3

Pointers

Pointer, başka bir değişkenin bellekteki adresini işaret eden bir değişkendir. O nedenle, pointere *işaretçi, gösterici* adları verilir. Değişkenler için olduğu gibi, pointerin değeri, işaret ettiği adresteki değerdir. Nasıl ki, değişken bildiriminde, değişkenin veri tipini belirtiyorsak, pointer bildiriminde de, onun işaret edeceği bellek adresine konulabilecek değerin veri tipini belirtmeliyiz. Aslında bütün değişkenler, bellekte birer adres işaret ederler. Dolayısıyla, bütün değişkenler birer pointerdir. Ama, C dilinde pointerlerle *pointer aritmetiği* denilen işlemler yapılır. Bu işlemler, bazı işlerin yapılmasını çok kolaylaştırır. Pointer aritmetiğini biraz sonra açıklayacağız.

Bildirimi sırasında, değişkenin pointer olduğunu belirtmek için, değişken adının önüne yıldız (*) karakteri konulur:

```
int gelir;
int *gelirPtr;
```

bildirimilerinin birincsi *int* değer alan *gelir* adlı bir değişkeni bildiriyor. İkincisi *int* tipi veri içerecek bir bellek adresini işaret eden **gelirPtr* adlı bir pointer bildirimidir. Her ikisi de *int* tipinden olduğu için, istenirse iki bildirim tek bildirim halinde yazılabilir:

```
int gelir, *gelirPtr;
```

Pointer bildiriminde, değişken adının sonuna *Ptr* ekledik. Bunu yapmak zorunlu değildir. Derleyici yalnızca değişkenin önündeki (*) karakterine bakar. *Ptr* eklememizin nedeni, kaynak programı okuyan kişinin pointer değişkenlerini kolay algılaması içindir. Zorunlu olmayan bu işi alışkanlık edinmek yararıdır.

3.1 Pointer'in Yararları

Her veri tipinden pointer tanımlanabilir. O nedenle, bazı kaynaklara göre, pointer, C dilinde türetilmiş (derived) bir veri tipidir. Pointere atanan değerler, değişken adresleridir. Pointer kendisine atanan adresteki değeri gösterir. Dolayısıyla, bellekteki verilerle işlem yapmak için güçlü bir araçtır. Pointerlerin başlıca yararları şöyle sıralanabilir:

1. Arrayler ve veri tabloları üzeride işlemleri kolaylaştırır.
2. Dinamik adres yönetimini sağlar.
3. *Değerle çağrı*'nın asıl olduğu fonksiyonlarda, parametrelere pointerler atanabilir ve böylece *referan ile çağrı* oluşur. Bu oluşumda, parametre değerleri değişebilir ve fonksiyon birden çok değer verebilir. (Bununla ilgili örnekleri birazdan inceleyeceğiz).
4. Pointerler, doğrudan adresleri gösterdiği için, programın koşma hızını artırır.
5. Pointer bağlı listeler, kuyruklar, yığıtlar (stacks) gibi yapılarla işlem yapmayı kolaylaştıran bir araçtır.

3.1.1 Bellek Adresleri

Bilgisayarın CPU'dan sonra en önemli bileşeni anabelleğidir. Buna İngilizcede RAM (Random Access Memory - Seçkili Erişilir Bellek) denilir. Türkçe'de ona *ana bellek* diyoruz. Donanımsal yapıları farklı olan RAM türleri vardır. Ama programcı için, RAM'in donanımsal yapısı önem taşımaz. Hepsi benzer kuralla işlev yapar. Genellikle, bellek adresleri 1 byt'lık (8 bit'lik) gözelerden (hücre) oluşur. Her gözenin pozitif tam sayı ile belirlenen bir adresi vardır. Göze adresleri artan sırada dizilirler. İşletim sistemi, her gözenin adresini bilir. *Seçkili Erişilir Bellek (RAM)* denmesinin nedeni, istenilen gözeye doğrudan erişilebiliyor olmasıdır.

