

Algoritma ve Programlama -1

11. HAFTA

Gösterici ve Diziler, Gösterici ve
Fonksiyonlar, Karakter Göstericiler

11. Hafta - Tekrar

- Göstericiler (Pointers)
- Değişken ve Bellek Adresi
- Gösterici Operatörleri
- Gösterici Aritmetiği

Göstericiler ve Diziler

- Diziler ile ilgili iki önemli özellik:
 1. Dizi elemanlarının hepsi aynı türdendir.
 2. Dizi elemanları bellekte sürekli bir biçimde bulunurlar.
- Başlangıç adresi ve uzunluğu bilinen bir dizinin, gösterici operatörü ile tüm elemanlarına erişebilir miyiz?
- C dilinde göstericiler ve diziler arasında yakın bir ilişki vardır.
- Bir dizinin adı, dizinin ilk elemanının adresini saklayan bir göstericidir.
- Bu yüzden, bir dizinin herhangi bir elemanına gösterici ile de erişilebilir.

Göstericiler ve Diziler

- Örneğin:
int kutle[5], *p, *q; şeklinde bir bildirim yapılsın.
- Buna göre aşağıda yapılan atamalar geçerlidir:
 - p = &kutle[0]; /* birinci elemanın adresi p göstericisine atandı */
 - p = kutle; /* birinci elemanın adresi p göstericisine atandı */
 - q = &kutle[4]; /* son elemanın adresi q göstericisine atandı */
- İlk iki satırdaki atamalar aynı anlamdadır. Dizi adı bir gösterici olduğu için, doğrudan aynı tipteki bir göstericiye atanabilir.
- Ayrıca, i bir tamsayı olmak üzere,
kutle[i] ile *(p+i) aynı anlamdadır.
- Bunun sebebi, p göstericisi kutle dizisinin başlangıç adresini tutmuş olmasıdır.
- p+i işlemi ile i+1. elemanın adresi, ve *(p+i) ile de bu adresteki değer hesaplanır.
- *p+i ne anlama gelir?

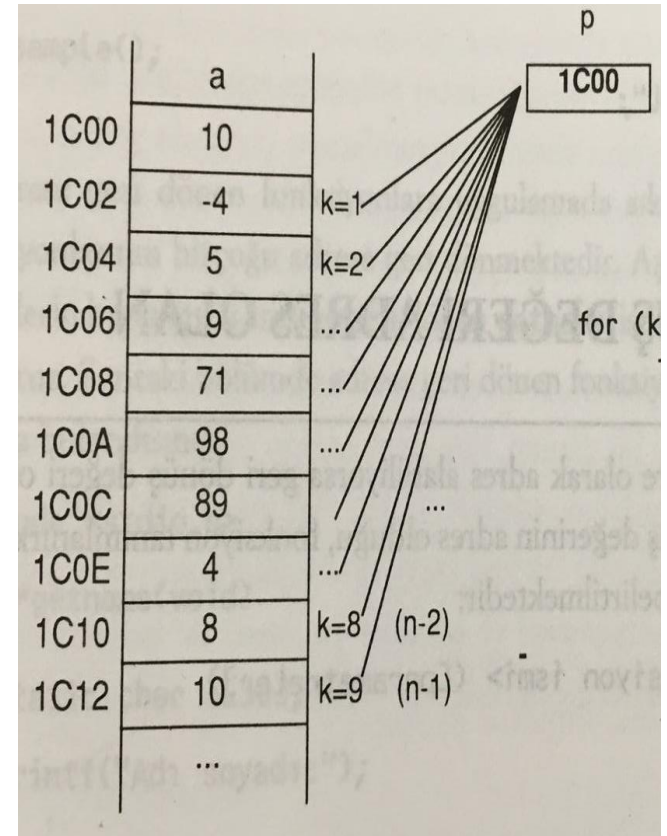
Göstericiler ve Diziler

```
int main()
{
    int a[10]= {10,-4,5,9,71,98,89,4,8,10};
    int *p, *q;
    p=&a[0]; //p=a;
    q=&a[9];

    printf("a dizisi adresi: %p \n",&a);
    printf("p pointerinin gosterdigi adres: %p \n",p);
    printf("q pointerinin gosterdigi adres: %p \n",q);

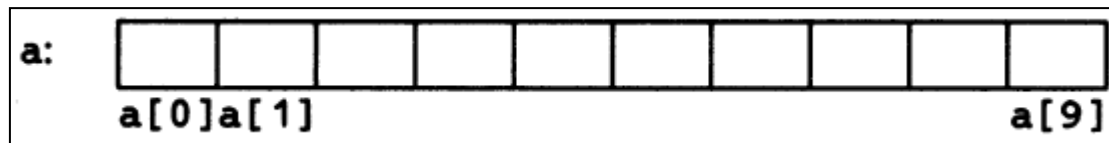
    printf("a[0] degeri: %d \n",a[0]);
    printf("*p : %d \n",*p);
    printf("*q : %d \n",*q);

    return 0;
}
```



