



BİLGİSAYAR PROGRAMLAMA 1

Ders Notu 10 – İşaretçiler

Konya Teknik Üniversitesi
Elektrik – Elektronik Mühendisliği Bölümü

6.05.2024

Konya

C dili bir değişkenin değerine erişmek için iki farklı yöntem sağlar:

- **Direkt erişim:** Şimdiye kadar yazdığımız kodlardaki gibi, bir değişkenin adını kullanarak o değişkenin değerine erişmeye denir.
- **Dolaylı erişim:** İşaretçiler aracılığıyla değişkenin bellekteki adresi üzerinden ilgili değere erişime denir.

İşaretçiler

- Bellekteki her byte'ın hexadecimal sistemde ifade edilen bir adresi vardır. **İşaretçi**, bellek alanındaki **adreslerin** saklandığı bir değişkendir. İşaret değişkenlerine referans değişken de denir.
- İşaretçilere veriler değil, verilerin bellekte saklı olduğu gözlerin başlangıç adresi atanır.
- C dilinde özellikle programın çalıştırıldığı bilgisayarın iç belleğinin verimli bir şekilde kullanılmasını sağlayan **dinamik bellek** uygulamalarında ve katarlarla (string) çalışırken etkin bir şekilde kullanılmaktadır.

Örnek Verirsek;

okul_no = 453 değişkeni için;

Tipi = int

Adi = okul_no

Değeri = 453

Adresi = 1005

1-A dersliğine ait 1005 adresindeki sırayı sabittir. Derse değişik öğrencilerin gelmesi durumunda değişen sadece öğrenci numaralarıdır.

1-A DERSLİĞİ

1001, 123	1011, 789	1021, 823
1002, 752	1012, 111	1022, 901
1003, 696	1013, 222	1023, 903
1004, 678	1014, 333	1024, 907
1005, 453	1015, boş	1025, boş
1006, 287	1016, 899	1026, boş
1007, 900	1017, 890	1027, 278
1008, 876	1018, boş	1028, boş
1009, boş	1019, boş	1029, boş
1010, boş	1020, boş	1030, boş

İşaretçi (pointer) bildirimi

İşaretçi değişkeni tanımlanırken, aşağıdaki ifade kullanılır:

tip *degisken_adi; // *degisken_adi: bir adres işaretçisidir.

İşaretçi değişkeni, ilgili adrese hangi tip değişken saklanacak olursa olsun, bellekte aynı uzunlukta yer işgal eder. Ne kadar yer işgal edeceği (32 bit, 64 bit gibi), derleyicinin bellek modeline ya da kullanılan mikroişlemcinin adres uzunluğuna bağlı olabilir.

int *p;	_____→	Tamsayı saklamak için bir adres işaretçisi
float *q;	_____→	Gerçel sayı saklamak için bir adres işaretçisi
char *r;	_____→	Karakter saklamak için bir adres işaretçisi

- İşaretçiye bir değişkenin **adresini** atamak için **&** operatörü kullanılır:



```
int *p;  
float *q;  
char *r;
```

p = &t ; //t'nin adresi p değişkeninin
//içine atanıyor.

q = &f ;

r = &kr ;

- Yani **&** operatörü bir değişkenin önüne konmuşsa, o değişkenin değeri/içeriği değil, adresi bildiriliyor anlamına gelir. Öte yandan bir işaretçi adının önüne ***** operatörü konmuşsa; işaretçinin tuttuğu adresle değil, işaret ettiği yerdeki veri ile ilgileniliyor demektir.

