İçindekiler

Fonksiyonlar

Bölüm 5

Adres Değişkenleri

Bölüm 6
Adres Değişkenleri ve Çok-Boyutlu Diziler

Bölüm 7
Dinamik Bellek Yönetimi

Bölüm 8
Yapı Değişkenleri

Bölüm 9
Giriş/Çıkış İşlemleri

■

BÖLÜM 1: **Giriş ve Temel Kavramlar**

Bu bölümde C programlama dilinin özelliklerine kısaca değinildikten sonra basit bir C programı oluşturulur. ■

1.1 Bir C Programının Oluşturulması

C programlama dili, Bell Laboratuarlarında 1972 yılında Dennis Ritchie tarafından geliştirilmiştir. C dili verimli, güçlü ve taşınabilir bir programlama dilidir. Metin işlemci, veri tabanı, grafik programı yada bilimsel çalışmalarda kullanılan uygulama programları, derleyiciler ve UNIX gibi işletim sistemleri geliştirmek için kullanılan genel amaçlı ve orta seviyeli bir dildir.

Taşınabilirlik (*portability*), aynı programın farklı donanımlarda ve işletim sistemleri altında işletilebilirliğini ifade eder. Bir dilin *verimliliği* (*efficiency*), hızlı fakat fazla yer kaplamayan yazılımlar geliştirmeye olanak sağladığı oranda artar. C programlama dili verimli, basit fakat güçlü yapısından ve bu dilde geliştirilen uygulamaların taşınabilir olmasından dolayı programcılar, mühendisler ve sistem programcıları tarafından tercih edilir ve yaygın olarak kullanılır.

C dilinde pek çok işlem standart C kütüphanesinde bulunan fonksiyonlar kullanılarak gerçekleştirilir; örneğin giriş/çıkış işlemleri. C dili, özel kütüphanelerin geliştirilerek programlarda kullanılabilmesine olanak sağlar. Programlarda, gerekli olduğunda bazı özel işlemler, yazılım firmaları tarafından geliştirilmiş kütüphaneler kullanılarak gerçekleştirilebilir.

Bir programlama dilinden bahsedilirken, hangi *ortam*da kullanıldığı belirtilir. Bir programlama dilinin kullanıldığı ortam donanım, işletim sistemi ve kaynak kodların (işletim sistemi altında programlama dili ile oluşturulan) çalıştırılabilir kodlara çevrilmesini sağlayan derleyici programdan oluşur.

Taşınabilirlik özelliğinden dolayı C ile kodlanmış programlar pek çok ortamda işletilebilir. Mevcut ortamda kullanılan C derleyicisinin yapısı öğrenildikten sonra diğer ortamlarda da benzer yazılımlar kolaylıkla geliştirilebilir.

Kitapta yer alan örnek programlar aşağıda görüldüğü gibi disket sembolü () ve bir örnek numarası ile başlar. Metin içinde programa değinilirken genel olarak program ismi yerine örnek numarası kullanılır. Fakat örnek programlar diskette program ismi ile bulunur.

Programlardaki ekran çıktıları monitör sembolü ve "çıktı" yazısı sonrası verilir ( **Çıktı**). Ayrıca eğer program klavyeden bilgi okuyor ise girilen bilgiler klavye işareti ve "bilgi girişi" yazısı sonrasında verilir ( **Bilgi Girişi**).

Aşağıda ekrana "C programlama dili" dizgisini yazan ve bir kaç satırdan oluşan basit bir C programı verilmiştir. Herhangi bir C programının oluşturulması, bir editör (*text editor*) program kullanılarak kaynak kodun (*source code*) yazılması ve bir dosyaya saklanması ile başlar. Bu dosya, C *kaynak dosyası* (*source file*) olarak adlandırılır.

Örnek 1.1-1 c.c programı.

```
#include <stdio.h>

int main(void)
{
   printf( "C programlama dili\n" );
   return 0;
}
```

Çıktı

```
C programlama dili
```

Kaynak dosya isminin izin verilen uzunluğu işletim sistemine göre değişir. Örneğin DOS işletim sisteminde en fazla 8 karakterden oluşur ve dosya isminde bazı noktalama işaretleri bulunamaz. C kaynak dosyaları, .c uzantılı dosyalardır.

C kaynak dosyası yada dosyaları, çalıştırılabilir program haline iki aşamada getirilir:

- Derleme (*compilation*): Bu aşamada derleyici program, C kaynak dosyalarını object dosyalara çevirir. Bir object dosyası (*object-code file*) binary kodlardan oluşur, fakat henüz çalıştırılabilir durumda değildir. Object dosyalar DOS işletim sisteminde .obj uzantılı, UNIX işletim sisteminde .o uzantılı dosyalardır.
- Bağlama (*linking*): Bu aşamada, *linker* olarak adlandırılan program (yada *linkage editor*), derleme sırasında oluşturulan object dosyalarını standart kütüphaneler (programlarda kullanılan standart kütüphane fonksiyonlarına ait kodları içeren (DOS işletim sisteminde .lib uzantılı, UNIX işletim sisteminde .a dosyalar), programcı tarafından belirtilen diğer object dosyalar ve kütüphaneler ile birleştirerek çalıştırılabilir program dosyasını oluşturur. Bu dosya (DOS işletim sisteminde .exe uzantılı, UNIX işletim sisteminde ise .out uzantılı yada uzantısızdır), işletim sistemi altında çalıştırılmaya hazırdır.

Derleyiciler, derleme sırasında kaynak kod içinde herhangi bir yazım hatasına (*syntax error*) yada programlama hatasına rastladığında çalışmasını durdurur ve ekrana hatayı anlatan ve kaynak program içinde hangi satırda bulunduğunu gösteren hata mesajını listeler. Bazı hatalar yukarıda anlatılan bağlama işlemi sırasında oluşur ve bağlama hatası (*linkage error*) olarak adlandırılır. Bu durumda, linker programı tarafından ekrana

hata mesajı listelenecektir. Hatasız olarak derlenen ve bağlanan bir programın çalışması sırasında oluşan hatalar ise çalışma zamanı (*run-time error*) hatalarıdır. ■

1.2 C Atomları

Atom (*token* olarakta adlandırılır), programlama dili için anlamı olan tek bir karakter yada karakterler (boşluk yada bazı özel karakterlerle bölünmemiş) bütünüdür. Bir C programı atomlardan oluşur. Derleyici çalışmaya başladığında kaynak kodu ilk olarak atomlara ayırır. Bir program içinde 6 çeşit atom bulunur: isimler (*identifiers*), anahtar sözcükler (*keywords*), sabitler (*constants*), dizgi sabitleri (*string literals*), operatörler (*operators*) ve ayraçlar (*separators*).

Bir isim, harf ve rakam dizisidir. C dilinde çeşitli tiplerde sabitler bulunur: tamsayı sabitler, karakter sabitleri, kayan-noktalı sabitler ve enum sabitleri. Dizgi sabiti (*string literal* yada *string constant*) çift tırnaklar arasında bulunan karakter dizisinden oluşur ("..."). Ayraçlar, C dilinin noktalama işaretlerdir: **, ; { } = ( ) :**

Operatörler ve diğer C atomları izleyen bölümde ayrıntılı olarak incelenecektir. Aşağıdaki isimler, C dilinde ayrılmış sözcüklerdir ve program içinde değişken ismi olarak kullanılamaz (*keywords*):

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

Derleyici tarafından program içindeki düz boşluklar (*blank space*), yatay ve düşey tab boşlukları (HT ve VT), yeni-satır karakterleri (NL, *newlines*), form-feed karakterleri (FF) (bunların tamamı boşluk karakteri olarak adlandırılır: *white space*) ve açıklama satırları (/* ile başlayan ve */ ile sonlanan) atom ayracı (*token delimiter*) olarak görülür. Kaynak kod içinde birbirini izleyen isimler, anahtar sözcükler ve sabitler bitişik yazılamazlar; atom ayracı ile birbirlerinden ayrılmalıdırlar. Bunun dışında program içinde C atomları arasında istenildiği kadar boşluk karakteri yada açıklama bulunabilir yada atomlar bitişik yazılabilir.

Buna göre c.c programı aşağıdaki şekillerde yazılabilir:

Her satırda bir atom bulunacak şekilde:

ATOM:	#include <stdio.h>
anahtar sözcük	int
anahtar sözcük	main

ayraç	(
anahtar sözcük	void
ayraç	)
ayraç	{
isim	printf
ayraç	(
dizgi sabiti	"C programlama dili\n"
ayraç	)
ayraç	;
anahtar sözcük	return
sabit	0
ayraç	;
ayraç	}

Tüm atomlar aynı satırda bulunacak şekilde:

```
#include <stdio.h>
int□main(void){printf("C programlama dili\n");return□0;}
```

NOT:
□ işareti, atom ayracı bulunması gereken kısımları göstermek için kullanılmıştır.

Görüldüğü gibi iki anahtar sözcük (**int** ve **main**); bir anahtar sözcük ve sabit (**return** ve 0) arasında mutlaka bir atom ayracı bulunmalıdır (boşluk karakteri yada açıklama). Aşağıdaki C deyiminde değişken isimleri ve operatörler arasında atom ayracı bulunması gerekmez:

```
c=a+b;
```

Her iki şekilde de program kodunun okunabilirliği oldukça sınırlıdır.

C dili birden fazla satırdan oluşan açıklama satırlarına izin verir; /* ile başlar ve */ ile sonlanır. İç içe bulunamazlar yada dizgi sabitlerinin içine yerleştirilemezler. Açıklama satırları ve düzenli bir şekilde yerleştirilen boşluk karakterleri, program kodunun okunabilirliğini (*readibility*) artırır ve program üzerinde daha sonra yapılacak çalışmaları kolaylaştırır. Kitaptaki örneklerde, programların kısa oluşları ve metin içinde açıklamaların bulunduğu göz önünde bulundurularak açıklama satırlarına çok az yer verilmiştir.

Her iki programda da görüldüğü gibi # ile başlayan önişlemci komutları tek bir satırda bulunabilir (satır devam ettirmek için kullanılan \ ile bölünmedikçe). Önişlemci komut satırında bulunan atomlar FF yada VT ile ayrılamazlar. # öncesinde yada sonrasında istenen miktarda düz boşluk ve HT (yada açıklama) bulunabilir. ■

BÖLÜM 2 : **Tipler, Operatörler ve İfadeler**

Bu bölümde değişkenler, sabitler, değişkenlerin bildirilmesi, operatörler ve ifadeler anlatılacaktır. Bildirimler, programda kullanılacak değişkenleri listeler, değişkenlerin özelliklerini ve ilk değerlerini bildirir. Operatörler, değişkenler ve sabitler üzerinde yapılacak işlemleri belirtir. Programlarda operatörler ve operand'lardan (değişken, sabit, fonksiyon çağrısı) oluşan C ifadeleri işlenerek yeni değerler elde edilir. Bir program deyimi, "**;**" ile sonlanmış bir ifadeden oluşur. Bu bölümde son olarak tip dönüşümleri anlatılacaktır. ■

2.1 Değişken isimleri

Değişken, sembolik sabit ve fonksiyon isimleri (*identifiers*) harf, rakam ve alt-çizgiden oluşur. İlk karakter bir harf olmalıdır. Alt-çizgi (*underscore*) karakteri "_", harf olarak işlem görür ve uzun değişken isimlerinde okunabilirliği artırmak için kullanılabilir. Aşağıdakiler geçerli isim örnekleridir:

```
SecenekSayisi
DIZI_BOYU
Listele
Sayi123
```

Bir isim boşluk, virgül yada **( ) & $ # . ! \ ?** gibi özel semboller içeremez. Kütüphane rutinleri genel olarak alt-çizgi ile başladığı için program içinde kullanılan değişken isimlerini alt-çizgi ile başlatmak uygun olmaz. C dilinde küçük ve büyük harf ayrımı yapılır; örneğin Dizi ile dizi birbirinden farklıdır. Değişken isimlerinin küçük harflerle, sembolik sabit isimlerinin ise büyük harflerle oluşturulması yaygın bir uygulamadır. Program içindeki dış başlanma özelliği olmayan (*internal*) bir ismin 31 karakteri belirgindir. Yani 31 karakterden uzun isimlerin kullanılması anlamsızdır. Dış bağlanma özelliği olan fonksiyon isimlerinin ve external değişkenlerin ise 6 karakteri belirgindir ve bu isimlerde büyük-küçük harf ayrımı yapılmayabilir. Anahtar sözcükler değişken ismi olarak kullanılamaz. ■

2.2 Veri Tipleri ve Boyutları

C dilindeki temel veri tipleri şunlardır: **char**, **int**, **float** ve **double**.

Tamsayı tipler (*integral types*):

char Karakter veri tipi (1- byte). C karakter tablosunda bulunan bir karakteri (daima pozitif) saklayabilir.
Bu tipin sınırları (minimum ve maksimum; yazılan değerler dahil):
unsigned char : 0 ve 255
signed char : -128 ve 127

int Tamsayı veri tipi (donanıma bağlı olarak 2 yada 4 byte).
Sınırları (minimum ve maksimum; yazılan değerler dahil):
unsigned int : 0 ve 65535 (2-byte)
signed int : -32768 ve 32767 (2-byte)

Bu tiplere **signed** yada **unsigned** anahtar sözcükleri eklenerek sırasıyla *işaretli* yada *işaretsiz* (*pozitif yada 0*) oldukları belirtilebilir. Bunlardan biri kullanılmadığında derleyiciye bağlı olarak tip işaretli yada işaretsiz kabul edilir. **unsigned** yada **signed** bulunan bildirimlerde **int** kullanılmayabilir; örneğin **unsigned i;**.

short ve **long** tip belirleyiciler bir tamsayı tipin uzunluğunu değiştirir; "**short int**", bir kısa tamsayıdır ve 16 bit'tir; "**long int**", bir uzun tamsayıdır ve 32 bit'tir.

Sınırları (minimum ve maksimum; yazılan değerler dahil):
unsigned short : 0 ve 65535
signed short : -32768 ve 32767

unsigned long :0 ve 4294967295
signed long: -2147483648 ve 2147483647

Bildirimlerde **int** kullanılmayabilir. Örneğin **unsigned short int** yerine **unsigned short** kullanılabilir. Ayrıca tip belirleyiciler istenilen sırada yerleştirilebilir; örneğin **int short unsigned**.

Bilgisayarda gerçel sayıları (*real numbers*) yani kesirli kısmı bulunan sayıları temsil etmek için kayan-nokta düzeni (*floating-point notation*) kullanılır.

Kayan-nokta tipler (*floating-point types*) şunlardır:

Pozitif yada negatif olabilirler. Hassasiyet (*precision*), ondalık noktadan sonraki hane sayısını belirtir. Aşağıda görüldüğü gibi 3 ayrı hassasiyet seviyesi vardır.

float	Tek-hassasiyetli kayan nokta (4-byte). *Hassasiyet* : en az 6 hane.
double	Çift-hassasiyetli kayan nokta (8-byte). *Hassasiyet* : en az 10 hane.
long double	Genişletilmiş hassasiyetli kayan nokta. *Hassasiyet* : en az 10 hane. Byte sayıları ve sınırlar donanıma göre değişir.

Standart başlık dosyaları limits.h ve float.h'de sınırlar için sembolik sabitler bulunur. ■

2.3 Sabitler

Tamsayı sabitler, **l** yada **L** harfi eklenmediği zaman **int** değerlerdir; örneğin 1234 değeri. Tamsayı sabit sonuna **l** yada **L** harfi eklenerek bunun bir **long** değer olduğu belirtilir; 123l yada 123L gibi. **unsigned** (işaretsiz) tamsayı sabitlere, **u** yada **U** harfleri eklenir; **unsigned long** sabitlere ise **ul** yada **UL** harfleri eklenir.
Tamsayı sabitler onluk (*decimal*) yerine önüne **0** yerleştirilerek sekizlik (*octal*); önüne **0x** yada **0X** yerleştirilerek onaltılık (*hexadecimal*) sayı sistemlerinde gösterilebilirler; örneğin onluk sabit 171, sekizlik sabit olarak 0253 yada onaltılık sabit olarak 0xAB veya 0xab şeklinde gösterilebilir. Bu sabitlere de **L** yada **U** eklenebilir.

Kayan-noktalı sabitler ondalık nokta içerebilir (123.4); üslü (*exponent*) olarak verilebilir (1e-2 yada 1E-2) yada her ikisi bir arada (1.23e-2) bulunabilir. **f** yada **F** harfleri eklenmiş ise **float** aksi taktirde **double** değerlerdir. **l** ve **L** ekleri ise **long double** sabitler için kullanılır.

Tırnaklar arasına yerleştirilmiş bir karakterden oluşan *karakter sabitleri* tamsayı değerlerdir; örneğin 'a', 'X'. Karakter sabiti, bilgisayarın karakter tablosunda sözkonusu karaktere karşılık gelen sayısal değeri verir. Karakter sabiti '0', ASCII karakter tablosunda (*ASCII character set*) 48 değerine karşılık gelir.
Bir programda bu karakteri temsil etmek için sayısal değer 48 yerine karakter sabiti '0' kullanımı daha pratiktir. Karakter sabitleri diğer tamsayılar gibi işlemlerde kullanılabilirler.

Dizgi sabitleri (*string constant*, *string literal*) çift tırnaklar arasında bulunan sıfır yada daha çok karakterden oluşur; "deneme" yada boş dizgi "" gibi. Dizgi sabitleri izleyen bölümlerde ayrıntılı olarak incelenecektir.

Dizgi sabitleri ve karakter sabitleri *escape dizileri* içerebilirler. Escape dizileri, ters kesme karakteri \ ve bunu izleyen harf yada rakamlardan oluşur; *white-space* (boşluk karakterleri) yada grafiksel olarak temsil edilemeyen karakterleri temsil etmek ve bazı özel anlamı olan karakterlerin bu özel anlamını derleyiciden gizlemek için kullanılır. Escape dizileri birden fazla karakter kullanılarak oluşturulur fakat sadece bir karakteri temsil ederler. Örneğin printf fonksiyonu çağrısındaki dizgi sabiti, ekranda bir alt satıra geçilmesini sağlayan \n (*newline*) escape dizisini içerir.

Escape dizileri:

\a	*bell*
\b	*backspace*
\f	*formfeed*
\n	*newline*
\r	*carriage return*
\t	*horizontal tab*
\v	*vertical tab*
\\	\ *karakteri*
\?	? *karakteri*
\'	' *karakteri*
\"	" *karakteri*

ASCII karakterleri temsil etmek için sekizlik yada onaltılık ASCII karakter kodları ile oluşturulan escape dizileri kullanılabilir:

> '\ooo' escape dizisinde ooo, 0'dan 7'ye kadar rakamlardan oluşan sekizlik ASCII karakter kodudur (burada sekizlik sayının ilk rakamı olarak 0 verilmesi gereksizdir). Örnek olarak *backspace* için '\010' yada '\10' kullanılabilir.
>
> '\xhhh' escape dizisinde hhh, 0'dan 9'a ve a'dan f'e (yada A'dan F'e) kadar rakamlardan oluşan onaltılık ASCII karakter kodudur. Örnek olarak *backspace* için '\x08' yada '\x8' kullanılabilir.

Escape dizileri dizgi sabitleri içinde yukarıdaki şekilde kullanıldığında, hatalara sebep vermemek için tüm haneler doldurulmalıdır. Örneğin derleyici "\x07Bell" dizgisini \x07B ({ karakteri) ve ell olarak yorumlar. Bunun yerine "\x007Bell" kullanılmalıdır.

'\0' karakter sabiti ('\000' yada '\x000'), ASCII karakter tablosunda sayısal kodu sıfır olan boş karakterdir (*null character*). Sayısal kodu 48 olan '0' ile karıştırılmamalıdır.

Karakter sabiti, tek karakter içeren dizgi sabiti ile karıştırılmamalıdır; örneğin 'a' ve "a" aynı değildir. 'a' bir tamsayıdır ve karakter tablosunda bir sayısal değere (koda) eşittir; "a" ise a karakteri ve dizgiyi sonlandıran null karakterden ('\0') oluşan bir karakter dizisidir.

C dilinde **enum** anahtar kelimesi kullanılarak her biri bir tamsayı sabite (*enumeration constant*) karşılık gelen isimlerden oluşan liste tanımlanabilir. Bir **enum** bildirimi çeşitli şekillerde yapılabilir. Aşağıdaki deyimde, **enum** tipi *etiket* tanımlanır:

enum *etiket* { *enum-listesi* ... };

Tanımlanan **enum** tipi kullanılarak **enum** değişkeni *isim* aşağıdaki şekilde bildirilir:

enum *etiket isim*;

enum tipi ve **enum** değişkeni bildirimi bir arada yapılabilir (seçime bağlı kullanımlar köşeli parantezler arasında verilmiştir):

enum [*etiket*] { *enum-listesi*...} [*isim*];

Aşağıda **enum** tipi olarak SAYI etiketi bildirilir:

```
enum  SAYI  { SIFIR, BIR, IKI, UC, DORT };
```

Listedeki her elemana aşağıdaki şekilde açık olarak sabit değer ataması yapılabilir:

... eleman_ismi = *sabit-ifade*, ...

Değer atanmadığı durumda listedeki ilk elemanın değeri 0 olur. Ayrıca değer atanmamış herhangi bir elemanın değeri kendinden önceki elemanın değerinin bir fazlasıdır. Bu nedenle sabit listesindeki hiç bir elemana ilk değer atama yapılmaz ise listedeki isimler sıfırdan başlayarak birer birer artan sıralı değerler alır. Örneğin SIFIR'ın değeri 0, BIR'in değeri 1, IKI'nin değeri 2 gibi.

Aşağıdaki **enum** bildiriminde görüldüğü gibi açık atama yapılabildiği için listedeki sabit değerlerin sıralı olması gerekmez:

```
enum  LISTE { a = 12, b = 0, c = 5 };
```

enum tipi SAYI kullanılarak **enum** değişkeni DEGER bildirilebilir:

```
enum  SAYI  DEGER;
```

enum tipi ve değişkeni bildirimi aynı deyimde yapılabilir:

```
enum SAYI  { SIFIR, BIR, IKI, UC, DORT }  DEGER;
```

Eğer aynı tipte başka değişken bildirilmeyecek ise **enum** bildirimi etiket olmadan yapılabilir:

```
enum   { SIFIR, BIR, IKI, UC, DORT }  DEGER;
```

Herhangi bir **enum** değişkeni bildirmeksizin isimlerle ifade edilen sabitlerden oluşan **enum** listesi bildirilebilir:

```
enum   { SIFIR, BIR, IKI, UC, DORT };
```

Örnek 2.3-1 enum.c programı.

```
#include <stdio.h>

enum SAYI { SIFIR,  BIR,  IKI, UC, DORT  };
enum { OCAK  = 1, SUBAT,  MART,  MAYIS  = 5, HAZIRAN};

int main(void)
{
  enum SAYI DEGER = SIFIR;
  int AY = OCAK;

  printf( "SIFIR : %d\n", DEGER );

  DEGER = DORT;
```

... Örnek 2.3-1 devam

```
  printf( "DORT : %d\n", DEGER );
  printf( "OCAK : %d\n", AY );
  AY = MART;
  printf( "MART : %d\n", AY );
  AY = HAZIRAN;
  printf( "HAZIRAN : %d\n", AY );
  return 0;
}
```

Çıktı

```
SIFIR : 0
DORT : 4
OCAK : 1
MART : 3
HAZIRAN : 6
```

Örnek 2-3-1'de **enum** tipi SAYI ve SAYI tipinde DEGER değişkeni bildirilir. Bildirim sırasında DEGER değişkenine SIFIR değeri atanır. Ayrıca **int** tipinde AY değişkeni bildirilir ve ilk değer olarak **enum** sabiti OCAK atanır. Bir **enum** listesi sadece tamsayı değerlerden oluşur.

Bu örnekte de görüldüğü gibi **enum** tipinde bir değişken, **enum** tipi ile tanımlanan değerlerden birini taşır ve dolayısıyla daima **int** tipindedir. Bundan başka bir **enum** listesinde bulunan sabit ismi, bir diğerinde kullanılamaz. Fakat aynı kısıtlama listede yer alan sabit değerler için sözkonusu değildir.

enum tipi, tamsayı sabitler yerine anlamlı isimler kullanılmasını sağladığı için programların okunabilirliği arttırır. Aynı işlem, bir grup #define komutu ile gerçekleştirilebilir. Fakat birbiri ile ilgili değerler tek bir listede toplandığı için **enum** kullanımı daha pratiktir.

C dilinde, veri tipi olarak mantıksal doğru ve yanlış (*Boolean TRUE* ve *FALSE*) değerleri yoktur. Herhangi bir test ifadesi, 0 dışında herhangi bir pozitif yada negatif değer veriyor (*non-zero value*) ise koşul doğru kabul edilir. Eğer 0 değeri veriyor ise koşul yanlıştır.

Doğru yada yanlış koşulunu test etmek için TRUE ve FALSE makroları kullanılabilir:

```
#define TRUE   1
#define FALSE  0
```

Fakat bu değerler **enum** sabiti olarakta tanımlanabilir:

```
enum bool { FALSE, TRUE };
```

Burada FALSE sabitinin değeri 0, TRUE sabitinin değeri ise 1'dir. ■

2.4 Bildirimler

Tüm değişkenler, program içinde kullanılmadan önce bildirim deyimleri ile bildirilmelidir. Bir bildirim, değişkeninin tipini belirtir. Aynı deyimde aynı tipte birden fazla değişken bildirilebilir; yada her değişkenin bildirimi ayrı bir bildirim deyimi ile yapılabilir:

```
int a, b, c;
char s, dizi [ 10 ];
```

yada

```
int a;
int b;
int c;
char s;
char dizi [ 10 ];
```

Bir değişkene bildirim deyiminde = ve bir ifade ile ilk değer atama yapılabilir:

```
char c = 'a';
int i = 9;
```

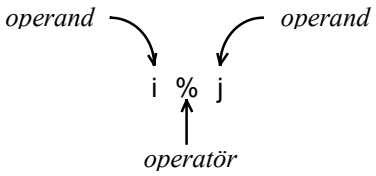
Bu bildirim deyimlerinde ilk değer olarak 'a' ve 9 kullanılır. Bildirim deyimlerinde çeşitli belirleyiciler (*qualifier* yada *storage class specifier*) kullanılabilir (const, volatile yada static, auto gibi). Bildirimler izleyen bölümlerde ayrıntılı olarak incelenecektir. ■

2.5 Operatörler

Aritmetik operatörler

Aritmetik işlemlerde kullanılan ve iki operand alan (*binary*) aritmetik operatörler şunlardır: +, -, *, / ve kalan operatörü %.

Aritmetik ifade:



Tamsayılarla yapılan bölme işleminde kesirli kısım atılır (*truncated*). **float** ve **double** operand'lara uygulanamayan % operatörü i'nin j'ye bölümünden kalan değeri verir.

İlişkisel ve Mantıksal Operatörler

Operand'ları karşılaştırmak için kullanılan ilişkisel operatörler ve anlamları şunlardır:

<	*küçüktür*
>	*büyüktür*
<=	*küçüktür yada eşittir*
>=	*büyüktür yada eşittir*

Aşağıdaki iki operatör yukardakilerden daha düşük önceliklidir:

==	*eşittir*
!=	*eşit değildir*

İlişkisel operatörler kullanılarak oluşturulan ilişkisel ifadeler, koşul ifadeleri olarakta adlandırılır. Mantıksal operatörler && (-**ve**- operatörü) ve || (-**veya**- operatörü) ile birleştirilen ifadeler soldan-sağa doğru işlenir; sonucun *doğru* yada *yanlış* olduğu saptanır saptanmaz işlem durur. C dilinde ilişkisel ve mantıksal ifadeler doğru ve yanlış sonuçları (*boolean* değer) vermez. İlişkisel ve mantıksal ifadelerin değerleri bir tamsayıdır; 0 (*yanlış*) yada 0 dışında bir değer (*doğru*). Aşağıdaki mantıksal ifadede ilk olarak *ifade1* işlenir; doğru ise *ifade2* işlenir ve bu ifade de doğru ise *ifade3* işlenir.

İfadelerden biri, örneğin *ifade1* yanlış sonucunu verir ise mantıksal ifadeden çıkılır ve diğerleri işlenmez:

ifade1 && *ifade2* && *ifade3*

Bu özellik kullanılarak sıfıra bölme hatası (*divide-by-zero*) aşağıdaki şekilde önlenebilir:

```
d != 0  &&  n/d < 10
```

İlişkisel ifade d != 0 ilk olarak işlenecek ve d'nin değeri 0 olduğunda yanlış sonucunu vereceği için n/d işlenmeden ifadenin tamamından çıkılacaktır. && operatörü, || operatöründen daha yüksek önceliğe sahiptir. Ayrıca her iki operatör de ilişkisel

operatörlerden daha düşük önceliğe sahip olduğu için ilişkisel operatörlerle oluşturulan *ifade1*, *ifade2* yada *ifade3* dışında parantezler gereksizdir.

Bir diğer mantıksal operatör, ! (-**değil**- operatörü: *unary negation operator*) operatörüdür; 0 dışında bir değere sahip operand'ın değerini 0, değeri 0 olan operand'ın değerini 1 yapar. Genel olarak test ifadelerinde kullanılır; örneğin `if (!dogru)` ifadesi `if (dogru == 0)` yerine kullanılır.

Artırma ve Eksiltme Operatörleri

C dilinde değişkenlerin değerlerinin artırılması ve eksiltilmesi için artırma (*increment*) ve eksiltme (*decrement)* operatörleri bulunur. Artırma operatörü **++**, operand'ının değerine 1 ekler; eksiltme operatörü -- ise operand'ının değerinden 1 çıkarır. Her iki operatör de, değişkenden önce önek (*prefix*) yada değişkenden sonra sonek (*postfix*) operatörler olarak kullanılabilir:

++i,	*önek artırma: değişkenin değeri artırıldıktan sonra kullanılır.*
i++,	*sonek artırma: değişkenin değeri kullanıldıktan sonra artırılır.*
--i,	*önek eksiltme: değişkenin değeri eksiltildikten sonra kullanılır.*
i--,	*sonek eksiltme: değişkenin değeri kullanıldıktan sonra eksiltilir.*

--(a + b) şeklinde bir ifade hatalıdır. Çünkü ++ ve -- operatörleri sadece değişkenlere uygulanır (*lvalue*). Bazı uygulamalarda operatörlerin önek ve sonek kullanımları farklı sonuçlar verebilir.

Bit Operatörleri

C dilinde mantıksal bit işlemleri için yalnızca işaretli yada işaretsiz **char**, **short**, **int**, ve **long** operand'lara (*integral types*) uygulanabilen 6 operatör bulunur:

&	**ve (**bitwise AND)
|	**veya** (bitwise OR)
^	*dışlayan* **veya** (bitwise exclusive-OR)
<<	sola bit kaydırma (shift left)
>>	sağa bit kaydırma (shift right)
~	**değil** yada 1'in tümlevi operatörü (bit-negate yada bitwise complement)

Aşağıda bu operatörler kullanılarak çeşitli bit işlemleri yapılır:

VE (AND)

&	**0**	**1**
0	0	0
1	0	1

Örnek:

```
  0101 0101      0x55
& 1111 0000      0xF0
  ---------
  0101 0000      0x50
```

VEYA (OR)

|	0	1
0	0	1
1	1	1

Örnek:

```
  0101 0101      0x55
| 1111 0000      0xF0
  1111 0101      0xF5
```

dışlayan - VEYA (EXCLUSIVE-OR)

^	0	1
0	0	1
1	1	0

Örnek:

```
  0101 0101      0x55
^ 1111 0000      0xF0
  1010 0000      0xA0
```

DEĞİL (NOT)

~	
0	1
1	0

```
unsigned char a = 0x55;                 /*  0101 0101  */
unsigned char b = 0xF0;                 /*  1111 0000  */
~a          ⇨    AA                     /*  1010 1010  */
a << 1      ⇨    AA                     /*  1010 1010  */
b >> 6      ⇨     03                    /*  0000 0011  */
```

Atama Operatörleri ve ifadeleri

Aþaðýdaki ifade basit bir atama ifadesidir:

degisken = *operand*

Bu ifade, sonuna **;** eklendiðinde atama deyimi olur. Bu ifadede *operand*'ýn deðeri *degisken*'e atanýr. Eþittir iþareti =, C dilinde *atama operatörü*dür (*assignment operator*). *operand* bir deðiþken, sabit bir deðer yada bir baþka geçerli C ifadesi olabilir. Bir atama ifadesinin deðeri saðda bulunan operand ile aynýdýr; tipi ise solda bulunan operand ile aynýdýr. Buna göre yukarýdaki atama ifadesinin değeri *degisken*'e atanan değer (sağda bulunan operand) ile aynıdır; tipi ise *degisken (*solda bulunan operand) ile aynıdır. Örneğin aşağıdaki printf satırı ekrana 9 değerini yazar:

```
printf( "%d\n", i = 9 );
```

Aynı ifadede birden fazla atama bulunabilir:

```
i = j = k = 7
```

Bu ifadedeki tüm atama operatörleri aynı operatör önceliğine sahiptir. Bu durumda ifade, operatörlerin grup ilişkisine göre işlenir.

Atama operatörünün grup ilişkisi (izleyen paragraflarda anlatılacak) sağdan-sola doğru olduğu için bu ifade aşağıdaki sırada işlenir:

```
k = 7
j = k
i = j
```

Bu ifadeler aşağıdaki ifade ile aynı etkiye sahiptir:

```
i = ( j =  (k = 7) )
```

İfade **;** ile sonlanır ise oluşturulan atama deyiminde tüm değişkenlere 7 değeri atanır:

```
i = j = k = 7;
```

Bu atama işlemi aşağıdaki 3 ayrı atama işlemi ile aynı işi yapar:

```
k = 7;
j = 7;
i = 7;
```

Aşağıdaki deyim, bir gizli atama deyimidir (*embedded assignment*):

```
( c = getchar( ) ) != EOF
```

getchar tarafından döndürülen değer ilk olarak c'ye atanır; atanan değer aynı zamanda bu atama ifadesinin değeridir ve daha sonra EOF ile karşılaştırılır. Parantezler gereklidir çünkü **=** operatörü **!=** operatöründen daha düşük önceliğe sahiptir. Bu ifade bir **while** döngüsünün test ifadesinde kullanılabilir.

```
i = i + 9
```

ifadesi iki aşamada işlenir: ilk olarak i + 9 ifadesinin değeri hesaplanır; hesaplanan değer i değişkenine atanır. C dilinde, **=** operatörünün sağında ve solunda aynı değişkenin bulunduğu ifadeler tek bir atama operatörü (+=) ile oluşturulabilir:

```
i += 9
```

İki operand alan pek çok operatör'e (*binary operators*) karşılık gelen atama operatörü vardır.

Aşağıdaki operatörler kullanılarak, *operator*= şeklinde atama operatörleri oluşturulabilir:

```
+  -  *  /  %  <<  >>  &  ^  |
```

Sonuç olarak,

ifade1 ***operator=*** *ifade2*

ifadesi ve

ifade1 **=** (*ifade1*) ***operator*** (*ifade2*)

ifadesi birbirine eştir.

Üstteki ifadede *ifade1* yalnızca bir kez işlenir. *ifade2* dışındaki parantezler mutlaka kullanılmalıdır. Örneğin,

```
a *= b + 1
```

ifadesi,

```
a = a * (b + 1)
```

ifadesine eştir; `a = a * b + 1` ifadesine değil. ■

2.6 Koşul İfadeleri

Üç operand alan **?:** operatörü (*ternary operator*) ile koşul ifadeleri (*conditional expressions*) oluşturulabilir:

(*ifade1*) **?** *ifade2* **:** *ifade3*

İlk olarak *ifade1* işlenir. Eğer sonuç 0 dışında bir değer (*non-zero value*), yani doğru ise *ifade2* işlenir ve koşul ifadesinin değeridir. Aksi taktirde koşul ifadesinin değeri *ifade3* olur. Aynı işlem aşağıdaki **if** deyimi ile gerçekleştirilebilir:

```
if (ifade1)
    ifade2
else
    ifade3
```

Koşul ifadesi, diğer C ifadelerine izin verilen her yerde kullanılabilir. *ifade2* ve *ifade3* farklı tiplerde ise sonuç tip dönüşüm kurallarına göre belirlenir. *ifade1* dışındaki parantezler **?:** operatörünün önceliği çok düşük olduğu için kullanılmayabilir. ■

2.7 Operatör Önceliği ve İşleme Sırası

Birden fazla operatör içeren ifadeler, parantezler bulunmuyor ise operatör *öncelik sırası*na (*precedence*) göre işlenir. Aynı önceliğe sahip operatörlerin bulunduğu ifadeler ise operatörlerin *grup özellikleri*ne (*associativity*) göre işlenir. Şekil 2.7-1'de operatörlerin öncelik ve grup özellikleri listelenir. Bu tabloda aynı satırda bulunan operatörler eşit önceliğe sahiptir; aşağı doğru operatör önceliği azalır. İfadelerde parantezler kullanılarak öncelikler değiştirilebilir.

C dili **&&**, **||**, **?:**, ve virgül operatörü "**,**" dışında bir operatörün operand'larının hangi sırada işleneceğini belirtmez. Aşağıdaki test ifadesinde ifadeler *soldan-sağa* doğru işlenir:

```
if (  (c = getchar()) == EOF || c == '\n' ) { ... }
```

Fakat aşağıdaki ifadede çağrılardan hangisinin önce işleneceği belli değildir. Dolayısıyla fonksiyonların iç yapısına bağlı olarak a değişkeninin değeri çağrıların işleniş sırasına göre değişebilir:

```
a = fonk1( ) + fonk2( );
```

Bu gibi belirsiz durumlar bazı teknikler kullanılarak önlenebilir; örneğin bu ifadede çağrıların döndürdüğü değerler ara değişkenlerde saklanarak belli bir işleniş sırası garanti edilebilir. Aynı şekilde fonksiyon argümanlarının işleniş sırası da derleyiciye göre değişebilir:

```
fonk1( ++i, fonk2( i ) );
```

Bu çağrıdaki belirsizlik aşağıdaki şekilde önlenebilir:

```
++i;
fonk1( i, fonk2( i ) );
```

Bir C ifadesinin iþlenmesi sýrasýnda ifadenin asýl amacý dýþýnda oluþan etkiler *yan etki* olarak adlandırılır. Örneğin aşağıdaki atama deyiminin asıl amacı, i değişkenine j değişkeninin değerinin atamaktır. Yan etki ise atama sonrası j değişkeninin değerinin artmasıdır:

```
i = j++;
```

Yan etkiler her zaman açık olmayabilir. Yan etkiler içeren ifadelerin işleniş sırası derleyiciye bağlı olarak değişebilir. Örneğin aşağıdaki ifadede indeks değerinin i değişkeninin ++ sonrası değeri mi, yoksa ++ öncesi değeri mi olduğu derleyiciye göre değişir:

```
dizi[ i ] = i++;
```

Bir diğer örnek olarak aşağıdaki printf çağrısı verilebilir:

```
int i = 9;
...
printf( "%d\n", i++ *  i++  );
```

Şekil 2.7-1 Operatör öncelik tablosu.

Operatörler *operatör önceliği aşağı doğru azalır*	***Grup İlişkileri***
Fonksiyon çağırma, indeks, ok ve nokta operatörleri: **( ) [ ] -> .**	soldan sağa ⇨
! ~ ++ -- + - * & (*tip*) sizeof	⇦ sağdan sola
Aritmetik operatörler (iki operand alır): *** / %**	soldan sağa ⇨
+ -	
<< >>	soldan sağa ⇨
İlişkisel operatörler: **< <= > >=**	soldan sağa ⇨
== !=	
&	soldan sağa ⇨
^	
|	
&&	
||	
?:	⇦ sağdan sola
= += -= *= /= %= &= ^= |= <<= >>=	⇦ sağdan sola
Virgül operatörü: **,**	soldan sağa ⇨
Tek operand alan (unary) **+**, **-** *ve* ***** *operatörleri aynı operatörlerin çift operand alanlarından (binary) daha yüksek önceliğe sahiptir.*	

Sonuç olarak, ifadelerin işleniş sırasına bağlı kodlar oluşturulmamalıdır. ■

2.8 Tip Dönüşümleri

C dilinde tip dönüşümleri otomatik olarak (tip genişletme -*promoting*-, tip dengeleme -*balancing*- yada atama sırasında -*assigning*-) yada program içinde tip dönüştürme operatörü kullanılması ile (*type casting*) gerçekleşir.

Eğer bir ifade de bulunan tamsayı değer **int**, **unsigned int**, **long** yada **unsigned long** tipinde değil ise tipi genişletilir (tamsayı değerin **sizeof** operatörünün operand'ı olarak bulunduğu ifade dışında). Bu işlem otomatik olarak integral tiplere uygulandığı için *integral promotion* olarak adlandırılır. Eğer orjinal tipteki tüm değerler bir **int** ile temsil edilebiliyor ise yeni tip **int** olur; aksi taktirde **unsigned int** olur.

Bu işlem, işaretli yada işaretsiz **char**, **short int**, tamsayı bit-alanı yada enum tipler üzerinde uygulanır. Kayan-nokta tiplere bu işlem uygulanmaz. Örneğin bir **signed char** değer **int**'e; **unsigned short** değer ise **int** yada **unsigned int**'e genişletilir.

Örnek 2.8-1'de çeşitli tamsayı tip genişletme işlemleri yer alır. Örnekte yer alan ilk **if** deyiminde,

```
c == ~0x5A
```

testi yapılır. Burada **unsigned char** tipinde c değişkeninin değeri olan 0xA5, 0x00A5'e genişler. Ayrıca yine 0x005A'ya genişleyen KARAKTERSABIT'in değeri ~ operatörü uygulandıktan sonra 0xFFA5 olur. Dolayısıyla test ifadesi yanlış sonucunu verir ve **if** deyiminin **else** kısmı çalışır. Bu sorun test ifadesinde tip dönüştürme operatörü (izleyen paragraflarda anlatılacaktır) kullanılarak KARAKTERSABIT'in **unsigned char**'a dönüştürülmesi ile önlenebilir.

İzleyen **if** deyiminde 0xC0 değerine (1100 0000) 3 kez *sola bit-kaydırma* işlemi uygulanır. Fakat bu değer kaydırma öncesi **int**'e genişlediğinden dolayı sonuç 0 olmaz. Yine örnekte de görüldüğü gibi tip dönüştürme yapılarak bu sorun önlenebilir.

Programda **char** tipinde k1, k2 ve k3 değişkenleri bildirilir. Program, karakter tipinin 8-bit olduğu donanımda k2 için ekrana -56 yazar (orjinal değer 200 yada 0xC8). Derleyici printf parametrelerine (format dizgisi hariç) tip kontrolü yapmaz ve k2'nin **int**'e genişlediğini saptayamaz. Bir **char** tip, derleyici tarafından **signed** yada **unsigned** (işaretli yada işaretsiz) olarak yorumlanabilir. Bu donanımda **signed** olarak yorumlar ve **signed char** veriyi **int**'e işaret genişlemesi (*sign extension*) ile genişletir; yani boş kalan yüksek anlamlı ilave bit pozisyonlarını orjinal işaretli **char** verinin işaret bit (en yüksek anlamlı bit) değeri olan 1 ile doldurur. Dolayısıyla **signed char** 0xC8 değeri **int**'e 0xFFC8 olarak genişler (-56). Toplama işlemi 0064 + FFC8 = 002C şeklinde olur ve 0x2C (yada 44) sonucunu verir.

unsigned char'ın **int**'e genişlemesi boş kalan yüksek anlamlı bit pozisyonlarının 0 ile doldurulması ile olur. Programda **unsigned char** kullanıldığında toplama sonucu elde edilen 300 değeri (0x012C) 1-byte alana sığmaz. Sonuç 44 yada 2C olur.
Burada daha kısa bir tipe dönüşüm (*demotion*) gerçekleşir. **signed** tiplerin bu tip dönüşümlerinde sonuç derleyiciye göre değişir.

İzleyen işlemde k1'in değeri 0x7C'dir (124). k1 + k1 ifadesinde her iki 0x7C değeri 0x007C değerine genişler ve toplamları 0x00F8'dir. Tekrar dönüşüm olmaz ve toplam değer ekranan yazılır. Fakat toplamın atandığı k2'nin değeri genişler; k2'nin (0xF8) yüksek anlamlı bit'i dönüşüm öncesi 1 olduğu için ekrana yazılan değer 0xFFF8 olur.

Bir operatör (+ yada * gibi iki operand alan -binary operator-) farklı tiplerde operand'larla işlem yapıyor ise operand'lar belli bazı dönüşüm kurallarına göre ortak bir

veri tipine dönüştürülür (*balancing*). İfadede tamsayılar bulunuyor ise integral promotion gerçekleşir. Bu işlem aşağıda verilen ve tiplerin önceliğinin sıralamada ileriye doğru arttığı listeye göre gerçekleşir:

int, unsigned int, long, unsigned long, float, double, long double

Buna göre eğer operand'lardan herhangi biri **long double** ise diğeri de bu tipe dönüştürülür; yada operand'lardan herhangi biri **double** ise diğeri de bu tipe dönüştürülür; yada operand'lardan herhangi biri **float** ise diğeri de bu tipe dönüştürülür; yada integral promotion uygulanır.

```
int i;
long l;
double d;
```

Örneğin yukarıda bildirilen değişkenlerle oluşturulan ((i + l) + d) ifadesinde ilke olarak i'nin tipi **long**'a daha sonra da i + l ifadesinin tipi **double**'a çevrilir.

Atama işlemlerinde (atama operatörleri ile oluşturulan) operand'lar farklı tiplerde ise tip dönüştürme gerçekleşir. Sağdaki değer sonucun tipini belirleyen soldaki tipe dönüştürülür. Atama işleminde, bir aritmetik tip (tamsayı ve kayan-nokta tipler) bir diğer aritmetik tipe dönüşebilir. Dönüşümler operand'ların tiplerine bağlı olarak veri kaybına yol açabilir. Fonksiyon çağrısında argümanlar aktarılırken atama işleminde olduğu gibi tip dönüştürme gerçekleşebilir. Prototip bulunmuyor ise **char** ve **short** tipler **int**'e; **float** tipler **double**'a dönüştürülür.

Tek operand alan (*unary*) tip dönüştürme operatörü (*cast operator*) kullanılarak tip dönüşümleri zorlanabilir. Aşağıdaki ifadede, *ifade*'nin tipi verilen tipe dönüştürülür. İzleyen bölümlerde ayrıca adres değişkenlerine tip dönüştürme uygulanması ve **void** tipi adres değişkenlerinin kullanımı anlatılacaktır. ■

(*tip-ismi***)***ifade*

Örnek 2.8-1 tip.c programı.

```
#include <stdio.h>

#define  KARAKTERSABIT  0x5A

int main(void)
{
   unsigned  char   c = 0xA5;
   unsigned  char   a = 0x40, b = 0xD0;

   char   k1   = 100,   /* 0x64  */
          k2   = 200,   /* 0xC8 */
          k3   = k1   + k2;
   unsigned
   char   uk1 = 100,   /* 0x64  */
          uk2 = 200,   /* 0xC8 */
          uk3 = uk1    + uk2;
   if ( c   ==  ~KARAKTERSABIT )
        ;
   else
        puts( "integral promotion" );
   if ( c   ==  (unsigned char)~KARAKTERSABIT )
        puts( "OK !" );
   else
        ;
   c = 0xC0;  /* 1100 0000 */
   if ( c << 3 )
        puts( "c << 3 --> Sonuc 0 degildir" );
   else
        ;

   if ( (unsigned char)(c << 3) )
     ;
   else
     puts( "c << 3 --> Sonuc 0 " );
   /* integral promotion: signed char --> int                    */
   printf( "k1 : %d, k2 : %d, k3 : %d\n",              k1,  k2,  k3 );
   printf( "k1 : 0x%X, k2 : 0x%X, k3 : 0x%X\n",  k1,  k2,  k3 );
   /* integral promotion: unsigned char --> unsigned int         */
   printf( "uk1 : %u, uk2 : %u, uk3 : %u\n",           uk1,   uk2,   uk3 );
   printf( "uk1 : 0x%X, uk2 : 0x%X, uk3 : 0x%X\n",   uk1,   uk2,   uk3 );

   k1   = 124;         /* 0x7C */
   k2   = k1   + k1;
```

...Örnek 2.8-1 devam

```
   printf( "k1          : 0x%02X\n",  k1 );
   printf( "k1 + k1   : 0x%02X\n",  k1   + k1   );
   printf( "k2          : 0x%02X\n",  k2 );

   return 0;
}
```

Çıktı

```
integral promotion
OK !

c << 3 --> Sonuc 0 degildir
c << 3 --> Sonuc 0

k1   : 100,    k2   :  -56,      k3   :  44
k1   : 0x64,  k2   :  0xFFC8,  k3   :  0x2C

uk1 :  100,     uk2 :  200,     uk3 :  44
uk1 :  0x64, uk2 :  0xC8,    uk3 :  0x2C

k1             :  0x7C
k1   + k1   :  0xF8
k2             :  0xFFF8
```

BÖLÜM 3: **Kontrol Akışı**

Kontrol Akışı (*flow of control*), program deyimlerinin çalıştırılma sırasını ifade eder. Program deyimleri, bu bölümde anlatılacak olan kontrol akışı deyimleri tarafından herhangi bir şekilde değiştirilmediği sürece, birbirini izleyen sırada çalıştırılır (*sequential flow of control*). ■

3.1 Deyimler ve Bloklar

C dilinde deyimler **;** ile sonlanır. Dolayısıyla bir C ifadesine **;** eklendiğinde *program deyimi* (*statement*) oluşturulmuş olur. Aşağıdakiler geçerli C deyimleridir (*expression statement*):

```
a = 10;
i = i + 2;
puts( "deneme" );
```

Sadece **;** ile oluşturulan deyim, *boş deyim* (*null statement*) olarak adlandırılır ve hiç bir işlem yapmaz. Çoğunlukla aşağıda görüldüğü gibi deyim bloğu bulunmayan döngülerin çalışması için kullanılır:

```
 ...
while (...) ;
 ...
 yada
 ...
for (...;...;...) ;
 ...
```

Oklu parantezler { ve } kullanılarak bildirimler ve deyimler grup haline getirilir ve *bileşik deyim* (*compound statement*) yada *blok* oluşturulur:

```
{
 ...
bildirimler;
deyimler;
 ...
}
```

Bu blok, şekil olarak tek bir deyime eşittir. Bloğu kapatan oklu parantezden sonra **;** kullanılmaz. Örnek olarak fonksiyon içindeki deyimleri sınırlayan parantezler; **if**, **else**, **while** yada **for** sonrasında birden fazla deyim bulunduğunda blok oluşturmak için kullanılan parantezler verilebilir. ■

3.2 if-else, else-if ve switch

if-else deyimi, bir yada daha fazla deyimin ancak belli bir koşul sağlandığında çalıştırılması için kullanılır:

```
if  (ifade)
    deyim1
else
    deyim2
```

else kısmı kullanılmayabilir. Test ifadesi *ifade* işlendikten sonra *doğru* sonucunu verdiğinde (yani 0 dışında bir değere sahip ise) *deyim1* çalıştırılır; eğer *yanlış* sonucunu

verir (*ifade*nin değeri 0'dır) ve **else** kısmı var ise *deyim2* çalıştırılır. **if** deyimi *ifade*'nin sayısal değerini kontrol ettiği için

```
if  (ifade != 0)
```

yerine

```
if  (ifade)
```

kullanılabilir. Test ifadeleri, ilişkisel ve mantıksal operatörler kullanılarak oluşturulur.

İç içe **if** deyimlerinde **else** kısmının kullanılmaması karışıklığa yol açabilir. Aşağıdaki deyimde **else** kısmı, kendine en yakın olan **else**'i bulunmayan **if** 'e aittir:

```
if ( ...)
  if ( ...)
        ...;
  else
        ...;
```

Eğer en üstteki **if**'e ait olması istenirse, oklu parantezler kullanılmalıdır:

```
if ( ...)
{
  if ( ...)
        ...;
}
else
  ...;
```

Aşağıdaki **else** deyimi, **for** bloğunda bulunan **if**'e aittir. Fakat program satırlarının pozisyonu en üstte yer alan **if**'e ait olduğu izlenimini verir:

```
if ( ...)
  for  ( ...)
        if ( ...)
        {
              ...;
        }
else
  ...;
```

İç içe **if**'lerde (*nested-if*) oklu parantezler kullanılarak olası hatalar önlenebilir.

Aşağıdaki **if** deyimlerinde birden fazla koşul test edilir:

```
if  (ifade)
    deyim
else if  (ifade)
    deyim
else if  (ifade)
    deyim
    .
    .
    .
else if  (ifade)
    deyim
else
    deyim
```

*ifade*ler sıralı olarak işlenir ve herhangi bir ifade *doğru* sonucunu verir ise ilgili bloktaki deyimler çalıştırılır ve tüm **if-else** zinciri sona erer.

"*deyim*"ler, bir deyimden yada parantezlerle sınırlı deyim grubundan oluşabilir. Zincirin sonunda bulunan **else** kısmı, önceki hiç bir koşul sağlanamaz ise çalıştırılır (*default case*) ve iptal edilebilir.

switch deyimi, *ifade*'nin tamsayı *sabit-ifade* yada sabit değerlerden birine eşit olup olmadığı test eder:

```
switch   (ifade)
{
   case sabit-ifade:    deyimler;
     .
     .
     .
   case sabit-ifade:    deyimler;
   default:             deyimler;
}
```

Herhangi bir **case**, *ifade*nin değerine eşit ise ilgili *deyimler* çalıştırılır; hiç biri eşit değil ise **default** kısmında bulunan deyimler çalıştırılır (**if-else** zincirinde bulunan son **else** kısmı gibi). Tüm **case** ifadeleri birbirinden farklı olmalıdır. **default** kısmı kullanılmayabilir. **default** kısmının sonda bulunması gerekmez. Programlarda **if-else** zincirleri yerine **switch** kullanılabilir. **break** ve **return** deyimleri kullanılarak **switch** bloğundan hemen çıkılabilir. Herhangi bir **case** içinde bulunan deyimler çalıştırıldıktan sonra **break** deyimine rastlanmaz ise izleyen **case**'ler taranır. Aynı deyimleri çalıştıran farklı **case**'ler peşpeşe sıralanabilir. ■

3.3 Döngüler: **while**, **for** ve **do-while**

Aşağıdaki **while** döngüsünde, *ifade* işlenir; değeri 0 değilse *deyim*ler çalıştırılır ve *ifade*'nin tekrar işlenmesi ile döngü devam eder:

```
while (ifade)
        deyim
```

ifade'nin değeri 0 oluncaya kadar döngü devam eder. Aşağıdaki **for** deyiminde yer alan 1. ve 3. ifadeler, atama yada fonksiyon çağrısı; 2. ifade ise bir ilişkisel ifadedir:

```
for (ifade1; ifade2; ifade3)
        deyim
```

ifade1 döngü başlangıcında ve sadece bir kez çalıştırılır. *ifade2* ise **for** döngüsünün test ifadesidir ve işlendikten sonra 0 dışında bir değer veriyor ise *deyim*ler çalıştırılır. *ifade3* ise *deyim*ler çalıştırıldıktan sonra çalıştırılır. Döngünün herhangi bir kısmı iptal edilebilir. for (;;) ; döngüsü, "sonsuz" döngüdür. Bu döngüden **break** yada **return** ile çıkılır. Yukarıdaki **for** döngüsü yerine aşağıdaki **while** döngüsü kullanılabilir:

```
ifade1;
while (ifade2)
{
        deyim
        ifade3;
}
```

Bir C operatörü olan virgül "," çoğunlukla **for** döngülerinde birden fazla ifadeyi bir arada yazmak için kullanılır. Virgül operatörü ile ayrılan ifadeler soldan sağa işlenir; sonucun tipi ve değeri sağ operand'ın tipi ve değeridir. Örnek olarak aşağıdaki **for** döngüsü verilebilir:

```
for ( ifade1, ifade2; ...; ... )
        ...;
```

while ve **for** döngülerinde test ifadesi üstte bulunur. **do-while** döngüsünde ise altta bulunur. Dolayısıyla, döngü bloğunda bulunan deyimler en az bir kez çalıştırılmış olur:

```
do
    deyim
while (ifade);
```

Önce *deyim*ler çalıştırılır daha sonra test ifadesi işlenir. Eğer doğru ise *deyim*ler tekrar çalıştırılır ve döngü devam eder. ■

3.4 **break** ve **continue**

Test ifadesi yanlış sonucunu vermeden önce **break** deyimi kullanılarak **for**, **while** ve **do-while** döngülerinden çıkılabilir. Ayrıca **switch** deyiminden çıkış için de kullanılabilir.

Sadece döngülerde kullanılabilen **continue** deyimi ise döngünün bir sonraki tekrara atlamasını sağlar. Bu işlem **while** ve **for** döngüsünde hemen test kısmına geçilmesini, **for** döngüsünde ise *ifade3*'ün çalıştırılmasını sağlar. ■

3.5 **goto** ve etiketler

break deyimi, iç içe döngülerde en içteki bloktan çıkılmasını sağlar. Tüm bloklardan çıkmak için **goto** deyimi kullanılabilir:

```
                for  ( ...)
                     for  ( ...)
                     {
                          ...
                          if ( ...)
                             goto etiket;
                     }
                ...
etiket:
                ...
```

etiket bir değişken ismi ile aynıdır ve **:** ile sonlanır. **goto** ile aynı fonksiyon içinde herhangi bir deyime bağlanabilir. Bir etiketin görünürlük alanı (*scope*) fonksiyonun tamamıdır.

Genel olarak **goto** kullanılarak yazılan kodların okunabilirliği azdır. Bu nedenle programlarda **goto** kullanımı tercih edilmez.

Bu bölümde C dilinde kontrol akışını değiştiren deyimler anlatıldı. ■

Örnek 3.1-1 kontrol.c programı.

```
#include <stdio.h>

int main(void)
{
   int i = 0, j = 0;

   printf( "for girisi - i : %d\n", i );
   for ( i = 0; i < 5; i++ )
           printf( "%d", i );
   printf( "\nfor cikisi - i : %d\n", i );
```

...Örnek 3.1-1 devam

```
    /*
          for girisi - i : 0
          01234
          for cikisi - i : 5
     */
  i = 0;
  printf( "do-while girisi - i : %d\n", i );
  do
  {
     printf( "%d", i );
     i++;
  } while ( i < 5 );
  printf( "\ndo-while cikisi - i : %d\n", i );
     /*
          do-while girisi - i : 0
          01234
          do-while cikisi - i : 5
     */
  i = 0;
  printf( "do-while girisi - i : %d\n", i );
  do
     printf( "%d", i );
  while ( i++ < 5 );
  printf( "\ndo-while cikisi - i : %d\n", i );
     /*
          do-while girisi - i : 0
          012345
          do-while cikisi - i : 6
     */
  i = 0;
  printf( "while girisi - i : %d\n", i );
  while ( i < 5 )
  {
     printf( "%d", i );
     i++;
  }
  printf( "\nwhile cikisi - i : %d\n", i );

     /*
          while girisi - i : 0
          01234
          while cikisi - i : 5
     */
  i = 0;
  printf( "while girisi - i : %d\n", i );
  while ( i++ < 5 )
     printf( "%d", i );
```

...Örnek 3.1-1 devam

```
  printf( "\nwhile cikisi - i : %d\n", i );

    /*
          while girisi - i : 0
          12345
          while cikisi - i : 6
     */
  i = 0;
  printf( "sonsuz while girisi - i : %d\n", i );
  while (1)
  {
    if ( i > 5 )
      break;
    else
      printf( "%d", i );
    i++;
  }
  printf( "\nsonsuz while cikisi - i : %d\n", i );
    /*
          sonsuz while girisi - i : 0
          012345
          sonsuz while cikisi - i : 6
     */
  i = 0;
  printf( "sonsuz for girisi - i : %d\n", i );
  for (;;)
  {
    if ( i > 5 )
      break;
    else
      printf( "%d", i );
    i++;
  }

  printf( "\nsonsuz for cikisi - i : %d\n", i );
    /*
          sonsuz for girisi - i : 0
          012345
          sonsuz for cikisi - i : 6
    */
  i = 0;
  printf( "for-continue girisi - i : %d\n", i );
  for ( i = 0; i < 5; i++ )
  {
       if ( i == 2 )
          continue;
       printf( "%d", i );
```

...Örnek 3.1-1 devam

```
   }
   printf( "\nfor-continue cikisi - i : %d\n", i );
      /*
            for-continue girisi - i : 0
            0134
            for-continue cikisi - i : 5
       */
   for ( i = 0, j = 0; i < 5; i++ )
         for ( ; j < 5; j++ )
               if ( (i * j) != 16 )
                     putchar( '.' );
               else
                     goto cikis;
cikis:
   printf( "i : %d, j : %d\n", i, j );
      /*
            .....i : 5, j : 5
       */
   printf( "switch -\n" );
   for ( i = 0; i < 4; i++ )
         switch (i)
         {
            case 0:
            case 1:
                        printf ( "i < 2\n" );
                        break;
            case 2:
                        printf ( "i : 2\n" );
                        break;

            default:
                        printf ( "i : %d\n", i );
         }
      /*
            switch -
            i < 2
            i < 2
            i : 2
            i : 3
       */
   printf( "if-else -\n" );
   for ( i = 0; i < 4; i++ )
         if ( i < 2 )
            printf ( "i < 2\n" );
         else if ( i == 2 )
            printf ( "i : 2\n" );
         else
```

...Örnek 3.1-1 devam

```
            printf ( "i : %d\n", i );

      /*
            if-else -
            i < 2
            i < 2
            i : 2
            i : 3
       */
   return 0;
}
```

Çıktı

Çıktılar programda döngü çıkışlarında açıklama satırı olarak verilmiştir.

BÖLÜM 4: **Fonksiyonlar**

Fonksiyonların kolaylıkla oluşturularak verimli bir şekilde kullanılabilmesi, C dilinin en önemli özelliklerinden biridir. *Belirli bir işlevi yerine getiren ve tekrarlanan program parçaları* okunabilirliği arttırmak, değişiklik yapılmasını ve hata bulmayı kolaylaştırmak amacıyla programın bütününden ayrılarak fonksiyon haline getirilebilir. Aynı amaçla, *tekrarlanmayan fakat programın bütününden farklı işlevleri yerine getiren kısımlar* da fonksiyon haline getirilebilir. Böylece tanımlanan fonksiyon gerekli oldukça çağrılır ve çağıran bloktaki kod tekrarı önlenmiş olur. Tanımlanan fonksiyon sadece bir kez çağrılacak da olsa bütünden ayrıldığı için program üzerinde kontrol artar. Ayrıca büyük bir programın tamamını tek bir **main** bloğu içine yerleştirmek iyi bir yol değildir. Programın esas işlevi ile ilgili olmayan ayrıntıların ana programdan ayrılması kolaylık sağlar. Fonksiyonlar oluşturularak çok kapsamlı projeler küçük parçalara bölünebilir. Daha önceden yazılarak fonksiyon haline getirilmiş kodlar, bir başka programda aynı işleve ihtiyaç duyulduğunda ilgili kodları yeniden yazmaya gerek kalmadan kullanılabilir.

C dilinde giriş/çıkış işlemlerini gerçekleştirmek için deyimler bulunmaz. Bu işlemler, standart C kütüphanesinde bulunan fonksiyonlar çağrılarak gerçekleştirilir. Önceki bölümlerde yer alan örnek programlarda standart C kütüphanesinde bulunan giriş/çıkış fonksiyonları (printf, scanf gibi) kullanıldı. C kütüphanesi bir çok hazır fonksiyon içerir. Hazır kütüphane fonksiyonları kullanıldığında program *link* işlemi sırasında bu fonksiyonların kütüphanede bulunan tanımları ile birleştirilir.

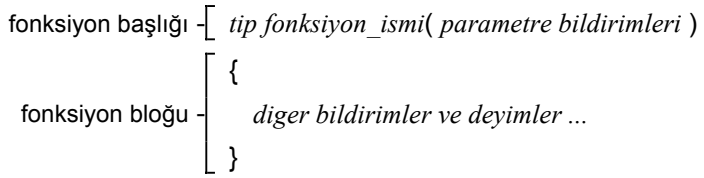
Bu bölümde fonksiyonların oluşturulmasına ve kullanımına yer verilecektir. ■

4.1 Fonksiyon Tanımı, Bildirimi, Çağırma ve Değer Döndürme

Fonksiyon, belli bir işlevi yerine getiren ve genellikle kendini çağıran program satırına bir değer döndüren deyimler grubudur.

Fonksiyon Tanımı

Fonksiyon tanımı aşağıdaki gibi yapılır:



Tanım iki kısımdan oluşur: fonksiyon başlığı ve fonksiyon bloğu. Fonksiyonun döndüreceği değerin tipi, istenen tip belirleyici fonksiyon ismi öncesine yerleştirilerek belirlenir. Dolayısıyla başlıkta yer alan *tip*, fonksiyonun döndüreceği değerin tipini belirtir. Eğer hiç bir tip belirleyici bulunmuyorsa fonksiyonun **int** tipinde değerler döndüreceği kabul edilir. Hiç bir değer döndürmeyen fonksiyonlar ise **void** tipi ile tanımlanır. Seçilen fonksiyon ismi (*fonksiyon_ismi*), fonksiyonun işlevini açıkça belirtmelidir.

Parantezler içinde virgülle ayrılmış olarak bildirilen değişkenler, *fonksiyon parametreleri* olarak adlandırılır ve çağrı sırasında fonksiyonun almayı beklediği verilerin sayısını ve tipini belirtir. Fonksiyonun hiç bir parametresi yoksa, yani fonksiyona hiç bir değer aktarılmıyor ise parantezler içinde **void** tip belirleyici kullanılır. Değer döndürmeyen ve parametresi olmayan bir fonksiyon tanımı aşağıdaki gibi yapılır:

```
void  isim( void )
{
   ...
}
```

Fonksiyon bildirimi

Fonksiyonlar da değişkenler gibi bildirilebilir. Bildirim fonksiyonun ismini, döndüreceği değerin tipini, çağrı ile aktarılan verilerin sayısını ve tipini belirtir.

Bu bildirim *fonksiyon prototipi* (yada kısaca *prototip*) olarak adlandırılır:

tip fonksiyon_ismi (*parametre tip listesi*);

Bir programda tanımlanan her fonksiyon için tekbir prototip bulunur. Prototipler, genel olarak kaynak programın başlangıç noktasına yakın ve bildirdikleri fonksiyonun ilk kullanımından önce yerleştirilir. Böylece derleyiciye programdaki çağrıları kontrol etme imkanı sağlanmış olur.

Prototipteki parantezler arasında fonksiyon parametrelerinin virgülle ayrılmış *tip listesi* yer alır. Tipleri sıralarken parametre isimleri de kullanılabilir; fonksiyon başlığında yer alan parametre bildirimleri listesi prototipte de kullanılabilir. Bu uygulama okunabilirliği arttırır, fakat C dilinin bir kuralı değildir. Eğer fonksiyonun hiç bir parametresi yoksa prototipte de **void** veri tipi kullanılır:

tip isim(void);

Fonksiyon prototipinde parantezlerin içi boş bırakılırsa derleyici hiçbir argüman kontrolu yapamaz. Eğer fonksiyon hiçbir değer döndürmüyor ise prototipte de tanımda olduğu gibi **void** tip belirleyici kullanılmalıdır. *tip* verilmediğinde ise **int** tipi değerler döndüreceği kabul edilir. Örnek programlarda kullanılan hazır C kütüphane fonksiyonlarının prototipleri, ilgili başlık dosyalarında (*header file* yada *include file*) bulunur ve programa önişlemci komutu #include kullanılarak alınır.

Fonksiyon çağırma ve değer döndürme

Yukarıda anlatılan şekilde tanımlanan ve bildirilen bir fonksiyona erişim *fonksiyon çağrısı* ile gerçekleşir. Fonksiyon oluşturmak için bloktan ayrılan program parçasının yerine fonksiyon çağrı satırı yerleştirilir. Örnek programlarda sık sık standart C kütüphanesinde bulunan hazır fonksiyonlar kullanıldı. Bunlardan biri de ekrana çeşitli formatlarda bilgi yazmak için kullanılan printf fonksiyonudur. Programlarda printf fonksiyonuna erişim, gerekli olan noktaya bu fonksiyonun çağrısı yerleştirilerek gerçekleştirilir.

Fonksiyonlar arasında veri alışverişi, çağırma ve çağıran fonksiyona dönme sırasında gerçekleşir. Çağrının yer aldığı fonksiyon (bu ana program yani **main** fonksiyonu yada bir başka fonksiyon olabilir) *çağıran fonksiyon* olarak adlandırılır. Çağrı ile erişilen fonksiyon ise *çağrılan fonksiyon* olarak adlandırılır.

Programın çalışması sırasında çağrı satırına gelindiğinde çağrılan fonksiyon bloğu içinde bulunan deyimler çalıştırılır. Fonksiyonun çalışması bittiğinde ise tekrar çağıran fonksiyona geri dönülür.

Örneklerde görüldüğü gibi fonksiyonlar çağıran fonksiyondan değer alabilirler. Fonksiyona çağrı sırasında aktarılan herhangi bir değer *argüman* olarak adlandırılır.

NOT:
Fonksiyon çağrısında yer alan (fonksiyona aktarılan) sabit değerler yada değişkenler **argüman**, **gerçek argüman** yada **gerçek parametre** olarak adlandırılır. Ayrıca tanımda parantezler içinde bildirilen değişkenler ise **parametre**, **formal parametre** yada **formal argüman** olarak adlandırılır.

Burada herhangi bir karışıklık olmaması için çağrıda yer alan değerler yada değişkenler **argüman**, tanımda bildirilen değişkenler ise **parametre** olarak adlandırılır.

Aşağıdaki satırda fonksiyon ismi ve bunu izleyen bir çift parantez kullanılarak hiçbir argüman değeri almayan ve döndürdüğü değer kullanılmayan (döndürüyor ise) fonk fonksiyonu çağrılır:

```
fonk( );
```

Bu en basit fonksiyon çağrı deyimidir. Eğer fonksiyon argüman değerleri bekliyor ise çağrı parantezler içine argümanların virgülle ayrılmış listesi verilerek yapılır:

```
fonk( argüman listesi ... );
```

Argüman listesi fonksiyon tanımındaki tip ve sayıda değişkenler, sabit değerler yada istenen tipte değerler veren ifadeler içerebilir.

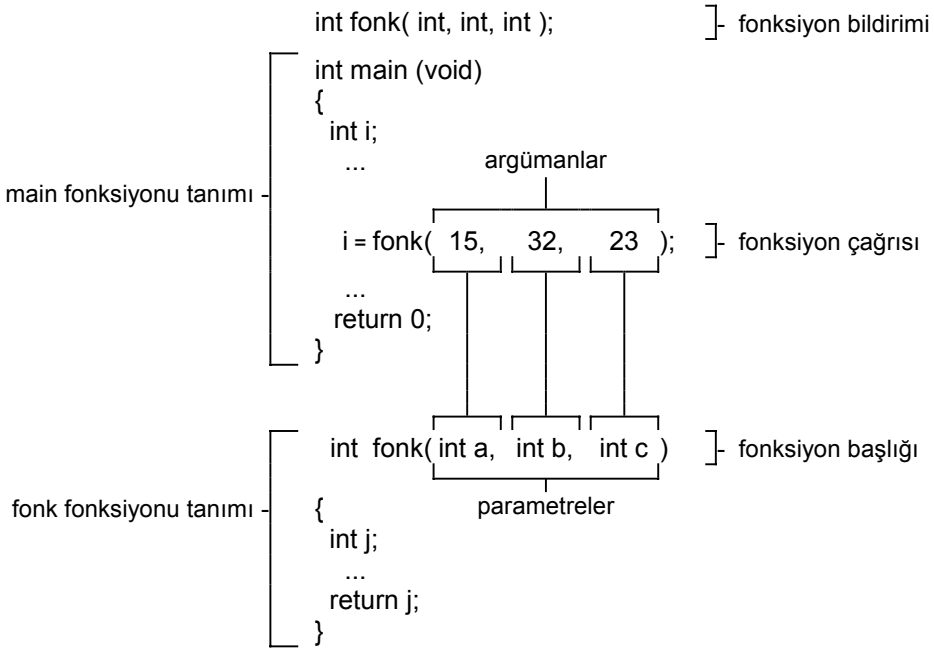
Çağrıda yer alan her bir argümana tanımdaki bir parametre karşılık gelir; birebir eşleme gerçekleşir. Eğer argüman tipleri ile parametre tipleri farklı ise argüman tipleri, atama işleminde olduğu gibi prototipteki karşılık gelen parametre tiplerine çevrilir. Aşağıda üç argüman değeri alan fonk fonksiyonu tanımlanır; **main** bloğunda yer alan çağrıdaki argümanların her biri tanımdaki parametrelerden birine karşılık gelir. Dolayısıyla a parametresi 15 değerini, b parametresi 32 değerini ve c parametresi de 23 değerini alır.

Bir fonksiyona argüman olarak herhangi bir basit değişken, union yada yapı değişkeni, sabit bir değer, dizi yada fonksiyon aktarılabilir (union ve yapı değişkenleri, izleyen bölümlerde ayrıntılı olarak anlatılacaktır).
Çağıran fonksiyona değer döndürmek için **return** deyimi kullanılır:

```
return ifade;
```

return deyimi hem fonksiyonun çalışmasını sona erdirebilir (kontrol çağrı satırına geri döner), hem de çağıran deyime değer döndürebilir. Eğer **return** deyiminde herhangi bir ifade bulunmuyor ise hiçbir değer döndürmez ve bulunduğu satıra gelindiğinde sadece fonksiyonun çalışmasını sona erdirerek, çağrı satırına dönülmesini sağlar. Bir fonksiyonda birden fazla **return** deyimi olabilir ve fonksiyon içinde herhangi bir yerde

bulunabilir. Fakat **return** deyimi ile sadece tek bir değer döndürebilir ve fonksiyonun her çağrılışında sadece bir **return** deyimi çalıştırılır.



Fonksiyon hiçbir değer döndürmese de, kapanış parantezine gelmeden önce herhangi bir koşula bağlı olarak fonksiyonu durdurmak için **return** deyimi kullanılabilir. Fakat fonksiyon hiçbir değer döndürmüyor ise **return** deyimi kullanımı zorunlu değildir.

Çağıran fonksiyon döndürülen değeri kullanmayabilir. Fakat bazı durumlarda (özellikle fonksiyonun işlediğini gösteren bir bilgi veya bir hata kodu döndüren bazı kütüphane fonksiyonlarında) kullanılması gerekebilir.
Örnek olarak printf fonksiyonu çağrı satırına ekrana yazılan karakter sayısını yada hata oluşması durumunda negatif değer döndürür. Fakat örnek programlarda görüldüğü gibi döndürdüğü değer kullanılmayabilir. Eğer fonksiyon değer döndürüyor ise ve bu değer kullanılacak ise fonksiyon çağrısı döndürülen tipteki bir değişkenin yer aldığı bir atama deyimine yada döndürdüğü değerin kullanıldığı bir ifadeye yerleştirilebilir. Böylece döndürülen değer uygun tipteki bir değişkene atanmış olur yada ifade içinde kullanılabilir. Örnek olarak;

degisken = *fonksiyon_ismi*(*arguman listesi*);

return *ifade*; deyiminde yer alan *ifade*'nin (*ifade* herhangi bir değişken, değişken yada sabitlerden oluşan aritmetik bir ifade olabilir) değerinin tipi ne olursa olsun, döndürülmeden önce fonksiyonun tipine çevrilecektir. Fonksiyonda hiçbir **return** deyimi bulunmadığı zaman fonksiyon bloğunda yer alan deyimler çalıştırıldıktan ve bloğu

kapatan paranteze (sağ parantez: }) gelindikten sonra yine çağrı satırına geri dönülür. Böylece kontrol çağıran fonksiyona geçmiş olur. Fakat eğer fonksiyon kapanış parantezine gelerek geri dönüyorsa yada bir çağrıda değer döndürdüğü halde başka bir çağrıda değer döndürmüyorsa hatalı çalışıyor olabilir. Her iki durumda da, bu fonksiyon çağıran bloğa anlamsız değerler döndürür.

Bir fonksiyon sabit bir değer, union yada yapı değişkeni değeri veya herhangi bir adres değeri döndürebilir. Fakat bir başka fonksiyon yada dizi döndüremez.

Fonksiyon çağrısı, program içinde iki şekilde bulunabilir: kendi başına bir program deyimi olarak yada herhangi bir ifadenin içinde, o ifadenin bir parçası olarak. İlk durumda, döndürülen değer kullanılmamış olur. İkinci durumda ise döndürülen değer ifade içinde kullanılacaktır. Ayrıca çağrı, başka bir fonksiyon çağrısında argüman olarak da kullanılabilir. Böylece döndürülen değer bir başka fonksiyona aktarılmış olur.

Argümanların aktarılması: değer-ile-çağırma ve referans-ile-çağırma

Programda çağrı satırına gelindiğinde çağrıda yer alan her bir argümanın değeri fonksiyondaki karşılık gelen parametreye atanır. Bir başka deyişle, her parametre karşılık gelen argüman değerinin kopyasını alır. Argüman değerlerinin aktarılması ile gerçekleşen bu şekildeki bir çağrı *değer-ile-çağırma* olarak adlandırılır. Değer-ile-çağırma gerçekleşiyor ise fonksiyon parametre değerlerini değiştirebilir, fakat çağıran blokta yer alan argüman değerlerini etkileyemez.

Argümanlar dizi ismi yada fonksiyon ismi olduğunda karşılık gelen parametreler adres değişkenleridir (izleyen bölümlerde ayrıntılı olarak anlatılacaktır). Bu durumda argümanların değerleri yerine adresleri aktarılır. Bu şekilde gerçekleşen fonksiyon çağrısı *referans-ile-çağırma* olarak adlandırılır ve çağıran bloktaki argüman değerlerine erişim mümkün olur.

Argüman değerlerinin aktarılması, aktarılan değerlere erişim ve çağıran bloğa değer döndürme işlemleri için gerekli olan kodlar, derleyici tarafından oluşturularak çağıran ve çağrılan fonksiyonlara yerleştirilir.

Fonksiyon parametreleri ve fonksiyon bloğu içinde bildirilen değişkenler çağrı ile bloğa girildiğinde *otomatik* olarak oluşturulur ve bloktan çıkıldığında yok olurlar.

Bu nedenle fonksiyon başlığında bildirilen parametreler ve fonksiyon bloğu içinde bildirilen değişkenler, o fonksiyonun *lokal* (*local*: yerel) değişkenleridir ve sadece o fonksiyon bloğu içinden erişilebilirler (**main** ve diğer fonksiyonlar erişemez).

Parametre değerleri değiştiğinde çağıran fonksiyondaki değerler etkilenmez dolayısıyla parametreler blok içinde herhangi bir lokal değişken gibi kullanılabilir.

C dilinde, tüm blokların dışında tanımlanan değişkenler *global* (genel) değişkenler olarak adlandırılır ve varlıklarını lokal değişkenlerin aksine programın çalışması boyunca sabit bellek adreslerinde sürdürürler. Bu değişkenlere program içinde yer alan tüm fonksiyonlardan erişilebilir. Fonksiyon çağrıları sırasında değerlerini korudukları için fonksiyonlar arasında veri taşınmasında kullanılabilir. Fakat yukarıda da belirtildiği gibi verilerin argümanlar aracılığı ile aktarılması, hem fonksiyondan çıkıldığında bu değişkenler yok olduğı için bellek tasarrufu sağlar, hem de argüman değerleri üzerindeki işlemler bu değerlerin kopyaları üzerinde gerçekleştirildiği için çağıran bloktaki değerleri etkilemez. Bu nedenlerden dolayı fonksiyonlara veri aktarılmasında sabit bellek adreslerinde bulunan global değişkenlerin kullanılması tercih edilmez. Global değişkenlerin kullanımı daha sonra ayrıntılı olarak anlatılacaktır. Bazı derleyiciler argüman değerlerinin mikroişlemcinin **register** birimlerine yerleştirilmesini sağlar.

Örnek 4.1-1'de yer alan program, Fahrenhayt olarak girilen sıcaklık değerini Santigrat'a çevirir. Ekrandan 33.8 değeri girildiğinde program 1.0 değerini verir. **main** fonksiyonu tanımı programda görüldüğü gibi başlık ve fonksiyon bloğundan oluşur. Programda, sıcaklık çevirme işlemi aşağıdaki formül kullanılarak yapılır:

$$t\_c = (t\_f - 32.0) / 1.8$$

Örnek 4.1-1 fonk1.c programı.

```
                                #include <stdio.h>

main fonksiyonu başlığı -[ int main(void)
                         [ {
                         |   float t_c, t_f;
                         |
                         |   printf("Fahrenhayt : ");
main fonksiyonu bloğu -  |   scanf("%f", &t_f );
                         |
                         |   t_c = ( t_f - 32.0 ) / 1.8;
                         |
                         |   printf("Santi. : %.1f\n", t_c);
                         |   return 0;
                         [ }
```

Bilgi Girişi

Fahrenhayt : **33.8**

Çıktı

Santi. : 1.0

Formülün bulunduğu kısım fonksiyon haline getirilerek **main** bloğundan ayrılabilir. Bu durumda çevirme işleminin yapıldığı satıra, bu işlemi gerçekleştiren fonksiyonun çağrısı

yerleştirilecektir. Fonksiyon çağrısı, fonksiyon isminden sonra parantezler içinde aktarılmak istenen argümanların verilmesi ile gerçekleşir. Fonksiyonun kendine aktarılan Fahrenhayt değerini formüle göre Santigrat ölçeğine çevirerek çağrı satırına geri döndürmesi ve döndürülen değerin de t_c değişkenine atanması gerekir. Buna göre bir **float** argüman değeri alan ve çağıran bloğa bir **float** değer döndüren SantiGrat fonksiyonu fahr parametresi ile tanımlanabilir:

SantiGrat fonksiyonu başlığı -
```
float SantiGrat( float fahr )
```

SantiGrat fonksiyonu bloğu -
```
{
   float  c;
   c = ( fahr - 32.0 ) / 1.8;
   return c;
}
```

Fonksiyon başlığında fahr parametresi bildirilir. Fonksiyon tanımının oklu parantezler arasında kalan ve fonksiyon bloğu olarak adlandırılan kısmında ise **float** tipindeki c değişkeni bildirilir.

Örnek 4.1-2'de **main** bloğu içinde Fahrenhayt değerleri için bildirilen t_f değişkeni ve Santigrat değerleri için bildirilen t_c değişkeni, **main** fonksiyonunun lokal değişkenleridir. Ayrıca SantiGrat tanımında bildirilen fahr ve c değişkenleri de bu fonksiyonun lokal değişkenleridir ve main bloğundan erişilemezler. Fonksiyon argümanları değerleri ile aktarılır; yani çağrılan fonksiyon çağrıda yer alan her argümanın değerinin kendine ait geçici bir kopyasını alır. Dolayısıyla fonksiyon, çağıran bloktaki esas argümanlara erişemez. SantiGrat fonksiyonu, çağrıda argüman olarak yer alan t_f argümanına erişemez. Sadece bu argümanın değeri fahr parametresine ilk değer olarak atanır. Böylece çağıran blokta kullanılan değişken isimleri, çağrılan blokta da değişken ismi olarak kullanılabilir. Ayrıca farklı bloklarda aynı isimlerin kullanılmasından doğabilecek hatalı erişimler önlenmiş olur.

Örnek 4.1-2 fonk2.c programı.

```
                                    #include <stdio.h>
SantiGrat fonksiyonu bildirimi, prototipi -[ float SantiGrat( float fahr );
                                    int main(void)
                                    {
lokal değişken bildirimleri -[        float t_c, t_f;
printf fonksiyonu çağrısı -[          printf("Fahrenhayt : ");
scanf fonksiyonu çağrısı -[           scanf("%f", &t_f );
SantiGrat fonksiyonu çağrısı -[       t_c = SantiGrat( t_f );
printf fonksiyonu çağrısı -[          printf("Santi. : %f\n", t_c);
                                      return 0;
                                    }
                                    float SantiGrat( float fahr )
                                    {
lokal değişken bildirimleri -[        float  c;
                                      c = ( fahr - 32.0 ) / 1.8;
çağıran bloğa değer döndürme -[       return c;
                                    }
```

Bilgi Girişi

Fahrenhayt : **33.8**

Çıktı

Santi. : 1.0

SantiGrat tanımında, return c; deyimi ile **return** anahtar sözcüğü kullanılarak **main** fonksiyonundaki çağrı satırına SantiGrat fonksiyonunun lokal değişkeni c'nin değeri döndürülür. **main** bloğundan SantiGrat fonksiyonuna yapılan çağrı, bir **float** argüman değeri aktarır.

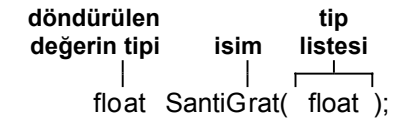
Programdaki **main** fonksiyonu tanımı standart kütüphane fonksiyonları olan printf ve scanf çağrılarını içerir. Bu fonksiyonların prototiplerinin bulunduğu başlık dosyası stdio.h, #include önişlemci komutu kullanılarak program içine alınır. Bu fonksiyonların önceden derlenmiş tanımları link işlemi sırasında program ile birleştirilir. SantiGrat fonksiyonu t_f argümanı ile çağrılır. SantiGrat fonksiyonunun döndürdüğü değer ise t_c değişkenine atanır.

Programdaki prototip aşağıdaki şekilde de yazılabilir:

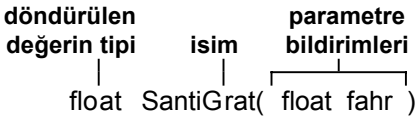
```
float SantiGrat( float );
```

Bu bildirim ile yine fonksiyonun bir **float** değer almayı beklediği ve sonuç olarak bir **float** değer döndüreceği belirtilir. Özetle;

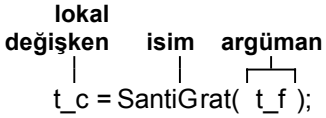
SantiGrat fonksiyonu prototipi:



SantiGrat fonksiyonu başlığı:



SantiGrat fonksiyonu çağrısı:



Program, SantiGrat fonksiyonu çağrısı printf fonksiyonu çağrısına argüman olarak kullanılarak yazılabilir. printf fonksiyonu çağrısı iki argüman aktarır; format dizgisi ve SantiGrat çağrısının döndürdüğü **float** değer:

```
printf("Santi. : %f\n", SantiGrat( t_f ) );
```

Ayrıca formül **return** deyimine yerleştirilebilir. Böylece daha kısa bir program elde edilir:

```
float SantiGrat( float fahr )
{
  return ( fahr - 32.0 ) / 1.8;
}
```

Bu programda yer alan formül, fonksiyon oluşturmak için pek uygun değildir. Bu işlem, #define önişlemci komutu ile basit bir makro tanımlayarak gerçekleştirilebilir:

```
#define SantiGrat( fahr )     (fahr - 32.0) / 1.8
```

Fonksiyon tanımları, program içinde herhangi bir sırada bulunabilir. Program çalışmasını **main** bloğu ile başlatır ve bu bloğun kapanış parantezinde sona erdirir. Bu nedenle **main** fonksiyonu genel olarak en başta yer alır ve diğer fonksiyon tanımları **main** bloğu sonrasına yerleştirilir. Programın okunabilirliğini arttıran bu uygulama, C dilinin kurallarından biri değildir. Bir programda herhangi bir fonksiyon *tanımlandığı yerden önce çağrılıyor* ise mutlaka çağıran blok öncesinde bildirimi yapılmalıdır. Aksi taktirde program satırındaki ilk kullanımı (çağrı sırasında) ile bildirilmiş olur ve **int** tipi değerler döndüreceği kabul edilir. Programda hata oluşmaması için bu fonksiyonun tanımında döndürdüğü değerin tipi olarak **int** verilmiş yada hiçbir tip belirtilmemiş olmalıdır (tip belirtilmediği için **int** tipi değerler döndüreceği varsayılır).

Aşağıdaki program satırlarında fonk fonksiyonu tanımı çağrının yer aldığı **main** fonksiyonu sonrasına yerleştirilmiştir:

```
int main(void)
{
  double x;
  x = fonk(...);
    ...
}
double fonk(...)
{
    ...
}
```

Prototip bulunmadığı için **main** bloğunda yer alan ilk çağrı ile **int** tipi değerler döndüreceği kabul edilerek bildirilmiş olur. Fakat tanımda **double** değerler döndüreceği belirtilir. Bu durumda hata oluşur. Aşağıdaki şekilde görüldüğü gibi fonksiyon tanımı **main** öncesine yerleştirilerek (1.program) yada **main** öncesinde bildirimi yapılarak (2.program) bu hata önlenebilir. Fakat derleyicinin çağrıları kontrolüne olanak sağladığı için prototip kullanımı tercih edilir:

1.program

```
/* fonk tanımı */
double fonk(...)
{
  ...
}

/* main tanımı */
int main(void)
{
  double x;
  x = fonk(...);
    ...
}
```

yada

2.program

```
/* prototip */
double fonk(...);

/* main tanımı */
int main(void)
{
  double x;
  x = fonk(...);
    ...
}

/* fonk tanımı */
double fonk(...)
{
  ...
}
```

Örnek 4.1-2'de SantiGrat fonksiyonu tanımı bu fonksiyonu çağıran **main** bloğundan sonra yerleştirilmiştir. Bu nedenle bildirimi **main** bloğundan önce yapılır. Böylece derleyici fonksiyon prototipinde verilen bilgileri kullanarak bu fonksiyona yapılan her çağrıda tip kontrolü (aktarılan argümanların ve döndürülen değerin tipi) yapar ve argüman sayısını kontrol eder.

Aşağıda iki **int** parametreye sahip ve yine **int** tipi değerler döndüren fonk fonksiyonu tanımlanır. fonk fonksiyonu **main** bloğu içinde iki **double** değer ile çağrılır ve döndürdüğü değer bir **double** değişkene atanır:

```
#include <stdio.h>

int main(void)
{
  double Toplam, a = 1, b = 2;
  Toplam = fonk( a, b );
  printf("Toplam : %1.1f\n", Toplam );
  return 0;
}

int fonk( int x, int y )
{
  return x+y;
}
```

fonk fonksiyonu tanımı çağrının yer aldığı **main** fonksiyonu sonrasında yer alır ve **main** öncesinde prototipi bulunmaz. Derleyici çağrı satırına geldiğinde fonk fonksiyonu henüz bildirilmemiş olduğu için **int** tipi değerler döndüreceği kabul edilir. Ayrıca argüman ve parametre tipleri uymadığı halde gerekli tip çevirmeleri yapamaz. Bu nedenle veri kaybı olur ve ekrana 0.0 değeri yazılır. Bu durum aşağıdaki görüldüğü gibi **main** fonksiyonu öncesinde fonk fonksiyonu bildirilerek önlenebilir. Böylece derleyici prototipte verilen bilgileri kullanarak gerekli tip çevirmeleri gerçekleştirir ve ekrana 3.0 değeri yazılır:

```
#include <stdio.h>
int fonk( int, int );
int main(void)
{
   double Toplam, a = 1, b = 2;
   Toplam = fonk( a, b );
   printf("Toplam : %1.1f\n", Toplam );
   return 0;
}

int fonk( int x, int y )
{
   return x+y;
}
```

Bir fonksiyon aynı program içinde yer alan diğer tüm fonksiyonlardan çağrılabilir. Ayrıca kendini çağıran C fonksiyonu (**main** hariç) tanımlanabilir. C dilinde, fonksiyonlar birbirlerini çağırabilirler, fakat bir fonksiyon başka bir fonksiyon içinde tanımlanamaz.

Aşağıdaki fonk1 ve fonk2 fonksiyonları **main** tarafından çağrılabilir ve ayrıca birbirlerini de çağırabilirler. Fakat **main** fonksiyonunu çağıramazlar:

```
    ...
int main(void)
{
    ...
}

fonk1( )
{
    ...
}
fonk2( )
{
    ...
}
```

main fonksiyonu:

Örneklerde de görüldüğü gibi her C programı en az bir fonksiyon içerir ve bu fonksiyon **main** olarak adlandırılır. **main** bir anahtar sözcük değildir ve programın başka bir yerinde kullanılamaz. Ayrıca her C programında sadece tek bir **main** fonksiyonu bulunabilir. Bütün çalıştırılabilir program deyimleri **main** yada bir başka fonksiyonun içinde yer alır.

main, **int** tipi değerler döndüren bir fonksiyon olarak komut satırı bilgilerinin program içinde kullanılmasını sağlayan parametreler ile tanımlanır.

Program çalıştığında **main** fonksiyonunun bu parametrelerine, komut satırına girilen bilgiler aktarılır:

```
int main( int argc, char **argv )
{
  ...
 return 0;
}
```

Komut satırı argümanlarının kullanılmadığı durumlarda parametre bildirimleri yerine **void** kullanılır. **main** fonksiyonu bir başka fonksiyon tarafından çağrılamaz ve bu nedenle çağıran fonksiyona değer döndürmesi sözkonusu olamaz. Fakat **main** fonksiyonu çalışmasını sona erdirdiğinde, işletim sistemine durum ifadesi için **return** deyimi ile bir tamsayı değer döndürebilir. Bu değer program içindeki hata kontrol kodlarına bağlı olarak düzenlenir ve genel olarak normal bir çıkışı ifade etmek için 0 (sıfır), hatalı bir çıkışı ifade etmek için ise sıfır hariç bir tamsayı olarak seçilir. İşletim sisteminde bulunan ve döndürülen değere erişebilen bir toplu işlem komutu kullanılarak programın hatasız çalışıp çalışmadığı saptanabileceği gibi başka komut grupları da çalıştırılabilir. **return** deyimi yerine standart C kütüphanesinde bulunan exit fonksiyonu kullanılarak programdan çıkılabilir.

Fonksiyon Argümanı Olarak Diziler

Bir dizi, aynı veri tipindeki bir grup değişkenden oluşur. Bu değişkenlerin her biri *dizi elemanı* olarak adlandırılır. Dizi elemanları bellekte birbirini izleyen eşit büyüklükte alanlarda bulunur. Elemanlara erişim ortak bir değişken ismi yani dizi ismi kullanılarak gerçekleştirilir. Aşağıda *tip* tipinde *n* elemandan oluşan *dizi* dizisi bildirilir:

tip dizi [*n*];

Her dizi elemanının bir indeks numarası vardır. Dizinin başlangıç elemanının indeks numarası 0'dır. Dizinin son elemanının indeks numarası ise *n - 1*'dir. Dizi elemanlarına

dizi_ismi[*indeks_no*] ifadesi ile erişilir. Dizi elemanlarına değer atama izleyen bölümlerde anlatılacağı gibi bildirim sırasında yada program içinde *dizi_ismi* [*indeks_no*] = *deger*; atama deyimi ile yapılabilir. *deger*, *tip* tipinde değer döndüren bir ifade, sabit yada değişken olabilir. Köşeli parantezler ([ ]) *indeks operatörü*dür. Diziler izleyen bölümlerde ayrıntılı olarak anlatılacaktır.

C dilinde dizi ismi (yukarıdaki bildirimdeki *dizi*), dizinin başlangıç elemanının bellek adresini veren sabittir. Bu nedenle, dizi ismi bir fonksiyon çağrısında argüman olarak yer aldığında dizi elemanlarının değerleri yerine dizinin başlangıç adresi aktarılır (referans-ile-çağırma). Çağrılan fonksiyon dizinin bellek adresini bildiği için çağıran bloktaki elemanların değerlerini değiştirebilir.

Aşağıdaki örnekte argüman olarak **int** tipi elemanlardan oluşan bir dizi ve bu dizinin eleman sayısı olarak yine **int** tipi bir değer bekleyen Dizix2 fonksiyonu tanımlanır. **main** bloğu içinde bildirilen ve ilk değer ataması yapılan 5 elemanlı tamsayı dizi dizisi (dizi dizisinin tipi **int**[5]'dir), bu fonksiyona argüman olarak aktarılır.

Örnek 4.1-3 fonkdizi.c programı.

```
#include <stdio.h>
#define DiziBoy(s)sizeof s / sizeof s[0]
void Dizix2( int[ ], int );
int main(void)
{
   int dizi[ 5 ] = { 1, 2, 3, 4, 5 };
   int i;
   /* cagri oncesi dizinin ekrana listelenmesi */
   for ( i = 0; i < DiziBoy( dizi ) ; i++ )
         printf("dizi [%d] : %d\n", i, dizi [ i ] );
   /* fonksiyon cagrisi */
   Dizix2( dizi, DiziBoy(dizi) );
   /* cagri sonrasi dizinin ekrana listelenmesi */
   for ( i = 0; i < DiziBoy(dizi) ; i++ )
         printf("dizi [%d] : %d\n", i, dizi [ i ] );
   return 0;
}
void Dizix2( int dizi[ ], int Boy )
{
   int n;

   for ( n = 0; n < Boy; n++ )
         dizi [ n ] = dizi [ n ] * 2;
}
```

...Örnek 4.1-3 devam

Çıktı

```
dizi [ 0 ]  :  1
dizi [ 1 ]  :  2
dizi [ 2 ]  :  3
dizi [ 3 ]  :  4
dizi [ 4 ]  :  5
dizi [ 0 ]  :  2
dizi [ 1 ]  :  4
dizi [ 2 ]  :  6
dizi [ 3 ]  :  8
dizi [ 4 ]  :  10
```

Fonksiyon çağrısında ilk argüman olarak dizi ismi dizi ve ikinci argüman olarakta dizinin eleman sayısını veren DiziBoy(dizi) makrosu yer alır. İndeksi olmayan dizi ismi (dizi), dizinin başlangıç elemanının adresini veren adres sabiti olduğu için aktarılan değer de yine başlangıç elemanının bellek adresidir ve bu adres değerinin kopyası fonksiyon tanımında bildirilen adres değişkenine (yine aynı isimdeki fonksiyon parametresi dizi) atanır. Böylece Dizix2 fonksiyonu, dizinin çağıran bloktaki değerlerine erişebilir. Programda dizinin elemanlarının değerleri, Dizix2 fonksiyonu çağrısından önce ve sonra listelenir. Dizix2 fonksiyonu içinde elemanların değerleri 2 ile çarpılır ve tekrar diziye yerleştirilir. Çıktıda da görüldüğü gibi çağrı sonrası elemanların değerleri değişmiştir.

