PROGRAMLAMAYA GİRİŞ · 1. DÖNEM

Programlamaya Giriş

Ders notları ve alıştırmalar

0 İlk C programı

1 Değişkenler, tipler, girdi-çıktı

2 Operatörler, koşullar, döngüler

3 Diziler ve karakter dizileri

4 Fonksiyonlar ve değişkenlerin ömrü

5 İşaretçiler

6 Dinamik bellek yönetimi

7 Özyineleme ve fonksiyon işaretçileri

8 Yapılar: struct

9 Union ve bitsel işlemler

10 Bağlı listeler

11 Dosya işlemleri

12 Önişlemci ve kütüphaneler

HEKZON

hekzon.com/dersler

13 bölüm · 38 konu

İçindekiler

Bu notlar, YouTube'daki Programlamaya Giriş dersinin yazılı hâlidir. Her konunun yanındaki zaman, uzun videoda o konunun başladığı yeri gösterir. Her bölümün sonunda sorular var; cevaplarını ve çözümlerini en sonda bulabilirsin. Aynı soruları hekzon.com/dersler adresinde, cevabını hemen göreceğin alıştırmalar olarak da çözebilirsin.

0 **İlk C programı** — 3
Neden 5 satır? · Neden hiç çalışmaz?

1 **Değişkenler, tipler, girdi-çıktı** — 11
Boş değişken · Harf – sayı? · −1 > 1 ?

2 **Operatörler, koşullar, döngüler** — 20
if (x = 5) · −7 % 3 · 5 kez mi? · Tek bir break

3 **Diziler ve karakter dizileri** — 32
Onuncu eleman · 3 harf, 4 bayt · Eşit değil

4 **Fonksiyonlar ve değişkenlerin ömrü**
100 yerde aynı hesap · Fonksiyon bitince? · Kaç kez çağrıldı?

5 **İşaretçiler** — 54
Değer bellekte nerede? · Yer değiştir · p + 1 kaç? · Neden çöktü?

6 **Dinamik bellek yönetimi** — 67
Dizi ne kadar büyük olmalı? · 6 sayılık yer · Bellek nereye gidiyor?

7 **Özyineleme ve fonksiyon işaretçileri**
Kendini çağıran fonksiyon · Beş satır, 331 milyon çağrı · Fonksiyona fonksiyon

8 **Yapılar: struct** — 86
Üç ayrı dizi · Aynı alanlar, iki boy

9 **Union ve bitsel işlemler** — 94
İkisi de var mı? · Lamba yandı, motor durdu · Aynı 4 bayt

10 **Bağlı listeler** — 106
Bir milyon adım · Durmadan 42 · 35 hâlâ orada

11 **Dosya işlemleri** — 118
Kayıp satırlar · Olmayan dosya

12 **Önişlemci ve kütüphaneler** — 128
KARE(2+3) = ? · Tek komut, dört aşama · DLL bulunamadı

Cevaplar ve çözümler — 139

PROGRAMLAMAYA GİRİŞ · 1. DÖNEM

0 İlk C programı

Bu bölümde ilk C programını satır satır tanıyacaksın: C'nin nereden geldiğini, her programda bulunan iskeleti, printf ile ekrana yazmayı ve yorum satırlarını. Sonra yazdığın kodun çalışan bir programa nasıl dönüştüğünü izleyecek, derleme hatası, uyarı ve çalışma zamanı hatası arasındaki farkı göreceksin.

Neden 5 satır?

videoda 0:11

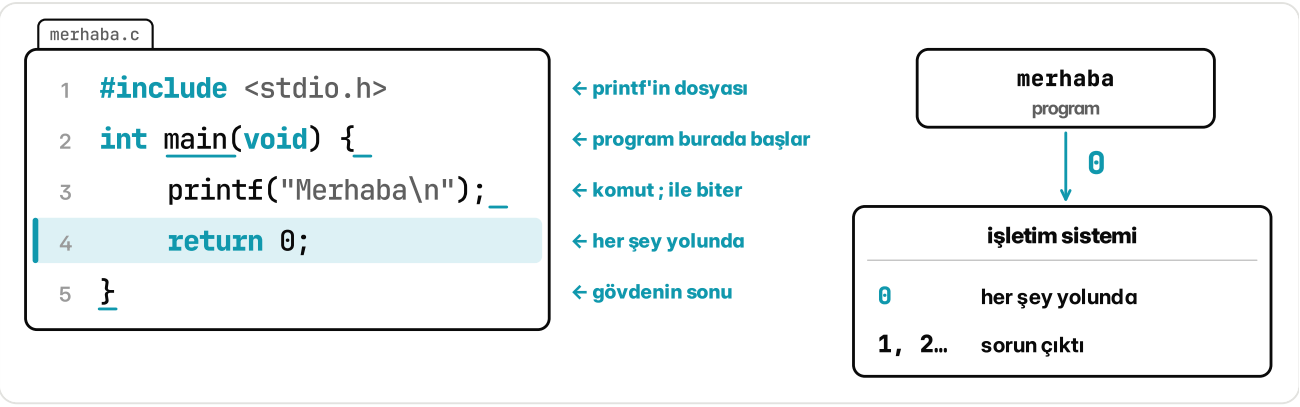
Python'da ekrana Merhaba yazdırmak tek satırlık iştir: print("Merhaba"). C dilinde aynı iş için 5 satır yazarız:

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

Tek satır yazdırmak için neden 5 satır gerekir? Cevabı bu kısmın sonunda; önce dilin kendisine bakalım.

C, 1972'de Dennis Ritchie'nin elinde, Unix işletim sistemini yazmak için doğdu. Bugün de Linux'un ve Windows'un çekirdeği büyük ölçüde C ile yazılır. Çamaşır makinesi, kombi ya da arabanın içindeki küçük bilgisayarlar, yani **gömülü cihazlar** da çoğunlukla C ile programlanır. Python'un kendisi de C ile yazılmıştır. C donanıma yakın ve hızlıdır, ama senin yerine hiçbir şeyi varsaymaz: her şeyi açıkça yazarsın.

Şimdi iskeleti satır satır okuyalım. İlk satırdaki #include, ekrana yazan printf'in tanıtıldığı dosyayı programa ekler; bu dosya stdio.h, yani standart girdi-çıktı dosyasıdır. main, programın başladığı yerdir; her C programında bir main bulunur. Süslü parantezler main'in **gövdesini** açıp kapatır ve gövdedeki her komut noktalı virgülle biter. return 0 programı bitirir ve işletim sistemine "her şey yolunda gitti" der; 0'dan başka bir sayı, bir sorun çıktığını bildirir.



Şekil 0.1 merhaba.c'nin 5 satırı yanlarındaki görevleriyle; program bitince işletim sistemine 0 gönderir, 0 her şey yolunda, 1, 2... sorun çıktı demektir.

printf, tırnak içindekini olduğu gibi yazar ve alt satıra kendiliğinden geçmez. Merhaba ile Dünya'yı iki ayrı printf ile yazdırırsak ekranda bitişik çıkar:

```
printf("Merhaba");
printf("Dünya");
```

Çıktı:

```
MerhabaDünya
```

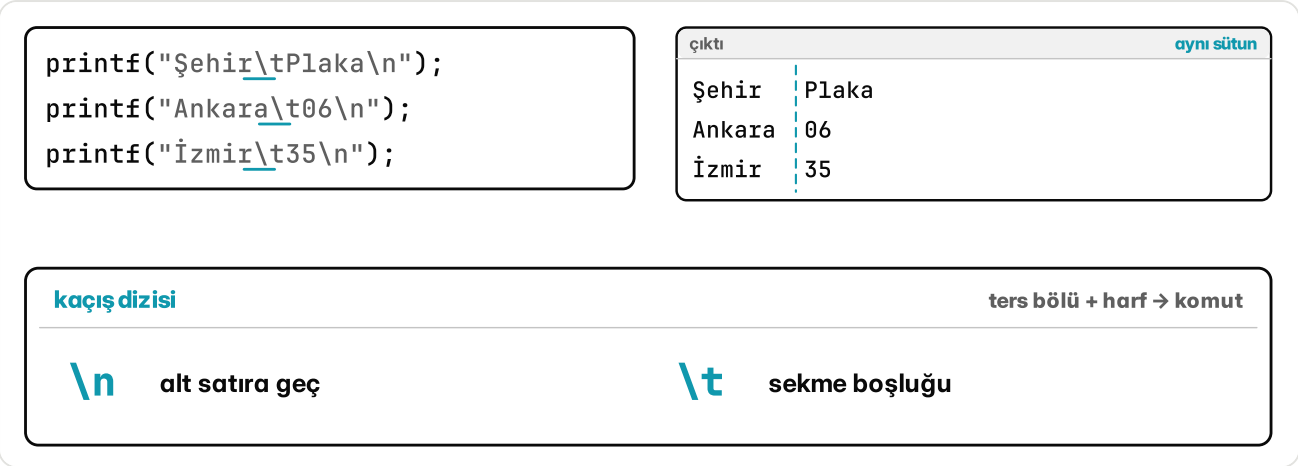
Alt satıra geçmek için tırnağın içine \n yazarız: printf("Merhaba\n"). Ters bölü, arkasındaki harfin yazılmayıp bir komut olarak okunacağını söyler; ters bölü ile harften oluşan bu ikiliye **kaçış dizisi** denir. \n alt satıra geçirir, \t ise bir sekme boşluğu bırakır ve yazıları sütun sütun hizalar:

```
printf("Şehir\tPlaka\n");
printf("Ankara\t06\n");
printf("İzmir\t35\n");
```

Çıktı:

```
Şehir   Plaka
Ankara  06
İzmir   35
```

Terminalde sekme durakları genellikle 8 sütunda birdir. Şehir, Ankara ve İzmir 8 harften kısa olduğu için üçünden sonra gelen yazı da 9. sütundan başlar.



Şekil 0.2 Sekmeli üç printf satırı ve çıktısı: plakalar aynı sütunda; altta \n alt satıra geçirir, \t sekme boşluğu bırakır.

// işaretinden sonrası satır sonuna kadar **yorumdur**; derleyici onu görmezden gelir, programa katmaz. Birden çok satırlık yorum /* ile açılır, */ ile kapanır:

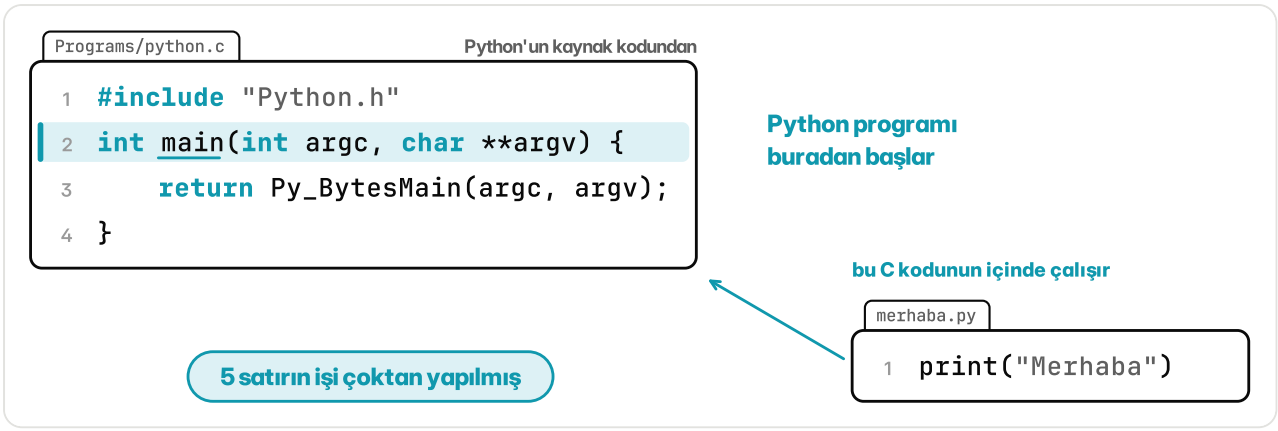
```
#include <stdio.h>
/* İlk programım:
   ekrana selam yazar. */
int main(void) {
    printf("Merhaba\n");  // selam
    return 0;
}
```

Bu programın çıktısı yine yalnızca Merhaba'dır. Yorum, kodu okuyacak insana bırakılmış bir nottur.

Baştaki sorunun cevabı Python'un kendi kaynak kodunda. Python'un C ile yazılmış kaynağında da bir main vardır ve Python programı oradan başlar. Programs/python.c dosyasının Windows dışındaki sistemler için yazılmış kısmı şudur:

```
#include "Python.h"

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

Python'da yazdığın tek satır bu hazır C kodunun içinde çalışır: 5 satırın işini senin yerine Python'u yazanlar çoktan yapmıştır.



Şekil 0.3 Python'un kaynağındaki Programs/python.c: Python programı buradaki main'den başlar, merhaba.py bu C kodunun içinde çalışır.

Neden hiç çalışmaz?

videoda 1:57

Şu programda tek bir noktalı virgül eksik, hem de 5. satırda:

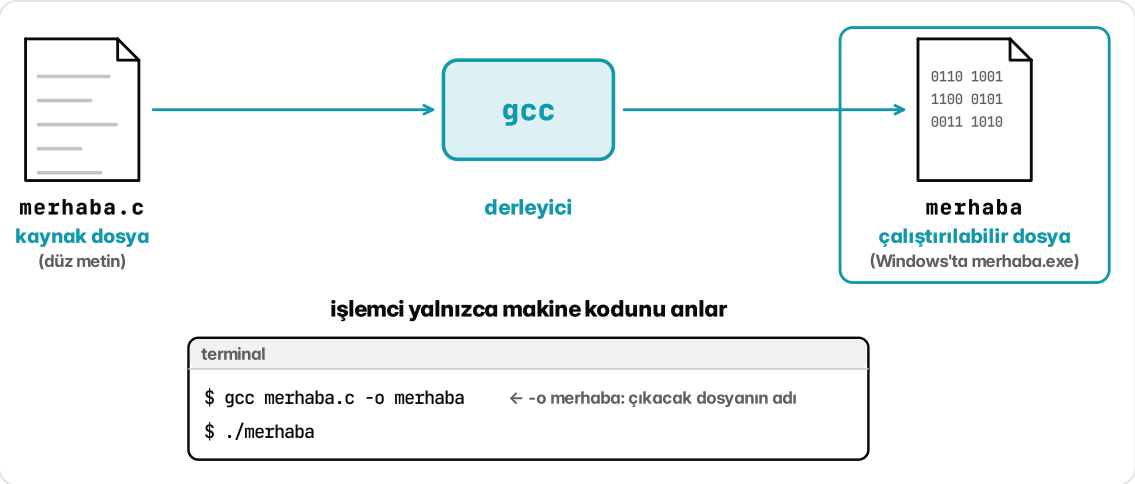
```
#include <stdio.h>

int main(void) {
    printf("Merhaba\n");
    printf("Dünya\n")
    return 0;
}
```

Ama program 4. satırdaki Merhaba'yı bile yazmaz; ekran bomboş kalır. Hata sonraki satırdayken önceki satırlar neden çalışmaz?

Cevap için kodun çalışan bir programa nasıl dönüştüğüne bakalım. Yazdığımız kod, merhaba.c adında düz bir metin dosyasıdır; buna **kaynak dosya** denir. İşlemci bu metni çalıştıramaz, yalnızca makine kodunu anlar. Çeviriyi **derleyici** yapar; en yaygınlarından biri gcc'dir. gcc kaynak dosyayı baştan sona okur, makine koduna çevirir ve bir **çalıştırılabilir dosya** üretir:

```
$ gcc merhaba.c -o merhaba
$ ./merhaba
```

İlk komuttaki -o merhaba, çıkacak dosyanın adını verir; Windows'ta dosya merhaba.exe olur. İkinci komut bu dosyayı çalıştırır. Programı çalıştırdığımızda işleyen kaynak dosya değil, bu yeni dosyadır.

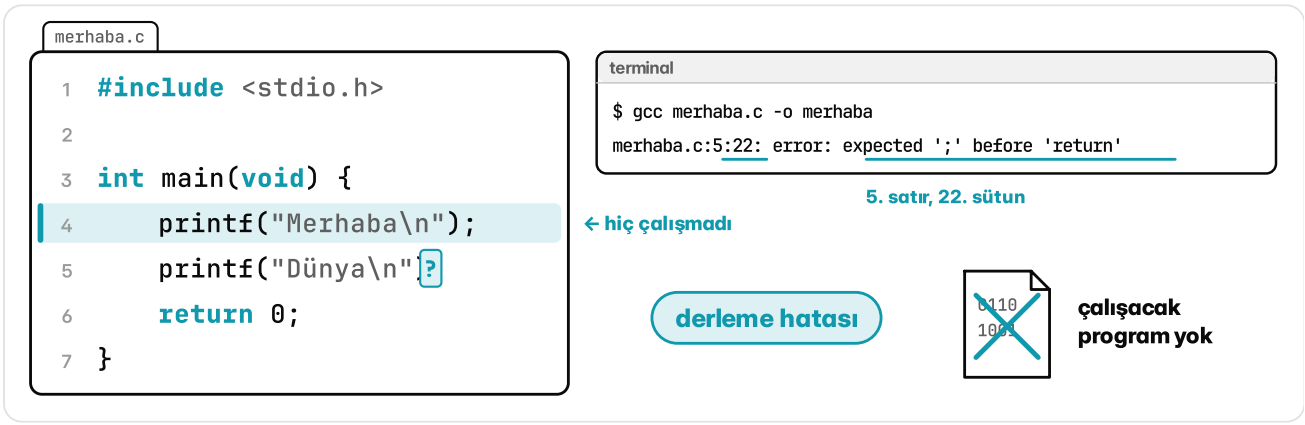


Şekil 0.4 merhaba.c kaynak dosyası gcc derleyicisinden geçer ve makine kodlu çalıştırılabilir merhaba dosyası çıkar; altta terminalde derleme ve çalıştırma komutları.

Noktalı virgül eksik olunca derleyici 5. satırın nerede bittiğini anlayamaz ve durur. Hatanın yerini de söyler:

```
merhaba.c:5:22: error: expected ';' before 'return'
```

5:22, 5. satırın 22. sütunu demektir: orada, return'den önce bir noktalı virgül bekleniyordu. İletinin sözleri derleyiciden derleyiciye değişebilir, ama satır numarası hatanın yerini gösterir. Bu tür hataya **derleme hatası** denir ve derleme hatası olunca çalıştırılabilir dosya hiç üretilmez. Merhaba'nın bile yazılmamasının sebebi budur: ortada çalışacak bir program yoktur.



Şekil 0.5 gcc 5. satırın 22. sütununda noktalı virgül bekliyor; derleme hatası yüzünden çalışacak program yok, 4. satır hiç çalışmadı.

Noktalı virgülü koyalım ve ikinci printf ile 10 / 0'ın sonucunu yazdıralım:

```
printf("Merhaba\n");
printf("%d\n", 10 / 0);
```

Buradaki %d, sonucun o yere tam sayı olarak yazılacağını söyler. Derleyici bu kez durmaz ama bir uyarı verir:

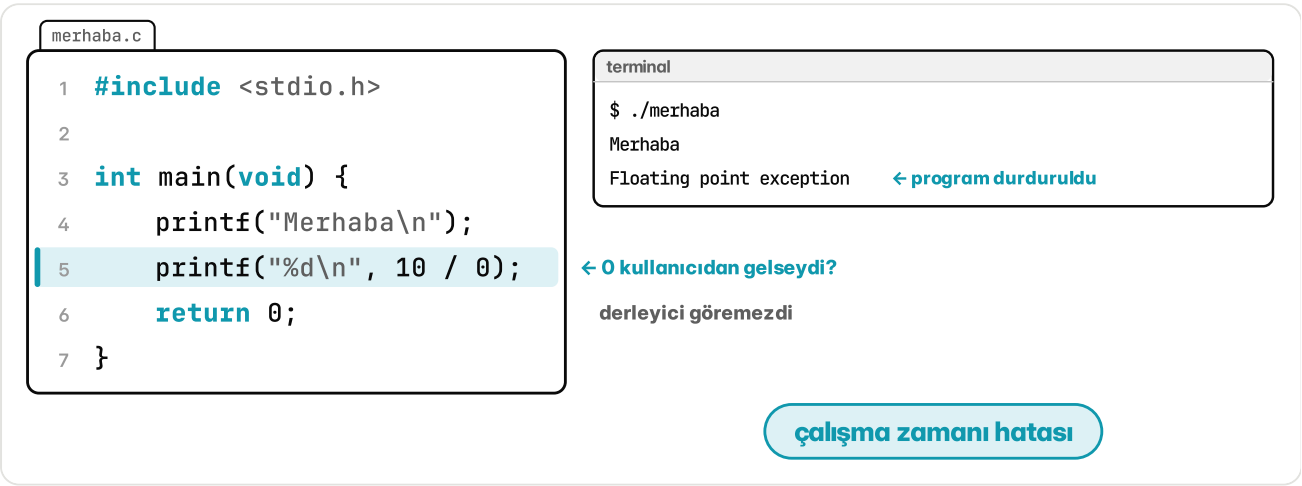
```
merhaba.c:5:23: warning: division by zero [-Wdiv-by-zero]
```

Uyarı, kodun kurallara uygun ama büyük olasılıkla yanlış olduğunu söyler; çalıştırılabilir dosya yine de üretilir.

Çalıştıralım: önce Merhaba yazılır, sonra sıra bölmeye gelir. Sıfıra bölmenin sonucu yoktur; çoğu bilgisayarda işletim sistemi programı o satırda durdurur. Linux'ta terminal şunu gösterir:

```
$ ./merhaba
Merhaba
Floating point exception
```

Program çalışırken ortaya çıkan bu tür hataya **çalışma zamanı hatası** denir. Burada 0 koda yazılıydı, derleyici görüp uyardı; sayı kullanıcıdan gelseydi derleyici bunu hiç göremezdi.



Şekil 0.6 Program Merhaba'yı yazdıktan sonra 5. satırdaki 0'a bölmede durduruluyor: çalışma zamanı hatası.

Telefonunda bir uygulama aniden kapandığında da çoğu zaman olan budur: derleyiciden sorunsuz geçmiş bir program, çalışırken bir hataya çarpmıştır.

Özet

- C, 1972'de Unix'i yazmak için doğdu; işletim sistemi çekirdeklerinde, gömülü cihazlarda ve Python'un kendisinde kullanılır.
- Her C programı main'den başlar; #include ile eklenen stdio.h printf'i tanıtır, her komut noktalı virgülle biter, return 0 işletim sistemine başarıyı bildirir.
- printf alt satıra kendiliğinden geçmez; \n alt satıra geçirir, \t sekme boşluğu bırakır. Ters bölüyle başlayan bu ikililere kaçış dizisi denir.
- // ile başlayan ve /* */ arasındaki yorumlar programa girmez.
- Derleyici kaynak dosyayı makine koduna çevirip çalıştırılabilir dosya üretir; derleme hatasında dosya hiç üretilmez, uyarıda üretilir.
- Çalışma zamanı hatası program çalışırken çıkar; tam sayıyı 0'a bölmek çoğu bilgisayarda programı durdurur.

Sorular

1 **Her C programında bulunan ve programın çalışmaya başladığı yer hangisidir?**

A main
B printf
C #include <stdio.h>
D return 0

2 **printf("Merhaba"); printf("Dünya"); satırları ekrana ne yazar?**

A Merhaba Dünya
B Merhaba ve alt satırda Dünya
C MerhabaDünya
D Hiçbir şey; derleme hatası verir.

3 **Şu iki satır ekrana kaç satır yazar?**

```
printf("A\nB\n");
printf("C");
```

Cevap: _____

4 **main'in sonundaki return 0 satırı ne yapar?**

A Ekrana 0 yazar.
B Programı en baştan yeniden başlatır.
C main'in gövdesini açar.
D Programı bitirir ve işletim sistemine her şeyin yolunda gittiğini bildirir.

5 **Şu satır ekrana ne yazar?**

```
printf("Merhaba\n");  // selam
```

Cevap: _____

6 **5. satırında tek bir noktalı virgül eksik olan programda, 4. satırdaki printf("Merhaba\n"); neden hiç çalışmaz?**

A Program 4. satırı çalıştırır ama ekran hemen silinir.

B Derleme hatası yüzünden çalıştırılabilir dosya hiç üretilmez; çalışacak bir program yoktur.

C printf yalnızca programın son satırında çalışır.

D İşletim sistemi programı 5. satıra gelince durdurur.

7 **printf("%d\n", 10 / 0); satırı için derleyici "division by zero" uyarısı veriyor. Ardından ne olur?**

A Çalıştırılabilir dosya üretilmez.

B Program ekrana 0 yazar ve sorunsuz biter.

C Uyarı yüzünden program çalıştırılamaz.

D Dosya üretilir; çalıştırınca çoğu bilgisayarda program bu satırda durdurulur.

8 **Bir program kullanıcının girdiği sayıya bölme yapıyor. Derleyiciden uyarısız geçti, ama kullanıcı 0 girince program kapandı. Bu ne tür bir hatadır?**

A Derleme hatası: çalıştırılabilir dosya hiç üretilmemiştir.

B Uyarı: kod kurallara uygun ama büyük olasılıkla yanlıştır.

C Çalışma zamanı hatası: program çalışırken bir hataya çarpmıştır.

D Kaçış dizisi hatası: \n yanlış yere yazılmıştır.

PROGRAMLAMAYA GİRİŞ · 1. DÖNEM

1 Değişkenler, tipler, girdi-çıktı

Bu bölümde değişken tanımlamayı, değişkene klavyeden ya da sabit olarak değer vermeyi, C'nin veri tiplerini ve printf'in biçim belirteçlerini öğreneceksin. Yol boyunca üç soruya cevap arayacağız: boş bir değişken ekrana ne yazar, bir harften sayı nasıl çıkarılır ve −1 nasıl 1'den büyük çıkar?

Boş değişken

videoda 3:30

Bir değişken açıp içine hiçbir şey koymadan ekrana yazdıralım:

```
int puan;
printf("%d\n", puan);
```

0 mı çıkar? C'de çoğu zaman anlamsız bir sayı çıkar; bu örnekte 418545520 çıktı ve bir sonraki çalıştırmada başka bir sayı çıkabilir. Bu sayı nereden geliyor?

C'de bir **değişkeni** kullanmadan önce tanımlarız: önce tipini, sonra adını yazarız. int puan dediğimizde derleyici bellekte bir tam sayı sığacak kadar yer ayırır (bugünkü bilgisayarlarda 4 bayt) ve o yere puan adını verir. printf'teki %d, puan'ın değerinin o yere tam sayı olarak yazılacağını söyler. Tanımlarken yanına bir değer de yazarsak değişken o değerle doğar; buna **ilk değer** denir:

```
int puan = 0;
```



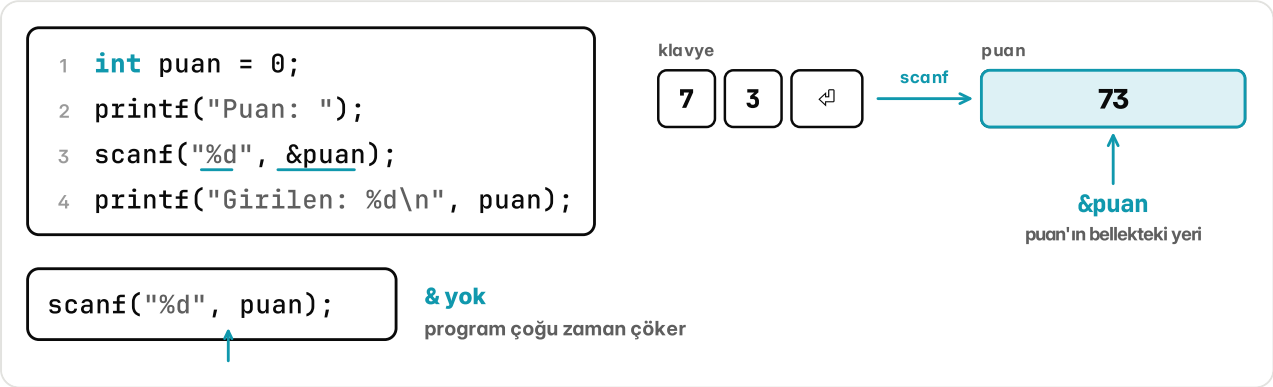
Şekil 1.1 int puan = 0; satırında tip, ad ve ilk değer; bellekte puan'a ayrılan 4 göz sıfırla dolu.

Peki main gibi bir fonksiyonun içindeki değişkene ilk değer vermezsek? C ayırdığı yeri temizlemez. O bellek gözleri az önce başka bir iş için kullanıldıysa, oradan kalan bitler olduğu gibi durur ve printf onları bir tam sayı olarak okur. Ekrana çıkan sayı o artıktır; buna **çöp değer** denir. C bunu hız için yapar: nasılsa üstüne yazılacak bir yeri sıfırlamakla vakit kaybetmez. Bu yüzden kural basit: değişkene ilk değerini hep kendin ver.

Değişkenin değeri klavyeden de gelebilir. scanf, kullanıcının yazdığı sayıyı okur ve değişkene koyar. Neyi okuyacağını, printf'te olduğu gibi %d gibi bir **biçim belirteci** söyler:

```
int puan = 0;
printf("Puan: ");
scanf("%d", &puan);
printf("Girilen: %d\n", puan);
```

Kullanıcı 73 yazıp Enter'a basarsa ekranda Puan: 73 ve altında Girilen: 73 görünür. Değişkenin adının önüne & işareti koyarız, çünkü scanf değişkenin değerini değil, bellekteki yerini ister: okuduğu sayıyı o yere kendisi yazar. Bu işareti unutmak, yani scanf("%d", puan) yazmak en sık yapılan hatalardan biridir; program çoğu zaman çöker.



Şekil 1.2 Klavyeden girilen 7 ve 3'ü scanf okuyup &puan ile verilen yere yazar; & unutulursa program çoğu zaman çöker.

Hiç değişmeyecek bir değer için iki yol var:

```
#include <stdio.h>
#define HAFTA 7

int main(void) {
    const double PI = 3.14159;
    int gun = 3 * HAFTA;
    printf("%d %.2f\n", gun, PI);
    return 0;
}
```

Çıktı:

```
21 3.14
```

Tipin önüne const yazarsak değişken salt okunur olur: PI = 3; gibi onu sonradan değiştirmeye kalkan satırda derleyici hata verir (gcc: assignment of read-only variable 'PI'). #define ise bir ada bir değer bağlar: program derlenmeden önce koddaki her HAFTA yazan yere 7 konur, yani derleyici 3 * 7 satırını görür. Buradaki double kesirli sayı tipidir; %.2f onu noktadan sonra 2 basamakla yazar. İkisine bu bölümün ilerisinde döneceğiz.

sabit.c

```c
#include <stdio.h>
#define HAFTA 7

int main(void) {
    const double PI = 3.14159;
    int gun = 3 * HAFTA;
    return 0;
}
```

ilk değerle `int puan = 0;`

klavyeden `scanf("%d", &puan`

sabit olarak `const · #define`

Şekil 1.3 sabit.c'de const ve #define; yanda bir değişkeni doldurmanın üç yolu: ilk değerle, klavyeden, sabit olarak.

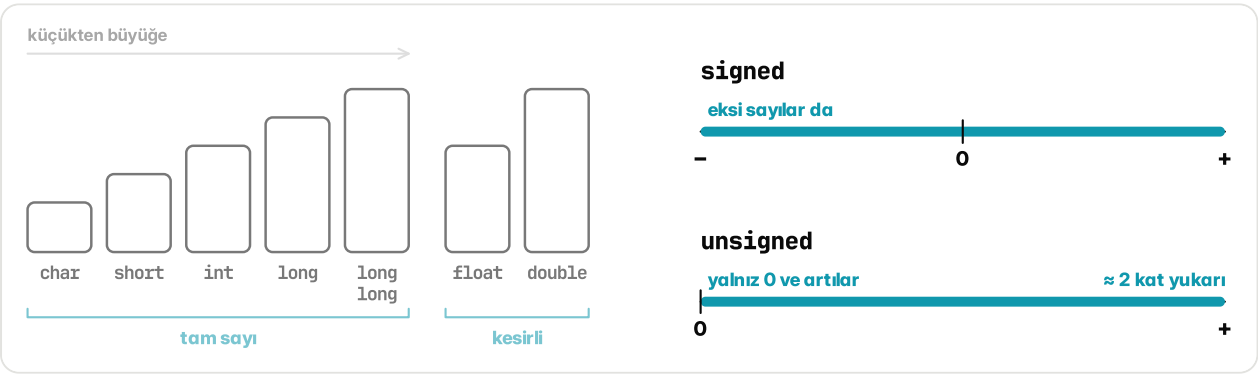
Kısacası C sana yalnız bir yer verir, içini sen doldurursun: ilk değerle, klavyeden ya da sabit olarak.

Harf – sayı?

videoda 5:02

C'de bir harfi büyük harfe çevirmek için hazır bir komuta gerek yok; tek bir çıkarma yeter. Bir harften sayı nasıl çıkarılır? Cevap C'nin tiplerinde.

C'nin tam sayı tipleri küçükten büyüğe char, short, int, long ve long long'dur; kesirli sayılar için de float ve double vardır. Yani char da bir tam sayı tipidir, hem de en küçüğü. Tam sayı tiplerinin önüne signed ya da unsigned yazılabilir. signed, yani **işaretli** tip eksi sayıları da tutar. unsigned, yani **işaretsiz** tip yalnız 0 ve artıları tutar; eksilere ayrılan değerleri de artılara kullandığı için artı yönde yaklaşık 2 kat yükseğe çıkar.



Şekil 1.4 C'nin tipleri küçükten büyüğe; signed tip 0'ın iki yanını, unsigned tip yalnız 0 ve artıları tutar ve yaklaşık 2 kat yukarı çıkar.

Peki int kaç bayt? C bunu kesin söylemez, yalnız alt sınır koyar: int en az 16 bit, long en az 32 bit, long long en az 64 bit olmalıdır. Bugünkü bilgisayarlarda int 4 bayttır, ama küçük bir denetleyici kartında (örneğin Arduino Uno'da) 2 bayt olabilir. long ise 64 bitlik Windows'ta 4, 64 bitlik Linux'ta 8 bayttır. Boy, derleyiciye ve makineye göre değişir.



Şekil 1.5 Tiplerin C'deki alt sınırı ve bugünkü bilgisayardaki boyu: int en az 16 bit, long en az 32 bit, long long en az 64 bit.

Bu yüzden boyu tahmin etmeyiz, sorarız: sizeof, bir tipin ya da değişkenin kaç bayt tuttuğunu verir.

```
printf("%zu\n", sizeof(int));
printf("%zu\n", sizeof(long));
printf("%zu\n", sizeof(char));
```

%zu, sizeof'un sonucu için kullanılan biçim belirtecidir. Bu üç satır 64 bitlik Linux'ta 4, 8 ve 1; 64 bitlik Windows'ta 4, 4 ve 1 yazar. Yalnız char'ın boyu her makinede aynıdır: 1 bayt.

char aslında 1 baytlık küçük bir tam sayıdır. Tırnak içinde 'h' yazınca derleyici yerine h'nin kodunu, 104'ü koyar. Aynı değişkeni printf ile %c belirteciyle yazdırırsak harf olarak h, %d ile yazdırırsak sayı olarak 104 görürüz. Bellekte değişen bir şey yoktur, bayt hep 01101000'dır; yalnız okuma biçimi değişir.

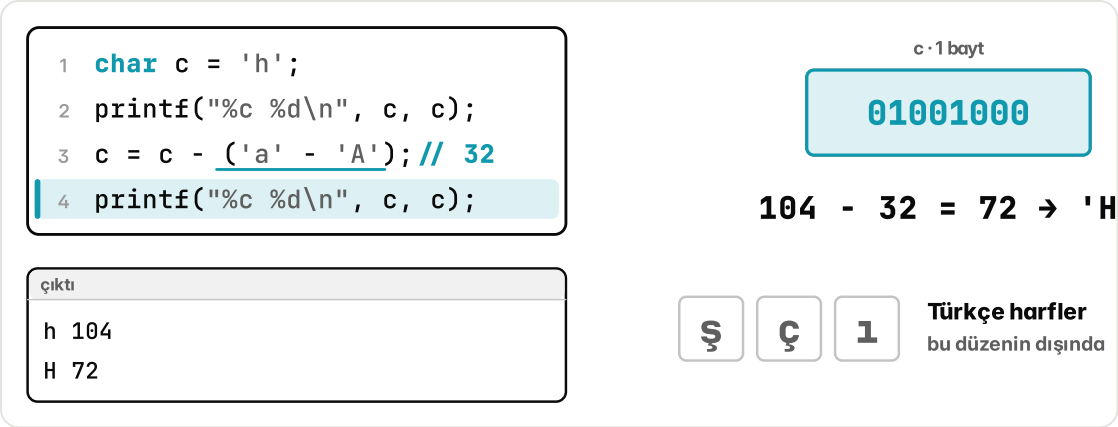
Kodlamada her küçük harf, büyüğünden aynı uzaklıktadır: 'a' ile 'A' arasındaki fark, 'h' ile 'H' arasındakiyle aynıdır. Bu farkı ezberlemeyiz, derleyiciye hesaplatırız: 'a' − 'A' = 32.

```
char c = 'h';
printf("%c %d\n", c, c);
c = c - ('a' - 'A');
printf("%c %d\n", c, c);
```

Çıktı:

```
h 104
H 72
```

104'ten 32 çıkınca 72 kalır, yani 'H'. İki bayt yalnız 32'yi taşıyan bitte ayrılır: 01101000 ve 01001000. Bu yöntem yalnız İngiliz alfabesindeki 26 harfte işler; ş, ç, ı gibi Türkçe harfler bu düzenin dışındadır.



Şekil 1.6 c değişkeninin baytı 01001000 oldu: 104 − 32 = 72, yani 'H'; çıktı h 104 ve H 72. Türkçe harfler bu düzenin dışında.

−1 > 1 ?

videoda 6:43

Şu program −1 ile 1'i karşılaştırıyor:

```
int a = -1;
unsigned int b = 1;
if (a < b) {
    printf("küçük\n");
} else {
    printf("büyük\n");
}
```

if, parantezindeki koşul doğruysa ilk bloğu, değilse else'ten sonraki bloğu çalıştırır. Program ekrana büyük yazar. Derleyici hata vermez, program da bozuk değildir. C'de −1 nasıl 1'den büyük çıkabilir?

Cevap için önce printf'in bir sayıyı nasıl yazdığına bakalım. Ona neyi nasıl yazacağını % ile başlayan biçim belirteçleri söyler:

- %d işaretli tam sayı, %u işaretsiz tam sayı yazar.
- %f kesirli sayıyı noktadan sonra 6 basamakla yazar; araya .2 koyup %.2f yazarsak 2 basamağa yuvarlar.
- %c tek karakter, %s metin yazar.

Örneğin x = 72.456 iken %f 72.456000, %.2f 72.46 yazar.



Şekil 1.7 Biçim belirteçleri tablosu: %d, %u, %f, %.2f, %c ve %s, her biri örnek çıktısıyla.

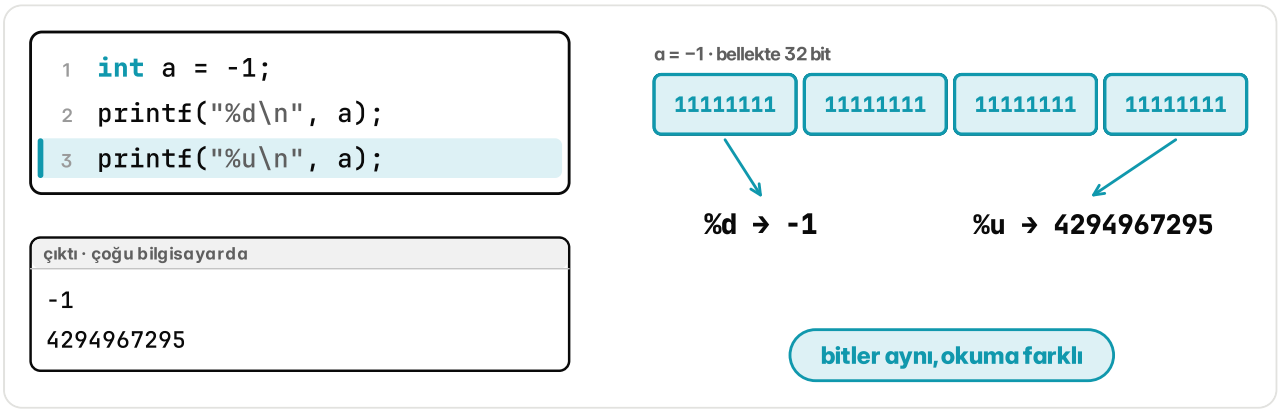
Belirteci yanlış seçersek printf bunu fark etmez, bitleri söylediğimiz gibi okur. −1 bellekte 32 tane 1 ile durur:

```
int a = -1;
printf("%d\n", a);
printf("%u\n", a);
```

Çıktı (çoğu bilgisayarda):

```
-1
4294967295
```

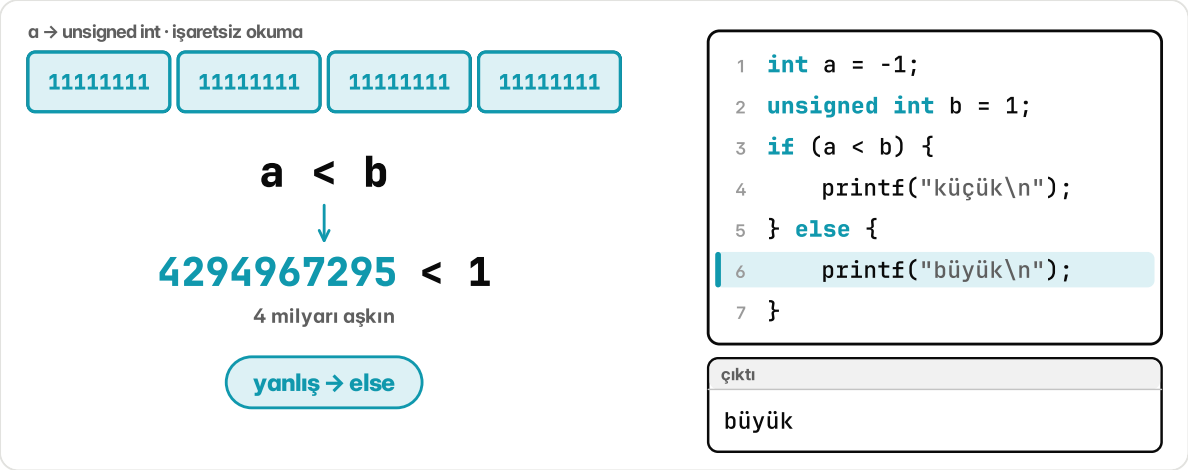
%d ile okursak −1, %u ile okursak 4 milyarı aşkın bir sayı görürüz: bitler aynı, okuma farklı. 32 bitin hepsi 1 olunca işaretsiz değer $2^{32} - 1 = 4.294.967.295$'tir.



Şekil 1.8 a = −1'in 32 biti de 1; %d ile okununca −1, %u ile okununca 4294967295.

Karşılaştırmada da aynısı olur. C iki farklı tipi doğrudan işleme sokmaz, önce ikisini ortak bir tipe çevirir. Kural kabaca şöyledir: küçük tip büyüğe, tam sayı kesirliye çevrilir. char ile int toplanırken char int'e, int ile double toplanırken int double'a çevrilir; int ile unsigned int karşılaşırsa int işaretsize çevrilir. Bu çevirme sessizce yapılır, kodda hiçbir iz bırakmaz.

Programımızda a < b karşılaştırılırken a işaretsize çevrilir. Bitleri aynı kalır ama artık işaretsiz okunur: 4.294.967.295. Karşılaştırma aslında 4.294.967.295 < 1 diye yapılır; sonuç yanlıştır, else çalışır ve ekrana büyük yazılır.



Şekil 1.9 a işaretsize çevrilince a < b karşılaştırması 4294967295 < 1 olur; koşul yanlış, else çalışır ve çıktı büyük.

Bu tuzak gerçek kodda sık çıkar, çünkü bir şeyin boyunu veren sizeof da işaretsiz bir sayı verir. if (-1 < sizeof(int)) koşulu da yanlış çıkar ve program yine büyük yazar: −1, sizeof(int)'ten bile büyük sayılır. Çözüm basit: işaretli ve işaretsiz sayıyı aynı işlemde karıştırma. gcc bu karşılaştırmada varsayılan ayarlarla bir şey demez; -Wall -Wextra ile derlersen uyarır. Bu uyarıyı açık tut.

Özet

- C'de değişken önce tipi, sonra adıyla tanımlanır; tanımlarken verilen değere ilk değer denir.
- Fonksiyonun içindeki değişkene ilk değer verilmezse önceki işten kalan bitler okunur: çöp değer.
- scanf değişkenin bellekteki yerini ister, bu yüzden adın önüne & konur; const salt okunur değişken yapar, #define derlemeden önce adın yerine değeri koyar.
- C tiplerin boyuna yalnız alt sınır koyar; boyu sizeof ile sorarız, char her makinede 1 bayttır.
- char 1 baytlık bir tam sayıdır: 'h' 104'tür ve 'h' − ('a' − 'A') = 72, yani 'H'.
- int ile unsigned int karşılaşınca int işaretsize çevrilir; −1, 4.294.967.295 olur ve 1'den büyük çıkar.

Sorular

1 **main içinde int puan; yazılıyor ve puan'a ilk değer verilmeden printf("%d\n", puan); çalıştırılıyor. Ekranda ne görülür?**

A Her zaman 0

B Çoğu zaman anlamsız bir sayı; bir sonraki çalıştırmada başka bir sayı çıkabilir.

C Her zaman −1

D Hiçbir şey; program derlenmez.

2 **Şu satırdaki hata nedir?**

```
scanf("%d", puan);
```

A %d yerine %u yazılmalı.

B Tırnağın içinde \n eksik.

C puan'ın önünde & yok; scanf değişkenin bellekteki yerini ister.

D scanf'ten önce puan'a ilk değer verilmemiş.

3 **Şu satırlar ekrana ne yazar?**

```
#define HAFTA 7
int gun = 3 * HAFTA;
printf("%d\n", gun);
```

Cevap:

4 **Hangi tipin boyu her makinede aynıdır?**

A int: 4 bayt

B long: 8 bayt

C long: 4 bayt

D char: 1 bayt

5 **Şu satır ekrana ne yazar?**

```
printf("%d\n", 'a' - 'A');
```

Cevap:

6 **double x = 72.456; iken printf("%.2f\n", x); ekrana ne yazar?**

A 72.46

B 72.456000

C 72.45

D 72

7 **int'in 32 bit olduğu bir bilgisayarda şu satırlar ne yazar?**

```
int a = -1;
printf("%u\n", a);
```

Cevap: _____

8 **Şu program neden "büyük" yazar?**

```
int a = -1;
unsigned int b = 1;
if (a < b) printf("küçük\n");
else printf("büyük\n");
```

A unsigned int eksi değer tutamadığı için b, −1 olur.

B Karşılaştırmada a işaretsize çevrilir ve 4.294.967.295 olarak okunur.

C if eşitliği sorar; a ile b eşit olmadığı için else çalışır.

D Derleyici karşılaştırmayı ters çevirir.

PROGRAMLAMAYA GİRİŞ · 1. DÖNEM

2 Operatörler, koşullar, döngüler

Bu bölümde C'nin koşulları nasıl değerlendirdiğini, bölme ve kalan işlemlerini, kısa yazımlı operatörleri, üç döngü yazımını ve switch ile break'i öğreneceksin. Her kısım C'de sık yapılan bir hatadan yola çıkar: hep doğru çıkan bir if, eksi çıkan bir kalan, bir kez dönen bir döngü ve yanlış yerden çıkan bir break.

if (x = 5)

videoda 8:14

Şu programda x 0, ama if ekrana x beş yazar:

```
int x = 0;
if (x = 5)
    printf("x beş\n");
printf("x şimdi %d\n", x);
```

Çıktı:

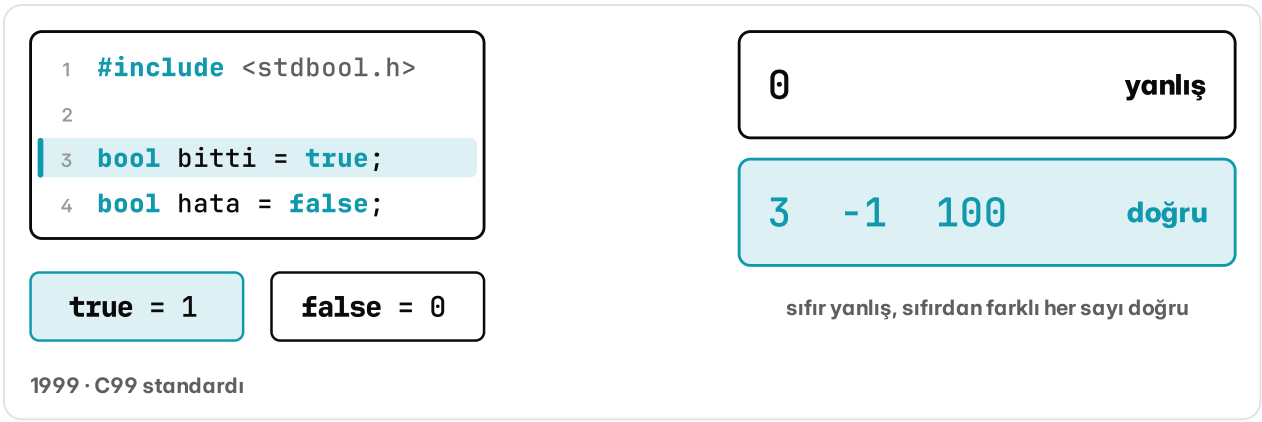
```
x beş
x şimdi 5
```

x 100 olsa da −1 olsa da yine yazar. Bu koşul neden hep doğru çıkar?

Cevap, C'nin doğruyu nasıl tuttuğunda saklı. C'de koşul bir sayıdır: 0 yanlış demektir, 0'dan farklı her sayı doğru. if içine 3 yazarsak çalışır, 0 yazarsak çalışmaz, −1 yazarsak yine çalışır:

```
if (3)  printf("üç\n");          /* yazar */
if (0)  printf("sıfır\n");        /* yazmaz */
if (-1) printf("eksi bir\n");     /* yazar */
```

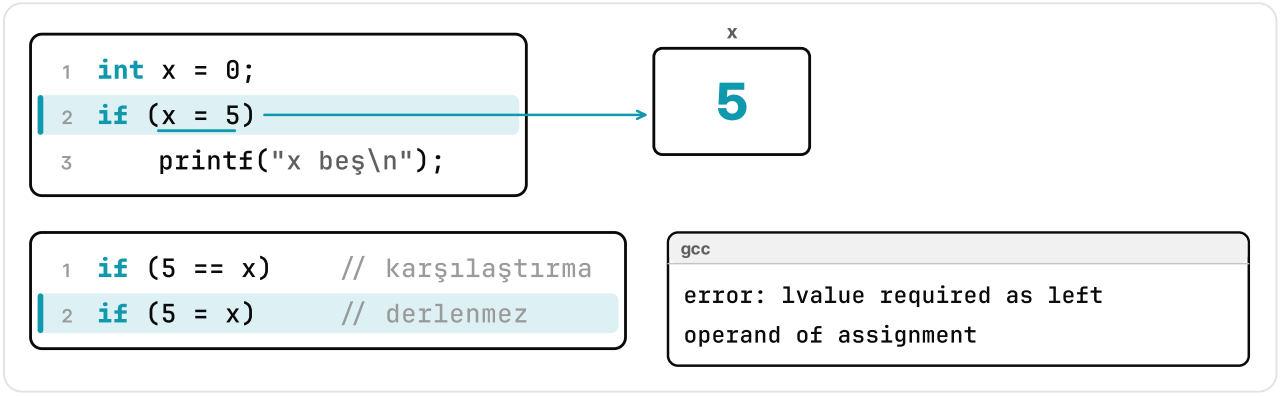
Karşılaştırma da bir sayı üretir: 7 > 2 ifadesinin değeri 1, 2 > 7'ninki 0'dır; printf("%d\n", 7 > 2) ekrana 1 yazar. 1999'dan beri (C99 standardı) stdbool.h dosyasıyla gelen bool tipi ve true, false sözcükleri de var; onlar da 1 ve 0'ın başka adıdır.



Şekil 2.1 0 yanlış, 3, −1 ve 100 doğru: sıfırdan farklı her sayı doğrudur; stdbool.h'deki true 1, false 0'dır.

Şimdi if (x = 5) satırına dönelim. Tek eşittir karşılaştırmaz, **atar**: x değişkenine 5 koyar. Üstelik C'de atamanın kendisi de bir değerdir: konan sayı, yani 5. Koşul 5 olur; 5 sıfır değildir, yani doğru. x önceden ne olursa olsun fark etmez ve satırdan sonra x de 5'tir. Karşılaştırmak için çift eşittir yazılır: if (x == 5).

Bu hata o kadar sık yapılır ki gcc, uyarılar açıkken (-Wall) koşulun içinde atama görünce uyarır: suggest parentheses around assignment used as truth value. Bazı yazılımcılar sabiti sola yazar: if (5 == x). Yanlışlıkla tek eşittir yazılırsa, yani if (5 = x) olursa, 5'e değer atanamayacağı için derleme durur.



Şekil 2.2 Sabit solda yazılınca if (5 == x) karşılaştırır; yanlışlıkla yazılan if (5 = x) ise derlenmez ve gcc hata verir.

C'de "ve" çift ve işaretiyle (&&), "veya" çift dik çizgiyle (||) yazılır. && işleminde ilk koşul yanlışsa sonuç zaten yanlıştır; C ikinci koşulu hiç hesaplamaz. || işleminde de ilk koşul doğruysa ikinciye bakılmaz. Buna **kısa devre** denir. Kısa devre işe de yarar:

```
int a = 10, b = 0;
if (b != 0 && a / b > 2)
    printf("büyük\n");
printf("bitti\n");          /* çıktı: bitti */
```

!= "eşit değil" demektir. b 0 olduğu için ilk koşul yanlıştır, bölme hiç yapılmaz ve ekrana yalnız bitti yazılır. Sıra ters yazılırsa, if (a / b > 2 && b != 0), önce bölme yapılır. Tam sayıyı sıfıra bölmek **tanımsız davranıştır**, yani C standardı ne olacağını söylemez; çoğu bilgisayarda program çöker.



Şekil 2.3 b != 0 önde olunca bölme hiç yapılmaz ve çıktı bitti olur; sıra ters olunca Linux'ta program Floating point exception ile durur.

Koşulun sayı olmasının bir kullanımı daha var: **koşul operatörü**. Soru işaretinden önce koşul, sonra iki nokta ile ayrılmış iki değer gelir; koşul doğruysa ilki, yanlışsa ikincisi seçilir:

```
int x = -7;
int mutlak = (x < 0) ? -x : x;
printf("%d\n", mutlak);    /* 7 */
```

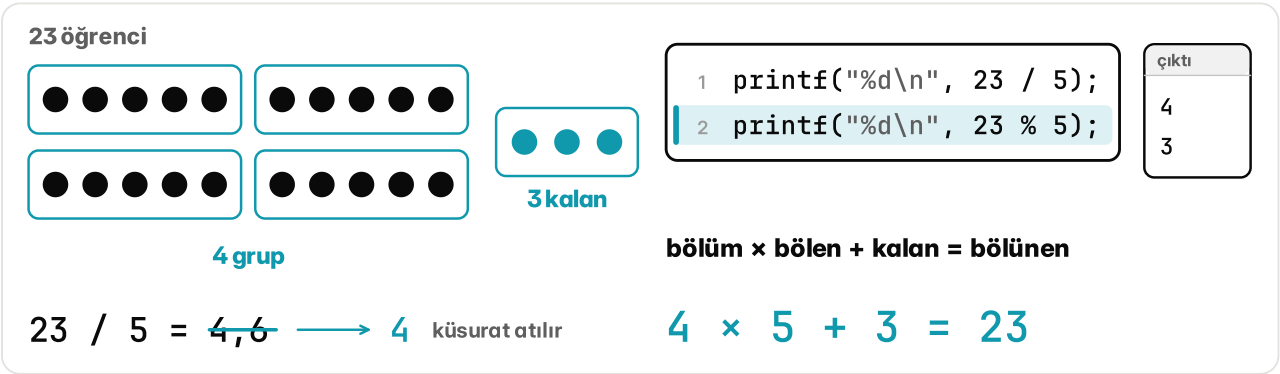
x 0'dan küçük olduğu için −x, yani 7 seçilir: mutlak değer tek satırda bulunur. Kısacası C'de her koşul bir sayıdır; if de, kısa devre de, koşul operatörü de yalnızca şuna bakar: sıfır mı, değil mi?

−7 % 3

videoda 10:04

−7'nin 3'e bölümünden kalan kaçtır? Matematik dersinde cevap 2'dir, çünkü −7 = 3 × (−3) + 2. C'de ise printf("%d\n", -7 % 3) ekrana −1 yazar. C kalanı başka bir kurala göre bulur.

Önce pozitif sayılarla bakalım: 23 öğrenci beşerli gruplara ayrılıyor. C'de iki tam sayının bölümü yine tam sayıdır, küsurat atılır: 23 / 5 işleminin sonucu 4,6 değil 4'tür, yani 4 grup. Kalanı yüzde işareti verir; bu işleme **mod** denir: 23 % 5 = 3, yani 3 öğrenci gruba giremez. İkisini bir kural bağlar: bölüm × bölen + kalan = bölünen. Burada 4 × 5 = 20, artı 3, 23.



Şekil 2.4 23 öğrenci 5'erli 4 gruba ayrılır, 3 öğrenci kalır: 23 / 5 = 4, 23 % 5 = 3 ve 4 × 5 + 3 = 23.

Negatif sayıda C bölümü sıfıra doğru keser: −7 / 3 = −2,33...; küsurat atılır ve sonuç −3 değil −2 olur. Kalan aynı kuraldan çıkar: −2 × 3 = −6; −7'ye ulaşmak için −1 gerekir. Yani C'de kalan, bölünenin işaretini taşır:

```
printf("%d\n", -7 / 3);
printf("%d\n", -7 % 3);
printf("%d\n", 7 % 3);
printf("%d\n", 7 % -3);
```

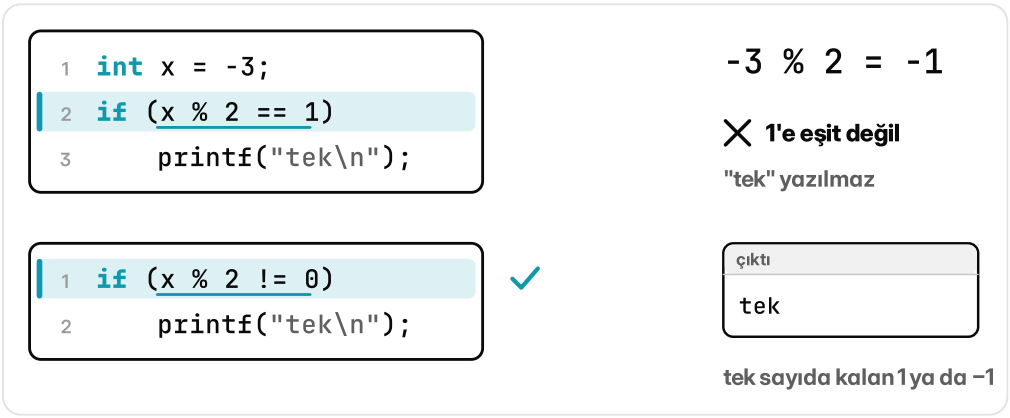
Çıktı:

```
-2
-1
1
1
```

Bu bir tuzak kurar. Sayının tek olup olmadığını x % 2 == 1 diye sorarsan, −3 için kalan −1 çıkar ve −3 tek sayı sayılmaz:

```
int x = -3;
if (x % 2 == 1)            /* -3 % 2 = -1: yazmaz */
    printf("tek\n");
if (x % 2 != 0)            /* doğru yol: yazar */
    printf("tek\n");
```

Doğrusu, kalanın sıfırdan farklı olup olmadığına bakmaktır: tek sayıda kalan 1 ya da −1'dir. Bu program ekrana yalnız bir kez tek yazar.



Şekil 2.5 −3 % 2 = −1 olduğu için x % 2 == 1 koşulu yanlış çıkar; x % 2 != 0 ise doğru çalışır ve tek yazar.

Bir değişkeni 1 artırmak o kadar sık yapılır ki C'nin kısa bir yazımı var: i++, i'yi 1 artırır; i--, 1 azaltır. 5 eklemek için i += 5 yazılır; bu, i = i + 5'in kısasıdır. -= de aynı biçimde çıkarır:

```
int i = 10;
i++;                       /* 11 */
i--;                       /* 10 */
i += 5;                    /* 15 */
i -= 2;                    /* 13 */
```

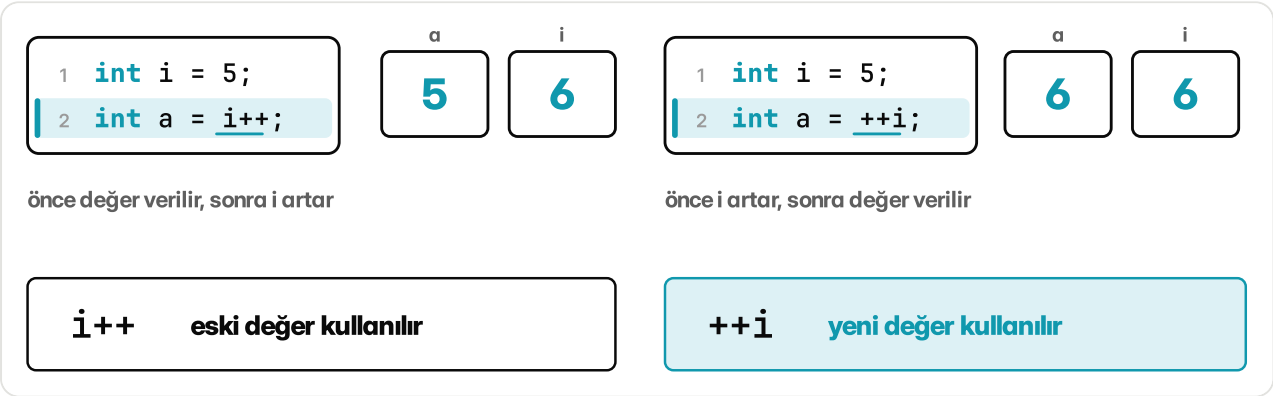
++ değişkenin önüne de yazılabilir. Satırda tek başına durunca i++ ile ++i aynıdır; ikisi de i'yi 1 artırır. Fark, değer başka bir yerde kullanılınca çıkar:

```
int i = 5;
int a = i++;                /* a 5, i 6 */
printf("a=%d i=%d\n", a, i);
int j = 5;
int b = ++j;
printf("b=%d j=%d\n", b, j);
```

Çıktı:

```
a=5 i=6
b=6 j=6
```

a = i++ dersek önce i'nin değeri a'ya verilir, sonra i artar: a 5, i 6 olur. ++ öne yazılırsa önce j artar, sonra değeri verilir: b de j de 6 olur. Kısacası ++ arkadaysa eski değer, öndeyse yeni değer kullanılır.



Şekil 2.6 int a = i++; ile a 5, i 6; int a = ++i; ile a 6, i 6: i++ eski değeri, ++i yeni değeri kullanır.

5 kez mi?

videoda 11:43

Bu döngü ekrana 5 kez merhaba yazmalı, ama yalnızca bir kez yazar:

```
for (int i = 0; i < 5; i++);
    printf("merhaba\n");
```

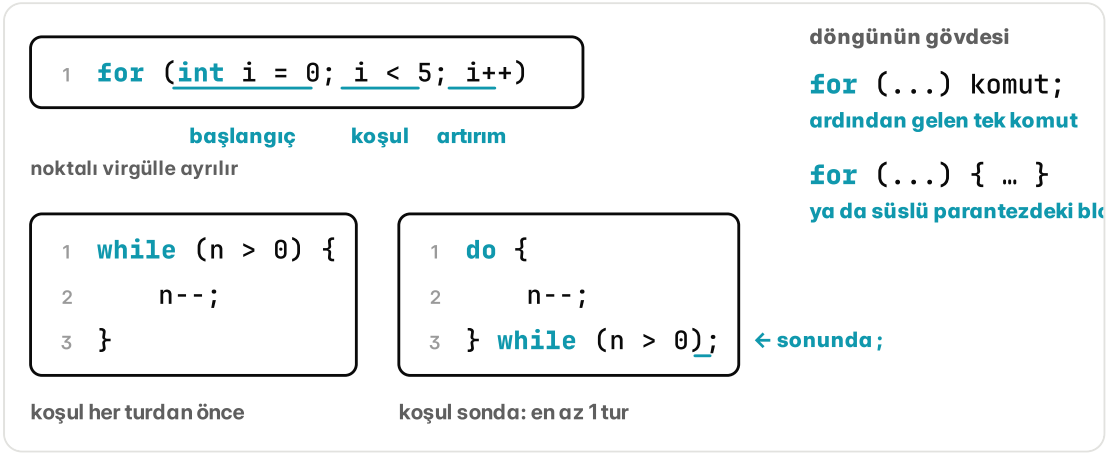
Sebep, ilk satırın sonundaki küçük bir işaret.

C'de üç döngü yazımı var. for'un parantezinde üç parça noktalı virgülle ayrılır: başlangıç (int i = 0), koşul (i < 5) ve artırım (i++). while yalnızca koşulu taşır ve onu her turdan önce sorar. do-while koşulu sonda sorar, bu yüzden gövde en az bir kez çalışır; while'dan sonraki noktalı virgül de unutulmamalıdır:

```
int n = 3;
while (n > 0) {
    n--;                    /* 3 tur, n 0 olur */
}
do {
    n--;
} while (n > 0);
printf("%d\n", n);          /* -1 */
```

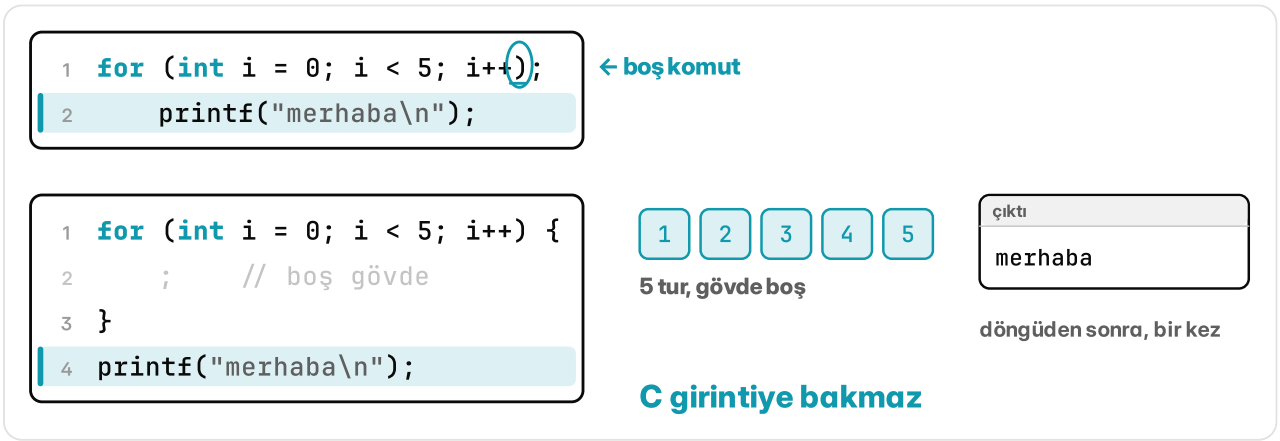
while 3 tur döner ve n'yi 0'a indirir. do-while'a gelindiğinde koşul zaten yanlıştır, ama koşul sonda sorulduğu için gövde bir kez çalışır: n −1 olur ve ekrana -1 yazılır.

Döngünün **gövdesi**, ardından gelen tek komut ya da süslü parantezdeki bloktur.



Şekil 2.7 for başlangıç, koşul ve artırımı taşır; while koşulu her turdan önce, do-while sonda sorar. Döngünün gövdesi ardından gelen tek komut ya da süslü parantezdeki bloktur.

Şimdi baştaki for satırına bakalım: parantezden hemen sonra bir noktalı virgül var. C'de tek başına noktalı virgül, hiçbir şey yapmayan **boş komuttur**. Döngünün gövdesi bu boş komut olur: döngü 5 tur hiçbir şey yapmadan döner ve biter. Alttaki printf döngünün dışında kalır ve bir kez çalışır. İçeri girintili yazılmış olması bir şey değiştirmez, çünkü C girintiye bakmaz. Uyarıları açmak burada da işe yarar: clang -Wall ile bu satır için for loop has empty body (döngünün gövdesi boş) uyarısını verir.

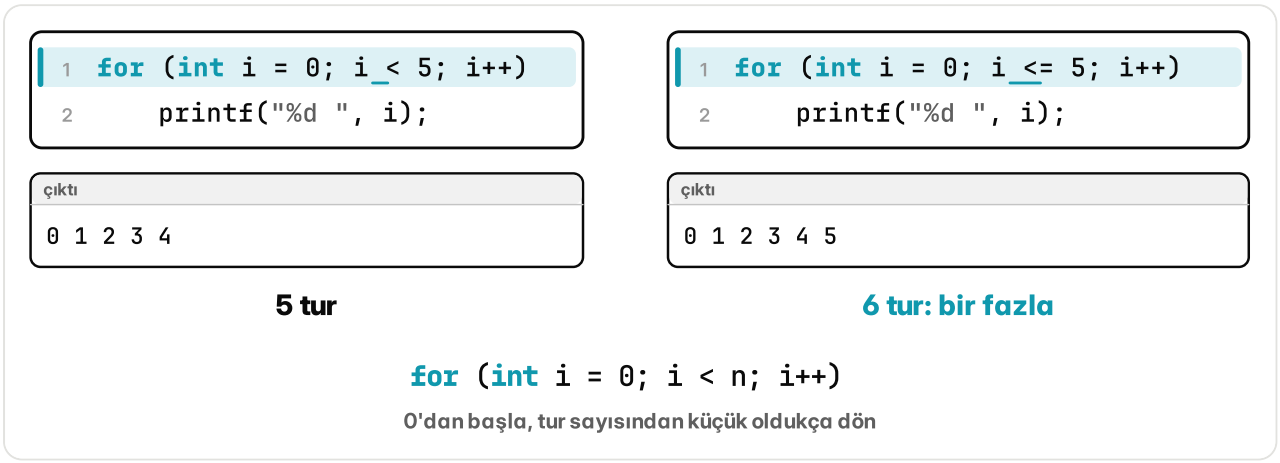


Şekil 2.8 Noktalı virgül döngüye boş bir gövde verir: döngü 5 tur boş döner, merhaba döngüden sonra bir kez yazılır.

Döngüde en sık hata ise bir tur fazla ya da eksik dönmektir. C'de sayım 0'dan başlar: 5 tur için i 0'dan başlar ve koşul i < 5'tir; i sırayla 0, 1, 2, 3, 4 olur. <= yazarsan i 5'ken de döner: 6 tur.

```
for (int i = 0; i < 5; i++)
    printf("%d ", i);       /* 0 1 2 3 4: 5 tur */
printf("\n");
for (int i = 0; i <= 5; i++)
    printf("%d ", i);       /* 0 1 2 3 4 5: 6 tur */
printf("\n");
```

Bu yüzden C'de en sık yazılan döngü satırı for (int i = 0; i < n; i++) biçimindedir: 0'dan başla, tur sayısından küçük oldukça dön.



Şekil 2.9 i < 5 ile çıktı 0 1 2 3 4 (5 tur), i <= 5 ile 0 1 2 3 4 5 (6 tur, bir fazla).

Tek bir break

videoda 12:53

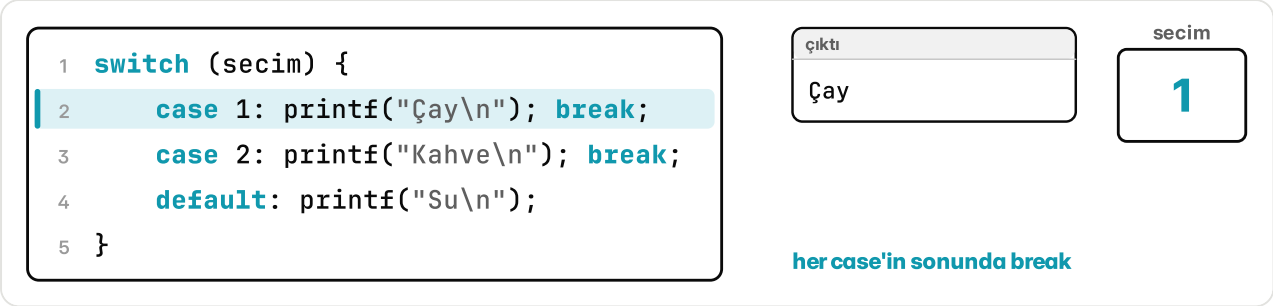
1990'da ABD'nin en büyük telefon şirketinin, AT&T'nin şebekesi 9 saat aksadı; milyonlarca arama bağlanamadı. Sebep, C ile yazılmış yazılımda yanlış yerden çıkan tek bir break komutuydu. break nereden çıkar?

break'i en çok switch içinde görürüz. switch bir değişkene bakar ve değeri eşit olan case etiketine atlar; hiçbiri eşit değilse default'a gider. Seçim 1 ise çay, 2 ise kahve, hiçbiri değilse su:

```
int secim = 1;
switch (secim) {           /* break yok: Çay, Kahve, Su */
    case 1: printf("Çay\n");
    case 2: printf("Kahve\n");
    default: printf("Su\n");
}
```

Ama case yalnızca bir giriş kapısıdır, duvar değil. break yoksa seçim 1 iken Çay yazılır ve akış aşağı devam eder: Kahve ve Su da yazılır. Buna **düşme** denir. Her durumun sonuna break konunca switch'ten orada çıkılır ve yalnızca Çay yazılır:

```
switch (secim) {           /* break var: yalnız Çay */
    case 1: printf("Çay\n"); break;
    case 2: printf("Kahve\n"); break;
    default: printf("Su\n");
}
```



Şekil 2.10 Her case'in sonuna break konunca secim 1 için yalnız Çay yazılır.

Düşme bazen işe yarar. 6 ve 7 için ayrı ayrı yazmak yerine iki case alt alta yazılır, ikisi de hafta sonu der:

```
switch (gun) {
    case 6:
    case 7: printf("Hafta sonu\n"); break;
    default: printf("Hafta içi\n");
}
```

gun 6 ise case 6'dan girilir; orada komut olmadığı için akış case 7'nin satırına düşer ve Hafta sonu yazılır.

break, içinde bulunduğu en yakın döngüden ya da switch'ten çıkar, daha dışarıya geçmez. İç içe iki döngüde yalnızca içtekini bitirir:

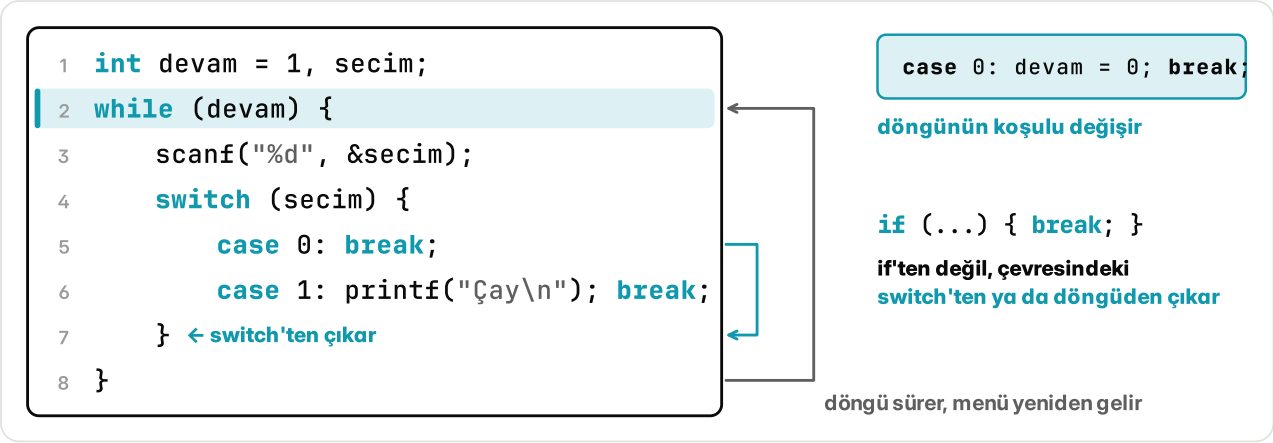
```
int i, j;
for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
        if (j == 1) break;  /* yalnız içteki döngüden çıkar */
        printf("i=%d j=%d\n", i, j);
    }
}
```

Çıktı:

```
i=0 j=0
i=1 j=0
i=2 j=0
```

j 1 olunca içteki döngü biter, dıştaki döngü sıradaki i ile sürer.

switch bir döngünün içindeyse break yalnızca switch'ten çıkar. while (devam) döngüsünün içinde bir switch ile yazılmış bir menüde, 0'a basınca çıkmak için case 0: break; yazarsan döngü bitmez, menü yeniden gelir. Döngüden çıkmak için döngünün koşulunu değiştirmek gerekir: case 0: devam = 0; break;. if içindeki break ise if'ten değil, çevresindeki switch'ten ya da döngüden çıkar.

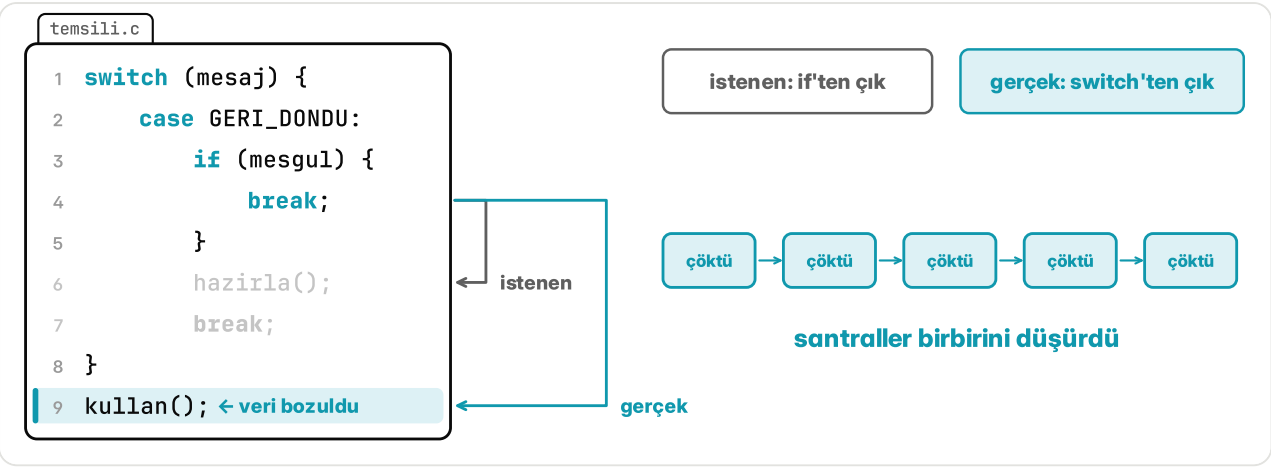


Şekil 2.11 Menüde case 0'daki break yalnız switch'ten çıkar, döngü sürer; döngüyü bitirmek için devam = 0 yapılır. if içindeki break if'ten değil, çevresindeki switch'ten ya da döngüden çıkar.

Telefon şebekesindeki hata da buydu. Aşağıdaki kod olayın sadeleştirilmiş, temsilî bir biçimidir, gerçek kod değil:

```
switch (mesaj) {
    case GERI_DONDU:
        if (mesgul) {
            break;          /* if'ten değil, switch'ten çıkar */
        }
        hazirla();          /* atlanır */
        break;
}
kullan();                   /* çıktı: veri bozuk */
```

Programcı bir if'ten erken çıkmak için break yazdı; break ise çevresindeki switch'ten çıktı. Hazırlık atlandı, program yapmaması gereken işe geçti ve veriyi bozdu. Çöken santral yeniden açılırken komşularına gönderdiği mesajlar onlarda da aynı hatayı tetikledi; santraller birbirini düşürdü.



Şekil 2.12 İstenen, break'in yalnız if'ten çıkmasıydı; gerçekte switch'ten çıktı, hazirla() atlandı ve veri bozuldu. Santraller birbirini düşürdü.

Özet

- C'de koşul bir sayıdır: 0 yanlış, 0'dan farklı her sayı doğru; karşılaştırmalar 1 ya da 0 verir.
- if (x = 5) atama yapar ve hep doğrudur; karşılaştırmak için == yazılır.
- && ve || kısa devre yapar: sonuç belli olunca ikinci koşul hesaplanmaz.
- Tam sayı bölmesi sıfıra doğru keser, kalan bölünenin işaretini taşır: −7 / 3 = −2, −7 % 3 = −1; tek sayı denetimi x % 2 != 0 ile yapılır.
- i++ eski değeri, ++i yeni değeri kullanır; for satırının sonundaki noktalı virgül döngüye boş bir gövde verir.
- switch'te break yoksa akış sonraki case'lere düşer; break yalnız en yakın döngüden ya da switch'ten çıkar, if'ten çıkmaz.

Sorular

1 Şu satırlar ne yapar?

```
int x = 0;
if (x = 5) printf("x beş\n");
```

A Hiçbir şey yazmaz, çünkü x 0'dır.

B Derleme hatası verir.

C x beş yazar; atama koşulu 5 yapar ve 5 doğrudur.

D x beş yazar ama x 0 olarak kalır.

2 **Şu satırlar ekrana ne yazar?**

```
int a = 10, b = 0;
if (b != 0 && a / b > 2) printf("büyük\n");
printf("bitti\n");
```

A büyük ve altında bitti

B Yalnız bitti

C Hiçbir şey; program sıfıra bölmede çöker.

D Yalnız büyük

3 **Şu satır ekrana ne yazar?**

```
printf("%d\n", -10 % 4);
```

Cevap: _____

4 **Şu satırlardan sonra a ve i kaçtır?**

```
int i = 5;
int a = i++;
```

A a 6, i 6

B a 5, i 5

C a 6, i 5

D a 5, i 6

5 **Şu satırlar ekrana kaç kez merhaba yazar?**

```
for (int i = 0; i < 5; i++);
    printf("merhaba\n");
```

A 1 kez

B 5 kez

C Hiç yazmaz.

D 6 kez

6 **for (int i = 0; i <= 5; i++) döngüsü kaç tur döner?**

Cevap: _____

7 **secim 2 iken şu switch ekrana ne yazar?**

```
switch (secim) {
    case 1: printf("Çay\n");
    case 2: printf("Kahve\n");
    default: printf("Su\n");
}
```

A Yalnız Kahve

B Çay, Kahve ve Su

C Kahve ve Su

D Yalnız Su

8 **Bir döngünün içindeki switch'in bir case'inde if (mesgul) { break; } satırı çalışıyor. Bu break nereden çıkar?**

A Yalnız if'ten; case'in kalan satırları çalışır.

B switch'ten; döngü sürer.

C Hem switch'ten hem de döngüden

D Döngüden; switch'in kalan satırları atlanır ve döngü biter.

PROGRAMLAMAYA GİRİŞ · 1. DÖNEM

3 Diziler ve karakter dizileri

Bu bölümde C'de dizi tanımlamayı, dizinin eleman sayısını bulmayı ve C'nin dizinin sınırını neden denetlemediğini öğreneceksin. Ardından metinlerin C'de nasıl tutulduğunu, sonlandırıcıyı, iki boyutlu dizileri ve metinlerle çalışan hazır fonksiyonları göreceksin.

Onuncu eleman

videoda 14:15

Beş elemanlı bir dizi tanımlayıp onuncu elemanına 99 yazalım:

```
#include <stdio.h>

int main(void) {
    int puan[5] = {12, 7, 30};
    puan[9] = 99;               /* dizinin dışı: tanımsız davranış */
    printf("%d\n", puan[0]);    /* program durmaz */
    return 0;
}
```

Program derlenir ve çalışır; puan[9] satırında programı durduran bir hata çıkmaz. C bizi neden durdurmaz? Cevaba geçmeden önce dizi tanımına bakalım.

Bir dizi tanımlarken önce elemanların türünü, sonra dizinin adını, köşeli parantez içinde de eleman sayısını yazarız. İlk değerler süslü parantez içinde verilir. Beş gözlük bir diziye yalnız üç değer verirsek kalan iki göz 0 ile dolar: puan bellekte 12, 7, 30, 0, 0 olur. Eleman sayısını hiç yazmazsak derleyici süslü parantezdeki değerleri sayar ve dizinin boyunu kendisi belirler. Aşağıdaki programda liste dizisinin boyu bu yüzden 3'tür:

```
int puan[5] = {12, 7, 30};  /* 12 7 30 0 0 */
int liste[] = {12, 7, 30};  /* boyu derleyici sayar: 3 */
printf("%d %d\n", puan[4], (int)(sizeof liste / sizeof liste[0]));  /* 0 3 */
```

Çıktı:

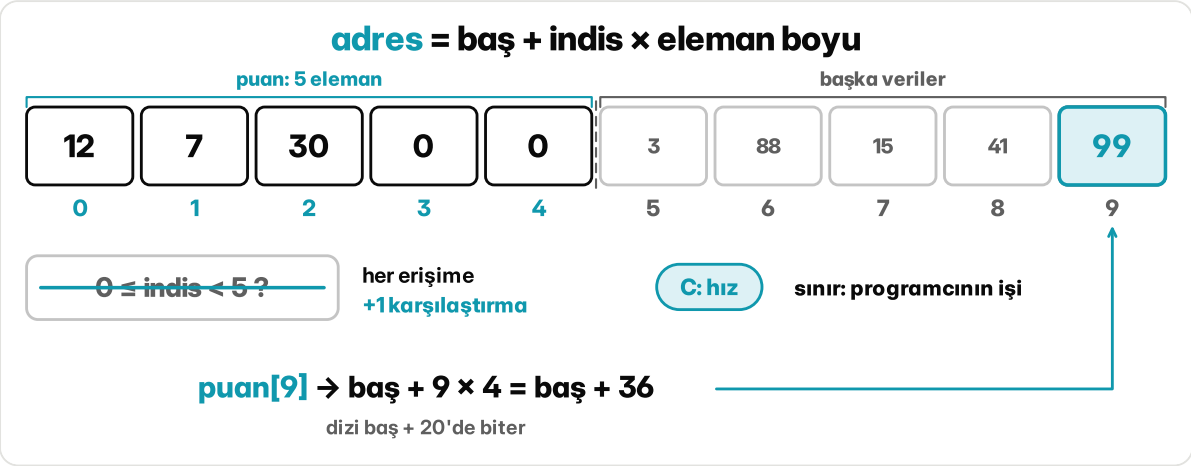
```
0 3
```

Dizinin kaç elemanlı olduğunu sizeof ile buluruz. **sizeof**, bir şeyin bellekte kaç bayt tuttuğunu söyler. int'in 4 bayt olduğu bir bilgisayarda (bugünkü bilgisayarların çoğu böyledir) bütün dizi 20 bayt, tek bir eleman 4 bayttır. 20'yi 4'e bölünce eleman sayısı çıkar:

```
int n = sizeof puan / sizeof puan[0];
printf("%d\n", n);          /* 5 (int 4 bayt: 20 / 4) */
```

Bu hesabın iyi yanı, dizinin boyunu sonradan değiştirsek de doğru kalmasıdır; 5'i elle yazmak gerekmez.

Şimdi asıl soruya dönelim. C bir elemana ulaşırken yalnız adresini hesaplar: dizinin başı artı indis çarpı eleman boyu. İndisin dizinin içinde kalıp kalmadığına bakmaz. Bakmak, her erişime "0 ≤ indis < 5 mi?" diye fazladan bir karşılaştırma eklemek demektir. C'nin tasarımcıları hızı seçti, sınırı korumayı programcıya bıraktı. puan[9]'un adresi baş + 9 × 4 = baş + 36'dır; dizi ise baş + 20'de biter. Program oraya, başka bir verinin üstüne yazar.

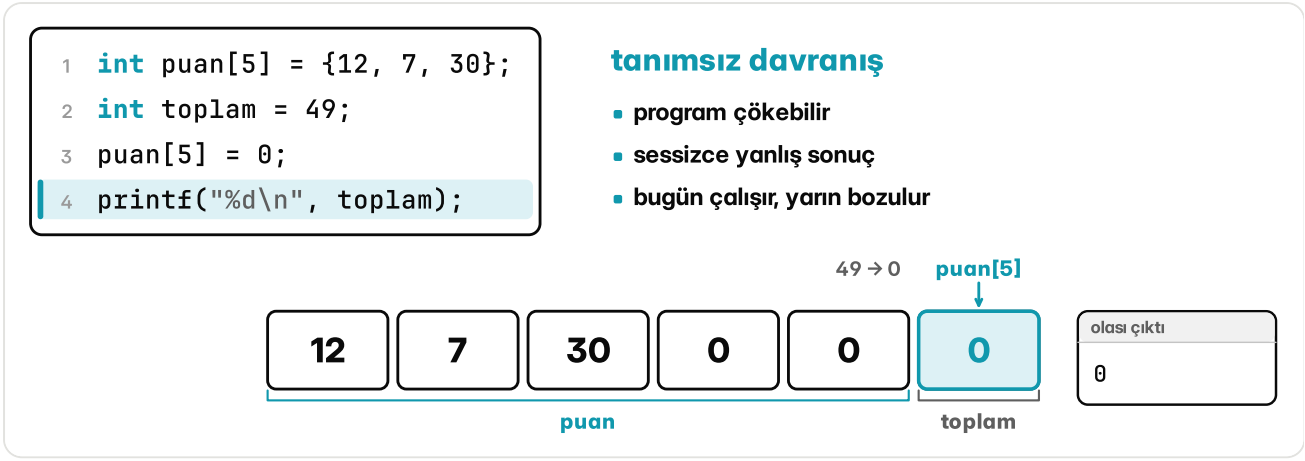


Şekil 3.1 Adres = baş + indis × eleman boyu: puan[9] baş + 36'ya, dizinin bittiği baş + 20'nin ötesine, başka verilerin üstüne düşer.

Bellekte dizinin hemen ardında başka bir değişken duruyorsa ne olur?

```
int puan[5] = {12, 7, 30};
int toplam = 49;
puan[5] = 0;                /* dizinin dışı: tanımsız davranış */
printf("%d\n", toplam);     /* olası: 0 (toplam dizinin ardındaysa) */
```

toplam dizinin ardındaysa puan[5]'e yazılan 0 onun üstüne gelir ve ekrana 49 yerine 0 çıkar. Değişkenlerin bellekteki sırası derleyiciye bağlıdır; bu örneği denediğimiz derleyicide toplam başka bir yerdeydi ve 49 yazdı. C standardı bu duruma **tanımsız davranış** (undefined behavior) der: program çökebilir, yanlış sonuçla sessizce sürebilir ya da bugün çalışıp yarın bozulabilir. İndis sabit bir sayıysa derleyici çoğu zaman uyarı verir; indis bir değişkenden geliyorsa bunu da göremez.

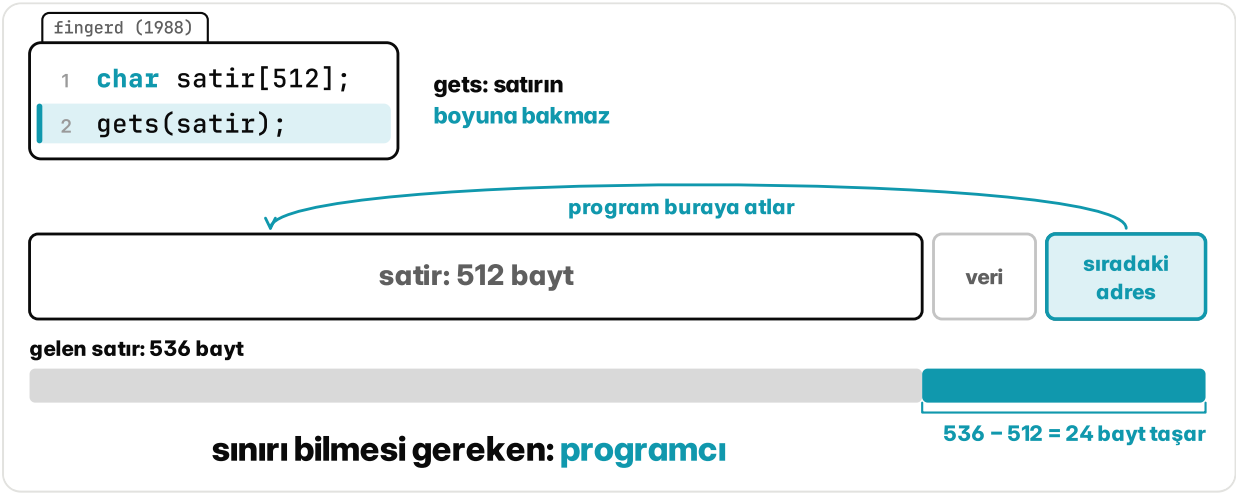


Şekil 3.2 toplam dizinin hemen ardındaysa puan[5]'e yazılan 0 onun üstüne gelir: 49 → 0, olası çıktı 0.

Bu hatanın bedeli 1988'de görüldü. 2 Kasım 1988'de Morris solucanı, internete bağlı binlerce bilgisayara kendi kendine yayıldı. Girdiği kapılardan biri, kullanıcılar hakkında bilgi veren fingerd programıydı. Program gelen satırı 512 baytlık bir diziye okuyordu:

```
char satir[512];
gets(satir);                /* satırın boyuna bakmaz */
```

Satırı okuyan gets fonksiyonu dizinin boyunu bilmez. Solucan 536 baytlık bir satır gönderdi; 536 − 512 = 24 bayt dizinin dışına taştı. Taşan baytlar, programın birazdan nereye atlayacağını tutan yere yazıldı ve program solucanın komutlarına atladı. C dizinin sınırını bilmez; bilmesi gereken programcıdır.



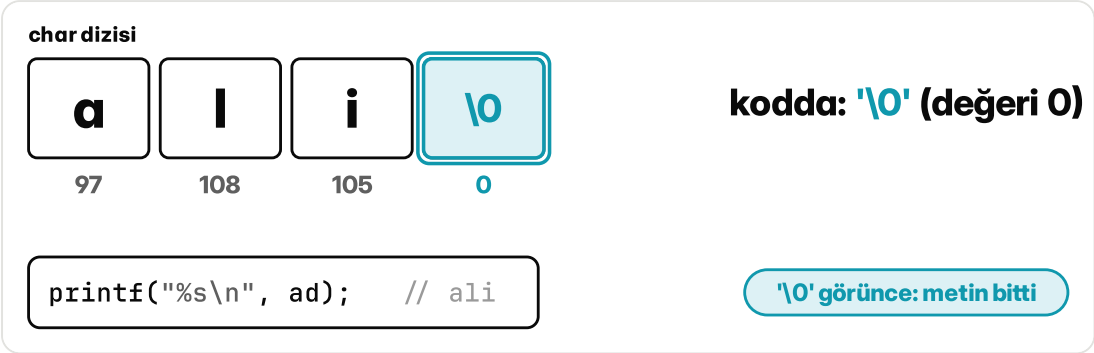
Şekil 3.3 fingerd'in 512 baytlık dizisine 536 baytlık satır gelir; taşan 24 bayt sıradaki adresi bozar ve program oraya atlar.

3 harf, 4 bayt

videoda 16:03

Üç harfli ali adını bir char dizisine koyup sizeof ile boyuna bakalım. Ekrana 4 çıkar. Fazladan bayt nereden geliyor?

C'de metin, harfleri sırayla tutan bir char dizisidir. Derleyici, tırnak içindeki her metnin sonuna değeri 0 olan bir karakter ekler; kodda '\0' diye yazılır. Bu karaktere **sonlandırıcı** denir. Bellekte ad dizisi a, l, i ve '\0' olarak durur; harflerin sayı değerleriyle 97, 108, 105, 0. Metinle çalışan her fonksiyon, örneğin metni ekrana yazan printf("%s"), bu 0'ı görünce metnin bittiğini anlar.



Şekil 3.4 ali bellekte a, l, i ve '\0' olarak durur; sayı değerleri 97, 108, 105 ve 0.

Metnin uzunluğunu string.h kütüphanesindeki strlen fonksiyonu bulur: sonlandırıcıya kadar sayar ve 3 der. sizeof ise diziye ayrılan yeri ölçer ve 4 der:

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

int main(void) {
    char ad[] = "ali";
    printf("%zu\n", sizeof ad);     /* 4: ayrılan yer */
    printf("%zu\n", strlen(ad));    /* 3: sıfıra kadar */
    return 0;
}
```

Çıktı:

```
4
3
```

Diziyi 3 gözlük tanımlarsak ne olur? C, char kisa[3] = "ali"; satırını yine kabul eder, ama sonlandırıcıya yer kalmaz. printf sıfırı bulana kadar okumayı sürdürür ve dizinin ötesine geçer; ali'nin ardından anlamsız harfler çıkabilir. Bu da tanımsız davranıştır; bu örnekte ekrana ali'nin ardından birkaç anlamsız karakter çıktı.

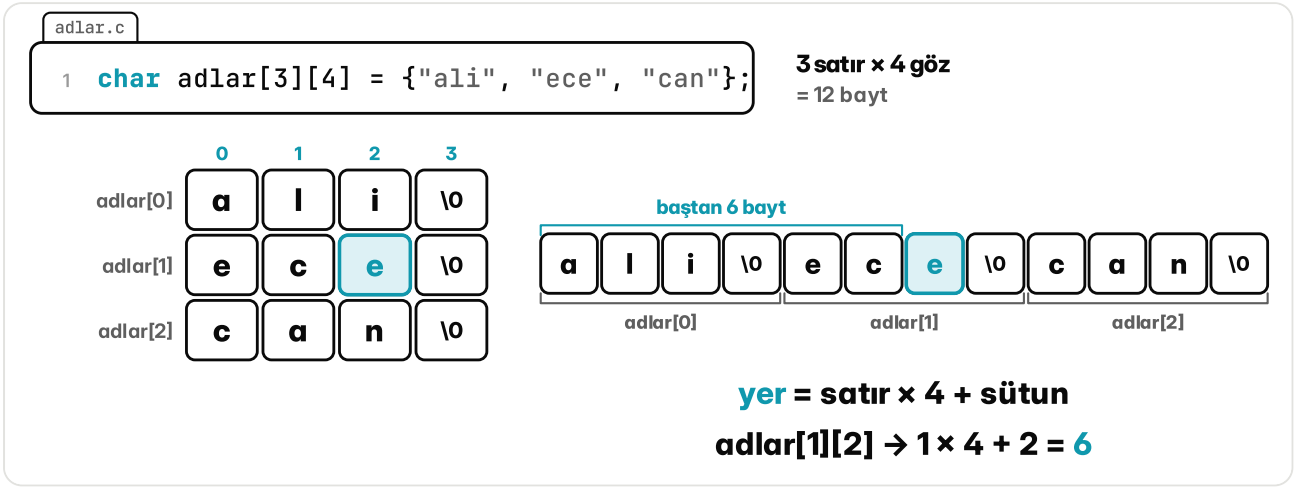
Birkaç adı bir arada tutmak için iki boyutlu bir dizi kurarız: 3 satır, her satırda 4 göz, toplam 12 bayt.

```
char adlar[3][4] = {"ali", "ece", "can"};
printf("%c\n", adlar[1][2]);    /* e: baştan 1 * 4 + 2 = 6 bayt */
printf("%s\n", adlar[2]);       /* can */
```

Çıktı:

```
e
can
```

Bellekte satırlar tek sıra hâlinde arka arkaya dizilir: önce ali'nin 4 baytı, sonra ece'nin, sonra can'ın. Bir elemanın yeri satır × 4 + sütun ile bulunur. adlar[1][2] baştan 1 × 4 + 2 = 6 bayt ileridedir; orada ece'nin son harfi e durur. adlar[2] ise tek başına bir satırın tamamıdır, yani can metni.

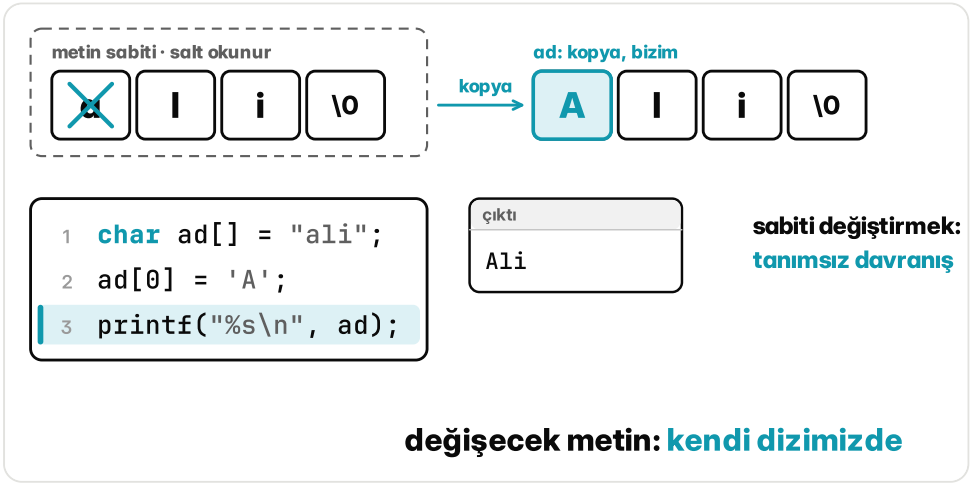


Şekil 3.5 3 × 4'lük adlar tablosu bellekte satır satır 12 bayta yerleşir; adlar[1][2] baştan 1 × 4 + 2 = 6 bayt ileridedir.

Tırnak içindeki "ali" bir **metin sabitidir**; çoğu derleyici onu programın değiştirilemeyen bir bölgesine koyar. Diziye ilk değer olarak verdiğimizde dizi bu sabitin bir kopyasını alır ve kopya bizimdir:

```
char ad[] = "ali";          /* metin sabitinin kopyası */
ad[0] = 'A';
printf("%s\n", ad);         /* Ali */
```

İlk harfi büyük A yapınca dizide Ali yazar. Sabitin kendisini değiştirmek ise tanımsız davranıştır; değiştireceğin metni her zaman kendi dizinde tut.



Şekil 3.6 Salt okunur metin sabitinin kopyası ad dizisine alınır; ad[0] = 'A' yalnız kopyayı değiştirir ve çıktı Ali olur.

Eşit değil

videoda 17:24

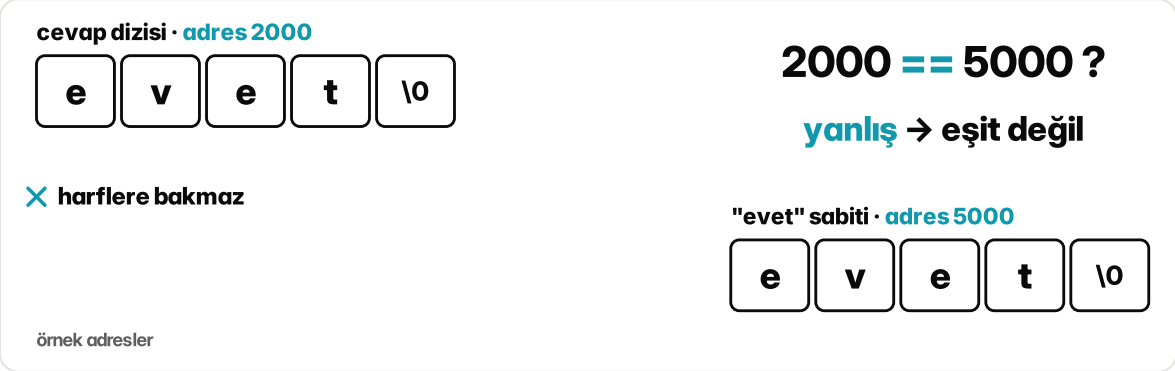
Bir char dizisine evet yazıp onu == ile "evet"e karşılaştıralım:

```
#include <stdio.h>

int main(void) {
    char cevap[] = "evet";
    if (cevap == "evet")        /* iki adres karşılaştırılır */
        printf("eşit\n");
    else
        printf("eşit değil\n"); /* bu satır çalışır */
    return 0;
}
```

Çıktı:

```
eşit değil
```

Harfler tıpatıp aynı, ama program eşit değil diyor. Çünkü == iki metnin harflerine değil, bellekteki yerlerine, yani adreslerine bakar. cevap dizisi bellekte bir yerde (örneğin 2000 adresinde), tırnaktaki "evet" sabiti başka bir yerde (örneğin 5000 adresinde) durur. 2000 == 5000 yanlış olduğu için sonuç hep "eşit değil" çıkar. Derleyici bu satırda uyarı da verir.



Şekil 3.7 cevap dizisi 2000'de, "evet" sabiti 5000'de durur; == harflere değil adreslere bakar ve 2000 == 5000 yanlıştır.

