23. Gösterici Gösteren Göstericiler

Jump to bottom

Necati Ergin edited this page on 28 Feb · 2 revisions

Gösterici değişkenler `(pointer variables)` değerleri adres olan yani adres bilgisi tutan değişkenlerdir.

```
int *ptr;
```

gibi bir tanımlamayla ismi `ptr` olan bir değişken oluşturulmuş olur. Programın çalışma zamanında bu nesne için bellekte ayrılan yer sistemden sisteme değişebilecek şekilde `2`, `4` ya da `8` `byte` olabilir. `ptr` değişkeni `int *` türündendir. `ptr` bir değişken olduğuna göre, `ptr` değişkeninin de adresinden söz edilebilir, değil mi?
Adres operatörü ile `ptr` değişkeninin kendi adresini elde edebiliriz:

```
&ptr
```

Yukarıdaki ifadenin türü nedir? Bu ifadenin türü, `int *` türünden bir nesnenin adresi olan türdür. Bu tür C dilinde

```
int **
```

olarak ifade edilir. O zaman yukarıdaki `ptr` gibi bir değişkenin adresi, bir başka değişkende tutulmak istenirse, bu adresi tutacak değişkenin türü `int **` olmalıdır. Aşağıdaki koda bakalım:

```
int main()
{
        int x = 10;
        int *ptr = &x;
        int **pp = &ptr;
        //
}
```

`main` işlevinde `int **` türünden olan `pp` isimli değişkene, `int *` türünden olan `ptr` değişkeninin adresi ile ilk değer veriliyor. Bunun anlamı şudur: `ptr` değişkeninin değeri, `p` değişkeninin adresidir. Başka bir deyişle, `pp` isimli gösterici değişken bir başka gösterici değişken olan `ptr` 'yi göstermektedir. Bu durumda

```
*pp
```

ifadesi `pp` nesnesinin gösterdiği nesne, `ptr` nesnesinin kendisidir. `*pp` ifadesine yapılan atama aslında `ptr` nesnesini değiştirir. Aşağıdaki programı inceleyin:

```
#include <stdio.h>

int main()
{
        int x = 10;
        int y = 20;
        int *ptr = &x, **pp = &ptr;

        printf("x = %d\n", x);
        printf("y = %d\n", y);
        *ptr = 100;
        *pp = &y;
        *ptr = 200;

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

        **pp = 2000;

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

        return 0;
}
```

`main` işlevinde tanımlanan `int *` türden `ptr` isimli gösterici değişkene `int` türden `x` değişkeninin adresi ile, `int **` türden `pp` isimli gösterici değişkene ise `ptr` 'nin adresi ile ilk değer veriliyor.

```
*ptr = 100;
```

deyimiyle, `ptr` gösterici değişkeninin gösterdiği nesneye, yani `x` değişkenine `100` değeri atanmış oluyor.

```
*pp = &y;
```

deyimi ile `pp` gösterici değişkeninin gösterdiği nesneye, yani `p` gösterici değişkenine bu kez `y` değişkeninin adresi atanıyor. Bu atamadan sonra yürütülecek

```
*ptr = 200;
```

ataması ile artık `ptr` gösterici değişkeninin gösterdiği nesneye yani `y` değişkenine `200` değeri atanmış olur. Şimdi de aşağıdaki deyimi inceleyelim:

```
**pp = 2000;
```

İçerik `(dereferencing )` operatörünün oluşturduğumuz operatör öncelik tablomuzun ikinci seviyesinde yer aldığını ve sağdan sola öncelik yönüne sahip olduğunu biliyorsunuz. Bu durumda önce

```
*pp
```

ifadesi ile `pp` gösterici değişkeninin gösterdiği nesneye yani `ptr` nesnesine erişilir. Daha sonra

ifadesiyle de `pp` gösterici değişkeninin gösterdiği nesnenin gösterdiği nesneye, yani `p` değişkeninin gösterdiği nesneye, yani `y` değişkenine erişilir. Bu deyimin yürütülmesiyle `y` değişkenine `2000` değeri atanmış olur. `**pp` ifadesi, `pp` 'nin gösterdiği nesnenin gösterdiği nesneye, yani `y` nesnesine karşılık gelir.

