Fonksiyonlar-2

Argümanları Değere ve Referansa Göre Geçirme

Birçok programlama dilinde argümanları iletmenin iki yolu vardır: değere göre geçiş ve referansa göre geçiş. Bir bağımsız değişken değere göre iletildiğinde, bağımsız değişkenin değerinin bir kopyası yapılır ve fonksiyona iletilir. Kopyada yapılan değişiklikler, çağırandaki orijinal değişkenin değerini etkilemez. Bir bağımsız değişken referans olarak iletildiğinde, çağıran, çağrılan fonksiyonun orijinal değişkenin değerini değiştirmesine izin verir.

Değere göre geçiş, çağrılan fonksiyonun çağıranın orijinal değişkeninin değerini değiştirmesi gerekmediği durumlarda kullanılmalıdır. Bu, doğru ve güvenilir yazılım sistemlerinin geliştirilmesini engelleyebilecek kazara yan etkileri önler. Referansla geçiş, yalnızca orijinal değişkeni değiştirmesi gereken güvenilir çağrılan fonksiyonlarla kullanılmalıdır.

C'de tüm bağımsız değişkenler değere göre iletilir. Bölüm 7, İşaretçiler'de, referansa göre geçişin nasıl elde edileceğini göstereceğiz. Bölüm 6'da, dizi bağımsız değişkenlerinin performans nedenleriyle referans olarak otomatik olarak iletildiğini göreceğiz. Bunun bir çelişki olmadığını Bölüm 7'de göreceğiz. Şimdilik, değer geçişine odaklanıyoruz.

Rastgele Sayı Üretimi

Şimdi simülasyon ve oyun oynamaya kısa bir giriş yapalım. Bu ve bir sonraki bölümde, çok sayıda özel fonksiyon içeren güzel bir şekilde yapılandırılmış bir oyun oynama programı geliştirelim. Program, üzerinde çalıştığımız fonksiyonları ve birkaç kontrol ifadesini kullanır. Şans öğesi için, <stdlib.h> başlığından C standart kütüphane fonksiyonu **rand** kullanılarak rastgele sayı üretimi yapılarak bilgisayar uygulamalarına tanıtılabilir.

Aşağıdaki ifadeyi göz önünde bulunduralım:

```
int value = rand();
```

rand fonksiyonu, 0 ile RAND_MAX (<stdlib.h> başlığında tanımlanan sembolik bir sabit) arasında bir tam sayı üretir. C standardı, RAND_MAX değerinin en az 32.767 olması gerektiğini belirtir; bu, iki baytlık (yani 16 bitlik) bir tamsayı için maksimum değerdir.

Bu bölümdeki programlar Microsoft Visual C++ üzerinde maksimum RAND_MAX değeri 32,767 ve GNU gcc ve Xcode Clang üzerinde maksimum RAND_MAX değeri 2,147,483,647 ile test edilmiştir. **rand** gerçekten rasgele tamsayılar üretiyorsa, 0 ile RAND_MAX arasındaki her sayı, **rand** her çağrıldığında eşit seçilme şansına (veya olasılığına) sahiptir.

rand tarafından doğrudan üretilen değer aralığı genellikle belirli bir uygulamada ihtiyaç duyulandan farklıdır. Örneğin, madeni para atmayı simüle eden bir program "tura" için yalnızca 0 ve "yazı" için 1 gerektirebilir. Altı kenarlı bir zarı simüle eden bir zar atma programı, 1'den 6'ya kadar rasgele tamsayılar gerektirir.