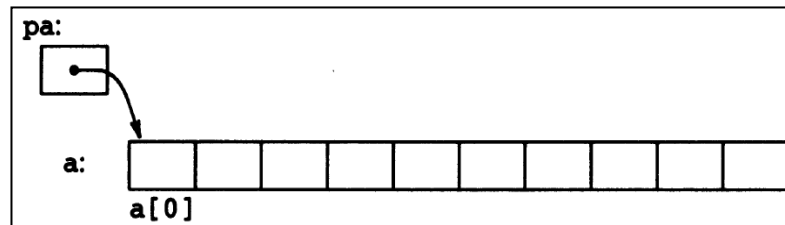
Pointers and Arrays

- In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously.
- Any operation that can be achieved by array subscripting can also be done with pointers.
- The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.
- `int a[10];`
 - defines an array `a` of size 10, that is, a block of 10 consecutive objects named `a[0]`, `a[1]`, ... , `a[9]`.

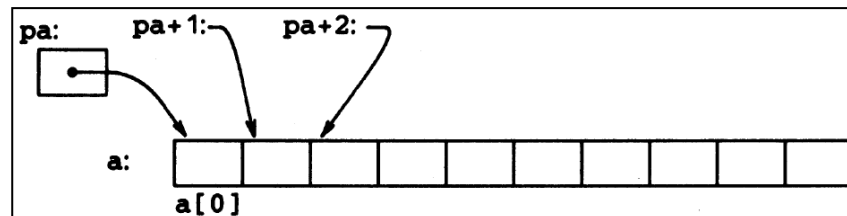


Pointers and Arrays

- The notation $a[i]$ refers to the i -th element of the array. If pa is a pointer to an integer, declared as
 - `int *pa;`
 - `pa = &a[0];`
- sets pa to point to element zero of a ; that is, pa contains the address of $a[0]$.



- `x = *pa;` will copy the contents of $a[0]$ into x .
- If pa points to a particular element of an array, then by definition $pa+1$ points to the next element, $pa+i$ points i elements after pa , and $pa-i$ points i elements before.
- Thus, if pa points to $a[0]$, $*(pa+1)$ refers to the contents of $a[1]$, $pa+i$ is the address of $a[i]$, and $*(pa+i)$ is the contents of $a[i]$.



Pointers and Arrays

- The correspondence between indexing and pointer arithmetic is very close.
- By definition, the value of a variable or expression of type array is the address of element zero of the array.
- Thus after the assignment `pa=&a[0]`, `pa` and `a` have identical values.
- Since the name of an array is a synonym for the location of the initial element, the assignment
 - `pa=&a[0]` can also be written as `pa=a`.

Pointers and Arrays

- Rather more surprising, at least at first sight, is the fact that a reference to $a[i]$ can also be written as $*(a+i)$.
- In evaluating $a[i]$, C converts it to $*(a+i)$ immediately; the two forms are equivalent.
- Applying the operator $\&$ to both parts of this equivalence, it follows that $\&a[i]$ and $a+i$ are also identical: $a+i$ is the address of the i -th element beyond a .
- As the other side of this coin, if pa is a pointer, expressions may use it with a subscript; $pa[i]$ is identical to $*(pa+i)$.
- In short, an array-and-index expression is equivalent to one written as a pointer and offset.
- There is one difference between an array name and a pointer that must be kept in mind.
 - A pointer is a variable, so $pa=a$ and $pa++$ are legal. But an array name is not a variable; constructions like $a=pa$ and $a++$ are illegal.