*  İşaretçi bildirimi
  Değere erişim operatörü

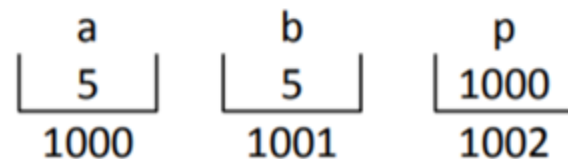
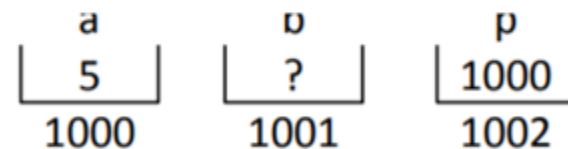
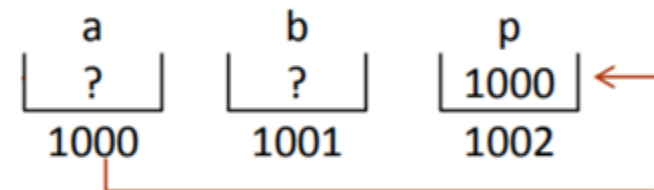
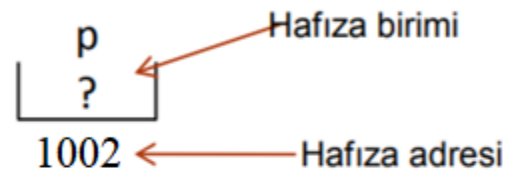
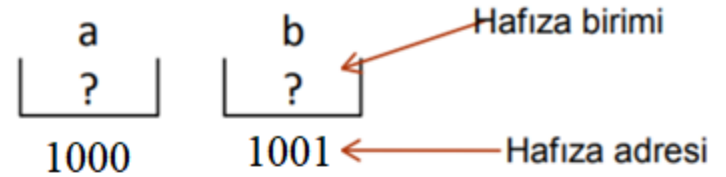

```
int a, b;
```

```
int *p;
```

```
p = &a;
```

```
*p = 5;
```

```
b = *p;
```



Örnek

```
int c, *pc;
```

```
/* pc' nin işaret ettiği adresteki değer c olsun istiyorsak: */
```

```
pc = c; // Yanlış! pc adres, ancak c bir adres değil.
```

```
*pc = c; // Doğru! *pc, pc adresindeki değerdir ve c de bir  
          değerdir (adres değil).
```

```
/* pc, c' nin adresini işaret etsin istiyorsak: */
```

Aynı işlevde

```
*pc = &c; // Yanlış! *pc, pc adresindeki değer, ancak &c bir adres.
```

```
pc = &c; // Doğru! pc bir adres ve &c de bir adres.
```


C dilinde oluşturduğumuz her tip hafızada belirli byte boyutunda yer kaplar.

- **32** bit işlemcili bilgisayarlarda;

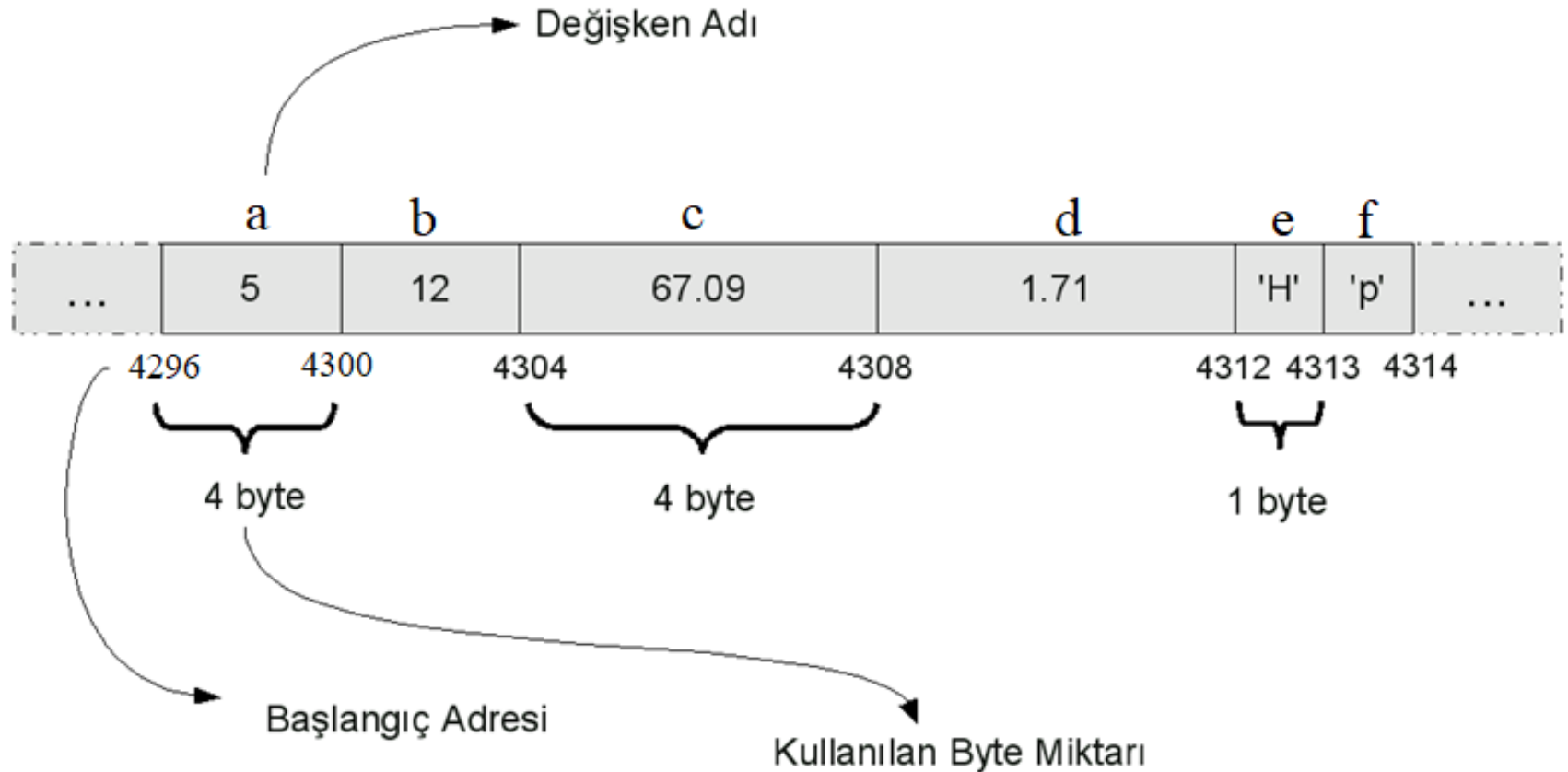
```
char    = 1 byte  
int     = 4 byte  
float   = 4 byte  
double  = 8 byte
```