Örnekte **main** bloğunda bildirilen dizi bir sabit, Dizix2 bloğundan bildirilen dizi ise bir adres değişkenidir. Fonksiyon parametresi olarak bildirilen bir dizi için bellek alanı ayrılmaz. Bunun yerine aktarılan adres değerini alabilecek bir adres değişkeni oluşturulur. Bu nedenle fonksiyon parametresi bir tek-boyutlu dizi olduğunda, tanım ve prototipteki parametre bildirimlerinde köşeli parantezler arasında dizinin eleman sayısının verilmesi gereksizdir. Parametre iki boyutlu bir dizi olduğunda ise ikinci boyuttaki eleman sayısı verilmeyebilir. Fakat blok içinde bildirilen bir dizi için boyutlar bellidir ve derleme sırasında bellek alanı ayrılır.

Sonuç olarak bir fonksiyona çağrı ile bir dizinin tamamının aktarılması mümkün değildir. Ancak dizinin herhangi bir elemanının değeri yada adresi aktarılabilir.

Fonksiyon Argümanı olarak Fonksiyon İsmi

C dilinde bir fonksiyonun ismi, o fonksiyonun adresini verir. Dolayısıyla bir fonksiyon ismi başka bir fonksiyon çağrısında argüman olarak yer aldığında aktarılan değer bu fonksiyonun adresidir.

Aşağıdaki örnekte argüman olarak fonksiyon ismi f2 kullanılarak, f1 fonksiyonu çağrılır. f1 fonksiyonu tanımında bildirilen p parametresi, hiçbir argüman değeri almayan ve hiçbir değer döndürmeyen fonksiyona işaret eden adres değişkenidir ve çağrı ile f2

fonksiyonunun adresini alır (fonksiyona işaret eden adres değişkeni, izleyen konularda anlatılmıştır):

```
/*
 * fonkadr.c
 */
#include <stdio.h>

void f1(  void(*)(void) );
void f2(  void );

int main( void)
{
  f1( f2 );  /* arguman olarak fonksiyon ismi */
  return 0;
}
/* p parametresi :  fonksiyona isaret eden adres degiskeni */
void f1( void (*p)(void) )
{
  (*p)( );
}
void f2( void )
{
  printf("f2 ...\n");
}

Çıktı:
f2 ...
```

Fonksiyonlara her çeşit verinin adresi aktarılabileceği için argümanlar dizi ismi yada fonksiyon ismi olmadığı durumlarda da, çağıran bloktaki argüman değerlerine erişim yada verimli ve hızlı programlar oluşturmak amacıyla referans-ile-çağırma gerçekleştirilebilir. Örneğin bir yapı değişkeninin kopyası yerine bellek adresi aktarılabilir. Özellikle argüman olarak büyük bir yapı aktarılmak istendiğinde, tüm yapının kopyası yerine adresi aktarılarak sınırlı *stack* alanı daha az kullanılabilir. referans-ile-çağırma, daha sonraki bölümlerde ayrıntılı olarak incelenecektir. ■

4.2 Değişkenlerin Görünürlük Alanı ve Varolma Süresi

Bir değişkenin program içindeki kullanımına etki eden iki önemli özelliği, o değişkenin *görünürlük alanı (scope* yada *visibility)* ve *varolma süresi*'dir (*lifetime* yada *duration*). Bir değişkenin görünürlük alanı, program içinde değişkenin değerine erişilebilen alandır. Varolma süresi ise o değişkenin program içinde var olduğu ve değerini koruduğu süredir.

C programları değişken ve fonksiyon tanımlarından oluşur. Görünürlük alanı ve varolma süresi, değişken tanımının veya bildiriminin fonksiyon bloğu içinde yada dışında

bulunuyor olmasına göre belirlenir. Bu nedenle fonksiyonlar, değişkenlerin görünürlük alanı ve varolma süresi üzerinde en önemli etkiye sahiptirler.

Global ve Lokal Değişkenler

Bir değişkene, bulunduğu dosya içinde her yerden ve başka dosyalardan erişilebilmesi *global görünürlük* olarak; fonksiyonların dışında tanımlanan değişkenler de bu şekilde erişilebildikleri için *global değişkenler* (*global variable. global:*genel) olarak adlandırılır. Bu değişkenler tüm blokların dışında bulunduğu için harici değişkenler (*external*) olarak da adlandırılırlar.

Daha önce de belirtildiği gibi fonksiyonlar birbiri içinde tanımlanamazlar ve fonksiyonlara erişim doğal olarak global seviyede gerçekleşir. Dolayısıyla uygun şekilde tanımlanan ve bildirimi yapılan bir fonksiyon, bulunduğu dosya içindeki tüm bloklardan ve diğer dosyalardan çağrılabilir.

Fonksiyon parametreleri ve fonksiyon bloğu içinde bildirilen değişkenler, *lokal değişkenler* (*local variable. local:*yerel yada lokal) olarak adlandırılır. Çünkü bu değişkenlere erişim sadece bildirildikleri blok içinde mümkündür (*lokal görünürlük*). Blok içinde erişilebildikleri için dahili değişkenler (*internal*) olarak da adlandırılırlar.

Aşağıdaki örnekte **main** bloğu içinde lokal değişken i bildirilir. Bu değişken **main** bloğu dışında tanımsızdır ve erişim sadece **main** fonksiyonunu sınırlayan parantezler içinde mümkündür. Dolayısıyla i değişkenine, fonk fonksiyonu tarafından yapılan erişim başarısız olur. Sonuç olarak program derlenemez ve derleyici hata verir:

```
#include <stdio.h>
void fonk( void );
int main(void)
{
  /*  lokal degisken bildirimi  */
  int i = 9;
  fonk();
  return 0;
}
void fonk( void )
{
  printf("i : %d\n", i );
}
```

i değişkenine fonk bloğu içinde erişim ancak bu değişken global seviyede bildirildiğinde gerçekleşebilir. Örnek 4.2-1'de bu amaçla **main** ve fonk fonksiyonlarının *dışında ve öncesinde*, global değişken i tanımlanır. Böylece her iki fonksiyon da aynı değişkene erişebilir.

Örnek 4.2-1 global1.c programı.

```
#include <stdio.h>
void fonk( void );
int i;                  /* global degisken tanimi */
int main(void)
{
   i = 9;
   fonk( );
   return 0;
}
void fonk( void )
{
   printf("i : %d\n", i );
}
```

... Örnek 4.2-1 devam

Çıktı

```
i : 9
```

Global değişkenlere, programdaki tüm fonksiyonlardan ve hatta birden fazla dosyadan oluşan programlarda diğer dosyalardan da erişilebilir. Dolayısıyla değişkenler global seviyede bildirildiğinde, hatalı bir erişim değişkenin değerini değiştirebilir. C dilinin en önemli özelliklerinden biri, programların fonksiyonlar oluşturularak parçalara (modüllere) bölünebilmesi ve oluşturulan fonksiyonların ihtiyaç duyuldukça tekrar kullanılabilmesi, dolayısıyla modüler programlamanın çok kolay uygulanabilmesidir. Global görünürlük, fonksiyonlar arasında çok sayıda veri bağlantısı oluşmasına sebep olabilir. Sonuç olarak, fonksiyonlar arasında veri transferi global değişkenler aracılığı ile yapıldığında fonksiyonların tekrar kullanılabilirliği azalır. Tüm bu nedenlerden dolayı global değişkenlerin kullanımı tercih edilmez ve veriler argümanlar aracılığı ile aktarılır. Fakat çok sayıda verinin aktarıldığı uygulamalarda, uzun argüman listeleri yerine global değişkenlerin kullanımı daha pratik olacaktır. Aşağıda i değişkeninin değeri fonk fonksiyonuna argüman olarak aktarılır. Böylece, bu değişkenin değeri fonk bloğu içinde kullanılabilir.

Bu program da yine aynı çıktıyı verir:

```
/*
 * lokal.c programi.
 */
#include <stdio.h>
void fonk( int );
int main(void)
{
  int i = 9;
  fonk( i );
  return 0;
}
void fonk( int i )
{
  printf("i : %d\n", i );
}
```

Bu programda fonk fonksiyonu başlığında bildirilen i parametresi ile **main** bloğu içinde bildirilen i değişkeni ilgisizdir. Her ikisi de içinde bulunduğu bloğun lokal değişkenidir. Fakat okunabilirliği arttırmak amacıyla parametre bildiriminde farklı isimler kullanılmalıdır.

fonk bloğu içinde bildirilecek olan diğer bir i değişkeni de, **main** bloğunda bildirilen aynı isimdeki değişken ile ilgisiz olur ve sadece fonk içinde erişilebilir.

Blok Yapısı kullanılarak Görünürlük Alanının Kısıtlanması

Fonksiyonlar birbirleri içinde tanımlanamazlar fakat bir fonksiyon bloğu içinde başka bloklar yer alabilir. Oklu parantezlerle sınırlanmış bildirimler ve deyimler, *deyim bloğu* yada *program bloğu* olarak adlandırılır. Değişken bildirimi içermeyen ve sadece deyimlerden oluşan bir blok ise, *bileşik deyim* olarak adlandırılır.

Değişkenlerin görünürlük alanları, çeşitli şekillerde kısıtlanabilir. Herhangi bir program bloğu içinde bildirilen değişkene sadece o blok içinde erişilebilir. Oklu parantezlerin her çifti, içinde bildirilen değişkenlerin Görünürlük Alanını kısıtlar. Aşağıdaki örnekte, **main** fonksiyonu içinde bulunan blokta i değişkeni bildirilir ve ilk değer ataması yapılır.

Örnek 4.2-2 blok1.c programi.

```
#include <stdio.h>
int main(void)
{
   int i = 9;       /* lokal degisken bildirimi   */
   printf("dis blok, i : %d\n", i );
   {
      int i;            /* lokal degisken bildirimi   */
      i = 20;
      printf("ic blok, i : %d\n", i );
   }
   printf("dis blok, i : %d\n", i );
   return 0;
}
```

Çıktı

```
dis blok,   i  :  9
ic   blok,  i  :  20
dis blok,   i  :  9
```

Çıktıda da görüldüğü gibi, ikinci bloğu başlatan sol parantez sonrası bildirilen yeni i değişkeni, dış bloktaki aynı isimde bildirilen i değişkeni ile ilgisizdir ve görünürlük alanı bloğu sona erdiren sağ parantez ile sınırlıdır. Blok yapısı kullanılarak görünürlük alanının kısıtlanmasının bazı avantajları vardır:

- Özellikle büyük programlarda, değişkenin bildirimi ile ilk kullanımı arasındaki mesafe uzun olduğunda, bu iki nokta arasında gidip gelmek güçleşir. Fakat değişken ilk kullanıldığı yere yakın bildirildiğinde, aradaki mesafe kısaltılmış olacağı için program kodunun okunabilirliği artar.

- Aşağıdaki **if** deyiminde, i değişkenine test ifadesi gerçekleştiği taktirde ihtiyaç olacaktır. Böylece, test ifadesi gerçekleşmez ise bu değişken için bellek alanı ayrılmamış olur:

```
    ...
if  (...)
{
  int i;
    ...
}
    ...
```

Ayrıca i değişkeni, programda kullanıldığı yerden önce görülmediği için yine program kodunun okunabilirliği artar.

Yukarıdaki örnekte, *blok yapısı* kullanılarak lokal değişkenlerin görünürlük alanı kısıtlandı. Blok yapısı, global değişkenlere de uygulanabilir.

Örnek 4.2-3'de, tamsayı a ve b global değişkenleri bildirilir ve bu değişkenlere ilk değer olarak sırasıyla 10 ve 20 değerleri atanır:

Örnek 4.2-3 blok2.c programı.

```
#include <stdio.h>

void fonk( double );

int a   = 10;    /* global degisken   */
int b   = 20;    /* global degisken   */

int main(void)
{
  fonk( 9.0 );
  printf("main,  a : %d, b : %d\n", a, b );
  return 0;
}

void fonk( double a )   /* lokal parametre*/
{
  double b = 3.0;        /* lokal degisken     */
  {
    int a  = 30;
    printf( "fonk, ic blok a : %d\n", a );
  }
  printf( "fonk,  a : %1.1f, b : %1.1f\n", a, b );
}
```

Çıktı

```
fonk,  ic blok   a :  30
fonk,  a : 9.0,  b :  3.0
main, a : 10,   b :  20
```

Programda tanımlanan fonk fonksiyonu bloğunda, aynı değişken isimleri kullanılarak **double** tipinde a parametresi ve yine **double** tipinde b değişkeni bildirilir.

Ayrıca fonk fonksiyonu içinde oluşturulan bir başka blokta, tamsayı a değişkeni bildirilir. Bu değişkenler fonk fonksiyonunun lokal değişkenleridir ve hiç biri blok dışında bulunan global tamsayı değişkenler ile ilgili değildir.

Fonksiyon parametrelerine, görünürlük alanları fonksiyonun en dış bloğunda bildirilen değişkenlerinki ile aynı olduğundan dolayı *aynı isimde değişken bildirimleri içeren iç bloklar hariç* fonksiyon içinde her yerden erişilebilir (sadece **goto** etiketi'nin görünürlük alanı fonksiyon bloğunun tamamıdır). Bu nedenle fonk içinde bulunan blokta bildirilen tamsayı a değişkeni, bu blok içinde **double** parametre a'nın değerine erişimi engeller.

Programda ilk olarak fonk fonksiyonu çağrılır ve argüman olarak 9.0 değeri aktarılır. Aktarılan bu değer, a parametresine atanır. **double** tipindeki b değişkeninin bildirimini izleyen blok içinde tamsayı değişken a bildirilir. Dolayısıyla bu blok içinde **double** tipindeki a parametresinin değerine erişmek mümkün değildir. Sonuç olarak, blok içinde yer alan printf fonksiyonu çıktıda da görüldüğü gibi ekrana tamsayı a değişkeninin değerini yazar (30). Blok çıkışında bulunan printf fonksiyonu ise ekrana iki değer yazar; a parametresinin çağrı ile argüman olarak aktarılan değeri (9.0) ve b değişkeninin bildirimde ilk değer olarak atanan **double** değeri (3.0).
a ve b global değişkenleri **main** bloğu içinde görünür (erişilebilir) oldukları için fonk çağrısı tamamlandıktan sonra izleyen satırdaki printf fonksiyonu ile değerleri ekrana yazılır.

Blok içinde bildirilen değişkenler ve fonksiyon parametreleri, aynı isimli fonksiyonları da gizler. Örnek olarak **main** bloğu içinde bildirilen bir fonk değişkeni, fonk fonksiyonuna erişimi engeller ve **main** içinde bu ismin tüm kullanımları fonk değişkenine erişim anlamına gelir. Aynı şekilde fonk parametresi ile tanımlanan bir fonksiyon içinde aynı isimde bir lokal değişken bildirildiğinde bu ismin kullanımı fonk değişkenine erişim anlamına gelir.

Global değişkenlerin görünürlük alanının, blok yapısı kullanılarak kısıtlanması okunabilirliği azalttığı için hatalara sebep olabilir. Fakat bazı durumlarda tercih edilen bir yoldur.

Örneklerde sadece basit değişkenler (tek bir veri elemanına sahip değişkenler) kullanıldı. Fakat görünürlük kuralları tüm diğer değişken tipleri (diziler, union ve yapı değişkenleri) için de geçerlidir.

Okunabilirliği arttırmak için prototipteki parantezler arasında verilen parametre tip listesinde, parametre isimlerinin de kullanılabileceği belirtildi. Bu isimlerin görünürlük alanı, prototipi sona erdiren parantez ile sonlanır.

Yukarıdaki örnekte yer alan fonk fonksiyonu prototipi, parametre ismi kullanılarak aşağıdaki şekilde yazılabilir:

```
void fonk( double a );
```

Burada yer alan a ismi, sadece parantezler arasında görünür (erişilebilir) olduğu için programın başka yerinde kullanılabileceği gibi prototipte de a yerine herhangi bir isim kullanılabilir:

```
void fonk( double DEGISKEN );
```

Görünürlük Alanı ve Global Değişken Tanımının Kaynak Dosyada Bulunduğu Yer

Yukarıda değişken bildiriminin fonksiyon bloğu (yada herhangi bir blok) içinde yada dışında bulunmasının görünürlük alanını etkilediği anlatıldı. Tüm blokların dışında bulunan *global bir değişkenin, kaynak dosya içinde tanımlandığı yer de, görünürlük alanını etkiler*. Çünkü derleyici, kaynak dosyayı başından başlayarak satır satır okur. Aşağıdaki programda, tamsayı i değişkeni global seviyede tanımlanır. Fakat tanım i değişkenine ilk erişimin (fonk fonksiyonu çağrısında argüman olarak) yapıldığı **main** fonksiyonu tanımından sonra yerleştirilir. Derleyici kaynak dosyayı satır satır okuyarak i değişkeninin kullanıldığı noktaya ulaşır. i değişkeni bu noktada henüz *bildirilmemiş* olduğu için derleyici tarafından tanımlanmadığı kabul edilir; ve i değişkeni bildirildiği yerden önce kullanıldığından dolayı derleyici hata verir. Sonuç olarak *kaynak dosyada bir global değişken'e erişim ancak bildirildiği yerden sonra tanımlanan fonksiyonlar içinden mümkündür.*

Derleyici kaynak dosyayı satır satır okuduğu için, bu kural lokal değişkenler için de geçerlidir. *Bir lokal değişken, blok içinde sadece bildirildiği nokta ile blok kapanış parantezi arasında erişilebilir.*

```
#include <stdio.h>
void fonk( int );
int main(void)
{
  fonk( i );
  return 0;
}
int i = 10;  /* global degisken */
void fonk( int j )
{
  printf("i : %d\n", j );
}
```

Görünürlük Alanının ve Varolma Süresinin Kontrolu

Değişkenlerin bellekte nasıl saklandıkları, görünürlük alanına ve varolma süresine etki eder. Değişkenler bellekte iki şekilde saklanırlar: *otomatik* ve *static* olarak. Otomatik olarak saklanan değişkenler herhangi bir blok içinde bulunurlar ve bu bloğa girildiğinde oluşturulur, bloktan çıkıldığında ise otomatik olarak yok olurlar. Static olarak saklanan değişkenler ise herhangi bir blok içinde yada dışında bulunabilirler. Bu değişkenler değerlerini programın çalışması boyunca ve içinde bulundukları blok yada fonksiyona giriş ve çıkışlarda muhafaza ederler. C dilinde, değişken bildiriminin önüne yerleştirilerek, derleyiciye değişkenin bellekte nasıl saklanacağını ve görünürlük alanını bildiren çeşitli *saklama sınıfı belirleyici*ler (*storage class specifiers*) bulunur. Bu belirleyiciler şunlardır: **auto**, **static**, **extern**, **register** ve **typedef**.. Her bildirimde en fazla bir belirleyici bulunabilir.

extern bildirim :

Bir global değişken, aynı kaynak dosya içinde tanımlandığı yerden önce kullanılacak ise, mutlaka kullanıldığı yerden önce bir **extern** bildirim yapılmalıdır. **extern** bildirim, normal bildirimin önüne **extern** anahtar kelimesi yerleştirilerek yapılır. Bu yol ile yukarıdaki örnekteki erişim problemi çözülebilir:

Örnek 4.2-4 extern.c programı.

```
#include <stdio.h>
void fonk( int );
int main(void)
{
  extern int i;     /* extern bildirim      */
  fonk( i );
  return 0;
}
int i = 10;       /* global degisken   */
void fonk( int j )
{
  printf("i : %d\n", j );
}
```

Çıktı

```
i : 10
```

Örnek 4.2-4'de yapılan **extern** bildirim, **main** bloğu içinde bulunduğu için sadece bu fonksiyon içinde geçerlidir. i değişkeni program içinde yine tanımlandığı yerden önce ve başka bir fonksiyonda kullanılacak ise bu fonksiyon bloğu içinde **extern** bildirim gereklidir. Fakat tanımlandığı yerden önce kullanıldığı tüm fonksiyonlar içine **extern**

bildirim yerleştirmek yerine tüm blokların dışında ve öncesinde bir **extern** bildirim yapmak en pratik yoldur.

Yukarıda bir global değişkenin, aynı kaynak dosyada tanımlandığı yerden önce kullanılabilmesi için **extern** bildirimin gerektiği belirtildi. Kaynak program birden fazla dosyadan oluşabilir ve global değişkene tanımlandığı dosyadan farklı bir dosyada erişim gerekebilir (yani bir dosyada bulunan fonksiyonun diğer dosyadaki herhangi bir değişkene erişimi gerekebilir). Bu durumda da yine global değişkene erişilen diğer dosyada **extern** bildirim gerekir.
Bir global değişkenin *bildirimi ve tanımı* aynı şey değildir. Bildirim ile değişkenin özellikleri (tipi, boyutları gibi) ifade edilir ve bu değişkenin başka bir yerde tanımlandığını belirtilir. Tanım ise aynı zamanda bir bildirimdir; değişken için bellek alanı ayrılmasını ve bu alana ilk değer atama yapılabilmesini sağlar.

Kaynak programı oluşturan bütün dosyalar arasında global değişkenin yalnızca bir tanımı olmalıdır; programı oluşturan diğer dosyalarda bu değişkene erişmek için **extern** bildirimler bulunabilir. Yukarıdaki örnek programda görüldüğü gibi tanımın bulunduğu dosyada da **extern** bildirimler bulunabilir.

Kaynak dosyada **extern** anahtar kelimesi bulunmayan ve ilk değer atama yapılmayan değişken bildirimleri, *geçici tanım* (*tentative definition*) olarak adlandırılır. Yine **extern** kullanılmadan ilk değer atama yapılan bir bildirim ise *gerçek tanım*dır. Eğer kaynak dosyada bir değişkene ilk değer atama yapılan bir gerçek tanım bulunmuyor ise tüm geçici tanımlar ilk değer olarak 0 (adres değişkenleri için ilk değer olarak boş adres değeri -null pointer-) ile o değişkenin tanımı olarak kabul edilir. Eğer gerçek tanım bulunuyor ise geçici tanım, bildirim olarak ele alınır:

```
/* gecici tanım */
int i;
 ...
int main(void)
{
      ...
      printf( "i : %d\n", i );
}
/* gercek tanım */
int i = 3;
```

Örnek 4.2-5'de yer alan program, iki ayrı kaynak dosyadan oluşur. Bu örnekte, birden fazla kaynak dosyadan oluşan bir programdaki erişim kuralları gösterilir. Basit programlar genellikle tek bir kaynak dosyaya yerleştirilir. Fakat karmaşık ve kapsamlı projeler birden fazla kaynak dosyaya bölünür. İlk dosyada,

```
int i = 10, j = 20;
```

satırında i ve j global değişkenleri *tanımlanır* ve ilk değer atamaları yapılır. Bu tanım ile bellekte her iki değişken için bellek alanı ayrılır. Ayrıca bu tanım, birinci dosyada bulunduğu satırdan itibaren *her iki değişkenin de bildirimi* olarak iş görür.

İkinci dosyada ise fonk fonksiyonu bloğunda, i ve j değişkenlerinin

```
extern int i, j;
```

satırı ile **extern** bildirimleri yapılır. Bu satır ile değişkenler için bellek alanı ayrılmaz, fakat ikinci dosyada bulunduğu noktadan blok sonuna kadar (çünkü blok içinde bulunur) i ve j'nin tamsayı değişkenler olduğunu ve başka bir yerde tanımlandığı ifade edilir.
Tanım yeri, bulunulan görünürlük alanı dışı yada bir başka dosya olabilir. Böylece bu değişkenlerin bir başka dosyada tanımlandığı belirtilir ve dolayısıyla fonk fonksiyonu diğer dosyadaki i ve j değişkenlerine erişebilir.

Örnek 4.2-5 Birden fazla kaynak dosyadan oluşan (iki-dosyalı) program.

```
/*
 * dosyaa.c
 */
#include <stdio.h>

/* global degisken tanimlari */
int i = 10, j = 20;

extern void fonk( void );

int main(void)
{
  fonk();
  return 0;
}
```

```
/*
 * dosyab.c
 */
#include <stdio.h>

  void fonk(void)
{
    /* extern bildirimler */
    extern int i, j;

    printf( "i : %d, j : %d\n", i, j );
  }
```

Çıktı

```
i : 10, j : 20
```

Programda ayrıca ikinci dosyada tanımlanan fonk fonksiyonu, birinci dosya içinde bulunan **main** fonksiyonu tarafından çağrılır. Erişimin gerçekleşmesi için, ilk dosyadaki **main** bloğundan önce fonk fonksiyonunun **extern** bildirimi yapılır. Ancak C dilinde değişkenlerin aksine fonksiyonlar daima kendiliğinden global olduğu için birinci dosyadaki fonksiyon bildiriminde, **extern** anahtar kelimesinin kullanımı gereksizdir. **extern** anahtar kelimesinin kullanımı fonksiyonun başka bir dosyada tanımlandığını açıkça belirtir ve böylece okunabilirliği arttırır. fonk fonksiyonu başka bir kaynak dosyada tanımlanmış olsa dahi **main** tarafından çağrılabilir. *Fonksiyonlar normal olarak, birden fazla kaynak dosyadan oluşan programlarda her yerden erişilebilir.*
İkinci dosyadaki **extern** bildirim, fonk bloğu içinde yer alır. Bu durumda bildirim sadece bu blok sonuna kadar geçerli olur. Tüm blokların dışında ve kaynak dosyanın başında yer

alan **extern** bildirim ise, aynı kaynak dosyada bulunduğu satırdan sonra tanımlanan tüm fonksiyonlar içinde geçerlidir.

static değişkenler :

Bazı durumlarda birden fazla kaynak dosyadan oluşan programlarda global bir değişkenin sadece tanımlandığı kaynak dosyada görünür (erişilebilir) olması istenebilir. Bu iş bildirimde tip öncesine **static** anahtar kelimesi yerleştirilerek gerçekleştirilir. Bu şekilde bildirilen bir değişken *static external (global) değişken* olarak adlandırılır ve sadece bulunduğu kaynak dosyada, tanımlandığı satır sonrasında erişilebilir.

static anahtar kelimesi, erişimi kısıtlamak için fonksiyon bildirimlerine de uygulanabilir. Bu şekilde bildirilen bir fonksiyon *static fonksiyon* olarak adlandırılır ve sadece tanımlandığı dosya içinde çağrılabilir. Örnek 4.2-5'de ikinci dosyada tanımlanan fonk fonksiyonunun başlık kısmında tip öncesine **static** eklenirse, ilk dosyadan erişim mümkün olmayacaktır. Linker hata verir ve program derlenemez:

```
static void fonk(void)
{
      ...
}
```

Fonksiyonun tanımlandığı dosyada prototipi bulunmuyor ise **static** anahtar kelimesi fonksiyon tanımına yerleştirilmelidir (başlıkta, tip belirleyici öncesine). **static** anahtar kelimesi bir fonksiyonun hem prototipinde hem de tanımında yer alabilir. Fakat **static** bildirimi yapılan bir fonksiyonun tanımında da **static** belirleyici kullanılması gerekmez.

static belirleyici, fonksiyon ismini programın diğer dosyalarından gizler. Böylece aynı isim diğer dosyalarda başka bir fonksiyon için kullanılabilir. Örneğin ticari fonksiyon kütüphaneleri oluşturulurken kütüphanedeki bütün fonksiyonlar **static** olarak tanımlanır. Bu yolla kullanıcının kendi oluşturduğu fonksiyonlarla isim benzerliğinden doğabilecek aksaklıklar önlenmiş olur.

Fonksiyon ismine, tanımda sadece **static** belirleyici; prototipte ise **extern** ve **static** belirleyiciler uygulanabilir.

otomatik değişkenler :

Bir değişkenin kullanımına etki eden diğer önemli özelliği, programın çalışması sırasında var olduğu ve değerini koruduğu süredir. Herhangi bir blok içinde (fonksiyon bloğu yada program bloğu) bildirilen lokal değişken, bloğa girildiği yada fonksiyon çağrıldığı zaman oluşturulur ve blok sonuna ulaşıldığında yada fonksiyondan çıkıldığında yok olur. Dolayısıyla, "lokal" kelimesi bu değişkenin görünürlük alanını ifade eder.

auto belirleyici kullanılarak bildirilen bir lokal değişken, fonksiyonla yada girilen blok ile birlikte *otomatik* olarak gelir ve gider (otomatik varolma süresi). Bu nedenle *otomatik değişken* olarak da adlandırılır. Fakat bildirimlerde **auto** kullanılması zorunlu değildir. Çünkü fonksiyonlar içinde **extern** *ve* **static** *belirleyici olmadan bildirilen tüm değişkenler, kendiliğinden otomatik* olacağı için bildirimde **auto** anahtar kelimesinin kullanımı sadece program kodunun okunabilirliğini arttırır. Fonksiyon parametreleri de lokal değişkenlerdir ve dolayısıyla otomatik olarak saklanırlar.

Otomatik değişkenlerin bazı avantajları vardır:

- bellek tasarrufu sağlarlar (fonksiyon çalışmasını bitirdiğinde kayboldukları için, ihtiyaç olmadığında bellek harcamazlar).
- otomatik değişkenler, programın işletim zamanı sırasında oluşturulduğu için çalıştırılabilir program kodunun boyutlarını yada bellekte kaplayacağı alanı arttırmazlar.
- fonksiyon çağrıları arasında değerleri muhafaza edilmediği için, önceki çağrılardan kalma artık değerlerin bir sonraki çağrıda kullanılması gibi bir sorun yoktur.

Örnek programlarda pek çok otomatik değişken bildirildi. Aşağıdaki fonksiyonda tamsayı i ve j değişkenlerinin her ikisi de otomatik değişkenlerdir:

```
void fonk( void )
{
   int i;              /* otomatik degisken i */
   auto int j;         /* otomatik degisken j */
}
```

Yukarıda da belirtildiği gibi otomatik değişkenlerin varolma süresi blok ile sınırlıdır. Ve sadece blok içinde bildirildikleri satırdan blok sonuna kadar geçerlidirler (görünürlük alanı).

Global değişkenler, programın çalışması boyunca fonksiyon çağrıları arasında değerlerini muhafaza ederler (global varolma süresi) ve tanımlandıkları satırdan program sonuna kadar ve diğer dosyalardan yapılan erişimlere izin verirler. Geniş görünürlük alanı ve uzun varolma süresine sahiptirler (static bellek alanında olarak saklanırlar). Bu nedenle birbirlerini çağırmayan ve bazı verileri paylaşmak durumunda olan fonksiyonlar arasında veri taşınmasında kullanılabilirler. Dolayısıyla argümanlar ile veri aktarımına ve **return** ile değer döndürülmesine alternatif olarak kullanılabilir. Çok sayıda veri paylaşılıyor ise, uzun argüman listeleri kullanımına gerek kalmaz.

Bu nedenlerden dolayı, lokal fonksiyon parametreleri yerine global değişkenler kullanılır. Ayrıca bazı durumlarda referans-ile-çağırma yerine, global veri kullanımı tercih edilebilir (örneğin, global dizi kullanımı).

Önceki bölümde de anlatıldığı gibi global değişkenlere **static** anahtar kelimesi uygulanarak görünürlük alanları bulundukları kaynak dosya ile sınırlanabilir (**external static** değişkenler). **static** belirleyici blok içinde bildirilen lokal değişkenlere de uygulanabilir. Blok içinde **static** olarak bildirilen değişkenler (**internal static** değişkenler), otomatik değişkenlerde olduğu gibi yine bildirildikleri blok içinde görünürler. Fakat otomatik değişkenlerin aksine bulundukları bloğa her giriş ve çıkışta yada fonksiyonun her çağrılışında gidip gelmek yerine daima var olur. Dolayısıyla **static** anahtar kelimesi bir lokal değişkene uygulandığında, bu değişkenin görünürlük alanı yerine varolma süresini etkiler. Sonuç olarak, herhangi bir blok içinde **static** olarak bildirilen bir değişken, bu blok ile sınırlı görünürlük alanına sahiptir ve programın çalışması boyunca sürekli olarak bellekte tutulur.

Aşağıdaki programda, **static** belirleyici kullanılarak fonksiyon çağrıları arasında değerini koruyan lokal değişken Deger bildirilir:

Örnek 4.2-6 static.c programı.

```
#include <stdio.h>

#define   ILK_DEGER       10
#define   IKINCI_DEGER    20

void fonk( int );

int main(void)
{
     printf("ilk cagri ..." );
     fonk( ILK_DEGER );

     printf("ikinci cagri ...");
     fonk( IKINCI_DEGER );

     return 0;
}

void fonk( int i )
{
  static int Deger;

  if ( i == ILK_DEGER )
       Deger = 0;

  Deger = Deger + i;

  printf(" Deger : %d\n", Deger );
}
```

Çıktı

```
ilk cagri ...      Deger : 10
ikinci cagri  ... Deger : 30
```

Programda, Deger değişkeni çağrılar arasında değerini koruduğu için ikinci çağrıda ekrana 30 değeri yazılır. Çünkü ikinci çağrıda aktarılan 20 değeri önceki çağrıdan kalan

10 değeri ile toplanır. Görüldüğü gibi **static** bildirim, Deger değişkeninin varolma süresini arttırır. Fakat Deger değişkenine erişim, sadece fonk fonksiyonu içinde mümkündür; görünüm alanı değişmez.

Program, fonksiyon içinde bildirilen lokal değişkenler (otomatik değişkenler) için *stack* alanını kullanır Fonksiyon çalışmasını bitirinceye kadar bu değişkenler stack alanında kalır. Dolayısıyla fonksiyonun lokal değişkenleri için çok fazla stack alanı gerekiyor ise yada stack alanında pek çok veri alanı oluşmasına sebep olan çok sayıda iç içe çağrı yapılmış ise (kendi kendini pek çok kez çağıran fonksiyonlarda olduğu gibi), program stack alanı dışına taşabilir (*stack overflow*). Bu durum işletim zamanı hatasına sebep olur. Örneğin çok sayıda elemana sahip bir otomatik dizi, bu taşmaya sebep olabilir. Aşağıda bildirilen s dizisi, 1 byte büyüklüğünde 5000 elemana sahiptir ve işletim zamanı sırasında stack taşması hatasına sebep olabilir:

```
void fonk(void)
{
  ...
  char s [ 5000 ];
  ...
}
```

Bildirimde **static** anahtar kelimesi kullanılarak derleyicinin lokal değişkenler için stack alanını kullanmaması sağlanabilir. s dizisi **static** olarak bildirildiğinde, bu dizinin elemanları stack yerine bellekteki *static veri alanı*na yerleştirilir. Böylece stack'ta diğer lokal değişkenler için kullanılabilecek alan miktarı artmış olur.

```
void fonk(void)
{
  ...
  static char s [ 5000 ];
  ...
}
```

Bazı derleyiciler, program için daha geniş stack alanı ayrılmasını sağlayan seçeneklere sahiptir. **static** veriler kullanılırken daha önce belirtilen noktalar dikkate alınmalıdır. Veriler static (kalıcı) olarak saklandığında, çağrılar arasında değerlerini korurlar. Dolayısıyla fonksiyonun her çağrılışında aynı değerlere erişilir.

Bir değişken bildiriminde, varolma süresi ve görünürlük alanını ifade eden en fazla bir belirleyici bulunabilir. Herhangi biri bulunmadığı durumda fonksiyon içinde *bildirilen* değişkenler (bütün değişken tipleri dahil) için **auto**; yine herhangi bir fonksiyon içinde *bildirilen* fonksiyonlar için ise, **extern** belirleyici kabul edilir. Tüm fonksiyonların dışında *bildirilen* değişkenler için **extern** kabul edilir. Tüm fonksiyonların dışında *bildirilen* fonksiyonlar ise global varoma sürelidir ve programın tüm dosyalarından görünürler.

Aşağıdaki örnekte, f1 ve f2 fonksiyonlarının prototipleri **main** fonksiyonu içine yerleştirilmiştir. Dolayısıyla bildirimler **main** bloğu içinde geçerli olur. Bu durumda f1 fonksiyonu f2 fonksiyonunu çağırdığında hata oluşur. Çünkü f2 fonksiyonu tanımı f1 bloğu sonrasında yer alır. Fakat bildirimler **main** dışına çıkarıldığında bu sorun önlenir.

```
#include <stdio.h>

int main(void)
{
  void fonk1( void );
  void fonk2( void );

  fonk1();
  fonk2();

  return 0;
}

void fonk1(void)
{
  fonk2();            /* bu cagri hata olusturur  */
  puts( "fonk1 ..." );
}

void fonk2(void)
{
  puts("fonk2 ...");
}
```

register değişkenler :

Yukarıdaki satırlarda **auto**, **static** ve **extern** belirleyiciler anlatıldı. **register** ise bir diğer belirleyicidir. Bir otomatik değişkenin yada fonksiyon parametresinin bildiriminde, tip öncesine yerleştirilen **register** anahtar kelimesi, değişkenin mikro-işlemcinin register birimlerinde saklanacağını ifade eder. Fakat bu kullanımda, bazı kısıtlamalar sözkonusudur. Donanıma bağlı olarak, **register** belirleyici sadece **char**, **short** ve **int** tamsayı tipler (**signed** yada **unsigned**) ve adres değişkeni tipleri ile kullanılabilir. Ayrıca sadece tek elemanlı veriler mikro-işlemcinin register birimlerine yerleştirilebilir. Diziler, yapı değişkenleri gibi birden fazla elemandan oluşan veri grupları, register birimlerine yerleştirilemeyecek kadar büyüktür. Donanıma bağlı olarak, sadece belli sayıda register birimi kullanılabilir ve bu sayıdan fazla register kullanımı söz konusu olduğunda, kalan değişkenler derleyici tarafından **register** bildirim dikkate alınmadan adreslenebilir bellek alanlarına yerleştirilir. Register birimleri adreslenebilir olmadığı için & operatörü kullanılarak, **register** ile bildirilmiş bir değişkenin adresi alınamaz. Bu kural, derleyicinin değişkenleri register birimlerine yerleştirip yerleştirmediğine bakılmaksızın geçerlidir. Dolayısıyla **scanf** fonksiyonu kullanılarak, bir **register** değişkene değer alınamaz.

Aşağıdaki satırda, i ve j **register** değişkenleri bildirilir:

```
register int i, j;
```

Yukarıdaki bildirimde **register** anahtar kelimesi kullanılarak, **int** tipi i ve j değişkenlerinin mikro-işlemcinin register birimlerinde saklanmasının istendiği ifade edilir. **register** bildirimde, tip belirleyici kullanılmadığında, veri tipi olarak **int** kabul edilir.

typedef :

C dilinde **typedef** anahtar kelimesi kullanılarak mevcut veri tipleri için yeni isimler tanımlanabilir. Program kodunun okunabilirliğini arttıran, yaygın bir uygulamadır. Aşağıdaki bildirim ile **int** tip belirleyicisi yerine kullanılabilecek TAMSAYI ismi tanımlanır:

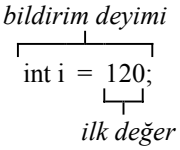
```
typedef  int  TAMSAYI;
```

Görüldüğü gibi **typedef** bildirimi, normal bildirim öncesine **typedef** anahtar kelimesi yerleştirilerek gerçekleştirilir. Böylece değişken bildirimi, tip ismi bildirimi olur ve bildirimde yer alan isim, veri tipi ismi haline gelir. **typedef** bildirimi, yeni bir veri tipi oluşturmaz. Sadece mevcut bir veri tipi için yeni bir isim tanımlanmasını sağlar.
Pratik ve yaygın kullanımına örnek olarak, birden fazla veri elemanından oluşan tipler için (diziler ve yapı değişkenleri gibi) kısa ve tanımlayıcı isimler oluşturulması verilebilir (Örnek 7.2-1, Örnek 8.3-3, Örnek 8.3-4, Örnek 8.3-5, Örnek 9.4-3). **typedef** ile tanımlanan yeni isimler, büyük harflerle yazılarak okunabilirlik arttırılabilir.

typedef bildirimi, taşınabilirlik problemlerinin daha kolay çözülebilmesini sağlar. Donanıma bağlı veri tipleri için **typedef** bildirimleri kullanılır ise program farklı bir çevreye taşındığında, sadece **typedef** bildirimlerini değiştirmek yeterli olacaktır. **typedef** kullanımı, önişlemci komutu #define kullanımına benzer. Fakat derleyici tarafından işlem gördüğü için **typedef** ile daha kapsamlı değiştirmeler yapılabilir. **typedef**, diğer belirleyicilerde olduğu gibi verilerin bellekte nasıl saklanacağına etki etmemekle birlikte, bildirimde bulunduğu yer dolayısıyla aynı şekilde yorumlanır. ■

4.3 İlk Deger Atama

İlk değer atama, herhangi bir değişkene (bu aşağıda görüldüğü gibi bir basit değişken, dizi değişkeni, yada daha sonraki bölümlerde ayrıntılı olarak anlatılan **union** yada yapı değişkeni olabilir) başlangıç değerinin bildirim sırasında atanmasıdır. Bu işlem, normal atama ile aynı değildir ve bazı kuralları vardır.



Bir değişkenin bildirimi sırasında bellek alanı ayrılıyor ise bu bildirim ile değişken tanımlanmış olur. Otomatik değişkenlerin bildirimleri, aynı zamanda bellek alanı ayrılmasını sağladığı için bu değişkenlerin tanımıdır. Bir otomatik basit değişkene yada bir **register** değişkene, tanımlandığı sırada ilk değer atama işlemi yapılabilir. Bunun için değişken bildirim deyiminde bir *sabit değere, önceden tanımlanmış değişkene, fonksiyon çağrısına yada bunlarla oluşturulan herhangi bir geçerli ifadeye* eşitlenebilir.

Daha önce de anlatıldığı gibi **auto** ve **register** belirleyiciler ile bildirilen değişkenler, otomatik varolma sürelidir. Programın çalışması sırasında bu değişkenler bulundukları fonksiyon bloğuna yada program bloğuna her normal girişte (**goto** ile sapmadan gerçekleşen) tekrar oluşturulur ve eğer var ise ilk değer atama işlemleri (bu iş için gerekli komutların çalıştırılması ile) yapılır. Dolayısıyla otomatik değişkenlerin önceki fonksiyon çağrılarında yada program bloğu girişlerinde aldıkları değerleri korumaları sözkonusu değildir. İlk değer atama yapılmadığında **auto** ve **register** değişkenlerin başlangıç değerleri belirsizdir. Bir otomatik değişkene ilk değer atama ile daha sonra yapılacak olan atama işlemi bildirim sırasında yapılarak işlemler kısaltılmış olur. Fakat, ilk değer atama işlemi değişkenin kullanım noktasından uzakta yapıldığı için ilk değerlerin görülmesi zor olabilir. Bu nedenle okunabilirliği arttırmak amacıyla açık olarak atama yapılması (herhangi bir atama deyiminde) tercih edilir.

Fonksiyon parametreleri de otomatik olarak saklanırlar, fakat bu değişkenlere ilk değer atama yapılamaz. Ayrıca global olarak bildirilemezler. Ancak parametrenin kendisi aşağıda görüldüğü gibi ilk değer atama işleminde kullanılabilir:

```
int fonk( int m )       /* parametre m */
{
  ...
  int i = m + 1;        /* ilk deger atama */
  ...
}
```

Yukarıdaki işlem, bir atama deyimi ile de gerçekleştirilebilir:

```
int fonk( int m )  /* parametre m */
{
  ...
  int i;
  ...
  i = m + 1;              /* atama deyimi */
  ...
}
```

Global ve **static** değişkenlere de ilk değer atama yapılabilir. Fakat atanan ilk değer otomatik değişkenlerin aksine, bir sabit yada önceden bildirilmiş global yada **static** verinin bellek adresine bir tamsayı sabitin eklendiği yada çıkarıldığı ifade olabilir. İlk değer atama yapılmadığında ise başlangıç değerleri 0 olur.

Örnek 4.3-1 ilkdeg1.c programı.

```
#include <stdio.h>

/* global degiskenler */
int   a = 0;
int   b;

int main(void)
{
   /* auto degiskenler */
   char c =  'c';
   int  i = 3;
   int  j;
   long  l, n, m = 5L;
   long  k  = m * 10L * 2L;

   /* internal static degiskenler */
   static s1 = 9;
   static s2;

   /* register degiskenler */
   register  r1 = i * 2;
   register  r2;

   /* atama deyimi */
   l = n = 3L;

   printf( "global degiskenler : \n a : %d, b : %d\n", a, b );
   printf( "otomatik degiskenler : \n");
   printf( "c : %c, i : %d, j : %d, l : %ld, n : %ld, m : %ld, k : %ld \n",  c, i, j, l, n, m, k );
   printf( "static degiskenler : \n s1 : %d, s2 : %d \n", s1, s2 );
   printf( "register degiskenler : \n r1 : %d, r2 : %d\n", r1, r2 );

   return 0;
}
```

... Örnek 4.3-1 devam

Çıktı

```
global degiskenler :
a : 0, b : 0
otomatik degiskenler :
c : c, i : 3, j : 235, l : 3, n : 3, m : 5, k : 100
static degiskenler :

s1 : 9, s2 : 0
register degiskenler :
r1 : 6, r2 : 916
```

Programda aşağıda açıklanan bildirim ve ilk değer atama işlemleri yer alır:

- otomatik değişkenler :

Programda c, i, j, l, n, m ve k otomatik değişkenleri bildirilir. c, i, m ve k değişkenlerine, bildirim sırasında ilk değer atama yapılır. İlk değer atama işleminde, her bir değer sadece tek bir değişkene atanır.

Dolayısıyla,

```
long l, n, m = 5L;
```

bildirim deyimindeki 5L değeri, sadece m değişkenine atanır. Okunabilirliği arttırmak amacıyla, her bir bildirim deyiminde sadece tek bir ilk değer atamanın yapılması uygun olur. l ve n değişkenlerine, aşağıdaki deyim ile 3L değeri atanır:

```
l = n = 3L;
```

Bu işlem, ilk değer atama ile bildirim sırasında aşağıdaki şekilde de yapılabilir:

```
long l = 3L,  n = 3L, m = 5L;
```

İlk değer atama yapılmayan j değişkeninin değeri, daha sonra da hiçbir değer atanmadığı için belirsizdir ve o anda bellekte bulunan anlamsız değerler ekrana yazılır.

- global değişkenler :

Global a değişkenine ilk değer olarak 0 atanır. Global b değişkenine ilk değer atama yapılmadığı halde başlangıç değeri 0 olur. b değişkenine de ilk değer atama yapılarak, global bildirimler aşağıdaki şekilde düzenlenebilir:

```
int a = 0, b = 0;
```

- **static** değişkenler :

Tamsayı s1 değişkenine ilk değer olarak 9 değeri atanır. Tamsayı s2 değişkenine ise ilk değer atama yapılmaz. Fakat çıktıda görüldüğü gibi ilk değer atama yapılmayan tamsayı **static** değişken s2'nin başlangıç değeri 0 olur.

- **register** değişkenler :

int tipi **register** değişken r1'e, bildirim deyiminde i * 2 ifadesi eşitlenerek ilk değer atama yapılır. **int** tipi **register** değişken r2'nin değeri ise ilk değer atama yapılmadığı için belirsizdir.

Static varolma süreli değişkenler (global ve **static** olarak bildirilen), program başlamadan önce oluşturulur ve ilk değer atama işlemleri sadece bir kez yapılır.

Dolayısıyla bu değişkenler çağrılar arasında değerlerini korurlar (aynı durum kendini çağıran fonksiyon bloğu içinde tanımlanan **static** değişkenler içinde geçerlidir). Örnek 4.3-2'de bildirilen j değişkeni, çıktıda görüldüğü gibi çağrılar arasında değerini korur.

Örnek 4.3-2 ilkdeg2.c programı.

```
#include <stdio.h>
int fonk( void );
int main(void)
{
   int i;
   for (  i = 0; i < 5; i++ )
         printf("%d\n", fonk() );
   return 0;
}
int fonk(void)
{
   static int j = 0; /* blok ici static bildirim */
   return j++;
}
```

Çıktı

```
0
1
2
3
4
```

Diziler ve yapı değişkenleri birden fazla veri elemanına sahiptir (izleyen bölümlerde anlatılacaktır). Bir diziye yada yapı değişkenine (otomatik yada static) ilk değer atama, bildirim sırasında virgül ile ayrılmış ve oklu parentezlerle sınırlanmış sabitlerden oluşan ilk değer listesi ile yapılabilir. Dizinin boyutları verilmediğinde derleyici ilk değerleri sayarak eleman sayısını hesaplar. Fakat bildirimde köşeli parantezler içinde eleman sayısı verildiğinde belirtilen sayıdan daha az ilk değer varsa, kalan elemanların ilk değerleri 0 olur. Daha çok sayıda ilk değer var ise derleyici hata mesajı verir. Bir otomatik değişkene ilk değer atama tek bir ifade ile yapılıyor ise bu ifade sabit ifade olmayabilir.

Bir otomatik diziye ilk değer atama yapılmadığında elemanların değerleri belirsiz olur. İlk değer atama yapılmayan global yada static bir dizinin elemanlarının değerleri ise 0 olur. Aynı kurallar bir **union** için de geçerlidir, fakat bir **union** değişkenine ilk değer atama sadece ilk elemanının tipinde bir sabit değer ile yapılabilir.

Aşağıda, tek boyutlu otomatik GUN dizisi bildirilir ve parantezlerle sınırlı 12 sabit değer ile ilk değer atama yapılır. Atanan değerler, aylardaki gün sayılarıdır:

```
int GUN[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

GUN dizisinde olduğu gibi dizinin boyutları verilmediğinde derleyici ilk değerleri sayarak eleman sayısını hesaplar. Bu durumda, GUN dizisi 12 elemanlı bir dizi olacaktır. Fakat bildirimde köşeli parantezler içinde eleman sayısı verildiğinde belirtilen sayıdan daha az ilk değer varsa kalan elemanların ilk değerleri belirsiz olur. Daha çok sayıda ilk değer var ise derleyici hata mesajı verir.

Bir yapıyı yada diziyi oluşturan veri grubunun herhangi bir elemanı, bir başka veri grubu oluşturabilir (alt-veri grubu). Herhangi bir yapı yada dizi değişkenine uygulanan ilk değer atama kuralları, bu alt-veri grupları için de geçerlidir. Yani, alt-veri grubu elemanlarına ilk değer atama, yine oklu parantezlerle sınırlanmış ve virgüllerle ayrılmış sabit değerler ile yapılır. Eleman sayısından fazla ilk değer var ise bu hata oluşturur. Alt-veri grubunun ilk değerleri parantezlerle sınırlanmamış ise gerekli sayıda ilk değer kullanılır ve kalan değerlerle ana grubun izleyen elemanlarına ilk değer atama yapılır.
Örnek 4.3-3'de iki boyutlu tamsayı b dizisi bildirilir ve tümüyle parantezlerle sınırlı sabit değerler ile ilk değer atama yapılır:

```
int b[4][3] = {
                { 1, 2, 3 },   /*  b[0][0..2] */
                { 4, 5, 6 },   /*  b[1][0..2] */
                { 7, 8, 9 },   /*  b[2][0..2] */
              };
```

b dizisi, 4 elemandan oluşur ve her eleman 3 elemanlı bir dizidir. b dizisinin ilk elemanı olan b[0] dizisine 1, 2, ve 3 iki değerleri atanır; yani alt-veri grubu olan b[0][0], b[0][1]

ve b[0][2] elemanlarına). İzleyen elemanlar ise b[1] ve b[2] dizisine ilk değer olarak atanır. Fakat b[3] dizisi için değer kalmadığından dolayı elemanlara 0 değeri atanır. Bu bildirim ve ilk değer atama aşağıdaki şekilde de yapılabilir:

```
int c[4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Fakat bu durumda ilk değer atama farklı yorumlanır. Çünkü sol parantez ile başlayan ilk değer listesi, c[0] için değil, c dizisi içindir. İlk üç değer kullanılır, izleyen değerler üçer üçer c[1] ve c[2] için kullanılır. Aşağıdaki bildirim ise d dizisinin ilk kolonuna ilk değer atama yapar (d[0][0], d[1][0], d[2][0] ve d[3][0] elemanlarına) :

```
int d[4][3] = {
                { 1 }, { 2 }, { 3 }, { 4 }
              };
```

Örnek 4.3-3 ilkdiz1.c programı.

```
#include <stdio.h>

int main(void)
{
   int i, j;
   int a[ ]            = { 1, 2, 3 };
   int b[ 4 ][ 3 ] = {   { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, };
   int c[ 4 ][ 3 ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
   int d[ 4 ][ 3 ] = {  { 1 }, { 2 }, { 3 }, { 4 }  };

   for ( i = 0; i < sizeof (a) / sizeof (a[0]); i++ )
      printf("a[%d] : %d\n", i, a[ i ] );

   for ( i = 0; i < 4; i++ )
        for ( j = 0; j < 3; j++ )
             printf("b[%d][%d] : %d\n", i, j, b[ i ][ j ] );

   for ( i = 0; i < 4; i++ )
        for ( j = 0; j < 3; j++ )
             printf("c[%d][%d] : %d\n", i, j, c[ i ][ j ] );

   for ( i = 0; i < 4; i++ )
        for ( j = 0; j < 3; j++ )
             printf("d[%d][%d] : %d\n", i, j, d[ i ][ j ] );

   return 0;
}
```

...Örnek 4.3-3 devam

Çıktı

```
a[0] : 1        b[0][0] : 1     c[0][0] : 1     d[0][0] : 1
a[1] : 2        b[0][1] : 2     c[0][1] : 2     d[0][1] : 0
a[2] : 3        b[0][2] : 3     c[0][2] : 3     d[0][2] : 0
                b[1][0] : 4     c[1][0] : 4     d[1][0] : 2
                b[1][1] : 5     c[1][1] : 5     d[1][1] : 0
                b[1][2] : 6     c[1][2] : 6     d[1][2] : 0
                b[2][0] : 7     c[2][0] : 7     d[2][0] : 3
                b[2][1] : 8     c[2][1] : 8     d[2][1] : 0
                b[2][2] : 9     c[2][2] : 9     d[2][2] : 0
                b[3][0] : 0     c[3][0] : 0     d[3][0] : 4
                b[3][1] : 0     c[3][1] : 0     d[3][1] : 0
                b[3][2] : 0     c[3][2] : 0     d[3][2] : 0
```

Örnek 4.3-4'de global dizi i, otomatik dizi j ve **static** dizi s bildirilir ve ilk değer atamaları yapılır. İlk değer atamada, herhangi bir değerin tekrarlandığını belirtmek için bir yol yoktur. Ayrıca sadece belli elemanlara ilk değer atamakta mümkün değildir.

İlk değer atama yapılmadığında, global i ve **static** s dizisinin elemanlarının değerleri 0 olur. Fakat otomatik j dizisinin elemanlarının değerleri belirsiz olur.

j dizisinin elemanlarının değerleri bir döngü ve atama deyimi ile 0 yapılabilir:

```
    ...
int j[ 3 ];
    ...
for ( indeks = 0; indeks  < 3; indeks++ )
        j[ indeks ] = 0;
    ...
```

Örnek 4.3-4 ilkdiz2.c programı.

```
#include <stdio.h>

int  i[ ] = { 0, 1, 2, 3 };

int main(void)
{
   int indeks;
   int j[ ] = { 4, 5, 6 };
   static int s[ ] = { 7, 8, 9 };

   for ( indeks = 0; indeks < sizeof (i) / sizeof (i[0]); indeks++ )
         printf(" i[%d] : %d\n", indeks, i[ indeks ] );

   for ( indeks = 0; indeks < sizeof (j) / sizeof (j[0]); indeks++ )
         printf(" j[%d] : %d\n", indeks, j[ indeks ] );
```

...Örnek 4.3-4 devam

```
   for ( indeks = 0; indeks < sizeof (s) / sizeof (s[0]); indeks++ )
        printf(" s[%d] : %d\n", indeks, s[ indeks ] );
   return 0;
}
```

Çıktı

```
i[0] :0
i[1] :1
i[2] :2
i[3] :3

j[0] :4
j[1] :5
j[2] :6

s[0] :7
s[1] :8
s[2] :9
```

Eğer bir global değişkenin bildiriminde ilk değer atama yapılıyor ise bu bildirim ile tanımlanmış olur. Bir global değişkenin sadece tek bir tanımı bulunabilir. Bir **extern** bildirim, bildirilen değişkenin özelliklerinin başka bir yerde tanımlanarak bu değişken için bellek alanı ayrıldığını ifade eder. Global dizilere ilk değer atama, sadece tanımda olur.

Tanımda dizi boyutları belirtilmelidir (ilk değer atama var ise ve boyutlar verilmezse derleyici tarafından değerler sayılarak hesaplanır); fakat dizinin **extern** bildiriminde boyutların verilmesi seçime bağlıdır. Aşağıda örnek olarak, SAYI dizisi verilir:

1. dosya :

```
        ...
#define ELEMAN   5
        ...
/* global dizi tanimi */
int SAYI [ ELEMAN ];
        ...
```

2. dosya :

```
        ...
/* extern bildirim */
extern int SAYI [ ];
        ...
```

Global bir değişkenin sadece bir tanımı olur. İlk değer atama yapılan bir bildirim o değişkenin tanımıdır. Yani herhangi bir **extern** bildirimde ilk değer atama girişimi, ikinci bir tanım oluşturma girişimi olacaktır. Ayrıca **extern** bildirimde değişken için (basit değişken, dizi yada yapı değişkeni) bellek alanı ayrılmaz. Bu nedenlerden dolayı **extern** bildirimlerde ilk değer atama yapılamaz.

enum tipi bir değişkene listedeki sabit değerlerden biri ilk değer olarak atanabilir.

Karakter dizilerine, dizgi sabitleri (*string literal*) ilk değer olarak atanabilir. Aşağıda karakter dizisi s bildirilir ve ilk değer atama yapılır:

```
char s [ ] = "abc";
```

Derleyici, "abc" dizgisinde bulunan karakterleri sayarak dizgi uzunluğunu bulur ve ona göre bellek düzenlemesi yapar. Bu nedenle ilk değer atama yapılıyor ise köşeli parantezler arasında eleman sayısı belirtmek gerekmez. Aşağıdaki deyimlerde s1 ve s2 karakter dizileri tanımlanır:

```
char s1[ ] = "abc";
char s2[ ] = { 'a', 'b', 'c', '\0' };
```

s1 dizisine ilk değer olarak "abc" karakter dizgisi atanır. Dizgi sabitleri, bellekte karakter dizisi olarak saklanırlar. Dizgi sonunun belirlenmesi için derleyici tarafından otomatik olarak her dizgi sonuna boş-karakter (NUL veya karakter sabiti olarak '\0') eklenir. Bu nedenle derleyici tarafından saptanan dizgi uzunluğu, karakter sayısının bir fazlasıdır. sizeof s1 / sizeof s1[0] ifadesi ile bulunan s1 dizisinin eleman sayısı, "abc" dizgisinin karakter sayısından bir fazladır (yani 4).

Dolayısıyla ilk değer atama yapılırken köşeli parantezler arasında eleman sayısı veriliyor ise boş-karakter için ilave yer ayrılmalıdır. Ayrıca dizgideki karakter sayısı (boş-karakter hariç), parantezler arasında verilen eleman sayısını geçmemelidir. Örnek olarak, aşağıda bildirilen c dizisinin eleman sayısı kullanılacak dizgi boyunun bir fazlası olarak verilmiştir:

```
...
#define  DIZGI_BOYU  20
...
char  s[ DIZGI_BOYU+1 ];
```

s1 dizisi, eleman sayısı belirtilerek aşağıdaki gibi bildirilebilir:

```
char  s1[ 4 ]  =  "abc";
```

İlk değer atama, karakter sabitlerinden oluşan ilk değer listesi ile de yapılabilir. Fakat bu durumda, derleyici tarafından karakter dizisi sonuna boş-karakter eklenmeyeceği için bu iş açık olarak yapılmalıdır:

```
char  s2 [ ]  =  { 'a', 'b',  'c', '\0' };
```

yada

```
char  s2 [ 4 ]  =  { 'a', 'b',  'c', '\0' };
```

Örnek 4.3-6'da sadece boş karakterlerden oluşan s1 ve s2 dizisi bildirilir. Fakat çıktıda da görüldüğü gibi strlen fonksiyonu her iki dizinin eleman sayısı için 0 değerini

döndürür. Eleman sayıları **sizeof** operatörü kullanılarak hesaplandığında s1 dizisi için 1, s2 dizisi için ise 2 bulunur. ■

Örnek 4.3-5 bosdizgi.c programı.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char s1 [ ] = "";
   char s2 [ ] = "\0";

   printf("eleman sayisi, s1 :  %d \n", sizeof s1 / sizeof s1[0] );
   printf("eleman sayisi, s2 :  %d \n", sizeof s2 / sizeof s2[0]  );
   printf("strlen(s1) : %d, strlen(s2) : %d\n", strlen(s1), strlen(s2) );

   return 0;
}
```

Çıktı

```
eleman sayisi, s1 :  1
eleman sayisi, s2 :  2
strlen(s1) : 0, strlen(s2) : 0
```

4.4 C Önişlemcisi

C derleyicisi, kaynak dosyaları ilk olarak aslında C dilinin parçası olmayan bazı özel komutlar için tarar. # işareti ile başlayan bu özel komutlar (*directives*), derleme işleminin önişleme olarak adlandırılan ilk adımında, *C önişlemcisi* (*C preprocessor*) tarafından işlenir.

Önişlemci komutları şunlardır: kaynak dosya içine başlık dosyalarının alınması için kullanılan #include; sembolik sabitler ve makro tanımlamak için kullanılan #define ve kaynak dosya içinde bazı satırların derleyici tarafından belli koşullara bağlı olarak görülmesini sağlayan #if, #endif, #elif ve #else ve daha önce tanımlanmış bir makro ismini iptal eden #undef komutlarıdır.

#include komutu

Kaynak dosya içinde bulunan her,

#include <*dosya*>

yada

#include "*dosya*"

satırı, önişlemci tarafından başlık dosyası *dosya*'nın metni ile değiştirilir. Başlık dosyaları (*include files* yada *header files*) sık sık kullanılan #define satırlarını yada extern bildirimleri içerir ve bunların tekrar yazılmasını önlemek amacıyla kaynak metin içine #include önişlemci komutu yerleştirilerek alınırlar. #include satırları kaynak metin içinde içerdikleri bilgilerin kullanımından önce herhangi bir yerde bulunabilirler. Fakat genel olarak başlık dosyası isminden de anlaşılacağı gibi kaynak metnin ilk satırları #include satırlarıdır. #include ile kaynak metin içine alınan bir dosya başka #include satırları içerebilir.

Standart kütüphane fonksiyonlarının prototiplerini ve bazı standart makroları içeren başlık dosyaları *sistem başlık dosyaları* yada *standart başlık dosyaları* olarak adlandırılır. Örneklerin ilk satırında #include komutuyla, kullanılan kütüphane fonksiyonlarının prototiplerini ve makroları içeren standart giriş/çıkış başlık dosyası stdio.h'nin programlar içine alınması sağlandı.

Dosya ismi, açılı parantezler (< ve >) ile sınırlanmış ise *dosya* dosyası sistem başlık dosyalarının bulunduğu yer olarak belirlenen fihristte (sistem *include* fihristi) aranır. Bu iş için derleyiciler genel olarak işletim ortamındaki bir değişkene (*environment variable*) atanan fihrist yolu bilgisini kullanır. Aşağıdaki #include satırındaki x.h başlık dosyası, sistem başlık dosyalarının bulunduğu fihristteki sys alt-fihristinde aranır (DOS işletim sisteminde fihrist ayracı ters kesme işaretidir. Ayrıca bazı diğer işletim sistemlerinde, dosya isimlerinde büyük-küçük harf ayrımı yapıldığı dikkate alınmalıdır):

```
#include <sys\x.h>
```

Dosya ismi, çift tırnaklar arasında bulunuyor ise *dosya* dosyası bulunulan fihristte aranır; dosya ismi, bulunulan fihriste göre (*relative path*) yada kök fihristine göre (*full path*) verilen fihrist yolu bilgisini içerebilir.

Örnek olarak aşağıdaki include satırları verilebilir;

```
#include "..\baslik.h"
```

yada

```
#include "c:\derle\hata.msg"
```

#include kullanımı bir programın tüm kaynak dosyalarının aynı tanım ve bildirimleri içermesini sağlar. Başlık dosyasında değişiklik yapıldıktan sonra bu dosyayı alan tüm kaynak dosyalar tekrar derlenmelidir. Başlık dosyaları genel olarak .h uzantılıdır.

#define komutu

Bu komut kullanılarak program yazmada kolaylık sağlayan sembolik sabitler ve makrolar tanımlanabilir. #define kullanılarak sembolik sabit tanımı aşağıdaki şekilde yapılır:

```
#define  isim   degistirilecek_metin
```

#define komutu, sabit ismi *isim* ve *degistirilecek_metin* boşluk yada tab ile ayrılır. *isim* bir değişken ismi ile aynıdır; *degistirilecek_metin* ise herhangi bir metin olabilir. Kaynak metin içinde #define satırını izleyen tüm satırlardaki *isim*'ler, *degistirilecek_metin* ile değiştirilir. Bir #define satırı daha önceden tanımlanmış isimleri içerebilir.

Aşağıdaki satırda, 5 değeri yerine kullanılmak üzere SATIRSAYI sembolik sabiti tanımlanır:

```
#define SATIRSAYI  5
```

Sabit sayılar yerine #define ile tanımlanan sembolik sabitler kullanılarak program kodunun daha kolay anlaşılır olması sağlanabilir. Örneğin yukarıda tanımlanan sembolik sabit SATIRSAYI kullanılarak aşağıdaki **for** döngüsü oluşturulabilir:

```
for ( satir = 0; satir < SATIRSAYI; satir++ )
...
```

Bu döngüde satir değişkeninin son değerinin satır sayısı sınırı olduğu açıkça bellidir. Ayrıca satır sayısında değişiklik yapılması gerektiğinde, bu değişikliğin tüm program yerine sadece #define satırında yapılması yeterli olacaktır.

Makro satırı çok uzun ise \ ile izleyen satırdan devam edilebilir:

```
#define UZUNDIZGI      \
   "Bu bir denemedir"
```

\ karakteri ile satır sonunu işaretleyen NL (*newline*) karakteri gizlenir ve bu satırın tek bir satır olarak işlem görmesi sağlanır. Çok uzun makrolar bu yolla birkaç satıra bölünebilir.

Değiştirme işlemi çift tırnaklar ile sınırlı dizgilerde gerçekleşmez. Örneğin tanımlanan bir abc ismi için aşağıdaki puts deyiminde herhangi bir işlem yapılmaz:

```
puts( "abc" );
```

Program içinde bulunan bir abcd dizgisinde de herhangi bir işlem yapılmaz. Bu işlemin gerçekleşmesi için program metni içinde boşluk karakterleri ile sınırlı abc ismi aranır.

#define kullanılarak argüman kabul eden makrolar tanımlanabilir. Örneğin iki argümanından büyük olan argümanın değerini döndüren Maksimum makrosu aşağıdaki gibi tanımlanabilir (makro ismi ile sol parantez arasında boşluk bulunmamalıdır):

```
#define  Maksimum( x, y )   ((x) > (y)  ? (x) : (y))
```

Maksimum makrosu kullanıldığı program satırında önişlemci tarafından açıldığında, x ve y argümanlarının yerini programda girilen argümanlar alır. Bu makro aynı işlevi yerine getiren bir fonksiyondan farklı olarak, argümanların her ikisinin de aynı tipte olması koşulu ile farklı tiplerde argümanlar ile çağrılabilir. Argüman olarak verilen ifadelerin iki kez işlendiği dikkate alınacak olursa, ++ yada -- gibi operatörlerle oluşturulan ifadeler argüman olarak kullanılmamalıdır. Ayrıca ifadelerin işleniş sırasının önemli olabileceği dikkate alınarak ilave parantezler ile öncelikler kontrol edilmelidir. Örneğin aşağıdaki SayiKare makrosu hatalıdır:

```
#define SayiKare(x)    x * x
```

Bu makro i + 3 gibi bir ifade ile çağrıldığında hata oluşur. Bu makro, hatasız çalışması için ilave parantezler kullanılarak aşağıdaki şekilde tanımlanmalıdır:

```
#define SayiKare(x)    (x) * (x)
```

degistirilecek_metin içinde argüman isminden önce bulunan # işareti, argümanın makro açılımında çift tırnaklar arasına yerleştirilmesini sağlar.

```
#define  Print( x )    printf( #x " : %d\n", x )
```

Yukarıdaki Print makrosu i argümanı ile çağrıldığında açılımı aşağıdaki gibi olur:

```
printf( "i" " : %d\n", i );    yada  printf( "i : %d\n", i );
```

Önişlemci operatörü ##, makro açılımı sırasında argümanları birleştirir. Aşağıdaki makro den, ve eme argümanları ile çağrıldığında deneme ismini verir:

```
#define Ekle( a, b )    a ## b
```

Programlarda makroların kullanımı çok kolaylık sağlar. Standart başlık dosyalarında pek çok makro tanımı bulunur.

#if komutu

Koşul deyimleri kullanılarak önişleme kontrol edilebilir. Böylece derleme sırasında belli koşullara bağlı olarak belli program satırlarının seçilerek programa dahil edilmesi

sağlanabilir. Bu amaçla önişlemci komutları #if, #endif, #else ve #elif ile oluşturulan deyimler kullanılır:

```
#if  ifade1
...
#elif  ifade2
...
#elif  ifadex
...
#else
...
#endif
```

Burada *ifade1*, ..., *ifadex* sabit tamsayı ifadelerdir ve işlendikten sonra herhangi biri sıfır harici bir değer veriyor ise izleyen #elif (else-if), #else yada son olarak deyimi kapatan #endif satırına kadar olan satırlar programa dahil edilir.

#if komutunda kullanılan ve önişlemci operatörü defined ile oluşturulan defined *isim* ifadesinin değeri, eğer *isim* tanımlı ise 1'dir; aksi taktirde 0'dır.

Bu operatör defined yada !defined olarak kullanılabilir:

```
#if  !defined isim
...
#endif
```

yada

```
#if  defined isim
...
#endif
```

#if defined *isim* satırı yerine #ifdef *isim*; #if !defined *isim* satırı yerine ise #ifndef *isim* kullanılabilir.

Aşağıdaki satırlarda önişlemci komutlarının kullanımı örneklenmiştir:

Örnek 4.4-1 onisle.c programı.

```
#include <stdio.h>
#include <ctype.h>

#define MODUL1    1
#define MODUL2    2
#define MODUL3    3
#define MODUL4    4
```

...Örnek 4.4-1 Çıktı devam

```
#define    PROG       MODUL3
#if !defined DENEME
    #define DIZGI1 "DENEME tanimli degil"
#endif
#define DENEME
#if defined DENEME
    #define DIZGI2 "DENEME tanimlidir"
#endif
#if     PROG  ==  MODUL1
        #define ISIM  "MODUL1"
#elif   PROG  ==  MODUL2
        #define ISIM  "MODUL2"
#elif   PROG  ==  MODUL3
        #define ISIM  "MODUL3"
#else
        #define ISIM  "MODUL4"
#endif
int main(void)
{
  puts( DIZGI1 );
  puts( DIZGI2 );
  puts( ISIM  );
  printf( "L : %s\n", islower( 'L' ) ? "buyuk harf" : "kucuk harf"  ); /* islower makrosu      */
  #undef islower
  printf( "L : %s\n", islower( 'L' ) ? "buyuk harf" : "kucuk harf"  ); /* islower fonksiyonu   */
  return 0;
}
```

Çıktı

```
DENEME tanimli degil
DENEME tanimlidir
MODUL3
L : kucuk harf
L : kucuk harf
```

#undef komutu

Standart C kütüphane fonksiyonlarının bildirimleri başlık dosyalarında bulunur. Standart başlık dosyalarında bu fonksiyonlardan bazıları aynı zamanda yine aynı isimde makrolar olarak tanımlanmışlardır. Bu makrolar kullanıldıkları satırlarda önişlemci tarafından açıldıklarında fonksiyon çağrısından daha hızlı çalışan C ifadelerine dönüşürler ve fonksiyon ismini gizlerler. Fakat makrolar programlarda hata giderme ve izleme

işlemleri yapılırken sorun olabilir. Makro karşılığı yerine fonksiyonun kendisini kullanmak için #undef önişlemci komutu kullanılarak standart başlık dosyasındaki makro tanımı iptal edilebilir. ■

4.5 C Programlarını ayrı ayrı Derleme ve Bağlama

Herhangi bir C programının oluşturulması, bir editör program kullanılarak kaynak kodun yazılması ve bir dosyaya saklanması ile başlar. Bu dosya, C *kaynak dosyası* olarak adlandırılır. Bir programın farklı kısımları, farklı kaynak dosyalara yerleştirilebilir ve bu dosyaların herbiri ayrı ayrı derlenebilir. C programları fonksiyon haline getirilebilecek kısmı kalmayıncaya kadar bölünebilir. Tek bir kaynak dosyada yer alan programı, birkaç kaynak dosyaya bölmenin bazı avantajları vardır. Programın herhangi bir parçası üzerinde yapılan hata düzeltme yada değişiklik sonrası derleme, programın sadece ilgili kısmında yapılabilir. Bu yolla özellikle büyük programlar üzerinde çalışırken zaman kazanılmış olur.
Ayrıca üzerindeki tüm çalışmaların tamamlandığı fonksiyonları, diğerleri üzerindeki çalışmalar devam ederken tekrar tekrar derlemek gerekmez. C kaynak dosyası yada dosyaları, çalıştırılabilir program haline iki aşamada getirilir:

- Derleme (*compilation*): Bu aşamada derleyici program, C kaynak dosyalarını object dosyalara çevirir. Bir object dosyası (*object-code file*) binary kodlardan oluşur, fakat henüz çalıştırılabilir durumda değildir. C kaynak dosyaları, .c uzantılı dosyalardır. Object dosyalar ise, DOS işletim sisteminde .obj uzantılı, UNIX işletim sisteminde .o uzantılı dosyalardır.

- Bağlama (*linking*): Bu aşamada, *linker* olarak adlandırılan program (yada *linkage editor*), derleme sırasında oluşturulan object dosyalarını standart kütüphaneler (programlarda kullanılan standart kütüphane fonksiyonlarına ait kodları içeren .lib uzantılı dosyalar), programcı tarafından belirtilen diğer object dosyalar ve kütüphaneler ile birleştirerek çalıştırılabilir program dosyasını oluşturur. Bu dosya (DOS işletim sisteminde .exe uzantılı, UNIX işletim sisteminde ise .out uzantılı yada uzantısızdır), işletim sistemi altında çalıştırılmaya hazırdır.

Yukarıda da belirtildiği gibi bir C programını oluşturan fonksiyonların ve external değişkenlerin aynı anda derlenmesi gerekmez. Programın kaynak kodu birkaç dosyaya yerleştirilebilir, ayrıca bu dosyalar daha önce derlenmiş kütüphane fonksiyonları ile bağlanabilirler. Fakat bu işlemlerin hatasız gerçekleşmesi dosyalar arasında paylaşılan verilerin bildirimlerinin uygun şekilde düzenlenmesine bağlıdır. Bunu sağlamak için başlık dosyaları kullanılır. Fonksiyon prototipleri, sembolik sabitler (#define ile tanımlanan) ve paylaşılan verilere ait bildirimler .h uzantılı bir başlık dosyasına yerleştirilir. Bu dosya ihtiyaç duyulan kaynak dosyalar içine #include önişlemci komutu kullanılarak dahil edilir. Her kaynak dosyanın sadece ihtiyaç duyduğu bilgilere erişmesi için ayrı bir başlık dosyası oluşturulabilir. Fakat bu pratik bir yol değildir. Genellikle

belli bir program büyüklüğüne kadar ihtiyaç duyulan tüm bilgileri içeren tek bir başlık dosyasının bulunması daha pratiktir. Fakat kapsamlı projelerde pek çok başlık dosyası bulunabilir.

Herhangi bir kaynak dosya (modül) içinde bir external değişken yada fonksiyona, bildirildiği noktadan dosya sonuna kadar her yerde erişilebilir. Fakat external değişkene, tanımı öncesinde erişim gerekiyor ise yada bu değişken kullanıldığı dosyadan farklı bir kaynak dosyada tanımlanmış ise **extern** bildirim gereklidir. **extern** bildirimler link programına bilgi verir. Aynı kural fonksiyonlar için geçerli olmadığından dolayı herhangi bir kaynak dosyada tanımlanan bir fonksiyona, **extern** bildirim olmadan diğer kaynak dosyalardan erişilebilir. Burada bir değişkenin bildirimi ve tanımı arasındaki fark önemlidir.
Bildirim değişkenin ismini, tipini ve diğer özelliklerini belirtir. Tanım ise bildirimi de içerir ve aynı zamanda derleyici yada linker tarafından değişken için bellekte yer ayrılmasını sağlar. Tanımlar belirtilen veriler için gerçekte bellek alanı ayrılmasını sağlar; bildirimler ise derleyiciye başka bir yerde tanımlanmış verinin özellikleri hakkında bilgi vermek için oluşturulur. ANSI standardında kaynak programı oluşturan tüm modüller arasında bir external değişkenin sadece tek bir tanımı bulunabilir; tanımı içeren dosyada ve diğer dosyalarda extern bildirimler bulunabilir.

Bu durum bir global (external) değişkenin biri hariç tüm bildirimlerinin önüne extern yerleştirilerek gerçekleştirilebilir. Fakat her zaman bu kadar basit olmayabilir. Birden fazla modülden oluşan programlar genellikle çok sayıda veriyi paylaşır ve bazıları extern içeren ve bazıları içermeyen bildirimlerin ayrı listelerini ayrı modüllerde bulundurmak kolay değildir. Tercih edilen yol yukarıda bahsedildiği gibi tüm verilerin tekbir .h dosyasında bulundurulmasıdır. Fakat ANSI standardında belirtildiği gibi eğer bildirimler/tanımlar ayrı ayrı dosyalarda farklı yazılmak durumunda ise tek bir başlık dosyası kullanılamaz. Bu durumda tek bir başlık dosyası kullanımı mümkün olmayabilir. Fakat bunun pratik bir çözümü vardır. Aşağıdaki satırlar verilerin sadece ana modülde tanımlanmış olmasını ve diğer modüllerde de extern olarak bildirilmiş olmasını garanti eder.

.h dosyası

```
...
#ifdef ANA_MODUL
  #define EXTERN
#else
  #define EXTERN extern
#endif
...
EXTERN bildirimler...
...
```

dosya1.c
...
#define ANA_MODUL
...
EXTERN *bildirimler...*
...

Sembolik sabit kullanılarak kritik bildirimler için **extern** anahtar kelimesinin kullanımı kontrol edilir. Eğer dosyada ANA_MODUL tanımı var ise o zaman programın ana modülü derleniyor demektir ve sembolik sabit EXTERN hiç bir şey anlamına gelmez.

Aksi taktirde EXTERN, **extern** olarak tanımlanır ve bulunduğu satırlar ise external verilerin bildirimi olarak iş görür. Ana modül derleniyor iken tanımların oluşturulmasını sağlamak için başlık dosyasının dahil edildiği satır öncesine #define ANA_MODUL yerleştirilir. Programın diğer modülleri bu tanımı içermez. Tanım ve bildirim ayrımı konusunda derleyiciler farklılık gösterebilir.

Bazı linker programları (bağlayıcılar) bir programın kaynak dosyaları arasında aynı veri elemanlarının birden fazla tanımının bulunmasına izin verirler. Fakat birden fazla ilk değer atama bulunamaz, aksi taktirde link sırasında hata oluşur. Bu durum, tek bir .h dosyası kullanımına izin verdiği için geliştirme işlemini basitleştirse de, programın taşınabilirliğini azaltır.

Dosyalar ayrı ayrı derlendiği için bir fonksiyonun gerçek adresi ile çağrı arasındaki ilişki ve global verilere erişim, çalıştırılabilir program kodunun oluşturulması sırasında linker tarafından kurulan *bağlar* (*link*) ile gerçekleşir. Linker bu işi yaparken derleme sırasında object dosyalar içine yerleştirilen bilgileri kullanır. Sonuç olarak bağlama (linking), farklı fonksiyonlar yada farklı kaynak dosyalar içinde aynı değişken isminin kullanımı ile aynı bellek alanına erişimin gerçekleşmesini sağlar. Bir modül, bir başka modülde tanımlanmış bir isme erişebiliyor ise bu ilişki, *dış erişim* (*external reference*) olarak adlandırılır. Dış erişimler linker tarafından oluşturulur.

Bir ismin (değişken yada fonksiyon ismi) *bağ özelliği*, bu ismin programın hangi kısımlarında kullanılabileceğine, dolayısıyla link programının bu isim üzerinde yapacağı işlemlere karar verir. Bir değişken isminin bağlanma özelliği, önceki bölümlerde anlatıldığı gibi bazı anahtar kelimelerle yada program içinde bulunduğu yer ile belirlenebilir.

Bir isim, *iç* yada *dış bağlanma özelliğ*ine (*internal yada external linkage*) sahip olabilir veya hiçbir bağlanma özelliği bulunmayabilir (*no linkage*). Çalıştırılabilir program oluşturmak üzere derlenip bağlanma işlemi yapılmış bir grup kaynak dosya içinde, dış bağlanma özelliğine sahip bir değişken isminin kullanımı, ortak bir bellek alanına erişimi gerçekleştirir. Programın farklı modüllerinde bulunan aynı isimli değişkenler, aynı bellek

alanında bulunurlar. Dolayısıyla farklı dosyalarda bu değişkenlerin kullanımı aynı veriye erişim anlamındadır.

Tek bir kaynak dosya içinde, iç bağlanma özelliğine sahip bir değişken isminin kullanımı da aynı şekilde ortak bir bellek alanına erişim anlamına gelir.
Bir değişken ismi, bu iki bağlanma özelliğinden herhangi birine sahip olmayabilir (no linkage). Bu durumda sadece tanımlandığı blok içinden erişim mümkündür.

Bir programdaki bağlanma özelliği olmayan isimler, blok erişimli (blok ile sınırlı görünürlük alanı) fakat **extern** ile bildirilmeyen isimlerdir (fonksiyon parametreleri, fonksiyon ve değişken isimleri hariç herşey; **goto** etiketi, yapı etiketi, yapı elemanı ismi, **typedef** ismi, yada **enum** sabiti).

Global değişkenler ve fonksiyonlar (tüm blokların dışında ve **static** anahtar kelimesi olmadan bildirilen), programdaki tüm kaynak modüllerden erişilebildikleri için (görünürlük alanları programın tamamıdır; *program scope*) kendiliğinden dış bağlanma özelliğine sahiptirler. Bir global değişken yada fonksiyon, **static** anahtar kelimesi ile bildirildiğinde bulunduğu dosya ile sınırlı görünürlük alanına (*file scope*) sahip olduğu için iç bağlanma özelliğine sahiptir. Fakat **static** anahtar kelimesinin, blok içi ve dışı kullanımına dikkat etmek gerekir. Çünkü blok ile sınırlı görünürlük alanına (*block scope*) sahip bir değişkene **static** uygulandığında, sadece varolma süresi etkilenir ve static süreli olur. Tüm blokların dışında bulunan (*file scope*) bir değişkene uygulandığında ise bu değişkenin bağlanma özelliği değişerek dış bağlanmadan iç bağlanmaya dönüşür.

Aşağıdaki örnekte yer alan iki ayrı kaynak dosyada, farklı bağ tipleri gösterilir. Bu iki dosya, tek bir çalıştırılabilir program oluşturur. Derleme ve birleştirme işlemleri, işletim ortamına göre değişir.

Örnek 4.5-1 bagla1.c ve bagla2.c programı.

```
/*
 * bagla1.c
 */
#include  <stdio.h>

static void fa( int );

int main( void )
{
   extern void fa( int );     /* internal linkage    */
   extern void fb( void );  /* external linkage   */

   fa( 123 );
   fb( );

   return 0;
}
```

...Örnek 4.5-1 devam

```
int i = 13;                 /* external linkage   */
static void fa( int i )     /* internal linkage   */
{
   for (;;)
   {
      double i = 12.0;      /* no linkage         */
      printf("%f\n", i );
      goto fa_son;
   }
fa_son:                     /* no linkage         */
   printf("%d\n", i );
}

/*
 * bagla2.c
 */
#include <stdio.h>
extern int i;               /* external linkage   */
static int j = 45;          /* internal linkage   */
void fb( void )             /* external linkage   */
{
   printf("%d\n", i );
   printf("%d\n", j );
}
```

Çıktı

```
12.000000
123
13
45
```

---

fa fonksiyonu, **static** ile bildirildiği için sadece 1. kaynak dosya içinde erişilebilir. Bu nedenle iç bağlanma özelliğine sahiptir. İlk dosyada tanımlanan i değişkeni, **static** olarak bildirilmediği için dış bağlanma özelliğine sahiptir ve ikinci dosyada aynı isimle **extern** kullanılarak bildirilen bir değişken buradaki değere erişebilir. fa içinde bildirilen **double** i değişkeninin bağlanma özelliği yoktur çünkü blok içinde ve **extern** olmadan bildirilmiştir. İkinci dosyada, **static** belirleyici ile bildirilmediği için fb fonksiyonu dış bağlanma özelliğine sahiptir. Dolayısıyla ilk kaynak dosyadan çağırma mümkün olur. İkinci dosyada bulunan j değişkeni **static** ile bildirildiği için iç bağlanma özelliğine sahiptir ve diğer dosyalardan erişim mümkün değildir.

```
extern void fa( int );    /* internal linkage   */
extern void fb( void );   /* external linkage   */
```

Yukarıdaki **extern** bildirimler, dosya içinde görünürlüğe sahip ve tüm blokların dışında tanımlanmış (*file scope*) bir isim ile bağlantı yapmak anlamına gelir.
main içindeki fa fonksiyonu bildirimi aynı dosyadaki aynı isimli fonksiyon ile bağlanır ve böylece iç bağ oluşturur. **main** bloğundaki **extern** bildirimler öncesinde fa için **static** bildirim yapılmalıdır çünkü derleyici bunun dış bağlanma özelliğine sahip olduklarını kabul edebilir ve aynı dosyada daha sonra bulunan gerçek fonksiyon tanımları ile uyuşmaz. Derleyici fb'nin dış bağlanma özelliğine sahip olduğunu kabul eder.

Aşağıdaki satırlarda bir kaç kaynak dosyadan oluşan bir programda, veri paylaşımı için dosya içinde tüm blokların dışında bulunan çeşitli bildirim ve tanımlar incelenmiştir:

Dosya1:
```
int a = 1;
```
Dosya2:
```
extern int a;
```

a değişkeni dosyalar arasında paylaşılır ve ilk değeri 1'dir.

Dosya1:
```
extern int a;
...
int a;
```
Dosya2:
```
extern int a;
```

a değişkeni dosyalar arasında paylaşılır ve ilk değer olarak 0 taşır.

Dosya1:
```
int a;
```
Dosya2:
```
int a;
```

Hatalı tanım. (bazı derleyiciler izin verir)

Dosya1:
```
int a = 1;
```
Dosya2:
```
int a = 2;
```

Linker hata verir. İki kez tanımlama girişimi.

Dosya1:
```
static int a = 1;
```
Dosya2:
```
int a = 2;
```
Dosya3:
```
extern int a;
```

Dosya1'de bulunan a değişkeni diğer dosyalarda bulunan değişkenler ile ilgisizdir.
Dosya2 ve Dosya3 ilk değeri 2 olan a değişkenini paylaşır.

Dosya1:
```
static int a = 1;
extern int a;
```
Dosya2:
```
int a = 2;
```
Dosya3:
```
extern int a;
```

Yukarıdaki bildirim/tanımlar ile aynıdır.

Dosya1:
```
extern int a;
```
Dosya2:
```
extern int a;
```

Hata. a değişkeni tanımlanmamıştır.

Standart C kütüphanesi standart girişten okuyan (klavye) ve standart çıkışa yazan (ekran) pek çok hazır fonksiyon ve makro tanımı içerir. Kitaptaki örnek programlarda klavyeden girilen bilgileri formatlı olarak almak için standart kütüphane fonksiyonu scanf

kullanıldı. Ayrıca çeşitli bilgileri ekrana formatlı olarak listelemek için yine standart kütüphane fonksiyonu printf kullanıldı. C kütüphanesinde karakter giriş/çıkış işlemleri için getchar ve putchar makroları bulunur. Bu makro ve fonksiyonların kullanımına Bölüm 9'da yer verilmiştir. ■

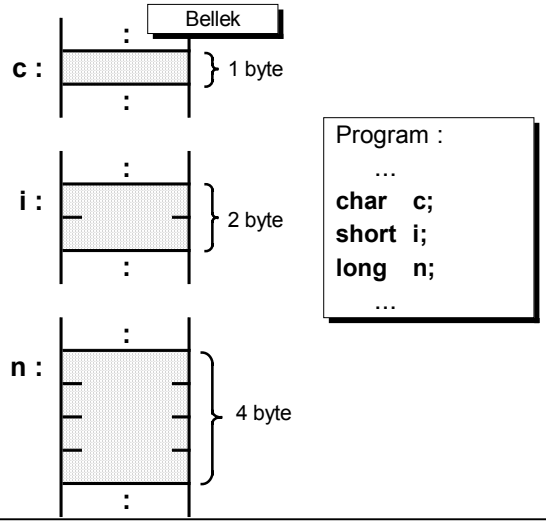
BÖLÜM 5: **Adres Değişkenleri**

Adres değişkenleri (*pointer variable* yada kısaca *pointer*) C dilinin güçlü olduğu konulardan biridir. Adres değişkenleri kullanılarak verimli ve kısa C programları oluşturulabilir. C dilinde adres değişkenleri ve diziler birbirleri ile çok yakın ilişkilidir. Bu bölümde adres değişkenlerinin bildirimi ve programlarda kullanımı anlatılacaktır. ■

5.1 Adres Değişkeni Bildirimi

Değişken bildirimleri içeren bir program derlendiğinde, bellekte değişkenlerin her biri için bildirimde yer alan *tip*te verileri saklayabilecek büyüklükte ayrı bir alan ayrılır. Ayrıca bellekte ayrılan her alanın (dolayısıyla değişkenin) bir bellek adresi vardır. Programda değişkene atanan değerler bu adres kullanılarak erişilebilen bellek alanında saklanır. Şekil 5.1-1'de, bildirilen üç değişken için bellekte ayrılan alanlar sembolik olarak gösterilir:

Şekil 5.1-1 c, i, ve n değişkenlerinin sembolik bellek görünümü.



Şekil 5.1-1'deki kutuların her biri 1 byte bellek alanını temsil eder. Buna göre **char** tipindeki c değişkeni için 1 byte bellek alanı ayrılmıştır. Bu değişkene 1 byte büyüklüğünde değerler atanabilir. Aynı şekilde, **short** tipindeki i ve **long** tipindeki n değişkenleri için sırasıyla 2 ve 4 byte bellek alanları ayrılmıştır. Sembolik olarak gösterilen bu bellek alanlarının her birinin ayrı bir bellek adresi vardır.

C dilinde, bellekte tutulan değerlerin bellek adreslerini almak için *adres operatörü* (*address operator*) & kullanılır. Adres operatörü bir *unary* operatör olduğu için tek operand kabul eder ve herhangi bir değerin atanabileceği ve verilerin saklanabileceği bellek alanına, dolayısıyla bellek adresine sahip değişkenlere ve dizi elemanlarına uygulanabilir (ayrıca izleyen bölümlerde anlatılacak olan yapı değişkenlerine de uygulanabilir). Bellekte tutulmadıklarından ve dolayısıyla bellek adresleri bulunmadığından sabit değerlere, **register** değişkenlere ve C operatörleri ile oluşturulan ifadelere uygulanamaz.

NOT:

Nesne, ***lvalue*** ve ***rvalue***

C programları, programcı tarafından tanımlanan ve programın çalışması sırasında var olan ***nesne***ler (değişkenler, diziler, sabitler gibi) içerir. Programda bulunan bildirimler ve deyimler aracılığı ile bu nesneler oluşturulur ve işlenir.

Bir nesne (*data object* yada *object*) değerlerin saklanabileceği ve daha sonra alınabileceği bir bellek alanıdır. Nesnenin özellikleri (*attributes*: önceki bölümlerde anlatılan saklama sınıfı ve tipi) saklayabileceği değerlerin sayısını, tipini ve bellek düzenini belirtir.

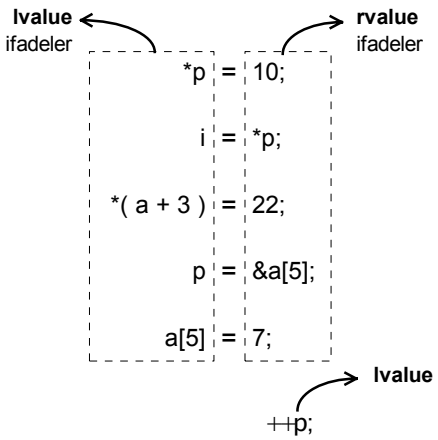
Nesnenin içerdiği *değer* (*data value*) sayı, karakter yada bir başka nesnenin adresi olabilir ve bir bit grubu (ikili sayı sisteminde) ile temsil edilir. Bir nesne tek bir değer (*scalar*; örnek olarak bir basit değişken) yada değer grubu (*aggregate*; örnek olarak dizi ve yapı değişkeni) içerebilir. Sonuç olarak, bir nesne bilgisayar belleğindeki bir bilgi saklama alanı; değer ise belli bir bit grubudur.

Bir değişken (*variable*), programcı tarafından tanımlanan ve isimlendirilen bir nesnedir. Değişkenin değeri programda var olduğu süre içerisinde atama işlemi ile değiştirilebilir. Değişken, nesnenin özel bir şeklidir ve dolayısıyla bir değişken için *değişkenin değeri* (*rvalue*), *değişkenin yeri* (*lvalue*), *değişkenin ismi* (*identifier*) yada *değişkenin tipi* terimleri kullanılabilir.

lvalue (*left value)*, bir atama deyiminin solunda kalan ifade yada isimdir ve bir nesneye erişir. Değişkenler, dizi elemanları ve adres değişkenleri lvalue'dır. &, ++ ve -- operatörleri ancak lvalue ifadelere uygulanabilir.

rvalue (*right value*) ise bir atama deyiminin sağında bulunan ifadedir ve değeri atama deyiminin solundaki lvalue ile erişilen nesneye atanır. Bir rvalue ifadesi herhangi bir nesne erişimi sağlamayabilir. Dizi ismi, fonksiyon, sabit ve lvalue olan her ifade bir rvalue'dır.

Aşağıda tamsayılara işaret eden p adres değişkeni, tamsayı değişken i ve tamsayı değerlerden oluşan a dizisi ile oluşturulan deyimler kullanılarak lvalue ve rvalue ifadeler örneklenir:



Adres operatörü'nün genel kullanım şekilleri aşağıda verilmiştir:

&*degisken_ismi*

yada

&*dizi_ismi*[*indeks*]

dizi_ismi[*indeks*] ifadesi, herhangi bir dizinin tamsayı *indeks* ile belirlenen elemanıdır. Bellek adresleri donanıma bağlı olarak 2 yada 4 byte büyüklüğünde değerlerdir. Adres operatörü (&) kullanılarak yukarıdaki şekilde alınan adres değeri, 2 yada 4 byte büyüklüğünde adres değerlerini saklayabilecek *adres değişkeni*ne (*gösterge değişkeni* yada kısaca *gösterge* olarakta adlandırılır) atanabilir.
Adres değişkenleri, taşıdıkları adres değerlerinin kullanılmasına ve bu değerler üzerinde işlemler yapılabilmesine olanak sağlar.

Aşağıdakiler, adres operatörü ile oluşturulabilecek geçerli C deyimleridir:

```
pi = &i;
adrDizi = &Dizi[10];
```

İlk deyimde daha önce uygun bir veri tipi ile bildirilmiş i değişkeninin adres operatörü kullanılarak alınan bellek adresi, adres değişkeni pi'ye atanır. İkinci deyimde ise Dizi dizisinin 10 no.'lu elemanının (11. elemanı) adresi, adrDizi adres değişkenine atanmıştır.

Örnek 5.1-1'de adres operatörü kullanılarak Dizi dizisinin elemanlarının bellek adresleri listelenir:

Örnek 5.1-1 adresop1.c programı.