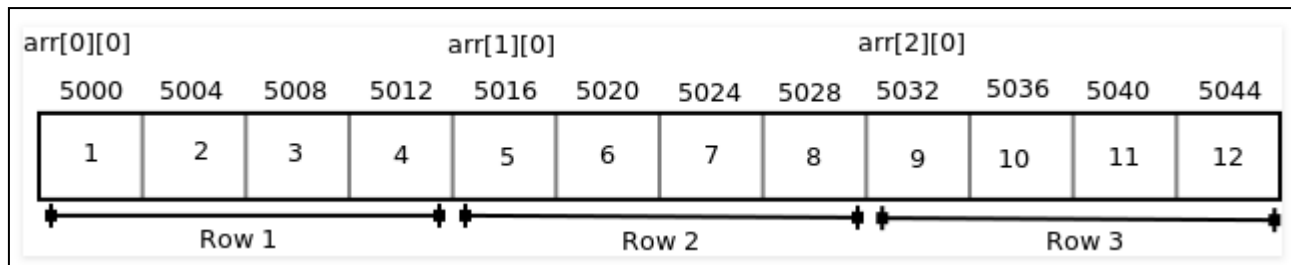
Göstericiler ve Diziler Uygulama

- n elemanlı bir dizinin aritmetik ortalamasını pointer kullanarak hesaplayan C kodunu yazınız.
- Matriste istenilen bir indisteki elemana pointer yardımıyla nasıl ulaşabiliriz?

Göstericiler ve İki Boyutlu Dizi

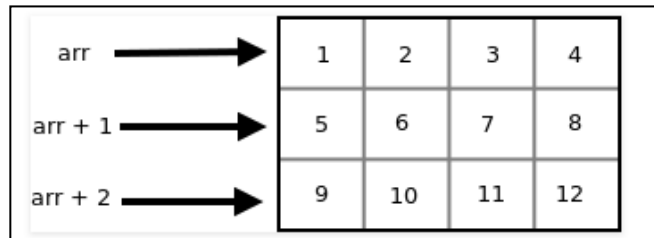
```
int arr[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };
```

| | Col 1 | Col 2 | Col 3 | Col 4 |
|-------|-------|-------|-------|-------|
| Row 1 | 1 | 2 | 3 | 4 |
| Row 2 | 5 | 6 | 7 | 8 |
| Row 3 | 9 | 10 | 11 | 12 |



- So here *arr* is an array is an array of 3 elements where each element is a 1-D array of 4 integers.
- We know that the name of an array is a constant pointer that points to 0th 1-D array and contains address 5000.
- Since *arr* is a 'pointer to an array of 4 integers', according to pointer arithmetic the expression *arr + 1* will represent the address 5016 and expression *arr + 2* will represent address 5032.

Göstericiler ve İki Boyutlu Dizi



arr - Points to 0th element of arr - Points to 0th 1-D array - 5000
arr + 1 - Points to 1th element of arr - Points to 1st 1-D array - 5016
arr + 2 - Points to 2th element of arr - Points to 2nd 1-D array - 5032

- We know, the pointer expression $*(arr + i)$ is equivalent to the subscript expression $arr[i]$. So $*(arr + i)$ which is same as $arr[i]$ gives us the base address of i^{th} 1-D array.

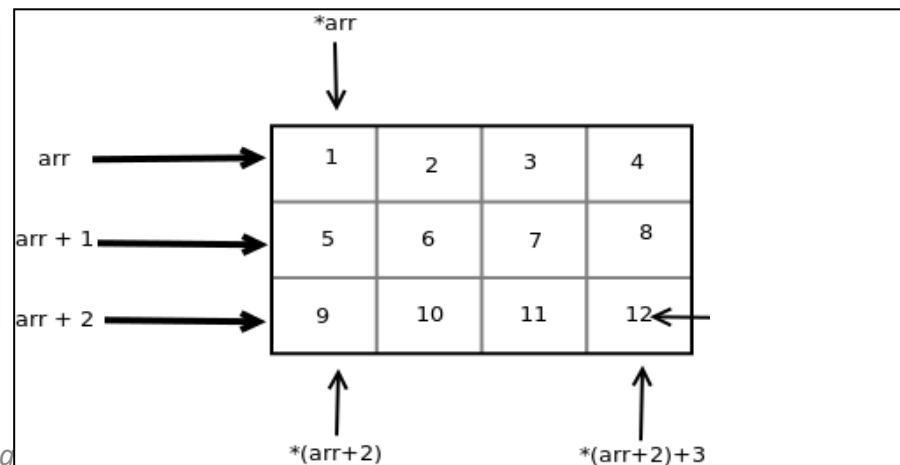
$*(arr + 0)$ - arr[0] - Base address of 0th 1-D array - Points to 0th element of 0th 1-D array - 5000
 $*(arr + 1)$ - arr[1] - Base address of 1st 1-D array - Points to 0th element of 1st 1-D array - 5016
 $*(arr + 2)$ - arr[2] - Base address of 2nd 1-D array - Points to 0th element of 2nd 1-D array - 5032