Metinleri karşılaştırmak için string.h'deki strcmp fonksiyonu kullanılır. strcmp harfleri tek tek karşılaştırır: iki metin eşitse 0 verir; ilk farklı harfte birincinin kodu küçükse eksi, büyükse artı bir sayı verir. "ali" ile "can"ın ilk harfleri farklıdır; a (97), c'den (99) küçük olduğu için sonuç eksidir.

```
char cevap[] = "evet";
if (strcmp(cevap, "evet") == 0)
    printf("eşit\n");                       /* eşit */
printf("%d\n", strcmp("ali", "can") < 0);   /* 1: eksi */
printf("%d\n", strcmp("can", "ali") > 0);   /* 1: artı */
```

Çıktı:

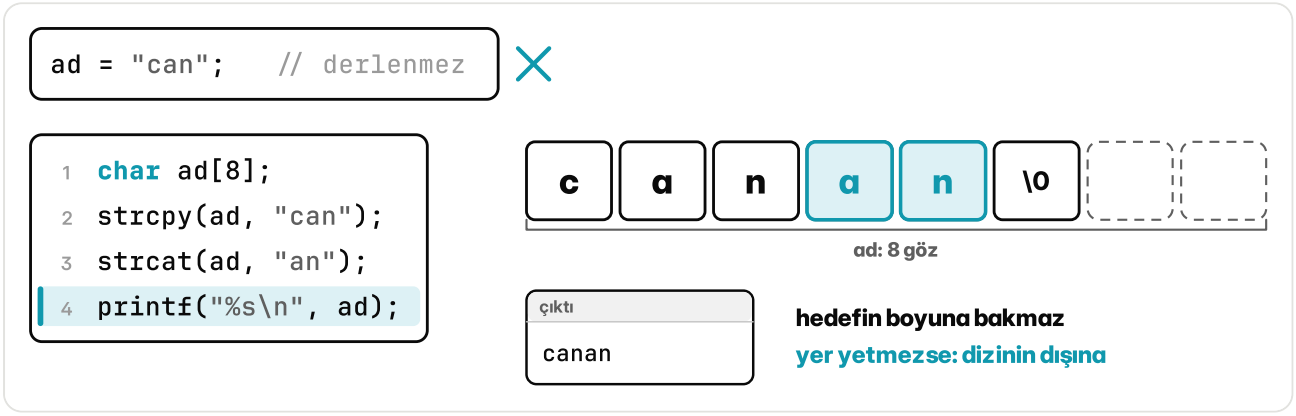
```
eşit
1
1
```

Son iki satır, karşılaştırma doğruysa 1 yazar. Burada bir tuzak var: strcmp eşitlikte 0 verir ve C'de 0 "yanlış" demektir. Koşula yalnız if (strcmp(cevap, "evet")) yazarsak koşul metinler farklıyken çalışır. Doğrusu, sonucun 0'a eşit olduğuna bakmaktır: == 0.

Tanımdan sonra bir metni = ile diziye atayamayız; ad = "can"; satırı derlenmez, çünkü dizinin tamamına tek seferde değer verilemez. Kopyalamak için strcpy, sona eklemek için strcat vardır:

```
char ad[8];
/* ad = "can";  derlenmez: diziye tek seferde atama yok */
strcpy(ad, "can");
strcat(ad, "an");
printf("%s\n", ad);         /* canan */
```

can'a an eklenince canan olur: 5 harf ve sonlandırıcı, 6 bayt; 8 gözlük diziye sığar. İki fonksiyon da sonlandırıcıyı taşır, ama hedef dizinin boyuna bakmaz; yer yetmezse dizinin dışına yazar.

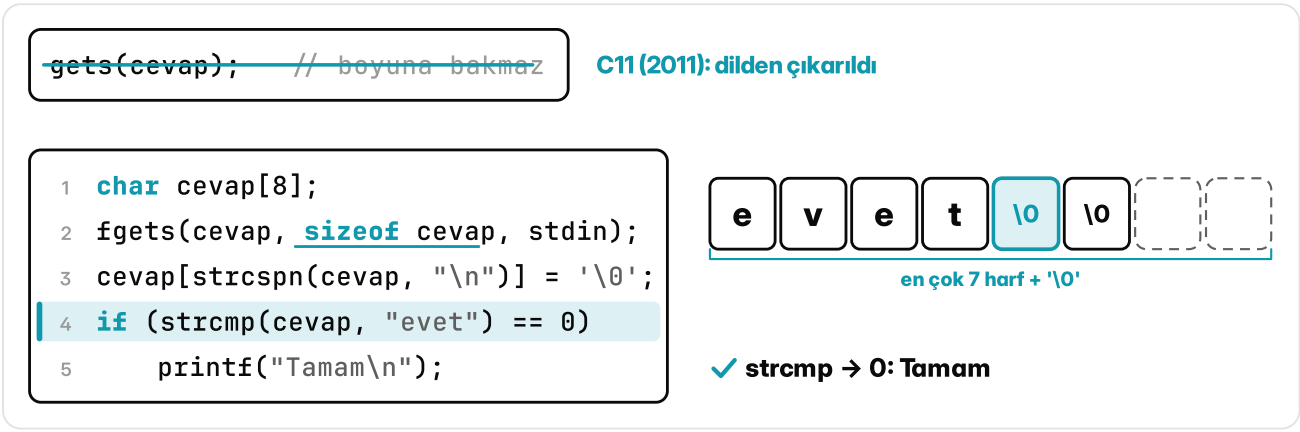


Şekil 3.8 8 gözlük ad dizisinde strcpy ve strcat sonrası c, a, n, a, n ve '\0' durur; çıktı canan.

Kullanıcıdan satır okumanın eski yolu gets'ti. gets satırı dizinin boyunu bilmeden yazar ve uzun bir satır dizinin dışına taşar. Güvenli kullanmanın bir yolu olmadığı için gets, C'nin 2011 standardında (C11) dilden çıkarıldı. Yerine fgets kullanılır: dizinin boyunu da alır, en çok boyun bir eksiği kadar karakter okur ve sona sonlandırıcıyı koyar.

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

int main(void) {
    char cevap[8];
    if (fgets(cevap, sizeof cevap, stdin) == NULL)
        return 1;
    cevap[strcspn(cevap, "\n")] = '\0';     /* satır sonunu sil */
    if (strcmp(cevap, "evet") == 0)
        printf("Tamam\n");
    return 0;
}
```

stdin standart girdidir, yani çoğu zaman klavye. 8 gözlük dizi en çok 7 karakter ve '\0' alır. fgets, Enter tuşunun bıraktığı satır sonu karakterini ('\n') de diziye yazar: dizide e, v, e, t, '\n', '\0' durur. strcspn, '\n'in metinde kaçıncı sırada olduğunu verir; o göze '\0' koyunca satır sonu silinir. Ancak bundan sonra strcmp 0 verir: kullanıcı evet yazıp Enter'a basarsa ekrana Tamam çıkar.



Şekil 3.9 gets dilden çıkarıldı; fgets en çok 7 harf ve '\0' okur, satır sonu '\0' ile silinince strcmp 0 verir ve Tamam yazılır.

Özet

- Dizi türü, adı ve eleman sayısıyla tanımlanır; eksik verilen ilk değerler 0 olur, boyu yazılmazsa derleyici sayar.
- sizeof bayt sayısını verir; eleman sayısı sizeof dizi / sizeof dizi[0] ile bulunur.
- C dizinin sınırını denetlemez; dışarı yazmak tanımsız davranıştır ve komşu veriyi bozabilir.
- C'de metin, sonu '\0' sonlandırıcısıyla biten bir char dizisidir; strlen sonlandırıcıya kadar sayar, sizeof ayrılan yeri ölçer.
- İki boyutlu dizi bellekte satır satır yerleşir; yer = satır × satır boyu + sütun.
- Metinler == ile değil, strcmp ile karşılaştırılır ve eşitlik == 0 ile denetlenir; satır okumak için gets değil fgets kullanılır.

Sorular

1 **int puan[5] = {12, 7, 30}; tanımından sonra puan[3]'ün değeri nedir?**

A 30

B 0

C Bellekte o an ne varsa, rastgele bir değer

D Derleme hatası verir, çünkü yalnız 3 değer verildi

2 **int'in 4 bayt olduğu bir bilgisayarda int dizi[] = {4, 8, 15, 16, 23, 42}; tanımı için sizeof dizi kaç verir?**

Cevap:

3 **int puan[5]; tanımlı bir programda puan[9] = 99; satırı çalışınca ne olur?**

A C, programı derleme sırasında hata vererek durdurur.

B C, program çalışırken indisi denetler ve hata vererek durdurur.

C C adresi hesaplar ve dizinin dışına, başka bir verinin yerine yazar; sonuç tanımsız davranıştır.

D Dizi kendiliğinden 10 elemana büyür.

4 **Aşağıdaki iki satırın çıktısı nedir?**

```
char ad[] = "kedi";
printf("%zu %zu\n", sizeof ad, strlen(ad));
```

Cevap:

5 **char adlar[3][4] = {"ali", "ece", "can"}; dizisinde adlar[2][1] hangi harftir ve dizinin başından kaç bayt ileridedir?**

A a; baştan 9 bayt

B e; baştan 6 bayt

C c; baştan 8 bayt

D n; baştan 10 bayt

6 **Aşağıdaki iki satır neden derlenmez?**

```
char ad[8];
ad = "can";
```

A 8 göz, can için yetmez.

B can tırnak yerine kesme işaretiyle yazılmalıydı: 'can'.

C char dizisine metin konamaz; metin için ayrı bir tür gerekir.

D Dizinin tamamına tanımdan sonra = ile tek seferde değer verilemez; strcpy kullanılmalıdır.

7 **Aşağıdaki kod ekrana ne yazar?**

```
char cevap[] = "evet";
if (strcmp(cevap, "evet"))
    printf("eşit\n");
else
    printf("farklı\n");
```

A eşit

B farklı

C Hiçbir şey yazmaz.

D Derlenmez.

8 **char cevap[8]; fgets(cevap, sizeof cevap, stdin); satırıyla kullanıcı evet yazıp Enter'a bastı. Dizinin sonlandırıcı dahil kaç baytı dolu olur?**

Cevap: _____

4 Fonksiyonlar ve değişkenlerin ömrü

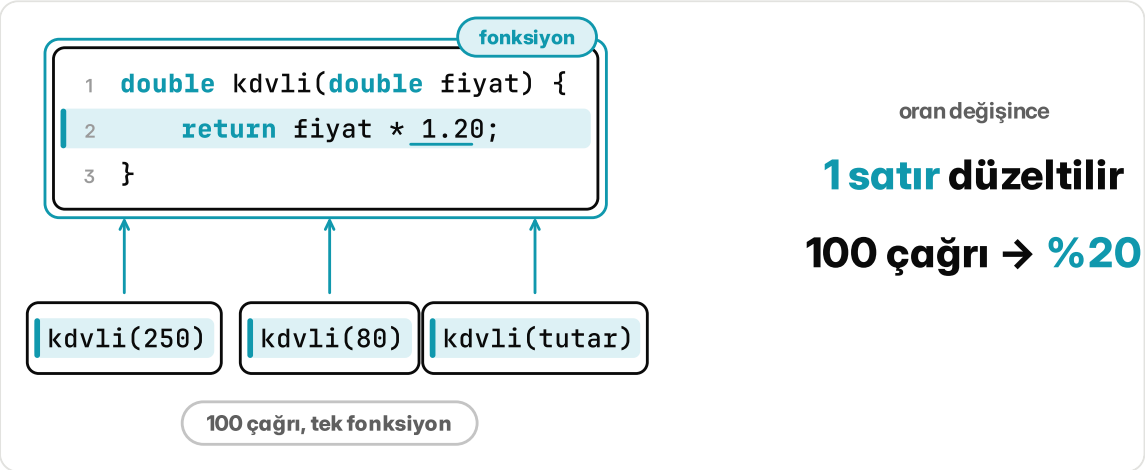
Bu bölümde aynı hesabı bir kez yazıp her yerde kullanmayı sağlayan fonksiyonları öğreneceksin: tanımın parçalarını, void'i, prototipi ve hazır kütüphane fonksiyonlarını. Ardından fonksiyona verilen değerin neden kopya olduğunu, değişkenlerin bellekte nerede doğup nerede öldüğünü, kapsamı ve ömrü göreceksin.

100 yerde aynı hesap

videoda 18:53

Bir mağaza programı, KDV dahil fiyatı kodun 100 ayrı yerinde hesaplıyor: fis = fiyat * 1.18;, sepet = toplam * 1.18;, fatura = tutar * 1.18; ve 97 satır daha. 2023'te genel KDV oranı yüzde 18'den yüzde 20'ye çıktı. Programcı 100 satırı tek tek mi düzeltmeli?

Hesabı bir kez yazıp ona bir ad veririz; gereken her yerde o adla çağırırız. Adı olan ve çağrılınca çalışan bu kod parçasına **fonksiyon** denir. Oran değişince yalnız fonksiyonun içindeki tek satırı düzeltiriz; 100 çağrının hepsi yeni oranla hesaplar.



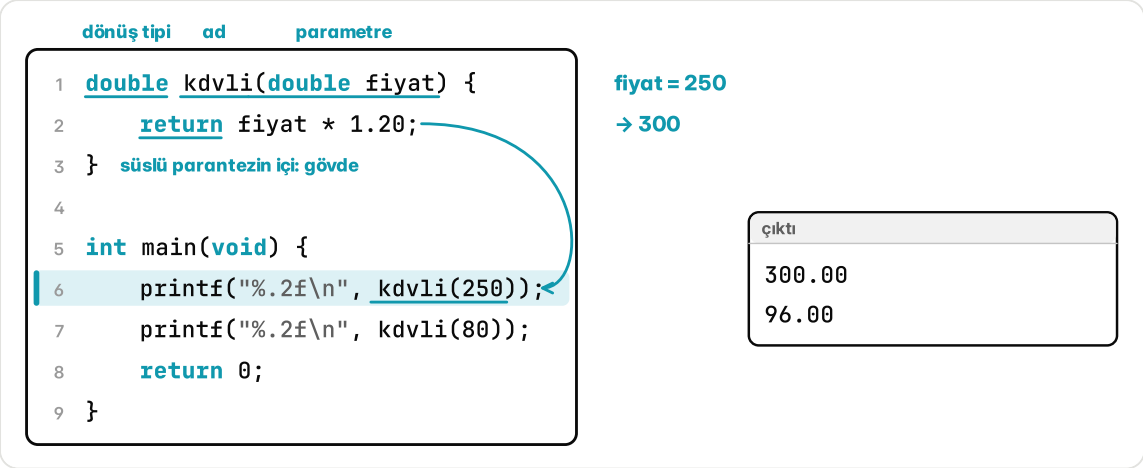
Şekil 4.1 kdvli bir kez yazılır, 100 yerden çağrılır; oran değişince yalnız 1 satır düzeltilir.

```
#include <stdio.h>

double kdvli(double fiyat) {
    return fiyat * 1.20;
}

int main(void) {
    printf("%.2f\n", kdvli(250));
    printf("%.2f\n", kdvli(80));
    return 0;
}
```

Çıktı:

```
300.00
96.00
```

Tanımın ilk sözcüğü, fonksiyonun geri vereceği değerin tipidir; burada kesirli sayı, yani double. Ardından fonksiyonun adı (kdvli), parantez içinde de dışarıdan aldığı değer gelir. Bu değere **parametre** denir: double fiyat. Süslü parantezin içi fonksiyonun gövdesidir. Gövdedeki return, sonucu fonksiyonu çağıran yere geri verir. kdvli(250) çağrılınca fiyat 250 olur, 250 × 1,20 = 300 hesaplanır ve bu sayı printf'e gider; printf de onu iki ondalık basamakla (%.2f) yazar. 80 için sonuç 96'dır.



Şekil 4.2 Tanımın parçaları: dönüş tipi double, ad kdvli, parametre double fiyat ve gövde; kdvli(250) → 300.

Her fonksiyon değer döndürmez. Ekrana yalnız bir çizgi çeken fonksiyonun tipine void yazarız: geri verdiği bir şey yoktur, gövdesinde return da gerekmez. Parantezin içindeki void ise dışarıdan hiçbir değer almadığını söyler.

```
void cizgi(void) {
    printf("----------\n");
}

int main(void) {
    cizgi();
    printf("Fiş\n");
    cizgi();
    return 0;
}
```

Çıktı:

```
----------
Fiş
----------
```

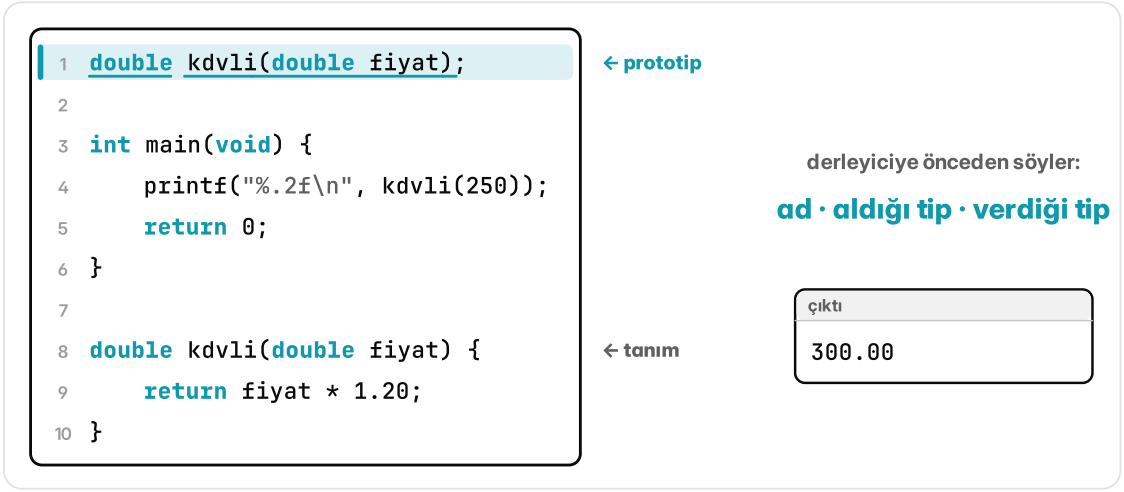
Fonksiyonu main'in altına yazarsak ne olur?

```
#include <stdio.h>

int main(void) {
    printf("%.2f\n", kdvli(250));
    return 0;
}

double kdvli(double fiyat) {
    return fiyat * 1.20;
}
```

Bu kod derlenmez. Derleyici dosyayı yukarıdan aşağı okur; main'deki çağrıyı gördüğü anda kdvli adını henüz tanımaz. gcc bu satırda "error: implicit declaration of function 'kdvli'" der; clang da benzer bir hata verir. Çözüm, fonksiyonun ilk satırını dosyanın başına bir kez daha yazıp sonuna noktalı virgül koymaktır:

```
double kdvli(double fiyat);
```

Buna **prototip** denir: derleyiciye fonksiyonun adını, aldığı ve geri verdiği tipleri önceden bildirir. Prototipi başa eklenen program derlenir ve 300.00 yazar; fonksiyonun asıl tanımı yine main'in altında durabilir.



Şekil 4.3 Başa yazılan prototip derleyiciye adı, aldığı ve verdiği tipi önceden söyler; tanım main'in altında kalır.

printf'i de biz yazmadık. Dosyanın başına eklediğimiz stdio.h başlık dosyası, onun prototipini getirir. Karekök ve üs alma gibi hesaplar da math.h başlık dosyasındaki hazır fonksiyonlardadır:

```c
#include <stdio.h>
#include <math.h>

int main(void) {
    printf("%.0f\n", sqrt(9));
    printf("%.0f\n", pow(2, 10));
    return 0;
}
```

Çıktı:

```
3
1024
```

sqrt karekökü, pow üssü hesaplar: 9'un karekökü 3, 2 üzeri 10 ise 1.024'tür. %.0f sonucu ondalık basamaksız yazar. Bu hazır fonksiyonlar da bizim yazdıklarımız gibi çalışır: değeri alır, hesaplar, sonucu geri verir.

Fonksiyon bitince?

videoda 20:29

Bir fonksiyonun içinde tanımladığımız değişkenler, fonksiyon bitince ne olur? Önce şaşırtıcı bir sonuca bakalım:

```c
#include <stdio.h>

void ikile(int sayi) {
    sayi = sayi * 2;
    printf("%d\n", sayi);
}

int main(void) {
    int puan = 10;
    ikile(puan);
    printf("%d\n", puan);
    return 0;
}
```

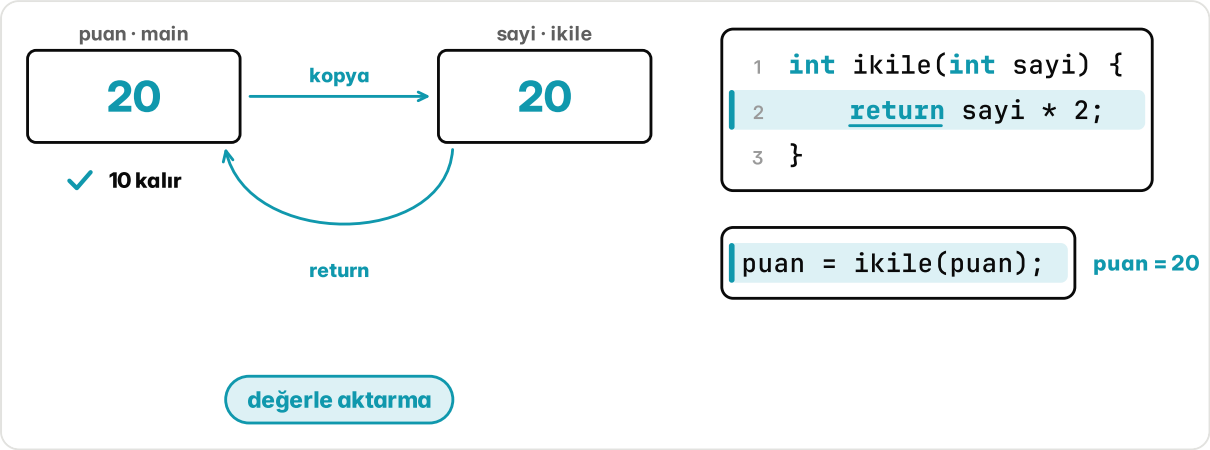
Çıktı:

```
20
10
```

puan'ı ikiye katlayan fonksiyona verdik. Fonksiyon içeride 20 yazdı, ama main'e dönüp puan'ı yazdırınca yine 10 çıktı. Çünkü C fonksiyona değişkenin kendisini değil, değerinin bir kopyasını verir; buna **değerle aktarma** denir. ikile kendi kopyası olan sayi'yı 20 yaptı; puan ise 10 olarak kaldı. Sonucu istiyorsak fonksiyon onu return ile geri verir, biz de puan'a yazarız:

```
int ikile(int sayi) {
    return sayi * 2;
}
```

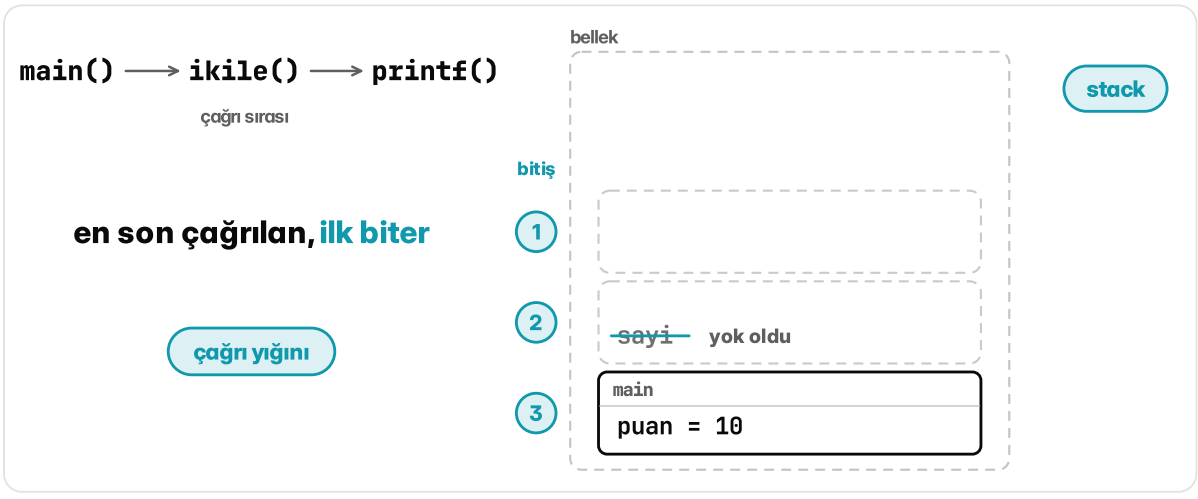
Bu sürümde main'de puan = ikile(puan); yazınca puan 20 olur.



Şekil 4.4 ikile, puan'ın kopyası olan sayi ile çalışır; puan 10 kalır, sonuç return ile alınınca puan 20 olur.

Peki kopya bellekte nerede durur? Program, her fonksiyon çağrısı için bellekte bir çerçeve açar; bu çerçevelerin durduğu bölgeye **stack** denir. main'in çerçevesinde puan durur. ikile çağrılınca bu çerçevenin üstüne yeni bir çerçeve kurulur; fonksiyonun parametresi ve içinde tanımlanan değişkenler orada doğar. Fonksiyonun içinde tanımlanan bu değişkenlere **yerel değişken** denir.

Fonksiyon başka bir fonksiyonu çağırırsa onun çerçevesi de en üste kurulur. ikile, printf'i çağırdığında zincir main → ikile → printf olur ve stack'te üç çerçeve üst üste durur. Biten fonksiyonun çerçevesi kaldırılır, içindeki değişkenler de onunla birlikte yok olur: önce printf, sonra ikile biter; sayi da ikile'nin çerçevesiyle birlikte gider. Çerçeveler üst üste konmuş tabaklar gibi üstten kalkar: en son çağrılan ilk biter. Bu yüzden bu yapıya **çağrı yığını** (call stack) denir.

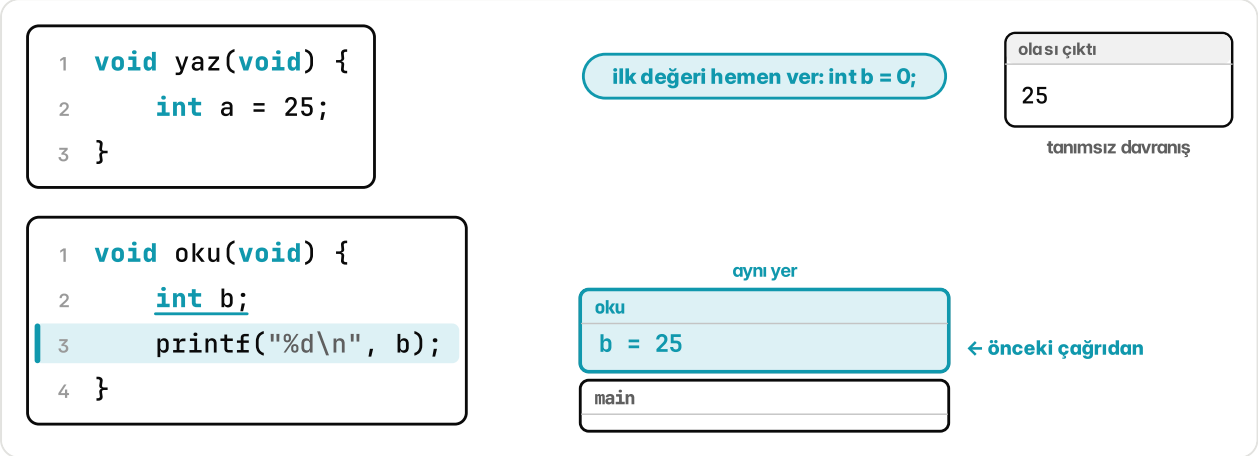


Şekil 4.5 Stack'te main, ikile ve printf çerçeveleri üst üste kurulur; bitiş sırası printf, ikile, main: en son çağrılan ilk biter.

Kaldırılan çerçevenin içi silinmez; sonraki çağrının çerçevesi aynı yere kurulur. Bunun sonucunu şu programda görürüz:

```
void yaz(void) {
    int a = 25;
}

void oku(void) {
    int b;
    printf("%d\n", b);
}

int main(void) {
    yaz();
    oku();
    return 0;
}
```

oku'nun çerçevesi, az önce kalkan yaz'ın çerçevesinin yerine kurulur. b'ye ilk değer verilmediği için onu okumak tanımsız davranıştır; ama çoğu bilgisayarda (iyileştirme kapalıyken) b, a'nın eski yerine düşer ve ekrana 25 çıkar. Bu örnekte de 25 çıktı. Yerel değişkene ilk değerini hemen vermek (int b = 0;) bu yüzden iyi bir alışkanlıktır.



Şekil 4.6 oku'nun çerçevesi yaz'ınkinin yerine kurulur; ilk değeri verilmemiş b'de önceki çağrıdan kalan 25 görünebilir.

Kaç kez çağrıldı?

videoda 21:45

Bir sitenin ziyaretçi sayacını düşünelim: her ziyarette bir fonksiyon çağrılıyor ve sayıyı bir artırıyor.

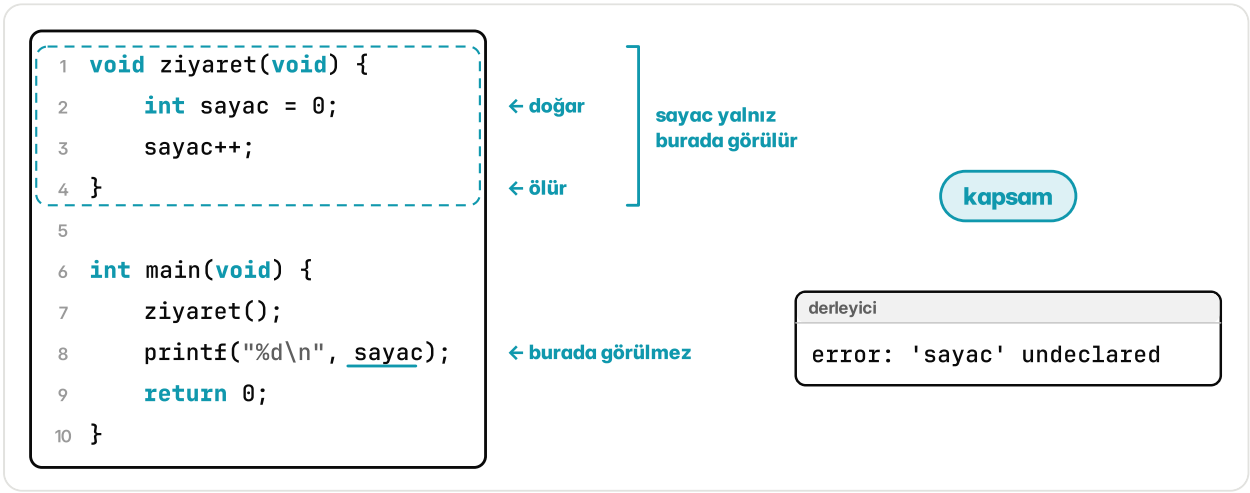
```c
#include <stdio.h>

void ziyaret(void) {
    int sayac = 0;
    sayac++;
    printf("%d\n", sayac);
}

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

Çıktı:

```
1
1
1
```

Üç ziyaretten sonra ekranda hep 1 var, çünkü sayac her çağrıda 0'dan başlıyor. Sayaç nerede durmalı?

sayac bir yerel değişkendir: çağrı başlayınca doğar, çağrı bitince ölür. Ayrıca yalnız tanımlandığı süslü parantezlerin içinden görülür. main onu adıyla kullanmaya kalkarsa, örneğin printf("%d\n", sayac); yazarsa, derleyici bu adı tanımaz: gcc "error: 'sayac' undeclared" der. Bir değişkenin görülebildiği bu bölgeye **kapsam** denir.



Şekil 4.7 sayac yalnız ziyaret'in içinde görülür; main'de kullanılınca derleyici 'sayac' undeclared der.

Bütün fonksiyonların dışında, dosyanın başında tanımlanan değişken ise **global değişkendir**. Her fonksiyon onu görür ve program bitene kadar yaşar:

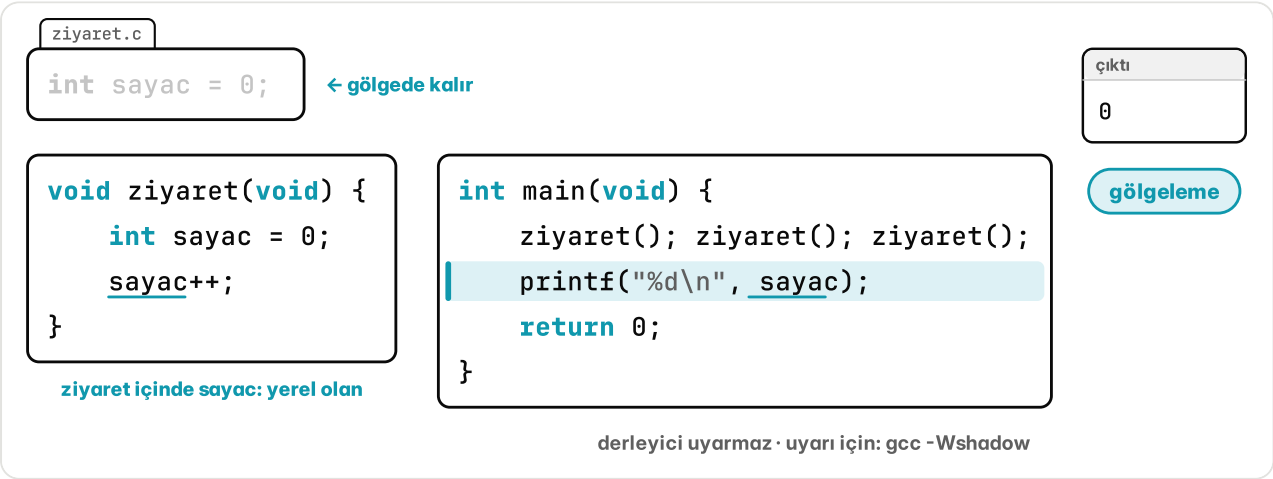
```
#include <stdio.h>

int sayac = 0;

void ziyaret(void) {
    sayac++;
}

int main(void) {
    ziyaret(); ziyaret(); ziyaret();
    printf("%d\n", sayac);
    return 0;
}
```

Bu program 3 yazar. Ama her fonksiyon global değişkeni değiştirebildiği için, büyük bir programda yanlış değeri kimin yazdığını bulmak zorlaşır.

Bir tuzak daha var. ziyaret'in içinde aynı adla bir yerel değişken tanımlarsak, o fonksiyonda sayac adı yereli gösterir:

```
void ziyaret(void) {
    int sayac = 0;
    sayac++;
}
```

Global sayac gölgede kalır: fonksiyon kendi yerel sayacını artırır, global olan 0'da kalır ve main ekrana 0 yazar. Buna **gölgeleme** denir. gcc varsayılan ayarlarla, hatta -Wall ve -Wextra ile bile uyarı vermez; uyarı için -Wshadow seçeneği gerekir.



Şekil 4.8 ziyaret'teki yerel sayac global olanı gölgeler; global sayac 0'da kalır ve çıktı 0 olur.

En temiz çözüm, yerel değişkenin başına static yazmaktır:

```
void ziyaret(void) {
    static int sayac = 0;
    sayac++;
    printf("%d\n", sayac);
}
```

Çıktı:

```
1
2
3
```

static yerel değişken program başlarken bir kez kurulur ve ilk değerini o zaman alır; fonksiyon bitince yok olmaz, değerini sonraki çağrıya taşır. Yine de yalnız o fonksiyonun içinden görülür; başka bir fonksiyon onu bozamaz.

Kısacası her değişkenin iki özelliği vardır: nereden görüldüğü, yani kapsamı; ne kadar yaşadığı, yani **ömrü**.

- Yerel değişken: kapsamı dar (yalnız kendi fonksiyonu), ömrü kısa (tek çağrı).
- Global değişken: kapsamı geniş (bütün dosya), ömrü uzun (program boyunca).
- static yerel değişken: kapsamı dar (yalnız kendi fonksiyonu), ömrü uzun (program boyunca).



Şekil 4.9 Yerel, global ve static yerel değişkenin kapsamı (nereden görülür) ve ömrü (ne kadar yaşar).

Özet

- Fonksiyon, adı olan ve çağrılınca çalışan kod parçasıdır; tanımı dönüş tipi, ad, parametreler ve gövdeden oluşur, return sonucu geri verir.
- void dönüş tipi değer döndürmediğini, parantezdeki void parametre almadığını söyler.
- Derleyici dosyayı yukarıdan aşağı okur; aşağıda tanımlanan fonksiyon için başa noktalı virgülle biten bir prototip yazılır.
- C fonksiyona değerin kopyasını verir; fonksiyon kopyayı değiştirir, çağıranın değişkeni değişmez.
- Her çağrı stack'te bir çerçeve açar; yerel değişkenler orada doğar ve çerçeve kalkınca yok olur, en son çağrılan ilk biter.
- Yerel değişken dar kapsamlı ve kısa ömürlü, global geniş ve uzun, static yerel ise dar kapsamlı ama uzun ömürlüdür.

Sorular

1 **double kdvli(double fiyat) tanımında en baştaki double neyi söyler?**

A Parametrenin tipini

B Fonksiyonun kaç parametre aldığını

C Fonksiyonun geri vereceği değerin tipini

D Fonksiyonun main'den önce çalışacağını

2 **math.h eklenmiş bir programda printf("%.0f\n", pow(3, 4)); satırı ekrana ne yazar?**

Cevap:

3 **Aşağıdaki fonksiyon tanımlıyken printf("%d\n", ucle(7) + 1); satırı ne yazar?**

```
int ucle(int x) {
    return x * 3;
}
```

Cevap:

4 **Aşağıdaki program derlenmez. Neden?**

```
#include <stdio.h>

int main(void) {
    printf("%d\n", kare(5));
    return 0;
}

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

A Derleyici dosyayı yukarıdan aşağı okur; çağrıyı gördüğünde kare adını henüz tanımaz.

B kare fonksiyonunun return satırı yanlıştır.

C printf'in içinde fonksiyon çağrılamaz.

D int döndüren fonksiyonlar main'den önce çağrılamaz.

5 **Aşağıdaki program ekrana ne yazar?**

```
#include <stdio.h>

void artir(int n) {
    n = n + 5;
}

int main(void) {
    int x = 3;
    artir(x);
    printf("%d\n", x);
    return 0;
}
```

Cevap: _____

6 **main fonksiyonu a'yı, a da b'yi çağırıyor. Stack'teki çerçevelerden ilk hangisi kaldırılır?**

A main'in çerçevesi

B a'nın çerçevesi

C b'nin çerçevesi

D Üçü aynı anda kaldırılır.

7 **Aşağıdaki fonksiyon 4 kez çağrılırsa ekrana ne yazılır?**

```
void say(void) {
    static int n = 0;
    n++;
    printf("%d ", n);
}
```

A 1 1 1 1

B 0 1 2 3

C 4 4 4 4

D 1 2 3 4

8 Aşağıdaki program ekrana ne yazar?

```
#include <stdio.h>

int toplam = 10;

void ekle(void) {
    int toplam = 0;
    toplam = toplam + 5;
}

int main(void) {
    ekle(); ekle();
    printf("%d\n", toplam);
    return 0;
}
```

A 20

B 10

C 5

D 0

PROGRAMLAMAYA GİRİŞ · 1. DÖNEM

5 İşaretçiler

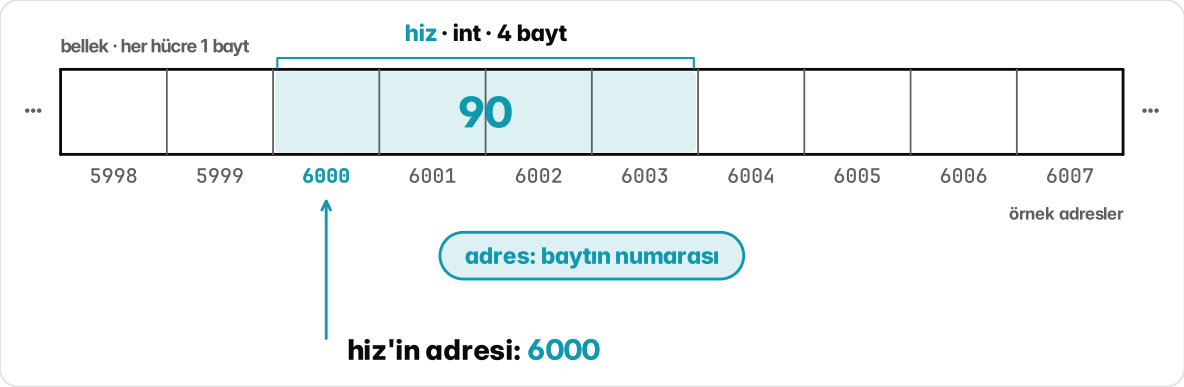
Bu bölümde değişkenlerin bellekteki adresini ve adres tutan değişkenleri, yani işaretçileri öğreneceksin. Bir fonksiyonun kendisine verilen değişkeni nasıl değiştirebildiğini, işaretçiye sayı eklemenin ne demek olduğunu, dizilerle işaretçilerin bağını, NULL'ı ve const'u göreceksin.

Değer bellekte nerede?

videoda 23:14

int hiz = 90; diye bir değişken tanımladık. Değerini printf ile ekrana yazdırmayı biliyoruz; peki bu 90 bellekte tam olarak nerede duruyor?

Bellek, baytlardan oluşan uzun bir sıradır ve her baytın bir numarası vardır; bu numaraya **adres** denir. int çoğu bilgisayarda 4 bayt yer kaplar. Değişkenin adresi bu 4 bayttan ilkinin numarasıdır: hiz 6000, 6001, 6002 ve 6003 numaralı baytları kaplıyorsa adresi 6000'dir. 6000 anlatım için seçilmiş örnek bir adrestir; gerçek adresler çok daha büyük sayılardır.



Şekil 5.1 Bellek 1 baytlık hücrelerden oluşur; 4 baytlık hiz 6000-6003'ü kaplar ve adresi 6000'dir (örnek adresler).

Değişkenin adının önüne & koyunca değerini değil, adresini alırız. printf adresi %p ile, onaltılık tabanda yazar:

```
#include <stdio.h>

int main(void) {
    int hiz = 90;
    printf("%d\n", hiz);
    printf("%p\n", &hiz);
    return 0;
}
```

Çıktı:

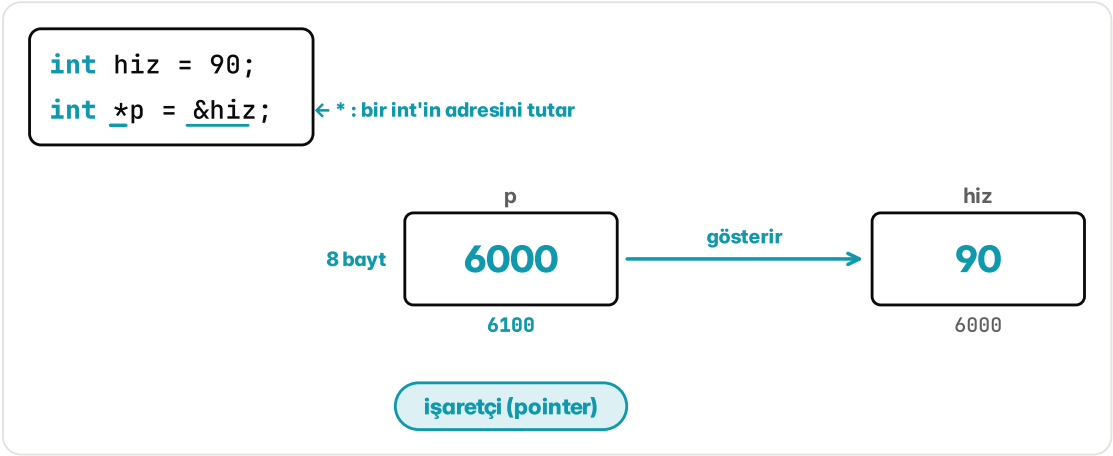
```
90
000000C8FA9FF904
```

İkinci satır, bu çalıştırmada Windows'ta alınan adrestir; 64 bit Linux'ta aynı program 0x7ffd5e8c3a4c gibi bir biçimde yazar. Adres her çalıştırmada farklı çıkabilir, ama değişken yaşadığı sürece hep aynı yerde durur.

Adres de bir sayıdır; o hâlde bir değişkende saklanabilir. Böylece bir değere adıyla değil, yeriyle ulaşabiliriz. Adres tutan değişkene **işaretçi** (pointer) denir:

```
int *p = &hiz;
```

Tanımda tipin yanına yazılan yıldız, p'nin bir tam sayı değil, bir tam sayının adresini tuttuğunu söyler. p'ye ilk değer olarak hiz'in adresini verdik; p artık hiz'i gösterir. p de bir değişkendir: kendi adresi vardır ve 64 bitlik bilgisayarda 8 bayt yer tutar.



Şekil 5.2 p, hiz'in adresi olan 6000'i tutar ve hiz'i gösterir; p'nin kendi adresi 6100'dür ve 8 bayt yer kaplar.

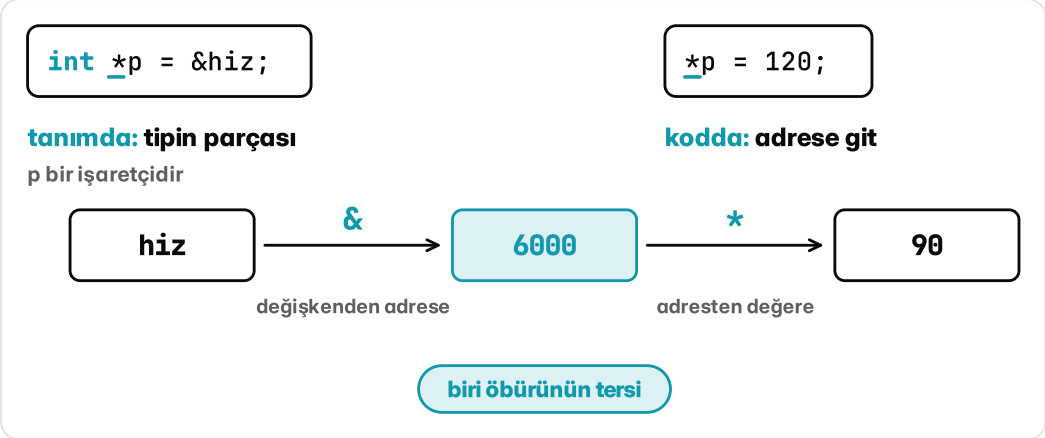
İşaretçinin önüne yıldız koyunca tuttuğu adrese gidip oradaki değere ulaşırız: *p 90 verir. *p'ye 120 yazınca hiz de 120 olur, çünkü p'nin gösterdiği yer hiz'in ta kendisidir:

```
#include <stdio.h>

int main(void) {
    int hiz = 90;
    int *p = &hiz;
    printf("%d\n", *p);
    *p = 120;
    printf("%d\n", hiz);
    printf("%zu\n", sizeof p);
    return 0;
}
```

Çıktı:

```
90
120
8
```

Yıldızın iki işi karışmasın. Tanımda (int *p) tipin parçasıdır ve bunun bir işaretçi olduğunu söyler; kodda (*p) ise "bu adrese git" demektir. & değişkenden adrese, yıldız adresten değere götürür; biri öbürünün tersidir. hiz'den &hiz ile 6000'e, 6000'den yıldızla yeniden 90'a varırız.



Şekil 5.3 Tanımdaki yıldız tipin parçasıdır, koddaki yıldız adrese gider; & değişkenden adrese, yıldız adresten değere götürür.

Yer değiştir

videoda 24:38

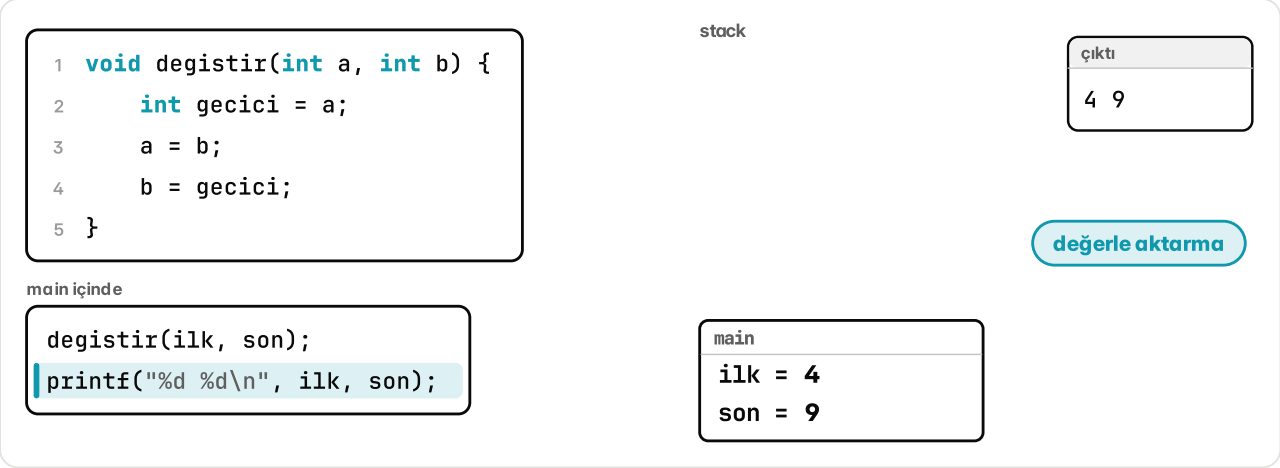
Bir fonksiyon, kendisine verilen değişkeni nasıl değiştirebilir? İki değişkenin değerlerini yer değiştiren bir fonksiyon yazalım; sıralama yaparken bu işe sürekli ihtiyaç duyulur. ilk 4, son 9; fonksiyondan sonra ilk 9, son 4 olmalı.

```
#include <stdio.h>

void degistir(int a, int b) {
    int gecici = a;
    a = b;
    b = gecici;
}

int main(void) {
    int ilk = 4, son = 9;
    degistir(ilk, son);
    printf("%d %d\n", ilk, son);
    return 0;
}
```

Çıktı:

```
4 9
```

Fonksiyon, geçici bir değişkenin yardımıyla iki değeri yer değiştiriyor; ama main'e dönünce ekrana yine 4 ve 9 çıkıyor. Çünkü C fonksiyona değişkenlerin kendisini değil, değerlerinin kopyasını verir; yer değiştiren yalnız kopyalardır. a ve b fonksiyonun kendi çerçevesinde doğar ve fonksiyon bitince onunla birlikte yok olur.



Şekil 5.4 degistir yalnız a ve b kopyalarını yer değiştirir; main'deki ilk 4, son 9 kalır.

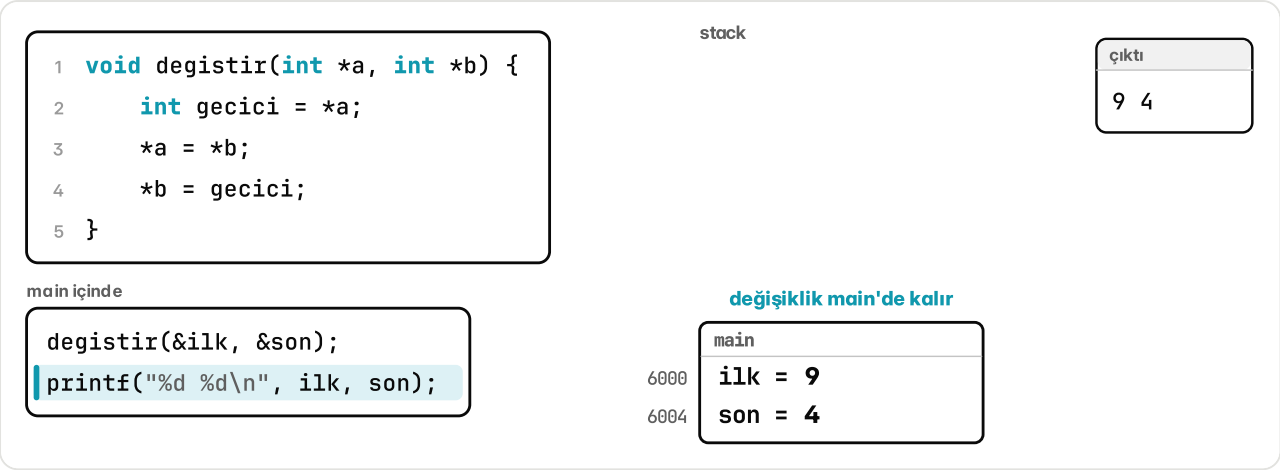
Çözüm, fonksiyona değerleri değil adresleri vermektir. Parametreler işaretçi olur; çağırırken ilk ve son'un önüne & koyarız:

```
#include <stdio.h>

void degistir(int *a, int *b) {
    int gecici = *a;
    *a = *b;
    *b = gecici;
}

int main(void) {
    int ilk = 4, son = 9;
    degistir(&ilk, &son);
    printf("%d %d\n", ilk, son);
    return 0;
}
```

Çıktı:

```
9 4
```

Fonksiyon adreslerin de kopyasını alır; ama adresin kopyası da aynı yeri gösterir. ilk 6000'de, son 6004'te duruyorsa a 6000'i, b 6004'ü tutar. Fonksiyonun içinde *a, main'deki ilk değişkeninin ta kendisidir. Önce ilk'in değeri (4) gecici'ye alınır; sonra son'un değeri (9) ilk'e, gecici'deki 4 de son'a yazılır. Fonksiyon bitip çerçevesi kalksa da main'deki değerler değişmiş kalır.

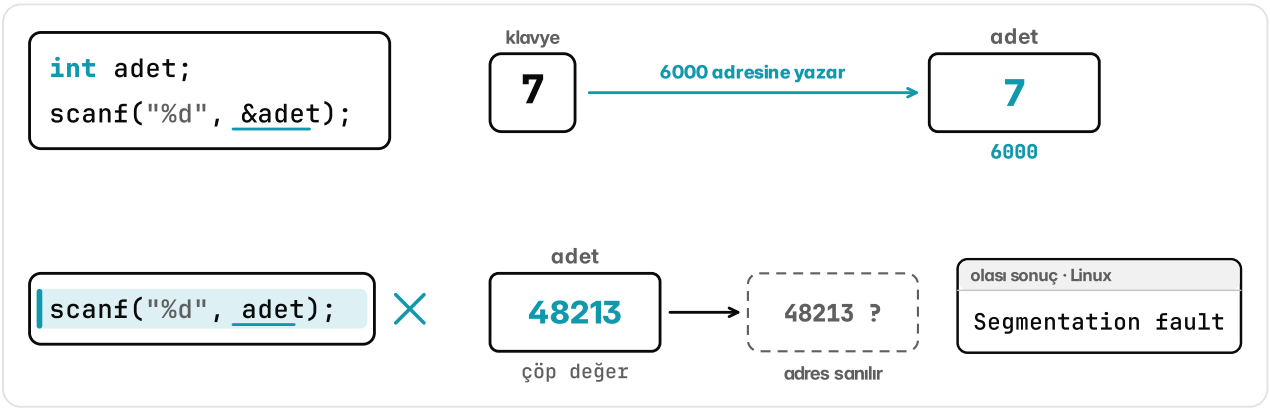


Şekil 5.5 a ve b, 6000 ve 6004 adreslerini tutar; değişiklik main'de kalır ve çıktı 9 4 olur.

scanf'in neden & istediği de buradan çıkar. scanf klavyeden okuduğu sayıyı bizim değişkenimize yazmalıdır; bunun için değişkenin değerini değil, adresini ister: scanf("%d", &adet);. & unutulursa ne olur?

