C Structures (Yapılar)

C standardında bazen kümeler olarak adlandırılan yapılar, tek bir ad altında ilgili değişkenlerin birleşimidir. Yapılar, yalnızca aynı veri türündeki öğeleri içeren dizilerin aksine, birçok farklı veri türünden değişkenler içerebilir. Yapılar genellikle dosyalarda saklanacak kayıtları tanımlamak için kullanılır. İşaretçiler ve yapılar kullanılarak; bağlantılı listeler, sıralar, yığınlar ve ağaçlar gibi daha karmaşık veri yapılarının oluşturulması kolaylaştırılır.

Yapılar, türetilmiş veri türleridir; diğer türlerdeki nesneler kullanılarak oluşturulurlar. Aşağıdaki yapı tanımını göz önünde bulunduralım:

```
struct card {
        char *face;
        char *suit;
};
```

Anahtar sözcük “struct”, bir yapı tanımı sunar. Tanımlayıcı card, “structure tag” olarak adlandıran ve yapı türündeki değişkenleri bildirmek için kullanılan yapı etiketidir. Yapı tanımının parantezleri içinde bildirilen değişkenler, yapının üyeleridir. Aynı yapı türünün üyeleri benzersiz adlara sahip olmalıdır, ancak iki farklı yapı türü aynı adı taşıyan üyeleri çakışma olmadan içerebilir. Her yapı tanımı noktalı virgülle bitmelidir.

"struct card" tanımı, her biri char * türünde olan üyeleri içerir. Yapı üyeleri, veri türlerinin değişkenleri (örneğin, int, float, vb.) veya diziler ve diğer yapılar gibi kümeler olabilir. Bir dizinin her elemanı aynı tipte olmalıdır. Bununla birlikte, yapı elemanları farklı türlerde olabilir. Örneğin, aşağıdaki yapı, bir çalışanın adı ve soyadı için karakter dizisi üyeleri, çalışanın yaşı için “unsigned int”, çalışanın cinsiyeti için 'M' veya 'F' içeren bir “char” ve çalışanın saatlik maaşını içeren “double” şöyle oluşturulabilir:

```
struct employee {
        char firstName[20];
        char lastName[20];
        unsigned int age;
        char gender;
        double hourlySalary;
};
```

Yapı tipindeki bir değişken, aynı yapı tipinin tanımında bildirilemez. Bununla birlikte, bu yapı türüne bir işaretçi dahil edilebilir. Aşağıdaki örnek için, employee2 yapısında: teamLeaderPtr bir işaretçi olduğundan (struct employee2 yazmak için), tanımda buna izin verilir. Aynı yapı tipine işaret eden bir yapıya, kendine referanslı yapı denir.

```
struct employee2 {
        char firstName[20];
        char lastName[20];
        unsigned int age;
        char gender;
        double hourlySalary;
        struct employee2 teamLeader; // ERROR
        struct employee2 *teamLeaderPtr; // pointer
};
```

Yapı tanımları bellekte yer ayırmaz; bunun yerine her tanım, değişkenleri tanımlamak için kullanılan yeni bir veri türü oluşturur; örneğin, söz konusu yapının örneklerinin nasıl oluşturulacağına ilişkin bir plan şöyle verilebilir;

struct card

aCard, deck[52], *cardPtr;

Yapı değişkenleri, diğer türlerdeki değişkenler gibi tanımlanır.

Tanım, aCard'ı struct card türünde bir değişken olarak bildirir,

deck değişkeni struct card türünde 52 elemanlı bir dizi olarak bildirir ve

cardPtr'yi struct card için bir işaretçi olarak bildirir.

Önceki ifadeden sonra, aCard adlı bir yapı nesnesi, deck dizisindeki 52 elemanlı bir dizi ve yapı kartı türünde başlatılmamış bir işaretçi için bellek ayırılır. Belirli bir yapı tipinin değişkenleri, yapı tanımının kapatma ayracı ile yapı tanımını sonlandıran noktalı virgül arasına değişken adlarının virgülle ayrılmış bir listesi yerleştirilerek de bildirilebilir. Örneğin, önceki tanım yapı kartı tanımına şu şekilde dahil edilmiş olabilir:

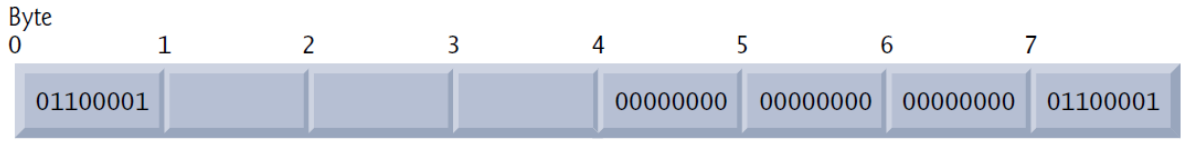
```
struct card {
        char *face;
        char *suit;
} aCard, deck[52], *cardPtr;
```

Yapılar, == ve != operatörleri kullanılarak karşılaştırılamaz, çünkü yapı üyeleri mutlaka ardışık bellek baytlarında saklanmaz. Aşağıda verilen example Yapı örneğinin sample1 ve sample2 tipinin bildirildiği aşağıdaki yapı tanımını göz önünde bulunduralım:

```
struct example {
        char c;
        int i;
} sample1, sample2;
```

4 baytlık sözcüklere sahip bir bilgisayar, yapı örneğinin her bir üyesinin bir sözcük sınırında, yani bir sözcüğün başında hizalanmasını gerektirebilir; bu, bilgisayara bağlı olarak değişebilir. Şekil 10.1, 'a' karakteri ve 97 tamsayısının atandığı yapı örneği türünde bir değişken için bir örnek depolama hizalamasını göstermektedir (değerlerin bit temsilleri gösterilmektedir).

Üyeler kelime sınırlarından başlayarak saklanırsa, yapı örneği tipindeki değişkenler için depolamada üç baytlık bir boşluk (şekilde bayt 1-3) oluşur. Üç baytlık boşluktaki değer tanımsızdır. Sample1 ve sample2'nin üye değerleri aslında eşit olsa bile, tanımlanmamış üç baytlık boşlukların aynı değerleri içermesi muhtemel olmadığından, yapıların mutlaka eşit olması gerekmez.



Şekil 10.1 Bellekte tanımsız bir alanı gösteren yapı örneği türünde bir değişken için olası depolama hizalaması.

Yapılar, dizilerde olduğu gibi başlatıcı listeleri kullanılarak başlatılabilir. Bir yapıyı başlatmak için, tanımdaki değişken adını bir eşittir işareti ve parantez içine alınmış, virgülle ayrılmış başlatıcılar kullanılır. Örneğin;

```
struct card aCard = { "Three", "Hearts" };
```

Bu deyim, struct card (Bölüm 10.2'de tanımlandığı gibi) türünde olacak bir aCard değişkeni yaratır ve "face" değişkenini "Three" olarak "suit" değişkeni "Hearts" olarak başlatır. Listede yapıdaki değişkenlerden daha az başlatıcı varsa, geri kalan değişkenler otomatik olarak 0'a (veya üye bir işaretçiyse NULL) ile başlatılır. Bir fonksiyon tanımı dışında (yani harici olarak) tanımlanan yapı değişkenleri, harici tanımda açıkça başlatılmamışlarsa 0 veya NULL olarak başlatılır. Yapı değişkenleri aynı türde bir yapı değişkeni atanarak veya yapının bireysel üyelerine değerler atanarak atama ifadelerinde de başlatılabilir.

Yapıların üyelerine erişmek için iki operatör kullanılır: Bunlar, nokta operatörü olarak da adlandırılan yapı elemanı operatörü (.) ve ok operatörü olarak da adlandırılan yapı işaretçi operatörü (->). Yapı üyesi operatörü, yapı değişkeni adı aracılığıyla bir yapı üyesine erişir. Örneğin, aCard yapı değişkeninin üye takımını yazdırmak için aşağıdaki ifade kullanılabilir;

```
printf("%s", aCard.suit); // displays Hearts
```