rand'ı göstermek için, altı kenarlı bir zarın 10 atışını simüle edecek ve her atışın değerini yazdıracak bir program geliştirelim. ***(Şekil 5.4). (NOT: int i; yi for döngüsü dışına alalım) ;***

```
1  // fig05_04.c
2  // Shifted, scaled random integers produced by 1 + rand() % 6.
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void) {
7
8     for (int i = 1; i <= 10; ++i) {
9        printf("%d  ", 1 + (rand() % 6)); // display random die value
10    }
11
12    puts("");
13 }
```

rand fonksiyonunun prototipi <stdlib.h> içindedir. 0 ile 5 arasında tamsayılar üretmek için 9. satırda, kalan operatörünü (%) aşağıdaki gibi **rand** ile birlikte kullanıyoruz

rand() % 6

Buna ölçeklendirme denir. 6 sayısı ölçeklendirme faktörü olarak adlandırılır. Daha sonra, önceki sonucumuza 1 ekleyerek üretilen sayı aralığını kaydırıyoruz. Çıktı, sonuçların 1 ila 6 aralığında olduğunu gösterir; bu rasgele değerlerin seçilme sırası derleyiciye göre değişebilir.

Altı Kenarlı Bir Zarı 60.000.000 Defa Atmak

Bu sayıların yaklaşık olarak eşit olasılıkla oluştuğunu göstermek için, Şekil 5.5'teki programla 60.000.000 atışlık bir zar simülasyonu yapalım. 1'den 6'ya kadar her tamsayı yaklaşık 10.000.000 kez görünmelidir. ***(Şekil 5.5). (NOT: int ROLL; u for döngüsü dışına alalım) ;***

```
1  // fig05_05.c
2  // Rolling a six-sided die 60,000,000 times.
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void) {
7     int frequency1 = 0; // rolled 1 counter
8     int frequency2 = 0; // rolled 2 counter
9     int frequency3 = 0; // rolled 3 counter
10    int frequency4 = 0; // rolled 4 counter
11    int frequency5 = 0; // rolled 5 counter
12    int frequency6 = 0; // rolled 6 counter
13
14    // loop 60000000 times and summarize results
15    for (int roll = 1; roll <= 60000000; ++roll) {
16       int face = 1 + rand() % 6; // random number from 1 to 6
17
18       // determine face value and increment appropriate counter
19       switch (face) {
20          case 1: // rolled 1
21             ++frequency1;
22             break;
23          case 2: // rolled 2
24             ++frequency2;
25             break;
26          case 3: // rolled 3
27             ++frequency3;
28             break;
29          case 4: // rolled 4
30             ++frequency4;
31             break;
32          case 5: // rolled 5
33             ++frequency5;
34             break;
35          case 6: // rolled 6
36             ++frequency6;
37             break; // optional
38       }
39    }
40
41    // display results in tabular format
42    printf("%s%13s\n", "Face", "Frequency");
43    printf("   1%13d\n", frequency1);
44    printf("   2%13d\n", frequency2);
45    printf("   3%13d\n", frequency3);
46    printf("   4%13d\n", frequency4);
47     printf("   5%13d\n", frequency5);
48     printf("   6%13d\n", frequency6);
49 }
```

Program çıktısının gösterdiği gibi, ölçekleme ve kaydırma yoluyla, altı kenarlı bir zarın atılmasını gerçekçi bir şekilde simüle etmek için rand işlevini kullandık. "face" ve "frequency" karakter dizilerini sütun başlıkları olarak yazdırmak için %s dönüştürme belirtiminin kullanıldığına dikkat edin (satır 42). Bölüm 6'da dizileri inceledikten sonra, bu 20 satırlık switch basitçe tek satırlık deyimle nasıl değiştireceğimizi göstereceğiz.

Rastgele Sayı Oluşturucuyu Rastgele Hale Getirme

Şekil 5.4'teki programın tekrar çalıştırılması yine 6 6 5 5 6 5 1 1 5 3'ü üretir. Bu, Şekil 5.4'te gösterdiğimiz değerlerin tam sırasıdır. Bunlar nasıl rastgele sayılar olabilir? İronik olarak, bu tekrarlanabilirlik **rand** fonksiyonunun önemli bir özelliğidir. Bir programda hata ayıklarken, bu tekrarlanabilirlik, bir programda yapılan düzeltmelerin düzgün çalıştığını kanıtlamak için gereklidir.