```
#include <stdio.h>
int main(void)
{
  int Dizi[ ] = { 0, 1, 2 };
  int i;
  for ( i = 0; i < 3; i++ )
      printf("Dizi[%d] icin bellek adresi : %u\n", i, &Dizi[ i ] );
  return 0;
}
```

Çıktı

```
Dizi[0] icin bellek adresi : 3458
Dizi[1] icin bellek adresi : 3460
Dizi[2] icin bellek adresi : 3462
```

Aşağıda adres operatörü'nün hatalı kullanımları örneklenmiştir. Bu deyimler, derleme sırasında hataya sebep verir ve program derlenemez:

```
/* adres operatörü register saklama sınıfına sahip değişkenlere uygulanamaz. */
register short i;
short *pi;
pi = &i;
pi = &( i + 5 );    /* ifadelerin adresi alınamaz.          */
pi = &++i;          /* ifadelerin adresi alınamaz.          */
pi = &4;            /* adres operatörü sabitlere uygulamaz. */
```

Örnek 5.1-2'de kullanılan ve bundan sonra da programlarda çok sık olarak kullanılacak olan **sizeof** operatörü, *ifade*'nin büyüklüğünü byte olarak döndürür. Döndürdüğü değer standart C kütüphanesinde işaretsiz tamsayı olarak tanımlanmış olan size_t tipindedir ve bu tip kullanılarak bildirilen bir değişkene atanabilir:

```
size_t i;
i = sizeof ifade;
```

ifade, bir değişken ismi (basit değişkenin, adres değişkeninin, dizinin yada yapının ismi) yada **(*tip-belirleyici*)** şeklinde oluşturulan bir tip-çevirme (*type-cast*) ifadesidir. Dizi ismine uygulandığında dizinin tamamının byte olarak büyüklüğünü; dizgi sabitine yada değişkenine uygulandığında ise toplam büyüklüğü dizgi sonunu işaretleyen '\0' karakteri de dahil olmak üzere verir.

NOT:

sizeof operatörü (*unary operator* : yani tek operand alır), daha sonraki bölümlerde anlatılacak olan *yapı tipi*ne yada *yapı değişkeni* ismine uygulandığında döndürdüğü byte sayısı gerçek byte (donanıma ve derleyiciye bağlı olarak bellek düzeni gereği yapı içinde bulunabilecek boşlukların da (*padding bytes*) dahil edildiği) miktarıdır. **sizeof** bir derleme-zamanı operatörüdür (*compile-time operator*). Dizi boyutları derleme sırasında belli olduğu için, **sizeof** kullanılarak dizinin eleman sayısı derleme sırasında saptanabilir.

sizeof operatörü dizgi uzunluğunu '\0' dahil olmak üzere verir. Standart C kütüphanesinde yer alan **strlen** fonksiyonu ise *s* dizgisinde bulunan karakterlerin sayısını dizgi sonunu işaretleyen '\0' karakteri hariç olmak üzere verir:

```
size_t strlen( const char *s );          /* string.h */
```

Aşağıdaki programda **sizeof** operatörünün çeşitli uygulamalarına yer verilmiştir:

```
/*
 * 💾 sizeof.c programi.
 */
#include <stdio.h>
#include <string.h>
#define  DiziBoy(s)  (sizeof s / sizeof s[0])

int main(void)
{
  long n;
  int  j[] = { 0, 1, 2 };
  static int  jj[ DiziBoy(j) ];

  char *s  = "1234";
  char t[]  = "1234";
  size_t ELSAYI = sizeof j / sizeof j[0];

  printf( "n : %u byte, j : %u byte, jj : %u byte\n",      sizeof n,  sizeof j,
                                                           sizeof jj            );
  printf( "-sizeof- s : %u byte, t : %u byte\n",           sizeof s,  sizeof t  );
  printf( "-strlen- s : %u karakter, t : %u karakter\n",   strlen(s), strlen(t) );
  printf( "char : %u, int : %u, long : %u byte \n",        sizeof (char),
                                                           sizeof (int),
                                                           sizeof (long)        );

  printf( "j dizisinin eleman sayisi : %u \n",       ELSAYI        );
  printf( "jj dizisinin eleman sayisi : %u \n",      DiziBoy( jj ) );
  printf( "j dizisi int[3] tipindedir : %u byte\n",  sizeof (int[3]) );

  ELSAYI = sizeof t / sizeof t[0];
  printf( "-sizeof- t dizisinin eleman sayisi : %u \n",  ELSAYI );
  printf( "sizeof 2L : %u byte, sizeof 2 : %u byte\n", sizeof 2L,  sizeof 2 );
```

... sizeof.c devam

```
  printf( "sizeof \"1234\" : %u byte\n",          sizeof "1234"  );
  printf( "strlen(\"1234\") : %u karakter\n",       strlen("1234") );

  return 0;
}
```

Çıktı

```
n : 4 byte, j : 6 byte, jj : 6 byte
-sizeof-s :  2 byte,        t  :  5 byte
-strlen-   s :  4 karakter,  t  :  4 karakter
char : 1, int : 2, long : 4 byte
j dizisinin eleman sayisi  :  3
jj dizisinin eleman sayisi :  3
j dizisi int[3] tipindedir  :  6 byte
-sizeof- t dizisinin eleman sayisi : 5
sizeof 2L : 4 byte, sizeof 2 : 2 byte
sizeof "1234"     :  5 byte
strlen("1234") :  4 karakter
```

Örnek 5.1-2 adresop2.c programı.

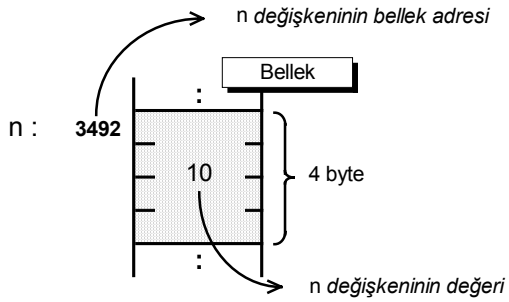
```
#include <stdio.h>
int main(void)
{
  long n = 10;
  printf( "n degiskeni : %u byte, bellek adresi : %u\n", sizeof ( n ), &n );
  printf( "adres degeri : %u byte.\n", sizeof ( &n ) );
  return 0;
}
```

Çıktı

```
n degiskeni : 4 byte, bellek adresi : 3492
adres degeri : 2 byte.
```

Çıktıda görüldüğü gibi, **long** tipinde tanımlanan n değişkeni için 4 byte bellek alanı ayrılmıştır. Bu alanın adresi (&n), 2 byte büyüklüğünde bir tamsayıdır (3492). Bu nedenle programın derlendiği ve çalıştırıldığı ortamda adres değişkenleri 2 byte büyüklüğündedir (farklı bir donanımda 4 byte olabilir).

Şekil 5.1-2 n değişkeninin sembolik bellek görünümü.



Bir değişkenin değerine, değişkenin bellek adresi kullanılarak erişilebilir. Adresler üzerinde aritmetik işlemler yapmak mümkündür. C dilinde, bu uygulamaları gerçekleştirebilmek için adres değişkenleri kullanılır. Örnek 5.1-3'de, long *p; deyimi ile p adres değişkeni bildirilir. Program derlendiğinde **long** tipindeki n ve k tamsayı değişkenleri için herbiri 4 byte büyüklüğünde, sırasıyla 3518 ve 3522 adresli bellek alanları ayrılır.

Örnek 5.1-3 adresop3.c programı.

```
#include <stdio.h>

int main(void)
{
  long  n;
  long  k;
  long  *p;

  n = 1650;
  p = &n;
  k = *p;

  printf( "adresler: &n : %u, &k : %u\n",  &n,  &k );
  printf( "degerler:  p : %u, *p : %ld, n : %ld, k : %ld\n",   p,  *p,  n,  k );
  printf( "p : %u, n : %u, k : %u byte\n",   sizeof p,   sizeof n,  sizeof k );

  return 0;
}
```

Çıktı

```
adresler:  &n  :  3518, &k  :  3522
degerler:  p   :  3518, *p  :  1650, n   :  1650, k :  1650
p : 2, n : 4, k : 4 byte
```

Adres değişkeni bildirim deyimi,

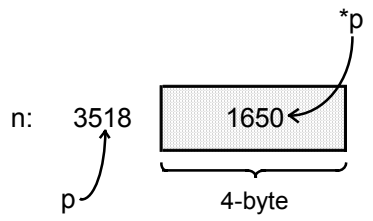
```
long *p;
```

ile p adres değişkeni için 2 byte büyüklüğünde bellek alanı ayrılır. Bildirim deyiminde **long** veri tipi yer aldığı halde p adres değişkeni için 4 byte yerine 2 byte bellek alanı ayrılmıştır.

n = 1650; atama deyimi ile n değişkeni için ayrılmış olan bellek alanına 1650 değeri yerleştirilir.

İzleyen satırda bulunan p = &n; deyimi ile n değişkeninin adres operatörü kullanılarak alınan bellek adresi (3518) p adres değişkenine atanarak, p adres değişkeninin n değişkenine *işaret etmesi* sağlanır (Şekil 5.1-3). Böylece p adres değişkeni kullanılarak n değişkeninin bellek alanına erişilebilir. Adres değişkeni bildirim deyiminde kullanılan * işaretinin, k = *p; atama deyiminde de kullanıldığı görülüyor. Bu satırda p adres değişkeninin işaret ettiği bellek alanında bulunan değer alınarak (yani n değişkeninin değeri olan 1650), k değişkeninin bellek alanına kopyalanır. Çıktıda da görüldüğü gibi, her iki değişkenin de (n ve k) bu atama işlemi sonrası değeri 1650 olur.

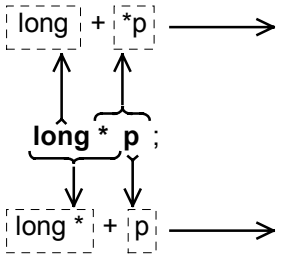
Şekil 5.1-3 p adres değişkeni.



Örnek programda, * operatörünün iki ayrı kullanımı yer alır:

- long *p; bildirim deyiminde,

  * operatörü bildirim deyimlerinde adres değişkeninin tipinin bir parçasıdır. Adres değişkeni bildirim deyimi iki şekilde yorumlanabilir: p adres değişkeninin indirek değeri bir **long** tamsayıdır (1); p adres değişkeni **long *** tipindedir (2):



(1) * *operatörü adres değişkeninin indirek değerini verdiği için (*k = *p; *atama deyiminde olduğu gibi),* *p *ifadesi ile alınan değer* **long** *tamsayıdır.*

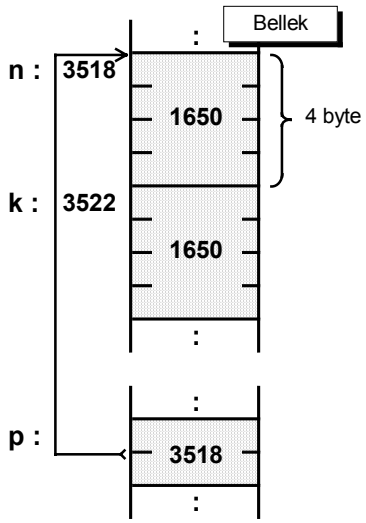
(2) p *adres değişkeni* **long *** *tipindedir. Yani* **long** *tamsayı değişkenlere işaret eder.*

- k = *p; atama deyiminde,

  * operatörü, C operatörleri ile oluşturulan ifadelerde adres değişkenine uygulanarak işaret edilen bellek alanındaki değere erişilir. Dolayısıyla adres değişkeninin indirek değerini verir.

n değişkeni, p adres değişkeninin *işaret ettiği (gösterdiği) değişken* olarak adlandırılır. Çünkü p'nin taşıdığı değer (3518) n değişkeninin bellek adresidir. n değişkeninin değeri olan 1650, adres değişkeninin *indirek değeri* olarak adlandırılır. n değişkeninin değerine indirek olarak erişim için kullanılan *p ifadesinde yer alan * işareti de, *indirek değer operatörü* (*indirection* yada *dereferencing operator*) olarak adlandırılır. Şekil 5.1-4'de değişkenlerin sembolik bellek görünümleri verilmiştir.

Şekil 5.1-4 p adres değişkeni, n değişkenine işaret eder.



! *Değişkenlerin bellekteki bulunuşlarının sembolik olarak gösterildiği şekillerde, değişkenler bellekte bitişik olarak gösterilir (örnek olarak n ve k değişkenleri). Fakat donanımın bellek düzenlemesinin bir gereği olarak (alignment) değişkenler bellekte her zaman peşpeşe bulunmayabilir.*

Adres değişkenleri işaret edecekleri değişkenlerin tiplerine bağlı olarak herhangi bir temel tip ile bildirilebilir. Fakat adres değişkeni bildirim deyiminde yer alan veri tipi (* işareti çıkarıldığında elde edilen temel veri tipi), adres değişkeninin işaret edeceği değişkenin tipi ile aynı olmalıdır.

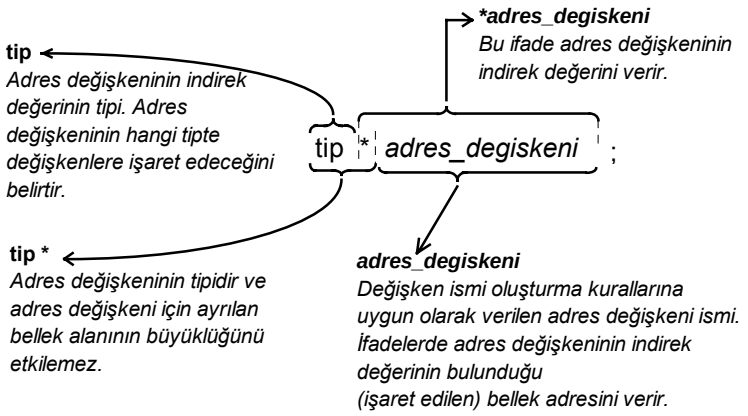
Böylece adres değişkeninin hangi tipteki değişkenlerin bellek adresini taşıyacağı belirlenmiş olur. Tipler farklı olduğunda, derleme sırasında hata oluşur.

NOT:
İndirek değer operatörü tek operand alır (*unary operator*) ve çift operand alan çarpma operatörü * (*binary operator*) ile karıştırılmamalıdır. Özellikle, i *= j; gibi deyimler yanıltıcı olabilir.

Örnek 5.1-4'de **int *** tipinde pi adres değişkeni, **long *** tipinde pl adres değişkeni ve **double** * tipinde px adres değişkeni bildirilir. Programda adres değişkenlerinin tiplerine **sizeof** operatörü uygulanarak byte sayıları listelenir. Çıktıda da görüldüğü gibi, bellekte i ve l değişkenleri için farklı büyüklüklerde alanlar ayrılmış olmasına rağmen bu değişkenlere işaret eden adres değişkenleri için aynı büyüklükte bellek alanları ayrılmıştır.

Bir bilgisayardaki bütün adres değişkenleri aynı büyüklüktedirler (donanımına bağlı olarak 2 yada 4 byte). Sonuç olarak adres değişkeninin bellek alanı büyüklüğü ve düzeni, işaret ettiği değişkenin tipine değil donanıma bağlıdır. Dikkat edilecek olursa ayrılan bellek alanı **int** tipi değişkenler için ayrılan bellek alanı ile aynı büyüklüktedir.

Şekil 5.1-5 Adres değişkeni bildirim deyimi.



Programlarda görüldüğü gibi bir adres değişkeni de bellekte saklanır ve dolayısıyla değer atanabilecek bellek alanına ve bir bellek adresine sahiptir (*lvalue*).

Bu nedenle adres değişkenleri ile **++** ve **--** operatörleri kullanılabilir ve hatta bir *adres değişkenine işaret eden adres değişkeni* bildirilebilir (bu konu izleyen bölümlerde anlatılacaktır).

Ayrıca ***** operatörü ile oluşturulan ******adres_degiskeni* ifadesine de değer atanabilir (*lvalue*); **++** ve **--** operatörleri uygulanabilir.

NOT:
Örnek programlarda adres değerleri ekrana listelenirken printf format dizgisinde %u format belirleyici kullanıldı. %u format belirleyici printf fonksiyonunun argümanı olan değerin, işaretsiz (-u-nsigned) tamsayı tip olarak işlem görmesini sağlar. Bu durumda adres değerleri ekrana ondalık formatta yazılır. Fakat bir bellek adresi işaretsiz tamsayı tipinde değildir; kendine özgü bir tiptir. Bu programların oluşturulduğu donanımda bellek adresi ile bir int değer aynı byte büyüklüğünde olmasına karşın, bir başka donanımda byte sayıları eşit olmayabilir. Sonuç olarak adres değeri bir tamsayı değildir ve bu değer kullanılarak belirli bazı adres işlemleri yapılabilir.

printf fonksiyonu adres değerlerinin derleyiciye bağlı formatta (genel olarak onaltılık sayılar olarak) temsil edilebilmesi için %p format belirleyicisine sahiptir.
Fakat programlarda adres aritmetiğinin anlaşılabilir olması için açısında çoğunlukla %u kullanılmıştır.

Örnek 5.1.4'deki *pi = 42; atama deyiminde adres değişkeninin indirek değerinin (yani i değişkeninin değeri) bulunduğu bellek alanına *pi ifadesi ile erişilerek 42 değeri atanır. Programda ayrıca pi = &j; atama deyimi ile j değişkeni için ayrılan bellek alanının adresi pi adres değişkenine atanır. Böylece pi'nin, j değişkenine işaret etmesi sağlanır. Bu durumda pi'nin indirek değeri 40 olur.

Örnek 5.1-4 adresop4.c programı.

```
#include <stdio.h>

int main(void)
{
  int       i  = 10, j  = 40, *pi;
  long      l  = 10,    *pl;
  double  x = 1.0, *px;

  pi   = &i;
  pl   = &l;
  px  = &x;

  printf( "int : %u, long : %u, double : %u\n", sizeof (int),          /* sizeof i */
                                                sizeof (long),         /* sizeof l */
                                                sizeof (double) );     /* sizeof x */
  printf( "int * : %u, long * : %u, double * : %u\n",
                                                sizeof (int*),         /* sizeof pi */
                                                sizeof (long*),        /* sizeof pl */
                                                sizeof (double*) );    /* sizeof px */
  printf( "&i : %u, &j : %u\n",  &i,  &j  );
  printf( "i : %d, j : %d, pi : %u, *pi : %d\n",  i,  j,  pi,  *pi  );

  i = 89;
  printf( "i : %d, j : %d, pi : %u, *pi : %d\n",  i,  j,  pi,  *pi  );

  *pi = 42;
  printf( "i : %d, j : %d, pi : %u, *pi : %d\n",  i,  j,  pi,  *pi  );

  pi = &j;
  printf( "i : %d, j : %d, pi : %u, *pi : %d\n",  i,  j,  pi,  *pi  );

  return 0;
}
```

Çıktı

```
int     : 2, long     : 4, double     : 8
int *: 2, long * : 2, double *   : 2
&i : 4362, &j : 4360
i  :  10, j  : 40,   pi :  4362, *pi  :  10
i  :  89, j  : 40,   pi :  4362, *pi  :  89
i  :  42, j  : 40,   pi :  4362, *pi  :  42
i  :  42, j  : 40,   pi :  4360, *pi  :  40
```

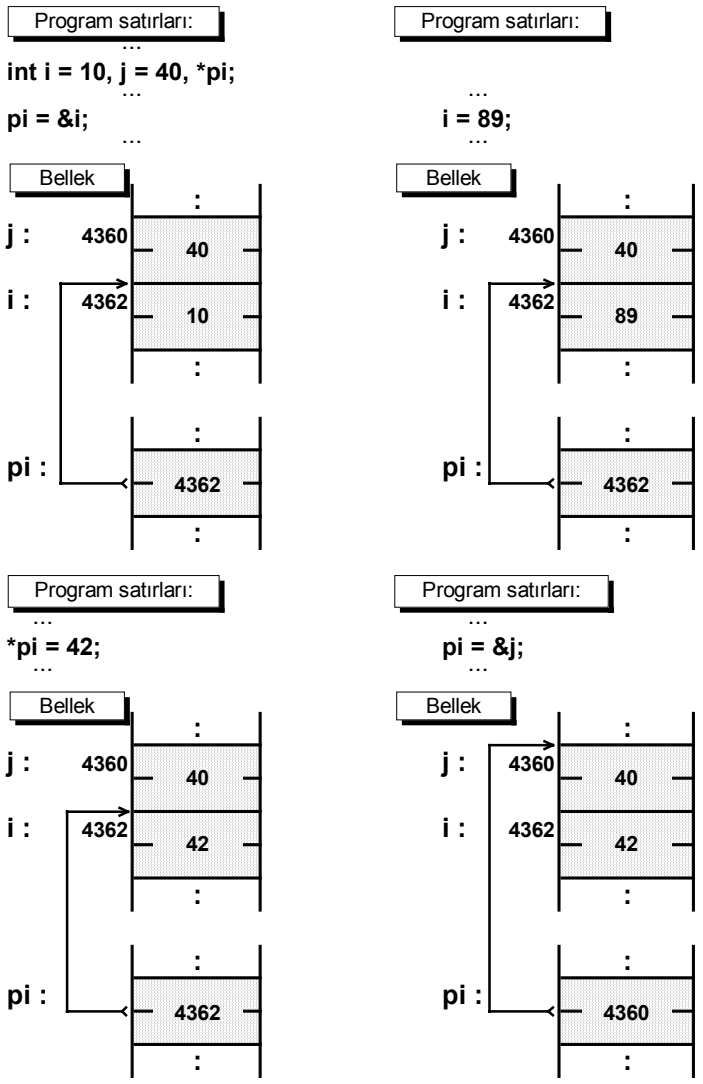
Şekil 5.1-6'da programda tanımlanan değişkenlerin yapılan işlemler sonrasındaki sembolik bellek görünümleri verilmiştir.

Örnek 5.1-4'de görüldüğü gibi aynı temel veri tipine sahip olduklarında, birkaç değişken aynı satırda virgülle ayrılmış olarak bildirilebilir:

```
short   i;
short   j;      } short i, j, *pi;
short   *pi;
long    l;
long    *pl;    } long l, *pl;
double  x;
double  *px;    } double x, *px;
```

Yukarıdaki bildirim deyimlerinde yer alan değişkenlerin tipleri şöyledir: i ve j değişkenlerinin tipi, short; pi adres değişkeninin tipi, short *; l değişkeninin tipi, long; pl adres değişkeninin tipi, long *; x değişkeninin tipi, double ve px adres değişkeninin tipi, double *. ■

Şekil 5.1-6 Örnek 5.1-4'de yer alan adres değişkeni işlemleri sonrasında sembolik bellek görünümleri.



5.2 Adres Değişkenleri ve Diziler

Örnek 5.2-1'de, 3 elemanlı Dizi dizisi tanımlanır. Bu program derlendiğinde bellekte, **int** verileri saklayabilecek sıralı 3 alan ayrılır. Ayrılan alanların herbiri 2 byte büyüklüğündedir. Çıktıda da görüldüğü gibi, her dizi elemanının bir bellek adresi vardır. Ayrılan bellek alanları sıralı olduğu için bu alanlara ait bellek adresleri de sıralı olacaktır. Dizi dizisinin elemanlarına, bu adresler kullanılarak erişilebilir.

Örnek 5.2-1 dizi1.c programı.

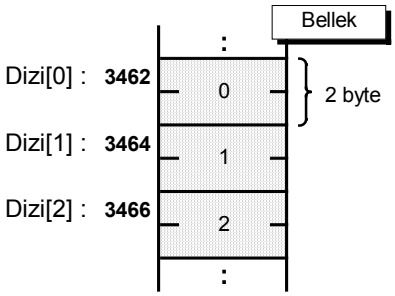
```
#include <stdio.h>

int main(void)
{
  int Dizi[ ] = { 0, 1, 2 };
  int i;

  for ( i = 0; i < 3 ; i++ )
       printf(" Dizi[%d] : %d, bellek adresi : %u \n", i, Dizi[ i ], &Dizi[ i ] );

  return 0;
}
```

Çıktı

```
Dizi[0]  :  0, bellek adresi  :  3462
Dizi[1]  :  1, bellek adresi  :  3464
Dizi[2]  :  2, bellek adresi  :  3466
```

Şekil 5.2-1'de görüldüğü gibi Dizi dizisinin eleman numarası (indeksi) birer birer artarken (0, 1, 2), elemanların & operatörü ile alınan bellek adresleri ikişer ikişer artar (3462, 3464, 3466).

Şekil 5.2-1 int[3] tipindeki Dizi dizisinin sembolik bellek görünümü.



Fakat Örnek 5.2-2'de olduğu gibi Dizi dizisi **long** temel veri tipi ile bildirildiğinde, sıralı adres değerleri arasındaki fark 4 olur.

Örnek 5.2-2 dizi2.c programı.

```
#include <stdio.h>

int main(void)
{
   long Dizi[ ] = { 0, 1, 2 };
   int i;

   for ( i = 0; i < 3 ; i++ )
        printf(" Dizi[%d] : %ld, bellek adresi : %u \n", i, Dizi[ i ], &Dizi[ i ] );
   return 0;
}
```

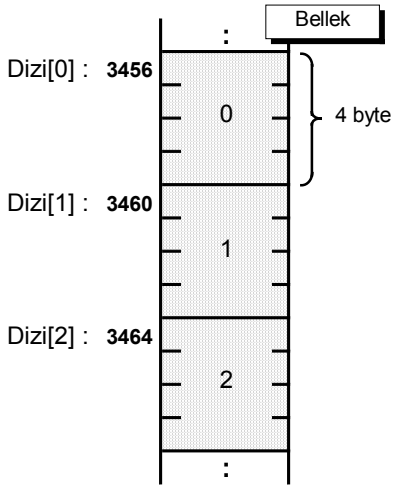
Çıktı

```
Dizi[0]  :  0, bellek adresi  :  3456
Dizi[1]  :  1, bellek adresi  :  3460
Dizi[2]  :  2, bellek adresi  :  3464
```

NOT: Adres değerleri donanıma göre değişebilir!

Bu durumda, sembolik bellek görünümü aşağıdaki gibi olacaktır.

Şekil 5.2-2 long[3] tipindeki Dizi dizisinin sembolik bellek görünümü.



Her iki örnekte de görüldüğü gibi dizinin indeksi birer birer artarken, karşılık gelen elemanlara ait bellek adresleri de dizinin bildirildiği temel veri tipi (**sizeof**(*tip*)) boyutlarında artar. Dizinin indeksi ve bellek adresleri arasındaki bu ilişki kullanılarak başlangıç elemanının adresi bilindiğinde dizinin diğer elemanlarına erişilebilir.

Buna göre **int** elemanlardan oluşan Dizi dizisinin birinci elemanı (Dizi[1]), başlangıç elemanın (Dizi[0]) bulunduğu 2 byte (**sizeof** Dizi[0] büyüklüğündeki) bellek alanını izleyen 2 byte'lık bellek alanında bulunacaktır. Başlangıç elemanının adresi (&Dizi[0] ifadesi ile alınan) 3462 olduğuna göre, birinci elemanın adresini elde etmek için başlangıç elemanının adresinden itibaren 1 dizi elemanı alanı kadar bellek alanı atlamak gerekir ve bu nedenle 3462 değerine 1***sizeof** (**int**) yani 2 byte eklenir. Böylece 3464 değeri elde edilir.

Aynı şekilde dizinin ikinci elemanı (Dizi[2]), başlangıç elemanını izleyen 2 bellek alanı sonrasında bulunduğundan bu elemanın bellek adresi 3462 değerine 2***sizeof** (**int**) eklenerek elde edilebilir (3462+4 = 3466). Yada dizinin ikinci elemanı, birinci elemanı izleyen bellek alanında bulunduğundan 3464 + 1***sizeof** (**int**) ifadesi de aynı sonucu (3466) verecektir. Böylece dizi elemanlarının adresleri, başlangıç adresine atlanacak alan sayısı ile **sizeof (***tip***)** çarpımı eklenerek elde edilebilir:

başlangıç adresi + i * sizeof(*tip*)

Bu ifadede i değişkeni atlanacak alan sayısını (dizinin indeks değişkeni) belirten tamsayı, *tip* ise dizinin bildirim deyiminde yer alan temel veri tipidir.

long Dizi[3] dizisi için uygulandığında, birinci elemanın adresi 3456 değerine 1***sizeof** (**long**)**;** ikinci elemanın adresi ise 3456 değerine 2***sizeof** (**long**) eklenerek bulunabilir:

başlangıç elemanının (Dizi[0]) *bellek adresi* : 3456
birinci elemanın (Dizi[1]) *bellek adresi* : 3456 + 1 * 4 = 3460
ikinci elemanın (Dizi[2]) *bellek adresi* : 3456 + 2 * 4 = 3464

Bilinen aritmetik kurallar uygulanarak adres değerleri yada adres değişkenleri ile işlemler yapılırken, yukarıdaki

başlangıç adresi + i * sizeof(*tip*)

ifadesi kullanılabilir. Fakat *C dilinde adres değerleri veya adres değişkenleri içeren ifadelerle yapılan işlemler **adres aritmetiği**ne göre yorumlanır*.
Adres aritmetiğine göre herhangi bir adres değerine yada adres değişkenine bir tamsayı değerin eklendiği ifadede, tamsayı değer direk olarak toplanmaz. Tamsayı değer, adres değerinin ait olduğu alanın yada bir başka deyişle adres değişkeninin indirek değerinin byte sayısı ile ölçeklendirildikten sonra işleme girer. Yani belli bir *ölçek faktörü* ile çarpılır. Bu ölçeklendirme C tarafından otomatik olarak yapılır.

Aşağıdaki programda adres aritmetiğini örneklemek için **char ***, **int ** ve **double ** tipindeki boş adres değerlerine (NULL) tamsayı 1 değeri eklenmiştir. Çıktıda da görüldüğü gibi **char**, **int** ve **double** tipi verilerin bellek adresleri için ölçek faktörleri sırasıyla 1, 2 ve 8'dir (bu değerler donanıma bağlı olarak değişebilir).

Örnek 5.2-3 arit1.c programı.

```
#include <stdio.h>

int main(void)
{
  printf(   "sizeof (char)       : %u\n"
            "sizeof (int)        : %u\n"
            "sizeof (double)     : %u\n",
            sizeof (char), sizeof (int), sizeof (double) );

  printf(   "(char *)0    + 1 : %u\n"
            "(int *)0     + 1 : %u\n"
            "(double *)0 + 1 : %u\n",
            (char *)0 + 1, (int *)0 + 1, (double *)0 + 1 );

  return 0;
}
```

Çıktı

```
sizeof(char)       : 1
sizeof(int)        : 2
sizeof(double)   : 8

(char *)0      + 1 : 1
(int *)0       + 1 : 2
(double *)0   + 1 : 8
```

Dizi dizisinin her bir elemanı, birbirini izleyen eşit büyüklükteki bellek alanlarına yerleştirilmiş aynı temel veri tipindeki sıralı değişkenlerdir. Dolayısıyla sıralı dizi elemanları için bellekte ayrılan eşit büyüklükteki alanların adresleri üzerinde işlemler yapılırken, dizideki herhangi bir elemanın bellek adresine tamsayı 1 değeri eklendiğinde bir sonraki elemanın bellek adresi elde edilir.

Çünkü adres aritmetiğine göre toplama iºleminden önce 1 değeri tek bir dizi elemanı için ayrılan bellek alanının byte sayısı (**sizeof** (*tip*) ifadesinin döndürdüğü değer, yani bildirimde yer alan temel veri tipinin byte olarak büyüklüğü) ile ölçeklendirilir. Bu nedenle bir dizinin başlangıç elemanının adresini kullanarak diğer elemanların bellek adreslerini veren,

başlangıç adresi + i * sizeof(*tip*)

ifadesi yerine,

başlangıç adresi + i

kullanılmalıdır. i'nin değeri (indeks değişkeni) başlangıç adresine eklenmeden önce, C dilinin yukarıda anlatılan özelliğinden (adres aritmetiği) dolayı bellek adresinin ait olduğu bellek alanının byte sayısı ile çarpılmış olacaktır. Dolayısıyla adres aritmetiği kullanılarak Örnek 5.2-1'de yer alan printf satırı aşağıdaki şekilde yazılabilir:

```
printf(" Dizi[%d] : %d, bellek adresi : %u \n", i, Dizi[ i ], &Dizi[ 0 ] + i );
```

Program derlenerek çalıştırıldığında, yine aynı çıktı elde edilir.

Dizi elemanlarına işaret eden adres değişkeni kullanılarak, dizi elemanlarının bellek adresleri üzerinde aritmetik işlemler kolaylıkla yapılabilir. Adres değişkenleri de bellekte saklanırlar (*lvalue*). Bu nedenle adres değişkenleri için ayrılan bellek alanlarına örneklerde görüldüğü gibi değer atanabilir. Ayrıca diğer değişkenlerde olduğu gibi adres değişkenlerine de artırma (++) ve eksiltme (--) operatörleri uygulanabilir.

Adres değişkenine tamsayı 1 eklemek, indirek değerinin byte büyüklüğündeki bir alan sonrasının bellek adresini verir. Eklenen tamsayı 2 olduğunda ise bulunulan bellek adresini izleyen ve her biri indirek değerin byte büyüklüğünde olan iki bellek alanı sonrasının adres değeri elde edilir. Örnek 5.2-4'de yer alan programın çıktısında görüldüğü gibi bildirilen adres değişkenlerine 1 değeri, adres değişkeninin indirek değerinin byte sayısı ile çarpılarak eklenir.

Örnek 5.2-4 arit2.c programı.

```
#include <stdio.h>
int main(void)
{
  short i,     *pi;
  long l,   *pl;
  double x,  *px;

  pi   = &i;
  pl   = &l;
  px   = &x;

  printf( "short i : %u byte, pi : %u, pi + 1 : %u\n",       sizeof( i ),    pi,   pi + 1    );

  printf( "long l : %u byte, pl :  %u,  pl + 1 : %u\n",         sizeof( l ),    pl,   pl + 1    );

  printf( "double x : %u byte, px :  %u,  px + 1 : %u\n",  sizeof( x ),   px,  px + 1   );

  printf("&i  : %d, &i + 1  : %d \n", &i,    &i + 1    );
  printf("&l  : %d, &l + 1  : %d \n", &l,    &l + 1    );
  printf("&x : %d, &x + 1 : %d \n", &x,     &x + 1   );

  return 0;
}
```

Çıktı

```
short i     : 2 byte,    pi   : 3614,      pi + 1   : 3616
long l      : 4 byte,    pl   : 3608,      pl + 1   : 3612
double x  : 8 byte,    px  : 3618,      px + 1  : 3626

&i:  3614,    &i + 1   : 3616
&l:  3608,    &l + 1   : 3612
&x  :  3618,    &x + 1  :  3626
```

Programda 1 sayısı i, l, ve x değişkenlerinin adres operatörü (&) kullanılarak alınan adres bellek adresleri ile toplanmadan önce otomatik olarak işlemde yer alan değişkenin veri tipinin byte sayısı ile çarpılmıştır.

Adres Değişkenleri ve Tek-Boyutlu Diziler

Örnek 5.2-5'de Dizi dizisinin elemanlarına adres değişkeni kullanılarak erişilir. Programda Dizi dizisinin başlangıç elemanının bellek adresi pDizi adres değişkenine atanır. Daha sonra **for** bloğunda yer alan printf çağrısında, pDizi'nin taşıdığı adres değerine her çevrimde dizinin bir sonraki elemanının numarası eklenir. İşlem adres değerleri üzerinde yapıldığı için, i'nin değeri toplama işlemine girmeden önce indirek değerin byte sayısı ile çarpılarak ölçeklendirilir.

 **Örnek 5.2-5** dizi3.c programı.

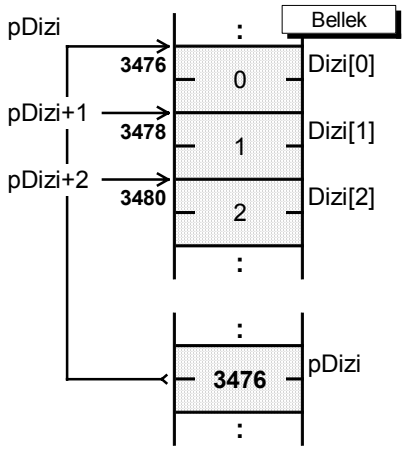
```
#include <stdio.h>

int main(void)
{
  int Dizi[ ] = { 0, 1, 2 }, i;
  int *pDizi;

  pDizi = &Dizi[0];

  for ( i = 0; i < 3 ; i++ )
       printf(" *(pDizi + %d) : %d, bellek adresi pDizi + %d : %u \n",
                                          i, *( pDizi + i ), i,     pDizi + i  );

  return 0;
}
```

 **Çıktı**

```
          *(pDizi + 0) :  0, bellek adresi  pDizi + 0  :  3476
          *(pDizi + 1) :  1, bellek adresi  pDizi + 1  :  3478
          *(pDizi + 2) :  2, bellek adresi  pDizi + 2  :  3480
```

Programda toplama işleminden sonra atama yapılmadığı için, pDizi'nin taşıdığı adres değeri program boyunca değişmez ve dizinin başlangıç elemanının adresidir.

Şekil 5.2-3 int[3] tipindeki Dizi dizisi ve int * tipindeki pDizi adres değişkeninin sembolik bellek görünümü.



for döngüsü ve printf deyimi aşağıdaki şekilde oluşturulabilir:

```
for ( i = 0; i++ < 3 ; pDizi++ )    /* yada for ( i = 0; i < 3 ; i++, pDizi++ ) */
    printf("*pDizi : %d, bellek adresi pDizi : %u \n", *pDizi, pDizi );
```

Yine aynı çıktı elde edilir. Fakat pDizi++ ifadesi (pDizi = pDizi + 1 ifadesi ile aynıdır) ile artırım sonunda elde edilen adres değeri pDizi adres değişkenine atandığı için, program sonunda pDizi'nin değeri dizinin başlangıç elemanının adresi olmayacaktır.

Özet olarak;

- Dizi bildirim deyimi içeren bir program derlendiğinde, bellekte dizinin eleman sayısı kadar ve her biri bildirimde yer alan veri tipi ile belirtilen byte büyüklüğünde (**sizeof**(*tip*)) sıralı bellek alanları ayrılır. Bellek alanları sıralı olduğu için, adresleri de sıralı olacaktır.
- Dizinin indeksi (eleman numarası) birer birer artarken, her indekse karşılık gelen elemanın bellek adresi de dizi bildirim deyiminde yer alan veri tipinin byte sayısı kadar artar.
- Adres değeri yada adres değişkeni ile bir tamsayı değerin toplandığı ifadede, tamsayı değer adresin ait olduğu alanın byte sayısı ile çarpılarak işleme girer.

- C dilinin yukarıda anlatılan özelliği (*adres aritmetiği*) bütün adres değerleri için geçerli olduğundan, dizi elemanlarının bellek adresleri üzerinde yapılan işlemler de bu şekilde yorumlanır. Dolayısıyla bir dizinin başlangıç adresine herhangi bir elemanın numarasını eklemek, o dizi elemanının bellek adresini verecektir. Herhangi bir adres değerine tamsayı 1 değeri eklendiğinde elde edilen sonuç, bu adres değeri ile dizinin temel veri tipinin byte sayısının toplamına eşittir. Adres değişkenleri ile yapılan işlemlerde, işaret edilen verinin boyutları daima yapılan işlemin sonucunu etkiler.

Aşağıda adres değişkenleri ile yapılabilecek işlemler sıralanmıştır :

- Adres değişkeninin değeri, ++ ve -- operatörleri ile artırılabilir yada eksiltilebilir. Örnek olarak, eğer pc adres değişkeni bellekteki herhangi bir **char** verinin bulunduğu bellek alanına işaret ediyorsa, pc++ ifadesi ile izleyen **char** alana işaret edecektir. Fakat bu işlemlerin tek bir veri elemanına işaret eden adres değişkeni (*basit adres değişkeni*) üzerinde yapılması anlamsızdır. Bu işlemlerin eğer adres değişkeni bir dizi elemanına işaret ediyorsa (*dizi adres değişkeni*) sıralı elemanlara erişim amacıyla yapılması anlamlıdır.

  Dizi elemanlarına erişim adres değişkeni aracılığı ile yapılırken, C dili tarafından dizi sınırlarının dışına taşma kontrol edilmez (aynı durum, indeks kullanılarak erişim sağlanırken de geçerlidir). Taşma durumunda istenmediği halde bitişik alanlardaki veriler bozulabileceği için, adres işlemleri dikkatli yapılmalıdır.

- Adres değişkenine dizi sınırları içinde kalarak bir tamsayı değer eklenebilir yada çıkarılabilir. Fakat bu işlem **float** yada **double** değerler ile yapılamaz.

  Örnek :

```
p += 2;
```

  Bu işlemler adres aritmetiğine göre yorumlanır. Adres değişkeni p herhangi bir dizinin elemanlarına işaret ediyorsa, p + i ifadesi (i bir tamsayı değişkendir) p değişkeninin işaret ettiği veriyi izleyen i'inci verinin adresini verir. Bu işlem, indirek değerin veri tipi (**sizeof** (*p)) C tarafından otomatik olarak dikkate alındığından, bütün adres değerleri ve adres değişkenlerine uygulanabilir.

- Adres değişkenine 0 (sıfır) değeri atanabilir. C derleyicisi, 0 değerini bellek adresi olarak kullanmaz. Dolayısıyla 0 değeri adres değişkenine atanarak özel bir durum bildirilebilir. *Boş adres değişkeni* (*null-pointer*) değeri olarak 0 değeri, stdio.h, stdlib.h, stddef.h, locale.h, string.h ve time.h başlık dosyalarında NULL makrosu ile tanımlanmıştır.

- Adres değişkenine adres değerleri yada adres değerleri döndüren ifadeler de atanabilir. Adres değişkeni, programlarda herhangi bir tamsayı değişken gibi kullanılamayacağı gibi herhangi bir tamsayı değer de adres değişkenine atanamaz.

- Adres değişkeni aynı tipte adres değeri bekleyen bir fonksiyona argüman olarak aktarılabilir.

Aynı tipteki adres değişkenleri arasında aşağıdaki işlemler yapılabilir. Fakat bu işlemler ancak adres değişkenleri aynı dizinin elemanlarına işaret ediyor ise anlamlıdır:

- Bir adres değişkeninin değeri aynı tipte bir başka adres değişkenine atanabilir.

  Örnek :

  ```
  p1 = p2;   yada   p2 = p1;
  ```

- İki adres değişkeni arasında toplama, çarpma ve bölme işlemleri yapılamaz.

- İki adres değişkeni birbirinden çıkarılabilir.

  Örnek :

  ```
  p2 - p1
  ```

  İşlemin sonucu stddef.h'de tanımlanmış bulunan ptrdiff_t tipindedir ve adres değişkenleri arasındaki farkın, indirek değerin byte sayısına bölünmesiyle elde edilir. Bu ölçeklendirme, yine C derleyicisi tarafından otomatik olarak gerçekleştirilir. Dolayısıyla aynı dizinin farklı elemanlarına işaret eden iki adres değişkeni birbirinden çıkarıldığında, aradaki eleman sayısı elde edilir.

  Aşağıdaki program parçasında ptrdiff_t tipinde bildirilen n değişkeni kullanılarak, iki adres değişkeni arasındaki fark taşınabilir şekilde saklanır. Ayrıca fark değeri **long** tipine çevrilerek taşınabilir şekilde ekrana listelenir:

  ```
  ...
  ptrdiff_t n;
  ...
  n = ( p + 1 ) - p;
  ...
  printf( "n : %ld\n", (long)n );
  ...
  ```

- Adres değişkeni ile bir başka adres değeri, adres değeri döndüren bir ifade yada diğer bir adres değişkeni karşılaştırılabilir.

  Örnek :

  ```
  p1 < p2
  ```

  Bu ifade, p1 adres değişkeni dizinin p2'den önceki herhangi bir elemanına işaret ediyorsa *doğru*dur. Aşağıdaki testlerin tümü, aynı dizinin elemanlarına ait adres değerlerine yada aynı dizinin elemanlarına işaret eden adres değişkenlerine uygulanabilir:

  ```
  <=, >=, <, >, !=, ==
  ```

- Adres değişkeni ve NULL değeri (0 değeri) arasında, == ve != karşılaştırması yapılabilir:

  Örnek :

  p == NULL (yada p == 0) ve p != NULL (yada p != 0)

  Aynı dizinin elemanlarına işaret eden iki adres değişkeninin farkı, bir tamsayı değer ile yada 0 ile karşılaştırılabilir:

  Örnek :

  p1 - p2 < 0, p1 - p2 < 0 yada p1 - p2 = = 0

Örnek 6.2-5'de atama deyimi,

```
pDizi = &Dizi[0];
```

ile Dizi dizisinin başlangıç elemanının adresi pDizi adres değişkenine atanır ve pDizi üzerinde adres aritmetiği uygulanarak dizi elemanlarına erişim sağlanır. Programda adres değişkeni pDizi yerine dizi ismi Dizi kullanıldığında da yine aynı çıktı elde edilir. Çünkü *C dilinde dizi ismi, dizinin başlangıç elemanının bellek adresini veren sabittir* (*constant pointer*).

Adres değişkeni olmadığı için, dizi ismine herhangi bir adres değeri atanamaz, ++ yada -- operatörleri uygulanamaz. Aşağıda, 3 elemanlı m dizisi ve tamsayı değişken n bildirilir:

```
...
int m[3], n;
...
```

Dizi ismine m = &n deyimi ile n değişkeninin bellek adresi atanamaz. Ayrıca &m, m++ yada m-- ifadeleri geçersizdir. Ancak dizi ismine * operatörü uygulanarak verdiği sabit adres değeri kullanılabilir (*m, *(m+1) gibi ...).

Örnek 5.2-6 dizi4.c programı.

```
#include <stdio.h>
#define DiziBoy(s)   sizeof (s) / sizeof (s[0])
int main(void)
{
   int Dizi[ ]  = { 0, 1, 2 }, i;

   int *pDizi = Dizi;
   puts( "dizi elemanlarinin adresleri" );
   for ( i = 0; i < DiziBoy(Dizi); i++ )
         printf( "&Dizi[%d] : %u, &Dizi[0] + %d : %u, Dizi + %d : %u\n",
                  i, &Dizi[i],  i, &Dizi[0] + i,  i, Dizi + i );

   puts( "dizi elemanlarinin degerleri" );
   for ( i = 0; i < DiziBoy(Dizi); i++ )
         printf( "Dizi[%d] : %d, *(Dizi + %d) : %d\n",    i, Dizi[i],  i, *(Dizi + i) );

   puts( "dizi elemanlarinin adresleri" );
   for ( i = 0; i < DiziBoy(Dizi); i++ )
         printf( "pDizi + %d : %u\n",i,  pDizi + i  );

   puts( "dizi elemanlarinin degerleri" );
   for ( i = 0; i < DiziBoy(Dizi); i++ )
         printf( "pDizi[%d] : %d, *(pDizi + %d) : %d\n",i,  pDizi[i],  i,  *(pDizi + i)  );

   puts( "pDizi = Dizi + 1 deyimi sonrasi degerler");
   pDizi = Dizi + 1;
   printf("Dizi : %u, pDizi : %u, *pDizi : %d\n", Dizi,  pDizi,  *pDizi  );
   printf("pDizi - Dizi : %ld\n", (long)(pDizi - Dizi) );
   puts( "pDizi++ deyimi sonrasi degerler" );
   pDizi++;
   printf("Dizi : %u, Dizi + 2 : %u, pDizi : %u, *pDizi : %d\n",  Dizi,     Dizi + 2,
                                                                  pDizi,*pDizi    );
   printf( "%s\n",    (pDizi > Dizi)          ? "pDizi  > Dizi"          :  "pDizi < Dizi"     );
   printf( "%s\n",    (pDizi == Dizi + 2) ? "pDizi == Dizi + 2" :  "pDizi != Dizi + 2"  );
   return 0;
}
```

Çıktı

```
dizi elemanlarinin adresleri
        &Dizi[0] :  3910, &Dizi[0] + 0 :  3910, Dizi + 0 :  3910
        &Dizi[1] :  3912, &Dizi[0] + 1 :  3912, Dizi + 1 :  3912
        &Dizi[2] :  3914, &Dizi[0] + 2 :  3914, Dizi + 2 :  3914
```

... Örnek 5.2-6 Çıktı devam

```
dizi elemanlarinin degerleri
        Dizi[0]   :  0,*(Dizi + 0)   :  0
        Dizi[1]   :  1,*(Dizi + 1)   :  1
        Dizi[2]   :  2,*(Dizi + 2)   :  2
dizi elemanlarinin adresleri
        pDizi  + 0 :  3910
        pDizi  + 1 :  3912
        pDizi  + 2 :  3914
dizi elemanlarinin degerleri
        pDizi[0] :  0,*(pDizi  + 0)   :  0
        pDizi[1] :  1,*(pDizi  + 1)   :  1
        pDizi[2] :  2,*(pDizi  + 2)   :  2
pDizi = Dizi + 1 deyimi sonrasi degerler
        Dizi :  3910, pDizi  :  3912, *pDizi :  1
        pDizi  -  Dizi :  1
pDizi++ deyimi sonrasi degerler
        Dizi :  3910, Dizi + 2 :  3914, pDizi  :  3914, *pDizi :  2
        pDizi >  Dizi
        pDizi == Dizi + 2
```

&Dizi[0] ve dizi ismi Dizi, her ikisi de aynı adres değerini verir (3476). Dolayısıyla bu adresteki değeri döndüren Dizi[0] ve *Dizi ifadeleri de aynı sonucu verecektir. Fakat dizi ismine bir tamsayının eklendiği ifadeler adres aritmetiğine göre yorumlandığından, dizi ismi kullanılarak;

Dizi + i

ifadesi ile diğer dizi elemanlarının bellek adreslerine,

*(Dizi+i)

ifadesi ile de bu adreslerdeki verilere erişilebilir.

Bildirim deyimi ile 3 elemanlı Dizi dizisi (Dizi[0], Dizi[1], Dizi[2]) bildirilir. Bu dizi bildirim deyiminde yer alan Dizi, **int**[3] tipindedir. **int**[3] ifadesinden indeks ([3] yada [eleman_sayısı]) çıkarıldığında elde edilen ise bildirimde kullanılan temel veri tipidir.

Örnek 6.2-6'da adres değişkenine diğer değişkenlerde olduğu gibi bildirim sırasında ilk değer atama yapılır. int *pDizi = Dizi deyimi ile bildirilen pDizi adres değişkenine aynı deyimde ilk değer ataması yapılarak Dizi dizisinin başlangıcına işaret etmesi sağlanır. Bildirim deyiminde Dizi yerine Örnek 6.2-5'de olduğu gibi &Dizi[0] kullanılabilir.

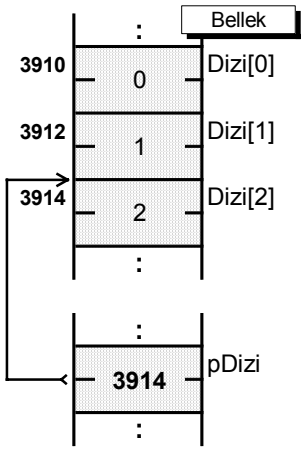
Örnek programda ilk olarak dizi ismi ile oluşturulan ifadeler kullanılarak dizi elemanlarının bellek adresleri listelenir. Dizi ismi, dizinin başlangıç adresini veren sabit

olduğu için Dizi + i ifadeleri adres aritmetiğine göre yorumlanır. Aynı şekilde dizinin başlangıç elemanına bir tamsayının eklendiği &Dizi[0] + i ifadeleri de adres aritmetiğine göre yorumlanır.
Dizi elemanlarının değerleri ise yine dizi ismi ile oluşturulan bu ifadelere indirek değer operatörü uygulanarak listelenir.

Örnek programda daha sonra adres değişkeni kullanılarak dizi elemanlarının adresleri ve değerleri listelenir. Adres değişkeni de yine ifadelerde dizi ismi gibi bulunur fakat adres değişkeni bir *lvalue*, dizi ismi ise değiştirilemeyen bir *lvalue*'dır (*non-modifiable lvalue*).

Daha sonra pDizi = Dizi + 1 deyimi ile dizinin birinci elemanının bellek adresi adres değişkenine atanır. pDizi adres değişkeni, dizinin birinci elemanına işaret ettiği için *pDizi ifadesi 1 değerini döndürür. Dizi'nin değeri olan başlangıç adresinin (3910), pDizi'nin değeri 3912'den çıkarıldığı işlem adres aritmetiğine göre yorumlanır ve iki adres değeri arasındaki fark otomatik olarak dizi bildiriminde yer alan temel veri tipinin (**int**) boyutlarına bölünür. Bu nedenle pDizi - Dizi işlemi 1 değerini döndürür. ptrdiff_t tipindeki bu değer ekrana listelenirken taşınabilir olması için **long** tipine çevrilir.

Şekil 5.2-4 int[3] tipindeki Dizi dizisi ve int * tipindeki pDizi adres değişkeninin program sonundaki sembolik bellek görünümü.



NOT: Adres değerleri Örnek 5.2-5 ile aynı değildir.

---

pDizi++ ifadesi ile adres değişkeninin taşıdığı adres değerine tamsayı 1 eklenir. Fakat yine adres aritmetiğine göre tamsayı 1 işleme girmeden önce pDizi'nin indirek değerinin byte sayısı ile çarpıldığından elde edilen sonuç 3914 (ikinci elemanın adresi) olacaktır. Dolayısıyla *pDizi ifadesi 2 değerini döndürür.

Dizi ismi Dizi'nin verdiği adres değeri (3910), dizinin ikinci elemanına işaret eden pDizi'nin değerinden (3914) küçüktür. Böylece pDizi > Dizi testi doğru olacağı için printf fonksiyonu ekrana "pDizi > Dizi" dizgisini yazar.
Programda son olarak pDizi = = Dizi + 2 testi yapılır. pDizi adres değişkeni dizinin ikinci elemanının adresini taşıdığından, test doğru sonucunu verir ve ekrana "pDizi == Dizi + 2" dizgisi yazılır.

Bu programda dizi ismi ve adres değişkeni ile oluşturulan ifadeler kullanılarak yapılan işlemler örneklendi. Şekil 5.2-4'de programın sonunda Dizi dizisinin ve pDizi adres değişkeninin sembolik bellek görünümleri yer alır.

Sonuç olarak Dizi dizisinin elemanlarının bellek adreslerini veren &Dizi[i] ifadesi, Dizi + i ifadesi ile; bu elemanların değerlerini veren Dizi[i] ifadesi ise *(Dizi + i) ifadesi ile aynıdır. Görüldüğü gibi dizi elemanlarının adreslerine ve bu adreslerde bulunan değerlere dizi ismi kullanılarak kolaylıkla erişilebilir.

C dilinde adres değişkenleri indeks operatörü ([ ]) ile kullanılabilir. Böylece oluşturulan pDizi[i] ifadesi ile *(pDizi + i) ifadesi aynıdır (tamsayı indeks i, dizinin herhangi bir elemanının numarasıdır) ve her ikisi de pDizi + i adresindeki değeri verir. İndeks operatörü ve indirek değer operatörü, hem dizi ismine hem de adres değişkenine uygulanabilir. Bu durumda dizi ismi Dizi ve adres değişkeni pDizi ile aşağıdaki ifadeler oluşturulabilir:

Dizi[i] yada *(Dizi + i) ifadeleri Dizi + i adresindeki değeri,

ve

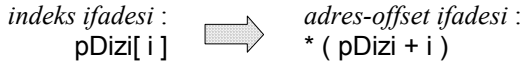
pDizi[i] yada *(pDizi + i) ifadeleri de, pDizi + i adresindeki değeri

verir.

Derleyici tarafından *dizi ismi ve indeks operatörü ile oluşturulan bir ifade, dizinin başlangıç elemanına işaret eden adres değeri ve offset ifadesine dönüştürülür* (**sizeof** ve & operatörünün operand'ı olan ifadeler ve bir **char** tipi dizinin dizgi sabiti ile ilk değer atamasının yapıldığı uygulama hariç).
Dolayısıyla aşağıda adres değeri ve indeks operatörü ile oluşturulan *indeks ifadesi* (*array notation* yada *subscript notation*) ile adres değeri ve offset ile oluşturulan *adres-offset ifadesi* (*pointer notation*) birbiri ile aynıdır :

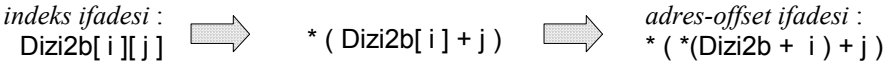
indeks ifadesi : Dizi[i] ⇨ *adres-offset ifadesi* : * (Dizi + i)

Bu ifadeler dizi ismi yerine adres değişkeni kullanılarak oluşturulabilir.



Ok işareti bu ifadelerin aynı olduğunu belirtir. Fakat bir adres sabiti olan dizi ismi ile oluşturulan ifadeler, adres değişkeni kullanılarak oluşturulan ifadelerle aynı değildir. Derleyici bu iki grup ifade için farklı kodlar oluşturur (Ancak izleyen bölümlerde de anlatılacağı gibi, argüman olarak dizi alan bir fonksiyonun parametre listesinde bildirilen dizi ismi, adres değişkenidir. Çünkü aktarılan adres değeridir).

Bu ifadeler, çok-boyutlu diziler için de oluşturulabilir:



Dizi indeksleme işlemi yukarıda da görüldüğü gibi aslında adres aritmetiğidir ve değişme özelliği vardır. Dolayısıyla aşağıdaki ifadeler birbirine eştir (değişme özelliğinden dolayı):

i [Dizi] ifadesinin gerekli olduğu hiç bir uygulama yoktur. Burada yalnızca dizi indeksleme işleminin değişme özelliği olduğunu göstermek amacıyla oluşturulmuştur.

Örnek programda aşağıdaki makro kullanılarak dizinin eleman sayısı bulunur:

```
#define DiziBoy(s)   sizeof (s) / sizeof (s[0])
```

Bir dizi ismine **sizeof** operatörü uygulandığında, dizinin toplam byte büyüklüğünü verir. Dizinin tek bir elemanının byte sayısı veren **sizeof** s[0] ifadesinde 0 yerine dizi sınırları içinde kalan bir başka indeks değeri kullanılabilir. Fakat her dizinin 0. elemanı mutlaka bulunacağı için, indeks değeri olarak 0 kullanımı ifadeyi daha anlaşılır hale getirir.
Dizi ismine negatif indeks uygulanabilir. Böylece dizinin başlangıç elemanından önceki aynı büyüklükteki bellek alanlarına erişilebilir. Bu uygulama derleme sırasında hata oluşturmayabilir, fakat dizi sınırları dışına çıkmak C dilinin kurallarına uymaz. Aynı şekilde diziye işaret eden adres değişkenine de negatif indeks uygulanabilir. Bu durumda işaret edilen dizi elemanından bir önceki elemana erişilir.

Örnek 5.2-7'de **int** tipinde i değişkeni, **int *** tipinde pDizi ve piDizi adres değişkenleri, **int**[5] tipinde iDizi dizisi bildirilir. Programda ayrıca **double *** tipinde pdDizi adres değişkeni bildirilir ve **double**[2] tipinde dDizi dizisi tanımlanır.

Örnek 5.2-7 dizi5.c programı.

```
#include <stdio.h>

#define  DiziBoy(s)  sizeof(s)/sizeof(s[0])

int main(void)
{
   int i = 0, *pDizi;
   int     *piDizi;
   double  *pdDizi;

   int     iDizi[ 5 ];
   double  dDizi [ ] = { 4.5, 5.4 };

   piDizi = &i;
   pdDizi = dDizi;

   for ( i = 0, pDizi = iDizi; i < DiziBoy(iDizi); i++, pDizi++ )
         *pDizi = i;
   pDizi = iDizi;

   ++pdDizi;
   ++*pDizi;

   *pdDizi  +=  (double)*pDizi;
   piDizi   =   pDizi;
   *piDizi  *=  2;

   if ( *piDizi == *(iDizi + 2) ) *piDizi = (int)*(pdDizi - 1);

   for ( i = 0, pDizi = iDizi; i < 5; i++, pDizi++)
         printf("iDizi[%d]: %d\n", i, *pDizi );

   pDizi = iDizi;

   printf("dDizi[0]: %.1f, dDizi[1]: %.1f\n", dDizi[0], dDizi[1] );

   return 0;
}
```

Çıktı

```
iDizi[0]: 4
iDizi[1]: 1
iDizi[2]: 2
iDizi[3]: 3
iDizi[4]: 4
dDizi[0]: 4.5, dDizi[1]: 7.4
```

Atama deyimi piDizi = &i ile adres değişkeninin i değişkenine; pdDizi = dDizi ile de pdDizi adres değişkeninin dDizi dizisinin başlangıç elemanına işaret etmesi sağlanır. **for** döngüsünde, adres değişkeni pDizi kullanılarak iDizi dizisinin elemanlarına karşılık gelen eleman numaraları değer olarak atanır. Döngü çıkışında taşıdığı değer değişen

pDizi adres değişkenine, pDizi = iDizi ile tekrar iDizi dizisinin başlangıç elemanının adresi atanır.

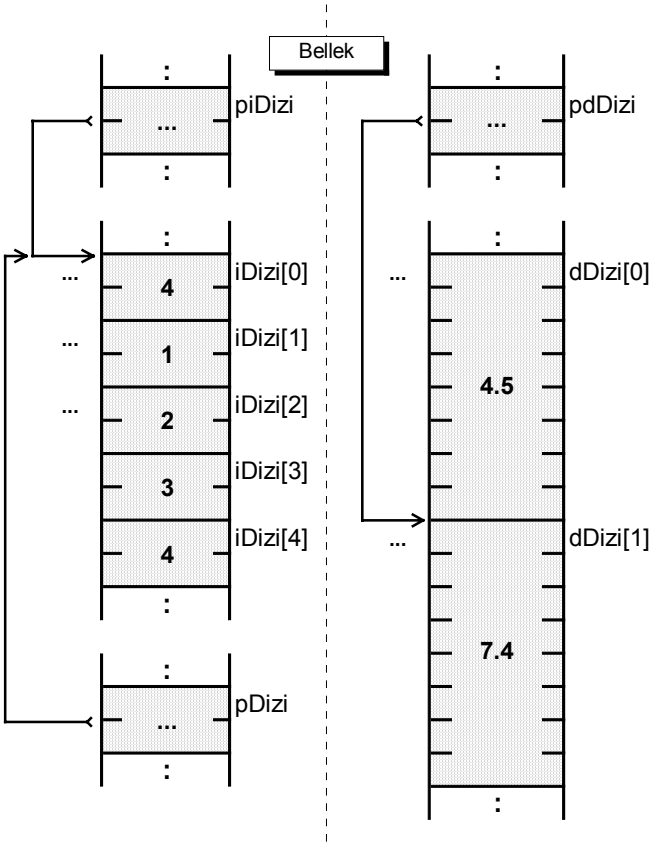
++pdDizi deyimi ile adres değişkeninin dizinin ilk elemanına işaret etmesi sağlanır. ++*pDizi deyimi kullanılarak iDizi'nin başlangıç elemanının değerine tamsayı 1 eklenir.

*pdDizi += (double)*pDizi deyiminde, iDizi[0]'ın değeri (yani *pDizi) tip değiştirme uygulanarak dDizi dizisinin ilk elemanının değeri ile toplanır ve elde edilen sonuç yine dDizi dizisinin ilk elemanına atanır: *pdDizi = *pdDizi + (double)*pDizi;

piDizi = pDizi deyimi, her iki adres değişkeninin de aynı veriye işaret etmesini sağlar (yani iDizi dizisinin başlangıç elemanı). piDizi adres değişkeni kullanılarak, iDizi dizisinin başlangıç elemanının değeri tamsayı 2 ile çarpılır. Elde edilen sonuç yine başlangıç elemanına atandıktan sonra dizinin ikinci elemanının değeri ile karşılaştırılır. Bu test *doğru* sonucu vereceğinden koşul sağlanır ve *piDizi = (int)*(pdDizi - 1) deyimi çalıştırılır. Böylece tip değiştirme yapılarak dDizi dizisinin başlangıç elemanının değeri iDizi dizisinin başlangıç elemanına atanır. Programın sonunda, pDizi adres değişkeni kullanılarak iDizi dizisinin elemanlarına erişilir ve **for** döngüsü ile değerleri ekrana listelenir. **for** döngüsü sonrasında pDizi = iDizi atama deyimi ile adres değişkeninin yine dizi başlangıcına işaret etmesi sağlanır.

Şekil 5.2-5'de örnek programda tanımlanan adres değişkenlerinin ve dizilerin program sonundaki bellek düzenleri sembolik olarak gösterilir.

Şekil 5.2-5 Program sonunda piDizi ve pDizi adres değişkenleri iDizi dizisine; pdDizi adres değişkeni ise dDizi dizisinin birinci elemanına işaret eder.



Yukarıdaki örnek programda görüldüğü gibi, ++ ve -- operatörleri adres değişkenlerine ön-ek (*prefix operator*) ve son-ek (*postfix operator*) operatörler olarak uygulanabilir.

Operatör öncelik kurallarına göre eğer bir ifadede birden fazla operatör bulunuyor ise, operatörler öncelik sırasına (operatör öncelik tablosunda belirtilen) göre uygulanır. Operatörler aynı önceliğe sahip ise, grup özelliklerine (*associativity*) göre işlenirler. Buna göre operatör öncelik tablosunda (*operator precedence table*) aynı sırada bulunan ++ ve * operatörlerinin grup özellikleri *sağdan sola* doğru olduğu için, ifadelerdeki gruplandırma da sağdan sola doğru yapılır.
Dolayısıyla *++p ifadesinde adres değişkenine sağdan sola doğru ilk olarak ++ operatörü bağlanır. p adres değişkeninin taşıdığı değer artırıldıktan sonra * operatörü ile indirek değeri alınır.

Aşağıdaki programda arttırma ve indirek değer operatörlerinin adres değişkenine bir arada uygulanması örneklenmiştir.

Örnek 5.2-8 adr.c programı.

```
#include <stdio.h>

int main(void)
{
   int dizi[ ]   = { 99, 88, 77, 66, 55 };
   int *p     = dizi;
   int  i         = 0;

   printf( "p : %u, *p : %d, i : %d\n",    p, *p, i ); puts( "i = *p++" );
   i = *p++;
   printf( "p : %u, *p : %d, i : %d\n\n", p, *p, i );

   printf( "p : %u, *p : %d, i : %d\n",    p, *p, i ); puts( "i = *++p" );
   i = *++p;
   printf( "p : %u, *p : %d, i : %d\n\n",    p, *p, i );

   printf( "p : %u, *p : %d, i : %d\n",       p, *p, i ); puts( "i = ++*p" );
   i = ++*p;
   printf( "p : %u, *p : %d, i : %d\n\n",    p, *p, i );

   printf( "p : %u, *p : %d, i : %d\n",       p, *p, i ); puts( "i = (*p)++" );
   i = (*p)++;
   printf( "p : %u, *p : %d, i : %d\n\n",    p, *p, i );

   for ( i = 0; i < sizeof dizi / sizeof dizi[0]; i++ )
   {
         dizi [ i ] = i;
         printf( "%d, ", dizi [ i ] );
   }
   putchar( '\n' );

   p = &dizi[ 2 ];
   *( p + 1 ) = 9;
   p[-2] = 7;
   dizi[ *p++ ] = *( dizi + dizi[ 1 ] ) + 2;
   *(p + 1 ) = 5;
   dizi[ *p - 8 ] = 8;

   for ( i = 0; i < sizeof dizi / sizeof dizi[0]; i++ )
         printf( "%d, ", dizi [ i ] );

   return 0;
}
```

... Örnek 5.2-8 devam

Çıktı

```
p : 3660, *p : 99, i : 0
i = *p++
p : 3662, *p : 88, i : 99

p : 3662, *p : 88, i : 99
i = *++p
p : 3664, *p : 77, i : 77

p : 3664, *p : 77, i : 77
i = ++*p
p : 3664, *p : 78, i : 78

p : 3664, *p : 78, i : 78
i = (*p)++
p : 3664, *p : 79, i : 78

0, 1, 2, 3, 4,
7, 8, 3, 9, 5,
```

Örnek 5.2-8'de arttırma (++) ve indirek değer operatörlerinin (*) p adres değişkenine bir arada uygulanması ile oluşturulan ifadelerde (Şekil 5.2-6'da 1. satır), operatörlerin işleme girme sırası ok yönünde yani sağdan sola doğru gerçekleşir. Çünkü bu operatörler aynı önceliğe sahiptir ve bir arada bulundukları ifadelerde grup özellikleri sağdan sola doğrudur. 2. satırda ise aynı işlemler operatör etkisinin açık olarak belirtilmesi için parantezler kullanılarak gerçekleştirilir. 3. satırda ise ifadelerin daha anlaşılır olması için arttırma işlemi ve indirek değerin i değişkenine atanması iki ayrı deyimde yapılır. 4. satırda ise ilk üç ifadede yer alan işlem, arttırma operatörü yerine toplama operatörü kullanılarak gerçekleştirilir.

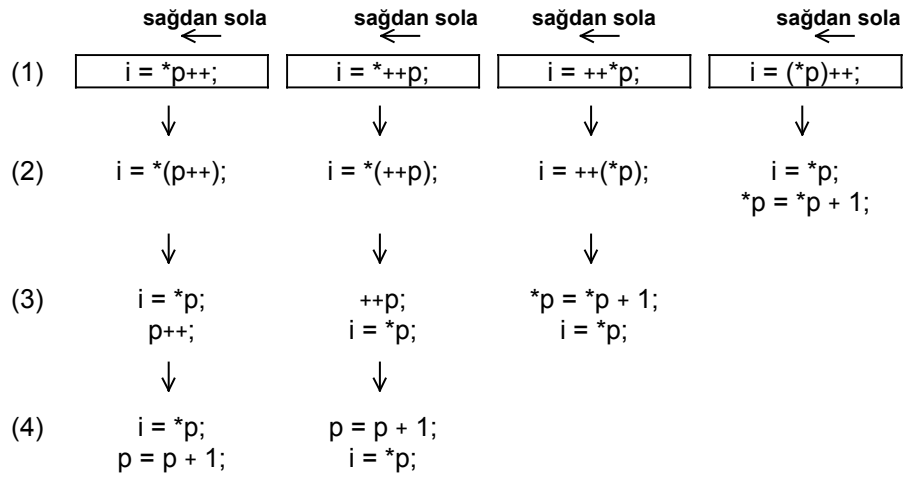
Programda son olarak, **for** döngüsünde dizi elemanlarına 0, 1, 2, 3 ve 4 değerleri atanır. p = &dizi[2] atama deyiminde dizinin 2. elemanının bellek adresi p adres değişkenine atanır. Dolayısıyla p+1 ifadesi dizinin 3. elemanına işaret eder. *(p + 1) = 9 deyimi ile bu elemana 9 değeri atanır.

p[-2] = 7 deyiminde p adres değişkenin işaret ettiği dizi elemanının (2. eleman) 2 eleman öncesinde bulunan dizi elemanına (dizinin 0. elemanı) 7 değeri atanır.

dizi[1]'in değeri 1 olduğu için *(dizi + dizi[1]) ifadesi *(dizi + 1) olur ve dizinin 1.elemanının değerini verir (yani tamsayı 1). Sonuç olarak oluşan dizi[*p++] = 1 + 2 ifadesi, p adres değişkeni dizinin 2. elemanına işaret ettiği için dizi[2] = 3 atama deyimidir. Böylece değeri artırılan p adres değişkeni dizinin 3. elemanına işaret eder ve *(p + 1) = 5 deyimi ile dizinin 4. elemanına 5 değeri atanır.

Dizinin 3. elemanının değeri 9 olduğu için dizi[*p - 8] = 8 atama deyimi ile dizinin 1. elemanına 8 değeri atanır. Dizi elemanlarının değerleri çıktıda da görüldüğü gibi, sırasıyla 7, 8, 3, 9, 5 olur. ■

Şekil 5.2-6 ++ ve * operatörleri ile oluşturulan ifadelerin işlenmesi.



5.3 Adres Değişkenleri ve Dizgiler

Bir *dizgi sabiti* yada kısaca *dizgi* (*string constant*, *character string* yada *string literal*) çift tırnaklar arasına yerleştirilen sıfır yada daha fazla karakterden oluşur. Örneğin "12345abc" bir dizgi sabitidir. Çift tırnaklar yalnızca dizgiyi sınırlamak için kullanılır ve dizginin bir parçası değildir. İki çift tırnaktan oluşan ve hiçbir karakter içermeyen dizgi boş dizgi olarak adlandırılır (""). Karakter sabitlerinde kullanılan escape dizinleri (\n, \t, \007 gibi) dizgiler içinde de kullanılabilir. Dizgi içinde yer alan \" karakterleri tek bir çift tırnağı ifade eder.

Derleme sırasında peşpeşe verilen dizgi sabitleri birbirine eklenir. Örneğin ekrana abcd karakterlerini yazan bir printf çağrısı iki şekilde de olabilir:

```
printf( "ab"
        "cd\n" );
```

yada

```
printf( "abcd\n" );
```

Dizgiler bellekte ASCII kodu 0 olan boş karakter (*null character*: \0 yada karakter sabiti olarak '\0') ile sonlanmış karakter dizileri olarak saklanırlar. Dolayısıyla bir karakter dizisine ilk değer atama parantezlerle sınırlı ve virgülle ayrılmış karakter sabitleri listesi yerine dizgi sabiti kullanılarak yapılabilir:

```
char s[ ] = { 'a', 'b', 'c', '\0' };
```

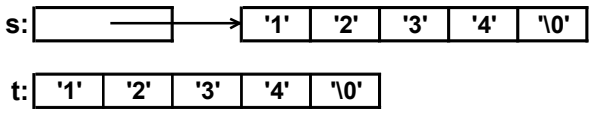
yerine

```
char s[ ] = "abc";
```

kullanılabilir. İkinci bildirimde dizgi sabitindeki karakterler s dizisinin elemanlarına atanır ve dizgi sonuna C tarafından otomatik olarak boş karakter yerleştirilir. Derleyici dizi uzunluğuna '\0' karakterini de dahil eder. Dizgiler üzerinde işlemler yapılırken bu karakter kullanılarak dizgi sonu belirlenebilir. Bildirimde dizi boyutları verildiğinde, dizgi sabitindeki karakter sayısı dizinin eleman sayısını geçmemelidir ('\0' hariç).

Herhangi bir dizgi sabiti karakter dizisi tipindedir ve değeri dizginin bellekte bulunduğu alanın başlangıç adresidir (karakter dizisinin başlangıç elemanının bellek adresi). Bir programda dizgi sabitinde bulunan karakterlere erişim, başlangıç elemanına işaret eden adres değeri kullanılarak gerçekleştirilir. Dizgi sabitleri örnek programlarda çoğunlukla printf fonksiyonu argümanı olarak kullanıldı. Dolayısıyla bir dizgiyi argüman olarak alan printf fonksiyonu karakter dizisi başlangıcına işaret eden adres değeri alır. Bu nedenle indeks yada adres-offset ifadeleri kullanılarak gerçekleştirilen dizi işlemleri, dizgilerle de yapılabilir.

Bölüm 5'de yer alan sizeof.c programında (sayfa 5-5) **char** tipi verilere işaret eden s adres değişkeni ve t karakter dizisi bildirilir ve "1234" dizgisi ile ilk değer atamaları yapılır. İlk değerleri aynı olsa dahi bu iki bildirim birbirinden oldukça farklıdır.
Çıktıda da görüldüğü gibi **sizeof** operatörü s için 2 byte, t için ise 5 byte değerini döndürür. Çünkü s bir dizgi sabitine işaret eden adres değişkenidir. İlk değer olarak bildirimde bir karakter dizisinin adresi atanmıştır. s adres değişkeninin değeri bir başka dizgiye işaret edecek şekilde değiştirilebilir. t ise bellekte 5 karakterlik alana sahip bir karakter dizisidir. İçerdiği karakterler değiştirilebilir fakat t daima aynı bellek alanına işaret eder (dizi ismi başlangıc adresini veren sabittir):



s adres değişkenine değer atama ayrı bir deyimde de yapılabilir:

```
char *s;
s = "1234";
```

Dizginin başlangıç adresini aldığı için printf("1234") çağrısı yerine s adres değişkeninin yada t dizisinin argüman olarak aktarıldığı aşağıdaki printf çağrıları da ekrana yine "1234" dizgisini yazar:

```
printf( s );   yada   printf( t );
```

Aşağıdaki örnek programda a ve b karakter dizileri tanımlanır. İlk değer atama iki şekilde de yapılabilir. Programda ayrıca dizgi sabiti ile oluşturulan ve C programlarında pek rastlanmayan çeşitli ifadeler yer alır: dizgi sabitine indirek değer operatörü uygulanarak dizginin ilk karakteri alınır; dizgi sabitine indeks uygulanarak bir karaktere erişilir ve adres aritmetiğinin değişme özelliğinden faydalanılarak dizgideki bir karaktere erişilir. Bu üç işlem sonucu ekrana 023 karakterleri listelenir. Çıktıda da görüldüğü gibi, her iki printf çağrısında da aynı dizgi sabiti ("a") yer aldığı halde bellek adresleri farklıdır. Dizgi sabiti "a", karakter sabiti 'a' ile karıştırılmamalıdır. 'a' bir karakter sabitidir ve ifadelerde a karakterinin ASCII kodunu veren tamsayıdır. "a" ise a karakteri ve '\0'den oluşan karakter dizisidir.

Örnek 5.3-1 karak.c programı.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char  a[ ] = "1234";
   char  b[ ] = { '1', '2', '3', '4', '\0' };

   printf( "sizeof a  : %u, sizeof b : %u\n", sizeof a,   sizeof b  );
   printf( "strlen(a) : %u, strlen(b): %u\n", strlen(a),  strlen(b) );
   printf( "%p\n", "a" );
   printf( "%p\n", "a" );

   putchar( *"0" );
   putchar( "012345"[2]   );
   putchar( *(3+"012345") );

   return 0;
}
```

Çıktı

```
sizeof a   :  5, sizeof b   :  5
strlen(a)  :  4, strlen(b)  :  4
0083
0089
023
```

Örnek 5.3-2'de karakter dizisi s bildirilir ve elemanlarına ilk değer olarak "01234ab" dizgi sabitinde bulunan karakterler atanır. s dizisinin son elemanı dizgi sonunu

işaretleyen '\0' karakteridir. Daha sonra **char** tipi verilere işaret eden ps adres değişkeni bildirilir. Adres değişkenine ilk değer olarak karakter dizisinin başlangıç elemanının bellek adresi atanır. Böylece adres değişkeni kullanılarak tüm dizi elemanlarına erişilebilir.

Örnek 5.3-2 dizgi.c programı.

```
#include <stdio.h>
#include <string.h>

#define  NL       '\n'

int main(void)
{
  char   s[ ]    = "01234ab";
  char   *ps     = s;
  int i, BOY = strlen( s );

  putchar( *ps++   );          /*       putchar( *ps );  ps++;                */
  putchar( *(ps++) );            /*      yukaridaki deyim ile ayni              */

  putchar( (*ps)++ );            /*      putchar( *ps );  *ps = *ps + 1;        */
  putchar( *++ps   );          /*       ++ps;  putchar( *ps );                */
  putchar( *(++ps) );            /*      yukaridaki deyim ile ayni              */
  putchar( ++*ps   );          /*       *ps = *ps + 1;   putchar( *ps );      */
  putchar( ++(*ps) );            /*      yukaridaki deyim ile ayni              */
  putchar( NL );                 /*    0123456                          */

  ps = s;
  while ( ps < s + BOY )
          putchar( *ps++ );
  putchar( NL );          /* 01336ab */

  ps = s + BOY - 1;
  while ( ps >= s )
          putchar( *ps-- );
  putchar( NL );          /* ba63310 */

  for ( ps = s, i = 0; i < BOY; i++ )
          putchar( ps[i] );
  putchar( NL );          /* 01336ab */

  for ( ps = s + BOY - 1, i = 0; i < BOY; i++ )
          putchar( ps[-i] );
  putchar( NL );          /* ba63310 */

  ps = s;
  while ( *ps ) /*     while ( *ps != '\0' )*/
          putchar( *ps++ );
  putchar( NL );          /* 01336ab */

  i  = 0;
```

... Örnek 5.3-2 devam

```
   ps = s;
   while ( ps[i] )   /*    while ( ps[i] != '\0' )   */
            putchar( ps[i++] );
   putchar( NL );          /* 01336ab */

   return 0;
}
```

Çıktı

```
0123456
01336ab
ba63310
01336ab
ba63310
01336ab
01336ab
```

Programda yer alan iki **while** döngüsünün test ifadesinde dizgi sonu, boş karakter kontrolu yapılarak belirlenir. Test ifadeleri koşul sağlandığında (test doğru olduğunda) 0 harici bir değer, koşul sağlanmadığında (test yanlış olduğunda) ise 0 değerini döndürür. Bu nedenle **while** döngülerinin test ifadelerinde *ps != '\0' ve ps[i] != '\0' yerine, sadece *ps ve ps[i] ifadeleri kullanılmıştır. Çünkü dizgi sonuna gelindiğinde *ps ve ps[i] ifadelerinin değeri boş karakter (0) olacaktır; yani != '\0' koşulunun *yanlış* olması durumunda döndüreceği değer.

Programda yer alan ilk putchar deyim grubu ekrana 0123456 karakterlerini yazar. Her putchar deyiminin açık formu, aynı satırda açıklama satırı olarak verilmiştir. Görüldüğü gibi, ilk iki deyimde ps adres değişkeninin işaret ettiği karakter ekrana yazılır ve ps adres değişkeninin değeri artırılarak bir sonraki karaktere işaret etmesi sağlanır. İzleyen deyimde ise işaret edilen karakter (2) ekrana yazılır ve bu karakterin ASCII kod değeri bir artırılır. Böylece karakter dizisinin ikinci elemanı 3 karakteri olur. Bundan sonra ps adres değişkeninin değeri yine artırılır ve işaret edilen karakter (3) ekrana listelenir. İşlem izleyen deyimde bir kez daha tekrarlanarak ekrana 4 karakteri yazılır. Son iki deyimde, 4 karakterinin ASCII kod değeri iki kez artırılır ve elde edilen değerler aynı alana yazıldıktan sonra ekrana listelenir. Sonuç olarak 4 karakterinin bulunduğu bellek alanında bu iki deyim sonrası 6 karakteri bulunur. Dizginin son iki karakterine erişim gerçekleşmez. Programda adres değişkeni ile oluşturulan deyimler kullanılarak, s karakter dizisinin elemanlarının değerleri ekrana listelenir (*ps++, *ps--, ps[i], ps[-i], ps[i++]).

Karakter dizisinin elemanlarının değerleri dizi ismi ile oluşturulan s[i] yada *(s+i) ifadeleri kullanılarakta listelenebilirdi:

```
...
i = 0;
while ( s[ i ] )
        putchar( s[i++] );
...
```

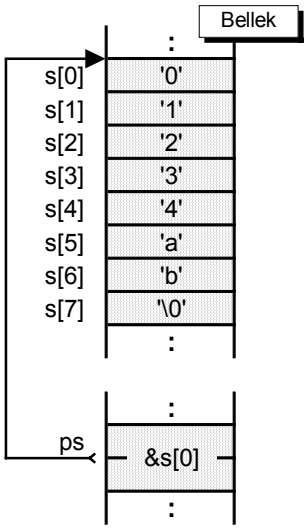
yada

```
...
i = 0;
while ( *(s + i) )
{
putchar( *(s + i) );
i++;
}
...
```

Fakat burada amaç dizgiye işaret eden adres değişkeni kullanımını örneklemektir.

s karakter dizisinin elemanları **char** tipinde veriler, yani 1 byte büyüklüğünde sıralı bellek alanlarında saklanan ASCII kodlardır ve sembolik bellek görünümleri aşağıdaki gibi olur:

Şekil 5.3-1 char *ps = s; deyimi sonrası s karakter dizisinin ve ps adres değişkeninin sembolik bellek görünümü.



Standart C Kütüphanesi dizgi giriş/çıkış işlemleri için, dizgileri bir bütün olarak yada karakter-karakter işleyen çeşitli fonksiyonlar içerir (gets, puts, getchar, putchar, scanf ve printf gibi). Ayrıca C kütüphanesinde dizgiler ve karakterler üzerinde işlemler yapan

çeşitli fonksiyon ve makrolar bulunur (strcat, strchr, strcmp, strcpy, strlen, isalpha, isupper, islower, isdigit, toupper, tolower gibi). İzleyen bölümde, elemanları dizgilere işaret eden **char *** tipi adres değişkenleri olan adres dizileri anlatılacaktır. ■

5.4 Adres Dizileri

C dilinde elemanları adres değerleri saklayabilecek diziler bildirilebilir. Böyle bir dizi *adres dizisi* olarak adlandırılır ve bu dizinin sıralı bellek adreslerinde bulunan her elemanı için, donanıma bağlı olarak tek bir adres değeri taşıyabilecek büyüklükte bellek alanı ayrılır. Adres dizisinin elemanları birbirini izleyen bellek adreslerinde bulunan ve taşıdıkları değerlere adres dizisi ismi ve indeks operatörü ile oluşturulan ifadelerle erişilen adres değişkenleridir. Fakat bu adres değişkenlerinin taşıdığı adres değerleri sıralı olmayabilir.

Aşağıdaki bildirim deyimi ile **char** tipi verilere işaret eden 4 adres değişkeninden oluşan AdrDizi adres dizisi bildirilir:

```
char *AdrDizi [ 4 ];
```

Bu bildirim deyimi, indeks operatörü öncelik listesinde indirek değer operatörü öncesinde bulunduğu için *adres dizisi değişkeni* (AdrDizi [...]) *ile erişilen her dizi elemanının tipi*nin **char *** olduğunu ifade eder. Bir başka deyişle AdrDizi, ***char **** *tipindeki 4 veriden oluşan dizi*; yani **char*[4]** tipindedir. **sizeof (char***[4]**)** yada **sizeof (**AdrDizi**)** ifadeleri dizinin toplam bellek alanının byte olarak büyüklüğünü verecektir.

Örnek 5.4-1 adrdizgi.c programı.

```
#include   <stdio.h>
#include   <stdlib.h>
#include   <string.h>

#define NL     '\n'

int main(void)
{
   int i, j;
   char *AdrDizgi [ ] = { "a", "bcd987", "123", NULL };

   printf( "sizeof AdrDizgi       : %u,  sizeof *AdrDizgi          : %u\n"
           "sizeof **AdrDizgi       : %u\n"
           "sizeof AdrDizgi[0]   : %u,  sizeof AdrDizgi[0][0] : %u\n",
           sizeof AdrDizgi,       sizeof *AdrDizgi,   sizeof **AdrDizgi,
           sizeof AdrDizgi[0], sizeof AdrDizgi[0][0] );
```

...Örnek 5.4-1 devam

```
printf( "AdrDizgi          : %p,  *AdrDizgi       : %p\n"
        "**AdrDizgi     : %u\n"
        "AdrDizgi[0]    : %p,  AdrDizgi[0][0] : %u\n",
        AdrDizgi,  *AdrDizgi,  **AdrDizgi, AdrDizgi[0],  AdrDizgi[0][0] );
for ( i = 0; i < sizeof AdrDizgi/ sizeof AdrDizgi[0] - 1; i++ )
    printf( "AdrDizgi [ %d ] : %p, *AdrDizgi [ %d ] : %c, "
            "AdrDizgi[ %d ] dizgisi : %s\n", i, AdrDizgi[ i ], i,  *AdrDizgi[ i ],
                                                i, AdrDizgi[ i ] );
 /*
  * Adres dizisini sonlandiran NULL adres degeri test edilerek
  * dizgilerin listelenmesi:
  *          while ( AdrDizgi[ i ] != NULL )
  */
i = 0;
while ( AdrDizgi[ i ] )
      puts( AdrDizgi[ i++ ] );
 /*
  * Adres dizisini sonlandiran NULL adres degeri ve her dizgi icinde
  * dizgileri sonlandiran '\0' (null character)  test edilerek
  * dizgilerde bulunan karakterlerin listelenmesi:
  *          while ( AdrDizgi[ i ] != NULL )
  *          ...

  *                while ( AdrDizgi[ i ][ j ] != '\0' )
  *                ...
  */
i = 0;
while ( AdrDizgi[ i ] )
{
      j = 0;
      while ( AdrDizgi[ i ][ j ] )
              putchar( AdrDizgi[ i ][ j++ ] );
    /*
     *   while ( ... )
     *   {
     *      putchar( *(*(AdrDizgi +  i ) + j) );
     *      j++;
     *   }
     */
      putchar( NL );
      i++;
}
i = 0;
while ( AdrDizgi[ i ] )
{
    printf( "AdrDizgi[ %d ] dizgisi %u karakter. \n", i, strlen( *(AdrDizgi +  i ) ) );
    /*      i, strlen( AdrDizgi[ i ] ) );    */
    i++;
}
```

...Örnek 5.4-1 devam

```
   puts(    "            Adres Karakter   ASCII kod"     );
   printf(  "            ------- ----------  ------------"  );
   for ( i = 0; i < sizeof AdrDizgi/ sizeof AdrDizgi[0] - 1; i++ )
   {
        putchar( NL );
        j = 0;
        do
        {
             printf( "AdrDizgi [ %d ][ %d ] : %p   %c       %d\n",
                    i, j, &AdrDizgi [ i ][ j ], AdrDizgi [ i ][ j ], AdrDizgi [ i ][ j ] );
        }  while ( AdrDizgi [ i ][ j++ ] );
   }
   return 0;
}
```

Çıktı

```
sizeof AdrDizgi       : 8, sizeof *AdrDizgi            : 2
sizeof **AdrDizgi     : 1
sizeof AdrDizgi[0] : 2, sizeof AdrDizgi [ 0 ][ 0 ]  : 1


AdrDizgi        : 0F3C, *AdrDizgi            : 0042
**AdrDizgi      : 97
AdrDizgi[ 0 ]   : 0042,  AdrDizgi [ 0 ][ 0 ]   : 97

AdrDizgi [ 0 ]  : 0042,  *AdrDizgi [ 0 ]    : a,  AdrDizgi [ 0 ] dizgisi : a
AdrDizgi [ 1 ]  : 0044,  *AdrDizgi [ 1 ]    : b,  AdrDizgi [ 1 ] dizgisi : bcd987
AdrDizgi [ 2 ]  : 004B,  *AdrDizgi [ 2 ]    : 1,  AdrDizgi [ 2 ] dizgisi : 123

a
bcd987
123

a
bcd987
123

AdrDizgi [ 0 ] dizgisi 1 karakter.
AdrDizgi [ 1 ] dizgisi 6 karakter.
AdrDizgi [ 2 ] dizgisi 3 karakter.
                        Adres   Karakter   ASCII kod
                        -------  ----------  ------------
AdrDizgi [ 0 ][ 0 ]  :  0042       a           97
AdrDizgi [ 0 ][ 1 ]  :  0043                   0

AdrDizgi [ 1 ][ 0 ]  :  0044       b           98
AdrDizgi [ 1 ][ 1 ]  :  0045       c           99
```

...Örnek 5.4-1 Çıktı devam

```
AdrDizgi [ 1 ][ 2 ]  :  0046     d        100
AdrDizgi [ 1 ][ 3 ]  :  0047     9        57
AdrDizgi [ 1 ][ 4 ]  :  0048     8        56
AdrDizgi [ 1 ][ 5 ]  :  0049     7        55
AdrDizgi [ 1 ][ 6 ]  :  004A              0

AdrDizgi [ 2 ][ 0 ]  :  004B     1        49
AdrDizgi [ 2 ][ 1 ]  :  004C     2        50
AdrDizgi [ 2 ][ 2 ]  :  004D     3        51
AdrDizgi [ 2 ][ 3 ]  :  004E              0
```

C dilinde *dizgilerden oluşan dizi*, iki şekilde oluşturulabilir: iki-boyutlu karakter dizisi (izleyen bölümlerde anlatılacaktır) yada adres dizisi olarak:

2-b karakter dizisi:

```
char  s [ ] [ 7 ] = { "a", "bcd987", "123" };
```

adres dizisi:

```
char  *m [ ] = { "a", "bcd987", "123" };
```

yada

```
char  *m [ 3 ];
m [ 0 ] = "a";
m [ 1 ] = "bcd987";
m [ 2 ] = "123";
```

s dizisi bildirimi ile, herbiri 7 karakter uzunluğunda (dizgi sonunu işaretleyen '\0' dahil) 3 karakter dizisi için bellek alanı ayrılır. Dizgiler ayrılan bellek alanının tamamını kullanmayabilir ("a" ve "123" dizgilerinde olduğu gibi).

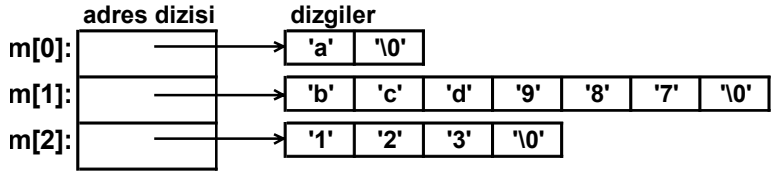
Bildirimde verilmeyen s dizisinin 1.boyut değeri, ilk değer listesinde bulunan dizgi sayısına göre derleyici tarafından saptanabilir. 2.boyut değeri ise en uzun dizgiye göre belirlenmiştir. Böylece dizgiler bellekte 2-b dizinin satırları olarak saklanırlar:

s dizisi

s[0]:	**'a'**	**'\0'**	**...**	**...**	**...**	**...**	**...**
s[1]:	**'b'**	**'c'**	**'d'**	**'9'**	**'8'**	**'7'**	**'\0'**
s[2]:	**'1'**	**'2'**	**'3'**	**'\0'**	**...**	**...**	**...**

Burada s[i], dizinin i. satırı olan dizginin başlangıç adresini veren sabittir. Adres dizisi bildirimi ile bellekte 3 adet **char *** tipinde adres değişkeni için ve ayrıca 1+1, 6+1 ve 3+1 karakter uzunluğunda (+1 karakter '\0' içindir) üç ayrı karakter dizisi için bellek alanı ayrılır.

Adres dizisinin elemanı olan her adres değişkeni bir dizgiye işaret eder:

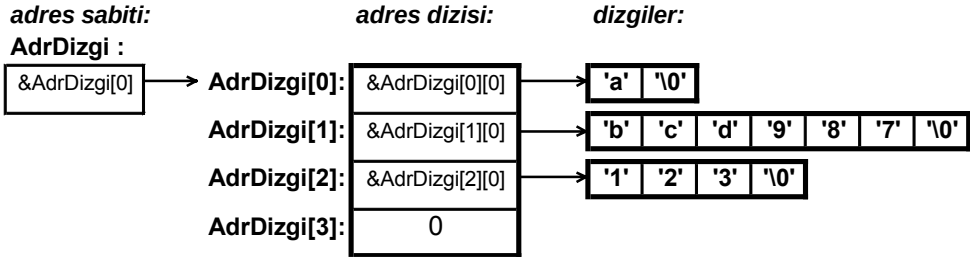


İkinci uygulama ile farklı uzunluklarda dizgilerden oluşan dizi (*ragged array*) oluşturulabilir. Komut satırı argümanları bu şekilde bir dizidir. Bu uygulama daha verimlidir. Fakat izleyen bölümlerde anlatılacak olan dinamik bellek alanı kullanımı söz konusu olduğunda, gerekli bellek alanı ayırma işleminin ayrı ayrı her dizgi için yapılması gerekir. Her iki uygulamada, bir dizgideki herhangi bir karaktere erişen ifade indeks operatörleri kullanılarak aynı şekilde oluşturulur: s[1][2] yada m[1][2] gibi. Bu iki ifade birbirinin aynı değildir ve söz konusu karaktere erişimi farklı şekilde gerçekleştirirler.

Örnek 5.4-1'de **char** tipi verilere işaret eden 4 adet adres değişkeninden oluşan AdrDizgi adres dizisi tanımlanır. İlk değer listesinde yer alan son değer boş adres değeridir (NULL). Dolayısıyla AdrDizgi[3] adres değişkeni hiçbir dizgiye işaret etmez. Bu değer adres dizisi sonunu belirlemek için kullanılır.

Dizi ismi AdrDizgi, adres dizisinin başlangıç adresini veren adres sabitidir. Bir ifadede AdrDizgi, *çift adres sabiti*ne dönüşür. Yani AdrDizgi'nin değeri, adres dizisinin ilk elemanının (ilk dizginin başlangıcına işaret eden) bellek adresidir.

Adres dizisinin ilk elemanı ilk dizginin başlangıcına işaret ettiği için, AdrDizgi'ye iki kez * operatörü uygulandığında, a karakterine erişilir:



Örnek programda, AdrDizgi adres dizisine bildirim sırasında ilk değer atama yapılarak farklı uzunlukta 3 dizginin başlangıç elemanlarının bellek adresleri atanır. Adres dizisinin elemanlarının adreslerini &AdrDizgi[i] yada AdrDizgi+i ifadeleri verir.

Çıktıda görüldüğü gibi, AdrDizgi[...][...] ifadesi ile dizgilerin elemanı olan 1-byte büyüklüğünde **char** verilere erişilir. AdrDizgi[0][0]'ın değeri, ilk dizginin başlangıç elemanı olan a karakterinin ASCII kodu olan 97 değeridir. Çünkü AdrDizgi[0] ilk dizginin başlangıç adresini veren adres değişkenidir (bu donanımda 2-byte) ve indirek değer operatörü uygulandığında işaret ettiği bellek alanında bulunan **char** değeri verir.

Programda dizgileri ve karakterleri listelemek için çeşitli döngüler yer alır. **sizeof(**AdrDizgi**)** ifadesi adres dizisinin toplam bellek alanı büyüklüğü olan 8-byte değerini; **sizeof(**AdrDizgi[0]**)** ise dizinin elemanı olan bir adres değişkeninin bellek alanı büyüklüğü olan 2-byte değerini verir. Dolayısıyla

sizeof AdrDizgi / **sizeof** AdrDizgi[0]

ifadesi de dizinin eleman sayısını verir ve döngülerin bir kısmında NULL testi yapılmadan adres dizisindeki eleman sayısının saptanmasını sağlar.

Aşağıdaki örnek programda, **double** tipinde verilere işaret eden 5 adres değişkeninden (AdrDizi[0], AdrDizi[1], AdrDizi[2], AdrDizi[3] ve AdrDizi[4]) oluşan adres dizisi bildirilir. **for** döngüsünde adres dizisinin elemanlarına **double** dizi ddizi'nin elemanlarının sıralı bellek adresleri atanır. Böylece ddizi'nin elemanlarına programda görüldüğü gibi adres dizisi kullanılarak erişilebilir. ■

Örnek 5.4-2 adrd.c programı.

```
#include   <stdio.h>
#define    ELSayi(dizi)    sizeof dizi / sizeof dizi[0]
int main(void)
{
   int i;
   double ddizi[ ] = { 3.5, 7.9, 1.2, 5.3, 4.4 }, *AdrDizi[ ELSayi(ddizi) ];
   for ( i = 0; i < ELSayi(ddizi); i++ )
         *( AdrDizi + i ) = ddizi + i; /* AdrDizi[ i ] = &ddizi[ i ]; */
   for ( i = 0; i < ELSayi(ddizi); i++ )
         printf( "ddizi[%d] : %1.1f, *AdrDizi[%d] : %1.1f\n",  i, ddizi[ i ], i, *AdrDizi[ i ]  );
   /*
   *    printf( "ddizi[%d] : %1.1f, *AdrDizi[%d] : %1.1f\n",  i, *(ddizi+i), i, **(AdrDizi+i) );
   */
   return 0;
}
```

...Örnek 5.4-2 devam

Çıktı

```
ddizi[ 0 ]  :  3.5, *AdrDizi[ 0 ] :  3.5
ddizi[ 1 ]  :  7.9, *AdrDizi[ 1 ] :  7.9
ddizi[ 2 ]  :  1.2, *AdrDizi[ 2 ] :  1.2
ddizi[ 3 ]  :  5.3, *AdrDizi[ 3 ] :  5.3
ddizi[ 4 ]  :  4.4, *AdrDizi[ 4 ] :  4.4
```

5.5 Çift Adres Değişkeni

Bir adres değişkeni bir başka adres değişkenine işaret edecek şekilde bildirilebilir. Aşağıdaki örnekte yer alan double **ppd = &pd deyimi, **double *** tipindeki pd adres değişkenine işaret eden ppd adres değişkenini bildirilir:

Örnek 5.5-1 ciftadr.c programı.

