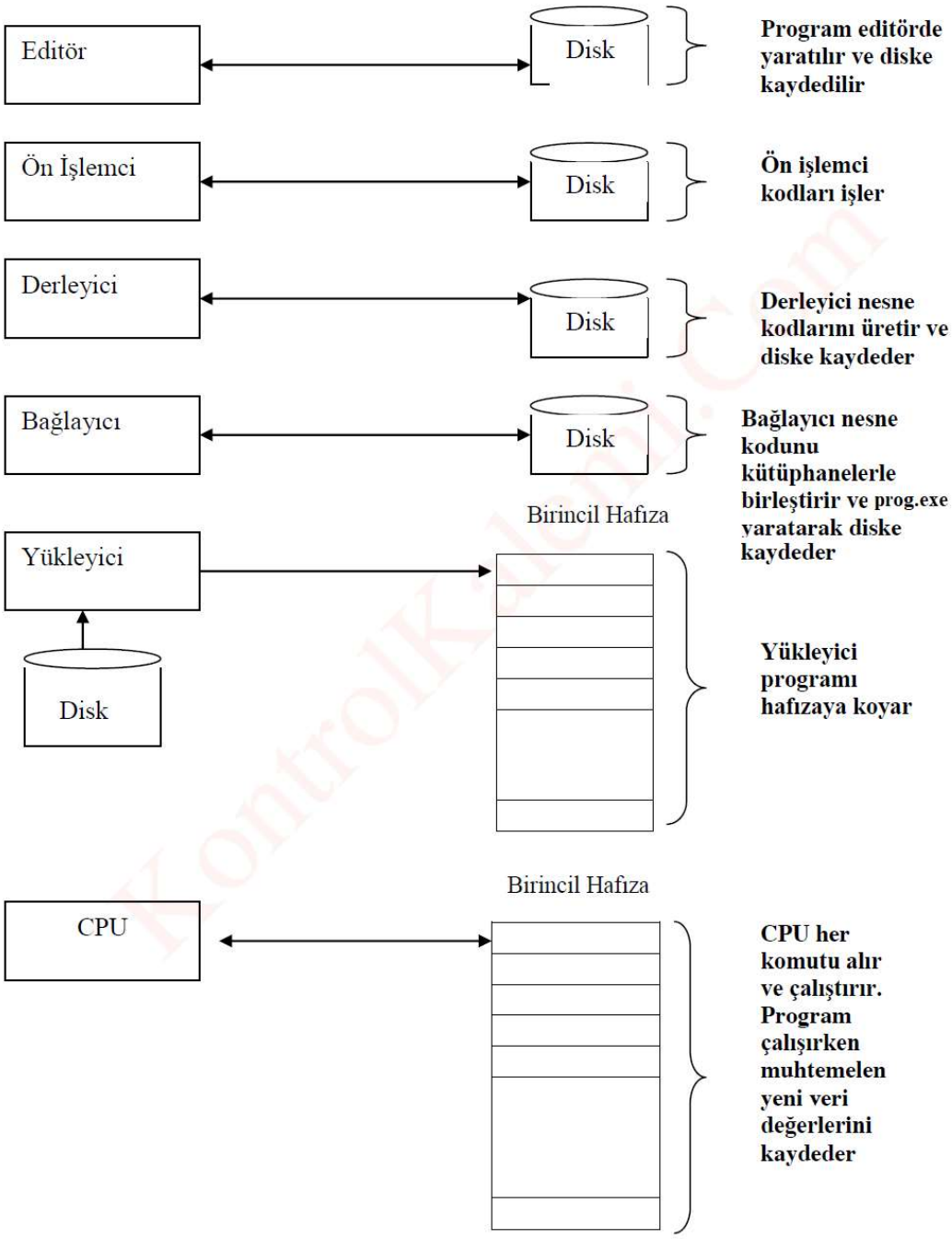
I. GİRİŞ
INTRODUCTION

Bilgisayar

- Bilgisayar; sayma ve hesaplama yapabilen, veri işleyebilen bir makinedir.
- Bilgisayar, bizim yaptığımız gibi akıl yürütemez.
- Bilgisayarları kullanarak bir iş yapmak için, işi yapmak için gereken bütün adımları (komutlar) ona söylememiz gerekir. Komutların listesine *program* denir.
- Bu komutlar, bir *programlama dili* kullanılarak bilgisayara verilir.
- Aslında, bir bilgisayar yalnızca ayrık (discrete) ikili sinyalleri, yani 0'ları ve 1'leri (bitler) anlayabilir. Bu yüzden, komutlar ikili sayı sistemindeki dizgiler (ikili dizgi = binary string) biçiminde yazılmalı veya bu dizgilere çevrilmelidir.
- Programlama dilleri aşağıdaki türlerden olabilir:
    1. Makine dili: Bilgisayarın, ana dilidir. Her komut ikili bir dizgidir.
    2. Yüksek düzey diller: İngilizceye benzeyen dillerdir, ör: C, Pascal, Fortran, Java.
- Her dilin kendi standartları ve önceden tanımlanmış komutları vardır. Çoğunda, başka kişiler tarafından kodlanmış program kitaplıkları (library) vardır.
- Yüksek düzey dillerde yazılmış komutlar, bir derleyici (compiler) tarafından makine diline çevrilir.
- Yüksek düzey bir dilde yazılan komutlardan *koşabilir* = *çalışabilir* = *executable* bir program oluşturmak için gereken bütün adımlar aşağıdaki şekilde gösterilmiştir.



Ön işlemci = Preprocessor

Derleyici = Compiler

Bağlayıcı = Linker

Yükleyici = Loader

Nesne kodu = Object code

- Yazılım geliştirme:

    - Problemi belirt.
    - Analiz et = Çözümle (girdi (input) ve çıktıları (output) belirt).
    - Bir algoritma tasarla, yani takip edilecek adımları söyle.
    - Adımları, bir programlama dili kullanarak gerçekle (implement).
    - Kaynak kodunu (source code) derle (compile) ve programı koş (execute).
    - Programın doğru çalışıp çalışmadığını test et.

Algoritma

- Bir problemin çözümü, bir dizi işlemin belirli bir sırada çalıştırılmasını içerir.

    - UYGULANACAK İŞLEMLER ve
    - BU İŞLEMLERİN HANGİ SIRADA UYGULANACAĞI

    algoritma olarak adlandırılır.

- Günlük hayattan bir algoritma örneği:
  Yataktan kalkıp işe gidene kadar, sırasıyla, yapılacak işlemler

    - Yataktan kalk
    - Pijamalarını çıkar
    - Duş al
    - Giyin
    - Kahvaltı yap
    - İşe doğru yola çık

Sözde kod (kaba kod)

- Sözde kodlar, bir programcının algoritma yazmada kullandığı yapay ve mantıksal dildir. Sözde kodlar, her gün konuştuğumuz dile oldukça yakındır. Bu kodları bilgisayarda çalıştıramayız. Ancak, bu kodlar programcıya programını yüksek düzey bir programlama diliyle yazmadan (gerçeklemeden) önce düşünme fırsatı verir.

- Sözde kod örnekleri:

    - *Eğer (if) öğrencinin notu 60'dan büyükse ya da 60'a eşitse*
      "Geçtiniz" yazdır

    - *Eğer (if) öğrencinin notu 60'dan büyükse ya da 60'a eşitse*
      "Geçtiniz" yazdır
      Aksi takdirde (else)
      "Kaldınız" yazdır

- *Eğer (if) öğrencinin notu 90'a eşit ya da 90'dan büyükse*
  "A" yazdır
  Aksi takdirde (else)
  Eğer (if) öğrencinin notu 80'e eşit ya da 80'den büyükse
  "B" yazdır
  Aksi takdirde (else)
  Eğer (if) öğrencinin notu 70'e eşit ya da 70'ten büyükse
  "C" yazdır
  Aksi takdirde (else)
  Eğer (if) öğrencinin notu 60'a eşit ya da 60'tan büyükse
  "D" yazdır
  Aksi takdirde (else)
  "F" yazdır

- *Alışveriş listesinde en az bir malzeme bulunduğu sürece (while)*
  Bir sonraki malzemeyi al ve bu malzemeyi listeden çıkart

- *"toplam" değişkenine 0 ata*
  "sayaç" değişkenine 0 ata
  "sayaç" 10'dan küçük iken (while)
  Diğer notu gir
  Girilen notu "toplam"a ekle
  "sayaç"a 1 ekle
  Sınıf ortalamasını "toplam"ı "sayaç"a bölerek bul
  Sınıf ortalamasını yazdır

Akış grafikleri

- Programlar, yalnızca 3 kontrol yapısıyla yazılabilir.
  - Sıra yapısı (art arda gelen komutlar)
  - Seçme yapısı (*if*)
  - Döngü yapısı (*while*)

- *Akış grafikleri*, algoritmanın ya da algoritmanın bir parçasının grafik gösterimidir. Bu grafikler genelde; dikdörtgen, çember, elmas gibi bazı özel kullanımlı sembollerden oluşur. Bu semboller, *akış çizgileri* ile birbirine bağlanır.

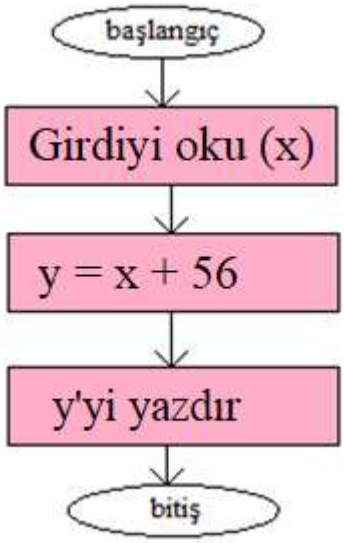
  Dikdörtgen sembolü veya *işlem sembolü,* girdi/çıktı veya herhangi bir işlemi belirtmek için kullanılır. İşlemin ne olduğu dikdörtgen kutunun içine yazılır.

  Elmas sembolü veya *karar işareti*, algoritmanın karşılık gelen anında bir karar verileceğini gösterir. Kararın nasıl verileceği (karar koşulu) elmas kutunun içine yazılır.

Akış grafiğiyle tam bir algoritma gösterilmek istenirse, grafiğin başına ve sonuna *oval* semboller yerleştirilir. İstenirse; baştaki oval sembolün içine "başlangıç", sondaki oval sembolün içine de "bitiş" yazılarak grafik daha anlaşılır hale getirilebilir.

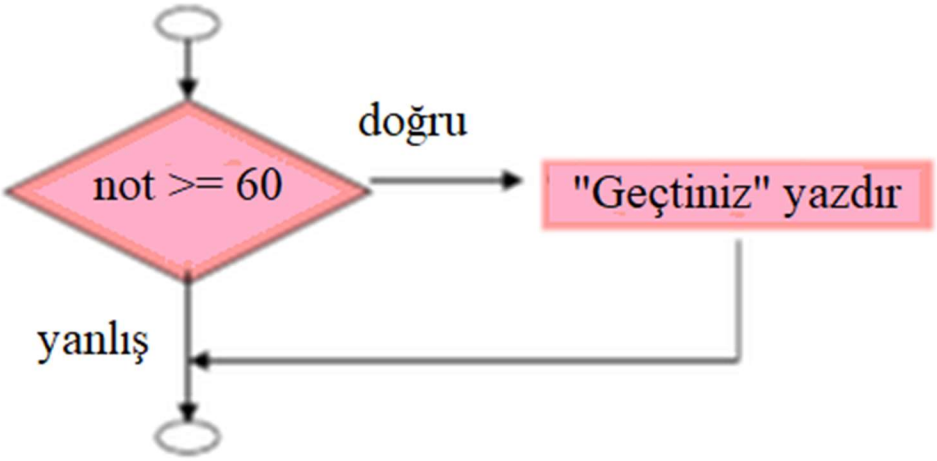
Akış çizgilerindeki okların yönleri, algoritmadaki komutların hangi sırayla aktığını gösterir.

Programlardaki sıra yapısının akış grafiği basit bir örnekle aşağıda gösteriliyor:

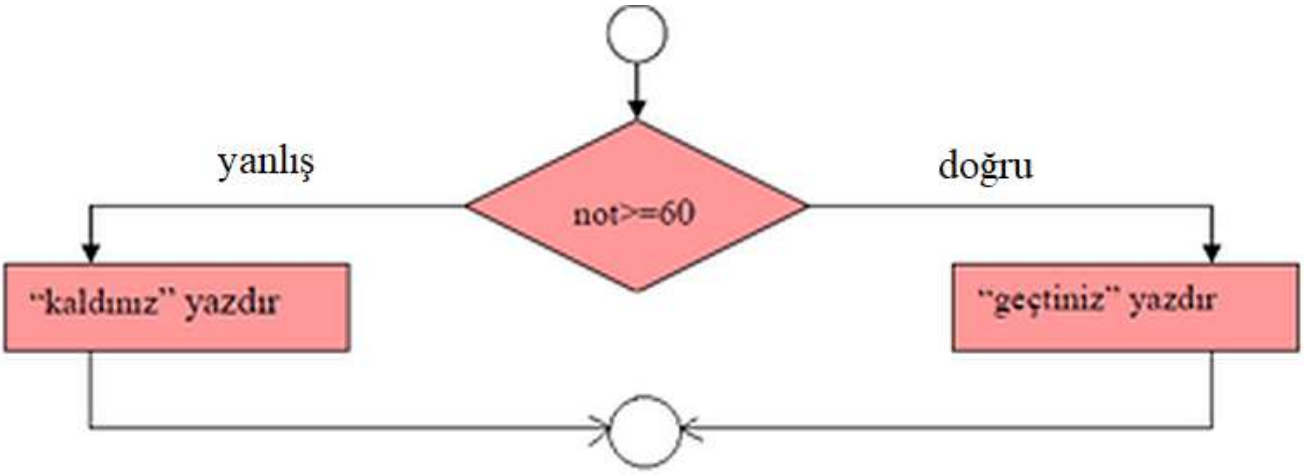


Şimdi, aşağıda gördüğümüz 3 akış grafiğini anlamlandıralım.

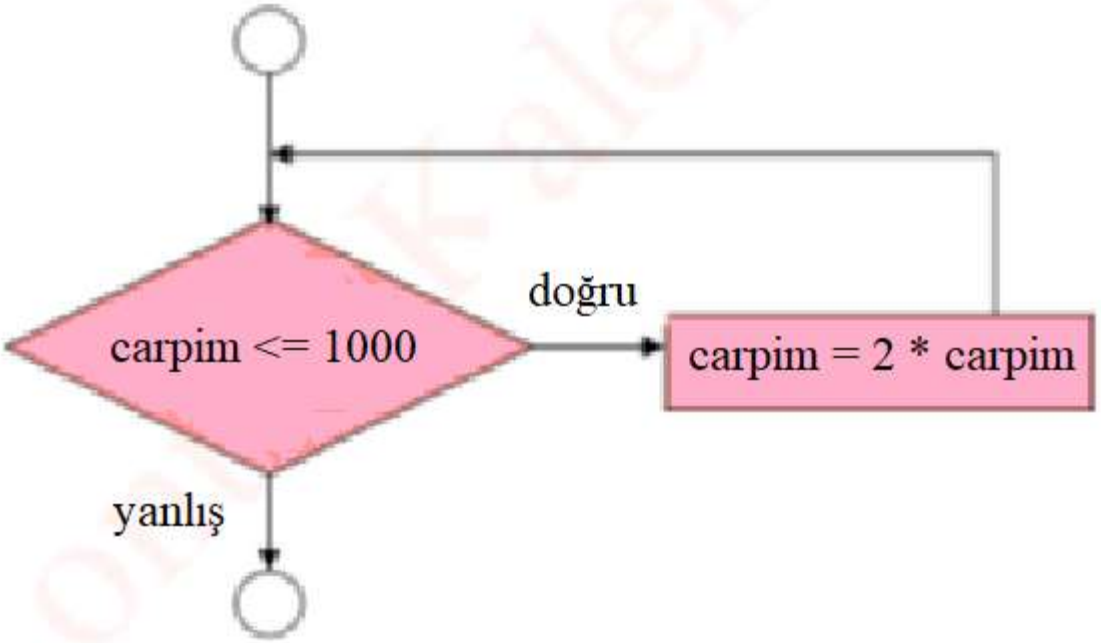
Bir yollu karar yapısı (*if*)



İki yollu karar yapısı (*if/else*)



Yineleme yapısı (*while*)



Yapısal programlama

- *Yapısal programlama***,** bir bilgisayar programının
    1. açıklığını = berraklığını = anlaşılırlığını
    2. kalitesini = niteliğini
    3. geliştirme zamanını (yazmak ve üretmek için harcanan zaman)

ilerletmek (daha iyi hale getirmek) için, aşağıdaki yapısal kontrol akışı unsurlarını kullanır:

    1. seçim (*if/else*)
    2. yineleme = döngü (*while, for*)
    3. blok yapıları
    4. altprogramlar = alt yordamlar (fonksiyonlar)

- Yapısal programlamada, basit testlere ve buna bağlı olarak C dilindeki "*goto*" deyiminin yaptığı gibi kod içinde sıçramalara izin verilmez. Sıçramalar, potansiyel olarak, takip edilmesi ve bakımının yapılması zor olan karışık program kodlarına yol açar. Hata ayıklama ve düzeltme, olanaksız hale gelebilir.

C diline giriş

Örnek 1:

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

- Yukarıdaki program ekrana "Merhaba dunya!" mesajını yazdırmaktadır.

- C programları bir veya daha fazla fonksiyondan oluşur (programlama blokları). Fonksiyonlar, yapılacak işlemleri belirten deyimler (statement) içerir. Her C programı `main` adında bir fonksiyon içerir. C'de kullanılan; `main`, `if`, `while`, `for`, `int`, `char`, … gibi bazı sözcükler özel amaçlar için saklanmıştır.

- C, büyük/küçük harfe duyarlı bir dildir. Örneğin, `main` ve `mAIN` aynı değildir.

- Küme parantezleri `{ }` bir bloğun başını ve sonunu gösterir. Bloklar (birleşik deyimler) deyimleri gruplamaya yarar. Verdiğimiz örnekte, küme parantezleri `main` fonksiyonunun başını ve sonunu işaretliyor.

- Koşulabilir (executable) bir C programının kaynak kodu mutlaka bir `main` fonksiyonu bulundurmalıdır.

- `printf("Merhaba dunya!\n");` çift tırnak işaretleri arasında kalan metni ekranda göstermektedir. Her deyim noktalı virgülle biter. Yeni satır (newline), durak (tab), vb. gibi bazı karakterler sola yatık bölü çizgisi (\) ve ardından gelen başka özel karakterlerle gösterilmektedir. Örneğin, `\n` yeni satır karakteridir, `\t` yatay durak karakteridir, vb. `\` kaçış karakteri olarak adlandırılır.

- `#include <...>`, derleyiciye önceden tanımlanmış bir program kitaplığını (library) programın kaynak kodunda bulundurmasını söyler. Standart kitaplıkların birinde tanımlanmış bir fonksiyon kullanmak istersek, kitaplığın ismini < ve > arasında belirtmeliyiz.

- Örneğimizde, `#include <stdio.h>` stdio.h kitaplığını kodda bulunduruyor. Bu kitaplıkta, standart girdi/çıktı (ekran, klavye, dosya sistemi, vb.) fonksiyonları bulunur. Örneğin, `printf()`, `scanf()`, `getchar()`, vb. Örnekte, programın girdi/çıktı araçlarıyla etkileşebilmesi için `<stdio.h>` kitaplığına ihtiyacımız var (`printf()` bu kitaplıkta tanımlanmıştır).

- C dilinde fonksiyonlar, kendilerini çağıran ortama (başka fonksiyonlar, işletim sistemi, vb.) bir değer döner. Örnekteki `main` fonksiyonunun başındaki `int` sözcüğü, fonksiyonun tamsayı tipinde bir değer döndüğünü belirtiyor. `main` bloğunun sonunda yer alan `return 0;` deyimi, 0 tamsayı değerinin dönülmesini sağlıyor. Örnekte, dönülen değerin bir önemi bulunmamakla beraber (istersek 149, 3, -95, vs. de dönebilirdik), C dili fonksiyon (altprogram) tabanlı yapısal bir dil olduğu için, genel durumda fonksiyonların döndüğü değerler önemlidir.