rand fonksiyonu aslında sözde rasgele sayılar üretir. **rand**'ı art arda çağırmak, rastgele görünen bir sayı dizisi üretir. Ancak, program her çalıştırıldığında dizi kendini tekrar eder. Bir program tamamen hata ayıklandıktan sonra, her yürütme için farklı bir rasgele sayı dizisi üretecek şekilde koşullandırılabilir. Buna rastgeleleştirme denir ve standart kütüphane fonksiyonu **srand** ile gerçekleştirilir. **srand** fonksiyonu bir **int** bağımsız değişkeni alır ve her program yürütmesi için farklı bir rasgele sayı dizisi üretmek üzere **rand** fonksiyonunu bir başlatıcı değer ile başlatır. **srand** fonksiyonunu Şekil 5.6'da gösteriyoruz. **srand** için fonksiyon prototipi **<stdlib.h>** içinde bulunur.

(Şekil 5.6). (NOT: int i; yi for döngüsü dışına alalım) ;

```
1  // fig05_06.c
2  // Randomizing the die-rolling program.
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void) {
7     printf("%s", "Enter seed: ");
8     int seed = 0; // number used to seed the random-number generator
9     scanf("%d", &seed);
10
11    srand(seed); // seed the random-number generator
12
13    for (int i = 1; i <= 10; ++i) {
14       printf("%d  ", 1 + (rand() % 6)); // display random die value
15    }
16
17    puts("");
18 }
```

Programı birkaç kez çalıştıralım ve sonuçları gözlemleyelim. Programın her çalıştırılışında, farklı bir başlatma değeri sağlanması koşuluyla, farklı bir rasgele sayı dizisinin elde edildiğine dikkat edin. İlk ve son çıktılar aynı başlatma değerini kullanır, dolayısıyla aynı sonuçları gösterirler. Her seferinde yeni bir başlatma değeri girmeden randomize etmek (rastgeleleştirmek) için şöyle bir ifade kullanın;

```
srand(time(NULL));
```

Bu, bilgisayarın başlatma değerini otomatik olarak elde etmek için saatini okumasını sağlar. Bu değer bir tamsayıya dönüştürülür ve rasgele sayı üretecinin başlatma değeri olarak kullanılır. **time** için fonksiyon prototipi **<time.h>** içindedir.

Genelleştirilmiş Ölçeklendirme ve Rastgele Sayıların Kaydırılması

rand fonksiyonu tarafından doğrudan üretilen değerler her zaman şu aralıktadır:
0 ≤ rand() ≤ RAND_MAX;
Bildiğimiz gibi, aşağıdaki ifade altı kenarlı bir zar atmayı simüle eder:

```
int face = 1 + rand() % 6;
```

Bu ifade, **face** değişkenine her zaman 1 ≤ face ≤ 6 aralığında bir tamsayı değeri (rastgele) atar. Bu aralığın genişliği (yani, aralıktaki ardışık tam sayıların sayısı) 6'dır ve sayı aralıkları ise 1'dir. Önceki ifadeye atıfta bulunarak, aralığın genişliğinin kalan % operatörle (yani 6) **rand**'ı ölçeklendirmek için kullanılan sayı tarafından belirlendiğini ve aralığın başlangıç sayısının (yani 1) rand % 6'ya eklendiğini görüyoruz. Bu sonucu şu şekilde genelleştirebiliriz:

```
int n = a + rand() % b;
```

Burada,
• a kaydırma değeridir (istenen ardışık tam sayılar aralığındaki ilk sayıya eşittir) ve
• b, ölçekleme faktörüdür (istenen ardışık tamsayı aralığının genişliğine eşittir).
Alıştırmalarda, ardışık tamsayı aralıkları dışındaki değer kümelerinden rastgele tamsayılar seçeceğiz.