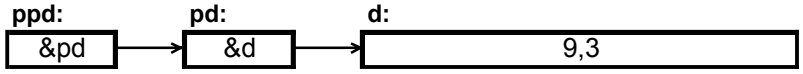
```
#include   <stdio.h>
int main(void)
{
   double d = 9.3, *pd = &d, **ppd = &pd;

   /*
   *     double d = 9.3;
   *     double *pd;
   *     double **ppd;
   *     ...
   *     pd = &d;
   *     ppd = &pd;
   */
   printf( "d : %1.1f, *pd : %1.1f, **ppd : %1.1f\n", d, *pd, **ppd );
   printf( "&d : %p, pd : %p, *ppd : %p\n", &d, pd, *ppd );
   printf( "&pd : %p, ppd : %p\n", &pd, ppd );
   return 0;
}
```

Çıktı

```
          d : 9.3      *pd : 9.3    **ppd : 9.3
         &d : 1090      pd : 1090    *ppd : 1090
                       &pd : 1098     ppd : 1098
```

---

ppd adres değişkeni, indirek değere (9.3) erişim iki seviyede gerçekleştiği için *çift adres değişkeni* (*double pointer*) olarak adlandırılır. Çıktıda da görüldüğü gibi ppd ifadesi, pd adres değişkeninin bellek adresini; *ppd ise d değişkeninin bellek adresini verir. ppd adres değişkeni, indirek değer operatörü iki kez uygulandığında d değişkeninin değerini verir (**ppd).



İndirek erişimin iki seviyeden fazla olduğu uygulamalar çok azdır. Çift adres değişkenleri izleyen bölümlerde de anlatılacağı gibi genel olarak adres dizileri fonksiyonlara argüman olarak aktarılırken, komut satırı argümanları kullanılırken, bir fonksiyonun çağıran bloktaki adres değişkeninin değerini değiştirmesi istendiğinde yada dinamik bellek alanı ayıran fonksiyonlar kullanılırken gereklidir. ■

5.6 Adres Değişkenlerine Tip Dönüştürme Uygulanması

C dilinde herhangi bir ifadeye, *tip dönüştürme operatörü* uygulanarak tip dönüştürme yapılabilir. Böylece derleyicinin bu ifadeyi farklı bir veri tipinde görmesi sağlanabilir:

(*tip-ismi*) *ifade*

Tip dönüştürme işlemi, adres değişkenlerine yada adres değeri veren ifadelere de uygulanabilir. Bu yolla adres değişkeninin yada adres değeri veren ifadenin değeri bildirildiğinden farklı bir tipte verinin adresi olarak işlem görebilir. Aşağıdaki örnek programda, 1-byte **char** verilerden oluşan cDizi dizisinin başlangıcına işaret eden **char *** tipindeki Adrc adres değişkenine tip dönüştürme uygulanarak bu donanımda 2-byte olan **int** verilere işaret etmesi sağlanır. Tip dönüştürme işleminin uygulandığı adres değişkeni 2-byte verilere işaret ettiği için taşıdığı adres değerine tamsayı 1'in eklendiği ifade adres aritmetiğine göre yorumlanır ve cDizi'nin 2. elemanının bellek adresi elde edilir (&cDizi[2]). Elde edilen adres değeri yine bir **int** veriye işaret eder. Bu adres değeri tekrar tip dönüştürme uygulanarak **char *** tipine dönüştürülür ve aynı tipteki Adrc adres değişkenine atanır. Adrc adres değişkenine indirek değer operatörü uygulandığında cDizi[2]'nin değeri olan 0xC3 elde edilir. Bu işlemde,

```
Adrc = (char *)( (int *)Adrc + 1);
```

ifadesi yerine,

```
( (int *)Adrc )++;
```

ifadesi kullanılamaz. Çünkü tip dönüştürme operatörü bir bellek alanında bulunan bit değerlerinin farklı tipte görülmesini ve buna göre işlenmesini sağlamaz; ancak atama yapılamayan bir *rvalue* (atama işleminde sağda bulunan değer) oluşturur. Artırma operatörü ++ ise *lvalue* gerektirir.

Programda ayrıca Adrc+2 adresinde bulunan tamsayı değer listelenir. Bunun için dizinin başlangıc adresine tamsayı 2 değerinin eklendiği ve dizinin 2.elemanının **char *** tipindeki bellek adresini veren Adrc+2 ifadesine yine tip dönüştürme 2-byte bellek alanına işaret etmesi sağlanır. Elde edilen adres değerine * operatörü uygulanarak bu alanda bulunan değer **unsigned int** olarak Tams değişkenine atanır.

C derleyicisi tip dönüştürme işleminden dolayı Adrc+2 ifadesinin döndürdüğü adres değerini **unsigned int** tipi verinin bellek adresi olarak görür ve atama deyimi ile bu adresteki 2-byte değeri (0xD4C3) Tams değişkenine atar.

Bu işlem aşağıdaki deyimler kullanılarak gerçekleştirilebilir:

```
unsigned *pi;
    ...
pi = (unsigned *)&cDizi[0];      /*   pi = (unsigned *)cDizi;   */
Tams = *( pi + 1 );              /*   Tams = pi[1];             */
```

Örnek 5.6-1 adrcevir.c programı.

```
#include <stdio.h>

#define  ELSayi(dizi)   sizeof dizi / sizeof dizi[0]

int main(void)
{
   unsigned  char cDizi [ ] = { 0xA1, 0xB2, 0xC3, 0xD4 }, *Adrc = cDizi;
   unsigned  Tams, i;

   printf( "cDizi dizisi %u byte.\n", sizeof cDizi );

   for( i = 0; i < ELSayi( cDizi ); i++ )
        printf( "%u.byte : 0x%02X\n", i+1, *(Adrc+i) );

   Adrc = (char *)( (int *)Adrc + 1 );
   printf( "*Adrc : 0x%02X\n", *Adrc );

   Adrc = cDizi;
   Tams = *(unsigned *)(Adrc + 2);
   printf( "Tams : 0x%04X\n", Tams );

   return 0;
}
```

Çıktı

```
cDizi dizisi 4 byte.
1.byte:  0xA1
2.byte:  0xB2
3.byte:  0xC3
4.byte:  0xD4

*Adrc :  0xC3

Tams   :  0xD4C3
```

Tek operand alan (*unary*) tip dönüştürme operatörü (*type cast operator* yada *type conversion operator*) ve adres operatörleri (* ve &) aynı operatör önceliğine sahiptir. Grup ilişkileri sağdan sola doğru olduğu için ifade içinde bir arada bulunduklarında sağdan sola doğru çalışırlar.

void * tipi adres değişkenleri

void tip belirleyici önceki bölümlerde değer döndürmeyen yada argüman almayan fonksiyonlar tanımlanırken kullanıldı. C dilinde, **void *** tipinde bildirilen bir adres değişkeni hiçbir veriye işaret etmez; indirek değerinin alınması anlamsızdır. Ancak **void *** tipinde bir adres değişkeninin değeri herhangi bir tipteki adres değişkenine tip dönüştürme yapılmadan atanabilir. Herhangi bir tipteki adres değişkeninin değeri de **void *** tipinde bir adres değişkenine yine tip dönüştürme yapılmadan atanabilir. Bazı uygulamalarda **void *** tipi adres değişkenleri kullanılır. Örneğin izleyen bölümlerde anlatılacak olan dinamik bellek yönetimi için kullanılan bazı standart C kütüphane fonksiyonları **void *** tipinde adres değeri döndürür. Böylece ayrılan bellek alanına erişim belli bir veri tipi ile sınırlandırılmadığı için döndürülen adres değeri herhangi bir tipteki adres değişkenine atanarak istenen şekilde erişim gerçekleştirilebilir (bazı programlarda okunabilirliği artırmak için zaman zaman döndürülen **void *** tipi adrese tip dönüştürme yapılır). Pek çok standart C kütüphane fonksiyonu da argüman olarak **void *** adres değeri alır. İzleyen bölümlerde bir bellek bloğunu bir başka adrese byte-byte kopyalamak üzere tanımlanan BlokKopya fonksiyonu da argüman olarak **void *** tipi adres değeri alan fonksiyon örneğidir.

Aşağıdaki programda **void *** tipi adres değişkenleri kullanılarak bir tamsayının byte değerleri listelenir:

Örnek 5.6-2 void.c programı.

```
#include <stdio.h>

int main(void)
{
   int  i = 0xABCD, ByteSayi;
   char c = 'A', *pc;
   void *pv;

   pv = &c;
   pc = pv;
   printf("c : %c\n", *pc );

   pv = &i;

   printf( "i : 0x%04X, &i : %p\n", i, &i );
   for ( ByteSayi = 0; ByteSayi < sizeof i; ByteSayi++ )
        printf( "%p adresinde bulunan byte : 0x%02X\n",
                (char *)pv + ByteSayi, *( (char *)pv + ByteSayi ) & 0xFF );

   return 0;
}
```

...Örnek 5.6-2 devam

Çıktı

```
c : A
i : 0xABCD, &i : 0DA2
0DA2 adresinde bulunan byte : 0xCD
0DA3 adresinde bulunan byte : 0xAB
```

Bir adres değişkenine farklı tipte bir verinin adresi atanırken tip dönüştürme işlemi uygulanarak derleyiciye bilgi verilir. Aksi taktirde derleyici uyarı mesajı verir. Programda görüldüğü gibi tip dönüştürme işlemi **void *** adres değişkenlerine atama yapılırken yada **void *** adres değişkeninin değeri bir başka adres değişkenine atanırken gerekmez.

Çıktı incelendiğinde programın derlendiği donanımda, değişkenin en düşük anlamlı (Least Significant) byte değeri düşük bellek adresinde saklanır (yani 0xCD byte'ı 0DA2 adresinde bulunur). Bu bellek düzeni *little-endian* olarak adlandırılır. Çünkü bellekte yukarı doğru giderken (artan bellek adresi yönüne doğru) verinin en düşük anlamlı byte değeri önce bulunur. Bazı donanımlar *big-endian* düzenini kullanır. Bu düzenlerden hangisinin kullanıldığı bit değerleri üzerinde işlemler yapılırken önem kazanır. ■

NOT:
Bilgisayar belleği byte'lara bölünmüştür. Her byte, 8 bit'ten oluşur (1 yada 0 olmak üzere iki değerin saklanabileceği en küçük bellek birimi 1 bit -*bi*nary digi*t*- büyüklüğündedir). Veriler bilgisayar belleğinde bit'lerden oluşan ikili sayılar (binary number) olarak saklanır (Örnek programlarda bellekte saklanan değerler sembolik bellek görünümlerinin kolay anlaşılır olabilmesi için ondalık sayılar olarak gösterildi).

Kullanılan iki ayrı fiziksel bellek organizasyon şekli vardır: ***little-endian*** ve ***big-endian***. Bunlar bilgisayarın bir ikili sayıdaki bit'leri nasıl organize ettiği ile ilgilidir. Little-endian düzeninde, birden fazla byte içeren bir ikili sayının *least significant byte*'ı düşük bellek adresine yerleştirilir. Dolayısıyla, *most significant byte* yüksek bellek adresinde saklanır. Bu farklılık bit işlemleri yapılan programlarda ve ikili tamsayıları disk dosyasına yazan programlarda taşınabilirliğini önler. Ayrıca iki farklı donanım arasında veri taşınması sözkonusu olduğunda byte sırası önem kazanır.

Binary dosya giriş/çıkış fonksiyonları fread ve fwrite sayıları ikili formatta yazarken yada okurken donanımın endian düzenine göre okur (bu programın yazıldığı donanımda byte sırası little-endian yani sağdan sola doğru olduğu için Örnek 5.6-2'de low-byte değeri high-byte öncesinde bulunur).

endian.c programında, onaltılık sayıların bit kaydırma işlemi (>>) yapılarak ikili sayı sistemindeki karşılıkları listelenmiştir. Programda ayrıca **unsigned int** ve **unsigned long int** tamsayı değişkenlerin bellek adresleri tip dönüştürme (*type-casting*) uygulanarak **char** tipi verilere işaret eden adres değişkenine atanır ve

sırasıyla bu değişkenler için bellekte ayrılan 2-byte ve 4-byte (bu donanımda) bellek alanlarına byte-byte erişilerek bu alanlarda bulunan değerler listelenir. Çıktı incelendiğinde programın oluşturulduğu donanımın bellek düzeninin **little-endian** olduğu anlaşılır.

```
/*
 *  endian.c programi.
 */

#include <stdio.h>

void   ListBit( unsigned long Sayi, size_t BitSayi );

int   main(void)
{
     unsigned   int      i    = 0xABCD,    ByteSayi;
     unsigned   long     n    = 0xAB12CD34;
     unsigned   char     *pc = (char *)&i;

     printf( "i : 0x%04X, &i : %p, ", i, &i );
     ListBit( i, sizeof(i) * 8 );

     puts( "Adres:  Deger: Hex,(Binary)\n------  --------------------");
     for ( ByteSayi = 0; ByteSayi < sizeof i; ByteSayi++ )
     {
          printf("%p      0x%02X,", pc + ByteSayi, *(pc + ByteSayi) );
          ListBit( *(pc + ByteSayi), sizeof(char) * 8 );
     }
     pc = (char *)&n;
     printf( "\nn : 0x%08lX, &n : %p, ", n, &n );
     ListBit( n, sizeof(n) * 8 );

     puts( "Adres:  Deger: Hex,(Binary)\n------  --------------------");
     for ( ByteSayi = 0; ByteSayi < sizeof n; ByteSayi++ )
     {
          printf("%p      0x%02X,", pc + ByteSayi, *(pc + ByteSayi) );
          ListBit( *(pc + ByteSayi), sizeof(char) * 8 );
     }
     return 0;
}

void   ListBit(unsigned long Sayi, size_t BitSayi)
{
        putchar( '(' );
        while ( BitSayi-- > 0 )
                putchar( '0' + (char)((Sayi >> BitSayi) & 0x01) );
        puts( ")" );
}
```

...endian.c devam

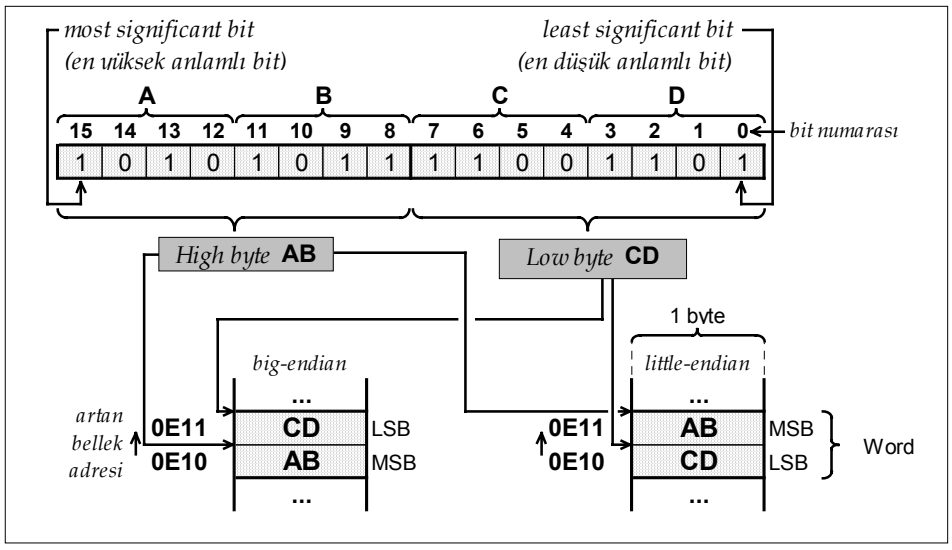
Çıktı

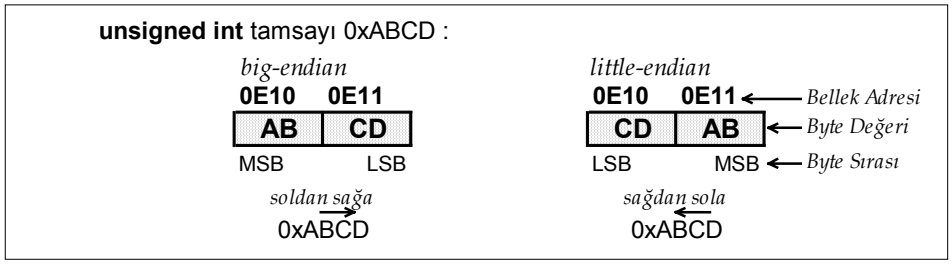
i : 0xABCD, &i : 0E10, (**10101011**11001101)
Adres: Deger: Hex,(Binary)
0E10 0xCD, (11001101)
0E11 0xAB,(10101011)

n : 0xAB12CD34, &n : 0E0A,
(**10101011**00010010**11001101**00110100)
Adres: Deger: Hex,(Binary)
0E0A 0x34, (00110100)
0E0B 0xCD, (11001101)
0E0C 0x12, (00010010)
0E0D 0xAB,(10101011)

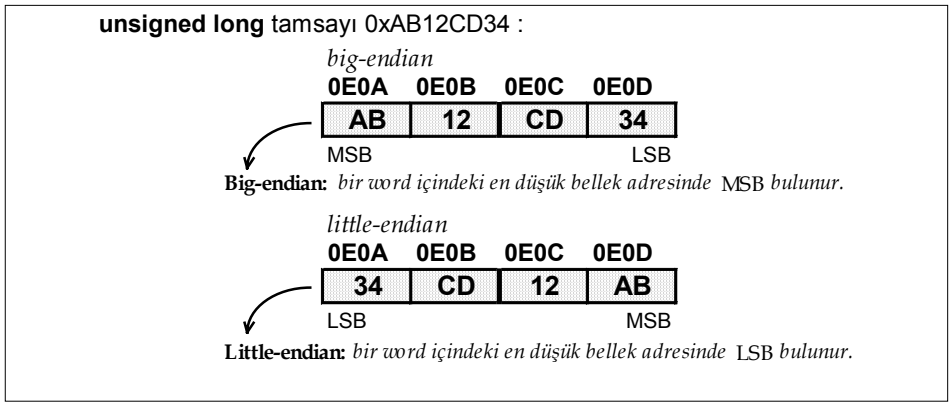
Little-endian ve ***big-endian*** donanımlarda bir ***word*** içindeki byte organizasyonunu göstermek için aşağıdaki şekiller hazırlanmıştır. Programın oluşturulduğu donanımda ***word*** büyüklüğü (***word size***) 2 byte'tır.

Onaltılık (hexadecimal) 0xABCD sayısının ikili sayı sisteminde (binary) gösterilmesi ve byte sırası:





0xAB12CD34 sayısının byte sırası:



Şekillerde de görüldüğü gibi byte sırası şöyledir:

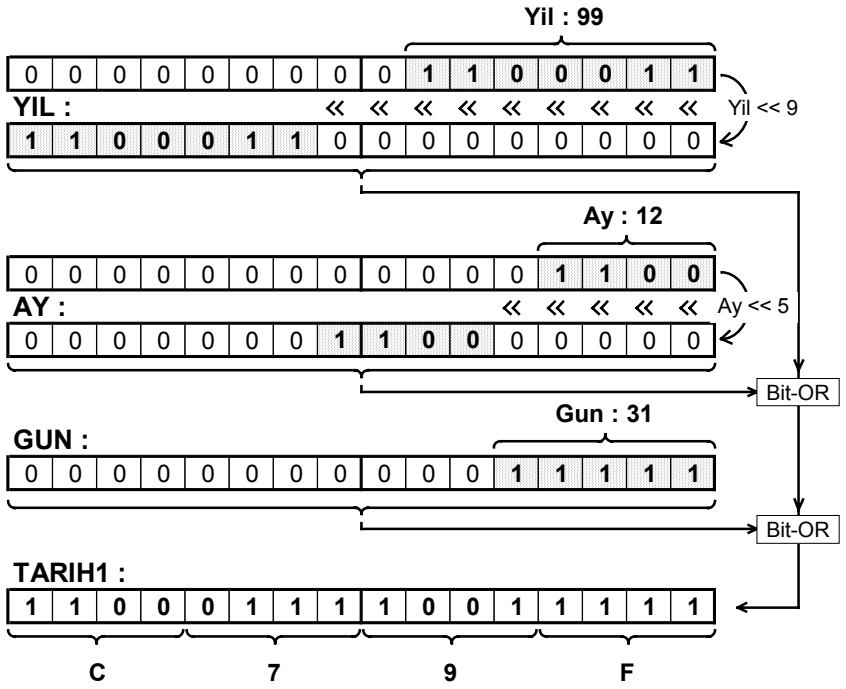
Little-endian'da artan bellek adresi yönünde ilk olarak LSB (Least Significant Byte) bulunur (en sağdaki byte: little-end). Big-endian'da ise ilk olarak MSB (Most Significant Byte) bulunur.

Standart binary giriş/çıkış fonksiyonları fread ve fwrite donanımın endian sırasına göre okur ve yazar. Yukarıdaki örnek programın ve Örnek 9.1-7'nin çıktıları incelendiğinde bu programların oluşturulduğu donanımın little-endian düzenini kullandığı anlaşılır. Byte verilerde bu şekilde bir sıralama söz konusu değildir.

Aşağıdaki program bit-alanları (Bölüm 8) konusundan sonra incelenebilir. Programda tarih bilgileri tek bir int değişkende saklanır: 0-99 arasında Yil değeri için en fazla 7 bit; 1-12 arasında ay değeri için en fazla 4 bit ve 1-31 arası gün değeri için en fazla 5 bit gereklidir. Toplam olarak 16 bit yani bu programın oluşturulduğu donanımda bir int değeri saklayabilecek bir tamsayı değişken bu iş için yeterli olacaktır. Tarih bilgisini oluşturan 16 bit'i bir int'e yerleştirmek için bit işlemleri gereklidir. 31-12-99 tarihi için bit dizilişi MSB solda bulunmak üzere onaltılık sayı sisteminde C79F değerine eşittir. Fakat bu değerin bellekte bulunuş sırası donanımın endian sırasına göre değişecektir. Aşağıdaki örnekte olduğu gibi bu iş için bit-kaydırma işlemleri yerine bit-alanı yapısı kullanıldığında donanımın ayrıntıları ile uğraşmak gerekmez. Fakat bir tamsayıyı bit-alanı yapısı olarak işleyebilmek için tamsayı değişkenin bellek adresi, tip dönüştürme uygulanarak struct GunAyYil tipi bit-alanı yapısına işaret eden adres

değerine dönüştürülür. Daha sonra ok operatörü kullanılarak bit-alanlarına erişilir ve değerler yerleştirilir.

Böylece belli bir bit sıralanışı esas alınarak gerçekleştirilen bit-kaydırma ve maskeleme işlemlerine gerek kalmaz. Örnek programda her iki uygulamaya da yer verilmiştir.




```
/*
 *  bits.c programi.
 */
#include <stdio.h>

/*  maksimum Gun 31       :  1 1 1 1 1                          */
#define BitGUN 5
/*  maksimum Ay  12         :            1 1 0 0                 */
#define BitAY   4
/*  maksimum Yil 99         :                  1 1 0 0 0 1 1    */
#define BitYIL   7
/*  Toplam 16 bit (2-byte)  :  1 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1   */

#define TarihBitKay(Gun,Ay,Yil)    \
(Yil << (BitGUN+BitAY)) | (Ay  <<  BitGUN) | Gun

void  ListBit( unsigned long Sayi, size_t BitSayi );

struct  GunAyYil
{
   unsigned Gun: BitGUN, Ay : BitAY, Yil: BitYIL;
```

```
   ... bits.c devam
};

int main(void)
{
   unsigned int Tarih1, Tarih2, Gun = 31, Ay = 12, Yil = 99;


   struct GunAyYil *pT = (struct GunAyYil *)&Tarih2;

   Tarih1     = TarihBitKay( Gun, Ay, Yil );
   pT->Gun  = Gun;
   pT->Ay    = Ay;
   pT->Yil    = Yil;

   printf( "Tarih1 : %04X, ", Tarih1 );
   ListBit( Tarih1, sizeof(Tarih1) * 8 );

   printf( "Tarih2 : %04X, ", Tarih2 );
   ListBit( Tarih2, sizeof(Tarih2) * 8 );

   printf( "Tarih2 - Gun %02d, Ay %02d, Yil 19%02d\n",
           pT->Gun, pT->Ay, pT->Yil  );

   printf("Tarih2 - Gun %02d, ", pT->Gun );
   ListBit( pT->Gun, BitGUN );

   printf("Tarih2 - Ay  %02d, ", pT->Ay  );
   ListBit( pT->Ay , BitAY  );

   printf("Tarih2 - Yil %02d, ", pT->Yil );
   ListBit( pT->Yil, BitYIL );

   return 0;
}
void ListBit( unsigned long Sayi, size_t BitSayi )
{
   putchar( '(' );
   while ( BitSayi-- > 0 )
           printf( "%1d", (Sayi >> BitSayi) & 0x01 );
   puts( ")" );
}
```

Çıktı

```
Tarih1   :  C79F,   (1100011110011111)
Tarih2   :  C79F,   (1100011110011111)
Tarih2 - Gun 31, Ay 12, Yil 1999
Tarih2 - Gun   31,  (11111)
Tarih2 - Ay     12,  (1100)
Tarih2 - Yil     99,  (1100011)
```

Aşağıdaki rutin kullanılarak donanımın hangi endian düzenini kullandığı anlaşılabilir:

```
        ...
short int WORD = 0x0001;
        ...
/* low-byte sifir degil ise : little-endian */
printf("%s\n", *((char *)&WORD) ? "little-endian" : "big-endian" );
        ...
```

5.7 Adres Değişkenleri ve Fonksiyonlar

Örnek 5.7-1'de tamsayı i ve j değişkenleri taşıdıkları değerlerin değiştirilmesi için DegerDeg fonksiyonuna argüman olarak aktarılır.

Örnek 5.7-1 degaktar.c programı.

```
#include <stdio.h>

void DegerDeg( int, int );

int main(void)
{
  int i = 10, j = 20;

  printf("i : %d, j : %d\n", i, j );
  DegerDeg( i, j );
  printf("i : %d, j : %d\n", i, j );

  return 0;
}

void DegerDeg( int a, int b )
{
  int t;

  t   = a;
  a   = b;
  b   = t;
}
```

Çıktı

```
i : 10, j : 20
i : 10, j : 20
```

DegerDeg fonksiyonu çağrısı ile i ve j değişkenlerinin bellek alanlarında saklanan değerlerin kopyaları, DegerDeg fonksiyonunun *lokal* değişkenlerine değer olarak atanır. Yani i değişkeninin değeri a parametresine, j değişkeninin değeri ise b parametresine aktarılır. DegerDeg bloğu içinde sadece bu lokal kopyalar değiştirilir. Bu nedenle çıktıda da görüldüğü gibi değiştirme işlemi sadece DegerDeg fonksiyonu içinde

gerçekleşir ve çağıran blokta yer alan i ve j değişkenlerinin değerleri aynı kalır. DegerDeg fonksiyonu **main** fonksiyonunun lokal değişkenleri olan tamsayı i ve j değişkenlerinin bellek alanlarına erişemez ve dolayısıyla bu değişkenlerin bellek alanlarında saklanan değerlerini de değiştiremez. Bu şekilde gerçekleşen bir fonksiyon çağrısı, çağrı sırasında yalnızca argüman değerleri aktarıldığı için *değer-ile-çağırma* (*call-by-value*) olarak adlandırılır.

DegerDeg fonksiyonu çağıran bloktaki i ve j değişkenlerine ancak bu değişkenlerin bellek adreslerini bildiği taktirde erişilebilir. Dolayısıyla değer-ile-çağırma yoluyla eriºemeyecektir.

C dilinde fonksiyonlara adres değerleri aktarılabilir. Fakat fonksiyon adres değerleri alabilecek şekilde tanımlanmalı ve bildirimi de (prototip) buna göre yapılmalıdır. Adres değerleri aktarılmak istendiğinde, tanımda bildirilen parametreler adres değişkenleri olmalıdır ve fonksiyon bildiriminde de karşılık gelen adres değişkeni tipleri listelenmelidir. Fonksiyonlara her çeşit bellek adresi aktarılabilir; bir basit değişkenin, bir dizinin, adres dizisinin, yapı değişkeninin ve hatta bir başka fonksiyonun bellek adresi. Bu yolla sadece basit değişkenlere değil diğer tiplere de erişilebilir.

Argüman olarak adres değeri alan bir fonksiyonun çağrısında aşağıdakilerden herhangi biri argüman olarak bulunabilir:

- sabit adres değeri,
- adres değeri döndüren bir ifade,
- adres değişkeni (basit değişkene, fonksiyona, yapı değişkenine yada bir diziye işaret eden)

Çağrıda yer alan argüman, fonksiyon tanımında parametre olarak bildirilen adres değişkeninin tipinde olmalıdır. Fonksiyon çağrısı, çağrıda argüman olarak adres değerleri yer aldığında *referans-ile-çağırma* olarak adlandırılır. Çünkü çağrı ile argümanların bellek alanlarına erişim için gerekli olan referans adres bilgisi aktarılır.

Dizi ismi, dizinin başlangıç adresini veren sabittir. Çağrıda yer alan argüman herhangi bir dizi ismi olduğunda aktarılan değer başlangıç elemanının bellek adresi olacağından dolayı otomatik olarak referans-ile-çağırma gerçekleşir.

Bir fonksiyon çağıran bloğa **return** deyimi ile sadece tekbir değer döndürür. Fakat bazı uygulamalarda birden fazla değer döndürmek gerekebilir. Böyle bir uygulamada da yine adres değişkenleri kullanılır. Ayrıca bir fonksiyon adres değeri döndürecek şekilde tanımlanabilir. Böyle bir fonksiyon adres değeri olarak herhangi bir durumu ifade etmek için boş adres değişkeni değeri de (NULL) döndürebilir.

Fonksiyon argümanı olarak adres değişkenleri

Örnek 5.7-1'deki değer değiştirme işlemi Örnek 5.7-2'de olduğu gibi DegerDeg fonksiyonuna adres değerleri aktarılarak gerçekleştirilebilir:

Örnek 5.7-2 refakt1.c programı.

```
#include <stdio.h>
void DegerDeg( int *, int * );
int main(void)
{
  int i = 10, j = 20;
  printf("i : %d, j : %d\n", i, j );
  DegerDeg( &i, &j );
  printf("i : %d, j : %d\n", i, j );
  return 0;
}
void DegerDeg( int *adr1, int *adr2 )
{
  int t;
  t     = *adr1;
  *adr1 = *adr2;
  *adr2 = t;
}
```

Çıktı

```
i : 10, j : 20
i : 20, j : 10
```

Programda görüldüğü gibi DegerDeg fonksiyonu bellek adresleri alabilecek şekilde tanımlanır; **int *** tipinde iki adres değişkeni olan adr1 ve adr2 parametreleri bildirilir. Fonksiyon prototipi de buna göre düzenlenir ve DegerDeg fonksiyonunun **int** tipinde değişkenlere işaret eden iki adres değeri beklediği bildirilir. Çağrı satırında argüman olarak &i ve &j ifadeleri kullanılarak i ve j değişkenlerinin bellek adresleri sabit adres değerleri olarak aktarılır. Bu işlem izleyen örnekte görüldüğü gibi adres değişkenleri kullanılarakta yapılabilir. Her iki uygulamada da argümanlar aynı çeşit veriyi aktarır: çağıran bloktaki lokal değişkenlerin bellek adresleri. Dolayısıyla, adres değerlerinin hangi yolla aktarıldığı DegerDeg fonksiyonunu etkilemez. DegerDeg fonksiyonu çalıştığında **int *** tipinde iki lokal adres değişkeni oluşturur (adr1 ve adr2). Aktarılan adresler bu adres değişkenlerine atanır. Çağrıdaki argümanlar ve tanımdaki parametreler birebir eşleştiği için, i değişkeninin adresi adr1 adres değişkenine, j değişkeninin adresi ise adr2 adres değişkenine atanır.

Örnek 5.7-3 refakt2.c programı.

```
#include <stdio.h>
void DegerDeg( int *adr1, int *adr2 );
int main(void)
{
   int i = 10, j = 20;
   int *pi = &i, *pj = &j;
   printf("i : %d, j : %d\n", *pi, *pj );

   DegerDeg( pi, pj );
   printf("i : %d, j : %d\n", i, j );
   return 0;
}
void DegerDeg( int *adr1, int *adr2 )
{
   int t;
   t     = *adr1;
   *adr1 = *adr2;
   *adr2 = t;
}
```

Çıktı

```
i : 10, j : 20
i : 20, j : 10
```

Fonksiyon içinde değer değiştirme sırasında geçici olarak kullanım için bellek alanı oluşturmak amacıyla, **int** tipinde t değişkeni bildirilir. İndirek değer operatörü (*) ile elde edilen değerler, t değişkeni aracılığı ile değiştirilir. Bu işlem direk olarak i ve j değişkenlerinin bellek alanları üzerinde yapılır. Böylece **main** fonksiyonunun lokal değişkenlerine indirek olarak erişilir ve değerleri değiştirilir.

Örnek 5.7-3'de yer alan fonksiyon prototipinde, okunabilirliği arttırmak için tanımda yer alan parametre isimleri de kullanılmıştır. Her iki örnekte de fonksiyon, çağıran bloğa değer döndürmeyecek şekilde tanımlanmıştır ve fonksiyon bloğunda **return** deyimi bulunmaz. Fonksiyonun çağrı satırına hiçbir değer döndürmediğini belirtmek için tanım ve prototipte **void** tip belirleyici kullanılmıştır. Fakat işlem, iki değişken üzerinde ve adres değerleri kullanılarak yapıldığı için birden fazla değer döndürülmüş gibi gerçekleşir.

Fonksiyonun adres değeri döndürmesi istendiğinde, **void** tip belirleyici yerine istenen adres değişkeni tipi kullanılır ve fonksiyonun döndürdüğü adres değeri de çağıran blokta aynı tip belirleyici ile bildirilmiş olan bir adres değişkenine atanır.

Fonksiyon argümanı olarak diziler

Önceki bölümlerde de belirtildiği gibi dizi ismi, başlangıç elemanının bellek adresini veren sabittir. Bu nedenle fonksiyon çağrılırken kullanılan argümanlar dizi ismi olduğunda aktarılan değerler bellek adresleridir (referans-ile-çağırma); bu argümanlara karşılık gelen ve fonksiyon tanımında bildirilen parametreler ise adres değişkenleridir.

Fonksiyona aktarılan dizi herhangi bir tipte olabilir. Fonksiyonlara argüman olarak dizgi sabitleri de aktarılabilir. C dilinde çift tırnaklar arasına yerleştirilen karakterlerle oluşturulan bir dizgi sabiti, karakter dizisi olarak bellekte tutulur. Bir dizgi sabitinin değeri, bellekte saklandığı alanın başlangıç adresidir. Bu nedenle bir fonksiyona herhangi bir dizgi sabiti argüman olarak verildiğinde aktarılan değer bir adres değeridir; yani bellekte bulunan bir karakter dizisinin başlangıç elemanının adresi. Çağrılan fonksiyonun tanımında bildirilen ve bu argümana karşılık gelen parametre ise **char** tipi verilere işaret eden bir adres değişkenidir.

Örnek 5.7-4'de argüman olarak aktarılan dizgi sabitinin uzunluğunu döndüren DizgiBoy fonksiyonu tanımlanır. Standart C kütüphanesinde bulunan **strlen** fonksiyonu da aynı işi yapar.

Örnek 5.7-4 dizgiboy.c programı.

```
#include <stdio.h>

unsigned DizgiBoy( char[ ] );

int main(void)
{
  char *p    = "123";
  char  s[ ] = "123";

  printf( "123 dizgisi : %u\n",         DizgiBoy ( p )         );
  printf( "s karakter dizisi : %u\n",  DizgiBoy ( s )       );
  printf( "123 dizgisi : %u\n",         DizgiBoy ( "123" )    );

  return 0;
}

unsigned DizgiBoy( char  m[ ] )
{
  unsigned i;

  for ( i = 0 ;  m[ i ] != '\0' ;  i++ ) ;

  return  i;
}
```

Çıktı

```
123 dizgisi : 3
s karakter dizisi : 3
123 dizgisi : 3
```

Programda DizgiBoy fonksiyonuna argüman olarak p adres değişkeni, karakter dizisi ismi s ve dizgi sabiti "123" aktarılır. İlk değer olarak bildirimde p adres değişkenine "123" dizgi sabitinin bellek adresi atandığı için, çağrıda argüman olarak yer aldığında DizgiBoy fonksiyonuna yine bu dizginin başlangıç adresi aktarılmış olur. Programda bildirilen s karakter dizisi **char** tipi verilerden oluşur. Dizi ismi s, çağrıda argüman olarak kullanıldığında fonksiyona karakter dizisinin başlangıç elemanı olan **char** verinin bellek adresi aktarılmış olur. Son çağrıda argüman olarak kullanılan "123" dizgi sabitinin değeri de yine bellekte bulunduğu alanın başlangıç adresi olduğu için her üç çağrıda da DizgiBoy fonksiyonuna adres değerleri aktarılmış olur. Aktarılan adres değerleri, DizgiBoy fonksiyonu başlığında bildirilen m adres değişkenine atanır. m bir adres değişkeni olduğuna göre bildirimi,

```
char m [  ];
```

yerine

```
char *m;
```

şeklinde yapılabilir. Ayrıca fonksiyon prototipi de aşağıdaki şekilde yazılabilir:

```
unsigned DizgiBoy ( char * );
```

Fakat bildirim indeks operatörü ([ ]) kullanılarak yapıldığında, aktarılan adres değerinin bir diziye ait olduğu açıkça belirtilmiş olur.

Sonuç olarak bir dizi fonksiyona argüman olarak aktarılırken, fonksiyon tanımında bildirilen parametre adres değişkenidir ve her iki şekilde de bildirilebilir: adres değişkeni olarak yada eleman sayısı verilmeden köşeli boş parantezlerle. Aktarılan değer dizinin tek bir elemanına işaret eden adres değeri olduğu için tanımda dizi boyutunun verilmesi anlamsızdır (İzleyen bölümlerde anlatılacağı gibi, çok-boyutlu diziler aktarılırken ilk boyuttaki eleman sayısı verilmeyebilir. Fakat izleyen boyut değerleri verilmelidir). Bu durum yalnızca fonksiyon tanımında bildirilen parametreler için geçerlidir.

Örneğin aşağıdaki a dizisi bildirimleri birbirinden farklıdır:

dosya 1:	***dosya 2:***
`char  a [ 5 ];`	`extern char *a;`

Dosya 1'deki bildirim 5 **char** veri için bellek alanı ayırır. Dosya 2'deki bildirim ise ilk değer ataması yapılmamış (hiçbir bellek alanına işaret etmeyen) bir adres değişkeni bildirimidir. Dolayısıyla **extern** erişimin gerçekleşmesi için bu iki bildirimin aynı şekilde yapılması gerekir:

dosya 1:	***dosya 2:***
`char  a [ 5 ];`	`extern char a [ ];`

Adres değişkenlerine indeks operatörü uygulanabilir. Bu nedenle **for** döngüsü içinde dizgi elemanlarına erişim m[i] ifadesi ile gerçekleştirilir. Döngü içinde indeks değişkeni i'nin değeri, m[i] ifadesi ile dizgi sonunu belirten '\0' karakterine (boş karakter) erişilinceye kadar arttırılır.

İndeks değişkeninin aldığı son değer dizgi uzunluğunu verir. m bir adres değişkeni olduğu için **for** döngüsü yerine aşağıdaki **while** döngüsü kullanılabilir:

```
...
i = 0;
while ( *m++ )
    i++;
...
```

DizgiBoy fonksiyonu dizgilerin uzunluğunu '\0' karakteri hariç olarak ekrana yazar. Örnek 5.7-5'deki programın çıktısında da görüldüğü gibi **sizeof** operatörü kullanıldığında, s ve "123" karakter dizilerinin uzunluğu dizgi sonunda yer alan '\0' karakteri de dahil olmak üzere verilir. p ise bu donanımda 2 byte bellek alanına sahip bir adres değişkenidir.

Örnek 5.7-5 size.c programı.

```
#include <stdio.h>

int main(void)
{
  char *p    = "123";
  char  s[ ] = "123";

  printf( "p adres degiskeni  : %u byte\n",  sizeof ( p )     );
  printf( "s karakter dizisi  : %u byte\n",  sizeof ( s )     );
  printf( "123 dizgisi        : %u byte\n",  sizeof ( "123" ) );

  return 0;
}
```

Çıktı

```
p adres degiskeni :  2 byte
s karakter dizisi :  4 byte
123 dizgisi       :  4 byte
```

Şimdiye kadar verilen tüm örnek programlar da ekran çıktısı almak için printf fonksiyonu kullanıldı. printf fonksiyonu ilk argüman olarak bir dizgi alır; izleyen argümanlar hakkında bilgi içeren bir format dizgisi. Örnek 5.7-4'de printf fonksiyonunda ikinci argüman olarak DizgiBoy fonksiyonu çağrısı yer aldı. Böylece bu fonksiyonun döndürdüğü **unsigned** değer, printf fonksiyonuna aktarılır.

Bu işlem aşağıdaki şekilde de gerçekleştirilebilir:

"123" dizgisi için:

```
    ...
char *p = "abc";
unsigned j;
    ...
j = DizgiBoy( p );
printf( "%s dizgisi : %u karakter \n",  p,  j );
    ...
```

Bildirim deyimi,

```
char *p = "123";
```

ile yapılan atama işlemi dizgi kopyalama değildir. Burada "123" dizgi sabitinin bellek adresi, p adres değişkenine değer olarak atanır. C dilinde bir dizginin tamamını tek bir birim olarak ele alarak işleyen operatörler yoktur. Dizgiler üzerinde yapılan dizgi kopyalama, dizgi karşılaştırma veya belli bir karakteri aramak amacıyla dizgi tarama gibi işlemler standart kütüphanede yer alan fonksiyonlar ve adres aritmetiği kullanılarak yapılabilir. İzleyen paragraflarda dizgiler üzerinde bu işlemleri gerçekleştiren ve argüman olarak dizgilerin bellek adreslerini alan birkaç fonksiyon tanımlanır.

Örnek 5.7-6'da DizgiKopyala fonksiyonu tanımlanır. Bu fonksiyon argüman olarak **char** tipi verilere işaret eden iki adres değeri bekler. Fonksiyon çağrısında argüman olarak karakter dizisi isimleri s1 ve s2 kullanılır. Böylece fonksiyona dizilerin başlangıç elemanlarının bellek adresleri aktarılır. Fonksiyona aktarılan adres değerleri, p1 ve p2 adres değişkenlerine atanır. DizgiKopyala fonksiyonu bu adresleri kullanarak "abc" ve "123" dizgilerine erişebilir. **while** döngüsünde yer alan p1[i] = p2[i] deyiminde, p2[i] ifadesi ile erişilen elemanın değeri p1 karakter dizisinin karşılık gelen pozisyonuna atanır. Bu işlem p2 dizisinin her elemanı ile son eleman olan '\0' karakterine erişilinceye kadar tekrarlanır. Elemanlara erişim, atama sonrası artırılan dizi indeks değişkeni i kullanılarak gerçekleştirilir. Dizgi sonunu saptamak için **while** döngüsünün test ifadesinde *gizli atama* (*embedded assignment*) uygulanarak boş karakter (*null* karakteri : '\0') kontrolu yapılır:

```
( p1[ i ] = p2[ i ] )  !=  '\0'
```

Test ifadesinde ilk olarak p2[i] ifadesi ile elde edilen değer, p1[i] ile erişilen elemana atanır ve daha sonra bu değer '\0' ile karşılaştırılır. Eğer eşit değilse döngü devam eder. Eşit ise p2 dizisinin son elemanına erişilmiştir ve döngüden çıkılır. Atama deyimi dışındaki parantezler gereklidir çünkü atama operatörü (=) diğer operatörlerden düşük önceliklidir.

Örnek 5.7-6 dizgikop.c programı.

```
#include <stdio.h>
void DizgiKopyala( char[ ], char[ ] );
int main(void)
{
   char s1[ ] = "abc";
   char s2[ ] = "123";
   printf("s1 : %s, s2 : %s\n", s1, s2 );
   DizgiKopyala( s1, s2 );
   printf("s1 : %s, s2 : %s\n", s1, s2 );
   return 0;
}
void DizgiKopyala( char p1[ ], char p2[ ] )
{
   int i = 0;
   while (  (p1[ i ] = p2[ i ]) != '\0'  )
            i++;
}
```

Çıktı

```
s1 : abc, s2 : 123
s1 : 123, s2 : 123
```

Fonksiyon içinde elemanları p1 = p2 atama işlemi ile kopyalamak mümkün değildir. Bu işlem sadece p2'nin taşıdığı adres değerini p1'e kopyalar. Ayrıca bu işlemin **main** bloğunda yer alan karakter dizileri üzerinde hiçbir etkisi yoktur. Örnek 5.7-6'da p1 ve p2 adres değişkenlerinin taşıdığı adres değerleri kopyalama öncesi ve sonrası aynı kalır. Yalnızca p2 ile indirek olarak erişilen bellek alanlarındaki değerler (s2 dizisinin elemanlarının değerleri), p1 ile erişilen alanlara (s1 dizisinin bellek alanı) kopyalanır.

Örnek 5.7-6'da adres değişkeni p1 ve p2 parametreleri, dizi bildiriminde olduğu gibi indeks operatörü kullanılarak bildirildi. DizgiKopyala fonksiyonu * operatörü kullanılarak aşağıdaki şekilde de yazılabilir:

```
void  DizgiKopyala( char  *p1, char  *p2 )
{
        while (   (*p1 = *p2) !=  '\0' )
        {
            p1++;
            p2++;
        }
}
```

++ işlemi, atama deyimi içinde yapılabilir. Ayrıca **while** döngüsü test ifadesinde dizgi sonunu saptamak için yapılan boş karakter kontrolu gereksizdir. Çünkü dizgi sonuna erişildiğinde *p2++ ifadesi, != '\0' testi ile aynı değeri verir (boş karakter değeri olan 0, koşul sağlanamadığında karşılaştırma işleminin döndürdüğü yanlış değeri ile aynıdır) ve döngüden çıkılır. Sonuç olarak DizgiKopyala aşağıdaki şekilde de yazılabilir:

```
void  DizgiKopyala( char  *p1, char  *p2 )
{
        while (   *p1++ = *p2++  ) ;
}
```

*p2++ ifadesi, p2'nin değeri artırılmadan önce işaret ettiği karakteri verir. p1'in taşıdığı adres değeri ise bu karakter p1 ile erişilen alana kopyalandıktan sonra artırılır. Aşağıdaki programda iki dizgiyi birbiri ile karşılaştıran DizgiKarsi fonksiyonu tanımlanır:

Örnek 5.7-7 dizgikar.c programı.

```
#include <stdio.h>
int DizgiKarsi( char*, char* );
int main(void)
{
  char s1 [ ]= "abcdf";
  char s2 [ ]= "12345";
  char s3 [ ]= "abcdf";
  printf( "s1 : %s, s2 : %s\n", s1, s2 );
  printf( "%s ve %s, %s\n", s1, s2, ( DizgiKarsi( s1, s2 ) == 0 ? "esit" : "esit degil") );
  printf( "s1 : %s, s3 : %s\n", s1, s3 );
  printf( "%s ve %s, %s\n", s1, s3, ( DizgiKarsi( s1, s3 ) == 0 ? "esit" : "esit degil") );
  return 0;
}
int DizgiKarsi( char *p1, char *p2 )
{
  for ( ; *p1 == *p2; p1++, p2++ )
       ;
  return( *p1 ? *p1 - *p2 : 0 );
}
```

Çıktı

```
s1 : abcdf, s2 : 12345
abcdf  ve 12345, esit degil
s1 : abcdf, s3 : abcdf
abcdf ve abcdf, esit
```

DizgiKarsi fonksiyonu argüman olarak karşılaştırılacak dizgilerin bellek adreslerini bekler. Programda argüman olarak karakter dizisi isimleri kullanılmıştır. Fakat fonksiyon dizgi sabitleri kullanılarak aşağıdaki şekilde de çağrılabilir:

```
...
DizgiKarsi ( "abcdf", "12345" );
...
DizgiKarsi ( "abcdf", "abcdf" );
...
```

Aktarılan adres değerleri p1 ve p2 adres değişkenlerine atanır. Her iki değişkenin de taşıdıkları adres değerleri birer birer arttırılır ve indirek değerler karşılaştırılır. Karakterlerin eşit olmadığı ilk durumda döngüden çıkılır. Eğer tüm karakterler birebir eşitse, **for** döngüsü dizi sonuna gelindiğinde '\0' karakterini saptar ve döngüden çıkılır. **return** deyimi içinde kullanılan ? operatörünün test ifadesinde, *p1'in değeri kontrol edilir. Dizi sonuna gelinmiºse *p1'in değeri '\0' olacağı için 0 değeri döndürülür. Eğer dizi sonuna gelmeden önce döngüden çıkılmış ise *p1 ifadesi null olmayan herhangi bir değere sahip olacağı için *p1 - *p2 ifadesinin değeri döndürülecektir. *p1-*p2 ifadesi, p1'in işaret ettiği karakter ASCII karakter tablosunda p2'nin işaret ettiği karakterden önce yer alıyorsa negatif tamsayı değer, aksi taktirde pozitif tamsayı değer verir.

DizgiKarsi fonksiyonundaki parametre bildirimleri indeks operatörü kullanılarakta yapılabilir.

Adres değeri döndüren fonksiyonlar

Bir fonksiyon adres değeri döndürecek şekilde tanımlanabilir. Örnek 5.7-8'de tanımlanan BlokKopyala fonksiyonu, ikinci parametre ile belirtilen bellek bloğunun son parametre ile belirtilen sayıda byte'ını, ilk parametre ile belirtilen bellek adresine kopyalar ve yine ilk parametre ile belirtilen bellek adresini döndürür.

Örnek 5.7-8 blokkopy.c programı.

```
#include <stdio.h>

void *BlokKopyala( void *hDizgi, const void *kDizgi, size_t ByteSayi );

int main(void)
{
  char dizgi[ ] = "abcdefg";

  printf("dizgi : %s\n", dizgi );
  BlokKopyala( dizgi, "123", 3 );
  printf("dizgi : %s\n", dizgi );

  puts( (char *)BlokKopyala( dizgi, "xyz", 3 ) );

  return 0;
}
```

... Örnek 5.7-8 devam

```
void *BlokKopyala( void *hDizgi, const void *kDizgi, size_t ByteSayi )
{
  char *h = hDizgi;
  const char *k = kDizgi;

  while ( ByteSayi-- )
        *h++ = *k++;
  return hDizgi;
}
```

Çıktı

```
dizgi : abcdefg
dizgi : 123defg
xyzdefg
```

Örnek programda **char** tipi veriler kopyalanır. Fonksiyon **void *** tipi adres değeri döndürür; ayrıca parametre listesinde de **void *** tipi adres değişkenleri bildirildiği için fonksiyonun kullanımı belli bir adres tipi ile sınırlı değildir. Dolayısıyla farklı tiplerde veriler saklayan bellek blokları aynı fonksiyon kullanılarak kopyalanabilir.

const değişkenler

Bir değişken bildiriminde kullanılan **const** anahtar sözcüğü (*type qualifier*), değişkenin değerinin değiştirilmesini önler. Değişkenin ancak bildirim sırasında ilk değer atama ile verilen değerini programın herhangi bir yerinde direk yada indirek (adres değişkeni aracılığı ile) değiştirme girişimi derleyici hatası oluşturur:

```
const int ci = 90;
ci = 90;  /* Hata */
```

Fonksiyonlar **const** olarak bildirilemezler. **const** anahtar sözcüğü dizilere, yapı değişkenlerine, union'lara, bit-alanlarına yada bunların elemanlarına uygulanabilir.

Bir bildirimde **const** anahtar sözcüğü değişken isminden önce herhangi bir yerde bulunabilir. Fakat bir bildirimde saklama sınıfı belirleyicinin (*storage class specifier*) ilk önce yazılması okunabilirlik açısından tercih edilir:

```
static const int ci = 90;
```

const, adres değişkenlerine de uygulanabilir. Pek çok standart kütüphane fonksiyonu bildiriminde olduğu gibi, BlokKopyala fonksiyonunun adres değişkeni kDizgi parametresi bildiriminde de **const** anahtar sözcüğü kullanılmıştır.

const anahtar sözcüğü (*type qualifier*) derleyiciye çağrılan fonksiyonun adresi aktarılan bellek alanında bulunan değerleri değiştiremeyeceğini bildirir. Böylece kaynak dizgiyi değiştirme girişimi, derleme sırasında hata oluşturacağından dolayı dizgi korunmuş olur. Atama işlemindeki tip uyumu ile argüman aktarmadaki tip uyumunun aynı olduğu düşünülecek olursa bu korumanın gerçekleşmesi için aktarılan adres değerinin **const** veriye ait olması gerekmez. Dolayısıyla **const** olmayan bir değişkenin adresi, bir **const**'a işaret eden adres değişkenine atanabilir. Yukarıdaki örnekte kDizgi parametresine **const** olmayan adres değeri aktarılır. Fakat fonksiyon içinde kDizgi ile erişilen alanlardaki değerleri değiştirme girişimi derleme hatası oluşturacaktır. Aynı şekilde Örnek 5.7-2'deki DegerDeg fonksiyonu **const** adres değişkeni parametrelerle tanımlanıp bildirilebilir ve **const** olmayan değişken adresleri ile çağrılabilir. Fakat yine derleme hatası oluşur. Çünkü fonksiyon içinde indirek olarak bu değerler değiştirilmek istenir. **const** olmayan bir adres değişkenine bir **const** değişkenin adresi atanamaz.
Bildirimde tip belirleyici bulunmadığı zaman **int** kabul edilir:

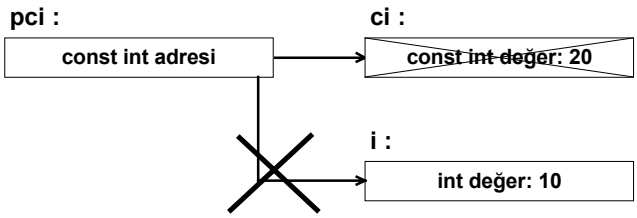
```
const ci = 20;  /*  const int ci = 20  */
```

Aşağıdaki bildirimlerde çeşitli **const** kullanımları örneklenir:

```
int i = 10;
int *pi;
const int ci = 20;          /*  const int bildirimi                                  */
const int *pci = &ci;       /*  const int'e isaret eden adres degiskeni bildirimi    */
        yada
const int *pci = &i;        /*  bildirim sırasında const olmayan degisken adresi
                                atanabilir. Tersi gecerli degildir !                 */
```

İzleyen şekilde de görüldüğü gibi pci adres değişkeni bildirim sırasında ilk değer olarak adresi atanan bir **const int** değere (ci) işaret eder. pci'nin taşıdığı adres değeri değiştirilebilir. Fakat sadece bir başka **const int** değişkenin bellek adresi atanabilir. pci adres değişkeninin indirek değeri değiştirilemez.

```
pci = &ci;
pci = &i;           /*  Hata !                                                      */
i = *pci;
*pci = 99;          /*  Hata !                                                      */
pi = &ci;           /*  Hata !                                                      */
pi = pci;           /*  Hata ! const olmayan adres değişkenine const adres
                        degeri atanamaz                                             */
```

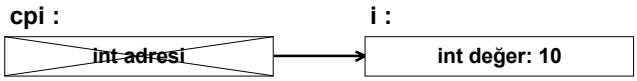


Aşağıda taşıdığı adres değeri değiştirilemeyen **const** adres değişkeni cpi bildirilir. Ancak cpi, **const** olmayan bir tamsayıya işaret eder. cpi'nin indirek değeri değiştirilebilir.

```
int i = 10;
int *pi = &i;
int *const cpi = &i;      /*  int'e isaret eden const adres degiskeni bildirimi      */
```

yada

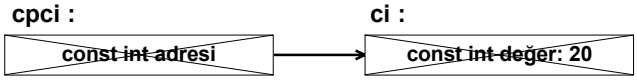
```
int *const cpi = pi;      /*  const adres degiskenine const olmayan adres degeri
                              atanabilir. Tersi gecerli degildir !                   */
```




```
*cpi = 10;
i = *cpi;
cpi = &i;                 /*  Hata !                                                 */
```

Aşağıda **const int**'e işaret eden **const** adres değişkeni cpci bildirilir.

```
const ci = 20;                  /*  const int bildirimi                            */
const int *const cpci = &ci;    /*  const int'e isaret eden const
                                    adres degiskeni  bildirimi                     */
```



cpci'nin taşıdığı adres değeri ve indirek değeri değiştirilemez:

```
*cpci = 10;               /*  Hata !                                                 */
cpci = &ci;               /*  Hata !                                                 */
```

const'a işaret eden adres değişkeni bildirimlerinde, **const** anahtar sözcüğü * öncesinde herhangi bir yerde; **const** adres değişkeni bildirimlerinde ise * sonrasında bulunur.

C dilinde bildirimlerde kullanılan bir diğer anahtar sözcük **volatile**'dir (*type qualifier*). Derleyiciye bildirilen değişkenin değerinin programın kontrolü ötesinde faktörler tarafından değiştirilebileceğini bildirir. Derleyici söz konusu değişken üzerinde optimizasyon yapmaz.

Örnek 5.7-9'de bir dizgiyi belli bir karakter için tarayan ve bulunduğu ilk pozisyonun adresini döndüren DizgiTarama fonksiyonu tanımlanır.

Örnek 5.7-9 dizgitar.c programı.

```
#include <stdio.h>

char *DizgiTarama( char *, char  );

int main(void)
{
   char dizgi[ ] = "abcdefg";
   char  c = 'e', *pdizgi;

   pdizgi = DizgiTarama( dizgi, c );

   if  ( pdizgi != NULL )
        printf(" %c, %s dizgisi icinde %d. karakterdir.\n", c, dizgi, (pdizgi-dizgi)+1 );
   else
        printf(" %s dizgisi icinde, %c karakteri bulunamadi.\n", dizgi, c );

   printf( "Bu bir %s\n", DizgiTarama( "Bu bir denemedir",  'd' ) );

   return 0;
}

char *DizgiTarama( char *pDizgi, char c )
{
   while ( *pDizgi != '\0' && *pDizgi != c )
          ++pDizgi;

   return ( *pDizgi == c ? pDizgi : NULL );
}
```

Çıktı

```
e, abcdefg dizgisi icinde 5. karakterdir.
Bu bir denemedir
```

Program "abcdefg" dizgisi içinde e karakterini bulur ve kaçıncı karakter olduğunu ekrana yazar. DizgiTarama fonksiyonu **char** tipi verilere işaret eden adres değeri döndürecek şekilde tanımlanmış ve prototipi de buna göre düzenlenmiştir.

Fonksiyon aranılan karaktere ilk kez rastladığında, bu karakteri saklayan elemanın bellek adresini döndürür. printf fonksiyonu, format dizgisinde yer alan %s format belirleyiciye karşılık gelen argüman olarak dizgi adresi bekler. Programdaki son printf çağrısında ekrana ilk olarak "Bu bir " dizgisi yazılır. Bundan sonra printf fonksiyonu %s'e karşılık gelen argüman olarak DizgiTarama fonksiyonu tarafından döndürülen "Bu bir denemedir" dizgisindeki d karakterinin bellek adresini alır; ve dizgiyi bu pozisyondan itibaren ekrana yazar.

Önceki bölümlerde fonksiyonların bir durumu ifade etmek için adres değeri olarak 0 değerini döndürebileceği ve 0 değerinin adres değişkenlerine atanabileceği belirtilmişti.

DizgiTarama fonksiyonu da verilen karakterin dizgi içinde bulunamadığını bildirmek için 0 adres değeri (NULL, yani boş adres değişkeni değeri) döndürür.

Fonksiyon prototipi ve tanımı, indeks operatörü kullanılarak hazırlanabilir:

Prototip:

```
char *DizgiTarama( char[ ], char);
```

Tanım:

```
char *DizgiTarama( char pDizgi[], char c )
{
  int i;
  i = 0;
  while ( pDizgi[ i ] != '\0' && pDizgi[ i ] != c )
          ++i;
  return ( pDizgi[ i ] == c ? &pDizgi[ i ] : NULL );
}
```

Fonksiyonda karakter bulunamadığında döndürülen değer olarak NULL ismi yerine **(char *)0** yada sadece **0** değeri kullanılabilir.

Fonksiyon Argümanı olarak Adres Dizileri

Örnek 5.7-10'da, 5 **double** veriden oluşan d dizisi bildirilir. Programda elemanlar küçükten büyüğe doğru sıralanır ve dizi bu sıralamaya göre yeniden düzenlenir. Çıktıda dizi elemanlarının sıralama öncesi ve sonrası değerleri listelenir. Sıralama işlemi dizi elemanları üzerinde yapıldığı için çıktıda da görüldüğü gibi elemanların sıralama sonrası yerleşimi farklı olacaktır. Fakat bazı uygulamalarda, dizi elemanlarının yerleşiminin bozulmaması gerekebilir.
Bu örnekte, sadece 5 elemanlı bir dizi kullanıldı. Eleman sayısı arttıkça, 8-byte bellek alanı kaplayan **double** verilerin sıralama işlemi oldukça yavaşlar.

Örnek 5.7-10 adrf10.c programı.

```
#include <stdio.h>

int main(void)
{
  double d[ ] = { 3.0, 1.9, 2.8, 5.7, 4.6 };
  int i, j, ELSayi = sizeof d  / sizeof d[0];
  double t;

  /* elemanlarin listelenmesi */
  for( i = 0; i < ELSayi; i++ )
       printf("d[%d] : %1.1f  \n",  i, d[ i ] );

  /* elemanlarin siralanmasi */
  for( i = 0; i < ELSayi - 1; i++ )
```

... Örnek 5.7-10 devam

```
        for( j = i + 1 ; j < ELSayi; j++ )
            if ( d[ i ] > d[ j ] )
            {
                t       = d[ j ];
                d[ j ]  = d[ i ];
                d[ i ]  = t;
            }
    /* siralama sonrasi elemanlarin listelenmesi */
    printf("\n");
    for( i = 0; i < ELSayi; i++ )
        printf("d[%d] : %1.1f  \n",  i, d[ i ] );
    return 0;
}
```

Çıktı

```
d[ 0 ] :  3.0
d[ 1 ] :  1.9
d[ 2 ] :  2.8
d[ 3 ] :  5.7
d[ 4 ] :  4.6

d[ 0 ] :  1.9
d[ 1 ] :  2.8
d[ 2 ] :  3.0
d[ 3 ] :  4.6
d[ 4 ] :  5.7
```

Örnek 5.4-2'de adres dizisi AdrDizi kullanılarak 5 **double** veriden oluşan ddizi dizisinin elemanlarına erişilir. Örnek 5.7-10'daki sıralama işlemi, Örnek 5.7-11'de görüldüğü gibi **double** d dizisine işaret eden adres dizisi kullanılarak gerçekleştirilebilir. Bu amaçla **double** verilere işaret eden pd adres dizisi bildirilir. Programda elemanların indirek olarak erişilen değerleri kullanılarak pd adres dizisinde saklanan bellek adresleri sıralanır. Sıralanan değerler 2-byte bellek alanında saklanan bellek adresleri olduğu için sıralama işlemi çok daha hızlı gerçekleşir. pd adres dizisinin taşıdığı adres değerleri sıralamaya göre yeniden düzenlenir ve böylece d dizisinin sıralama işlemi öncesi dizilişi de bozulmamış olur.

Örnek 5.7-11 adrf11.c programı.

```
#include <stdio.h>

#define ELSayi  5

int main(void)
{
    double d[ ] = { 3.0, 1.9, 2.8, 5.7, 4.6 };
    double *pd[ ELSayi ];
```

... Örnek 5.7-11 devam

```
   double *t;
   int i, j;
   /*  adres dizisine deger atama ve elemanlarin listelenmesi  */
   for( i = 0; i < ELSayi; i++ )
   {
        pd[ i ] = &d[ i ];
        printf("pd[%d] : %p, *pd[%d] : %1.1f, d[%d] : %1.1f \n",
                i, pd[ i ], i, *pd[ i ], i, d[ i ] );
   }
   /*  elemanlarin degerlerine gore bellek adreslerinin siralanmasi  */
   for( i = 0; i < ELSayi - 1 ; i++ )
        for( j = i + 1 ; j < ELSayi; j++ )
             if ( *pd[ i ] > *pd[ j ] )
             {
                  t       = pd[ j ];
                  pd[ j ] = pd[ i ];
                  pd[ i ] = t;
             }
   /*  degerlerin listelenmesi  */
   printf("\n");
   for( i = 0; i < ELSayi; i++ )
        printf("pd[%d] : %p, *pd[%d] : %1.1f, d[%d] : %1.1f \n",
                i, pd[ i ], i, *pd[ i ], i, d[ i ] );
   return 0;
}
```

Çıktı

```
pd[0] :  1074,   *pd[0]:  3.0, d[0]:  3.0
pd[1] :  107C, *pd[1]:  1.9, d[1]:  1.9
pd[2] :  1084,   *pd[2]:  2.8, d[2]:  2.8
pd[3] :  108C, *pd[3]:  5.7, d[3]:  5.7
pd[4] :  1094,   *pd[4]:  4.6, d[4]:  4.6

pd[0] :  107C, *pd[0]:  1.9, d[0]:  3.0
pd[1] :  1084,   *pd[1]:  2.8, d[1]:  1.9
pd[2] :  1074,   *pd[2]:  3.0, d[2]:  2.8
pd[3] :  1094,   *pd[3]:  4.6, d[3]:  5.7
pd[4] :  108C, *pd[4]:  5.7, d[4]:  4.6
```

Çıktıda görüldüğü gibi işlem adres değerleri üzerinde yapıldığı için pd adres dizisinin taşıdığı bellek adreslerinin sıralama sonrası dizilişi farklı olacaktır. Fakat d dizisinin eleman yerleşimi aynı kalır. Programdaki listeleme ve sıralama işlemleri fonksiyon haline getirilerek **main** bloğundan ayrılabilir. Örnek 5.7-12'de listeleme ve sıralama işlemleri için listele ve sirala fonksiyonları tanımlanmıştır. Bu program da yine aynı çıktıyı verir.

Bu programda yer alan fonksiyon prototipleri okunabilirliği arttırmak için parametre isimleri kullanılarak aşağıdaki gibi yazılabilir:

```
/* fonksiyon bildirimleri */
void sirala(  int   ELSayi, double *p[ ] );
void listele(  int   ELSayi, double *p[ ], double dd[ ] );
```

Yukarıdaki prototiplerde de görüldüğü gibi listele ve sirala fonksiyonları argüman olarak *adres dizisine işaret eden adres değeri* almayı bekler. Her iki fonksiyona da adres değeri (adres dizisi adresi) aktarma işlemi, çağrı satırlarında argüman olarak adres dizisi ismi pd kullanılarak gerçekleştirilir:

```
listele(   BOY, pd,  d );
...
sirala(   BOY, pd );
```

Argüman pd ile alınan adres değeri, fonksiyonların kendi lokal adres değişkeni olan p adres değişkenine (her iki fonksiyonda da, adres değişkeni için aynı parametre ismi kullanılabilir) atanır. Böylece fonksiyonlar p adres değişkeni aracılığı ile çağıran bloktaki adres dizisine ve indirek olarakta d dizisine erişebilir ve bu diziler üzerinde işlemler yapabilir. listele ve sirala fonksiyonlarının parametre bildiriminde indeks operatörü (**[ ]**) yer aldığı halde,

```
double *p[ ]
```

bildiriminde yer alan p, sabit değil bir adres değişkenidir. Aynı şekilde,

```
double dd[ ]
```

bildiriminde yer alan dd, bir adres değişkenidir.

Önceki bölümlerde indeksi olmayan dizi isminin (çağrıda argüman olarak yer alan dizi isimleri pd ve d gibi) bir adres değişkeni olmadığı, fakat dizinin başlangıç elemanının adresini veren sabit olduğu belirtildi. Dolayısıyla dizi ismine (**double** dizi ismi d yada adres dizisi ismi pd gibi) atama yapmak, ++ yada -- operatörleri uygulamak mümkün değildir. Herhangi bir dizi ismi bir fonksiyona argüman olarak aktarıldığında, fonksiyon argüman değeri olarak dizi isminin verdiği başlangıç adresini alır ve bu adres değerini, kendi lokal adres değişkenine (sadece fonksiyon bloğu içinden erişilebilen) atar.
Bu adres değişkenine başka bir adres değeri atanabilir; taşıdığı adres değeri değiştirilebilir; ++ yada -- operatörleri uygulanabilir. Fonksiyona adres dizisi aktarıldığında ise alınan adres değeri bir başka adres değişkeninin (adres dizisinin başlangıç elemanı) adresi olduğu için fonksiyon tanımında bildirilen lokal adres değişkeni çift adres değişkeni'dir.

Bu nedenle prototipler ve parametre bildirimleri aşağıdaki gibi yazılabilir:

Bildirimler:

```
/* fonksiyon bildirimleri */
void sirala(  int,  double ** );
void listele(  int,  double **, double [ ] );
```

yada

```
/* fonksiyon bildirimleri */
void sirala(  int   ELSayi, double **p );
void listele(  int   ELSayi, double **p, double dd[ ] );
```

Parametre bildirimleri:

```
void sirala( int  ELSayi, double **p )
{
  ...
}
void listele( int  ELSayi, double **p, double dd[] )
{
  ...
}
```

Ayrıca dd dizisi için prototip ve tanımlarda indeks operatörü kullanılarak yapılan,

double dd[ ]

bildirimi yerine,

double *dd

bildirimi kullanılabilir.

Fakat prototip ve parametre bildirimlerinde indeks operatörü ([ ]) kullanılması, adres değişkeninin bir dizinin başlangıcına işaret ettiğini açıkça belirtir.

Örnek 5.7-12 adrf12.c programı.

```
#include <stdio.h>

#define BOY  5

void   sirala(   int,  double *[ ] );
void   listele(   int,  double *[ ], double [ ] );
```

... Örnek 5.7-12 devam

```
int main(void)
{
   int n;
   double d[ ] = { 3.0, 1.9, 2.8, 5.7, 4.6 };
   double *pd[BOY];

   for( n = 0; n < BOY ; n++ )
         pd[ n ] = &d[ n ];
   listele(  BOY, pd, d );
   sirala(   BOY, pd );

   printf("\n");
   listele(  BOY, pd, d );

   return 0;
}
void sirala( int  ELSayi, double *p[ ] )
{
   int i, j;
   double *t;

   for( i = 0; i < ELSayi - 1 ; i++ )
         for( j = i + 1 ; j < ELSayi; j++ )
               if ( *p[ i ] > *p[ j ] )
               {
                     t       = p[ j ];
                     p[ j ]  = p[ i ];
                     p[ i ]  = t;
               }
}
void listele( int  ELSayi, double *p[ ], double dd[ ] )
{
   int l;
   for( l = 0; l < ELSayi; l++)
         printf("pd[%d] : %p,  *pd[%d] : %1.1f, d[%d] : %1.1f \n",
                  l, p[ l ], l, *p[ l ], l, dd[ l ] );
}
```

Çıktı

Örnek 5.7-11 ile aynı.

Örnek 5.7-12'de diziler üzerinde işlemler indeks operatörü kullanılarak yapıldı. Bu işlemler Örnek 5.7-13'de olduğu gibi adres aritmetiği uygulanarak gerçekleştirilebilir.

Bu amaçla pd adres dizisine d dizisinin elemanlarının bellek adreslerini atayan,

```
pd[ n ] = &d[ n ];
```

deyimi yerine,

```
*( pd+i ) = d + i;
```

deyimi kullanılabilir.

Örnek 5.7-13 adrf13.c programı.

```
#include <stdio.h>

#define BOY  5

void   sirala(   int,  double ** );
void   listele(   int,  double **, double [ ]);

int main(void)
{
   int i;
   double d[ ] = { 3.0, 1.9, 2.8, 5.7, 4.6 };
   double *pd[BOY];

   for( i = 0; i < BOY ; i++ )
        *( pd+i ) = d + i;

   listele(   BOY, pd, d );
   sirala(   BOY, pd );

   printf("\n");
   listele(   BOY, pd, d );

   return 0;
}

void sirala( int  ELSayi, double **p )
{
   int i, j;
   double *t;

   for( i = 0; i < ELSayi - 1 ; i++ )
        for( j = i + 1 ; j < ELSayi ; j++ )
             if ( **(p+i) > **(p+j) )
             {
                t              = *( p + j );
                *( p + j )= *( p + i );
                *( p + i )= t;
             }
}
```

...Örnek 5.7-13 devam

```
void listele( int  ELSayi, double **p, double dd[ ])
{
   int l;
   for( l = 0; l < ELSayi; l++)
        printf("pd[%d] : %p,  *pd[%d] : %1.1f, d[%d] : %1.1f \n",
                l, *( p + l ), l, **( p + l ), l, dd[ l ] );
}
```

Çıktı

Örnek 5.7-11 ile aynı.

Örnek 5.7-13'de listele fonksiyonunda yer alan *(p + l) ifadesi adres değeri, **(p + l) ifadesi ise bu adreste bulunan değeri verir. p parametresi çift adres değişkeni olduğundan indirek değere erişim için * operatörü iki kez uygulanmalıdır. Aynı şekilde sirala fonksiyonundaki işlemler de adres aritmetiği kullanılarak gerçekleştirilebilir. Örnek programlarda adres dizisi kullanılarak **double** verilerden oluşan bir dizi üzerinde sıralama işlemleri yapıldı. Dizgiler ve yapılar gibi bellekte çok yer kaplayan verilerin sıralama işlemlerinin bu verilere işaret eden adres dizileri üzerinde yapılması oldukça verimlidir.

Fonksiyona işaret eden adres değişkeni

Bir fonksiyon değişken değildir fakat fonksiyona işaret eden adres değişkeni bildirilebilir. Herhangi bir fonksiyonun adresi bu adres değişkenine atanabilir; argüman olarak fonksiyonlara aktarılabilir yada adres dizilerine yerleştirilebilir. Ayrıca bir fonksiyon başka bir fonksiyonun adresini döndürebilir. Fonksiyona işaret eden bir adres değişkeni aşağıdaki şekilde bildirilir:

tip (**adr*) (*parametre tip listesi*);

Bu bildirimde *tip*, *adr* adres değişkeninin işaret ettiği fonksiyonun döndüreceği değerin tipidir. **adr* ifadesi dışında bulunan parantezler kullanılmadığında, operatör önceliğinden dolayı bu bildirim ile adres değeri döndüren bir fonksiyon bildirilmiş olur (bildirilen *adr*, *tip* tipinde verilere işaret eden adres değeri döndüren fonksiyon olur). Çünkü * operatörü *adr* yerine *tip* ile birleşir. Bildirimin sonunda bulunan ve parantezlerle sınırlanmış parametre tip listesi, adres değişkeninin bir fonksiyona işaret ettiğini belirtir. Bu bildirimde *adr*, *tip* tipinde değerler döndüren fonksiyona işaret eden adres değişkeni; **adr* ise fonksiyonun kendisidir.

Örnek programda,

```
int (*adr_fonk) (const char *);
```

bildirim deyimi ile **int** değer döndüren ve bir **const char *** tipinde argüman alan fonksiyona işaret eden adr_fonk adres değişkeni bildirilir.

Bu adres değişkenine **const char *** tipi argüman alan bir fonksiyonun adresi atanabilir. *adr_fonk ifadesinin dışında bulunan parantezler kullanılmadığında,

```
int  *adr_fonk (const char *);
```

bildirim deyimi ile bir **int** verinin adresini döndüren adr_fonk fonksiyonu bildirilmiş olur.

Örnek 5.7-14 adrfonk1.c programı.

```
#include <stdio.h>
int main(void)
{
  int (*adr_fonk) (const char *);
  adr_fonk = puts;
  (*adr_fonk)("abc");
  return 0;
}
```

Çıktı

```
abc
```

Programda,

```
adr_fonk = puts;
```

atama deyimi ile adres değişkenine puts fonksiyonun adresi atanır. Çünkü yukarıdaki atama deyiminde olduğu gibi bir ifadede fonksiyon ismi (puts), yine aynı fonksiyona işaret eden adres değerine dönüşür. Bu nedenle yukarıdaki atama deyiminde puts öncesinde & operatörü gerekli değildir. Standart kütüphane fonksiyonu puts, argüman olarak aktarılan dizgiyi ekrana yazar ve durum gösteren bir **int** değer döndürür. Fonksiyon ismi yukarıdaki atama deyiminde parantezler olmadan kullanılır çünkü bu bir fonksiyon çağrısı değildir. Dolayısıyla fonksiyon çağırma operatörü (()) kullanılmaz. Çağrı satırı,

```
(*adr_fonk) ("abc");
```

ile puts fonksiyonuna adres değişkeni aracılığı ile indirek olarak erişilir ve ekrana abc dizgisi yazılır. Fonksiyona işaret eden adres değişkeni ile gerçekleştirilen çağrıda * operatörü kullanılmayabilir.
Dolayısıyla çağrı aşağıdaki şekilde gerçekleştirilebilir:

```
adr_fonk("abc");
```

Bir fonksiyonun argüman olarak bir başka fonksiyona aktarılması

Bir fonksiyon ismi, bir başka fonksiyon çağrısında argüman olarak kullanıldığında aktarılan değer bu fonksiyonun adresidir. Bu uygulamaya örnek olarak, son argümanı fonksiyona işaret eden adres değişkeni olan qsort fonksiyonu gösterilebilir:

```
void qsort (  void *baslangic,  size_t el_sayi,
                                size_t byte,
                                int (*Adr)( const void *adr1, const void *adr2 )  );
/* stdlib.h */
```

Standart kütüphane sıralama fonksiyonu qsort, herbiri *byte* büyüklüğünde *el_sayi* elemandan oluşan ve bellek adresi *baslangic* olan eleman ile başlayan diziyi sıralar. Dizinin iki elemanını birbiri ile karşılaştırmak için adresi *Adr* olan karşılaştırma fonksiyonunu çağırır. Argüman olarak karşılaştırılan iki elemanın adresini alan bu fonksiyon indirek olarak erişilen değerler arasındaki ilişkiye göre negatif (deger1 < deger2), 0 (deger1 == deger2) yada pozitif (deger1 > deger2) bir değer döndürmelidir. qsort fonksiyonu her tipte dizi ile çalışabilir. Ancak dizi tipine göre aynı prototipe sahip fakat iç yapısı farklı karşılaştırma fonksiyonları geliştirilmelidir. Örneğin aşağıdaki fonksiyon bir tamsayı dizinin elemanlarını karşılaştırmak için kullanılabilir:

```
int IntKarsi( const void *i, const void *j )  /* tamsayi dizi */
{
      if ( *(int*)i < *(int*)j )
         return -1;
      else if ( *(int*)i  == *(int*)j )
         return 0;
      else
         return 1;
}
```

Sonuç olarak fonksiyon adresi alacak şekilde tanımlanmış herhangi bir fonksiyon bir diğerini indirek olarak adres değişkeni aracılığı ile çağırabilir.

Örnek 5.7-15 qsort.c programı (Ornek 5.7-11).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BOY  5

void   List( int, double *[ ], double [ ] );
int   dAdrKarsi(      const void *adr1, const void *adr2 );
int   DoubleKarsi(  const void *adr1, const void *adr2 );
int   DizgiKarsi(   const void *adr1, const void *adr2 );

int main(void)
{
   int n;
   double d[ ]      = { 3.0, 1.9, 2.8, 5.7, 4.6 }, *pd[ BOY ];
   char *Adr[ ]  = { "defgabc", "cdefgab", "bcdefga", "abcdefg" };

   for( n = 0; n < BOY ; n++ )
         pd[ n ] = &d[ n ];

   List( BOY, pd, d );

   puts( "adres dizisi sirala" );
   qsort( pd, BOY, sizeof (double*), dAdrKarsi );
   List( BOY, pd, d );

   puts( "double dizi sirala" );
   qsort( d, BOY, sizeof (double), DoubleKarsi );
   List( BOY, pd, d );

   puts( "adres dizisi sirala" );
   qsort( Adr, sizeof Adr / sizeof Adr[0], sizeof Adr[0], DizgiKarsi );
   for ( n = 0; n < sizeof Adr / sizeof Adr[0]; n++ )
         puts( Adr[ n ] );

   return 0;
}
void List( int  ELSayi, double *p[ ], double dd[ ])
{
   int i;

   putchar( '\n' );
   for( i = 0; i < ELSayi; i++)
         printf("*pd[%d] : %1.1f, d[%d] : %1.1f \n", i, *p[ i ], i, dd[ i ] );
   putchar( '\n' );
}

int  dAdrKarsi( const void *adr1, const void *adr2 )
{
   const double d1 = **(double **)adr1;
   const double d2 = **(double **)adr2;

   if ( d1 < d2 )
```

...Örnek 5.7-15 devam

```
        return -1;
    else if ( d1 == d2 )
        return 0;
    else
        return 1;
}
int  DoubleKarsi( const void *adr1, const void *adr2 )
{
    const double d1 = *(double *)adr1;
    const double d2 = *(double *)adr2;

    if ( d1 < d2 )


        return -1;
    else if ( d1 == d2 )
        return 0;
    else
        return 1;
}
int  DizgiKarsi( const void *adr1, const void *adr2 )
{
    const char *s1 = *(char **)adr1;
    const char *s2 = *(char **)adr2;

    return strcmp( s1, s2 );
}
```

Çıktı

```
*pd[0] : 3.0, d[0] : 3.0
*pd[1] : 1.9, d[1] : 1.9
*pd[2] : 2.8, d[2] : 2.8
*pd[3] : 5.7, d[3] : 5.7
*pd[4] : 4.6, d[4] : 4.6

adres dizisi sirala

*pd[0] : 1.9, d[0] : 3.0
*pd[1] : 2.8, d[1] : 1.9
*pd[2] : 3.0, d[2] : 2.8

*pd[3] : 4.6, d[3] : 5.7
*pd[4] : 5.7, d[4] : 4.6

double dizi sirala

*pd[0] : 2.8, d[0] : 1.9
*pd[1] : 3.0, d[1] : 2.8
*pd[2] : 1.9, d[2] : 3.0
*pd[3] : 5.7, d[3] : 4.6
```

...Örnek 5.7-15 devam

```
*pd[4] : 4.6, d[4] : 5.7

adres dizisi sirala
abcdefg
bcdefga
cdefgab
defgabc
```

Örnek 5.7-16'da argüman olarak **int** tipi değerler döndüren bir fonksiyona işaret eden adres değişkeni bekleyen Liste fonksiyonu tanımlanmıştır.

Fonksiyon bildirimi okunabilirliği arttırmak için parametre ismi kullanılarak aşağıdaki gibi yazılabilir:

```
void  Liste(  int (*adr_fonk)( const char *s )  );
```

Örnek 5.7-16 adrfonk2.c programı.

```
#include <stdio.h>
void Liste( int (*)(const char *) );
int main(void)
{
   Liste( puts );
   return 0;
}
void Liste( int (*adr_fonk)(const char *) )
{
    adr_fonk("abc");
}
```

Çıktı

```
abc
```

Liste fonksiyonu başlığında adr_fonk parametresinin **int** tipi değerler döndüren ve bir **char *** argümana sahip fonksiyona işaret eden adres değişkeni olduğu bildirilir. Liste fonksiyonu aktarılan fonksiyon adresini lokal adres değişkeni adr_fonk'a atar. Bu adres değişkeni ile indirek olarak puts fonksiyonuna erişir ve abc dizgisini ekrana yazar. Bildirimde Liste fonksiyonunun argüman olarak fonksiyona işaret eden adres değişkeni beklediği belirtildiği için çağrıda & operatörü kullanımı gereksizdir.

Örnek programlarda adres dizisi kullanılarak **double** veriler sıralandı. Sıralama ve listeleme işlemleri argüman olarak adres dizisi alan fonksiyonlarla gerçekleştirildi. Örnek

5.7-17'de listeleme işlemi sirala fonksiyonu içinde yapılır. sirala fonksiyonu argüman olarak fonksiyon adresi alacak şekilde tanımlanır.

Örnek 5.7-17 adrfonk3.c programı.

```
#include <stdio.h>

#define BOY  5

void   sirala(   int,  double **, void (*)( int, double ** ) );
void   listele(   int,  double ** );

int main(void)
{
   int i;
   double d[ ] = { 3.0, 1.9, 2.8, 5.7, 4.6 };
   double *pd[ BOY ];
   void (*adr_fonk)( int, double **, void (*)(int, double **) );

   for( i = 0; i < BOY ; i++ )
        *( pd + i ) = d + i;

   adr_fonk = sirala;
   (*adr_fonk)( BOY, pd, listele );

   return 0;
}

void sirala( int  ELSayi, double **ps, void (*adr_list)( int, double ** ) )
{
   int i, j;
   double *t;

   for( i = 0; i < ELSayi - 1 ; i++ )
        for( j = i + 1 ; j < ELSayi ; j++ )
             if ( **( ps + i ) > **( ps + j ) )
             {
                t              = *( ps + j );
                *( ps + j )  = *( ps + i );
                *( ps + i )  = t;
}
   (*adr_list)( ELSayi, ps );
}

void listele( int  ELSayi, double **pl )
{
   int l;
   for( l = 0; l < ELSayi; l++)
        printf("eleman %d - %1.1f \n",  l + 1, **( pl + l ) );
}
```

...Örnek 5.7-17 devam

Çıktı

```
eleman 1 - 1.9
eleman 2 - 2.8
eleman 3 - 3.0
eleman 4 - 4.6
eleman 5 - 5.7
```

Aşağıdaki fonksiyon prototipi, sirala fonksiyonunun argüman olarak bir **int** değer, bir çift adres değeri ve bir fonksiyon adresi (**int** ve **double** ** tipi iki argüman alan ve hiç bir değer döndürmeyen) beklediğini belirtir:

Prototip :

```
void sirala( int, double **, void (*)( int, double ** ) );
```

Fonksiyon başlığında ise aktarılan fonksiyon adresinin atanacağı adr_list parametresi (sirala fonksiyonunun lokal adres değişkeni) bildirilir:

Fonksiyon başlığı:

```
void sirala( int  ELSayi, double **ps, void (*adr_list)(int, double **) )
```

main bloğu içinde bildirilen adr_fonk adres değişkeni, sirala fonksiyonuna erişim için kullanılır. adr_fonk adres değişkeni ile indirek olarak erişilen sirala fonksiyonu, çağrı satırı,

```
(*adr_fonk)( BOY, pd, listele );
```

ile listele fonksiyonunun üçüncü argüman olarak aktarılan ve kendi lokal adres değişkeni adr_list'e atanan adresini alır ve bu yolla eriştiği listele fonksiyonunu kullanarak sıralanmış değerleri ekrana yazar.

Fonksiyon Tablosu

Önceki bölümlerde dizgilere ve **double** verilere işaret eden adres dizileri bildirildi. Adres dizisi, aynı tipte elemanlardan oluşan her çeşit veri grubunun bellek adreslerini taşıyabilir. Elemanları aynı prototipe sahip fonksiyonların adresini saklayan adres dizisi *fonksiyon tablosu* olarak adlandırılır. Dizi elemanlarının hepsi aynı tipte olduğu için işaret edilen fonksiyonlar da aynı tipte değerler döndürmeli ve aynı parametre listesine sahip olmalıdır. Fonksiyon tablosu aşağıdaki şekilde bildirilir:

tip (**tablo_ismi* [n]) (*parametre-tip-listesi*);

Bu bildirimde *tip*, fonksiyonların döndürdüğü değerin tipi; *tablo_ismi*, C dilinin kurallarına uygun olarak belirlenen fonksiyon tablosu ismi; n, tabloda yer alan fonksiyon sayısı; *parametre-tip-listesi* ise fonksiyonların virgül ile ayrılmış olarak verilen parametre tiplerinin listesidir. Bildirim sırasında ilk değer atama yapılabilir:

tip (**tablo_ismi* [ ]) (*parametre-tip-listesi*) = { *adres_1*, ... , *adres_n* };

Programlarda belli bir koşula göre yapılan fonksiyon çağrıları, **if** ve **switch** deyimleri yerine fonksiyon tablosundan yine seçime bağlı olarak belirlenen dizi indeksi ile elde edilen fonksiyon adresi kullanılarak gerçekleştirilebilir. Tablo kullanılarak fonksiyon çağrılması aşağıdaki şekilde olur:

(**tablo_ismi* [indeks]) (aktarılan-argumanlarin-listesi);

Eğer fonksiyon herhangi bir değer döndürüyor ise bu değer çağrı ifadesi uygun şekilde bildirilmiş bir değişkene atanarak alınabilir.

Örnek 5.7-18'de üç ayrı fonksiyon tanımlanır: menu, giris ve liste fonksiyonları. Adres değişkeni aracılığı ile erişilen menu fonksiyonu, basit bir menu listeler ve ekrandan aldığı seçenek numarasını döndürür. Ana programda **switch** deyimi kullanılarak seçime göre giris yada liste fonksiyonları çağrılır.

Örnek 5.7-19'da **switch** deyimi kullanılmadan yine menu fonksiyonunun döndürdüğü seçenek numarasına göre giris yada liste fonksiyonları çağrılır. Bu amaçla fonk_t adres dizisi bildirilir ve ilk değer olarak giris ve liste fonksiyonlarının bellek adresleri atanır. Değer atama yapılırken, önceki örnekler de olduğu gibi fonksiyon isimleri & operatörü olmadan kullanılır. Kullanım kolaylığı amacıyla fonksiyonlara erişim için kullanılan ifadeler için **#define** ön-işlemci komutu ile sembolik isimler tanımlanmıştır. Programda adres dizisinin 0 indeksli başlangıç elemanı kullanılmamıştır. Ayrıca bu elemana atanan 0 değeri, atama öncesi tip çevirme uygulanarak fonksiyon tipine çevrilir. Sonuç olarak **main** bloğu iki satırdan oluşan bir program haline gelir. Çok sayıda fonksiyon içeren programlarda fonksiyon tablosu kullanımı oldukça pratiktir.

Örnek 5.7-18 fonktab1.c programı.

```
#include <stdio.h>

int menu(void)
{
  int secim;

  while ( 1 ) {
        printf( "Secenekler\n"
                "----------------  \n"
                "1-Giris          \n"
                "2-Liste          \n"
```

...Örnek 5.7-18 devam

```
                "3-Cikis          \n"
                "Seciminiz :    "   );
  scanf("%d", &secim );
  if ( secim < 1 || secim > 3 )
       printf("\t\t hatali giris ...\n");
  else
       break;
  } /* while */
  return secim;
}
void giris(void)
{
  printf("\t\t giris ...\n");
}
void liste(void)
{
  printf("\t\t liste ...\n");
}
int main(void)
{
  int i, (*adr_menu)(void) = menu;
  while (  (i = (*adr_menu)( ) ) != 3 )
           switch( i )
           {
                case 1:
                          giris();
                          break;
                case 2:
                          liste();
                          break;
                default:
                          break;
           }
  return 0;
}
```

Çıktı

```
Secenekler
1 -  Giris
2 -  Liste
3 -  Cikis
Seciminiz :  1
             giris ...
Secenekler
1 -  Giris
```

...Örnek 5.7-18 Çıktı devam

```
2 -  Liste
3 -  Cikis
Seciminiz :  2
             liste ...
Secenekler
1 -  Giris
2 -  Liste
3 -  Cikis
Seciminiz :  3
```

Aşağıdaki program da yine aynı çıktıyı verir.

Örnek 5.7-19 fonktab2.c programı.

```
#include <stdio.h>
int menu(void)
{
  int secim;
  while ( 1 )
  {
       printf( "Secenekler\n"
               "----------------  \n"
               "1-Giris          \n"
               "2-Liste          \n"
               "3-Cikis          \n"
               "Seciminiz :    "  );
       scanf("%d", &secim );
       if ( secim < 1 || secim > 3 )
            printf("\t\t hatali giris ...\n");
       else
            break;
  } /* while */
  return secim;
}
void giris(void)                          /* giris fonksiyonu tanimi */
{
  printf("\t\t giris ...\n");
}
void liste(void)                          /* liste fonksiyonu tanimi */
{
  printf("\t\t liste ...\n");
}
```

...Örnek 5.7-19 devam

```
void (*fonk_t [ ] )(void) = {  (void (*)())0, giris, liste };    /*  fonksiyon tablosu tanimi */
int i, (*adr_menu)(void) = menu;              /* global degisken i ve adr_menu bildirimi  */
#define FONK   (*fonk_t[ i ])                   /* fonksiyon cagrilari icin isim tanimlanmasi */
#define MENU  (*adr_menu)
int main(void)
{
  while ( (i = MENU( ) ) != 3 )
  FONK( );
  return 0;
}
```

Çıktı

Örnek 5.7-18 ile aynı çıktı.

Komut Satırı Argümanları

Önceki bölümlerde yer alan programlarda, **main** fonksiyonu parametresiz olarak tanımlandı. Komut satırına boşluk ile ayrılmış olarak komut isminden sonra girilen ve *komut satırı argümanları* olarak adlandırılan bilgilerin program içinde kullanılabilmesi için **main** fonksiyonu iki parametre ile tanımlanır. Program çalışmaya başladığında komut satırına girilen bu bilgiler işletim sistemi tarafından **main** fonksiyonunun parametrelerine değer olarak aktarılır. Örnek 5.7-20'de **main** fonksiyonu argc (***arg****ument* ***c****ount*) ve argv (***arg****ument* ***v****ector*) parametreleri ile tanımlanmıştır. argc ve argv parametreleri aracılığı ile komut satırı argümanlarına program içinden erişilebilir. Yaygın olarak kullanılan argc ve argv isimleri yerine başka isimler de kullanılabilir.

Örnek 5.7-20 arg1.c programı.

```
#include <stdio.h>
#include <string.h>
int main( int argc, char **argv )  /*  char *argv[ ]  */
{
  int i;
  printf("argc : %d\n", argc );
  for ( i = 0; i <= argc; i++ )
       printf("argv[%d] : %12s, %2u karakter\n",  i, argv[ i ], strlen( argv[i] ) );
  return 0;
}
```

...Örnek 5.7-20 devam

Bilgi Girişi

Komut satiri (DOS):

```
C:\>arg arg1 arg2 <enter>
```

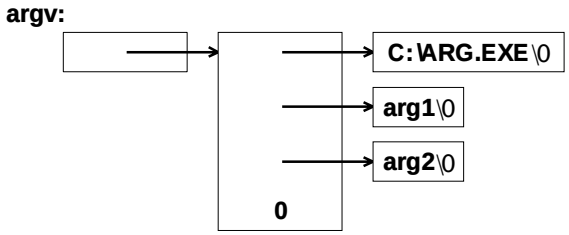
Çıktı

```
argc    : 3
argv[0] :C:\ARG.EXE,   10 karakter
argv[1] :        arg1,  4 karakter
argv[2] :        arg2,  4 karakter
argv[3  :      (null),  0 karakter
```

NOT:
Program DOS ortamında derlenmiştir. Dolayısıyla farklı ortamlarda argv[0], komutun bulunduğu fihrist bilgisini içermeyebilir (örneğin UNIX'te).

---

main fonksiyonun ilk parametresi olan **int** tipindeki **argc** değişkeni dizgilerin sayısını (bu örnekte; 3) verir. **İ**kinci parametre argv ise dizgilerin adres dizisine işaret eden çift adres değişkenidir. Dolayısıyla bildirimi **char** **argv şeklinde yapılabilir. Fakat [ ] kullanımı, adres değişkeninin bir diziye işaret ettiğini açık olarak belirttiği için tercih edilebilir. Adres dizisi argv'nin elemanları **char *** tipindeki adres değişkenleridir ve karakter dizgilerine işaret ederler. Örnek programda adres dizisi "C:\ARG.EXE", "arg1" ve "arg2" dizgilerinin başlangıç elemanlarının bellek adresleri ve son eleman olarakta boş adres değerinden (NULL) oluşur.



Yukarıdaki örnekte komut satırına iki argüman girilmiştir: arg1 ve arg2 dizgileri. argv[1] ve argv[2] adres değişkenleri ile erişilen bu argümanlar, '\0' ile sonlanmış karakter dizgileridir.

Adres dizisinin başlangıç elemanı argv[0], işletim sistemine bağlı olarak sürücü ismini, programın ve bulunduğu fihristin ismini içerir (DOS altında). argv[argc] ise NULL değer taşıyan boş adres değişkenidir ve hiçbir veriye işaret etmez. Boş adres değeri, adres dizisinin sonunu işaretler. Programlar içinde genellikle argv[0] ve argv[argc] kullanılmaz. Komut isminden sonra hiç bir argüman girilmediğinde argc'nin değeri 1'dir.

Sonuç olarak adres dizisinin argv[1] ifadesi ile erişilen ikinci dizgisi, komut satırına program isminden sonra girilen ilk argümandır. Son argümana ise argv[argc-1] ifadesi ile erişilir.

Komut satırı argümanlarına erişim tekniklerini göstermek için arg2.c ve arg3.c programlarında çeşitli **while** ve **for** döngülerine yer verilmiştir. Bölüm 9'da bu teknikler kullanılarak program içine komut satırından çeşitli seçeneklerin alınması ile ilgili örneklere yer verilmiştir. ■

Örnek 5.7-21 arg2.c programı.

