Adresler (`pointer` veri tipi) (Aha yandık!)

C'nin kafa karıştıran konularından biri olan adresleri tanıyalım, aslında korkulacak bir şey olmadığını görelim.

Değişkenlerin belirli bir bellek alanı kapladığı ve kapladıkları bu alanın da bir adresi olduğunu biliyoruz. Peki, bu adresi değişken olarak kullanabilir miyiz?
Evet!

`pointer`, yani işaretçi, tipi bellekteki adresleri işaret etmek için kullanılır. Bunu açık adres gibi düşünebilirsiniz.
Adresi kullanan birisi evinize, veya program dilinde değişkeninize ulaşabilecek.

C dilinde pointerların temeli şu şekildedir:

```c
//Pointer değişken tanımlamak için ismin başına yıldız koyarız:
int *adres;
//değişken adresi almak için, adresini alacağımız değişkenin başına & koyarız.
int sayi = 3;
adres = &sayi;
//pointer üzerindeki değişkene erişmek için pointer'ın başına yıldız koyarız
*adres = 5;
//veya
printf("Degiskenden: %d , Adresten: %d", sayi, *adres);

//Pointerlar iç içe geçebilir
int katman0, *katman1, **katman2, ***katman3;
katman1 = &katman0;
katman2 = &katman1;
katman3 = &katman2;
//iç içe geçtiği durumda katman kadar yıldız koyarak değere erişiriz.
***katman3 = 40;

//sadece adres bilgisi gerekiyorsa void* tipini kullanırız.
void* kat2_adres = &kat2_adres;

//adres yazdırmak için %p kullanırız
printf("kat2_adres: %p, katman3: %p", kat2_adres, katman3);
```

Pointerların aritmatiği bildiğimiz sayılardan daha farklıdır:

```c
//Pointer aritmatiğini anlamada iki yardımcımız var:
//sizeof( degisken ): verdiğimiz değişkenin boyutunu bulan bir yardımcı. (long int döndürür)

int* test_ptr = 0;
long ptr_aritmatik_sonuc = (long)(test_ptr + 1);
printf("%ld\n", ptr_artimatik_sonuc);
printf("%ld", sizeof(int));
```

Potansiyel çıktısı (sistemden sisteme göre değişiklik olabilir):

```
4
4
```

Ayrıca, programınızda hata ayıklarken pointer tipleri kodda yazdığınızdan biraz daha farklı gözükecektir.
Bunu bir örnek ile gösterelim, anlaması daha kolay olur:

```c
// Örnek Kod
int sayi = 40;
int *isaretci = &sayi; // <- Bir sayı işaretçisi tanımladık.

printf("Sayımız: %d", isaretci); // <- Sayı istenen yere sayı İŞARETÇİsi verdik.
//Bu, bir yere işaretçinin gösterdiği değere ulaşmaya çalışırken * koymayı unuttuğumuzda
//başımıza gelebilir.
```

Programa çevirilirken verilen uyarı:

> **ornek.c**: In function '**main**':
> **ornek.c**:6:19: **warning:** format **'%d'** expects argument of type **'int'**, but argument 2 has type **'int *'**

Bunu türkçeye çevirirsek:

> **ornek.c**: '**main**' fonksiyonunda:
> **ornek.c**:6:19: **Uyarı:** '**%d**' şekli argümanın '**int**' olmasını bekler ama argüman 2 '**int ***' tipine sahip

Bundan alayacağımız üzere, program çeviricisi, kodda yazdığımız `int *isaretci` tipini şu şekilde algılıyor:

> Değişkenin ismi `isaretci`,
> tipi "int işaretçisi" yani `int*`,
> değeri de sayi değişkenin adresi.

Yani işaretçi işareti tip işaretinin hemen sonuna geliyor.

"Neden pointer kullanmalıyım?" sorusuna daha sonra değineceğiz. (bkz. Ders 5 ve 6)

Pointer'a bağlı türler

Pointerların kendisi yararlı ama şimdi anlatacağım tipler ile çok çok daha yararlı olacak!