Arada boşluk olmayan bir eksi (-) işareti ve büyüktür (>) işaretinden oluşan yapı işaretçisi operatörü, yapıya bir işaretçi aracılığıyla bir yapı üyesine erişir. cardPtr işaretçisinin yapı kartını işaret etmek üzere bildirildiğini ve aCard yapısının adresinin cardPtr'ye atandığını varsayalım. CardPtr işaretçisi ile aCard yapısının üye takımını yazdırmak için aşağıdaki ifadeyi kullanın;

```
printf("%s", cardPtr->suit); // displays Hearts
```

cardPtr->suit ifadesi, işaretçiyi referanssız bırakan ve yapı üye operatörünü kullanarak "suit" üyesine erişen (*cardPtr).suit ifadesine eşdeğerdir. Burada parantezler gereklidir, çünkü yapı üyesi operatörü(.), işaretçi referans kaldırma operatöründen (*) daha yüksek önceliğe sahiptir. Dizi indeksleme için kullanılan parantezler (fonksiyonları çağırmak için) ve parantezler ([]) ile birlikte yapı işaretçisi operatörü ve yapı üyesi operatörü, en yüksek operatör önceliğine sahiptir ve soldan sağa ilişkilendirir.

Şekil 10.2'deki program, yapı elemanı ve yapı işaretçisi operatörlerinin kullanımını göstermektedir. Yapı üyesi operatörünü kullanarak, aCard yapısının üyelerine sırasıyla "Ace" ve "Spades" değerleri atanır (satır 17 ve 18). Sonraki işaretçi kullanarak cardPtr işaretçisine aCard yapısının adresi atanır (satır 20). printf fonksiyonu ile , sırasıyla aCard değişken adına sahip yapı kullanılarak, , cardPtr işaretçisi kullanılarak ve işaretçi adresi kullanılarak "Ace" ve "Spades" yazdırılır (satır 22–24).

```c
// Fig. 10.2: fig10_02.c
// Structure member operator and
// structure pointer operator

#include <stdio.h>

// card structure definition
struct card {
   char *face; // define pointer face
   char *suit; // define pointer suit
};

int main(void)
{
   struct card aCard; // define one struct card variable

   // place strings into aCard
   aCard.face = "Ace";
   aCard.suit = "Spades";

   struct card *cardPtr = &aCard; // assign address of aCard to cardPtr

   printf("%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
      cardPtr->face, " of ", cardPtr->suit,
      (*cardPtr).face, " of ", (*cardPtr).suit);
}
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Yapılar, fonksiyonlara şu şekilde geçirilebilir:

• Bireysel yapı elemanlarının geçirilmesi.

• bütün bir yapıyı geçirmek.

• bir yapıya işaretçi geçirmek.

Yapılar veya tek tek yapı üyeleri bir fonksiyona geçirildiğinde, bunlar değere göre geçirilir. Bu nedenle, çağıranın yapı üyeleri, çağrılan fonksiyon tarafından değiştirilemez. Bir yapıyı referans olarak geçirmek için, yapı değişkeninin adresi iletilir. Diğer tüm diziler gibi yapı dizileri de otomatik olarak referansa göre iletilir.

“typedef” anahtar sözcüğü, önceden tanımlanmış veri türleri için eşanlamlılar (veya takma adlar) oluşturmak için bir mekanizma sağlar. Yapı türlerinin adları, daha kısa tür adları oluşturmak için genellikle “typedef” ile tanımlanır. Örneğin, aşağıdaki ifade, yeni tür adı Card'ı, yapı türü card ile eşanlamlı olarak tanımlar.

```
typedef struct card Card;
```

C programcıları genellikle bir yapı tipini tanımlamak için typedef'i kullanır, bu nedenle bir yapı etiketi gerekli değildir. Örneğin, aşağıdaki tanım, ayrı bir "typedef" ifadesine ihtiyaç duymadan Card yapı tipini oluşturur.

```
typedef struct {
char *face;
char *suit;
} Card;
```

Cart artık cart yapı tipindeki değişkenleri bildirmek için kullanılabilir. Aşağıdaki bildirim, 52 elemanlı Cart yapısından oluşan bir dizi bildirir.

```
Card deck[52];
```

"typedef" ile yeni bir ad oluşturmak, yeni bir tür oluşturmaz; typedef basitçe, mevcut bir tür adı için takma ad olarak kullanılabilecek yeni bir tür adı oluşturur. Anlamlı bir ad, programın kolay anlaşılmasına yardımcı olur. Örneğin bir önceki bildirimi okuduğumuz zaman “deck'in 52 Karttan oluşan bir dizi olduğunu” biliyoruz.

Şekil 10.3'teki program, kart karıştırma ve dağıtma simülasyonuna dayanmaktadır. Program, kart destesini bir dizi yapı olarak temsil eder ve yüksek performanslı karıştırma ve dağıtma algoritmaları kullanır. Program çıktısı Şekil 10.4'te gösterilmiştir.

