

Fonksiyonlar-3

Özyineleme (Recursion)

Bazı problem türleri için, fonksiyonların kendilerini çağırması gerçekten yararlıdır. Yinelemeli bir fonksiyon, kendisini başka bir fonksiyon aracılığıyla doğrudan veya dolaylı olarak çağırarak bir fonksiyondur. Özyineleme, üst düzey bilgisayar bilimi derslerinde tartışılan karmaşık bir konudur. Bu bölümde ve sonraki bölümde, basit özyineleme örnekleri sunulacaktır.

Temel Durumlar ve Özyinelemeli Çağrılar

Özyinelemeli bir fonksiyon, bir sorunu çözmek için çağrılır. Fonksiyon aslında yalnızca en basit veya temel durumların nasıl çözüleceğini bilir. Fonksiyon bir temel durumla çağırılırsa, basitçe bir sonuç döndürür. Daha karmaşık bir problemle çağırıldığında, fonksiyon tipik olarak problemi iki kavramsal parçaya ayırır:

- fonksiyonun nasıl yapılacağını bildiği ve
- nasıl yapılacağını bilmediği şeyler.

Özyineleme adımı aynı zamanda bir dönüş ifadesi de içerir, çünkü orijinal çağırana geri gönderilecek bir sonuç oluşturulur. Bu kavramlara bir örnek olarak, popüler bir matematiksel hesaplamayı gerçekleştirmek için özyinelemeli bir program yazalım.

Yinelemeli Faktöriyel Hesaplama

Negatif olmayan bir n tamsayısının faktöriyeli, $n!$ (" n faktöriyel"), şu çarpımdır;

$$n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

Örneğin $5!$ 120'ye eşit olan $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ çarpımıdır.

0'dan büyük veya 0'a eşit bir tamsayının faktöriyeli, aşağıdaki gibi bir for ifadesi kullanılarak yinelemeli (iterasyon) olarak hesaplanabilir:

```
unsigned long long int factorial = 1;  
for (int counter = number; counter > 1; --counter)  
factorial *= counter;
```

Faktöriyel fonksiyonun özyinelemeli bir tanımı, aşağıdaki ilişkiyi gözlemleyerek elde edilir:

$$N! = n \cdot (n - 1)!$$

Örneğin, $5!$ açıkça $5 \cdot 4!$ 'e eşittir aşağıda gösterildiği gibi:

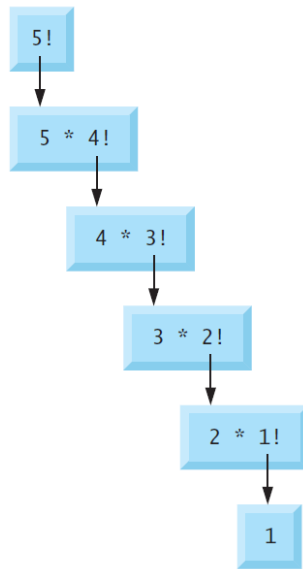
$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$

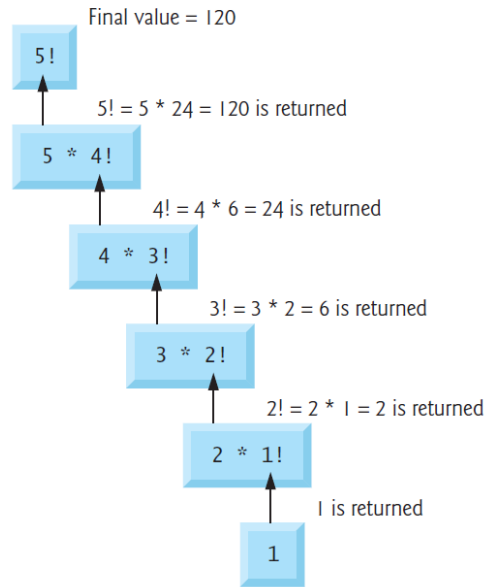
$$5! = 5 \cdot (4!)$$

$5!$ değerlendirmesi aşağıdaki diyagramda gösterildiği gibi ilerleyecektir. Aşağıdaki şemanın (a) kısmı, yinelemeli çağrının ardışıklığının 1'e kadar nasıl ilerlediğini gösterir! 1 (yani temel durum) olarak değerlendirilerek özyineleme sonlandırılır. Kısım (b), son değer hesaplanıp döndürülene kadar her özyinelemeli çağrıdan çağırana döndürülen değerleri gösterir.

a) Sequence of recursive calls



b) Values returned from each recursive call



Özyinelemeli Faktör Hesaplamalarını Uygulama

Şekil 5.9, 0-21 tamsayılarının faktöriyelerini hesaplamak ve yazdırmak için özyinelemeyi kullanır (unsigned long long int türünün seçimi yeri geldiğinde açıklanacaktır).