Rastgele Sayı Simülasyonu Vaka Çalışması: Bir Şans Oyunu

Bu bölümde, "craps" olarak bilinen popüler zar oyununu simüle ediyoruz. Oyunun kuralları basittir:
Bir oyuncu iki zar atar. Her zarın altı yüzü vardır. Bu yüzler 1, 2, 3, 4, 5 ve 6 nokta içerir. Zarlar durduktan sonra, yukarı bakan iki yüzdeki noktaların toplamı hesaplanır. İlk atışta toplam 7 veya 11 ise oyuncu kazanır. İlk atışta toplam 2, 3 veya 12 ise ("craps" denir), oyuncu kaybeder (yani "ev" kazanır). İlk atışta toplam 4, 5, 6, 8, 9 veya 10 ise, bu toplam oyuncunun "puanı" olur. İlk atışta Continue (Devam) geldi ise sonraki atışlarda oyuncu puanını atan kazanır. 7 atan kaybeder.

Şekil 5.7, zar oyununu simüle eder ve birkaç örnek uygulama gösterir.

```c
// fig05_07.c
// Simulating the game of craps.
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // contains prototype for function time

enum Status {CONTINUE, WON, LOST}; // constants represent game status

int rollDice(void); // rollDice function prototype

int main(void) {
   srand(time(NULL)); // randomize based on current time

   int myPoint = 0; // player must make this point to win
   enum Status gameStatus = CONTINUE; // may be CONTINUE, WON, or LOST
   int sum = rollDice(); // first roll of the dice

   // determine game status based on sum of dice
   switch(sum) {
      // win on first roll
      case 7: // 7 is a winner
      case 11: // 11 is a winner
         gameStatus = WON;
         break;
      // lose on first roll
      case 2: // 2 is a loser
      case 3: // 3 is a loser
      case 12: // 12 is a loser
         gameStatus = LOST;
         break;
      // remember point
      default:
         gameStatus = CONTINUE; // player should keep rolling
         myPoint = sum; // remember the point
         printf("Point is %d\n", myPoint);
         break; // optional
   }

   // while game not complete
   while (CONTINUE == gameStatus) { // player should keep rolling
      sum = rollDice(); // roll dice again

      // determine game status
      if (sum == myPoint) { // win by making point
         gameStatus = WON;
      }
      else if (7 == sum) { // lose by rolling 7
         gameStatus = LOST;
      }
   }

   // display won or lost message
   if (WON == gameStatus) { // did player win?
      puts("Player wins");
   }
   else { // player lost
      puts("Player loses");
   }
}

// roll dice, calculate sum and display results
int rollDice(void) {
   int die1 = 1 + (rand() % 6); // pick random die1 value
   int die2 = 1 + (rand() % 6); // pick random die2 value

   // display results of this roll
   printf("Player rolled %d + %d = %d\n", die1, die2, die1 + die2);
   return die1 + die2; // return sum of dice
}
```

Player wins on the first roll:

```
Player rolled 5 + 6 = 11
Player wins
```

Player wins on a subsequent roll:

```
Player rolled 4 + 1 = 5
Point is 5
Player rolled 6 + 2 = 8
Player rolled 2 + 1 = 3
Player rolled 3 + 2 = 5
Player wins
```

Player loses on the first roll:

```
Player rolled 1 + 1 = 2
Player loses
```

Player loses on a subsequent roll:

```
Player rolled 6 + 4 = 10
Point is 10
Player rolled 3 + 4 = 7
Player loses
```

Oyunda, oyuncu her atışta iki zar atmalıdır. Zarları atmak ve toplamlarını hesaplamak ve yazdırmak için rollDice fonksiyonunu tanımlarız. Fonksiyon bir kez tanımlanır, ancak iki kez çağrılır (satır 16 ve 41). Fonksiyon argüman almaz, bu nedenle parametre listesinde (satır 62) ve fonksiyon prototipinde (satır 9) **void** olduğunu belirttik. rollDice fonksiyonu, iki zarın toplamını döndürür, bu nedenle fonksiyon başlığında ve fonksiyon prototipinde bir **int** dönüş türü belirtilir.