Şimdi de aşağıdaki kodu inceleyelim:

```
#include <stdio.h>

int main()
{
        int a[5] = {10, 20, 30, 40, 50};
        int *p = a;
        int **pp = &p;

        ++*pp;
        ++**pp;
        (*pp)[2] = -1;

        for (int k = 0; k < 5; ++k)
                printf("%d ", a[k]);

        return 0;
}
```

Yukarıdaki kodda `p` gösterici değişkeni `a` dizisini, `pp` gösterici değişkeni ise `p` gösterici değişkenini gösteriyor.

```
++*pp;
```

deyimi ile `pp` 'nin gösterdiği nesneyi yani `p` 'yi `1` arttırmış oluyoruz. `p` 'nin `1` artması demek artık `a` dizisinin `1` indisli elemanını gösteriyor olması demek, değil mi?

```
++**pp;
```

deyimi ile ise `pp` 'nin gösterdiği nesnesinin gösterdiği nesneyi yani `a` dizisinin `1` indisli elemanını `1` arttırmış oluyoruz.

```
(*pp)[2] = -1;
```

deyiminde ise, `*pp` gösterici değişkeninin yani `p` 'nin gösterdiği dizinin `2` indisli elemanına `-1` değeri atanıyor. `*pp` , yani `p` , `a` dizisinin `1` indisli elemanını gösterdiğine göre bu ifade ile `a` dizisinin `3` indisli elemanına `-1` değeri atanmış olur.

yerel bir gösterici değişkenin değerini değiştiren işlevler

Yerel bir nesnenin değerini değiştirecek bir işlev, yerel nesnenin adresi ile çağrılmalıdır `(call by reference`). Yerel bir gösterici değişkenin değerini değiştirecek bir işlev de, yerel göstericinin değerini değil adresini almalıdır:

```
void swap_ptr(int **p1, int **p2)
{
        int *ptemp = *p1;
        *p1 = *p2;
        *p2 = ptemp;
}
```

Örnek kodda `int *` türünden iki nesnenin değerini takas etmek amacıyla `swap_ptr` isimli bir işlev tanımlanıyor. İşlevin `int **` türünden iki parametresi olduğunu görüyorsunuz. Bu işlev şüphesiz değerlerini takas edeceği nesnelerin adresleri ile çağrılmalıdır:

```
#include <stdio.h>

void swap_ptr(int **p1, int **p2);

int main()
{
        int x = 10, y = 20, *p = &x, *q = &y;

        printf("*p = %d\n", *p);
        printf("*q = %d\n", *q);
        swap_ptr(&p, &q);
        printf("*p = %d\n", *p);
        printf("*q = %d\n", *q);

        return 0;
}
```

`main` işlevi içinde `int` türden `x` ve `y` isimli değişkenler ile `p` ve `q` isimli gösterici değişkenler tanımlanıyor. Verilen ilk değerlerle `p`, `x`'i, `q` ise `y`'yi gösteriyor. Daha sonra, çağrılan `swap_ptr` işlevine `p` ve `q` gösterici değişkenlerinin adresleri gönderiliyor. İşleve yapılan çağrıdan sonra artık, `p` gösterici değişkeni `y` nesnesini, `q` gösterici değişkeni ise `x` nesnesini gösterir hale geliyor. Şimdi de aşağıdaki işlevi inceleyin:

```
void pp_swap(int **p1, int **p2)
{
        int ptemp = **p1;
        **p1 = **p2;
        **p2 = ptemp;
}
```

`pp_swap` isimli işlev hangi nesneleri takas ediyor? İşlevimiz `int *` türünden iki göstericinin adresini alarak bunların gösterdiği `int` türden nesneleri takas ediyor:

```
#include <stdio.h>

int main()
{
        int x = 10, y = 20, *px = &x, *py = &y;

        pp_swap(&px, &py);

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

        return 0;
}
```

bir gösterici dizisi üzerinde işlem yapan işlevler

Bir dizi üzerinde işlem yapan işlevin dizinin başlangıç adresi ile dizinin boyutunu alması gerektiğini biliyorsunuz. `int` türden bir dizi ile ilgili işlem yapan bir işlevin bildirimi şöyle olabilir:

```
void processArray(int *p, size_t size);
```

Böyle bir işlev için eşdeğer bir başka bildirim şöyle olabilir:

```
void processArray(int p[], size_t size);
```

Böyle bir işlev dizinin ilk elemanının adresi ve dizinin boyutu ile çağrılır, değil mi?

```
int a[10];
```

gibi bir dizi söz konusu olduğunda, dizi ismi olan `a` bir ifade içinde kullanıldığında otomatik olarak bu dizinin adresine dönüştürülür `(array to pointer conversion / array decay)`. Yani derleyici açısından bakıldığında `a` ifadesi

```
&a[0]
```

ifadesine eşdeğerdir. Bu işlev

```
processArray(a, 10);
```

biçiminde çağrılabilir.