```
#include <stdio.h>

int main(void) {
    int adet;
    scanf("%d", adet);
    printf("%d\n", adet);
    return 0;
}
```

scanf, adet'in içindeki rastgele değeri adres sanıp oraya yazmaya kalkar. Derleyici bu satırda uyarı verir (%d bir int adresi bekler, ona int verilmiştir) ama programı derler. Sonuç tanımsız davranıştır ve program çökebilir; bu örnekte program, Windows'un erişim ihlali hatasıyla (0xC0000005) çöktü.



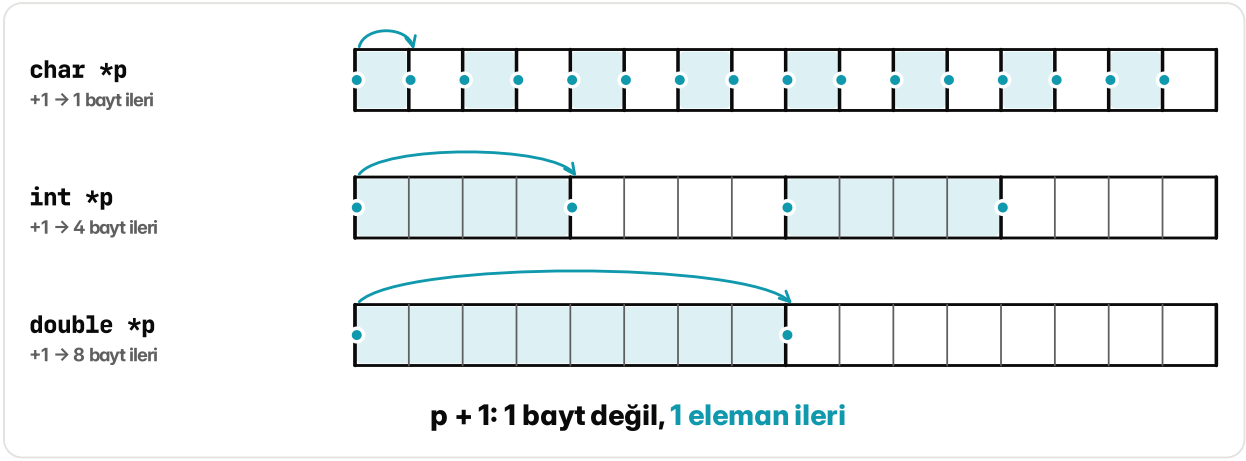
Şekil 5.6 & ile scanf okuduğu 7'yi adet'in adresine yazar; & unutulursa adet'teki çöp değer adres sanılır ve program çökebilir.

p + 1 kaç?

videoda 25:55

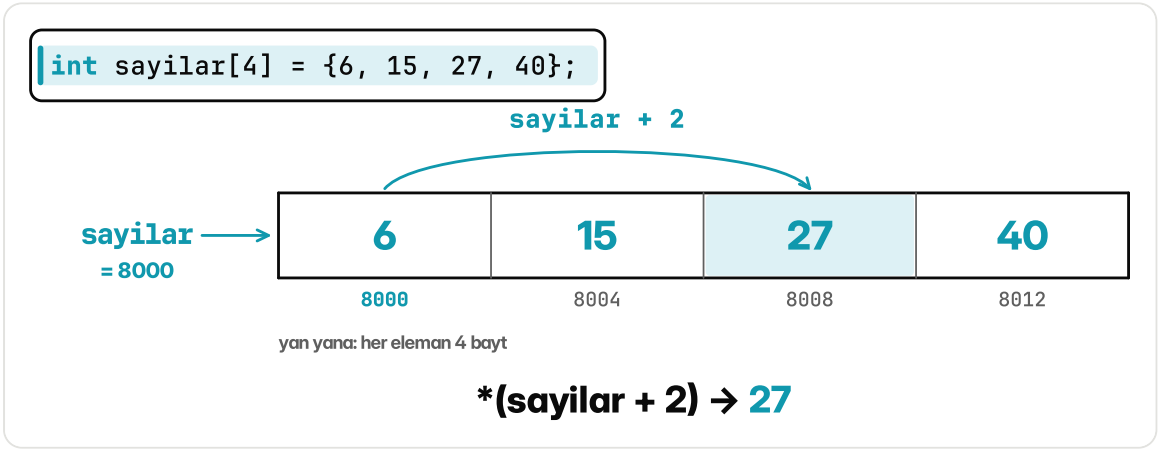
Bir işaretçi 8000 numaralı adresi tutuyor. Ona 1 eklersek yeni adres kaç olur? İşaretçi bir tam sayıyı gösteriyorsa cevap 8001 değil, 8004'tür.

Çünkü C, işaretçiye eklenen sayıyı bayt olarak değil, eleman olarak sayar. int 4 bayt olduğu için sonraki tam sayı 4 bayt ileridedir; char işaretçisi 1, double işaretçisi 8 bayt ilerler. Böylece p + 1 her zaman bir sonraki elemanın başına çıkar. Buna **işaretçi aritmetiği** denir.



Şekil 5.7 p + 1, int işaretçisinde 4, char işaretçisinde 1, double işaretçisinde 8 bayt ilerler: bir bayt değil, bir eleman ileri.

Bu kural asıl dizilerde işe yarar, çünkü dizinin elemanları bellekte yan yana durur. int sayilar[4] = {6, 15, 27, 40}; dizisi 8000'den başlıyorsa elemanlar 8000, 8004, 8008 ve 8012'dedir. Dizinin adı bir ifadede kullanılınca ilk elemanının adresine dönüşür: sayilar, 8000 demektir. sayilar + 2 ilk elemandan 2 eleman ilerideki adrestir, yani 8008; önüne yıldız koyunca 27 çıkar.



Şekil 5.8 sayilar 8000'den başlar ve elemanlar 4'er bayt arayla durur; sayilar + 2 = 8008 ve *(sayilar + 2) → 27.

Köşeli parantezle sayilar[2] yazmak tam olarak bunun kısaltmasıdır: derleyici onu adrese 2 ekleyip yıldızla okumaya, yani *(sayilar + 2)'ye çevirir. İndisin 0'dan başlaması da buradan gelir: indis, ilk elemandan kaç adım uzakta olduğumuzu söyler; ilk eleman *(sayilar + 0)'dır.

Aşağıdaki program bunları sınar. int, char ve double ile başlayan üç printf satırı, iki komşu elemanın adresleri arasındaki farkı bayt olarak yazdırır:

```
#include <stdio.h>

int main(void) {
    int sayilar[4] = {6, 15, 27, 40};
    int *p = sayilar;
    char harf[2];
    double kesir[2];
    printf("%p\n%p\n", (void *)p, (void *)(p + 1));
    printf("int: %d\n", (int)((char *)(p + 1) - (char *)p));
    printf("char: %d\n", (int)((char *)(harf + 1) - (char *)harf));
    printf("double: %d\n", (int)((char *)(kesir + 1) - (char *)kesir));
    printf("%d %d %d\n", *(sayilar + 2), sayilar[2], *(p + 2));
    printf("%d\n", p == &sayilar[0]);
    return 0;
}
```

Çıktı:

```
00000080837FFCC0
00000080837FFCC4
int: 4
char: 1
double: 8
27 27 27
1
```

İlk iki satır bu çalıştırmadaki p ve p + 1 adresleridir: onaltılık C0'dan C4'e, yani 4 bayt ilerlenmiştir. Son satırdaki 1, p'nin dizinin ilk elemanının adresine eşit olduğunu söyler.

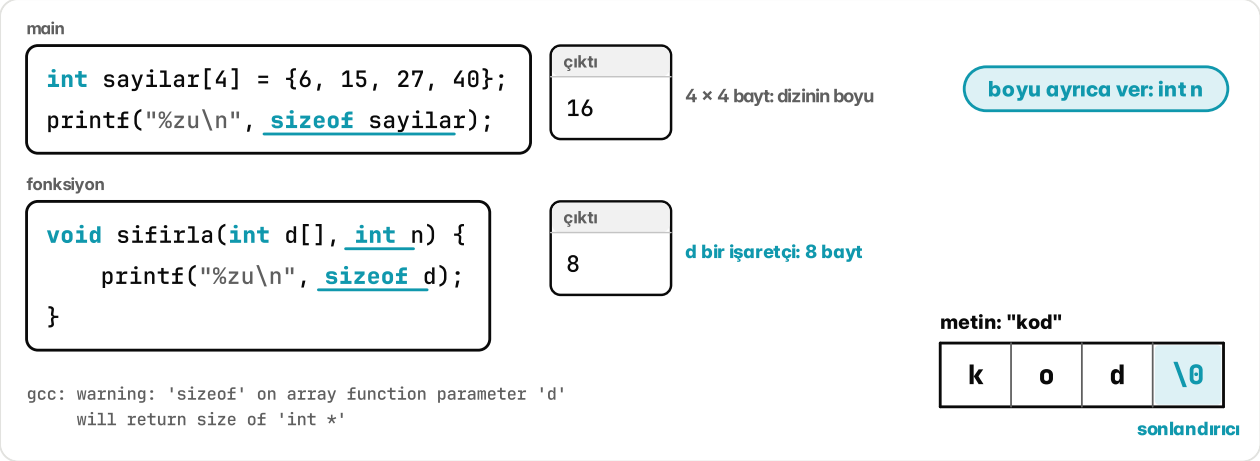
Diziyi bir fonksiyona verince fonksiyona dizinin kopyası değil, ilk elemanının adresi gider. Bu yüzden fonksiyon elemanları sıfırlarsa main'deki dizi de sıfırlanır:

```
void sifirla(int d[], int n) {
    printf("%zu\n", sizeof d);
    for (int i = 0; i < n; i++) {
        d[i] = 0;
    }
}

int main(void) {
    int sayilar[4] = {6, 15, 27, 40};
    printf("%zu\n", sizeof sayilar);
    sifirla(sayilar, 4);
    printf("%d\n", sayilar[0]);
    return 0;
}
```

Çıktı:

```
16
8
0
```

Ama adresle birlikte boy bilgisi kaybolur. main'de sizeof sayilar dizinin 16 baytını verir; fonksiyonun içinde ise sizeof d işaretçinin 8 baytını verir, çünkü int d[] diye yazılan parametre aslında bir int işaretçisidir. Derleyici bu satırda uyarır; gcc'nin uyarısı "'sizeof' on array function parameter 'd' will return size of 'int *'" der. Bu yüzden fonksiyona diziyle birlikte eleman sayısını da ayrı bir parametre olarak veririz: sifirla'daki n budur. Metinlerde buna gerek kalmaz; sondaki sonlandırıcı, metnin nerede bittiğini zaten söyler.



Şekil 5.9 main'de sizeof sayilar 16, fonksiyonda sizeof d 8 verir; d bir işaretçidir, bu yüzden boy ayrıca verilir.

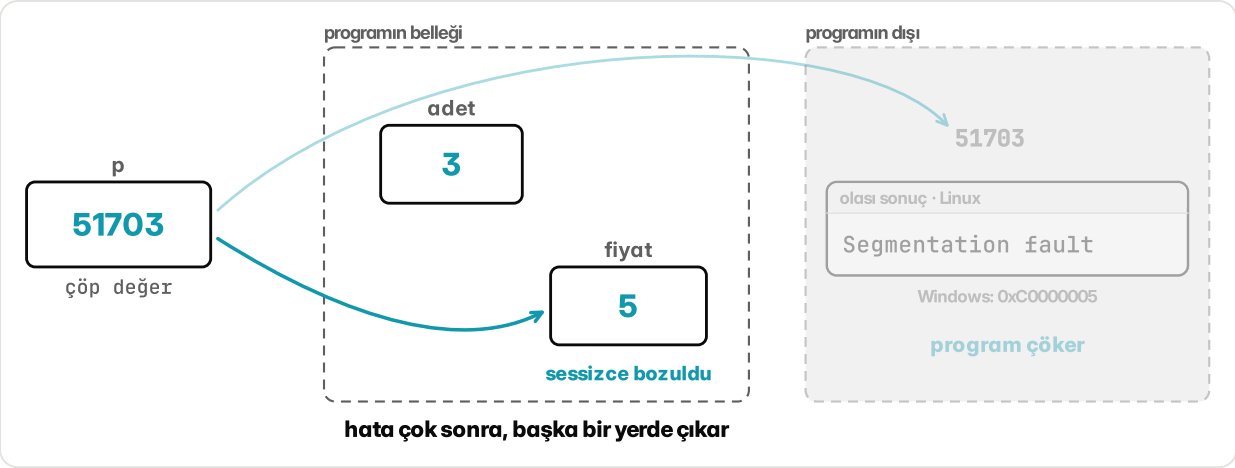
Neden çöktü?

videoda 27:16

Program derleniyor, ama çalışınca daha ilk satırlarda çöküyor; neden? Sebep çoğu zaman ilk değeri verilmemiş bir işaretçidir:

```
#include <stdio.h>

int main(void) {
    int *p;
    *p = 5;
    printf("%d\n", *p);
    return 0;
}
```

p'nin içinde, o bellek yerinden kalmış rastgele bir sayı vardır; *p = 5 ile oraya yazınca nereye yazdığımızı bilemeyiz. O adres programa ait değilse işletim sistemi programı anında durdurur, yani program çöker: Linux'ta "Segmentation fault" yazar, Windows'ta erişim ihlali hatası verir. Daha kötüsü, adres programın kendi belleğine düşerse çökme olmaz; başka bir değişken sessizce bozulur ve hata çok sonra bambaşka bir yerde ortaya çıkar. Bu örnekte program çökmedi ve 5 yazdı: p'deki rastgele değer, programın kendi belleğinden bir yeri gösteriyordu.



Şekil 5.10 İlk değeri verilmemiş p'deki çöp değer programın dışını gösterirse program çöker, içini gösterirse başka bir değişken sessizce bozulur.

Bu yüzden henüz bir yeri göstermeyen işaretçiye NULL veririz. **NULL**, hiçbir yeri göstermediği bilinen özel adrestir. NULL işaretçiyi kullanmak da programı çökertir, ama hata rastgele bir yerde değil, tam o satırda çıkar. Kullanmadan önce de işaretçinin NULL olup olmadığına bakarız:

```
#include <stdio.h>

int main(void) {
    int *p = NULL;
    if (p != NULL) {
        *p = 5;
    } else {
        printf("p bir yeri göstermiyor\n");
    }
    return 0;
}
```

Çıktı:

```
p bir yeri göstermiyor
```

Boş adres fikrini 1965'te ALGOL W diline koyan kişi Tony Hoare'du. Hoare 2009'daki bir konuşmasında buna "milyar dolarlık hatam" dedi; yol açtığı sayısız çökmeyi ve güvenlik açığını kastediyordu.

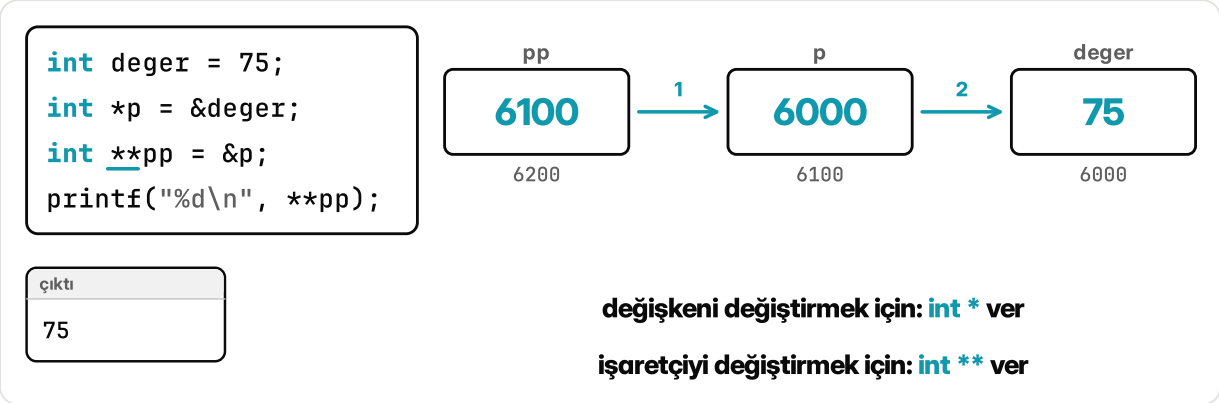
İşaretçi de bir değişken olduğuna göre onun da adresi vardır. Bu adresi tutan değişkene **işaretçinin işaretçisi** denir ve tanımında iki yıldız bulunur:

```c
#include <stdio.h>

int main(void) {
    int deger = 75;
    int *p = &deger;
    int **pp = &p;
    printf("%d\n", **pp);
    return 0;
}
```

Çıktı:

```
75
```

pp, p'nin adresini tutar; p de deger'in adresini. pp'nin önündeki iki yıldız önce p'ye, oradan da deger'in kendisine gider ve 75 verir. Bir değişkeni fonksiyonda değiştirmek için adresini verdiğimiz gibi, bir işaretçiyi fonksiyonda değiştirmek için de onun adresini veririz.



Şekil 5.11 pp, p'nin adresini (6100), p de deger'in adresini (6000) tutar; zincirin sonunda 75 durur.

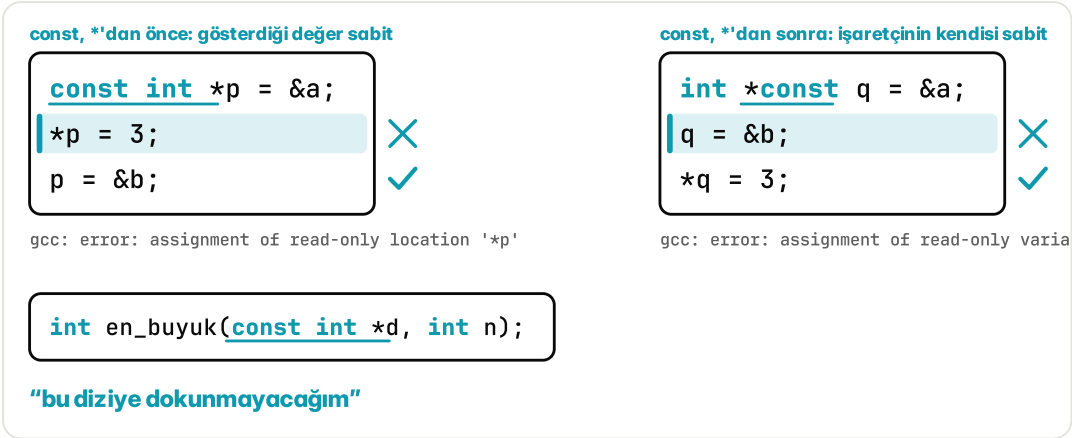
Son olarak const. const bir söz verir. Yıldızdan önce yazılırsa (const int *p) işaretçinin gösterdiği değer onun üzerinden değiştirilemez, ama p başka bir yeri gösterebilir. Yıldızdan sonra yazılırsa (int *const q) işaretçinin kendisi sabittir ve başka bir yeri gösteremez, ama gösterdiği değer değişebilir:

```c
#include <stdio.h>

int main(void) {
    int a = 1, b = 2;
    const int *p = &a;
    int *const q = &a;
    p = &b;
    *q = 3;
    printf("%d %d\n", *p, a);
    return 0;
}
```

Çıktı:

```
2 3
```

Tersini denersek, yani *p = 3; ya da q = &b; yazarsak program derlenmez: gcc ilki için "assignment of read-only location '*p'", ikincisi için "assignment of read-only variable 'q'" der. const, diziyi bir fonksiyona verirken "buna dokunmayacağım" demenin yoludur: int en_buyuk(const int *d, int n); gibi bir prototip, fonksiyonun dizinin elemanlarını yalnız okuyacağını söyler.



Şekil 5.12 const yıldızdan önceyse gösterilen değer, yıldızdan sonraysa işaretçinin kendisi sabittir; buna aykırı satırlar derlenmez.

Özet

- Her baytın bir numarası, yani adresi vardır; & bir değişkenin adresini verir, printf adresi %p ile onaltılık yazar.
- İşaretçi adres tutan değişkendir; tanımdaki yıldız tipin parçasıdır, koddaki yıldız adresteki değere gider.
- Fonksiyona değişkenin adresi verilirse fonksiyon çağıranın değişkenini değiştirebilir; scanf bu yüzden & ister.
- İşaretçiye 1 eklemek bir sonraki elemana geçmektir; adres tipin boyu kadar ilerler ve a[i], *(a + i) demektir.
- Fonksiyona verilen dizi ilk elemanının adresine dönüşür; boy bilgisi kaybolduğu için eleman sayısı ayrıca verilir.
- İlk değeri verilmemiş işaretçi tehlikelidir; henüz bir yeri göstermeyen işaretçiye NULL verilir, const da gösterilen değeri ya da işaretçinin kendisini sabitler.

Sorular

1 **int x = 7; int *p = &x; satırlarından sonra p'nin içinde ne vardır?**

A 7 sayısı

B x'in adresi

C p'nin kendi adresi

D x'in bir kopyası

2 **Aşağıdaki satırlar ekrana ne yazar?**

```
int x = 7;
int *p = &x;
*p = *p + 3;
printf("%d\n", x);
```

Cevap:

3 **Bir fonksiyonun main'deki int sayi değişkenini 0 yapabilmesi için fonksiyon ve çağrı nasıl yazılmalıdır?**

A void sifirla(int n) { n = 0; } ve sifirla(sayi);

B void sifirla(int *n) { n = 0; } ve sifirla(&sayi);

C void sifirla(int *n) { *n = 0; } ve sifirla(&sayi);

D void sifirla(int n) { n = 0; } ve sifirla(&sayi);

4 **int adet; scanf("%d", adet); satırındaki hata nedir?**

A adet'in önündeki & unutulmuştur; scanf okuduğu sayıyı yazacağı yerin adresini ister.

B %d yerine %s yazılmalıdır.

C scanf yalnız metin okuyabilir, sayı okuyamaz.

D adet'e önce bir değer verilmeden scanf çağrılamaz.

5 **int'in 4 bayt olduğu bir bilgisayarda int *p işaretçisi 8000 adresini tutuyor. p + 3 hangi adresi verir?**

Cevap:

6 **int a[4] = {6, 15, 27, 40}; dizisi için *(a + 1) + a[3] ifadesinin değeri nedir?**

A 46

B 67

C Derlenmez, çünkü adrese sayı eklenemez.

D 55

7 64 bitlik bir bilgisayarda main'de int sayilar[10]; tanımlı ve dizi void f(int d[]) fonksiyonuna veriliyor. f'nin içinde sizeof d kaç verir?

A 40

B 10

C 8

D 4

8 int a = 1, b = 2; const int *p = &a; tanımlarından sonra aşağıdaki satırlardan hangisi derlenmez?

A p = &b;

B *p = 3;

C printf("%d\n", *p);

D int c = *p;

PROGRAMLAMAYA GİRİŞ · 1. DÖNEM

6 Dinamik bellek yönetimi

Bu bölümde boyu ancak program çalışırken belli olan veriler için belleğin nereden ve nasıl alındığını göreceksin. Önce çalışan bir programın belleğinin bölgelerini ve stack ile heap arasındaki farkı, sonra heap'ten bellek alıp geri veren dört fonksiyonu, en sonda da dinamik bellekte en sık yapılan dört hatayı öğreneceksin.

Dizi ne kadar büyük olmalı?

videoda 28:43

Kullanıcıdan bir sayı listesi alan bir program yazıyoruz. Kullanıcı kaç sayı gireceğini ancak program çalışınca söyleyecek: belki 6, belki yarım milyon. Oysa dizinin boyunu kodu yazarken vermek zorundayız. Bu sayıları tutacak dizi kaç elemanlı olmalı?

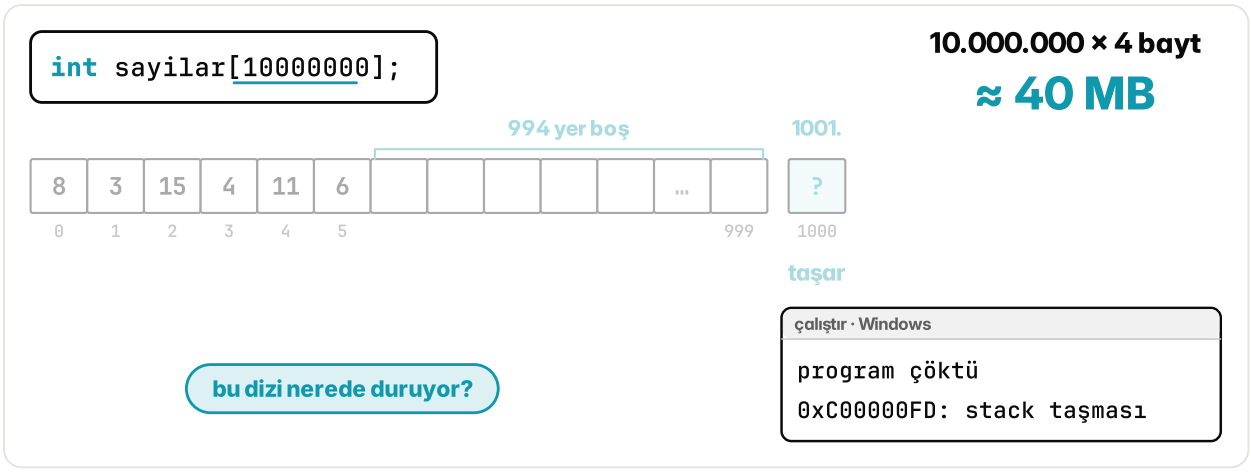
İlk akla gelen, diziyi büyük tutup yetmesini ummaktır. Aşağıdaki program 1000 elemanlı bir diziyle çalışıyor; kullanıcı önce 6, sonra 8, 3, 15, 4, 11 ve 6 sayılarını giriyor:

```
int sayilar[1000];
int adet;
scanf("%d", &adet);
for (int i = 0; i < adet; i++)
    scanf("%d", &sayilar[i]);
printf("%d sayı alındı, %d yer boş\n", adet, 1000 - adet);
```

Çıktı:

```
6 sayı alındı, 994 yer boş
```

Yerin neredeyse tamamı boşa gitti. Kullanıcı 1001. sayıyı girerse de dizi taşar: C dizinin sınırını denetlemez, sayı dizinin dışındaki belleğe yazılır ve bu tanımsız davranıştır. Diziyi 10 milyon elemanlı yapsak bu kez program daha başlarken çöker. 10.000.000 × 4 bayt = 40.000.000 bayt, yani yaklaşık 40 MB'lık bu yerel dizi, durduğu yere sığmaz. Bu örneği çalıştırdığımız Windows'ta program hiçbir şey yazmadan 0xC00000FD koduyla, yani stack taşmasıyla kapandı.



Şekil 6.1 10 milyon elemanlı yerel dizi 10.000.000 × 4 bayt, yaklaşık 40 MB tutar; Windows'ta program 0xC00000FD koduyla, yani stack taşmasıyla çöker.

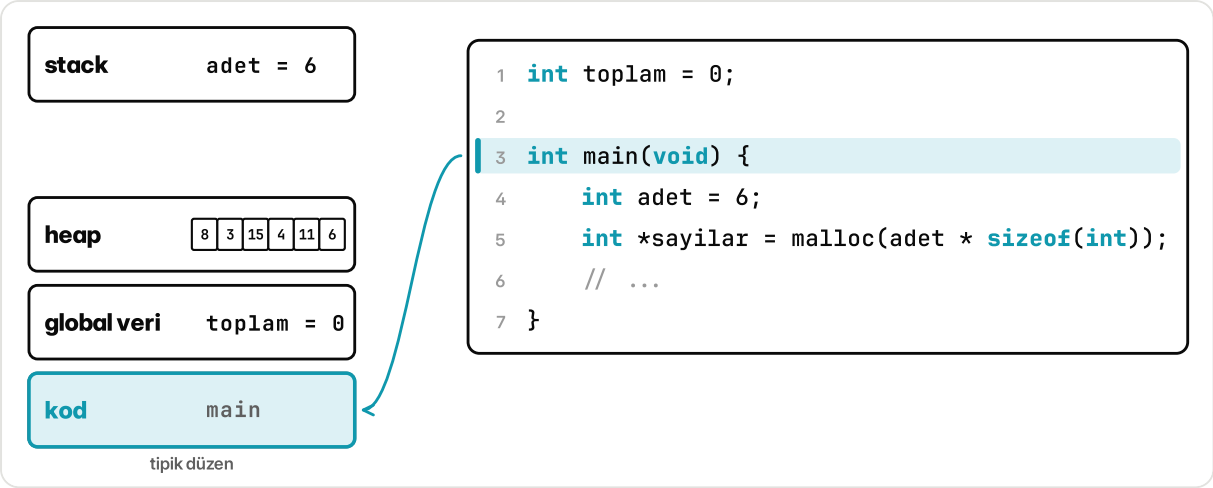
Nedenini anlamak için çalışan bir programın belleğine bakalım. Bellek dört bölgeye ayrılır:

- **Kod bölgesi**: derlenmiş komutlar burada durur.
- **Global veri bölgesi**: global ve static değişkenler; program başlarken kurulur, program bitene kadar yaşarlar.
- **Stack**: fonksiyon çağrılarının çerçeveleri; yerel değişkenler ve yerel diziler buradadır, fonksiyon bitince çerçeveyle birlikte kalkarlar.
- **Heap**: program çalışırken istenen belleğin verildiği bölge.

Bölgelerin bellekteki sırası sistemden sisteme değişir; şekillerde gösterilen tipik düzendir. Küçük bir örnekte hangisinin nerede durduğunu görelim:

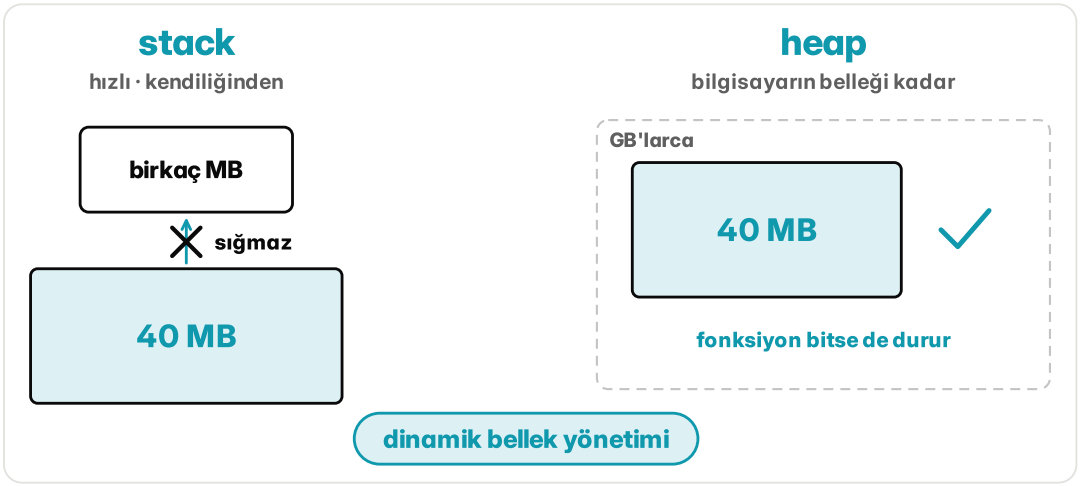
```
int toplam = 0;                 /* global veri */
int main(void) {                /* kod */
    int adet = 6;               /* stack */
    int *sayilar = malloc(adet * sizeof(int));   /* heap */
```

Dosyanın başındaki toplam global veride, main'in içindeki adet stack'te durur. malloc ile program çalışırken istenen 6 sayılık yer heap'tedir; bunları işleyen komutlar da kod bölgesindedir.



Şekil 6.2 Tipik düzende dört bölge: toplam = 0 global veride, adet = 6 stack'te, 6 sayılık blok heap'te, main'in komutları kod bölgesinde.

Stack hızlıdır ve kendiliğinden yönetilir: fonksiyon çağrılınca çerçeve kurulur, bitince kalkar. Ama küçüktür; çoğu sistemde birkaç megabayt (Windows'ta çoğunlukla 1 MB, Linux'ta 8 MB). 40 MB'lık dizi bu yüzden sığmadı. Heap ise bilgisayarın belleği kadar büyüyebilir; aynı 40 MB orada sorun olmaz. Üstelik heap'ten alınan bellek, onu alan fonksiyon bitse de biz geri verene kadar yerinde durur. Bedeli şudur: ayırmayı da geri vermeyi de programcı yapar. Belleği program çalışırken gerektiği kadar heap'ten alıp işi bitince geri vermeye **dinamik bellek yönetimi** denir.



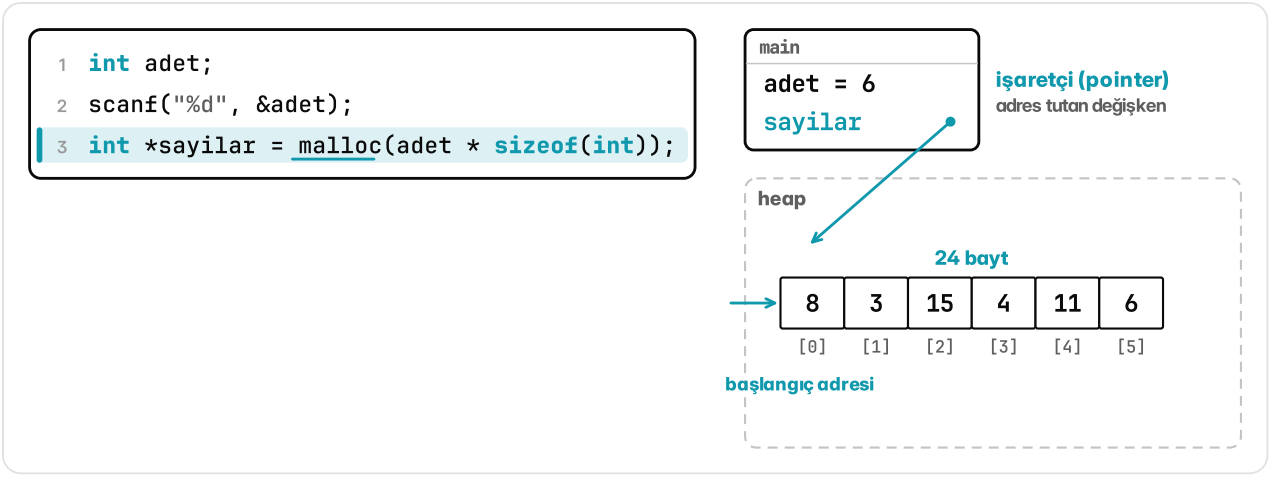
Şekil 6.3 Stack birkaç MB'tır ve 40 MB'lık diziye yetmez; heap bilgisayarın belleği kadar büyür ve oradaki blok fonksiyon bitse de durur.

6 sayılık yer

videoda 30:02

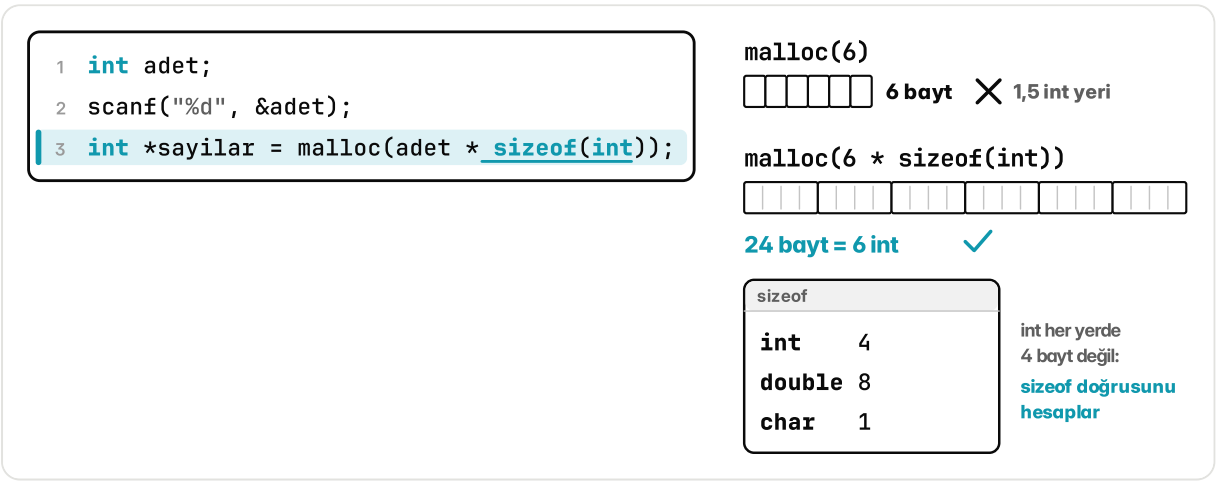
Kullanıcı 6 sayı gireceğini söyledi. Bu 6 sayının yerini program çalışırken heap'ten nasıl isteriz? Bu iş için C'nin stdlib.h başlığında dört fonksiyon vardır.

malloc, istediğimiz kadar baytı heap'ten ayırır ve ayırdığı bloğun başlangıç adresini geri verir. Bu adresi bir işaretçide, yani adres tutan bir değişkende saklarız; sonra bloğu dizi gibi kullanırız.



Şekil 6.4 malloc heap'ten 24 baytlık bir blok ayırır; main'deki sayilar işaretçisi bloğun başlangıç adresini tutar ve blok [0]'dan [5]'e dizi gibi kullanılır.

malloc bayt sayısı ister, eleman sayısı değil. malloc(6) yazarsak 6 sayılık değil, 6 baytlık yer alırız; int 4 bayt olduğuna göre bu ancak 1,5 sayılık yerdir. 6 sayı için 24 bayt gerekir. 24'ü elle yazmak yerine adet * sizeof(int) ile hesaplatırız: int'in boyu başka bir bilgisayarda farklıysa da sizeof doğrusunu verir.



Şekil 6.5 malloc(6) yalnız 6 bayt, yani 1,5 int'lik yer verir, malloc(6 * sizeof(int)) ise 24 bayt; sizeof int için 4, double için 8, char için 1 bayt söyler.

malloc her zaman başarmaz: yeterli bellek yoksa **NULL** döndürür, yani hiçbir yeri göstermeyen adresi. NULL adresine yazmaya kalkan program çöker. Bu yüzden ayırmanın hemen ardından kontrol ederiz; NULL geldiyse hata yazıp programdan çıkarız. Girdi yine 6 ve 8, 3, 15, 4, 11, 6 olsun:

```
int adet;
scanf("%d", &adet);
int *sayilar = malloc(adet * sizeof(int));
if (sayilar == NULL) {
    printf("Bellek yetmedi\n");
    return 1;
}
for (int i = 0; i < adet; i++)
    scanf("%d", &sayilar[i]);
printf("%zu bayt, ilk sayı %d\n", adet * sizeof(int), sayilar[0]);
free(sayilar);
```

Çıktı:

```
24 bayt, ilk sayı 8
```

Son satırdaki **free**, işi biten bloğu heap'e geri verir. Stack'teki yerel değişkenler fonksiyon bitince kendiliğinden kalkar; heap'teki blok ise biz free ile geri verene kadar yerinde durur.

malloc'un verdiği bloğun içinde önceden kalmış çöp değerler olabilir. **calloc** ise iki şey ister: eleman sayısını ve bir elemanın boyunu. Bloğu verirken her baytını sıfırlar:

```
int *sayilar = calloc(adet, sizeof(int));
```

adet 6 iken elemanlar yazdırılınca 6 sayının hepsi 0 çıkar:

Çıktı:

```
0 0 0 0 0 0
```

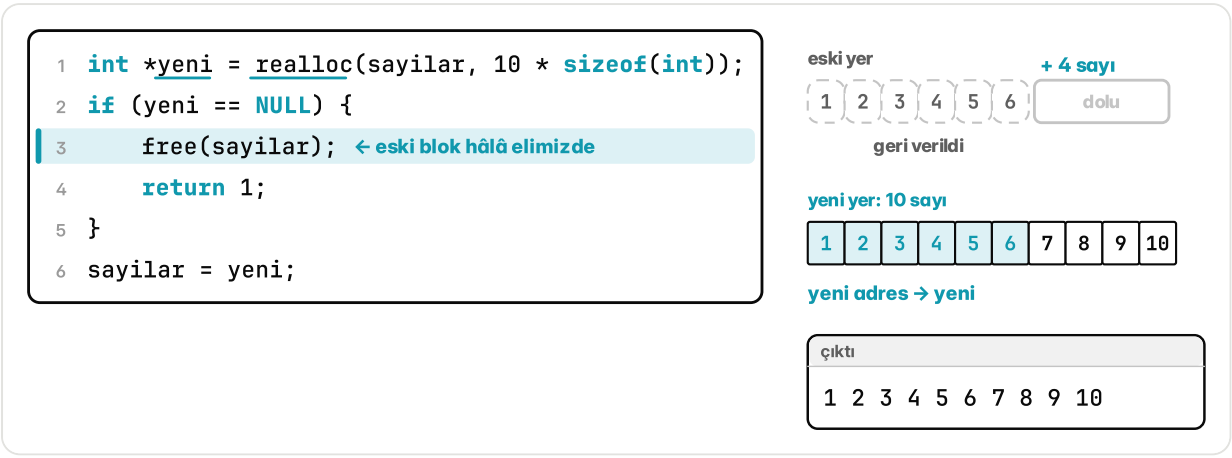
Kullanıcı 4 sayı daha girmek isterse **realloc** bloğu 10 sayılık yapar. Blok yerinde büyüyemiyorsa realloc heap'te yeni bir yer bulur, eski sayıları oraya taşır, eski yeri geri verir ve yeni adresi döndürür. Başaramazsa NULL döndürür ve eski blok olduğu gibi kalır. Bu yüzden sonucu önce ayrı bir işaretçiye alırız:

```
int *yeni = realloc(sayilar, 10 * sizeof(int));
if (yeni == NULL) {
    free(sayilar);          /* eski blok hâlâ bizde */
    return 1;
}
sayilar = yeni;
```

Sonucu doğrudan sayilar'a yazsaydık, realloc başarısız olduğunda sayilar NULL olur ve eski bloğun adresi kaybolurdu. Örnekte ilk 6 elemanda 1'den 6'ya kadar sayılar vardı, yeni 4 elemana 7'den 10'a kadarı yazıldı; taşınmada ilk 6 sayı korunur:

Çıktı:

```
1 2 3 4 5 6 7 8 9 10
```



Şekil 6.6 realloc: 6 sayılık blok yerinde büyüyemeyince sayılar 10 sayılık yeni yere taşınır ve eski yer geri verilir; yeni adres önce yeni işaretçisine alınır.

Bellek nereye gidiyor?

videoda 31:29

Haftalarca açık kalan bir sunucu programı her gün biraz daha çok bellek kullanıyor ve sonunda çöküyor. Bellek nereye gidiyor? Cevap, dinamik bellekte en sık yapılan dört hatanın ilkinde.

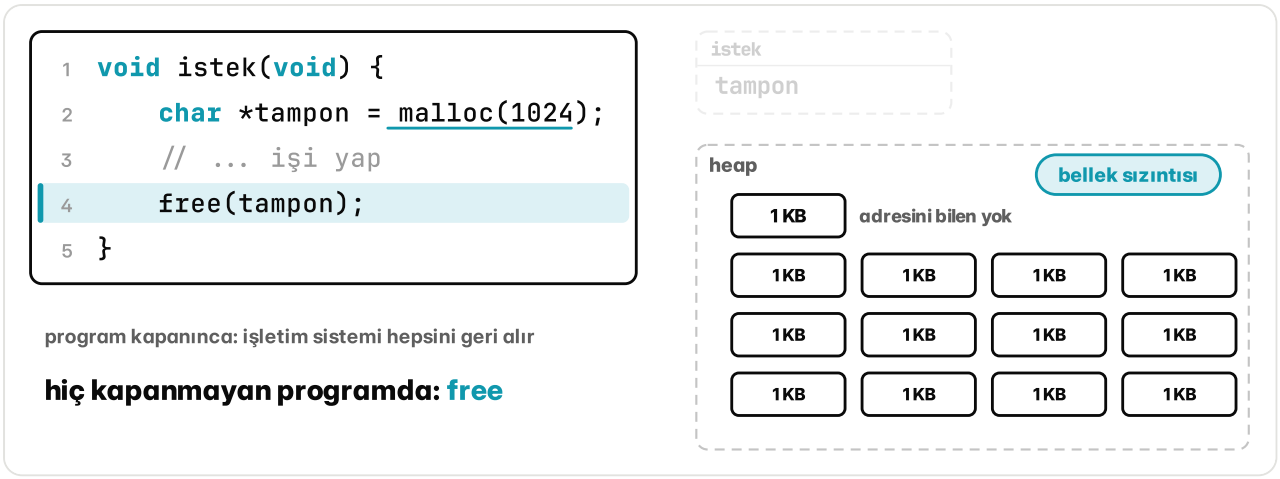
Program her istekte heap'ten 1 KB alıyor ve geri vermiyor:

```
void istek(void) {
    char *tampon = malloc(1024);
    if (tampon == NULL)
        return;
    tampon[0] = 0;
}   /* free yok: tampon kalktı, 1 KB'lık blok heap'te kaldı */
```

istek bitince adresi tutan tampon stack'ten kalkar; blok ise heap'te durur, ama adresini bilen kimse kalmaz. Buna **bellek sızıntısı** (memory leak) denir. Saniyede 100 istekle bir günde kaybolan bellek:

```
100 istek/sn × 1 KB     = 100 KB/sn
100 KB/sn × 86.400 sn   = 8.640.000 KB ≈ 8,6 GB
```

Program kapanınca işletim sistemi ona verdiği belleğin hepsini geri alır; ama hiç kapanmayan bir programda tek çare, işi biten bloğu free etmektir.

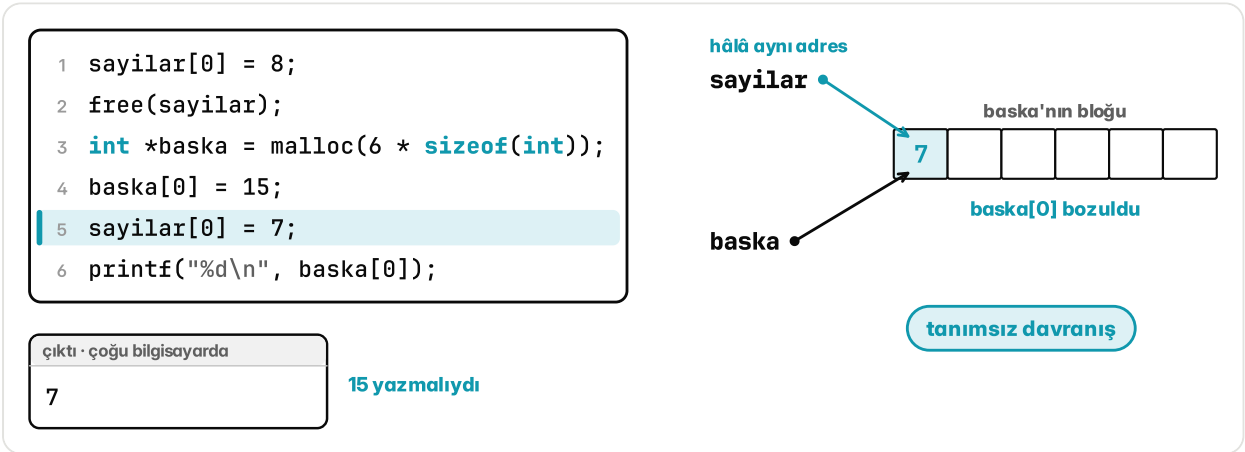


Şekil 6.7 istek her çağrıda 1 KB alır; tampon kalkınca adresini bilen kalmayan bloklar heap'te birikir; çare, fonksiyonun sonuna eklenen free(tampon).

İkinci hata, geri verilmiş belleği kullanmaktır. free bloğu geri verir ama işaretçideki adresi silmez; işaretçi hâlâ aynı yeri gösterir. O yer bu arada malloc ile başka bir değişkene verilmiş olabilir:

```
sayilar[0] = 8;
free(sayilar);
int *baska = malloc(6 * sizeof(int));   /* aynı yer buna verilebilir */
if (baska == NULL)
    return 1;
baska[0] = 15;
sayilar[0] = 7;          /* tanımsız davranış: geri verilmiş belleğe yazmak */
printf("%d\n", baska[0]);
```

Ekrana 15 çıkmalıydı. Çoğu bilgisayarda baska, sayilar'ın eski bloğunu alır; sayilar[0] = 7 satırı baska'nın değerini bozar ve ekrana 7 çıkar. Bu örneği çalıştırdığımız Windows'ta heap aynı bloğu hemen vermedi, 15 çıktı. Sonuç güvenilmez; bu tanımsız davranıştır.



Şekil 6.8 free'den sonra sayilar hâlâ aynı adresi gösterir; o bloğu alan baska'nın değeri sayilar[0] = 7 ile bozulur ve çoğu bilgisayarda 15 yerine 7 yazılır.

Üçüncü hata, aynı bloğu iki kez free etmektir. Bu, heap'in boş yerleri izlediği kayıtları bozar. Program çoğu zaman hemen çöker; daha kötüsü, bu hatayı bulan bir saldırgan programın denetimini ele geçirebilir. Çaresi kısadır: free'den hemen sonra işaretçiye NULL yazarız. free(NULL)

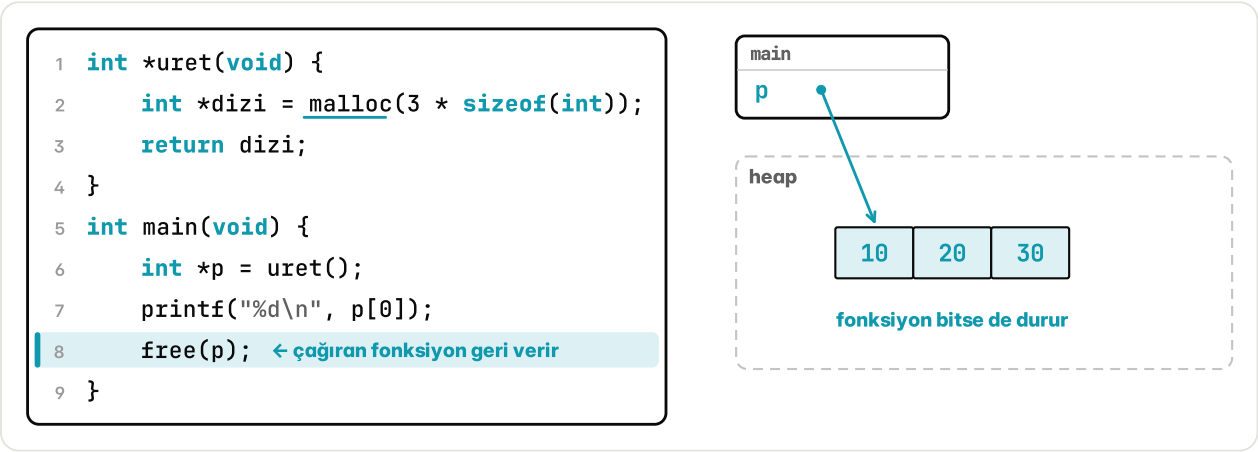
hiçbir şey yapmaz, ikinci free zararsız kalır:

```
free(sayilar);
sayilar = NULL;
free(sayilar);           /* free(NULL) hiçbir şey yapmaz */
```

Dördüncü hata stack'tedir: fonksiyon kendi yerel dizisinin adresini döndürür.