Numaralandırmalar

Oyun makul bir şekilde başlatılır. Oyuncu, ilk atışta veya sonraki herhangi bir atışta kazanabilir veya kaybedebilir. Yeni bir tür olan **enum Status** olarak tanımlanan **gameStatus** değişkeni, mevcut durumu saklar. Satır 7, numaralandırma adı verilen yeni bir tür oluşturur. **enum** anahtar kelimesi tarafından tanıtılan bir numaralandırma, tanımlayıcılarla temsil edilen bir tamsayı sabitleri kümesidir. Numaralandırma sabitleri, programları daha okunabilir ve bakımını kolaylaştırmaya yardımcı olur. Bir **enum**'daki değerler 0 ile başlar ve 1 artırılır. 7. satırda, **CONTINUE** sabiti 0 değerine, **WON** 1 değerine ve **LOST** 2 değerine sahiptir. Bir numaralandırmadaki tanımlayıcılar benzersiz olmalıdır, ancak değerler yinelenebilir. Bir programda öne çıkmalarını sağlamak ve değişken olmadıklarını belirtmek için **enum** sabit adlarında yalnızca büyük harfler kullanılmalıdır.

Oyun kazanıldığında, **gameStatus** WON olarak ayarlanır. Oyun kaybedildiğinde, **gameStatus** LOST olarak ayarlanır. Aksi takdirde **gameStatus**, CONTINUE olarak ayarlanır ve oyun devam eder. Oyun ilk atıştan sonra biterse, **gameStatus** DEVAM etmez, bu nedenle program 53-58. satırlardaki **if…else** ifadesine geçer;

Oyun Sonraki Bir Atışta Sona Eriyor

İlk atıştan sonra eğer oyun bitmemişse toplam **myPoint**'e kaydedilir. **GameStatus** CONTINUE olduğundan yürütme **while** deyimi ile devam eder. Süre boyunca her seferinde, yeni bir toplam üretmek için **rollDice** çağrılır:

• Toplam, **myPoint** ile eşleşirse, **gameStatus** WON olarak ayarlanır, **while** döngüsü sona erer, **if…else** ifadesi "Oyuncu kazanır" yazar ve yürütme sona erer.

• Toplam 7 ise (satır 47), **gameStatus** LOST olarak ayarlanır, **while** döngüsü sona erer, **if…else** deyimi "Oyuncu kaybeder" yazar ve yürütme sona erer.

Kontrol Yapısı

Programın kontrol mimarisine dikkat edin. **main** ve **rollDice** olmak üzere iki fonksiyonu ve anahtarı, **while** ve iç içe **if…else** ifadelerini kullandık.

Depolama Sınıfları

Bölüm 2-4'te, değişken adları için tanımlayıcılar kullandık. Değişkenlerin öznitelikleri arasında ad, tür, boyut ve değer bulunur. Bu bölümde, tanımlayıcıları kullanıcı tanımlı fonksiyonlar için adlar olarak da kullanıyoruz. Aslında, bir programdaki her tanımlayıcının, depolama sınıfı, depolama süresi, kapsam ve bağlantı dahil olmak üzere başka öznitelikleri vardır. C, auto, register, extern ve static depolama sınıfı tanımlayıcılarını verir. Bir depolama sınıfı, bir tanımlayıcının depolama süresini, kapsamını ve bağlantısını belirler. Depolama süresi, bir tanımlayıcının bellekte bulunduğu süredir. Bazıları kısaca var olur, bazıları tekrar tekrar yaratılır ve yok edilir ve diğerleri tüm program yürütmesi için vardır.

Kapsam, bir programın bir tanımlayıcıya nerede başvurabileceğini belirler. Bazılarına bir program boyunca, bazılarına da programın yalnızca bölümlerinden başvurulabilir. Çok kaynaklı dosya programı için, tanımlayıcının bağlantısı, tanımlayıcının yalnızca geçerli kaynak dosyada mı yoksa uygun bildirimlere sahip herhangi bir kaynak dosyada mı bilindiğini belirler. Bu bölüm, depolama sınıflarını ve depolama süresini tartışacaktır.