Bu kez elemanları `int *` türden olan bir dizi tanımlanmış olsun:

```
int *a[100];
```

Yine `a` ifadesi bir işleme sokulduğunda, bu dizinin ilk elemanı olan nesnenin adresine yani dizinin başlangıç adresine dönüştürülür.

Bu dizinin ilk elemanı olan nesne `int *` türünden olduğuna göre, bu nesnenin adresi `int **` türündendir. Böyle bir dizi üzerinde işlem yapacak işlev, bu dizinin başlangıç adresi ile boyutunu alacağına göre, şöyle bildirilmelidir:

```
void processArray(int **parray, size_t size);
```

Böyle bir işlev için alternatif bir bildirim şöyle olabilir:

```
void processArray(int *parray[], size_t size);
```

Derleyicinin yaptığı kontroller açısından her iki bildirim eş değerdedir. Bazı kodlayıcılar, işlevin bir dizi adresi istediği vurgusunu yaptığı için yukarıdaki bildirimi tercih ederler.Bu işlev

```
processArray(a, 10);
```

biçiminde çağrılabilir.

gösterici gösteren gösterici ve `const` anahtar sözcüğü

Gösterici gösteren göstericilerde `const` anahtar sözcüğünün kullanıldığı yere bağlı olarak `const` anahtar sözcüğünün anlamı ve buna bağlı olarak derleyicinin yaptığı kontroller değişir. Şimdi tüm senaryoları tek tek inceleyelim. Önce aşağıdaki koda bakalım:

```
int main()
{
        int x = 10;
        int y = 10;

        int *p1 = &x;
        int *p2 = &y;
        int **const pp = &p1;
        //pp = &p2; //gecersiz
        *pp = &y; //gecerli
        **pp = 40; //gecerli

        return 0;
}
```

Yukarıdaki kodda `pp` göstericisinin

```
int **const pp = &p1
```

biçiminde tanımlandığını görüyorsunuz. Burada `const` olan `pp` değişkeninin kendisidir. `pp` değişkeni hayatı boyunca, kendisine ilk değer olarak verilen `p1` değişkeninin adresini tutacaktır. `pp` değişkeninin değerini değiştirmeye yönelik ifadeler geçersizdir. Ancak `*pp` nesnesine, yani `pp` 'nin gösterdiği nesneye ya da `**pp` 'ye, yani `pp` 'nin gösterdiği nesnenin gösterdiği nesneye yapılacak atamalar geçerlidir. Burada tanımlanan `pp` 'yi sözel olarak şu şekilde ifade edebiliriz: Bir göstericiyi gösteren kendisi `const` gösterici `(const pointer to int *`).

Şimdi de aşağıdaki koda bakalım:

```
int main()
{
        int x = 10;
        int y = 10;

        int *p1 = &x;
        int *p2 = &y;
        int *const *pp = &p1;
        pp = &p2; //gecerli
        //*pp = &y; //gecersiz
        **pp = 40; //gecerli
```

```
        return 0;
}
```

Burada ise `pp` isimli göstericinin

```
int *const *pp = &p1;
```

biçiminde tanımlandığını görüyorsunuz. Bu durumda `const` olan `*pp` nesnesidir. `*pp` nesnesine yapılan atamalar geçersizdir. Yani `pp` 'nin gösterdiği nesneyi `pp` yoluyla değiştiremeyiz. `pp` ve `**pp` nesnelerine yapılan atamalar geçerlidir. Burada tanımlanan `pp` 'yi sözel olarak şu şekilde ifade edebiliriz: Kendisi `const` bir göstericiyi gösteren gösterici `(pointer to const pointer to int / pointer to top level const pointer)`.