- `array`, yani diziler:
  Arrayler aynı tipteki birden fazla veriyi tek değişkende tutmamızı sağlar.
  Yani, 10 kişinin ismini tutmak için 10 tane farklı değişken yerine tek bir değişken kullanabiliriz!
  Yaşasın daha az kod yazmak!
  Gelin nasıl kullanıldıklarına da bakalım:

  ```c
  // 10 sayı kapasitesi olan bir sayılar dizisi oluşturalım
  int dizi1[10];

  //arrayler'de sıra indeks sırasına göre işler:
  ```

```
//yani sayma sayılarıyla başlamaz, (1, 2, 3, ...)
//doğal sayılarla başlar. (0, 1, 2, ...)

//hadi arraydeki değerlerin ilkine 3, ikincisine 5 değeri verelim.
dizi1[0] = 3;
dizi1[1] = 5;

//değerleri okumak için atadığımız şekilde köşeli parantez kullanıyoruz:
printf("%d %d", dizi1[0], dizi1[1]);

//değerini önceden atadığımız bir dizi yapalım.
int dizi2[3] = {1, 2, 3};
int dizi3[5] = {1, 3};

//başlangıç değerlerini verdiğimiz bir dizinin kapasite sayısını yazmazsak
//o dizinin kapasitesi başlangıç değerler sayısı kadar olur:
int dizi4[] = {1, 2, 3, 4}; //bu dizinin boyutu 4 int olacaktır.
```

Burada not almanız gereken önemli bir şey var: Dizi boyutları **dinamik değildir, program esnasında kolayca değiştirilemez.**
Çünkü diziler özel bir pointer türüdür:
Başlangıçtaki pointer değerleri, verdiğimiz kapasite boyutunda bir değişkenin adresidir.
Yani, bu pointerin adresi önden tanımlanır. Hatta, **arraylerin adresi değiştirilemez**.
Arraylerin bu atanış biçiminden dolayı `*dizi` erişimi ile `dizi[0]` aynı sonucu verecektir.

- C Metinleri `char*`, `char[]`:
  C'de metinler, karakter dizilerinin özel bir kullanımı ile temsil edilir.

  Hadi bir metin yapalım, ve onu bir değişkene atayalım, sonra da yazdıralım:

```
char *metin = "Selam!";
printf("%s\n", metin);
```

  Çıktı:

```
Selam!

```

  Metinleri birleştirmek için yanına bir metin daha açabiliriz. Hatta, daha iyi okunması için boşluk da koyabiliriz.

```
printf("Selam " "Dünyalı");
```

  Çıktısı:

```
Selam Dünyalı
```

  Eğer `"` karakterini yazdırmamız gerekiyorsa, eğik çizgi kullanarak metinin bitmesini önleyebiliriz.

```
printf("Vazoyu mu kırdın? \"Aferin\" sana.");
```

Çıktısı:

```
Vazoyu mu kırdın? "Aferin" sana.
```

C Metinlerinin diğer dillerden bir farkı var. Analtmak yerine, birlikte görmeye ne dersiniz? Şimdi, bu metni inceleyelim:

```
char* metin = "Selam!";
printf("%d. karakter: '%c' , sayısı: %d\n", 0 +1 , metin[0], metin[0]);
// S
printf("%d. karakter: '%c' , sayısı: %d\n", 1 +1, metin[1], metin[1]);
// e
printf("%d. karakter: '%c' , sayısı: %d\n", 2 +1, metin[2], metin[2]);
// l
printf("%d. karakter: '%c' , sayısı: %d\n", 3 +1, metin[3], metin[3]);
// a
printf("%d. karakter: '%c' , sayısı: %d\n", 4 +1, metin[4], metin[4]);
// m
printf("%d. karakter: '%c' , sayısı: %d\n", 5 +1, metin[5], metin[5]);
// !
```

Çıktısı:

```
1. karakter: 'S' , sayısı: 83
2. karakter: 'e' , sayısı: 101
3. karakter: 'l' , sayısı: 108
4. karakter: 'a' , sayısı: 97
5. karakter: 'm' , sayısı: 109
6. karakter: '!' , sayısı: 33
```

Peki, ben bunun 7. karakterine bakmak istesem, ne olurdu?

```
printf("%d. karakter: '%c' , sayısı: %d\n", 6 +1, metin[6], metin[6]);
// ????
```