- 64 bit sistemlerde ise, değerler iki katı olarak değişir.

Pointerlar ise tipe bakmaksızın her zaman;

- **32** bit sistemlerde 4 byte,
- **64** bit sistemlerde 8 byte yer kaplar.

```
int a = 5;  
int b = 12;  
float c = 67.09;  
float d = 1.71;  
char e = 'H';  
char f = 'p';
```




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

```
main() {
```

```
    int deg, *isaret;
```

```
    deg = 999;
```

```
    isaret = &deg ;
```

```
    printf(“degiskenin adresi: %p\n”, isaret);
```

```
    printf(“degiskenin degeri: %d\n”, *isaret);
```

```
    printf(“degiskenin adresi: %p\n”, &deg);
```

```
    printf(“degiskenin degeri: %d\n”, deg);
```

```
    return 0; }
```

Adres

Bellek

FAFF

...

FB00

FB01

999

FB02

FB03

...

```
degiskenin adresi: 000000000062FE14
degiskenin degeri: 999
degiskenin adresi: 000000000062FE14
degiskenin degeri: 999
```

```
-----
Process exited after 0.04861 seconds with return value 0
Press any key to continue . . .
```

```
#include<stdio.h>
main() {
    int sayi = 15;
    printf("sayi degiskeninin degeri: %d\n", sayi);
    printf("sayi degiskeninin bellek adresi: %p\n", &sayi);

    int *p = NULL;
    printf("isaretci degiskeninin tanimlama esnasinda bellek adresi: %p\n", p);

    p = &sayi;
    printf("isaretci degiskeninin yeni bellek adresi: %p\n", p);
    printf("isaretci degiskeninin tuttugu bellek adresindeki deger: %d\n", *p);

    *p = 16;
    printf("isaretci degiskeninin tuttugu bellek adresindeki yeni deger: %d\n", *p);
    printf("sayi degiskeninin yeni degeri: %d\n", sayi);

    return 0; }
```

```
sayi degiskeninin degeri: 15
sayi degiskeninin bellek adresi: 0060FF34
isaretci degiskeninin tanimlama esnasinda bellek adresi: 00000000
isaretci degiskeninin yeni bellek adresi: 0060FF34
isaretci degiskeninin tuttugu bellek adresindeki deger: 15
isaretci degiskeninin tuttugu bellek adresindeki yeni deger: 16
sayi degiskeninin yeni degeri: 16
```



```
#include<stdio.h>
main() {
    int sayi = 100;
    int *ref;
    ref = &sayi ;
    int sayi2 = *ref;
    *ref = 150;
    printf("sayi degiskeninin degeri: %d\n", sayi);
    printf("sayi2 degiskeninin degeri: %d\n", sayi2);
    return 0; }
```

```
sayi degiskeninin degeri: 150
sayi2 degiskeninin degeri: 100
```

```
-----
Process exited after 0.03967 seconds with return value 0
Press any key to continue . . .
```