Yerel Değişkenler ve Otomatik Depolama Süresi

Depolama sınıfı belirleyicileri, otomatik depolama süresi ve statik depolama süresi arasında bölünmüştür. **auto** anahtar sözcüğü, bir değişkenin otomatik depolama süresine sahip olduğunu bildirir. Bu tür değişkenler, program kontrolü tanımlandıkları bloğa girdiğinde oluşturulur. Blok aktifken var olurlar ve program kontrolü bloktan çıktığında yok edilirler.

Yalnızca değişkenler otomatik depolama süresine sahip olabilir. Bir fonksiyonun yerel değişkenleri (parametre listesinde veya fonksiyon gövdesinde bildirilenler) varsayılan olarak otomatik depolama süresine sahiptir, bu nedenle **auto** anahtar sözcüğü nadiren kullanılır. Otomatik depolama süresi, belleği korumanın bir yoludur çünkü yerel değişkenler yalnızca ihtiyaç duyulduğunda bulunur. Otomatik depolama süresine sahip değişkenlere yalnızca yerel değişkenler olarak atıfta bulunacağız.

Statik Depolama Sınıfı

extern ve **static** anahtar sözcükleri, statik depolama süresine sahip değişkenler ve fonksiyonlar için tanımlayıcıları bildirir. Statik depolama süresinin tanımlayıcıları, programın yürütülmeye başladığı andan sona erene kadar mevcuttur. Statik değişkenler için, program yürütülmeye başlamadan önce depolama yalnızca bir kez tahsis edilir ve başlatılır. Fonksiyonlar için, program yürütmeye başladığında fonksiyonun adı bulunur. Ancak, bu adlar programın yürütülmesinin başlangıcından itibaren mevcut olsalar da, her zaman erişilebilir değildirler. Depolama süresi ve kapsamı (bir adın kullanılabileceği durumlarda) ayrı konulardır.

Statik depolama süresine sahip birkaç tanımlayıcı türü vardır: harici tanımlayıcılar (global değişkenler ve fonksiyon adları gibi) ve depolama sınıfı belirticisi **static** ile bildirilen yerel değişkenler. Global değişkenler ve fonksiyon adları, varsayılan olarak **extern** depolama sınıfına sahiptir. Global değişkenler, değişken bildirimlerini herhangi bir fonksiyon tanımının dışına yerleştirerek oluşturulur. Program yürütme boyunca değerlerini korurlar.

Global değişkenlere ve fonksiyonlara, dosyadaki bildirimlerini veya tanımlarını izleyen herhangi bir fonksiyon tarafından başvurulabilir. Fonksiyon prototiplerini kullanmamızın bir nedeni de budur; printf'i çağıran bir programa <**stdio.h**>'yi dahil ettiğimizde, dosyanın geri kalanının **printf** adını bilmesini sağlamak için fonksiyon prototipi dosyamızın başına yerleştirilir. Bir değişkeni yerel yerine global olarak tanımlamak, değişkene erişmesi gerekmeyen bir fonksiyon yanlışlıkla değişkeni değiştirdiğinde istenmeyen yan etkilerin ortaya çıkmasına izin verir. Global olarak, benzersiz performans gereksinimleri olan durumlar dışında global değişkenlerden kaçınmalısınız. Yalnızca belirli bir fonksiyonda kullanılan değişkenler, o fonksiyonda yerel değişkenler olarak tanımlanmalıdır.

Yerel statik değişkenler hala yalnızca tanımlandıkları fonksiyonda bilinirler ve fonksiyon geri döndüğünde değerlerini korurlar. Fonksiyonun bir sonraki çağrılışında, statik yerel değişken, fonksiyondan en son çıkıldığında sahip olduğu değeri içerir.

Kapsam Kuralları
Bir tanımlayıcının kapsamı, programın tanımlayıcıya başvurulabilen bölümüdür. Örneğin, bir bloktaki yerel bir değişkene, yalnızca o bloktaki veya o blok içinde iç içe geçmiş bloklardaki tanımı izlenerek başvurulabilir. Dört tanımlayıcı kapsam; fonksiyon kapsamı, dosya kapsamı, blok kapsamı ve fonksiyon prototipi kapsamıdır.