```
#include <stdio.h>
#include <string.h>

int main( int argc, char **argv )
{
  /*
   *    av ve ac degiskenleri, parametrelerin ilk degerlerinin korunmasi icindir.
   */
  int i, ac = argc;
  char **av = argv;
  char *p;

  puts( "while - 1" );
  /*
   *   dizgilerin listelenmesi :
   *   *argv her artirimdan sonra dizgilerin baslangic adreslerini ve son olarakta  donguden
   *   cikilmasini saglayan bos adres degerini (NULL) verir.
   */
  while ( *++argv )
        puts( *argv );

  puts( "while - 2" );
  /*   while  dongusunde  *++argv  ifadesi,  dizgilerin  baslangic  elemanlarinin  bellek
   *   adreslerini verir. Bu adresler kullanilarak erisilen  dizgi boyunca  ilerlenerek karakterler
   *   listelenebilir.
   *   Yukaridaki  dongu  argv'nin  degerini degistirdigi icin orijinal  deger gereklidir.
   */
  argv = av;
  while ( *++argv )    /*    while ( *++argv != NULL )    */
  {
        printf( "argv[...] --> %u\n", argv[0] );
        for ( p = argv[0]; *p; p++ )
           /*
            *    asagidaki for donguleri de kullanilabilir:
            *    for ( p = argv[0]; *p != '\0'; p++ )
            *              yada
            *    for ( p = argv[0]; *p++; )
            */
        printf( "\t%u adresindeki karakter --> %c\n", p, *p );
        putchar( '\n' );
  }
```

...Örnek 5.7-21 devam

```
puts( "while - 3" );
argv = av;
while ( *++argv )
{
        printf( "argv[...] --> %u\n", argv[0] );
        for ( i = 0; *(argv[0]+i); i++ )
        {
              /*
               *     for ( i = 0; argv[0][i]; i++ )
               */
              printf( "\t%u adresindeki karakter --> ", argv[0] + i );
              putchar( *(argv[0] + i) );
              putchar( '\n' );
             /*
              *   putchar( argv[0][i] );
              */
        }
        putchar( '\n' );
}
puts( "for - 1" );
/*  argv + i yada &argv[ i ] ifadeleri adres dizisinin elemanlarinin bellek adreslerini;
 *  *(argv + i), &argv[ i ][ 0 ] yada argv[ i ] ifadeleri ise dizgilerin baslangic elemanlarinin
 *  bellek adreslerini (yani adres dizisinin elemanlarinin degerlerini) verir:
 */

argv = av;
for ( i = 0; i <= argc; i++ )
     printf( "%s dizgisi %u karakter\n", *(argv + i), strlen( argv[i] ) );

puts( "for - 2" );
/*  argv, adres dizisinin baslangic elemanina (&argv[0]) isaret eden cift adres
 *  degiskenidir. Dolayisiyla argv+i yerine ++argv ifadesi ile argv'nin adres dizisinin bir
 *  sonraki elemanina isaret etmesi saglanir. Fakat  argv+i  kullanildiginda argv'nin
 *  orijinal degeri degismez. argv adres degiskenine ++ isleminden sonra  * operatoru
 *  uygulandiginda  isaret  ettigi  adres  dizisi elemaninin degerini (dizginin baslangic
 *  adresi) verecektir.
 */
for ( i = 0; i <= argc; i++, argv++ )
{
     printf( "%s dizgisi %u karakter\n", *argv, strlen( *argv ) );
       /*
        *   for dongusu yerine buraya yerlestirilebilir:
        *   argv++;
        */
}
puts( "while - 5" );
argv = av;
while ( --argc > 0 )
        puts( *++argv );
```

...Örnek 5.7-21 devam

```
puts( "while - 6" );
argv = av;
argc = ac;
while ( --argc > 0 )
{
/*
 *  **++argv ifadesi karakter dizilerinin (dizgilerin) baslangic adreslerini verdigine  gore
 *  baslangic elemanina erisim icin **++argv  yerine  (*++argv)[0]  ifadesi kullanilabilir.
 *  Yani  **++argv  ==  *(*++argv + 0)  ==  (*++argv)[0]
 *  Indeks operatoru ([]),  oncelik  listesinde  indirek  deger operatorunden (*) once
 *  bulunur.  Dolayisiyla ifadenin *++(argv[0])  seklinde  yorumlanmasini  onlemek
 *  icin  parantezler kullanilir.  ++ uygulanmadigi zaman asagidaki ifadeler aynidir:
 *        **argv,  (*argv)[i],  *(argv[i])  yada  *argv[i]
 */
    putchar( (*++argv)[0] );
    putchar( '\n' );
}
puts( "while - 7" );

argv = av;
argc = ac;
while ( --argc > 0 )
{
/*
 *  argv cift adres degiskenidir. Iki  kez  *  uygulandiginda isaret ettigi dizginin baslangic
 *  elemani olan char veriyi verir.
 */
  putchar( **++argv );
  putchar( '\n' );
}
puts( "while - 4" );
/*
 *  adres dizisinin elemanlari olan argv[i] adres degiskenlerinin tasidigi adres degerlerini
 *  degistirdigi icin bu uygulama program sonuna yerlestirilmistir.
 */
argv = av;
while ( *++argv )         /*  != NULL  */
{
       while ( *argv[0] )  /*  != '\0'  */
       {
          /*
           * komut satirina girilen arguman dizgilerde bulunan karakterlerin listelenmesi.
           */
          putchar( *argv[0] );
          ++argv[0];
       }
       putchar( '\n' );
}
```

...Örnek 5.7-21 devam

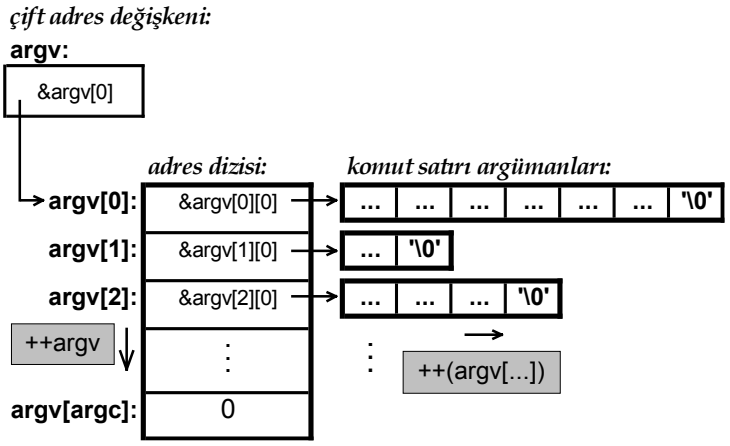
```
   /*
    *    argv      : char ** tipinde adres degiskeni.
    *    *argv     : char *  tipinde adres degiskeni.
    *    **argv    : char tipi degisken.
    */
   printf( "sizeof argv : %u, sizeof *argv : %u, sizeof **argv : %u\n",
              sizeof argv, sizeof *argv, sizeof **argv );
   return 0;
}
```

Çıktı

```
while - 1
123
abc
while - 2
argv[...] --> 3760
   3760 adresindeki karakter --> 1

   3761 adresindeki karakter --> 2
   3762 adresindeki karakter --> 3
argv[...] --> 3764
   3764 adresindeki karakter --> a
   3765 adresindeki karakter --> b
   3766 adresindeki karakter --> c
while - 3
argv[...] --> 3760
   3760 adresindeki karakter --> 1

   3761 adresindeki karakter --> 2
   3762 adresindeki karakter --> 3
argv[...] --> 3764
   3764 adresindeki karakter --> a
   3765 adresindeki karakter --> b
   3766 adresindeki karakter --> c
for - 1
C:\ARG2.EXE dizgisi 11 karakter
123    dizgisi 3 karakter
abc    dizgisi 3 karakter
(null) dizgisi 0 karakter
for - 2
C:\ARG2.EXE dizgisi 11 karakter
123    dizgisi 3 karakter
abc    dizgisi 3 karakter
(null) dizgisi 0 karakter
while - 5
123
```

...Örnek 5.7-21 Çıktı devam

```
abc
while - 6
1
a
while - 7
1
a
while - 4
123
abc
sizeof argv : 2, sizeof *argv : 2, sizeof **argv : 1
```



++(argv[...]) ifadesi dizgilerde ilerler;**char *** tipinde adres değeri döndürür.
++argv ifadesi adres dizisinde ilerler; **char **** tipinde adres değeri döndürür.

Örnek 5.7-22 arg3.c programı.

```
#include <stdio.h>
#include <string.h>

int main( int argc, char **argv )
{
   int i, j;
      /*      Asagidaki ifadeler ayni degeri verir:
                i. dizginin baslangic elemani     :  *argv[ i ]  yada  argv[ i ][ 0 ]
                0.dizginin i.elemaninin adresi:   argv[0] + i yada  *argv + i
       */
```

...Örnek 5.7-22 devam

```
   for ( i = 0; argv[i]; i++ )
   {
        for ( j = 0; argv[i][j]; j++ )
             putchar ( argv[ i ][ j ] );          /* 2-boyutlu dizi olarak listeleme */
        putchar( '\n' );
   }
   for ( i = 0; *(argv+i); i++ )
   {
        for ( j = 0; *(*(argv+i)+j); j++ )
             putchar (  *( *(argv+i) + j)  ); /* adres aritmetigi kullanarak listeleme */
                /*
                 *       (*(argv+i))[ j ] yada *(argv[ i ] + j)  deneyiniz !!!
                 */
        putchar( '\n' );
   }

   while ( --argc > 0 )
   {
          putchar( (*++argv)[0] );
   /*
    *  (*++argv)[ 0 ] == *( *++argv + 0)  == **++argv
    *  Bu ifadede cift adres degiskeni argv'nin degeri artirildigi icin adres dizisinde
    *  ilerlenir. argv, adres dizisi baslangic elemanina, ++argv ise izleyen elemanina
    *  isaret eder. * yada [ ] uygulandiginda isaret edilen dizginin baslangic elemaninin
    *  adresini verir (adres dizisi elemaninin degeri).
    *  Tekrar * uygulandiginda bu adresteki char veri elde edilir. Dolayisiyla bu ifade
    *  daima dizginin baslangic elemanini verir.
    */
        putchar( *++(argv[0]) );   /*  *++argv[0]   */
   /*
    *  Bu ifadede adres dizisi elemani olan ve dizgilerin baslangicina isaret eden
    *  adres degiskeni argv[0]'in degeri artirildigi icin dizgide ilerlenir. argv adres dizisine
    *  isaret eder. Onceki putchar cagrisinda argv artirildigi icin argv[0] artik adres
    *  dizisinin baslangic elemani degildir. Bir baska dizginin baslangic adresidir ve ++
    *  uygulandiginda bu dizginin izleyen karakterine isaret eder. * uygulandiginda bu
    *  karakteri verir.
    */
   }
   return 0;
}
```

Çıktı

```
C:\ARG3.EXE
123
abc
C:\ARG3.EXE
123
abc
12ab
```

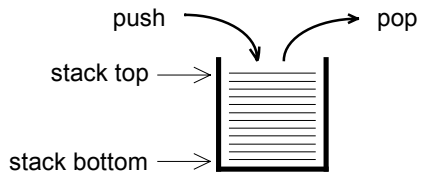
5.8 Değişen Sayıda Argüman Alan Fonksiyonlar

C dilinde, *değişen sayıda argüman alan* (her çağrıda farklı sayıda ve farklı tiplerde argüman alabilecek) *fonksiyon* oluşturmak mümkündür (*variable-length argument list function*). Örnek programlarda kullanılan scanf ve printf, bu şekilde çalışan standart kütüphane fonksiyonlarıdır.

C dilinde bir fonksiyon çağrıldığı zaman, argümanlar programın *stack* alanına çağrıda yer aldıkları sıranın tersine yani sağdan sola doğru aktarılır. Buna göre *C fonksiyon çağırma tekniği*'nde (*C calling convention*) son argüman stack alanına ilk olarak, ilk argüman ise en son yerleştirilir. Dolayısıyla ilk argüman her fonksiyon çağrısında aynı sabit offset'te bulunur. Çağrılan fonksiyon ilk argümanın bu sabit stack pozisyonunu esas alır ve argüman listesinde ilerleyerek argümanlara erişir.

NOT:

Bir C fonksiyonu çağrıldığında, çağrı satırında yer alan argümanların değerleri programın *stack* alanına kopyalanır. Stack son giren ilk çıkar (LIFO, Last In First Out) şeklinde çalışan bir veri yapısıdır; stack'a eklenen bir değer stack tavanına (*stack top*) yerleştirilir (*push*) ve böylece stack genişlemiş olur.
Yada stack'tan tavanda bulunan bir veri çıkarılabilir (*pop*). Bu durumda da stack küçülmüş olur.



Programın stack alanı ise bu şekilde yönetilen ve fonksiyon çağrıları sırasında kullanılan geçici verileri saklamak için ayrılmış ve çağrıdan dönüldüğünde boşaltılan bellek alanıdır. Donanıma ve derleyiciye göre değişmekle birlikte, çağrı sırasında stack işlemleri pek çok derleyici tarafından aşağıdaki şekilde gerçekleştirilir:

CPU'nun *stack pointer register*'i (SP), stack'ın tavan pozisyonunu izler (en son veri eklenen pozisyon). Stack azalan bellek adresi yönünde (aşağı doğru) büyür. Yani bir fonksiyon diğerini çağırdığında, stack daha çok bellek alanı kullanır ve SP'nin değeri azalır. Çağrıdan dönüldüğünde, stack küçülür ve SP'nin değeri artar.

C fonksiyon çağırma tekniği'nde (*C calling convention*), argümanlar stack alanına çağrıda yer aldıkları sıranın tersine yani sağdan sola doğru aktarılır. Buna göre son argüman ilk olarak, ilk argüman ise en son yerleştirilir. İlk argüman sonrasına ise fonksiyon çalışmasını tamamladıktan sonra kontrolün aktarılacağı *dönüş adresi* (*return address*) yerleştirilir. Aşağıdaki şekilde de görüldüğü gibi C çağırma tekniği'nde, ilk argüman daima stack'ta SP'ye göre belli bir sabit offset'te

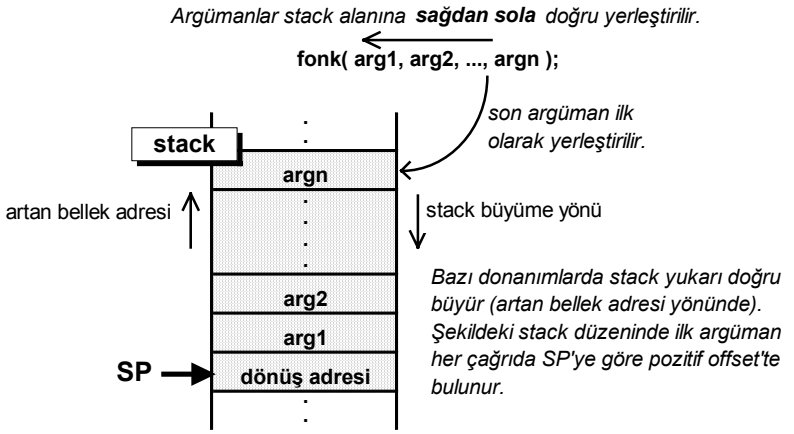
ve dönüş adresi öncesinde bulunur (bu offset, dönüş adresini saklamak için gerekli olan byte sayısına eşittir).

Çaðrýlan fonksiyon, daima aynı offset'te bulunduðu için (aktarýlan argüman sayýsýndan baðýmsýz olarak) ilk argümana ve dolayýsýyla ilk argümanýn stack adresini esas alarak izleyen argümanlara eriþebilir.

Diðer bazý programlama dillerinde argümanlar stack alanına programda yer aldıkları sırada (soldan sağa doğru) yerleştirilir ve bu durumda çağrılan fonksiyon ilk argümanın yerini belirleyemeyeceği için, değişen sayıda argüman listesine sahip fonksiyon oluşturulamaz. Bazı C derleyicileri diğer dillerdeki çağırma tekniklerinin de kullanılabilmesine olanak sağlar.

C çağırma tekniği'nde, çağrılan fonksiyon aktarılacak argüman sayısını bilmez. Dolayısıyla çağıran fonksiyon, stack'a yerleştirilen argüman sayısını bildiği için SP'yi ayarlayarak stack'ı boşaltır. Bu durum programın hızı ve oluşturulan kod uzunluğu açısından bir dezavantajdır. Çünkü programın farklı yerlerindeki her çağrının stack alanını boşaltan kodları içermesi gerekir.

Şekil 5.8-1 C çağırma tekniği'nde, fonksiyona giriş sırasında stack görünümü.



Fonksiyon argümanlarının yerleştirildiği stack düzeni donanıma ve derleyiciye göre değişebilir ve ayrıca derleyicilerin fonksiyon argümanlarına erişim teknikleri farklı olabilir. Bu nedenle değişen sayıda argümanların değerlerine erişmek için taşınabilir bir tekniğe ihtiyaç vardır. ANSI C'de stdarg.h başlık dosyası değişen sayıda argüman alan fonksiyon oluşturmak için kullanılabilecek standart makro tanımları içerir. Aşağıda listelenen bu makroların kullanılması, donanıma ve derleyiciye göre farklılık gösterebilecek iç yapıları gizleyerek basit ve taşınabilir bir teknik sağlar:

```
va_start(  va_list arguman_gostergesi, son_arguman );   /* stdarg.h */
va_arg(    va_list arguman_gostergesi, tip );           /* stdarg.h */
va_end(    va_list arguman_gostergesi );                /* stdarg.h */
```

Yukarıdaki makrolar kullanılarak değişen sayıda argüman alan fonksiyon aşağıdaki şekilde oluşturulur:

- Fonksiyon bildirim ve tanımında parantezler içinde sabit argümanlar ve bunları izleyen üç nokta (...) bulunur. Üç nokta derleyiciye bu fonksiyonun sabit argümanlardan başka sayıları ve tipleri belli olmayan (0 yada daha fazla) argüman alabileceğini belirtir. Bu argümanların tipleri ve sayıları belli olmadığı için derleyici tip kontrolu yapamaz.

- İlk olarak fonksiyon içinde, stdarg.h'de tanımlanmış olan va_list tipinde bir argüman göstergesi (*arguman_gostergesi*) bildirilir:

  va_list *arguman_gostergesi*;

  Bu değişken, listede bulunan argümanlara işaret eden bir basit adres değişkenidir (*argument pointer*).

  arguman_gostergesi'nin değer atama işlemi ilk olarak va_start makrosu tarafından yapılır. Daha sonra bu değişken, izleyen argümanlara erişmek için va_arg tarafından kullanılır.

- Fonksiyonun argüman listesi en az bir sabit argüman (her çağrıda aynı olan) ile başlamalıdır. Çünkü kullanılan makrolar, listede yer alan diğer argümanların yerlerini bu argümanı esas alarak saptar. va_start makrosu çalıştırılarak *arguman_gostergesi*'nin verilen sabit argümanı (fonksiyonun bildirilen son sabit argümanı olan *son_arguman*) izleyen ilk argümana işaret etmesi sağlanır. *son_arguman*, **register** saklama sınıfına sahip olamaz.

  va_start(*arguman_gostergesi*, *son_arguman*);

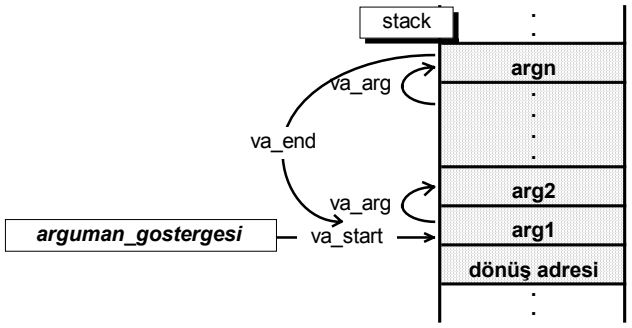
  Bundan sonra *arguman_gostergesi* ilk parametresi olan va_arg çalıştırılarak listedeki argümanlara erişilir.

- Her bir argümana erişmek için va_arg makrosu çalıştırılır. İlk parametresi olan *arguman_gostergesi*'nin işaret ettiği argümanın değerini veren bu makronun ikinci parametresi sözkonusu argümanın tipidir. *arguman_gostergesi*, her va_arg sonrası izleyen argümana işaret eder.

  Bu makro, değişen sayıda argüman listesindeki argüman sayısı kadar çalıştırılabilir. Çağrılan fonksiyon aktarılan argüman sayısını bilmediği için, herhangi bir yolla va_arg makrosunun kaç kez çalıştırılacağı saptanmalıdır. Bunun için çeşitli yollar mevcuttur: eğer liste aynı tipte argümanlardan oluşuyor ise liste sonu herhangi bir değer ile işaretlenebilir (tamsayılar için -1, dizgi argümanlar için uygun tip dönüşümü yapılarak 0 yada NULL gibi); yada bir sabit argüman ile bu sayı aktarılabilir. Hem argümanların sayısını hem de tiplerini belirtmek için printf fonksiyonunda olduğu gibi sabit argüman olarak bir format dizgisi aktarılabilir. Bir başka teknik ise, standart C kütüphanesinde bulunan ve daha sonra açıklanacak olan vprintf gibi bir başka değişen sayıda argüman alan fonksiyonun kullanılmasıdır.

- Argüman listesi tarandıktan sonra son olarak va_end makrosu çalıştırılır. Bu makro fonksiyondan dönülmeden önce *arguman_gostergesi*'ni sıfırlar. va_end sonrasında tekrar va_start çalıştırılarak liste baştan taranabilir. Liste sonuna gelmeden önce yine va_end ve va_start çalıştırılarak liste tekrar baştan taranabilir.

Şekil 5.8-2 va_start, va_arg ve va_end makrolarının kullanımı.



Aşağıdaki örnek programda yer alan her iki fonksiyon da aynı işi yapar. Üç sabit argüman alan ve bunları ekrana listeleyen SabitArg fonksiyonu, aynı işi yapan değişen sayıda argüman alan bir fonksiyon olarak tanımlanır.

Örnek 5.8-1 sabitarg.c programı.

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

void SabitArg(int i, long l, char *s );
void DegisArg(  int i, ... );

int main(void)
{
  SabitArg(    9,19L,   "DENEME"  );
  DegisArg(9,19L,   "DENEME"  );

  return 0;
}

void SabitArg( int i, long l, char *s )
{
  puts("- SabitArg -");
  printf("i : %d, l : %ld, s : %s\n", i, l, s );
}

void DegisArg( int i, ... )
{
  va_list AdrArg;

  puts("- DegisArg -");

  va_start( AdrArg, i );

  printf("i : %d, ",  i );
  printf("l : %ld, ",     va_arg( AdrArg, long   ) );
  printf("s : %s\n ",   va_arg( AdrArg, char * ) );

  va_end( AdrArg );
}
```

Çıktı

```
- SabitArg -
i : 9, l : 19, s : DENEME
- DegisArg -
i : 9, l : 19, s : DENEME
```

DegisArg fonksiyonunun prototipinde ve tanımında **int** tipi sabit parametre i ve bunu izleyen üç nokta bulunur. Program makroların kullanımını örneklemek amacıyla geliştirildiği için, DegisArg fonksiyonuna argüman sayısını ve tiplerini aktarmak için herhangi bir teknik içermemektedir. İlk olarak va_list tipinde AdrArg değişkeni bildirilir. Argüman göstergesi AdrArg ve sabit argüman i kullanılarak va_start makrosu çalıştırılır. Böylece AdrArg'nin değişen sayıda argüman listesinde yer alan ve i değişkenini izleyen ilk argümana işaret etmesi sağlanır. Bundan sonra iki kez çalıştırılan

va_arg makrosu ilk olarak ikinci parametresi olan **long** tipinde 19 değerini ve daha sonra da "DENEME" dizgisi başlangıcına işaret eden **char *** tipindeki adres değerini verir. Fonksiyondan dönülmeden önce va_end çalıştırılır.

Değişen sayıda argüman alan fonksiyon oluştururken argümanlar fonksiyon çağrısı sırasında genişleyen (*promoting*) veri tipinde, dizi tipi yada fonksiyon tipinde olamaz. Dolayısıyla **float** tipler **double** ile; **char** (**signed** yada **unsigned**) gibi genişleyen tamsayı tipler ise **int** (**signed** yada **unsigned**) ile değiştirilmelidir.
Ayrıca listede adres değeri olarak sadece ***** eklenerek adres tipine çevrilebilen (**int** yada **char** gibi) veri tipleri kullanılabilir. Örneğin **int*[10]** kullanılamaz.

NOT:

UNIX System V'de kullanılan makrolar varargs.h'de bulunur. Aşağıda UNIX System V'de değişen sayıda argüman listesi alan bir fonksiyon örneği verilmiştir:

```
/*
 * varunix.c
 */
#include <stdio.h>
#include <varargs.h>

void PrntDizgi();

int main(void)
{
   PrntDizgi( "abc", "def", "xyz", (char *)NULL );
   return 0;
}
void PrntDizgi(va_alist)
va_dcl
{
   char *s;
   va_list AdrArg;

   va_start(AdrArg);

   while ( s = va_arg( AdrArg, char *) )
         puts( s );

   va_end( AdrArg );
}
/*
 * Cikti:
         abc
         def
         xyz
 */
```

Örnek 5.8-2'de değişen sayıda argüman alan PrintDizgi, PrintTams, PrintXYf, DizgiEkle ve HataYaz fonksiyonları tanımlanmıştır.

Örnek 5.8-2 degisarg.c programı.

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

void   PrintDizgi(   char *Format, ... );
void   PrintTams(  int  Sayi, ... );
void   PrintXYf(     int  Satir, int Kolon, char *Format, ... );
char   *DizgiEkle(  char *Dizgi, ... );
void   HataYaz(     char *Format, ... );

char Dizgi[ BUFSIZ ];

int main(void)
{
   Dizgi[0] = '\0';

   PrintDizgi( "1-sN2-sN3-sN", "abc", "def", "xyz" );
   PrintDizgi( "1-s,T2-sN", "bir", "iki" );

   PrintTams ( 3, 1, 2, 3 );
   PrintTams ( 2, 1, 2 );

   PrintXYf  ( 10, 40, "%s : %d\n", "Tamsayinin degeri", 34 );

   puts( DizgiEkle( Dizgi, "Bu ", "bir ", "denemedir. \n", (char *)NULL ) );

   HataYaz( "%s dosyasi bulunamadi", "DENEME" );

   return 0;
}
void PrintDizgi( char *Format, ... )
{
   va_list AdrArg;

   puts("- PrintDizgi -");

   va_start( AdrArg, Format );

   for ( ;*Format; ++Format )
        switch (*Format)
        {
                case 's':
                        printf("%s", va_arg( AdrArg, char * ) );
                        break;

                case 'N':
                        putchar( '\n' );
                        break;

                case 'T':
                        putchar( '\t' );
```

... Örnek 5.8-2 devam

```
                    break;
                default :
                    putchar( *Format );
            }
    va_end( AdrArg );
}
void PrintTams( int Sayi, ... )
{
    int i;
    va_list AdrArg;
    puts("- PrintTams -");
    printf("ekrana yazilan tamsayi sayisi : %d\n", Sayi );
    va_start( AdrArg, Sayi );
    for ( i = 0; i < Sayi; i++ )
        printf( "%d. sayi : %d\n", i+1, va_arg( AdrArg, int ) );
    va_end( AdrArg );
}
void PrintXYf( int  Satir, int Kolon, char *Format, ... )
{
    va_list AdrArg;
    puts( "- PrintXYf -" );
    if ( strlen(Format) > 0 )
    {
        printf("Kursor Pozisyonu Satir : %d, Kolon : %d\n", Satir, Kolon );
        va_start( AdrArg, Format );
        vprintf( Format, AdrArg );
        va_end( AdrArg );
    }
}
char *DizgiEkle( char *Dizgi, ... )
{
    char *s;

    va_list AdrArg;
    puts("- DizgiEkle -");
    va_start( AdrArg, Dizgi );
    while (  ( s = va_arg( AdrArg, char * ) )
                != NULL )
            strcat( Dizgi, s );
    va_end( AdrArg );
    return Dizgi;
}
```

...Örnek 5.8-2 devam

```
void HataYaz( char *Format, ... )
{
  va_list AdrArg;
  puts("- HataYaz -");
  va_start( AdrArg, Format );
  fprintf(  stderr, "Hata: " );
  vfprintf( stderr, Format, AdrArg );
  fprintf(  stderr, "\n" );
  va_end( AdrArg );
}
```

Çıktı

```
- PrintDizgi -
1-abc
2-def
3-xyz
- PrintDizgi -
1-bir,  2-iki
- PrintTams -
ekrana yazilan tamsayi sayisi : 3
1. sayi : 1
2. sayi : 2
3. sayi : 3
- PrintTams -
ekrana yazilan tamsayi sayisi : 2
1. sayi : 1
2. sayi : 2
- PrintXYf -
Kursor Pozisyonu Satir : 10, Kolon : 40

Tamsayinin degeri : 34
- DizgiEkle -
Bu bir denemedir.
- HataYaz -
Hata: DENEME dosyasi bulunamadi
```

Değişen sayıda dizgi argümanı ekrana yazan PrintDizgi fonksiyonunun sabit argümanı bir format dizgisidir. Format dizgisi, format belirleyici karakterler olarak dizgiler için s harfini, NL (*newline*) karakteri için N harfini ve tab boşluğu için de T harfini içerir. PrintDizgi çağrılarında format dizgisinde bulunan karakterler birer birer ekrana yazılır. Bu arada format dizgisinde s karakterine rastlandığında ise argüman listesinden bir dizgi va_arg makrosu ile alınarak ekrana yazılır. N harfine rastlandığında ekrana '\n', T karakterine rastlandığında ise ekrana '\t' yazılır.

PrintTams fonksiyonunun ilk argümanı, argüman listesinde yer alan tamsayı değerlerin sayısını aktarır. DizgiEkle fonksiyonu ise, NULL ile sonlandırılmış olan değişen sayıda argüman listesinde bulunan dizgileri, ilk argümanı olan bellek adresine strcat fonksiyonunu kullanarak peşpeşe yerleştirir ve yine bu bellek adresini döndürür. puts fonksiyonu DizgiEkle çağrısından dönen değer olan bellek adresini argüman olarak alır ve bu adreste bulunan dizgiyi ekrana yazar.

PrintXYf ve HataYaz fonksiyonları ise yine değişen sayıda argüman alan vprintf ve vfprintf'i çağırır. Standart C kütüphanesinde bulunan vfprintf, vprintf ve vsprintf fonksiyonları fprintf, printf ve sprintf fonksiyonlarına benzer:

```
int vfprintf( FILE *dosya_gostergesi, const char *Format, va_list AdrArg );   /* stdio.h */

                        int vprintf( const char *Format, va_list AdrArg );   /* stdio.h */

              int int vsprintf( char *Dizgi, const char *Format, va_list AdrArg );   /* stdio.h */
```

Ancak bu fonksiyonlar argüman listesi yerine argüman listesine işaret eden bir adres değeri alırlar (va_list tipinde bildirilen *AdrArg*'nin değeri).

vfprintf fonksiyonu argüman listesinde bulunan değerleri *Format* ile belirtilen şekilde formatlanmış "text" olarak *dosya_gostergesi* ile ifade edilen akıma (bir disk dosyasına ait akım, stdout yada stderr); vprintf fonksiyonu, standart çıkış'a (stdout); vsprintf fonksiyonu ise başlangıç adresi *Dizgi* ile belirtilen bellek alanına yerleştirir (vsprintf, son karakter olarak olarak boş karakter ekler). *Format*, printf fonksiyonunun format dizgisi ile aynı şekilde oluşturulur ve aynı işleve sahiptir. vfprintf ve vprintf fonksiyonları başarılı ise oluşan karakter sayısını, hata durumunda ise (fonksiyon, akımın hata göstergesini değiştirmiş ise) negatif değer döndürür. vsprintf ise boş karakter (*null character*, '\0') hariç *Dizgi*'ye yazılan karakter sayısını döndürür. Yine bu fonksiyon da çıktı hatası durumunda negatif değer döndürür. Programda bu fonksiyonlar çağrılmadan önce ilk olarak va_start makrosu çalıştırılmalıdır. Çağrı sonrası ise va_end makrosu çalıştırılmalıdır.

PrintXYf fonksiyonu 3 sabit argümana sahiptir. İlk iki tamsayı argümanı, format dizgisi ve değişen sayıda argüman listesi izler. Bu fonksiyon içinde vprintf fonksiyonu, format dizgisini ve argüman göstergesi olan AdrArg ile çağrılır. PrintXYf fonksiyonu, printf fonksiyonunun karakterleri belirtilen satır ve kolona yazan versiyonu haline getirilebilir. HataYaz fonksiyonunun ilk argümanı format dizgisidir. Bunu değişen sayıda argüman listesi izler. İlk olarak fprintf çağrısı standart hata çıkışı'na "Hata: " dizgisini yazar. Format dizgisi (printf format dizgisi ile aynı şekilde çalışan) ve argüman göstergesi AdrArg (va_start makrosu çalıştırılarak değişen argüman listesine işaret etmesi sağlanan) ile çağrılan vfprintf fonksiyonu listede bulunan argümanları format dizgisinde belirtilen formata göre ekrana yazar. Son olarak ekrana NL karakteri gönderilir.

Bu bölümde, değişen sayıda argüman alabilen fonksiyonların oluşturulması ve kullanımı anlatıldı. Derleyicinin prototip dışında kalan parametreler için tip ve sayı kontrolü yapamayacağı göz önüne alınarak bu uygulamalar çok dikkatli gerçekleştirilmelidir. ■

5.9 Kendini Çağıran Fonksiyonlar

C dilinde, bir fonksiyon kendisini direk yada indirek olarak çağırabilir. Bu şekilde çalışan fonksiyonlar, *özyineli* (*recursive*) fonksiyonlar olarak adlandırılır. Direk çağrıda, kendini çağıran tek bir fonksiyon yer alır. İndirek çağrıda ise, fonksiyon kendisini bir başka fonksiyon üzerinden çağırır (Örnek olarak, f1 fonksiyonu f2 fonksiyonunu çağırır ve f2 fonksiyonu da tekrar f1 fonksiyonunu çağırır). Bu bölümde, direk olarak gerçekleşen özyineli çağrılara yer verilmiştir. Aşağıdaki programda, ekrandan girilen bir tamsayının faktöriyeli hesaplanır.

Örnek 5.9-1 fakt1.c programı.

```
#include <stdio.h>

int main(void)
{
  long fakt = 1, sayac, sayi;

  printf("bir tamsayi giriniz : ");
  scanf("%ld", &sayi );

  for ( sayac = 1; sayac <= sayi; sayac++ )
      fakt = fakt * sayac;

  printf("%ld !   :  %ld\n", sayi, fakt );

  return 0;
}
```

Bilgi Girişi

```
bir tamsayi giriniz : 3
```

Çıktı

```
3 !   :  6
```

Program ekrandan girilen 3 değeri için 6 sonucunu verir. n sayısının faktöriyeli, n! şeklinde yazılır ("n faktöriyel" olarak okunur) ve n dahil n'e kadar tüm pozitif tamsayıların çarpımına eşittir. Örnek olarak;

```
n! == 1 x 2 x ... x n
3! == 3 x 2 x 1,   yani 6
4! == 4 x 3 x 2 x 1,  yani 24
```

Faktöriyel fonksiyonu için aşağıdaki özyineli tanım mevcuttur:

$$n! = \begin{cases} 1, & \text{eğer } n <= 1 \text{ ise} \\ n * (n - 1)!, & \text{diğer durumlarda} \end{cases}$$

Örnek 5.9-2 fakt2.c programı.

```
#include <stdio.h>

long fakt( long );

int main(void)
{
   long sayi, sonuc;

   printf("bir tamsayi giriniz : ");
   scanf("%ld", &sayi );

   sonuc = fakt( sayi );
   printf("%ld !   :  %ld\n", sayi, sonuc );

   return 0;
}

long fakt( long n ) /* n! : n faktoriyel */
{
   long f;

   printf("cagri, n : %ld\n", n );
   f = ( n <= 1 ) ? 1L : n * fakt( n-1 );
   printf("donus, faktoriyel degeri : %ld\n", f );

   return f;
}
```

Bilgi Girişi

```
bir tamsayi giriniz : 3
```

Çıktı

```
cagri, n  : 3 ------------------------> fakt(3) çağrısı:  (3 <= 1) ? 1 : 3 * fakt(2)   (1)
cagri, n  : 2 ------------------------> fakt(2) çağrısı:  (2 <= 1) ? 1 : 2 * fakt(1)   (2)
cagri, n  : 1 ------------------------> fakt(1) çağrısı:  (1 <= 1) ? 1 : 3 * fakt(0)   (3)
donus, faktoriyel degeri : 1 ------> fakt(1) çağrısından dönüş   (4)
donus, faktoriyel degeri : 2 ------> fakt(2) çağrısından dönüş   (5)
donus, faktoriyel degeri : 6 ------> fakt(3) çağrısından dönüş   (6)
3 !   :  6
```

Örnek 5.9-2'de yer alan programda, yukarıdaki tanım kullanılarak faktöriyel hesaplama işlemi için kendini çağıran fakt fonksiyonu oluşturulur.

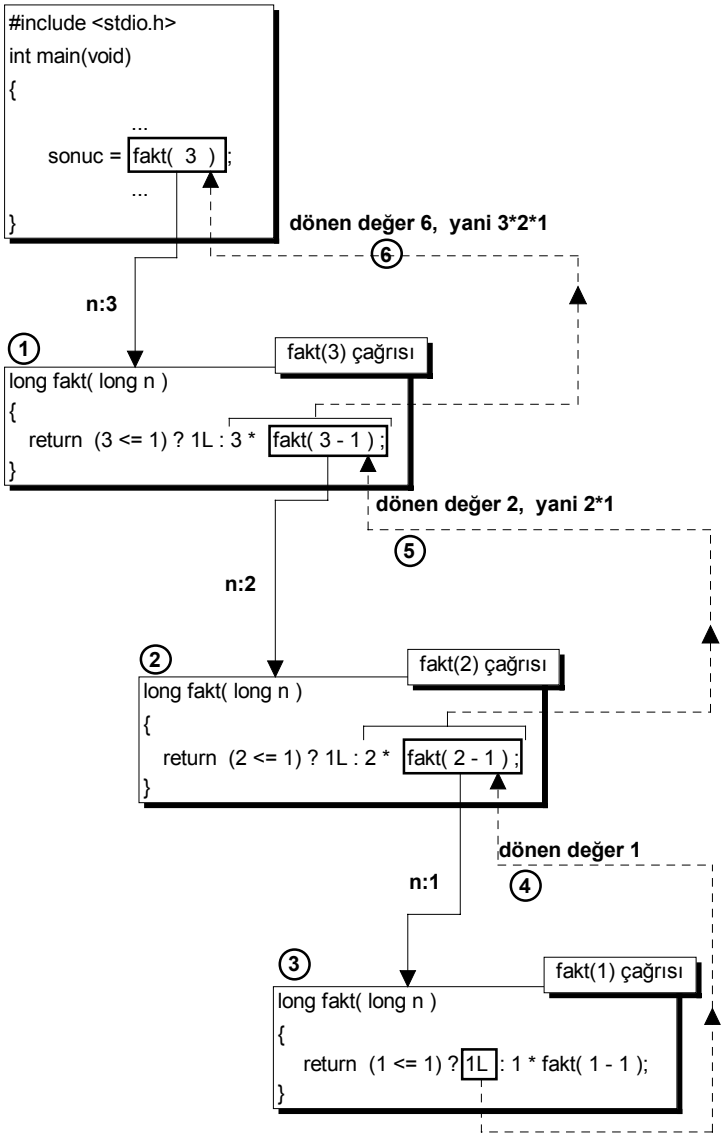
Bu program ekrandan 3 değeri girildiğinde yine aynı çıktıyı verir. fakt fonksiyonuna, çağrıları izlemek için iki ayrı printf satırı eklenmiştir. Çıktıda da görüldüğü gibi n'in değeri 1'den büyük olduğu zaman, fonksiyon fakt(n-1) ifadesi ile kendini çağırır.

Bu çağrı ile argüman olarak, n'in o andaki değerinin bir eksiği aktarılır. Bu sırada, çağıran fakt bloğundaki işlemler askıya alınır. İç içe çağrılar, n'in değeri 1 olana kadar devam eder ve son çağrı bu değer ile gerçekleştirilir. Son çağrı, çağıran bloğa 1 değerini döndürür ve diğer çağrılar da dönmeye başlar. Bu işlemler, Şekil 5.9-1'de şematik olarak gösterilmiştir.

Bir fonksiyonun kendini çağıracak şekilde yazılması, bellek tasarrufu sağlamaz ve programın hızını arttırmaz. Çünkü fonksiyonun üzerinde işlem yaptığı değerler için zaten stack alanı ayrılır ve stack işlemleri yapılır. Fakat bu şekilde yazılan fonksiyonların kodları daha kısadır ve okunabilirliği daha fazladır. Ayrıca bazı veri yapıları üzerinde işlemler yapmak için kod oluşturma sırasında çok kolaylık sağlarlar (ağaç veri yapısında olduğu gibi).

Bazı algoritmalar da özyineli olarak tanımlanabilir (hızlı-sıralama (*quick sort*) algoritması gibi). Fakat fonksiyon kendini bir kez çağırıyor ise bu işlemin özyineli bir tanım yerine tekrar ile yapılması daha uygundur. ■

Şekil 5.9-1 3!'in fakt fonksiyonu kullanılarak özyineli olarak hesaplanması.



BÖLÜM 6:
Adres Değişkenleri ve Çok-Boyutlu Diziler

C dilinde çok-boyutlu dizi (*multi-dimensional array*) "*elemanları dizi olan dizi*" 'dir. Bu bölümde çok-boyutlu dizilerin ve bu dizilerin elemanlarına işaret eden adres değişkenlerinin bildirilmesi ve çok-boyutlu dizilerin fonksiyonlara argüman olarak aktarılması anlatılacaktır. ■

6.1 Çok-Boyutlu Dizi Bildirimi

C dilinde çok-boyutlu dizi bildirimi, elemanları dizi olan bir dizinin bildirimidir. İstenen sayıda boyuta (*dimension*) sahip dizi bildirilebilir. Bildirim deyimine eklenen her indeks, diziye bir boyut daha ekler:

Dizi [*1.boyut***] [** *2.boyut***] [***3.boyut***] ... [***n.boyut***]**

Aşağıdaki bildirim deyiminde, 5 tamsayıdan oluşan tek-boyutlu tamsayı dizi iDizi tanımlanır:

```
int iDizi [ 5 ] = { 0, 1, 2, 3, 4 };
```

İki-boyutlu (*two-dimensional*; *2-d array*) iDizi2b ise aşağıdaki gibi tanımlanır:

```
int iDizi2b [ 4 ] [ 5 ] =
{
{ 0, 1, 2, 3, 4 },   /*  iDizi2b[0] dizisi  */
{ 5, 6, 7, 8, 9 },   /*  iDizi2b[1] dizisi  */
{ 2, 3, 4, 5, 6 },   /*  iDizi2b[2] dizisi  */
{ 7, 8, 9, 0, 1 }    /*  iDizi2b[3] dizisi  */
};
```

Üçüncü indeks eklenerek üç-boyutlu tamsayı dizi iDizi3b tanımlanabilir:

```
int iDizi3b [ 3 ] [ 4 ] [ 5 ] =
{
      /*  iDizi3b[0] dizisi   */
      {
          { 0, 1, 2, 3, 4 },          /*  iDizi3b[0][0] dizisi   */
          { 5, 6, 7, 8, 9 },          /*  iDizi3b[0][1] dizisi   */
          { 2, 3, 4, 5, 6 },          /*  iDizi3b[0][2] dizisi   */
          { 7, 8, 9, 0, 1 }           /*  iDizi3b[0][3] dizisi   */
      },
      /*  iDizi3b[1] dizisi   */
      {
          { 0, 1, 2, 3, 4 },          /*  iDizi3b[1][0] dizisi   */
          { 5, 6, 7, 8, 9 },          /*  iDizi3b[1][1] dizisi   */
          { 2, 3, 4, 5, 6 },          /*  iDizi3b[1][2] dizisi   */
          { 7, 8, 9, 0, 1 }           /*  iDizi3b[1][3] dizisi   */
      },
      /*  iDizi3b[2] dizisi   */
      {
          { 0, 1, 2, 3, 4 },          /*  iDizi3b[2][0] dizisi   */
          { 5, 6, 7, 8, 9 },          /*  iDizi3b[2][1] dizisi   */
          { 2, 3, 4, 5, 6 },          /*  iDizi3b[2][2] dizisi   */
          { 7, 8, 9, 0, 1 }           /*  iDizi3b[2][3] dizisi   */
      }
};
```

Eksiksiz olduğu için ilk değer listelerinde yer alan değerleri sınırlayan oklu parantezler kullanılmayabilir. Fakat parantezlerin kullanımı okunabilirliği artırır. İki-boyutlu (2-b) iDizi2b, "*5 tamsayıdan oluşan 4 diziden oluşan*" dizidir. Yani iDizi2b dizisi iDizi2b[0], iDizi2b[1], iDizi2b[2] ve iDizi2b[3] dizilerinden oluşur.

İki-boyutlu dizi, *satir* ve *sutun*'lardan oluşan bir tablo (matriks) olarak düşünülebilir. Dizide yer alan tamsayılara *satir* ve *sutun* indeksi (*row indeks* ve *column index*) kullanılarak iDizi2b [*satir*] [*sutun*] ifadesi ile erişilir. &iDizi2b [*satir*] [*sutun*] ifadesi ise dizi elemanlarının bellek adreslerini verir:

Tablo:

		1.sutun	2.sutun	3.sutun	4.sutun	5.sutun
iDizi2b[0]:	1.satir	0	1	2	3	4
iDizi2b[1]:	2.satir	5	6	7	8	9
iDizi2b[2]:	3.satir	2	3	4	5	6
iDizi2b[3]:	4.satir	7	8	9	0	1

Üç-boyutlu (3-b) iDizi3b dizisi ise, iki-boyutlu dizilerden oluşur: "*5 tamsayıdan oluşan 4 dizinin (1-b) oluşturduğu 3 diziden (2-b) oluşan dizi*". iDizi3b dizisinin ilk elemanı olan iki-boyutlu iDizi3b[0] dizisi, "*5* **int** *veriden oluşan 4 dizinin (1-b) oluşturduğu dizi*"'dir. iDizi3b dizisinin elemanları olan tamsayı değerlere iDizi3b[*tablo*][*satir*][*sutun*] ifadesi ile erişilir. Dizi elemanlarının bellek adreslerini ise &iDizi3b[*tablo*][*satir*][*sutun*] ifadesi verir:

3. Tablo

2. Tablo

	0	1	2	3	4
0	1	2	3	4	9
5	6	7	8	9	6
2	3	4	5	6	1
7	8	9	0	1	

1. Tablo

	0	1	2	3	4
0	1	2	3	4	9
5	6	7	8	9	6
2	3	4	5	6	1
7	8	9	0	1	

iDizi2b dizisi, tek-boyutlu dizilerin oluşturduğu satırlardan oluşur ve bellekte birbirini izleyen satırlar olarak bulunur (Şekil 6.1-2). Dolayısıyla bellekte ilk olarak dizinin ilk satırını oluşturan iDizi2b[0] dizisi bulunur. Bunu diğer satırlar yani iDizi2b[1], iDizi2b[2] ve iDizi2b[3] dizileri izler. Önceki bölümlerde, tek-boyutlu diziler tanımlanırken ilk değer listesi bulunuyor ise eleman sayısı verilmeden boş köşeli parantezlerle tanımlandı. Derleyici, bildirim deyimindeki ilk değer listesinde bulunan değerleri (0-boyutlu satırlar olarak düşünülebilir) sayarak dizi boyutlarını tesbit eder. İlk değer listesi içeren çok-boyutlu bir dizi bildirim deyiminde yada izleyen bölümlerde anlatılacağı gibi argüman olarak çok-boyutlu dizi alan fonksiyonların parametre bildirim listesinde, satır sayısını belirten 1.boyut kullanılmayabilir. Çünkü dizi bellekte satırları oluşturan alt-diziler olarak saklanır ve dizinin satır sayısı değil satır uzunluğu önemlidir.

Bu durumda iDizi2b aşağıdaki gibi tanımlanabilir:

```
int iDizi2b [  ] [ 5 ] =  { ... };
```

Aynı şekilde üç-boyutlu iDizi3b dizisi de aşağıdaki gibi tanımlanabilir:

```
int iDizi3b [  ] [ 4 ] [ 5 ] =  { ... };
```

Sonuç olarak, bir dizi eksiksiz ilk değer listesi ile tanımlanırken ilk boyut verilmeyebilir.

İki-boyutlu iDizi2b dizisinin içerdiği tek-boyutlu dizi sayısı dizinin 1.boyutu olan satır sayısını verir:

2-b dizinin 1.boyutu:

2-b dizi satır sayısı = Dizinin toplam byte sayısı / Bir satırının byte sayısı

iDizi2b için;

```
sizeof (iDizi2b) / sizeof (iDizi2b[0])
```

Satırları oluşturan bir tek-boyutlu dizinin içerdiği tamsayı eleman sayısı da, dizinin sütun sayısını (2.boyut) verir:

2-b dizinin 2.boyutu:

2-b dizi sütun sayısı = Bir satırının byte sayısı /
Bir tamsayı elemanın (0-boyutlu) byte sayısı

iDizi2b için;

```
sizeof (iDizi2b[0]) / sizeof (iDizi2b[0][0])
```

Bir üç-boyutlu dizi iDizi3b, 2-b dizilerin oluşturduğu *tablo*lardan oluşan bir dizidir. Buna göre dizinin ilk boyutu olan *tablo sayısı*nı, dizinin içerdiği 2-b dizi sayısı verir (Örnek 6.1-2):

3-b dizinin 1.boyutu (tablo sayısı):

3-b dizinin tablo sayısı = Dizinin toplam byte sayısı /
Bir tablonun (2-b dizilerden oluşan) byte sayısı

iDizi3b için;

```
sizeof (iDizi3b) / sizeof (iDizi3b[0])
```

3-b dizinin 2.boyutu olan bir tablodaki *satır sayısı*nı, tablonun içerdiği tek-boyutlu dizi sayısı verir:

3-b dizinin 2.boyutu:

3-b dizinin satır sayısı = Bir tablonun (2-b dizilerden oluşan) byte sayısı / Bir tablodaki bir satırın (tek-boyutlu dizinin) byte sayısı

iDizi3b için;

```
sizeof (iDizi3b[0]) / sizeof (iDizi3b[0][0])
```

3-b dizinin 3.boyutu olan *sütun sayısı*nı ise herhangi bir tabloda bulunan herhangi bir satırı oluşturan tamsayı elemanların sayısı verir:

3-b dizinin 3.boyutu:

3-b dizinin sütun sayısı = Bir tablodaki bir satırın (tek-boyutlu dizinin) byte sayısı / Bir satırı oluşturan bir tamsayı elemanın (0-boyutlu) byte sayısı

iDizi3b için;

```
sizeof (iDizi3b[0][0]) / sizeof (iDizi3b[0][0][0])
```

Aşağıdaki ifadeler 2-b ve 3-b **int** dizilerin eleman sayısını (*rank*) verir:

2-b **int** dizi için:

```
sizeof (iDizi2b) / sizeof (iDizi2b[0][0])
```

yada

```
sizeof (iDizi2b) / sizeof (int)
```

3-b **int** dizi için:

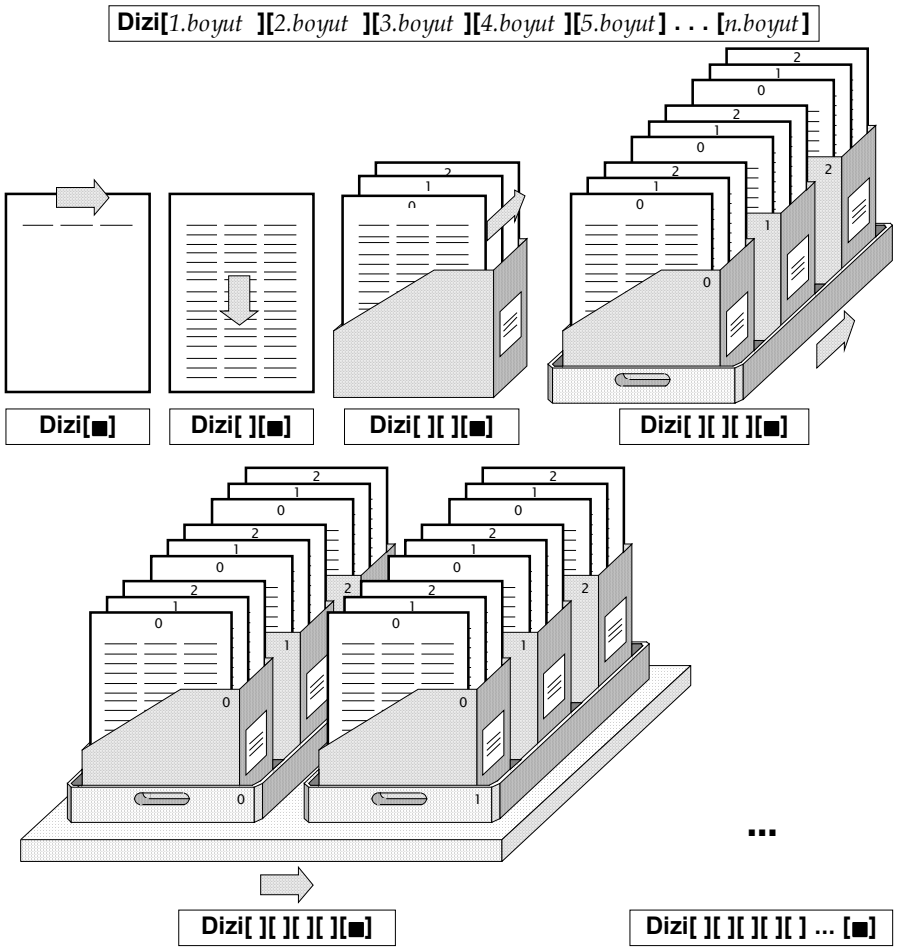
```
sizeof (iDizi3b) / sizeof (iDizi3b[0][0][0])
```

yada

```
sizeof (iDizi3b) / sizeof (int)
```

Örnek 6.1-1 ve Örnek 6.1-2'de bu şekilde oluşturulan makrolar kullanılarak dizilerin tablo, satır ve sütun sayıları bulunur. Çok-boyutlu dizi kavramı Şekil 6.1-1'de gösterilmiştir. Bir tek-boyutlu dizi, tek bir satırdan oluşur. 2-b dizi ise, birden fazla satır içeren ve satır-sütun düzeninde indekslenen bir tablodur. Bir 3-b dizi birden fazla tablo içeren bir klasör (tablo-satır-sütun şeklinde indekslenen); 4-b dizi birden fazla klasör içeren bir kutu; 5-b dizi ise birden fazla kutu içeren bir raf vb... olarak düşünülebilir.

Şekil 6.1-1 Çok-boyutlu diziler.



Örnek 6.1-1'de ilk olarak, 2-b dizi iDizi2b'nin elemanlarının değerleri ve bellek adresleri listelenir. Örnek programlarda satır ve sütun sayılarını hesaplayan makrolar yerine, **#define** ile tanımlanan isimler kullanılabilir:

```
#define   SATIR   4
#define   SUTUN   5
```

Fakat bu değerlerin diziye değer eklendiğinde değiştirilmesi gerekir. Makrolar kullanıldığında ise diziye değer eklendiğinde programda hiç bir değişiklik gerekmez.

Örnek 6.1-1 adr2b.c programı.

```
#include <stdio.h>

#define Satir2b(s) sizeof(s)/sizeof(s[0])          /* satir sayisi       */
#define Sutun2b(s)  sizeof(s[0])/sizeof(s[0][0])  /* sutun sayisi       */

int main(void)
{
   int i, j;                        /* i ve j : satir ve sutun indeksleri     */
   int iDizi2b [  ] [ 5 ]  = {
                                 { 0, 1, 2, 3, 4 },     /*   iDizi2b [ 0 ]     */
                                 { 5, 6, 7, 8, 9 },     /*   iDizi2b [ 1 ]     */
                                 { 2, 3, 4, 5, 6 },     /*   iDizi2b [ 2 ]     */
                                 { 7, 8, 9, 0, 1 }   /*   iDizi2b [ 3 ]     */
                           };

   /*
    *    Adr2b adres degiskeni, 5 tamsayidan olusan diziye -iDizi2b dizisinin
    *    satirlarina- isaret eder
    */
   int (*Adr2b) [ 5 ]  = iDizi2b;

   puts( "satir, sutun: deger(adres) " );
   for ( i = 0; i < Satir2b( iDizi2b ); i++ )
   {
        for ( j = 0; j < Sutun2b( iDizi2b ); j++ )
             printf( "%d,%d: %d (%p) ",  i,  j, iDizi2b [ i ] [ j ], &iDizi2b [ i ] [ j ]  );
        putchar( '\n' );
   }

   printf( "iDizi2b : %p, sizeof(iDizi2b) : %u\n",         iDizi2b,
                                                              sizeof (iDizi2b)  );
   printf( "iDizi2b+1 : %p, sizeof(iDizi2b+1) : %u\n",  iDizi2b+1,
                                                              sizeof (iDizi2b+1) );
   printf( "iDizi2b[1] : %p, sizeof(iDizi2b[1]) : %u\n",       iDizi2b[1],
                                                                 sizeof (iDizi2b[1]) );
   printf( "iDizi2b[1]+1 : %p, sizeof(iDizi2b[1]+1) : %u\n",iDizi2b[1]+1,
   printf( "iDizi2b[1][1] : %d, sizeof(iDizi2b[1][1]): %u\n", iDizi2b[1][1],

   puts( "\nDizi elemanlarinin degerlerinin listelenmesi : \n");

   puts( "*( iDizi2b[ satir ] + sutun ) :" );
   for ( i = 0; i < Satir2b(iDizi2b); i++ )
   {
        for ( j = 0; j < Sutun2b(iDizi2b); j++ )
             printf( "%d ", *( iDizi2b[ i ] + j )  );
        putchar( '\n' );
   }

   printf( "\nsizeof(Adr2b) : %u, sizeof(*Adr2b) : %u\n",  sizeof ( Adr2b  ),
                                                              sizeof ( *Adr2b ) );
   printf( "Adr2b : %p, Adr2b + 1 : %p\n",                    Adr2b,  Adr2b + 1 );
```

...Örnek 6.1-1 devam

```
   puts( "\nAdr2b[ satir ][ sutun ] :" );
   for ( i = 0; i < Satir2b(iDizi2b); i++ )
   {
        for ( j = 0; j < Sutun2b(iDizi2b); j++ )
             printf( "%d ", Adr2b[ i ][ j ] );
        putchar( '\n' );
   }
   return 0;
}
```

Çıktı

```
satir, sutun: deger(adres)
0,0: 0 (0F28)  0,1: 1 (0F2A)  0,2: 2 (0F2C)  0,3: 3 (0F2E)  0,4: 4 (0F30)
1,0: 5 (0F32)  1,1: 6 (0F34)  1,2: 7 (0F36)  1,3: 8 (0F38)  1,4: 9 (0F3A)
2,0: 2 (0F3C)  2,1: 3 (0F3E)  2,2: 4 (0F40)  2,3: 5 (0F42)  2,4: 6 (0F44)
3,0: 7 (0F46)  3,1: 8 (0F48)  3,2: 9 (0F4A)  3,3: 0 (0F4C)  3,4: 1 (0F4E)
iDizi2b              :  0F28, sizeof (iDizi2b)             : 40
iDizi2b + 1          :  0F32, sizeof (iDizi2b + 1)      : 40
iDizi2b[ 1 ]         :  0F32, sizeof (iDizi2b[ 1 ])        : 10
iDizi2b[ 1 ] + 1     :  0F34, sizeof (iDizi2b[ 1 ] + 1)    : 10
iDizi2b[ 1 ][ 1 ]   :  6,      sizeof (iDizi2b[ 1 ][ 1 ]): 2
Dizi elemanlarinin degerlerinin listelenmesi :
*( iDizi2b [ satir ] + sutun ) :
0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1
sizeof(Adr2b) : 2,  sizeof(*Adr2b) : 10
Adr2b : 0F28, Adr2b + 1 : 0F32
Adr2b [ satir ][ sutun ] :
0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1
```

Çıktıda da görüldüğü gibi tamsayı dizi elemanları bellekte tek-boyutlu bir dizide olduğu gibi 2-byte büyüklüğünde sıralı bellek alanlarında bulunur. Önceki bölümde, tek-boyutlu dizinin elemanlarına dizinin başlangıcına işaret eden bir basit adres değişkeni aracılığı ile erişilirken kullanılan adres aritmetiği, çok-boyutlu diziler için de geçerlidir.

Dolayısıyla 2-b dizi iDizi2b için dizi ismi kullanılarak aşağıdaki ifadeler oluşturulabilir :

iDizi2b ifadesi:

İki-boyutlu dizinin başlangıç adresini verir (0F28) ve "5 **int**'ten oluşan 4 dizinin oluşturduğu dizi" tipindedir. **sizeof(**iDizi2b**)** yada **sizeof(int**[4][5]**)** ifadeleri iDizi2b dizisinin toplam büyüklüğü olan 40 byte değerini verir.

iDizi2b + i ifadesi:

Dizi ismi iDizi2b'ye offset eklenerek, satırları oluşturan alt-dizilerin (1-b) başlangıç adresleri alınabilir (i : satır indeksi). Dolayısıyla iDizi2b + i, dizinin i. satırına işaret eder. Bu şekilde oluşturulan ifadelere **sizeof** operatörü uygulandığında yine 40 byte değeri elde edilir. Programın çıktısında da görüldüğü gibi, yine bu ifadeler de adres aritmetiğine göre yorumlanır:

```
iDizi2b  +  0  :  0F28   ( yada iDizi2b : 0F28 )
iDizi2b  +  1  :  0F32
iDizi2b  +  2  :  0F3C
iDizi2b  +  3  :  0F46
```

*(iDizi2b + i) ifadesi:

Yukarıdaki ifadelere indirek değer operatörü uygulandığında elde edilen değer "5 **int**'ten oluşan dizi" tipindedir (10 byte) ve yine alt-dizilerin başlangıç adresini verir:

```
*(iDizi2b  +  0):  0F28  ( yada *iDizi2b : 0F28 )
*(iDizi2b  +  1):  0F32
*(iDizi2b  +  2):  0F3C
*(iDizi2b  +  3):  0F46
```

iDizi2b[i] ifadesi de yine alt-dizilerin başlangıç adresini verir.

Programın çıktısında da görüldüğü gibi (satır indeksi i : 1 için), yukarıda verilen iDizi2b + i ve iDizi2b[i] ifadelerinin her ikisi de alt-dizilerin başlangıç adresini verir. Fakat bu iki ifadenin tipleri farklıdır ve dolayısıyla birbirlerine eşit değildirler.

*(iDizi2b + i) + j ve *(*(iDizi2b + i) + j) ifadeleri:

*(iDizi2b + i) ifadesi alt-dizilerin başlangıcına işaret ettiğine göre, bu ifadeye sütun offseti (j) eklenerek alt-dizilerin (1-b) elemanlarının bellek adresleri elde edilebilir:

*(iDizi2b + i) + j yada iDizi2b[i] + j (yada &iDizi2b [i][0] + j)

Tekrar indirek değer operatörü uygulandığında işaret edilen bellek alanlarında bulunan tamsayı değerler elde edilir (iDizi2b [i][j]):

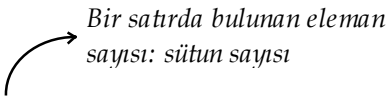
((iDizi2b + i) + j) yada *(iDizi2b[i] + j)

j : 0 değeri için **(iDizi2b + i) yada *iDizi2b[i] ifadeleri alt-dizilerin başlangıç elemanlarının değerlerini verir.

Örnek 6.1-1'de dizi ismi ile oluşturulan *(iDizi2b[i] + j) ifadesi kullanılarak tüm elemanların değerleri listelenir. 2-b dizi elemanlarına adres değişkeni kullanılarakta erişilebilir. Bu amaçla, 2-b dizi iDizi2b içindeki tamsayı değerlerden oluşan 1-b diziye işaret eden adres değişkeni Adr2b tanımlanır ve ilk değer olarak iDizi2b dizisinin başlangıç adresi atanır:

```
int (*Adr2b)[ 5 ] = iDizi2b;
```

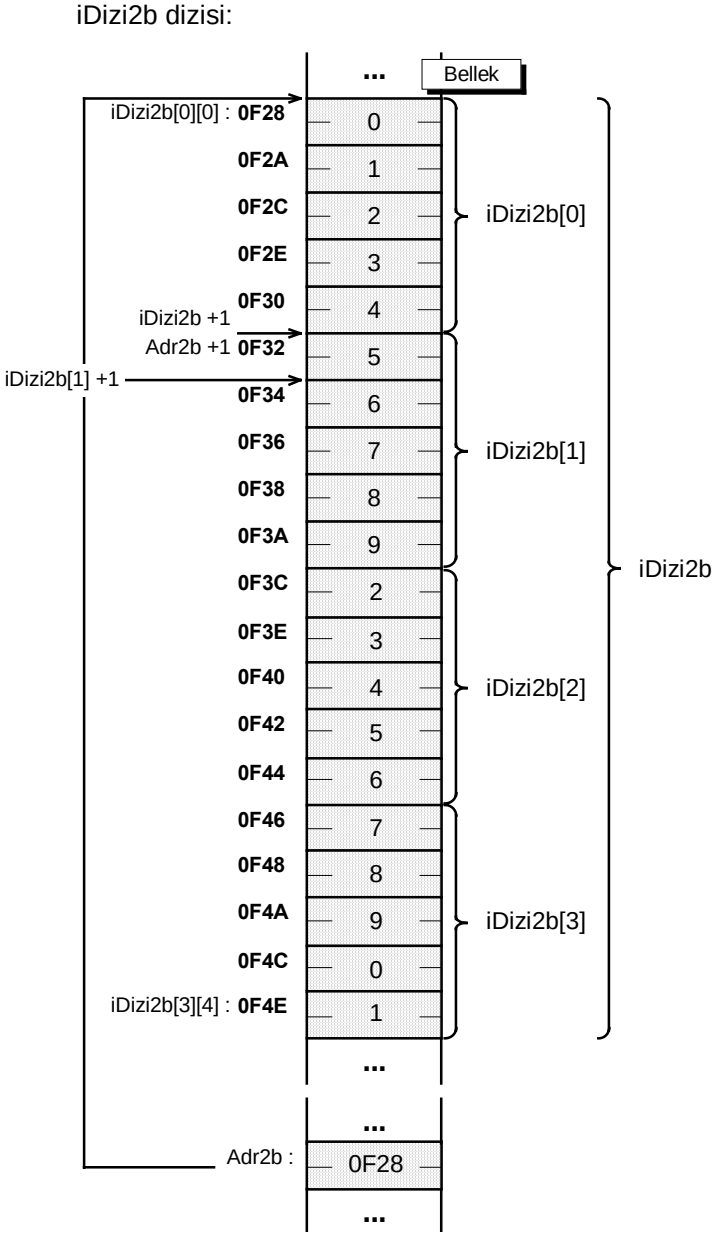
Genel format aşağıdaki gibidir:



tip **(***adres_degiskeni***) [*** *satir_uzunlugu* **] =** *2b_dizi_ismi* ;

Böylece iDizi2b'nin satırlarını oluşturan tek-boyutlu dizilere işaret eden Adr2b tanımlanmış olur.

Şekil 6.1-2 2-b dizi iDizi2b ve Adr2b adres değişkeni.



Adr2b adres değişkeni "*5* **int** *veriden oluşan diziye işaret eden adres değişkeni*" tipinde olduğu için, değeri ++ operatörü uygulanarak artırıldığında bellekte bir sonraki tek-boyutlu diziye (dizinin bir sonraki satırına) işaret eder (yani değeri **sizeof(int**(*)[5]**)**

yada **sizeof(*Adr2b)** kadar artar). Çıktıda da görüldüğü gibi Adr2b adres değişkeninin indirek değeri 10 byte büyüklüğündedir.

Dizi ismi kullanılarak oluşturulan ifadelerde olduğu gibi, adres değişkeni de satır-sütun şeklinde indekslenerek elemanlara erişilebilir. Adr2b[*satir*][*sutun*] dizi elemanlarının değerlerini verir. Bu ifade iDizi2b[*satir*][*sutun*] ifadesi ile aynı değeri döndürür. Fakat derleyici indeks operatörünün adres değişkenine uygulandığı ifadeleri, dizi ismine uygulandığı ifadelerden farklı yorumlar (izleyen bölümde anlatılacak olan argüman olarak çok-boyutlu dizi alan fonksiyonların parametre listesi hariç) ve her ikisi için farklı kod üretir. Dizi ismi iDizi2b bir sabit göstergedir. Dizi ismi ile oluşturulan ifade, "dizi başlangıcından itibaren i * *sutun_sayisi* + j sayısı kadar bellek alanı (her biri tek bir 0-boyutlu dizi elemanı büyüklüğünde) atlanarak gelinen yerde bulunan değere erişim" anlamında yorumlanır. Adres değişkeni ile oluşturulan ifade ise, "adres değişkeninin değerine adres aritmetiğine göre i * *sutun_sayisi* + j eklenerek elde edilen adreste bulunan değere erişim" anlamında yorumlanır.

İki-boyutlu dizinin elemanı olan 1-b dizilere işaret eden adres değişkeni ile oluşturulan,

Adr2b[*satir*][*sutun*]

ifadesi aşağıdaki ifade ile,

*(Adr2b[*satir*] + *sutun*)

ve dolayısıyla,

*(*(Adr2b + *satir*) + *sutun*)

ifadesi ile aynıdır.

iDizi2b dizisinin elemanları sıralı bellek adreslerinde bulunduğu için erişim, dizinin tek bir elemanına işaret eden Adr adres değişkeni kullanılarak aşağıdaki şekilde gerçekleştirilebilir:

```
...
int *Adr = &iDizi2b[ 0 ] [ 0 ];
...
        ...

/* indeks hesaplanarak degerlerin listelenmesi */
for ( i = 0; i < Satir2b(iDizi2b); i++ )
{
    for ( j = 0; j < Sutun2b(iDizi2b); j++ )
        printf( "%d ", Adr [ i * Sutun2b(iDizi2b) + j ] );
    putchar( '\n' );
}
        ...
```

Fakat dizi elemanlarına bu şekilde erişim, 2-b dizi görünümü vermez ve ayrıca izleyen bölümlerde anlatılacak olan bazı uygulamalarda kullanılması mümkün değildir.

Örnek 6.1-2'de, 3-b dizi iDizi3b'nin satırlarını oluşturan 2-b dizilere (tablolara) işaret eden adres değişkeni Adr3b tanımlanır ve yine Örnek 6.1-1'de olduğu gibi adres aritmetiği ile oluşturulan ifadeler kullanılarak dizi elemanlarına erişilir.

Örnek 6.1-2 adr3b.c programı.

```
#include <stdio.h>

#define Tablo3b(s)  sizeof(s)/sizeof(s[0])              /* tablo sayisi    */
#define Satir3b(s)  sizeof(s[0])/sizeof(s[0][0])        /* satir sayisi    */
#define Sutun3b(s)  sizeof(s[0][0])/sizeof(s[0][0][0])  /* sutun sayisi    */

int main(void)
{
  int i, j, k;                  /* i, j ve k : tablo, satir ve sutun indeksleri*/
  int iDizi3b [ ] [ 4 ] [ 5 ] =
                                {
                                  {  /*  iDizi3b [0]  */
                                    { 0, 1, 2, 3, 4 },          /*  iDizi3b[0][0]  */
                                    { 5, 6, 7, 8, 9 },          /*  iDizi3b[0][1]  */
                                    { 2, 3, 4, 5, 6 },          /*  iDizi3b[0][2]  */
                                    { 7, 8, 9, 0, 1 }           /*  iDizi3b[0][3]  */
                                  },
                                  {  /*  iDizi3b [1]  */
                                    { 0, 1, 2, 3, 4 },          /*  iDizi3b[1][0]  */
                                    { 5, 6, 7, 8, 9 },          /*  iDizi3b[1][1]  */
                                    { 2, 3, 4, 5, 6 },          /*  iDizi3b[1][2]  */
                                    { 7, 8, 9, 0, 1 }           /*  iDizi3b[1][3]  */
                                  },

                                  {  /*  iDizi3b [2]  */
                                    { 0, 1, 2, 3, 4 },          /*  iDizi3b[2][0]  */
                                    { 5, 6, 7, 8, 9 },          /*  iDizi3b[2][1]  */
                                    { 2, 3, 4, 5, 6 },          /*  iDizi3b[2][2]  */
                                    { 7, 8, 9, 0, 1 }           /*  iDizi3b[2][3]  */
                                  }
                                };
  /*
   * Adr3b adres degiskeni, tablolara -iDizi3b dizisinin satirlarina- isaret eder.
   */
  int (*Adr3b) [ 4 ][ 5 ] = iDizi3b;

  puts( "Tablo, Satir, Sutun: Deger(Adres) :" );
  for ( i = 0; i < Tablo3b(iDizi3b); i++ )
  {
       printf("Tablo No : %d\n", i );
```

...Örnek 6.1-2 devam

```
        for ( j = 0; j < Satir3b(iDizi3b); j++ )
        {
            for ( k = 0; k < Sutun3b(iDizi3b); k++ )
                printf( "%d,%d,%d: %d(%p) ", i, j, k, iDizi3b[i][j][k], &iDizi3b[i][j][k] );
            putchar( '\n' );
        }
        putchar( '\n' );
    }
    puts( "ifade : degeri, sizeof (ifade)" );
    printf( "iDizi3b             : %p, %u\n",   iDizi3b,                sizeof (iDizi3b)                );
    printf( "iDizi3b + 1         : %p, %u\n",   iDizi3b + 1,            sizeof (iDizi3b + 1)            );
    printf( "iDizi3b[1]          : %p, %u\n",   iDizi3b[1],             sizeof (iDizi3b[1])             );
    printf( "iDizi3b[1] + 1      : %p, %u\n",   iDizi3b[1] + 1,         sizeof (iDizi3b[1] + 1)         );
    printf( "iDizi3b[1][1]       : %p, %u\n",   iDizi3b[1][1],          sizeof (iDizi3b[1][1])          );
    printf( "iDizi3b[1][1] + 1   : %p, %u\n",   iDizi3b[1][1] + 1,      sizeof (iDizi3b[1][1] + 1)      );
    printf( "iDizi3b[1][1][1]    : %d, %u\n",   iDizi3b[1][1][1],       sizeof (iDizi3b[1][1][1])       );

    puts( "\nDizi elemanlarinin degerlerinin listelenmesi:\n" );

    puts( "*( *( *(iDizi3b + i ) + j ) + k ) :" );
    for ( i = 0; i < Tablo3b(iDizi3b); i++ )
    {
        printf("Tablo No : %d\n", i );
        for ( j = 0; j < Satir3b(iDizi3b); j++ )
        {
            for ( k = 0; k < Sutun3b(iDizi3b); k++ )
                printf( "%d ", *( *( *(iDizi3b + i) + j ) + k )  );
            putchar( '\n' );
        }
        putchar( '\n' );
    }
    printf( "sizeof(Adr3b) : %u, sizeof(*Adr3b) : %u\n",  sizeof (Adr3b),   sizeof (*Adr3b)  );
    printf( "Adr3b : %p, Adr3b + 1 : %p\n",               Adr3b,            Adr3b + 1        );

    puts( "\nAdr3b[ tablo ][ satir ][ sutun ]: " );
    for ( i = 0; i < Tablo3b(iDizi3b); i++ )
    {
        for ( j = 0; j < Satir3b(iDizi3b); j++ )
        {
            for ( k = 0; k < Sutun3b(iDizi3b); k++ )
                printf( "%d ", Adr3b[ i ][ j ][ k ] );
                /*
                 *      *( *( *(Adr3b + i) + j ) + k )
                 */
            putchar( '\n' );
        }
        putchar( '\n' );
    }
```

...Örnek 6.1-2 devam

```
   return 0;
}
```

Çıktı

```
Tablo, Satir, Sutun: Deger(Adres) :
Tablo No : 0
0,0,0: 0 (0EB6)  0,0,1: 1 (0EB8)  0,0,2: 2 (0EBA)  0,0,3: 3 (0EBC)  0,0,4: 4 (0EBE)
0,1,0: 5 (0EC0)  0,1,1: 6 (0EC2)  0,1,2: 7 (0EC4)  0,1,3: 8 (0EC6)  0,1,4: 9 (0EC8)
0,2,0: 2 (0ECA)  0,2,1: 3 (0ECC)  0,2,2: 4 (0ECE)  0,2,3: 5 (0ED0)  0,2,4: 6 (0ED2)
0,3,0: 7 (0ED4)  0,3,1: 8 (0ED6)  0,3,2: 9 (0ED8)  0,3,3: 0 (0EDA)  0,3,4: 1 (0EDC)

Tablo No : 1
1,0,0: 0 (0EDE)  1,0,1: 1 (0EE0)  1,0,2: 2 (0EE2)  1,0,3: 3 (0EE4)  1,0,4: 4 (0EE6)
1,1,0: 5 (0EE8)  1,1,1: 6 (0EEA)  1,1,2: 7 (0EEC)  1,1,3: 8 (0EEE)  1,1,4: 9 (0EF0)
1,2,0: 2 (0EF2)  1,2,1: 3 (0EF4)  1,2,2: 4 (0EF6)  1,2,3: 5 (0EF8)  1,2,4: 6 (0EFA)
1,3,0: 7 (0EFC)  1,3,1: 8 (0EFE)  1,3,2: 9 (0F00)  1,3,3: 0 (0F02)  1,3,4: 1 (0F04)

Tablo No : 2
2,0,0: 0 (0F06)  2,0,1: 1 (0F08)  2,0,2: 2 (0F0A)  2,0,3: 3 (0F0C)  2,0,4: 4 (0F0E)
2,1,0: 5 (0F10)  2,1,1: 6 (0F12)  2,1,2: 7 (0F14)  2,1,3: 8 (0F16)  2,1,4: 9 (0F18)
2,2,0: 2 (0F1A)  2,2,1: 3 (0F1C)  2,2,2: 4 (0F1E)  2,2,3: 5 (0F20)  2,2,4: 6 (0F22)
2,3,0: 7 (0F24)  2,3,1: 8 (0F26)  2,3,2: 9 (0F28)  2,3,3: 0 (0F2A)  2,3,4: 1 (0F2C)

ifade : degeri, sizeof (ifade)
iDizi3b              : 0EB6,  120
iDizi3b + 1          : 0EDE,  120
iDizi3b[1]           : 0EDE,  40
iDizi3b[1] + 1       : 0EE8,  40
iDizi3b[1][1]        : 0EE8,  10
iDizi3b[1][1] + 1 : 0EEA,  10
iDizi3b[1][1][1]     : 6, 2

Dizi elemanlarinin degerlerinin listelenmesi:

*( *( *(iDizi3b + i ) + j ) + k ) :
Tablo No : 0
0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1

Tablo No : 1
0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1
Tablo No : 2
0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
```

...Örnek 6.1-2 Çıktı devam

```
7 8 9 0 1

sizeof (Adr3b) : 2, sizeof (*Adr3b) : 40
Adr3b : 0EB6, Adr3b + 1 : 0EDE

Adr3b[ tablo ][ satir ][ sutun ]:
0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1

0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1

0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1
```

---

iDizi3b dizisi iDizi3b[0], iDizi3b[1] ve iDizi3b[2] *tablo*larından (2-b diziler) oluşur. Tablolar ise 1-b dizilerden oluşur. Buna göre iDizi3b[0] tablosunun *satır*ları olan 1-b diziler: iDizi[0][0], iDizi[0][1], iDizi[0][2] ve iDizi[0][3]. Aynı şekilde iDizi3b[1] tablosunun satırları; iDizi[1][0], iDizi[1][1], iDizi[1][2] ve iDizi[1][3] dizileri; iDizi3b[2] tablosunun satırları ise iDizi[2][0], iDizi[2][1], iDizi[2][2] ve iDizi[2][3] dizileridir. Her 1-b dizi ise 5 **int** değer (*sutun* sayısı) içerir. Örnek programda Adr3b adres değişkenine indeks operatörü uygulanarak oluşturulan,

```
Adr3b [ i ] [ j ] [ k ]
```

ifadesi,

```
*( *( *( Adr3b + i ) + j ) + k )
```

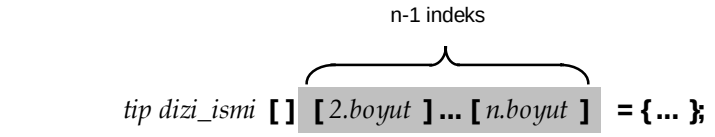
ifadesi ile aynıdır.

Adr3b adres değişkeni, "*5* **int** *değerden oluşan 4 (1-b) diziden oluşan diziye (2-b) işaret eden adres değişkeni*" tipindedir: yani **int**(*)[4][5] tipindedir. Dolayısıyla, indirek değerine **sizeof** operatörünün uygulandığı **sizeof** (*Adr3b) ifadesi 40 değerini verir (**sizeof** (**int**(*)[4][5]) ifadesi de aynı değeri verir). Bu nedenle Adr3b + 1 ifadesi, iDizi3b dizisinin bir sonraki tablosuna (satırına) işaret eder (adres aritmetiğine göre onaltılık 0EB6 sayısına 1 değeri eklendiğinde, bu değer indirek değerin tipi ile ölçeklendiği için gerçekte 1*40 değeri eklenir ve sonuçta 0EDE sayısı elde edilir).

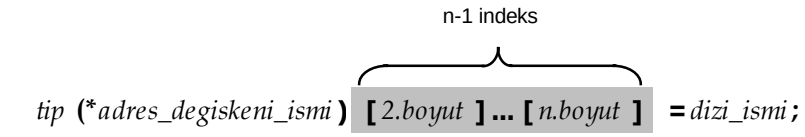
Bu bölümde son olarak 4-b dizi örneklenmiştir. Şekil 6.1-3'de bu diziyi oluşturan tüm alt-diziler şematik olarak gösterilmiştir.

n boyutlu bir dizi için aşağıdaki ifadeler oluşturulabilir:

n boyutlu dizi tanımı (eksiksiz ilk değer listesi ile):



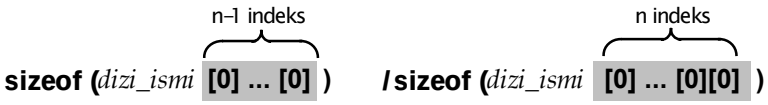
n boyutlu diziye işaret eden adres değişkeni tanımı:



Aşağıdaki ifade dizinin 1.boyut değerini verir:

sizeof (*dizi_ismi*) / sizeof (*dizi_ismi* [0])

Aşağıdaki ifade ise dizinin n.boyut değerini verir:



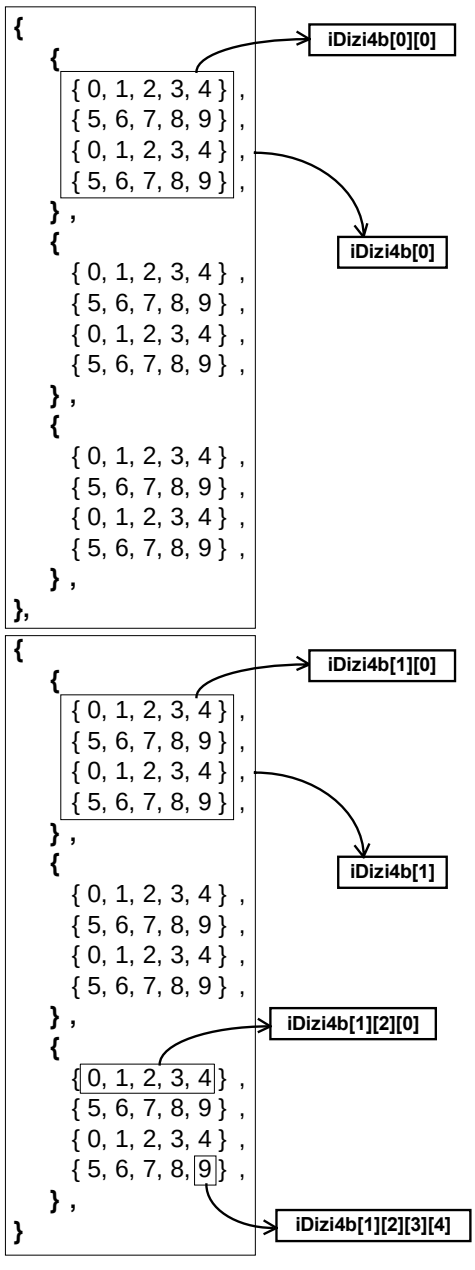
Bu bölümde çok-boyutlu dizilerin tanımlanması ve dizi elemanlarına adres değişkeni aracılığı ile erişim anlatıldı. ■

Örnek 6.1-3 adr4b.c programı.

```
#include <stdio.h>

#define Boyut1(s) sizeof(s)/sizeof(s[0])
#define Boyut2(s) sizeof(s[0])/sizeof(s[0][0])
#define Boyut3(s) sizeof(s[0][0])/sizeof(s[0][0][0])
#define Boyut4(s) sizeof(s[0][0][0])/sizeof(s[0][0][0][0])

int main(void)
{
   int i, j, k, l;

  /* 1.boyut (2) verilmeyebilir */
```

...Örnek 6.1-3 devam

```
int  iDizi4b[ 2 ][ 3 ][ 4 ][ 5 ] =
{
  {
    {
      { 0, 1, 2, 3, 4 } ,
      { 5, 6, 7, 8, 9 } ,
      { 0, 1, 2, 3, 4 } ,
      { 5, 6, 7, 8, 9 } ,
    } ,
    {
      { 0, 1, 2, 3, 4 } ,
      { 5, 6, 7, 8, 9 } ,
      { 0, 1, 2, 3, 4 } ,
      { 5, 6, 7, 8, 9 } ,
    } ,
    {
      { 0, 1, 2, 3, 4 } ,
      { 5, 6, 7, 8, 9 } ,
      { 0, 1, 2, 3, 4 } ,
      { 5, 6, 7, 8, 9 } ,
    } ,
  },
  {
    {
      { 0, 1, 2, 3, 4 } ,
      { 5, 6, 7, 8, 9 } ,
      { 0, 1, 2, 3, 4 } ,
      { 5, 6, 7, 8, 9 } ,
    } ,
    {
      { 0, 1, 2, 3, 4 } ,
      { 5, 6, 7, 8, 9 } ,
      { 0, 1, 2, 3, 4 } ,
      { 5, 6, 7, 8, 9 } ,
    } ,
    {
      { 0, 1, 2, 3, 4 } ,
      { 5, 6, 7, 8, 9 } ,
      { 0, 1, 2, 3, 4 } ,
      { 5, 6, 7, 8, 9 } ,
    } ,
  }
};
```




```
int (*Adr4b) [ 3 ][ 4 ][ 5 ] = iDizi4b;

  for ( i = 0; i < Boyut1(iDizi4b); i++ )
  {
      for ( j = 0; j < Boyut2(iDizi4b); j++ )
```

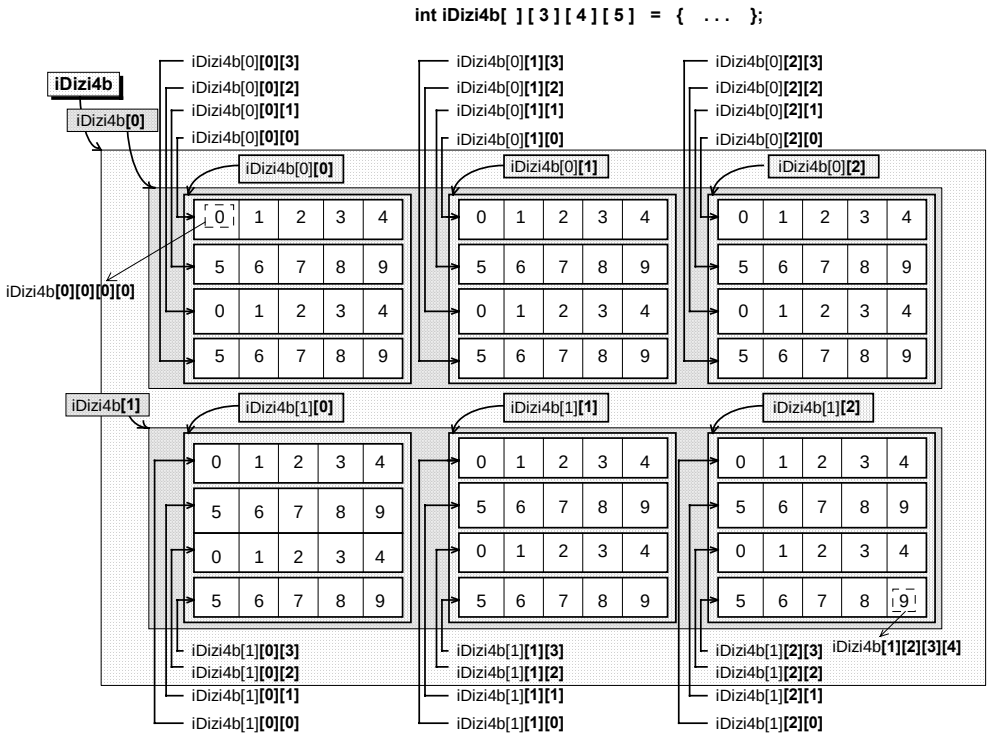
...Örnek 6.1-3 devam

```
        {
            for ( k = 0; k < Boyut3(iDizi4b); k++ )
            {
                for ( l = 0; l < Boyut4(iDizi4b); l++ )
                    printf( "%d ", Adr4b[ i ][ j ][ k ][ l ] );
                putchar( '\n' );
            }
            putchar( '\n' );
        }
        putchar( '\n' );
    }
    return 0;
}
```

Çıktı

```
0 1 2 3 4
5 6 7 8 9
0 1 2 3 4
5 6 7 8 9

0 1 2 3 4
5 6 7 8 9
0 1 2 3 4
5 6 7 8 9

0 1 2 3 4
5 6 7 8 9
0 1 2 3 4
5 6 7 8 9

0 1 2 3 4
5 6 7 8 9
0 1 2 3 4
5 6 7 8 9

0 1 2 3 4
5 6 7 8 9
0 1 2 3 4
5 6 7 8 9

0 1 2 3 4
5 6 7 8 9
0 1 2 3 4
5 6 7 8 9
```

Şekil 6.1-3 4-boyutlu iDizi4b dizisini oluşturan alt-dizilerin şematik olarak gösterilmesi.



6.2 Çok-Boyutlu Dizilerin Fonksiyonlara Aktarılması

C dilinde argümanlar fonksiyonlara değerleri ile aktarılır (*call-by-value*). Çağrılan fonksiyon argüman listesinde yer alan bir basit değişkenin yada izleyen bölümlerde anlatılacak olan bir yapı değişkeninin lokal kopyasını aldığı için argümanların çağıran bloktaki değerlerini değiştiremez.

Bu durum, argümanlar dizi olduğunda geçerli değildir. Çünkü önceki bölümlerde de anlatıldığı gibi, bir tek-boyutlu dizi ismi fonksiyon çağrısında argüman olarak yer aldığında dizinin değerleri yerine başlangıç elemanının bellek adresi aktarılır (*call-by-reference*) ve fonksiyon çağıran bloktaki dizi elemanlarını değerlerini değiştirebilir.
Fonksiyon tanımında karşılık gelen parametre ise, aktarılan adres değerinin atanabileceği bir adres değişkenidir. Parametre bildiriminin indeks operatörü kullanılarak dizi bildirimi şeklinde (*array notation* yada *subscript notation*) yapıldığında da yine parametre adres değişkenidir.

Örnek olarak tamsayı elemanlardan oluşan tek-boyutlu iDizi1b dizisini argüman olarak kabul eden bir fonksiyon, aşağıdaki şekillerde tanımlanabilir :

Dizi tanımı:

```
int iDizi1b[ ] = { 0, 1, 2 };
```

Fonksiyon çağrısı:

```
fonk( iDizi1b );
```

Fonksiyon tanımı:

```
void fonk( int Adr1b [ 3 ] )
{ ... }
```

yada

```
void fonk( int Adr1b [ ] )
{ ... }
```

yada

```
void fonk( int *Adr1b )
{ ... }
```

Yukarıdaki üç parametre tanımı da birbirinin aynıdır ve bildirilen Adr1b, bir adres değişkenidir. Dizinin başlangıç elemanının çağrı sırasında aktarılan adresi, tek bir **int** değişkene işaret eden Adr1b adres değişkenine atanır ve bu adres değişkeni aracılığı ile çağıran bloktaki dizi elemanlarına erişilir.

Çok-boyutlu diziler bellekte alt-dizilerden oluşan satırlar olarak (*row-major order*) saklanır. Bir çok-boyutlu dizi ismi, dizinin ilk satırına işaret eder ve ilk satırın başlangıç

adresini veren sabittir. Dolayısıyla izleyen paragraflarda da anlatılacağı gibi, bir çok-boyutlu dizi ismi çağrıda argüman listesinde yer aldığında, fonksiyona çağrı sırasında dizinin ilk satırına işaret eden adres değeri aktarılır. Bir tek-boyutlu dizinin satırları 0-boyutlu alt-diziler olan tamsayı değişkenler oluşturur. Dolayısıyla aynı durum tek-boyutlu diziler için de geçerlidir ve Adr1b adres değişkeni çağıran bloktaki tek-boyutlu iDizi1b dizisinin satırlarını oluşturan 0-boyutlu tamsayı dizi elemanlarına yada bir başka deyişle dizinin satırlarına işaret eder.

Çağrılan fonksiyon dizi boyutlarını bilmediği için, dizinin eleman sayısı aşağıda olduğu gibi fonksiyona aktarılmalıdır:

Fonksiyon çağrısı:

```
fonk( iDizi1b,  sizeof iDizi1b / sizeof iDizi1b[0]  );
```

Fonksiyon tanımı:

```
void fonk( int Adr1b [ ],  size_t  ELEMAN_SAYI )
{ ... }
```

Çok-boyutlu bir dizi bellekte birbirini izleyen satırlar olarak bulunduğu için, dizi elemanlarına sıralı olarak erişilirken en sağda bulunan indeks en hızlı değişen indeks olacaktır.

Dizi bildiriminde dizinin satır sayısını belirten ilk indeks, derleme sırasında dizi için gerekli olan bellek alanı büyüklüğünün saptanmasına yardımcı olur. İndeks hesaplarında etkili olmaz. Dolayısıyla bir çok-boyutlu diziyi bir fonksiyona argüman olarak aktarmak için, fonksiyon tanımındaki karşılık gelen parametre bildirimi dizinin satır uzunluğunu (sütun sayısını) içermelidir. Satır sayısı gereksizdir çünkü çağrı sırasında aktarılan, dizinin satırlarını oluşturan dizilere işaret eden adres değeridir.

Örnek 6.2-1'de iki, üç ve dört-boyutlu diziler, dizi değerlerini listeleyen fonksiyonlara argüman olarak aktarılır. Ayrıca programda bir 2-b dizinin 1. ve 2. satırlarını değiştiren fonksiyon tanımlanmıştır.

Örnek 6.2-1 fonkdizi.c programı.