Çıktısı:

```
7. karakter: '' , sayısı: 0
```

Hmm, boş bir karakter verdi... Hadi başka bir metin ile tekrar deneyelim:

```c
char *metin = "robot";
printf("%s\n", metin);

printf("%d. karakter: '%c' , sayısı: %d\n", 0 +1, metin[0], metin[0]);
// 'r'
printf("%d. karakter: '%c' , sayısı: %d\n", 1 +1, metin[1], metin[1]);
// 'o'
printf("%d. karakter: '%c' , sayısı: %d\n", 2 +1, metin[2], metin[2]);
// 'b'
printf("%d. karakter: '%c' , sayısı: %d\n", 3 +1, metin[3], metin[3]);
// 'o'
printf("%d. karakter: '%c' , sayısı: %d\n", 4 +1, metin[4], metin[4]);
// 't'
printf("%d. karakter: '%c' , sayısı: %d\n", 5 +1, metin[5], metin[5]);
// Yine mi boş?
```

Çıktısı:

```
robot
1. karakter: 'r' , sayısı: 114
2. karakter: 'o' , sayısı: 111
3. karakter: 'b' , sayısı: 98
4. karakter: 'o' , sayısı: 111
5. karakter: 't' , sayısı: 116
6. karakter: '' , sayısı: 0
```

Evet, gördüğünüz üzere, yine metnin sonuna bir boş karakter ekleniyor. Peki neden?

Çünkü, C Metinleri, metin sonunun geldiğini anlamak için metnin sonundan sonra koydukları bu boş karakteri ararlar.
Hatta, çalışma şekilleri şu şekilde kısaca açıklanabilir:

1. İşaretçinin o anda işaret ettiği karakteri al: `char karakter = *isaretci;`
2. Karakteri ekrana yazdır
3. İşaretçiyi bir ileriye al: `isaretci = isaretci + 1`
4. Eğer şimdiki işaret ettiği karakter boş karakter değilse, 1. adıma dön.

"Eğer mi?" diyebilirsiniz, bende cevap veriyorum: Evet, *eğer*. Programlar bir koşulun doğru olup olmadığına göre farklı işlemler yapabilir.

Gelin ilk eğer'imiz'i yazalım.

Programlarda koşullar (eğer, `if`, `else`)

Her gün, rutinimiz farklı şeylerden dolayı değişebiliyor.

Mesela, kahvaltı için yumurtalı bir şeyler yapacakken bir baktınız ki evde yumurta bitmiş. Kahvaltı yapabilmek için bir markete gittiniz, yumurta aldınız, döndünüz ve kahvaltı hazırlamaya devam ettiniz.
Bunun algoritmasını şu şekilde açıklayabiliriz: **Eğer yumurta bittiyse, git yumurta al.**

C içinde tam sayılarda ve diğer veri tiplerindeki karşılaştırmayı hatırlarsanız, karşılaştırma doğru bir sonuç verdiğinde `1` geliyordu. `if` blokları da bunu kullanarak çalışır. `else` özel kod parçası da `if` bloğu değerinin 0 olduğu, yani çalışmayacağı zaman çalışır.

Önceki örneğimizin kodunu yazmak istersek:

```c
#include <stdio.h>
int main(){
  int yumurta_sayisi;
  printf("Evde kaç yumurta var?: ");
  scanf("%d", &yumurta_sayisi);
  if(yumurta_sayisi == 0){
    printf("Yumurta al, sonra da kahvaltı hazırla." "\n");
  } else {
    printf("Kahvaltı hazırla." "\n");
  }
}
```

`if`in anatomisine bakmak istersek, şunu görürüz:

```c
if (bir_deger)
{
    //bir_deger 0'dan farklıysa çalışacak kod bloğu
}
else
{
    //bir_deger 0 ise çalışacak kod bloğu.
}
```

Peki, bu kod blokları nelerdir?