İşaretçi aritmetiği

- İşaretçiler kullanılırken, işaret edilen adres temel alınıp o adresten önceki ya da sonraki adreslere erişilmesi istendiğinde aritmetik işlemlerden yararlanılabilir.
- İşaretçiler üzerinde yalnız bir tam sayı ile **toplama**, bir tam sayı **çıkarma**, **artırma** ve **azaltma** operatörleri kullanılabilir (+, ++, -, --).
- İşaretçilerin bu operatörlere verdiği tepki, her bir veri tipi için farklıdır. Örneğin 32 bit işlemcili bilgisayarlarda int * tipindeki bir işaretçiye ++ operatörü uygulanırsa adres değeri *4 byte*, char * tipine uygulanırsa *1 byte* artacaktır.


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

```
main() {
```

```
    int tsayi = 17;
```

```
    double osayi = 21.3;
```

```
    char kr = 'C';
```

```
    int *intPtr;
```

```
    double *doublePtr;
```

```
    char *charPtr;
```

```
    intPtr = &tsayi ;
```

```
    doublePtr = &osayi;
```

```
    charPtr = &kr;
```

```
printf("Tam sayi isaretcisinin ilk degeri: %p\n", intPtr);
```

```
printf("Ondalik sayi isaretcisinin ilk degeri: %p\n", doublePtr);
```

```
printf("Karakter isaretcisinin ilk degeri: %p\n", charPtr);
```

```
printf("-----\n");
```

```
    intPtr ++;
```

```
    doublePtr --;
```

```
    charPtr ++;
```

```
printf("Tam sayi isaretcisinin son degeri: %p\n", intPtr);
```

```
printf("Ondalik sayi isaretcisinin son degeri: %p\n", doublePtr);
```

```
printf("Karakter isaretcisinin son degeri: %p\n", charPtr);
```

```
printf("-----\n");
```

```
    return 0; }
```

```
Tam sayi isaretcisinin ilk degeri: 000000000062FE04
```

```
Ondalik sayi isaretcisinin ilk degeri: 000000000062FDF8
```

```
Karakter isaretcisinin ilk degeri: 000000000062FDF7
```

```
-----
```

```
Tam sayi isaretcisinin son degeri: 000000000062FE08
```

```
Ondalik sayi isaretcisinin son degeri: 000000000062FDF0
```

```
Karakter isaretcisinin son degeri: 000000000062FDF8
```

```
-----
```

Bazı Uyarılar

- `*ip++;` /* İşaretçinin gösterdiği bellek adresini artırır. */
`(*ip)++;` /*İşaretçinin gösterdiği adresteki değişken değerini artırır. */
- İki işaretçi direkt toplanamaz ve çarpılamaz. Mantıksal olarak iki işaretçinin toplanması ya da çarpılmasının bir anlamı yoktur.
- İşaretçi değişkenler üzerinde karşılaştırma operatörleri (`<`, `>`, `<=`, `>=`, `==`, `!=`) kullanılabilir.
 - İki işaretçi değeri birbirine eşitse, aynı bellek alanındaki değere işaret ediyorlar demektir.
 - Bir işaretçinin değeri bir diğer işaretçinin değerinden küçükse, bellekte daha önce gelen bir alandaki değeri işaret ediyor demektir. Dizilerde önceki/sonraki elemanın belirlenmesi, sınır aşımı kontrolü gibi uygulamalarda kullanılabilir.