Örnek 2:

```
#include <stdio.h>
/*Bu iki tamsayiyi toplayan programin kaynak kodudur*/

int main() {
      int sayi1, sayi2, toplam;
      printf("Sayilari girin: ");
      scanf("%d%d",&sayi1,&sayi2);
      toplam = sayi1 + sayi2;
      printf("Toplamin degeri %d'dir.\n", toplam);
      return 0;
}
```

- `/* … */` bir yorumu göstermektedir. `/*` ve `*/` arasındaki her şey, C derleyicisi tarafından görmezden gelinir. Yani makine diline çevrilmezler. Yalnızca; okunabilirlik, belgeleme ve kodu okuyanlar için bilgi amaçlıdırlar. Yorumlar `/*` ile başlamalı ve `*/` ile bitmelidir (`*` ve `/` arasında boşluk olmamalıdır). Yorumlar, aşağıdaki gibi, birden fazla satıra da dağıtılabilir.

```
/*Bu iki tams
ayiyi toplayan
programin kaynak kodudur*/
```

- `;` deyimlerin sonunu gösterir. Bir satırda birden çok deyim olabileceği gibi, bir deyim birden çok satırda bölünmüş de olabilir. Aşağıdakilerin tümü geçerli deyimlerdir:

```
int sayi1,
sayi2, toplam; printf("Sayilari girin: ");
scanf("%d%d",
&sayi1,
&sayi2);
toplam = sayi1 +
sayi2;printf("Toplamin degeri %d'dir.\n", toplam);
return 0;
```

- Boşluk, durak, yeni satır gibi karakterler derleyici tarafından göz ardı edilir. Bu yüzden; bu karakterleri deyimler arasına, virgülden sonra, artı işaretinden önce, vb. koyabiliriz. Ancak; `printf`, `sum` gibi sözcülerin karakterleri arasına, çift tırnakların arasındaki metnin karakterleri arasına, vb. koyamayız. Aşağıdaki örneklerin tümü yanlıştır:

```
in                                  /* int bolunemez */
t sayi1, sayi2, toplam;
prin                                /* printf bolunemez */
tf("Sayilari girin: ");
scanf("%d%d",&sa                    /* Degisken adı sayi1 bolunemez */
yi1,&sayi2);
toplam = sayi1 + sayi2;
printf("Toplamin de                 /* Cift tirnaklar arasi bolunemez*/
geri %d'dir.\n", toplam);ret        /* return bolunemez */
urn 0;
```

- Örnek 2'de; birinci ve ikinci sayının toplamını hesaplamak için bu sayıların değerlerini ve toplam değeri ekrana yazdırmak için o değeri akılda (bilgisayarın belleğinde) tutmalıyız. Bellekte, bellek hücreleri (memory cells) denen bir dizi sıralı depolama birimleri vardır. Her bellek hücresinin, kendisiyle ilgili tek bir adresi vardır. Bellek hücreleri, daha küçük birimlerin (bayt = byte) topluluklarıdır. Örneğin, tamsayı değerleri tutabilen bir bellek hücresi 4 bayttan oluşabilir.
- `int sayi1, sayi2, toplam`; bir bildirim deyimidir (declaration statement). `sayi1, sayi2` ve `toplam`'ın her birine *değişken* (variable) denir. Değişkenler, hesaplamalar sırasında değerleri tutmak için kullanılır. Değişkenlerin değerlerini tutmak için, bellekte yer ayırmak gerekir. Bildirim deyiminde, her değişken adı için bir bellek hücresi ayrılır. Bir değişkene erişmek istediğimizde, adıyla çağırırız ve program ona karşılık gelen bellek adresini bilir. Değişkenleri, kullanmadan önce mutlaka tanımlamak gerekir.
- Tanımladığımız her değişken için değişken tipini belirtmeliyiz. Örneğimizdeki `int` tamsayı (integer) veri tipini gösteriyor (başka veri tipleri de var). Bildirimde, önce veri tipini ve daha sonra tek bir değişken adını veya birden çok değişken adının listesini belirtmek gerekir.

Listedeki değişkenler virgüllerle ayrılmalıdır. Aşağıdaki bildirimlerin hepsi doğrudur. Ayrıca, her bildirimin noktalı virgül (;) ile bittiğine de dikkat edelim.

```
int a, b, toplam;
int d;
int bir,
iki, uc;
```

- `scanf("%d%d",&sayi1,&sayi2);` standart girdi aracından (klavye) girdi (input) okur. Program bu deyimi koşarken, kullanıcıdan değer girip bu değeri bilgisayarın belleğine göndermek için "enter" tuşuna basmasını bekler. Eğer `scanf()` fonksiyonunu kullanacaksak, programımıza `<stdio.h>` kitaplığını dahil etmeliyiz.
- `%d` (dönüşüm belirteci), belirtilen değişkene tamsayı bir değerin okunacağını gösteriyor. Örnekte, iki tane `%d` var. Bu yüzden, çift tırnaktan sonra belirtilen ve virgüllerle ayrılan `sayi1` ve `sayi2` değişkenlerine iki değer okunacaktır.
- `&` adres işlecidir (adress operator). Bir değişkene uygulandığında, o değişkenin bellek adresi elde edilir. `scanf()` fonksiyonunda bir değişken adından önce kullanılması gerektir. `scanf("%d%d",&sayi1,&sayi2);` ile yapılan işin aynısı aşağıdaki deyimlerle de yapılabilir:

  ```
  scanf("%d",&sayi1);
  scanf("%d",&sayi2);
  ```

- `toplam = sayi1 + sayi2;` bir atama deyimidir (assignment statement). Önce = işaretinin sağ tarafı hesaplanır, sonra da sağ tarafın hesaplanan değeri sol tarafa atanır. Örnekte, `sayi1` ve `sayi2` değişkenlerinin değerleri toplanır ve işlemin sonucu (bulunan değer) `toplam` değişkeninde depolanır.
- `printf("Toplamin degeri %d'dir.\n", toplam);` çift tırnaklar arasındaki metni ekranda gösterir. Bunun yanında, `%d` ekrana tamsayı bir değerin yazıldığını gösterir. Örneğimizde, `toplam` değişkeninin değeri ekrana yazılmaktadır. Ekrana yazılan değerler; sabitler, değişkenler, sabitlerin ve değişkenlerin kombinasyonu da olabilir. Bir değerden fazlası yazılabilir. Aşağıdaki örneklerin tümü geçerlidir:

  ```
  printf("Toplamin degeri %d'dir.", toplam);
  printf("Toplamin degeri %d'dir.", 3);
  printf("Toplamin degeri %d'dir.", a + b);
  printf("Toplamin degeri %d'dir.", toplam + a * 3);
  printf("%d ve %d toplandiginda %d elde ederiz.", a, B, a+B);
  printf("9 ile %d'yi topla ve %d elde et.", sayi, sayi+9);
  ```

- C'de her fonksiyon aksi belirtilmediği zaman mutlaka bir değer dönmelidir. Dönülen değerler, fonksiyonu çağırana bilgi verir. Örnekte, `return 0;` deyimi 0 değerini dönme işini yapar.

Veri tipleri (Data types)

C programlama dili bazı temel veri tiplerini sunmaktadır.

Tamsayılar (integers):

Büyüklüğü farklı olan tamsayı tipleri vardır. Her birinin en az olan büyüklüğünü ANSI standardı belirler, ama tam büyüklük bilgisayara bağlıdır. Büyüklüğe göre 3 şekilde sınıflandırılabilir:

1. `short int` (en az 16 bit)
2. `int` (`short` veya `long`)
3. `long int` (en az 32 bit)

Aşağıdaki gibi de sınıflandırılabilirler:

1. `signed int`: hem pozitif hem negatif değerler tutulabilir. `int` kullanıldığında, bunun işaretli (signed) `signed int` olduğu varsayılır.
2. `unsigned int`: sadece negatif olmayan sayılar tutulabilir. Bir tamsayının gösteriminde, ilk bit, işaret biti olmak yerine sayının parçası olarak kullanılır.

Aşağıdakilere dikkat edelim:

- `short int, long int` vb.de `int` kullanılmayabilir.
- Veri tiplerinin sınırları olduğunu anlamaya dikkat edelim. Eğer taşma varsa, yani bir sayı değeri belleğe sığmak için çok büyükse, ANSI standardının belirttiğine göre davranış tanımsızdır ve her şey olabilir. Bilgisayarımızda, tamsayıların 16 bitlik bellek hücrelerinde tutulduğunu varsayalım. Yani, -32768 ve 32767 arasındaki tamsayılar tutulabiliyor olsun. Tipi `int` olan bir değişkende 43567 değerini tutmak isteyelim. Bu durumda taşma vardır. Söylenen değeri tutmak için değişkenimizi `long` olarak tanımlayabiliriz. Öte yandan, `long` değişkenleri kullanan aritmetik işlemler `int` değişkenleri kullananlardan daha çok zaman alır.
- Her farklı veri tipinin dönüşüm belirteci farklıdır. `short, int` ve `long` için, sırasıyla, `%hd, %d` ve `%ld` kullanmak gerekir. Değişkenin tipi, karşılık gelen dönüşüm belirtecine uymalıdır, aksi halde tanımlanmamış davranış oluşur.
- İşaretsiz (`unsigned`) sabit değerler için `u` veya `U` kullanılır. `long` sabit değerler için `l` veya `L` kullanılır. Örneğin, `300Ul`, bir işaretsiz (`unsigned`) `long` sabittir.
- C'de ayrıca octal (8 tabanında) ve hexadecimal (16 tabanında) gösterimler vardır. Octal gösterimde, rakamlar 0 ve 7 arasındadır. Hexadecimal gösterim için 0, …9, A,…,F, veya 0,…9, a,…,f, rakam olarak kullanılır.

Kayan nokta sayılar (floating point numbers):

Gerçek sayılardır (yani kesirli bölümü olan sayılar). Farklı kayan nokta sayı tiplerinin büyüklüğü farklıdır ve bilgisayara göre değişir. Tamsayılar gibi, ANSI standartlarına göre, kabul edilir en az büyüklükleri vardır. Anlamlı basamaklarda kayan nokta hatalarının olabileceğini dikkate almalıyız. Farklı kayan nokta tipleri, boyları, alabileceği değerler, dönüşüm belirteçleri (DöBe), hassasiyetleri ve sabit örnekleri aşağıdaki tabloda belirtilmiştir:

	Boy	Alabileceği değerler	DöBe	Hassasiyet	Sabitler
`float`	4 bayt	1.2E-38 … 3.4E+38	`%f`	6 ondalık basamak	3.7, 3.7f, 3.7F, 1.4e2, 1.4E-2
`double`	8 bayt	2.3E-308 … 1.7E+308	`%lf`	15 ondalık basamak	3.7, 1.4e2, 1.4E-2
`long double`	10 bayt	3.4E-4932 … 1.1E+4932	`%LG`	19 ondalık basamak	3.7, 3.7l, 3.7L, 1.4e2, 1.4E-2

Karakterler (characters):

Bir karakter, tek bir bayta karşılık gelir. `char veri;` karakter veri tipine sahip bir değişkeni tanımlayan bildirim deyimidir. Bir karakterin dönüşüm belirteci `%c`'dir.

Karakter sabitleri tek tırnaklar arasında gösterilir, örneğin `'a', '9', 'B', '+', '&'`, vb. Geçiş dizisi (escape sequence) kullanılarak gösterilebilen bazı karakterler de vardır: `'\n'` (yeni satır), `'\t'` (yatay durak), `'\\'` (sola yatık bölü), `'\''` (tek tırnak), vb. Bu gösterimlerin, yalnızca bir karaktere karşılık geldiğine dikkat edelim.

Makinelerde, yerel bir karakter kümesi ve bu kümede karakter sabitleri vardır. Karakter sabitleri, herhangi bir karaktere karşılık gelen (o karakteri kodlayan) tamsayı tipindeki sayı değerleridir. En yaygın karakter kodlama standardı ASCII tablosudur. Her karakter tek bir değere sahiptir. Örneğin, `'0'`, ASCII karakter tablosunda 48 değerine sahiptir. Karakter değerleri -128 ve 127 arasındadır. Yazdırılabilen karakterlerin değeri her zaman pozitiftir. Bir karakter değişkenini `unsigned char` olarak tanımlanırsa, tutulabilen değerler 0 ve 255 arasındadır.

Ayrıca, sıfır veya daha çok karakterden oluşan ve başı ve sonunda çift tırnak (") bulunan dizgi sabitleri vardır. Örnek: `"Bu bir dizgidir", "Merhaba dunya", ""` (boş dizgi), vb.

Değişkenler ve sabitler (variables and constants)

Değişken bildirimi (variable declaration):

Değişkenler, bellekte değer tutmak için kullanılan veri nesneleridir. Bir değişken aşağıdaki sözdizimi (sentaks = syntax) ile bildirilir:

veriTipi degiskenAdiListesi;

- Her değişkenin bir veri tipi ve tek bir adı olmalıdır, örneğin, aynı kapsamda aynı ada sahip farklı değişkenler tanımlanamaz.
- Bir değişken kullanılmadan önce tanımlanmalıdır.
- Değişkenler, herhangi bir bloğun yalnız başında tanımlanabilir.
- Tek bir bildirim deyiminde, virgüllerle ayrılmış birden çok değişken adı olabilir. Tek bir bildirim deyiminde aynı tipten değişkenler tanımlanabilir.
- Bir değişken tanımlandığında, değeri çöptür (değerin ne olduğunu bilemeyiz). Ancak; bir değişkeni tanımladığınızda ona ilk değer atamak mümkündür. İlk değerin, bir sabit olması gerekir.
- `printf()` ve `scanf()` fonksiyonlarında, değişkenlerin tipi karşılık gelen dönüşüm belirtecine uygun olmalıdır, aksi halde tanımlanmamış davranış görülür.
- `x ≠ 'x' ≠ "x"` (`x`'in tamsayı olarak tanımlandığını varsayalım)
  - `x`: bir değişken adıdır
  - `'x'`: x karakteridir
  - `"x"`: x dizgisidir (tek karakterden oluşan dizgi)

- `9 ≠ '9' ≠ "9"`
  - `9`: tamsayı sabiti olan 9'dur
  - `'9'`: 9 karakteridir
  - `"9"`: 9 dizgisidir (tek karakterden oluşan dizgi)

Değişken örnekleri:

```
int sayi1 = 0, sayi2, toplam;

float Sayi1;

long double vergi = 4.3;

double saniye;

long faktoriyel = 40L;

char adim = 'a', t;

char birinci = '\n';

double SAyi1 = 3.4, SAyi2 = 5.6, SAyi3, SAyi4 = -132;

float sonuncu;


scanf("%d%lf%ld%c", &sayi1, &ikinci, &faktoriyel, &birinci);

scanf("%f", &Sayi1);

printf("Farkli sayilar: %d\t%ld\t%Lf\n", sayi2, faktoriyel, vergi);

printf("Daha fazlasi, %c %c", adim, birinci);
```

Değişken adlandırma kuralları:

Burada söylenecek kurallar, aynı zamanda; kimlik tanıtıcı adlar (identifier names) için de geçerlidir. Kimlik tanıtıcı adlara örnekler: değişken adları, fonksiyon adları, sembolik sabit adları, vs.

1. Harflerden ve rakamlardan oluşmalıdır (altçizgi '_' karakteri harflerden sayılır). Değişken adlarının anlamlı olması gerekmez.
2. Değişken adları, bir harf ile başlamalıdır. Altçizgi karakteri ile başlamak da mümkündür, ama önerilmez çünkü program kitaplılarındaki fonksiyonlar sıklıkla bu türden adlandırılır.
3. C'nin küçük/büyük harfe duyarlı olduğunu unutmayalım. Örneğin, `x` ve `X` farklıdır.
4. Bir adın en az 31 karakteri önemlidir (derleyiciye göre değişir).
5. `int, main, float, return, long, unsigned, const`, vs. gibi anahtar sözcükler saklıdır ve kimlik tanıtıcı adlar olarak kullanılamaz.

İşte bazı geçerli kimlik tanıtıcı adlar:

```
Benim_Adim8
sayi_
_sayi
BLM
```

```
Fdjkfhdg34ffs
x9875424
```

Aşağıdakiler, geçerli olmayan kimlik tanıtıcı adlardır:

```
8Ad_4
4563
sayi+
veri\%
return
```

Sabitler (constants):

Sabit olarak tanımlanan değerlerdir, programın çalışması sırasında değerleri değiştirilemez. Sabitler iki farklı şekilde tanımlanabilir:

1. Saklı sözcük olan `const` kullanarak. Bildirim kısmında tanımlanmalıdır, sabit için bellekte yer ayrılır.

   const tipAdi degiskenAdi = sabitDeger;

   ```
   const double PI = 3.14;
   ```

2. `define` kullanarak. Bir yerine koymadır, bellekte yer ayrılmaz.

   ```
   #define PI 3.14
   ```

   `PI`, `long double` olarak tanımlanmak istenirse, aşağıdaki gibi tanımlanır:

   ```
   #define PI 3.14L
   ```

Sayım sabitleri (enumeration constants):

Sabit tamsayı değerlerinin bir listesini tanımlamak içindir. Sabit karakter değerlerinin aslında küçük tamsayı değerler olduğunu hatırlayalım. Birden çok `define` için bir alternatiftir.

```
enum Ay{OCAK = 1, SUBAT, MART};
enum BOOLEAN{hayir, evet};
enum sayilarim{uc = 3, bir, DORT = 2, bes};
enum ozelKarakterler{TAB = '\t', YENISATIR = '\n', GERIYEYATIKBOLU = '\\'};
```

- Tamsayılar gibi davranırlar.
- Başka şekilde belirtilmezse, sayım sabitlerinin değerleri 0 ile başlar. Belirtilmeyen değerler, en son belirtileni takip eder.
- Farklı sayımlardaki adlar farklı olmalıdır.
- Aynı ya da farklı sayımlardaki değerler farklı olmak zorunda değildir.

Örnek

```
enum boolean {FALSE, TRUE};
int main (){
      int a;
      a = FALSE;
      return 0;
}
```

Örnek

```
enum boolean {FALSE, TRUE};
int main (){
      enum boolean b;
      b = 9;                  /* uyari verir. */
      b = TRUE;
      return 0;
}
```

Örnek

```
int main (){
      enum boolean {FALSE, TRUE};
      int a;
      enum boolean b;
      a = FALSE;
      b = 9;                  /* uyari verir. */
      b = TRUE;
      return 0;
}
```

Atama ve tip dönüşümleri (assignment and type conversions):

Dar olan bir tipteki değer, geniş olan bir tipe (değişkene) atandığı zaman, değer bilgi kaybı olmadan atanır. Geniş olan tipteki değer, dar olan tipe atanırsa, bilgi kaybı olabilir.

```
int a, float b, char c;
a = 3.14;         /* a = 3 */
b = 8;            /* b = 8.0 */
c = '0';          /* c = '0' */
b = 9.2;          /* b = 9.2 */
a = b;            /* a = 9 */
b = a;            /* b = 9.0 */
a = c;            /* a = 48 ('0'ın ASCII karakter kodu 48 olduğu için) */
b = c;            /* b = 48.0 */
c = 49;           /* c = '1' ('1'in ASCII karakter kodu 49 olduğu için) */
c = 50.2;         /* c = '2' */
```

İşleçler (operators)

Atama işleci:

`x = a + y;`

`x = x + y;` veya `x += y;`

`x *= y + 1` eşittir `x = x * (y + 1);`

Aritmetik işleçler:

- `+, -, *, /` → bütün veri tiplerine uygulanabilir (karakter tipine de)
- `%` → mod işleci, tamsayılara uygulanabilir
- `/` → tamsayı bölmesi (iki işlenen (operand) de tamsayıysa kesir olan parçayı çöpe atar)
- Bir işlecin farklı tipte işlenenlere uygulanıyorsa, dar olan işlenen geniş olana dönüştürülür

Örnek

```
float x; int a, b;
x = 5/2;                        /* x = 2.0 */
x = 5.0 / 2;                    /* x = 2.5 */
a = 5.0 / 2;                    /* a = 2 */
a = 5 / 2 + 3 / 2;              /* a = 3 */
b = 5 / 2.0 + 3 / 2;            /* b = 3 */
b = 5 / 2.0 + 3.0 / 2;          /* b = 4 */
x = a / b;                      /* x = 0.0 */
b = b + a;                      /* b = 7 */
```

Örnek

```
int c = 5002;
char a = '0';           /* ASCII kodu 48 */
a = a + 3;              /* a = '3' *, ASCII kodu 51/
a += c;                 /* a = ?, tasma ve tanimlanamayan davranis*/
```

Örnek

```
int a;
char b = 'A';
a = '9' - '2';          /* a = 7 */
a = 'Z' - b;            /* a = 25 */
b = 'c' + 4             /* b = 'g' */
```

Tipe zorlama (casting)

Doğrudan tip dönüşümleri (tipe dönüştürme) zorlanabilir.