```
#include   <stdio.h>
#include   <string.h>   /* memcpy icin */

void   Liste2b(   int   (*)[ 5 ]           );
void   Liste3b(   int   (*)[ 4 ][ 5 ]      );
void   Liste4b(   int   (*)[ 3 ][ 4 ][ 5 ] );
void   Degis2b(   int   (*)[ 5 ]           );
```

... Örnek 6.2-1 devam

```
int   iDizi2b [ ] [ 5 ] =
                        {
                            { 0, 1, 2, 3, 4 },
                            { 5, 6, 7, 8, 9 },
                            { 2, 3, 4, 5, 6 },
                            { 7, 8, 9, 0, 1 }
                        };

int   iDizi3b [ ] [ 4 ] [ 5 ] =
                        {
                          { { 0, 1, 2, 3, 4 }, { 5, 6, 7, 8, 9 }, { 2, 3, 4, 5, 6 }, { 7, 8, 9, 0, 1 } },
                          { { 0, 1, 2, 3, 4 }, { 5, 6, 7, 8, 9 }, { 2, 3, 4, 5, 6 }, { 7, 8, 9, 0, 1 } },
                          { { 0, 1, 2, 3, 4 }, { 5, 6, 7, 8, 9 }, { 2, 3, 4, 5, 6 }, { 7, 8, 9, 0, 1 } }
                        };
int   iDizi4b [ ] [ 3 ] [ 4 ] [ 5 ] =
                        {
                          {
                            { { 0, 1, 2, 3, 4 }, { 5, 6, 7, 8, 9 }, { 2, 3, 4, 5, 6 }, { 7, 8, 9, 0, 1 } },
                            { { 0, 1, 2, 3, 4 }, { 5, 6, 7, 8, 9 }, { 2, 3, 4, 5, 6 }, { 7, 8, 9, 0, 1 } },
                            { { 0, 1, 2, 3, 4 }, { 5, 6, 7, 8, 9 }, { 2, 3, 4, 5, 6 }, { 7, 8, 9, 0, 1 } }
                          },
                          {
                            { { 0, 1, 2, 3, 4 }, { 5, 6, 7, 8, 9 }, { 2, 3, 4, 5, 6 }, { 7, 8, 9, 0, 1 } },
                            { { 0, 1, 2, 3, 4 }, { 5, 6, 7, 8, 9 }, { 2, 3, 4, 5, 6 }, { 7, 8, 9, 0, 1 } },
                            { { 0, 1, 2, 3, 4 }, { 5, 6, 7, 8, 9 }, { 2, 3, 4, 5, 6 }, { 7, 8, 9, 0, 1 } }
                          },
                        };
int main(void)
{
  printf( "sizeof iDizi2b : %u\n", sizeof iDizi2b );
  Liste2b( iDizi2b );
  printf( "sizeof iDizi3b : %u\n", sizeof iDizi3b );
  Liste3b( iDizi3b );
  printf( "sizeof iDizi4b : %u\n", sizeof iDizi4b );
  Liste4b( iDizi4b );
  Degis2b( iDizi2b );
  puts( "2b dizinin 1. satiri ile 2. satirinin degistirilmesi: " );
  Liste2b( iDizi2b );
  return 0;
}
void Liste2b( int (*Adr2b)[ 5 ] )
{
  int i, j;
  puts( "- 2b dizi listeleme-" );
```

... Örnek 6.2-1 devam

```
   printf("sizeof  Adr2b : %u, sizeof *Adr2b : %u, ",  sizeof Adr2b, sizeof *Adr2b  );
   printf("sizeof *Adr2b / sizeof (int) : %u\n",       sizeof *Adr2b / sizeof (int)  );

   for ( i = 0; i < 4; i++ )
   {
        for ( j = 0; j < 5; j++ )
             printf( "%d ", Adr2b[ i ][ j ] );
        putchar( '\n' );
   }
}
void Liste3b( int (*Adr3b)[ 4 ][ 5 ] )
{
   int i, j, k;

   puts( "- 3b dizi listeleme-" );
   printf("sizeof  Adr3b : %u, sizeof *Adr3b : %u, ",  sizeof Adr3b, sizeof *Adr3b  );
   printf("sizeof *Adr3b / sizeof (int) : %u\n",       sizeof *Adr3b / sizeof (int) );
   for ( i = 0; i < 3; i++ )
   {
        for ( j = 0; j < 4; j++ )
        {
             for ( k = 0; k < 5; k++ )
                  printf( "%d ", Adr3b[ i ][ j ][ k ] );
             putchar( '\n' );
        }
        putchar( '\n' );
   }
}
void Liste4b( int (*Adr4b)[ 3 ][ 4 ][ 5 ] )
{
   int i, j, k, l;

   puts( "- 4b dizi listeleme-" );
   printf("sizeof   Adr4b: %u, sizeof *Adr4b  : %u, ",  sizeof Adr4b, sizeof *Adr4b  );
   printf("sizeof *Adr4b / sizeof (int) : %u\n",        sizeof *Adr4b / sizeof (int) );

   for ( i = 0; i < 2; i++ )
   {
        for ( j = 0; j < 3; j++ )
        {
             for ( k = 0; k < 4; k++ )
             {
                  for ( l = 0; l < 5; l++ )
                       printf( "%d ", Adr4b[ i ][ j ][ k ][ l ] );
                  putchar( '\n' );
             }
             putchar( '\n' );
        }
```

... Örnek 6.2-1 devam

```
        putchar( '\n' );
   }
}
void Degis2b( int (*a2b)[ 5 ] )
{
   int t[ 5 ];
   memcpy( t,    a2b+1,  sizeof t    );
   memcpy( a2b+1, a2b,   sizeof a2b[1] );
   memcpy( a2b,  t,      sizeof a2b[0] );
}
```

Çıktı

```
sizeof iDizi2b : 40
- 2b dizi -
sizeof  Adr2b : 2, sizeof *Adr2b : 10, sizeof *Adr2b / sizeof (int) : 5
0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1
sizeof iDizi3b : 120
- 3b dizi -
sizeof  Adr3b : 2, sizeof *Adr3b : 40, sizeof *Adr3b / sizeof (int) : 20
0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1

0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1

0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1

sizeof iDizi4b : 240
- 4b dizi -

sizeof  Adr4b : 2, sizeof *Adr4b : 120, sizeof *Adr4b / sizeof (int) : 60
0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1

0 1 2 3 4
```

... Örnek 6.2-1 Çıktı devam

```
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1

0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1


0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1

0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1

0 1 2 3 4
5 6 7 8 9
2 3 4 5 6
7 8 9 0 1

2b dizinin 1. satiri ile 2. satirinin degistirilmesi:
- 2b dizi -
sizeof  Adr2b : 2, sizeof *Adr2b : 10, sizeof *Adr2b / sizeof (int) : 5
5 6 7 8 9
0 1 2 3 4
2 3 4 5 6
7 8 9 0 1
```

Programda tanımlanan 2-b dizi iDizi2b'nin bir satırı, 5 *tamsayıdan oluşan dizi*dir (int[5] tipinde 4 dizi bulunduğu için, satır sayısı : 4). Bu nedenle aktarılan adres değeri 5 tamsayıdan oluşan diziye işaret eder ve dolayısıyla int(*)[5] tipindedir (Şekil 6.2-1). Parametre bildirimleri aşağıdaki üç şekilde de yapılabilir:

Dizi şeklinde bildirim:

```
void Liste2b( int Adr2b [4] [5] )
{ . . . }
```

İlk boyut değeri verilmeden dizi şeklinde bildirim:

```
void Liste2b( int Adr2b [ ] [ 5 ] )
{ . . . }
```

Dizinin satırlarına işaret eden adres değişkeni şeklinde bildirim:

```
void Liste2b( int (*Adr2b) [ 5 ] )
{ . . . }
```

iDizi3b'nin bir satırı ise *5 tamsayıdan oluşan 4 dizi*dir (int[4][5] tipinde 3 dizi bulunduğu için satır sayısı : 3). Aktarılan adres değeri 5 tamsayıdan oluşan 4 dizinin oluşturduğu diziye işaret eder ve int(*)[4][5] tipinde olacaktır. Aşağıdaki parametre bildirimlerinin üçünde de dizinin satırlarına işaret eden adres değişkeni bildirilir:

Dizi şeklinde bildirim:

```
void Liste3b( int Adr3b [ 3 ] [ 4 ] [ 5 ] )
{ . . . }
```

İlk boyut değeri verilmeden dizi şeklinde bildirim:

```
void Liste3b( int Adr3b [ ] [ 4 ] [ 5 ] )
{ . . . }
```

Dizinin satırlarına işaret eden adres değişkeni şeklinde bildirim:

```
void Liste3b(  int (*Adr3b) [ 4 ] [ 5 ] )
{ . . . }
```

iDizi4b'nin bir satırı, *5 tamsayıdan oluşan 4 dizinin oluşturduğu 3 diziden oluşan dizi*dir (int[3][4][5] tipinde 2 dizi bulunduğu için satır sayısı : 2). Çağrı sırasında aktarılan adres değeri bu dizilere işaret eder ve int(*)[3][4][5] tipindedir.
Aşağıdaki her üç parametre bildirimi de aynıdır ve int(*)[3][4][5] tipinde Adr4b adres değişkeni bildirilir:

Dizi şeklinde bildirim:

```
void Liste4b( int Adr4b [ 2 ] [ 3 ] [ 4 ] [ 5 ] )
{ . . . }
```

İlk boyut değeri verilmeden dizi şeklinde bildirim:

```
void Liste4b( int Adr4b [ ] [ 3 ] [ 4 ] [ 5 ] )
{ . . . }
```

Dizinin satırlarına işaret eden adres değişkeni şeklinde bildirim:

```
void Liste4b( int (*Adr4b) [ 3 ] [ 4 ] [ 5 ] )
{ . . . }
```

Sonuç olarak parametre bildirimlerinde dizinin alt-dizilerini oluşturan satırların sayısını veren ilk boyut değeri gereksizdir. İzleyen boyut değerleri bir satırın uzunluğunu verdiği için parametre bildiriminde verilmelidir. Bildirim üç şekilde de yapılabilir. Fakat

programlarda parametre bildiriminin dizi şeklinde ilk boyut değeri verilmeden yapılması, aktarılan adres değerinin bir diziye ait olduğunu belirttiği için tercih edilebilir.

Programın çıktısında da görüldüğü gibi Adr2b, Adr3b ve Adr4b, bu programın oluşturulduğu donanımda 2-byte büyüklüğünde adres değişkenleridir. Fakat bu adres değişkenlerinin indirek değerlerine uygulanan **sizeof** operatörünün çıktılarında da görüldüğü gibi tipleri farklıdır (farklı uzunlukta dizi satırlarına işaret ederler). Dolayısıyla bu adres değişkenlerinden herhangi birine herhangi bir tamsayı değer eklendiğinde, eklenen değer adres aritmetiğine göre bir satırın byte uzunluğu ile ölçeklenir ve adres değişkeni bir sonraki satıra işaret eder. Örnek olarak iDizi2b'nin satırlarına işaret eden Adr2b adres değişkenine eklenen tamsayı 1 değeri, adres aritmetiğine göre otomatik olarak aşağıdaki şekilde ölçeklenir:

1 * *satır uzunluğu* yada 1 * sizeof (*Adr2b) yada 1 * 5 * sizeof (int)

Adr3b için:

1 * *satır uzunluğu* yada 1 * sizeof (*Adr3b) yada 1 * 4 * 5 * sizeof (int)

Adr4b için:

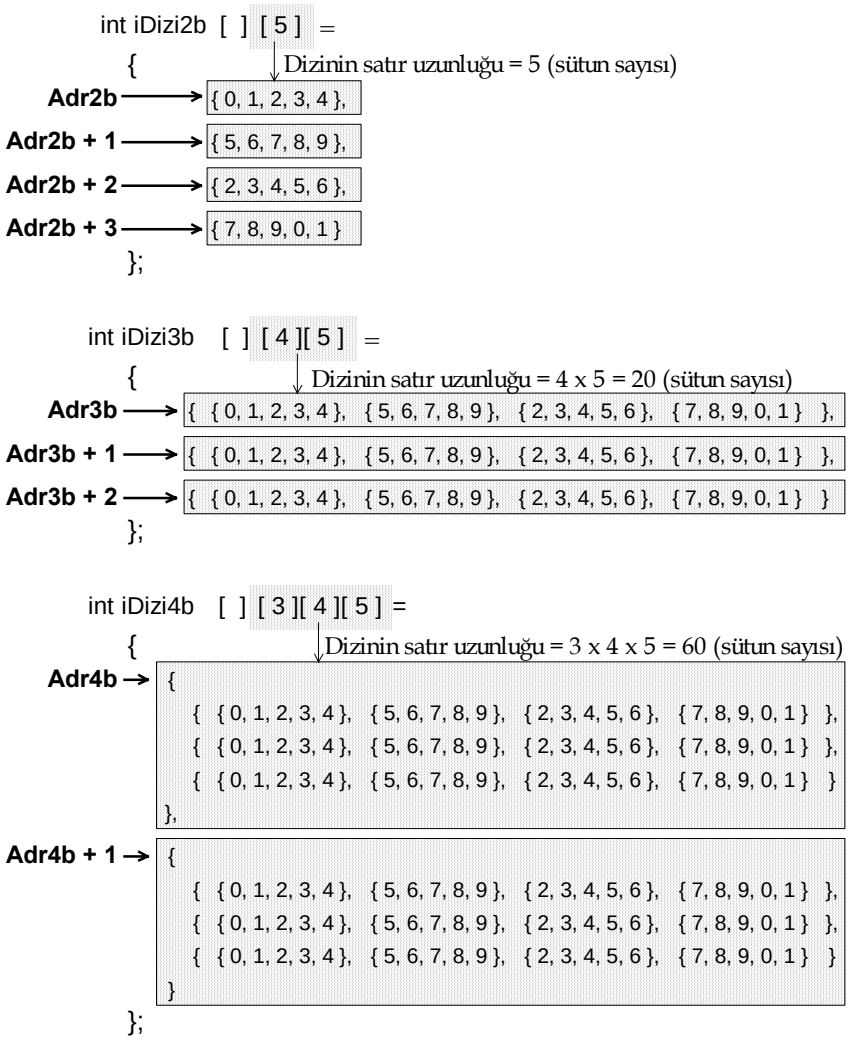
1 * *satır uzunluğu* yada 1 * sizeof (*Adr4b) yada 1 * 3 * 4 * 5 * sizeof (int)

Örnek programda yer alan fonksiyon tanımlarında, dizi elemanlarının değerlerini adres değişkenleri aracılığı ile listeleyen **for** döngülerinde, döngü sayacı sınırları sabit değerler olarak verildi.

Bu değerler fonksiyonlara, size_t tipinde bildirilen parametrelerle aktarılabilir. Böylece ilgili fonksiyonun her çağrısında, aynı boyut sayısına sahip fakat farklı sayıda elemanı olan diziler listelenebilir.

Dizi satırlarına işaret eden adres değişkeni parametrelerin bildiriminde parantezler gereklidir çünkü indeks operatörü ([ ]), indirek değer operatöründen (*) daha yüksek önceliklidir. Parantezler kullanılmadığı taktirde bildirimler adres dizisi bildirimleri olacaktır.

Şekil 6.2-1 iDizi2b, iDizi3b ve iDizi4b dizilerinin satırlarını oluşturan alt-diziler ve bu alt-dizilere işaret eden adres değişkenleri.



Dizilerin indekslenmesi işleminin (*array subscripting*) tanımına göre *dizi_ismi*[*indeks*] ifadesi, *(*dizi_ismi* + *indeks*) ifadesine eştir ve bu değişme özelliği olan bir işlemdir.

Örnek programda yer alan çok-boyutlu dizi isimleri ile aşağıdaki birbirine eş ifadeler oluşturulabilir:

İndeks ifadeleri:

```
iDizi2b [ i ] [ j ]
iDizi3b [ i ] [ j ] [ k ]
iDizi4b [ i ] [ j ] [ k ] [ l ]
```

Adres ve offset ifadeleri:

*(iDizi2b [i] + j)		*(*(iDizi2b + i) + j)
*(iDizi3b [i] [j] + k)		*(*(*(iDizi3b + i) + j) + k)
*(iDizi4b [i] [j] [k] + l)		*(*(*(*(iDizi4b + i) + j) + k) + l)

İkinci grupta yer alan ifadeler işlenirken iDizi2b [i], iDizi3b [i] [j] ve iDizi4b [i] [j] [k] ifadeleri dizi tipindedir ve *tamsayılardan oluşan diziye işaret eden sabit adres değeri*ne çevrilir. Bu ifadelere eklenen j, k ve l değerleri ise adres aritmetiğine göre bir tamsayının byte sayısı (**sizeof** (**int**)) ile çarpılır. Bu ifadeler, dizi listeleme fonksiyonları içinde tanımlanan adres değişkenleri kullanılarakta oluşturulabilir. ■

BÖLÜM 7: **Dinamik Bellek Yönetimi**

C dilinde, bir dizinin büyüklüğü derleme sırasında (*compile-time*) sabittir. Dolayısıyla dizi bildirilirken, programda gerekli olabilecek maksimum sayıda elemanı saklayacak kadar büyük olduğundan emin olunmalıdır. Bu nedenle büyük miktarda bellek alanı programın çalışma süresinin çoğunda kullanılmadan bekler. Sonuç olarak, gerek duyulduğunda istenen miktarda bellek alanını programın çalışması sırasında ayırmak (*allocation*) ve gerek kalmadığı zaman bu alanı başka kullanımlar için serbest bırakmak (*deallocation*) oldukça verimli olacaktır.

Programın çalışması sırasında (*run-time*) bellek alanı ayırma işlemi *dinamik bellek yerleştirme* (*dynamic memory allocation*) olarak; bu amaçla yapılan işlemler ise *dinamik bellek yönetimi* (*dynamic memory management*) olarak adlandırılır. Programlarda static ve global veriler belleğin *static veri alanı*na; otomatik veriler ise *stack alanı*na yerleştirilir. Dinamik veriler ise belleğin *heap* olarak adlandırılan kısmına yerleştirilir. Dinamik bellek alanına erişim, bu amaçla kullanılan fonksiyonlar tarafından döndürülen ve ayrılan bellek bloğunun başlangıcına işaret eden adres değerleri aracılığıyla gerçekleşir.

Bu bölümde, dinamik bellek yönetimi için kullanılan standart kütüphane fonksiyonları (malloc, calloc, realloc, ve free) ve diziler için dinamik bellek alanı ayrılması anlatılacaktır. Ayrıca bu amaçla çeşitli fonksiyonlar tanımlanacaktır. ■

7.1 Standart C'de Dinamik Bellek Kullanımı

Standart C kütüphanesinde dinamik bellek kullanımı için malloc, calloc, realloc ve free fonksiyonları bulunur. Bu fonksiyonların prototipleri ise stdlib.h başlık dosyasında bulunur.

malloc fonksiyonu, *n* byte büyüklüğünde bellek alanı ayırır ve hatasız çalışır ise bu alanın başlangıç adresini döndürür. Döndürülen adres değeri herhangi bir adres tipine (indirek değerinin büyüklüğü *n*'den fazla olmayan) çevrilebilir. Hata durumunda boş adres değeri (NULL; *null pointer*) döndürür.

```
void *malloc( size_t  n );                    /*  stdlib.h  */
```

malloc tarafından ayrılan bellek alanında, herhangi bir değer atanmadan önce bulunan değerler belirsizdir.

calloc fonksiyonu ise, her biri *n* büyüklüğünde *el_sayi* kadar elemandan oluşan dizi için bellek alanı ayırır (*el_sayi* * *n* byte); dizinin her byte'ına 0 (sıfır) değerini atar ve dizinin başlangıç elemanına işaret eden adres değerini döndürür. Hata durumunda boş adres değeri döndürür. Döndürülen adres değeri, byte olarak büyüklüğü *n*'den fazla olmayan herhangi bir tipe işaret eden adres tipine çevrilebilir.

```
void *calloc( size_t el_sayi, size_t  n );    /*  stdlib.h  */
```

realloc fonksiyonu önceki malloc, calloc yada realloc çağrısı ile ayrılmış olan heap alanını genişletmek yada daraltmak için kullanılır. Dolayısıyla ilk argüman olarak önceki malloc, calloc yada realloc çağrısının döndürdüğü adres değerini alır (*adr*). realloc fonksiyonu istenen miktarda bellek alanı (*n* büyüklüğünde) ayırabiliyor ise, orijinal verileri (*adr* tarafından işaret edilen bellek alanında bulunan) yeni ayrılan alana kopyalar ve yeni adresi döndürür. Önceki bellek alanı serbest bırakılır. Yeni ayrılan bellek alanı öncekinden daha büyük ise, önceki veriler bu alanın başlangıcından itibaren kopyalanır. Fakat artan bellek alanındaki değerler belirsizdir. Eğer daha küçük bir bellek alanı ayrılıyor ise, önceki verilerin sadece bu alana sığabilecek miktarı kopyalanır. realloc çağrısı, *adr*'nin değeri boş adres değeri olduğunda malloc(*n*) çağrısı ile aynıdır (orijinal veriler ayrılan bellek alanına kopyalanmaz).

```
void *realloc( void *adr, size_t  n );        /*  stdlib.h  */
```

Döndürdüğü adres değerine yine diğer fonksiyonlarda olduğu gibi tip çevirme uygulanabilir. Hata durumunda boş adres değeri döndürür ve önceki ayrılan bellek alanı değişmeden kalır.

```
void free( void *adr );                       /*  stdlib.h  */
```

Eğer *adr* boş adres değişkeni değil ise, işaret ettiği bellek alanı serbest bırakılır. Aksi taktirde fonksiyon hiç birşey yapmaz. Sadece malloc, calloc yada realloc çağrısı ile ayrılmış olan bellek alanı serbest bırakılır. Programdan çıkılırken free çağrısı gerekmeyebilir. Fakat programın anlaşılabilirliği açısından mutlaka ayrılmış bulunan bellek alanları free fonksiyonu çağrılarak serbest bırakılmalıdır.

Örnek 6.1-1'de 2-b dizi iDizi2b'ye işaret eden adres değişkeni Adr2b bildirildi. Adr2b adres değişkeni, 2-b dizinin satırlarını oluşturan 1-b dizilere işaret eder ve **int**(*)[5] tipindedir. Örnek 6.1-2'de ise 3-b dizi iDizi3b'ye işaret eden Adr3b adres değişkeni bildirilir. Bu adres değişkeni **int**(*)[4][5] tipindedir ve iDizi3b'nin satırlarını oluşturan 2-b dizilere işaret eder. Örnek 7.1-1'de 2-b ve 3-b diziler için dinamik olarak bellek alanı ayrılır ve bu alanların başlangıc adresleri 2-b ve 3-b dizilere işaret eden adres değişkenlerine atanır. Ayrılan bellek alanlarına değer atamak için rastgele sayı üreten standart kütüphane fonksiyonu rand kullanılmıştır:

```
int rand( void  );                                  /*  stdlib.h  */
```

Örnek 7.1-1 din2b3b.c programı.

```
#include   <stdio.h>
#include   <stdlib.h>

#define    NL                  '\n'
#define    RandomSayi(min, max)  \
           ((rand() % (int)(((max)+1) - (min))) + (min))

#define    NoSatir2b      4
#define    NoSutun2b      5

#define    NoTablo3b      3
#define    NoSatir3b      4
#define    NoSutun3b      5

int main(void)
{
  int  i, j, k,
       dNoSatir2b   = NoSatir2b,
       dNoTablo3b  = NoTablo3b;
  int (*Adr2b)[ NoSutun2b ], (*Adr3b) [ NoSatir3b ] [ NoSutun3b ];

  /*  2-b ve 3-b diziler icin dinamik bellek alani ayrilmasi.  */
  Adr2b= ( int(*)[ NoSutun2b ] )malloc(   dNoSatir2b*
                                          NoSutun2b *  sizeof (int) );
  if ( Adr2b == NULL )
  {
       fprintf( stderr, "Bellek alani ayrilamadi !!!\n" );
       exit( 1 );
```

... Örnek 7.1-1 devam

```
    }
    Adr3b= ( int(*)[ NoSatir3b ][ NoSutun3b ] )malloc(     dNoTablo3b  *
                                                            NoSatir3b      *
                                                            NoSutun3b      *
                                                            sizeof (int) );
    if ( Adr3b == NULL )
    {
        fprintf( stderr, "Bellek alani ayrilamadi !!!\n" );
        exit( 2 );
    }

    for ( i = 0; i < dNoSatir2b; i++ )
        for ( j = 0; j < NoSutun2b; j++ )
            Adr2b [ i ] [ j ] = RandomSayi( 0, 9 );

    for ( i = 0; i < dNoTablo3b; i++ )
        for ( j = 0; j < NoSatir3b; j++ )
            for ( k = 0; k < NoSutun3b; k++ )
                Adr3b [ i ] [ j ] [ k ] = RandomSayi( 0, 9 );

    /* Elemanlarin degerlerinin listelenmesi. */
    puts( "\n2-b dizi : " );
    for ( i = 0; i < dNoSatir2b; i++ )
    {
        for ( j = 0; j < NoSutun2b; j++ )
            printf( "%d ", *(*(Adr2b + i) + j) );
        putchar( NL );
    }
    puts( "\n3-b dizi : " );
    for ( i = 0; i < dNoTablo3b; i++ )
    {
        for ( j = 0; j < NoSatir3b; j++ )
        {
            for ( k = 0; k < NoSutun3b; k++ )
                printf( "%d ", *(*(*(Adr3b + i) + j) + k) );
            putchar( NL );
        }
        putchar( NL );
    }
    free( Adr2b );
    free( Adr3b );
    return 0;
}
```

... Örnek 7.1-1 devam

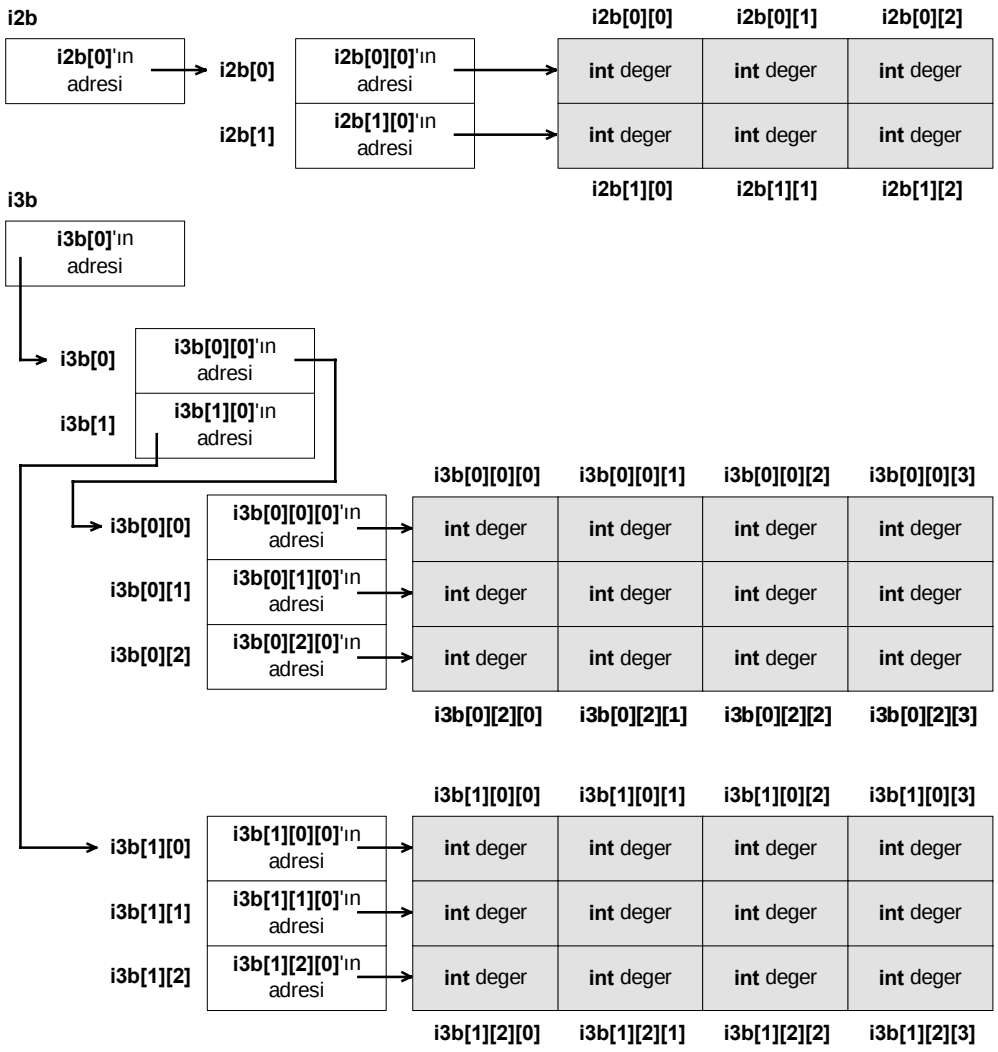
Çıktı

```
2-b dizi :
1 7 4 0 9
4 8 8 2 4
5 5 1 7 1
1 5 2 7 6

3-b dizi :
1 4 2 3 2
2 1 6 8 5
7 6 1 8 9
2 7 9 5 4

3 1 2 3 3
4 1 1 3 8
7 4 2 7 7
9 3 1 9 8

6 5 0 2 8
6 0 2 4 8
6 5 0 9 0
0 6 1 3 8
```

Programda da görüldüğü gibi malloc fonksiyonu dizilerin tamamı için bellek alanı ayırır ve bu alanların başlangıcına işaret eden adres değerini döndürür. Döndürülen adres değeri, dizilerin satırlarına işaret eden adres değeri tipine çevrilerek yine bu şekilde bildirilmiş olan adres değişkenlerine atanır. Erişim, bu adres değişkenleri aracılığı ile gerçekleştirilir.

Program derlenirken, adres değişkenleri bildirimlerinde kullanılan NoSutun2b, NoSatir3b ve NoSutun3b değerlerinin sabit olması gerekmektedir.

Şekil 7.1-1 İki-boyutlu i2b dizisi ve üç-boyutlu i3b dizisi.



Ancak 2-b dizinin satır sayısını belirten NoSatir2b ve 3-b dizinin satır sayısını belirten NoTablo3b değerleri, programın çalışması sırasında belirlenebilir (programda bunu ifade etmek için aynı isimde fakat d harfi ile başlayan dNoSatir2b ve dNoTablo3b değişkenleri tanımlanır ve malloc çağrılarında kullanılır). Dolayısıyla bu teknik,

dizilerin tüm boyut değerlerinin program derlenirken belli olmadığı durumlarda dinamik bellek alanı ayırmak için kullanılamaz.

2-b ve 3-b diziler için dinamik bellek alani ayrılması

Şekil 7.1-1'de şematik olarak gösterildiği gibi 2-b ve 3-b dizi bildirimleri ile dizi elemanları ve bir grup adres sabiti (*pointer constant*) oluşturulur.

İki-boyutlu i2b bildirimi:

```
int i2b [ 2 ] [ 3 ];
```

Üç-boyutlu i3b bildirimi:

```
int i3b [ 2 ] [ 3 ] [ 4 ];
```

Şekilde görüldüğü gibi 2-b i2b dizisi bildirimi ile i2b, i2b[0] ve i2b[1] adres sabitleri oluşur. Ayrıca 6 tamsayı değer için bellek alanı ayrılır. i3b dizisi bildirimi ile i3b, i3b[0], i3b[1], i3b[0][0], i3b[0][1], i3b[0][2], i3b[1][0], i3b[1][1], ve i3b[1][2] adres sabitleri oluşur. Bunlardan hiç biri lvalue değildir. İfadelerde sabit adres değerlerine dönüşürler. Ayrıca i3b bildirimi ile 24 tamsayı değer için bellek alanı ayrılır.

Dizi ismi i2b'ye iki kez indirek değer operatörü uygulandığında dizide bulunan ilk tamsayı değer elde edilir (**i2b). i2b, alt-dizilerin başlangıç elemanı olan tamsayı değerlerin adreslerini veren adres sabitleri dizisine işaret eder. i2b[...] ifadeleri dizinin satırlarını oluşturan 1-b dizilere işaret eder ve *i2b[...] ifadeleri başlangıç elemanlarının değerlerini verir.

3-b dizide bulunan ilk elemanın değerini almak için ise dizi ismi i3b'ye indirek değer operatörü üç kez uygulanmalıdır (***i3b). i3b, çift adres sabitleri dizisine (i3b[...]) işaret eder. i3b[...] ifadeleri ise, alt-dizilerin başlangıç adreslerini veren i3b[...][...] adres sabiti dizilerine işaret eder. Buna göre, **i3b[...] ifadeleri ve *i3b[...][...] ifadeleri alt-dizilerin başlangıç elemanlarının değerlerini verir.

Bu ifadelerle oluşturulan aşağıdaki örnek programda tamsayı değerler ekrana listelenir:

```
/*
 * i2b3b.c programi.
 */
#include <stdio.h>

int main(void)
{
   int i, j;
   int i2b [ 2 ] [ 3 ]        = {
                                  9,1,2,
                                  7,4,5
```

... Örnek i2b3b.c devam

```
                                    };
    int i3b [ 2 ] [ 3 ] [ 4 ]  = {
                                      8,1,0,0,
                                      7,2,1,0,
                                      6,3,2,1,
                                      5,4,3,2,
                                      4,5,4,3,
                                      3,6,5,4
                                    };

    printf( "**i2b : %d\n", **i2b );
    for ( i = 0; i < 2; i++ )
          printf( "*i2b[%d] : %d\n", i, *i2b[i] );
    putchar( '\n' );

    printf( "***i3b : %d\n", ***i3b );
    for ( i = 0; i < 2; i++ )
          printf( "**i3b[%d] : %d\n", i, **i3b[i] );
    putchar( '\n' );

    for ( i = 0; i < 2; i++ )
          for ( j = 0; j < 3; j++ )
                printf( "*i3b[%d][%d] : %d\n", i, j, *i3b[i][j] );

    return 0;
}
```

Cikti :

```
        **i2b     : 9
        *i2b[0]   : 9
        *i2b[1]   : 7

        ***i3b    : 8
        **i3b[0]  : 8
        **i3b[1]  : 5

        *i3b[0][0]  : 8
        *i3b[0][1]  : 7
        *i3b[0][2]  : 6
        *i3b[1][0]  : 5
        *i3b[1][1]  : 4
        *i3b[1][2]  : 3
```

2-b ve 3-b diziler için programın çalışması sırasında dinamik bellek alanı kullanmak için aynı erişim sistemi adres değişkenleri ile oluşturulmalıdır. Örnek 7.1-2 ve Örnek 7.1-3'de 2-b ve 3-b diziler için bu şekilde dinamik bellek alanı ayrılır.

Örnek 7.1-2 din2b.c programı.

```
#include <stdio.h>
#include <stdlib.h>

#define    NL     '\n'
#define    RandomSayi(min, max)    ((rand() % (int)(((max)+1) - (min))) + (min))

void List2b( int **Adr2b, int NoSatir2b, int NoSutun2b );

int main(void)
{
   int i, j, dNoSatir2b = 2, dNoSutun2b = 3;
   int **Adr2b;

   if ( (Adr2b = (int **)malloc( dNoSatir2b * sizeof (int*) )) == NULL )
   {
        fprintf( stderr, "Bellek alani ayrilamadi !!!\n" );
        exit(1);
   }

   for ( i = 0; i < dNoSatir2b; i++ )
        if ( (Adr2b[ i ] = (int *) malloc( dNoSutun2b * sizeof (int) )) == NULL )
        {
             fprintf( stderr, "Bellek alani ayrilamadi !!!\n" );
             exit(2);
        }

   for ( i = 0; i < dNoSatir2b; i++ )
        for ( j = 0; j < dNoSutun2b; j++ )
             Adr2b[ i ][ j ] = RandomSayi( 0, 9 );

   for ( i = 0; i < dNoSatir2b; i++ )
   {
        for ( j = 0; j < dNoSutun2b; j++ )
             printf( "%d ", *(*(Adr2b + i ) + j ) );
        putchar( NL );
   }
   List2b( Adr2b, dNoSatir2b, dNoSutun2b );

   free( Adr2b );
   for ( i = 0; i < dNoSatir2b; i++ )
        free( Adr2b[ i ] );

   return 0;
}

void   List2b( int **Adr2b, int NoSatir2b, int NoSutun2b )
{
        int i, j;
```

... Örnek 7.1-2 devam

```
        for ( i = 0; i < NoSatir2b; i++ )
        {
            for ( j = 0; j < NoSutun2b; j++ )
                printf( "%d ", Adr2b[ i ][ j ] );
            putchar( NL );
        }
}
```

Çıktı

```
1 7 4
0 9 4
1 7 4
0 9 4
```

Programda görüldüğü gibi 2-b dizinin satır ve sütun sayıları (dNoSatir2b ve dNoSutun2b) tamsayı değişkenlerdir ve programın çalışması sırasında girilebilir.

Dinamik 2-b dizi için ilk olarak bir çift adres değişkeni (*double pointer*) bildirilir:

```
int **Adr2b;
```

Bu adres değişkenine, dNoSatir2b sayısı kadar **int*** tipinde adres değişkeni için malloc çağrılarak ayrılan bellek alanının başlangıç adresi **int**** tipine çevrilerek atanır:

```
Adr2b = (int **)malloc( dNoSatir2b * sizeof (int*) )
```

Böylece 2-b dizinin satırlarına işaret eden adres değişkenleri için bellek alanı ayrılmış olur. Bu alanda, 2-b dizinin satırlarının başlangıç adreslerinden oluşan adres dizisi saklanacaktır. Çift adres değişkeni bu alana işaret ettiği için indeks operatörü ile oluşturulan Adr2b[i] ifadesi de bu alana erişir. **for** döngüsü içinde bir başka malloc çağrısı yer alır:

```
Adr2b[ i ] = (int *)malloc( dNoSutun2b * sizeof (int) )
```

Bu çağrı ile tamsayı değerleri saklamak için ayrılan dNoSatir2b sayısı kadar dNoSutun2b*sizeof (int) uzunluğunda her alanın başlangıç adresi **int*** tipine çevrilerek Adr2b[i] ile erişilen ve önceki malloc çağrısında ayrılmış olan bellek alanına atanır. Böylece dizi elemanları ve dizi elemanlarına erişim için gerekli olan tüm adres değişkenleri için bellek alanları ayrılmış olur. RandomSayi makrosu ile üretilen 0 - 9 arası tamsayı değerler, Adr2b[i][j] ile erişilen bellek alanlarına atanır. Daha sonra dizi elemanlarının değerleri ekrana listelenir. Listeleme, List2b fonksiyonunda olduğu gibi uygun şekilde tanımlanmış bir fonksiyona çift adres değişkeni değeri aktarılarak gerçekleştirilebilir.

Örnek 7.1-3 din3b.c programı.

```
#include <stdio.h>
#include <stdlib.h>

#define    NL     '\n'
#define    RandomSayi(min, max)  ((rand() % (int)(((max)+1) - (min))) + (min))

void List3b( int ***Adr3b, int NoTablo3b, int NoSatir3b, int NoSutun3b );

int main(void)
{
   int i, j, k, dNoTablo3b = 2, dNoSatir3b = 3, dNoSutun3b = 4;
   int ***Adr3b;

   if ( (Adr3b = (int ***)malloc( dNoTablo3b * sizeof (int**))) == NULL )
   {
        fprintf( stderr, "Bellek alani ayrilamadi !!!\n" );
        exit( 1 );
   }

   for ( i = 0; i < dNoTablo3b; i++ )
   {
        if ( (Adr3b[ i ] = (int **)malloc( dNoSatir3b * sizeof (int*))) == NULL )
        {
             fprintf( stderr, "Bellek alani ayrilamadi !!!\n" );
             exit( 2 );
        }
        for ( j = 0; j < dNoSatir3b; j++)
        {
             if ( (Adr3b[ i ][ j ] = (int *)malloc( dNoSutun3b * sizeof (int))) == NULL )
             {
                  fprintf( stderr, "Bellek alani ayrilamadi !!!\n" );
                  exit( 3 );
             }
        }
   }

   for ( i = 0; i < dNoTablo3b; i++ )
        for ( j = 0; j < dNoSatir3b; j++ )
             for ( k = 0; k < dNoSutun3b; k++ )
                  Adr3b[i ][ j ][ k ] = RandomSayi( 0, 9 );

   for ( i = 0; i < dNoTablo3b; i++ )
   {
        for ( j = 0; j < dNoSatir3b; j++ )
        {
             for ( k = 0; k < dNoSutun3b; k++ )
                  printf( "%d ", Adr3b[i ][ j ][ k ] );
             putchar( NL );
        }
        putchar( NL );
```

... Örnek 7.1-3 devam

```
  }
  List3b( Adr3b, dNoTablo3b, dNoSatir3b, dNoSutun3b );
  free( Adr3b );
  for ( i = 0; i < dNoTablo3b; i++ )
  {
      free ( Adr3b[ i ] );
      for ( j = 0; j < dNoSatir3b; j++)
          free( Adr3b[ i ][ j ] );
  }
  return 0;
}
void List3b( int ***Adr3b, int NoTablo3b, int NoSatir3b, int NoSutun3b )
{
    int i, j, k;
    for ( i = 0; i < NoTablo3b; i++ )
    {
        for ( j = 0; j < NoSatir3b; j++ )
        {
            for ( k = 0; k < NoSutun3b; k++ )
                printf( "%d ", Adr3b[i ][ j ][ k ] );
            putchar( NL );
        }
        putchar( NL );
    }
}
```

Çıktı

```
1 7 4 0
9 4 8 8
2 4 5 5

1 7 1 1
5 2 7 6
1 4 2 3

1 7 4 0
9 4 8 8
2 4 5 5

1 7 1 1
5 2 7 6
1 4 2 3
```

3-b dizi için ise, ilk olarak çift adres değişkenleri dizisine işaret eden bir üçlü adres değişkeni (*triple pointer*) bildirilir:

```
int ***Adr3b;
```

Bu adres değişkenine çift adres değişkenleri için ayrılan bellek alanının başlangıç adresi atanır:

```
Adr3b = (int ***)malloc( dNoTablo3b * sizeof (int**))
```

Tekrar malloc fonksiyonu çağrılarak dNoTablo3b sayısı kadar adres dizisi için (herbiri dNoSatir3b sayısı kadar adres değişkeninden oluşan) bellek alanı ayrılır:

```
Adr3b[ i ] = (int **)malloc( dNoSatir3b * sizeof (int*))
```

Bu alanların başlangıç adresleri önceki malloc çağrısı ile çift adres değişkenleri dizisi için ayrılmış bulunan bellek alanına, Adr3b[i] indeks ifadesi ile erişilerek atanır. Yine aynı döngü içinde çift adres dizisinin elemanlarının işaret ettiği adres dizilerinin elemanlarına, tamsayı değerler için ayrılan bellek alanlarının başlangıç adresleri atanır:

```
Adr3b[ i ][ j ] = (int *)malloc( dNoSutun3b * sizeof (int))
```

Böylece 3-b dinamik dizinin elemanları ve elemanlarına erişim için gerekli olan tüm adres değişkenleri için bellek alanı ayrılmış olur. Yine önceki örnekte olduğu gibi, ayrılan bellek alanlarına atanan rastgele tamsayı değerler ekrana listelenir.

Örnek 7.1-4'de 2-b tamsayı dizilere dinamik bellek alanı ayırmak için Din2b fonksiyonu tanımlanır:

Örnek 7.1-4 dinfonk.c programı.

```
#include <stdio.h>
#include <stdlib.h>

#define   NL              '\n'
#define   RandomSayi(min, max)  ((rand() % (int)(((max)+1) - (min))) + (min))
#define   MAXSATIR   10
#define   MAXSUTUN   10

int     **Din2b(       int NoSatir2b, int NoSutun2b                    );
void    Free(          int **Adr2b, int NoSatir2b                      );
void    DegerAta2b(    int **Adr2b, int NoSatir2b, int NoSutun2b       );
void    List2b(        int **Adr2b, int NoSatir2b, int NoSutun2b       );

int main(void)
{
  int dNoSatir2b = 0, dNoSutun2b = 0;
  int **Adr2b;

  printf( "Satir Sayisi ? " );
  scanf(  "%d", &dNoSatir2b );
  if ( dNoSatir2b > MAXSATIR )    dNoSatir2b    = MAXSATIR;

  printf( "Sutun Sayisi ? " );
  scanf( "%d", &dNoSutun2b );
```

... Örnek 7.1-4 devam

```
   if ( dNoSutun2b > MAXSUTUN ) dNoSutun2b   = MAXSUTUN;
   if ( (Adr2b = Din2b( dNoSatir2b, dNoSutun2b )) == NULL )
   {
        fprintf( stderr, "Din2b: Bellek alani ayrilamadi !!!\n" );
        exit(1);
   }
   DegerAta2b(  Adr2b,  dNoSatir2b, dNoSutun2b );
   List2b(      Adr2b,  dNoSatir2b, dNoSutun2b );
   Free  (      Adr2b,  dNoSatir2b );
   return 0;
}
int **Din2b( int NoSatir2b, int NoSutun2b )
{
   int i;
   int **Adr2b;
   if ( (Adr2b = (int **)malloc( NoSatir2b * sizeof (int *) )) == NULL )
        return (int**)NULL;
   for ( i = 0; i < NoSatir2b; i++ )
        if ( (Adr2b[ i ] = (int *)malloc( NoSutun2b * sizeof (int) )) == NULL )
             return (int**)NULL;
   return Adr2b;
}
void Free( int **Adr2b, int NoSatir2b )
{
   int i;
   free( Adr2b );
   for ( i = 0; i < NoSatir2b; i++ )
        free( Adr2b[ i ] );
}
void DegerAta2b( int **Adr2b, int NoSatir2b, int NoSutun2b )
{
   int i, j;
   for ( i = 0; i < NoSatir2b; i++ )
```

... Örnek 7.1-4 devam

```
        for ( j = 0; j < NoSutun2b; j++ )
             Adr2b[ i ][ j ] = RandomSayi( 0, 9 );
}
void List2b( int **Adr2b, int NoSatir2b, int NoSutun2b )
{
   int i, j;
   for ( i = 0; i < NoSatir2b; i++ )
   {
        for ( j = 0; j < NoSutun2b; j++ )
```

... Örnek 7.1-4 devam

```
            printf( "%d ", Adr2b[ i ][ j ] );
        putchar( NL );
    }
}
```

Bilgi Girişi

```
Satir Sayisi ?   5
Sutun Sayisi ?   20
```

Çıktı

```
1 7 4 0 9 4 8 8 2 4
5 5 1 7 1 1 5 2 7 6
1 4 2 3 2 2 1 6 8 5
7 6 1 8 9 2 7 9 5 4
3 1 2 3 3 4 1 1 3 8
```

Programda malloc çağrılarının döndürdüğü değerler hata durumlarının saptanması için test edilmiştir. Dinamik bellek kullanırken en çok yaşanan problem, ayrılan alanların dışına taşılmasıdır (*memory leak*). Ayrıca programda farkında olmadan artık kullanılmadığı halde serbest bırakılmamış olan bellek alanları, programın ihtiyaç olan bir başka yerinde bellek yetmemesi sorununu doğurabilir.

malloc yada calloc ile ayrılan bellek alanının büyüklüğünü değiştirmek için aşağıdaki programda tanımlanan remalloc fonksiyonu bloğunda görüldüğü gibi ilk olarak yine malloc kullanılarak istenen büyüklükte bellek alanı ayrılır. Önceki malloc yada calloc çağrısı ile ayrılmış olan bellek alanında bulunan veriler bu yeni ayrılan alana kopyalanır (memcpy çağrısı). Son olarak önceki bellek alanı serbest bırakılır.

Örnek 7.1-5 remalloc.c programı.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MaxInt          19
#define BlokBoy         5
#define ToplamByte(x)   ((x+BlokBoy)*sizeof(int))

void *remalloc( void *Adr, size_t YeniBlokByte, size_t EskiBlokByte );

int main(void)
{
  int *Adr = NULL, *OncekiAdr = NULL;
  int i, Onceki = 0;

  for ( i = 0; i < MaxInt; i++ )
  {
```

... Örnek 7.1-5 devam

```
        if ( i % BlokBoy == 0 )
        {
            OncekiAdr = Adr;
            Adr = remalloc( Adr, ToplamByte(i), ToplamByte(Onceki) );
            Onceki = i;
            if ( Adr == NULL )
            {
                free( OncekiAdr );
                fprintf( stderr, "Bellek alani ayrilamadi !!!\n" );
                exit( 1 );
            }
            printf( "%p : %u byte bellek alani ayrildi !!!\n", Adr, ToplamByte(i) );
        }
        Adr[ i ] = i;
    }
    for ( i = 0; i < MaxInt; i++ )
        printf( "%d, ", Adr[ i ] );
    putchar( '\n' );
    free( Adr );
    return 0;
}
void *remalloc( void *Adr, size_t YeniBlokByte, size_t EskiBlokByte )
{
    void *t;

    if ( (t = malloc( YeniBlokByte )) == NULL )
        return NULL;
    if ( Adr != NULL )
    {
      memcpy( t, Adr, ( YeniBlokByte > EskiBlokByte ) ? EskiBlokByte : YeniBlokByte );
      free( Adr );
    }
    return t;
}
```

Çıktı

```
0EFA :  10 byte bellek alani ayrildi !!!
1108 :  20 byte bellek alani ayrildi !!!
111E :  30 byte bellek alani ayrildi !!!
113E :  40 byte bellek alani ayrildi !!!
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
```

Tüm bu işlemler gelişmiş standart kütüphane fonksiyonu realloc kullanılarak gerçekleştirilebilir (Örnek 7.1-6). realloc fonksiyonu önceki çağrıda ayrılan blok yeterli

boşluğa sahip ise yerinde genişletir ve veri kaydırma yapmaz. Programda **for** döngüsü içinde 19 tamsayı değer için dinamik bellek alanı ayrılır ve ayrılan alanlara döngü sayacının değeri atanır. Döngü bloğunda yer alan ilk **if** deyiminde döngü sayacının değerinin 5'in katları olup olmadığı kontrol edilir. Çünkü programda realloc fonksiyonu ayırdığı bellek alanını 5 tamsayılık bloklar halinde artırır. Bu uygulama, her defasında ayrılan alanı bir tamsayılık alan kadar genişletmekten daha verimlidir. Programdaki ilk realloc çağrısına, ilk argüman olarak boş adres değeri (NULL) aktarılır. Bu çağrı malloc çağrısı ile aynıdır.

Örnek 7.1-6 realloc.c programı.

```
#include <stdio.h>
#include <stdlib.h>

#define MaxInt              19
#define BlokBoy         5
#define ToplamByte(x)   ((x+BlokBoy)*sizeof(int))

int main(void)
{
   int *Adr = NULL, *OncekiAdr = NULL;
   int i;

   for ( i = 0; i < MaxInt; i++ )
   {
        if ( i % BlokBoy == 0 )
        {
           OncekiAdr = Adr;
           Adr = realloc( Adr, ToplamByte(i) );
           if ( Adr == NULL )
           {
                free( OncekiAdr );
                fprintf( stderr, "Bellek alani ayrilamadi !!!\n" );
                exit( 1 );
           }
           printf( "%p : %u byte bellek alani ayrildi !!!\n", Adr, ToplamByte(i) );
        }
        Adr[ i ] = i;
   }

   for ( i = 0; i < MaxInt; i++ )
        printf( "%d, ", Adr[ i ] );
   putchar( '\n' );
   free( Adr );

   return 0;
}
```

Çıktı

```
0EFA :  10 byte bellek alani ayrildi !!!
1108  :  20 byte bellek alani ayrildi !!!
1108  :  30 byte bellek alani ayrildi !!!
```

...Örnek 7.1-6 Çıktı devam

```
1108 :  40 byte bellek alani ayrildi !!!
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
```

Ekrandan girilecek olan değer sayısı belli olmayan bir uygulamada, programdaki **for** döngüsü yerine bilgi girişi deyimleri içeren bir döngü yerleştirilir ve döngü içinde bilgi girişi devam ettikçe realloc ile dizi genişletilir. Program döngüden çıkış ve bilgi girişini sona erdirmek için gerekli deyimleri içermelidir (örneğin bilgi girişini sonlandırmak için kullanıcı tarafından özel bir sayı girilmesi istenebilir).

Dizgilere işaret eden adres dizisi ve dinamik bellek kullanımı

Önceki bölümlerde elemanları dizgilere işaret eden adres değişkenleri olan adres dizileri tanımlandı. Adres dizisinin elemanları olan adres değişkenlerinin işaret ettiği farklı uzunluktaki dizgiler için dinamik bellek alanı ayrılabilir. Aşağıdaki örnek programda, farklı uzunluklarda 6 dizgi (*ragged array*) için dinamik bellek alanı ayrılır ve bu alanların başlangıç adresleri adres dizisinin elemanları olan adres değişkenlerine atanır. Programda dizgilerin uzunluklarının ve içerdikleri karakterlerin rastgele olması için RandomSayi makrosu ve RandomDizgi fonksiyonu tanımlanmıştır.

Örnek 7.1-7 adrdindz.c programı.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define   NoSatir                    6
#define   RandomSayi( min, max )   ((rand() % (int)(((max)+1) - (min))) + (min))

char  *RandomDizgi( int BOY );

int main(void)
{
  int     i, dNoSutun;
  char   *RndDizgi;
  char   *AdrDinDz [ NoSatir ];

  for ( i = 0; i < NoSatir; i++ )
  {
      dNoSutun  = RandomSayi( 5, 30 );
      if ( (RndDizgi = RandomDizgi( dNoSutun )) == NULL )
      {
          dNoSutun = 10;
          RndDizgi = "123456789";
      }
      if ( (AdrDinDz [ i ] = (char *)malloc( dNoSutun * sizeof(char) )) == NULL )
      {
          fprintf( stderr, "Bellek alani ayrilamadi !!!\n" );
```

...Örnek 7.1-7 devam

```
            exit( 1 );
        }
        strncpy( AdrDinDz[ i ],  RndDizgi, dNoSutun );
        free ( RndDizgi );
        printf( "AdrDinDz [ %d ] : %s\n", i, AdrDinDz[ i ], AdrDinDz[i] );
    }
    printf( "AdrDinDz [ 2 ]        : %s\n",    AdrDinDz [ 2 ]         );
    printf( "AdrDinDz [ 2 ] [ 3 ] : %c\n",    AdrDinDz [ 2 ] [ 3 ]   );
    printf( "*AdrDinDz             : %s\n",    *AdrDinDz              );
    printf( "(*AdrDinDz)[ 4 ]      : %c\n",    (*AdrDinDz) [ 4 ]      );
    printf( "*(*AdrDinDz + 4)      : %c\n",    *(*AdrDinDz + 4)       );
    for ( i = 0; i < NoSatir; i++ )
        free( AdrDinDz [ i ] );
    return 0;
}
char *RandomDizgi( int BOY )
{
    char *RndDizgi;
    int i;
    if ( (RndDizgi = (char *)malloc( BOY * sizeof (char) )) == NULL )
        return NULL;
    for ( i = 0; i < BOY-1 /* bos karakter icin ilave byte */; i++ )
        RndDizgi[ i ] = (char)(RandomSayi( 97, 122 ));
    RndDizgi[ i ] = '\0';
    printf( "Random Dizgi --> %s\n", RndDizgi );
    return RndDizgi;
}
```

Çıktı

```
Random Dizgi --> hqghumeaylnlfdxfirc
AdrDinDz [ 0 ] :    hqghumeaylnlfdxfirc
Random Dizgi --> scxggbwkfnqduxwfnfozvsrtk
AdrDinDz [ 1 ] :    scxggbwkfnqduxwfnfozvsrtk
Random Dizgi --> prepggxrpnrvy
AdrDinDz [ 2 ] :    prepggxrpnrvy
Random Dizgi --> tmwcysyycqpevikeffmzni
AdrDinDz [ 3 ] :    tmwcysyycqpevikeffmzni
Random Dizgi --> kkasvwsrenzkycxf
AdrDinDz [ 4 ] :    kkasvwsrenzkycxf
Random Dizgi --> tlsgypsfadpooefxzbcoejuvpva
AdrDinDz [ 5 ] :    tlsgypsfadpooefxzbcoejuvpva
AdrDinDz [ 2 ]          :  pre**p**ggxrpnrvy
AdrDinDz [ 2 ] [ 3 ]    :  **p**
*AdrDinDz           :  hqgh**u**meaylnlfdxfirc
```

...Örnek 7.1-7 Çıktı devam

```
(*AdrDinDz)[ 4 ]      : u
*(*AdrDinDz + 4)      : u
```

Bu uygulamanın pratik olmayan yönü, her dizgi için ayrı bir malloc çağrısının gerekiyor olmasıdır. ■

7.2 Yapılar ve Dinamik Bellek Kullanımı

Yapı değişkenleri izleyen bölümde anlatılacaktır. Fakat konu bütünlüğü açısından dinamik yapılara bu bölümde yer verilmiştir. Örnek 8.4-5'de bir adet yapı değişkeni için malloc fonksiyonu kullanılarak dinamik bellek alanı ayrılır. Aşağıdaki örnek programda ise yine tek bir yapı değişkeni, yapi değişkenlerinden oluşan bir yapı dizisi, 2-b yapı dizisi ve elemanları 2-b yapı dizisinin elemanlarına işaret eden 2-b adres dizisi için dinamik bellek alanı kullanımı yer alır. Örneklerde, program kodunun nasıl çalıştığı metinde anlatıldığı için çok fazla açıklama satırına yer verilmemiştir. Aşağıdaki dinstru.c örneğinde program kodu, bir program kodunun anlaşılabilir olmasını sağlamak için açıklama satırlarının etkili kullanımını göstermek amacıyla açıklama satırları ile anlatılmıştır. ■

Örnek 7.2-1 dinstru.c programı.

```
/*
 *    Bu programda  yapilar icin dinamik bellek alani ayrilmasi orneklenmistir.
 */
#include <stdio.h>
#include <stdlib.h>          /* exit  ve  malloc,calloc, free  fonksiyonlari icin */
#include <string.h>          /* strcpy  fonksiyonu icin                           */

#define   NL          '\n'   /* newline karakteri                                 */
#define   BOY         10
#define   ELSayi      5

/* Kayit yapi tipinin tanimlanmasi */
typedef  struct KAYIT
{
  char dizgi [ BOY + 1 ];
  int  i;
}  Kayit;

/* fonksiyon prototipleri */
Kayit  **Din2as       (  int NoSatir2b,        int NoSutun2b   );
Kayit  ***Dinap_2as   (  int NoSatir2b,        int NoSutun2b   );
void   Free2as     (  Kayit    **p_2as,     int NoSatir2b     );
void   Freeap_2as  (  Kayit ***p_ap_2as,  int NoSatir2b     );

int main(void)
{
  int i, j, dNoSatir2b = 2, dNoSutun2b = 3;
```

... Örnek 7.2-1 devam

```
Kayit                  *p_s,   /* pointer to structure                                  */
                       *p_as,  /* pointer to array of structures                        */
                     **p_2as,  /* pointer to 2-d array of structures                    */
                   ***p_ap_2as; /* pointer to 2-d array of pointers to 2-d array of
                                   structures                                           */
/*
   p_s      : Kayit yapisina isaret eden adres degiskeni.
   p_as     : Kayit yapilarindan olusan diziye (yapi dizisi) isaret eden adres degiskeni.
   p_2as    : 2-boyutlu Kayit yapi dizisine isaret eden adres degiskeni.
   p_ap_2as : 2-b Kayit yapi dizisinin elemanlarina isaret eden 2-b adres dizisine isaret
              eden adres degiskeni.
 */

/* tek bir Kayit yapisi icin dinamik bellek ayrilmasi */
if ( (p_s = (Kayit *)calloc( 1, sizeof (Kayit))) == NULL )
{
        fprintf( stderr, "Bellek alani ayrilamadi !!!\n" );
        exit( 1 );
}
/* Kayit yapisina deger atama   */
strcpy( p_s->dizgi, "XyZabc123K" );
p_s->i = 99;
/* Kayit yapisinin listelenmesi */
puts( "Kayit Yapisi :" );
printf( "(*p_s).dizgi   : %s, (*p_s).i  : %d\n", (*p_s).dizgi, (*p_s).i );
printf( "p_s[0].dizgi   : %s, p_s[0].i  : %d\n", p_s[0].dizgi, p_s[0].i );
/* ayrilan dinamik bellek alaninin serbest birakilmasi */
free( p_s  );
/*
 * ELSayi adet yapidan olusan yapi dizisi icin dinamik bellek
 * alani ayrilmasi:
 * eleman sayisi programin calismasi sirasinda girilebilir.
 */
if( (p_as = (Kayit *)calloc( ELSayi, sizeof (Kayit))) == NULL )
{
     fprintf( stderr, "Bellek alani ayrilamadi !!!\n" );
     exit( 2 );
}
/* yapi dizisine deger atama   */
for (i = 0; i < ELSayi; i++ )
{
     strcpy( p_as[ i ].dizgi, "XyZabc123K" );
     p_as[ i ].i = i+1;
}
/* yapi dizisinin listelenmesi */
puts( "Kayit Yapi Dizisi :" );
for (i = 0; i < ELSayi; i++ )
```

... Örnek 7.2-1 devam

```
        printf( "%s  %d\n", p_as[ i ].dizgi, p_as[ i ].i );
    /* yapi dizisi icin ayrilan bellek alaninin serbest birakilmasi */
    free( p_as );
    /*
     *    2-b yapi dizisi icin dinamik bellek alani ayrilmasi:
     *       Satir ve Sutun sayilari programin calismasi sirasinda
     *       girilebilir.
     */
    if ( (p_2as = Din2as( dNoSatir2b, dNoSutun2b )) == NULL )
    {
        fprintf( stderr, "Din2as: Bellek alani ayrilamadi !!!\n" );
        exit( 3 );
    }

    /* 2-b yapi dizisine deger atama  */
    for ( i = 0; i < dNoSatir2b; i++ )
        for ( j = 0; j < dNoSutun2b; j++ )
        {
            strcpy( p_2as[ i ][ j ].dizgi, "XyZabc123K" );
            p_2as[ i ][ j ].i = i * j + 2;
        }
    /* 2-b yapi dizisinin listelenmesi */
    puts( "2-b Yapi Dizisi :" );
    for ( i = 0; i < dNoSatir2b; i++ )
    {
        for ( j = 0; j < dNoSutun2b; j++ )
            printf( "%s %d ", p_2as[ i ][ j ].dizgi, p_2as[ i ][ j ].i );
        putchar( NL );
    }
     /*
      *    2-b adres dizisi icin bellek alani ayrilmasi.
      */
    if ( (p_ap_2as = (Kayit***)Dinap_2as( dNoSatir2b, dNoSutun2b )) == NULL )
    {
        fprintf( stderr, "Dinap_2as: Bellek alani ayrilamadi !!!\n" );
        exit( 4 );
    }


     /*
      *    adres dizisine yukaridaki deyimlerde olusturulan 2-b yapi
      *    dizisinin elemanlarinin adreslerinin atanmasi.
      */
    for ( i = 0; i < dNoSatir2b; i++ )
        for ( j = 0; j < dNoSutun2b; j++ )
            p_ap_2as[ i ][ j ] = &p_2as[ i ][ j ];

     /*
      *    2-b adres dizisinin elemanlarinin isaret ettigi yapilarin degerlerinin listelenmesi.
      */
```

... Örnek 7.2-1 devam

```
   puts( "2-b adres dizisi ile 2-b Yapi Dizisine erisim :" );
   for ( i = 0; i < dNoSatir2b; i++ )
   {
        for ( j = 0; j < dNoSutun2b; j++ )
             printf( "%s %d ", p_ap_2as[ i ][ j ]->dizgi, p_ap_2as[ i ][ j ]->i );
        putchar( NL );
   }
   /* 2-b yapi dizisi icin ayrilan dinamik alanin serbest birakilmasi  */
   Free2as( p_2as,  dNoSatir2b );
   /* 2-b adres dizisi icin ayrilan dinamik alanin serbest birakilmasi          */
   Freeap_2as( p_ap_2as, dNoSatir2b );
   return 0;
}

/*
 *    Din2as fonksiyonu 2-boyutlu yapi dizisi icin dinamik bellek  alani ayirir ve bu alana
 * isaret eden adres degerini dondurur.
*/
Kayit  **Din2as( int NoSatir2b, int NoSutun2b )
{
   int i;
   Kayit **p_2as;
   /* 2-b dizinin satirlarina isaret eden adres degiskenleri icin bellek alani ayrilmasi. */
   if ( (p_2as = (Kayit **)malloc( NoSatir2b * sizeof (Kayit*) )) == NULL )
        return (Kayit**)NULL;
   /*
    *2-b dizinin satirlarini olusturan 1-b diziler (elemanlari Kayit yapilari olan) icin bellek
    *alani  ayrilmasi.
    */
   for ( i = 0; i < NoSatir2b; i++ )
          if ( (p_2as[ i ] = (Kayit *)malloc( NoSutun2b * sizeof (Kayit) )) == NULL )
               return (Kayit**)NULL;
   return p_2as;
}
/*
 * Dinap_2as fonksiyonu Kayit tipinde yapilara isaret eden adres degiskenlerinden
 * olusan 2-b   adres dizisi icin dinamik bellek alani ayirir ve bu alana isaret eden
 * adres degerini dondurur.
 */
Kayit  ***Dinap_2as( int NoSatir2b, int NoSutun2b )
{
   int i;
   Kayit ***p_ap_2as;
   /*
    * 2-b adres dizisinin satirlarina isaret eden cift adres degiskenleri icin bellek alani ayrilmasi.
    */
   if ( (p_ap_2as = (Kayit ***)malloc( NoSatir2b * sizeof (Kayit**) )) == NULL )
```

... Örnek 7.2-1 devam

```
        return (Kayit***)NULL;
    /*
     *   2-b adres dizisinin satirlarini olusturan 1-b diziler (elemanlari Kayit tipi yapilara
     *   isaret eden adres degiskenleri olan) icin dinamik bellek alani ayrilmasi.
     */
    for ( i = 0; i < NoSatir2b; i++ )
        if ( (p_ap_2as[ i ] = (Kayit **)malloc( NoSutun2b * sizeof (Kayit*) ))
            == NULL )
            return (Kayit***)NULL;
    return p_ap_2as;
}
/*
 * Free2as fonksiyonu, Din2as fonksiyonu tarafindan ayrilan dinamik bellek
 * bloklarini serbest birakir.
 */
void  Free2as( Kayit **p_2as, int NoSatir2b )
{
    int i;
      /*
       *  2-b dizinin satirlarina isaret eden adres degiskenleri icin ayrilan bellek alaninin serbest
       *  birakilmasi.
       */
    free( p_2as );
      /*
       *  2-b dizinin satirlarini olusturan 1-b diziler (elemanlari Kayit yapilari olan) icin ayrilan
    *  bellek  alaninin serbest birakilmasi.
       */
    for ( i = 0; i < NoSatir2b; i++ )
        free( p_2as[ i ] );
}
/*
 *  Dinap_2as fonksiyonu tarafindan ayrilan dinamik bellek alanini serbest birakir.
 */
void  Freeap_2as( Kayit ***p_ap_2as, int NoSatir2b )
{
    int i;
      /*
       *     2-b adres dizisinin satirlarina isaret eden cift adres degiskenleri icin ayrilan dinamik
       *     bellek alaninin serbest birakilmasi.
       */
    free( p_ap_2as );
      /*
       *  2-b adres dizisinin satirlarini olusturan 1-b diziler (elemanlari Kayit tipi yapilara isaret
       *  eden adres degiskenleri olan) icin ayrilan dinamik bellek alaninin serbest birakilmasi.
       */
```

...Örnek 7.2-1 devam

```
   for ( i = 0; i < NoSatir2b; i++ )
        free( p_ap_2as[ i ] );
}
```

Çıktı

```
Kayit Yapisi :
(*p_s).dizgi :  XyZabc123K, (*p_s).i :  99
p_s[0].dizgi :  XyZabc123K, p_s[0].i :  99

Kayit Yapi Dizisi :
XyZabc123K  1
XyZabc123K  2
XyZabc123K  3
XyZabc123K  4
XyZabc123K  5

2-b Yapi Dizisi :
XyZabc123K  2 XyZabc123K  2 XyZabc123K  2
XyZabc123K  2 XyZabc123K  3 XyZabc123K  4
2-b adres dizisi ile 2-b Yapi Dizisine erisim :
XyZabc123K  2 XyZabc123K  2 XyZabc123K  2
XyZabc123K  2 XyZabc123K  3 XyZabc123K  4
```

BÖLÜM 8: **Yapı Değişkenleri**

C dili, mevcut değişken tiplerine ilave olarak (basit değişkenler, diziler gibi), diğer bazı tiplerin tanımlanmasına olanak sağlar. Bunlardan biri de, *yapı değişkeni*dir. Bu bölümde, yapı değişkeninin bildirimi ve programlarda kullanımı anlatılmıştır. ■

8.1 Yapı Değişkeni Bildirimi

Birbiri ile ilgili değişkenlerin ortak bir isim altında toplanması ile *yapı değişkeni* (yada kısaca *yapı*) oluşturulur. Yapı elemanı olan değişkenler farklı tiplerde olabilir. Değişkenlerin bu şekilde organize edilerek bir bütün halinde işlenebilmesi, özellikle çok sayıda değişken içeren programlarda büyük kolaylık sağlar. Bir yapı, tek bir değişkenden de oluşabilir.

Yapı değişkeni çeşitli şekillerde bildirilebilir:

- İlk olarak elemanların bellekte bulunuş şekillerini temsil eden bir *yapı şablonu* bildirilir.

  Yapı şablonu bildiriminin genel şekli:

  ```
  struct etiket_ismi
  {
     yapı elemanı bildirimleri ... ;
  };
  ```

  Daha sonra, şablon bildiriminde yer alan **struct** anahtar kelimesi ve bunu izleyen *yapı etiketi* (yada *yapı ismi*) kullanılarak, aynı şablona sahip yapı değişkenleri bildirilir:

  ```
  struct etiket_ismi  degisken_listesi;
  ```

Örnek olarak aşağıda, proje bilgileri içeren proje şablonu bildirilir. Şablon, projenin ismi için isim karakter dizisini, proje başlangıç ve bitiş tarihleri için ise **long** tamsayı değişkenler basla ve son'u içerir:

```
/* sablon bildirimi */
struct proje
{
   char *isim;
   long basla;
   long son;
};
```

struct anahtar kelimesinden sonra yer alan "proje" ismi, yapı etiketidir ve sadece yapının şablonunu yada bir başka deyişle iskeletini tanımlama işlevini yerine getirir. proje olarak adlandırılan şablonu bildirildikten sonra, "**struct** proje" kelimeleri diğer veri tipleri gibi kullanılabilecek *yapı veri tipi*ni oluşturur. Bu şablona sahip p1, p2 ve p3 yapı değişkenleri aşağıdaki şekilde bildirilir:

```
/* yapi degiskeni bildirimi */
struct proje p1, p2, p3;
```

- Yapı değişkeni bildirimi aşağıdaki şekilde de yapılabilir:

```
struct
{
   yapı elemanı bildirimleri ... ;
} degisken_listesi;
```

Fakat bu bildirimde etiket bulunmadığından dolayı, başka bir yerde aynı şablona sahip değişken bildirmek için şablonun tekrar yazılması gerekir.

Örnek olarak aşağıda etiket kullanılmadan, elemanları sınırlayan blok sonu oklu parantezinden sonra virgülle ayrılmış değişken listesi yazılarak, aynı deyimde aynı şablona sahip birden fazla yapı değişkeni bildirilir:

```
struct
{
   char *isim;
   long basla;
   long son;
} p1, p2, p3;
```

- Diğer bir yol ise, ilk yapı değişkeni bildiriminde etiket kullanılmasıdır. Böylece, aynı şablon daha sonraki bildirimlerde kullanılabilir:

```
struct proje
{
   char *isim;
   long basla;
   long son;
} p1;
...
struct proje p2, p3;
```

Yukarıda anlatılan bildirim şekillerinden herhangi biri kullanılabilir. Programın okunabilirliğini arttıracak ve daha sonra bazı değişiklerin kolayca yapılabilmesini sağlayacak en pratik yol seçilmelidir. Genellikle, şablon bildirimleri bir **include** dosyasına yada programın ilk satırlarına yerleştirilir. Daha sonra bu şablonların etiket isimleri kullanılarak, aynı şablonlara sahip yapı değişkenleri bildirilir. Bu uygulama çok daha pratiktir.

Sonuç olarak yapı değişkeni bildiriminin genel şekli aşağıdaki gibidir:

```
struct [etiket_ismi]
{
   eleman bildirimleri...;
} degisken_listesi;
```

Burada etiket kullanımı, seçime bağlı olduğu için köşeli parantezler ([ ]) arasında verilmiştir. Örnek 8.1-1'de, proje1 ve proje2 şablonları kullanılarak **struct** proje1 tipinde p1 yapı değişkeni ve **struct** proje2 tipinde p2 yapı değişkeni bildirilir. Şablon bildiriminde değişken ismi yer almadıkça, bellek alanı ayrılmaz. Dolayısıyla, proje1 ve proje2 yapı etiketleri bellek alanına sahip değildir, fakat yapı tipindeki p1 ve p2 değişkenleri için bellek alanı ayrılır.

isim, basla ve son değişkenleri, p1 ve p2 yapısının elemanlarıdır. Programda da görüldüğü gibi, bir yapının eleman isimleri diğer yapılarda eleman ismi olarak yada değişken ismi olarak kullanılabilir.

Örnek 8.1-1 yapi1.c programı.

```
#include <stdio.h>
#include <string.h>

struct proje1
{
  char *isim;
  long  basla;
  long  son;
};

#define ELEMAN_SAYI 10

struct proje2
{
  char  isim[ ELEMAN_SAYI + 1 ];
  long  basla;
  long  son;
};

int main(void)
{
  struct proje1 p1;
  struct proje2 p2;
  long son = 1997;

  p1.isim = "xyz";
  p1.basla = 1991L;
  p1.son = 1997L;

  printf("p1.isim: %s, p1.basla: %ld, p1.son: %ld\n", p1.isim, p1.basla, p1.son);
  printf("p1          : %d  byte\n",  sizeof( p1 )          );
```

... Örnek 8.1-1 devam

```
   printf("p1.isim    : %d  byte\n",  sizeof( p1.isim )    );
   printf("p1.basla  : %d  byte\n",  sizeof( p1.basla )  );
   printf("p1.son    : %d  byte\n",  sizeof( p1.son )    );

   strcpy( p2.isim, "xyz" );
   p2.basla = 1991L;
   p2.son = son;

   printf("p2.isim: %s, p2.basla: %ld, p2.son: %ld\n", p2.isim, p2.basla, p2.son);
   printf("p2          : %d  byte\n",  sizeof( p2 )          );
   printf("p2.isim    : %d  byte\n",  sizeof( p2.isim )    );
   printf("p2.basla  : %d  byte\n",  sizeof( p2.basla )  );
   printf("p2.son    : %d  byte\n",  sizeof( p2.son )    );

   return 0;
}
```

Çıktı

```
p1.isim: xyz, p1.basla: 1991, p1.son: 1997
p1          : 10  byte
p1.isim    : 2  byte
p1.basla  : 4  byte
p1.son    : 4  byte
p2.isim: xyz, p2.basla: 1991, p2.son: 1997
p2          : 20  byte
p2.isim    : 11  byte
p2.basla  : 4  byte
p2.son    : 4  byte
```

Bir yapının elemanlarına *yapı elemanı operatörü* (.) yada diğer adıyla *nokta operatörü* kullanılarak erişilebilir. Bu operatörün kullanılması ile oluşturulan *yapı_degiskeni.eleman_ismi* ifadesi, yapı elemanları için bellekte ayrılan alanlara erişimi sağladığı için, bir değişken isminin kullanılabileceği her yerde kullanılabilir. Programda yer alan atama deyimleri ile yapı elemanlarına çeşitli değerler atanır. p1 yapısının isim elemanı **char *** tipinde bir adres değişkenidir. p2 yapısının isim elemanı ise bir karakter dizisidir. **#define** ön-işlemci komutu kullanılarak ELEMAN_SAYI sembolik sabiti tanımlanır. Bildirimde isim karakter dizisinin eleman sayısı, dizgi sonunu belirten boş karakter için yer ayırmak amacıyla ELEMAN_SAYI + 1 olarak verilir. p1.isim adres değişkenine atama deyimi ile proje ismi olarak "xyz" dizgisi atanır. p2.isim karakter dizisine ise proje ismi olarak yine aynı dizgi, dizgi kopyalama fonksiyonu strcpy fonksiyonu kullanılarak atanır. **long** tamsayı tipindeki p1.basla ve p2.basla değişkenlerine 1991 değeri, p1.son değişkenine ise 1997 değeri atanır. p2.son değişkenine ise atama deyiminde **long** tamsayı değişken son kullanılarak yine aynı değer atanır.

Program ekrana p1 ve p2 yapılarının bellek alanı büyüklüklerini ve ayrıca yapı elemanlarının bellek alanı büyüklüklerini ve değerlerini listeler. Program 2-byte bellek adreslerine sahip bir donanımda yukarıdaki çıktıyı verir. Fakat bu değerler 4-byte bellek

adreslerine sahip bir donanımda farklı olacaktır. Bir yapının bellek alanı büyüklüğü donanıma göre değişebilir. Dolayısıyla, C programlarının taşınabilir olması için, yapı elemanlarının bellek adresleri üzerinde yapılan işlemler buna göre düzenlenmelidir.

Örnek 8.1-2'de görüldüğü gibi, bir yapı değişkeni bir başka yapı değişkeni içerebilir. Örnek programda proje başlama ve sona erme tarihleri gün ve ay olarak kaydedilir. Bu amaçla proje şablonu içinde GunAy şablonuna sahip basla ve son elemanları bildirilir ve böylece p.basla.gun, p.basla.ay, p.son.gun ve p.son.ay elemanları oluşur. Program p yapısının elemanlarının değerlerini ekrana listeler.

Örnek 8.1-2 yapi2.c programı.

```
#include <stdio.h>

struct GunAy
{
  short gun;
  short ay;
};

struct proje
{
  char  *isim;
  struct GunAy  basla;
  struct GunAy  son;
};

int main(void)
{
  struct proje p = { "xyz", 15, 12, 31, 12 };

  printf("proje ismi: %s\n", p.isim );
  printf("proje baslama gun/ay: %d/%d, ", p.basla.gun, p.basla.ay );
  printf(" sona erme gun/ay: %d/%d\n", p.son.gun, p.son.ay);

  return 0;
}
```

Çıktı

```
proje ismi: xyz
proje baslama gun/ay: 15/12,  sona erme gun/ay: 31/12
```

Örnek programlarda yalnızca yapı elemanı operatörü (.) kullanılarak yapı elemanlarına erişim işlemi yer aldı. Yapılar üzerinde, yapı değişkeninin bir bütün olarak ele alındığı diğer bazı işlemler de gerçekleştirilebilir. Bir yapı, aynı tipte (aynı şablona sahip) bir başka yapıya atanabilir, bir fonksiyona argüman olarak aktarılabilir. Ayrıca, yapı değeri döndüren fonksiyon tanımlanabilir. Fakat, iki yapı birbiri ile karşılaştırılamaz. Bu işlem, yapı elemanları ile gerçekleştirilebilir.

Global yada **static** (blok içi yada blok dışı) yapı değişkenlerine ilk değer atama sadece elemanlarına karşılık gelen sabit değerler yada sabit ifadelerin, oklu parantezlerle sınırlı listesi ile yapılabilir.

Otomatik yapı değişkenlerine de aynı şekilde ilk değer atama yapılabilir, fakat eğer bu işlem ilk değer listesi yerine tek bir ifade ile yapılacak ise, aynı tipteki bir başka yapı yada yapı değeri döndüren bir fonksiyon çağrısı eşitlenerek de gerçekleştirilebilir.

İlk değer listesinde yer alan sabit değer yada sabit ifade sayısı, eleman sayısından az ise, kalan elemanların ilk değerleri 0 olur. Daha fazla ilk değerin bulunması derleme sırasında hata oluşturur.

İlk değer atama yapılmadığında, global yada **static** yapı değişkenlerinin elemanlarının ilk değerleri 0 olur. Otomatik yapı değişkenlerinin elemanlarının ilk değerleri ise belirsizdir.

Örnek 8.1-3'de, global yapı değişkenleri g1 ve g2 bildirilir. g1'e sabit değer listesi ile ilk değer atama yapılır. Ayrıca, fonk bloğu içinde de s1, s2, s3 ve s4 otomatik yapı değişkenleri ve **static** yapı değişkeni s5 bildirilir. s2 yapısına sabit değer listesi ile ilk değer atama yapılır. s1'e aynı tipteki g1, s3'e ise koşul operatörünün test ifadesine bağlı olarak s1 yada s2 atanarak ilk değer atama işlemi gerçekleştirilir. Program ekrana yapı elemanlarının değerlerini listeler. Çıktıda da görüldüğü gibi, ilk değer atama yapılmayan global yapı g2 ve blok içi **static** yapı s5'in elemanlarının ilk değerleri 0 olur. Otomatik yapı s4'ün elemanlarının değerleri ise belirsizdir.

Örnek 8.1-3 ilkyapi1.c programı.

```
#include <stdio.h>

#define SECIM 1

struct yapi
{
   int  i;
   long x;
}  g1   = { 66, 33L }, g2;

int main(void)
{
   struct yapi s1 = g1, s2 = { 77, 55L }, s3 = ( SECIM ? s1 : s2 ), s4;
   static struct yapi s5;

   printf("g1.i : %d, g1.x : %ld\n", g1.i, g1.x );
   printf("g2.i : %d, g2.x : %ld\n", g2.i, g2.x );
   printf("s1.i : %d, s1.x : %ld\n", s1.i, s1.x );
   printf("s2.i : %d, s2.x : %ld\n", s2.i, s2.x );
```

...Örnek 8.1-3 devam

```
   printf("s3.i : %d, s3.x : %ld\n", s3.i, s3.x );
   printf("s4.i : %d, s4.x : %ld\n", s4.i, s4.x );
   printf("s5.i : %d, s5.x : %ld\n", s5.i, s5.x );

   return 0;
}
```

Çıktı

```
g1.i  : 66,   g1.x : 33
g2.i  : 0,    g2.x : 0
s1.i  : 66,   s1.x : 33
s2.i  : 77,   s2.x : 55
s3.i  : 66,   s3.x : 33
s4.i  : 689,  s4.x : 277479424
s5.i  : 0,    s5.x : 0
```

Örnek 8.1-4'de, 4 tamsayı elemana sahip global yapı değişkeni g1, blok dışı (external) **static** yapı değişkenleri e1 ve e2, blok içi (internal) **static** yapı değişkeni s3 ve otomatik yapı değişkenleri s1 ve s2 bildirilir.

Örnek 8.1-4 ilkyapi2.c programı.

```
#include <stdio.h>

struct yapi
{
   int     i, j;
   long    x, y;
}  g1 = { 1, 2 };

static struct yapie1 = { 1, 2 };
static struct yapie2;

int main(void)
{
   struct yapi s1 = { 1, 2 }, s2;
   static struct yapi s3 = { 1, 2 };

   printf( "g1.i : %d, g1.j : %d, g1.x : %ld, g1.y : %ld\n", g1.i, g1.j, g1.x, g1.y );

   printf( "e1.i : %d, e1.j : %d, e1.x : %ld, e1.y : %ld\n", e1.i, e1.j, e1.x, e1.y );

   printf( "e2.i : %d, e2.j : %d, e2.x : %ld, e2.y : %ld\n", e2.i, e2.j, e2.x, e2.y );

   printf( "s1.i : %d, s1.j : %d, s1.x : %ld, s1.y : %ld\n", s1.i, s1.j, s1.x, s1.y );

   printf( "s2.i : %d, s2.j : %d, s2.x : %ld, s2.y : %ld\n", s2.i, s2.j, s2.x, s2.y );

   printf( "s3.i : %d, s3.j : %d, s3.x : %ld, s3.y : %ld\n", s3.i, s3.j, s3.x, s3.y );

   return 0;
}
```

...Örnek 8.1-4 devam

Çıktı

```
g1.i : 1,    g1.j : 2,    g1.x : 0,          g1.y : 0
e1.i : 1,    e1.j : 2,    e1.x : 0,          e1.y : 0
e2.i : 0,    e2.j : 0,    e2.x : 0,          e2.y : 0
s1.i : 1,    s1.j : 2,    s1.x : 0,            s1.y : 0
s2.i : 689,  s2.j : 0,    s2.x : 65409162,   s2.y : 49363
s3.i : 1,    s3.j : 2,    s3.x : 0,            s3.y : 0
```

Programda, e2 ve s2 yapılarına ilk değer atama yapılmamıştır. Bu durumda, çıktıda da görüldüğü gibi, e2 yapısının elemanlarının ilk değerleri 0 olur. s2 yapısının elemanlarının ilk değerleri ise belirsizdir. g1, e1, s1 ve s3 yapı değişkenlerine ilk değer atama, eleman sayısından daha az sayıda sabit değer içeren liste ile yapılır. Listede bulunan değerler, yapı elemanlarına sırayla atanır. Kalan elemanların ilk değerleri, çıktıda da görüldüğü gibi 0 olur. ■

8.2 Yapı Dizisi

Aynı şablona sahip birden fazla yapı değişkenine ihtiyaç duyulduğunda *yapı dizisi* kullanılır:

struct *etiket_ismi dizi_ismi*[*eleman_sayısı*];

Örnek 8.2-1'de yer alan **struct** bildirimi ile, bellekte **struct** proje veri tipinde 3 yapı değişkeni için yer ayrılır:

```
struct proje p[ EL_SAYI ];
```

Burada p, proje yapı şablonuna sahip 3 adet yapı değişkeninden oluşan yapı dizisidir. p'nin tipi struct proje[3] olur.

Örnek 8.2-1 yapidiz1.c programı.

```
#include <stdio.h>

#define EL_SAYI      3
#define DIZGI_BOY    10

struct proje
{
  char  *isim;
  long  basla;
  long  son;
};
```

...Örnek 8.2-1 devam

```
int main(void)
{
  struct proje p[ EL_SAYI ];
  char dizgi[ EL_SAYI ][ DIZGI_BOY+1 ];
  int  i;
  for ( i  = 0; i < EL_SAYI; ++i )
  {
       p[i].isim = dizgi[i];
       printf( "proje ismi, baslama, bitis ? " );
       scanf( "%10s%ld%ld",
                                  p[i].isim,   &p[i].basla,  &p[i].son );
  }
  for ( i  = 0; i < EL_SAYI; ++i )
       printf( "%10s %4ld %4ld\n",
                                  p[i].isim,   p[i].basla, p[i].son );
  return 0;
}
```

Bilgi Girişi

```
proje ismi, baslama, bitis ? proje1 1991 1992
proje ismi, baslama, bitis ? proje2 1992 1994
proje ismi, baslama, bitis ? proje3 1994 1995
```

Çıktı

```
proje1 1991 1992
proje2 1992 1994
proje3 1994 1995
```

Dizide bulunan yapı elemanlarına erişim için,

dizi_ismi[*indeks*]**.***eleman_ismi*

ifadesi kullanılır. Örnek 8.2-1'de yapı elemanlarına bu şekilde oluşturulan p[i].isim, p[i].basla ve p[i].son ifadeleri ile erişilir. scanf fonksiyonu, klavyeden girilen bilgileri argüman olarak aktarılan bellek adreslerine atar. Bu fonksiyonun argüman listesinde, yapı elemanlarının bellek adreslerini veren p[i].isim, &p[i].basla ve &p[i].son ifadeleri bulunur. p[i].isim bir adres değişkenidir ve bir karekter dizisinin bellekte bulunduğu alanın başlangıcına işaret eder. Bu nedenle p[i].isim ifadesine & operatörü uygulanmaz. p[i].isim aracılığı ile, klavyeden girilen proje ismini dizgi sabiti olarak belleğe yerleştirmek için, bu adres değişkeninin geçerli bir bellek alanına işaret etmesi gerekir. Programda, yapı dizisi bildirimini izleyen satırda dizgi karakter dizisi bildirilir. Bu bildirim ile en fazla 10 karekter uzunluğunda 3 adet dizgi sabiti için bellek alanı ayrılır (11. eleman, dizgi sonunu belirleyen \0 karakteri içindir).

Daha sonra **for** döngüsü içinde,

```
p[i].isim = dizgi[i];
```

atama deyimi ile ayrılan bu alanların başlangıç adresleri, adres değişkeni p[i].isim'e aktarılır. Böylece scanf çağrısı öncesinde p[i].isim adres değişkeninin dizgilerin saklanabileceği geçerli bir bellek alanına işaret etmesi sağlanır. Klavyeden girilen proje isimleri bu bellek alanlarına yerleştirilir. Bu işlem yapılmadığı taktirde, adres değişkeni p[i].isim'in değeri geçerli herhangi bir bellek adresi olmayacağı için programın çalışması sırasında, "*boş adres değişkenine değer atama*" hatası (*null pointer assignment*) oluşacaktır.

Örnek 8.2-1'de kullanılan proje şablonunda, isim elemanı **char** tipi verilere işaret eden bir adres değişkeni olarak bildirildi. isim elemanı, **char** tipi bir dizi olarak bildirildiğinde, dizgi dizisi bildirimi ile bellek alanı ayırmaya gerek kalmayacaktır. Çünkü her isim elemanı için ayrılan alan, p[i] yapısında bulunan bir karakter dizisi olacaktır. Bu durumda Örnek 8.2-1 aşağıdaki şekilde yazılabilir:

Örnek 8.2-2 yapidiz2.c programı.

```
#include <stdio.h>

#define EL_SAYI       3
#define DIZGI_BOY    10

struct proje
{
  char   isim[ DIZGI_BOY+1 ];
  long   basla;
  long   son;
};

int main(void)
{
  struct proje p[ EL_SAYI ];
  int  i;

  for ( i  = 0; i < EL_SAYI; ++i )
  {
       printf( "proje ismi, baslama, bitis ? " );
       scanf( "%10s%ld%ld",
                                  p[i].isim,   &p[i].basla,  &p[i].son );
  }

  for ( i  = 0; i < EL_SAYI; ++i )
       printf( "%10s %4ld %4ld\n",
                                  p[i].isim,   p[i].basla, p[i].son );
  return 0;
}
```

...Örnek 8.2-2 devam

Bilgi Girişi

```
proje ismi, baslama, bitis ? proje1 1991 1992
proje ismi, baslama, bitis ? proje2 1992 1994
proje ismi, baslama, bitis ? proje3 1994 1995
```

Çıktı

```
proje1 1991 1992
proje2 1992 1994
proje3 1994 1995
```

Bu örnekte de yine scanf fonksiyonunun ikinci argümanı olan p[i].isim ifadesine & operatörü uygulanmamıştır. Çünkü C dilinde indeksi verilmeyen dizi ismi, dizinin başlangıç elemanının bellek adresini veren adres sabitidir. Okunabilirliği arttırmak amacıyla, örnek programlarda hata kontrol deyimlerine yer verilmemiştir.

Örnek 8.2-3 yapidiz3.c programı.

```
#include <stdio.h>
#define EL_SAYI(s)    sizeof(s)/sizeof(s[0])
struct proje
{
   char   *isim;
   long   basla, son;
} p[ ] = {
                  "proje1", 1991, 1992,
                  "proje2", 1992, 1994,
                  "proje3", 1994, 1995,
                };
int main(void)
{
   int  i;
   for ( i  = 0; i < EL_SAYI(p); ++i )
        printf( "%10s %4ld %4ld\n",  p[i].isim, p[i].basla, p[i].son );
   return 0;
}
```

Çıktı

```
proje1 1991 1992
proje2 1992 1994
proje3 1994 1995
```

Örnek 8.2-3'de proje şablonu bildirilir ve bu şablona sahip yapılardan oluşan p yapı dizisi tanımlanır. Yine aynı deyimde, p yapı dizisine yapı elemanlarına karşılık gelen sabit değer listesi ile ilk değer atama yapılır. p yapı dizisi global olarak (blok dışında) tanımlandığı için, ilk değer atama işlemi (listede yer alan değerlerin ayrılan bellek alanlarına yerleştirilmesi) program çalışmaya başlamadan önce ve sadece bir kez yapılır.

Örnek programda, yapı dizisi tanımında köşeli parantezlerin içi boş bırakılmıştır. Derleyici, ilk değer listesinde bulunan değerleri sayarak dizi boyutlarını hesaplar ve böylece p'nin tipi tamamlanmış olur. Eğer ilk değer listesi bulunmuyor ise, dizi boyutları mutlaka belirtilmelidir. Programda **for** döngüsü ile dizi boyunca ilerleyerek elemanlar listelenir.

Döngünün test ifadesinde kullanılan EL_SAYI(s) makrosu, derleme öncesi C ön-işlemcisi tarafından **#define** satırında tanımlandığı gibi,

```
sizeof(p)/sizeof(p[0])
```

ifadesi ile değiştirilir. Bu ifade, derleme sırasında dizideki yapı değişkeni sayısını yani dizinin eleman sayısını verir. **sizeof**(p[0]) yerine **sizeof**(**struct** proje) kullanılabilir. Çünkü her iki **sizeof** ifadesi de, dizide yer alan tek bir yapının bellek alanı büyüklüğünü verir. Döngüde dizinin eleman sayısı sayısal olarak verilmediği için, dizide yer alan yapıların tipi değiştirildiğinde yada ilk değer listesine ilave yapıldığında program içinde eleman sayısı ile ilgili herhangi bir değişiklik gerekmez. Döngüde dizi sonunun belirlenmesi, ilk değer listesi sonuna yerleştirilecek olan boş adres değişkeni değeri (NULL) kontrol edilerek de yapılabilir.

Örneklerde görüldüğü gibi, yapı dizileri de diğer diziler gibi bildirilir ve ilk değer atama yapılır. Yapı dizisinin her bir elemanı, bir başka alt-veri grubu oluşturur. Herhangi bir yapı değişkeni için geçerli olan ilk değer atama kuralları, yapı dizisinin elemanı olan ve alt-veri grubu oluşturan yapılar için de geçerlidir. Dolayısıyla, alt-veri grubu elemanlarına ilk değer atama, yine oklu parantezlerle sınırlanmış ve virgüllerle ayrılmış sabit değer yada sabit ifadelerle yapılır. Çok boyutlu dizilerin aksine, yapı dizisinde yer alan alt-veri gruplarının elemanları farklı veri tiplerinde olabileceği için, iç parantezler sadece ilk değer listesinin eksiksiz olduğu durumlarda kullanılmayabilir. Örnek 8.2-3'de ilk değer listesi eksiksiz olduğu için, dizideki alt-veri gruplarını oluşturan yapılara karşılık gelen ilk değerler parantezlerle sınırlandırılmamıştır. Fakat listenin tam olmadığı durumlarda ilk değerlerin dizinin her sırasına yada bir başka deyişle her yapısına karşılık gelecek şekilde parantezlerle sınırlandırılması zorunludur. Aksi taktirde, veri tipleri birbirine uymayacağı için hata oluşacaktır.

Örneğin, sadece isim elemanına ilk değer atama yapılacak ise aşağıdaki ifade kullanılmalıdır:

```
struct proje
{
  char *isim;
  long basla, son;
} p [ ] = {  { "proje1" },  { "proje2" },  { "proje3" },  };
```

Bu yapı dizisi tanımında yer alan ilk değer listesi, alt-veri grubu oluşturan yapı değişkenlerinin basla ve son elemanları için herhangi bir ilk değer içermez. Dolayısıyla, basla ve son elemanlarının ilk değeri 0 olur.

Bütün global ve **static** dizilerde olduğu gibi, ilk değer atama yapılmayan global ve **static** yapı dizilerinin elemanlarının ilk değerleri 0 olur; otomatik yapı dizilerinin elemanlarının ilk değerleri ise belirsizdir.

Örnek 8.1-2'de proje yapısı içinde yer alan her GunAy yapısı bir alt-veri grubu oluşturur. İlk değer atama aşağıdaki şekilde yapıldığında, p.son.ay elemanının ilk değeri 0 olur:

```
static struct proje p = { "xyz", 15, 12, 31 };
```

Bu deyimdeki ilk değer listesinde, alt-veri gruplarına karşılık gelen ilk değerler parantezlerle sınırlanmadığı için, gerekli sayıda ilk değer kullanılır ve kalan değerlerle yapının izleyen elemanı olan alt-veri grubuna ilk değer atama yapılır. Buna göre, xyz dizgisi p.isim elemanına, 15 ve 12 değerleri ilk alt-veri grubunu oluşturan p.basla yapısının p.basla.gun ve p.basla.ay elemanlarına ve 31 değeri de p yapısının izleyen elemanı olan ve bir başka alt-veri grubu oluşturan p.son'un p.son.gun elemanına ilk değer olarak atanır. Listede daha başka sabit değer bulunmadığından dolayı, p.son.ay elemanının ilk değeri 0 olur.

Aşağıdaki bildirim deyiminde, 15 sayısını sınırlayan iç parantezler, ilk alt-veri grubuna atanacak değerleri ifade eder. Sonuç olarak p.basla.gun'ün ilk değeri 15, p.basla.ay'ın ilk değeri 0, p.son.gun ve p.son.ay'ın ilk değerleri ise sırasıyla 12 ve 31 olur.

```
static struct proje p = { "xyz",  { 15 },  12, 31 };
```

Örnek 8.2-4'de **struct** Yapi1 tipinde t yapısı bildirilir ve bu yapının elemanı olan iki karakter dizisi (s1 ve s2) için bellekte her biri 5 karakter uzunluğunda sabit büyüklükte alanlar ayrılır.

Örnek 8.2-4 yapidiz4.c programı.

```
#include <stdio.h>

struct Yapi1
{
   char s1[5], s2[5];
};

struct Yapi2
{
   char *s1, *s2;
};

int main(void)
{
   struct Yapi1 t  = {  { 'a', 'b', 'c' }, "xyz" };
   struct Yapi2 p = {  "abc", "xyz" };
   int i;

   printf("t.s1 : %d, t.s2 : %d, p.s1 : %d, p.s2 : %d byte\n",
           sizeof( t.s1 ), sizeof( t.s2 ), sizeof( p.s1 ), sizeof( p.s2 ) );

   printf("t.s1 : %s, t.s2 : %s, p.s1 : %s, p.s2 : %s\n",  t.s1, t.s2, p.s1, p.s2 );

   for ( i = 0; i < sizeof(t.s1); i++ )
         printf("t.s1[%d] : %c, ", i, t.s1[i] );
   puts( "" );

   for ( i = 0; i < sizeof(t.s2); i++ )
         printf("t.s2[%d] : %c, ", i, t.s2[i] );
   puts( "" );

   i = 0;
   while ( p.s1[i] )
   {
           printf("p.s1[%d] : %c, p.s1+%d : %d\n", i, p.s1[i], i, p.s1+i );
           i++;
   }

   for ( i = 0; p.s2[i] ; i++ )
         printf("p.s2[%d] : %c, p.s2+%d : %d\n", i, *(p.s2+i), i, p.s2+i );

   return 0;
}
```

Çıktı

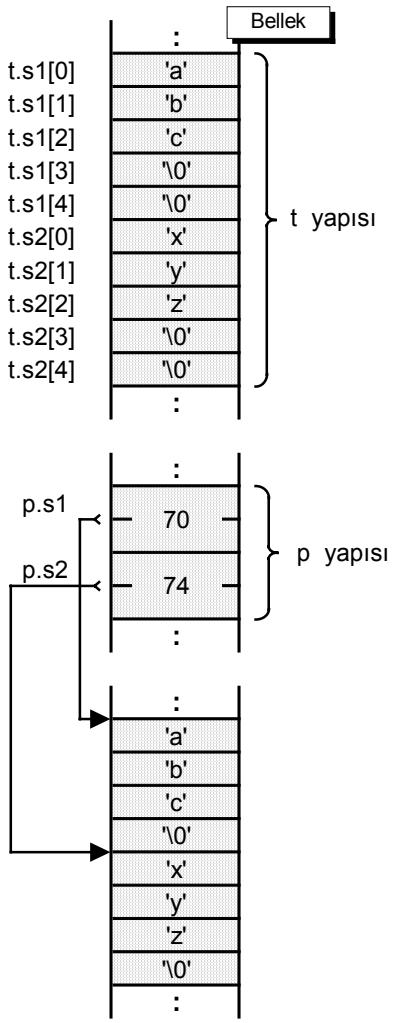
```
t.s1 : 5,    t.s2 : 5,    p.s1 : 2,   p.s2 : 2 byte
t.s1 : abc, t.s2 : xyz, p.s1 : abc,   p.s2 : xyz

t.s1[0] : a, t.s1[1] : b, t.s1[2] : c, t.s1[3] :  , t.s1[4] :  ,
t.s2[0] : x, t.s2[1] : y, t.s2[2] : z, t.s2[3] :  , t.s2[4] :  ,

p.s1[0] : a,   p.s1+0 : 70
p.s1[1] : b,   p.s1+1 : 71
```

...Örnek 8.2-4 devam

```
p.s1[2] : c,   p.s1+2 : 72
p.s2[0] : x,   p.s2+0 : 74
p.s2[1] : y,   p.s2+1 : 75
p.s2[2] : z,   p.s2+2 : 76
```

Bildirimde ilk değer atama yapılarak bu alanlara abc ve xyz dizgileri yerleştirilir. s1 ve s2 karakter dizilerinin ilk değer atama yapılmayan elemanlarının değerleri boş karakter ('\0') olur. Fakat bellekte ayrılan alanların sadece bir kısmı kullanılmıştır.

Şekil 8.2-1 t ve p yapı değişkenlerinin sembolik bellek görünümü.



Bellek tasarrufu söz konusu olduğunda, yapı elemanları olarak karakter dizileri yerine, dizgilere işaret eden adres değişkenleri kullanılır. Programda bildirilen p yapısının elemanları, bellekte başka bir yerde bulunan dizgilere işaret eden adres değişkenleridir. Böylece dizgiler için sadece gerekli miktarda bellek alanı ayrılır (dizgi sonunu belirten NUL karakteri (*null character*) ve dizgilerde bulunan karakter sayısı kadar bellek alanı). Özellikle eleman sayısının çok fazla olduğu uygulamalarda bu yol önemli ölçüde bellek tasarrufu sağlar.

Yapı elemanı operatörü (.) ve indeks operatörü ([ ]) aynı operatör önceliğine sahiptir. Fakat grup ilişkileri soldan sağa doğru olduğu için, t ve p yapı değişkenlerindeki karakter dizilerinin elemanlarına erişim için kullanılan ifadelerde parantez kullanımı gereksizdir. Örneğin, (t.s1)[i] ve t.s1[i] ifadeleri aynıdır. Programda değişik kullanımları örneklemek amacıyla, p.s1 adres değişkeninin işaret ettiği dizgideki karakterler **while** döngüsünün test ifadesinde dizgi sonunu belirleyen boş karekter kontrolü yapılarak listelenir. Ayrıca, p.s2 aracılığı ile erişim için adres aritmetiği kullanılmıştır. Şekil 8-2.1'de t ve p yapı değişkenlerinin ilk değer atamaları yapıldıktan sonraki sembolik bellek görünümü verilmiştir. ■

8.3 Yapılar ve Adres Değişkenleri

Yapılar da bellekte saklandıkları için, herhangi bir yapının bellek adresi alınabilir. Dolayısıyla, bir yapıya işaret eden adres değişkeni (*yapı adres değişkeni*) bildirilebilir.

Örnek 8.3-1'de, **struct** proje tipinde p yapı değişkeni ve **struct** proje * tipinde ap adres değişkeni bildirilir.

Örnek 8.3-1 adryapi1.c programı.