İşaretçiler & Diziler

- İşaretçilerin kullanım amaçlarından biri de dizilerin daha kolay ve verimli şekilde kullanılmasını sağlamaktır.
- Dizideki elemanların tipleriyle uyumlu bir işaretçiye dizinin ilk elemanının adresi kopyalandığında, bu işaretçi aracılığıyla dizinin tüm elemanlarına erişilebilir.
- Dizinin adı, o dizinin bellekteki başlangıç adresine karşılık gelir.
- Dizinin bellekteki başlangıç adresi, dizinin ilk elemanının bellek adresidir.


```
#include<stdio.h>
main() {
    int dizi[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int *ptr = &dizi[0];
    int *ptr2 = dizi;
    int i;
```



```
printf("Dizinin 1. elemani: %d\n", dizi[0]);
printf("Dizinin 1. elemani: %d\n", *ptr);
printf("Dizinin 1. elemani: %d\n", *ptr2);
printf("-----\n");
printf("Dizinin 5. elemani: %d\n", dizi[4]);
printf("Dizinin 5. elemani: %d\n", *(ptr+4));
printf("Dizinin 5. elemani: %d\n", *(ptr2+4));
printf("-----\n");
printf("Dizinin 9. elemani: %d\n", dizi[8]);
printf("Dizinin 9. elemani: %d\n", *(ptr+8));
printf("Dizinin 9. elemani: %d\n", *(ptr2+8));
```

```
    for(i=0; i<10; i++) {
printf("Dizi[%d] -> Adresi: %p, Degeri: %d\n", i, ptr+i, *(ptr+i));    }
    return 0; }
```

```
Dizinin 1. elemani: 1
Dizinin 1. elemani: 1
Dizinin 1. elemani: 1
```

```
-----
Dizinin 5. elemani: 5
Dizinin 5. elemani: 5
Dizinin 5. elemani: 5
```

```
-----
Dizinin 9. elemani: 9
Dizinin 9. elemani: 9
Dizinin 9. elemani: 9
```

```
Dizi[0] -> Adresi: 00000000062FDE0, Degeri: 1
Dizi[1] -> Adresi: 00000000062FDE4, Degeri: 2
Dizi[2] -> Adresi: 00000000062FDE8, Degeri: 3
Dizi[3] -> Adresi: 00000000062FDEC, Degeri: 4
Dizi[4] -> Adresi: 00000000062FDF0, Degeri: 5
Dizi[5] -> Adresi: 00000000062FDF4, Degeri: 6
Dizi[6] -> Adresi: 00000000062FDF8, Degeri: 7
Dizi[7] -> Adresi: 00000000062FDFC, Degeri: 8
Dizi[8] -> Adresi: 00000000062FE00, Degeri: 9
Dizi[9] -> Adresi: 00000000062FE04, Degeri: 10
```

Adresler
arasında
4' er fark
olduğuna
dikkat !!!



Bir A[] dizisinin elemanlarının toplamını bulan bir fonksiyon yazmamız gerekirse;

```
int topla1(int A[], int n)
{
    int *p, toplam=0, sayac;
    p=&A[0];

    for(sayac=0; sayac<n; sayac++)
        toplam+=*(p+sayac);

    return toplam;
}
```

```
int topla2(int A[], int n)
{
    int *p, toplam=0;

    for(p=A; p<&A[n]; p++)
        toplam+=*p;

    return toplam;
}
```


- Matrisler, bilindiği gibi iki boyutlu dizilerdir. Matrisler üzerinde işaretçi atamaları yapmak da mümkündür:

```
int A[10][20], *p, *q, *r;
```

```
p = A;  
p = &A[0][0];
```



Aynı adres ?

```
q = &A[3][4];  
r = &A[9][19];
```


Dinamik bellek kullanımı nedir ?

Dizileri önceki haftalarda bahsettiğimiz klasik yöntemlerle tanımlarsak, derleyici bu dizi için gerekli bellek alanını, bildirildiği fonksiyon etkin olduğu sürece tutacak/meşgul edecek ve bu adresteki değer kullanılsa bile belleğin bu kısmını kullanılamaz hale getirecektir.

Eğer bu alan seyrek kullanılıyorsa ya da kullanılmadığı zamanlarda saklanan değerlerin bir önemi yoksa; bu alan kullanılmadığı zamanlarda serbest bırakılarak başka amaçlarla kullanılabilir. Bunun için belleğin dinamik kullanımı gereklidir.

Dinamik bellek kullanımı; gerekli bellek alanlarının program ya da fonksiyon başında ayrılması yerine, gerektiği anda istenmesi ve kullanılmadığında serbest bırakılması/iade edilmesidir. Boş bellek alanı istemek ya da bir alanı serbest bırakmak için standart kütüphanede bazı fonksiyonlar mevcuttur.

Dinamik belleğin **AVANTAJ**ları

- C dilinin bu kadar yaygın olmasının önemli sebeplerinden biri, işletim sistemi kaynaklarına en alt (detay) seviyede erişim imkanı sağlamasıdır. Bunda işaretçilerin payı büyük olup, kullanmayı bilene oldukça **esnek** imkanlar sunar.
- Bellek **daha verimli** bir şekilde kullanılır ve **daha hızlı** programlar yazılır.
- Dizilerin yönetiminde kullanılması **performansı** artırır.
- Fonksiyonların birden fazla değer döndürmesine olanak verir.
- Klasik yöntemle diziler tanımlanırken boyutları baştan tanımlanır ve değiştirilemezken, dinamik bellek kullanımıyla ihtiyaç halinde fazladan alan sağlanabilir.