(tipAdi) ifade

Örnek

İki tamsayımız olduğunu, bunların ortalamalarını ve oranlarını hesaplamak istediğimizi varsayalım.

```
int x = 5, y = 4;
float ortalama, oran;
ortalama = (x + y) / 2;                  /* ortalama = 4.0 */
ortalama = (x + y) / 2.0;                /* ortalama = 4.5 */
ortalama = ((float)x + y) / 2;           /* ortalama = 4.5 */
ortalama = (float)(x + y) / 2;           /* ortalama = 4.5 */
ortalama = (float)((x + y) / 2);         /* ortalama = 4.0 */

oran = x / y;                        /* oran = 1.0  */
oran = (float)(x / y);               /* oran = 1.0  */
oran = (float)x / y;                 /* oran = 1.25, x'in degeri degismez  */
oran = (x + 0.0) / y;                /* oran = 1.25  */
```

Artırma ve azaltma işleçleri (increment and decrement operators):

`++` artırma ve `--` azaltma işleçleridir. `++` işlenenine 1 ekler ve `--` işleneninden 1 çıkarır. Öntakı (prefix) veya arttakı (postfix) işleçler olarak kullanılabilirler.

`n++` (önce `n`'nin değerini kullan, sonra `n`'nin değerini 1 artır)

`n--` (önce `n`'nin değerini kullan, sonra `n`'nin değerini 1 azalt)

`++n` (önce `n`'nin değerini 1 artır, sonra `n`'nin değerini kullan)

`--n` (önce `n`'nin değerini 1 azalt, sonra `n`'nin değerini kullan)

Örnek

```
int n = 5, x;
n++;              /* n = 6 */
x = n++;          /* x = 6, n = 7 */
x = n--;          /* x = 7, n = 6 */
```

Örnek

```
int n = 5, x;
++n;               /* n = 6 */
x = ++n;           /* x = 7, n = 7 */
x = --n;           /* x = 6, n = 6 */
```

Örnek

```
int a, b, x;
b = 3++;     /* olmaz */
++(x = 3);   /* x = 4 */
++x = 3;     /* x = 3 */
a = 3;
++a += 2     /* a = 6 */
a++ += 2;    /* olmaz, hata verir */
```

İlişkisel ve mantıksal işleçler (relational and logical operators):

Değişkenlerin değerini, sabitleri ve bunlardan oluşan ifadeleri karşılaştıran ilişkisel işleçler bulunur:

`>` : büyüktür

`>=` : büyük eşittir

`<` : küçüktür,

`<=` : küçük eşittir

`==` : eşittir

`!=` : eşit değildir

Eşitlik ve eşitsizlik işleçlerini kayan nokta sayılarla kullanırken dikkatli olmak gerekir. Beklenen sonuçlar alınmayabilir.

İfadeler, mantıksal işleçlerle bağlanabilir:

`&&` (VE = AND)

`||` (VEYA = OR)

`!` (DEĞİL = NOT)

Eğer mantıksal bir ifade doğruysa, ifadenin sayısal değeri 1 olur. Yanlışsa 0 olur.

Mantıksal işleçlerin, doğruluk tablosu aşağıdadır:

a	b	a && b	a || b	!a
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Bitsel işleçler (bitwise operators):

&	bitsel VE (bitwise AND)
|	bitsel VEYA (bitwise OR)
^	bitsel DIŞLAYICI VEYA (bitwise XOR)
<<	bitsel sola kaydırma (bitwise left shift)
>>	bitsel sağa kaydırma (bitwise right shift)
~	bitsel ters/değilleme (bitwise 1's complement)

Örnek

```
int a, b, c;
a = 10; b = 86;              /* a = 1010, b = 1010110 */
c = a & b;                   /* c = 0000010 = 2 */
c = a | b;                   /* c = 1011110 = 94 */
c = a ^ b;                   /* c = 1011100 = 92 */
c = b << 1;                  /* c = 10101100 = 172  (c = b*2) */
c = b << 2;                  /* c =101011000 = 344 (c = b*2*2) */
c = b >> 1;                  /* c = 101011 = 43 (c = b/2) */
c = b >> 2;                  /* c = 10101 = 21 (c = (b/2)/2) */
c = ~b;                      /* c = 101001 = -87 */
```

İşleçlerin öncelikleri (precedence of operators):

Aşağıdaki tabloda öncelikler azalan sırada verilmiştir. Eğer öncelik konusunda şüpheye düşülürse, parantezleri kullanmak işe yarar:

`( )` parantezler	içteki önce, soldan sağa
`!, ++, --, +` (tekli), `-` (tekli)	sağdan sola
`*, /, %`	soldan sağa
`+, -`	soldan sağa
`<, <=, >, >=`	soldan sağa
`==, !=`	soldan sağa
`&&`	soldan sağa
`||`	soldan sağa
`=, +=, -=, *=, /=, %=`	atama işleçleri, soldan sağa

Örnek

```
a = 4; b = 5; c = 7;
(a < b || b > 7 ) && c > = a  /* 1 verir, yani doğru (TRUE) */
a < b || b > 7  && c > = a    /* 1 verir, yani doğru (TRUE) */
!(a > b) && c > = a           /* 1 verir, yani doğru (TRUE) */
!a > b && c >= a              /* 0 verir, yani yanlış (FALSE) */
```

Girdi/Çıktı (Input/Output = I/O) işlemleri

`printf()` fonksiyonu

```
printf(bicimlendirmeliDizgi, degisken1, degisken2, ...);
printf("XXX555 \ndersindeyiz.");          /* " ve " arasindaki dizgiyi yazdirir*/
printf("Sayi %d'dir\n", a);               /*a değişkeninin değerini yazdirir*/
printf("Toplam %f'dir.", 3+2.0);          /*Islemin sonucunu yazdirir*/
```

- `%` argümanların yerini tutar (nereye yazılmaları gerektiğini söyler)
- Dönüşüm belirteçleri, yazdırılacak değer tipleriyle uyuşmalıdır. Aksi halde, tanımlanmamış davranış olur. Farklı veri tiplerinden değişkenler/değerler kullanmaya ihtiyaç duyulursa, tipe zorlama kullanılabilir.

- Dönüşüm belirteçlerini hatırlayalım:

```
%hd → short       %d → int          %ld → long        %c → char
%f → float        %lf → double      %LG → long double
```

Örnek 3:

```
#include <stdio.h>
int main(){
      char karakter = '0';
      int kod;
      kod = karakter;
      printf("%c karakterinin ASCII kodu %d'dir.\n", karakter, kod);
      return 0;
}
```

`scanf()` fonksiyonu

- Belirlenen girdi aracından değişkenlere değerler okur. Dönüşüm belirteçleri değişken tiplerine uymalıdır. Aşağıdaki örnekte, kullanıcının değişkenler için değerler girmesi ve "enter" tuşuna basması beklenir.

```
scanf("%d%f", &sayi11, &sayi2);
```

- Karakterleri okurken sorun oluşabilir:

```
scanf("%c", &b);  /*'t' yazip entere basarsanız, b = 't'*/
scanf("%c", &b);  /*bir şey yazmadan entere basarsanız b = '\n'*/
```

`getchar()` fonksiyonu

- Standart girdi aracından bir seferde bir karakter okur ve bir tamsayı döner. Kullanıcının enter tuşuna basmasını da bekler.

```
int ch, char a;
ch = getchar();
a = getchar();
```

`putchar()` fonksiyonu

- Standart çıktı aracında bir karakterin değerini yazdırır.

```
int a, char c;
putchar(a);
putchar(c);
```

Örnek 4: Kullanıcıdan bir büyük harf alan ve aldığı harfi küçük harfe döndüren programı yazalım. Kullanıcının yalnızca büyük harfler gireceğini varsayabiliriz.

```
#include <stdio.h>
/* Okudugu buyuk harfi kucuk harfe cevirir.*/
int main(){
	char a;
	printf("Buyuk harf girin: ");
	a = getchar();
	a += 'a'-'A';
	printf("Harfin kucuk hali %c'dir.\n", a);
	return 0;
}
```

Örnek 5: Kullanıcıdan iki kayan nokta (reel) sayı alan ve yerlerini değiştiren programı yazalım.

```
#include <stdio.h>
int main(){
	float sayi1, sayi2;
	float gecici;
	printf("Iki sayi girin: ");
	scanf("%f%f", &sayi1, &sayi2);
	printf("Sayilarin ilk degerleri: %f %f\n", sayi1, sayi2);
	gecici = sayi1;
	sayi1 = sayi2;
	sayi2 = gecici;
	printf("Sayilarin son degerleri: %f %f\n", sayi1, sayi2);
	return 0;
}
```

II. SEÇİM DEYİMLERİ

SELECTION STATEMENTS

(if, switch)

`if` deyimi

- Şimdiye kadar, bir program sırayla koşulan deyimlerden (sıralı deyimler) oluşmuştu. Yani, deyimler bütün koşullarda bir kez koşuluyordu.

Bölen sıfıra eşit değilse, iki sayı üzerinde bölme işlemi yapmak istediğimizi varsayalım. Bu yüzden, deyimlerin sırasını kontrol etmeliyiz, yani program bir sonraki adımda ne yapacağına karar vermeli.

```
if (bolen != 0)
      bolum = sayi / bolen;
```

Bir öğrenci dersten geçtiyse (`puan ≥ 60` ise) bir mesaj yazdırmak istediğimizi düşünelim:

```
if (puan >= 60)
      printf("Gectiniz");
```

Şimdi aynı zamanda, öğrenciye kaldıysa da (`puan < 60` ise) bilgi vermek istediğimizi varsayalım. O zaman, seçmeli bir `else` olmalı. Her `else`, başka bir `else` ile eşleşmeyen bir `if` ile eşleşmeli.

```
if (puan >= 60)
      printf("Gectiniz");
else
      printf("Kaldiniz");
```

Başarılı olan öğrencilerin sayısını da bilmek istediğimizi düşünelim. {…} sözdiziminin bir bloğu veya birleşik deyimi gösterdiğini ve bir deyimden fazla deyimi gruplandırdığını hatırlayalım.

```
basariliSayac = 0;
if (puan > = 60) {
      printf("Gectiniz");
      basariliSayac++;
}
else
      printf("Kaldiniz");
```

`if` deyiminin sözdizimi (sentaks) aşağıdaki gibidir:

```
if (ifade)
      deyim blogu 1;
else
      deyim blogu 2;
```

- Eğer (`if`) `ifade` doğruysa (`TRUE`), `deyim blogu 1` koşulur; aksi durumda (`else`), yani ifade yanlışsa (`FALSE`), `deyim blogu 2` koşulur (`else` bölümü yoksa hiçbir şey koşulmaz).
- `ifade` tamsayı bir değer olmalı, 0 yanlış (`FALSE`) ve 0'dan farklı değerler DOĞRU (`TRUE`) anlamına gelir.
- `else` seçmelidir.
- Bir deyim bloğu tek bir deyim, birleşik bir deyim ya da boş bir deyim olabilir.
- `if/else`'ten sonra programın çalışması normal biçimde devam eder.

Bazı ifadeler sayısal değerlerdir ve olası kısa yollarımız vardır. Sıfır ile bölme örneğini ele alın:

```
if (y != 0)
      sonuc = x / y;
```

ve

```
if (y)
      sonuc = x / y;
```

denktir (aynıdır).

Örnek 1: Kullanıcıdan üç tamsayı alan; ortalamayı, en küçüğünü ve en büyüğünü gösteren program

```
#include <stdio.h>
/*Uc tamsayinin ortalamasini, en kucugunu ve en buyugunu hesapla ve bastir */

int main(){
            int a, b, c;
            int min, max;
```

```
        printf("Uc sayi girin: ");
        scanf("%d%d%d", &a, &b, &c);
        printf("Ortalama: %f\n", (a + b + c)/3.0);
        if (a < b) {
            min = a;
            max = b;
        }
        else {
            min = b;
            max = a;
        }
        if (c < min)
            min = c;
        else if (c > max)
            max = c;

        printf("En kucuk sayi: %d\n", min);
        printf("En buyuk sayi: %d\n", max);
        return 0;
}
```

İç-içe `if` deyimleri

`if-else` yapısını tek bir deyim olarak düşünebilir ve başka `if`ler veya `else`ler için bir deyimin yerine kullanabiliriz. Çok yönlü kararları kodlamak için iç-içe `if`ler kullanırız.

Öğrencileri aldıkları puana göre üçe ayırmak (puanı 60'dan fazla olanlar, puanı 50 ile 60 arasında olanlar ve puanı 50'den az olanlar) ve buna göre bir mesaj göstermek istediğimizi varsayalım:

```
if (puan >= 60)
      printf("Gectiniz");
else
      if (puan >= 50 && puan < 60)
            printf("Baska bir sinava girmelisiniz");
```

```
      else
            printf("Kaldiniz");
```

elselerle ifleri birbiriyle eşleştirme:

```
if (n > 0)
      if (a > 0)
            n = a;
      else
            n = b;
else
      if (a < 0)
            b = 1;
      else if (b == 0)
            b = n;
```

```
if (n > 0) {
      if (a > b)
            b = a;
}
else
      n = b;
```

- Eşleştirme yapılırken {…} dikkate alınmalı; yardımcı olacak {…} yoksa her bir else kendisinden önce gelen en yakın if ile eşleştirilmelidir.

Örnek 2: Alınan puana harf notu atayan programı yazalım. Kural şöyle: 90-100 → A, 80-89 → B, 70-79 → C, 60-69 → D ve 0-59 → F

```
#include <stdio.h>

int main() {
            int puan;
            char harf;

            printf("Puan giriniz: ");
            scanf("%d", &puan);

            if (puan > 100 || puan < 0)
                  printf("Gecersiz puan!\n");
            else {
                  if (puan >= 90)
                        harf = 'A';
                  else if (puan >= 80)
                        harf = 'B';
                  else if (puan >= 70)
                        harf = 'C';
                  else if (puan >= 60)
                        harf = 'D';
                  else
                        harf = 'F';

                  printf("Harf notunuz: %c\n", harf);
            }
            return 0;
}
```

Örnek 3: Kullanıcıdan 10'luk tabanda iki sayı ve bir karakter alınacak (`sayi1 karakter sayi2` biçiminde) ve basit bir hesap makinesi tasarlanacak. Eğer karakter `'+'`, `'-'`, `'*'` veya `'/'` ise; program gerekli işlemi yapıp sonucu gösterecek, aksi durumda kullanıcıya bir uyarı mesajı verecek.

```
#include <stdio.h>
#define KUCUK 0.0000001

/* x islem y biciminde bir ifade girilecek
ve sonuc hesaplanip ekrana bastirilacak  */

int main(){
            float x, y, sonuc;
            char islec;

            printf("Islem girin: ");
            scanf("%f %c %f", &x, &islec, &y);

            if (islec == '+')
                 sonuc = x + y;
            else if (islec == '-')
                 sonuc = x - y;
            else if (islec == '*')
                 sonuc = x * y;
            else if (işleç == '/' && ((y-0.0>=KUCUK) || (y-0.0 <= -1*KUCUK)))
                 sonuc = x / y;
            else {
                 printf("Gecersiz islec...\n");
                 return 0;
            }

            printf("Islemin sonucu = %f.\n", sonuc);
            return 0;
}
```

Örnek 4: İkinci dereceden bir denklemin ($ax^2 + bx + c = 0$) gerçek köklerini hesaplayan ve gösteren bir program yazalım. Katsayılar kullanıcıdan alınacak. Gerçek kökler aşağıdaki biçimde hesaplanır:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$\Delta = b^2 - 4ac$ olsun. $\Delta < 0$ ise gerçek kök yoktur. $\Delta = 0$ ise yalnız bir gerçek kök vardır. $\Delta > 0$ ise, iki gerçek kök vardır.

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

/*Ikinci derece denklemin gercek koklerini bulalim*/
```

```
int main(){

            float a, b, c, delta, kok1, kok2;

            printf("Ikinci derece denklemin katsayilarini girin: ");
            scanf("%f%f%f", &a, &b, &c);

            if (a == 0.0)
                 printf("Denklem ikinci derece degil!\n");
            else {
                 delta = b*b-4*a*c;
                 if (delta < 0.0)
                       printf("Gercek kok yok...\n");
                 else
                       if (delta > 0.0) {
                             printf("Iki kok var...\n");
                             kok1 = (-b+sqrt(delta))/(2*a);
                             kok2 = (-b-sqrt(delta))/(2*a);
                             printf("Kokler %f ve %f'dir.\n", kok1, kok2);
                       }
                       else {
                             printf("Yalniz bir kok var...\n");
                             kok1 = (-b)/(2*a);
                             printf("Kok %f'dir.", kok1);
                       }

            }
            return 0;
}
```

Örnek 5: Büyük/küçük olmasına göre; büyük harfleri küçük harfine, küçük harfleri de büyük harfine çeviren bir program yazalım. Karakter bir harf değilse bir mesaj gösterilmeli.

```
#include <stdio.h>
//Buyuk harfi kucuge, kucuk harfi buyuge cevirir.*/

int main(){
            char C;

            printf("Harf girin: ");
            scanf("%c", &C);

            if (C>='A' && C<='Z')
                 printf("%c'nin kucugu %c'dir.\n", C, C + 'a' -'A');
            else if (C>='a' && C<='z')
                 printf("%c'nin buyugu %c'dir.\n", C, C + 'A' -'a');
            else
                 printf("Girdiginiz karakter harf degil...\n");

            return 0;
}
```

`switch` deyimi

Sırası verilen ayın gün sayısını seçmek istediğimizi varsayalım:

```
if (ay == 4 || ay == 6 || ay == 9 || ay == 11)
      gunler = 30;
else if (ay == 2)
      gunler = 28;
else
      gunler = 31;
```

Çok seçenekli bir kararı kodlamanın bir diğer yolu `switch` deyimini kullanmaktır.

```
switch(ay) {
      case 4: case 6: case 9: case 11:
             gunler = 30;
             break;
      case 2:
             gunler = 28;
             break;
      default:
             gunler = 31;
             break;
}
```

`switch` deyiminin sözdizimi (sentaks) aşağıdaki gibidir:

```
switch(ifade) {
      case sabit deger1:
             deyim1;
             break;
      case sabit deger2:
             deyim2;
             break;
      . . .
```

```
      default:
            deyimN;
            break;
}
```

- `ifade`nin değeri test edilir ve `switch` bu değere göre dallanır.
- `İfade`deki değerin tipi tamsayı olmalıdır (`int, short, long, char`). Eğer olmazsa, derleme zamanı (compile-time) hatası olur.
- Her `case` tamsayı bir sabitle etiketlenmelidir.
- `default` seçmelidir ve `ifade` `case`lerin hiçbirine uymazsa koşulur.
- `break` deyimi `switch`ten hemen gerçekleşen bir çıkışa neden olur. Bir `case` bloğu koşulduğunda, koşma bir sonraki `case` bloğuna düşer ve `break` kullanıp (kırıp) `switch`ten ayrılarak başka deyimlerin koşması engellenebilir.

```
switch(sayi1) {
      case 1: printf("bir"); break;
      case 2: printf("iki");
      case 3: printf("uc");
      case 4: printf("dort"); break;
      case 5: printf("bes");
      case 6: printf("alti");
      default: printf("baska");
}
```

```
switch(sayi1) {
      case 1: printf("bir"); break;
      default: printf("baska");
      case 2: printf("iki");
}
```

Örnek 6: Kare (`k,K`), dikdörtgen (`d,D`) veya çember (`c,C`) olabilen ve seçilen geometrik şeklin alanını hesaplayıp gösteren bir program yazalım.