```c
// Fig. 10.3: fig10_03.c
// Card shuffling and dealing program using structures
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define CARDS 52
#define FACES 13

// card structure definition
struct card {
   const char *face; // define pointer face
   const char *suit; // define pointer suit
};

typedef struct card Card; // new type name for struct card

// prototypes
void fillDeck(Card * const wDeck, const char * wFace[],
   const char * wSuit[]);
void shuffle(Card * const wDeck);
void deal(const Card * const wDeck);


int main(void)
{
   Card deck[CARDS]; // define array of Cards

   // initialize array of pointers
   const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
      "Six", "Seven", "Eight", "Nine", "Ten",
      "Jack", "Queen", "King"};

   // initialize array of pointers
   const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};

   srand(time(NULL)); // randomize

   fillDeck(deck, face, suit); // load the deck with Cards
   shuffle(deck); // put Cards in random order
   deal(deck); // deal all 52 Cards
}


// place strings into Card structures
void fillDeck(Card * const wDeck, const char * wFace[],
   const char * wSuit[])
{
   // loop through wDeck
   for (size_t i = 0; i < CARDS; ++i) {
      wDeck[i].face = wFace[i % FACES];
```

```
50          wDeck[i].suit = wSuit[i / FACES];
51       }
52    }
53
54    // shuffle cards
55    void shuffle(Card * const wDeck)
56    {
57       // loop through wDeck randomly swapping Cards
58       for (size_t i = 0; i < CARDS; ++i) {
59          size_t j = rand() % CARDS;
60          Card temp = wDeck[i];
61          wDeck[i] = wDeck[j];
62          wDeck[j] = temp;
63       }
64    }
65
66    // deal cards
67    void deal(const Card * const wDeck)
68    {
69       // loop through wDeck
70       for (size_t i = 0; i < CARDS; ++i) {
71          printf("%5s of %-8s%s", wDeck[i].face  , wDeck[i].suit  ,
72             (i + 1) % 4 ? "  " : "\n");
73       }
74    }
```

```
Three of Hearts     Jack of Clubs     Three of Spades      Six of Diamonds
 Five of Hearts    Eight of Spades    Three of Clubs     Deuce of Spades
 Jack of Spades     Four of Hearts    Deuce of Hearts      Six of Clubs
Queen of Clubs     Three of Diamonds  Eight of Diamonds   King of Clubs
 King of Hearts    Eight of Hearts    Queen of Hearts    Seven of Clubs
Seven of Diamonds   Nine of Spades     Five of Clubs     Eight of Clubs
  Six of Hearts    Deuce of Diamonds   Five of Spades     Four of Clubs
Deuce of Clubs      Nine of Hearts    Seven of Hearts     Four of Spades
  Ten of Spades     King of Diamonds    Ten of Hearts     Jack of Diamonds
 Four of Diamonds    Six of Spades     Five of Diamonds    Ace of Diamonds
  Ace of Clubs      Jack of Hearts      Ten of Clubs     Queen of Diamonds
  Ace of Hearts      Ten of Diamonds   Nine of Clubs      King of Spades
  Ace of Spades     Nine of Diamonds  Seven of Spades    Queen of Spades
```

Fig. 10.4 | Output for the high-performance card shuffling and dealing simulation.

Programda, fillDeck fonksiyonu (satır 44–52), her türden "As"tan "Papaz"a kadar Card dizisini başlatır. Card dizisi, yüksek performanslı karıştırma algoritmasının uygulandığı karıştırma fonksiyonuna (satır 55-64) geçirilir.

Shuffle fonksiyonu, bağımsız değişken olarak 52 elemanlı Card dizisini alır. Fonksiyon, 52 Kart (satır 58–63) arasında döngü yapar. Her Kart için 0 ile 51 arasında rastgele bir sayı seçilir. Ardından, mevcut Kart ve rastgele seçilen Kart dizide değiştirilir (60-62. satırlar). Tüm dizinin tek geçişinde toplam 52 takas yapılır ve Kart dizisi karıştırılır!