(NOT: int i; yi for döngüsü dışına almayı unutmayalım.)

```

1  // fig05_09.c
2  // Recursive factorial function.
3  #include <stdio.h>
4
5  unsigned long long int factorial(int number);
6
7  int main(void) {
8      // calculate factorial(i) and display result
9      for (int i = 0; i <= 21; ++i) {
10         printf("%d! = %llu\n", i, factorial(i));
11     }
12 }
13
14 // recursive definition of function factorial
15 unsigned long long int factorial(int number) {
16     if (number <= 1) { // base case
17         return 1;
18     }
19     else { // recursive step
20         return (number * factorial(number - 1));
21     }
22 }
  
```

Faktöriyel Fonksiyonu

Özyinelemeli faktöriyel fonksiyonu önce bir sonlandırma koşulunun doğru olup olmadığını, yani sayının 1'den küçük veya 1'e eşit olup olmadığını test eder. Sayı gerçekten 1'den küçük veya ona eşitse, faktöriyel 1 döndürür, daha fazla yineleme gerekmez ve program sonlandırılır. Sayı 1'den büyükse aşağıdaki ifade, “number” ve “number – 1”in faktöriyelini yinelemeli bir faktöriyel çağrısı olarak ifade eder.

```
return number * factorial(number - 1);
```

factorial(number - 1) çağrısı, orijinal hesaplama factorial(number)'dan biraz daha basit bir problemdir. Temel durumu atlamak veya özyineleme adımı, temel duruma yaklaşmayacak şekilde yanlış yazmak, sonsuz yinelemeye neden olur ve sonunda belleği tüketir. Bu, problem yinelemeli olmayan bir çözümdeki sonsuz döngü sorununa benzer.

Faktöriyeler Hızla Büyür

Faktöriyel fonksiyonu (satır 15–22) bir **int** alır ve işaretsiz bir **long long int** döndürür. C standardı, **unsigned long long int** türünde bir değişkenin en az 18.446.744.073.709.551.615 kadar büyük bir değer tutabileceğini belirtir. Şekil 5.9'da görülebileceği gibi, faktöriyel değerler hızla büyür. Programın daha büyük faktöriyel değerleri hesaplayabilmesi için **unsigned long long int** veri türünü seçtik. **%llu** dönüştürme belirteci, **unsigned long long int** değerleri yazdırmak için kullanılır. Ne yazık ki, factorial fonksiyonu büyük değerleri o kadar hızlı üretir ki, **unsigned long long int** bile birçok faktöriyel değeri yazdırmamıza yardımcı olmaz, çünkü bu türün maksimum değeri hızla aşılır.

Tamsayı Türlerinin Sınırlamaları Vardır

unsigned long long int kullandığımızda bile, 21'den sonraki faktöriyelleri hesaplayamıyoruz! Bu, C gibi prosedürel programlama dillerindeki bir zayıflığa işaret eder; dil, çeşitli uygulamaların benzersiz gereksinimlerini karşılayacak şekilde kolayca genişletilemez. Ancak esne yönelimli diller genişletilebilir. Programcılar, sınıflar adı verilen bir dil özelliği aracılığıyla, gelişigüzel büyük tamsayılar tutabilen yeni veri türleri oluşturabilir.

Özyinelemeyi Kullanma Örneği: Fibonacci Serisi

Fibonacci serisi

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

0 ve 1 ile başlar ve sonraki her Fibonacci sayısının önceki iki Fibonacci sayısının toplamı olma özelliğine sahiptir. Dizi özellikle sarmalın bir biçimini anlatır. Ardışık Fibonacci sayılarının oranı, 1.618... gibi sabit bir değere yakınsar. Bu sayı da doğada tekrar tekrar oluşur ve altın oran veya altın ortalama olarak adlandırılır. İnsanlar estetik açıdan hoşta giden altın oranı bulma eğilimindedir. Mimarlar genellikle uzunlukları ve genişlikleri altın ortalama oranında olan pencereler, odalar ve binalar tasarlar. Kartpostallar genellikle altın oran uzunluk/genişlik oranıyla tasarlanır. Fibonacci serisi özyinelemeli olarak şu şekilde tanımlanabilir:

fibonacci(0) = 0

fibonacci(1) = 1

fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

Şekil 5.10, fibonacci fonksiyonunu kullanarak n'inci Fibonacci sayısını yinelemeli olarak hesaplar. Fibonacci sayıları hızla büyük olma eğilimindedir. Bu nedenle, fibonacci fonksiyonunda dönüş türü için **unsigned long long int** veri türü tercih edilir.