```
#include <stdio.h>
#include <string.h>

struct proje
{
  char   *isim;
  long   basla;
  long   son;
};

int main(void)
{
  struct proje p       = { "yeni proje", 1991L,  1994L  };
  struct proje *ap  = &p;
  char *dizgi          =    "YENI PROJE";
```

...Örnek 8.3-1 devam

```
   printf("sizeof( struct proje * ) : %d, sizeof( ap ) : %d,  sizeof( p ): %d byte\n\n",
                                                                  sizeof( struct proje * ),
                                                                  sizeof(ap),
                                                                  sizeof(p) );
   printf("ap->isim: %s, ap->basla: %ld, ap->son: %ld\n",
                                                                  ap->isim,
                                                                  ap->basla,
                                                                  ap->son  );
   printf("(*ap).isim : %s, (*ap).basla : %ld, (*ap).son : %ld\n",
                                                                  (*ap).isim,
                                                                  (*ap).basla,
                                                                  (*ap).son );
   printf("(&p)->isim: %s, (&p)->basla: %ld, (&p)->son: %ld\n\n",
                                                                  (&p)->isim,
                                                                  (&p)->basla,
                                                                  (&p)->son );
   strcpy( ap->isim, dizgi );
   ap->basla = 1990L;
   puts("YENI PROJE dizgisi ve 1990 degeri atandiktan sonra ...\n");
   printf("p.isim : %s, p.basla : %ld, p.son : %ld\n",        p.isim, p.basla, p.son );
   printf("proje isminin 3. karakteri : %c\n\n",                *(ap->isim + 2) );
   puts("bellek adresleri :");
   printf("&p : %d, &(p.isim) : %d, &(p.basla) : %d, &(p.son) : %d\n",
                                                                  &p,
                                                                  &(p.isim),
                                                                  &(p.basla),
                                                                  &(p.son) );
   printf("ap : %d, &(ap->isim) : %d, &(ap->basla) : %d, &(ap->son) : %d\n",
                                                                  ap,
                                                                  &(ap->isim),
                                                                  &(ap->basla),
                                                                  &(ap->son) );
   return 0;
}
```

Çıktı

```
sizeof(struct proje *) : 2, sizeof(ap) : 2, sizeof(p): 10 byte
ap->isim     : yeni proje, ap->basla      : 1991,    ap->son          : 1994
(*ap).isim   : yeni proje, (*ap).basla    : 1991,    (*ap).son        : 1994
(&p)->isim   : yeni proje, (&p)->basla    : 1991,    (&p)->son        : 1994
YENI PROJE dizgisi ve 1990 degeri atandiktan sonra ...
p.isim : YENI PROJE, p.basla : 1990, p.son : 1994
proje isminin 3. karakteri : N
```

...Örnek 8.3-1 Çıktı devam

```
bellek adresleri :
&p  : 3940,   &(p.isim)      : 3940,   &(p.basla)     : 3942,  &(p.son)       : 3946
ap  : 3940,   &(ap->isim)    : 3940,   &(ap->basla)   : 3942,  &(ap->son)     : 3946
```

ap adres değişkenine bildirim deyiminde ilk değer olarak, p'nin &p ifadesi ile alınan bellek adresi atanır. Böylece, ap adres değişkeninin p yapısına işaret etmesi sağlanır. Çıktıda da görüldüğü gibi ap, 2 byte büyüklüğünde bir adres değişkenidir ve taşıdığı değer p yapısının bellek adresidir (3940).

Adres değişkeni aracılığı ile yapı elemanlarına indirek erişim için, *ok operatörü* (->) ile oluşturulan ifadeler kullanılır. Örnek programda p yapısının isim, basla ve son elemanlarına erişim için sırasıyla,

ap->isim, ap->basla ve ap->son

ifadeleri kullanılır. Adres değişkeni aracılığı ile erişim, nokta operatörü (.) ile oluşturulan aşağıdaki ifadelerle de gerçekleştirilebilir:

(*ap).isim, (*ap).basla ve (*ap).son

Fakat ok operatörü ile oluşturulan ifadeler, sadece tek bir operatör içerdiği için daha pratiktir. Programda görüldüğü gibi ap yerine yapı değişkeninin bellek adresini veren &p kullanılarak oluşturulan,

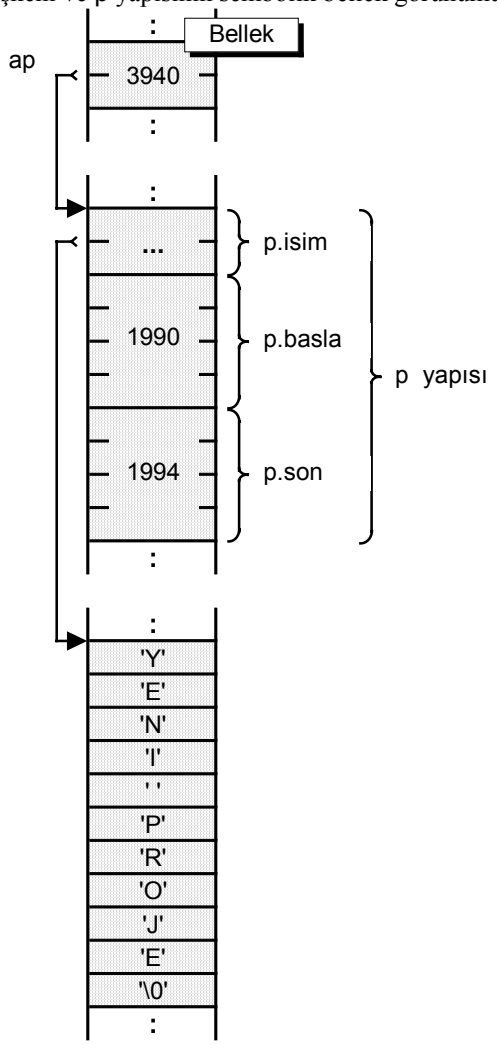
(&p)->isim, (&p)->basla ve (&p)->son

ifadeleri ile de yapı elemanlarına erişmek mümkündür.

Bildirim sırasında, p yapısına parantezlerle sınırlı sabit değer listesi ile ilk değer atama yapılır. Yapının isim elemanı, bellekte başka bir yerde bulunan "yeni proje" dizgi sabitinin bellek adresini alır.
Dolayısıyla ap->isim, bu karakter dizisinin başlangıcına işaret eder. Programda daha sonra, strcpy fonksiyonu kullanılarak ap->isim tarafından işaret edilen bellek alanına "YENI PROJE" dizgisi kopyalanır. Çıktıda da görüldüğü gibi ap->isim + 2 ifadesi, ikinci offset'te bulunan elemanın (dizgi sabitinin 3. karakteri) bellek adresini; *(ap->isim + 2) ise bu adreste bulunan 1 byte değeri (N karakteri) verir. Programda son olarak p yapısının ve yapı elemanlarının bellek adresleri listelenir.

Şekil 8.3-1'de, ap adres değişkeni ve p yapısının sembolik bellek görünümü verilmiştir:

Şekil 8.3-1 ap adres değişkeni ve p yapısının sembolik bellek görünümü.



Sonuç olarak, herhangi bir yapı elemanına erişim için aşağıdaki kombinasyonlar kullanılır:

nokta operatörü kullanılarak:

yapi_degiskeni.*eleman_ismi* (1)
(**adres_degiskeni*).*eleman_ismi* (2)

ok operatörü kullanılarak:

(&*yapi_degiskeni*)->*eleman_ismi* (3)
adres_degiskeni->*eleman_ismi* (4)

Yapı operatörleri (. ve ->), adres değişkeni operatörlerinden (* ve &) daha yüksek önceliğe sahiptir. Dolayısıyla, (2) ve (3) no.'lu ifadelerde hata oluşmaması için parantezlerin kullanımı zorunludur. Örneğin (2) no.'lu ifadede parantezler kullanılmadığı taktirde, **adres_degiskeni.eleman_ismi* ifadesi **(adres_degiskeni.eleman_ismi)* ile aynı olacaktır ve parantezler içindeki ifade bir adres değişkeni olmadığı için hata oluşacaktır. Yapı elemanının bellek adresini veren,

&(adres_degiskeni->eleman_ismi) yada *&(yapi_degiskeni.eleman_ismi)*

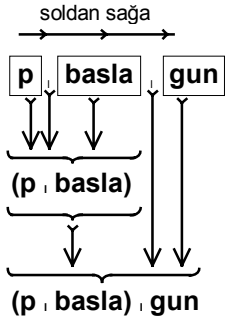
ifadelerinde ise operatör önceliğinden faydalanılarak parantezler kaldırılabilir. Aynı operatör önceliğine sahip **.** ve **->** operatörlerinin grup ilişkileri (*associativity*) soldan sağa doğrudur. Dolayısıyla Örnek 8.1-2'deki (yapi2.c programı) p yapısı ve bu yapıya işaret eden **struct** proje * tipindeki ap adres değişkeni ile oluşturulan ve her iki operatörün de kullanıldığı aşağıdaki 4 ifade birbirine eşittir:

```
p.basla.gun
ap->basla.gun
(p.basla).gun
(ap->basla).gun
```

Þekil 8.3-2'de görüldüðü gibi ifade soldan saða doðru iþlenirken ilk rastlanan nokta operatörü ile p.basla oluþturulur. Bundan sonra rastlanan ikinci nokta operatörü p.basla ile gun elemanýný birleþtirir.

Yapý operatörleri . ve ->, fonksiyon çaðýrma ve indeks operatörleri ile birlikte (sýrasýyla ( ) ve [ ]) öncelik listesinin en üst sýrasýnda bulunurlar. Bu nedenle operatör etkileri (*binding*) diðer operatörlerden fazladýr.

Şekil 8.3-2 p.basla.gun ifadesinin soldan sağa işlenmesi.



Örnek 8.3-2'de ok operatörü, ++ ve -- operatörleri ile birlikte kullanılır. Programda 3 yapıdan oluşan t yapı dizisi ve bu yapı dizisinin elemanlarına işaret eden p adres değişkeni bildirilir. Adres değişkenine bildirim sırasında yapı dizisinin başlangıç elemanı olan yapının bellek adresi atanır (dizi ismi t, başlangıç elemanının bellek adresini veren

sabittir). Bu nedenle, ilk printf deyimi ekrana t[0] yapısının elemanlarının değerlerini listeler. Bu işlem aşağıdaki deyim ile de gerçekleştirilebilir:

```
printf( "t->i : %d, t->s : %s\n", t->i, t->s );
```

Aşağıdaki her iki **for** döngüsü de yapı dizisinde bulunan yapıların elemanlarının değerlerini listeler:

```
for ( j = 0; j < sizeof(t)/sizeof(t[0]); j++ )
        printf( "%d, %s\n", (t+j)->i, (t+j)->s );

for ( j = 0; j < sizeof(t)/sizeof(t[0]); j++ )
        printf( "%d, %s\n", t[j].i, t[j].s );
```

Örnek 8.3-2 adryapi2.c programı.

```
#include <stdio.h>

struct yapi
{
   int i;
   char *s;
};

struct yapi   t[ ]  = {  30,  "ABC",  40,  "EFG",  50,  "MNO" };
struct yapi   *p   = t;

int main(void)
{
   char c;
   int j;

   printf( "p->i : %d, p->s : %s\n", p->i, p->s );

   j = ++p->i;
   printf( "islem 1 : \t p->i : %d, p->s : %s, j : %d\n", p->i, p->s, j );

   j = p++->i;
   printf( "islem 2 : \t p->i : %d, p->s : %s, j : %d\n", p->i, p->s, j  );

   c = *p->s++;
   printf( "islem 3 : \t p->i : %d, p->s : %s, c : %c\n", p->i, p->s, c );

   c = *p++->s;
   printf( "islem 4 : \t p->i : %d, p->s : %s, c : %c\n", p->i, p->s, c );

   c = *p--->s;
   printf( "islem 5 : \t p->i : %d, p->s : %s, c : %c\n", p->i, p->s, c );

   j = (++p)->i;
   printf( "islem 6 : \t p->i : %d, p->s : %s, j : %d\n", p->i, p->s, j );

   j = p->i++;
   printf( "islem 7 : \t p->i : %d, p->s : %s, j : %d\n", p->i, p->s, j );
```

...Örnek 8.3-2 devam

```
   c = (*p->s)++;
   printf( "islem 8 : \t p->i : %d, p->s : %s, c : %c\n", p->i, p->s, c );
   return 0;
}
```

Çıktı

```
p->i : 30, p->s : ABC
islem 1 :       p->i : 31, p->s : ABC, j  : 31
islem 2 :       p->i : 40, p->s : EFG, j  : 31
islem 3 :       p->i : 40, p->s : FG, c      : E
islem 4 :       p->i : 50, p->s : MNO, c   : F
islem 5 :       p->i : 40, p->s : FG, c      : M
islem 6 :       p->i : 50, p->s : MNO, j    : 50
islem 7 :       p->i : 51, p->s : MNO, j    : 50
islem 8 :       p->i : 51, p->s : NNO, c   : M
```

Programda yer alan diğer 8 işlem ve ilgili deyimler şunlardır:

İşlem 1: **j = ++p->i;**

Bu deyim ile yapı dizisinin, p adres değişkenin işaret ettiği yapısının i elemanının değeri arttırılır. Daha sonra yeni değer j değişkenine atanır. Ekrana 31 değeri ve ABC dizgisi yazılır. Bu deyim parantezler kullanılarak aşağıdaki şekilde yazılabilir:

```
j = ++(p->i);
```

İşlem 6: **j = (++p)->i;**

i elemanının değerini almadan önce adres değişkeninin değerini arttırmak için (++p)->i ifadesi kullanılabilir. Çıktıda da görüldüğü gibi, bu deyim sonrasında p adres değişkeni dizideki bir sonraki yapıya işaret eder ve ekrana bu yapının elemanlarının değerleri listelenir. j değişkenine arttırma sonrası erişilen i elemanının değeri (50) atanır. Bu işlemde parantezler kullanılarak operatör etkisi değiştirilmiştir.

İşlem 2: **j = p++->i;**

Çıktıda da görüldüğü gibi, bu deyim ile i elemanının değeri j değişkenine atandıktan sonra adres değişkeninin değeri arttırılarak bir sonraki yapıya işaret etmesi sağlanır. Parantezler kullanılarak yazılan aşağıdaki ifade ile aynıdır:

```
j = (p++)->i;
```

İşlem 7: **j = p->i++;**

Bu deyimde, i elemanının değeri j değişkenine atandıktan sonra arttırılır. Dolayısıyla ekranan j'nin değeri olarak 50, i elemanının değeri olarak ise 51 yazılır.

İşlem 3: **c = *p->s++;**

p->s adres değişkeni "EFG" karakter dizisinin başlangıç elemanına işaret eder. Dolayısıyla bu deyimde, *p->s ile alınan 'E' değeri c değişkenine atanır ve s'in bir sonraki karakter alanına (F karakterinin bulunduğu alan) işaret etmesi sağlanır (*s++ gibi).

İşlem 4: **c = *p++->s;**

s'in işaret ettiği karakter (F) alındıktan sonra p'nin değeri arttırılır. p adres değişkeni dizide bulunan bir sonraki yapıya işaret eder.

İşlem 5: **c = *p--->s;**

Yukarıdaki işlemin tersi. s'in işaret ettiği karakter (M) alındıktan sonra p'nin değeri eksiltilir. p adres değişkeni dizide bulunan bir önceki yapıya işaret eder.

İşlem 8: **c = (*p->s)++;**

Bu deyimde s'in işaret ettiği değer c değişkenine atandıktan sonra arttırılır. s'in işaret ettiği alanda saklanan M karakterinin ASCII değeri arttırıldığında N karakterinin ASCII değerine eşit olacağı için ekrana "NNO" dizgisi yazılır.

Yapı değişkenlerine işaret eden adres dizisi :

Aynı şablona sahip birden fazla yapı değişkenine ihtiyaç duyulduğunda, yapı dizisi bildirilebilir. **struct** proje yapı tipindeki, N sayıda yapı değişkeni için, p yapı dizisi aşağıdaki şekilde bildirilir :

```
struct proje p[ N ];
```

Fakat eleman sayısı arttığında, dizi elemanlarına erişim yavaşlar ve işlemlerde kullanılan bellek miktarı artar. C dilinde, elemanları yapılara işaret eden adres değişkenleri olan adres dizisi bildirilebilir.

Adres dizisi ile erişim daha hızlı gerçekleşir ve yapı dizisi üzerinde yapılan işlemlerde bellek tasarrufu sağlanır. Aşağıda, **struct** proje tipindeki yapıların bellek adresini taşıyan N adet adres değişkeninden oluşan adr_dizi adres dizisi bildirilir:

```
struct proje *adr_dizi[ N ];
```

Örnek 8.3-3'de **typedef** kullanılarak BILGI yapı veri ismi tanımlanır. Yeni veri ismi tanımlanırken büyük harf kullanılarak sözkonusu veri isminin programcı tarafından oluşturulduğu belirtilir ve bu yolla programın okunabilirliği artmış olur. Aynı veri ismi **#define** ön-işlemci komutu kullanılarakta tanımlanabilir. Tek fark **typedef** deyimleri derleyici tarafından, **#define** deyimleri ise ön-işlemci tarafından işlenir. Fakat bazı durumlarda ön-işlemci tarafından yapılamayan karmaşık metin değiştirme işlemleri, **typedef** kullanıldığında derleyici tarafından yapılabilir. Programda BILGI[3] tipinde p yapı dizisi bildirilir. Daha sonra, yine 3 elemanlı adres dizisi adr_dizi bildirilir. İlk **for**

döngüsü ile adr_dizi dizisine p yapı dizisinin elemanı olan yapı değişkenlerinin bellek adresleri atanır. İkinci **for** döngüsünde ise, adres dizisi aracılığıyla erişilen yapıların elemanlarının değerleri listelenir. Programda son olarak iç içe iki **for** döngüsü kullanılarak yapıların isim elemanlarının işaret ettiği dizgilerin karakterleri birer birer listelenir. Örneğin i değişkeninin değeri 0 olduğunda (adr_dizi[0]->isim) adres değişkeni abc dizgisine işaret eder. (adr_dizi[0]->isim)[j] ifadesindeki j değişkeninin değeri dizgi sonunu belirten '\0' karakterine rastlanıncaya kadar arttırılarak abc dizgisi boyunca ilerlenir ve karakterler ekrana listelenir.
Boş karaktere rastlanınca **for** döngüsünden çıkılır ve aynı işlemler p dizisinin bir sonraki yapısının isim elemanı ile tekrarlanır.

Örnek 8.3-3 adryapi3.c programı.

```
#include <stdio.h>

#define EL_SAYI(s)   sizeof(s)/sizeof(s[0])

typedef struct
{
   char  *isim, *adres, *tel_no;
   int yas;
} BILGI;

BILGI p[ ] = {
                    { "abc",    "xxx",    "123 456", 23 },
                    { "def",    "yyy",    "789 012", 34 },
                    { "mno",    "zzz",    "890 123", 30 },
                };

BILGI *adr_dizi[ EL_SAYI(p) ];

#include <stdio.h>

int main(void)
{
   int i, j;

   for ( i = 0; i < EL_SAYI(p); i++)
         *( adr_dizi + i ) = p + i;
   printf("sizeof( p ) : %d, sizeof( adr_dizi ) : %d\n",  sizeof( BILGI[ EL_SAYI(p) ]     ),
                                                          sizeof( BILGI*[ EL_SAYI(p) ]    ) );
   puts("yapi elemanlari :");
   for ( i = 0; i < EL_SAYI(p); i++)
         printf("%4s, %4s, %8s, %3d\n",  adr_dizi[i]->isim,
                                         adr_dizi[i]->adres,
                                         adr_dizi[i]->tel_no,
                                         adr_dizi[i]->yas  );
   for ( i = 0; i < EL_SAYI(p); i++)
         for  (j = 0; (adr_dizi[i]->isim)[ j ] ; j++ )
               printf(" %c\n",  (adr_dizi[i]->isim)[ j ] );

   return 0;
}
```

...Örnek 8.3-3 devam

Çıktı

```
sizeof( p ) : 24, sizeof( adr_dizi ) : 6
yapi elemanlari :
 abc,     xxx,   123 456,  23
 def,     yyy, 789 012,  34
 mno, zzz, 890 123,  30
 a
 b
 c
 d
 e
 f
 m
 n
 o
```

Örnek 8.3-4'de yapı adres değişkeni kullanımları örneklenmiştir. İlk olarak **typedef** kullanılarak yapı veri tipi ismi YAPI tanımlanır. YAPI tipi, tamsayı i değişkeni, tamsayı verilere işaret eden pi adres değişkeni ve dizgilere işaret eden t adres değişkeninden oluşur. Programda, YAPI tipinde 4 yapı değişkeninden oluşan yapı dizisi YDizi, YAPI tipindeki yapı değişkenlerine işaret eden aYapi adres değişkeni ve yine elemanları YAPI tipindeki yapılara işaret eden yapı adres dizisi aDizi bildirilir.

Örnek 8.3-4 adryapi4.c programı.

```
#include <stdio.h>

#define BOY 4
#define ELNo(s)   sizeof(s)/sizeof(s[0])

typedef struct
{
   int  i, *pi;
   char *t;
}  YAPI;

YAPI  YDizi[BOY], *aYapi,  *aDizi[BOY];

int      n[ ]   = {  2,4,1,3 };
char   *s[ ]   = {  "iki",   "dort",   "bir",  "uc"};

void   Listele(  YAPI  ** );
void   Sirala(   YAPI  ** );

int main( void )
{
   int j;

   /* yapi dizisine ve adres dizisine deger atama */
   for ( j = 0; j < BOY; j++ )
```

...Örnek 8.3-4 devam

```
{
    YDizi[j].i= *(n + j);
    YDizi[j].pi = (n + j);
    YDizi[j].t  = *(s + j);
    aDizi[j]    = &YDizi[j];
}
printf( "YDizi'nin eleman sayisi : %d\n",                ELNo(YDizi) );
printf( "YDizi'nin bir elemaninin byte sayisi : %d\n", sizeof(YAPI) );

/* yapi dizisinin elemanlarinin degerlerinin listelenmesi */
for ( j = 0; j < BOY; j++ )
    printf( "YDizi[%d].i : %d, *YDizi[%d].pi : %d, YDizi[%d].t : %s\n",
                                                                    j, YDizi[j].i,
                                                                    j, *YDizi[j].pi,
                                                                    j, YDizi[j].t  );
/* yapi dizisinin elemanlarinin degerlerinin listelenmesi */
for ( j = 0; j < BOY; j++ )
{
    printf("(*(YDizi+%d)).i  : %d,",     j, (*(YDizi+j)).i      );
    printf("*(*(YDizi+%d)).pi : %d,",   j, *(*(YDizi+j)).pi   );
    printf("(*(YDizi+%d)).t  : %s\n",j, (*(YDizi+j)).t      );
}

/* yapi dizisinin elemanlarinin degerlerinin listelenmesi */
aYapi = YDizi;
printf("\naYapi->i : %d, *aYapi->pi : %d, aYapi->t : %s, ", aYapi->i,
                                                            *aYapi->pi,
                                                            aYapi->t );
printf("*(++aYapi)->pi : %d\n", *(++aYapi)->pi );
--aYapi;
Listele( &aYapi );
printf("aYapi->i : %d, *aYapi->pi : %d, aYapi->t : %s\n", aYapi->i,
                                                          *aYapi->pi,
                                                          aYapi->t  );
printf("\nSiralama Oncesi degerler ...\n");
for ( j = 0; j < BOY; j++ )
{
    printf( "aDizi[%d]->i : %d, *aDizi[%d]->pi : %d,",   j, aDizi[j]->i,   j, *aDizi[j]->pi  );
    printf( "aDizi[%d]->t : %s\n",                       j, aDizi[j]->t );
}
Sirala( aDizi );
printf( "\nSiralama Sonrasi degerler ...\n" );
for ( j = 0; j < BOY; j++ )
{
    printf( "aDizi[%d]->i : %d, *aDizi[%d]->pi : %d,",   j, aDizi[j]->i,
                                                         j, *aDizi[j]->pi);
```

...Örnek 8.3-4 devam

```
        printf( "aDizi[%d]->t : %s\n",                    j,  aDizi[j]->t );
   }
   return 0;
}
void Listele( YAPI **aaYapi )
{
   puts("Listele ...  ");
   printf("(*aaYapi)->i : %d, *(*aaYapi)->pi : %d,  (*aaYapi)->t : %s\n",
                                                                    (*aaYapi)->i,
                                                                    *(*aaYapi)->pi,
                                                                    (*aaYapi)->t );
                                                                    (*aaYapi)++;
}
void Sirala( YAPI *aaDizi[ ] )
{
   int n, m;
   YAPI *T;
   for ( n = 0; n < BOY; n++ )
        for ( m = n+1; m < BOY; m++ )
             if ( *aaDizi[n]->pi > *aaDizi[m]->pi )
             {
                  T           = aaDizi[m];
                  aaDizi[m] = aaDizi[n];
                  aaDizi[n]  = T;
             }
}
```

Çıktı

```
YDizi'nin eleman sayisi : 4
YDizi'nin bir elemaninin byte sayisi : 6
YDizi[0].i : 2, *YDizi[0].pi : 2, YDizi[0].t : iki
YDizi[1].i : 4, *YDizi[1].pi : 4, YDizi[1].t : dort
YDizi[2].i : 1, *YDizi[2].pi : 1, YDizi[2].t : bir
YDizi[3].i : 3, *YDizi[3].pi : 3, YDizi[3].t : uc

(*(YDizi+0)).i  : 2,*(*(YDizi+0)).pi : 2,(*(YDizi+0)).t  : iki
(*(YDizi+1)).i  : 4,*(*(YDizi+1)).pi : 4,(*(YDizi+1)).t  : dort
(*(YDizi+2)).i  : 1,*(*(YDizi+2)).pi : 1,(*(YDizi+2)).t  : bir
(*(YDizi+3)).i  : 3,*(*(YDizi+3)).pi : 3,(*(YDizi+3)).t  : uc

aYapi->i : 2, *aYapi->pi : 2, aYapi->t : iki, *(++aYapi)->pi : 4
Listele ...
(*aaYapi)->i : 2, *(*aaYapi)->pi : 2, (*aaYapi)->t : iki
aYapi->i : 4, *aYapi->pi : 4, aYapi->t : dort

Siralama Oncesi degerler ...
aDizi[0]->i : 2, *aDizi[0]->pi : 2,aDizi[0]->t : iki
aDizi[1]->i : 4, *aDizi[1]->pi : 4,aDizi[1]->t : dort
```

...Örnek 8.3-4 devam

```
aDizi[2]->i : 1, *aDizi[2]->pi : 1,aDizi[2]->t : bir
aDizi[3]->i : 3, *aDizi[3]->pi : 3,aDizi[3]->t : uc
Siralama Sonrasi degerler ...
aDizi[0]->i : 1, *aDizi[0]->pi : 1,aDizi[0]->t : bir
aDizi[1]->i : 2, *aDizi[1]->pi : 2,aDizi[1]->t : iki
aDizi[2]->i : 3, *aDizi[2]->pi : 3,aDizi[2]->t : uc
aDizi[3]->i : 4, *aDizi[3]->pi : 4,aDizi[3]->t : dort
```

---

for döngüsü bloğunda, YDizi ve aDizi dizilerine değer atama yapılır. YDizi'deki yapıların i ve pi elemanlarına sırasıyla, n dizisinin elemanlarının değerleri ve adresleri atanır; t elemanına ise dizgilere işaret eden s adres dizisinin elemanlarının değerleri olan dizgi adresleri atanır. Ayrıca aDizi adres dizisine, YDizi'de bulunan yapıların bellek adresleri atanır. Böylece, aDizi'nin elemanları olan adres değişkenleri, YDizi'nin elemanları olan yapılara işaret eder. YDizi, YAPI[BOY] tipindedir. aDizi ise YAPI*[BOY] tipindedir.

Atama deyimi,

```
aYapi  = YDizi;
```

ile aYapi adres değişkeninin YDizi'nin başlangıç elemanına işaret etmesi sağlanır. Bunu izleyen printf satırında, aYapi kullanılarak yapı dizisinin başlangıç elemanının değerleri listelenir. Ok operatörü (->), indirek değer operatöründen (*) daha yüksek önceliğe sahip olduğu için, *aYapi->pi ifadesinde parantez kullanılmamıştır. Fakat okunabilirliği arttırmak için *(aYapi->pi) ifadesi kullanılabilir. aYapi adres değişkeni başlangıçta, YDizi[0] yapısına işaret ettiği için,

```
*(++aYapi)->pi
```

ifadesi dizideki bir sonraki yapının elemanlarına erişimi sağlar. aYapi adres değişkenine -- operatörü uygulanarak, tekrar dizi başlangıcına işaret etmesi sağlanır.

Listele fonksiyonu başlığında bildirilen çift adres değişkeni aaYapi, çağrı ile aktarılan aYapi adres değişkeninin adresini alır (Bu bildirim, YAPI *aaYapi[ ] şeklinde yapılabilir). Böylece çağıran bloktaki aYapi adres değişkeninin taşıdığı adres değerini değiştirebilir. Çıktıda da görüldüğü gibi Listele fonksiyonu, aaYapi aracılığı ile yapı elemanlarına erişir ve (*aaYapi)++ ifadesi ile aYapi adres değişkeninin değerini arttırarak dizideki bir sonraki yapıya işaret etmesini sağlar ve çağıran bloğa döner. Listele bloğunda yer alan *(*aaYapi)->pi ifadesinde yine ilave parantezlere gerek yoktur. Bunun yerine *((*aaYapi)->pi) kullanımı sadece okunabilirliği arttırır. Listele bloğundaki, (*aaYapi)->i ifadesinde *aaYapi dışındaki parantezler gereklidir. Çünkü -> operatörü daha yüksek önceliğe sahiptir ve parantezler kullanılmadığı taktirde, *aaYapi->i ifadesi *(aaYapi->i) ile aynıdır ve i bir adres değişkeni olmadığı için hata oluşur.

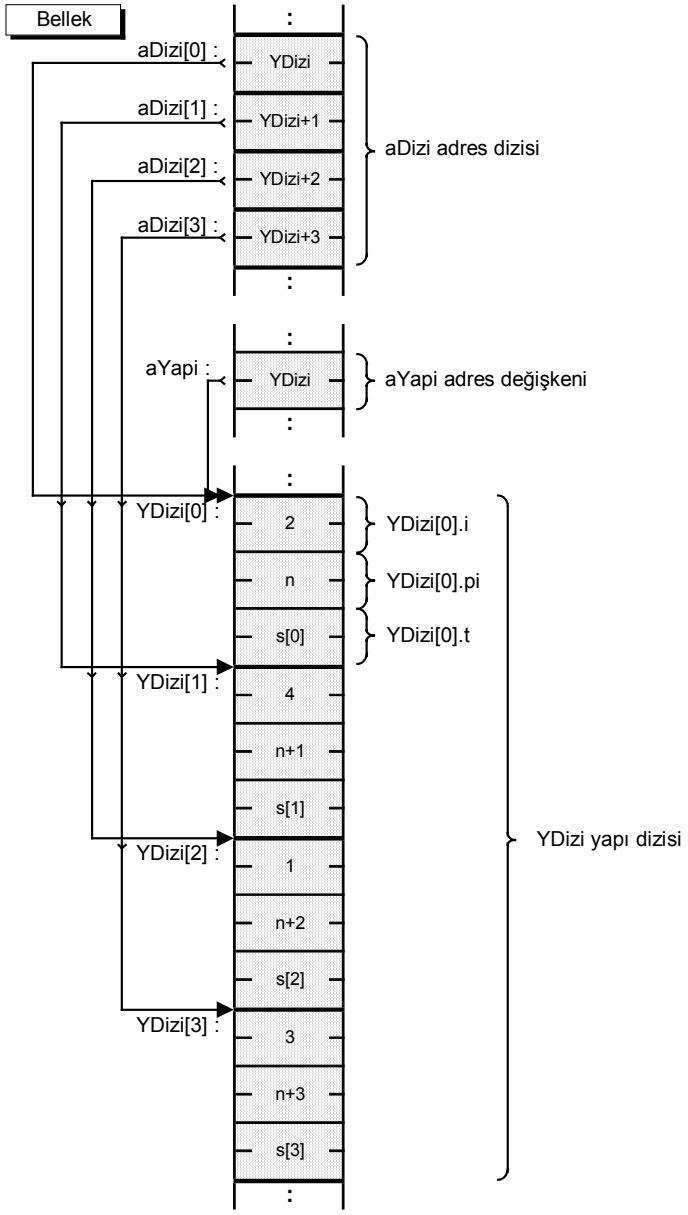
Listele çağrısından sonra aYapi adres değişkeni kullanılarak gerçekleştirilen erişim, dizideki bir sonraki yapıyı listeler.

Programda son olarak aDizi adres dizisi kullanılarak yapı dizisi elemanları sıralanır. Sıralama işlemi adres dizisi üzerinde yapıldığı için, YDizi'deki sıralama değişmez (Şekil 8.3-3). Sirala fonksiyonuna argüman olarak aDizi aktarılır. Sirala fonksiyonu adres dizisinin başlangıç elemanının bellek adresini alarak aDizi dizisinin elemanlarının değerlerine erişir. Bu adres değerleri aracılığı ile yapının elemanlarına erişerek aşağıdaki test ifadesi ile karşılaştırma yapar:

```
*aaDizi[n]->pi > *aaDizi[m]->pi
```

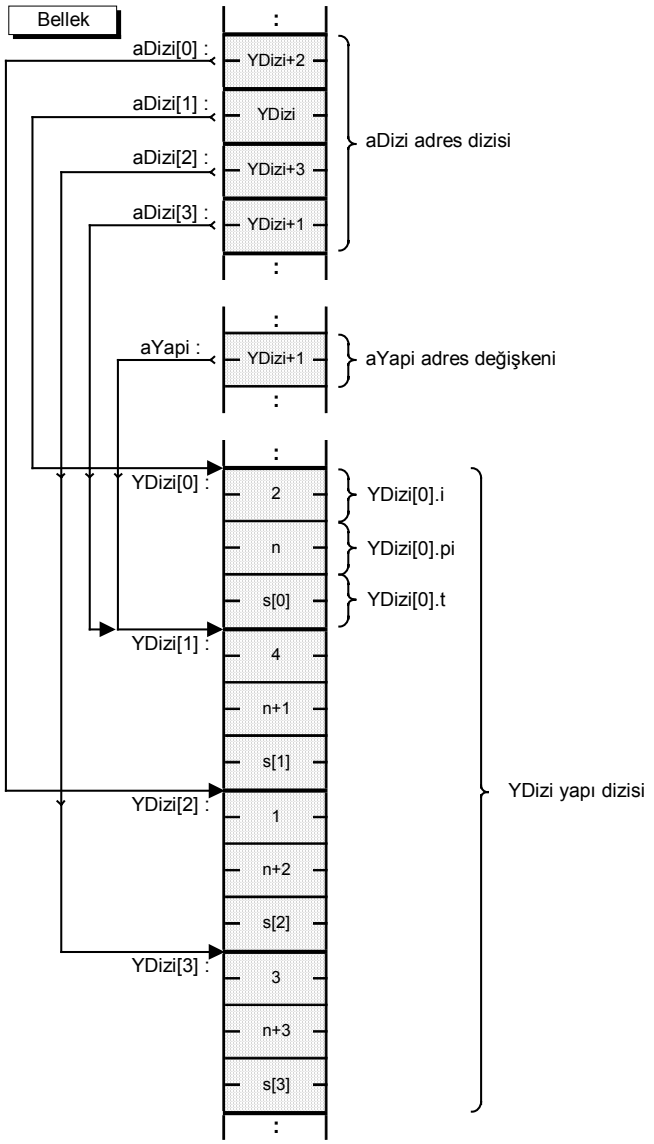
Daha sonra, adres değerleri aktarılarak sıralama yapılır. Çağrı sonrası aDizi adres dizisi kullanılarak sıralanmış değerler ekrana yazılır. Şekil 8.3-3'de Sirala ve Listele çağrıları öncesi ve sonrası sembolik bellek görünümü verilmiştir.

Şekil 8.3-3 Sıralama öncesi sembolik bellek görünümü.



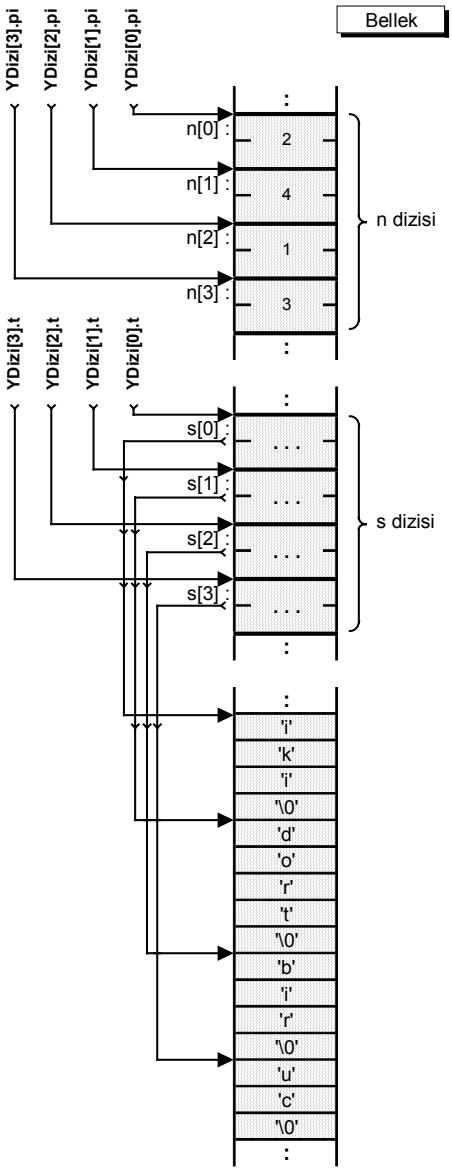
Şekil 8.3-3 devam...

Sıralama sonrası aDizi adres dizisinin ve Listele fonksiyonu çağrıldıktan sonra aYapi adres değişkeninin sembolik bellek görünümü:



Şekil 8.3-3 devam...

n ve s dizilerinin sembolik bellek görünümü:



Yapı elemanı olarak fonksiyona işaret eden adres değişkeni :

Bir fonksiyon ismi, fonksiyonun adresini verir. Önceki bölümlerde anlatıldığı gibi, basit bir fonksiyona işaret eden adres değişkeni aşağıdaki şekilde oluşturulur:

fonksiyon bildirimi:

```
void fonk1( void );
```

fonksiyona işaret eden adres değişkeni bildirimi:

```
void (*adr_fonk)( void );
```

atama ile adres değişkeninin fonksiyona işaret etmesi sağlanır:

```
adr_fonk = fonk1;
```

Aşağıdaki örnek programda bildirilen YDizi yapı dizisi, elemanları karakter dizisine ve fonksiyona işaret eden adres değişkenleri olan FYapi tipindeki yapılardan oluşur. YDizi[0] ve YDizi[1], FYapi tipinde yapılardır.

YDizi[0].adr_fonk ve YDizi[1].adr_fonk yapı elemanları ise fonksiyona işaret eden adres değişkenleridir. Bu ifadelerde indeks operatörü ([ ]) ile dizinin elemanı olan herhangi bir yapı değişkenine ve yapı elemanı operatörü (.) kullanılarakta bu yapının elemanı olan adres değişkenine erişim gerçekleştirilir. Programda görüldüğü gibi fonksiyon çağrıları, (*YDizi[i].adr_fonk)( ) deyimi ile yapılır. Nokta operatörü (.), indirek değer operatöründen (*) daha yüksek önceliğe sahip olduğu için YDizi[i].adr_fonk ifadesi parantezlerle sınırlandırılmamıştır.

Örnek 8.3-5 adryapi5.c programı.

```
#include <stdio.h>
#define EL_SAY(s)  sizeof(s)/sizeof(s[0])
typedef struct       /* etiket ismi kullanilabilir */
{
  char   *FAdi;
  void   (*adr_fonk)(void);
}  FYapi;
void fonk1( void ), fonk2( void ), fonk3( void );

FYapi    YDizi[ ] =  {  {  "fonk1", fonk1 }, {  "fonk2", fonk2 }  };
int main(void)
{
  int i;
```

...Örnek 8.3-5 devam

```
  for ( i = 0; i < EL_SAY( YDizi ); i++ )
  {
       printf(" YDizi[%d].FAdi : %s --> ", i, YDizi[i].FAdi );
       (*YDizi[ i ].adr_fonk)();
  }
  puts( " fonksiyon adresleri : ");
  printf("\t\t fonk1 : %p, fonk2 : %p, fonk3 : %p\n", fonk1, fonk2, fonk3 );
  puts( " eleman adresleri : ");
  printf("\t\t  YDizi[0].adr_fonk : %p,   YDizi[1].adr_fonk : %p\n",    YDizi[0].adr_fonk,
                                                                         YDizi[1].adr_fonk );
   /* fonk2 referansinin fonk3 olarak degistirilmesi */
  YDizi[1].adr_fonk = fonk3;
  YDizi[1].FAdi = "fonk3";
  puts( " atama sonrasi eleman adresleri : ");
  printf("\t\t  YDizi[0].adr_fonk : %p,   YDizi[1].adr_fonk : %p\n",
                YDizi[0].adr_fonk,          YDizi[1].adr_fonk );
  printf(" YDizi[1].FAdi : %s --> ", YDizi[ 1 ].FAdi );
  (*YDizi[ 1 ].adr_fonk)();
  return 0;
}
void fonk1(void)
{
  puts("fonksiyon 1 ...");
}
void fonk2(void)
{
  puts("fonksiyon 2 ...");
}
void fonk3(void)
{
  puts("fonksiyon 3 ...");
}
```

Çıktı

```
YDizi[0].FAdi : fonk1 --> fonksiyon 1 ...
YDizi[1].FAdi : fonk2 --> fonksiyon 2 ...
fonksiyon adresleri :
        fonk1 : 00AC, fonk2 : 00BC, fonk3 : 00CC
eleman adresleri :
        YDizi[0].adr_fonk : 00AC,  YDizi[1].adr_fonk : 00BC
atama sonrasi eleman adresleri :
        YDizi[0].adr_fonk : 00AC,  YDizi[1].adr_fonk : 00CC
YDizi[1].FAdi : fonk3 --> fonksiyon 3 ...
```

Çıktıda listelenen adres değerlerinden de anlaşılacağı gibi programda atama ile YDizi[1].adr_fonk adres değişkeninin fonk2 yerine fonk3 fonksiyonuna işaret etmesi sağlanır.

Yapı elemanlarının *offset*'lerinin hesaplanması :

Bir yapıya işaret eden adres değişkeni ile herhangi bir diziye işaret eden adres değişkeni kavramsal olarak aynıdır. Dizi adres değişkeni dizinin her elemanına erişebildiği gibi, yapı adres değişkeni de yapının her elemanına erişebilir. Fakat kullanım şekilleri farklıdır. Dizideki her eleman aynı tipte olduğu için, eşit büyüklükte ve sıralı bellek alanlarında bulunurlar. Dolayısıyla herhangi bir dizi elemanına, bu elemanın dizinin başlangıcından itibaren kaç byte sonra bulunduğu (yani *offset* miktarı) belli olduğu için, sıralanışa bağlı olarak kullanılan sayısal indeks yardımı ile erişilebilir. Bu amaçla, aşağıdaki her iki ifade de kullanılabilir:

dizi ismi ve indeks ifadesi : dizi[i]

adres değeri ve offset ifadesi : *(dizi+i)

Tamsayı i değişkeni ilk ifadede indeks değişkeni, ikincisinde ise offset değişkenidir. Fakat yapının her bir elemanı farklı tipte olabileceği için, bu şekilde oluşturulan herhangi bir ifade kullanılarak elemanlara erişim mümkün olmaz. Her yapı elemanının sembolik bir ismi vardır ve bu isim, ayrı bir *tip* ve *offset* ifade eder. Yapı elemanları bellekte, yapı içinde bildirildikleri sırada ve artan bellek adreslerinde bulunurlar.

ANSI C'nin bir özelliği olarak, yapı elemanlarının offset değerlerinin taşınabilir bir şekilde saptanabilmesi için stddef.h başlık dosyasında offsetof makrosu tanımlanmıştır. Bu makro ön-işlemci tarafından genişletildiğinde size_t tipinde sabit bir ifadeye dönüşür ve bir yapı elemanının yapı başlangıcından itibaren offset değerini byte olarak verir:

offsetof(*yapi_veri_tipi*, *eleman_ismi*); /* stddef.h */

yapi_veri_tipi'deki herhangi bir p yapısı için &p.*eleman_ismi* ifadesi bir adres sabiti olmalıdır. Bu makro daha sonra anlatılacak olan bit-alanları ile kullanılamaz.

Aşağıdaki örnek programda ilk olarak **sizeof** operatörü kullanılarak s yapısının bellek alanı büyüklüğü ve elemanlarının offset değerleri hesaplanır. Çıktıda görüldüğü gibi, adr elemanının &s.adr ifadesi ile alınan bellek adresi (4628) ve hesaplanan bellek adresi (4627) aynı değildir. Donanıma bağlı olarak, yapı elemanı olan veriler derleyici tarafından 2 (yada 4) ile bölünebilen bellek adreslerine yerleştirilir (bu işlem *alignment* olarak adlandırılır). Bu nedenle, bellekte bulunan herhangi bir yapı içinde isimsiz boş alanlar (donanıma bağlı olarak bir yada daha fazla byte) oluşabilir.

Örnek 8.3-6 adryapi6.c programı.

```
#include <stdio.h>
#include <stddef.h>

struct YAPI
{
   int   i;
   double  f;
   char   c,*adr;
};

int main(void)
{
   struct YAPI  s = { 1, 0.7, 'A', "dizgi" };

   puts("elemanlarin degerleri ve bellek adresleri:");
   printf("\ti : %d, f : %1.1f, c : %c, adr : %s \n",  s.i,  s.f,  s.c,  s.adr );
   printf("\t&s : %d, &s.i : %d, &s.f : %d, &s.c : %d, &s.adr : %d\n",
          &s, &s.i, &s.f, &s.c, &s.adr );

   puts("elemanlarin -hesaplanan- offset degerleri :");
   printf("\ts.i : %d, s.f : %d, s.c : %d,  s.adr : %d byte.\n",
          /* s.i      */   0,
          /* s.f      */   sizeof( s.i ),
          /* s.c      */   sizeof( s.i )  + sizeof( s.f ),
          /* s.adr    */   sizeof( s.i )  + sizeof( s.f )  + sizeof( s.c )  );
   printf("s yapisinin -hesaplanan- byte sayisi : %d byte.\n",
          sizeof( s.i )  + sizeof( s.f )  + sizeof( s.c ) + sizeof( s.adr)  );

   printf("adr elemaninin -hesaplanan- bellek adresi : %d\n",
          (unsigned)&s + sizeof(s.i) + sizeof( s.f ) + sizeof( s.c ) );

   puts("\nelemanlarin gercek offset degerleri (offsetof makrosu):");
   printf("\ts.i : %d, s.f : %d, s.c : %d,  s.adr : %d byte.\n",
          /* s.i      */   offsetof( struct YAPI, i ),
          /* s.f      */   offsetof( struct YAPI, f ),
          /* s.c      */   offsetof( struct YAPI, c ),
          /* s.adr    */   offsetof( struct YAPI, adr ) );
   printf("s yapisi : %d byte.\n", sizeof(s) );
   printf("adr elemaninin bellek adresi : %d\n",
          (unsigned)&s + offsetof( struct YAPI, adr ) );

   return 0;
}
```

Çıktı

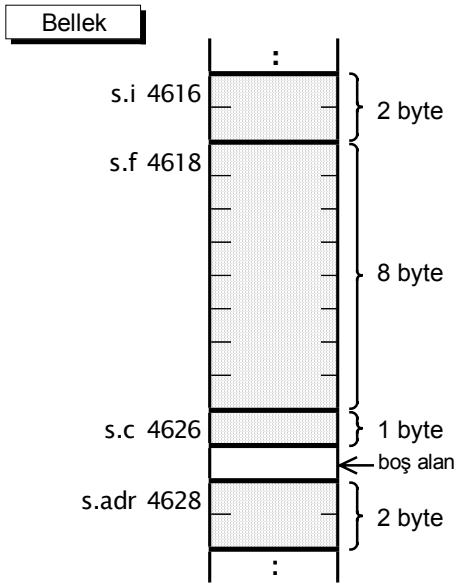
```
elemanlarin degerleri ve bellek adresleri:
        i : 1, f : 0.7, c : A, adr : dizgi
        &s : 4616, &s.i : 4616, &s.f : 4618, &s.c : 4626, &s.adr : 4628
elemanlarin -hesaplanan- offset degerleri :
        s.i : 0, s.f : 2, s.c : 10,  s.adr : 11 byte.
```

... Örnek 8.3-6 Çıktı devam

```
s yapisinin -hesaplanan- byte sayisi : 13 byte.
adr elemaninin -hesaplanan- bellek adresi : 4627
elemanlarin gercek offset degerleri (offsetof makrosu):
        s.i : 0, s.f : 2, s.c : 10,  s.adr : 12 byte.
s yapisi : 14 byte.
adr elemaninin bellek adresi : 4628
```

Örnek 8.3-6'da s yapısının c elemanı, bulunduğu adresteki (4626) bir byte'lık bellek alanını kullanır. adr elemanı yukarıda bahsedilen donanım özelliğinden dolayı, hemen c elemanını izleyen bellek adresi (4627) yerine bir sonraki çift adrese (4628) yerleştirilir. Bu nedenle, c elemanı ile adr elemanı arasında bir byte'lık *boş alan* (*hole*) oluşur (Şekil 8.3-4).

Şekil 8.3-4 s yapısının sembolik bellek görünümü.



Çýktýda da görüldüğü gibi, adr'nin offset değerini elemanlarýn byte sayýlarýnýn toplamýný kullanarak hesaplayan,

```
sizeof(s.i) + sizeof(s.f) + sizeof(s.c)
```

ifadesi boş alanı dikkate almadığı için hatalı sonuç verir. Ayrıca aynı teknik kullanılarak hesaplanan s yapısının byte sayısı (13) ve adr elemanının adresi (4627) hatalıdır. Fakat çıktıda görüldüğü gibi offsetof makrosu kullanılarak doğru değerler elde edilir. Sonuç olarak bir yapının bellek alanı, elemanlarının **sizeof** ile alınan byte sayılarının

toplamına eşit olmayabilir. Fakat **sizeof** operatörü yapının tamamına uygulandığında, yapı içinde bulunan boş alanları da dikkate aldığından dolayı doğru sonucu verir. Aynı şekilde, Örnek 8.1-1'de **sizeof** operatörü p2 yapısı için 19 yerine 20 byte sonucunu verir.

Pek çok C derleyicisi, yapı elemanlarının bellek düzeni üzerinde kontrol sağlayan derleyici seçeneklerine yada derleyici komutlarına sahiptir. Yukarıda bahsedilen donanım özelliğinden başka, veri tiplerinin byte sayıları da donanıma göre değişir.

Dolayısıyla yapı elemanlarının offset değerleri de donanıma göre farklılık gösterir (2-byte bellek adreslerine sahip donanımdaki değerler, 4-byte bellek adreslerine sahip donanımdakilerden farklı olacaktır). Sonuç olarak, C programlarının taşınabilir olması ve hatasız çalışabilmesi için, yapı elemanlarının bellek adresleri üzerinde yapılan işlemlerde, sayısal offset değerleri kullanılmamalıdır. Bu işlemler, yukarıda anlatılan teknikler kullanıldığında, C derleyicisi tarafından hatasız olarak gerçekleştirilir.

Örnek 8.3-7'de global yapı şablonları AltProje ve Proje tanımlanır. Proje şablonunun elemanlarından biri AltProje şablonuna sahip 2 elemanlı Alt yapı dizisidir. Programda malloc fonksiyonu kullanılarak tek bir Proje yapısı için bellek alanı ayrılır ve bu alanın bellek adresi ap yapı adres değişkenine atanır. Daha sonra bu adres değişkeni ile yapı elemanlarına erişim gerçekleştirilir ve elemanlara çeşitli değerler atanır. Programın sonunda atanan bu değerler ekrana listelenir. ■

Örnek 8.3-7 adryapi7.c programı.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct AltProje
{
  char  isim[15];
  long  Basla;
  long  Bitis;
};

struct Proje
{
  char  isim[15];
  long  Basla;
  long  Bitis;
  struct AltProje Alt[2];
};

struct Proje *ap;

int main(void)
{
  if (  (  ap = (struct Proje *)malloc( sizeof(struct Proje) ) ) == NULL )
```

...Örnek 8.3-7 devam

```
  {
        puts("hata: bellek alani ayrilamadi");
        exit(0);
  }
  strcpy( ap->isim, "Ana Proje " );
  ap->Basla = 1994;
  ap->Bitis = 1997;
  strcpy( (ap->Alt[0]).isim, "Alt Proje1" );
  (ap->Alt[0]).Basla = 1994;
  (ap->Alt[0]).Bitis = 1995;
  strcpy( (ap->Alt[1]).isim, "Alt Proje2" );
  (ap->Alt[1]).Basla = 1995;
  (ap->Alt[1]).Bitis = 1997;
  printf("PROJE: %s, Basla : %ld, Bitis : %ld\n", ap->isim, ap->Basla, ap->Bitis );
  printf("PROJE: %s, Basla : %ld, Bitis : %ld\n",  (ap->Alt[0]).isim,
                                                   (ap->Alt[0]).Basla,
                                                   (ap->Alt[0]).Bitis );
  printf("PROJE: %s, Basla : %ld, Bitis : %ld\n",  (ap->Alt[1]).isim,
                                                   (ap->Alt[1]).Basla,
                                                   (ap->Alt[1]).Bitis );
  return 0;
}
```

Çıktı

```
PROJE:   Ana Proje ,  Basla : 1994,  Bitis : 1997
PROJE:   Alt Proje1,  Basla : 1994,  Bitis : 1995
PROJE:   Alt Proje2,  Basla : 1995,  Bitis : 1997
```

8.4 Yapı Değişkenleri ve Fonksiyonlar

Yapılar fonksiyonlara argüman olarak aktarılabilir. Bu durumda, yapının tamamı fonksiyonun argüman alanına kopyalanır. Fonksiyon, aktarılan yapının lokal kopyasını aldığı için çağıran bloktaki yapı elemanlarına erişemez. Fakat fonksiyona yapının bellek adresi aktarıldığında, çağıran bloktaki yapı elemanlarına erişim mümkün olur. Sonuç olarak, yapılar fonksiyonlara aşağıdaki şekillerde aktarılabilir:

- yapının bellek adresinin aktarılmasıyla :

  Bunun için fonksiyon argümanı olarak, yapıya işaret eden adres değişkeni yada yapı değişkenine adres operatörü (&) uygulanması ile oluşturulan ifade kullanılır.

Ayrıca bir fonksiyon çağrıda argüman olarak yapı adresi bekliyor ise, bu fonksiyonun tanımında yer alan ve argümana karşılık gelen parametre de yapı adres değişkeni olarak bildirilmelidir.

- çağrıda yapı ismini argüman olarak kullanarak yapının tamamının kopyasının aktarılması ile:

  Fakat bu durumda fonksiyon içinden çağıran bloktaki yapıya erişilemez.

Aşağıdaki örnekte, p1 ve p2 yapısı listele fonksiyonuna argüman olarak aktarılır. listele fonksiyonu bu yapıların lokal kopyalarını alır ve programın başında tanımlanan global şablon kullanılarak bildirilen proje parametresine atar. Daha sonra alınan bu değerleri ekrana listeler. listele fonksiyonu çağıran blokta yer alan yapıya erişemez. Programda görüldüğü gibi yapı şablonu etiket ismi, yapı elemanı isimleri yada fonksiyonun parametre isimleri aynı olabilir (listele fonksiyonunun proje parametresi).

Örnek 8.4-1 yapfonk1.c programı.

```
#include <stdio.h>
struct proje
{
   char   *isim;
   long   basla;
   long   son;
};
void listele( struct proje );
int main(void)
{
   static struct proje  p1  = {  "proje1",     0L,       0L       };
   static struct proje  p2  = {  "proje2",     1994L,  2000L   };
   listele(   p1   );
   listele(   p2   );
   p1 = p2;
   puts( "atama sonrasi yapi elemanlarinin degerleri :" );
   listele(   p1   );
   listele(   p2   );
   return 0;
}
void listele( struct proje proje )
{
    printf("%6s %4ld %4ld\n",  proje.isim,  proje.basla,  proje.son );
}
```

...Örnek 8.4-1 devam

Çıktı

```
proje1        0        0
proje2     1994     2000
atama sonrasi yapi elemanlarinin degerleri :
proje2     1994     2000
proje2     1994     2000
```

Programda, p2 yapısı p1 yapısına atanır ve atama işlemi sonrası elemanların değerleri tekrar listele fonksiyonu çağrılarak ekrana listelenir.

ANSI C'de bir yapı, aynı tipte (aynı şablona sahip) bir başka yapıya atanabilir. Fakat boş alanlar bir yapı diğerine atanırken, yapılar fonksiyonlara argüman olarak aktarılırken yada fonksiyonlardan değer olarak döndürme sırasında kopyalanmayabilir. İki yapı == operatörü ile karşılaştırılamaz. Yapılarda boşluk (*padding bytes*) bulunmadığından emin olunduğu taktirde string.h başlık dosyasında bulunan memcmp fonksiyonu kullanılarak karşılaştırma yapılabilir. Açık olarak bildirim sırasında ilk değer atama yapılmadığında otomatik yapıların ilk değerleri belirsizdir. Bu nedenle boşluklarda bulunan değerler de belirsiz olacaktır. Ayrıca boşluklar yapının bir parçası olmasına karşın isimsiz alanlar oldukları için yapı değişkeni **static** yada global bildirildiği durumlarda bile bu alanlarda bulunan ilk değerler 0 yerine belirsiz değerler olacaktır. Bu durum yapıların bir bütün olarak karşılaştırılmasını olanaksız kılar. Bir yapı için calloc fonksiyonu (stdlib.h) kullanılarak bellek alanı ayrıldığı taktirde, ayrılan bellek alanındaki tüm bit değerleri boşluk alanları da dahil olmak üzere 0 olur. Fakat bu sıfırlama işlemi malloc yada realloc fonksiyonları (stdlib.h) tarafından yapılmaz. memset fonksiyonu (string.h) kullanılarak bellek alanı sıfırlanabilir ve daha sonra memcmp ile karşılaştırma işlemi yapılabilir. Sonuç olarak karşılaştırma işlemi elemanlar karşılaştırılarak yapılmalıdır. Bu taşınabilir bir tekniktir. C programlarında yapıların atanması ve karşılaştırılması pek sık gerek duyulan işlemler değildir.

Programda p1 ve p2 yapıları oluşturulan program kodu ve hızı açısından daha verimli olacağı için blok içi **static** olarak bildirilmiştir.

Örnek 8.4-2'de, struct proje tipinde p yapı dizisi ve s yapısı bildirilir. Daha sonra yapı dizisinin herbir elemanının adres operatörü ile alınan bellek adresi proje_oku fonksiyonuna argüman olarak aktarılır. Yapıya ekrandan değer okuyan proje_oku ve yapının değerlerini ekrana listeleyen proje_list fonksiyonlarının tanımında parametre olarak, struct proje tipindeki yapılara işaret eden adres değişkeni bildirilir. Dolayısıyla, her iki fonksiyon da aldıkları adres değerlerini bu adres değişkenine atayarak, çağıran blokta yer alan yapı (**main** fonksiyonunun lokal yapısı) üzerinde işlemler yapar.

Örnek 8.4-2 yapfonk2.c programı.

```
#include <stdio.h>
#include <string.h>

#define EL_SAYI       3
#define DIZGI_BOY   10

struct proje
{
   char isim[ DIZGI_BOY + 1 ];
   long basla;
   long son;
};

void proje_oku ( struct proje * );
void proje_list( struct proje * );

int main(void)
{
   static struct proje s = { "proje s", 1991L, 2000L  };
   static struct proje p[ EL_SAYI ];
   int i;

   for ( i = 0; i < EL_SAYI; ++i )
         proje_oku ( &p[i] );

   for ( i = 0; i < EL_SAYI; ++i )
         proje_list( &p[i] );

   proje_list( &s );
   printf( "%10s %4ld %4ld\n",  s.isim, s.basla, s.son );

   return 0;
}

void proje_oku( struct proje *ap )
{
   printf( "proje ismi, baslama, bitis ? " );
   scanf( "%10s%ld%ld",  ap->isim,  &ap->basla,  &ap->son );
}

void proje_list( struct proje *ap )
{
   printf("%10s %4ld %4ld\n",  ap->isim, ap->basla, ap->son );

   if ( !strcmp( ap->isim, "proje s" ) )
         strcpy( ap->isim, "PROJE S" );
}
```

...Örnek 8.4-2 devam

Bilgi Girişi

```
proje ismi, baslama, bitis ?   proje1 1991  1992
proje ismi, baslama, bitis ?   proje2 1992  1994
proje ismi, baslama, bitis ?   proje3 1994  1995
```

Çıktı

```
   proje1  1991 1992
   proje2  1992 1994
   proje3  1994 1995
  proje s  1991 2000
  PROJE S  1991 2000
```

---

proje_list fonksiyonu sadece yapı elemanlarının değerlerini listelediği için, çağıran blokta yer alan yapıya erişimi gerekmez. Yani proje_list fonksiyonu, parametre olarak struct proje tipindeki adres değişkeni yerine, struct proje tipindeki herhangi bir yapı değişkeni ile tanımlanabilir. Bu durumda, çağrı ile yapının tamamı aktarılır ve fonksiyon yapının lokal kopyasını alır. Programda ayrıca, çağıran bloktaki yapı elemanlarının değerlerine adres değişkeni aracılığıyla erişimi göstermek için proje_list fonksiyonu içinde s yapısının isim elemanına "PROJE S" dizgisi atanır.
Bellek tasarrufu sözkonusu olduğunda, adres aktarımı tercih edilmelidir. Programda her iki fonksiyona da yapılan çağrılarda, yapı dizisi elemanlarından birinin adresi aktarıldı. Fonksiyonlar, tekbir yapı adresi alacak şekilde tanımlandığı için, yapı dizisinin herhangi bir elemanın adresinin aktarımı ile tek bir yapının adresinin aktarımı arasında fark olmayacaktır. Dolayısıyla, proje_list(&p[i]) ve proje_list(&s) çağrılarının her ikisi de, tek bir proje yapısının bellek adresini aktarır.

C dilinde bir diziyi bir fonksiyona değeri ile aktarmak mümkün değildir. Ancak dizinin adresi aktarılabilir. Dolayısıyla fonksiyon, dizinin bellek alanlarına erişebilir. Fakat herhangi bir dizinin, aşağıdaki indirek metod kullanılarak adresi yerine değerleri aktarılabilir ve bu yolla fonksiyonun asıl diziye erişimi önlenmiş olur.
Bu amaçla Örnek 8.4-3'de elemanı dizi olan global yapı değişkeni s tanımlanır. Bu yapının elemanı olan Dizi dizisine, **main** bloğu içinde strcpy fonksiyonu kullanılarak "Dizilerin degerleri " dizgisi (karakter dizisi) kopyalanır. Yapılar fonksiyonlara değerleri ile aktarıldığı için, s yapısı fonk fonksiyonu çağrısında argüman olarak kullanılır ve bu yolla aynı dizgi fonk fonksiyonuna aktarılır. Programda da görüldüğü gibi fonk fonksiyonunun parametre ismi global yapı değişkeni ile aynı olabilir.

Örnek 8.4-3 yapfonk3.c programı.

```
#include <stdio.h>
#include <string.h>

struct YAPI
{
  char Dizi[ 50 ];
};

struct YAPI s;

void fonk( struct YAPI s )
{
    strcat( s.Dizi, "ile aktarilmasi ... " );
    printf("fonk icinde s.Dizi : \n\t%s\n", s.Dizi );
}
int main(void)
{
  strcpy( s.Dizi,  "Dizilerin degerleri " );
  printf("fonk cagrisi oncesi - main - s.Dizi :  \n\t%s\n", s.Dizi );
  fonk( s );
  printf("fonk cagrisi sonrasi - main - s.Dizi :  \n\t%s\n", s.Dizi );
  return 0;
}
```

Çıktı

```
fonk cagrisi oncesi - main - s.Dizi :
        Dizilerin degerleri
fonk icinde s.Dizi :
        Dizilerin degerleri ile aktarilmasi ...
fonk cagrisi sonrasi - main - s.Dizi :
        Dizilerin degerleri
```

Çıktıda da görüldüğü gibi Dizi dizisi, fonk bloğu içinde yeni bir dizgi eklendiği halde değişmeden kalır. Fakat bu uygulamanın bazı sakıncaları vardır. Fonksiyon çağrısı sırasında argüman değerleri programın yığıt (*stack*) alanına kopyalanır. Eğer bu yolla büyük bir dizi aktarılıyor ise, dizi elemanları sınırlı yığıt alanında çok yer kaplar. Fakat dizi isminin argüman olarak yer aldığı bir çağrıda, tek bir adres değeri aktarılacağı için fazla yığıt alanı kullanılmamış olur.

Bir fonksiyon, yapı değeri döndürecek şekilde tanımlanabilir. Aşağıdaki programda struct SAYI tipinde s1, s2 ve stoplam yapıları bildirilir. Ayrıca prototipte de görüldüğü gibi, struct SAYI tipindeki iki yapıyı argüman olarak alan ve bu yapıların elemanlarının toplamını yine aynı tipteki n yapısı ile döndüren YapiTopla fonksiyonu tanımlanır. Fonksiyon çağrısı ile YapiTopla fonksiyonunun argüman alanına s1 ve s2 yapılarının a

ve b elemanlarının değerleri kopyalanır. Çağrı satırına döndürülen değerler stoplam yapısına atanır. Böylece aynı tipteki iki yapının elemanlarının değerlerinin toplamı, argüman olarak yapı değeri alan ve yine yapı değeri döndüren bir fonksiyon kullanılarak elde edilir.

Örnek 8.4-4 yapfonk4.c programı.

```
#include <stdio.h>

struct SAYI
{
  int a, b;
};

struct SAYI  YapiTopla( struct SAYI, struct SAYI );

int main(void)
{
  struct SAYI s1 = { 10, 20 }, s2 = { 50, 100 }, stoplam;

  printf( "s1.a : %d, s1.b : %d\n", s1.a, s1.b );
  printf( "s2.a : %d, s2.b : %d\n", s2.a, s2.b );

  stoplam = YapiTopla( s1, s2 );
  printf( "stoplam.a : %d, stoplam.b : %d\n", stoplam.a, stoplam.b );

  return 0;
}

struct SAYI YapiTopla( struct SAYI i, struct SAYI j )
{
  struct SAYI n;

  n.a = i.a + j.a;
  n.b = i.b + j.b;

  return ( n );
}
```

Çıktı

```
s1.a  :  10, s1.b  :  20
s2.a  :  50, s2.b  :  100
stoplam.a : 60, stoplam.b : 120
```

Örnek 8.4-5'de fonk fonksiyonuna yapı adresi aktarılır ve fonksiyon yapıya işaret eden adres değişkeni döndürür. fonk fonksiyonu içinde malloc fonksiyonu kullanılarak P1 şablonuna sahip tek bir yapı için bellek alan ayrılır ve bu alanın adresi yine aynı fonksiyon içinde tanımlanan ap1 yapı adres değişkenine atanır. Bu adres değişkeni kullanılarak bellekte yapı için ayrılan alanlara çeşitli değerler atanır. fonk çağrısı, ap1'in taşıdığı adres değerini döndürür ve çağrı satırında bu değer s1 yapı adres değişkenine atanır. s1 ile oluşturulan ifadeler kullanılarak bu alanlara atanan değerler ekrana listelenir. ■

Örnek 8.4-5 yapfonk5.c programı.

```
#include <stdio.h>
#include <stdlib.h>

struct P1
{
   char   *m[3];
   int     n[3];
};

struct P2
{
   char   *m;
   int     n;
};

struct P1 *fonk( struct P2 * );

int main(void)
{
   struct P1 *s1;
   struct P2  s2 = { "bir", 1 };

   s1 = fonk( &s2 );

   printf("%s : %d\n", s1->m[0], s1->n[0] );
   printf("%s : %d\n", s1->m[1], s1->n[1] );
   printf("%s : %d\n", s1->m[2], s1->n[2] );

   return 0;
}

struct P1 *fonk( struct P2 *ap2 )
{
   struct P1 *ap1;

   if (  (  ap1 = (struct P1 *)malloc( sizeof( struct P1 ) ) )
          == NULL )
   {
        puts("fonk hata: bellek alani ayrilamadi");
        exit(0);
   }
   ap1->m[0] = "sifir";
   ap1->m[1] = ap2->m;
   ap1->m[2] = "iki";

   ap1->n[0] = 0;
   ap1->n[1] = ap2->n;
   ap1->n[2] = 2;

   return ap1;
}
```

...Örnek 8.4-5 devam

Çıktı

```
sifir : 0
bir : 1
iki : 2
```

8.5 Union Değişkeni

Bir *union değişkeni* (yada kısaca *union*), format olarak yapı değişkenine benzer fakat kullanım amacı farklıdır. Union değişkeni kullanılarak farklı zamanlarda, farklı tiplerde veriler aynı bellek alanında saklanabilir.

Bir yapı değişkeni, derleyicinin bildirilmiş olan elemanların tamamı için bellek alanı ayırmasını sağlar. Bir union ise sadece en büyük elemanı kadar bellek alanı kullanır.
Bir yapı değişkeninin elemanları birbirini izleyen ayrı bellek alanlarında bulunur. Bir union değişkeninin elemanları ise bellekte üst üste bulunur. Bir union değişkeni, en büyük elemanını saklayacak kadar bellek alanına sahip ve tüm elemanlarının offset'leri 0 olan yapı gibi düşünülebilir.

Union değişkeni çok sık kullanılmaz. Ancak bazı sistem seviyesinde programlama gerektiren uygulamalarda kullanılırlar. Bildirim şekli yapı değişkenine benzer, fakat bildirimde **struct** anahtar sözcüğü yerine **union** anahtar sözcüğü kullanılır:

```
union [etiket_ismi]
{
        union elemanı bildirimleri ...;
} union_degiskeni;
```

Union değişkenine bildirilen elemanların herhangi birinin tipindeki değerler atanabilir. Elemanlara erişim yapı değişkenlerinde olduğu gibi nokta operatörü kullanılarak gerçekleştirilir:

union_degiskeni.eleman_ismi

yada union değişkenine işaret eden bir adres değişkeni kullanılarak aşağıdaki ifade ile gerçekleştirilir:

union_adres_degiskeni->eleman_ismi

Örnek 8.5-1'de 3 elemanlı union değişkeni u bildirilir ve elemanlarına çeşitli değerler atanır. Daha sonra Listele fonksiyonu kullanılarak atanan değerler ekrana listelenir.

Bir union değişkeninin en son değer atanan elemanına erişilebilir. Farklı bir elemana erişim yapıldığında elde edilecek değer kullanılan derleyici ve donanıma göre

(*implementation-dependent*) değişir ve belirsizdir (Örnek 8.5-1'de yer alan son atama işleminde olduğu gibi). Bir union değişkeninin bellek alanı düzenlemesi derleyici tarafından yapılır fakat en son hangi elemanına atama yapıldığı programcı tarafından izlenmelidir. Dolayısıyla Örnek 8.5-1'de bu amaçla utip değişkeni kullanılmıştır.

Bir union değişkenine ilk değer atama, bildirim sırasında ilk elemanın tipinde oklu parantezlerle sınırlı sabit bir değer kullanılarak yapılabilir. Diğer elemanlara ilk değer atama yapılamaz. Dolayısıyla sadece ilk değer atama sırasında elemanların sırası söz konusudur. Atanan herhangi bir ilk değer ilk elemanın tipine göre yorumlanır. İlk değer atama yapılmayan global yada **static** union değişkeninin ilk değerleri 0, otomatik union'ların ise belirsizdir.

Yapı değişkenleri ile yapılan işlemlerin aynısı union değişkenlerine de uygulanır: atama, bir bütün olarak kopyalama, adresinin alınması ve elemanlarına erişim gibi.

Örnek 8.5-1 union1.c programı.

```
#include <stdio.h>

#define TAMSAYI 1
#define DOUBLE  2
#define DIZGI   3

union SAYILAR
{
  int    i;
  double d;
  char   *s;
};

void Listele( union SAYILAR, int );

int main(void)
{
  union SAYILAR u = { 55 };  /* u.i = 55 */
  int utip;

  utip = TAMSAYI;
  Listele( u, utip );

  u.d  = 27.91;
  utip = DOUBLE;
  Listele( u, utip );

  u.s  = "dizgi";
  utip = DIZGI;
  Listele( u, utip );

  u.s  = "dizgi";
  utip = 4;             /* hatali tip */
  Listele( u, utip );

  u.d  = 27.91;
```

...Örnek 8.5-1 devam

```
   utip = TAMSAYI;   /* hatali tip */
   Listele( u, utip );
   return 0;
}
void Listele( union SAYILAR u, int utip )
{
     switch( utip )
     {
          case  TAMSAYI:
                              printf("%d\n", u.i );
                              break;
          case  DOUBLE:
                              printf("%.2f\n", u.d );
                              break;
          case  DIZGI:
                              printf("%s\n", u.s );
                              break;
          default:
                              printf("bilinmeyen union tipi : %d\n", utip );
     } /* switch */
}
```

Çıktı

```
55
27.91
dizgi
bilinmeyen union tipi : 4
23593
```

Bir union değişkeni, yapı değişkeni yada dizi içinde bulunabilir. Bunun tersi de geçerlidir yani bir union değişkeni içinde bir yapı değişkeni yada dizi bulunabilir. Erişim yine yapı değişkenlerinde olduğu gibi gerçekleştirilir. Örnek 8.5-2'de yapı dizisi YDizi'nin elemanları bir union içeren yapı değişkenleridir. Programda ekrana YDizi[0] ve YDizi[1] yapılarının elemanı olan union değişkeni UnDeg'in tamsayı elemanının değeri, daha sonra da karakter dizisi olan elemanının ilk karakteri listelenir. ■

Örnek 8.5-2 union2.c programı.

```
#include <stdio.h>

struct
{
       char *s;
       int i;
       union
```

... Örnek 8.5-2 devam

```
        {
                int j;
                char *c;
        } UnDeg;

}  YDizi[ ] =  {  {  "dizgi1",12,     27      },    /* YDizi[0] */
                  {  "dizgi2",34,  {  42  }  },    /* YDizi[1] */
                  };

int main(void)
{
     printf("YDizi[0].UnDeg.j : %d\n", YDizi[0].UnDeg.j );
     printf("YDizi[1].UnDeg.j : %d\n", YDizi[1].UnDeg.j );

     YDizi[0].UnDeg.c = "abc";
     YDizi[1].UnDeg.c = "xyz";

     printf("*YDizi[0].UnDeg.c   : %c\n", *YDizi[0].UnDeg.c    );
     printf("YDizi[1].UnDeg.c[0] : %c\n",  YDizi[1].UnDeg.c[0] );

     return 0;
}
```

Çıktı

```
YDizi[0].UnDeg.j   :  27
YDizi[1].UnDeg.j   :  42
*YDizi[0].UnDeg.c   :  a
YDizi[1].UnDeg.c[0] :  x
```

8.6 Bit-Alanı

Bir yapı yada union değişkeninin elemanı olarak bildirilen *bit-alanı* (*bit-field*) bellekte bir tamsayı için ayrılan alanın belli bir kısmındaki bir yada daha fazla yan yana bit'ten oluşur.

Donanımla ilgili bazı özel uygulamalar bir **int** verinin bulunduğu bellek alanının parçalarına erişimi ve buralardaki bit değerleri üzerinde işlemler yapılmasını gerektirir. Bu durumda bit değerlerine erişim için, bit pozisyonlarına karşılık gelen maskeleri ve bit-operatörlerini kullanmak gerekmez. Büyük bir programda bazı kontrol parametrelerinin farklı zamanlarda aldığı değerleri saptamak için 1 ve 0 değerlerini saklamak için iki ayrı tamsayı değişken yerine, bu değerleri bit-büyüklüğünde erişilebilir alanlara bölünmüş tek bir tamsayı değişkende saklamak bellek alanı tasarrufu sağlar.

Bir bit-alanı (yada kısaca *alan*) yalnızca **int**, **signed int** yada **unsigned int** olarak bildirilebilir. **int** olarak bildirildiğinde kullanılan derleyiciye bağlı olarak **signed** yada **unsigned** olabilir. Dolayısıyla programların taşınabilir olması için **signed** yada **unsigned** olduğu belirtilmelidir.

Bit alanı tanımı:

```
struct etiket
{
        /* bit-alanı bildirimleri */
        unsigned alan_ismi : genislik;
        ...
} degisken_ismi;
```

Her bit-alanı bir alan ismi (*alan_ismi*), ":" ve alanın bit sayısını belirten genislik (*genislik*) ile bildirilir. Bit-alanlarına adres operatörü (&) uygulanamaz. Bit-alanlarının bellekteki dizilişi (soldan sağa yada sağdan sola doğru) kullanılan derleyici ve donanıma göre değişir ve bu nedenle belli bir bellek düzenini esas alarak geliştirilen programlar taşınabilir değildir. Bit-alanları tamsayı değerlerdir ve aritmetik işlemlerde kullanılabilir. Bit-alanı bir tamsayı için ayrılan bellek sınırını geçemez bu nedenle bir **int** alandan daha geniş bit-alanı bildirilemez.

Örnek 8.6-1'de, 3 adet 1-bitlik alandan oluşan bit-alanı değişkeni BitDeg bildirilir. Daha sonra bildirilen bit-alanlarına sırasıyla 1, 0 ve 1 değerleri atanır. Atanan değerler ? operatörü ile oluşturulan ifadeler kullanılarak ekrana listelenir.■

Örnek 8.6-1 bitalan.c programı.

```
#include <stdio.h>

struct BitSablon
{
  unsigned alan1 : 1, alan2 : 1, alan3 : 1;
};

struct BitSablon BitDeg;

int main(void)
{
  BitDeg.alan1 = 1;
  BitDeg.alan2 = 0;
  BitDeg.alan3 = 1;

  printf("BitDeg.alan1 : %c\n", BitDeg.alan1 ? '1' : '0' );
  printf("BitDeg.alan2 : %c\n", BitDeg.alan1 ? '1' : '0' );
  printf("BitDeg.alan3 : %c\n", BitDeg.alan1 ? '1' : '0' );

  return 0;
}
```

Çıktı

```
BitDeg.alan1 : 1
BitDeg.alan2 : 0
BitDeg.alan3 : 1
```

BÖLÜM 9: **Giriş/Çıkış İşlemleri**

Giriş/çıkış işlemleri (I/O, **I**nput/**O**utput) C dilinin dışında tutulmuştur. Bu nedenle, giriş/çıkış işlemleri kullanılan C derleyicisiyle birlikte sağlanan kütüphane fonksiyonları ile gerçekleştirilir. Diğer programlama dillerinin aksine, I/O işlemlerinin ayrıntıları dilin dışında tutularak standart C kütüphanesine yerleştirilmiştir ve böylece C dilinin taşınabilir bir dil olma özelliği korunmuştur. Çünkü farklı bilgisayar sistemleri, I/O işlemleri için farklı teknikler kullanır. Derleyiciler, ANSI standart I/O kütüphanesinden başka, birkaç I/O fonksiyon grubu sağlar.

Bir giriş/çıkış işlemi, herhangi bir disk dosyasından okumak ve bu dosyaya yazmak, klavyeden girilen bilgileri okumak, çıktıları ekrana göndermek yada çevre birimlerine çeşitli komutlar göndermek olabilir. Klavye ve ekran giriş/çıkış işlemleri (console I/O) önceki bölümlerde anlatıldı. Bu bölümde, disk dosyalarına erişim (disk I/O) ve ilgili teknikler anlatılacaktır. ■

9.1 Standart Disk Giriş/Çıkış İşlemleri

Disk dosyaları floppy disket, hard disk yada manyetik teyp gibi kalıcı ortamlarda saklanan programlar ve verilerdir. Disk dosyalarına erişim için iki tip I/O fonksiyon grubu bulunur:

- **Akım I/O** (buffered stream I/O) fonksiyonları,
- **Düşük-seviyeli I/O** (unbuffered low-level I/O yada system I/O) fonksiyonları.

Akım fonksiyonları ANSI standardında (standart fonksiyonlar) tanımlandığı için, bu fonksiyonlar kullanılarak yazılan programlar bir donanımdan diğerine taşınabilir. Akım fonksiyonları, disk erişimi sırasında transfer edilen verilerin geçici olarak saklanması için işletim sistemi tarafından bellekte oluşturulan bir tampon alanı (*buffer*) kullanırlar.

Diskten bilgiler byte grupları halinde okunur ve tampon alana yazılır. Diskten bilgi okuyan I/O fonksiyonu, aslında daha önceden diskten byte grubu halinde okunarak tampon alana aktarılmış olan bilgileri okur. Aynı işlem yazma sırasında da fakat tampon alandan diske doğru gerçekleşir. Böylece, tampon alan kullanılarak farklı hızlarda çalışan ve verileri farklı şekillerde yöneten donanım birimlerinin birlikte çalışması sağlanır.

Pek çok akım I/O fonksiyonu, veriler disk dosyasından okunurken yada dosyaya yazılırken çeşitli format çevrimleri yapar (binary değerlerin, ASCII değerlere çevrilmesi). Yine standart kütüphanede yer alan akım I/O fonksiyonları fread ve fwrite, binary verilerin format değiştirme yapılmadan okunması ve yazılması için kullanılır.

Akım fonksiyonları, erişim sırasında akımı kontrol etmek için gerekli olan bilgileri bir veri yapısında tutar. FILE olarak adlandırılan bu veri yapısı, başlık dosyası stdio.h'de tanımlanmıştır. Bir disk dosyası, daha sonra anlatılacak olan akım I/O fonksiyonu fopen ile açıldığında, bu fonksiyon tarafından çağıran bloğa, disk dosyasının erişim bilgilerini (tampon alanın yeri, tampon alandaki karakter pozisyonu, dosyadan okuma yada dosyaya yazma işlemlerinden hangisinin yapıldığı, I/O sırasında hata yada okuma sırasında dosya sonu durumunun oluşup oluşmadığı bilgisi gibi) taşıyan FILE yapısına işaret eden bir adres değeri döndürülür. Programda izleyen tüm akım I/O işlemleri, *akım* (*stream*) olarak adlandırılan bu adres değeri aracılığı ile erişilen bilgiler kullanılarak gerçekleştirilir.

Buna göre bir *akımın açılması*, bir disk dosyası yada bir çevre birimi ile (klavye, ekran, modem yada yazıcı gibi) veri iletişim hattı oluşturulması anlamına gelir. Bir *akıma yazmak*, bir disk dosyasına, ekrana, modem'e, yazıcıya yada bir başka çevre birimine bilgilerin gönderilmesini; bir *akımdan okumak* ise bir disk dosyasından, klavyeden, modemden yada bir başka çevre biriminden bilgilerin alınmasını ifade eder. Sözkonusu *akım kapatıldığında* ise, bu veri iletişim hattı kesilmiş olur.

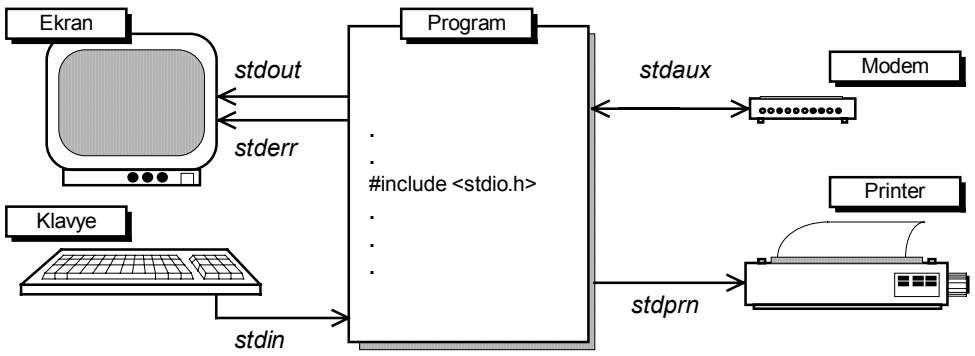
Bir C programı çalışmaya başladığı anda, bazı *standart I/O akımları* otomatik olarak açılır. Standart akımlar, C programlarının ortam ile iletişimini sağlar ve disk dosyaları gibi açılıp, kapatılmaz. DOS ve UNIX işletim sistemlerinin her ikisinde de bulunan 3 akım şunlardır: stdin (standart giriş), stdout (standart çıkış), ve stderr (standart hata). Diğer iki akım sadece DOS işletim sisteminde açılır: stdaux (standart yardımcı) ve stdprn (standart yazıcı).

Buna göre *stdin akımından okumak* ifadesi ile, stdin'in okunan dosyanın FILE yapısının adresini taşıdığı ifade edilir.
stdin, yukarıda bahsedildiği gibi bir C programı çalışmaya başladığında otomatik olarak açılan ve klavye ile ilgili olan akımdır. Şekil 9.1-1'de, DOS altında açılan 5 standart I/O akımı ve bu akımlarla ilgili donanım birimleri görülmektedir. Örnek programlarda, standart giriş ve standart çıkış akımları kullanıldı. Ekrana bilgi yazmak için sık sık kullanılan printf fonksiyonu, stdout akımına yazar. Fakat printf fonksiyonu daha genel amaçlı bir fonksiyon olan ve herhangi bir akıma yazabilen fprintf'in özel bir şeklidir. Aşağıdaki her iki deyim de bilgisayar ekranına deneme dizgisini yazar:

```
printf( "deneme\n" );
fprintf( stdout, "deneme\n" );
```

Klavyeden bilgi girişi için kullanılan scanf fonksiyonu ise stdin akımından okur. Bu fonksiyonun, herhangi bir akımdan okuyabilen karşılığı ise fscanf fonksiyonudur. Bu fonksiyonlar, izleyen paragraflarda anlatılacaktır.

Şekil 9.1-1 Standart giriş/çıkış akımları.



Düşük-seviyeli I/O fonksiyonları, tampon alan kullanmazlar ve hiçbir format çevirme işlemi yapmazlar. Bu işlemlerin, programda yapılması gerekir. Bu nedenle daha sonra anlatılacak olan text tipi dosyalara erişim için uygun değillerdir. Bu fonksiyonlar kullanılarak yazılan programlar taşınabilir değildir. Fakat donanım üzerinde kontrol artar ve direk erişim sağladıkları için daha hızlıdırlar.

Bu fonksiyonlar, akım I/O fonksiyonları ile birlikte kullanılamaz. Kullanıldığında dosya erişimi sırasında oluşabilecek hatalar veri kaybına sebep olabilir. Bu bölümde, akım I/O fonksiyonları anlatılacaktır.

Bir disk dosyasının oluşturulması, dosyaya yazma ve dosyanın kapatılması

Örnek 9.1-1'deki textyaz.c programı, A: sürücüsünde bulunan diskte test.asc isimli bir *text dosyası* oluşturur ve bu dosyaya deneme dizgisini yazar. Standart giriş/çıkış fonksiyonları çağrılacağı zaman gerekli bazı tip, makro rutini tanımları ve fonksiyon prototiplerini içeren standart giriş/çıkış başlık dosyası stdio.h, önceki örneklerde olduğu gibi aşağıdaki deyim kullanılarak programa dahil edilmelidir:

```
#include <stdio.h>
```

Disk dosyaları üzerinde işlemler aşağıdaki şekilde gerçekleştirilir:

- İlk olarak bir *dosya göstergesi* bildirilir; FILE * tipinde bir adres değişkeni.

  Dosya göstergesi bildiriminin genel şekli aşağıda verilmiştir:

  FILE **dosya_göstergesi*;

  Örnek programlarda çoğunlukla dosya göstergesi ismi olarak fp (**f**ile **p**ointer) kullanılmıştır. Fakat bunun yerine geçerli herhangi bir isim kullanılabilir.

NOT:
FILE adres değeri ve bu değerin atandığı FILE adres değişkeninden bahsedilirken "akım" yada "dosya göstergesi" ifadeleri kullanılabilir.

Program çalışmaya başlamadan önce otomatik olarak açılan 5 akım da (stdin, stdout, stderr, stdaux (DOS), stdprn (DOS)) birer dosya göstergesidir. Fakat bunlara adres değeri atanamaz, çünkü her biri stdio.h'de tanımlanmış ve ayrı bir FILE * tipi sabit adres değerine karşılık gelen sembolik isimdir (dosya gösterge sabitleri).

stdin (standart giriş akımı) normal olarak klavyeye (standart giriş dosyası) bağlıdır; stdout ve stderr ise ekrana (standart çıkış dosyası) bağlıdır. stdin ve stdout yönlendirilebilir (> ve | kullanılarak), fakat stderr daima ekrana bağlıdır ve tampon alan kullanmaz.

Bu bölümde disk dosyaları ile ilgili işlemler anlatıldığı için yerine göre karışıklığı önlemek amacıyla, "akıma yazmak" ifadesi yerine, "bir disk dosyasına yada standart çıkış'a yazmak"; "bir akımdan okumak" ifadesi yerine "bir disk dosyasından yada standart giriş'ten okumak" ifadeleri kullanılabilir.
Standart akımlar açılıp kapatılmaz (bazı özel durumlar dışında). Dolayısıyla "bir akımın açılması yada kapatılması", "bir disk dosyasının açılıp kapatılması"'nı ifade eder.

- fopen fonksiyonu kullanılarak disk dosyası açılır. fopen fonksiyonu dosya erişimini sağlayan FILE tipi veri adresi döndürür. FILE veri tipi, stdio.h başlık dosyasında _iobuf şablonuna sahip ve dosya erişimi ile ilgili daha önceki paragraflarda bahsedilen bilgileri içeren bir yapı tipi (**struct** veri tipi) olarak tanımlanmıştır. Bu adres değeri, dosya göstergesi değişkenine atanır.

- Disk dosyası açıldıktan sonra, standart kütüphanede yer alan çeşitli giriş/çıkış fonksiyonları ile erişim gerçekleştirilir. Bu fonksiyonların isimleri pek çok ekran ve klavye fonksiyonu ismine benzer fakat "f" harfi ile başlar (fprintf, fscanf, fputs, fputc, fgets, fgetc gibi) ve ayrıca parametre olarak dosya göstergesi alırlar.

- İşlemler tamamlandığında disk dosyası, fclose fonksiyonu kullanılarak kapatılır.

Örnek 9.1-1 textyaz.c programı.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  FILE *fp;

  if (  (fp = fopen( "a:\\test.asc", "w" ))  == NULL )
  {
    puts("test.asc dosyasi acilamadi");
    exit(1);
  }

  fputs( "deneme", fp );
  fputc( '\n', fp );
  fclose( fp );

  return 0;
}
```

Örnek 9.1-1'de ilk olarak,

```
FILE *fp;
```

deyimi ile dosya göstergesi fp bildirilir. Bunu izleyen if deyiminin test ifadesinde, fopen çağrısı yer alır. fopen fonksiyonuna iki argüman aktarılır. İlk argüman açılacak dosyanın diskteki ismini (*dosya_ismi*), diğeri ise dosyanın hangi amaçla açılacağını yani *erişim tipi*ni (*mod*) belirtir:

```
FILE *fopen( char *dosya_ismi, char *mod );     /* stdio.h */
```

Bu argümanlar, dizgi sabiti yada dizgi adresi olabilir. Tablo 9.1-1'de dosya işlemlerinde kullanılan erişim tipleri ve işlevleri listelenmiştir. Dosyanın disk ismi ("a:\\test.asc") yalnızca fopen fonksiyonu tarafından dosya açılırken kullanılır. Program içinde diğer giriş/çıkış fonksiyonları kullanılarak dosyaya yapılan tüm erişimler, dosyanın disk ismi yerine dosya göstergesi fp kullanılarak gerçekleştirilir.

Aynı anda açılacak olan dosyaların her biri için, ayrı bir dosya göstergesi bildirilmelidir. Bir defada açılabilecek maksimum dosya sayısı (standart başlık dosyası stdio.h'de FOPEN_MAX makrosu ile tanımlanır ve bu sayıya standart dosyalar da dahildir) sınırlıdır. test.asc dosyası, oluşturulurken erişim tipi olarak "w" kullanıldığı için tabloda da görüldüğü gibi yazma amacıyla açılmıştır.

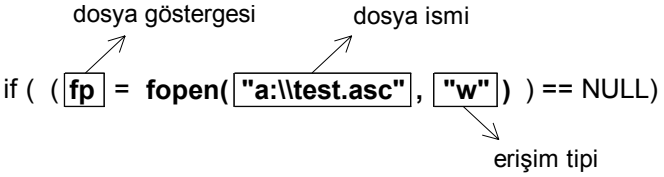
Tablo 9.1-1 Erişim tipleri ve işlevleri.

r	Diskte mevcut bir dosyayı **okumak** için açmak,
r+	Diskte mevcut bir dosyayı **okumak/yazmak** için açmak.
w	Bir dosya oluşturmak ve **yazmak** için açmak, Eğer aynı isimde bir dosya mevcut ise içindekiler silinir ve yeni veriler aynı dosyaya yazılır.
w+	Bir dosya oluşturmak ve **okumak/yazmak** için açmak, Eğer aynı isimde bir dosya mevcut ise içindekiler silinir ve yeni veriler aynı dosyaya yazılır.
a	Diskte mevcut bir dosyayı verileri sonuna **eklemek** amacıyla açmak, Eğer dosya diskte mevcut değilse oluşturulur.
a+	Diskte mevcut bir dosyayı verileri sonuna **eklemek**/**okumak** amacıyla açmak, Eğer dosya diskte mevcut değilse oluşturulur.

NOT:
***w** ve **w+** erişim tipleri kullanılırken, diskte aynı isimde bir dosya bulunuyor ise içindeki mevcut bilgiler kaybedileceği için dikkatli olmak gerekir.*

---

fopen çağrısından dönen FILE tipi veri adresi, fp adres değişkenine atanır:



NOT:
Dosya ismi dizgisinde peşpeşe iki ters kesme (\\) kullanılmıştır. Bunun nedeni, \t'nin tab boşluğunu ifade etmesidir. Çift ters kesme kullanılarak, t karakterinden

önceki ters kesmenin etkisi kaldırılır. Böylece derleyici \ ve t karakterlerinin kombinasyonunu, tab escape dizini olarak almaz. Burada ayrıca dosyanın erişim tipini bildiren argümanın, üstten tek tırnaklarla sınırlı bir karakterden oluşan ve bu karakterin ASCII kodunu döndüren karakter sabiti ('w') değilde, çift tırnaklarla sınırlı ve bellek adresi döndüren bir dizgi sabiti ("w") olduğuna dikkat edilmelidir.

Dosya ismi ("test.asc"), DOS işletim sisteminde en fazla 8 karakterden oluşur ve seçime bağlı olarak nokta ile ayrılmış en fazla 3 karakterden oluşan uzantısı olabilir. Genel olarak dosyanın ismi, içerdiği verilerin türünü; uzantısı ise uygulamayı ifade edecek şekilde belirlenir. Bu örnekte dosya ismi, deneme amacıyla ASCII karakterler içeren bir dosya oluşturulduğunu ifade etmek ve \t (tab) kullanımını örneklemek için test.asc olarak belirlenmiştir. Dosya isminde izin verilen maksimum karakter sayısı, kullanılabilecek karakterler ve dosyanın bulunduğu yeri gösteren dizin (path) bilgisinin yazılışı işletim sistemine göre değişebilir.

if deyiminde, fp'ye gizli atama ile atanan adres değerinin NULL adres değeri olup olmadığı test edilir:

```
                    gizli atama
              ┌──────────────────────┐
if ( (fp = fopen("a:\\test.asc", "w")) == NULL)
```

Çünkü herhangi bir sebepten dolayı dosya açılamıyor ise, fopen fonksiyonu FILE tipi adres değeri yerine NULL adres değeri (boş adres değeri 0) döndürür ve if deyiminin doğru kısmı çalışarak programdan çıkılır. Örnek programda, dosya açıldıktan sonra fputs, fputc ve fclose fonksiyonları çağrılır. puts fonksiyonu, argüman olarak verilen *dizgi*yi (dizgi sabiti yada dizgi adresi olarak), sonuna bir yeni-satır karakteri (NL, newline, \n) ekleyerek standart çıkış akımı, stdout'a (yani bilgisayar ekranına) yazar:

```
int puts( char *dizgi );    /* stdio.h */
```

fputs fonksiyonu da aynı şekilde çalışır. Fakat ikinci argüman olarak bir dosya göstergesi alır. Dolayısıyla, dizgiyi bir disk dosyasına yada standart çıkışa yazabilir. Ayrıca dizgi sonuna NL karakteri eklemez:

```
int fputs( char *dizgi, FILE *dosya_gostergesi );   /* stdio.h */
```

Her iki fonksiyon da, dizgi sonunu belirleyen null (\0) karakterini çıktıya yazmaz. Bu fonksiyonlar hatasız olarak çalıştığında negatif olmayan bir değer, hata oluşması durumunda ise EOF döndürür. Aşağıdaki deyimlerin her ikisi de ekrana deneme dizgisini yazar:

```
puts( "deneme" );
fputs( "deneme\n", stdout );
```

Programda bir NL karakterini dosyaya yazmak için fputc fonksiyonu kullanıldı. Bu fonksiyon karakter *c*'yi (**unsigned char** olarak) çıkış akımına yazar:

```
int fputc( int c, FILE *dosya_gostergesi );   /* stdio.h */
```

İkinci argüman *dosya_gostergesi*, daha önce de bahsedildiği gibi yazma amacıyla açılmış bir disk dosyasına ait dosya göstergesi yada stdout, stderr gibi bir standart çıkış akımıdır. Fonksiyon hatasız çalışır ise, akıma yazılan karakteri (**int**)(**unsigned char**) olarak döndürür. Hata durumunda ise EOF döndürür. fputc fonksiyonu, putc makrosu ile aynıdır:

```
int putc( int c, FILE *dosya_gostergesi );   /* stdio.h */
```

Buna göre aşağıdaki deyimler aynı işi yapar:

```
putc( 'A', stdout );
fputc( 'A', stdout );
```

Standart çıkış akımına yazan ve stdio.h'de tanımlanan putchar makrosu, putc makrosu ile aynıdır:

```
int putchar( int c );   /* stdio.h */
```

Buna göre, putchar(c) ve putc(c, stdout) aynıdır.

fopen ile yazma amacıyla açılarak, deneme dizgisi ve NL karakteri yazılan disk dosyası, fclose fonksiyonu ile kapatılarak programdan çıkılır. fclose fonksiyonu, hatasız olarak dosya kapatma işlemini gerçekleştirir ise 0 değerini, hata oluşur ise EOF döndürür:

```
int fclose( FILE *dosya_gostergesi );     /* stdio.h */
```

fopen ile açılan dosyalar, programdan çıkılırken otomatik olarak kapatılır. Fakat fclose kullanılarak dosyaların kapatılması daha uygundur. fclose fonksiyonu *dosya gostergesi* ile disk dosyası arasındaki bağı koparır. Bir programda, bir defada açılabilecek dosya sayısı sınırlıdır (en fazla FOPEN_MAX kadar dosya açılabilir. Bu sabit stdio.h'de tanımlanmıştır ve genel olarak 20'dir). Bu nedenle, çok fazla sayıda dosya ile çalışan bir programda işlemleri tamamlanan bir dosyanın kapatılması önem kazanabilir.

Bölüm girişinde anlatıldığı gibi fputs ve fputc gibi standart disk I/O fonksiyonları, dizgileri ve karakterleri bellekteki bir tampon alana yazarlar. Bu tampon alan tamamen dolduktan sonra, buradaki bilgiler disk dosyasına aktarılır. Bu işlem *flushing* olarak adlandırılır.

fclose fonksiyonu, bu alanda bulunan yazılmamış bilgilerin dosyaya aktarılmasını sağlar ve veri kaybı önlenir. Ayrıca serbest bırakılan tampon alan başka bir işlemde kullanılabilir. fflush ve fclose fonksiyonları kullanılarak bu tampon alanın tam olarak dolmadan önce boşaltılması sağlanabilir:

```
int fflush( FILE *dosya_gostergesi );     /* stdio.h */
```

fflush fonksiyonu, tampon alanda bulunan tüm çıktı bilgisini *dosya_gostergesi* ile belirtilen akıma yazar ve hata oluşmaz ise 0, aksi taktirde EOF döndürür. Eğer argüman olarak boş adres değişkeni (*null pointer*) verilir ise bu işlemi çıktı amacıyla açılmış olan tüm dosyalar için yapar.

Dosya fclose ile kapatılmamış olsa da, bir C programından normal çıkış sırasında otomatik olarak dosya kapatma işlemi yapılır. Fakat programdan normal olmayan bir şekilde (örneğin abort fonksiyonu çağrılarak yada programın çalışması kesilerek) dosya kapatılmadan çıkılır ise, flushing işlemi yapılarak tampon alan boşaltılmamış olacağı için, bu alanda kalan bilgiler kaybolacaktır.

Bir disk dosyasından okumak

textoku.c programında, test.asc dosyası okuma amacıyla (erişim tipi: r) açılır. **while** döngüsünün test ifadesinde, fgetc fonksiyonu tarafından döndürülen değer tamsayı c değişkenine atanır ve bu değerin dosya sonunu belirten EOF (end-of-file) olup olmadığı kontrol edilir:

```
int fgetc( FILE *dosya_gostergesi );    /* stdio.h */
```

fgetc fonksiyonu, her çağrıda *dosya_gostergesi* ile tanımlanan dosyanın o andaki karakter pozisyonundan bir karakter okur ve pozisyon göstergesini bir sonraki karaktere işaret edecek şekilde arttırır. Hatasız çalıştığında (**int**)(**unsigned char**) olarak okuduğu karakteri, hata durumunda ise EOF döndürür. Fakat fgetc fonksiyonu, dosya sonuna ulaşıldığında da EOF döndürür. Daha sonra anlatılacak olan ferror ve feof makroları kullanılarak bu iki durum birbirinden ayırt edilebilir.

Örnek 9.1-2 textoku.c programı.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int c;
  FILE *fp;

  if ( (fp = fopen( "a:\\test.asc", "r" )) == NULL )
  {
```

... Örnek 9.1-2 devam

```
    puts("test.asc dosyasi acilamadi");
    exit(1);
  }
  while (  (c = fgetc( fp )) != EOF )
    fputc( c, stdout );
  fclose( fp );
  return 0;
}
```

Çıktı

```
deneme
```

Standart C kütüphanesinde yer alan getc makrosu da, fgetc fonksiyonuna benzer. Dosyadan okuduğu karakteri döndürür. Hata oluştuğunda yada dosya sonuna ulaşıldığında ise yine EOF döndürür.

```
int getc( FILE *dosya_gostergesi );     /* stdio.h */
```

Ayrıca, stdin akımından okuyan getchar makrosu da, getc(stdin) ile aynıdır:

```
int getchar( void );     /* stdio.h */
```

Text ve Binary Modlar

Bir *text dosyası* (metin dosyası yada ASCII dosya) okunabilir ASCII karakterler içerir (alfanumerik karakterler, noktalama işaretleri ve bazı kontrol karakterleri) ve bir yada daha fazla satırdan oluşur. Text dosyasının içeriği, işletim sisteminde bulunan herhangi bir dosya listeleme komutu ile satırlar halinde ekrana listelenebilir (örneğin DOS işletim sisteminde TYPE komutu, UNIX işletim sisteminde cat komutu kullanılarak). Dosya içinde bulunan byte serilerinin (karakterlerin) satırlar halinde düzenlenmesini *satır-sonu karakteri* (EOL, *end-of-line*) sağlar. Satır-sonu karakteri, işletim sistemine göre farklılık gösterebilir.