Son olarak aşağıdaki koda bakalım:

```
int main()
{
        int x = 10;
        int y = 10;

        int *p1 = &x;
        int *p2 = &y;
        const int **pp = &p1;
        pp = &p2; //gecerli
        *pp = &y; //gecerli
        **pp = 40; //gecersiz

        return 0;
}
```

Burada `pp` isimli göstericinin

```
const int **pp = &p1;
```

biçiminde tanımlandığını görüyorsunuz. Tanımlama aşağıdaki gibi yapılsaydı da bir anlam değişikliği olmayacaktı:

```
int const **pp = &p1;
```

Bu durumda `const` olan `**pp` nesnesidir. `**pp` nesnesine yapılan atamalar geçersizdir. Yani `pp` 'nin gösterdiği göstericinin gösterdiği nesneyi `pp` yoluyla değiştiremeyiz. `pp` ve `*pp` nesnelerine yapılan atamalar geçerlidir. Burada tanımlanan `pp` 'yi sözel olarak şu şekilde ifade edebiliriz: Gösterdiği yer `const` olan bir göstericiyi gösteren gösterici. `(pointer to pointer to const int / pointer to low level const pointer)`.

Aklınızda kalması için şu basit kuralı anımsayın: `const` anahtar sözcüğü neden önce geliyorsa `const` olan odur:

```
int **const pp = &p1;
```

`const` anahtar sözcüğü `pp` 'den önce gelmiş. `const` olan `pp` . `pp` 'ye değer atayamayız.

```
int *const *pp = &p1;
```

`const` anahtar sözcüğü `*pp` 'den önce gelmiş. `const` olan `*pp` . `*pp` 'ye değer atayamayız.

```
const int **pp = &p1;
```

`const` anahtar sözcüğü `**pp` 'den önce gelmiş. `const` olan `**pp` . `**pp` 'ye değer atayamayız.

`const` anahtar sözcüğü bu yerlerin hepsinde birden kullanılabilir:

```
const int * const* const pp = &p1;
```

`pp` kendisi `const` ve gösterdiği yer `const` olan bir göstericiyi gösteren kendisi `const` bir göstericidir. `(pp is a const pointer to a const pointer to const int)`

Şimdi de aşağıdaki programı inceleyin:

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

void swap_ptr(const char **p1, const char **p2)
{
        const char *p_temp = *p1;
        *p1 = *p2;
        *p2 = p_temp;
}

void display_str_array(const char *const *p, size_t size)
{
        size_t k;

        for (k = 0; k < size; ++k)
                printf("%s ", p[k]);
        printf("\n");
}

void sort_str_array(const char **p, size_t size)
{
        size_t i, k;
```

```
        for (k = 0; k < size - 1; ++k) {
                for (i = 0; i < size - 1 - k; ++i) {
                        if (strcmp(p[i], p[i + 1]) > 0)
                                swap_ptr(p + i, p + i + 1);
                }
        }
}

int main()
{
        const char *pnames[10] = {"Eda", "Abdurrahman",
"Berk", "Zarife", "Yusuf",
                "Levent", "Sezgi", "Deniz", "Ufuk", "Cansu" };

        display_str_array(pnames, 10);
        sort_str_array(pnames, 10);
        getchar();
        display_str_array(pnames, 10);

        return 0;
}
```

Yukarıdaki programda tanımlanan işlevleri inceleyelim: `swap_ptr` işlevi kendisine gönderilen `const char *` türünden iki gösterici değişkeni takas ediyor. `display_str_array` işlevi adresini ve boyutunu aldığı elemanları `const char *` türünden olan bir gösterici dizisinin elemanlarının gösterdiği yazıları standart çıkış akımına yazdırıyor. `sort_str_array` işlevi adresini ve boyutunu aldığı elemanları `const char *` türünden olan bir dizinin elemanlarını, gösterdiği yazılar küçükten büyüğe doğru olacak şekilde sıralıyor.

Pages 19

Find a Page...

Home

00. Necati Ergin C Ders Notları

021 _Bool Türü

06. virgül operatörü

07. sizeof Operatörü

08. Koşul Operatörü

13. switch Deyimi

22. Yazı Sabitleri (String Literals)

23. Gösterici Gösteren Göstericiler

24. void Göstericiler(eksik)

26. İşlev Göstericileri 1 (Function Pointers 1)

28. Çok Boyutlu Diziler (Multi dimensional Arrays)

36. Bitsel İşlemler (Bitwise Operations)

42. Variadic İşlevler (Variadic Functions)

43. Bileşik Sabitler (Compound Literals)

Show 4 more pages…

Clone this wiki locally

```
https://github.com/necatiergin/C_Ders_Notlari.wiki.git
```