```
#include <stdio.h>
#define PI 3.14
/* Secilen kare (k,K), dikdortgen (d,D) veya cember (c,C) geometrik
sekillerinden birinin alanini hesaplar ve ekrana yazdirir. */

int main() {
            char secim;
            float a, b, alan;
            printf("Seciminizi girin: ");
            scanf("%c", &secim);

            switch(secim) {
                  case 'k': case 'K':
                        printf("Kenar boyunu girin: ");
                        scanf("%f",&a);
                        alan = a * a;
                        printf("Karenin alani: %.4f\n", alan);
                        break;
```

```
                case 'd': case 'D':
                        printf("Kenar boylarini girin: ");
                        scanf("%f%f",&a,&b);
                        alan = a * b;
                        printf("Dikdortgenin alani: %.4f\n", alan);
                        break;
                case 'c': case 'C':
                        printf("Yaricapi girin: ");
                        scanf("%f",&a);
                        alan = PI * a * a;
                        printf("Cemberin alani: %.4f\n", alan);
                        break;
                default:
                        printf("Gecersiz secim\n");
                        break;
        }
        return 0;
}
```

Örnek 7: `switch` kullanarak harf notlarını atayan bir program yazalım.

```
#include <stdio.h>
int main() {
        int puan;
        char harf;
        printf("Puaninizi girin: ");
        scanf("%d",&puan);

        if (puan > 100 || puan < 0)
                printf("Gecersiz not...\n");
        else {
                switch (puan/10) {
                        case 10: case 9:
                                harf = 'A';
                                break;
                        case 8:
                                harf = 'B';
                                break;
                        case 7:
                                harf = 'C';
                                break;
                        case 6:
                                harf = 'D';
                                break;
                        default:
                                harf = 'F';
                                break;
                }
                printf("Harf notunuz: %c\n", harf);
        }
        return 0;
}
```

Örnek 8: Verilen bir tarihi, eğer geçerliyse, yeniden biçimlendiren bir program yazalım. Gün, ay ve yıl tamsayı olarak verilecek ve tarihin geçerli olup olmadığı kontrol edilecektir.

```
#include <stdio.h>

int main(){

		int gun, ay, yil;

		printf("Gun, ay ve yil girin: ");
		scanf("%d%d%d", &gun, &ay, &yil);

		if (ay < 1 || ay > 12) {
			printf("Gecersiz ay...\n");
			return 0;
		}

		if (gun < 1) {
			printf("Gecersiz gun...\n");
			return 0;
		}

		switch(ay) {
			case 2:
				if ((yil%4 != 0 || (yil%100 == 0 && yil%400 != 0)) &&
					gun>28) {
					printf("Gecersiz gun...\n");
					return 0;
				}
				else
					if (gun > 29) {
						printf("Gecersiz gun...\n");
						return 0;
					}
				break;
			case 4: case 6: case 9: case 11:
				if (gun > 30) {
					printf("Gecersiz gun...\n");
					return 0;
				}
				break;
			default:
				if (gun > 31) {
					printf("Gecersiz gun...\n");
					return 0;
				}
				break;
		}
```

```
                printf("%d/",gun);
                switch(ay){
                     case 1: printf("Ocak/"); break;
                     case 2: printf("Subat/"); break;
                     case 3: printf("Mart/"); break;
                     case 4: printf("Nisan/"); break;
                     case 5: printf("Mayis/"); break;
                     case 6: printf("Haziran/"); break;
                     case 7: printf("Temmuz/"); break;
                     case 8: printf("Agustos/"); break;
                     case 9: printf("Eylul/"); break;
                     case 10: printf("Ekim/"); break;
                     case 11: printf("Kasim/"); break;
                     case 12: printf("Aralik/"); break;
                     default: break;
                }
                printf("%d\n",yil);

                return 0;
}
```

III. YİNELEME DEYİMLERİ = DÖNGÜLER

REPETITION STATEMENTS = LOOPS

(while, for, do-while)

Şimdiye kadar, sıralı deyimleri ve seçim deyimlerini gördük. Geldiğimiz aşamada, aşağıdaki problemleri dikkate alalım:

1. Her satırın başında satır numaralarını yazan bir program yazmak (100 satır olsun)
2. 1000 öğrencinin sınav notlarının ortalamasını hesaplamak
3. Kullanıcıdan bir metin alarak, yeni satır (‘\n’) karakterini görene kadar bu metindeki küçük harfleri karşılık gelen büyük harflere dönüştürmek
4. Kullanıcının girdiği sayının faktöriyelini hesaplamak

1. ve 2. durumlarda, sıralı deyimler kullanarak programlar yazmak mümkündür ama bu pratik değildir. 3. ve 4. durumlarda, deyimlerin kaç kere koşulacağını önceden bilmediğimiz için (örneğin, programı koşmadan önce kullanıcının ne zaman yeni satır karakteri (‘\n’) gireceğini bilemeyiz) sıralı deyimlerden oluşan programlar yazmak mümkün değildir.

Bu durumlarda, deyimleri ***yineleyen*** biçimde koşma yapılarına ihtiyacımız vardır.

Bir ***döngü*** **(loop),** bazı koşullar sağlanıyor iken bilgisayar tarafından koşulan deyimler grubudur.

`while`

1. 100 satırı, satır numaralarıyla yazdırmak istediğimizi varsayalım

```
i = 1;
while (i <= 100) {
          printf("satir&d\n", i);
          i++;
}
```

Doğruluğu kontrol edilen koşul yanlış (`FALSE` veya 0) olana kadar, deyimlerin koşulması tekrarlanacaktır. Eğer kontrol edilen koşul doğruysa (`TRUE` veya 0’dan farklı değerler), deyim bloğu koşulacaktır. Bir deyim bloğu, tek bir deyimden veya birleşik bir deyimden (birden fazla tekli deyim) oluşabilir. Yukarıdaki örnekte, deyim bloğu `i>100` olana kadar koşulmaktadır. Bu türden `while` döngüsünde, deyimlerin tam olarak kaç kere koşulduğunu bilmekteyiz.

Aşağıdaki kodu dikkate alalım:

```
i = 1;
while (i++ <= 100)                  /* ++i kullanırsak ne olur? */
             printf("satir%d\n",i);
```

2. Kullanıcının girdiği *n* tane sayının ortalamasını hesaplayalım. Sayıları negatif bir sayı girilene kadar almalıyız.

```
toplam = 0.0;

n = 0;

scanf("%d", &sayi);     /* sayi while dongusunun kosmasini kontrol ediyor */

while (sayi>=0){  /* (sayi >= 0) sınanan (test edilen doğruluğu kontrol edilen)
                  koşuldur */

             toplam += sayi;

             n++;

             scanf("%d",&sayi);     /* kontrol degiskeninin yeni degeri */

}

if (n>0)

             ortalama = toplam/n;

else

             ortalama = 0;
```

Bu örnekte, döngü `sayi < 0` olana kadar koşulmaktadır. `toplam` ve `n` ilk değerleri atanan diğer değişkenlerdir. Bu türden `while` döngüsünde, döngünün kaç kere koşulacağını tam olarak bilemeyiz. *sonDeger* (sentinel) olarak bildiğimiz değer, koşmanın sonuna geldiğimizi gösterir.

Örnek 1: Kullanıcı tarafından girilen `n` tane sayının ortalamasını hesaplayan program yazalım. `n` programın başında kullanıcı tarafından giriliyor.

```
#include <stdio.h>

int main(){

             int i = 1, n;

             float sayi, ort = 0.0;

             printf("Sayilarin sayisini girin: ");

             scanf("%d",&n);


             while (i<=n) {

                  printf("Siradaki sayiyi girin: ");

                  scanf("%f",&sayi);

                  ort += sayi;

                  i++;

             }
```

```
            if (n>0) {
                  ort /= n;
                  printf("Ortalama: %.2f\n", ort);
            }
            else
                  printf("Sayilarin sayisi 0'dan buyuk olmali.\n");
            return 0;
}
```

Örnek 2: Serbest düşüş yapan bir nesne üzerinde yerçekiminin etkisini düşünelim. Bir kuleden düşmekte olan bir cismin yere vurana kadar her saniye için yerden yüksekliğini gösteren bir program yazalım. Nesne yere vurduğunda da bir mesaj gösterilmeli. Kulenin yüksekliği olan h_0 kullanıcı tarafından söylenmektedir ve t zamanında nesnenin yüksekliği $h = h_0 – 1/2gt^2$ olarak bulunur.

```
#include <stdio.h>
#define g 9.81

int main() {
            float h0, h;
            int t;
            printf("Kulenin yuksekligini girin: ");
            scanf("%f",&h0);
            h = h0;
            t = 0;
            while (h>0) {
                  printf("%d\t%f\n",t,h);
                  t++;
                  h = h0 - 0.5*g*t*t;
            }

            printf("NESNE YERE VURDU!\n");
            return 0;
}
```

Örnek 3: Kullanıcı sırayla karakterler girmektedir. '*' karakteri girilene kadar görülen küçük harfleri sayan bir program yazalım.

```
#include <stdio.h>

int main(){
            const char sonDeger = '*';
            char c;
            int sayac = 0;
            printf("Karakterleri teker teker girin: ");
            scanf("%c",&c);
            while (c!='*') {
                  if (c >= 'a' && c<='z')
                        sayac++;
                  scanf("%c",&c);
            }
            printf("Kucuk harflerin sayisi: %d\n",sayac);
            return 0;
}
```

for

100 satırı satır numaralarıyla yazdıran aşağıdaki örneği dikkate alalım.

```
for (i = 1;  i<=100; i++)
            printf("satir%d\n", i);
```

Bu örnekte, `i` değişkeni döngüyü kontrol etmektedir. İlk değeri, ilkleme deyimi olan `i = 1` kullanılarak atanmıştır ve bu deyim döngünün başında yalnız bir kere koşulur. Sonra, sınama koşulu kontrol edilir. Bu koşul doğru (0'dan farklı) olursa, döngü vücudundaki deyim veya deyimler koşulur. Deyim(ler)in koşmasından sonra, değiştirme bölümü (`i++`) koşulur ve sınama koşulu kontrol edilir. Döngü, sınama koşulu yanlış (0) olana kadar aynı biçimde koşmasını sürdürür.

- `for` döngüsü üç kısımdan oluşur: ilkleme (`i=1`), sınama (`i<=100`) ve değiştirme (`i++`). Bu üç kısım noktalı virgüllerle (`;`) birbirinden ayrılmalıdır. İlkleme ve değiştirme kısımlarına virgüllerle ayrılan birden çok deyim yazılabilir.

```
for (i = 1, j = 100;  i<=100; i++, j--)
            printf("satir%d\n", i);
```

İlkleme ve değiştirme bölümlerindeki deyimler aritmetik ifade olmak zorunda değildir. Örneğin, buralarda `scanf(…)` kullanılabilir ama bu kötü bir programcılık tarzıdır.

- `for` döngüsünde, bütün ifadeler seçmelidir ama noktalı virgüller kullanılmalıdır. Aşağıdaki döngü sonsuz bir döngüye yol açar (yani sonsuza kadar koşar, böyle döngülerden kaçının)

```
for ( ; ; )
            printf("merhaba\n");
```

- `while` ve `for` döngüleri birbirine dönüştürülebilir. Öte yandan, koşmanın kaç kere olması gerektiği biliniyorsa (sabit sayıda), sıklıkla `for` döngüsü kullanılır.

Örnek 4: Verilen bir *n* sayısının faktöriyelini hesaplayan bir program yazalım.

```
#include <stdio.h>
int main() {
            int sayi, i;
            long fakt = 1;
            printf("Sayi girin: ");
            scanf("%d",&sayi);
            for (i=2; i<=sayi; i++)
                 fakt *=i;
            if (sayi>=0)
                 printf("Faktoriyel: %ld\n",fakt);
            else
                 printf("Negatif sayi girildi...\n");
            return 0;
}
```

Örnek 5: Kullanıcı tarafından girilen bir sayının *mükemmel* sayı olup olmadığını kontrol eden bir program yazalım. Mükemmel sayılar, kendisi dışındaki bütün pozitif bölenlerinin toplamı kendisine eşit olan pozitif sayılardır. Örneğin, 6 ve 28 mükemmel sayılardır. (6 = 1+2+3, 28 = 1+2+4+7+14)

```
#include <stdio.h>
int main(){
            int sayi, i, toplam = 0;
            printf("Sayi girin: ");
            scanf("%d",&sayi);
```

```
            for (i=1; i<=sayi/2; i++)
                  if (sayi % i == 0)
                        toplam += i;
            if (sayi == toplam)
                  printf("%d mukemmel sayidir.\n", sayi);
            else
                  printf("%d mukemmel sayi degildir.\n", sayi);
            return 0;
}
```

do-while

`while` ve `for` döngüleri döngüden çıkma koşulunu en başta kontrol ederler, ama `do-while` döngüsü, döngü vücudunu koştuktan sonra en sonda kontrol eder. `do-while` döngüsünün vücudu her zaman en az bir kere koşulur.

100 satırı satır numaralarıyla gösterme problemine geri dönelim.

```
i = 1;
do {
            printf("satir%d\n", i);
            i++;
} while (i<=100);
```

Bu örnekte, döngüdeki deyimler önce koşulur, sonra ifade doğruysa deyimler tekrar koşulur. Sınama koşulu yanlış olana kadar döngü devam eder. Eğer döngünün en az bir kere koşulacağı biliniyorsa `do-while` döngüsü kullanılabilir.

Örnek 6: Karenin alanını hesaplayan bir program yazalım. Her hesaplamanın sonunda, program kullanıcıya devam edip etmemek istediğini sormalı ve kullanıcı '`q/Q`' karakterini girdiyse sonlanmalı. Aksi halde, hesaplamaya devam etmeli.

```
#include <stdio.h>
int main() {
            char secim;
            float a;
            do {
                  printf("Kenar uzunlugunu girin: ");
                  scanf("%f",&a);
```

```
                printf("Alan: %f\n",a*a);
                printf("Sonlandirmak icin q/Q girin...\n");
                printf("Surdurmek icin herhangi baska bir karakter girin...\n");
                scanf("\n%c",&secim);
        }while (secim!='q' && secim!='Q');
        return 0;
}
```

Örnek 7: Kenar sayısı kullanıcı tarafından belirtilen bir çokgenin çevresini hesaplayan bir program yazalım. Eğer kenar sayısı çokgenler için geçersiz bir değerse, program bir uyarı mesajı göstermeli. Aksi halde, her kenarın boyu kullanıcıya sorulup çevre hesaplanmalı.

```
#include <stdio.h>


int main() {
        int kenar_sayisi = 0;
        float uzunluk, cevre = 0;
        printf("Kenar sayisini girin: ");
        scanf("%d",&kenar_sayisi);
        if (kenar_sayisi<3)
                printf("Cokgen degil...\n");
        else {
                do {
                        printf("Kenar uzunlugu girin: ");
                        scanf("%f",&uzunluk);
                        cevre += uzunluk;
                        kenar_sayisi--;
                } while (kenar_sayisi>0);
                printf("Cevre: %f\n",cevre);
        }
        return 0;
}
```

İç-içe döngüler

Şimdi, birbirinin içinde iki veya daha çok döngü olacak (iç-içe `if`ler veya `switch`ler gibi).

Örnek 8: Satır sayısı kullanıcı tarafından verilen ikizkenar bir üçgeni '`*`' karakterleriyle çizen bir program yazalım.

```
#include <stdio.h>
int main(){
		int satir, i, j;
		printf("Satir sayisini girin: ");
		scanf("%d",&satir);
		for (i=1; i<=satir; i++) {
			for (j=0; j<satir-i; j++)
				printf(" ");
			for (j=0; j<i*2-1; j++)
				printf("*");
			printf("\n");
		}
		return 0;
}
```

Örnek 9: Girdiyi '`!`' ile sonlanan karakterler dizisi olarak alan bir program yazalım. Program, tekrarlanan en uzun ve aynı harften oluşan küçük harf dizisini tespit etmeli, hangi harf olduğunu ve tekrarlamanın uzunluğunu çıktılamalı.

```
#include <stdio.h>
int main() {
		char simdiki, onceki, enuzun;
		int enBuyukUzunluk = 0, simdikiUzunluk = 0;
		printf("! ile biten karakter dizisini girin: ");
		scanf("%c", &simdiki);
		onceki = simdiki;
		while (onceki != '!') {
			if (onceki == simdiki)
				simdikiUzunluk++;
```

```
                else {
                        if (simdikiUzunluk > enBuyukUzunluk && onceki >='a' &&
                        onceki <='z') {
                                enuzun = onceki;
                                enBuyukUzunluk = simdikiUzunluk;
                        }
                        simdikiUzunluk = 1;
                }
                onceki = simdiki;
                scanf("%c",&simdiki);
        }
        if (enBuyukUzunluk > 0)
                printf("%c karakteri %d kere art arda var.\n",
                        enuzun,enBuyukUzunluk);
        else
                printf("Hic kucuk harf yok\n");
        return 0;
}
```

Örnek 10: Son okunan sayı bir önce okunan sayının iki katı olana kadar kullanıcıdan tamsayılar okuyan bir program yazalım. Program okunan sayıların ortalamasını hesaplayıp çıktılamalı.

```
#include <stdio.h>
int main(){
        int sayi, onceki, sayac = 1;
        float ort;
        printf("Bir sayi girin: ");
        scanf("%d",&sayi);

        ort = onceki = sayi;
        while (sayi != 2*onceki) {
                onceki = sayi;
                printf("Bir sayi girin: ");
                scanf("%d",&sayi);
```

```
                ort += sayi;
                sayac++;
            }
            ort /= sayac;
            printf("Ortalama: %.2f\n", ort);
            return 0;
}
```

break

Aynen switchteki gibi; for, while ve do-while döngülerinden erken bir çıkışa neden olur. breaki çevreleyen en içteki döngüden veya switchten hemen çıkılmasını sağlar.

Kullanıcının girdiği sayıların ortalamasını hesaplayalım. Negatif bir sayı girildiğinde durmalıyız.

```
ortalama = n = 0;
while (1) {
            scanf("%d", &sayi);
            if (sayi < 0)
                break;
            ortalama += sayi;
            n++;
}
if (n > 0)
            ortalama /= (float)n;
```

Örnek 11: Kullanıcıdan tek bir tamsayı almak istiyoruz, ama girilen sayının geçerli olup olmadığını kontrol edeceğiz. Rakamlardan farklı karakter içeren sayılar geçersizdir. Ayrıca, biliyoruz ki boşluk (' '), yatay durak ('\t') veya yeni satır ('\n') karakterlerinden biri girilirse sayı sonlanmıştır.

```
#include <stdio.h>
int main() {
            char c;
            long sayi = 0;
            int gecersiz = 0;
            printf("Bir tamsayi girin: ");
            c = getchar();
```

```
        while (c!=' ' && c!='\n' && c!='\t') {
              gecersiz = 0;
              if (c >= '9' || c <='0') {
                    gecersiz = 1;
                    break;
              }
              sayi = sayi*10+c-'0';
              c = getchar();
        }
        if (gecersiz)
              printf("Girilen sayi gecersiz bir sayidir.\n");
        else
              printf("Girdiginiz sayi: %ld\n", sayi);
        return 0;
}
```

continue

Çevreleyen `for`, `while` veya `do-while` döngüsünün sıradaki yinelenmesinin (tekrarının) başlamasına sebep olur. `switch` ile kullanılamaz.

`while` ve `do-while` döngülerinde, hemen sınama bölümüne geçilir.

`for` döngüsünde, kontrol değiştirme adımına geçer.

Aşağıdaki döngülerin koşmalarından önce programın girdisinin 5 -4 3 2 -5 8 9 1 olduğunu varsayalım. Son satırlardaki yorumlar kod parçaları koştuğunda `toplam` değişkeninin alacağı son değeri belirtiyor.