Kod blokları, süslü parantezler (`{` `}`) ve bunların arasına yazdığımız kodlardır.
Ayrıca, tek bir komut işleyeceksek, kod bloğu için süslü parantez kullanmamıza gerek yok!
Gelin, biraz önceki örneğin `if` kısmındaki süslü parantezleri kaldıralım:

```c
if(yumurta_sayisi == 0) printf("Yumurta al, sonra da kahvaltı hazırla."
"\n");
else printf("Kahvaltı hazırla." "\n");
```

Kimilerine göre daha okunaklıdır, kimine göre değildir. Kullanma seçimi sizde.

Her programımıza yazdığımız koddaki kod bloğuna dikkat çekmek isterim.
Hangisi mi? `main` fonksiyonunun kod bloğuna.

```c
int main() {}
```

Bu kod bloğunun içine başka kod bloğu yazabiliyoruz. Peki, bunu `if` ile yapabilir miyiz?
Evet!

```c
if(birinci_kosul){
    printf("Birinci koşul sağlandı.\n");
    if(ikinci_kosul){
        printf("İkinci koşul da sağlandı!\n");
        if (ucuncu_kosul) printf("Hatta üçüncü koşul bile sağlandı!");
    }
}
```

`else` içine `if` kod bloğu yazarak birden fazla durum için kod yazabiliriz.

```c
if( yag_var_mi ) printf("Yağ var.");
else if( bal_var_mi ) printf("Yağ yok ama bal var");
else printf("Ne yağ ne de bal var");
```

`if`'i belirli değerleri kontrol etmek için kullanma

Doğum gününe göre isim oluşturan bir kodunuz var diyelim:

```c
#include <stdio.h>
int main(){
    int ay;
    printf("Doğum gününüzün ayını girin: ");
    scanf("%d", &ay);
    if(gun == 1) {
        printf("1. ay: ");
        printf("İnci\n");
    }
    else if (gun == 2) {
        printf("2. ay: ");
        printf("Sami\n");
    }
    else if (gun == 3) {
        printf("3. ay: ");
        printf("Selami\n");
    }
    //böyle böyle gidiyor...
    return 0;
}
```

Bunun yerine `switch(){case...}` bloğu kullanabiliriz:

```c
#include <stdio.h>
int main(){
    int ay;
    printf("Doğum gününüzün ayını girin: ");
    scanf("%d", &ay);
    switch(ay){
        case 1:
            printf("1. ay: ");
            printf("İnci\n");
            break;
        case 2:
            printf("2. ay: ");
            printf("Sami\n");
            break;
        case 3:
            printf("3. ay: ");
            printf("Selami\n");
            break;
        //böyle böyle gidiyor...
    }
    return 0;
}
```

`switch-case` ile `if`'in farkları

`switch-case` kod **akışı** kullanırken, `if` kod **blokları** kullanır.

Nedir bu kod akışı?
Basitçe, switch-case içine gelen değer, eşleştiği bir değerin başlangıç koduna gelir, sonra da **`break` komutunu görene kadar veya kod bloğunun sonuna kadar devam eder**. İkinci durumun olması - yani bir karşılaştırmanın kodundan başlayıp diğerine geçmesine - fall-through (düşme) denir.

Bazı algoritmalar için bu işlem gerekli olabilir.

Hadi, düşen `switch-case`li bir kod parçası yazalım:

```
switch(bas_harf){
    case 'A':
        printf("Büyük ");
    case 'a':
        printf("A. \n");
        break;
    case 'B':
        printf("Bababababa\n");
        break;
    //Daha fazla harf için test ettiğimizi düşünün.
}
```

Ve buna girdi olarak `'A'` verdiğimizi düşünelim. Çıktımız şu şekilde olacaktı:

```
Büyük A.
```

Eğer girdimiz `'a'` olsaydı, çıktımız şu şekilde olurdu:

```
A.
```

Peki, bunu `if` ile yapamaz mıyız? Yapabiliriz!

```
if(bas_harf == 'A'){
    printf("Büyük ");
    printf("A. \n");
}
else if (bas_harf == 'a')
    printf("A. \n");
else if (bas_harf == 'B')
    printf("Bababababa\n");
```

Her ne kadar "yapabiliyor" olsak da çok bariz bir şey var: Kod tekrarlıyoruz.
Kod tekrarlamak genel olarak kötü bir şeydir. Kodun daha sonradan üzerinde çalışmasını zorlaştırır.