

`matematik.h` adında bir başlık dosyası oluşturalım ve bu dosyada matematiksel işlemleri gerçekleştiren fonksiyon prototiplerini tanımlayalım.

```
// matematik.h
#ifndef MATEMATIK_H
#define MATEMATIK_H

// Fonksiyon prototipleri
int toplama(int a, int b);
int cikarma(int a, int b);
int carpma(int a, int b);
double bolme(int a, int b);

#endif // MATEMATIK_H
```

2. **Başlık Dosyasının Kullanılması**

Başlık dosyamızı kullanarak ana programı (`main.cpp`) oluşturalım.

```cpp
#include <iostream>
#include "matematik.h" // Başlık dosyasını dahil et
using namespace std;
// Fonksiyon tanımları
int toplama(int a, int b) {
    return a + b;
}
int cikarma(int a, int b) {
    return a - b;
}
int carpma(int a, int b) {
    return a * b;
}
double bolme(int a, int b) {
    if (b != 0)
        return static_cast<double>(a) / b; // int to double dönüşümü
    else {
        cout << "Hata: Bölme sıfıra yapılamaz." << endl;
        return 0; // Sıfıra bölme durumunda 0 döndür
    }
}
```

```cpp
int main() {
    int x = 10, y = 5;

    cout << "Toplama: " << toplama(x, y) << endl;
    cout << "Çıkarma: " << cikarma(x, y) << endl;
    cout << "Çarpma: " << carpma(x, y) << endl;
    cout << "Bölme: " << bolme(x, y) << endl;

    return 0;
}
```

Açıklama:

1. **Başlık Dosyası (`matematik.h`):**
   - `#ifndef`, `#define` ve `#endif` direktifleri, çoklu dahil etmeyi önlemek için kullanılır. Bu yapıya "include guard" denir.
   - Fonksiyon prototipleri, ana programda kullanılacak matematiksel işlemleri tanımlar.

2. **Ana Program (`main.cpp`):**
   - Başlık dosyası `#include "matematik.h"` ile dahil edilir. Bu sayede `toplama`, `cikarma`, `carpma`, ve `bolme` fonksiyonları kullanılabilir.
   - Fonksiyon tanımları, başlık dosyasında tanımlandığı gibi ana programda gerçekleştirilir.
   - `main` fonksiyonu içinde bu matematiksel işlemler gerçekleştirilir ve sonuçları ekrana yazdırılır.

Başlık dosyaları, C++ programlama dilinde fonksiyonlar ve diğer öğeler arasında düzeni sağlamanın yanı sıra, kodun yeniden kullanılabilirliğini artırır. Projelerinizde fonksiyonlarınızı ve diğer öğelerinizi başlık dosyalarında tanımlayarak daha modüler ve düzenli bir yapı oluşturabilirsiniz. Başlık dosyalarının kullanımı, büyük projelerde kodun yönetimini kolaylaştırır ve hata ayıklama sürecini hızlandırır.

Uygulama 1

Problem: Bir endüstriyel robotun hareketlerini kontrol etmek için bir C++ programı yazmak istiyoruz. Robot, belirli bir noktaya gitmek için x, y ve z eksenlerinde hareket edecektir.

- Bunun için gerekli C++ kodunu yazınız.

İpucu: `void hareketEttir(int eksen, int hedef)`

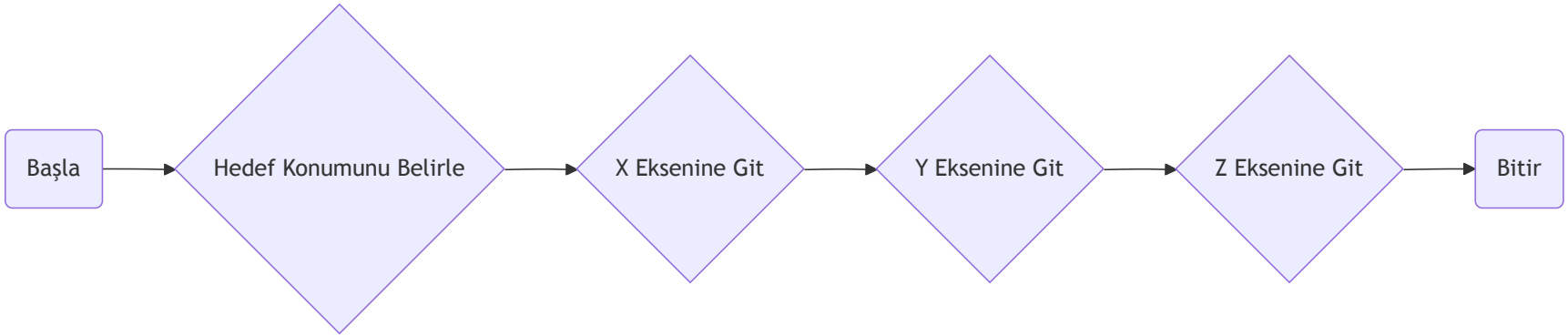
```cpp
#include <iostream>

// Robotun bir eksen üzerinde hareket etmesini sağlayan fonksiyon
void hareketEttir(int eksen, int hedef) {
    // Bu kısımda, robot kontrol kartına gerekli komutlar gönderilir.
    // Örneğin, bir seri port üzerinden veri gönderilebilir.
    std::cout << "Robot " << eksen << " ekseninde " << hedef << " konumuna hareket ettiriliyor." << std::endl;
}

int main() {
    // Robotun hedef konumu (x, y, z)
    int hedefX = 100;
    int hedefY = 50;
    int hedefZ = 20;

    // Robotu hedef konuma taşı
    hareketEttir(1, hedefX); // X ekseni
    hareketEttir(2, hedefY); // Y ekseni
    hareketEttir(3, hedefZ); // Z ekseni

    return 0;
```