```
i = 0; toplam = 0;
while (i<6) {
      scanf("%d", &sayi);
      if (sayi<=0)
          continue;
      toplam += sayi;
      i++;
}
/* toplam 28 olur */
```

```
toplam = 0;
for (i = 0; i<6; i++)
      scanf("%d", &sayi);
      if (sayi<=0)
          continue;
      toplam += sayi;
}

/* toplam 18 olur */
```

IV. FONKSİYONLAR
FUNCTIONS

Fonksiyonlar neden gereklidir?

1. Gerçek dünyadaki problemler, şu ana kadar ele aldıklarımızdan çok daha büyüktür. Bu yüzden, problemi daha küçük alt problemlere bölme ihtiyacını duyarız. Bir problem verilince, önce soyut formulasyonuyla başlarız (formüle etme) ve sonra daha detaylı alt problemler üzerinde çalışırız.

2. Fonksiyonlar, önceden yazılan kodları yeniden kullanmamıza olanak tanır. Örneğin, `<stdio.h>` program kitaplığında tanımlı standart fonksiyonlar vardır ve biz onları kendi amaçlarımız için kullanırız.

   Bir diğer örnek olarak, iki sayının permutasyonunu hesaplamayı ele alalım: *P(n,k) = n!/(n-k)!* Bu örnekte, faktöriyeli iki defa hesaplamalıyız, yani kodunu iki kere yazmalıyız. Bunun yerine, bu amaç için bir fonksiyon tanımlayıp gereken yerlerde kullanabiliriz (`printf()`, `scanf()`, vs. fonksiyonları gibi). Böylelikle derli-toplu kodlarımız olabilir.

Farklı durumlar için, '`-`' ve '`|`' karakterlerini kullanarak kare çizme problemini ele alalım:

3. 2 x 2 bir kare çizen program yazmak istiyoruz.

```
#include <stdio.h>


int main() {
            printf(" --\n");
            printf("|  |\n");
            printf("|  |\n");
            printf(" --\n");
            return 0;
}
```

4. Yukarıdaki programda, programımızın farklı yerlerinde 2 x 2 'lik kareler çizmek istiyorsak, aynı kodu birçok kez yazmak zorunda kalırız. O zaman, bu kodu yalnız bir kere yazalım ve sonra adını kullanarak çağıralım.

```
#include <stdio.h>

void kareCiz() {
            printf(" --\n");
            printf("|  |\n");
            printf("|  |\n");
            printf(" --\n");
}

int main() {
            kareCiz();
            return 0;
}
```

5. Ancak, yukarıdaki kodda, fonksiyon yalnız 2 x 2 kareler için çalışıyor. *n x n* kareler için çalışması amacıyla fonksiyonu genelleştirelim. *n*'yi parametre olarak fonksiyona geçmeliyiz.

```
#include <stdio.h>

void kareCiz(int n) {
            int i, j;

            printf(" ");
            for (i=0; i<n; i++)
                  printf("-");
            printf("\n");

            for (i=0; i<n; i++) {
                  printf("|");
                  for (j=0; j<n; j++)
                        printf(" ");
                  printf("|\n");
            }
```

```
            printf(" ");
            for (i=0; i<n; i++)
                 printf("-");
            printf("\n");
}

int main() {
            int n = 3;
            kareCiz(n);

            printf("\n");

            n = 5;
            kareCiz(n);

            return 0;
}
```

6. Yukarıdaki fonksiyonda, tek bir yatay çizgiyi bastıran (çıktılayan) kodu iki kere yazdık. O halde, bu amaç için yeni bir fonksiyon tanımlayabiliriz. Bu yeni fonksiyonun da karenin boyunu bilmesi gerekir, öyleyse bu değeri yeni fonksiyona parametre olarak geçmeliyiz.

```
#include <stdio.h>

void yatayCizgiCiz (int n) {
       int i;
       printf(" ");

       for (i=0; i<n; i++)
            printf("-");
       printf("\n");
}

void kareCiz(int n) {
       int i, j;

       yatayCizgiCiz(n);

       for (i=0; i<n; i++) {
            printf("|");
```

```
                for (j=0; j<n; j++)
                      printf(" ");
                printf("|\n");
          }

          yatayCizgiCiz(n);
    }

    int main() {
          int n = 3;
          kareCiz(n);

          printf("\n");

          n = 5;
          kareCiz(n);

          return 0;
}
```

7. Aşağıdaki gibi, dik bir çizgi (‘|’) ile başlayıp biten bir satır çizmek için de bir fonksiyon tanımlayabiliriz.

```
#include <stdio.h>

void yatayCizgiCiz (int n) {

          int i;

          printf(" ");


          for (i=0; i<n; i++)

               printf("-");

          printf("\n");

}

void dikeyCizgiCiz (int n) {

          int i;

          printf("|");

          for (i=0; i<n; i++)

               printf(" ");

          printf("|\n");

}
```

```
void kareCiz(int n) {
            int i;
            yatayCizgiCiz(n);
            for (i=0; i<n; i++)
                 dikeyCizgiCiz(n);
            yatayCizgiCiz(n);
}
int main() {
            int n = 3;
            kareCiz(n);
            printf("\n");
            n = 5;
            kareCiz(n);

            return 0;
}
```

8. Eğer `kareCiz()` fonksiyonunu, *n* x *m* kareler çizmesi için genelleştirmek istiyorsak, bu iki bilgiyi de, aşağıdaki gibi, fonksiyona göndermeliyiz.

```
#include <stdio.h>

void yatayCizgiCiz (int n) {
            int i;
            printf(" ");
            for (i=0; i<n; i++)
                 printf("-");
            printf("\n");
}

void dikeyCizgiCiz (int n) {
            int i;
            printf("|");
```

```
            for (i=0; i<n; i++)
                 printf(" ");
            printf("|\n");
}

void kareCiz(int n, int m) {
            int i;

            yatayCizgiCiz(m);
            for (i=0; i<n; i++)
                 dikeyCizgiCiz(m);
            yatayCizgiCiz(m);
}

int main() {
            int n = 2, m = 5;
            kareCiz(n, m);

            printf("\n");

            n = 8;
            m = 3;
            kareCiz(n, m);

            return 0;
}
```

Bir fonksiyonu tanımlamak için sözdizimi (sentaks) aşağıdaki gibidir:

```
donulen_degerin_veri_tipi     fonksiyon_adi(parametre_listesi) {
            bildirimler;
            deyimler;
}
```

- Fonksiyon adı, geçerli bir kimlik belirten ad olabilir ve fonksiyon adıyla çağırılır.

- `{…}`, bildirimlerden ve deyimlerden oluşan fonksiyon vücudunu barındırmaktadır.

- Parametre listesi, çağıranın fonksiyona veri göndermesini sağlar. C dilinde, fonksiyonlara yalnızca değerler geçilebilir, değişkenler değil. Çağıranda, parametrelerin değerleri bir değişken, bir sabit veya aşağıdaki örnekteki gibi bir ifade olabilir.

```
void fonksiyonum(int n, float k, char c) {
            . . .
}

int main() {
            int b =3, float a, char d = 'C';
            fonksiyonum(b, 5.6, d + 2 - '+');
            return 0;
}
```

Parametre listesi, virgüllerle ayrılan birden çok parametreden oluşabilir. Her biri için, veri tipi fonksiyon tanımında belirtilmelidir (çağıran tarafından çağırıldığı zaman değil). Hiç parametre yoksa çağıran ile fonksiyon arasında hiç iletişim yoktur.

- Başka türlü belirtilmedikçe, bir fonksiyon bir değer dönmelidir. Fonksiyonun değer dönmediğini belirtmek için `void` anahtar sözcüğü kullanılır. Fonksiyonun döndüğü değer, fonksiyondan çağırıldığı noktaya veri gönderebilmesi için tek yoldur ve yalnızca tek bir değer dönülebilir. Dönülen değerin veri tipi fonksiyon tanımının başında belirtilmelidir. Eğer belirtilmezse bir tamsayı değer (`int`) dönülür. Aşağıdaki örneğe bakalım:

```
float F1(int n, char c) {
            . . .
            return (n / 0.2);
}
```

Bu durumda; çağıran, fonksiyona iki değer göndermekte (`n` ve `c`) ve fonksiyon, çağırana tek bir `float` tipinde değer (`n / 2.0`) göndermektedir. Bir fonksiyon, değer döndüğünde veya sağ küme paranteziyle (`}`) sonlanır. Fonksiyon, fonksiyon vücudu içinde herhangi bir yerde değer dönebilir.

- Fonksiyonlar, kullanılmadan önce bildirilmelidir. Bu yüzden, fonksiyonlar, çağıranlarından önce bildirilmelidir veya çağıranlarından önce *prototipleri* tanımlanmalıdır. Bir prototip

tanımlarken, aşağıdaki örnekteki gibi; dönülen tip, fonksiyon adı ve parametre listesi yazılmalıdır.

```
void dikeyCizgiCiz(int boy);
void dikeyCizgiCiz(int);
long faktoriyel(int);
```

Bir fonksiyon bir başka fonksiyon içinde tanımlanamaz.

Kapsamlar (Scopes)

Bir değişkenin, yalnız içinde tanımlı olduğu bir kapsam vardır. Üç farklı kapsam vardır:

1. *Blok kapsamı* (Block scope): Bloklar, blokların başında bildirilen değişkenlere sahip olabilirler. Bu değişkenler, yalnızca tanımlandıkları blok içinde kullanılabilirler. Örnek:

```
if (a > 10 && a < 25) {
            float b; char c;
            b = a/3.0;
            c = a + 'b';
}
```

2. *Fonksiyon kapsamı* (Function scope): Bir fonksiyonun başında tanımlanan değişkenler fonksiyon kapsamı içindedir. Bu değişkenlere *yerel değişken* (local variable) denir. Fonksiyon içinde her yerde kullanılabilirler ve fonksiyon vücudu dışında kullanılamazlar.

3. *Dosya kapsamı* (File scope): Tüm fonksiyonların dışında tanımlanan değişkenler dosya kapsamı içindedir. Değişkenin tanımlandığı noktadan dosya sonuna kadar, bütün fonksiyonlarda kullanılabilir. Bunlara *global değişken* (global variable) denir. Gerekli olmadıkça, programlarda global değişkenleri kullanmaktan kaçınmak gerekir.

```
double G;                           /* G global degiskendir */
int fonksiyon1() {
            int a;                  /* a yerel degiskendir */
            if (a < 0) {
                  float b;
/* b blok kapsamindadir, yalnızca bu if icinde kullanilabilir */
            }
```

```
            for (a = 1; a < 100; a++) {
                 char c;
/* c blok kapsamindadir, yalnizca bu for icinde kullanilabilir */
            }
}
```

Eğer bir değişken bir kodda kullanılıyorsa, şu sırada aranır:

1. Kendi bloğunda
2. Yerel değişkenler arasında (fonksiyon kapsamında)
3. Global değişkenler arasında

Eğer ilk değerleri atanmadıysa, blok veya fonksiyon kapsamı olan değişkenlerin içeriği çöptür (belirsizdir). Öte yandan, global değişkenlerin ilk değeri atanmazsa, sıfır değerini barındırırlar.

Örnek 1: Aşağıdaki programın çıktısı nedir?

```
#include <stdio.h>
int a = 5, b = 3;
float c = 10;

int ilk(float a, float b, int ch) {
            printf("ilk #1 a = %.2f, b = %.2f, c = %.2f, ch = %d\n",
            (float)a, b, c, ch);
            a = b + ch;
            c--;
            return(a + b--);
}
float ikinci(int a, float c) {
            int ch = 2;
            printf("ikinci #1 a = %.2f, b = %d, c = %.2f, ch = %d\n",
            (float)a, b, c, ch);
            b = ilk(b,c,ch);
            printf("ikinci #2 a = %.2f, b = %d, c = %.2f, ch = %d\n",
            (float)a, b, c, ch);
            return(b*c);
}
```

```
int main() {
            float b = 5.99, ch = 1.2;
            printf("main #1 a = %.2f, b = %.2f, c = %.2f, ch = %d\n",
            (float)a, b, c, (int)ch);
            a = ilk(ch,c,b);
            printf("main #2 a = %.2f, b = %.2f, c = %.2f, ch = %d\n",
            (float)a, b, c, (int)ch);
            ch = ikinci(c,b);
            printf("main #3 a = %.2f, b = %.2f, c = %.2f, ch = %d\n",
            (float)a, b, c, (int)ch);
            return 0;
}
```

Çıktı:

```
main #1 a = 5.00, b = 5.99, c = 10.00, ch = 1
ilk #1 a = 1.20, b = 10.00, c = 10.00, ch = 5
main #2 a = 25.00, b = 5.99, c = 9.00, ch = 1
ikinci #1 a = 9.00, b = 3, c = 5.99, ch = 2
ilk #1 a = 3.00, b = 5.99, c = 9.00, ch = 2
ikinci #2 a = 9.00, b = 13, c = 5.99, ch = 2
main #3 a = 25.00, b = 5.99, c = 8.00, ch = 77
```

Örnek 2: n^k'yı hesaplayan bir program yazalım. k'nın tamsayı olduğunu varsayacağız.

```
#include <stdio.h>
double us(double n, int k) {
            int i, a;
            double gecici = 1.0;
            if (k == 0)
                  return 1.0;
            if (k < 0)
                  a = -1*k;
            else
                  a = k;
            for (i = 0;  i < a; i++)
                  gecici *= n;
```

```
            if (k < 0)
                 gecici = 1.0 / gecici;
            return gecici;
}

int main() {
            int n = 3, k = 5;
            printf("2.3 uzeri 5 = %.2f\n", us(2.3,k));
            printf("3 uzeri 4 = %.2f\n", us(n,4));
            printf("5.99 uzeri -2 = %.2f\n", us(5.99,-2));
            return 0;
}
```

Örnek 3: n ve k sayılar, $n > 0$, $k > 0$ ve $n \geq k$ olmak üzere; faktöriyeli, permutasyonu [$P(n,k) = n! / (n-k)!$] ve kombinasyonu [$C(n,k) = n! / (n-k)! \, . \, k!$] hesaplayan fonksiyonlar yazalım.

```
#include <stdio.h>

long faktoriyel(int n) {
            int i;
            long sonuc = 1;
            if (n == 0)
                 return 1;
            for (i = 2; i <= n; i++)
                 sonuc *= i;
            return sonuc;
}
long permutasyon(int n, int k) {
            long sonuc;
            if (n<0 || k<0 || n<k)
                 return 0;
            sonuc = faktoriyel(n)/faktoriyel(n-k);
            return sonuc;
}
```

```
long kombinasyon(int n, int k) {
            if (n<0 || k<0 || n<k)
                  return 0;
            return (faktoriyel(n)/(faktoriyel(n-k)*faktoriyel(k)));
}
int main() {
            printf("5! = %ld\n", faktoriyel(5));
            printf("P(5,2) = %ld\n", permutasyon(5,2));
            printf("C(5,2) = %ld\n", kombinasyon(5,2));
            return 0;
}
```

Makro yerleştirme:

```
#define PI 3.14
```

Derleme zamanında (compile time), bütün `PI`'lerin yerine 3.14 yerleştirilir.

`#define`, bir *jeton*un (token) yerine herhangi bir karakter dizisini koymak için kullanılır. Bunlara *makro* (macro) denir ve genel biçimi aşağıdadır:

```
#define ad adin_yerine_gececek_metin
```

`ad`, kimlik belirten adların kurallarına uymalıdır.

Argümanları olan makrolar da tanımlamak mümkündür. Verilen bir sayının karesini hesaplamak için, aşağıdaki makroyu tanımlayabiliriz:

```
#define kare(x) (x)*(x)
```

Bu örnekte, parantezler yanlış anlamlandırmaları önlemek için önemlidir. Programda, bu makro aşağıdaki gibi kullanılabilir:

```
a = kare(3);
b = kare(a + 3 * b);
```

Verilen iki sayının büyük olanını bulmak için aşağıdaki maximum (`max`) makrosu tanımlanabilir:

```
#define max(A,B) (((A)>(B)) ? (A) : (B))
```

Programda, aşağıdaki gibi kullanılabilir:

```
z = max(p+q, r+s);
```

Makroların, derleme zamanında gerçekleştirilen metin yerleştirmeler olduğunu unutmayalım.

V. İŞARETÇİLER
POINTERS

İşaretçi (pointer), içinde bir *bellek adresi*nin (memory address) değerini tutabilen bir değişkendir.

Tamsayı işaretçisi (integer pointer) tipinde `p` değişkeninin bildirimini ele alalım:

```
int *p;
```

`p` içinde bir bellek adresi tutar ve bu adreste bir tamsayı tutulur çünkü `p` bir tamsayı işaretçisidir. Bir `float` işaretçisi (float pointer) olan `A`'dan bahsediyorsak (`float *A` diye bildirilir), `A` içinde bir adres tutulur ve bu adresteki bellek hücresinde bir reel sayı (`float`) tutulur.

`A`'nın tuttuğu adreste (adresi `A`'da yazılı olan bellek hücresinde) bir tamsayının da (`int`) tutulabileceğini bilelim ama böyle yapmak kimi sorunlara yol açacaktır. Ayrıca, bir işaretçinin de bir değişken olduğunu unutmayalım, o halde işaretçiler de bellekte bir yerlerde tutulur, yani onların da bir bellek adresi vardır ve bu adres değişkenin içindeki değerle aynı değildir.

`void` işaretçileri dışındaki her işaretçi, belli bir veri tipine işaret eder. `void` işaretçileri, her türde işaretçiyi tutmak için kullanılabilir.

İşaretçi işleçleri (pointer operators)

1. `&` (*adres işleci*): Tekli bir işleçtir ve işleneninin adresini verir. Bellekteki nesnelere uygulanabilir; ifadelere, sabitlere veya tutmaç (register) değişkenlerine uygulanamaz. Aşağıdaki deyimlere bakalım:

   ```
   int *p, c;
   p = &c;
   ```

   İlk satırda iki değişken tanımlanıyor, biri bir tamsayı, diğeri bir tamsayı işaretçisi. İkinci satırda, `c` değişkeninin bellek adresi `p` değişkeninin değeri olarak atanıyor.

2. `*` (*dolaylama (başvurudan ayırma) işleci*) = (indirection (dereferencing) operator): Bu da tekli bir işleçtir ve işaretçilere uygulanabilir. Bu işleçle, adresi işaretçi içinde bulunan değişkene erişebiliriz. Yalnızca, daha önceden kullanım için ayrılmış adreslere erişmeliyiz, aksi halde kimi sorunlarla karşılaşabiliriz. Örneğin,

   ```
   int *p
   *p = 3;
   ```

   Burada `p` adında bir tamsayı işaretçisi tanımlanıyor, ama içinde bilinmeyen (çöp) değer var, bu yüzden bu değer herhangi bir adres olabilir. Bu adrese (`p`'nin değeri) erişmek istediğimiz zaman, muhtemelen kullanım için ayrılmamış bir adrese erişiyor olabiliriz.

Örnek 1:

```
#include <stdio.h>

int main() {

            int a = 3, b = 2, *ptr, *x, *y;

            ptr = &a;        /*ptr a'nin adresinin degerini tutuyor*/

            printf("%d\n",a);

            b = *ptr;        /*ptr'de a'nin adresi var, *ptr
                             kullanilarak a'daki degere erisiliyor,
                             sonra b = 3 oluyor*/

            printf("%d\n",b);

            *ptr = 0;        /*a = 0*/

            printf("%d\n",*ptr);

            x = ptr;         /*ptr degiskeninin degeri x'e ataniyor*/

            printf("%d\n",x);

            x = &*ptr; /*once ptr adresinde tutulan degiskene
                       eris, sonra adresini al, sonuc aslinda
                       ptr'nin degeridir*/

            printf("%d\n",x);

            y = *&ptr; /*ptr degiskeninin adresini al, bu
                       adresteki degiskene eris(ptr degeri)*/

            printf("%d\n",y);

            a = *&b;         /*b degiskeninin adresini al, sonra o
                             adresteki degere eris (b'nin değeri).*/

            printf("%d\n",a);

            a = &*b;         /*gecersiz,* sadece isaretcilere
                             (adreslere) uygulanabilir.*/

            return 0;

}
```

- İşaretçiler ile toplama (+) ve çıkarma (-) işleçleri de kullanılabilir.

Örnek 2:

```
#include <stdio.h>

int main() {

            int a = 3, *x, *y;

            y = &a;          /*y degiskenine a'nin adresi ataniyor*/
```

```
            printf("%d\n",y);

            x = y;          /*y x'e kopyalaniyor, yani x de a'nin
                            adresini tutuyor*/

            printf("%d\n",x);

            *y += 1;        /*y tarafindan tutulan adresteki degiskene
                            eris, sonra o degiskenin degerini 1 artir,
                            yani a = 4 olsun*/

            printf("%d\n",a);

            y = y+1;        /*y isaretcisinin degerini 1 artir, simdi
                            y a degiskeninden hemen sonraki degiskenin
                            adresini tutuyor (tamsayilara gore)*/

            printf("%d\n",y);

            y--;            /*y isaretcisinin degerini 1 azalt, simdi
                            y a degiskeninin adresini tutuyor.*/

            printf("%d\n",y);

            return 0;

}
```

İşaretçiler ve fonksiyon argümanları

Bir fonksiyonumuz olduğunu, fonksiyon içinde iki değerin yerini karşılıklı değiştirmek istediğimizi ve bunları fonksiyonu çağırana dönmek istediğimizi düşünelim. Ancak, bir fonksiyon yalnızca tek bir değer dönebilir ve C dilinde geçilen bir parametrenin değiştirilmesinin doğrudan bir yolu yoktur. Bu etkiyi başarmanın yolu, fonksiyonlara işaretçiler geçmektir.

Parametre olarak gönderilen iki değerin yerinin karşılıklı değiştirilmesi (*kdegistir* = swap) örneği:

```
#include <stdio.h>

void kdegistir(int a, int b) {

            int gecici = a;

            a = b;

            b = gecici;

}

int main() {

            int m = 3, n = 4;

            kdegistir(m,n);

            printf("%d %d\n", m, n);

            return 0;

}
```

`kdegistir` fonksiyonunu `main` içinde çağırınca, bu fonksiyonun hiçbir etkisi olmaz. `a` ve `b` göstermelik değişkenlerinin değeri fonksiyon içinde değiştirilir ama; `m` ile `a` ve `n` ile `b` arasında bağlantılar olmadığından, `m` ve `n` değişkenlerinin değeri aynı kalır.

Şimdi `kdegistir` fonksiyonun aşağıdaki gibi yazalım:

```
#include <stdio.h>
void kdegistir(int *a, int *b) {
            int gecici = *a;
            *a = *b;
            *b = gecici;
}
int main() {
            int m = 3, n = 4;
            kdegistir(&m,&n);
            printf("%d %d\n", m, n);
            return 0;
}
```

Bu durumda, fonksiyona `m` ve `n` değişkenlerinin adreslerini gönderiyor ve `kdegistir` fonksiyonunda bu adreslerde tutulan değişkenlerin değerini karşılıklı olarak yer değiştiriyoruz. Başvurudan ayırma işleci `*` kullanılarak, herhangi bir adreste tutulan değişkenlere erişebileceğimizi biliyoruz. Böylelikle, `main` fonksiyonunun yerel değişkenleri olan `m` ve `n`'ye, `kdegistir` fonksiyonunda dolaylı olarak erişiyoruz. Yani `main` fonksiyonu tarafından `kdegistir` fonksiyonu çağırılınca, `m`'nin değeri 4, `n`'nin değeri 3 oluyor, başka bir deyişle değişkenlerin değeri karşılıklı değişiyor.

Örnek 3: İkinci dereceden bir denklemin gerçek köklerini bulan programı ele alalım. Bu sefer, fonksiyon kullanarak problemi çözmek istiyoruz.

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

#define kucuk 0.000001
#define esit(a,b) ((a)-(b) <= kucuk && (b)-(a) <= kucuk)

float delta(float a, float b, float c) {
            return (b*b-4*a*c);
}
```

```
int coz(float a, float b, float c, float *kok1, float *kok2) {
            float d;
            if (esit(a,0.0))
                 return -1;
            d = delta(a, b, c);
            if esit(d, 0.0) {
                 *kok1 = (-b)/2*a;
                 return 1;
            }
            if (d<0)
                 return 0;
            *kok1 = (-b + sqrt(d))/(2*a);
            *kok2 = (-b - sqrt(d))/(2*a);
            return 2;
}

int main() {
            float k1, k2;
            float a, b, c;
            printf("Denklem katsayilarini girin: ");
            scanf("%f%f%f", &a,&b,&c);
            switch (coz(a, b, c, &k1,&k2)) {
                 case -1:
                       printf("Denklem ikinci dereceden degil\n");
                       break;
                 case 0:
                       printf("Gercek kok yok\n");
                       break;
                 case 1:
                       printf("Bir gercek kok var\n");
                       printf("Kok1 = Kok2 = %f\n",k1);
                       break;
```

```
                case 2:
                        printf("Iki gercek kok var\n");
                        printf("Kok1 = %f\n",k1);
                        printf("Kok2 = %f\n",k2);
                        break;
                default:
                        break;
        }
        return 0;
}
```

VI. DİZİLER

ARRAYS

Kullanıcıdan N tane tamsayı, x_i, alalım ve

1. Bu sayıların ortalamasını hesaplamak isteyelim

$$ort = \frac{\sum_{i=1}^{N} x_i}{N}$$

```
ort = 0;
for (i=0; i<N; i++) {
       scanf("%d), &sayi);
       ort += 0;
}
ort /= N;
```

2. Bu sayıların varyansını (değişinti) hesaplamak isteyelim

$$var = \frac{\sum_{i=1}^{N} (x_i - ort)^2}{N - 1}$$

   Bu defa, x_i değerlerini tutmak zorundayız; çünkü önce ortalamayı hesaplamaya, sonra da değerleri varyans hesaplamasında teker teker kullanmaya ihtiyacımız var. Yollardan biri her sayıyı ayrı değişkenlerde tutmaktır. Örneğin, N = 100 olursa100 tane değişken tanımlamak zorundayız, ama bu hiç uygulanabilir değil. Başka alternatif bir yol bu değişkenleri aynı ad altında gruplandırmak ve her birinin farklı ve tek bir indeksi olmasıdır.

Bir *dizi*(array) bir grup bellek konumudur. Bu konumlar aynı ada ve aynı veri tipine sahiptir. Belli bir konuma, yani dizi içindeki bir elemana, başvurmak (erişmek) için, dizinin adını ve köşeli parantezler içinde değişkenin (elemanın) konum numarasını (indeksini) belirtmek gerekir.

```
int X[100];
int ort = 0, sayi;
float var = 0.0;
for (i=0; i<100; i++) {
       scanf("%d",&sayi);
       ort += sayi;
       X[i] = sayi;
}
ort /= 100;

for (i=0; i<100; i++)
       var += (X[i] - ort) * (X[i] - ort);
var /= 99;
```

Kullanmadan önce dizileri bildirmek (tanımlamak) gerekir.

```
int X[100];

float Y[90], Dizi[10];
```

- `float Y[90];` derleyiciden float dizisi `Y` için 90 konum ayırmasını ister, yani her biri bir float değer tutabilen 90 bellek hücresi.

- Diziler (elemanları) bellekte yer işgal eder, bu yüzden ayrılacak hücrelerin sayısını belirtmek gerekir, örneğin `Y` dizisi için 90. Bu sayılar için sabit ifadelere gerek vardır. Örneğin, aşağıdaki tanım geçerlidir.

```
#define b 3

const int a = 2;

int y [a*b+2];
```

- Dizi elemanlarını değişkenler gibi kullanabiliriz, örneğin:

```
float A[10];
A[3] = 2;               /* A'nin 4. elemanina 2 degerini ata */
A[4] = A[3] * 2;        /* A'nin 4. elemaninin degerini al, 2 ile carp,
                        sonucu A'nin 5. elemanina ata */
A[3]++;                 /* A'nin 4. elemanının degerini 1 artir */
```

- Dizinin ilk elemanının indeksi sıfırdır, yani diziler 0. eleman ile başlar. Dizinin boyu N olduğunda, dizinin son elemanının indeksi $N - 1$'dir.

- İndeksler, yalnızca tamsayılar veya tamsayı ifadeler olabilir.

```
int k[5], i;

for (i=0; i<5; i++)

             k[i] = i*2;

k[2*2-1] = 3;

k[k[4] / k[1]] = 2;

k[1.5] = 3;             /* hata verir */
```

- Dizinin elemanları değişkenler olarak değerlendirilebilir, ama dizi adı bir değişken değildir. (Dizi adlarının aslında neyi ifade ettiği "Diziler ve İşaretçilerin İlişkisi" bölümünde anlatılmaktadır.) `double YeniDizi[5];` tanımını yaptığımızı varsayalım. O zaman `YeniDizi[0]`, `YeniDizi[1]`, `YeniDizi[2]`, `YeniDizi[3]` ve `YeniDizi[4]` değişkenlerimiz var ama `YeniDizi` diye bir değişkenimiz yok.

- Dizinin boyunu geçen elemanlara erişmeyi denemeyelim. `int k[5]` diye bir dizi tanımlandığını varsayalım, bu durumda `k[-45], k[-1], k[5], k[22],` vs. gibi elemanlara erişmeyi denememeliyiz. Aslında derleyici böyle erişime izin verir ama belirtilen indekslerle gösterilen hücreler ayrılmamışsa sorunlarla karşılaşabiliriz.

Bildirim sırasında dizilere ilk değerler atamak da mümkündür, değerler virgüllerle ayrılmış bir liste halinde verilir.

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

int n[4] = {1, 2, 3, 4, 5};   /* çok fazla deger (dizinin boyundan
                                 buyuk), derleme zamani hatasi verir */

int n[4] = {2, 3};                  /* az ilk değer, kalan elemanlara
                                       sifir degeri atanir */

int n[] = {1, 2, 5, 3, 7, 7, 0};

/*calisir, derleyici ilk deger olarak verilen elemanların sayısı kadar boyda bir
dizi yaratir, bu durumda 7 boyunda */

int n[];                            /* derleme zamani hatasi verir */
```

Diziler statik yapılardır. Bu dizinin boyunu kodumuzda belirlediğimiz ve boyunu daha sonra değiştiremeyeceğimiz anlamına gelir. Özet olarak, ya dizinin boyunu belirlemeli ya da diziye ilk değerler atamalıyız.

Örnek 1: Tamsayı veri tipinde 100 tane elemanımız olduğunu düşünelim. Bu sayıların simetrik olup olmadığını kontrol etmek istiyoruz. Elemanlar kullanıcı tarafından sırayla veriliyor.

```
#include <stdio.h>

int main() {

            int sayilar[100], i;


            for (i=0; i<100; i++)

                 scanf("%d",&sayilar[i]);


            for (i=0; i<50; i++)

                 if (sayilar[i] != sayilar[99-i])

                       break;
```

```
            if (i==50)
                printf("Sayilar simetrik\n");
            else
                printf("Sayilar simetrik degil\n");

            return 0;
}
```

Örnek 2: Kullanıcı karakterler giriyor. Bu karakterlerdeki her bir küçük harfi ve her bir büyük harfi sayacağız. Girilen karakterler `%` karakteri okununca bitiyor. Bütün diğer karakterler göz ardı edilecek.

```
#include <stdio.h>
int main() {
            int kucuk[26] = {0}, buyuk[26] = {0}, i;
            char a;
            printf("Karakterleri gir: ");
            scanf("%c",&a);
            while (a != '%') {
                if (a >= 'a' && a <= 'z')
                      kucuk[a -'a']++;
                else if (a >= 'A' && a <= 'Z')
                      buyuk[a - 'A']++;
                scanf("%c",&a);
            }
            printf("Kucuk harfler\n");
            for (i= 0; i<26; i++)
                printf("%d ", kucuk[i]);
            printf("\n");
            printf("Buyuk harfler\n");
            for (i= 0; i<26; i++)
                printf("%d ", buyuk[i]);
            printf("\n");
            return 0;
}
```

Dizilerin önemli bir eksiği vardır: Dinamik olarak yaratılamaz ve kullanılamazlar. Örneğin, kullanıcının 10'luk tabanda birbiri ardına tamsayılar girdiğini ama programın başında bu sayıların kaç tane olduğunu bilmediğimizi düşünelim (sayıların sayısının da kullanıcı tarafından verildiğini veya kullanıcı bir son değer girene kadar sayıların okunduğunu varsayalım). Bu sorunu çok büyük diziler tanımlayarak ve kullanıcının bu sınırı aşmayacak kadar değer girmesini umarak çözebiliriz. Aşağıdaki örneklere bakalım.

```
float kullaniciSayilari[100000], toplam;
int i, n;
scanf("%d",&n);
for (i=0; i<n; i++)
            scanf("%f",&kullaniciSayilari[i]);
toplam = 0;
for (i=0; i<n; i++)
            toplam += kullaniciSayilari[i];
```

Kullanıcı 100000'e eşit veya daha küçük bir boy girerse, yukarıdaki kod çalışır. 100500 girerse, indeksler sınırı aşar ve sorunlarla karşılaşılabilir. Şimdi kullanıcının dizinin boyu olarak 5 girdiğini düşünelim, o zaman da 99995 bellek hücresi boşu boşuna ayrılmış olur. Çok büyük boylu dizileri tanımlamanın verimsiz olduğunun fark ediyoruz, bu yüzden dinamik yapılara ihtiyacımız var.

Örnek 3: Kullanıcıdan iki polinom alan, sonra bunları toplayan ve çarpan bir program yazalım.

```
#include <stdio.h>
#define MAX 100
int main() {
            int p1[MAX] = {0}, p2[MAX] = {0};
            int top[MAX] = {0}, carp[MAX] = {0};
            int i,j,n1,n2, buyukn;
            int carpmaYapilamaz = 0;
            printf("Ilk polinomun derecesini gir: ");
            scanf("%d",&n1);
            if (n1 >= MAX)
                  return 0;
            printf("Katsaylari gir:\n");
            for (i=n1; i>=0; i--)
                  scanf("%d",&p1[i]);
            printf("Ikinci polinomun derecesini gir: ");
```

```
        scanf("%d",&n2);
        if (n2 >= MAX)
             return 0;
        printf("Katsayilari gir:\n");
        for (i=n2; i>=0; i--)
             scanf("%d",&p2[i]);
        if (n1 > n2)
             buyukn = n1;
        else
             buyukn = n2;
        for (i = 0; i<=buyukn; i++)        /*toplama*/
             top[i] = p1[i] + p2[i];
        if (n1+n2 < MAX)                   /*carpma*/
             for (i=0; i<=n1; i++)
                   for (j=0; j<=n2; j++)
                         carp[i+j] += p1[i]*p2[j];
        else
             carpmaYapilamaz = 1;
        printf("Toplama sonucu\n");
        for (i=buyukn; i>=0; i--)
             printf("%d ",top[i]);
        printf("\n");
        if (!carpmaYapilamaz) {
             printf("Carpma sonucu\n");
             for (i=n1*n2; i>=0; i--)
                   printf("%d ",carp[i]);
             printf("\n");
        }
        else
             printf("Carpma yapilamiyor\n");
        return 0;
}
```

VII. DİZİLER VE İŞARETÇİLERİN İLİŞKİSİ
DİNAMİK BELLEK AYIRMA
RELATIONSHIP OF ARRAYS AND POINTERS
DYNAMIC MEMORY ALLOCATION

- Dizi adları aslında dizinin ilk elemanının adresleridir ve işaretçi tipindeki değişkenlerdir.

```
int dizim[10];
int *isaretcim;
isaretcim = &dizim[0];
isaretcim = dizim;
```

- İşaretçiler ve dizilerin gösterimleri birbirlerinin yerine kullanılabilir. Aşağıdaki örneklerde, ≡ sembolü "denktir (aynıdır)" anlamında kullanılıyor.

```
dizim[i]          ≡      *(dizim+i)
dizim + i         ≡      &dizim[i]
isaretcim[i]      ≡      *(isaretcim + i)
isaretcim + i     ≡      &isaretcim[i]
```

Programın koşması sırasında ayrılan bloğun (dizinin) boyunu değiştirmek isteyelim:

- Bir dizinin bildiriminde (tanımında), dolaylı olarak uygun bir bellek bloğu ayrılır ve sonra boyunu değiştiremeyiz.
- Eğer yer ayırma işini doğrudan yapıyorsak, ayrılan bloğun boyunu değiştirebiliriz.
- `<stdlib.h>` kitaplığında tanımlanan üç yer ayırma fonksiyonu vardır:

1. **`malloc`:** Bellekten bir byte bloğu ayırır. Ayrılan bloğun boyu fonksiyona argüman olarak verilir. Başarılı olunursa, yeni ayrılan bloğun başlangıç adresi dönülür. Hata olursa `NULL` dönülür, örneğin, ayrılacak yeterli bellek yoksa.

```
int *a;
a = (int *)malloc (8 * sizeof(int));
```

2. **`calloc`:** Bellekten bir byte bloğu ayırır (`malloc`'un yaptığı gibi). Fark, ayrılan bloğun içeriğinin sıfırlar (0) ile doldurulmasıdır.

```
long *elemanlar;
elemanlar = (long *)calloc (348, sizeof(long));
```

3. **`realloc`:** Ayrılan bloğun boyunu, istenen yeni boya ayarlar. Gerekirse, içeriği kopyalar ve başka bir bellek konumuna aktarır. Başarı durumunda, yeniden ayrılan bloğun başlangıç adresini döner. Bu adres, ilk bloğun adresinden farklı olabilir. Başarısız olunursa, `NULL` dönülür. `realloc`; daha önceden `malloc, calloc` veya `realloc` ile yer ayrılan veya değeri `NULL` olarak belirlenmiş bloklara uygulanır.

```
vektor = (long *)realloc(vektor, 4 * sizeof(long));
```

Örnek 1: Kullanıcıdan sayılar alan ve standart sapmasını bulan program yazalım. Kullanıcının kaç tane sayı gireceğini bilmiyoruz ve bu sayıyı programın koşması sırasında kullanıcıdan alıyoruz. Bu yüzden, gereken belleği koşma sırasında doğrudan (dinamik olarak) ayırıyoruz.

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

int main() {
	int n, i;
	float ort = 0.0;
	float std = 0.0;
	float *sayilar;
	printf("Kac sayi var?: ");
	scanf("%d",&n);
	sayilar = (float *)malloc(n*sizeof(float));
	for (i = 0; i<n; i++) {
		printf("Sayi gir: ");
		scanf("%f", &sayilar[i]);
		ort += sayilar[i];
	}
	ort/=n;
	for (i = 0; i<n; i++)
		std += (ort - sayilar[i])*(ort - sayilar[i]);
	std = sqrt(std/(n-1));
	printf("Standart sapma = %.3f\n", std);
	return 0;
}
```

Örnek 2: Kullanıcıdan sayılar alan ve standart sapmasını bulan program yazalım. Eksi (negatif) bir değer girilene kadar sayıları okuyoruz. Kaç sayı olduğunu negatif değer girildiğinde bilebiliriz.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
int main() {
            int n=0, i;
            float ort = 0.0, std = 0.0;
            float *sayilar;
            float sayi;
            sayilar = NULL;
            printf("Sayi gir: ");
            scanf("%f", &sayi);
            while (sayi >= 0) {
                  n++;
                  ort += sayi;
                  sayilar = (float *)realloc(sayilar, n*sizeof(float));
                  if (sayilar == NULL) {
                        printf("Hata oldu\n");
                        return -1;
                  }
                  sayilar[n-1] = sayi;
                  printf("Sayi gir: ");
                  scanf("%f", &sayi);
            }
            ort/=n;
            for (i = 0; i<n; i++)
                  std += (ort - sayilar[i])*(ort - sayilar[i]);
            std = sqrt(std/(n-1));
            printf("Standart sapma = %.3f\n", std);
            return 0;
}
```

VIII. DİZİLERİ VEYA İŞARETÇİLERİ FONKSİYONLARA GEÇME
PASSING ARRAYS OR POINTERS TO FUNCTIONS

- Dizileri fonksiyonlara geçme veya işaretçileri fonksiyonlara geçme arasında kavramsal olarak bir fark yoktur; çünkü dizi adları aslında işaretçi tipinde değişkenlerdir.

- Dinamik bellek ayırma sırasında, dizileri işaretçi değişkenleri kullanarak oluşturduğumuz için, diziler (dizi adları) ve işaretçiler arasındaki eşliği (aynılığı) daha kolay görebiliriz.

- İşaretçileri fonksiyonlara geçmek aslında birbirine benzer iki bağlamda ele alınabilir:

    - İşaretçiler (adresler) aracılığıyla, söz konusu fonksiyonu çağıran fonksiyondaki değişkenlerin değerini değiştirebilmek

    - Dizi adları dizinin 0. (ilk) elemanının adresini bulunduran değişkenler olduğu için çağrılan fonksiyon içinde dizi elemanlarına erişebilmek ve değerlerini değiştirebilmek

- Diziler fonksiyonlardan dönülemez ama işaretçiler dönülebilir.

- Tek boyutlu dizileri fonksiyonlara geçerken, geçilen dizinin boyunu `[ ]` içinde belirtmek gerekmez, ancak kaç tane eleman üzerinde işlem yapılacağını fonksiyona bildirmek için fonksiyonun parametrelerinden (argüman) biri dizinin boyu olmalıdır.

- Fonksiyonlara geçilen dizinin (0. elemanın adresinin) veya işaretçinin (tuttuğu adresin) değerini değiştiremeyiz; ancak işaret edilen bellek konumlarındaki değerler (örneğin, dizinin elemanlarından her biri) değiştirilebilir.

Örnek 1: 10 tane tamsayımız var (bir dizi içinde). Verilen bir tamsayıya eşit olan bütün elemanları başka bir verilen sayıyla değiştirmek istiyoruz. Bu işlemi yapan ve değiştirilen eleman sayısını dönen bir fonksiyon yazalım.

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

int degistir(int *sayilar, int boy, int deger, int yeni_deger) {
      int i, sayac = 0;

      for (i=0; i < boy; i++) {
            if (sayilar[i] == deger) {
                  sayilar[i] = yeni_deger;
                  sayac++;
            }
      }
      return sayac;
}
```

```
int main() {

	int a[] = {2, 3, 4, 5, 6, 2, 2, 2, 1, 3}, k, *p, i;

	k = degistir(a,10,2,10);

	printf("%d eleman degistirildi\n",k);
	for (i=0; i<10; i++)
		printf("%d ",a[i]);
	printf("\n");

	p = (int *)calloc(10,sizeof(int));
	k = degistir(p,10,0,10);

	printf("%d eleman degistirildi\n",k);
	for (i=0; i<10; i++)
		printf("%d ",p[i]);
	printf("\n");

	return 0;
}
```

Örnek 2: Bir vektör ve bir skaler alan, ikisini çarpan ve sonuç vektörünü dönen bir fonksiyon yazalım.