Göstericiler ve İki Boyutlu Dizi

- To access an individual element of our 2-D array, we should be able to access any j^{th} element of i^{th} 1-D array.
- Since the base type of $*(arr + i)$ is *int* and it contains the address of 0^{th} element of i^{th} 1-D array, we can get the addresses of subsequent elements in the i^{th} 1-D array by adding integer values to $*(arr + i)$.
- For example $*(arr + i) + 1$ will represent the address of 1^{st} element of 1^{st} element of i^{th} 1-D array and $*(arr + i) + 2$ will represent the address of 2^{nd} element of i^{th} 1-D array.
- Similarly $*(arr + i) + j$ will represent the address of j^{th} element of i^{th} 1-D array. On dereferencing this expression we can get the j^{th} element of the i^{th} 1-D array.

| | |
|--------------------------|---|
| arr | Points to 0^{th} 1-D array |
| *arr | Points to 0^{th} element of 0^{th} 1-D array |
| (arr + i) | Points to i^{th} 1-D array |
| *(arr + i) | Points to 0^{th} element of i^{th} 1-D array |
| *(arr + i) + j) | Points to j^{th} element of i^{th} 1-D array |
| *(*(arr + i) + j) | Represents the value of j^{th} element of i^{th} 1-D array |

Uygulama



Göstericiler ve Fonksiyonlar

- Göstericiler bir fonksiyon için parametre olabileceği gibi, geri dönüş değeri olarakta kullanılabilirler.
 - Fonksiyon Parametresi Olan Göstericiler
 - Fonksiyon Geri Dönüş Değeri Olan Göstericiler

Fonksiyon Parametresi Olan Göstericiler

- C (ve C++) programlama dilinde fonksiyon parametreleri değer (pass by value) yada adres (pass by reference) olarak geçilebilir.
- Daha önceki uygulamalarda fonksiyonlara parametreler değer geçerek taşınmıştı.
- Bu şekilde geçirilen parametreler, fonksiyon içersinde değiştirilse bile, fonksiyon çağrıldıktan sonra bu değişim çağrılan yerdeki değerini değiştirmez.
- Fakat, bir parametre adres geçerek aktarılırsa, fonksiyon içindeki değişiklikler geçilen parametreyi etkiler.
- Adres geçerek aktarım, gösterici kullanmayı zorunlu kılar.

Değer ve Adres Geçerek Aktarım

```

void f1(int ); /* iki fonksiyon */
void f2(int *);
int main()
{
    int x = 55;
    printf("x in degeri,\n");
    printf("Fonksiyonlar cagrilmadan once: %d\n",x);
    /* f1 fonksiyonu cagriliyor...*/
    f1(x);
    printf("f1 cagirildikten sonra : %d\n",x);
    /* f2 fonksiyonu cagriliyor...*/
    f2(&x);
    printf("f2 cagirildikten sonra : %d\n",x);
    return 0;
}

/* Değer gecerek aktarım */
void f1(int n)
{
    n = 66;
    printf("f1 fonksiyonu icinde : %d\n",n);
}

/* Adres gecerek aktarım */
void f2(int *n)
{
    *n = 77;
    printf("f2 fonksiyonu icinde : %d\n",*n);
}

```

```

x in degeri,
Fonksiyonlar cagrilmadan once: 55
f1 fonksiyonu icinde : 66
f1 cagirildikten sonra : 55
f2 fonksiyonu icinde : 77
f2 cagirildikten sonra : 77

```


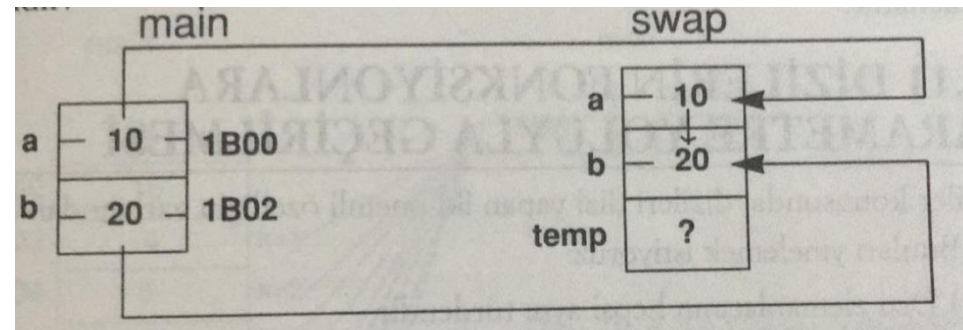
- Bir yerel değişkenin içeriğini bi fonksiyonun değiştirebilmesi için, fonksiyona o yerel değişkenin adresi geçirilmelidir.

