Fonksiyonlar

Table of contents

Bölüm 5: Fonksiyonlar - Kodun Yapı Taşları: Derinlemesine Bir Bakış **1**
5.1 Fonksiyon Tanımlama: Bir Görevi Paketleme 2
5.2 Fonksiyon Çağırma: Görevi Başlatma 4
5.3 `return` ve `printf` Arasındaki Fark 5
5.4 Fonksiyon Prototipleri: Derleyiciye Bilgi Vermek 5
5.5 Özyinelemeli Fonksiyonlar: Kendini Çağırmak 7
5.5.1 Döngüler vs. Özyineleme 8
5.5.2 Özyineleme Ne Zaman Kullanılır? 8
5.6 `static` Değişkenler: Değerini Koruma 8
Örnek: Sayaç Fonksiyonu 9
5.7 Değişkenlerin Kapsamı: Local ve Global 9
Örnek 10
5.8 Fonksiyonların Avantajları: Kodun Düzenlenmesi ve Yeniden Kullanımı 10
5.9 Alıştırmalar 10
5.10 Sonuç 11

Bölüm 5: Fonksiyonlar - Kodun Yapı Taşları: Derinlemesine Bir Bakış

Fonksiyonlar, C programlama dilinde kodun temel yapı taşlarıdır. Karmaşık problemleri daha küçük, yönetilebilir ve yeniden kullanılabilir parçalara ayırmamızı sağlarlar. Bu bölümde, fonksiyonların nasıl tanımlandığını, çağrıldığını ve kullanıldığını detaylı olarak inceleyeceğiz. Ayrıca fonksiyon prototiplerinin önemini, değişkenlerin kapsamını ve kullanımını öğreneceğiz.

5.1 Fonksiyon Tanımlama: Bir Görevi Paketleme

Fonksiyon tanımlamak, belirli bir görevi yerine getiren bir kod bloğu oluşturmak anlamına gelir. C dilinde bir fonksiyon tanımlamak için şu adımları izleriz:

1. **Geri Dönüş Tipi (Return Type):** Fonksiyonun döndüreceği değerin veri tipini belirtir. Fonksiyon bir değer döndürmüyorsa, geri dönüş tipi `void` olarak belirtilir.
2. **Fonksiyon Adı (Function Name):** Fonksiyonun adı, fonksiyona erişmek için kullanılır. Fonksiyon adları, değişken isimlendirme kurallarına uymalıdır.
3. **Parametre Listesi (Parameter List):** Fonksiyona geçirilen değerlerin listesidir. Parametreler, fonksiyon içinde kullanılacak değişkenlerdir.
4. **Fonksiyon Gövdesi (Function Body):** Fonksiyonun gerçekleştirdiği işlemleri içeren kod bloğudur.

Fonksiyon Tanımlama Sözdizimi:

```
geri_dönüş_tipi fonksiyon_adı(parametre_listesi) {
  // Fonksiyon gövdesi (işlemler)
  return değer; // İsteğe bağlı
}
```

Örnek 1: İki Sayıyı Toplayan Fonksiyon

```
int topla(int sayi1, int sayi2) {
  int toplam = sayi1 + sayi2;
  return toplam;
}
```

Bu örnekte, `topla` adında bir fonksiyon tanımladık. Fonksiyon, iki tam sayı (`sayi1` ve `sayi2`) alır ve bunları toplar. Sonucu, `toplam` değişkenine atar ve `return` ifadesi ile geri döndürür.

Örnek 2: Faktöriyel Hesaplayan Fonksiyon

```
int faktoriyel(int sayi) {
  if (sayi == 0) {
    return 1;
  } else {
    return sayi * faktoriyel(sayi - 1);
  }
}
```

Bu örnekte, `faktoriyel` adında bir fonksiyon tanımladık. Fonksiyon, bir tam sayı (`sayi`) alır ve bu sayının faktöriyelini hesaplar. Bu fonksiyon, özyinelemeli (recursive) bir fonksiyondur, yani kendini çağırır.

Daha Fazla Fonksiyon Tanımlama Örneği:

- **Bir sayının karesini hesaplayan fonksiyon:**

```
int kare(int sayi) {
  return sayi * sayi;
}
```

- **Bir sayının tek mi çift mi olduğunu kontrol eden fonksiyon:**

```
int tekMiCiftMi(int sayi) {
  if (sayi % 2 == 0) {
    return 1; // Çift
  } else {
    return 0; // Tek
  }
}
```

- **Bir string'in uzunluğunu hesaplayan fonksiyon (strlen fonksiyonunu kullanmadan):**

```
int stringUzunlugu(char str[]) {
  int uzunluk = 0;
  while (str[uzunluk] != '\0') {
    uzunluk++;
  }
  return uzunluk;
}
```

- **Void Fonksiyon Örneği: Merhaba Dünya Yazdırma**

```
void merhabaDunyaYazdir() {
  printf("Merhaba Dünya!\n");
}
```

Bu örnekte, `merhabaDunyaYazdir` fonksiyonu, ekrana "Merhaba Dünya!" mesajını yazdırır. Fonksiyon hiçbir değer döndürmediği için `void` geri dönüş tipine sahiptir ve `return` ifadesi kullanılmaz.