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

float * carpma(float *V, int boy, float skaler) {

	float *sonuc;
	int i;
	sonuc = (float *)malloc(boy*sizeof(float));

	for (i=0; i < boy; i++)
		sonuc[i] = V[i]*skaler;

	return sonuc;
}

int main() {

	float vektor[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
	float katsayi = 10;
	float *carpim;
	int i;

	carpim = carpma(vektor,10,katsayi);

	printf("ORIJINAL VEKTOR\n");
	for (i=0; i<10; i++)
		printf("%.2f ",vektor[i]);
```

```
            printf("\n");
            printf("KATSAYI\n");
            printf("%.2f\n", katsayi);

            printf("CARPIM SONUCU\n");
            for (i=0; i<10; i++)
                  printf("%.2f ",carpim[i]);
            printf("\n");

            return 0;
}
```

Örnek 3: Dizi adlarının gerçekten işaretçi olduğunu ve öyle davrandığını görmek için bazı çıktılamalar yapalım. Programı koşup çıktıyı gözlemleyelim.

Not: `%p` işaretçi (adres) değerlerini çıktılamak için biçim (dönüşüm) belirtecidir.

```
#include <stdio.h>

int main() {

      char dizi[5];

      printf("%p\n",dizi);
      printf("%p\n",&dizi);
      printf("%p\n",&dizi[0]);
      printf("%p\n",&dizi[2]);
      printf("%p\n",dizi+3);
      printf("\n");

      return 0;
}
```

IX. SIRALAMA, DOĞRUSAL ARAMA VE İKİLİ ARAMA
SORTING, LINEAR SEARCH AND BINARY SEARCH

ÖN BİLGİ:

- **free** fonksiyonu <stdlib.h> kitaplığında tanımlı, parametre olarak herhangi bir tipte işaretçi alan ve o işaretçi aracılığıyla ayrılmış bellek bloğunu serbest bırakan fonksiyondur.

- **rand** fonksiyonu <stdlib.h> kitaplığında tanımlı, rastgele bir tamsayı üreten ve ürettiği tamsayıyı dönen fonksiyondur.

Örnek 1: Verilen bir diziyi (elemanlarını) sıralayan fonksiyon yazalım.

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

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

void dizi_sirala(int *D, int boy) {
      int i, j, gecici;
      for (i = 0; i<boy; i++)
            for (j = i+1; j < boy; j++)
                  if (D[i] > D[j])
                        kdegistir(&D[i], &D[j]);
}

int main() {
      int *S, boy, i;
      printf("Dizinin boyunu gir: ");
      scanf("%d",&boy);
      S = (int *)malloc(boy*sizeof(int));
      for (i=0; i<boy; i++) {
            S[i] = rand()%10;
            printf("%d ",S[i]);
      }
      printf("\n");
      dizi_sirala(S,boy);
      for (i=0; i<boy; i++)
            printf("%d ",S[i]);
      printf("\n");
      free(S);
      return 0;
}
```

Örnek 2: *N* elemanı olan bir dizide verilen bir sayıyı arayan ve bulursa konumunun indeksini dönen fonksiyon yazalım. Sayı dizide yoksa fonksiyon -1 dönmeli. Arama doğrusal arama (linear search) algoritmasıyla yapılacak.

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

int dogrusalArama(int A[], int boy, int sayi) {

	int i;

	for (i=0; i< boy; i++)
		if (A[i] == sayi)
			return i;

	return -1;
}

int main() {

	int *S, boy, sayi, i, bulunan_indeks;

	printf("Dizinin boyunu gir: ");
	scanf("%d",&boy);

	S = (int *)malloc(boy*sizeof(int));

	for (i=0; i<boy; i++) {
		S[i] = rand()%10;
		printf("%d ",S[i]);
	}
	printf("\n");

	printf("Aranan sayiyi girin: ");
	scanf("%d",&sayi);

	bulunan_indeks = dogrusalArama(S, boy, sayi);

	if (bulunan_indeks == -1)
		printf("%d bulunamadi\n",sayi);
	else
		printf("%d sayisi %d konumunda bulundu\n",sayi, bulunan_indeks);

	free(S);

	return 0;
}
```

Örnek 3: *N* elemanı olan bir diziyi sıraladığımızı varsayalım. Verilen bir sayıyı bu dizi içinde bulan ve konumunun indeksini dönen bir fonksiyon yazalım. Eğer sayı dizide bulunamazsa, fonksiyon -1 dönmeli. Aramayı "ikili arama" algoritmasıyla yapalım.

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

void kdegistir(int *a, int *b) {
            int gecici;
            gecici = *a;
            *a = *b;
            *b = gecici;
}
void dizi_sirala(int *D, int boy) {
            int i, j, gecici;
            for (i = 0; i<boy; i++)
                  for (j = i+1; j < boy; j++)
                        if (D[i] > D[j])
                              kdegistir(&D[i], &D[j]);
}
int ikiliArama(int *A, int boy, int sayi) {
            int alt = 0, ust = boy-1, orta;
            while (alt <= ust) {
                  orta = (alt + ust)/2;
                  if (A[orta] == sayi)
                        return orta;
                  if (A[orta] < sayi)
                        alt = orta + 1;
                  else
                        ust = orta - 1;
            }
            return -1;
}
```

```
int main() {
            int *S, boy, sayi, i, bulunan_indeks;

            printf("Dizinin boyunu gir: ");
            scanf("%d",&boy);
            S = (int *)malloc(boy*sizeof(int));
            for (i=0; i<boy; i++) {
                 S[i] = rand()%10;
                 printf("%d ",S[i]);
            }
            printf("\n");

            dizi_sirala(S,boy);

            for (i=0; i<boy; i++)
                 printf("%d ",S[i]);
            printf("\n");

            printf("Aranan sayiyi girin: ");
            scanf("%d",&sayi);
            bulunan_indeks = ikiliArama(S, boy, sayi);
            if (bulunan_indeks == -1)
                 printf("%d bulunamadi\n",sayi);
            else
                 printf("%d sayisi %d konumunda bulundu\n", sayi,
                       bulunan_indeks);

            free(S);

            return 0;
}
```

X. DİZGİLER

STRINGS

Dizgi (karakterler dizisi), son elemanı boş karakter olan `'\0'` ile gösterilen bir bellek blokudur. Boş karakter blokun sonunu bilmemize yarar. Bu özellik önemlidir çünkü programlarımızda metin bilgisini kullanma oldukça yaygındır. Bellek bloku bir karakter dizisi olarak tutulabilir veya ilk karakterin adresi bir karakter işaretçisi değişkende tutulabilir. `P` diye bir karakter işaretçimiz olduğunu varsayalım, bir dizgi değil. İçeriği bastırmak için boyu (karakter sayısını) bilmeliyiz. Bu iş için gereken kod aşağıda gösteriliyor.

```
for (i = 0; i<boy; i++)
           printf("%c", P[i]);
```

Eğer dizgileri kullanırsak, `P`'nin son elemanını boş karakter (`'\0'`) olarak belirlemeliyiz. Böyle olursa, dizinin boyunu bilmeye gerek olmaz. Kodumuz aşağıdaki gibi olur:

```
for (i = 0; P[i] != '\0'; i++)
           printf("%c", P[i]);
```

Aslında, C dilinde dizgiler için birçok kolaylık bulunmaktadır. Örneğin, `printf()` fonksiyonu dizgiler üzerinde doğrudan uygulanabilir, dizgilerin dönüşüm belirteci `%s`'dir. Kod aşağıdadır:

```
printf("%s", P);
```

Bir dizgi sabiti çift tırnaklar (`"…"`) arasında verilebilir (`printf()` fonksiyonunda çift tırnaklar arasında verilen dizgileri hatırlayalım). Dizgi sabiti olan `"merhaba dunya"` bir karakterler dizisidir. İç gösterimde, dizi boş karakterle sonlandırılır, böylece program dizginin sonunu bulabilir. Bir dizginin uzunluğu (boyu), dizgideki karakterlerin sayısı artı 1'dir (boş karakter için). Dizgilerin belirtilmelerini (tanımlanmalarını) inceleyelim:

1. `char a[] = "mavi";`

   Bu bildirim 5 boylu bir dizi yaratır ve dizinin içine ilk değerler atar. Bu aşağıdaki deyimle tamamen aynıdır.

   ```
   char a[] = {'m','a','v','i','\0'};

   a[0] = 'a'; /* a "aavi" oldu */
   a[2] = '+'; /* a "aa+i" oldu *
   a[6] = 'a'; /* altinci elemana erismeyi denememeliyiz
                  cunku burasi ayrilmis bir konum degil */
   a[4] = '+'; /*  bu olabilir ama dizginin sonlandigi noktayi
                  kaybederiz */
   ```

Karakter dizilerinde olduğu gibi, dizgilerin elemanlarına teker teker erişilebilir ve değiştirilebilir, ama dikkat edilmeli. Boş karakter değiştirilirse, son eleman bilgisi kaybolur.

2. `char *p = "mavi";`

   `p` değişkeni, bellekte herhangi bir yerde olan `"mavi"` dizgisine işaret etmektedir. Dizginin içeriğini değiştirmeye çalışırsak, sonuç tanımsızdır. Ama `p` işaretçisini değiştirebilir ve bellekte başka bir yere işaret etmesini sağlayabiliriz, örneğin `p = "kirmizi"`.

3. `char *p;`

   Önce dizgi için gereken yeri ayırırız, sonra da dizginin içeriğini değiştirmek güvenli olur. Boş karakter (`'\0'`) için yer ayırmayı unutmamalıyız.

```
char *p;
p = (char *)malloc(5*sizeof(char));
p[0] = 'm';
p[1] = 'a';
p[2] = 'v';
p[3] = 'i';
p[4] = '\0';
p[1] = 's';       /* dizginin icerigini degistirebiliriz */
```

Örnek 1: Verilen bir dizginin uzunluğunu hesaplayan fonksiyon yazalım.

```
int strUzunluk(char *str) {
        int i = 0;
        while (str[i]!='\0')
             i++;
        return i;
}
```

Örnek 2: Bir kaynak dizgisini bir hedef dizgiye kopyalayan bir fonksiyon yazalım. Hedef dizgi için daha önce bellekte yer ayrıldığını varsayabiliriz.

```
void strKopyala(char *kaynak, char *hedef) {

         int i = 0;
         while (kaynak[i] != '\0') {
            hedef[i] = kaynak[i];
            i++;
         }
         hedef[i] = '\0';
}
```

Örnek 3: Örnek 1 ve Örnek 2'de tanımlanan fonksiyonları (`strUzunluk()` ve `strKopyala()`) kullanarak, `"mavi"` dizgisini `P` işaretçisi tarafından gösterilen bir konuma kopyalayalım. Sonra `P` ile gösterilen blokun içeriğini başka bir bellek blokuna kopyalayalım.

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

int main() {

        char *P, *C;
        P = (char *)malloc((strUzunluk("mavi")+1)*sizeof(char));
        strKopyala("mavi",P);
        printf("P = %s\n", P);
        C = (char *)malloc((strUzunluk(P)+1)*sizeof(char));
        strKopyala(P, C);
        printf("C = %s\n", C);
        return 0;
}
```

Aslında, `<string.h>` kitaplığında tanımlanmış yararlı fonksiyonlar vardır. Bazılarını inceleyelim:

1. `strlen(char *S): S` dizgisindeki karakter sayısını döner (boş karakteri saymadan).

2. `strcpy(char *hedef, char *kaynak): kaynak` dizgisini `hedef` dizgisine kopyalar. Fonksiyon, boş karakter aktarıldıktan sonra durur. Fonksiyonu çağırmadan önce, hedef dizgisi için gereken yer ayrılmalıdır.

   ```
   char *a, *b = "dunya";
   int t;
   t = strlen(b);          /* t, 5 olur */
   a = (char *) malloc((strlen(b)+1)*sizeof(char));
   strcpy(a,b);            /* b dizgisi a'ya kopyalanir,
                           a'nin yeri daha once ayrilmis olmalidir */
   ```

3. `strcmp(char *S1,char *S2): S1` ve `S2` dizgilerini karşılaştırır ve

   `S1 < S2` ise < 0 döner
   `S1 == S2` ise = 0 döner
   `S1 > S2` ise >0 döner

   Örneğin; `"abc" < "abd", "dac" > "cbd"`, vs.

4. `strcat(char *hedef, char *kaynak): kaynak` dizgisinin bir kopyasını `hedef` dizgisinin sonuna ekler. Bu fonksiyonu çağırmadan önce, `hedef` dizgisinin boyunu artırmalıyız.

```
char *a, *b;
a = (char *)malloc((strlen("dunya")+1)*sizeof(char);
strcpy(a,"dunya");      /* "dunya" a'ya kopyalanir */
b = (char *)malloc((strlen("mars")+1)*sizeof(char);
strcpy(b,"mars");       /* "mars" b'ye kopyalanir */
a = (char *)realloc((strlen(a)+ strlen(b)+1)*sizeof(char);
strcat(a, b);
```

5. `getchar(char *S)`: standart girdi aracından (klavye) yeni satır (`'\n'`) karakteriyle sonlanan bir dizgi alır ve `S` dizgisi içine koyar. Yeni satır karakterini `S`'ye koyduğu bir boş karakter (`'\0'`) ile değiştirir. `S` için de yer ayrılmış olmalıdır.

6. `puts(char *S)`: Boş karakter ile sonlanan `S` dizgisini standart çıktı aracına (monitör) kopyalar ve bu kopyanın sonuna bir yeni satır ekler.

7. `int atoi (char *S)`: Bir `S` dizgisini karşılık gelen tamsayısına çevirir. Eğer dizgi bir tamsayıya çevrilemezse 0 dönülür.

   `atoi("345")` 345 tamsayısını döner.
   `atoi("abc")` 0 döner.
   `atoi("12ad3")` 12 döner.

8. `double atof (char *S)`: Bir `S` dizgisini karşılık gelen kayan nokta sayısına (floating point number = reel sayı) çevirir.

9. `char *strtok(char * S1, char *S2)`: `S1`'i `S2`'de bulunmayan ilk jeton için tarar. İlk çağrıda, `S1`'deki ilk jetonun ilk karakterine bir işaretçi döner ve `S1` içinde dönülen jetondan hemen sonrasına boş karakteri yazar. Bu fonksiyonu çağırdıktan sonra `S1` değiştiği için, kullanırken dikkatli olmak gerekir.

Örnek 4: `strtok()` kullanımı

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define max 100

int main() {

        char S[max], *gecici;
        char *jeton = " ,\t";
        int *sayilar = NULL, i = 0, j;
        printf("Girin: \n");
        gets(S);
        gecici = strtok(S, jeton);
        while (gecici != NULL) {
              sayilar = (int *)realloc(sayilar,(i+1)*sizeof(int));
```

```
            sayilar[i] = atoi(gecici);
            i++;
            gecici = strtok(NULL, jeton);
        }

        for (j=0; j<i; j++)
        printf("%d ",sayilar[j]);
        printf("\n");
        free(gecici);
        free(sayilar);

        return 0;
}
```

10. `sprintf()`: Biçimlendirilmiş çıktıyı bir dizgiye gönderir.

```
char *S = (char *)malloc(100 * sizeof(char));
float derssaati = 0.8;
sprintf(S, "CCC%d dersi haftada %f saat\n", 999, 4*derssaati);
```

11. `sscanf()`: Verilen dizgiden değerler okur

```
gets(S);
sscanf(S, "%d%f", &a, &b);
```

Kullanıcının `"12 3.4 4.5"` girdiğini varsayarsak `a`, 12; `b`, 3.4 değerlerine sahip olur.

XI. ÇOK BOYUTLU DİZİLER, YAPILAR
MULTIDIMENSIONAL ARRAYS, STRUCTURES

Çok boyutlu diziler

Çok boyutlu diziler tanımlayabiliriz. İki boyutta, çok boyutlu diziler bellek konumlarını bir tablo olarak düşünmemize olanak tanır. Esasında, bellek konumlarında bir tablo biçiminde veri tutamayız ve iki boyutlu diziler elemanları satır satır sıralanmış olan tek boyutlu dizilerdir. Bildirim (tanım) sırasında, çok boyutlu dizilerin bütün boyutlarını belirtiriz. Aşağıdaki iki boyutlu dizi bildirimleri var:

```
int a[3][4];            /* a dizisini 3 satır ve 4 sutundan oluşan
                        bir tablo olarak dusunebiliriz, ama
                        aslında a'nin elemanlarini tutmak icin 12
                        tane komsu oda ayrilmistir */

int b[3][5]  = {{0, 1, 2}, {3}}     /* Cok boyutlu dizilerin
                                    elemanlarina da ilk değerler
                                    atayabiliriz. Bir boyutlu dizilerde
                                    oldugu gibi, butun deger atanmamis
                                    elemanlar 0 ile doldurulur */

int c[][5] = {{1, 2},{3, 4}}        /* Bu durumda ilk boyut 2'dir */

int a[][] = {{1, 2},{3, 4},{5, 6}}        /* Bu olmaz, yalnız ilk boyut
                                          belirtilmemis olabilir */

int a[3][4][5];         /* Uc(3) boyutlu bir dizi. Aslında 60 tane
                        komsu olan tamsayi buyuklugunde bellek
                        hucresinden olusur */
```

İki boyutlu bir dizi bir fonksiyona geçilecekse, geçilen değer elemanları tek boyutlu diziler olan bir diziyi gösteren bir işaretçidir. Fonksiyondaki parametre bildiriminde, dizinin ilk boyutu hariç diğer bütün boyutlar belirtilmelidir. Aşağıdaki fonksiyon bildirimlerine ve fonksiyon çağrılarına bakalım:

```
void fonk1(int a[][]) {}      /* Olmaz. Yalnız ilk boyut bos olabilir */

void fonk2(int a[3][4]) {}    /* Gecerli */

void fonk3(int a[][4]) {}     /* Gecerli */

int a[3][7], b[5][4], c[3][4];

fonk3(a);                     /* Gecersiz */

fonk3(b);                     /* Gecerli */

fonk3(c);                     /* Gecerli */
```

Örnek 1: Her biri 2 daire içeren 3 binadan oluşan bir bloğu ele alalım. Önce her dairedeki insan sayısını tutmak istiyoruz, sonra her binada yaşayan insanların ortalama sayısını hesaplamak istiyoruz.