Fonksiyon Kapsamı
Etiketler, start: gibi iki nokta üst üste ile takip edilen tanımlayıcılardır. Etiketler, fonksiyon kapsamına sahip tek tanımlayıcılardır. Etiketler, göründükleri fonksiyonda herhangi bir yerde kullanılabilir, ancak fonksiyon gövdesi dışında referans alınamaz. Etiketler, switch ifadelerinde (case etiketleri olarak). Etiketler, tanımlandıkları fonksiyonda gizlidir. Bu bilgi gizleme, iyi yazılım mühendisliğinin temel bir ilkesi olan en az ayrıcalık ilkesini uygulamanın bir yoludur. Bir uygulama bağlamında ilke, koda yalnızca belirlenen görevi yerine getirmek için ihtiyaç duyduğu kadar ayrıcalık ve erişim verilmesi gerektiğini, daha fazlasının verilmemesi gerektiğini belirtir.

Dosya Kapsamı
Herhangi bir fonksiyonun dışında bildirilen bir tanımlayıcı, dosya kapsamına sahiptir. Böyle bir tanımlayıcı, tanımlayıcının bildirildiği andan dosyanın sonuna kadar tüm fonksiyonlarda "bilinir" (yani erişilebilir). Bir fonksiyonun dışına yerleştirilen global değişkenler, fonksiyon tanımları ve fonksiyon prototiplerinin tümü dosya kapsamına sahiptir.

Blok Kapsamı
Bir blok içinde tanımlanan tanımlayıcıların blok kapsamı vardır. Blok kapsamı, bloğun sonlandırıcı sağ parantezinde (}) sona erer. Bir fonksiyonun başında tanımlanan yerel değişkenler, fonksiyon tarafından yerel değişkenler olarak kabul edilen fonksiyon parametrelerinde olduğu gibi blok kapsamına sahiptir. Herhangi bir blok değişken tanımları içerebilir. Bloklar iç içe geçtiğinde ve bir dış bloğun tanımlayıcısı, bir iç bloğun tanımlayıcısıyla aynı ada sahip olduğunda, dış bloğun tanımlayıcısı, iç blok sona erene kadar gizlenir. İç blokta yürütülürken, iç blok, çevreleyen bloğun aynı adlı tanımlayıcısının değerini değil, yerel tanımlayıcısının değerini görür. Bu nedenle, genellikle dış kapsamlarda adları gizleyen değişken adlarından kaçınmalısınız. Statik olarak bildirilen yerel değişkenler, program başlangıcından önce var olmalarına rağmen hala blok kapsamına sahiptir. Bu nedenle, depolama süresi bir tanımlayıcının kapsamını etkilemez.

Fonksiyon-Prototip Kapsamı
Fonksiyon prototipi kapsamına sahip tek tanımlayıcılar, bir fonksiyon prototipinin parametre listesinde kullanılanlardır. Daha önce bahsedildiği gibi, fonksiyon prototipleri parametre listesinde ad gerektirmez; yalnızca türler gereklidir. Bir fonksiyon prototipinin parametre listesinde bir ad kullanılırsa, derleyici bunu yok sayar. Bir fonksiyon prototipinde kullanılan tanımlayıcılar, programın herhangi bir yerinde belirsizlik olmaksızın yeniden kullanılabilir.