5.2 Fonksiyon Çağırma: Görevi Başlatma

Bir fonksiyonu çağırmak için, fonksiyonun adını ve parametrelerini (varsa) belirtmemiz gerekir. Fonksiyon çağrıldığında, program akışı fonksiyona geçer ve fonksiyon gövdesi çalıştırılır.

Fonksiyon Çağırma Sözdizimi:

```
fonksiyon_adı(parametreler);
```

Örnek: `topla` ve `faktoriyel` Fonksiyonlarını Çağırma

```
int main() {
  int toplam = topla(5, 3);  // toplam değişkenine 8 değeri atanır.
  int fakt = faktoriyel(4);  // fakt değişkenine 24 değeri atanır.

  printf("Toplam: %d\n", toplam);
  printf("Faktoriyel: %d\n", fakt);

  return 0;
}
```

Daha Fazla Fonksiyon Çağırma Örneği:

```
int main() {
  int sayi = 5;
  int karesi = kare(sayi);  // karesi değişkenine 25 değeri atanır.

  int sayi2 = 6;
  int tekCift = tekMiCiftMi(sayi2); // tekCift değişkenine 1 (Çift) değeri atanır.

  char metin[] = "Merhaba";
  int uzunluk = stringUzunlugu(metin); // uzunluk değişkenine 7 değeri atanır.

  printf("Karesi: %d\n", karesi);
  printf("Tek mi Çift mi: %d\n", tekCift);
  printf("Uzunluk: %d\n", uzunluk);

  merhabaDunyaYazdir(); // void fonksiyon çağrısı

  return 0;
}
```

5.3 `return` ve `printf` Arasındaki Fark

- `return` ifadesi, bir fonksiyonun değerini döndürür ve **fonksiyonun çalışmasını sonlandırır**. Döndürülen değer, fonksiyonu çağıran yere gönderilir ve başka işlemlerde kullanılabilir.
- `printf` fonksiyonu ise, bir değeri ekrana yazdırır, ancak fonksiyonun değerini etkilemez veya fonksiyonu sonlandırmaz. `printf` fonksiyonu, ekrana bilgi yazdırmak için kullanılır, ancak bu bilgi programın çalışmasını etkilemez.

Örnek:

```c
#include <stdio.h>

int ikiKati(int x) {
  int sonuc = x * 2;
  printf("Fonksiyon içinde: %d\n", sonuc); // Ekrana yazdırır, ancak fonksiyon devam eder
  return sonuc; // Fonksiyonun değeri döndürülür ve fonksiyon sona erer
}

int main() {
  int a = 5;
  int b = ikiKati(a);
  printf("Fonksiyon dışında: %d\n", b);
  return 0;
}
```

Bu örnekte, `ikiKati` fonksiyonu, aldığı değeri iki ile çarpar ve sonucu hem ekrana yazdırır (`printf`) hem de `return` ifadesi ile döndürür. `main` fonksiyonunda, `ikiKati` fonksiyonu çağrıldığında döndürülen değer `b` değişkenine atanır.

Çıktı:

```
Fonksiyon içinde: 10
Fonksiyon dışında: 10
```

5.4 Fonksiyon Prototipleri: Derleyiciye Bilgi Vermek

Fonksiyon prototipleri, derleyiciye fonksiyonun geri dönüş tipi, adı ve parametreleri hakkında bilgi verir. Bu bilgiler, derleyicinin fonksiyon çağrılarını doğru bir şekilde kontrol etmesini ve hataları önceden tespit etmesini sağlar. Fonksiyon prototipleri, fonksiyon tanımlamasından önce bildirilmelidir.

Fonksiyon Prototipi Sözdizimi:

```
geri_dönüş_tipi fonksiyon_adı(parametre_listesi);
```

Örnek 1: Prototip Kullanımı

```
#include <stdio.h>

// Fonksiyon prototipi
int usAl(int taban, int us);

int main() {
  int x = 2, y = 3;
  int sonuc = usAl(x, y); // Fonksiyon prototipi sayesinde derleyici bu çağrıyı anlayabilir.
  printf("%d üssü %d = %d\n", x, y, sonuc);
  return 0;
}

// Fonksiyon tanımı
int usAl(int taban, int us) {
  int sonuc = 1;
  for (int i = 0; i < us; i++) {
    sonuc *= taban;
  }
  return sonuc;
}
```