Bilgisayar çalışırken, işletim sisteminin komutları, uygulama programının komutları ve veriler anabellekte tutulur. Uygulama programının gerekli gördüğü komutlar ve veriler anabelleğe yerleştirilir. Bu işi işletim sistemi ve uygulama programı birlikte yaparlar. Anabellek, içine veri yazılan hücrelerden oluşur. Hücrelerin her birisi numarasıyla belirlidir. Buna hücrenin adresi diyoruz. İşletim sistemi ve dolayısıyla derleyici ana bellekteki her hücrenin numarasını (adresini) bilir.

Ana bellekte göze sayısı ne kadar çoksa, orada tutulanlar (veriler, komutlar vb) o kadar çok olur. Bu da programın daha hızlı çalışmasını sağlar. Tabii, RAM büyüklüğü sistemden sisteme değişir. Ama hepsinin bir sınırı vardır. Çok sayıda veri işleyen programlar, RAM'e sığmayan verileri hard disk, flopy disk vb gibi ikincil bir kayıt ortamına yazar ve gerektikçe oradan alır. Oraya süpürme alanı *(swap space)* denilir. Genellikle büyük veri tabanlarında bu iş kaçınılmazdır. İkincil kayıt ortamı ile veri alış-verişi RAM'dekilere göre daha yavaştır.

C dilinde bir değişkenin anabellek adresi, değişkenin önüne konulan & karakteri ile belirtilir. Pointerlerin değerleri de bellek adresleri olduğuna göre * ile & karakterleri birbirlerinin tümleyeni olarak işlev yaparlar. Bunu bir örnekle açıklayalım.

```
int x = 77;

int *p;

p = &x;
```

bildirimlerinden birincisi *int* tipinden x değişkeni tanımlıyor ve ona 77 değerini atıyor. İkincisi *int* tipinden verilere ayrılan bir adresi gösterecek olan *p* pointer değişkenini tanımlıyor. *p* yerine istenilen başka bir ad verilebilir. Bu aşamada *p* pointeri henüz bir adres göstermiyor. Üçüncü deyim, *p* pointer değişkenine x değişkeninin adresini atıyor. Böylece *p* pointeri x 'in adresini işaret etmeye başlıyor; o adreste 77 değeri vardır. Dolayısıyla *p* pointeri 77 değerini işaret etmeye başlıyor. Buna göre,

```
p = &x;

*p = x;
```

atamaları birbirlerine denk olur. Bunu bir örnekle gösterelim:

Program 3.1.

```
#include <stdio.h>

int main( void ) {

  int x, *p;

  x = 77;
  p = &x;


  printf( "\nx degiskeninin degeri   : %d"   , x );
  printf( "\nx degiskeninin adresi   : %d"   , &x );
  printf( "\n");
  printf( "\n*p in degeri             : %d"   , *p );
  printf( "\n  *p = x                 :%ld = %ld"   , *p, x );
  printf( "\n");
```

```
    printf( "\np pointerinin degeri     : %d"   , p );

    printf( "\n   p = &x   (hex)          :%x = %x"   , p, &x );
    printf( "\n   p = &x   (long)         :%ld = %ld"   , p, &x );
}
```

```
/**
  x degiskeninin degeri    : 77
  x degiskeninin adresi    : 2293316

  *p in degeri             : 77
     *p = x                : 77 = 77

  p pointerinin degeri     : 2293316
    p = &x   (hex)         :22fe44 = 22fe44
    p = &x   (long int)    :2293316 = 2293316
*/
```