Fonksiyon Parametresi Olan Göstericiler

- İki yerel değişkenin içeriğini birbiriyile değiştiren fonksiyonu C dilinde yazınız.

```
#include <stdio.h>
void swap(int a, int b);
void main()
{
    int a = 10, b = 20;
    swap(a, b);
    printf("a = %d b = %d\n", a, b);
}

void swap(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

- Burada değiştirme işlemi yapılamaz.
- İçerikleri değişenler main fonksiyonunun yerel değişkenleri değil, swap fonksiyonun parametre değişkenleridir.

Fonksiyon Parametresi Olan Göstericiler

- İki yerel değişkenin içeriğini birbiriyile değiştiren fonksiyonu C dilinde yazınız.

```
#include <stdio.h>

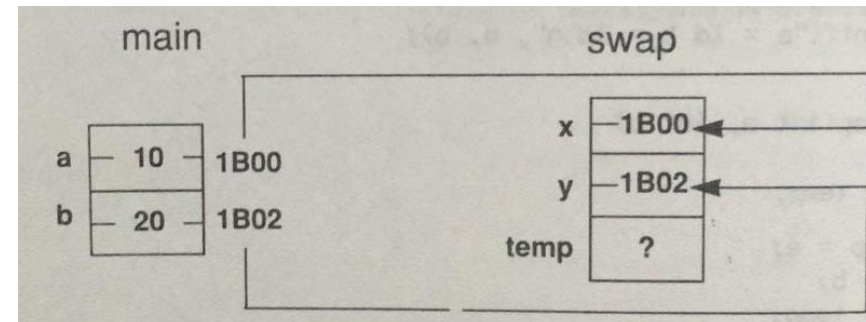
void swap(int *, int *);

void main()
{
    int a = 10, b = 20;

    swap(&a, &b);
    printf("a = %d b = %d\n", a, b);
}

void swap(int *x, int *y)
{
    int temp;

    temp = *x;
    *x = *y;
    *y = temp;
}
```



- Burada değiştirme işlemi gerçekleştirilir.
- Main fonksiyonunun yerel değişkenlerine ait adresler swap fonksiyonun parametre değişkenleridir.
- Değişiklik doğrudan main fonksiyonun yerel değişkenlerinin bulunduğu adresler üzerinden yapılır.

Fonksiyon Geri Dönüş Değeri Olan Göstericiler

- Fonksiyonların geri dönüş değeri bir gösterici olabilir.
- Bu durumda fonksiyon bir değer değil adres döndürecektir.
- Programda önce bir dizinin indisleri, dizi değerleri ve dizi elemanlarının adresleri ekrana basılır.
- Daha sonra, maxAdr(); fonksiyonu ile dizinin en büyük elemanının adresi döndürülür.
- Bu örnek program, göstericilerin gücünü çok zarif bir biçimde bize sunmaktadır.*

```
#include <stdio.h>

double* maxAdr(double a[], int boyut){
    double ebd = a[0];
    double *eba = &a[0];
    int i;
    for(i=1; i<boyut; i++){
        if(a[i]>ebd){
            ebd = a[i]; // en büyük deger
            eba = &a[i]; // en büyük adres
        }
    }
    return eba;
}

int main()
{
    double x[6] = {1.1, 3.3, 7.1, 5.4, 0.2, -1.5};
    double *p;
    int k;
    // indis, dizi ve adresini ekrana bas
    for(k=0; k<6; k++){
        printf("%d %lf %p\n", k, x[k], &x[k]);
    }

    p = maxAdr(x, 6);

    printf("En büyük deger: %lf\n", *p);
    printf("En büyük adres: %p \n", p);
    printf("En büyük konum: %d \n", int(p-&x[0]));

    return 0;
}
```

Fonksiyon Geri Dönüş Değeri Olan Göstericiler

- Bir dizi geriye döndürülebilir mi?
- Parametre olarak aldığı bir diziyi sıralayarak geriye döndüren fonksiyonu C dilinde yazınız.
- Uygulama