```
int *uret(void) {
    int dizi[3] = {10, 20, 30};
    return dizi;             /* yerel dizinin adresi: çerçeve kalkınca geçersiz */
}
```

uret bitince çerçevesi kalkar; dönen adres, sonraki çağrının çerçevesinin kurulacağı yeri gösterir. Derleyici uyarır. Uyarıyı dinlemezsek sonuç derleyiciye göre değişir: gcc ile derlenen program çöker; clang ile derlediğimiz örnekte ilk okuma 10 verdi, araya bir printf çağrısı girince ikinci okuma çöp bir değer (189) verdi. Doğrusu, diziyi heap'ten alıp adresini döndürmektir; uret'in ilk satırı int *dizi = malloc(3 * sizeof(int)); olur. Bloğu işi bitince free etmek de uret'i çağıran fonksiyonun görevidir.



Şekil 6.9 Doğru sürüm: uret diziyi heap'ten alır, 10, 20, 30 fonksiyon bitse de durur; main işi bitince free(p) ile geri verir.

Microsoft ve Google, yazılımlarındaki ciddi güvenlik açıklarının yaklaşık %70'inin bu tür bellek hatalarından çıktığını açıkladı. Kural bu yüzden kısadır: heap'ten alınan her bloğu tam bir kez geri ver, geri verdikten sonra ona dokunma, yerel değişkenin adresini fonksiyondan dışarı verme.

Özet

- Çalışan bir programın belleği dört bölgeye ayrılır: kod, global veri, stack ve heap.
- Stack hızlıdır ve kendiliğinden yönetilir ama küçüktür; heap büyüktür ve oradan alınan bellek geri verilene kadar durur.
- malloc istenen kadar baytı heap'ten ayırıp başlangıç adresini döndürür; bayt sayısı sizeof ile hesaplanır, sonuç NULL mı diye kontrol edilir.
- calloc bloğu sıfırlayarak verir; realloc bloğu büyütür ve sonucu ayrı bir işaretçiye alınır.
- free bloğu geri verir ama işaretçideki adresi silmez; free'den sonra işaretçiye NULL yazmak iki kez free hatasını önler.
- Sızıntı, geri verilmiş belleği kullanmak, iki kez free ve yerel değişkenin adresini döndürmek dinamik belleğin dört tuzağıdır.

Sorular

1 **Bir fonksiyonun içinde tanımlanan int dizi[100]; dizisi bellekte nerede durur ve fonksiyon bitince ne olur?**

A Heap'te durur; free ile geri verilene kadar kalır.

B Global veri bölgesinde durur; program bitene kadar kalır.

C Stack'te durur; fonksiyon bitince çerçeveyle birlikte kalkar.

D Kod bölgesinde durur; derlenmiş komutlarla birlikte kalır.

2 **int'in 4 bayt olduğu bir bilgisayarda int *p = malloc(6); satırı ne kadar yer ayırır?**

A 6 int'lik, yani 24 baytlık yer

B 6 baytlık, yani 1,5 int'lik yer

C 6 KB'lık yer

D Hiç yer ayırmaz; derleme hatası verir

3 **int'in 4 bayt olduğu bir bilgisayarda malloc(25 * sizeof(int)) kaç bayt ayırır?**

Cevap: _____

4 **calloc(adet, sizeof(int)) ile malloc(adet * sizeof(int)) arasındaki fark nedir?**

A calloc belleği stack'ten, malloc heap'ten alır.

B calloc ile alınan blok free ile geri verilmez.

C calloc hiçbir zaman NULL döndürmez.

D calloc bloğun her baytını sıfırlayarak verir; malloc'un bloğunda önceden kalmış çöp değerler olabilir.

5 **free(sayilar); satırından hemen sonra sayilar işaretçisi için hangisi doğrudur?**

A sayilar hâlâ eski adresi tutar, ama o blok artık bizim değildir.

B sayilar kendiliğinden NULL olur.

C Bloğun içi sıfırlanır ve sayilar ile güvenle okunabilir.

D sayilar stack'ten silinir ve artık kullanılamaz bir ad olur.

6 **Bloğu büyütmek için sayilar = realloc(sayilar, 10 * sizeof(int)); yazmanın sakıncası nedir?**

A realloc yalnız bloğu küçültebilir, büyütemez.

B realloc eski sayıları taşımaz; ilk 6 sayı kaybolur.

C realloc başarısız olursa NULL döner ve eski bloğun adresi kaybolur; blok geri verilemez.

D realloc'un sonucu her zaman aynı adres olduğu için atamaya gerek yoktur.

7 **Aşağıdaki fonksiyonun sorunu nedir?**

```
int *uret(void) {
    int dizi[3] = {10, 20, 30};
    return dizi;
}
```

A dizi heap'te durduğu için free edilmesi unutulmuştur.

B Dönen adres, fonksiyon bitince kalkan stack çerçevesindeki diziyi gösterir.

C Bir fonksiyon dizinin adresini hiçbir şekilde döndüremez.

D dizi'ye ilk değer süslü parantezle verilemez.

8 **Bir sunucu programı her istekte heap'ten 2 KB alıyor ve geri vermiyor. Saniyede 50 istek gelirse 1 saatte kaç KB sızar?**

Cevap: _____

PROGRAMLAMAYA GİRİŞ · 1. DÖNEM

7 Özyineleme ve fonksiyon işaretçileri

Bu bölümde kendini çağıran fonksiyonları ve bir fonksiyonu başka bir fonksiyona vermeyi göreceksin. Önce özyinelemenin nasıl durduğunu ve stack'te nasıl çalıştığını, sonra ne zaman çok yavaşladığını ve ne zaman en doğal yol olduğunu, en sonda da fonksiyon işaretçileriyle C'nin hazır sıralama fonksiyonu qsort'u öğreneceksin.

Kendini çağıran fonksiyon

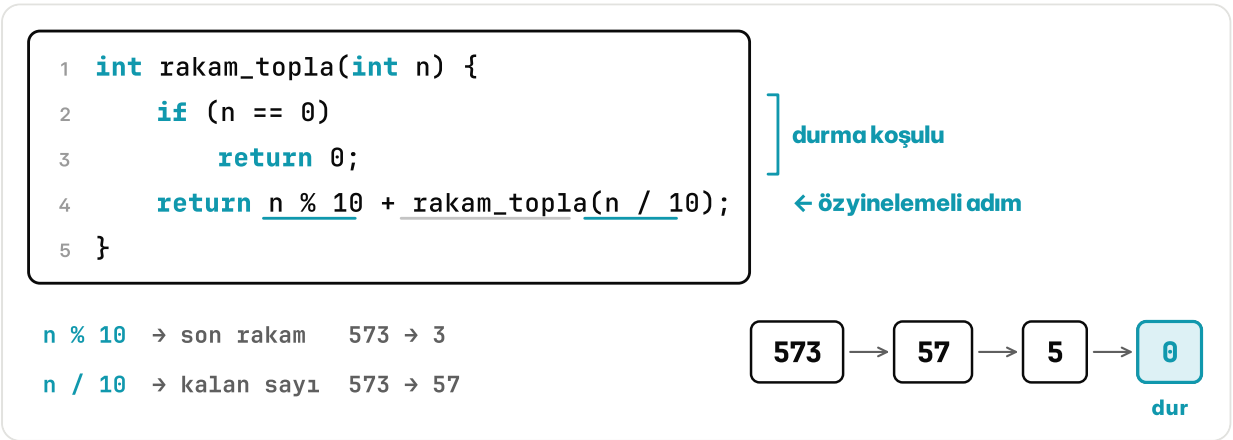
videoda 33:09

Bir fonksiyon, kendi içinden yine kendini çağırabilir. Peki bu çağrılar neden sonsuza kadar sürmez?

573'ün rakamlarını toplayalım. Son rakam olan 3'ü ayırır, kalan 57'nin rakam toplamına ekleriz. 57'nin rakam toplamı da aynı sorudur; yalnız sayı bir basamak kısalmıştır. Bir fonksiyonun, sorunun küçük hâlini çözmek için kendini çağırmasına **özyineleme** (recursion) denir:

```
int rakam_topla(int n) {
    if (n == 0)
        return 0;
    return n % 10 + rakam_topla(n / 10);
}
```

Kodda iki parça var. İlki **durma koşulu**: sayı 0 ise toplanacak rakam kalmamıştır; fonksiyon kendini çağırmadan 0 döndürür. İkincisi **özyinelemeli adım**: n % 10, yani 10'a bölümden kalan, son rakamı verir; n / 10, yani 10'a bölüm, son rakamı atar. Fonksiyon kısalan sayıyla kendini çağırır. Her adım sayıyı bir basamak kısaltır ve sonunda durma koşuluna varır; çağrıların sonsuza kadar sürmemesinin nedeni budur.

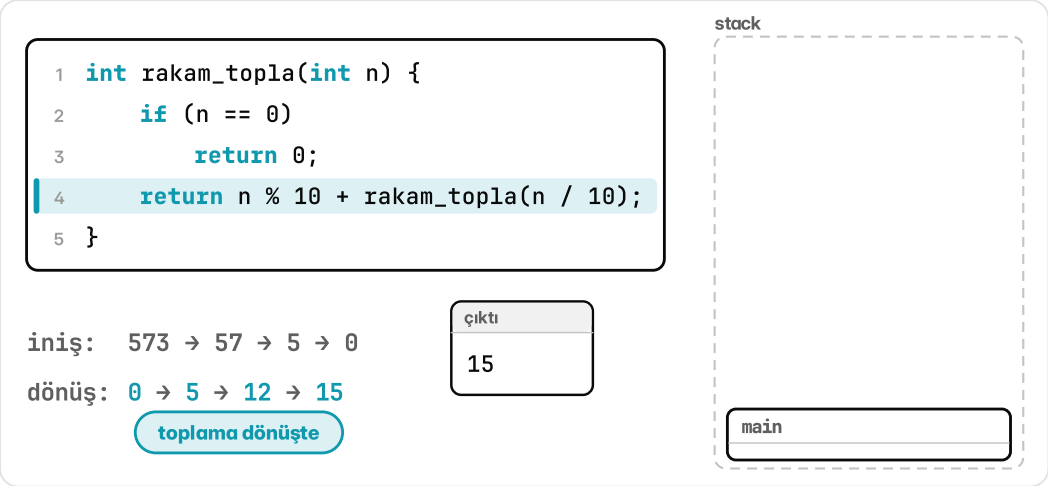


Şekil 7.1 rakam_topla'nın iki parçası: n == 0 iken durma koşulu, değilse özyinelemeli adım; n % 10 son rakamı, n / 10 kalan sayıyı verir ve sayı 573, 57, 5, 0 diye kısalır.

Her fonksiyon çağrısı stack'te kendi çerçevesini açar; fonksiyon kendini çağırsa da öyle. Kod kopyalanmaz; aynı kod her çağrıda yeni bir çerçeveyle yeniden çalışır. rakam_topla(573) çağrılınca 573, 57, 5 ve 0 için dört çerçeve üst üste kurulur. Her çerçevenin kendi n'si vardır ve her biri, üstündeki çağrının cevabını bekler:

```
rakam_topla(573) = 3 + rakam_topla(57)
rakam_topla(57)  = 7 + rakam_topla(5)
rakam_topla(5)   = 5 + rakam_topla(0)
rakam_topla(0)   = 0
```

En üstteki çerçeve 0 döndürür ve kalkar. 5'in çerçevesi buna 5 ekler, 5 döndürür. 57'ninki 7 ekler, 12 döndürür; 573'ünkü 3 ekler, 15 döndürür. Yani toplama inişte değil dönüşte yapılır. printf("%d\n", rakam_topla(573)); satırı ekrana 15 yazar.



Şekil 7.2 Stack'te iniş 573, 57, 5, 0; dönüş 0, 5, 12, 15; toplama dönüşte yapılır ve ekrana 15 çıkar.

Şimdi durma koşulunu silelim:

```
int rakam_topla(int n) {
    return n % 10 + rakam_topla(n / 10);
}
```

Sayı 0'a inince fonksiyon yine kendini çağırır; 0'ın 10'a bölümü yine 0'dır, çağrılar hiç bitmez. Derleyicinin uyarıları açıksa (-Wall) bu sonsuz çağrıyı daha derlerken fark edip haber verir. Her çağrı yeni bir çerçeve açar, ama stack sınırlıdır: çoğu sistemde 1 ile 8 megabayt arası. Stack dolunca program çöker; buna **stack taşması** (stack overflow) denir. Bu örneği çalıştırdığımız Windows'ta program hiçbir şey yazmadan 0xC00000FD koduyla kapandı. Yazılımcıların en çok soru sorduğu sitelerden Stack Overflow da adını bu hatadan alır. Bu yüzden özyinelemeli bir fonksiyonda önce durma koşulu yazılır.

Beş satır, 331 milyon çağrı

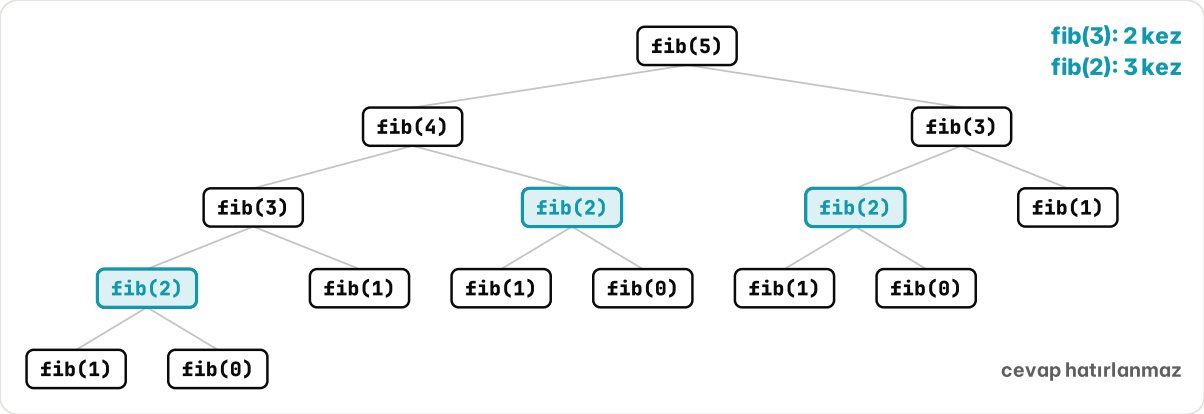
videoda 34:55

5 satırlık bir fonksiyon, tek bir sayıyı bulmak için neden 300 milyondan fazla kez çağrılır?

Fibonacci dizisinde her sayı, kendinden önceki iki sayının toplamıdır: 0, 1, 1, 2, 3, 5, 8, 13. Tanım koda neredeyse aynen geçer: istenen sıra 2'den küçükse sıranın kendisi döner; değilse fonksiyon önceki iki sırayla kendini çağırıp sonuçları toplar. Çağrıları saymak için her çağrıda global bir sayacı (cagri) bir artıralım:

```
int fib(int n) {
    cagri++;
    if (n < 2)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```

Ama her çağrı iki yeni çağrı açar. fib(5) hesaplanırken fib(4) ve fib(3) istenir; fib(4) de yine fib(3)'ü ister. Bu küçük ağaçta bile fib(3) 2 kez, fib(2) 3 kez baştan hesaplanır, çünkü fonksiyon bulduğu cevabı hatırlamaz.



Şekil 7.3 fib(5)'in 15 düğümlü çağrı ağacı: cevap hatırlanmadığı için fib(3) 2 kez, fib(2) 3 kez hesaplanır.

main fib(40)'ı isteyip sonucu ve sayacı yazdırınca:

Çıktı:

```
102334155
331160281 cagri
```

Cevap 100 milyonun biraz üstünde, ama fonksiyon 331 milyon kez çağrıldı; fib(2) tek başına 63 milyon kez yeniden hesaplandı.

Fonksiyona hafıza verebiliriz: bulduğu her sayıyı bir diziye yazar, aynı sayı yeniden istenince hesaplamadan oradan alır. Buna **hafızalı özyineleme** (memoization) denir. Global int bilinen[41]; dizisinin bütün elemanları 0'la başlar; 0 görülen yer henüz hesaplanmamış demektir:

```
int fib(int n) {
    cagri++;
    if (n < 2)
        return n;
    if (bilinen[n] == 0)
        bilinen[n] = fib(n - 1) + fib(n - 2);
    return bilinen[n];
}
```

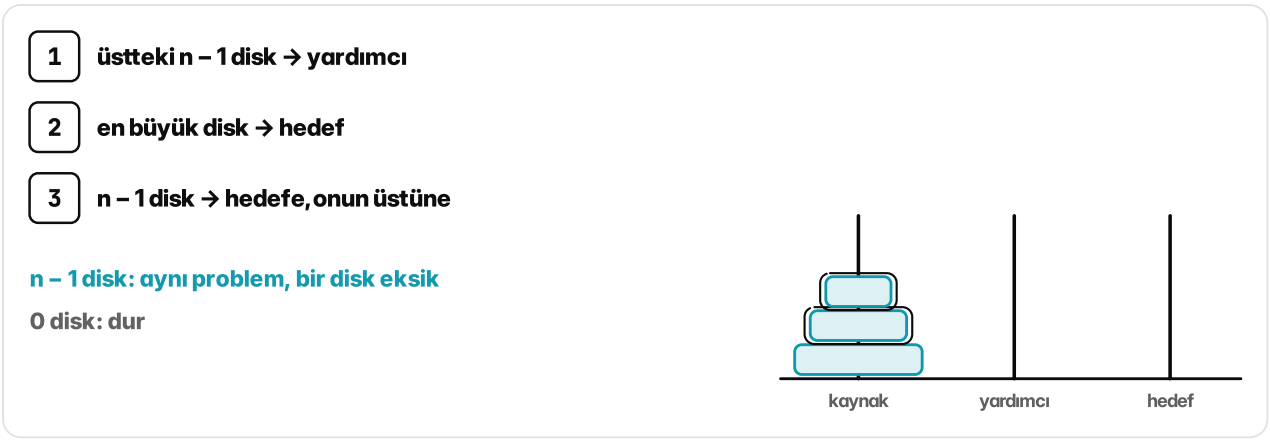
Sonuç yine 102334155, ama çağrı sayısı 331 milyondan 79'a iner. Daha da basiti döngüdür: son iki sayıyı iki değişkende tutar, her turda toplamlarını bulup bir adım ilerleriz.

```
int fib(int n) {
    int a = 0, b = 1;
    for (int i = 0; i < n; i++) {
        int t = a + b;
        a = b;
        b = t;
    }
    return a;
}
```

40 tur yeter; özyinelemeli sürüm saniyeler sürerken döngü anında biter.

Bazı problemlerde ise özyineleme en doğal yoldur. **Hanoi kuleleri** bunlardan biridir: üç çubuktan birindeki diskleri tek tek taşıyıp hepsini başka bir çubuğa dizeceğiz ve büyük disk küçüğün üstüne hiç konmayacak. Çözüm üç adımdır: üstteki diskleri yardımcı çubuğa taşı, en büyüğü hedefe koy, sonra o diskleri onun üstüne taşı. Üstteki diskleri taşımak da aynı problemdir, yalnız bir disk eksiktir; disk kalmayınca durulur. Kodda da üç adım üç satırdır:

```
void hanoi(int n, char kay, char hed, char yrd) {
    if (n == 0) return;
    hanoi(n - 1, kay, yrd, hed);
    printf("%c -> %c\n", kay, hed);
    hanoi(n - 1, yrd, hed, kay);
}
```



Şekil 7.4 Hanoi kulelerinin üç adımı: üstteki n – 1 disk yardımcıya, en büyük disk hedefe, n – 1 disk onun üstüne; n – 1 disk aynı problemdir, 0 diskte durulur.

hanoi(3, 'A', 'C', 'B') çağrısı 3 diski A'dan C'ye 7 hamlede taşır: A → C, A → B, C → B, A → C, B → A, B → C, A → C. Her yeni disk hamle sayısını iki katına çıkarıp bir ekler: 1, 3, 7, 15 ... 10 disk 1023 hamle. Yani n disk için $2^n - 1$ hamle gerekir. 64 disk için bu 18.446.744.073.709.551.615 hamledir; saniyede bir hamleyle yaklaşık 585 milyar yıl sürer, evrenin yaşının (13,8 milyar yıl) 40 katından fazla.



Şekil 7.5 Her disk hamle sayısını iki katına çıkarıp bir ekler: 1, 3, 7, 15, 31, 10 diskte 1023; 64 disk $2^{64} - 1$ hamle, saniyede bir hamleyle yaklaşık 585 milyar yıl.

Hangisini ne zaman seçmeli? Problem kendi küçüğüne bölünüyor ve aynı hesap tekrarlanmıyorsa özyineleme kısa ve açık yazılır. Aynı hesap tekrarlanıyorsa hafıza ya da döngü gerekir; çağrılar çok derine iniyorsa döngü seçilir, çünkü stack'i tüketmez.

Fonksiyona fonksiyon

videoda 37:07

Bir fonksiyona sayı da, dizi de verebiliyoruz. Peki bir fonksiyona başka bir fonksiyonu verebilir miyiz?

Programın kodu da bellekte durur; her fonksiyonun, kodunun başladığı bir adresi vardır. Dizinin adı ilk elemanının adresi olduğu gibi, fonksiyonun adı da bu adrestir. Bu adresi tutan değişkene **fonksiyon işaretçisi** denir. İki fonksiyonumuz olsun:

```
int kare(int x) { return x * x; }
int kup(int x) { return x * x * x; }
```

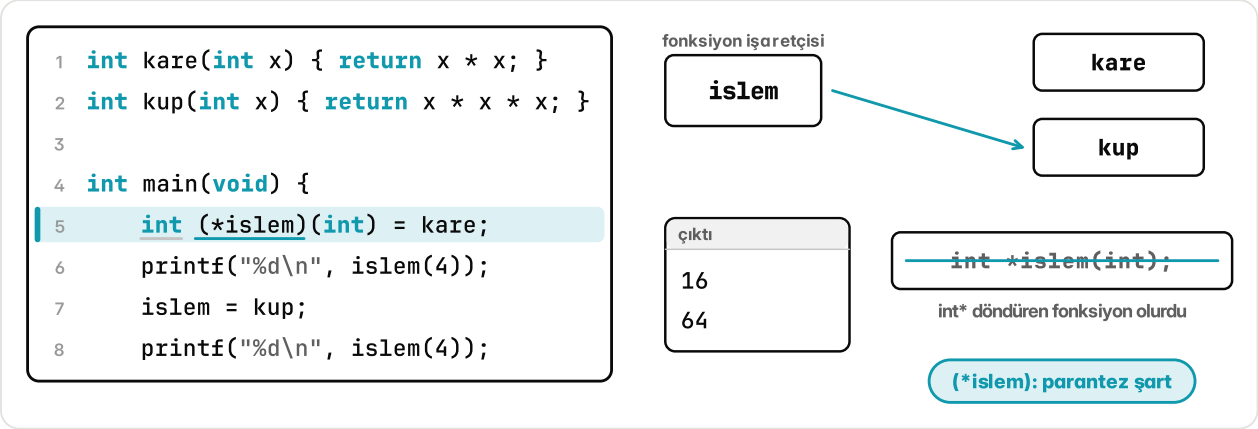
main'de islem adlı bir fonksiyon işaretçisi tanımlayalım:

```
int (*islem)(int) = kare;
printf("%d\n", islem(4));
islem = kup;
printf("%d\n", islem(4));
```

Çıktı:

```
16
64
```

islem, int alıp int döndüren bir fonksiyonun adresini tutar ve önce kare'yi gösterir; islem(4) 16 verir. Ona kup'u verip aynı çağrıyı yapınca 64 çıkar. Yıldızın çevresindeki parantez şarttır: int *islem(int); yazsaydık bu, int adresi döndüren islem adlı bir fonksiyonun bildirimi olurdu.

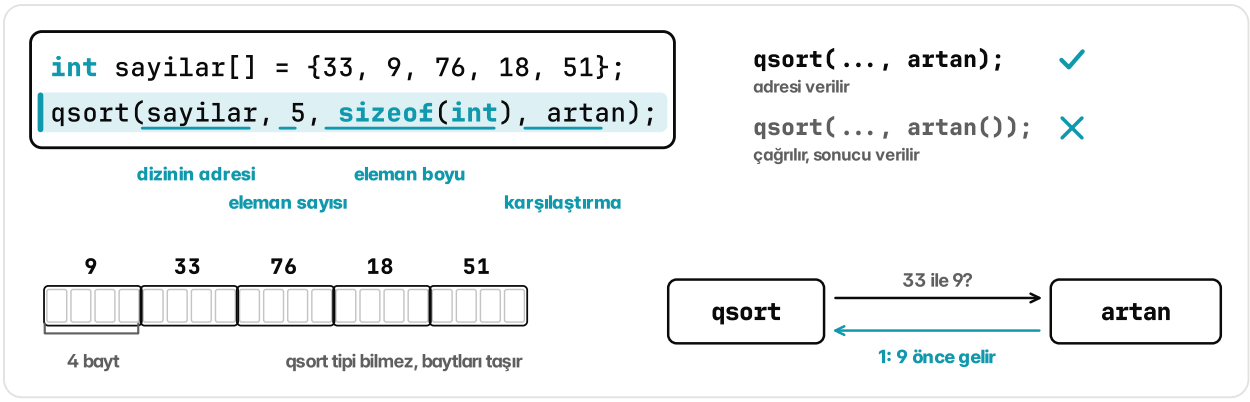


Şekil 7.6 islem fonksiyon işaretçisi önce kare'yi, sonra kup'u gösterir ve aynı islem(4) çağrısı 16 ile 64 verir; parantezsiz int *islem(int); ise int adresi döndüren bir fonksiyondur.

C'nin hazır sıralama fonksiyonu **qsort** (stdlib.h) bu fikirle çalışır. Ona dört şey veririz: dizinin adresi, eleman sayısı, bir elemanın boyu ve bir karşılaştırma fonksiyonu.

```
int sayilar[] = {33, 9, 76, 18, 51};
qsort(sayilar, 5, sizeof(int), artan);
```

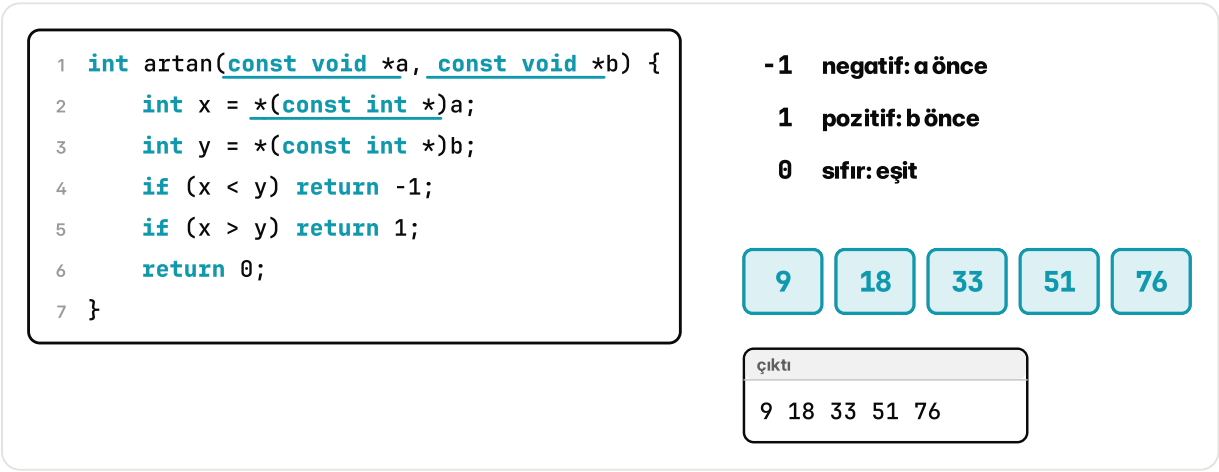
qsort hangi tipte veri sıraladığını bilmez; eleman boyunu bu yüzden ister ve yer değiştirirken o kadar baytı taşır. İki elemanı karşılaştırması gerektikçe de bizim fonksiyonumuzu çağırır. Fonksiyonu verirken yalnız adını yazarız, parantez koymayız; artan() yazmak onu çağırır, adresini vermez.



Şekil 7.7 qsort'un dört argümanı: dizinin adresi, eleman sayısı, eleman boyu (4 bayt) ve karşılaştırma fonksiyonu; qsort tipi bilmez, baytları taşır ve hangi elemanın önce geleceğini artan'a sorar.

Karşılaştırma fonksiyonu iki elemanın adresini tipsiz adres, yani void * olarak alır, çünkü qsort her tipte diziyle çalışmalıdır. Başındaki const, fonksiyonun bu adreslerdeki değerleri değiştirmeyeceğini söyler. Biz adresi int adresine çevirir, yıldızla adresteki değeri okuruz. İlk eleman önce gelmeliyse negatif, sonra gelmeliyse pozitif, eşitlerse 0 döndürürüz:

```
int artan(const void *a, const void *b) {
    int x = *(const int *)a;
    int y = *(const int *)b;
    if (x < y) return -1;
    if (x > y) return 1;
    return 0;
}
```



Şekil 7.8 artan fonksiyonunun dönüşleri: negatif a'nın, pozitif b'nin önce geldiğini, sıfır eşitliği söyler; 33, 9, 76, 18, 51 dizisi 9 18 33 51 76 olur.

Büyükten küçüğe sıralamak için qsort'a dokunmayız. azalan adlı ikinci fonksiyon artan'ın aynısıdır; yalnız küçük ve büyük işaretlerinin yeri değişir:

```
if (x > y) return -1;
if (x < y) return 1;
```

Dizi önce artan, sonra azalan ile sıralanıp yazdırılınca:

Çıktı:

```
9 18 33 51 76
76 51 33 18 9
```

Aynı sıralama kodu sayıyı da, metni de sıralar; neyin önce geleceğini ona verdiğimiz fonksiyon söyler.

Özet

- Özyinelemeli fonksiyon, sorunun küçük hâlini çözmek için kendini çağırır; durma koşulu ve sorunu küçülten bir adımı olmalıdır.
- Her çağrı stack'te kendi çerçevesini açar; durma koşulu unutulursa stack dolar ve program stack taşmasıyla çöker.
- Aynı hesabı tekrarlayan özyineleme çok yavaşlar: fib(40) 331 milyon çağrı ister; hafızalı sürüm 79 çağrı, döngü 40 tur yeter.
- Hanoi kuleleri gibi kendi küçüğüne bölünen problemlerde özyineleme en doğal yoldur; n disk $2^n - 1$ hamle ister.
- Fonksiyonun adı kodunun adresidir; bu adresi tutan değişkene fonksiyon işaretçisi denir.
- qsort, karşılaştırmayı ona verilen fonksiyona sorar: negatif, sıfır ya da pozitif dönüş sırayı belirler.

Sorular

1 **Özyinelemeli bir fonksiyonun çağrılarının sonsuza kadar sürmemesini ne sağlar?**

A Derleyicinin çağrı sayısını kendiliğinden sınırlaması

B Stack'in her çağrıdan sonra boşaltılması

C Fonksiyonun kodunun her çağrıda kopyalanması

D Her adımda sorunu küçültüp sonunda varılan bir durma koşulu

2 **Aşağıdaki fonksiyona göre rakam_topla(2026) kaç döndürür?**

```
int rakam_topla(int n) {
    if (n == 0)
        return 0;
    return n % 10 + rakam_topla(n / 10);
}
```

Cevap: _____

3 **rakam_topla(573) çalışırken stack'te aynı anda en çok kaç rakam_topla çerçevesi bulunur?**

A 3
B 4
C 1
D 15

4 **fib(n - 1) + fib(n - 2) ile yazılan düz özyinelemeli Fibonacci fonksiyonu fib(40) için neden 331 milyon kez çağrılır?**

A Durma koşulu yanlış yazıldığı için
B Fonksiyon bulduğu cevabı hatırlamaz; aynı değerler tekrar tekrar baştan hesaplanır
C Her çağrıda fonksiyonun kodu kopyalandığı için
D fib(40) 331 milyondan büyük bir sayı olduğu için

5 **Hanoi kulelerinde 5 diski bir çubuktan ötekine taşımak en az kaç hamle sürer?**

Cevap: _____

6 **kup(x), x * x * x döndüren bir fonksiyon olsun. Aşağıdaki satırlar ekrana ne yazar?**

```
int (*islem)(int) = kup;
printf("%d\n", islem(3));
```

Cevap: _____

7 **int (*islem)(int); ile int *islem(int); satırları arasındaki fark nedir?**

A İlki int alıp int döndüren bir fonksiyonun adresini tutan değişkendir; ikincisi int adresi döndüren bir fonksiyonun bildirimidir.
B İkisi aynıdır; parantez yalnız okunurluk içindir.
C İlki bir fonksiyon bildirimidir; ikincisi fonksiyon işaretçisidir.
D İlki derlenmez; fonksiyon işaretçisi ikinci biçimde yazılır.

8 **qsort ile küçükten büyüğe sıralayan artan fonksiyonu varken diziyi büyükten küçüğe sıralamak için ne değiştirilir?**

A qsort'a verilen eleman sayısı negatif yazılır.
B qsort'un kendi kodu değiştirilir.
C Karşılaştırma fonksiyonundaki küçük ve büyük işaretlerinin yeri değiştirilir.
D Eleman boyu olarak sizeof(int) yerine -sizeof(int) verilir.

PROGRAMLAMAYA GİRİŞ · 1. DÖNEM

8 Yapılar: struct

Bu bölümde birbirine ait bilgileri tek bir pakette toplamayı göreceksin. Önce struct ile yeni bir tip kurmayı, alanlara ulaşmayı, typedef, yapı dizisi ve enum'u; sonra yapının bellekte nasıl yerleştiğini, fonksiyona nasıl verildiğini ve ok işaretini öğreneceksin.

Üç ayrı dizi

videoda 38:44

Bir sınıfın adlarını bir dizide, numaralarını ikincisinde, notlarını üçüncüsünde tutuyoruz:

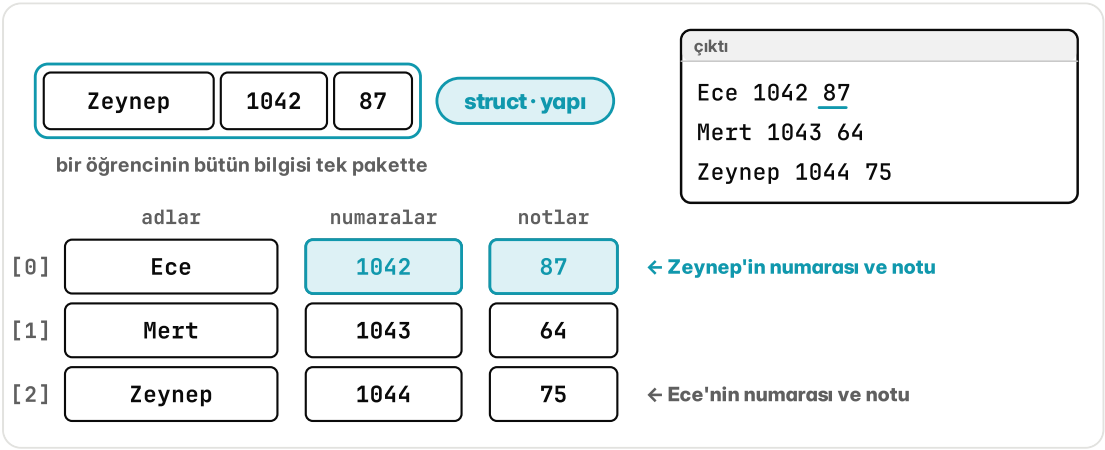
```
char adlar[3][16] = {"Zeynep", "Mert", "Ece"};
int numaralar[3] = {1042, 1043, 1044};
int notlar[3] = {87, 64, 75};
```

Zeynep'in notu 87, Ece'ninki 75. Adları alfabeye göre sıraladık: yalnız adlar dizisinde Zeynep ile Ece yer değiştirdi. Üç dizi satır satır yazdırılınca ekranda Ece'nin notu kaç görünür?

Çıktı:

```
Ece 1042 87
Mert 1043 64
Zeynep 1044 75
```

87. Adlar yer değiştirdi, numaralar ve notlar yerinde kaldı; Zeynep'in bilgileri artık Ece'nin satırında. Üç diziyi her işlemde birlikte taşımayı unutmak çok kolaydır. C'nin çözümü, bir öğrencinin bütün bilgisini tek pakette toplamaktır: **struct**, yani **yapı**.

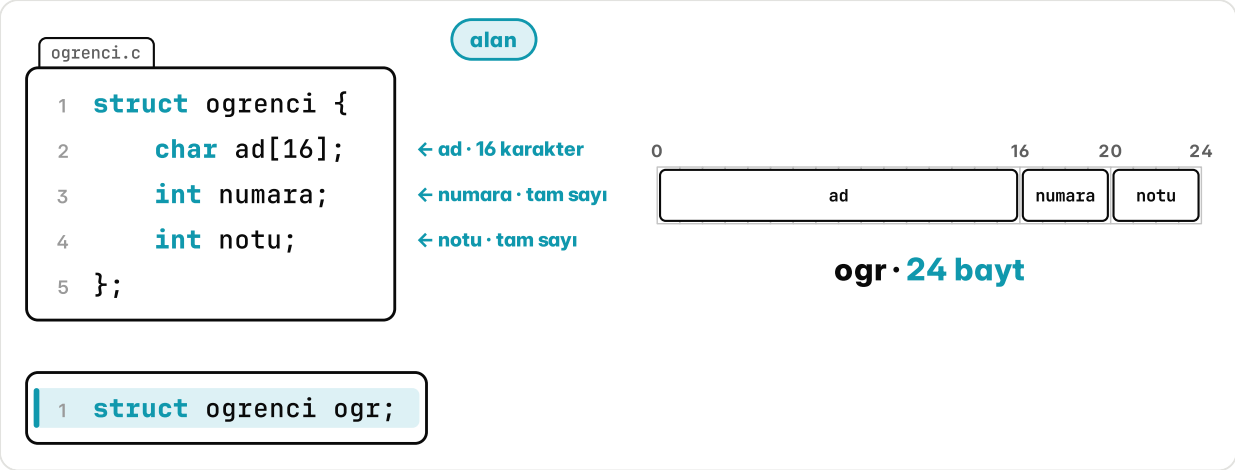


Şekil 8.1 Yalnız adlar sıralanınca Ece'nin satırında Zeynep'in numarası ve notu (1042, 87) kalır; çözüm, bir öğrencinin bütün bilgisini tek pakette toplayan struct.

Yapıyı tanımlarken içindeki her bilginin tipini ve adını yazarız; bunlara **alan** denir. Öğrenci yapısında üç alan var: 16 karakterlik ad, tam sayı numara ve tam sayı not.

```
struct ogrenci {
    char ad[16];
    int numara;
    int notu;
};
```

Tanım yalnız bir kalıptır; bellekte yer, bu tipte bir değişken tanımlanınca ayrılır. struct ogrenci ogr; yazınca ogr için 16 + 4 + 4 = 24 baytlık yer ayrılır.



Şekil 8.2 struct ogrenci'nin üç alanı ve struct ogrenci ogr; ile ayrılan 24 bayt: ad için 16, numara ve notu için 4'er bayt.

İlk değerler süslü parantez içinde, alanların sırasıyla verilir. Alana ulaşmak için değişkenin adı, nokta ve alanın adı yazılır:

```
struct ogrenci ogr = {"Zeynep", 1042, 87};
printf("%s %d\n", ogr.ad, ogr.notu);
ogr.notu = 92;
printf("%s %d\n", ogr.ad, ogr.notu);
```

Çıktı:

```
Zeynep 87
Zeynep 92
```

Notu 92 yapınca yalnız o alan değişir; ad ve numara olduğu gibi kalır.

Değişken tanımlarken her seferinde iki sözcük yazarız: önce struct, sonra ogrenci. **typedef**, var olan bir tipe yeni bir ad verir. typedef struct ogrenci Ogrenci; satırından sonra Ogrenci ogr2; yazmak yeter; iki yazım da aynı tipi gösterir. typedef yapının tanımıyla birlikte de yazılabilir:

```
typedef struct {
    char ad[16];
    int numara;
    int notu;
} Ogrenci;
```

Yapılardan dizi de kurulur: üç öğrenci, her biri kendi üç alanıyla yan yana durur. Önce köşeli parantezle öğrenciyi, sonra noktayla alanını seçeriz; ogrenciler[1].notu Mert'in notu olan 64'tür. Sıralarken Zeynep ile Ece'yi yer değiştirmek artık her öğrenci için tek atamadır: eşittir işareti üç alanı birlikte kopyalar.

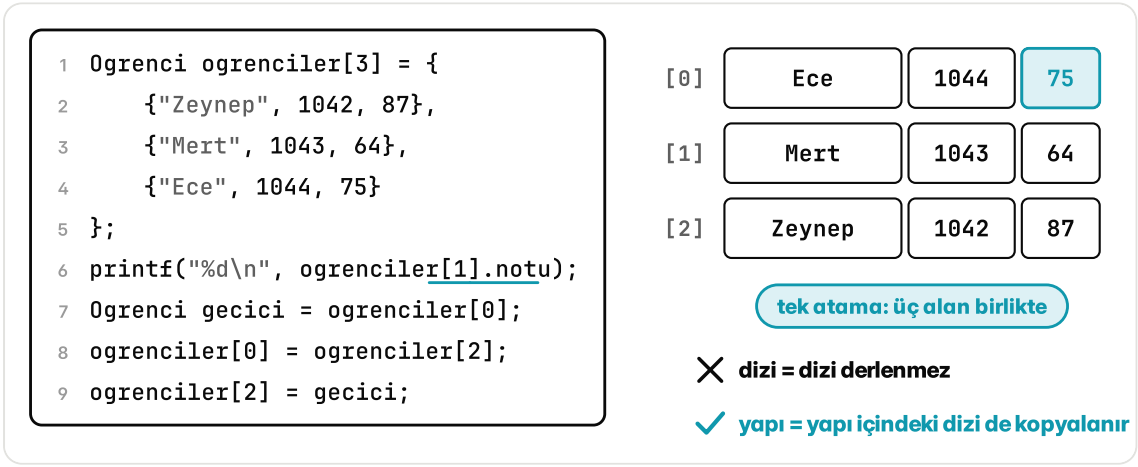
```
Ogrenci gecici = ogrenciler[0];
ogrenciler[0] = ogrenciler[2];
ogrenciler[2] = gecici;
```

Dizi bu atamalardan sonra satır satır yazdırılınca:

Çıktı:

```
Ece 1044 75
Mert 1043 64
Zeynep 1042 87
```

C'de bir dizi eşittir işaretiyle başka bir diziye kopyalanamaz; ama içinde dizi taşıyan yapı kopyalanabilir, ad dizisi de onunla birlikte gelir. Ece'nin notu artık hep 75.



Şekil 8.3 Üç Ogrenci kaydından oluşan yapı dizisi; [0] ile [2], üç alanıyla birlikte paket olarak yer değiştirir.

Öğrencinin dersteki sonucunu da sayıyla tutalım: 0 devam ediyor, 1 geçti, 2 kaldı. Ama kodu okuyan, 2'nin kaldı demek olduğunu bilemez. **enum** bu sayılara ad verir:

```
enum sonuc { DEVAM, GECTI, KALDI };
```

Derleyici adları 0'dan başlayarak sırayla numaralar: DEVAM 0, GECTI 1, KALDI 2. Kodda artık 2 değil, KALDI yazar:

```
enum sonuc s = KALDI;
if (s == KALDI)
    printf("Bütünlemeye girer\n");
s = 7;
printf("%d\n", s);
```

Çıktı:

```
Bütünlemeye girer
7
```

Yalnız C'de enum aslında bir tam sayıdır; değişkene 7 yazsak da derleyici ses çıkarmaz. Adlar okuyan içindir; doğru değeri vermek yine programcının işidir.

Aynı alanlar, iki boy

videoda 40:42

İki yapıda aynı üç alan var: tek karakterlik şube, tam sayı numara ve tek karakterlik sınıf. Tek fark sıraları. Birincisinde numara ortada:

```
struct kayit1 {
    char sube;
    int numara;
    char sinif;
};
```

İkincisinde en başta:

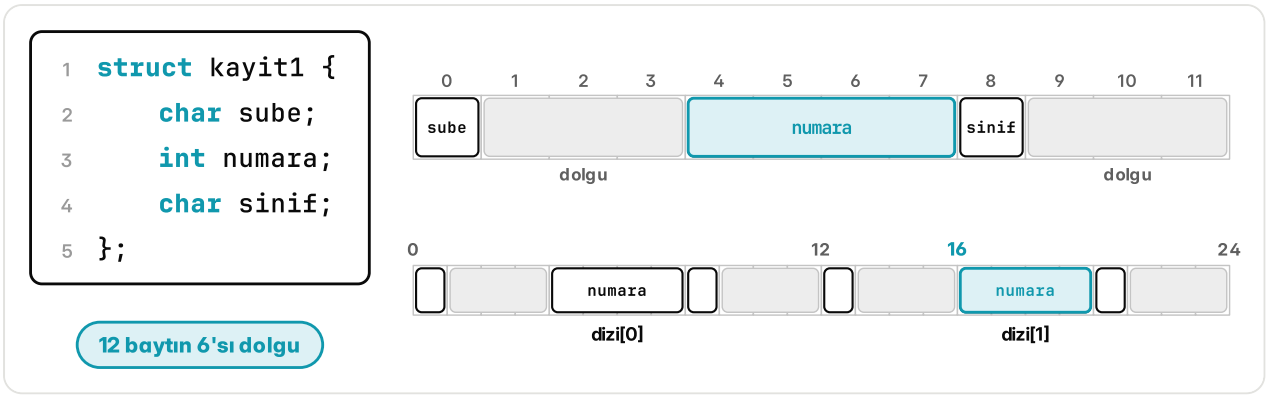
```
struct kayit2 {
    int numara;
    char sube;
    char sinif;
};
```

printf("%zu %zu\n", sizeof(struct kayit1), sizeof(struct kayit2)); satırı ekrana 12 8 yazar: biri 12 bayt, öteki 8. Alanlar aynıysa aradaki 4 bayt nereden geliyor?

char 1 bayttır; int 4 bayt. İşlemci belleği bayt bayt değil, dörder sekizer baytlık bloklarla okur. 4 baytlık sayı 4'ün katı olan bir adreste başlarsa tek okumada gelir; iki bloğa taşarsa iki okuma gerekir, bazı işlemciler hiç okuyamaz. Bu yüzden derleyici int'i 4'ün katı olan adrese koyar; buna **hizalama** (alignment) denir.

İlk yapıda şube 0. bayta oturur. Numara 1. baytta başlayamaz; derleyici 3 baytı boş bırakıp numarayı 4. bayttan başlatır. Sınıf 8. bayta düşer; ardından 3 bayt daha boş kalır. Sondaki boşluk dizi içindir: kayıtlar yan yana dizildiğinde her birinin numarası yine 4'ün katına düşsün. İki kayıtlık

bir dizide ikinci kaydın numarası 12 + 4 = 16. bayttadır. Bu boş baytlara **dolgu** (padding) denir; 12 baytın 6'sı dolgudur.

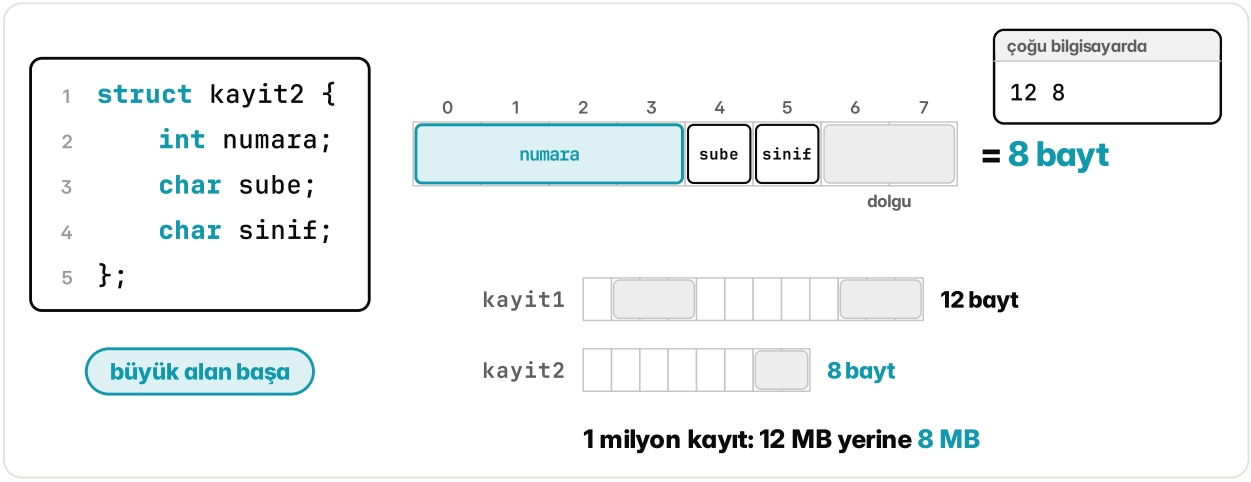


Şekil 8.4 kayit1'in 12 baytı: sube 0, dolgu 1-3, numara 4-7, sinif 8, dolgu 9-11; iki kayıtlık dizide ikinci kaydın numarası 16. baytta başlar.

İkinci yapıda numara en başta, ilk 4 baytta. İki karakter 4. ve 5. bayta yan yana oturur; sona yalnız 2 bayt dolgu eklenir, toplam 8:

```
kayit1:  sube 0 | dolgu 1-3 | numara 4-7 | sinif 8 | dolgu 9-11   = 12 bayt
kayit2:  numara 0-3 | sube 4 | sinif 5 | dolgu 6-7                 =  8 bayt
```

Kural basit: büyük alanları başa yaz. 1 milyon kayıtlık bir dizide bu fark 12 MB yerine 8 MB, yani 4 MB eder. Bu boylar çoğu bilgisayarda böyledir; kesin boyu sizeof söyler.



Şekil 8.5 kayit2'nin 8 baytı: numara 0-3, sube 4, sinif 5, dolgu 6-7; büyük alan başa yazılınca 1 milyon kayıt 12 MB yerine 8 MB tutar.

Yapıyı fonksiyona verince bütün baytları kopyalanır:

```
void ekle(Ogrenci o, int ek) {
    o.notu = o.notu + ek;
}
```

Ogrenci yapısı 24 bayttır. Mert'in kaydı (Mert, 1043, 64) ile ekle(ogr, 6); çağrılınca fonksiyon notu artırır, ama yalnız kopya değişir; ogr.notu 64 kalır. Bunun yerine yapının adresini veririz; adres yalnız 8 bayttır ve fonksiyon asıl kayda ulaşır. Adresi tutan değişkene işaretçi denir.

İşaretçiden alana ulaşmak için önce yıldızla yapıya gidip sonra nokta koyarız: (*p).notu. Noktanın önceliği yıldızdan yüksek olduğu için parantez şarttır; *p.notu, *(p.notu) demektir ve p yapı değil işaretçi olduğu için derlenmez. Bu uzun yazımın kısası **ok işareti**dir (→); p→notu ile (*p).notu aynı alanı gösterir:

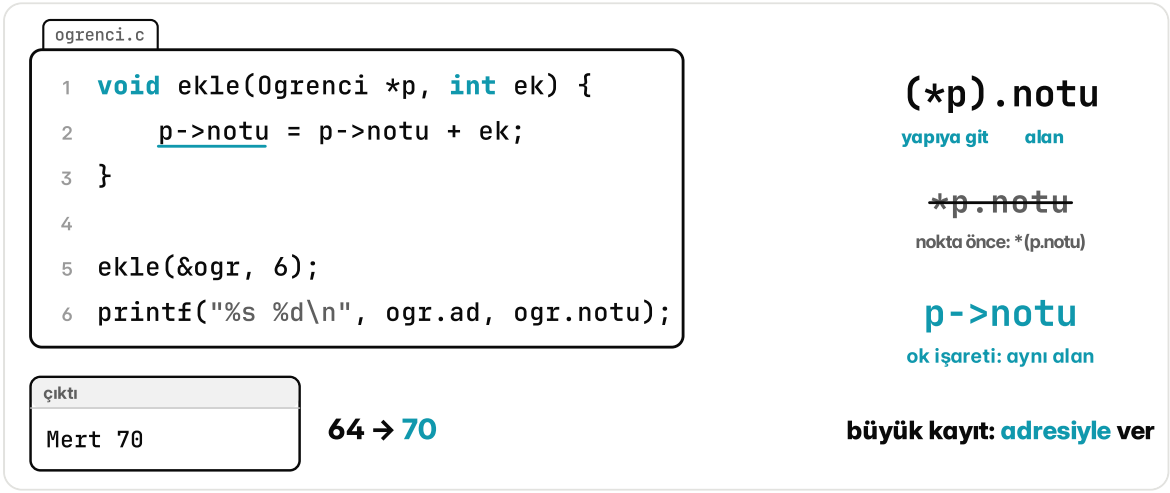
```
void ekle(Ogrenci *p, int ek) {
    p->notu = p->notu + ek;
}
```

ekle(&ogr, 6); çağrısından sonra ogr.ad ile ogr.notu yazdırılınca:

Çıktı:

```
Mert 70
```

Fonksiyon Mert'in notunu 64'ten 70'e çıkardı; değişen asıl kayıttır. Büyük kayıtları fonksiyonlara bu yüzden çoğunlukla adresleriyle veririz.