Bir C programında satır sonunu belirtmek için kullanılan *newline* karakteri (Sembolik olarak NL, karakter sabiti olarak '\n', ASCII kod olarak 10 yada 0x0A), UNIX işletim sisteminde bir text dosyasında yine satır sonunu işaretler. Fakat DOS işletim sisteminde durum farklıdır ve satır sonu *carriage-return* (CR, karakter sabiti olarak '\r', ASCII 13 yada 0x0D) ve *line-feed* (LF, ASCII 10 yada 0x0A) karakter kombinasyonu ile işaretlenir.
Bu nedenle, DOS işletim sisteminde programda (bellekte) bulunan NL karakteri, disk dosyasına CR-LF karakter çiftine çevrilerek yazılır. Yine aynı şekilde, disk dosyasından bu karakter çifti okunduğunda, tek bir NL karakterine çevrilerek belleğe alınır. DOS

ortamında kullanılan derleyiciler tarafından sağlanan standart kütüphane fonksiyonları bu çevirme işlemlerini otomatik olarak yapar. Fakat istendiğinde diskten okuma yada diske yazma sırasında bu çevirmelerin yapılmaması sağlanabilir.

DOS derleyicileri kullanılırken bu işlemleri gerçekleştirmek için, dosya iki ayrı modda açılır: *text mod* ve *binary mod*. Buna göre, DOS işletim sisteminde bir dosyadan okuma yada dosyaya yazma işlemi sırasında transfer edilen bu karakterlerin nasıl yorumlanacağı dosyanın hangi modda açıldığına bağlıdır. Bu modlar, *translation mode* (çevirme modu) olarakta adlandırır. Her iki mod da, standart C kütüphanesi tarafından desteklenir. Fakat sözkonusu çevirme işlemleri C standardında tanımlanmamıştır ve tamamen belli bir işletim ortamında çalışan derleyici ile birlikte verilen standart C kütüphanesi tarafından gerçekleştirilir. Böylece bir DOS dosyası text modunda açıldığında çevirme işlemleri otomatik olarak yapılır ve programda hiç bir etki olmaz. Bazı işletim sistemlerinde, örneğin UNIX'te text mod ve binary mod ayrımı olmadığı göz önünde bulundurulacak olursa, bu yaklaşım ile C programındaki text dosyası modeli UNIX yada DOS altında çalışırken korunmuş olur. Çeşitli işletim ortamlarında program geliştirirken, bu tip farklılıklar dikkate alınmalıdır.

Dosyanın hangi modda açılacağı, fopen fonksiyonunun ikinci argümanı olan *mod* dizgisine, erişim tipinden (yada erişim modu) sonra eklenen ikinci bir harf ile belirtilir. *mod* dizgisinde sadece erişim tipi yer aldığı zaman (Tablo 9.1-1), dosyanın text modunda açılacağı kabul edilir (*default mode*). Buna göre önceki örneklerde oluşturulan test.asc dosyası text modunda oluşturulmuş bir dosyadır ve biraz sonra da anlatılacağı gibi okuma-yazma işlemleri sırasında bazı çevirmeler gerçekleşir. *mod* dizgisine erişim tipinden sonra yerleştirilen b harfi (örneğin "rb", "w+b" yada "wb+" gibi), dosyanın binary modunda açılacağını belirtir. Dosya binary modunda açıldığında okuma ve yazma işlemleri sırasında hiç bir çevirme yapılmaz.

NOT:
Text mod, DOS derleyicilerinde *translated mode* olarakta adlandırılır. Bazı DOS derleyicileri ile çalışırken, text mod açık olarak ifade edilmek istendiğinde *mod* dizgisinde t harfi kullanılır (örneğin "rt", "wt" gibi). Bunun bir ANSI standart modu olmadığına dikkat edilmelidir. Bu harf kullanılmadığı zaman yine text mod kabul edilir.

Aşağıdaki program, yukarıda bahsedilen NL ve CR karakterlerinin kullanımını örnekler:

```
/*  CRNL.C  */
#include <stdio.h>

int main(void)
{
  int i;

  for ( i = 0; i < 5000; i++ )
```

... crnl.c devam

```
 {
     printf("i : %04d", i );
     if ( i == 2000 )
       putchar('\n');
     else
       putchar('\r');
 }
 printf("\nNL : 0x%02X, CR : 0x%02X", '\n', '\r' );
 printf("\nNL : %d, CR : %d", '\n', '\r' );

 return 0;
}
```

Çıktı:
```
i : 2000
i : 4999
NL : 0x0A, CR : 0x0D
NL : 10, CR : 13
```

Çıktıda da görüldüğü gibi 2000'e kadar olan sayılar aynı satıra, bu sayıdan sonrası ise bir alt satıra yazılır. Programda son olarak NL ve CR karakterlerinin ASCII karakter kodları onaltılık ve onluk sayı sistemlerinde ekrana listelenir. Programın çıktısı, standart çıkış bir disk dosyasına yönlendirilip, oluşturulan bu dosya işletim sisteminde bulunan bir listeleme komutu ile ekrana listelendiğinde, çıktı yine aynı şekilde görülür.

mod dizgisi sadece erişim tipi bilgisini içerdiği için, textyaz.c ve textoku.c programlarında okuma ve yazma işlemleri text modunda gerçekleşmiştir. textyaz.c programı tarafından oluşturulan test.asc dosyası DOS işletim sisteminin DIR komutu ile listelendiğinde, bulunduğu fihristte 8 byte büyüklüğünde bir dosya olarak görülür. Fakat dosyaya, 6 karakterden oluşan deneme dizgisi ve bir NL karakteri olmak üzere toplam 7 karakter yazıldı. Bunun nedeni text modunda yapılan çevirme işlemidir.
Örnek 9.1-3'de, text modunda oluşturulan test.asc dosyasında bulunan karakterler ve ASCII kodları, dosya önce text modunda açılarak ve daha sonrada binary modda açılarak listelenir:

Örnek 9.1-3 mod1.c programı.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(void)
{
  int c;
  FILE *fp;
```

... Örnek 9.1-3 devam

```
  puts( "test.asc: r - text modunda okuma." );
  if ( !(fp = fopen( "a:\\test.asc", "r" )) )
  {
    puts("test.asc dosyasi acilamadi");
    exit(1);
  }
  while ( (c = fgetc( fp )) != EOF )
        printf( "%c 0x%02X, %d\n", isprint(c) ? c : ' ', c, c );
  printf( "EOF (end-of-file) : %d\n", c );
  fclose( fp );

  puts( "test.asc: rb - binary modunda okuma." );
  if ( !(fp = fopen( "a:\\test.asc", "rb" )) )
  {
    puts("test.asc dosyasi acilamadi");
    exit(2);
  }
  while ( (c = fgetc( fp )) != EOF )
        printf( "%c 0x%02X, %d\n", isprint(c) ? c : ' ', c, c );
  printf( "EOF (end-of-file) : %d\n", c );
  fclose( fp );
  return 0;
}
```

Çıktı

test.asc: r - text modunda okum	**test.asc: rb - binary modunda okuma.**
d 0x64, 100	**d** 0x64, 100
e 0x65, 101	**e** 0x65, 101
n 0x6E, 110	**n** 0x6E, 110
e 0x65, 101	**e** 0x65, 101
m 0x6D, 109	**m** 0x6D, 109
e 0x65, 101	**e** 0x65, 101
0x0A, 10 } **NL**	**0x0D, 13** } **CR-LF**
EOF (end-of-file) : -1	**0x0A, 10** } **CR-LF**
	EOF (end-of-file) : -1

Okuma kolaylığı açısından çıktılar yanyana listelenmiştir !

---

mod1.c programının çıktısı incelendiğinde, text modunda okuma yapıldığında dizgi sonunda sadece bir karakter (toplam 7 karakter), binary modunda ise iki karakter (toplam 8 karakter) görülür. Çünkü, dosya text modunda yazma amacıyla açıldığında NL karakteri yazılmadan önce CR-LF karakter çiftine çevrilir. Aynı dosya yine text modunda okuma amacıyla açıldığında, okunan CR-LF karakter çifti tekrar tek bir NL karakterine çevrilir. Dolayısıyla, test.asc dosyası text modunda oluşturulduğu için diskte fazladan bir karakter içerir. Yine text modunda okunduğunda bu iki karakter tek bir karaktere çevrildiği için çıktıda sadece NL karakteri görülür. Fakat, binary modunda

böyle bir çevirme işlemi yapılmaz. Bu nedenle, text modunda yazılan test.asc dosyası binary modunda okunduğu zaman çıktıda CR-LF kombinasyonu görülür. Dosya binary modunda yazılmış olsaydı, diskte 7 byte büyüklüğünde görülecek ve her iki modda da, son karakter olarak sadece 0x0A görülecekti. Genel olarak dosya hangi modda yazılmış ise o modda okunur. Burada, içeriğini çevirme yapılmadan okumak için text modunda oluşturulan bir dosya binary modunda açılmıştır. UNIX işletim sisteminde bu çevirmeler yapılmadığı için ve mod ayrımı bulunmadığı için, dosya diskte 7 byte olarak bulunur (ls komutu ile listelendiğinde).

NL ve LF karakterleri aynıdır (0x0A). Fakat örneklerde de görüldüğü gibi bu karakter DOS işletim sisteminde bellekte ve diskte farklı yorumlanırlar. Bu konu ile ilgili olarak verilen son örnek olan Örnek 9.1-4'de, NL içeren NLDizgi dizgisi, ilk olarak text modunda açılan text dosyasına daha sonrada binary modunda açılan binary1 dosyasına yazılır. CR ve NL karakterleri içeren CR_NLDizgi dizgisi ise binary modunda açılan binary2 dosyasına yazılır. Program çalıştırıldıktan sonra oluşturulan bu üç dosyanın içeriği izleyen bölümlerde anlatılacak olan dump programı kullanılarak hiç bir çevirme yapılmaksızın onaltılık karakter kodları olarak listelenir.
Ayrıca her bir dump çıktısından sonra, DOS işletim sisteminde bulunan TYPE komutu kullanılarak dosyalar listelenir. Çıktıda açık olarak görüldüğü gibi, DOS işletim sisteminde bir text satırının CR-LF kombinasyonu ile sonlanması gerekir. UNIX'te ise C programlarında olduğu gibi tek bir NL ile sonlanır.

Yukarıdaki paragraflarda, işletim ortamına göre farklılık gösteren satır-sonu (EOL) karakteri anlatıldı. İşletim ortamına göre farklılık gösteren bir diğer konu da, text modunda dosya sonunun saptanmasıdır.

DOS işletim sisteminin eski sürümlerinde dosya sonu pozisyonunu saptamak için dosya sonu işareti olarak ^Z (Ctrl-Z, ASCII kod olarak 26 yada 0x1A) kullanıldı. Fakat bu teknik terkedilmiştir ve dosya sonu, dosyanın byte sayısı bilgisi kullanılarak saptanır. Böylece dosyada bulunan son karakter sonrasını okuma girişimi bir dosya-sonu (*end-of-file)* durumu oluşturur. Fakat dosya sonunu belirlemek için dosya büyüklüğü bilgisi kullanılıyor olsa da, text modunda okuma yapılırken dosya içinde ^Z karakterine rastlanırsa yine dosya-sonu durumu oluşur ve işlem durdurulur (Örnek 9.1-7).

Binary modunda ^Z karakterinin özel bir anlamı yoktur ve dosyada bulunan tüm karakterler okunduktan sonra dosya-sonu durumu oluşur. UNIX işletim sisteminde de (ve C dilinde) yine özel bir dosya sonu işareti yoktur. Programlarda kullanılan fonksiyonlar dosya-sonuna erişildiğini ve okunacak karakter kalmadığını belirtmek için stdio.h'de tanımlı olan EOF değerini döndürür. Fakat aynı değer bazı fonksiyonlar tarafından hata durumunda da döndürüldüğü için daha sonra anlatılacak olan bazı özel makrolar kullanılarak bu iki durum birbirinden ayırt edilebilir.

EOF, dosyada bulunan bir karakter değildir fakat belli bir durumu ifade eden özel bir koddur. EOF, stdio.h'de negatif bir tamsayı değer (sisteme bağlı olmakla birlikte genel

olarak 0xFFFF, yani -1 kullanılır) olarak tanımlanmıştır. fgetc fonksiyonu (ve getc makrosu) (**int**)(**unsigned char**) değerler döndürür ve böylece 127'den büyük ASCII değerlerin negatif olmaması sağlanır. Ancak fonksiyonun hata yada dosya-sonu durumunu bildirmek için döndürdüğü negatif değerin saptanması gerekir. Bu nedenle programda, okunan değerler 8 bit karakterler olmasına rağmen, dosya sonunu belirten EOF kodunun tesbit edilebilmesi için c değişkeni **char** yerine **int** tipi değişken olarak bildirilmiştir. Örnek 9.1-2 ve 9.1-3'de görüldüğü gibi fgetc fonksiyonu, dosyada (yada akımda) bulunan son karakter pozisyonundan ötesini okuma girişiminde bulunduğu anda EOF döndürür ve döngüden çıkılır.

Örnek 9.1-4 mod2.c programı.

```
#include <stdio.h>
#include <stdlib.h>

#define NLDizgi          "abc\n123\ndef"
#define CR_NLDizgi       "abc\r\n123\r\ndef"

void DosyaYaz( char *, char *, char * );

int main(void)
{
  DosyaYaz( "text",    "w",   NLDizgi    );
  DosyaYaz( "binary1", "wb",  NLDizgi    );
  DosyaYaz( "binary2", "wb",  CR_NLDizgi );

  return 0;
}

void DosyaYaz( char *DosyaAdi, char *Mod, char *Dizgi )
{
  FILE *fp;

  if (  (fp = fopen( DosyaAdi, Mod )) == NULL  )
  {
      fprintf( stderr, "%s acilamadi.\n", DosyaAdi );
      exit(1);
  }
  fputs( Dizgi, fp );
  fclose(fp);
}
```

... Örnek 9.1-4 devam

Çıktı

abc\n123\ndef dizgisinin **text** dosyasına *text modu* nda yazılması.

```
A:\> dump      <enter>
Dosya adi ? : text
Hex (Dec)                Hexadecimal Kodlar                    ASCII Kod
000D(0013)   61 62 63 0D 0A 31 32 33 0D 0A 64 65 66 |abc..123..def |
Toplam byte: 13
                         CR-LF       CR-LF
```

DOS TYPE çıktısı:

```
A:\> type text  <enter>
abc
123
def
```

abc\n123\ndef dizgisinin **binary1** dosyasına *binary modu* nda yazılması.

```
A:\> dump      <enter>
Dosya adi ? : binary1
Hex (Dec)                Hexadecimal Kodlar                    ASCII Kod
000B(0011)   61 62 63 0A 31 32 33 0A 64 65 66          |abc.123.def  |
Toplam byte: 11
                         LF          LF
```

DOS TYPE çıktısı:

```
A:\> type binary1  <enter>
abc
    123
        def
```

abc\r\n123\r\ndef dizgisinin **binary2** dosyasına *binary modu* nda yazılması.

```
A:\> dump      <enter>
Dosya adi ? : binary2
Hex (Dec)                Hexadecimal Kodlar                    ASCII Kod
000D(0013)   61 62 63 0D 0A 31 32 33 0D 0A 64 65 66 |abc..123..def |
Toplam byte: 13
                         CR-LF       CR-LF
```

DOS TYPE çıktısı:

```
A:\>type binary2   <enter>
abc
123
def
```

Programlarda, sabit değerler yerine NULL ve EOF gibi sembolik sabitlerin kullanımı taşınabilirliği arttırır. Çünkü bir başka donanımda ve işletim sisteminde bu değerler farklı olabilir. Böyle bir durumda, sembolik sabitler kullanıldığı için programlarda değişiklik yapmak gerekmeyecektir.

mod1.c programında **if** deyimlerinde "==NULL" ifadesi kullanılmamıştır. Bir **if**, **while** yada **for** deyiminin test ifadesinin döndürdüğü 0 değeri *yanlış*, başka herhangi bir değer ise *doğru* kabul edilir. Burada **if** deyiminin test kısmında fp adres değişkeninin değeri kontrol edilir. Dosya herhangi bir sebeple açılamaz ise fp adres değişkenine fopen çağrısının döndürdüğü boş adres değeri (0) atanır ve !0 ifadesi ile test doğru kabul edilerek **if** deyiminin doğru bloğunda bulunan hata mesajı ve programdan çıkış deyimleri çalıştırılır.
fp değişkenine fopen çağrısından 0 harici herhangi bir geçerli değer (yani FILE tipi veri adresi) atanır ise test yanlış kabul edileceği için program çalışmasına devam eder.

Örnek 9.1-5'de tamsayılardan oluşan **int** tipi bir dizinin elemanlarının değerleri text modunda açılan sayilar.txt dosyasına yazılır.

Örnek 9.1-5 sayitext.c programı.

```
#include <stdio.h>
#include <stdlib.h>

#define SATIRBOY    80
#define DIZIBOY(s)    sizeof( s )/sizeof(s[0])

int dizi[] = { 123, 45, -25678, 500, -22 };

int main(void)
{
  FILE *fp;
  char DosyaAd[] = "sayilar.txt", Buf[SATIRBOY+1];
  int i;

  if (  (fp = fopen( DosyaAd, "w" ))  == NULL )
  {
    printf( "%s olusturulamadi\n", DosyaAd );
    exit(1);
  }

  for ( i = 0; i < DIZIBOY(dizi); i++ )
      fprintf(  fp, "dizi[%d] : %6d\n", i, dizi[i] );
  fclose( fp );

  if (  (fp = fopen( DosyaAd, "r" ))  == NULL )
  {
    printf( "%s acilamadi\n", DosyaAd );
    exit(2);
  }

  dizi[0] = 0;
```

... Örnek 9.1-5 devam

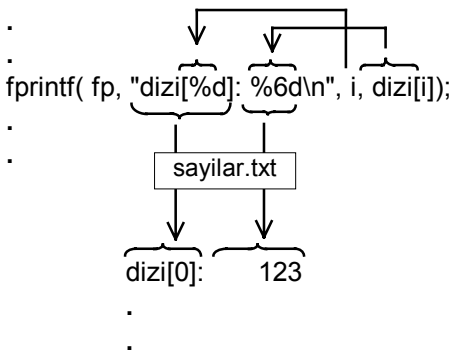
```
   fscanf( fp, "dizi[%d] : %d\n", &i,  &dizi[0] );
   printf( "dizi[%d] : %6d\n", i, dizi[0] );
   fgets( Buf, SATIRBOY, fp );
   printf( "%s", Buf );
   while (  (i = fgetc( fp )) != '\n' )
        printf( "%c", i );
   rewind( fp );

   printf( "\n-- rewind --\n" );
   printf( "%s\n", fgets( Buf, SATIRBOY, fp )  );
   fclose( fp );
   return 0;
}
```

Çıktı

```
dizi[0] :       123
dizi[1] :        45
dizi[2] :   -25678
-- rewind --
dizi[0] :       123
```

Programda ilk olarak sayilar.txt dosyası yazma amacıyla text modunda açılır. Dosya ismi argümanı, fopen fonksiyonuna bir karakter dizisi ismi (DosyaAd) olarak verilir. Bu uygulama, dosya isminin dizgi sabiti olarak yazılmasından daha pratiktir. Dosya açıldıktan sonra 5 elemanlı tamsayı dizi dizi'nin elemanları, fprintf fonksiyonu kullanılarak formatlanmış text şeklinde dosyaya yazılır. fprintf fonksiyonu printf fonksiyonuna benzer ve değişken sayıda argüman kabul eder. Ancak format dizgisi *format* ile belirtilen şekilde formatlanmış değerleri, printf fonksiyonundan farklı olarak sadece stdout akımına değil, *dosya_gostergesi* ile belirtilen disk dosyasına yada standart çıkışa yazar. stderr akımına hata mesajları yazılmasında çoğunlukla fprintf fonksiyonu kullanılır (Örnek 9.1-4). fprintf fonksiyonu yazılan karakter sayısını döndürür. Akımda hata oluşması durumunda ise negatif bir değer döndürür.

```
int fprintf( FILE *dosya_gostergesi, const char *format, ... );     /* stdio.h */
```



Sonuç olarak for döngüsü ile dosyaya, gerekli dönüşümler fprintf tarafından yapılarak 5 adet dizgi yazılır. Dosya işletim ortamında bulunan bir komut ile listelendiğinde aşağıdaki şekilde görülür:

```
dizi[0] :    123
dizi[1] :     45
dizi[2] : -25678
dizi[3] :    500
dizi[4] :    -22
```

Dizgilerin dosyada bulunuş şekillerine *format* dizgisi (printf fonksiyonunun format dizgisi ile aynı şekilde çalışır) karar verir. Aşağıdaki her iki deyim de aynı işi yapar:

```
fprintf( stdout, "deneme\n" ); ve  printf( "deneme\n" );
```

Daha sonra dosya text modunda okuma amacıyla açılır. Dosyadan ilk olarak fscanf fonksiyonu kullanılarak text bilgileri okunur.

```
int fscanf( FILE *dosya_gostergesi, const char *format, ... );     /* stdio.h */
```

Bu fonksiyon scanf gibi çalışır, fakat ilk argüman olarak dosya göstergesini alır. İkinci argüman olan format dizgisinin formu ve işlevi, scanf format dizgisi ile aynıdır ve dosyanın bulunulan pozisyonundan itibaren taranan girdi alanlarının nasıl yorumlanacağına karar verir. Sonuç olarak disk dosyasından yada standart girişten okunan dizgiler, belirtilen formata göre gerekli dönüşümler yapılarak C değişken değerlerine çevrilir ve argüman listesinde sıralanan bellek adreslerine atanır. Fonksiyon hatasız olarak çevrilen ve atanan alanların sayısını döndürür. Bu sayıya, okunan fakat atanmayan alanların sayısı dahil değildir. Döndürülen 0 değeri, hiç bir atama yapılmadığını gösterir. Dosya sonuna gelindiğinde EOF değeri döndürür. fscanf fonksiyonu scanf fonksiyonu ile aynıdır. Fakat scanf fonksiyonu sadece standart girişten okur. fscanf ise *dosya_gostergesi* ile belirtilen bir disk dosyasından da okuyabilir. Aşağıdaki her iki deyim de aynı işi yapar:

```
fscanf( stdin, "%d", &i );  ve  scanf( "%d", &i );
```

Programda görüldüğü gibi fscanf format dizgisi, fprintf format dizgisi ile aynı olmalıdır. Yani bilgiler yazıldıkları formatta okunmalıdırlar. fscanf iki sayısal değişkene, dosyadaki ilk dizgiden gerekli dönüştürmeleri yaparak iki tamsayı değeri okur. Okunan bilgiler printf ile ekrana listelenir.

Bu noktada, dosyanın ilk satırı okunmuş ve iki tamsayı değere dönüştürülmüş bulunuyor. Dosyanın içeriği düz metin olduğu için izleyen satırı da bir dizgi olarak işlem görebilir. fgets fonksiyonu kullanılarak ikinci dizgi okunur ve ekrana yazılır:

```
fgets( Buf, SATIRBOY, fp );
printf( "%s", Buf );
```

Bu fonksiyon, son argümanı olan *dosya_gostergesi* ile ifade edilen disk dosyasından yada standart girişten okuduğu dizgiyi, ilk argümanı olan *dizgi* karakter dizisine (bu bir dizgiye işaret eden **char** tipi adres değişkeni de olabilir) saklar:

```
char *fgets( char *dizgi, int n, FILE *dosya_gostergesi );    /* stdio.h */
```

Karakterler, bulunulan dosya pozisyonundan itibaren ilk NL karakterine rastlayıncaya kadar, dosya sonuna kadar yada okunan karakter sayısı *n*-1'e (son karakter dizgi sonunu belirten NUL byte için ayrılır) eşit oluncaya kadar okunur. Okunan karakterler ilk argüman olan karakter dizisine saklanır ve sonuna boş karakter (*null* karakter, \0) eklenir. Eğer okunmuş ise NL karakteri de karakter dizisine dahil edilir. Eğer ikinci argüman tamsayı *n*'in değeri 1 ise, karakter dizisi boş olur (""). fgets fonksiyonu hatasız çalışır ise, ilk argümanı olan adres değerini döndürür. Dosya sonu (*end-of-file*) yada hata durumunu bildirmek için NULL (boş adres değeri) döndürür. Yine daha sonra anlatılacak olan feof yada ferror makroları kullanılarak bu iki durum birbirinden ayırt edilebilir. Bu fonksiyon gets fonksiyonuna benzer. fgets fonksiyonu NL karakterine rastlar ise bu karakter de, girilen diğer karakterle birlikte saklanır. Fakat gets fonksiyonu okunan NL karakterini dizgiye aktarmaz. Ayrıca gets fonksiyonu sadece standart giriş akımı stdin'den okur. gets tarafından okunan ve saklanan karakterler sınırlandırılamaz. Bu nedenle fgets kullanımı daha uygundur. Her iki fonksiyon da, girilen karakterleri bir boş karakter ile sonlandırır:

```
char *gets( char *dizgi );    /* stdio.h */
```

gets fonksiyonu fgets ile aynı değerleri döndürür.

sayilar.txt dosyasının izleyen satırında bulunan karakterler fgetc fonksiyonu kullanılarak birer birer okunur. **while** döngüsünün test ifadesinde bulunan fgetc fonksiyonu, satır sonunu işaretleyen NL karakterine rastlayıncaya kadar karakterleri okumaya devam eder.

rewind fonksiyonu, *dosyanın pozisyon göstergesi*ni dosyanın başına ayarlar. Bu nedenle, son fgets çağrısı dosyada bulunan ilk dizgiyi okur. Programdaki ilk fgets çağrısında, okunan dizgi Buf dizisine aktarılır. fgets aynı zamanda ilk argümanı olan adres değerini (dizi ismi Buf, dizinin başlangıç adresini verir) döndürdüğü için, okunan dizgi aşağıdaki gibi listelenebilir:

```
printf( "%s\n", fgets( Buf, SATIRBOY, fp )  );
```

Çok büyük ASCII *text dosyaları* ile çalışırken, dizgilerle işlem yapmak (fputs, fgets fonksiyonları kullanılarak), karakter karakter işlem yapmaktan (fputc, fgetc fonksiyonları kullanılarak) daha hızlıdır.

Text ve Binary Format

Yukarıdaki örnekte, sayilar.txt dosyasına sayısal değerler fprintf fonksiyonu kullanılarak yazıldı. fprintf fonksiyonu gerekli dönüşümleri yaparak, tamsayı değerleri *text modunda açılan* sayilar.txt *dosyasına text formatında* yazdı. Dosya okunurken yine aynı şekilde, gerekli işlemler fscanf fonksiyonu tarafından yapıldı ve dosyada bulunan karakterler sayısal değerlere çevrilerek format dizgisinde belirtildiği şekilde tamsayı değişkenlere atandı. Formatlama ve çevirme yapan bu fonksiyonlarla okuma-yazma işlemleri yapıldığında veriler ASCII karakterler olarak okunur ve yazılır. Standart C kütüphanesinde, değerlerin hiç bir çevirme ve formatlama yapılmaksızın binary değerler olarak yazılmasını ve okunmasını sağlayan fwrite ve fread fonksiyonları bulunur. Örnek 9.1-5'de, text formatında ASCII karakterler olarak sayilar.txt dosyasına yazılan tamsayı değerler, binary formatta sayilar.bin dosyasına yazılır. Programda ilk olarak sayilar.bin dosyası binary modunda yazma amacıyla açılır ("wb"). Daha sonra fwrite fonksiyonu kullanılarak dizi dizisinin elemanı olan 5 tamsayı değerin tamamı binary formatta sayilar.bin dosyasına yazılır. fwrite fonksiyonu, *adr* tarafından işaret edilen bellek alanından *dosya_gostergesi* ile belirtilen disk dosyasının bulunulan pozisyonuna, belirtilen sayıda ve büyüklükteki veri elemanlarını yazar. *boy* bir veri elemanının byte olarak büyüklüğünü, *elsayi* ise bu büyüklükteki verilerden kaç tane yazılacağını belirtir:

```
size_t fwrite( const void *adr, size_t boy, size_t elsayi, FILE *dosya_gostergesi );  /* stdio.h */
```

Yazma işlemi sonrası pozisyon göstergesi güncellenir. Fonksiyon yazılan veri elemanı sayısını döndürür. Bu sayının, yazılacak olan eleman sayısından az olması hata oluştuğunu gösterir.

Örnek 9.1-6 sayibin.c programı.

```
#include <stdio.h>
#include <stdlib.h>

#define  DIZIBOY(s)   sizeof(s)/sizeof(s[0])

int dizi[] = { 123, 45, -25678, 500, -22 }, Buf[ DIZIBOY(dizi) ];

int main(void)
{
   FILE *fp;
   char DosyaAd[] = "sayilar.bin";
   int i;

   if (  (fp = fopen( DosyaAd, "wb" ))  == NULL )
   {
      perror( "yazma hatasi" );
      exit(1);
   }

   fwrite(  dizi, sizeof(dizi), 1, fp );
   fclose( fp );

   if (  (fp = fopen( DosyaAd, "rb" ))  == NULL )
   {
      perror( "okuma hatasi" );
      exit(2);
   }

   fread( Buf, sizeof(Buf), 1, fp );

   for ( i = 0; i < DIZIBOY(Buf); i++ )
         printf("dizi[%d] : %6d\n", i, Buf[i] );
   fclose( fp );

   return 0;
}
```

Çıktı

```
dizi[0] :    123
dizi[1] :     45
dizi[2] : -25678
dizi[3] :    500
dizi[4] :    -22
```

Örnek programda, ilk argüman olarak elemanları dosyaya yazılacak olan dizinin başlangıç adresini veren dizi ismi, ikinci argüman olarakta bu dizinin bellekte kapladığı alanın **sizeof** operatörü ile alınan byte sayısı (DOS ortamında, 5 adet 2-byte tamsayıdan oluşan dizi olduğu için 10 byte) verilir. Üçüncü argüman ise, verilen bellek adresinden

disk dosyasına bir adet 10 byte veri yazılacağını belirten 1 değeridir. Son argüman ise dosya göstergesi fp'dir:

```
fwrite(  dizi, sizeof(dizi), 1, fp );
```

Bu çağrı aşağıdaki şekilde de yapılabilir:

```
fwrite(  &dizi[0], sizeof(dizi[0]), DIZIBOY(dizi), fp );
```

Bu durumda, 10 byte büyüklüğünde 1 dizi yerine, 2 byte büyüklüğünde 5 eleman yazılır. Dolayısıyla her iki çağrıda aynı işi yapar. Programda dosya tekrar okuma amacıyla binary modunda ("rb") açılır. Bilgileri okumak için fwrite fonksiyonun karşılığı olan fread fonksiyonu kullanılır:

```
size_t fread( const void *adr, size_t boy, size_t elsayi, FILE *dosya_gostergesi );
/* stdio.h */
```

Bu fonksiyon, belirtilen sayıda (*elsayi*) ve büyüklükteki (*boy*) verileri, dosyanın bulunulan pozisyonundan okur ve *adr* ile işaret edilen bellek adresine saklar. İlk argüman basit bir değişkenin, dizinin yada yapı değişkeninin adresi olabilir. Okuma işlemi sonrası pozisyon göstergesi güncellenir. Fonksiyon okunan değerlerin sayısını döndürür. Bu sayı, dosya sonuna erişildiğinde yada okuma hatası oluştuğunda beklenenden az olacaktır. Daha sonra anlatılacak olan ferror ve feof makroları kullanılarak bu iki durumdan hangisinin oluştuğu saptanabilir.

Programda yer alan fread çağrısı ile okunan değerler, 10 byte uzunluğundaki Buf dizisine bir defada aktarılır:

```
fread( Buf, sizeof(Buf), 1, fp );
```

Örnekteki fread çağrısı aşağıdaki şekilde de olabilir:

```
fread( &Buf[0], sizeof(Buf[0]), DIZIBOY(Buf), fp );
```

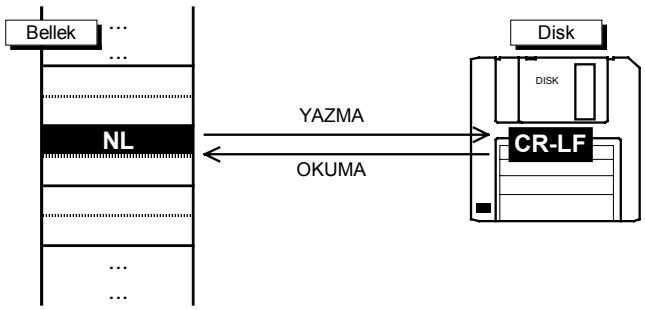
Burada tek bir elemanın byte olarak boyutu verilir ve okunacak eleman sayısı belirtilir. Her iki çağrıda da 10 byte veri transferi gerçekleşir. Buf ve &Buf[0] ifadelerinin her ikisi de, dizinin başlangıç adresini verir.
Binary mod ve binary format birbirine karıştırılmamalıdır. *Modlar* (binary ve text) fopen fonksiyonuna aktarılan argümanlardır ve daha öncede anlatıldığı gibi NL karakterinin çevrilmesine ve EOF işaretlerinin yorumlanmasına etki eder. *Formatlar* ise (binary ve text) sayısal değerlerin temsil edilme şekilleridir. Dolayısıyla, text modunda açılmış bir dosyaya yazmak için fwrite fonksiyonu kullanılır ise, her bir NL karakteri CR-LF çifti ile değiştirilir. fread fonksiyonu kullanılarak text modunda açılan bir dosyadan okuma yapıldığında da, yine CR-LF karakterleri tek bir NL karakterine çevrilir.

Text mod, ASCII text dosyalarından okumak ve bu dosyalara yazmak için uygundur. Text mod kullanılır ise, NL karakteri CR-LF çiftine çevrileceği için DOS'un TYPE komutu kullanılarak dosya ekrandan listelenebilir. Okuma-yazma tekniği birbiri ile aynı olduğu sürece, ASCII karakter dizgileri ile işlemler yapılırken ve bunları diske yazarken text yada binary formatın kullanılması pek farketmez. Fakat text mod ve text format sayısal değerleri yazmak ve okumak için uygun değildir. Sayısal değerleri (tamsayılar ve kayan-noktalı sayılar) *binary moddaki dosyalara, binary formatta* saklamak daha uygundur. Bunun sebepleri şunlardır:

- Binary format hemen hemen her zaman disk alanından tasarruf sağlar. Text formatında, "12345.678" rakamları ASCII kodları için 8 byte, ondalık nokta için bir byte ve değişkenler arasındaki ayraçlar için de bir yada daha fazla byte gerektirir. Binary formatta ise kayan-noktalı bir sayı, değeri ne olursa olsun 4 byte kullanır. **short** tamsayılar sadece 2 byte kullanır (DOS ortamında sayilar.bin dosyası diskte 10 byte, sayilar.txt dosyası ise 90 byte yer kaplar). Donanıma bağlı olarak binary formattaki bir tamsayı diskte daima 2 byte (DOS altında) yer kaplar. Text formatındaki bir tamsayı ise, çeşitli sayıda byte kullanır; bu tek bir karakter olabileceği gibi (örnek; "5"), altı karakter de olabilir (örnek "-10186").

- Binary format kullanımı bilgisayar zamanından tasarruf sağlar. Sayısal bir değeri diske yazmak için fprintf kullanıldığında, bilgisayar kendi kullandığı iki tabanlı sayıları karakter serilerine çevirmelidir. Yine bu işlemler sırasında kullanılan fscanf ise, karakterleri belleğe okurken tersini yapar ve ikili sayılara çevirir. Binary formatta bu çevrimlerin hiç biri olmaz.

Şekil 9.1-2 Text modunda okuma ve yazma işlemleri sırasında satır-sonu karakterinin yorumlanması.



- Binary format kayan-noktalı sayıların hassasiyetini korur. İkili tabandan ASCII ondalık koda çevirmek ve tekrar işlemi tersine tekrarlamak hassasiyeti etkileyebilir.

- Dizilerin ve yapıların binary olarak saklanması hızlıdır. Örneğin 100 elemanlı bir dizinin tamamını okumak ve her birini disk dosyasına yazmak gereksizdir (döngü vs... ile). Yerine sadece bir kez fwrite fonksiyonu çağrılır ve saklanacak olan dizinin boyutları aktarılır.

Görüldüğü gibi, binary format birçok avantaj sunar. Aşağıdaki program diskte 20 byte büyüklüğünde sayi.bin dosyası oluşturur. fwrite fonksiyonu kullanılarak yazılan bilgiler fread fonksiyonu ile okunur. Bundan sonra dosyanın pozisyon göstergesi rewind fonksiyonu ile başa alınır ve binary dosya fgetc fonksiyonu kullanılarak byte-byte okunur. Okunan değerler onaltılık sayılar olarak listelenir.

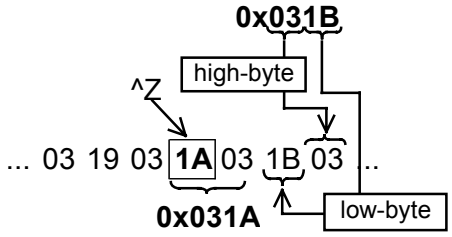
Örnek 9.1-7 bin.c programı.

```
#include <stdio.h>
#include <stdlib.h>

#define  EL_SAY  10

int main(void)
{
   FILE *fp;

   int dizi[ EL_SAY ], a_dizi[ EL_SAY ];
   int i;

   for ( i = 0; i < EL_SAY; i++ )
         dizi[ i ] = 790 + i;

   if (  (fp = fopen( "sayi.bin", "wb" ))  == NULL )
   {
      perror( "yazma hatasi" );
      exit(1);
   }
   fwrite(  &dizi[0], sizeof(dizi[0]), EL_SAY, fp );
   fclose( fp );

   if (  (fp = fopen( "sayi.bin", "rb" ))  == NULL )
   {
      perror( "okuma hatasi" );
      exit(2);
   }
   fread( &a_dizi[0], sizeof(a_dizi[0]), EL_SAY, fp );
   for ( i = 0; i < EL_SAY; i++ )
         printf("dizi[%d] : %4d, 0x%04X\n", i, a_dizi[i], a_dizi[i] );
   rewind( fp );
   while (  (i = fgetc( fp )) != EOF )
            printf( "%02X ", i );
   fclose( fp );
   return 0;
}
```

...Örnek 9.1-7 devam

Çıktı

```
dizi[0] : 790,   0x0316
dizi[1] : 791,   0x0317
dizi[2] : 792,   0x0318
dizi[3] : 793,   0x0319
dizi[4] : 794,   0x031A
dizi[5] : 795,   0x031B
dizi[6] : 796,   0x031C
dizi[7] : 797,   0x031D
dizi[8] : 798,   0x031E
dizi[9] : 799,   0x031F
16 03 17 03 18 03 19 03 1A 03 1B 03 1C 03 1D 03 1E 03 1F 03
```



Çıktıda görüldüğü gibi, diskteki 2 byte sayıların *low byte*'ı, *high byte* öncesinde bulunur. İlk iki byte 0x0316 yani 790 sayısıdır. Diğerleri de bu şekilde devam eder.

Diskte 20 byte yer kaplayan sayi.bin dosyası DOS işletim sisteminde bulunan TYPE komutu ile listelendiğinde, dosya ASCII karakterler içermediği için ekranda çeşitli grafik karakterler olarak listelenir. Fakat bu karakterlerin yalnızca 8 tane olduğu görülür. Bunun nedeni, sayi.bin dosyası içinde bulunan 1A karakterinin DOS işletim sisteminde bir text dosyasının sonunu işaretleyen EOF (Ctrl+Z) işareti olmasıdır. Aslında bu karakter, 0x031A (794) sayısının low byte'ıdır. Ayrıca, bu dosya okumak amacıyla text modunda açılarak, yine fgetc fonksiyonu ile byte-byte okunduğunda ancak 1A karakterine kadar olan 8 byte listelenir (16 03 17 03 18 03 19 03). Örnekte görüldüğü gibi binary modunda 1A karakterinin hiç bir anlamı yoktur. Bu nedenle tüm karakterler sorunsuz olarak listelenir. ■

9.2 Giriş/Çıkış Hatalarının İşlenmesi

Giriş/Çıkış hatalarının saptanmasında ve işlenmesinde giriş/çıkış fonksiyonlarının döndürdüğü değerler ve C kütüphanesinde bulunan bazı özel makro ve fonksiyonlar kullanılabilir. Örnek programlarda dosya açma hatalarını saptamak için fopen fonksiyonunun döndürdüğü adres değerinin NULL olup olmadığı kontrol edildi. Ayrıca bazı fonksiyonların döndürdükleri değerin dosya sonuna gelindiğini belirten EOF olup olmadığı kontrol edildi. Daha sonra Örnek 9.1-4'de fprintf fonksiyonu kullanılarak, Örnek 9.1-6 ve Örnek 9.1-7'de ise perror fonksiyonu kullanılarak ekrana hata mesajları yazıldı. Örnek programlarda genel olarak, programın okunabilirliğini arttırmak için giriş/çıkış fonksiyonları kullanılırken hata kontrolü yapılmadı.

FILE veri yapısı, veri transferinde kullanılan tampon alana ait bilgileri (tampon alanın yeri, tampondaki o anda okunacak yada yazılacak olan karakterin pozisyon bilgisi gibi), *handle* olarak adlandırılan dosya numarasını, giriş/çıkış işlemleri sırasında hata oluşup oluşmadığını yada okuma sırasında dosya sonuna erişilip erişilmediğini gösteren kontrol ve durum bilgisi (*hata göstergesi* ve *dosya-sonu göstergesi*) ve dosyadan okuma yada yazma işlemlerinden hangisinin yapıldığı bilgisini içerir.

Okuma sırasında dosya sonuna erişen bir fonksiyon dosya-sonu göstergesine (*end-of-file indicator*) 0 olmayan bir değer atar. Aynı şekilde, okuma yada yazma hatası ile karşılaşan bir fonksiyon hata göstergesine (*error indicator*) 0 olmayan bir değer atar. Bazı kütüphane fonksiyonları hata durumunda da yanıltıcı olabilecek EOF bilgisi döndürürler. Standart kütüphane makroları feof ve ferror hangi durumun gerçekleştiğini saptamak için kullanılabilir. clearerr fonksiyonu, hata göstergesini silerek hata durumunu ortadan kaldırır. Örnek 9.2-1'de bu fonksiyon ve makrolar kullanılarak giriş/çıkış hataları saptanır ve buna göre bazı deyimler çalıştırılır.

Örnek 9.2-1 hatatest.c programı.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
  FILE *fp_yaz, *fp_oku;
  int i;
  char isim[13] = "";

  printf("Diskte bulunmayan bir dosya adi giriniz: ");
  gets( isim );
  if ( *isim == '\0' )
  {
```

...Örnek 9.2-1 devam

```
    fprintf( stderr, "Dosya ismi girilmedi\n" );
    /* exit(1); */
  }
  if ( (fp_oku = fopen( isim, "r" ) ) == NULL )
  {
    perror( "Dosya okunamadi " );
    fprintf( stderr, "Dosya okunamadi : %s\n", strerror( errno ) );
    /* exit(2); */
  }
  do
  {
    printf("Diskte bulunmayan bir baska dosya adi giriniz: ");
    gets( isim );
  }  while ( (fp_yaz = fopen( isim, "r" )) != NULL );
  if ( (fp_yaz = fopen( isim, "w" )) == NULL )
  {
    fprintf( stderr, "Dosya acilamadi" );
    exit(3);
  }
  printf("\n%s dosyasi yazma amaciyla acildi.\n\n", isim );
  if ( fgetc( fp_yaz ) == EOF )
       fprintf( stderr, "Hata: fgetc okuyamadi\n");
  if ( feof( fp_yaz ) )
       fprintf( stderr, "EOF\n");
  if ( ferror( fp_yaz ) )
       fprintf( stderr, "Hata: ferror\n");
  for ( i = 0; i < 10; i++ )
  {
       printf("i: %d, ", i );
       if ( fprintf( fp_yaz, "%d", i ) < 0 )
            fprintf( stderr, "Hata: fprintf, ");
       if ( ferror( fp_yaz) )
            fprintf( stderr, "Hata: ferror");
       puts( "" );
       if ( i == 4 )
       {
         puts( "--clearerr--" );
         clearerr( fp_yaz );
       }
       if ( i == 7 )
       {
         puts( "--fclose--" );
         fclose( fp_yaz );
       }
```

...Örnek 9.2-1 devam

```
  } /* for */
  if ( fclose( fp_yaz ) )  fprintf( stderr, "Hata: fclose\n");
  if ( (fp_oku = fopen( isim, "r" ) ) == NULL )
  {
    fprintf( stderr, "%s dosyasi okunamadi\n", isim );
    exit(4);
  }
  printf( "%s dosyasinda bulunan karakterler : ", isim );
  while ( (i = fgetc( fp_oku )) != EOF )
        fputc( i, stdout );
  fclose( fp_oku );
  puts( "" );
  return 0;
}
```

Çıktı

```
Diskte bulunmayan bir dosya adi giriniz: deneme
Dosya okunamadi : No such file or directory
Dosya okunamadi : No such file or directory
Diskte bulunmayan bir baska dosya adi giriniz: deneme
deneme dosyasi yazma amaciyla acildi.
Hata: fgetc okuyamadi
Hata: ferror
i: 0, Hata: ferror
i: 1, Hata: ferror
i: 2, Hata: ferror
i: 3, Hata: ferror
i: 4, Hata: ferror
--clearerr--
i: 5,
i: 6,
i: 7,
--fclose--
i: 8, Hata: fprintf, Hata: ferror
i: 9, Hata: fprintf, Hata: ferror
Hata: fclose
deneme dosyasinda bulunan karakterler : 01234567
```

Örnek 9.2-1'de ilk olarak diskte bulunmayan bir dosya (deneme dosyası) fopen fonksiyonu tarafından okuma amacıyla açılmak istenir. Bu durumda fopen boş adres değeri döndürür ve **if** bloğunda yer alan ve ekrana hata mesajları yazan perror ve fprintf deyimleri çalıştırılır.

Hata durumunu oluşturan kütüphane fonksiyonu çağrısının hemen sonrasına yerleştirilen perror fonksiyonu, ekrana ilk olarak argüman olarak verilen *dizgi*'yi yazar. Bu dizgi, boş dizgi olabilir (""). Bunu, ":" işareti, sistem hata mesajı, bir boşluk ve NL karakteri izler:

```
void perror( const char *dizgi );        /* stdio.h */
```

perror fonksiyonu çıktıyı, stderr'e yazar. Böylece hata mesajları, yönlendirme yapılsa bile ekrana yazılır. Programda izleyen satırda görüldüğü gibi aynı sistem hata mesajı fprintf ve strerror fonksiyonu kullanılarak stderr'ye yazılabilir. Bu fonksiyon, *hata_kodu* tamsayı değişkeninin değerine karşılık gelen sistem hata mesajı dizgisinin adresini döndürür:

```
char *strerror( int hata_kodu );     /* string.h */
```

Programda, strerror fonksiyonuna argüman olarak errno değişkeni aktarılır. Bir C kütüphane fonksiyonunda hata oluştuğunda, fonksiyon global tamsayı sistem değişkeni errno'ya belli bir değer atar. perror fonksiyonu sistem hata mesajları tablosunda bulunan ve errno'nun değerine karşılık gelen mesajı ekrana yazar. İzleyen satırdaki standart hata akımına yazan fprintf çağrısında yer alan strerror fonksiyonu da, yine errno'nun değerine karşılık gelen bu hata mesajına işaret eden bir adres değeri döndürür.
Dolayısıyla çıktıda da görüldüğü gibi her iki çağrı da aynı işi yapar. Programların taşınabilir olması için direk errno değişkeni kullanılarak hata mesajı listelemek uygun değildir. Çünkü hata numaraları sisteme bağlı olarak değişebilir. Programdan çıkışı önlemek amacıyla, ilk iki exit çağrısı açıklama satırı haline getirilir.

Programa tekrar bir dosya ismi girilir (deneme). Fakat bu dosya yazma amacıyla açılacağı için, yanlışlıkla diskte bulunan bir dosya ismi girildiğinde içindeki bilgiler kaybolacaktır. Girilen isimde bir dosyanın diskte bulunmadığından emin olmak için, dosya okuma amacıyla açılır. Eğer dosya mevcut ise, **do-while** döngüsü diskte bulunmayan bir dosya ismi girilinceye kadar devam eder. Yine deneme dosyası yazma amacıyla açılır ve hata oluşturmak için fgetc fonksiyonu ile dosyadan okuma girişiminde bulunulur. Dosya yazma amacıyla açılmış olduğu için okuma girişimi akımda hata oluşturur. Sonuç olarak fgetc fonksiyonu EOF döndürür. Fakat bunun gerçekten bir EOF durumu olup olmadığı feof ve ferror makroları kullanılarak anlaşılabilir. feof makrosu, açılan dosyanın FILE veri yapısında bulunan dosya-sonu göstergesini belli bir değer için kontrol eder. Dosyadan okuyan bir fonksiyon dosya sonuna erişir ise, bu göstergeye 0 olmayan bir değer atar. Bu değer, **if** deyiminin test ifadesinde yer alan feof makrosu tarafından saptanır.

Bu durumda makro 0 dışında bir değer döndürerek (aksi taktirde 0 döndürür) if deyiminin doğru bloğunun çalışmasını sağlar:

```
int feof( FILE *dosya_gostergesi );     /* stdio.h */
```

Aynı şekilde, ferror makrosu da hata göstergesini belli bir değer için kontrol eder:

```
int ferror( FILE *dosya_gostergesi );        /* stdio.h */
```

dosya_göstergesi ile belirtilen dosyadan en son okuma yada dosyaya en son yazma işleminde hata oluşur ise ilgili fonksiyon hata göstergesinin değerini değiştirir ve 0 olmayan bir değer atar. Bunu saptayan ferror makrosu da 0 dışında bir değer döndürerek (aksi taktirde 0 döndürür) **if** deyiminin doğru bloğunu çalıştırır. Sonuç olarak çıktıda da görüldüğü gibi, yazma amacıyla açılmış bir dosyadan okuma girişiminde bulunulduğu için akımda hata oluşmuştur ve bu bir EOF durumu değildir.

Dosyaya **for** döngüsü içinde fprintf fonksiyonu ile i değişkeninin değerleri yazılır. Hata göstergesi otomatik olarak silinmediği için akımda hata durumu devam eder. i değişkeninin değeri 4 olduğunda çağrılan clearerr fonksiyonu, *dosya_gostergesi* ile ifade edilen dosyanın hata ve dosya-sonu göstergelerini siler:

```
void clearerr( FILE *dosya_gostergesi );        /* stdio.h */
```

Görüldüğü gibi, clearerr çağrılarak sıfırlanıncaya kadar hata göstergesi değerini korur (daha sonra anlatılacak olan bir fseek, fsetpos yada rewind çağrısı da hata göstergesini siler). i değişkeninin değeri 7 olduğunda dosya kapatılır. Dosyaya yazamadığı için fprintf fonksiyonu negatif bir değer döndürür ve if bloğundaki hata mesajı listelenir. Ayrıca fprintf fonksiyonu hata göstergesi aktif hale getirir (0 olmayan bir değer atar). Bunu saptayan ferror fonksiyonu da **if** bloğunun doğru kısmını çalıştırarak ekrana hata mesajı yazar. Döngü çıkışında, fclose ile dosya tekrar kapatılmaya çalışılır. Fonksiyon EOF döndürür ve **if** deyiminin doğru kısmı çalışarak ekrana hata mesajı listelenir. Programda son olarak deneme dosyasına yazılan karakterler listelenir. Görüldüğü gibi, fclose ile dosya kapatılmadan önceki tüm değerler yazılır. Programlarda, herhangi bir hata durumunda programın çalışmasını durdurarak çıkmak için standart kütüphane fonksiyonu exit kullanıldı. Bu fonksiyonun prototipi standart başlık dosyası stdlib.h'de bulunur:

```
void exit( int tamsayı );  /* stdlib.h */
```

exit fonksiyonuna argüman olarak aktarılan *tamsayı* çıkış kodu olarak adlandırılır ve programı çalıştıran ortama (DOS'ta ERRORLEVEL parametresine yada UNIX işletim sisteminde benzer bir shell değişkenine) aktarılır. Eğer program bir DOS toplu komut dosyası içinden çalıştırılıyor ise, bu dosyada bulunan IF ERRORLEVEL komutu ile döndürülen çıkış kodu test edilerek sonuca göre çeşitli dallandırmalar yapılarak bazı işlemler gerçekleştirilebilir. Genel olarak, normal bir çıkışı ifade etmek için 0 değeri, hata oluşması sonucu çıkışı ifade etmek için ise 0 dışında herhangi bir değer (*non-zero value*) kullanılabilir. Programdan çıkmak için main bloğu içinde kullanılan bir **return** *ifade* deyimi, exit(*ifade*) ile aynı işi yapar. exit fonksiyonu giriş/çıkış işlemleri için açılan dosyalar ile ilgili tüm tampon alanları boşaltır (*flushing*) ve tüm dosyaları kapatır. Çıkış kodu olarak, stdlib.h'de tanımlanmış olan sembolik sabitler EXIT_SUCCESS ve

EXIT_FAILURE kullanılabilir. Bu sabitler, başarılı yada hatalı bir çıkışı ifade etmek ve programların taşınabilir olması için kullanılabilir.

Örnek 9.2-2'de ilk olarak global sistem değişkeni errno'ya 0 değeri atanır. Daha sonra diskte bulunmayan bir text dosyası (DOSYA) okuma amacıyla açılmak istenir. fopen fonksiyonu, DOSYA dosyası diskte bulunmadığı için okuma amacı ile açamaz ve boş adres değeri döndürür. Bu nedenle, if deyiminin doğru bloğunda bulunan deyimler çalıştırılır.

Örnek 9.2-2 error.c programı.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>   /*  errno degiskeni bildirimi icin  */
int main(void)
{
  FILE *fp_oku;
  errno = 0;
  printf( "errno : %d\n", errno );
  if ( (fp_oku = fopen( "DOSYA", "r")) == NULL )
  {
    printf( "errno : %d\n", errno );
    perror( "DOSYA bulunamadi\n" );
    exit(1);
  }
  return 0;
}
```

Çıktı

```
errno : 0
errno : 2
DOSYA bulunamadi
: No such file or directory
```

Çıktıda da görüldüğü gibi fopen çağrısı sonrası errno değişkeninin değeri değişir. Bu değere göre çalışmak üzere, programlar içinde çeşitli hata rutinleri dallandırılabilir. Fakat daha önce de belirtildiği gibi programların taşınabilirliği açısından bu değerin kullanılması uygun değildir. Bunun yerine, hata mesajları perror fonksiyonu ile ekrana yazılır.

printf fonksiyonu çıktısını stdout'a yani standart çıkış birimi ile ilgili akıma gönderir. Standart çıkış birimi yönlendirme yapılmadığı sürece ekrandır. Dolayısıyla printf çıktıları yönlendirilmediği sürece ekranda listelenir. Komut satırında error > hata.err uygulandığında, printf fonksiyonunun çıktıları disk dosyası hata.err'e yönlendirilir ve bu dosyaya yazılır. perror çıktısı ise standart hata çıkış akımı olan stderr'e (dolayısıyla

ekrana) gönderilir. Böylece standart çıkışın bir çıktı dosyasına (hata.err) yönlendirilip yönlendirilmediğine yada bir *pipe* işleminin ("|" kullanılarak bir programın standart çıkışının başka bir programın standart girişine yöneltilmesi. Örnek : prog1 | prog2) olup olmadığına bakılmaksızın hata çıktısı ekrana gönderilir:

```
A:\> error > hata.err   } printf çıktısı hata.err dosyasına yazılır
DOSYA bulunamadi
: No such file or directory } perror çıktısı ekrana yazılır
```

Bu nedenle hata mesajlarının ekrana yazılmasında printf yerine perror fonksiyonu yada çoğunlukla stderr'e yazan fprintf çağrısı tercih edilir. ■

9.3 Giriş/Çıkış İşlemlerinde Komut Satırı Argümanlarının Kullanılması

Dosya isimleri, disk dosyaları üzerinde giriş/çıkış işlemleri gerçekleştiren programlara komut satırından aktarılabilir. Komut satırına girilen tüm bilgiler program ismi de dahil olmak üzere main fonksiyonunun argümanlarıdır.

Aşağıdaki örnekte yer alan dump programı, herhangi bir disk dosyasını binary modunda açarak içerdiği verileri onaltılık tabanda iki haneli tek byte değerler olarak standart çıkışa yazar. Ayrıca bu değerlerin ASCII karakter karşılıkları da her satırda listelenir. Grafik olarak ekrana yazılamayan karakterler "." ile değiştirilir. Program isminden sonra komut satırına girilen argümanlar verilerin okunacağı disk dosyası isimleri olarak ele alınır. Eğer argüman listesinde bulunan herhangi bir dosya açılamaz ise ekrana hata mesajı yazılarak programdan çıkılır. Herhangi bir argüman girilmez ise, ekrandan dosya ismi sorulur. Dosya ismi girilmediğinde ise, programın kullanım formatını belirten bilgi ve hata mesajı yazılarak programdan çıkılır. Dosyalar binary modunda açılarak, CR/LF karakter çiftlerinin NL karakterine çevrilmesi önlenmiştir. Bu nedenle, DOS işletim sisteminde bulunan dir komutunun verdiği dosya büyüklüğü ile dump çıktısında yer alan byte sayısı aynıdır.

Örnek 9.3-1 dump.c programı.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define BASLIK   "Hex (Dec )  ------------ Hexadecimal Kodlar \
-------------- --- ASCII Kod ----"

void Listele( char * );

int main( int argc, char *argv[])
{
   char DosyaAdi[ 13 ];

   if  ( argc == 1 )
   {
      printf( "Dosya adi ? : " );
      gets( DosyaAdi );
      if  ( '\0' == *DosyaAdi )
      {
         fprintf( stderr, "Dosya adi girilmedi.\n" );
         fprintf( stderr, "Kullanim: dump <dosya listesi>\n" );
         exit(0);
      }
      Listele( DosyaAdi );
   }
   else
      while ( *(++argv) )
      {
         printf( "Dosya adi : %s\n", *argv );
         Listele( *argv );
      }

   return 0;
}

void Listele( char *GirisDosya )
{
   FILE *fp_oku;
   unsigned char Dizgi[ 16 ];
   unsigned ToplamOku = 0;
   int i, Okunan;

   if ( (fp_oku = fopen( GirisDosya, "rb")) == NULL )
   {
      fprintf( stderr, "%s acilamadi.\n", GirisDosya );
      exit(1);
   }

   puts( BASLIK );
```

...Örnek 9.3-1 devam

```
  while ( (Okunan = fread( Dizgi, 1, 16, fp_oku )) > 0 )
  {
       ToplamOku += Okunan;
       printf( "%04X(%04u) ", ToplamOku, ToplamOku );
       for ( i = 0; i < 16; ++i )
            if ( i < Okunan )
            {
              printf( " %02X", Dizgi[i]);
              if ( !isprint( Dizgi[i] ) )   Dizgi[i] = '.';
            }
            else
            {
              printf( "    " );
              Dizgi[i] = ' ';
            }
            printf( " |%16.16s|\n", Dizgi );
  } /* while */
  printf( "Toplam byte: %d\n", ToplamOku );
  fclose( fp_oku );
}
```

Çıktı

Önceki bölümlerde dump çıktıları verildi.

Örnek 9.3-2'de tipik bir kelime, karakter ve satır sayma programı olan say.c yer alır. Program komut satırından girilen seçeneklere (*option* yada *switch*) göre, yine komut satırında yer alan bir yada daha fazla dosyanın kelime, karakter yada satır sayısını ekrana listeler. Komut satırında sadece seçenekler bulunuyor ise giriş dosyası olarak standart giriş kabul edilir.

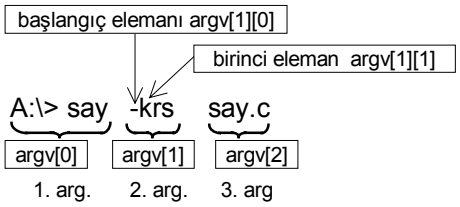
Programda yer alan ilk if deyiminin test ifadesinin doğru olması için, komut satırında program isminden başka argümanların bulunması (argc > 1) ve bu argümanlardan ikincisinin ilk karakterinin seçenek argümanı başlangıcını ifade eden '-' karakteri olması gerekir:

```
*argv[1] == '-'
```

Burada, argv[1][0] == '-'' yada **++argv == '-' kullanılabilir. Fakat ikinci ifade argv'nin değerini değiştirdiği için tercih edilmez. Bu iki koşul sağlandığında if deyiminin doğru bloğunda bulunan for döngüsü ile girilen seçenekler belirlenir. Bunun için char * tipindeki Opsiyon adres değişkenine ikinci argümanın birinci elemanının (argv[1]+1) bellek adresi atanır. Opsiyon değişkeninin değeri arttırılarak izleyen seçenek harfleri taranır. k, r ve s harflerinden başka herhangi bir harf girilir ise ekrana

girilen harf ile birlikte hata mesajı yazılarak programdan çıkılır. Yine aynı blokta argüman sayısı 2'den büyük ise seçenek argümanından sonra dosya isimleri girildiği kabul edilir ve bu girilen isimlerin herbiri Say fonksiyonuna argüman dizgisi bellek adresi olarak aktarılarak teker teker işlenir. Seçenek argümanından başka herhangi bir argüman bulunmuyor ise bilgiler standart girişten okunur.

DOS işletim sisteminde say.c programı için
3 argümanlı bir komut satırı:



argc = 3

Örnek 9.3-2 say.c programı.

```
#include <stdio.h>
#include <stdlib.h>

#define KELIME      1       /* kelime icin isaret                    */
#define BTABNL      0         /* bosluk, tab yada NL icin isaret     */
#define NL          '\n'
#define TAB         '\t'
#define BOSLUK      ' '
#define DOGRU       1
#define YANLIS      0

void Say( int S, int K, int R, char *DosyaAd );

main( int argc, char *argv[] )
{
  char *Opsiyon;
  int S = YANLIS, K = YANLIS, R = YANLIS;
  int i;

  if ( argc > 1 && *argv[1] == '-' )
  {
      for ( Opsiyon = argv[1]+1; *Opsiyon; ++Opsiyon )
          switch( *Opsiyon )
          {
                case 'k':
                      K = DOGRU;
                      break;
                case 'r':
                      R = DOGRU;
                      break;
                case 's':
```

... Örnek 9.3-2 devam

```
                        S = DOGRU;
                        break;
                default :
                        fprintf( stderr, "Bilinmeyen opsiyon %c\n", *Opsiyon );
                        exit(0);
            }
        if ( argc > 2 )
            for ( i = 2; i < argc; ++i )
                Say( S, K, R, argv[i] );
        else
                Say( S, K, R, "" );
    }
    else
    {
        fprintf( stderr, "Kullanim     : say -opsiyon [dosya_listesi]\n"   );
        fprintf( stderr, "Opsiyonlar : (k)elime, ka(r)akter, (s)atir \n"   );
        fprintf( stderr, "Ornek      : say -krs say.c\n"                   );
        exit(0);
    }
    return 0;
}

void Say( int S, int K, int R, char *DosyaAd )
{
    FILE *fp_oku;
    long SatirSayi = 0, KelimeSayi = 0, KarSayi = 0;
    int Onceki = BTABNL, c;

    if ( *DosyaAd == '\0' )
        fp_oku = stdin;
    else if ( (fp_oku = fopen( DosyaAd, "r" )) == NULL )
    {
        fprintf( stderr, "%s acilamadi\n", DosyaAd );
        exit(1);
    }
    while (  (c = fgetc( fp_oku )) != EOF  )
    {
        ++KarSayi;
        if ( c == NL )
            ++SatirSayi;
        if ( Onceki == BTABNL && c != BOSLUK && c != TAB && c != NL )
        {
            ++KelimeSayi;
            Onceki = KELIME;
        }
        if ( Onceki == KELIME  &&  (c == BOSLUK || c == TAB || c == NL) )
            Onceki = BTABNL;
    } /* while */
```

... Örnek 9.3-2 devam

```
  if (  ferror( fp_oku ) )
  {
       fprintf( stderr, "%s okunamadi\n", DosyaAd );
       exit(2);
  }
  if ( *DosyaAd == '\0' )
       puts("   Giris Dosyasi : stdin");
  else
  {
       printf("   Giris Dosyasi : %s\n", DosyaAd );
       if ( fclose(fp_oku) )
       {
            fprintf( stderr, "%s kapatilamadi\n", DosyaAd );
            exit(3);
       }
  }

  if ( S == DOGRU )
       printf( "\tsatir    : %ld\n", SatirSayi        );
  if ( K == DOGRU )
       printf( "\tkelime   : %ld\n", KelimeSayi   );
  if ( R == DOGRU )
       printf( "\tkarakter : %ld\n", KarSayi       );
}
```

Çıktı

```
A:\>say -krs say.c
        Giris Dosyasi :  say.c
                satir    :  175
               kelime    :  553
              karakter :  3451
```

NOT: DOS işletim sistemi altında say.c dosyası dir komutu ile listelendiğinde, 3626 karakterden oluştuğu görülür. Bunun nedeni programda dosyanın text modunda açılmış olması ve dolayısıyla diskteki CR/LF karakter çiftlerinin bellekte tekbir NL ile temsil ediliyor olmasıdır (175 + 3451 = 3626).

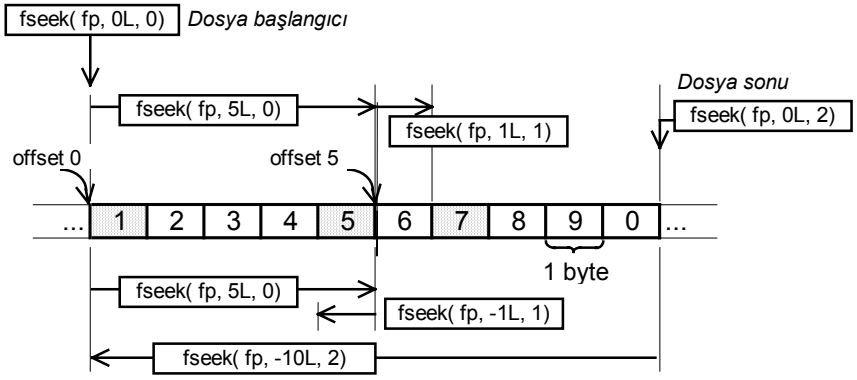
Argüman girilmediğinde yada ikinci argümanın ilk karakteri '-' değil ise **if** deyiminin **else** kısmında bulunan deyimler çalıştırılır ve ekrana programın kullanım formatı yazılarak programdan çıkılır. ■

9.4 Rastgele Erişim

Örnek programlarda görüldüğü gibi, giriş ve çıkış işlemleri normal olarak sıralı gerçekleşir. Yani her disk dosyasından okuma yada disk dosyasına yazma, bir önceki işlemin yapıldığı dosya pozisyonunu izleyen pozisyonda gerçekleşir. Okuma yada yazma işlemini gerçekleştiren fonksiyon, işlem sonrası dosyanın pozisyon göstergesini (yada okuma yazma kursorü) izleyen pozisyona işaret edecek şekilde arttırır. C kütüphanesinde bulunan bazı fonksiyonlar kullanılarak istenen dosya pozisyonuna ulaşılabilir ve böylece sıralı olmayan (rastgele) erişim gerçekleştirilebilir. Bu fonksiyonlar fseek, ftell, fsetpos, fgetpos ve rewind fonksiyonlarıdır.

Şekil 9.4-1 fseek fonksiyonu.

1234567890 karakterlerini içeren bir dosyada fseek ile pozisyon belirleme:




```
fseek( fp, 0L, 0);      } Dosya başlangıcına gider (1 değeri)

fseek( fp, 5L, 0);      } Dosya başlangıcından itibaren 5 karakter ileri gider (6. karakter pozisyonu).
fseek( fp, 1L, 1 );     } Sonra yine 1 karakter ilerler. (7. karakter yani 7 değeri)

fseek( fp, 5L, 0);      } Dosya başlangıcından itibaren 5 karakter ileri gider (6. karakter pozisyonu).
fseek( fp, -1L, 1 );    } Sonra 1 karakter geri gider. (5. karakter yani 5 değeri)

fseek( fp, -10L, 2 );   } Dosya sonundan itibaren 10 karakter geri gider (1 değeri)
```

Önceki bölümlerde de anlatıldığı gibi bir disk dosyası fopen fonksiyonu ile açılırken fonksiyonun ikinci argümanı olan mod dizgisinde r,w yada a erişim tipleri ile birlikte kullanılan + (artı işareti), dosyanın güncellemek yani hem okumak hem de yazmak amacıyla açılacağını ifade eder.
Okuma ve yazma işlemleri arasında gidip gelirken, dosya pozisyonu belirleyen fonksiyonlar kullanılarak bu iki işlemin birbirinden ayrılması sağlanır.

Argüman olarak bir disk dosyasına ait dosya göstergesini alan rewind fonksiyonu, dosyanın pozisyon göstergesini dosya başlangıcına ayarlar:

```
void rewind( FILE *dosya_gostergesi );      /* stdio.h */
```

rewind fonksiyonu daha önce Örnek 9.1-5 ve 9.1-7'de kullanıldı. rewind çağrısından sonra erişilen karakter dosyada bulunan ilk karakterdir. Bir disk dosyası okuma amacıyla açıldığında bu işlem otomatik olarak yapılır.

Dosyada herhangi bir pozisyona ulaşılmak istendiğinde fseek fonksiyonu kullanılır. Bir veri dosyasında, herhangi bir karakterin pozisyonu yada bir başka deyişle dosya başlangıcından itibaren byte olarak *offset*'i, o karakterin dosyada bulunduğu yeri ifade eder. Buna göre dosyadaki ilk karakter 0 pozisyonunda (offset değeri 0), ikinci karakter ise 1 pozisyonunda (offset değeri 1) bulunur. fseek fonksiyonu 3 argüman alır: bir disk dosyasına ait dosya göstergesi (*dosya_gostergesi*), erişilmek istenen pozisyonun byte offset değeri olarak bir **long** tamsayı değer (*offset*), offset'in nereye göre hesaplanacağını belirten tamsayı (*mod*):

```
int fseek( FILE *dosya_gostergesi, long offset, int mod );      /* stdio.h */
```

mod değeri, offset dosya başlangıcından itibaren ise 0, pozisyon göstergesinin o anda bulunduğu yere göre ise 1, dosya sonuna göre ise 2 olarak verilir. Ayrıca pozitif offset değeri, dosyada ileri gitmek; negatif offset değeri ise dosyada geri gitmek anlamına gelir. Buna göre Şekil 9.4-1'de çeşitli fseek çağrıları verilmiştir.

stdio.h başlık dosyasında *mod* değeri için tanımlanmış sembolik sabitler bulunur: SEEK_SET (0 değeri), SEEK_CUR (1 değeri) ve SEEK_END (2 değeri).

Bir rewind çağrısı ile aşağıdaki fseek çağrısı aynı işi yapar:

```
fseek( fp, 0L, SEEK_SET );
```

Fakat rewind çağrısı, akımın hata göstergesini sıfırlar. Fakat fseek fonksiyonu bu işlemi yapmaz. Her iki fonksiyon da, akımın dosya-sonu göstergesini sıfırlar. Bir diğer fark ise, rewind fonksiyonu hiç bir değer döndürmez. fseek fonksiyonu, hata oluşmaz ise 0 döndürür.
Hata durumunda ise 0 dışında herhangi bir değer döndürür. fseek fonksiyonu text modunda, CR-LF dönüşümlerinden dolayı kısıtlı kullanıma sahiptir. Text modunda yalnızca aşağıdaki fseek kullanımları güvenilir sonuçlar verir:

- *offset* değeri 0 ile, herhangi bir *mod* değerine göre yapılan pozisyon ayarlama işlemi:

```
fseek( fp, 0L, SEEK_SET );      dosya başlangıcına gider
fseek( fp, 0L, SEEK_CUR );      bulunulan pozisyonda kalır
fseek( fp, 0L, SEEK_END );      dosya sonuna gider
```

- Dosya başlangıcından itibaren, daha sonra anlatılacak olan ftell fonksiyonunun döndürdüğü değer kullanılarak yapılan pozisyon belirleme işlemi:

```
Pozisyon = ftell( fp );
fseek( fp, Pozisyon, SEEK_SET );
```

Binary modunda yukarıda bahsedilen kısıtlamalar söz konusu değildir.

ftell fonksiyonu, argüman olarak bir dosya göstergesi alır ve ilgili disk dosyasında o anda bulunulan pozisyonun dosya başlangıcından itibaren byte olarak offset değerini döndürür:

```
long ftell( FILE *dosya_gostergesi );   /* stdio.h */
```

Hatasız çalışır ise dosyanın pozisyon göstergesinin değerini, hata durumunda ise -1L değerini döndürür ve errno değişkenine pozitif bir değer atar. Text modunda döndürülen offset değeri, yapılan dönüşümlerden dolayı hatalı olabilir.

Örnek 9.4-1'de binary modunda yazma ve güncelleme amacıyla oluşturulan veriler dosyasına, bir **int** ve bir **double** değer içeren yapılardan oluşan yapı dizisi Dizi yazılır. Programda daha sonra do-while döngüsü içinde ekrandan girilen kayıt numarasına göre arama yapılır. Bunun için fseek fonksiyonuna argüman olarak aktarılmak üzere offset değeri hesaplanır. Kayıt numaraları 1'den başlar. Fakat offset değerleri 0'dan başlar. Bu nedenle, offset değeri hesaplanırken girilen kayıt numarasının 1 eksiği kullanılır (KayitNo - 1).

Dosyada bulunan herbir kayıt, EL_BOY(Dizi) makrosu ile hesaplanan miktar kadar alan kaplar (yani sizeof(struct KAYIT) ifadesinin döndürdüğü byte sayısı kadar).

Bu nedenle herhangi bir kaydın offset miktarı, kayıt numarasının 1 eksiği ile tekbir yapının byte sayısının çarpımına eşittir:

```
( KayitNo - 1 ) * EL_BOY( Dizi )
```

Bu işlemin sonucunda elde edilen değer **long** tipine çevrildikten sonra, herhangi bir kaydın dosya başlangıcına göre offset değeri saptanmış olur ve bu değer fseek fonksiyonuna 2. argüman olarak aktarılarak istenen pozisyona erişilebilir:

```
Offset = (long) ( (KayitNo - 1) * EL_BOY(Dizi) );
fseek( fp, Offset, SEEK_SET );
```

Kayıt numarası 0 girildiğinde döngüden çıkılır ve fseek çağrısı ile dosya başlangıcına gidilir. Dosya sonuna erişilinceye kadar, i = 4 ve x = 16.4 için dosyada bulunan kayıtlar taranır. Bu değerleri içeren kaydın offset değeri ftell çağrısı ile alınır. Dosya

başlangıcından itibaren byte olarak alınan bu offset değeri tekbir kaydın byte sayısına bölünerek kayıt numarası bulunur:

Offset / EL_BOY(Dizi)

Örnek 9.4-1 ftlfsk.c programı.

```
#include <stdio.h>
#include <stdlib.h>

#define EL_SAYI(s)    (sizeof(s)/sizeof(s[0]))
#define EL_BOY(s)     (sizeof(s[0]))

struct KAYIT
{
   int i;
   double x;
} Dizi[] =   {  { 1, 13.1 },   { 4, 16.4 },   { 3, 15.3 },
                { 4, 16.4 },   { 5, 17.5 },   { 4, 16.4 },
             }, kk;

int main(void)
{
   FILE  *fp;
   long  Offset;
   int     KayitNo;

   if ( (fp = fopen( "veriler", "w+b")) == NULL )
   {
      fprintf( stderr, "VERILER dosyasi acilamadi\n");
      exit(1);
   }
   if ( fwrite( Dizi, sizeof(Dizi), 1, fp ) < 0 )
   {
      fprintf( stderr, "VERILER dosyasina yazilamadi\n");
      exit(2);
   }
   puts("-ftell/fseek-");
   printf( "Kayit No (1-%d) giriniz, cikis icin 0\n", EL_SAYI(Dizi) );
   do
   {
        printf("Kayit No ? ");
        scanf( "%d", &KayitNo );
        if(  ( KayitNo >= 1 ) && ( KayitNo <= EL_SAYI(Dizi) )  )
        {
             Offset = (long)( (KayitNo - 1) * EL_BOY(Dizi) );
             fseek( fp, Offset, SEEK_SET );
             fread( &kk, EL_BOY(Dizi), 1, fp );
             if (  feof(fp)  ||  ferror(fp)  )
```

...Örnek 9.4-1 devam

```
            {
               fprintf( stderr, "VERILER dosyasi okunamadi\n");
               exit(3);
            }
            printf( "\ti:%d, x:%.1f\n", kk.i, kk.x );
         }
   } while( KayitNo );
   fseek( fp, 0L, SEEK_SET );   /* rewind( fp ); */
   printf("i:4 ve x:16.4 icin kayitlarin listelenmesi\n");
   while ( 1 )
   {
         fread( &kk, EL_BOY(Dizi), 1, fp );
         if ( feof(fp) || ferror(fp) )
            break;
         if ( (kk.i == 4) && (kk.x == 16.4) )
         {
            Offset = ftell(fp);
            printf("\tKayit no: %ld --> i=%d, x=%.1f\n",     Offset / EL_BOY(Dizi),
                                                            kk.i, kk.x );
         }
   }
   fclose(fp);
   return 0;
}
```

Çıktı

```
A:\>ftlfsk
-ftell/fseek-
Kayit No (1-6) giriniz, cikis icin 0
Kayit No ? 1
        i:1, x:13.1
Kayit No ? 2
        i:4, x:16.4
Kayit No ? 3
        i:3, x:15.3
Kayit No ? 4
        i:4, x:16.4
Kayit No ? 5
        i:5, x:17.5
Kayit No ? 6
        i:4, x:16.4
Kayit No ? 9
Kayit No ? 0
i:4 ve x:16.4 icin kayitlarin listelenmesi
```

...Örnek 9.4-1 Çıktı devam

```
Kayit no: 2 --> i=4, x=16.4
Kayit no: 4 --> i=4, x=16.4
Kayit no: 6 --> i=4, x=16.4

A:\>
```

Örnek 9.4-2'de, binary modunda açılan veri.dat dosyasına "Bu Bir DENEMEdir." dizgisi yazılır. Disket dolu ise, bu durum fputs çağrısını izleyen if deyiminde bulunan fflush fonksiyonu tarafından saptanır ve ekrana hata mesajı yazılarak programdan çıkılır. Aynı dosya tekrar okuma ve güncelleme amacıyla açılır.

> **NOT:**
> Programda fopen fonksiyonunun ilk argümanı olan dosya isminde fihrist ayracı olarak ters kesme "\" kullanılmıştır ("A:\\VERI.DAT"). Bu dosya ismi, programın yazıldığı DOS işletim sistemi altında şunu ifade eder: A: sürücüsünün kök fihristinde bulunan ve uzantısı DAT olan dosya.
> Programların taşınabilir olması için, dosya isminin ve uzantısının uzunlukları ve içerdiği karakterlerin ve dosyanın bulunduğu yeri ifade eden "path" bilgisinin içerdiği karakterlerin (burada A:\\) işletim sistemine göre değişebileceği dikkate alınarak taşınabilir dosya isimleri kullanılmalıdır.

Programda while döngüsü içinde, dosya sonuna erişilinceye yada herhangi bir hata oluşuncaya kadar çeşitli işlemler gerçekleştirilir. İlk olarak, ftell kullanılarak izleyen fgetc çağrısı ile dosyadan okunacak olan karakterin offset değeri saptanır ve **long** tamsayı Offset değişkenine atanır. fgetc ile okunan karakter tamsayı i değişkenine atanır ve izleyen if deyiminde bu karakterin küçük harf olup olmadığı test edilir. Küçük harf değil ise, fgetc tarafından dosyanın pozisyon göstergesi güncellendiği için, Offset değişkeninin taşıdığı offset değeri kullanarak okuma öncesi pozisyona erişilir ve okunan karakter küçük harfe çevrilerek fputc fonksiyonu ile aynı pozisyona yazılır. Daha sonra izleyen karakter pozisyonuna bir arttırılan offset değerinin argüman olarak kullanıldığı fseek çağrısı ile erişilir.

Örnek 9.4-2 harfcev.c programı.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(void)
{
  FILE  *fp;
  long  Offset;
  int     i;

  if ( (fp = fopen( "A:\\VERI.DAT", "wb" ))   ==  NULL  )
```

...Örnek 9.4-2 devam

```
  {
       fprintf( stderr, "VERI.DAT olusturulamadi\n" );
       exit(1);
  }
  fputs("Bu Bir DENEMEdir.", fp );
  if ( fflush( fp ) )
  {
       fprintf( stderr, "VERI.DAT dosyasina yazilamadi\n" );
       exit(2);
  }
  fclose( fp );
  if ( (fp = fopen( "A:\\VERI.DAT", "rb+" ))  == NULL  )
  {
       fprintf( stderr, "VERI.DAT bulunamadi\n" );
       exit(3);
  }
  while ( 1 )
  {
        Offset = ftell( fp );
        i = fgetc(fp);
        if ( (i == EOF) && (feof(fp) || ferror(fp)) )
             break;
        if ( i != tolower(i) )
        {
             fseek( fp, Offset, SEEK_SET );
             fputc( tolower(i), fp );
             printf("[%04ld] : %c --> %c\n", Offset, i, tolower(i) );
             fseek( fp, Offset+1, SEEK_SET );
        }
        else
             printf("[%04ld] : %c\n", Offset, i );
  }
  return 0;
}
```

Çıktı

```
A:\> type veri.dat
Bu Bir DENEMEdir.

A:\> harfcev
[0000] : B --> b
[0001] : u
[0002] :
[0003] : B --> b
[0004] : i
[0005] : r
```

...Örnek 9.4-2 Çıktı devam

```
[0006] :
[0007] : D --> d
[0008] : E --> e
[0009] : N --> n
[0010] : E --> e
[0011] : M --> m
[0012] : E --> e
[0013] : d
[0014] : i
[0015] : r
[0016] : .
A:\>
A:\> type veri.dat
bu bir denemedir.
```

Çıktıda da görüldüğü gibi program veri.dat dosyası içindeki tüm büyük harfleri küçük harfe çevirir. Bu program, dosyada pozisyon belirleme işleminin hatasız yapılabilmesi için ftell ve fseek fonksiyonlarının birlikte kullanımını göstermek için hazırlanmıştır. Karakterlerin küçük harfe çevirme işlemi basit bir filtre programı (standart girişten okuyan ve standart çıkışa yazan) ile yapılabilir:

```
/* filtre.c */
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int c;
   while ( (c=getchar()) != EOF )
         putchar( tolower(c) );

   return 0;
}
```

Ayrıca komut satırında yönlendirme yapılarak çıktının bir disk dosyasına yazılması sağlanabilir:

```
A:\> filtre < veri.dat > cikti
A:\> type cikti
bu bir denemedir.

      yada

A:\> type veri.dat | filtre > cikti
(  UNIX işletim sisteminde "type" komutunun karşılığı olarak "cat" komutu
kullanılabilir.)
```

fgetpos ve fsetpos fonksiyonları:

fgetpos fonksiyonu, argüman olarak bir disk dosyasına ait dosya göstergesini alır ve bu dosyanın pozisyon göstergesinin değerini **pozisyon*'a (*pozisyon* tarafından işaret edilen bellek alanına) atar:

```
int fgetpos( FILE *dosya_gostergesi, fpos_t *pozisyon );    /* stdio.h */
```

fsetpos fonksiyonu ise, dosyanın pozisyon göstergesini daha önceki fgetpos çağrısı ile **pozisyon*'a atanmış olan değere ayarlar. Ayrıca fsetpos çağrısı, dosya-sonu göstergesini sıfırlar:

```
int fsetpos( FILE *dosya_gostergesi, const fpos_t *pozisyon );    /* stdio.h */
```

Her iki fonksiyon da hatasız çalışır ise 0 değerini, hata durumunda ise 0 harici bir değer döndürür ve errno değişkenine bir pozitif değer atar. Dosyanın pozisyon göstergesinin değerini taşıyan *pozisyon*, stdio.h'de tanımlanmış olan fpos_t veri tipinde bildirilen bir adres değişkenidir.

Aşağıdaki örnekte, Örnek 9.4-1'de de kullanılan yapı dizisinin elemanları teker teker binary modunda yazma ve güncelleme amacıyla oluşturulan veriler dosyasına yazılır. Her bir yapının yazılacağı dosya pozisyonu değeri, yazma işlemi öncesi fpos_t tipinde bildirilen PozDizi dizisinin elemanına fgetpos çağrısı ile aktarılır. Dolayısıyla herhangi bir kayit numarası dizi indeksi olarak kullanılarak erişilen PozDizi elemanı, aynı kaydın pozisyon bilgisini taşır. Programda bu teknik kullanılarak, dosyada bulunan kayıtlar sondan başa doğru listelenir.

Örnek 9.4-3 fgpfsp.c programı.

```
#include <stdio.h>
#include <stdlib.h>

#define EL_SAYI(s)    (sizeof(s)/sizeof(s[0]))
#define EL_BOY(s)     (sizeof(s[0]))

typedef struct
{
  int   i;
  double  x;
} KAYIT;

KAYIT   Dizi[] =  {
                     { 1, 13.1 },   { 4, 16.4 },   { 3, 15.3 },
                     { 4, 16.4 },   { 5, 17.5 },   { 4, 16.4 },
                  };
KAYIT   kk;
```

...Örnek 9.4-3 devam

```
int main(void)
{
   FILE     *fp;
   fpos_t  PozDizi[ EL_SAYI(Dizi) ];
   int        index;
   if (  (fp = fopen( "veriler", "w+b"))  ==  NULL  )
   {
        fprintf( stderr, "VERILER dosyasi acilamadi\n");
        exit(1);
   }
   for ( index = 0; index < EL_SAYI(Dizi); ++index )
   {
        fgetpos( fp, PozDizi+index );
        if ( fwrite( &Dizi[index], EL_BOY( Dizi ), 1, fp ) < 0 )
        {
             fprintf( stderr, "VERILER dosyasina yazilamadi\n");
             exit(2);
        }
    }
   puts("Kayit:   i     x  \n------ ----- -----");
   for ( index = EL_SAYI(Dizi); index > 0; --index )
   {
        fsetpos( fp, &PozDizi[ index-1 ] );
        fread( &kk, EL_BOY(Dizi), 1, fp );
        printf( "%04d     %-3d %-4.1f\n", index, kk.i, kk.x );
   }
   fclose(fp);
   return 0;
}
```

Çıktı

```
A:\>fgpfsp
Kayit: i       x
------  -----   -----
0006    4    16.4
0005    5    17.5
0004    4    16.4
0003    3    15.3
0002    4    16.4
0001    1    13.1
A:\>
```

Programda da görüldüğü gibi, fgetpos tarafından alınan pozisyon değeri (**pozisyon* ile erişilen), herhangi bir offset hesaplamasında kullanılmamalıdır. Pozisyon değeri kapalı bir formatta saklanır ve sadece fsetpos çağrısında kullanılmalıdır. Aynı şekilde fsetpos fonksiyonu da, hesaplanan herhangi bir offset değeri ile değil, sadece fgetpos tarafından alınan bir pozisyon değeri ile çağrılmalıdır. Eğer dosya başlangıcından itibaren byte offset değeri olarak ifade edilen pozisyon bilgisine ihtiyaç var ise ftell fonksiyonu kullanılmalıdır. Herhangi bir byte offsetine gitmek için fseek kullanılmalıdır.

sscanf ve sprintf fonksiyonları:

Örnek programlarda formatlı veri girişi için scanf ve fscanf fonksiyonları, formatlı veri çıkışı için ise printf ve fprintf fonksiyonları kullanıldı. scanf fonksiyonu verileri standart girişten, fscanf ise herhangi bir dosyadan okur. printf fonksiyonu verileri standart çıkışa, fprintf ise herhangi bir dosyaya yazar. sscanf ve sprintf fonksiyonları aynı grupta yer alan bir diğer fonksiyon çiftidir. sscanf fonksiyonu verileri herhangi bir dizgiden (*dizgi*) okur ve ikinci argüman olarak verilen *format* dizgisinde belirtildiği şekilde çevirerek argüman listesinde karşılık gelen değişkenlere atar:

```
int sscanf( const char *dizgi, const char *format, ... );     /* stdio.h */
```

İlk iki argüman olan dizgi adreslerini, "..." ile ifade edilen ve değer atanacak olan değişkenlerin adreslerinden oluşan değişken uzunluktaki argüman listesi izler. *format* dizgisi scanf fonksiyonu ile aynıdır.

sscanf fonksiyonu başarılı olarak okunan ve çevrilerek değişkenlere atanan alan sayısını döndürür. Okuma işlemi tamamlanmadan önce dizgi sonunu belirten boş karaktere rastlanır ise EOF döndürür.

sprintf fonksiyonu da yine bir dizgiye formatlı olarak yazar. "..." ile ifade edilen değişken uzunluktaki argüman listesinde bulunan değişkenlerin değerlerini, *format* dizgisinde (printf format dizgisi ile aynıdır) belirtildiği şekilde karakterlere çevirerek *dizgi* dizgisine atar:

```
int sprintf( char *dizgi, const char *format, ...);    /* stdio.h */
```

Fonksiyon aktarılan karakterlerin sonuna boş karakter ekleyerek dizgiyi sonlandırır. Boş karakter hariç olmak üzere atanan karakter sayısını döndürür. Bu iki fonksiyon dizgi giriş/çıkış fonksiyonları olarak adlandırılabilir.

Aşağıdaki örnek programda, yapı dizisi Dizi'nin elemanlarının değerleri, sprintf fonksiyonu kullanılarak "%-9s %5.2f\n" formatına göre karakter dizisine çevrilerek Buf dizisine aktarılır. Daha sonra Buf karakter dizisi, fputs fonksiyonu kullanılarak text modunda yazma ve güncelleme amacıyla oluşturulan veri.dat dosyasına yazılır.

Aynı işlem, fprintf fonksiyonu kullanılarak aşağıdaki şekilde de yapılabilir:

```
fprintf( fp, "%-9s %5.2f\n",  Dizi[i].c, Dizi[i].d );
```

Örnek 9.4-4 dizgi.c programı.

```
#include <stdio.h>
#include <stdlib.h>

#define EL_SAYI(s)  (sizeof(s)/sizeof(s[0]))

struct KAYIT
{
   char   c[5];
   double d;
}  Dizi[ ]    = {  { "abc", 23.4 }, { "def", 4.56 }, { "xyz", 6.08 },  };

int  main(void)
{
   FILE *fp;
   char cc[5], Buf[BUFSIZ];
   int i;
   double x;

   if (  (fp = fopen( "veri.dat", "w+" ))  ==  NULL )
   {
        fprintf( stderr, "VERI.DAT dosyasi olusturulamadi\n" );
        exit(1);
   }

   for ( i = 0; i < EL_SAYI(Dizi); ++i  )
   {
        sprintf( Buf, "%-9s %5.2f\n", Dizi[i].c,  Dizi[i].d  );
        fputs( Buf, fp );
   }

   rewind(fp);

   while ( fgets( Buf, BUFSIZ, fp ) )
   {
        sscanf( Buf, "%s %lf", cc, &x );
        printf("%-9s %5.2f\n", cc, x );
   }

   fclose(fp);

   return 0;
}
```

Çıktı

```
abc       23.40
def        4.56
xyz        6.08
```

---

rewind fonksiyonu çağrılarak pozisyon göstergesi dosya başlangıcına ayarlanır. veri.dat dosyasında bulunan satırlar ilk olarak fgets fonksiyonu ile Buf dizisine aktarılır. Buf dizisinde bulunan karakterler sscanf fonksiyonu tarafından "%s %lf" formatına göre çevrilerek cc karakter dizisine ve x değişkenine atanır. Aynı işlem, fscanf fonksiyonu kullanılarakta yapılabilir:

```
while (  fscanf( fp, "%s %lf", cc, &x )  !=  EOF )
        printf("%-9s %5.2f\n", cc, x );
```

Programda Buf karakter dizisi bildirilirken eleman sayısı BUFSIZ olarak verildi. BUFSIZ sembolik sabiti, stdio.h'de 256'dan büyük bir tamsayı değer olarak tanımlanmıştır (pek çok derleyicide 512'dir).

Programlarda kullanılmak üzere oluşturulan tampon alanların büyüklüğü Örnek 9.4-4'de olduğu gibi BUFSIZ olarak verilebilir. Aynı zamanda akım giriş/çıkış fonksiyonları tarafından kullanılan tampon alanlar herhangi bir şekilde değiştirilmedikçe (*default* olarak) BUFSIZ büyüklüğündedir. ■