Dinamik belleğin **DEZAVANTAJ**ları

- Bellek adreslerine erişim işaretçi değişkenlerinin kullanımını gerektirir. Bu, nispeten öğrenmesi zor bir yöntem olduğundan, programlamaya yeni başlayanlar için işleri karmaşık bir hale getirebilir.
- Bellek adreslerini kullanıcı erişimine açmak, olası programlama hatalarının ihtimalini artırır.
- Belleğin tüm hücrelerine erişim imkanı sağladığı için işaretçilerin yanlış kullanımı, sadece yazılan programın değil, tüm sistemin çökmesine dahi yol açabilir.

Dinamik Diziler

Diziler, programın başında kaç boyutlu olduğu verilerek bildirilirse, derleyici o dizi için gerekli bellek alanını ayırır. Dizi üzerindeki işlemler tamamlanmış ve artık bu bellek alanına dizi için ihtiyaç kalmamış olsa bile program içinde başka amaçla kullanılamaz.

Ancak dinamik bellek kullanımı söz konusu ise, program yürütülürken dizi için gerekli bellek alanı talep edilir ve işi bittiğinde iade edilebilir.

Bu amaçla standart kütüphanede;
calloc(), ***malloc()***, ***realloc()*** ve ***free()***
fonksiyonları tanımlanmıştır.

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

```
main() {
```

```
    int *p, i, j, enkucuk, enbuyuk;
```

```
    p = (int *) malloc(20*sizeof(int));
```

```
    for(i=0; i<20; i++) {
```

```
        *(p+i) = rand();
```

```
        printf("%d\n", *(p+i)); }
```

```
    enkucuk = *p;
```

```
    for(j=0; j<20; j++)
```

```
        if(*(p+j)<enkucuk)
```

```
            enkucuk = *(p+j);
```

```
    printf("\n En kucuk eleman: %d' dir.\n", enkucuk);
```

```
    enbuyuk = *p;
```

```
    for(j=0; j<20; j++)
```

```
        if(*(p+j)>enbuyuk)
```

```
            enbuyuk = *(p+j);
```

```
    printf("\n En buyuk eleman: %d' dir.\n", enbuyuk);
```

```
    free(p);
```

```
    return 0;
```

```
}
```

Bellekten istenen alanı ayırır ve o alanın başlangıç adresini gönderir.

Bellekten daha önce istenen alanı boşaltıp iade eder.

41

18467

6334

26500

19169

15724

11478

29358

26962

24464

5705

28145

23281

16827

9961

491

2995

11942

4827

5436

En kucuk eleman: 41' dir.

En buyuk eleman: 29358' dir.

İşaretçi Tipli Fonksiyonlar

Fonksiyonlar tanımlanırken işaretçi tipli değişkenleri olabileceği gibi, fonksiyonun kendi tipi de işaretçi olabilir.

Yani fonksiyon, çağırıldığı yere bir değer yerine adres de gönderebilir.

Standart kütüphanede string üzerinde işlem yapmayı sağlayan işaretçi tipli fonksiyonlar da vardır ve bu fonksiyonlar uygulamaların etkinliğini artırır.

String ile ilgili işaretçi uygulamalarına bir sonraki dersimizde yer verilecektir.


```

#include<stdio.h>

main() {
    int dizi[] = {11, 15, 37, 2, 43, -2, 7};
    int *q;
    q = EnKucugunAdresi(dizi, 7);
    printf("En küçük eleman: %d\n", *q);
    printf("En küçük elemanın adresi: %p\n", q);
    return 0; }

int EnKucugunAdresi(int A[], int n) {
    int EnKucuk, *p, i;
    EnKucuk = A[0];
    p = &A[0];

    for(i=1; i<n; i++)
        if (A[i] < EnKucuk) {
            EnKucuk = A[i];
            p = &A[i];}
    return p;
}