Şekil 8.6 (*p).notu ile p→notu aynı alanı gösterir, *p.notu ise *(p.notu) demektir; ekle(&ogr, 6) Mert'in notunu 64'ten 70'e çıkarır.

Özet

- struct, birbirine ait bilgileri alanlar hâlinde tek bir tipte toplar; alana değişken.alan ile ulaşılır.
- typedef var olan bir tipe yeni bir ad verir; yapı dizisinde önce köşeli parantezle eleman, sonra noktayla alan seçilir.
- Yapılar eşittir işaretiyle bütün alanlarıyla kopyalanır; içlerindeki dizi de kopyalanır.
- enum sayılara ad verir; adlar 0'dan başlayarak numaralanır, ama C'de enum değişkeni bir tam sayıdır.
- Derleyici alanları hizalar ve araya dolgu koyar; büyük alanları başa yazmak yapıyı küçültür.
- Yapı fonksiyona kopyalanarak verilir; asıl kaydı değiştirmek için adresi verilir ve alana p→alan ile ulaşılır.

Sorular

1 **struct ogrenci { char ad[16]; int numara; int notu; }; tanımı yazıldığında bellekte ne olur?**

A 24 baytlık bir ogrenci değişkeni ayrılır.

B Bellekte yer ayrılmaz; tanım yalnız bir kalıptır, yer bu tipte bir değişken tanımlanınca ayrılır.

C Yalnız ad dizisi için 16 bayt ayrılır.

D Heap'ten 24 bayt istenir ve adresi saklanır.

2 **Bölümdeki struct ogrenci tipine göre aşağıdaki satırlar ekrana ne yazar?**

```
struct ogrenci ogr = {"Mert", 1043, 64};
ogr.notu = ogr.notu + 10;
printf("%s %d\n", ogr.ad, ogr.notu);
```

Cevap:

3 **enum donem { GUZ, BAHAR, YAZ }; tanımında YAZ'ın değeri kaçtır?**

A 3

B 1

C 0

D 2

4 **a ve b aynı yapı tipinde iki değişken, x ve y de char x[16], y[16]; diye tanımlanmış iki dizi. Hangisi doğrudur?**

A a = b; derlenmez; yapılar alan alan kopyalanmalıdır.

B x = y; dizinin bütün elemanlarını kopyalar.

C a = b; yapının bütün alanlarını, içindeki dizi dahil kopyalar; x = y; ise derlenmez.

D İkisi de yalnız ilk elemanı ya da ilk alanı kopyalar.

5 **int'in 4 bayt olduğu ve 4'ün katı olan adreste başladığı bir bilgisayarda struct k { char a; char b; int c; }; yapısının sizeof değeri kaçtır?**

Cevap: _____

6 **struct kayit1 { char sube; int numara; char sinif; }; yapısının sonuna da 3 bayt dolgu eklenir. Bunun nedeni nedir?**

A Kayıtlar yan yana dizildiğinde her kaydın numara alanı yine 4'ün katı olan bir adrese düşsün diye

B sinif alanı 4 bayta tamamlanmadan okunamadığı için

C Yapıların boyu her zaman 12 bayt olmak zorunda olduğu için

D Sondaki baytlar bir sonraki değişkenin adresini tutmak için

7 **void ekle(Ogrenci o, int ek) { o.notu = o.notu + ek; } fonksiyonu, notu 64 olan ogr için ekle(ogr, 6); diye çağrılıyor. Çağrıdan sonra ogr.notu kaçtır?**

A 70

B 6

C 64

D Derleme hatası verir

8 **p bir Ogrenci işaretçisi olsun. p→notu yazımı neyin kısasıdır?**

A *p.notu

B &p.notu

C p.notu

D (*p).notu

9 Union ve bitsel işlemler

Bir tam sayının bitleri tek tek de işlenebilir. Bu bölümde C'nin bit bit çalışan altı operatörünü ve tuzaklarını, bu operatörlerle bir baytın tek bir bitini açıp kapatmayı, bit alanlarını ve aynı baytları farklı tiplerle okumaya yarayan union'ı öğreneceksin.

İkisi de var mı?

videoda 42:35

Sepette 4 elma ve 2 armut var. Aşağıdaki program ikisinin de olup olmadığını soruyor, ama ekrana hiçbir şey yazmıyor.

```
#include <stdio.h>

int main(void) {
    int elma = 4, armut = 2;
    if (elma & armut)
        printf("ikisi de var\n");
    return 0;
}
```

Hata koşuldaki tek bir karakterdedir: & işareti bir kez yazılmış. C'de iki ayrı "ve" vardır. Çift && **mantıksal ve** yapar: iki tarafın da 0'dan farklı olup olmadığına bakar; ikisi de farklıysa sonuç 1 (doğru), değilse 0 (yanlış) olur. 4 de 2 de 0 olmadığı için elma && armut doğrudur. Tek & ise **bitsel ve** yapar: sayıları bit bit karşılaştırır ve yalnız aynı sırada ikisinde birden 1 olan bitleri 1 bırakır. 4 ikilikte 100, 2 ise 010'dur; aynı sırada ikisinde birden 1 olan bit yoktur:

```
100     (4)
010     (2)
000     4 & 2 = 0: yanlış
```

İki yazım da geçerli C olduğu için derleyici uyarmaz; hata ancak sonuç yanlış çıkınca fark edilir. Doğrusu &&'dir. Koşul düzeltilip iki ifadenin değeri de yazdırılınca fark açıkça görünür:

```
int elma = 4, armut = 2;
if (elma && armut)
    printf("ikisi de var\n");
printf("%d %d\n", elma & armut, elma && armut);
```

Çıktı:

```
ikisi de var
0 1
```



Şekil 9.1 elma && armut iki tarafın da 0'dan farklı olduğunu görüp 1 veriyor; elma & armut 100 ile 010'u bit bit karşılaştırıp 000 = 0 veriyor.

Bit bit çalışan altı operatör vardır: & (ve), | (veya), ^ (özel veya), ~ (değil), << (sola kaydırma) ve >> (sağa kaydırma). Ve, iki bit de 1 ise 1 verir; veya en az biri 1 ise, özel veya ise bitler farklıysa. Değil tek sayıyla çalışır ve bütün bitleri çevirir; kaydırma bitleri verilen basamak kadar sola ya da sağa iter. 12 (1100) ile 10 (1010) alt alta yazılınca dört bit çiftinin hepsi (1-1, 1-0, 0-1, 0-0) bir arada görülür:

```
1100     a = 12
1010     b = 10
1000     a & b = 8
1110     a | b = 14
0110     a ^ b = 6
```

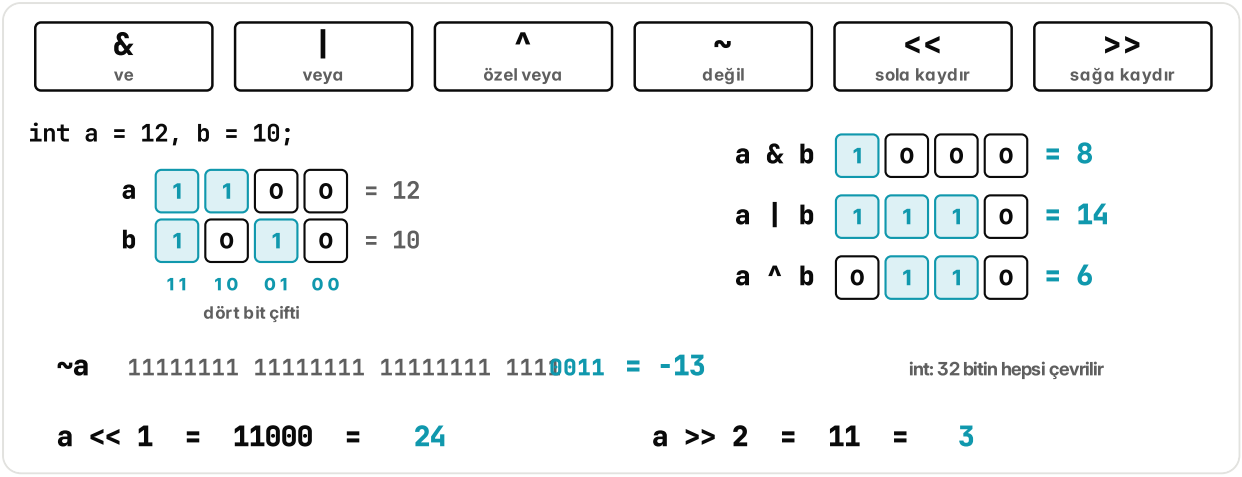
Aynı işlemler C'de:

```
int a = 12, b = 10;    /* 1100, 1010 */
printf("%d %d %d\n", a & b, a | b, a ^ b);
printf("%d\n", ~a);
printf("%d %d\n", a << 1, a >> 2);
```

Çıktı:

```
8 14 6
-13
24 3
```

~a neden −13 verir? int 32 bittir ve değil işlemi 32 bitin hepsini çevirir; 1100'ün önündeki 28 sıfır da 1'e döner: 11111111 11111111 11111111 11110011. En soldaki bit 1 olduğu için sayı eksidir: ikinin tümleyeninde bu bitler −13'tür. Genel olarak ~x = −x − 1 olur. Kaydırmada a << 1 sağa bir 0 ekler: 11000 = 24. a >> 2 sağdaki iki biti atar: 11 = 3.



Şekil 9.2 1100 ve 1010 için ve 8, veya 14, özel veya 6 veriyor; ~a 32 bitin hepsini çevirip −13 veriyor, a << 1 = 24, a >> 2 = 3.

En sağdaki bit, sayının tek mi çift mi olduğunu söyler: x & 1 tek sayıda 1, çift sayıda 0'dır. 7 = 0111 için 1, 8 = 1000 için 0 çıkar. Kalana bakmaktan (x % 2) farkı eksi sayılardadır: C'de kalan bölünenin işaretini alır ve −9 % 2 = −1 olur; −9'un en sağdaki biti ise 1'dir, −9 & 1 = 1 verir. Yalnız parantezi unutma: C, == karşılaştırmasını &'den önce yapar.

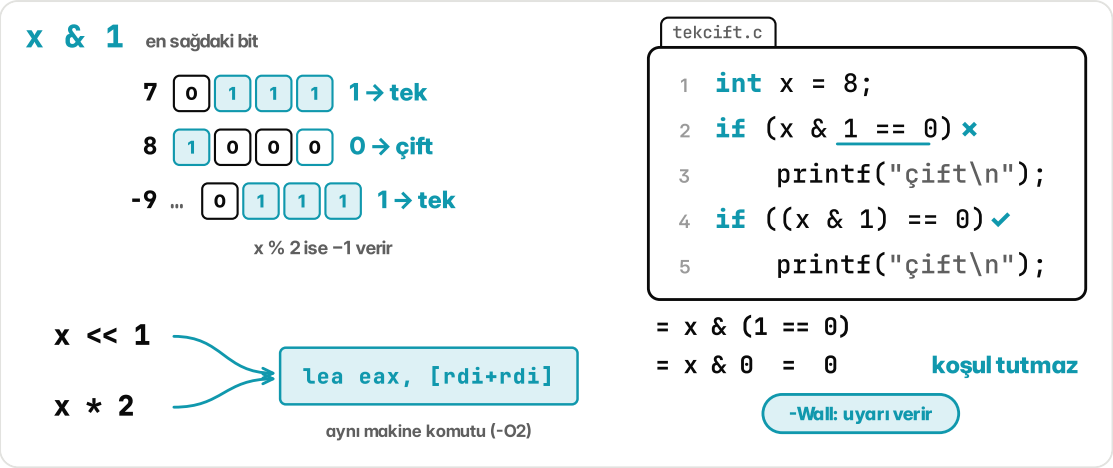
```
#include <stdio.h>

int main(void) {
    int y = -9;
    printf("%d %d\n", y & 1, y % 2);
    int x = 8;
    if (x & 1 == 0)
        printf("çift\n");
    if ((x & 1) == 0)
        printf("çift (parantezli)\n");
    printf("%d\n", x * 2);
    return 0;
}
```

Çıktı:

```
1 -1
çift (parantezli)
16
```

Parantezsiz koşul x & (1 == 0) diye okunur. 1, 0'a eşit olmadığı için 1 == 0 ifadesi 0 verir; x & 0 da her x için 0'dır. 8 çift olduğu hâlde ilk koşul tutmaz, yalnız parantezli satır yazar. Neyse ki derleyici, uyarılarını açan -Wall seçeneğiyle bu satırı işaretler.

Son kısa yol kaydırmadır: bir sayıyı sola 1 basamak kaydırmak onu 2 ile çarpar, 8 << 1 = 16'dır. Yine de x * 2 yaz. Derleyici iyileştirme yaparken ikisini de aynı makine komutuna çevirir; x * 2 ise ne yapıldığını okuyana açıkça söyler.



Şekil 9.3 x & 1, 7 ve −9 için 1, 8 için 0 veriyor; parantezsiz x & 1 == 0 koşulu x & 0 = 0'a dönüşüyor, x << 1 ile x * 2 aynı makine komutuna çevriliyor.

Lamba yandı, motor durdu

videoda 44:16

Arduino gibi küçük denetleyici kartlarında birkaç pinin açık-kapalı durumunu tek bir bayt tutar. Her bit bir pine bağlıdır; bit 1 olunca o pin 5 volt verir. Arduino Uno'da bu baytın adı PORTB'dir: 0. biti 8 numaralı pine, 5. biti 13 numaralı pine gider. Örnekte 0. bite bir motor, 5. bite bir lamba bağlı. Motor çalışırken lambayı yakmak için port = (1 << 5) yazılıyor; lamba yanıyor, ama motor da duruyor. Aşağıdaki programlarda bu baytın yerine port adlı sıradan bir unsigned char değişkeni kullanılıyor; bitler fonksiyonu baytın 8 bitini soldan sağa yazıyor.

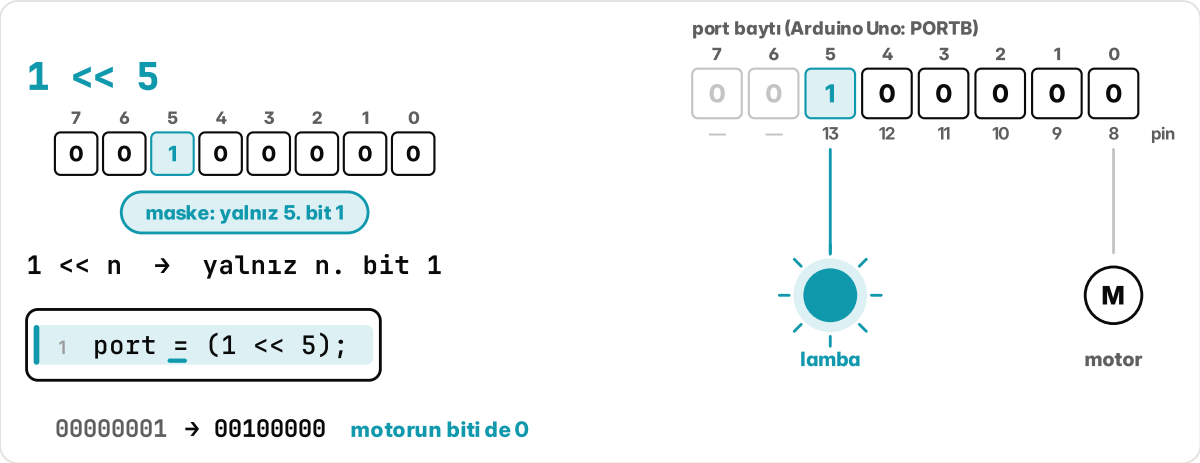
```
#include <stdio.h>

void bitler(unsigned char b) {    /* baytı 8 bit olarak yazar */
    for (int i = 7; i >= 0; i--)
        putchar((b >> i) & 1 ? '1' : '0');
    putchar('\n');
}

int main(void) {
    unsigned char port = 0x01;    /* motor: 0. bit */
    port = (1 << 5);              /* lambayı yak? */
    bitler(port);
    return 0;
}
```

Çıktı:

```
00100000
```

C'de bitler sağdan ve 0'dan sayılır: en sağdaki bit 0. bittir. Motor 0. bitte, lamba 5. bitte durur. 1 << 5 ifadesi 00000001'deki tek 1'i 5 basamak sola iter ve 00100000 = 32 olur. Yalnız 5. biti 1 olan bu sayıya **maske** denir. Hangi bit gerekiyorsa 1, onun sırası kadar sola kaydırılır: 1 << n, yalnız n. biti 1 olan maskedir.

Hata = işaretindedir. Atama baytın tamamını maskeyle değiştirir: port 00000001 iken 00100000 olur. Lambanın biti 1 olur, ama motorun biti de 0'a iner ve motor durur.



Şekil 9.4 Port baytında motor 0., lamba 5. bitte; 1 << 5 maskesi 00100000, port = (1 << 5) ataması baytı 00000001'den 00100000'a çevirip motorun bitini de siliyor.

Doğrusu |= yazmaktır. port |= (1 << 5), port = port | (1 << 5) satırının kısa yazımıdır: maskedeki 1 lambanın bitini açar, maskedeki 0'lar öteki bitleri olduğu gibi bırakır. Söndürmek için maske ~ ile ters çevrilir: 5. bit 0, ötekilerin hepsi 1 olur. &= ile bu tek 0 lambanın bitini kapatır, 1'ler öteki bitleri korur. ^= biti çevirir: lamba yanıyorsa söner, sönükse yanar. Okumak için & yeter: sonuç 0 değilse lamba yanıyordur.

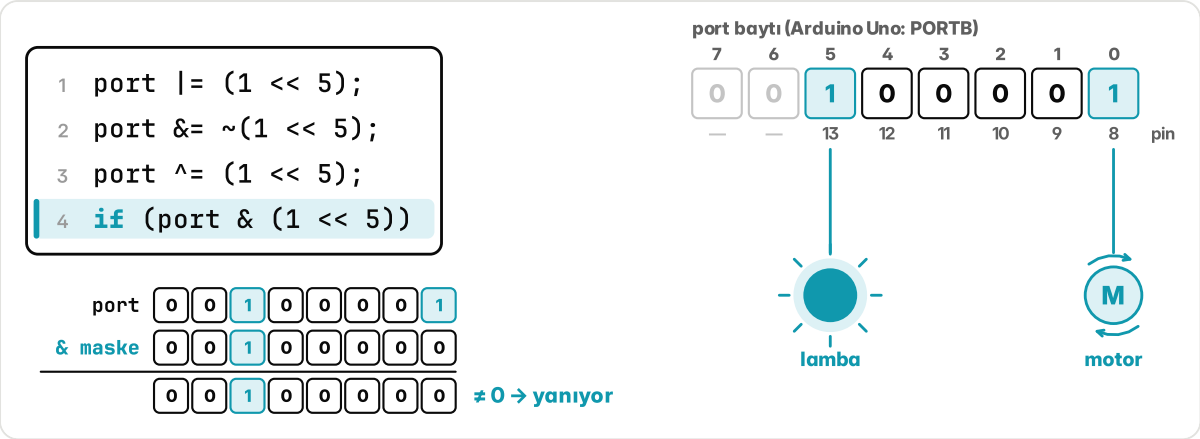
```
unsigned char port = 0x01;    /* motor çalışıyor */
port |= (1 << 5);             /* lambayı yak */
bitler(port);
port &= ~(1 << 5);            /* lambayı söndür */
bitler(port);
port ^= (1 << 5);             /* lambayı çevir */
bitler(port);
if (port & (1 << 5))          /* lamba yanıyor mu? */
    printf("lamba yanıyor\n");
```

Çıktı:

```
00100001
00000001
00100001
lamba yanıyor
```

Her satırda bayt, maskeyle bit bit işlenir (~ ile çevrilen maskenin baytın 8 bitine düşen kısmı 11011111'dir):

```
00000001 | 00100000 = 00100001     lamba yandı, motor çalışıyor
00100001 & 11011111 = 00000001     lamba söndü
00000001 ^ 00100000 = 00100001     lamba çevrildi: yandı
00100001 & 00100000 = 00100000     0 değil: lamba yanıyor
```



Şekil 9.5 Aç, kapat, çevir ve oku satırları port baytını maskeyle bit bit işliyor; lamba yanıp sönerken motorun biti 1 kalıyor.

Bu maskelere çoğu zaman ad verilir; adı olan maskeye **bayrak** denir. #define bir ada bir değer bağlar: derlemeden önce ad, yazıldığı her yerde bu değerle değiştirilir.

```
#define MOTOR (1 << 0)
#define LAMBA (1 << 5)
```

Artık satırda 1 << 5 yerine LAMBA yazar ve okuyan ne yapıldığını hemen anlar. Birden çok bayrak | ile birleşir: MOTOR | LAMBA iki biti birden 1 olan maskedir ve ikisini tek satırda açar.

```
unsigned char port = 0;
port |= LAMBA;
printf("%d\n", port);
port |= MOTOR | LAMBA;
printf("%d\n", port);
```

Çıktı:

```
32
33
```

Lamba zaten yanarken ikinci satır onu değiştirmez; yalnız motorun biti eklenir ve bayt 00100001 = 33 olur.

C, bitlerle çalışmanın bir yolunu daha sunar. Alanları bir arada tutan struct'ta bir alanın yanına : 1 yazılınca alan yalnız 1 bit yer kaplar; buna **bit alanı** denir. unsigned, unsigned int'in kısa yazımıdır.

```
struct durum {
    unsigned motor : 1;
    unsigned lamba : 1;
};
```

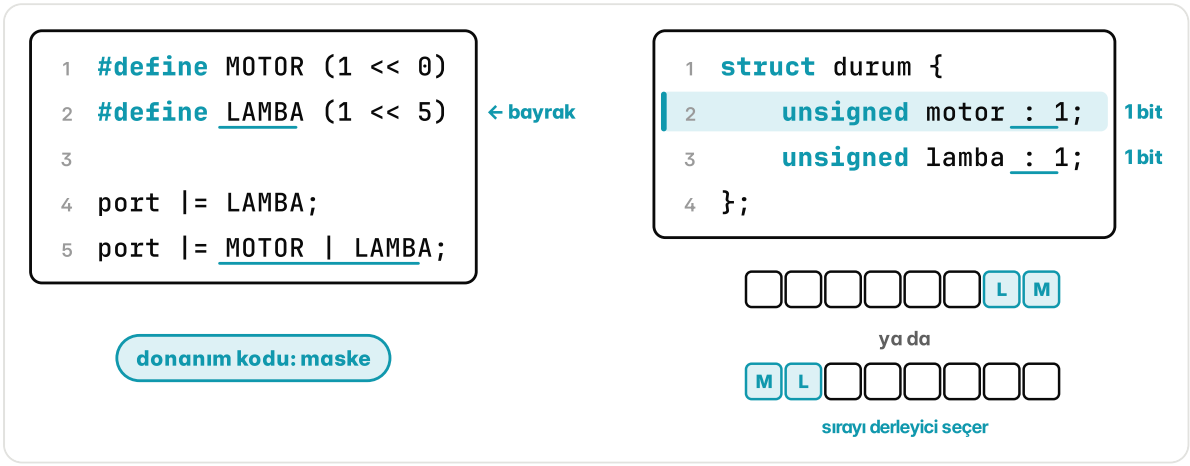
Bit alanlarına sıradan alanlar gibi erişilir:

```
struct durum d = {1, 0};
d.lamba = 1;
printf("%u %u\n", d.motor, d.lamba);
```

Çıktı:

```
1 1
```

Ama bu bitlerin baytta hangi sırayla dizileceğini, yani motorun sağda mı solda mı duracağını C belirlemez; derleyici seçer. Donanıma yazan kodun bitin yerini kesin bilmesi gerektiği için C'de donanım kodu çoğunlukla maskeleri kullanır.



Şekil 9.6 struct durum'da motor ve lamba alanları birer bit kaplıyor; bu iki bitin baytta hangi sırayla duracağını derleyici seçiyor.

Aynı 4 bayt

videoda 45:41

Aynı 4 bayt hem tam sayı hem kesirli sayı olabilir mi? C'de olabilir: 1,0 kesirli sayı olarak yazılıp aynı 4 bayt tam sayı diye okununca 1.065.353.216 çıkar.

```
#include <stdio.h>

union sayi {
    int   tam;
    float kesirli;
};

int main(void) {
    union sayi s;
    s.kesirli = 1.0f;
    printf("%d\n", s.tam);
    return 0;
}
```

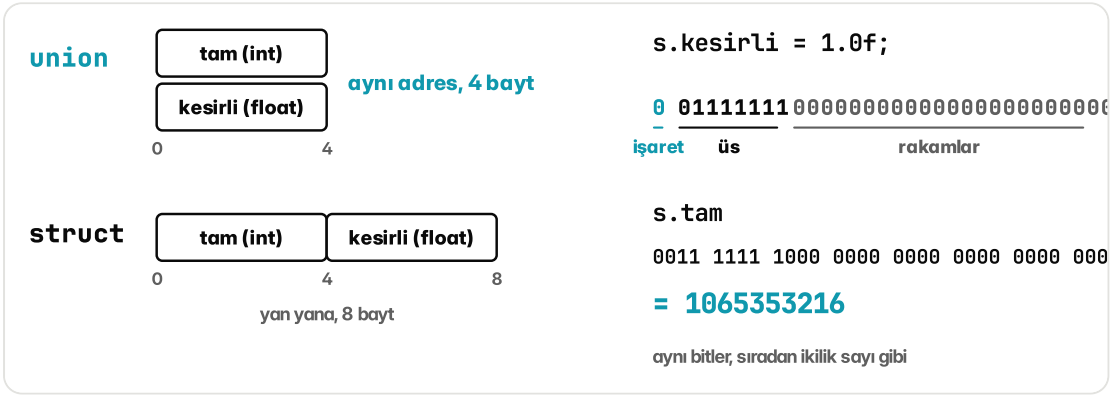
Çıktı:

```
1065353216
```

Buradaki sayi bir **union**'dır. union, struct gibi yazılır, ama alanları yan yana değil, aynı adreste üst üste durur: tam (int) da kesirli (float) da union'ın ilk baytından başlar ve aynı 4 baytı kullanır. 1.0f yazımındaki f, sabitin float olduğunu söyler. s.kesirli alanına yazılan 4 bayt, s.tam alanından okununca olduğu gibi tam sayı sayılır; C'de union'ın son yazılan alanından başka bir alanı okununca baytlar o alanın tipiyle yorumlanır.

Kesirli sayıda bu 32 bit işaret, üs ve rakamlar (mantis) diye bölünmüştür. 1,0 = 1,0 × 2^0 olduğu için işaret 0, rakamlar hep 0'dır; üs 127 eklenerek saklandığı için 0 üssü 01111111 diye yazılır. Tam sayı ise aynı bitleri sıradan bir ikilik sayı gibi okur:

```
işaret   üs         rakamlar
0        01111111   00000000000000000000000

0011 1111 1000 0000 0000 0000 0000 0000  =  0x3F800000  =  1.065.353.216
```



Şekil 9.7 union'da tam ve kesirli alanları aynı adreste üst üste, struct'ta yan yana duruyor; 1.0f'nin bitleri tam sayı olarak okununca 1.065.353.216 çıkıyor.

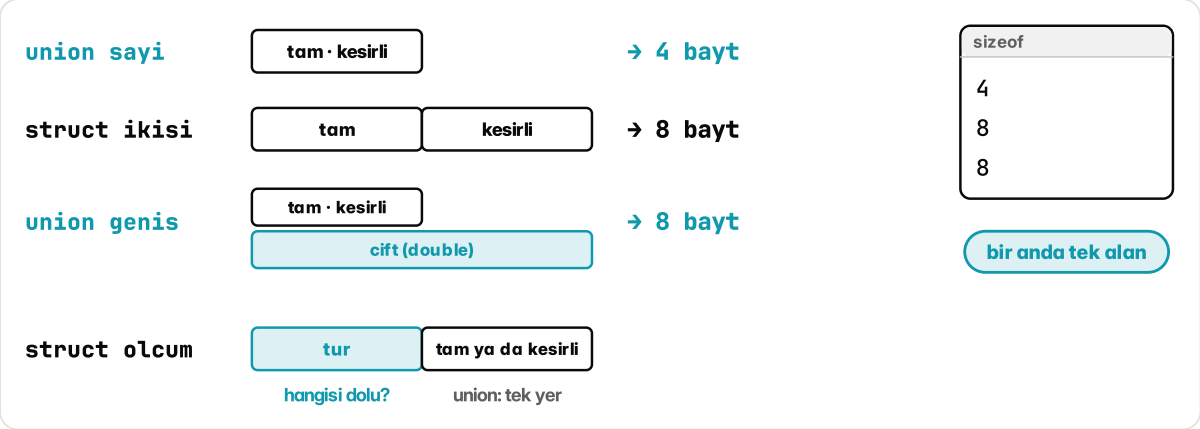
union'ın boyu en büyük alanı kadardır. sizeof bir tipin bellekte kaç bayt tuttuğunu verir; %zu bu değeri yazdırır.

```
#include <stdio.h>

union sayi   { int tam; float kesirli; };
struct ikisi { int tam; float kesirli; };
union genis  { int tam; float kesirli; double cift; };

int main(void) {
    printf("%zu\n", sizeof(union sayi));
    printf("%zu\n", sizeof(struct ikisi));
    printf("%zu\n", sizeof(union genis));
    return 0;
}
```

Çıktı:

```
4
8
8
```

int de float da 4 bayttır: union 4 bayt tutar, aynı iki alan struct içinde yan yana 8 bayt tutar. 8 baytlık bir double eklenince union da 8 bayta çıkar. union'da bir anda yalnız bir alan anlamlıdır, çünkü bir alana yazmak ötekinin baytlarını da değiştirir. Bu yüzden union, aynı anda gerekmeyen değerlerin belleği paylaşması için kullanılır. Örneğin bir ölçüm kimi zaman tam sayı, kimi zaman kesirli sayı olarak geliyorsa ikisine ayrı yer açılmaz: değer bir union'da tutulur, hangisinin dolu olduğunu da yanındaki küçük bir alan söyler; ikisi bir struct içinde bir arada durur.



Şekil 9.8 union sayi 4, struct ikisi 8, union genis 8 bayt tutuyor; altta türü bir alanda, değeri tek bir union'da tutan struct olcum.

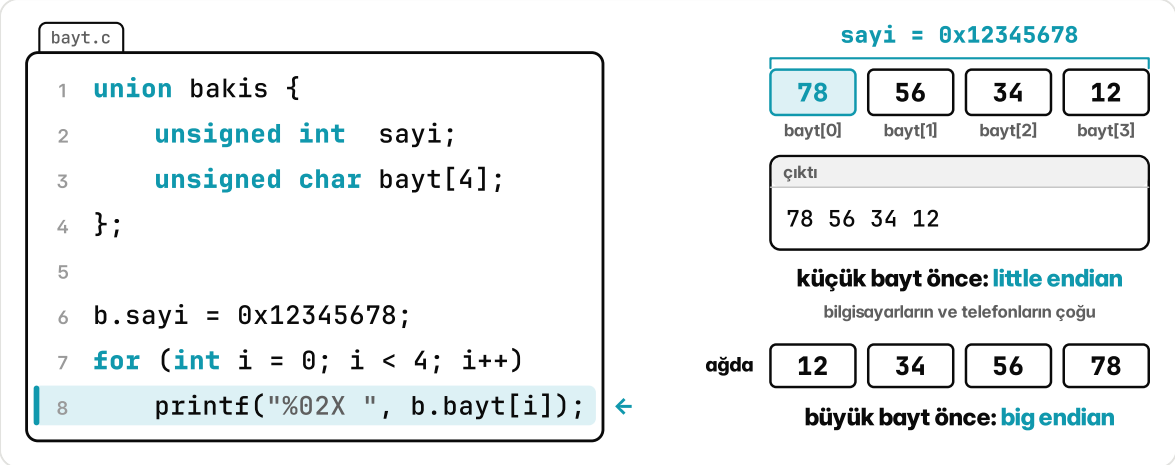
union, bir sayının baytlarına bakmanın da yoludur: 4 baytlık bir sayıyla 4 elemanlı bir bayt dizisi üst üste konur.

```
#include <stdio.h>

union bakis {
    unsigned int  sayi;
    unsigned char bayt[4];
};

int main(void) {
    union bakis b;
    b.sayi = 0x12345678;
    for (int i = 0; i < 4; i++)
        printf("%02X ", b.bayt[i]);
    printf("\n");
    return 0;
}
```

Çıktı:

```
78 56 34 12
```

%02X biçimi baytı onaltılık, iki haneli ve büyük harfle yazar. Sayıya 0x12345678 yazıldı, ama bayt[0]'dan başlayan sıra 78 56 34 12 çıktı. Yani bilgisayar sayının en küçük basamaklı baytını (78) önce, en küçük adrese koyar; buna **little endian** (küçük bayt önce) denir. Bugünkü bilgisayarların ve telefonların neredeyse hepsi böyle çalışır. Ağ ise sayıları büyük bayt önde gönderir: 12 34 56 78. Buna da **big endian** (büyük bayt önce) denir. Bu yüzden dosyadan ya da ağdan bayt bayt veri okuyan C kodu, sayının hangi sırayla geldiğini bilmek zorundadır; aynı 4 bayt ters sırayla okunursa bambaşka bir sayı çıkar.



Şekil 9.9 0x12345678 sayısının baytları bellekte 78 56 34 12 sırasıyla (little endian) duruyor; ağda 12 34 56 78 sırasıyla (big endian) gönderiliyor.

Özet

- & ve && farklıdır: && iki tarafın 0'dan farklı olup olmadığına, & aynı sıradaki bitlere bakar; 4 && 2 = 1, 4 & 2 = 0.
- Bit operatörleri &, |, ^, ~, << ve >>'dir; ~ int'in 32 bitinin hepsini çevirdiği için ~12 = −13 olur.
- C, == karşılaştırmasını &'den önce yapar; x & 1 == 0 yerine (x & 1) == 0 yazılır.
- 1 << n yalnız n. biti 1 olan maskedir; |= biti açar, &= ~ kapatır, ^= çevirir, & okur, = ise baytın tamamını değiştirir.
- union'ın alanları aynı adreste üst üste durur; boyu en büyük alanı kadardır ve bir anda tek alan anlamlıdır.
- Bugünkü bilgisayarlar little endian'dır: sayının en küçük basamaklı baytı en küçük adrestedir; ağ ise big endian kullanır.

Sorular

1 **int elma = 4, armut = 2; olduğuna göre elma & armut ile elma && armut ifadelerinin değerleri sırasıyla nedir?**

A 1 ve 1

B 6 ve 1

C 0 ve 1

D 0 ve 0

2 **printf("%d\n", 6 ^ 3); satırı ne yazar?**

Cevap:

3 **unsigned char port = 0x01;**

```
port |= (1 << 3);
Bu iki satırdan sonra port'un değeri onluk tabanda kaçtır?
```

Cevap:

4 **int x = 6;**

```
if (x & 1 == 0)
    printf("çift\n");
6 çift olduğu hâlde bu kod neden hiçbir şey yazmaz?
```

A C, == karşılaştırmasını &'den önce yapar; koşul x & (1 == 0), yani x & 0 olur ve bu her zaman 0'dır.

B 6'nın en sağdaki biti 1 olduğu için x & 1 sonucu 1'dir.

C & yalnız unsigned tiplerde çalışır; int'te sonuç hep 0'dır.

D x & 1 yalnız eksi sayılarda doğru sonuç verir.

5 **port baytında lamba 5. bite bağlı ve lambanın o anda yanıp yanmadığını bilmiyorsun. Öteki bitlere dokunmadan lambayı her durumda söndüren satır hangisidir?**

A port = ~(1 << 5);

B port ^= (1 << 5);

C port |= ~(1 << 5);

D port &= ~(1 << 5);

6 **int a = 5;**

```
printf("%d\n", ~a);
Bu kod ne yazar?
```

Cevap:

7 union u { int tam; float kesirli; double cift; }; tanımında int ve float 4, double 8 bayt tutuyorsa sizeof(union u) kaçtır?

A 16

B 8

C 4

D 12

8 union bakis { unsigned int sayi; unsigned char bayt[4]; };

```
union bakis b;
b.sayi = 0x0A0B0C0D;
Little endian bir bilgisayarda bayt[0]'dan bayt[3]'e kadar sırayla %02X ile
yazdırılırsa ne çıkar?
```

A 0A 0B 0C 0D

B 0B 0A 0D 0C

C 0D 0C 0B 0A

D 0C 0D 0A 0B

PROGRAMLAMAYA GİRİŞ · 1. DÖNEM

10 Bağlı listeler

Bu bölümde elemanları bellekte yan yana değil, adreslerle birbirine bağlanmış düğümlerde tutan bağlı listeyi öğreneceksin: düğümün nasıl tanımlandığını, listeye eleman eklemeyi, listeyi dolaşmayı ve düğüm silmeyi, bağlı listenin diziye ne zaman tercih edildiğini.

Bir milyon adım

videoda 47:11

Bir milyon elemanlı bir dizinin başına tek bir sayı eklemek istiyoruz. Bilgisayar bu işi neden bir milyon adımda yapar?

Dizinin elemanları bellekte yan yana, arada boşluk olmadan durur: 4 baytlık int'lerden oluşan bir dizide [0] 1000 adresindeyse [1] 1004'te, [2] 1008'dedir. Başta boş yer yoktur; yeni sayıya yer açmak için her eleman bir sağa kaymalıdır. Aşağıdaki program 20, 35, 50 dizisinin başına 5 ekler ve kaç kopya yaptığını sayar:

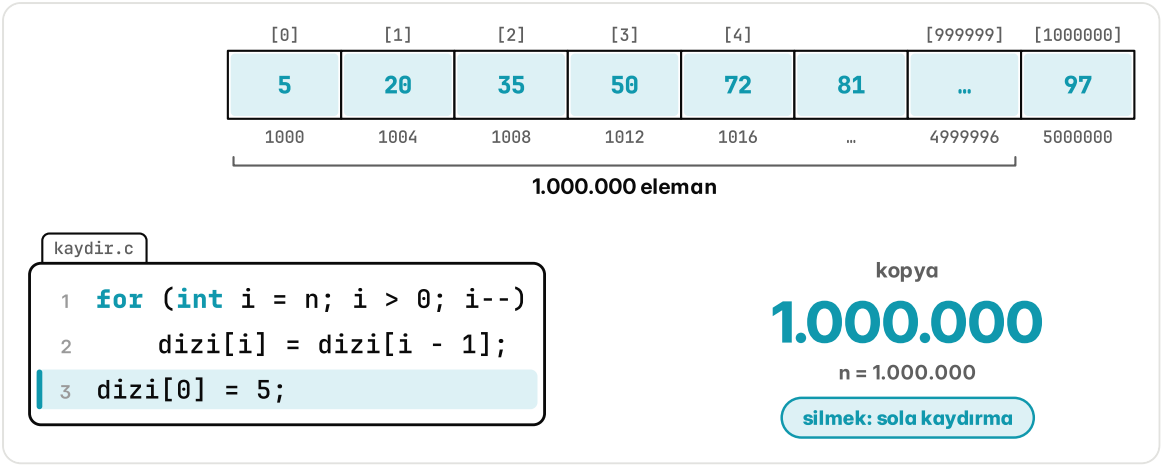
```
int dizi[10] = {20, 35, 50};
int n = 3;
int kopya = 0;

for (int i = n; i > 0; i--) {
    dizi[i] = dizi[i - 1];
    kopya++;
}
dizi[0] = 5;
n++;
```

Program sonunda diziyi ve kopya sayısını yazar.

Çıktı:

```
5 20 35 50
3 kopya
```

Döngü sondan başlar: önce dizi[3] = dizi[2] ile 50 bir sağa geçer, sonra 35, en son 20. Baştan başlasaydı ilk adımda dizi[1] = dizi[0], 35'in üstüne 20 yazardı; sonraki adımlar da bu 20'yi sağa taşırdı ve dizi 5 20 20 20 olurdu. n elemanlı dizinin başına eklemek n kopya ister: 3 elemanda 3, bir milyon elemanda bir milyon. Ortadan bir eleman silmek de aynı kaydırmayı ters yönde, sola doğru ister.



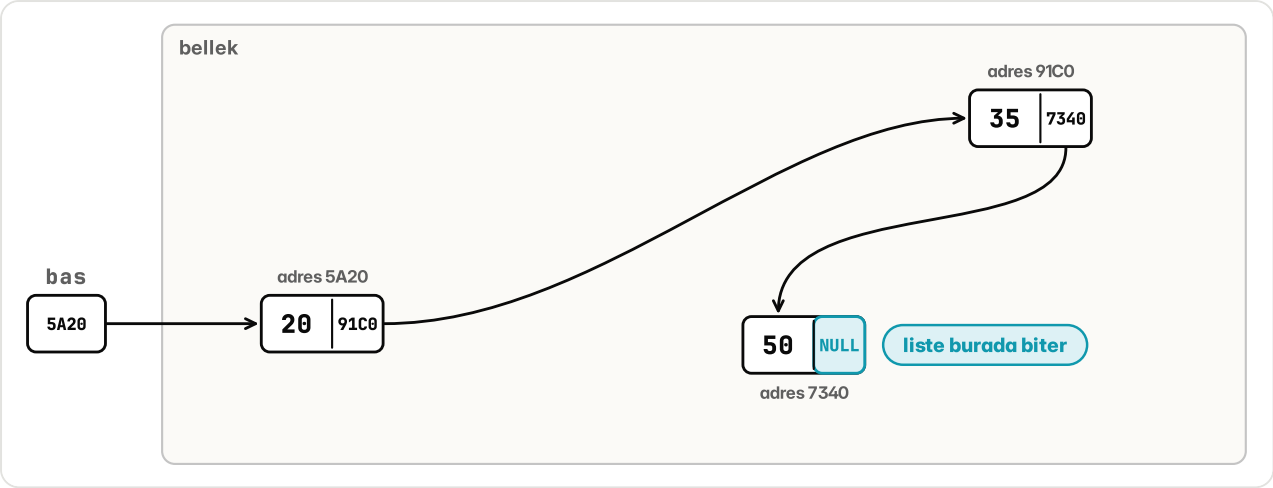
Şekil 10.1 Dizinin elemanları sondan başlayarak birer hücre sağa kopyalanıyor ve 5 ilk hücreye yazılıyor; 1.000.000 elemanda 1.000.000 kopya gerekiyor.

Bağlı liste (linked list) elemanları yan yana dizmez. Her eleman **düğüm** denen küçük bir pakette durur; pakette iki şey vardır: veri ve sıradaki düğümün bellekteki adresi. C'de düğüm bir yapıdır (struct): içinde bir tam sayı ve aynı tipten bir düğümü gösteren işaretçi bulunur.

```
typedef struct dugum {
    int veri;
    struct dugum *sonraki;
} Dugum;
```

Yapı kendini içine alamaz: alan struct dugum sonraki diye yazılsaydı her düğümün içinde bir düğüm, onun içinde bir düğüm daha olurdu ve boyu sonsuz olurdu. Kendi tipinden bir düğümün adresini ise taşıyabilir, çünkü işaretçi yalnız bir adrestir ve 64 bitlik bir bilgisayarda 8 bayt tutar. typedef bir tipe kısa ad verir: bundan sonra struct dugum yerine Dugum yazılır.

Düğümler bellekte herhangi bir yerde olabilir; her birine malloc, nerede boş yer bulursa orayı verir. Örneğin 20'nin düğümü 5A20 adresinde, 35'inki 91C0'da, 50'ninki 7340'ta olabilir. Sırayı adresler kurar: 20'nin düğümü 35'inkinin adresini tutar, 35'inki de 50'ninkini. İlk düğümün adresini bas adlı bir işaretçi saklar. bas'ı kaybeden bütün listeyi kaybeder, çünkü öteki düğümlerin adresleri yalnız birbirlerinin içinde yazılıdır. Son düğümde adresin yerinde NULL durur; NULL hiçbir yeri göstermeyen adrestir ve liste orada biter.



Şekil 10.2 Bellekte dağınık duran 20, 35 ve 50 düğümleri sıradakinin adresini tutuyor; bas ilk düğümü gösteriyor, son düğümde NULL var.

Başa 5'i eklemek için önce malloc ile yeni bir düğüm alınır ve verisine 5 yazılır. Yeni düğüm, şimdiki başın adresini sıradaki olarak alır; fonksiyon yeni düğümün adresini döndürür ve bas artık onu gösterir. → işareti, işaretçinin gösterdiği yapının alanına ulaşır: yeni→veri, yeni'nin gösterdiği düğümün veri alanıdır.

```
Dugum *basa_ekle(Dugum *bas, int x) {
    Dugum *yeni = malloc(sizeof(Dugum));
    yeni->veri = x;
    yeni->sonraki = bas;
    return yeni;
}
```

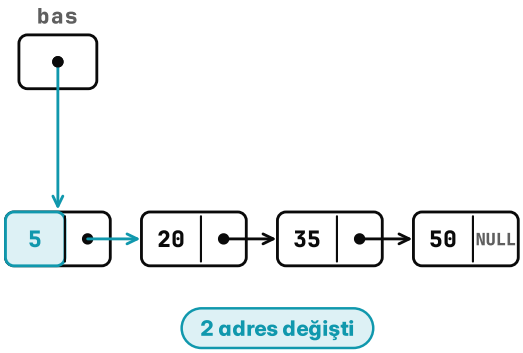
Boş liste bas = NULL ile başlar. Aynı fonksiyonla önce 50, 35 ve 20 başa eklenir, sonra 5:

```
Dugum *bas = NULL;
bas = basa_ekle(bas, 50);
bas = basa_ekle(bas, 35);
bas = basa_ekle(bas, 20);

bas = basa_ekle(bas, 5);
```

Liste baştan yazdırılınca 5 20 35 50 çıkar. Liste bir milyon elemanlı da olsa başa eklemede yalnız iki adres değişir: yeni düğümün sonraki alanı ve bas. Hiçbir eleman yerinden kıpırdamaz.

liste.c

```c
Dugum *basa_ekle(Dugum *bas, int x) {
    Dugum *yeni = malloc(sizeof(Dugum));
    yeni->veri = x;
    yeni->sonraki = bas;
    return yeni;
}

bas = basa_ekle(bas, 5);
```



Şekil 10.3 5'in düğümü listenin başına ekleniyor; yalnız yeni düğümün sonraki alanı ve bas değişiyor, öteki düğümler yerinde kalıyor.

Durmadan 42

videoda 48:35

Bağlı listede her düğüm, verisini ve sıradaki düğümün adresini taşır. 5, 20, 35, 50, 65 listesinde 35 ile 50 arasına 42'yi ekleyen iki satırın yerini değiştirdik; p, 35'in düğümünü gösteriyor:

```
p->sonraki = yeni;
yeni->sonraki = p->sonraki;
```

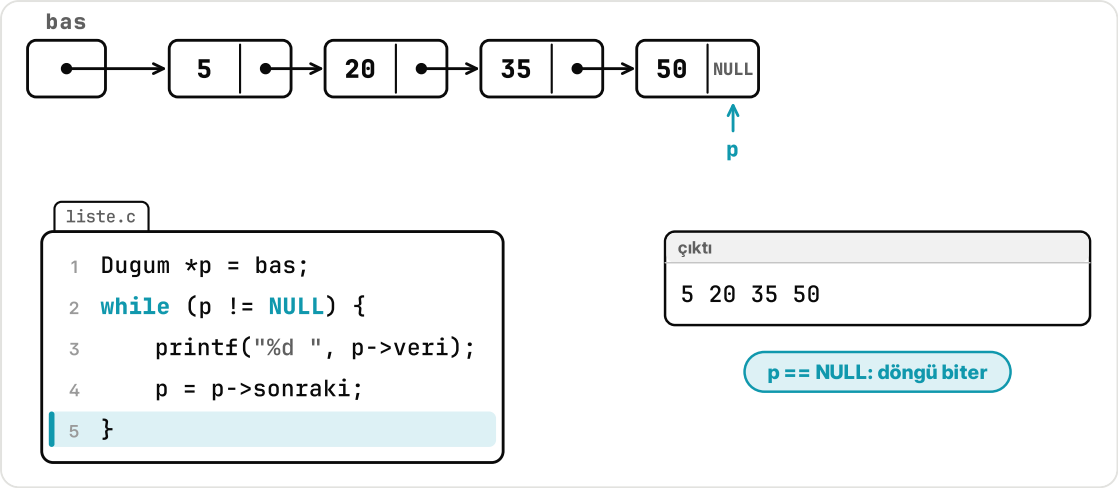
Program artık durmadan 42 yazıyor. Örnek program dolaşmayı 12 düğümde kesiyor; kesmese sonsuza kadar sürerdi.

Çıktı:

```
5 20 35 42 42 42 42 42 42 42 42 42 ...
```

Nedenini görmek için önce listenin nasıl dolaşıldığına bakalım. Listede indeks yoktur; bir düğüme ancak baştan başlayıp adres adres yürüyerek varılır. p işaretçisi başın adresini alır. p NULL olmadığı sürece döngü, düğümün verisini yazar ve p'yi sıradaki düğümün adresine geçirir. Son düğümden sonra p NULL olur ve döngü biter.

```
Dugum *p = bas;
while (p != NULL) {
    printf("%d ", p->veri);
    p = p->sonraki;
}
```



Şekil 10.4 p işaretçisi 5'ten 50'ye düğüm düğüm yürüyor ve her durakta veriyi yazıyor; p NULL olunca döngü bitiyor.

Arama da aynı yürüyüştür: düğümün verisi aranan sayıya eşit olunca durulur. Fonksiyon bulduğu düğümün adresini döndürür; bulamazsa liste biter ve NULL döner.

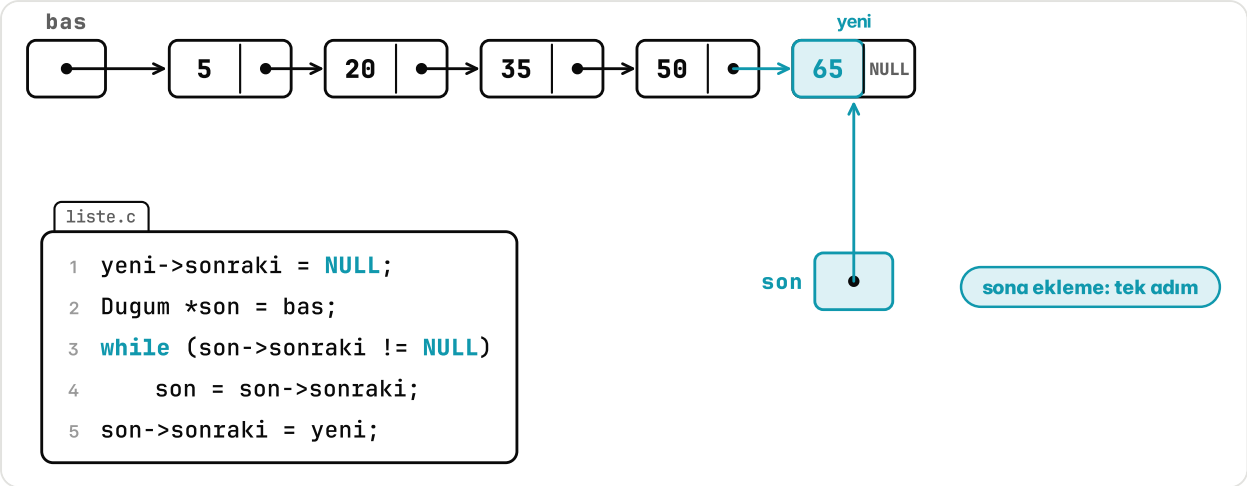
```
Dugum *bul(Dugum *p, int x) {
    while (p != NULL && p->veri != x)
        p = p->sonraki;
    return p;
}
```

Koşulun sırası önemlidir: && sol tarafı yanlış çıkınca sağ tarafa hiç bakmaz. p NULL olduğunda p->veri okunmaz; NULL bir düğüm değildir, verisi okunamaz.

Sona 65'i eklemek için önce son düğüm bulunur: sıradaki adresi NULL olan düğüm. Bu düğümün sonraki alanına yeni düğümün adresi, yeni düğümünkine de NULL yazılır:

```
Dugum *yeni = malloc(sizeof(Dugum));
yeni->veri = 65;
yeni->sonraki = NULL;
Dugum *son = bas;
while (son->sonraki != NULL)
    son = son->sonraki;
son->sonraki = yeni;
```

Bir milyon elemanlı listede son düğümü bulan yürüyüş bir milyon adım sürer. Bu yüzden çoğu program son düğümün adresini de ayrı bir işaretçide tutar ve sona eklemeyi tek adımda yapar.



Şekil 10.5 son işaretçisi son düğüme yürüyor ve 65 onun arkasına bağlanıyor; son düğümün adresi ayrıca tutulursa sona ekleme tek adımda yapılıyor.