(NOT: int number; ı for döngüsü dışına almayı unutmayalım.)

```
1 // fig05_10.c
2 // Recursive fibonacci function.
3 #include <stdio.h>
4
5 unsigned long long int fibonacci(int n); // function prototype
6
7 int main(void) {
8     // calculate and display fibonacci(number) for 0-10
9     for (int number = 0; number <= 10; number++) {
10         printf("Fibonacci(%d) = %llu\n", number, fibonacci(number));
11     }
12
13     printf("Fibonacci(20) = %llu\n", fibonacci(20));
14     printf("Fibonacci(30) = %llu\n", fibonacci(30));
15     printf("Fibonacci(40) = %llu\n", fibonacci(40));
16 }
17
18 // Recursive definition of function fibonacci
19 unsigned long long int fibonacci(int n) {
20     if (0 == n || 1 == n) { // base case
21         return n;
22     }
23     else { // recursive step
24         return fibonacci(n - 1) + fibonacci(n - 2);
25     }
26 }
```

main'den gelen fibonacci çağrıları özyinelemeli değildir (satır 10 ve 13-15), ancak fibonacci'ye yapılan sonraki tüm çağrılar özyinelemelidir (satır 24). Fibonacci her çağrıldığında, hemen temel durumu test eder; n, 0 veya 1'e eşit ise, n döndürülür. Eğer n, 1'den büyükse, özyineleme adımı, iki özyinelemeli çağrı üretir. Bunlar: Fibonacci (n-1) ve Fibonacci (n-2).

İşlenenlerin Değerlendirme Sırası

Yukarıdaki örnek, fibonacci(3) değerlendirilirken, fibonacci(2) ve fibonacci(1) olmak üzere iki özyinelemeli çağrı yapılacağını göstermektedir. Ancak bu aramalar hangi sırayla yapılacak? İşlenenlerin soldan sağa değerlendirileceğini varsayabilirsiniz. Optimizasyon nedenleriyle C, çoğu operatörün işlenenlerinin (+ dahil) değerlendirileceği sırayı belirtmez. Bu nedenle, bu çağrıların yürütüleceği sıra hakkında hiçbir varsayımda bulunmamalısınız. Çağrılar önce fibonacci(2)'yi ve ardından fibonacci(1)'i çalıştırabilir veya çağrılar ters sırayla, fibonacci(1) sonra fibonacci(2)'yi çalıştırabilir. Bu ve diğer birçok programda, nihai sonuç aynı olacaktır. Ancak bazı programlarda, bir işlenenin değerlendirilmesi, ifadenin nihai sonucunu etkileyebilecek yan etkilere sahip olabilir.

İşlenenin Değerlendirme Sırasının Belirtildiği Operatörler

C yalnızca dört operatörün işlenen değerlendirme sırasını belirtir— &&, ||, virgül (,) operatörü ve ?:. İlk üçü, işlenenlerinin soldan sağa değerlendirilmesi garanti edilen ikili operatörlerdir.

[Not: Bir fonksiyon çağrısında bağımsız değişkenleri ayırmak için kullanılan virgüller virgül operatörleri değildir.]

Son operatör, C'nin tek üçlü operatörüdür. En soldaki işleneni her zaman önce değerlendirilir. En soldaki işlenen sıfır dışında (doğru) olarak değerlendirilirse, ortadaki işlenen bir sonraki olarak değerlendirilir ve son işlenen göz ardı edilir. En soldaki işlenen sıfır (yanlış) olarak değerlendirilirse, sonraki üçüncü işlenen değerlendirilir ve orta işlenen göz ardı edilir.

Üstel Karmaşıklık

Fibonacci sayıları oluşturmak için burada kullandığımız gibi özyinelemeli programlar hakkında bir uyarı vardır. Fibonacci fonksiyonundaki her özyineleme düzeyi, arama sayısı üzerinde iki katına çıkan bir etkiye sahiptir. n'inci Fibonacci sayısını hesaplamak için yürütülen özyinelemeli çağrılarının sayısı " 2^n mertebesinde"dir. Bu hızla kontrolden çıkar. Yalnızca 20. Fibonacci sayısını hesaplamak için 220 veya yaklaşık bir milyon arama gerekir, 30. Fibonacci numarasını hesaplamak için 230 veya yaklaşık bir milyar arama gerekir, vb. Bilgisayar bilimcileri buna üstel karmaşıklık diyor. Bu tür sorunlar dünyanın en güçlü bilgisayarlarını bile küçük düşürebilir! Genel olarak karmaşıklık sorunları ve özel olarak üstel karmaşıklık, üst düzey bilgisayar bilimi dersinde ayrıntılı olarak tartışılır.