```

```

En küçük eleman: -2
En küçük elemanın adresi: 000000000062FE04

```

ÖRNEK: İşaretçi kullanan ortalama değer bulma fonksiyonu

```
#include<stdio.h>
#define NOTSAY 10

NotOku(int *, int);
double OrtBul(int *, int);

main() {
    int notlar[NOTSAY];
    int *p = notlar; /* int *p = &notlar[0]
                       olarak da yazılabilirdi. */
    NotOku(p, NOTSAY);
    double Ort = OrtBul(p, NOTSAY);

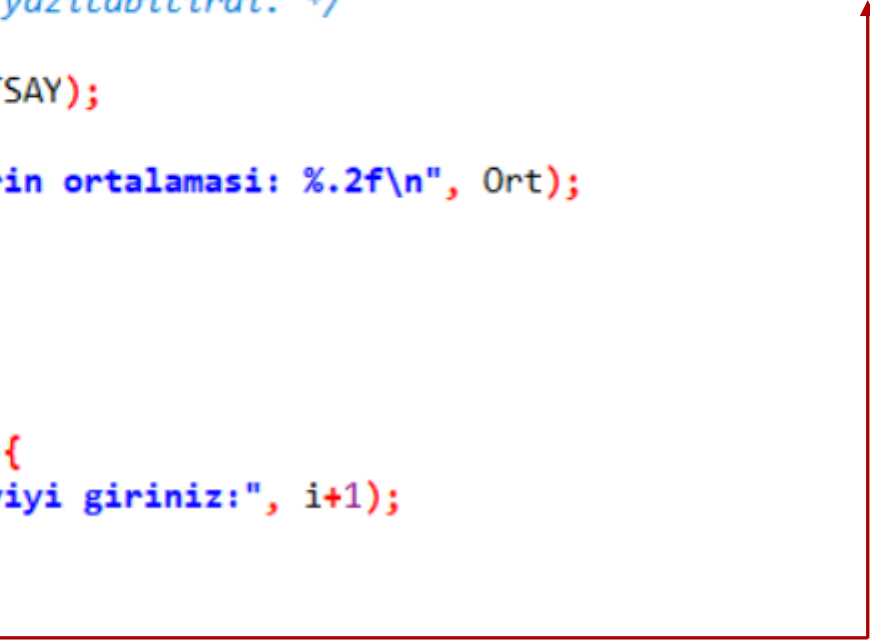
    printf("\n Girilen sayilarin ortalaması: %.2f\n", Ort);

    return 0; }

NotOku(int *p, int sayi) {
    int i;
    for(i=0; i<sayi; i++) {
        printf("\n%2d. sayiyi giriniz:", i+1);
        scanf("%d", p+i);
    }
}
```

```
double OrtBul(int *p, int sayi) {
    int i;
    double toplam = 0;

    for(i=0; i<sayi; i++)
        toplam+= *(p+i);
    return (double) toplam / (double) sayi;
}
```



ÖRNEK: İşaretçi tipli faktöriyel hesaplama fonksiyonu

```
#include<stdio.h>

int Faktoriyel(int);

main() {
    int sayi = 0;

    printf("\n Faktoriyel Hesaplama Fonk. Adresi: %p\n", Faktoriyel);
    printf("\n Faktoriyeli Hesaplanacak Sayi:");
    scanf("%d", &sayi);

    int faktor = Faktoriyel(sayi);
    printf("\n Sonuc: %d", faktor);
}

int Faktoriyel(int sayi) {
    int sonuc = 1, i;

    if( sayi<=0 )
        return sonuc;

    for(i=sayi; i>=1; i--) {
        sonuc = sonuc*i; }
    return sonuc;
}
```

ÖDEV:

Verilen ek pdf dosyasında bir sayının karesinin hesaplandığı Program 3.7'ye benzer şekilde işaretçilerden yararlanarak, kullanıcı tarafından girilen bir sayının %2,5'unu hesaplayan ve sonucu ekranda gösteren bir program yazınız. Yazacağınız program, tek bir hesaplama yapıp sonlanmamalı, kullanıcının istediği kadar tekrarda hesaplama yapabilmesine olanak sağlamalıdır.

KAYNAKLAR

- Her Yönüyle C, Tefik KIZILÖREN, Kodlab
- İşte C Programlama Dili, Dr. Rifat ÇÖLKESEN, Papatya Yayıncılık
- <http://www.baskent.edu.tr/~tkaracay/etudio/ders/prg/c/pointers.pdf>
- <http://www.alikeskin.org/algoritma/Pointer.pdf>