```

int main()
{
    int a=5, b=10;
    int dizi[5]= {1,8,3,12,9};
    int matris[3][3]={0,7,4},{6,2,11},{15,18,22}};
    int *psayi,*qsayi,*tsayi,*pdizi,*pmat;
    psayi=&a;
    qsayi=&b;
    //Level 1
    printf("Level-1 / SAYI \n");
    printf("----- \n");
    printf("psayi Deger: %d, qsayi Deger:%d \n",*psayi,*qsayi);
    printf("%d psayi Tuttugu Adres: %d, qsayi Tuttugu Adres:%d \n",psayi,qsayi);
    printf("psayi Deger: %d, qsayi Deger:%d \n",++*psayi,++*qsayi);
    printf("psayi Deger: %d, qsayi Deger:%d \n",*--psayi,*++qsayi);
    printf("psayi Deger: %d, qsayi Deger:%d \n",*psayi++,*qsayi--);
    printf("psayi Deger: %d, qsayi Deger:%d \n",*psayi,*qsayi);
    printf("----- \n");
    //Level 2
    pdizi=dizi;
    printf("Level-2/ DIZI \n");
    printf("----- \n");
    printf("dizi Tuttugu Adres: %d, pdizi Tuttugu Adres:%d \n",dizi,pdizi);
    printf("dizi Deger:%d, pdizi Deger:%d \n",*dizi,*pdizi);
    printf("(dizi+1) Tuttugu Adres: %d, (pdizi+1) Tuttugu Adres:%d \n",dizi+1,pdizi+1);
    printf("(dizi+1) Tuttugu Adres: %d, pdizi+1 Tuttugu Adres:%d \n",dizi+1,++pdizi);
    pdizi=dizi;
    printf("(dizi+3) Deger:%d, (pdizi+3) Deger:%d \n",*(dizi+3),*(pdizi+3));
    pdizi=dizi;
    printf("(*pdizi++) Deger:%d \n",*pdizi++);
    pdizi=dizi;
    printf("(++pdizi) Deger:%d \n",++pdizi);
    pdizi=dizi;
    printf("(++pdizi) Deger:%d \n",++pdizi);
    printf("----- \n");

    //Level 3
    printf("Level-3 / MATRIS \n");
    printf("----- \n");
    pmat=matris;
    printf("matris Tuttugu Adres: %d, pmat Tuttugu Adres:%d \n",matris,pmat);
    printf("matris Deger: %d, pmat Deger:%d \n",*matris,*pmat);
    printf("matris Deger: %d, pmat Deger:%d \n",**matris,*pmat);
    printf("matris+1 Tuttugu Adres: %d, pmat+1 Tuttugu Adres:%d \n",matris+1,pmat+1);
    printf("(matris+1) Deger: %d, (pmat+1) Deger:%d \n",*(matris+1),*(pmat+1));
    printf("(matris+1) Deger: %d, (pmat+1) Deger:%d \n",**matris,**pmat);
    printf("(matris+1)+1 Tuttugu Adres: %d, (pmat+1)+1 Tuttugu Adres:%d \n",*(matris+1)+1,*(pmat+1)+1);
    printf("(matris+1)+2 Deger: %d, (pmat+1)+2 Deger:%d \n",*(matris+1)+2,*(pmat+1)+2);
    printf("(matris+1)+1 Tuttugu Adres: %d, (pmat+3)+1 Tuttugu Adres:%d \n",*(matris+1)+1,*(pmat+3)+1);
    printf("(matris+1)+1 Deger: %d, (pmat+7) Deger:%d \n",*(matris+1)+1,*(pmat+7));