Örnek 2: Prototip Kullanılmaması

```
#include <stdio.h>

int main() {
  int x = 2, y = 3;
  int sonuc = usAl(x, y); // Hata! Derleyici usAl fonksiyonunu henüz tanımıyor.
  printf("%d üssü %d = %d\n", x, y, sonuc);
  return 0;
}

// Fonksiyon tanımı
int usAl(int taban, int us) {
  int sonuc = 1;
  for (int i = 0; i < us; i++) {
    sonuc *= taban;
```

```
  }
  return sonuc;
}
```

Daha Fazla Fonksiyon Prototipi Örneği:

```
#include <stdio.h>

int kare(int sayi);          // Fonksiyon prototipi
int tekMiCiftMi(int sayi); // Fonksiyon prototipi
int stringUzunlugu(char str[]); // Fonksiyon prototipi
void yildizYazdir(int adet);  // Fonksiyon prototipi (void fonksiyonlar için de prototip kul

int main() {
  // ... (Fonksiyon çağrıları ve diğer kodlar)
}

// Fonksiyon tanımları ...
```

5.5 Özyinelemeli Fonksiyonlar: Kendini Çağırmak

Özyinelemeli fonksiyonlar, kendini çağıran fonksiyonlardır. Bu tür fonksiyonlar, bir problemi daha küçük alt problemlere bölerek çözer ve her alt problem için kendini tekrar çağırır. Bu işlem, temel bir duruma ulaşılana kadar devam eder.

Örnek: Faktöriyel Hesaplama (Özyinelemeli)

```
#include <stdio.h>

int faktoriyel(int n) {
  if (n == 0) {
    return 1; // Temel durum: 0! = 1
  } else {
    return n * faktoriyel(n - 1); // Özyinelemeli adım
  }
}

int main() {
  int sayi = 5;
  int sonuc = faktoriyel(sayi);
  printf("%d! = %d\n", sayi, sonuc);
```

```
  return 0;
}
```

Açıklama:

1. `faktoriyel(5)` çağrılır. 5, 0'dan büyük olduğu için `else` bloğuna girilir.
2. `5 * faktoriyel(4)` hesaplanır. `faktoriyel(4)`'ü hesaplamak için fonksiyon kendini tekrar çağırır.
3. `faktoriyel(4)`, `4 * faktoriyel(3)` olarak hesaplanır. Fonksiyon tekrar çağrılır.
4. Bu işlem, `faktoriyel(0)` çağrısına kadar devam eder.
5. `faktoriyel(0)`, temel duruma ulaşır ve 1 değerini döndürür.
6. Döndürülen değerler, çağrı zinciri boyunca geriye doğru işlenir: 1, 2, 6, 24, 120.
7. Sonuç olarak, `faktoriyel(5)`, 120 değerini döndürür.

5.5.1 Döngüler vs. Özyineleme

- Hem döngüler hem de özyinelemeli fonksiyonlar, tekrarlayan işlemleri gerçekleştirmek için kullanılabilir.
- Döngüler, genellikle daha basit ve anlaşılır bir yapıya sahiptir ve daha az bellek kullanır.
- Bazı problemler özyinelemeli fonksiyonlar kullanılarak daha zarif ve anlaşılır bir şekilde çözülebilir, özellikle de problem kendi kendini tekrar eden bir yapıya sahipse (örneğin, ağaç veri yapıları gibi).
- Özyinelemede her fonksiyon çağrısı için bellek yığınında (stack) bir alan ayrılır. Çok derin özyineleme seviyeleri, **"stack overflow"** hatasına neden olabilir. Bu durum, belleğin tükendiği anlamına gelir. Döngüler ise bu riski taşımazlar.

5.5.2 Özyineleme Ne Zaman Kullanılır?

- Problem, daha küçük alt problemlere bölünebiliyorsa
- Temel bir durum (recursive çağrıların durduğu nokta) varsa
- Döngüler kullanarak çözüm karmaşıklaşıyorsa

5.6 `static` Değişkenler: Değerini Koruma

- Bir fonksiyon içinde `static` anahtar kelimesi ile tanımlanan değişkenler, fonksiyon çağrıları arasında değerlerini korur.
- Normalde, local değişkenler fonksiyon sona erdiğinde bellekten silinir, ancak `static` değişkenler programın sonuna kadar bellekte kalır. Bu, fonksiyonun bir sonraki çağrısında değişkenin değerinin korunmasını sağlar.