- `hareketEttir` **fonksiyonu:**
  - `eksen` parametresi: Hangi eksenin hareket ettirileceğini belirtir.
  - `hedef` parametresi: Hedef konumu belirtir.
  - Fonksiyon içerisinde, gerçek dünyadaki robot kontrol kartına gerekli komutlar gönderilecek kod yer almalıdır. Bu kısım, kullanılan robot kontrol kartına ve iletişim protokolüne göre değişir. Bu örnekte, sadece ekrana bir mesaj yazdırılarak gösterilmiştir.

- `main` **fonksiyonu:**
  - Robotun hedef konumu belirlenir.
  - `hareketEttir` fonksiyonu, her bir eksen için çağırılarak robotun hedef konuma taşınması sağlanır.

Uygulama 2

Bir mekatronik sistemde, sıcaklık ve nem sensörlerinden alınan verilerin işlenerek, belirli eşik değerler aşıldığında alarmın tetiklenmesi gerekmektedir. Bu sistem, veri işleme ve alarm kontrolünü fonksiyonlar aracılığıyla gerçekleştirecektir. Ayrıca, başlık dosyaları kullanılarak kodun modülerliği ve yeniden kullanılabilirliği sağlanacaktır.

Proje Bileşenleri

1. **Başlık Dosyası (`sensor.h`):**
   - Sensör verilerini okuma fonksiyonları
   - Alarm kontrol fonksiyonu
2. **Ana Program (`main.cpp`):**
   - Sensör verilerini alma
   - Verileri işleme ve alarm kontrolü
3. **Fonksiyon Tanımları (`sensor.cpp`):**
   - `sensor.h` başlık dosyasındaki fonksiyonların tanımları

1. Başlık Dosyası (`sensor.h`)

Başlık dosyası, fonksiyon prototiplerini ve gerekli tanımlamaları içerir. `#ifndef`, `#define` ve `#endif` direktifleri ile **include guard** kullanılarak çoklu dahil etme önlenir.

```
// sensor.h
#ifndef SENSOR_H
#define SENSOR_H

// Fonksiyon Prototipleri
double okuSicaklik();
double okuNem();
void kontrolEt(double sicaklik, double nem, bool &alarm, double sicaklikEsik, double nemEsik);

#endif // SENSOR_H
```

2. Fonksiyon Tanımları (`sensor.cpp`)

Bu dosya, `sensor.h` başlık dosyasında tanımlanan fonksiyonların gerçek tanımlarını içerir.

```cpp
// sensor.cpp
#include <iostream>
#include "sensor.h"

using namespace std;

// Sıcaklık sensöründen veri okuma fonksiyonu (Simülasyon)
double okuSicaklik() {
    // Gerçek uygulamada burada sensör verisi okunur
    // Bu örnekte rastgele bir sıcaklık değeri döndürülmektedir
    return 25.0 + rand() % 10; // 25.0 ile 34.0 arasında bir değer
}
```

(`sensor.cpp`) DEVAM

```cpp
// Nem sensöründen veri okuma fonksiyonu (Simülasyon)
double okuNem() {
    // Gerçek uygulamada burada sensör verisi okunur
    // Bu örnekte rastgele bir nem değeri döndürülmektedir
    return 40.0 + rand() % 20; // 40.0 ile 59.0 arasında bir değer
}

// Alarm kontrol fonksiyonu
void kontrolEt(double sicaklik, double nem, bool &alarm, double sicaklikEsik, double nemEsik) {
    if (sicaklik > sicaklikEsik || nem > nemEsik) {
        alarm = true;
    }
}
```