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

void insan_sayilarini_oku(int blok[][2],int binalar, int daireler) {
      int i, j;
      for (i=0; i<binalar; i++)
            for (j=0; j<daireler; j++)
                  scanf("%d",&blok[i][j]);
}

void binalardaki_toplam_nufusu_bul(int blok[][2], int toplam[], int
                              binalar, int daireler) {
      int i, j;
      for (i=0; i<binalar; i++) {
            toplam[i] = 0;
            for (j=0; j<daireler; j++)
                  toplam[i]+=blok[i][j];
      }
}

int main() {
      int Blok[3][2], Toplam[3],i;
      insan_sayilarini_oku(Blok,3,2);
      binalardaki_toplam_nufusu_bul(Blok,Toplam,3,2);

      for (i=0; i<3; i++)
            printf("%d ",Toplam[i]);
      printf("\n");
      return 0;
}
```

Örnek 2: Boyutları 2x4 olan iki tane iki-boyutlu matris var. Bu iki matrisin toplamını hesaplayan bir fonksiyon ve verilen bir matrisin çaprazlamasını (transpose) bulan başka bir fonksiyon yazalım.

```
#include <stdio.h>


void toplam(int A[][4], int B[][4], int sonuc[][4], int satir, int
            sutun) {
      int i,j;
      for (i=0; i<satir; i++)
            for (j=0; j<sutun; j++)
                  sonuc[i][j] = A[i][j]+B[i][j];
}
void caprazlama_bul(int A[2][4],int cap[4][2], int satir, int sutun) {
      int i,j;
      for (i=0; i<satir; i++)
            for (j=0; j<sutun; j++)
                  cap[j][i] = A[i][j];
}
int main(){
      int M1[2][4] = {{1,2,3,4},{5,6,7,8}};
      int M2[2][4] = {{10,20,30,40},{50,60,70,80}};
      int Toplam[2][4];
      int Capraz[4][2];
      int i,j;


      toplam(M1,M2,Toplam,2,4);
      caprazlama_bul(Toplam,Capraz,2,4);


      for (i=0; i<2; i++) {
            for (j=0;j<4; j++)
                  printf("%d ",Toplam[i][j]);
            printf("\n");
      }
      printf("\n");
```

```
	for (i=0; i<4; i++) {
		for (j=0;j<2; j++)
			printf("%d ",Capraz[i][j]);
		printf("\n");
	}
	return 0;
}
```

Örnek 3: Her birinde 3 öğrenci bulunan 2 sınıf var. Bu 3 öğrenci 4 ders alıyor (farklı sınıflardaki bütün öğrenciler aynı dersleri alıyor).

1. Kullanıcıdan her öğrencinin notlarını alan bir fonksiyon yazalım.
2. 4 dersin ortalamasını hesaplayan bir fonksiyon yazalım.

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

void notlari_oku(float notlar[][3][4], int siniflar, int ogrenciler,
	int dersler) {
	int i,j,k;
	for (i=0; i<siniflar; i++)
		for (j=0; j<ogrenciler; j++)
			for (k=0; k<dersler; k++)
				scanf("%f",&notlar[i][j][k]);
}

float *derslerin_ortalamasini_bul(float notlar[][3][4], int
		siniflar, int ogrenciler, int dersler) {
	int i,j,k;
	float *ort;
	ort = (float *)malloc(dersler * sizeof(float));
	for (i=0; i<dersler; i++) {
		ort[i] = 0.0;
		for (j=0; j<siniflar; j++)
```

```
                  for (k=0; k<ogrenciler; k++)
                        ort[i]+=notlar[j][k][i];
            ort[i]/= siniflar*ogrenciler;
      }
      return ort;
}

int main(){
      float notlar[2][3][4], *ders_ort;
      int i;
      notlari_oku(notlar,2,3,4);
      ders_ort = derslerin_ortalamasini_bul(notlar,2,3,4);

      for (i=0; i<4; i++)
            printf("%f ",ders_ort[i]);
      printf("\n");
      return 0;
}
```

Yapılar

Bir *yapı* (structure), bir veya daha çok değişkenden oluşan tek ad altında gruplanmış bir topluluktur. Topluluktaki değişkenlerin veri tipi farklı olabilir. Yapılar türetilmiş veri tipleridir.

```
struct vektor {
      int elemanSayisi;
      float *elemanlar;
};
```

Yukarıdaki tanımda, `struct` yapı bildirimi için kullanılan anahtar sözcüktür. `vektor`, yapıya ait etikettir. Küme parantezleri arasındaki bildirimler `vektor` yapısının üyeleridir. Üyeler temel tiplerden olabilir (`int, short, double, char`, vs.) veya başka yapılardan olabilir. Bir yapı kendisinin bir üyesi olamaz ama kendi türünden bir yapıya bir işaretçiyi üye olarak içerebilir. Yapı etiketleri (yapıların adları) aynı kapsam içinde tek olmalıdır. Öte yandan, farklı yapılar aynı ada sahip olan üyeler bulundurabilir. Bildirdiğimiz `vektor` yapısını ele alalım. Yapı veri tipindeki değişkenler aşağıdaki gibi bildirilebilir (tanımlanabilir):

```
struct vektor a, b;
struct vektor d[20], *c, k;
```

VEYA

```
struct vektor {
      int elemanSayisi;
      float *elemanlar;
} a, b;
```

Bir yapıya işaretçi tanımlar ve onu bir dizi olarak kullanmak istersek, bellekte yer ayırmak gerekir. Yapıları iç içe yerleştirmek de mümkündür. Aşağıdaki örneklere bakalım:

```
struct nokta {
      int X, Y;
};
struct dogru {
      struct nokta N1, N2;
};
struct cember {
      struct nokta merkez;
      float yaricap;
};
```

Bir yapının üyelerine erişebiliriz. Bir üyeye erişmek için,

```
yapi_degiskeninin_adi.üye_adi
```

sözdizimi kullanılır. Örneğin,

```
struct vektor {
      int elemanSayisi;
      float *elemanlar;
};
struct vektor V, *W, U[10];

V.elemanSayisi = 4;
V.elemanlar = (float *) malloc(4 * sizeof(float));
U[0].elemanSayisi = 2;
```

```
U[8].elemanSayisi = 3;
U[8].elemanlar = (float *) malloc(3 * sizeof(float));
U[8].elemanlar[0] = 4.5;
U[8].elemanlar[2] = -44.5;
W = (struct vektor *)malloc(2 * sizeof (struct vektor));
(*W).elemanSayisi = 4;
/* VEYA ALTERNATIF OLARAK */
W->elemanSayisi=4; /* -> yalniz yapi isaretcileriyle kullanilir */
*W.elemanSayisi = 4     /*olmaz */
```

Aynı veri tipinden yapı değişkenleri kopyalanabilir. Aşağıdakilerin tümü geçerlidir.

```
struct nokta {
      int X, Y;
};
struct dogru {
      struct nokta N1, N2;
};
struct dogru L, K;
struct nokta n1;

L.N1.X = 3;
L.N1.Y = 4;
n1.X = 5;
n1.Y = 7;
L.N2 = n1;
K = L;
L.N2 = K.N1;
```

Bir yapı değişkeninin adresini de alabiliriz. Yapının adresini tutan yapı işaretçilerini veya teker teker elemanları yapılar olan yapı dizilerini kullanabiliriz. Bir işaretçi tanımlanırsa, onun için yer ayırmak gerekir. Gerekirse, ayrılan yeri de serbest bırakmalıyız.

```
struct nokta {
      int X, Y;
};
struct nokta *dikdortgen;
dikdortgen = (struct nokta*)malloc(4 * sizeof(struct nokta));
dikdortgen[0].X = 5;
free(dikdortgen);


struct OGR {
      long numara;
      char *ad, *dogumYeri;
      int dersSayisi;
      int *notlar;
      float GNO;
};
typedef struct OGR ogrenci;


ogrenci **Sinif
Sinif = (ogrenci **)malloc(4 * sizeof(ogrenci *));
for (i = 0; i < 4; i++)
      Sinif[i] = (ogrenci *)malloc(50 * sizeof(ogrenci));
Sinif[3][5].numara = 2018001;
Sinif[3][5].dersSayisi = 6;
Sinif[3][5].notlar = (int *)malloc(Sinif[3][5].dersSayisi *
            sizeof(int));
…
for (i = 0; i < 4; i++)
      free(Sinif[i]);
free(Sinif);
```

Yapılar fonksiyonlara geçilebilir. Yapıların bütününü veya üyelerini fonksiyonlara geçebiliriz (*değer ile çağırma* = call by value). Yapıların adresini fonksiyonlara geçebiliriz (*başvuru ile çağırma* = call by reference). Fonksiyonlar bir yapı veya yapı işaretçisi (adres) dönebilir.

Örnek 4: Bu örnekte, polinomlar için temel işlemleri gerçeklememiz isteniyor. Polinomlar için gereken tanımı ve aşağıdaki fonksiyonları bulunduran bir program yazalım:

1. Derecesi parametre olarak verilen bir polinom için gerekli belleği ayıran fonksiyon
2. Kullanıcıdan bir polinomun katsayılarını okuyan fonksiyon
3. Bir polinomun katsayılarını ekrana çıktılayan fonksiyon
4. İki polinomu toplayan fonksiyon

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

struct polinom {
	int derece;
	float *katsayilar;
};
typedef struct polinom poli;

poli yer_ayir(int sayi) {
	poli gecici;
	gecici.derece = sayi;
	gecici.katsayilar =(float*)malloc((gecici.derece+1)*sizeof(float));
	return gecici;
}

void poli_oku(poli *P) {
	int i;
	for (i=0; i<=P->derece; i++) {
		printf("%d. dereceli terimin katsayisini girin: ",i);
		scanf("%f",&(P->katsayilar[i]));
	}
}
void poli_yazdir(poli P) {
	int i;
	for (i=P.derece; i>0; i--)
```

```
            if (P.katsayilar[i] != 0)
                  printf("%f x^%d + ",P.katsayilar[i],i);
      printf("%f\n",P.katsayilar[0]);
}

poli poli_topla(poli P1, poli P2) {
      poli gecici;
      int max, i;

      if (P1.derece > P2.derece)
            max = P1.derece;
      else
            max = P2.derece;

      gecici = yer_ayir(max);

      for (i=0; i<=gecici.derece; i++) {
            gecici.katsayilar[i] = 0.0;
            if (P1.derece >= i)
                  gecici.katsayilar[i] += P1.katsayilar[i];
            if (P2.derece >= i)
                  gecici.katsayilar[i] += P2.katsayilar[i];
      }
      return gecici;
}

int main() {
      poli P1,P2,P3;
      P1 = yer_ayir(4);
      printf("POLINOM 1\n");
      poli_oku(&P1);
      P2 = yer_ayir(5);
```

```
	printf("POLINOM 1\n");

	poli_oku(&P2);

	P3 = poli_topla(P1,P2);


	poli_yazdir(P1);

	poli_yazdir(P2);

	poli_yazdir(P3);

	free(P1.katsayilar);

	free(P2.katsayilar);

	free(P3.katsayilar);

	return 0;

}
```

Örnek 5: Programda doğruları tutacağız. İki uç noktasını alarak bir doğru yaratmak, bazı noktalarda karşılıklı değiştirme işini yapmak ve bir doğrunun uzunluğunu hesaplamak istiyoruz.

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

struct nokta {
	int X,Y;
};
struct dogru {
	struct nokta n1,n2;
};

struct nokta nokta_yap(int a, int b) {
	struct nokta gecici;
	gecici.X = a;
	gecici.Y = b;
	return gecici;
}

void kdegistir_noktalar(struct nokta *n1, struct nokta *n2) {
	struct nokta gecici;
	gecici = *n1;
	*n1 = *n2;
	*n2 = gecici;
}

float uzunluk(struct dogru L) {
	float d;
	d = sqrt((L.n1.X-L.n2.X)*(L.n1.X-L.n2.X)+(L.n1.Y-L.n2.Y)*(L.n1.Y-L.n2.Y));
	return d;
}
```

```
int main() {

	struct nokta a,b;
	struct dogru L;
	float k;

	a = nokta_yap(0,0);
	b = nokta_yap(3,4);
	kdegistir_noktalar(&a,&b);
	L.n1 = a;
	L.n2 = nokta_yap(6,8);
	k =  uzunluk(L);

	printf("UZUNLUK = %f\n",k);

	return 0;
}
```

- `typedef` yeni veri tipleri yaratır. Örneğin,

```
typedef int uzunluk;
```

Burada `uzunluk`, `int` ile aynı anlamı taşır hale geliyor. Aşağıdaki iki bildirim de geçerli ve aynı anlama gelmektedir.

```
int uzunluk1;

uzunluk uzunluk1;
```

`typedef` genelde karmaşık veri tipleri için gereklidir. Örneğin,

```
struct vektor {

	int elemanSayisi;

	float *elemanlar;

};

typedef struct vektor VektorTipi;
```

Bu tanımlamadan sonra, bildirimlerde `VektorTipi` bir veri tipi olarak kullanılabilir:

```
VektorTipi V, v[40], *vv;
```

`typedef` yeni bir veri tipi yaratmaz. Zaten var olan bir tip için yeni bir ad kullanabilmemizi sağlar.

XII. DOSYALAR, ÖZYİNELEME
FILES, RECURSION

Dosyalar

Şimdiye kadar, veriyi hep standart girdi aracından (klavye) okuduk ve hep standart çıktı aracına (monitör) yazdık. Bu işlemler, işletim sistemi tarafından programlar için otomatik olarak tanımlanmıştır. Bunun ötesinde, programlar kendilerine bağlanmış dosyalara da erişebilir.

Dosya hakkında bilgi içeren bir yapı vardır (bir arabellek (buffer) konumu (örneğin, kalıcı bellek olan hard diskte bir yer), arabellekteki güncel konum, okuma/yazma belirteçleri, vs.). Bu yapıya işaret eden bir *dosya işaretçisi* (file pointer) vardır. Bir dosyanın bildirimi aşağıdaki gibidir:

```
FILE *id;
```

`FILE *` veri tipi `<stdio.h>` kitaplığında tanımlıdır. Bir dosyaya erişmeden önce, `fopen` fonksiyonu ile ona bağlanmalıyız. Dosya adını (*dizin*i (directory) ile beraber) ve işlem kipini (modunu) bu fonksiyonda belirtmeliyiz. Kip; yazma (`w`), okuma (`r`), sonuna ekleme (`a`), vs. olabilir. Örneğin, bir dosyayı okuma kipinde açıp içine veri yazmaya çalışırsak, sorunla karşılaşırız. Aşağıdaki örnek *veri.txt* adındaki dosyayı yazma işlemi için programa bağlar. Hata olursa, örneğin belirtilen isimde bir dosya bulunamazsa, fopen fonksiyonu `NULL` döner.

```
FILE *fid;

fid = fopen("veri.txt","w");
```

Dosyalar üzerinde tanımlanmış bazı fonksiyonlar aşağıdadır. Bu işlemleri yapmadan önce, karşılık olan dosyaya bağlanmak, yani o dosyayı açmak gerekir.

1. `fprintf`: Açılmış olan bir dosyanın içine biçimlendirilmiş veriyi yazar

```
fprintf(fid, "%dMerhaba dunya!\n%f\t%s\n",12,18.78,str);
```

2. `fscanf`: Açılmış olan bir dosyadan veri okur

```
fscanf(fid, "%d%c", &x, &karakter);
```

3. `fgets`: `s` dizgisine *dosya katarı*ndan (file stream) karakterler okur. `n-1` karakter okunduğunda veya yeni satır (`'\n'`) karakteri okunduğunda durur (hangi olay önce olursa). Başarılı olursa, `s`'nin işaret ettiği dizgiyi döner. Hata olursa veya dosyanın sonuna gelinirse, `NULL` döner.

```
char *fgets(char *s, int n, FILE *katar);
```

4. `fputs`: `s` dizgisini verilen çıktı katarına kopyalar. Yeni satır karakteri eklemez ve sonlandıran boş karakteri (`'\0'`) kopyalamaz. Başarı durumunda, yazılan son karakteri döner. Başarısız olunursa, `EOF` (end-of-file) döner.

```
int fputs(const char *s, FILE *katar);
```

5. `EOF`: Bir dosyanın sonuna ulaşıldığını gösteren sabit değerdir.

Dosya işaretçisi ve dışarıdaki ad (dosyanın adı) arasındaki bağlantıyı, `fclose` fonksiyonunu kullanıp dosya işaretçisini serbest bırakarak, koparmalıyız.

```
fclose(fid);
```

- Programda açılacak dosyaların sayısı üzerinde kısıtlamalar vardır, bu yüzden programımızda açılmış olan ve ihtiyaç duyulmayan dosyaları kapatmalıyız.

- Program sonlanınca, `fclose` açılmış olan her dosya için otomatik olarak çağırılır.

- Açılmış olmayan bir dosyayı kapatmaya çalışmayalım. Sorun çıkabilir.

Örnek 1: Bu örnekte, kullanıcının girdiği veri dosyaya yazılıyor ve sonra bu veri dosyadan okunuyor. Bu amaçlar için, dosyaya yazma ve dosyadan okuma işlerini yapan fonksiyonlar yazalım.

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

void dosyayaYaz(char *dosya_adi) {

	FILE *id;
	int n, i;
	double x;

	id = fopen(dosya_adi,"w");

	printf("Girilecek veri sayisi: ");
	scanf("%d",&n);

	for (i=0; i<n; i++) {
		printf("Veri gir: ");
		scanf("%lf",&x);
		fprintf(id,"%lf\n",x);
	}
	fclose(id);
}

double *dosyadanDiziyeOku(char *dosya_adi,int *n) {

	FILE *dosyam = fopen(dosya_adi, "r");
```

```
	double *sayilar = NULL, gecici;

	*n = 0;

	while (fscanf(dosyam,"%lf",&gecici) != EOF) {
		(*n)++;
		sayilar =(double*)realloc(sayilar,(*n)*sizeof(double));
		sayilar[(*n)-1] = gecici;
	}

	fclose(dosyam);
	return sayilar;
}

int main() {

	double *SAYILAR;
	int n, i;
	dosyayaYaz("bilgi.txt");
	SAYILAR = dosyadanDiziyeOku("bilgi.txt",&n);

	for (i=0; i<n; i++)
		printf("%.2lf\n",SAYILAR[i]);

	free(SAYILAR);

	return 0;
}
```

Özyineleme

Örnek 2: Faktoriyel hesaplamayı özyinelemeli fonksiyon ile yapalım.

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


long faktoriyel(int n) {
	if (n<0)
		return -1;
	if (n == 0 || n == 1)
		return 1;
	else
		return (n*faktoriyel(n-1));
}
```

```
int main() {
	int sayi;
	printf("Sayi gir: ");
	scanf("%d", &sayi);
	printf("%d! = %ld\n",sayi,faktoriyel(sayi));
	return 0;
}
```

Örnek 3: Fibonacci dizisinin elemanlarını özyinelemeli fonksiyon ile hesaplayalım. Fibonacci dizisinde her sayı kendisinden önceki iki sayının toplamıdır. Fibonacci(0) = 0 ve Fibonacci(1) = 1.

Yani, Fibonacci dizisi şöyledir:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, …

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

int fibonacci(int n) {
	if (n<0)
		return -1;
	if (n == 0 || n == 1)
		return n;
	else
		return (fibonacci(n-1)+fibonacci(n-2));
}

int main() {
	int sayi;
	printf("Sayi gir: ");
	scanf("%d", &sayi);
	printf("fibonacci(%d) = %d\n",sayi,fibonacci(sayi));
	return 0;
}
```