42'yi 35'ten sonraya eklerken önce yeni düğüm 50'nin adresini alır, sonra 35 yeni düğümü gösterir:

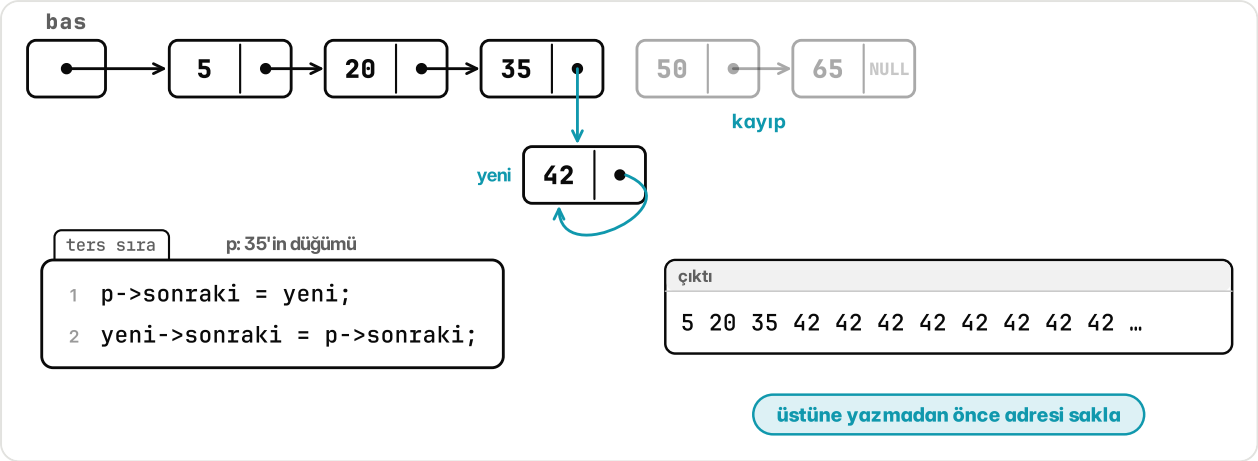
```
Dugum *p = bul(bas, 35);
yeni = malloc(sizeof(Dugum));
yeni->veri = 42;
yeni->sonraki = p->sonraki;
p->sonraki = yeni;
```

Dolaşma, 50 ile 42'nin aranması, sona ve araya ekleme aynı programda art arda çalıştırılır.

Çıktı:

```
5 20 35 50
50 bulundu
42 yok
5 20 35 50 65
5 20 35 42 50 65
```

Satırlar yer değiştirince 35 önce yeni düğümü gösterir ve 50'nin adresi hiçbir yerde kalmaz. İkinci satır, 35'in artık tuttuğu adresi yeni düğüme verir: bu, yeni düğümün kendi adresidir. 42 kendini gösterir; dolaşma ondan çıkamaz, 50 ve 65'e bir daha ulaşılamaz. Kural basittir: bir adresin üstüne yazmadan önce, onun gösterdiği düğümü başka bir yerde tut.



Şekil 10.6 Satırlar ters yazılınca 35 yeni düğümü, 42 de kendini gösteriyor; 50 ve 65'e giden bağ kayboluyor.

35 hâlâ orada

videoda 50:01

Bağlı listede düğümler birbirine adresle bağlıdır: 20, 35, 50. 35'in düğümü free ile belleğe geri verildi; free, malloc ile alınmış belleği geri veren fonksiyondur:

```
Dugum *p = bas->sonraki;   /* 35 */
free(p);
```

Ardından liste baştan yazdırıldı. Aşağıdaki çıktı, bu örneğin çalıştırıldığı bilgisayarda alındı.

Çıktı:

```
20 35 50
```

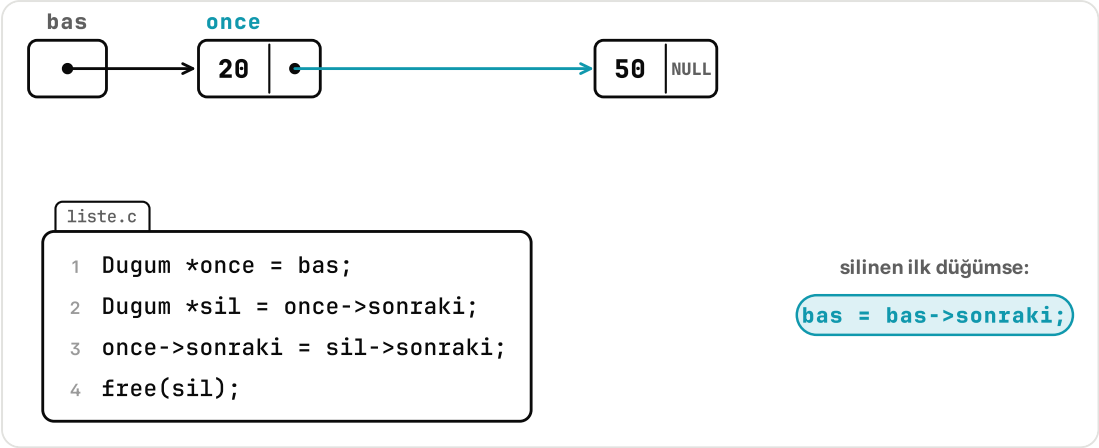
35 hâlâ görünüyor; başka bir bilgisayarda ise aynı program çökebilir. Neden?

free yalnız belleği geri verir; 20'nin düğümü 35'in adresini hâlâ tutar. Döngü o adrese gider ve artık programın olmayan belleği okur. Bu tanımsız davranıştır: C, böyle bir programın ne yapacağını belirlemez. Eski değer orada durabilir; bu örnekte geri verilen bellek henüz değişmediği için 35 göründü. Bellek başka bir işe de verilmiş olabilir. Kimi sistemlerde bellek yöneticisi geri verilen bloğun ilk baytlarına kendi bilgisini yazar; sonraki alanı anlamsız bir adres olur ve program büyük olasılıkla çöker.

Doğru sıra şudur: önce 20, 50'nin adresini alır ve 35 listeden çıkar; sonra free ile düğüm geri verilir.

```
Dugum *once = bas;
Dugum *sil = once->sonraki;
once->sonraki = sil->sonraki;
free(sil);
```

Liste 20 35 50'den 20 50'ye iner. Silinen ilk düğümse önceki düğüm yoktur: ilk düğümün adresi bir değişkende saklanır, bas ikinci düğüme geçer (bas = bas→sonraki), sonra eski ilk düğüm geri verilir.



Şekil 10.7 Önce 20'nin düğümü 50'yi gösteriyor, sonra 35'in düğümü free ile geri veriliyor; silinen ilk düğümse bas ikinci düğüme geçiyor.

Liste işi bitince bütün düğümler tek tek geri verilir. free'den sonra düğüme dokunulmaz; bu yüzden sıradaki düğümün adresi free'den önce bir değişkene alınır:

```
while (bas != NULL) {
    Dugum *sonra = bas->sonraki;
    free(bas);
    bas = sonra;
}
```

Dizi mi, bağlı liste mi? Hangisinin seçileceği yapılacak işe bağlıdır:

- i. elemana gitmek: dizide 1 adım, çünkü adresi hesaplanır; listede baştan düğüm düğüm i adım.
- Başa ya da yeri bilinen bir noktaya eklemek: dizide eleman sayısı kadar kaydırma, listede birkaç atama.
- Bir int için yer: dizide 4 bayt; listede 16 bayt. 64 bitlik bir bilgisayarda düğümde 4 baytlık sayı, 8 baytlık adres ve adresin 8'in katı bir yerden başlaması için araya konan 4 baytlık dolgu vardır.
- Bellekte dizi bitişik, düğümler dağınıktır. İşlemci belleği küçük bloklar hâlinde okur; dizinin bir bloğunda birkaç eleman birden gelir, dağınık düğümlerin her biri ayrı bir okumadır.

Bu yüzden çoğu zaman dizi seçilir; liste, araya sık ekleme ve silme yapılan, elemanların sırayla gezildiği yerlerde öne çıkar.



Şekil 10.8 Dizi ile bağlı listenin karşılaştırması: elemana gitmek, başa ya da araya eklemek, bir int için yer ve bellekteki düzen.

Çift yönlü listede her düğüm önceki düğümün adresini de tutar:

```
typedef struct dugum {
    int veri;
    struct dugum *onceki, *sonraki;
} Dugum;
```

Liste iki yönde gezilir; bir düğümü silerken önceki düğümü bulmak için baştan yürümek gerekmez. **Dairesel liste**de ise son düğüm, NULL yerine ilk düğümü gösterir; gezinti NULL'a değil, yeniden başa varınca biter. Aşağıdaki program 20, 35, 50 düğümlerini hem çift yönlü hem dairesel bağlar ve listeyi önce ileri, sonra geri gezer:

```
Dugum *p = bas;          /* ileri: başa varınca dur */
do {
    printf("%d ", p->veri);
    p = p->sonraki;
} while (p != bas);
printf("| ");
p = bas->onceki;         /* geri: sondan başa */
do {
    printf("%d ", p->veri);
    p = p->onceki;
} while (p != bas->onceki);
printf("\nsizeof(Dugum) = %zu\n", sizeof(Dugum));
```

Çıktı:

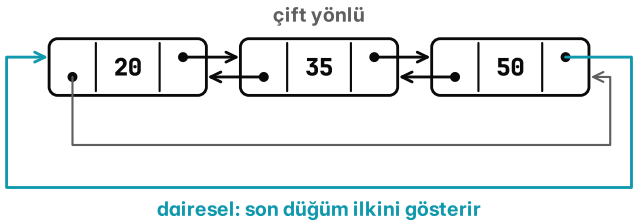
```
20 35 50 | 50 35 20
sizeof(Dugum) = 24
```

do-while gövdeyi önce bir kez çalıştırır, koşula sonra bakar; bu yüzden p başta bas'a eşit olduğu hâlde ilk düğüm yazılır ve döngü bas'a yeniden varınca durur. Dairesel listede ilk düğümün öncekisi son düğümdür; geri gezinti bas→onceki'den, yani 50'den başlar. Bir adres daha eklendiği

için düğüm 24 bayta çıkar. Linux çekirdeği, bütün süreçleri, yani çalışan programları tam da böyle bir listede tutar: çift yönlü ve dairesel.

liste.c

```c
typedef struct dugum {
    int veri;
    struct dugum *onceki, *sonraki;
} Dugum;
```



Linux çekirdeği: süreç listesi

Şekil 10.9 Üç bölmeli düğümlerle çift yönlü liste ve son düğümün ilk düğümü gösterdiği dairesel liste.

Özet

- Dizinin başına ya da ortasına eklemek için elemanlar kaydırılır: n elemanlı dizinin başına eklemek n kopya ister.
- Bağlı listede her düğüm veriyi ve sıradaki düğümün adresini tutar; ilk düğümün adresi bas'tadır, son düğümde NULL vardır.
- Başa ya da yeri bilinen bir düğümden sonraya eklemek birkaç adres atamasıdır; bir düğüme varmak ise baştan yürümeyi ister.
- Bir adresin üstüne yazmadan önce gösterdiği düğüm başka bir yerde tutulur; yoksa listenin geri kalanı kaybolur.
- Silinecek düğüm önce listeden çıkarılır, sonra free ile geri verilir; free'den sonra düğüme dokunulmaz.
- Çoğu zaman dizi seçilir; çift yönlü listede düğüm önceki düğümü de, dairesel listede son düğüm ilk düğümü gösterir.

Sorular

1 **Bağlı listede bir düğümde ne tutulur?**

- **A** Veri ve düğümün listedeki sıra numarası
- **B** Veri ve sıradaki düğümün adresi
- **C** Yalnız veri; sırayı malloc tutar
- **D** Veri ve listenin ilk düğümünün adresi

2 **1.000 elemanlı bir dizinin başına bir sayı eklemek için kaç eleman kopyalanır?**

Cevap:

3 **Hangi işte bağlı liste diziden üstündür?**

A 500. elemanın değerini okumak

B Bir int'i olabildiğince az bellekle saklamak

C Elemanları işlemcinin hızlı okuyacağı biçimde bitişik tutmak

D Yeri bilinen bir düğümden sonra yeni eleman eklemek

4 **Liste 10, 20, 30 düğümlerinden oluşuyor ve bas ilk düğümü gösteriyor. Bu kod ne yazar?**

```
Dugum *p = bas;
int t = 0;
while (p != NULL) {
    t += p->veri;
    p = p->sonraki;
}
printf("%d\n", t);
```

Cevap:

5 **bul fonksiyonundaki koşul while (p→veri != x && p != NULL) diye ters sırayla yazılsaydı ne olurdu?**

A Hiçbir şey değişmezdi; && iki tarafa da aynı anda bakar.

B Fonksiyon aranan sayı listede olsa bile onu bulamazdı.

C Aranan sayı listede yoksa, liste bitince NULL'ın verisi okunmaya çalışılırdı.

D Derleyici bu koşulu kabul etmez, hata verirdi.

6 **p, 35'in düğümünü gösteriyor ve 42'nin düğümü (yeni) 35 ile 50 arasına eklenecek. Satırlar bu sırayla yazılırsa ne olur?**

```
p->sonraki = yeni;
yeni->sonraki = p->sonraki;
```

A 42 kendini gösterir; dolaşma ondan çıkamaz, 50 ve sonrasına ulaşılamaz.

B 42, 35 ile 50 arasına doğru biçimde eklenir.

C 42 listenin sonuna eklenir.

D 35 listeden çıkar, yerine 42 geçer.

7 **Listeyi boşaltan bu döngüdeki hata nedir?**

```
while (bas != NULL) {
    free(bas);
    bas = bas->sonraki;
}
```

A Döngü son düğümü geri vermeden durur.

B free'den sonra bas→sonraki okunuyor; geri verilmiş belleğe dokunuluyor.

C free yalnız ilk düğümü geri verir, ötekileri bırakır.

D bas NULL olunca free(NULL) çağrılır ve program çöker.

8 **64 bitlik bir bilgisayarda bir int (veri) ve iki işaretçi (onceki, sonraki) taşıyan çift yönlü liste düğümü kaç bayt tutar?**

Cevap: _____

PROGRAMLAMAYA GİRİŞ · 1. DÖNEM

11 Dosya işlemleri

Program kapanınca bellekteki her şey kaybolur; kalıcı bilgi dosyada tutulur. Bu bölümde C'de dosya açmayı, dosyaya yazmayı ve dosyadan okumayı, yazılanların neden hemen dosyaya gitmediğini, metin ve ikili dosya farkını ve programa dosya adını komut satırından vermeyi öğreneceksin.

Kayıp satırlar

videoda 51:41

Programın nerede çöktüğünü bulmak için her adımı sırayla bir log dosyasına yazıyoruz; log, bir programın yaptıklarını sırayla kaydettiği dosyadır. Aşağıdaki programda üç adım tamamlanıyor ve üçü de log.txt'ye yazılıyor. Sonra program bilerek çökertiliyor: NULL'ı tutan p işaretçisinin gösterdiği yere 7 yazılıyor. Bu tanımsız davranıştır; çoğu bilgisayarda program çöker.

```
#include <stdio.h>

int main(void) {
    FILE *f = fopen("log.txt", "a");
    for (int adim = 1; adim <= 3; adim++)
        fprintf(f, "%d. adım tamam\n", adim);
    int *p = NULL;
    *p = 7;                 // program çöker
    fclose(f);
    return 0;
}
```

Program çöktükten sonra (Linux'ta ekranda Segmentation fault yazar) log.txt oluşmuştur, ama bomboştur: 0 bayt. Yazılan üç satır nereye gitti?

Programın değişkenleri RAM'de durur. Program kapanınca işletim sistemi o belleği geri alır ve başka programlara verir; içindekiler kaybolur. Bir bilginin program kapandıktan sonra da kalması için diskteki bir dosyaya yazılması gerekir.

C'de dosyayla her iş, dosyayı açan fopen fonksiyonuyla başlar. fopen dosyanın adını ve kipini alır, geriye bir FILE işaretçisi döndürür; FILE, açık dosyanın bilgilerini tutan yapıdır. O dosyayla bundan sonraki her iş bu işaretçi üzerinden yapılır. Kip, İngilizce bir sözcüğün baş harfidir:

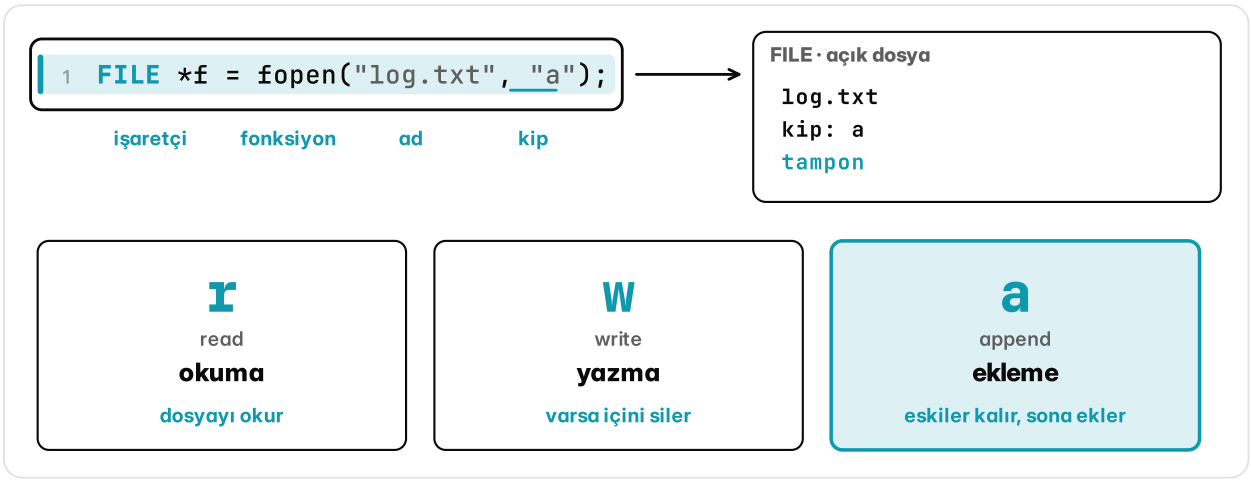
- "r" (read, okumak): dosyayı okur.
- "w" (write, yazmak): dosyaya yazar; dosya varsa içini baştan siler.
- "a" (append, eklemek): eskileri korur, yeni yazılanları sona ekler.

Log dosyaları bu yüzden çoğunlukla "a" kipiyle açılır. Aşağıdaki program not.txt'yi sırayla "w", "a" ve yeniden "w" kipiyle açıp birer satır yazıyor; fclose dosyayı kapatıyor, goster fonksiyonu da kipi ve dosyanın içini ekrana yazıyor:

```
FILE *f = fopen("not.txt", "w");
fputs("ilk\n", f);
fclose(f);
goster("w");
f = fopen("not.txt", "a");
fputs("ikinci\n", f);
fclose(f);
goster("a");
f = fopen("not.txt", "w");
fputs("yeni\n", f);
fclose(f);
goster("w");
```

Çıktı:

```
w: ilk
a: ilk
 ikinci
w: yeni
```



Şekil 11.1 fopen satırının parçaları (fonksiyon, dosya adı, kip, dönen işaretçi) ve r, w, a kiplerinin kartları; a kipi eskileri koruyup sona ekliyor.

Dosyaya yazmak, ekrana yazmaktan farksızdır. Ekrana yazan fonksiyonların başına f harfi gelince dosyaya yazan sürümleri olur: printf'in karşılığı fprintf, puts'un karşılığı fputs'tur. Bu sürümler dosyayı da parametre olarak alır. Bir fark var: puts satır sonunu kendisi ekler, fputs eklemez.

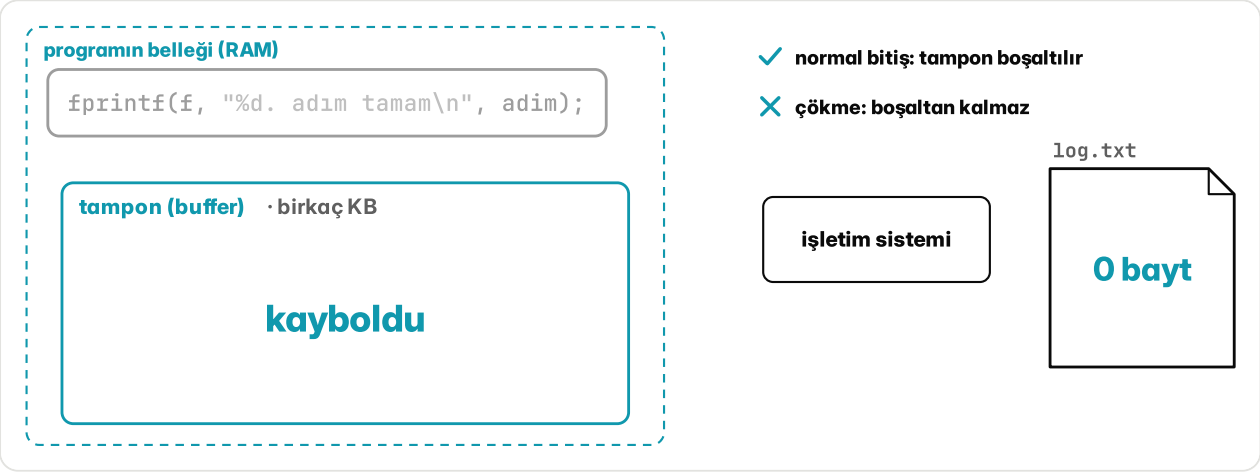
```c
#include <stdio.h>

int main(void) {
    FILE *f = fopen("log.txt", "w");
    printf("%d. adım tamam\n", 1);      // ekrana
    fprintf(f, "%d. adım tamam\n", 1);  // dosyaya
    puts("bitti");                      // ekrana, satır sonunu kendisi ekler
    fputs("bitti\n", f);                // dosyaya, satır sonu eklemez
    fclose(f);
    return 0;
}
```

Çıktı:

```
1. adım tamam
bitti
```

Aynı iki satır log.txt'ye de yazılır.

Oysa yazılan, dosyaya hemen gitmez. Her satır için ayrı ayrı işletim sistemine başvurmak yavaştır. Bu nedenle C, yazılanları önce programın belleğindeki bir **tampon** (buffer) alana biriktirir; bu alan genellikle birkaç kilobayttır. Tampon dolunca ya da dosya kapanınca içindekiler tek seferde dosyaya gönderilir. Log programının üç satırı da tampondaydı; program çökünce tampon da belleğin geri kalanıyla birlikte gitti. Program normal biterse C açık dosyaların tamponlarını kendisi boşaltır; fclose unutulsa bile satırlar dosyaya geçer. Çöken programda bunu yapacak kimse kalmaz.

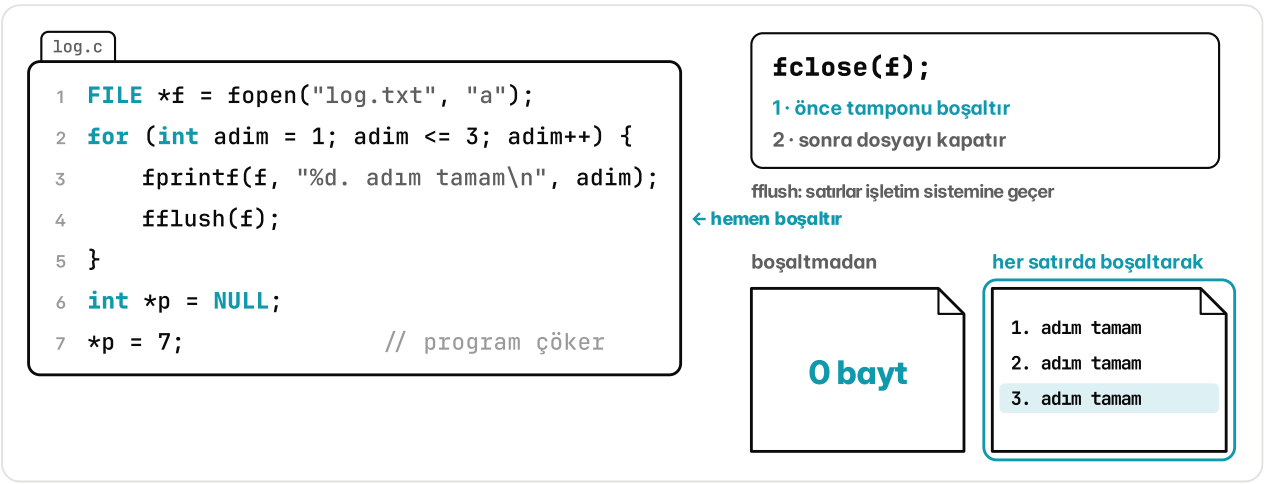


Şekil 11.2 fprintf'in yazdığı satırlar programın belleğindeki tamponda birikiyor; tampon dolunca ya da dosya kapanınca dosyaya gidiyor, program çökünce kayboluyor.

Dosyayla iş bitince dosya fclose ile kapatılır; fclose önce tamponu boşaltır, sonra dosyayı kapatır. Log gibi her satırın önemli olduğu dosyalarda tampon her satırdan sonra fflush ile elle boşaltılır:

```c
#include <stdio.h>

int main(void) {
    FILE *f = fopen("log.txt", "a");
    for (int adim = 1; adim <= 3; adim++) {
        fprintf(f, "%d. adım tamam\n", adim);
        fflush(f);          // tampon her satırdan sonra boşaltılır
    }
    int *p = NULL;
    *p = 7;                 // program çöker
    fclose(f);
    return 0;
}
```

Satırlar böylece işletim sistemine teslim edilir; program çökse de dosyaya yazılır. Boşaltmadan çöken programda log.txt 0 bayt kalır, her satırdan sonra boşaltanda 3 satır vardır. Çöken bir programda en değerli bilgi, çoğu zaman tam da kaybolan o son satırlardır.



Şekil 11.3 fflush'lı log programı ve fclose'un iki işi; boşaltmadan çöken programın log.txt'si 0 bayt, her satırdan sonra boşaltanınki 3 satır.

Olmayan dosya

videoda 53:37

Programımız satış kayıtlarını satislar.txt dosyasından okuyacak, ama dosyanın adını satis.txt diye yanlış yazmışız. fgets, dosyadan bir satır okuyan fonksiyondur.

```c
#include <stdio.h>

int main(void) {
    FILE *f = fopen("satis.txt", "r");   // dosyanın adı satislar.txt
    char satir[64];
    fgets(satir, sizeof satir, f);       // f NULL: program çöker
    printf("%s", satir);
    fclose(f);
    return 0;
}
```

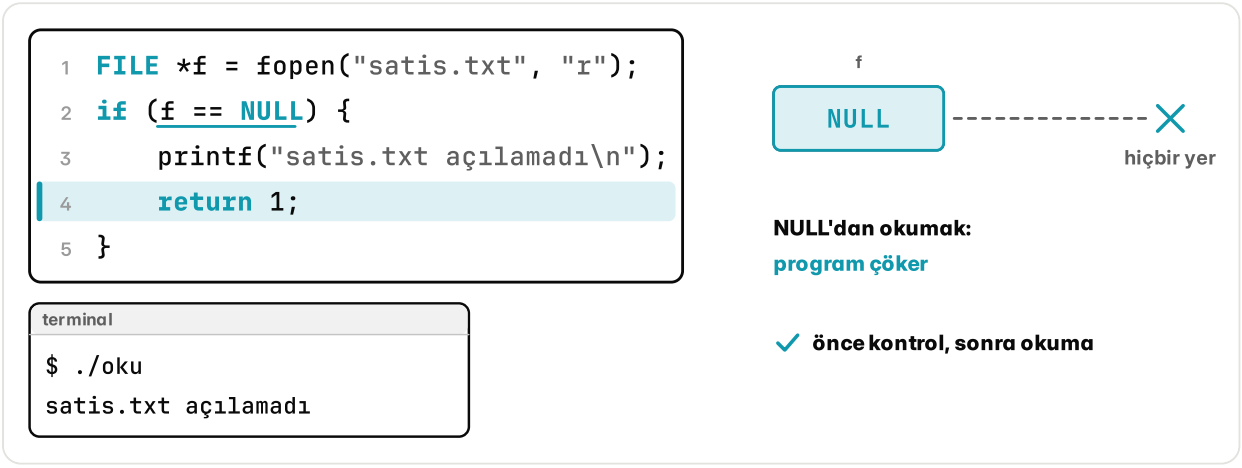
Linux'ta program, sorunun ne olduğunu söyleyen bir mesaj yazmadan, ilk satırı okumaya kalktığı anda çöker. Neden?

fopen, açamadığı dosya için hata vermez; geriye NULL döndürür. NULL hiçbir yeri göstermeyen adrestir; onu dosya diye kullanıp okumaya kalkmak tanımsız davranıştır: Linux'ta program çöker, kimi sistemlerde hiçbir şey yazmadan biter. Bu yüzden her fopen'dan sonra dönen işaretçiye bakılır; NULL ise anlaşılır bir mesaj yazılır ve program durur:

```
FILE *f = fopen("satis.txt", "r");
if (f == NULL) {
    printf("satis.txt açılamadı\n");
    return 1;
}
```

Çıktı:

```
satis.txt açılamadı
```



Şekil 11.4 fopen'dan sonra eklenen NULL kontrolü; f NULL olduğu için program "satis.txt açılamadı" yazıp duruyor.

Dosyadan okumak klavyeden okumaya benzer; fonksiyonlara klavye yerine dosya verilir. satislar.txt'nin içi:

```
kalem 120
silgi 85
defter 240
```

fgets(satir, sizeof satir, f) her çağrıda bir sonraki satırı satir dizisine okur ve dizinin boyunu aşmaz. Adı ve sayıyı biçime göre ayırmayı ise scanf'in dosya sürümü fscanf yapar; okuyabildiği değer sayısını döndürür. %15s, 16 baytlık ad dizisine en çok 15 karakter okur; son bayt dizgeyi bitiren '\0' içindir.

```
char satir[64];
while (fgets(satir, sizeof satir, f) != NULL)
    printf("%s", satir);
printf("son dönüş NULL mu: %s\n", fgets(satir, sizeof satir, f) == NULL ? "evet" :
"hayır");
```

Çıktı:

```
kalem 120
silgi 85
defter 240
son dönüş NULL mu: evet
```

Aynı dosyadaki adetleri toplayan döngü her turda 2 değer okur:

```
char ad[16];
int adet, toplam = 0, donus;
while ((donus = fscanf(f, "%15s %d", ad, &adet)) == 2)
    toplam += adet;
printf("toplam: %d\n", toplam);
printf("son dönüş: %d (EOF = %d)\n", donus, EOF);
```

Çıktı:

```
toplam: 445
son dönüş: -1 (EOF = -1)
```

Dosyanın sonunda saklı bir işaret yoktur; işletim sistemi dosyanın kaç bayt olduğunu bilir. Okunacak bir şey kalmayınca okuma fonksiyonları bunu dönüş değeriyle bildirir: fgets NULL, fscanf EOF döndürür. EOF (end of file, dosya sonu) stdio.h'de tanımlı eksi bir sayıdır, yaygın sistemlerde −1'dir. Döngü orada durur.

Metin dosyasında sayılar rakam rakam yazılır: 1000000, 7 karakter, yani 7 bayttır. İkili dosyaya ise bellekteki baytlar olduğu gibi kopyalanır: int türündeki her sayı, değeri ne olursa olsun 4 bayttır. fwrite(adres, bir öğenin boyu, öğe sayısı, dosya) bellekten dosyaya bayt kopyalar; fread aynı biçimde geri okur.

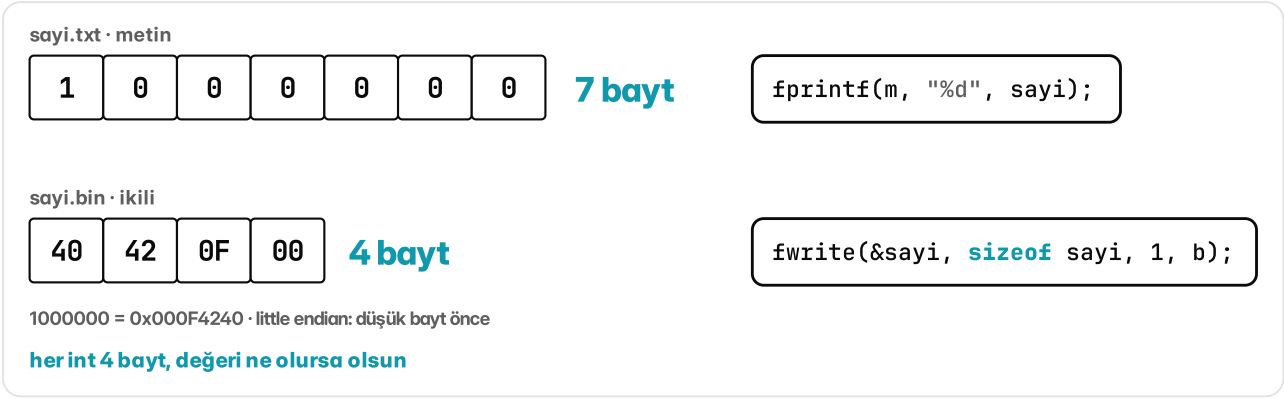
```
int sayi = 1000000;
FILE *m = fopen("sayi.txt", "w");
fprintf(m, "%d", sayi);                 // rakam rakam: 7 karakter
fclose(m);
FILE *b = fopen("sayi.bin", "wb");
fwrite(&sayi, sizeof sayi, 1, b);       // bellekteki 4 bayt
fclose(b);
```

Programın sonu iki dosyanın boyunu yazar.

Çıktı:

```
metin: 7 bayt
ikili: 4 bayt
```

1000000 = 0x000F4240; sayi.bin'de baytlar küçük bayt önce sıralanır: 40 42 0F 00.



Şekil 11.5 1000000 metin dosyasında 7 rakam, 7 bayt; ikili dosyada bellekteki 4 bayt (40 42 0F 00).

Bir yapının tamamı da tek seferde yazılıp aynı biçimde geri okunur. struct urun, 12 baytlık ad ve 4 baytlık adet alanlarından oluşur; 16 bayt tutar.

```
struct urun yaz[3] = {{"kalem", 120}, {"silgi", 85}, {"defter", 240}};
FILE *f = fopen("urunler.bin", "wb");
fwrite(yaz, sizeof(struct urun), 3, f);     // 3 yapı tek seferde
fclose(f);

struct urun oku[3];
f = fopen("urunler.bin", "rb");
size_t n = fread(oku, sizeof(struct urun), 3, f);
fclose(f);
```

Çıktı:

```
kalem 120
silgi 85
defter 240
dosya: 48 bayt
```

İkili dosya açılırken kipe b harfi eklenir ("wb", "rb"), çünkü Windows metin kipinde satır sonu baytını (0A) 2 bayta (0D 0A) çevirir ve ikili veriyi bozar. 10 sayısı bellekte 0A 00 00 00'dır; ilk baytı satır sonu baytıyla aynıdır. Aynı sayıyı "w" ve "wb" kipiyle yazıp geri okuyan bir program Windows'ta şunu yazar.

Çıktı:

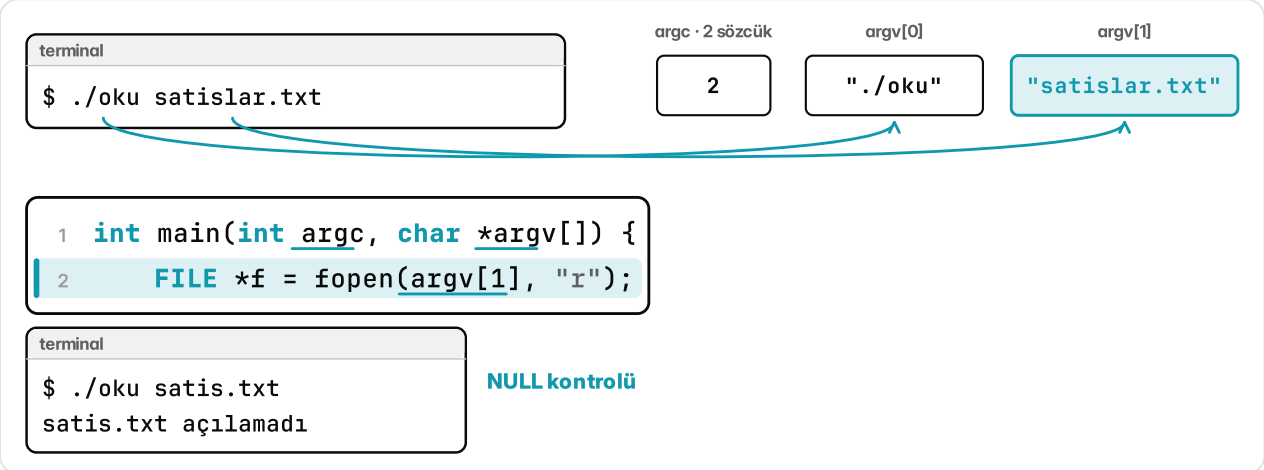
```
kip w : 5 bayt, geri okunan 2573
kip wb: 4 bayt, geri okunan 10
```

"w" kipinde dosyaya 0D 0A 00 00 00 yazılmış, geri okunan ilk 4 bayt 0D 0A 00 00 de 2573 olmuştur. Linux'ta metin ve ikili kip aynıdır. Yalnız dosyaya yapının bellekteki düzeni de geçer (dolgu baytları, bayt sırası); başka bir derleyici ya da işlemci aynı baytları farklı okuyabilir.

Dosyanın adını koda yazmak yerine programı çalıştırırken verebiliriz. Komut satırında yazılan sözcükler main'e iki parametreyle gelir: argc sözcüklerin sayısı, argv sözcüklerin kendisidir.

```
int main(int argc, char *argv[]) {
    printf("argc = %d\n", argc);
    for (int i = 0; i < argc; i++)
        printf("argv[%d] = %s\n", i, argv[i]);
    if (argc < 2) {
        printf("kullanım: oku <dosya>\n");
        return 1;
    }
    FILE *f = fopen(argv[1], "r");
    if (f == NULL) {
        printf("%s açılamadı\n", argv[1]);
        return 1;
    }
```

Program ./oku satislar.txt diye çalıştırılınca argc 2 olur: argv[0] programın adı, argv[1] dosyanın adıdır. Böylece aynı program her çalıştırmada başka bir dosyayı okur; ad yanlışsa NULL kontrolü programı durdurur.



Şekil 11.6 ./oku satislar.txt komutunda argc 2, argv[0] "./oku", argv[1] "satislar.txt"; yanlış adda NULL kontrolü programı durduruyor.

Özet

- Program kapanınca bellekteki her şey kaybolur; kalıcı bilgi dosyaya yazılır.
- fopen dosyayı açar ve FILE işaretçisi döndürür; "w" dosyanın içini siler, "a" sona ekler, "r" okur. Açamazsa NULL döner, bu yüzden her açmadan sonra NULL kontrolü yapılır.
- Yazılanlar önce tamponda birikir; fclose ve fflush tamponu boşaltır, çöken programın tamponu ise kaybolur.
- fgets dosya sonunda NULL, fscanf EOF döndürür; dosyanın içinde dosya sonu işareti yoktur.
- Metin dosyasında sayılar rakam rakam, ikili dosyada bellekteki baytlarla yazılır; ikili dosya "wb" ve "rb" kipleriyle açılır.
- Komut satırındaki sözcükler main'e argc ve argv ile gelir; argv[1] programa verilen ilk sözcüktür.

Sorular

1 **Var olan bir log dosyasının eski satırlarını silmeden sonuna yeni satır eklemek için fopen hangi kiple çağrılır?**

A "r"
B "w"
C "a"
D "rb"

2 **fopen istenen dosyayı açamazsa ne olur?**

A NULL döndürür; program bunu kontrol etmeden okumaya kalkarsa tanımsız davranış olur.
B Ekrana bir hata mesajı yazar ve programı kendisi durdurur.
C Dosyayı boş olarak oluşturur ve okuma hiçbir şey okumadan biter.
D EOF döndürür; okuma döngüsü hiç başlamaz.

3 **Bir program fprintf ile log dosyasına 3 satır yazdıktan sonra çöküyor ve dosya 0 bayt kalıyor. Satırlar neden dosyada yok?**

A fprintf satırları dosyaya değil, yalnız ekrana yazar.
B Dosya "a" kipiyle açıldığı için her açılışta içi silinir.
C Çöken programın dosyasını işletim sistemi siler.
D Satırlar programın belleğindeki tamponda bekliyordu; program çökünce tamponla birlikte kayboldu.

4 **Okunacak satır kalmadığında fgets ne döndürür?**

A EOF, yani −1

B NULL

C 0

D Boş bir satır ("")

5 **Program ./oku a.txt b.txt diye çalıştırılıyor. main'e gelen argc kaçtır?**

Cevap:

6 **int sayi = 250000; değeri fprintf(m, "%d", sayi) ile metin dosyasına yazılıyor. Dosya kaç bayt olur?**

Cevap:

7 **Dosyanın içi şöyle:**

```
elma 3
armut 5
erik 4
f bu dosyayı gösteriyor. Aşağıdaki kod ne yazar?
char ad[16];
int adet, toplam = 0;
while (fscanf(f, "%15s %d", ad, &adet) == 2)
    toplam += adet;
printf("%d\n", toplam);
```

Cevap:

8 **Windows'ta int türündeki 10 sayısı "w" kipiyle açılmış bir dosyaya fwrite ile yazılınca dosya 4 değil 5 bayt oluyor. Neden?**

A fwrite her sayının sonuna bir satır sonu baytı ekler.

B Windows'ta int 5 bayttır.

C 10'un ilk baytı 0A, satır sonu baytıyla aynıdır; metin kipi onu 0D 0A'ya çevirir.

D Metin kipi sayıyı rakam rakam yazar ve sonuna boşluk koyar.

PROGRAMLAMAYA GİRİŞ · 1. DÖNEM

12 Önişlemci ve kütüphaneler

Bir C programı tek komutla derlenir, ama bu komutun arkasında birkaç ayrı iş vardır. Bu bölümde # ile başlayan önişlemci komutlarını ve makroların tuzaklarını, kaynak kodun çalışan dosyaya dönüştüğü dört aşamayı, çok dosyalı programları, statik ve dinamik kütüphaneleri öğreneceksin.

KARE(2+3) = ?

videoda 55:26

Bir sayının karesini alan kısa bir KARE kalıbını #define ile yazalım ve 2 + 3'ün karesini isteyelim:

```
#include <stdio.h>
#define KARE(x) x*x

int main(void) {
    printf("%d\n", KARE(2+3));
    return 0;
}
```

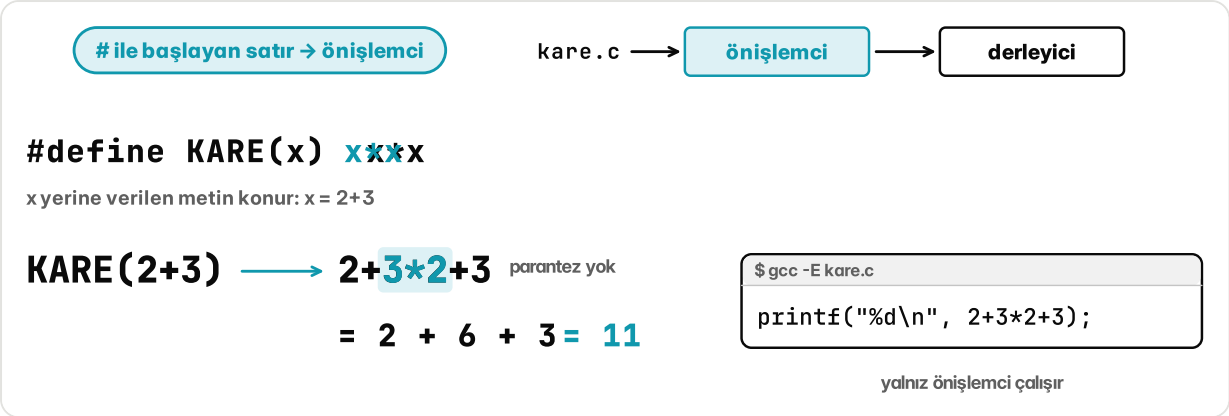
Çıktı:

```
11
```

Beklenen 25'ti; program 11 yazıyor. Hesapta hata yok: bu satır derleyiciye hiç bu hâliyle gitmiyor.

Diyez (#) ile başlayan satırlar C'nin değil, **önişlemci**nin komutlarıdır. Önişlemci derleyiciden önce çalışır ve yalnız metinle uğraşır. #define bir ad tanımlar; KARE gibi parametre alan bir tanıma **makro** denir. Önişlemci KARE adını görünce x'in yerine verileni koyar ve kalıbı satıra yazar. Hesap yapmaz, parantez de eklemez: KARE(2+3), 2+3*2+3 olur. Çarpma önce yapılır: 3 * 2 = 6, sonra 2 + 6 + 3 = 11. gcc'ye -E seçeneğiyle yalnız önişlemciyi çalıştırmasını söylersek (gcc -E kare.c) bu satır kendi gözümüzle görülür:

```
printf("%d\n", 2+3*2+3);
```



Şekil 12.1 Önişlemci KARE(2+3) yazımını 2+3*2+3 metnine çeviriyor; çarpma önce yapıldığı için sonuç 2 + 6 + 3 = 11, gcc -E çıktısında da bu satır görülüyor.

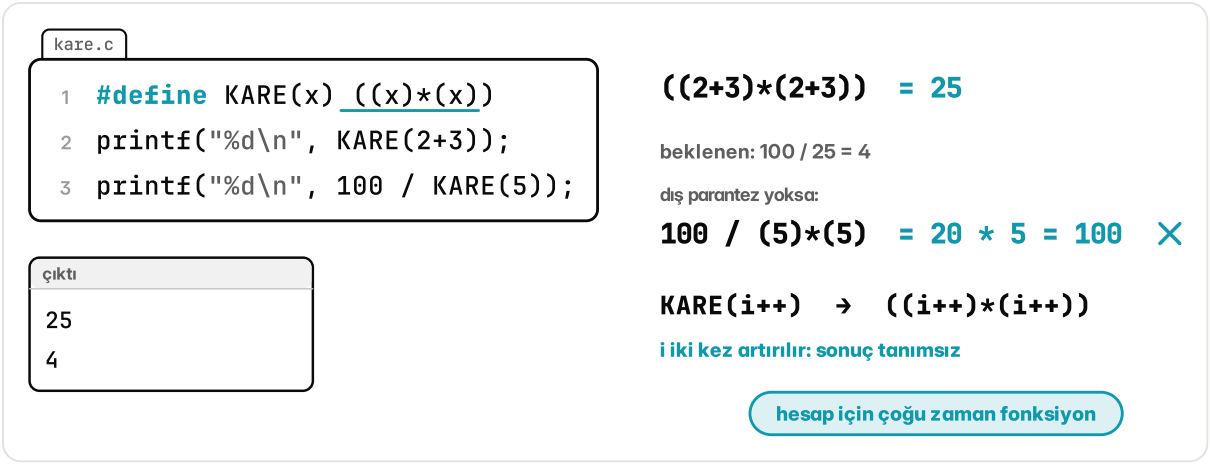
Çözüm parantezdir: makroda her x ve bütün ifade paranteze alınır. Şimdi önce 2 + 3 toplanır ve 25 çıkar.

```
#include <stdio.h>
#define KARE(x) ((x)*(x))

int main(void) {
    printf("%d\n", KARE(2+3));
    printf("%d\n", 100 / KARE(5));
    return 0;
}
```

Çıktı:

```
25
4
```

Dıştaki parantez de gereklidir. 100'ü 5'in karesine bölerken beklenen 4'tür. Makro (x)*(x) diye dış parantezsiz yazılsaydı satır 100 / (5)*(5) olurdu; bölme ve çarpma soldan sağa yapıldığı için önce 100 / 5 = 20, sonra 20 * 5 = 100 çıkardı. Bir tuzak daha var: makro, verileni iki kez yazar. KARE(i++), ((i++)*(i++)) olur ve i iki kez artırılır; aynı ifadede bir değişkeni iki kez değiştirmek tanımsız davranıştır, sonuç derleyiciye göre değişir. Bu yüzden hesap işinde çoğu zaman fonksiyon daha güvenlidir.



Şekil 12.2 Parantezli KARE makrosu 25 ve 4 veriyor; dış parantez olmasa 100 / KARE(5) 100 çıkardı, KARE(i++) ise i'yi iki kez artırıyor.

Önişlemci kod silebilir de. #ifdef ile #endif arasındaki satırlar yalnız o ad tanımlıysa derlenir:

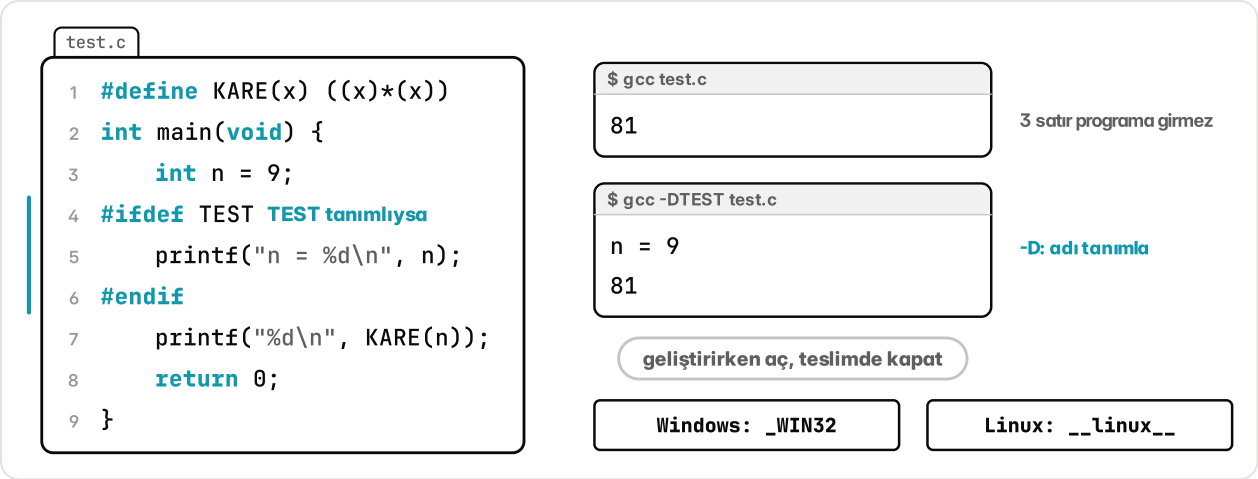
```
#include <stdio.h>
#define KARE(x) ((x)*(x))

int main(void) {
    int n = 9;
#ifdef TEST
    printf("n = %d\n", n);
#endif
    printf("%d\n", KARE(n));
    return 0;
}
```

Burada TEST adı tanımlı değildir: ara değeri yazan satır programa hiç girmez ve gcc test.c ile derlenen program yalnız 81 yazar. Ad, komut satırından -D seçeneğiyle verilince (gcc -DTEST test.c) o satır da derlenir ve önce ara değer gelir.

Çıktı:

```
n = 9
81
```

Geliştirirken açılıp teslimde kapatılan satırlar böyle yazılır. Derleyiciler bazı adları kendiliğinden de tanımlar: Windows'taki derleyiciler _WIN32 adını, Linux'takiler __linux__ adını. Böylece aynı kaynak kod Windows'ta ve Linux'ta farklı satırlarla derlenir.



Şekil 12.3 #ifdef TEST ile #endif arasındaki satır yalnız TEST tanımlıysa derleniyor; gcc test.c 81, gcc -DTEST test.c önce n = 9 sonra 81 yazıyor.

Tek komut, dört aşama

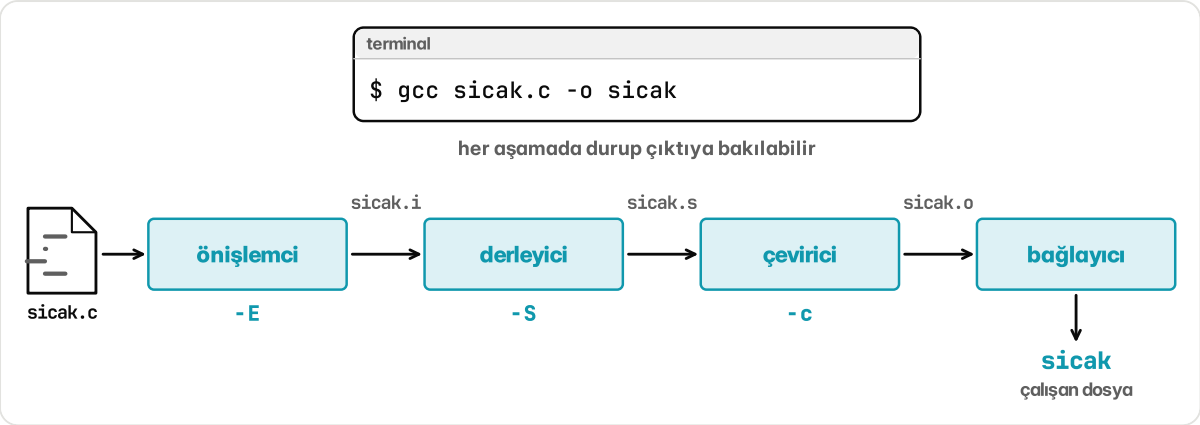
videoda 57:00

gcc'ye tek bir komut veririz ve çalışan bir dosya çıkar: gcc sicak.c -o sicak (-o, çıkacak dosyanın adını verir). Bu tek komutun arkasında dört ayrı iş vardır: önişlemci, derleyici, çevirici ve bağlayıcı. Her birinin çıktısı ayrı ayrı durdurulup görülebilir: -E seçeneği önişlemciden sonra (sicak.i), -S derleyiciden sonra (sicak.s), -c çeviriciden sonra (sicak.o) durur.

```
#include <stdio.h>
#define ESIK 30

int main(void) {
    int derece = 34;   // ölçülen
    if (derece > ESIK)
        printf("%d derece\n", derece);
    return 0;
}
```

Çıktı:

```
34 derece
```



Şekil 12.4 gcc sicak.c -o sicak komutunun dört aşaması: önişlemci, derleyici, çevirici ve bağlayıcı; ara dosyalar sicak.i, sicak.s ve sicak.o.