3. Ana Program (`main.cpp`)

Ana program, sensör verilerini alır, bu verileri işler ve gerektiğinde alarmı tetikler.

```cpp
// main.cpp
#include <iostream>
#include <cstdlib> // rand() fonksiyonu için
#include <ctime>   // srand() fonksiyonu için
#include "sensor.h"

using namespace std;

int main() {
    // Rastgele sayı üretimi için tohumlama
    srand(time(0));

    // Eşik Değerler
    double sicaklikEsik = 30.0; // 30°C
    double nemEsik = 50.0;      // %50

    // Alarm Durumu
    bool alarm = false;

    // Sensör Verilerini Oku
    double sicaklik = okuSicaklik();
    double nem = okuNem();
```

(`main.cpp`) DEVAM

```
    // Verileri Ekrana Yazdır
    cout << "Sıcaklık: " << sicaklik << "°C" << endl;
    cout << "Nem: " << nem << "%" << endl;

    // Alarm Kontrolü
    kontrolEt(sicaklik, nem, alarm, sicaklikEsik, nemEsik);

    if (alarm) {
        cout << "ALARM: Eşik değerler aşıldı!" << endl;
    } else {
        cout << "Sistem normal çalışıyor." << endl;
    }

    return 0;
}
```

https://github.com/marmara-mekatronik/MRM3049_20

4. Proje Yapısı ve Derleme

Projenin dosya yapısı şu şekilde olmalıdır:

```
proje_adi/
├── main.cpp
├── sensor.h
└── sensor.cpp
```

Ödev 1: Akıllı Sulama Sistemi

Açıklama:

Öğrenciler, bir akıllı sulama sistemi simüle eden bir C++ programı geliştirecekler. Program, toprak nem sensöründen alınan verileri kullanarak bitkilerin sulanıp sulanmayacağına karar verecektir.

Gereksinimler:

1. **Sensör Veri Okuma:**
   - Toz, sıcaklık ve nem sensörlerinden rastgele veriler okunmalıdır.
   - Her sensör için ayrı fonksiyonlar tanımlanmalıdır (`okuNem()`, `okuSicaklik()`, `okuToz()`).
2. **Eşik Değerler:**
   - Nem seviyesi için bir eşik belirleyin (örneğin, %30).
   - Sıcaklık ve toz seviyeleri için de eşikler belirleyin.

3. **Alarm Sistemi:**
    - Nem seviyesi eşik değerinin altına düştüğünde sulama sistemini aktif hale getirin ve ekrana "Sulama başlatıldı" mesajını yazdırın.
    - Sıcaklık veya toz seviyesi eşik değerlerini aştığında alarm mesajı yazdırılmalıdır.
4. **Fonksiyon Prototipleri ve Başlık Dosyaları:**
    - Programda fonksiyon prototiplerini kullanarak modüler bir yapı oluşturulmalıdır.
5. **Örnek Çıktı:**
    - Programın çıktısı, sensörlerden okunan verileri ve sulama durumunu göstermelidir.

Ödev 2: Robotik Kol Kontrol Sistemi

Açıklama:

Öğrenciler, bir robotik kolun kontrol sistemini simüle eden bir C++ programı geliştirecekler. Program, robotik kolun farklı hareketlerini (yukarı, aşağı, sola, sağa) kontrol etmek için fonksiyonlar içerecektir.

Gereksinimler:

1. **Robotik Kol Hareketleri:**
   - `yukariHareket()`, `asagiHareket()`, `solHareket()`, `sagHareket()` isimli fonksiyonlar tanımlayın.
   - Her fonksiyon, robot kolun belirli bir hareketini simüle etmelidir (örneğin, ekrana "Robot kol yukarı hareket ediyor" mesajı yazdırmak).
2. **Kontrol Paneli:**
   - Kullanıcıdan bir hareket girişi almak için bir kontrol paneli oluşturun (örneğin, bir menü sistemi).
   - Kullanıcıdan hangi hareketin yapılacağını seçmesini isteyin ve ilgili fonksiyonu çağırın.

3. **Durum Kontrolü:**
    - Robot kolun mevcut konumunu takip eden bir değişken kullanın ve bu değişkene göre uygun hareketi gerçekleştirin.
4. **Fonksiyon Prototipleri ve Başlık Dosyaları:**
    - Programda fonksiyon prototiplerini kullanarak modüler bir yapı oluşturulmalıdır.
5. **Örnek Çıktı:**
    - Kullanıcının yaptığı seçimlere göre robot kolun hareketlerini gösteren bir çıktı oluşturun.