Örnek: Sayaç Fonksiyonu

```c
#include <stdio.h>

int sayac() {
  static int count = 0; // static değişken, ilk değer 0 olarak atanır
  count++;
  return count;
}

int main() {
  printf("Sayaç: %d\n", sayac()); // Çıktı: 1
  printf("Sayaç: %d\n", sayac()); // Çıktı: 2
  printf("Sayaç: %d\n", sayac()); // Çıktı: 3
  return 0;
}
```

Açıklama:

- `count` değişkeni `static` olarak tanımlandığı için, fonksiyon ilk kez çağrıldığında 0 olarak başlatılır ve değeri bellekte saklanır.
- Fonksiyon sonraki çağrılarda, `count` değişkeni tekrar başlatılmaz, bellekteki son değeri kullanılır.
- Her fonksiyon çağrısında `count` değişkeni 1 artırılır ve yeni değeri döndürülür.

Eğer `static` anahtar kelimesi kullanılmasaydı, `count` değişkeni her fonksiyon çağrısında 0 olarak başlatılır ve çıktı her zaman 1 olurdu.

`static` Değişkenler Hakkında Uyarı:

`static` değişkenlerin değerleri fonksiyon çağrıları arasında korunurken, bu durum beklenmedik sonuçlara yol açabilir. Özellikle, aynı `static` değişkeni kullanan birden fazla fonksiyon varsa, bir fonksiyonun yaptığı değişiklik diğer fonksiyonları etkileyebilir. Bu nedenle, `static` değişkenleri dikkatli kullanmak ve mümkün olduğunca local değişkenleri tercih etmek iyi bir programlama uygulamasıdır.

5.7 Değişkenlerin Kapsamı: Local ve Global

- **Local Değişkenler:** Bir fonksiyonun içinde tanımlanan değişkenlerdir. Sadece o fonksiyon içinde kullanılabilirler. Fonksiyon sona erdiğinde, local değişkenler bellekten silinir.
- **Global Değişkenler:** Bir fonksiyonun dışında tanımlanan değişkenlerdir. Programın her yerinde kullanılabilirler. Programın başlangıcından sonuna kadar bellekte kalırlar.

Örnek

```c
#include <stdio.h>

int globalDegisken = 10; // Global değişken

void fonksiyon() {
  int localDegisken = 5; // Local değişken
  printf("Fonksiyon içinde globalDegisken: %d\n", globalDegisken);
  printf("Fonksiyon içinde localDegisken: %d\n", localDegisken);
}

int main() {
  printf("main içinde globalDegisken: %d\n", globalDegisken);
  fonksiyon();
  // printf("main içinde localDegisken: %d\n", localDegisken); // Hata! localDegisken, main i
  return 0;
}
```

5.8 Fonksiyonların Avantajları: Kodun Düzenlenmesi ve Yeniden Kullanımı

Fonksiyonlar kullanmanın birçok avantajı vardır:

- **Kodun Okunabilirliğini Artırır:** Fonksiyonlar, programı daha küçük ve daha anlaşılır parçalara böler.
- **Kodun Yeniden Kullanımını Sağlar:** Bir fonksiyon, programın farklı bölümlerinden tekrar tekrar çağrılabilir.
- **Kodun Bakımını Kolaylaştırır:** Bir fonksiyonda değişiklik yapıldığında, programın diğer bölümlerini etkilemez.
- **Hata Ayıklamayı Kolaylaştırır:** Fonksiyonlar, programı daha küçük parçalara böldüğü için hata ayıklamayı kolaylaştırır.

5.9 Alıştırmalar

1. Kullanıcıdan alınan iki tam sayının en büyüğünü bulan bir fonksiyon yazın.
2. Kullanıcıdan alınan bir sayının mükemmel sayı olup olmadığını kontrol eden bir fonksiyon yazın. (Mükemmel sayı, kendisi hariç pozitif bölenlerinin toplamına eşit olan sayıdır. Örneğin, 6 mükemmel bir sayıdır çünkü $1 + 2 + 3 = 6$)
3. Kullanıcıdan alınan iki sayının OBEB'ini (Ortak Bölenlerin En Büyüğü) bulan bir fonksiyon yazın.

4. Kullanıcıdan bir sayı alıp bu sayıya kadar olan tüm asal sayıları ekrana yazdıran bir fonksiyon yazın.
5. Kullanıcıdan bir sayı alıp bu sayının faktöriyelini hesaplayan bir fonksiyon yazın (Döngü kullanarak).
6. 0 ile 100 arasında rastgele bir sayı üreten bir fonksiyon yazın.

5.10 Sonuç

Bu bölümde, fonksiyonları daha detaylı inceledik, fonksiyon prototiplerinin önemini vurguladık, değişkenlerin kapsamını anlattık ve özyinelemeli fonksiyonları öğrendik. Ayrıca, daha fazla örnek ile konuyu pekiştirdik. Unutmayın, fonksiyonlar kodun yapı taşlarıdır ve iyi bir programcı olmak için fonksiyonları etkin bir şekilde kullanmayı öğrenmek çok önemlidir.