    return 0;
}

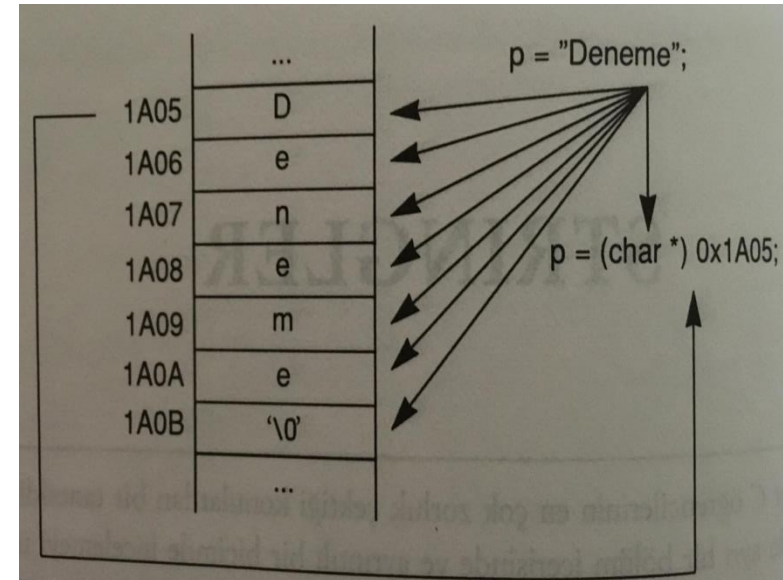
```

Stringler (Karakter Katarları)

- İki tırnak içerisindeki ifadeler string ifadeler denir.
- C’de stringler aslında karakteri gösteren bir adrestir.
- C derleyicileri, derleme aşamasında bir stringle karşılaştığında sırasıyla,
 - Onu belleğin güvenli bir bölgesine yerleştirir,
 - Sonuna null karakteri ekler,
 - String yerine yerleştirdiği yerin başlangıç adresini koyar.
- Bu durumda string ifadeleri aslında stringlerin bellekteki başlangıç yerini gösteren karakter türünden birer adrestir.
 - `char *p;`
 - `p="deneme";`

Stringler(Karakter Katarları)

- `char *p="deneme"` vs `char s[10]="deneme"`
- Göstericilere ilk değer verildiğinde;
 - String belleğe yerleştirilir sonra başlangıç adresi göstericiye atanır.
- Dizilerde;
 - Önce dizi açılır, sonra karakterler tek tek dizi elemanlarına yerleştirilir.



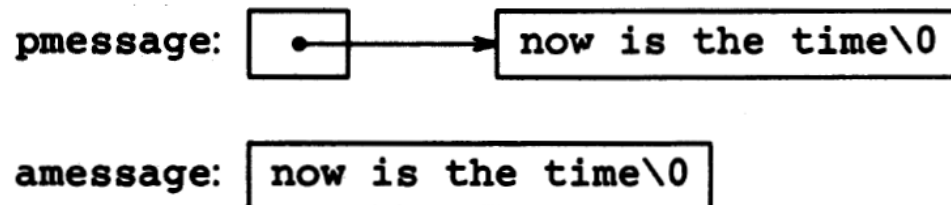
```
int main()
{
    char *p="Programlama";
    char s[20]="Programlama";
    // p=s;
    printf("p Tuttuğu Adres: %d,s Tuttuğu Adres: %d \n",p,s);
    printf("%s %s \n",p+7,s+7);
    printf("%c %c \n",p[4],s[4]);
    return 0;
}
```

Character Pointers

- A *string constant*, written as "I am a String" is an array of characters.
- In the internal representation, the array is terminated with the null character '\0' so that programs can find the end.
- Perhaps the most common occurrence of string constants is as arguments to functions.
 - `printf("hello, world\n");`
- When a character string like this appears in a program, access to it is through a character pointer; `printf` receives a pointer to the beginning of the character array.
- That is, a string constant is accessed by a pointer to its first element.

Character Pointers

- String constants need not be function arguments. If pmessage is declared as
 - `char *pmessage="now is the time";`
 - And this assigns to pmessage a pointer to the character array.
 - This is *not* a string copy; only pointers are involved.
 - C does not provide any operators for processing an entire string of characters as a unit.
- There is an important difference between these definitions:
 - `char amessage[] = "now is the time"` vs `char *pmessage = "now is the time"`
- amessage is an array, just big enough to hold the sequence of characters and '\0' that initializes it.
- Individual characters within the array may be changed but amessage will always refer to the same storage.
- c



Initialization Pointer

- The pointers x and y are allocated as local variables.
- The type int* means "pointer which points to ints".
- The pointers do not automatically get pointees.
- The syntax *x dereferences x to access its pointee.

```
void main() {  
    int *x; // Allocate the pointers x and y  
    int *y; // (but not the pointees)  
    int a=5;  
  
    // x = malloc(sizeof(int));  
    x = &a; // Allocate an int pointee,  
            // and set x to point to it  
  
    *x = 42; // Dereference x to store 42 in its pointee  
  
    *y = 13; // CRASH -- y does not have a pointee yet  
  
    y = x; // Pointer assignment sets y to point to x's pointee  
  
    *y = 13; // Dereference y to store 13 in its (shared) pointee  
}
```

Initialization Pointer

- `char amessage[] = "now is the time"` vs `char *pmessage = "now is the time"`
- There is difference between character array initialization and char pointer initialization.
- Whenever you initialize a char pointer to point at a string literal, the literal will be stored in the code section.
- You can not modify code section memory.
- If you are trying to modify unauthorised memory then you will get a segmentation fault.
- But if you initialize a char array, then it will be stored in the data or stack section, depending on at where you declared the array.
- So you can then modify the data.

[Character Array vs Character Pointer](#)

Copy the String

- We will illustrate more aspects of pointers and arrays by studying versions of two useful functions adapted from the standard library.
- The first function is `strcpy(s, t)`, which copies the string `t` to the string `s`.
- It would be nice just to say `s=t` but this copies the pointer, not the characters.