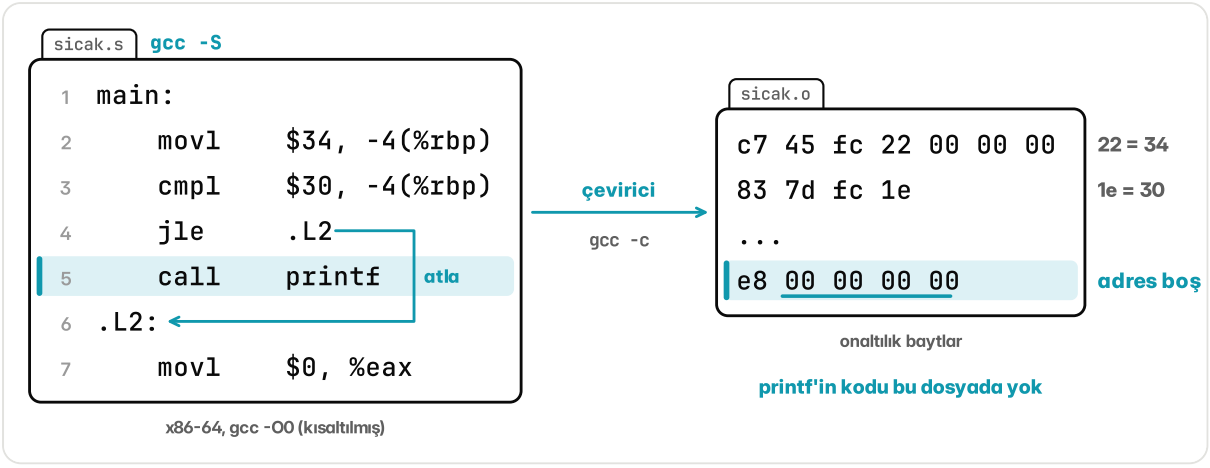
Önişlemci, #include satırını görünce o başlık dosyasının içeriğini olduğu gibi satırın yerine yapıştırır. 9 satırlık programımız yüzlerce satıra çıkar; printf'in tanıtıldığı satır da bunların arasındadır. ESIK adının yerine 30 yazılır, yorum silinir. Çıkan dosya hâlâ C'dir, ama içinde önişlemci komutu kalmamıştır. Dosyanın sonu şöyledir:

```
int main(void) {
    int derece = 34;
    if (derece > 30)
        printf("%d derece\n", derece);
    return 0;
}
```

Derleyici bu C metnini assembly diline çevirir: işlemcinin komutları, okunabilir adlarla. Önce derece değişkenine 34 yazılır ve 30 ile karşılaştırılır; derece 30'dan küçük ya da 30'a eşitse printf'in üstünden atlanır. **Çevirici** (assembler) bu satırları makine koduna, yani baytlara çevirir ve bir **nesne dosyası** (object file) yazar. x86-64 işlemcide bu komutlardan üçü ve baytları şöyledir:

```
movl $34, -4(%rbp)    c7 45 fc 22 00 00 00    derece = 34 (0x22 = 34)
cmpl $30, -4(%rbp)    83 7d fc 1e             30 ile karşılaştır (0x1e = 30)
call printf           e8 00 00 00 00          printf'i çağır: adres boş
```

Nesne dosyası henüz çalışmaz: printf'i çağıran komutta adresin yerinde sıfırlar durur, çünkü printf'in kodu bu dosyada yoktur.



Şekil 12.5 sicak.s'deki movl, cmpl, jle ve call komutları ve sicak.o'daki baytları; call printf komutunda adres henüz 00 00 00 00.

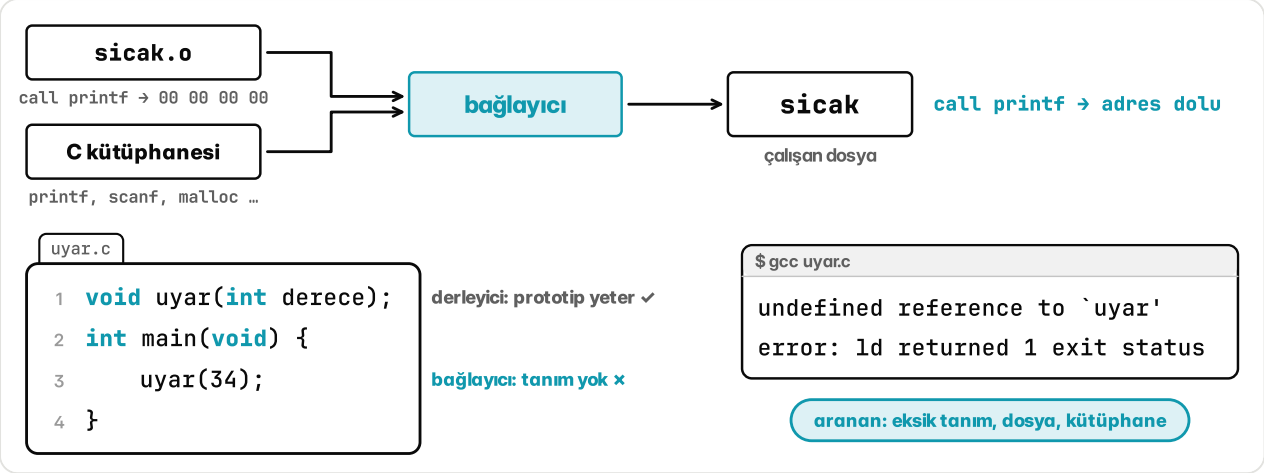
Bağlayıcı (linker), nesne dosyalarını ve C kütüphanesini birleştirir. printf'in makine kodu kütüphanededir; bağlayıcı boş adresi doldurur ve çalışan dosyayı yazar. Derleyici her dosyayı ayrı ayrı çevirdiği için başka dosyalarda ne olduğunu bilmez. Bir fonksiyonun prototipi olup tanımı hiçbir yerde yoksa derleyici ses çıkarmaz, çünkü prototipe güvenir:

```
#include <stdio.h>

void uyar(int derece);

int main(void) {
    uyar(34);
    return 0;
}
```

Hatayı bağlayıcı verir: bu ad hiçbir dosyada tanımlı değildir. gcc'nin bağlayıcısı bunu undefined reference (tanımsız başvuru) diye bildirir; bu örneğin derlendiği clang'ın bağlayıcısı şöyle yazar:

```
lld-link: error: undefined symbol: uyar
```

Mesaj bağlayıcıdan geliyorsa sorun satırda değildir; eksik bir tanım, dosya ya da kütüphane aranır.



Şekil 12.6 Bağlayıcı sicak.o'yu C kütüphanesiyle birleştirip printf'in adresini dolduruyor; tanımı olmayan uyar için bağlama hatası veriyor.

DLL bulunamadı

videoda 58:19

Bir oyunu açmak isteriz ve Windows, MSVCP140.dll dosyası bulunamadığı için kodun yürütülemediğini söyler. Program bilgisayarda duruyor; eksik olan ne? Cevap, bir programın kodunun her zaman tek dosyada durmamasında.

Büyük bir program tek dosyaya yazılmaz. Sıcaklığı Fahrenheit'a çeviren fonksiyon kendi kaynak dosyasında, sicaklik.c'de durur:

```
#include "sicaklik.h"

int cevrim = 0;

double fahrenheit(double c) {
    cevrim++;
    return c * 1.8 + 32;
}
```

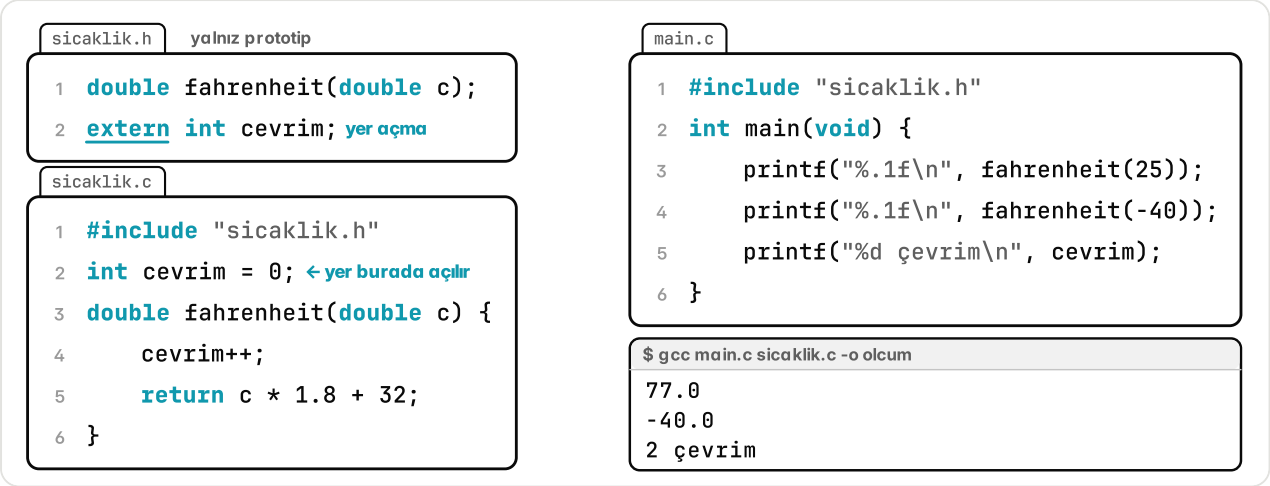
Başlık dosyası sicaklik.h'de ise yalnız fonksiyonun prototipi vardır. Ana dosya başlığı #include ile ekler ve fonksiyonu, nerede yazıldığını bilmeden çağırır. Kendi başlıklarımız "sicaklik.h" diye tırnakla, sistemin başlıkları <stdio.h> diye köşeli ayraçla eklenir.

```
#include <stdio.h>
#include "sicaklik.h"

int main(void) {
    printf("%.1f\n", fahrenheit(25));
    printf("%.1f\n", fahrenheit(-40));
    printf("%d çevrim\n", cevrim);
    return 0;
}
```

Başlıktaki extern int cevrim; satırı aynı işi bir değişken için yapar: bu değişken başka bir dosyada var, burada yer açma der. cevrim'in yeri yalnız sicaklik.c'de açılır. Dosyalar ayrı ayrı derlenir, bağlayıcı birleştirir (gcc main.c sicaklik.c -o olcum); bir dosya değişince yalnız o yeniden derlenir.

Çıktı:

```
77.0
-40.0
2 çevrim
```

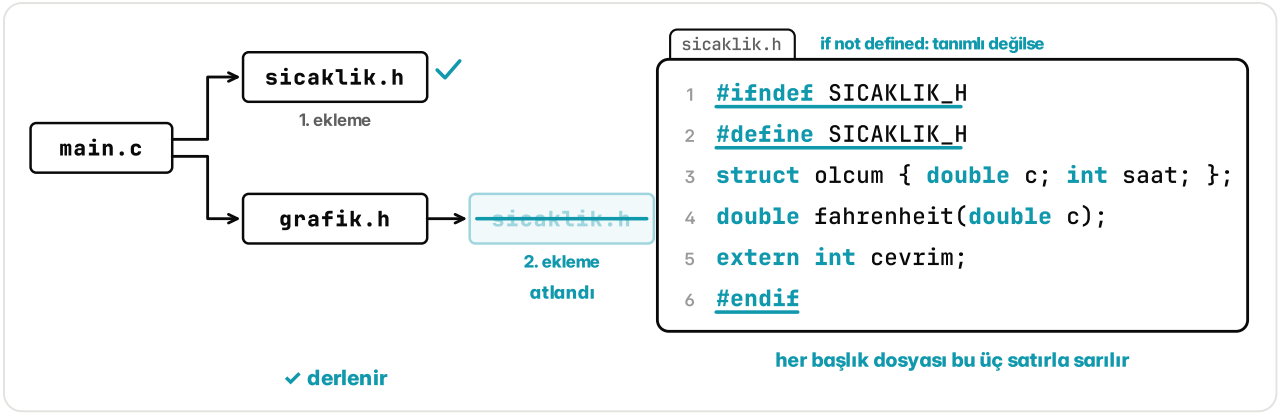


Şekil 12.7 sicaklik.h'de prototip ve extern, sicaklik.c'de fahrenheit ve cevrim, main.c'de çağrılar; birlikte derlenen program 77.0, -40.0 ve 2 çevrim yazıyor.

Aynı başlık, başka başlıkların içinden ikinci kez eklenebilir: grafik.h de sicaklik.h'yi ekler ve ikisini birden ekleyen bir dosyaya sicaklik.h iki kez yapıştırılır. İçinde struct tanımı varsa aynı struct iki kez tanımlanır ve derleme durur. Önlemi **başlık koruması**dır:

```
#ifndef SICAKLIK_H
#define SICAKLIK_H

struct olcum { double c; int saat; };
double fahrenheit(double c);
extern int cevrim;

#endif
```

Buradaki #ifndef, "bu ad tanımlı değilse" demektir. İlk eklemede SICAKLIK_H tanımsızdır; içerik alınır ve ad hemen tanımlanır. İkinci eklemede soru yanlış çıkar ve önişlemci #endif'e kadar her şeyi atlar. Bu yüzden her başlık dosyası bu üç satırla sarılır.



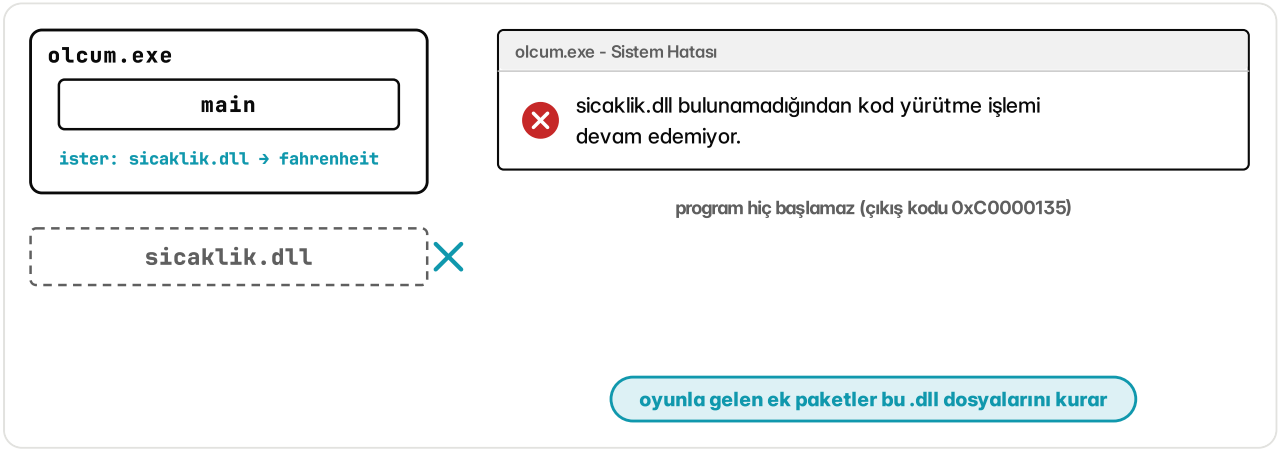
Şekil 12.8 sicaklik.h hem main.c'den hem grafik.h'den ekleniyor; korumasız başlık struct olcum'u iki kez tanımlıyor, korumalıda ikinci ekleme atlanıyor.

Derlenmiş nesne dosyaları tek bir arşivde toplanınca **kütüphane** olur. **Statik kütüphanede** (Linux'ta .a, Windows'ta .lib) bağlayıcı, kullanılan kodu programın içine kopyalar. ar arşivi kurar; -L. kütüphanenin bu klasörde aranacağını, -lsicaklik de libsicaklik.a'nın kullanılacağını söyler:

```
ar rcs libsicaklik.a sicaklik.o
gcc main.c -L. -lsicaklik -o olcum
```

Program büyür ama tek başına çalışır, yanında başka dosya istemez. Kütüphanede bir hata düzeltilirse programın yeniden bağlanması gerekir.

Dinamik kütüphanede (Linux'ta .so, Windows'ta .dll) kod programa kopyalanmaz; programa yalnız kütüphanenin ve fonksiyonların adı yazılır. Program açılırken işletim sistemi kütüphaneyi bulup belleğe yükler; birçok program aynı kopyayı paylaşır. Kütüphane güncellenince hepsi yeni kodu kullanır. Bedeli şudur: dosya bulunamazsa program hiç başlamaz; sicaklik.dll silinince olcum tek satır yazamadan kapanır. "DLL bulunamadı" hatası budur. Oyunla birlikte kurulan ek paketler de işte bu kütüphaneleri getirir.



Şekil 12.9 olcum.exe açılırken sicaklik.dll'i yüklüyor ve birçok program aynı DLL'i paylaşıyor; DLL yoksa program hiç başlamıyor.

Özet

- Diyez (#) ile başlayan satırlar önişlemcinin komutlarıdır; önişlemci derleyiciden önce çalışır ve yalnız metni değiştirir.
- Makroda her parametre ve bütün ifade paranteze alınır: KARE(x) ((x)*(x)); makro verileni iki kez yazabileceği için hesapta çoğu zaman fonksiyon daha güvenlidir.
- Koşullu derlemede #ifdef ile #endif arasındaki satırlar yalnız o ad tanımlıysa derlenir; ad, gcc'ye -D ile verilebilir.
- Tek gcc komutunun arkasında dört aşama vardır: önişlemci, derleyici, çevirici ve bağlayıcı; tanımı bulunamayan fonksiyonun hatasını bağlayıcı verir.
- Başlık dosyası prototipleri ve extern bildirimlerini taşır; başlık koruması aynı başlığın iki kez işlenmesini önler.
- Statik kütüphanenin kodu programa kopyalanır; dinamik kütüphane program açılırken yüklenir ve bulunamazsa program başlamaz.

Sorular

1 **gcc sicak.c -o sicak komutunun arkasındaki dört aşama hangi sırayla çalışır?**

A Derleyici, önişlemci, çevirici, bağlayıcı

B Önişlemci, bağlayıcı, derleyici, çevirici

C Önişlemci, çevirici, derleyici, bağlayıcı

D Önişlemci, derleyici, çevirici, bağlayıcı

2 **gcc'ye yalnız önişlemciyi çalıştırıp sonucunu göstermesini söyleyen seçenek hangisidir?**

Cevap:

3 **#define KARE(x) x*x**

```
olduğuna göre printf("%d\n", KARE(1+2)); ne yazar?
```

Cevap:

4 **#define IKI_KAT(x) (x)+(x)**

```
olduğuna göre printf("%d\n", 3 * IKI_KAT(4)); ne yazar?
```

A 24

B 16

C 28

D 12

5 **#include <stdio.h>**

```
int main(void) {
#ifdef HATA_AYIKLA
    printf("başladı\n");
#endif
    printf("bitti\n");
    return 0;
}
Bu program gcc -DHATA_AYIKLA prog.c komutuyla derlenip çalıştırılırsa ne yazar?
```

A Önce başladı, sonra bitti

B Yalnız bitti

C Yalnız başladı

D Derleme hatası verir, çünkü HATA_AYIKLA koda #define ile yazılmamış.

6 **Bir fonksiyonun prototipi yazılmış ve fonksiyon çağrılmış, ama tanımı hiçbir dosyada yok. Bu hatayı hangi aşama bildirir?**

A Önişlemci

B Derleyici

C Bağlayıcı

D Hiçbiri; program çalışırken çöker.

7 **Bir başlık dosyası neden #ifndef SICAKLIK_H, #define SICAKLIK_H ve #endif satırlarıyla sarılır?**

A Başlığın yalnız Windows'ta derlenmesi için.

B Başlık bir dosyaya ikinci kez eklenince içeriği atlansın ve içindeki struct iki kez tanımlanmasın diye.

C Başlıktaki fonksiyonların kodu programa kopyalansın diye.

D Başlıktaki değişkenlere bellekte yer açılsın diye.

8 **Bir program açılırken "sicaklik.dll bulunamadı" diye hata veriyor ve hiç başlamıyor. Hangisi doğrudur?**

A Program sicaklik.dll'deki kodu dinamik kütüphaneden kullanıyor; işletim sistemi dosyayı açılışta yükleyemedi.

B Program statik kütüphane kullanıyor; kütüphanenin kodu programın içinde eksik kalmış.

C Bağlayıcı derleme sırasında hata vermiş, program hiç oluşmamış.

D Başlık dosyasında başlık koruması unutulmuş.

Cevaplar ve çözümler

0 İlk C programı

1. **A** · main

   Program main'den başlar; her C programında bir main bulunur. #include printf'in tanıtıldığı dosyayı ekler, return 0 ise programı bitirir.

2. **C** · MerhabaDünya

   printf tırnak içindekini olduğu gibi yazar; boşluk bırakmaz, alt satıra da kendiliğinden geçmez. İki yazı bitişik çıkar. Alt satıra geçmek için \n yazmak gerekir.

3. **3**

   İlk printf A yazar ve \n ile alt satıra geçer, sonra B yazar ve yine alt satıra geçer. İkinci printf C'yi üçüncü satıra yazar. Ekranda 3 satır vardır: A, B ve C.

4. **D** · Programı bitirir ve işletim sistemine her şeyin yolunda gittiğini bildirir.

   return 0 programı bitirir; 0 işletim sistemine başarıyı, 0'dan başka bir sayı bir sorun çıktığını bildirir. Ekrana bir şey yazmaz; gövdeyi açan da süslü parantezdir.

5. **Merhaba**

   // işaretinden sonrası satır sonuna kadar yorumdur; derleyici onu programa katmaz. Ekrana yalnızca Merhaba yazılır, \n de imleci alt satıra geçirir.

6. **B** · Derleme hatası yüzünden çalıştırılabilir dosya hiç üretilmez; çalışacak bir program yoktur.

   Derleyici kaynak dosyanın tamamını önce okur; 5. satırda hata bulunca durur ve çalıştırılabilir dosya üretmez. Satırlar tek tek çalıştırılmadığı için 4. satır da hiç çalışmaz. Programı çalışırken durduran hata, çalışma zamanı hatasıdır; burada program hiç çalışmamıştır.

7. **D** · Dosya üretilir; çalıştırınca çoğu bilgisayarda program bu satırda durdurulur.

   Uyarı derlemeyi durdurmaz, çalıştırılabilir dosya yine de üretilir. Program çalışırken sıfıra bölmeye gelince çoğu bilgisayarda işletim sistemi onu durdurur. Sıfıra bölmenin sonucu olmadığı için 0 yazılmaz.

8. **C** · Çalışma zamanı hatası: program çalışırken bir hataya çarpmıştır.

   Program derlenmiş ve çalışmıştır, yani derleme hatası yoktur. Sayı kullanıcıdan geldiği için derleyici 0'ı göremez; hata ancak program çalışırken ortaya çıkar. Bu, çalışma zamanı hatasıdır.

1 Değişkenler, tipler, girdi-çıktı

1. **B** · Çoğu zaman anlamsız bir sayı; bir sonraki çalıştırmada başka bir sayı çıkabilir.

   C, fonksiyonun içindeki değişkene ayırdığı yeri temizlemez; orada önceki işten kalan bitler okunur. Buna çöp değer denir ve çalıştırmadan çalıştırmaya değişebilir. Program derlenir ve çalışır; 0 görmek için ilk değeri kendin vermelisin.

2. **C** · puan'ın önünde & yok; scanf değişkenin bellekteki yerini ister.

   scanf okuduğu sayıyı değişkenin bellekteki yerine kendisi yazar, bu yüzden adın önüne & konur: scanf("%d", &puan);. & unutulursa program çoğu zaman çöker.

3. 21

Program derlenmeden önce HAFTA yazan yere 7 konur; derleyici int gun = 3 * 7; satırını görür. 3 × 7 = 21.

4. D · char: 1 bayt

C tiplerin boyuna yalnız alt sınır koyar: int bir denetleyici kartında 2 bayt olabilir, long 64 bitlik Windows'ta 4, 64 bitlik Linux'ta 8 bayttır. Yalnız char'ın boyu her makinede 1 bayttır; emin olmak için boyu sizeof ile sorarız.

5. 32

char bir tam sayıdır ve derleyici tırnaktaki harfin yerine kodunu koyar. Her küçük harf büyüğünden aynı uzaklıktadır; 'a' ile 'A' arasındaki fark 32'dir. Bu yüzden 'h' − 32 = 104 − 32 = 72, yani 'H'.

6. A · 72.46

%.2f noktadan sonra 2 basamak yazar ve yuvarlar: üçüncü basamak 6 olduğu için 72.46. 72.456000, %f'nin 6 basamaklı yazımıdır; 72.45 yuvarlamak yerine kesmek olurdu.

7. 4294967295

−1 bellekte 32 tane 1 ile durur. %u bu bitleri işaretsiz okur: $2^{32} - 1 = 4.294.967.295$. Bitler aynıdır, yalnız okuma biçimi değişir.

8. B · Karşılaştırmada a işaretsize çevrilir ve 4.294.967.295 olarak okunur.

int ile unsigned int karşılaşınca C, int'i sessizce işaretsize çevirir. −1'in 32 biri işaretsiz okununca 4.294.967.295 olur; 4.294.967.295 < 1 yanlıştır ve else çalışır. b'nin değeri 1'dir, değişmez.

2 Operatörler, koşullar, döngüler

1. C · x beş yazar; atama koşulu 5 yapar ve 5 doğrudur.

Tek eşittir karşılaştırmaz, x'e 5 atar; atamanın değeri de konan sayıdır, yani 5. 5 sıfır olmadığı için koşul doğrudur ve x beş yazılır. Satırdan sonra x de 5'tir; gcc -Wall bu satır için uyarır ama derleme durmaz.

2. B · Yalnız bitti

&& işleminde ilk koşul yanlışsa sonuç zaten yanlıştır ve C ikinci koşulu hesaplamaz. b 0 olduğu için b != 0 yanlıştır, bölme hiç yapılmaz ve yalnız bitti yazılır. Sıra ters olsaydı önce 0'a bölme yapılırdı.

3. -2

C bölümü sıfıra doğru keser: −10 / 4 = −2,5, küsurat atılır, −2. Kural bölüm × bölen + kalan = bölünen: −2 × 4 = −8 ve −10'a ulaşmak için −2 gerekir. Kalan bölünenin işaretini taşır; matematikteki cevap olan 2 çıkmaz.

4. D · a 5, i 6

++ arkada olduğu için önce i'nin eski değeri, 5, a'ya verilir; sonra i 1 artar ve 6 olur. int a = ++i; yazılsaydı önce i artardı ve a da 6 olurdu.

5. A · 1 kez

for satırının sonundaki noktalı virgül boş bir komuttur ve döngünün gövdesi olur. Döngü 5 tur boş döner; printf döngünün dışında kalır ve bir kez çalışır. İçeri girintili yazılması bir şey değiştirmez, C girintiye bakmaz.

6. **6**

   i 0'dan başlar ve 5'e eşitken de koşul doğrudur: i sırayla 0, 1, 2, 3, 4 ve 5 olur. Bu 6 turdur; 5 tur için koşul i < 5 olmalıdır.

7. **C** · Kahve ve Su

   switch değeri eşit olan case 2'ye atlar ve Kahve yazar. break olmadığı için akış aşağı düşer, default'taki Su da yazılır. case 1 atlandığı için Çay yazılmaz.

8. **B** · switch'ten; döngü sürer.

   break, içinde bulunduğu en yakın döngüden ya da switch'ten çıkar; if bunlardan biri değildir. En yakını switch olduğu için break switch'ten çıkar, döngü ise sürer. 1990'daki telefon şebekesi hatası da if'ten çıkmak isteyen böyle bir break'ten doğdu.

3 Diziler ve karakter dizileri

1. **B** · 0

   Beş gözlük diziye üç değer verilince kalan gözler 0 ile dolar: dizi 12, 7, 30, 0, 0 olur. puan[3] dördüncü elemandır, yani 0.

2. **24**

   Boyu yazılmadığı için derleyici değerleri sayar: 6 eleman. Her eleman 4 bayt olduğu için dizi 6 × 4 = 24 bayt tutar. Eleman sayısı da 24 / 4 = 6 diye bulunur.

3. **C** · C adresi hesaplar ve dizinin dışına, başka bir verinin yerine yazar; sonuç tanımsız davranıştır.

   C yalnız adresi hesaplar: baş + 9 × 4 = baş + 36. Dizi baş + 20'de biter; C sınırı denetlemez ve oraya yazar. Program çökebilir ya da başka bir değişken sessizce bozulabilir.

4. **5 4**

   Derleyici "kedi"nin sonuna sonlandırıcıyı ('\0') ekler; dizi 4 harf + 1 = 5 bayttır. strlen sonlandırıcıya kadar sayar ve 4 verir. Çıktı: 5 4.

5. **A** · a; baştan 9 bayt

   Satırlar bellekte arka arkaya durur ve her satır 4 bayttır. Yer = 2 × 4 + 1 = 9. Satır 2 can'dır; sütun 1 onun ikinci harfi, a'dır.

6. **D** · Dizinin tamamına tanımdan sonra = ile tek seferde değer verilemez; strcpy kullanılmalıdır.

   C'de dizi tanımdan sonra = ile atanamaz. Metni diziye koymak için strcpy(ad, "can"); yazılır. can 3 harf ve sonlandırıcıyla 4 bayttır, 8 göze rahatça sığar.

7. **B** · farklı

   Metinler eşit olduğu için strcmp 0 verir; C'de 0 yanlış demektir ve else kolu çalışır. Eşitliği denetlemek için koşul strcmp(cevap, "evet") == 0 olmalıdır.

8. **6**

   fgets Enter'ın bıraktığı satır sonu karakterini de diziye yazar ve sona sonlandırıcıyı koyar: e, v, e, t, '\n', '\0'. Toplam 6 bayt; 8 gözlük diziye sığar.

4 Fonksiyonlar ve değişkenlerin ömrü

1. **C** · Fonksiyonun geri vereceği değerin tipini

   Tanımın ilk sözcüğü dönüş tipidir: kdvli kesirli bir sayı geri verir. Parametrenin tipi parantezin içindeki double'dır.

2. **81**

   pow üssü hesaplar: 3 üzeri 4 = 3 × 3 × 3 × 3 = 81. %.0f sonucu ondalık basamaksız yazar.

3. **22**

   ucle(7) çağrısında x 7 olur ve return 7 × 3 = 21'i çağrıldığı yere geri verir. 21 + 1 = 22 yazılır.

4. **A** · Derleyici dosyayı yukarıdan aşağı okur; çağrıyı gördüğünde kare adını henüz tanımaz.

   Fonksiyon main'in altında tanımlı ve başta prototipi yok. Başa int kare(int x); prototipi yazılırsa program derlenir ve 25 yazar.

5. **3**

   C fonksiyona x'in kendisini değil, değerinin kopyasını verir. artir kendi kopyası olan n'yi 8 yapar; main'deki x 3 olarak kalır.

6. **C** · b'nin çerçevesi

   Çerçeveler üst üste kurulur: main, a, en üstte b. En son çağrılan ilk biter; önce b'nin çerçevesi kalkar, sonra a'nınki, en son main'inki.

7. **D** · 1 2 3 4

   static yerel değişken program başlarken bir kez 0 ile kurulur ve çağrılar arasında değerini korur. Her çağrı n'yi bir artırıp yazar: 1 2 3 4. static olmasaydı her çağrıda 1 yazılırdı.

8. **B** · 10

   ekle'nin içindeki int toplam global olanı gölgeler; fonksiyon kendi yerel değişkenini değiştirir. Global toplam hiç değişmez ve main 10 yazar.

5 İşaretçiler

1. **B** · x'in adresi

   & değişkenin değerini değil adresini verir; p'ye x'in adresi yazılır ve p x'i gösterir. 7'ye ulaşmak için *p yazılır.

2. **10**

   *p, p'nin gösterdiği yere, yani x'in kendisine gider. *p + 3 = 7 + 3 = 10 olur ve bu değer x'e yazılır.

3. **C** · void sifirla(int *n) { *n = 0; } ve sifirla(&sayi);

   Fonksiyona sayi'nın adresi verilmeli ve fonksiyon o adresteki değeri *n ile değiştirmelidir. Birinci seçenekte yalnız kopya değişir; ikincide işaretçinin kendi kopyası 0 olur, sayi'ya dokunulmaz.

4. **A** · adet'in önündeki & unutulmuştur; scanf okuduğu sayıyı yazacağı yerin adresini ister.

   Doğrusu scanf("%d", &adet); olur. & unutulunca scanf, adet'in içindeki rastgele değeri adres sanıp oraya yazmaya kalkar; bu tanımsız davranıştır ve program çökebilir.

5. **8012**

   C işaretçiye eklenen sayıyı eleman olarak sayar: 3 eleman, 3 × 4 = 12 bayt demektir. 8000 + 12 = 8012.

6. **D** · 55

   a ilk elemanın adresidir; a + 1 bir sonraki elemanın adresi, *(a + 1) de a[1], yani 15'tir. a[3] 40'tır: 15 + 40 = 55.

7. **C** · 8

   Fonksiyona dizinin kopyası değil, ilk elemanın adresi gider; int d[] parametresi aslında bir int işaretçisidir. 64 bitlik bilgisayarda işaretçi 8 bayttır. Bu yüzden eleman sayısı ayrı bir parametreyle verilir.

8. **B** · *p = 3;

   const yıldızdan önce yazıldığı için p'nin gösterdiği değer p üzerinden değiştirilemez; *p = 3; derlenmez. p'nin kendisi sabit değildir, başka bir yeri gösterebilir; değeri okumak da serbesttir.

6 Dinamik bellek yönetimi

1. **C** · Stack'te durur; fonksiyon bitince çerçeveyle birlikte kalkar.

   Yerel değişkenler ve yerel diziler fonksiyonun stack'teki çerçevesindedir; fonksiyon bitince çerçeve kalkar ve onlar da gider. Heap'te yalnız program çalışırken malloc gibi fonksiyonlarla istenen bellek durur.

2. **B** · 6 baytlık, yani 1,5 int'lik yer

   malloc eleman sayısı değil bayt sayısı ister; 6 yazınca 6 bayt verir, bu da 1,5 int'lik yerdir. 6 sayılık yer için malloc(6 * sizeof(int)) yazılmalıdır. Satır derlenir; hata ancak bloğun dışına yazılınca ortaya çıkar.

3. **100**

   sizeof(int) 4'tür; 25 × 4 = 100 bayt.

4. **D** · calloc bloğun her baytını sıfırlayarak verir; malloc'un bloğunda önceden kalmış çöp değerler olabilir.

   İkisi de aynı boyda bloğu heap'ten alır ve ikisi de free ile geri verilir; yer yetmezse ikisi de NULL döndürür. Tek fark, calloc'un bloğu sıfırlayarak vermesidir: 6 elemanı yazdırınca 0 0 0 0 0 0 çıkar.

5. **A** · sayilar hâlâ eski adresi tutar, ama o blok artık bizim değildir.

   free bloğu heap'e geri verir ama işaretçideki adresi silmez. O blok başka bir değişkene verilmiş olabileceği için oraya yazmak tanımsız davranıştır. Bu yüzden free'den hemen sonra sayilar = NULL; yazılır; böylece yanlışlıkla ikinci kez free edilse bile free(NULL) hiçbir şey yapmaz.

6. **C** · realloc başarısız olursa NULL döner ve eski bloğun adresi kaybolur; blok geri verilemez.

   realloc başarısız olursa NULL döndürür ve eski blok olduğu gibi kalır. Sonuç doğrudan sayilar'a yazılırsa eski adres silinir, blok sızar. Bu yüzden sonuç önce ayrı bir işaretçiye (yeni) alınır, NULL değilse sayilar'a yazılır. Başarılı olduğunda realloc eski sayıları yeni yere taşır.

7. **B** · Dönen adres, fonksiyon bitince kalkan stack çerçevesindeki diziyi gösterir.

   dizi yerel bir dizidir ve uret'in stack'teki çerçevesinde durur. Fonksiyon bitince çerçeve kalkar; dönen adresten okumak tanımsız davranıştır. Doğrusu diziyi malloc ile heap'ten alıp adresini döndürmektir; heap'teki bir bloğun adresini döndürmek sorun değildir.

8. **360000**

   Saniyede 50 × 2 = 100 KB sızar. 1 saat 3.600 saniyedir: 100 × 3.600 = 360.000 KB, yani yaklaşık 360 MB.

7 Özyineleme ve fonksiyon işaretçileri

1. **D** · Her adımda sorunu küçültüp sonunda varılan bir durma koşulu

   Durma koşulu, fonksiyonun kendini çağırmadan cevap verdiği durumdur; özyinelemeli adım da sorunu her seferinde küçültüp oraya ulaştırır. Derleyici çağrıları sınırlamaz; durma koşulu yoksa çağrılar stack dolana kadar sürer ve program çöker.

2. **10**

   Her adım son rakamı (n % 10) alır ve kalan sayıyla (n / 10) devam eder: 6 + rakam_topla(202), 2 + rakam_topla(20), 0 + rakam_topla(2), 2 + rakam_topla(0) = 2 + 0. Dönüşte toplanınca 2 + 0 + 2 + 6 = 10.

3. **B** · 4

   573, 57, 5 ve 0 için dört çerçeve üst üste kurulur; 0'ın çerçevesi durma koşuluyla 0 döndürene kadar hiçbiri kalkmaz. 3 rakam olduğu için 3 demek yaygın yanılgıdır: durma koşulunun çağrısı da bir çerçeve açar.

4. **B** · Fonksiyon bulduğu cevabı hatırlamaz; aynı değerler tekrar tekrar baştan hesaplanır

   Her çağrı iki yeni çağrı açar ve aynı sıradaki sayılar ağacın farklı dallarında yeniden hesaplanır; fib(2) tek başına 63 milyon kez hesaplanır. Cevap ise yalnız 102.334.155'tir. Bulunan değerleri bir dizide tutan hafızalı sürüm 79 çağrıyla biter.

5. **31**

   n disk için $2^n - 1$ hamle gerekir: $2^5 - 1 = 32 - 1 = 31$. Aynı sonuç, her yeni diskin hamle sayısını iki katına çıkarıp bir eklemesinden de çıkar: 1, 3, 7, 15, 31.

6. **27**

   islem, kup fonksiyonunun adresini tutar; islem(3) çağrısı kup(3)'ü çalıştırır: 3 × 3 × 3 = 27.

7. **A** · İlki int alıp int döndüren bir fonksiyonun adresini tutan değişkendir; ikincisi int adresi döndüren bir fonksiyonun bildirimidir.

   Parantez yıldızı islem'e bağlar: islem bir işaretçidir ve gösterdiği şey int alıp int döndüren bir fonksiyondur. Parantez olmazsa yıldız dönüş tipine bağlanır ve satır, int * döndüren islem adlı bir fonksiyonu bildirir.

8. **C** · Karşılaştırma fonksiyonundaki küçük ve büyük işaretlerinin yeri değiştirilir.

   qsort sırayı kendisi bilmez; iki elemandan hangisinin önce geleceğini karşılaştırma fonksiyonuna sorar. Fonksiyonda x > y için -1, x < y için 1 döndürülünce büyük sayı önce gelir: 33, 9, 76, 18, 51 dizisi 76 51 33 18 9 olur.

8 Yapılar: struct

1. **B** · Bellekte yer ayrılmaz; tanım yalnız bir kalıptır, yer bu tipte bir değişken tanımlanınca ayrılır.

   Yapı tanımı yeni bir tip kurar; hangi alanların hangi sırayla duracağını söyler. Yer, struct ogrenci ogr; gibi bir değişken tanımlanınca ayrılır; o zaman 16 + 4 + 4 = 24 bayt tutar.

2. **Mert 74**

   İlk değerler alanların sırasıyla verilir: ad Mert, numara 1043, notu 64. İkinci satır yalnız notu alanını değiştirir: 64 + 10 = 74. printf adı ve notu yazar: Mert 74.

3. **D** · 2

   Derleyici enum'daki adları 0'dan başlayarak sırayla numaralar: GUZ 0, BAHAR 1, YAZ 2. Saymaya 1'den başlamak yaygın yanılgıdır; o zaman 3 bulunur.

4. **C** · a = b; yapının bütün alanlarını, içindeki dizi dahil kopyalar; x = y; ise derlenmez.

   C'de dizi eşittir işaretiyle başka bir diziye atanamaz. Yapılar ise atanabilir: eşittir işareti yapının bütün baytlarını, içindeki char ad[16] dizisi dahil kopyalar. Yapı dizisinde iki öğrenciyi yer değiştirmek bu yüzden öğrenci başına tek atamadır.

5. **8**

   a 0. bayta, b 1. bayta oturur. c 4'ün katında başlamalıdır; 2. ve 3. baytlar dolgu olur, c 4-7. baytları alır. Toplam 8 bayt; 8 zaten 4'ün katı olduğu için sona dolgu eklenmez.

6. **A** · Kayıtlar yan yana dizildiğinde her kaydın numara alanı yine 4'ün katı olan bir adrese düşsün diye

   Dizide kayıtlar arka arkaya durur. Yapı 9 bayt olsaydı ikinci kaydın numarası 9 + 4 = 13. bayta düşerdi; 12 bayt olunca 12 + 4 = 16. bayta, yani 4'ün katına düşer. char alanları hizalama istemez, 1 baytta okunur.

7. **C** · 64

   Yapı fonksiyona kopyalanarak verilir; fonksiyon kopyanın notunu 70 yapar, ogr'nin kendisi değişmez ve 64 kalır. Asıl kaydı değiştirmek için adres verilir: void ekle(Ogrenci *p, int ek) ve ekle(&ogr, 6);.

8. **D** · (*p).notu

   Önce yıldızla işaretçinin gösterdiği yapıya gidilir, sonra noktayla alan seçilir: (*p).notu. Noktanın önceliği yıldızdan yüksek olduğu için parantez şarttır; *p.notu, *(p.notu) demektir ve derlenmez. Ok işareti bu uzun yazımın kısasıdır.

9 Union ve bitsel işlemler

1. **C** · 0 ve 1

   && iki tarafın da 0'dan farklı olup olmadığına bakar; 4 de 2 de 0 olmadığı için 1 verir. & ise bitlere bakar: 100 ile 010'da aynı sırada ikisinde birden 1 olan bit yoktur, sonuç 0'dır. 6, 4 | 2 işleminin sonucudur.

2. **5**

   6 = 110, 3 = 011. Özel veya, iki bit farklıysa 1 verir: 110 ^ 011 = 101, yani 5.

3. **9**

   0x01 = 00000001. 1 << 3, yalnız 3. biti 1 olan 00001000 = 8 maskesidir. VEYA ile bayt 00001001 olur: 8 + 1 = 9.

4. **A** · C, == karşılaştırmasını &'den önce yapar; koşul x & (1 == 0), yani x & 0 olur ve bu her zaman 0'dır.

   == karşılaştırması &'den önce yapılır: önce 1 == 0 hesaplanır ve 0 çıkar, x & 0 da her x için 0'dır. Doğrusu (x & 1) == 0 yazmaktır; 6 = 110'un en sağdaki biti 0 olduğu için o zaman koşul tutar.

5. **D** · port &= ~(1 << 5);

   ~(1 << 5) yalnız 5. biti 0 olan maskedir; &= ile bu 0 lambanın bitini kapatır, 1'ler öteki bitleri korur. ^= biti çevirir, lamba sönükse onu yakar. = baytın tamamını değiştirir; |= ~(1 << 5) ise öteki bütün bitleri açar.

6. **-6**

   ~ int'in 32 bitinin hepsini çevirir; öndeki 0'lar da 1 olduğu için sonuç eksidir. ~x = −x − 1 olduğundan ~5 = −6.

7. **B** · 8

   union'ın alanları aynı adreste üst üste durur; boyu en büyük alanı, yani double kadardır: 8 bayt. 16, üç alanın toplamıdır (4 + 4 + 8); alanlar struct'ta yan yana dururdu.

8. **C** · 0D 0C 0B 0A

   Little endian bilgisayar, sayının en küçük basamaklı baytını (0D) en küçük adrese, yani bayt[0]'a koyar; ardından 0C, 0B ve 0A gelir. 0A 0B 0C 0D ise big endian sıradır.

10 Bağlı listeler

1. **B** · Veri ve sıradaki düğümün adresi

   Düğüm, veriyle sıradaki düğümün bellekteki adresinden oluşur; listenin sırasını bu adresler kurar. Listede indeks yoktur; ilk düğümün adresini de ayrı bir bas işaretçisi saklar.

2. **1000**

   Dizinin başında boş yer yoktur; yeni sayıya yer açmak için her eleman bir sağa kaymalıdır. n elemanda n kopya gerekir: 1.000 eleman için 1.000 kopya.

3. **D** · Yeri bilinen bir düğümden sonra yeni eleman eklemek

   Listede yeri bilinen bir düğümden sonra eklemek birkaç adres atamasıdır; dizide sağdaki bütün elemanlar kaydırılır. 500. elemana dizide tek adımda gidilir, listede baştan yürünür; liste ayrıca bir int için 16 bayt harcar ve düğümleri bellekte dağınıktır.

4. **60**

   p baştan başlar ve NULL olana kadar her düğümü bir kez gezer. t'ye sırayla 10, 20 ve 30 eklenir: 60.

5. **C** · Aranan sayı listede yoksa, liste bitince NULL'ın verisi okunmaya çalışılırdı.

   && önce sol tarafa bakar. Sayı listede yoksa p sonunda NULL olur ve soldaki p→veri ile NULL'ın verisi okunmaya çalışılır; oysa NULL bir düğüm değildir. Doğru sırada p != NULL önce sınanır, yanlış çıkınca sağ tarafa hiç bakılmaz.

6. **A** · 42 kendini gösterir; dolaşma ondan çıkamaz, 50 ve sonrasına ulaşılamaz.

   İlk satır 35'in tuttuğu 50 adresinin üstüne yeni düğümün adresini yazar; 50'nin adresi hiçbir yerde kalmaz. İkinci satır 35'in artık tuttuğu adresi, yani yeni düğümün kendi adresini yeniye verir. Doğrusu önce yeni→sonraki = p→sonraki, sonra p→sonraki = yeni yazmaktır.

7. **B** · free'den sonra bas→sonraki okunuyor; geri verilmiş belleğe dokunuluyor.

   free'den sonra düğüme dokunulmaz; geri verilmiş düğümün sonraki alanını okumak tanımsız davranıştır. Sıradaki düğümün adresi free'den önce bir değişkene alınmalıdır: Dugum *sonra = bas->sonraki; free(bas); bas = sonra;

8. **24**

   int 4 bayt, her adres 8 bayttır. Adresler 8'in katı bir yerden başlasın diye int'ten sonra 4 baytlık dolgu konur: 4 + 4 + 8 + 8 = 24. Örnek programda da sizeof(Dugum) = 24 çıkar.

11 Dosya işlemleri

1. **C** · "a"

   "a" (append) eskileri korur ve yeni yazılanları sona ekler. "w" dosya varsa içini baştan siler, "r" ve "rb" ise yalnız okumak içindir.

2. **A** · NULL döndürür; program bunu kontrol etmeden okumaya kalkarsa tanımsız davranış olur.

   fopen açamadığı dosya için hata vermez, NULL döndürür. NULL hiçbir yeri göstermez; onu dosya diye kullanmak tanımsız davranıştır ve Linux'ta program çöker. Bu yüzden her fopen'dan sonra if (f == NULL) kontrolü yapılır.

3. **D** · Satırlar programın belleğindeki tamponda bekliyordu; program çökünce tamponla birlikte kayboldu.

   C yazılanları önce programın belleğindeki tampona biriktirir; tampon dolunca ya da dosya kapanınca dosyaya gönderir. Çöken programda tamponu boşaltacak kimse kalmaz. Her satırdan sonra fflush çağrılsaydı satırlar dosyaya geçerdi. "a" kipi silmez, sona ekler.

4. **B** · NULL

   fgets dosya sonunda NULL döndürür; okuma döngüsü bu yüzden while (fgets(...) != NULL) diye yazılır. EOF'u döndüren fscanf'tir.

5. **3**

   argc komut satırındaki sözcüklerin sayısıdır ve programın adı da sayılır: argv[0] "./oku", argv[1] "a.txt", argv[2] "b.txt". Toplam 3 sözcük.

6. **6**

   Metin dosyasında sayı rakam rakam yazılır: 250000'de 6 rakam vardır, her rakam 1 bayt, toplam 6 bayt. Aynı sayı fwrite ile ikili dosyaya yazılsaydı int'in boyu kadar, 4 bayt olurdu.

7. **12**

   fscanf her turda bir ad ve bir sayı okur ve 2 döndürür; toplam'a 3, 5 ve 4 eklenir. Dosya bitince fscanf EOF döndürür, döngü durur ve 3 + 5 + 4 = 12 yazılır.

8. **C** · 10'un ilk baytı 0A, satır sonu baytıyla aynıdır; metin kipi onu 0D 0A'ya çevirir.

   10 bellekte 0A 00 00 00'dır. Windows metin kipinde satır sonu baytı 0A'yı 0D 0A'ya çevirir; dosyaya 5 bayt gider ve geri okunan sayı bozulur. İkili veri bu yüzden "wb" kipiyle yazılır.

12 Önişlemci ve kütüphaneler

1. **D** · Önişlemci, derleyici, çevirici, bağlayıcı

   Önce önişlemci # satırlarını işler (sicak.i), derleyici C'yi assembly diline çevirir (sicak.s), çevirici makine koduna çevirip nesne dosyasını yazar (sicak.o), en son bağlayıcı nesne dosyasını kütüphaneyle birleştirip çalışan dosyayı üretir.

2. **-E**

   -E seçeneği gcc'yi önişlemciden sonra durdurur: #include'lar yapıştırılmış, makrolar açılmış, yorumlar silinmiş C metni görülür. -S derleyiciden, -c çeviriciden sonra durur.

3. **5**

   Önişlemci x'in yerine 1+2'yi parantezsiz koyar: 1+2*1+2. Çarpma önce yapılır: 2 * 1 = 2, sonra 1 + 2 + 2 = 5. Beklenen 9, makro ((x)*(x)) diye yazılınca çıkar.

4. **B** · 16

   Satır 3 * (4)+(4) olur. Çarpma önce yapılır: 3 * 4 = 12, sonra 12 + 4 = 16. Beklenen 24 için bütün ifadenin de paranteze alınması gerekir: ((x)+(x)).

5. **A** · Önce başladı, sonra bitti

   -DHATA_AYIKLA adı komut satırından tanımlar; #ifdef ile #endif arasındaki satır derlenir ve önce "başladı" yazılır. -D olmadan derlenseydi o satır programa hiç girmez, yalnız "bitti" yazılırdı.

6. **C** · Bağlayıcı

   Derleyici prototipe güvenir ve dosyayı hatasız çevirir. Çağrılan fonksiyonun kodunu bütün nesne dosyalarında ve kütüphanelerde arayan bağlayıcıdır; bulamayınca hata verir ve çalışan dosya hiç oluşmaz.

7. **B** · Başlık bir dosyaya ikinci kez eklenince içeriği atlansın ve içindeki struct iki kez tanımlanmasın diye.

   İlk eklemede SICAKLIK_H tanımsızdır; içerik alınır ve ad tanımlanır. İkinci eklemede #ifndef'in sorusu yanlış çıkar ve önişlemci #endif'e kadar her şeyi atlar. Korumasız başlıkta aynı struct iki kez tanımlanır ve derleme durur.

8. **A** · Program sicaklik.dll'deki kodu dinamik kütüphaneden kullanıyor; işletim sistemi dosyayı açılışta yükleyemedi.

   Dinamik kütüphanenin kodu programa kopyalanmaz; programda yalnız kütüphanenin ve fonksiyonların adı yazılıdır. İşletim sistemi kütüphaneyi program açılırken yükler; dosya yoksa program hiç başlamaz. Statik kütüphanenin kodu programın içinde olduğu için böyle bir dosya gerekmez.

Öğrendiğini pekiştir

Bu notlardaki bütün soruları ve daha fazlasını sitede, her cevabın hemen ardından çözümüyle birlikte çözebilirsin.

hekzon.com/dersler

HEKZON