Bu bölümde, recursive veya iterasyon olarak kolayca uygulanabilen iki fonksiyon inceledik.

İterasyon ve Özyinelemenin Ortak Özellikleri

- Hem iterasyon hem de yineleme bir kontrol ifadesine dayalıdır: iterasyon bir yineleme ifadesi kullanır; özyineleme bir seçim ifadesi kullanır.
- Hem iterasyon hem de özyineleme, tekrarı içerir: İterasyon, bir yineleme ifadesi kullanır; özyineleme, tekrarlanan fonksiyon çağrıları yoluyla tekrarı sağlar.
- İterasyon ve özyinelemenin her birinin bir sonlandırma testi vardır: Döngü devam koşulu başarısız olduğunda iterasyon sona erer; bir temel durum tanındığında ise özyineleme sonlanır.
- Sayaç kontrollü iterasyon ve özyinelemenin her ikisi de yavaş yavaş sona yaklaşır: İterasyon, sayaç, döngü devam koşulunun başarısız olmasına neden olan bir değer üstlenene kadar bir sayacı değiştirmeye devam eder; özyineleme, temel duruma ulaşılan kadar orijinal problemin daha basit versiyonlarını üretmeye devam eder.
- Hem iterasyon hem de özyineleme sonsuz olarak gerçekleşebilir: Döngü devam testi hiçbir zaman yanlış olmazsa, iterasyonla birlikte sonsuz bir döngü oluşur; özyineleme adımı sorunu her seferinde temel duruma yakınsayacak şekilde azaltmazsa sonsuz yineleme oluşur. Sonsuz yineleme ve özyineleme tipik olarak bir programın mantığındaki hataların bir sonucu olarak ortaya çıkar.

Özyinelemenin Negatifleri

Özyinelemenin bazı olumsuzluğu vardır. Fonksiyon çağrılarının mekanizmasını ve dolayısıyla ek yükü tekrar tekrar çağırır. Bu, hem işlemci süresi hem de bellek alanı açısından yükü fazla olabilir. Her özyinelemeli çağrı, fonksiyonun değişkenlerinin başka bir kopyasının oluşturulmasına neden olur; bu önemli miktarda bellek tüketebilir. Öyleyse neden özyinelemeyi seçelim?

Özyinelemeli olarak çözülebilen herhangi bir problem, iterasyonla da çözülebilir. Özyinelemeli yaklaşım sorunu daha doğal bir şekilde yansıttığında ve anlaşılması ve hata ayıklaması daha kolay bir programla sonuçlandığında, iterasyonlu bir yaklaşım yerine özyinelemeli bir yaklaşım seçilir. Özyinelemeli bir çözüm seçmenin diğer bir nedeni, iterasyonlu bir çözümün belirgin olmayabileceğidir.

Bazı Gözlemler

İyi yazılım mühendisliği önemlidir ve yüksek performans önemlidir. Ne yazık ki, bu hedefler genellikle birbiriyle çelişir. İyi yazılım mühendisliği, ihtiyacımız olan daha büyük ve daha karmaşık yazılım sistemlerini geliştirme görevini daha yönetilebilir hale getirmenin anahtarıdır. Yüksek performans, donanım üzerinde her zamankinden daha fazla bilgi işlem talebi oluşturacak geleceğin sistemlerini gerçekleştirmenin anahtarıdır.

Yazılım Mühendisliđi

Büyük bir programı fonksiyonlara bölmek, iyi yazılım mühendisliđini destekler. Ama bir bedeli vardır. Fonksiyonları olmayan yekpare (yani tek parça) bir programla karşılaştırıldığında, büyük ölçüde işlevselleştirilmiş bir program, potansiyel olarak çok sayıda fonksiyon çağırısı yapar. Bunlar, bir bilgisayarın işlemci(ler)inde yürütme süresini tüketir. Yekpare programlar daha iyi performans gösterebilse de programlamak, test etmek, hata ayıklamak, sürdürmek ve geliştirmek daha zordur.

Verim

Günümüzün donanım mimarileri, fonksiyon çağrılarını verimli hale getirecek şekilde ayarlanmıştır. C derleyicileri, kodunuzu optimize etmenize yardımcı olur ve günümüzün donanım işlemcileri ve çok çekirdekli mimarisi inanılmaz derecede hızlıdır. Oluşturacağınız uygulamaların ve yazılım sistemlerinin büyük çoğunluğu için, iyi yazılım mühendisliđine odaklanmak, yüksek performanslı programlamadan daha önemli olacaktır. Bununla birlikte, oyun programlama, gerçek zamanlı sistemler, işletim sistemleri ve gömülü sistemler gibi birçok uygulama ve sistemde performans çok önemlidir.