```
/* strcpy: copy t to s; array subscript version */
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

```
/* strcpy: copy t to s; pointer version 1 */
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Copy the String

```
/* strcpy: copy t to s; pointer version 1 */  
void strcpy(char *s, char *t)  
{  
    while ((*s = *t) != '\0') {  
        s++;  
        t++;  
    }  
}
```

- Because arguments are passed by value, strcpy can use the parameters s and t in any way it pleases.
- Here they are conveniently initialized pointers, which are marched along the arrays a character at a time, until the '\0' that terminates t has been copied to s.
- In practice, strcpy would not be written as we showed it above.
- Experienced C programmers would prefer

```
/* strcpy: copy t to s; pointer version 2 */  
void strcpy(char *s, char *t)  
{  
    while ((*s++ = *t++) != '\0')  
        ;  
}
```

Copy the String

- This moves the increment of s and t into the test part of the loop.
- The value of *t++ is the character that t pointed to before t was incremented; the postfix ++ doesn't change t until after this character has been fetched.
- In the same way, the character is stored into the old s position before s is incremented.
- This character is also the value that is compared against '\0' to control the loop.
- The net effect is that characters are copied from t to s, up to and including the terminating '\0'.
- As the final abbreviation,

```
/* strcpy:  copy t to s; pointer version 3 */  
void strcpy(char *s, char *t)  
{  
    while (*s++ = *t++)  
        ;  
}
```

while(*s++ = *t++);

- There are several things going on here:
 - An assignment statement, $x = y$, is actually an *expression* (equal to the just-assigned value), which can itself be used as part of more complex expression.
 - This is useful for multiple assignments ($x = y = z$), and loop conditions like while (($c = \text{getchar}()$) != EOF).
 - Unlike, say, Pascal or Java, C isn't strict about Boolean values. Any nonzero value is true (i.e., causes the body of an if, while, or for statement to be executed), and zero (whether 0, 0.0, '\0', or NULL) is false.
 - In C, every string is terminated by '\0' character whose ASCII Code is 0.
 - Strings are represented as an array of characters (or a pointer to such an array) terminated by a NUL (zero) character. So, if p is a pointer to a string, $*p==0$ means it's at the end of the string.
 - A *postfix* ++ operator increments its variable, but returns the old value.
 - A while loop may have an empty body.

String ve Pointer Uygulamalar

- Pointer kullanarak string içerisinde kopyalama yapan C kodunu yazınız.
- Pointerlar ile string içerisinde bir string ifadeden kaç tane olduğunu bulan C kodunu yazınız.
- Farklı uzunluktaki string ifadelerin tek bir dizi içinde pointer ile tutulmasını sağlayan C kodunu yazınız.
- Pointer kullanarak bir string içerisindeki sesli harfleri silen C kodunu yazınız.

Ödev-4 (Metin Bulmaca)

- Sizin tarafından tanımlanan $m \times n$ boyutundaki karakter matrisinin içerisinde pointer yardımıyla gezinerek kendisine verilen string ifadenin soldan sağa ve yukarıdan aşağıya hangi pozisyonlarda bulunduğunu ve kaç adet olduğunu bulan C kodunu yazınız.
- Sağdan sola ve aşağıdan yukarıya pozisyonları için yapanlara bonus puan verilecektir.
- 22.12.2019 Salı günü saat 23:59'a kadar ders için oluşturulan Classroom sınıfına yüklenecektir.
- Belirtilen süre içinde gönderilmeyen ödevler değerlendirilmeyecektir.***
- Ödevler benzerlik tespit uygulaması tarafından kontrol edilecek olup kopya ödevler ödev puanı kadar – puan ile cezalandırılacaktır.***

| | | | | | | |
|---|---|---|---|---|---|---|
| N | L | I | F | N | E | U |
| E | N | E | U | A | L | H |
| U | E | G | E | U | E | N |
| T | U | A | N | E | U | W |
| P | N | E | U | J | C | Z |

Önümüzdeki Hafta

- Dinamik Bellek Yönetimi
 - malloc
 - calloc
 - realloc
- Yapılar - Struct

Kaynakça

1. Aslan, K., A'dan Z'ye C Kılavuzu, Pusula Yayınları.
2. Bişkin F., C Programlama Diline Giriş Notları.
3. Kernighan, K.W., Ritchie, D.M., The C Programming Language, Prentice Hall Software Series.
4. <https://www.geeksforgeeks.org/pointer-array-array-pointer/>
5. <http://cslibrary.stanford.edu/106/>
6. <https://stackoverflow.com/questions/34787202/can-a-string-be-assigned-to-a-char-pointer-without-it-becoming-a-literal>
7. <https://stackoverflow.com/questions/24545497/while-s-t-in-c>