Kapsam Belirleme Örneği
Şekil 5.8, global değişkenler, yerel değişkenler ve statik yerel değişkenlerle ilgili kapsam belirleme sorunlarını göstermektedir. Global bir x değişkeni tanımlanır ve 1 olarak başlatılır (satır 9). Bu global değişken, x adlı bir değişkenin tanımlandığı herhangi bir blokta (veya fonksiyonda) gizlidir. **main** olarak, yerel bir x değişkeni tanımlanır ve 5 olarak başlatılır (satır 12). Bu değişken daha sonra global x'in main'de gizli olduğunu göstermek için yazdırılır. Daha sonra, başka bir yerel değişken x ile 7'ye (satır 17) başlatılan main'de yeni bir blok tanımlanır. Bu değişken, main'in dış bloğunda x'i gizlediğini göstermek için yazdırılır. 7 değerine sahip x değişkeni, bloktan çıkıldığında otomatik olarak yok edilir ve main'in dış bloğundaki yerel x değişkeni, artık gizli olmadığını göstermek için tekrar yazdırılır.

```c
// fig05_08.c
// Scoping.
#include <stdio.h>

void useLocal(void); // function prototype
void useStaticLocal(void); // function prototype
void useGlobal(void); // function prototype

int x = 1; // global variable

int main(void) {
   int x = 5; // local variable to main

   printf("local x in outer scope of main is %d\n", x);

   { // start new scope
      int x = 7; // local variable to new scope

      printf("local x in inner scope of main is %d\n", x);
   } // end new scope

   printf("local x in outer scope of main is %d\n", x);

   useLocal(); // useLocal has automatic local x
   useStaticLocal(); // useStaticLocal has static local x
   useGlobal(); // useGlobal uses global x
   useLocal(); // useLocal reinitializes automatic local x
   useStaticLocal(); // static local x retains its prior value
   useGlobal(); // global x also retains its value

   printf("\nlocal x in main is %d\n", x);
}

// useLocal reinitializes local variable x during each call
void useLocal(void) {
   int x = 25; // initialized each time useLocal is called

   printf("\nlocal x in useLocal is %d after entering useLocal\n", x);
   ++x;
   printf("local x in useLocal is %d before exiting useLocal\n", x);
}

// useStaticLocal initializes static local variable x only the first time
// the function is called; value of x is saved between calls to this
// function
void useStaticLocal(void) {
   static int x = 50; // initialized once

   printf("\nlocal static x is %d on entering useStaticLocal\n", x);
   ++x;
   printf("local static x is %d on exiting useStaticLocal\n", x);
}

// function useGlobal modifies global variable x during each call
void useGlobal(void) {
   printf("\nglobal x is %d on entering useGlobal\n", x);
   x *= 10;
   printf("global x is %d on exiting useGlobal\n", x);
}
```

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Program, her biri argüman almayan ve hiçbir şey döndürmeyen üç fonksiyon tanımlar. useLocal fonksiyonu, yerel bir x değişkeni tanımlar ve onu 25 olarak başlatır (satır 36). useLocal fonksiyon çağrıldığında, değişken yazdırılır, artırılır ve fonksiyondan çıkmadan önce tekrar yazdırılır. Bu fonksiyon her çağrıldığında, x yerel değişkeni 25 olarak yeniden başlatılır.
useStaticLocal fonksiyonu x statik değişkenini tanımlar ve 47. satırda bunu 50 olarak başlatır (statik değişkenler için depolamanın program yürütülmeye başlamadan önce yalnızca bir kez tahsis edildiğini ve başlatıldığını hatırlayın). Statik olarak bildirilen yerel değişkenler, kapsam dışında olsalar bile değerlerini korur. useStaticLocal çağrıldığında, x yazdırılır, artırılır ve fonksiyondan çıkmadan önce tekrar yazdırılır. Bu fonksiyona yapılan bir sonraki çağrıda, x statik yerel değişkeni önceden artırılan 51 değerini içerecektir.
useGlobal fonksiyonu herhangi bir değişken tanımlamaz, bu nedenle x değişkenine başvurduğunda global x (satır 9) kullanılır. useGlobal çağrıldığında, global değişken yazdırılır, 10 ile çarpılır ve fonksiyondan çıkmadan önce tekrar yazdırılır. useGlobal fonksiyonu bir sonraki çağrıldığında, genel değişkenin değiştirilmiş değeri hala 10'dur.
Son olarak, program, fonksiyonların tümü diğer kapsamlardaki değişkenlere atıfta bulunduğundan, hiçbir fonksiyonun değiştirilmiş x'in değerini çağırmadığını göstermek için yerel x değişkenini ana satırda yeniden yazdırır (satır 31).