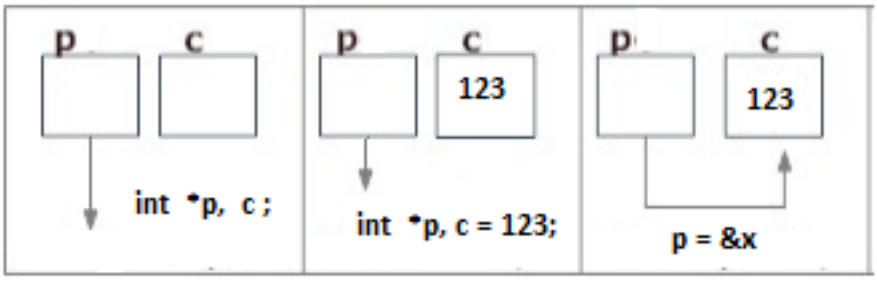
Açıklama:

Çıktının 6. satır $*p = x$ eşitliğini, 9. ve 10 satırları ise $p = \&x$ eşitliğini *hexadecimal* ve *long int* olarak gösteriyor.

Uyarı: Pointer bildiriminde * karakterini iki farklı konumda kullanabiliriz:

1. int *p;
2. int* p;

Bu derste kullandığımız *gcc* derleyicisi bu iki bildirimi eşdeğer sayar. Notasyonlarda birliği sağlamak için biz ilk gösterimi kullanacağız. Program 3.2 ikinci bildirimin de kullanılabileceğini gösteriyor.



Şekil 3.1: Pointer'in İşlevi

Program 3.2.

```c
#include <stdio.h>

int main() {
  int* p;
  int c;

  c=123;
  p = &c;
  printf("c = 123 ise\n");
  printf("c nin adresi:%d\n",&c);
  printf("c nin degeri:%d\n\n",c);
  printf("p pointerinin degeri:%d\n",p);
  printf("p pointerinin gosterdigi deger:%d\n\n",*p);

  return 0;
}
```

```
/**
c = 123 ise
c nin adresi:2293316
c nin degeri:123

p pointerinin degeri:2293316
p pointerinin gosterdigi deger:123
*/
```

3.2 NULL Pointer

NULL, *stdio.h* dosyasında tanımlı sembolik bir sabittir. Türkçe'de *boş*, *hiç* anlamlarına gelir. Bir pointerin hiçbir adresi işaret etmediğini belirtmek için, ona NULL değer atamak iyi bir alışkanlıktır. Aksi halde, pointer çöp (garbage) toplar; yani herhangi bir değer alabilir, dolayısıyla öngörülemeyen bir bellek adresini işaret edebilir. NULL değer atanmış pointerin; yani hiçbir adresi işaret etmeyen pointerin değeri daima 0 olur. Kendisine hiç değer atanmamış, NULL atanmış ya da bir değişken adresi atanmış pointerlerin değerlerini Program 3.3 örneğinden görebiliriz:

Program 3.3.

```c
#include <stdio.h>

int main () {
  int x, *p1, *p2 , *p3;
  x = 123;
  p2 = NULL;
  p3 = &x;

  printf("Deger atanmamis p1 pointerinin degeri : %x\n", p1  );
```

```
    printf("p2 = NULL atanmis pointerin degeri     : %x\n", p2  );
    printf("p3= &x atanmis pointerinin degeri      : %x\n", p3  );
    printf("&x   adresinin degeri                  : %x\n", p3  );

    return 0;
}
```

```
/**
Deger atanmamis p1 pointerinin degeri : 1e
p2 = NULL pointerinin degeri          : 0
p3        pointerinin degeri          : 22fe34
&x   adresinin degeri                 : 22fe34
```

Açıklama:

Çıktının 2. satırında hiç değer atanmamış bir pointerin aldığı değer görülüyor. Bu değer öngörülemez çöp (garbage) değerdir. Pointere NULL ya da bir adres atanmadan önce kullanılması hatalara yol açar.

Çıktının 3. satırı NULL atanmış bir pointerin değerinin 0 olduğunu gösteriyor.

Çıktının 4. ve 5. satırları bir değişken adresi atanmış pointerin değerinin atanan adres olduğunu gösteriyor.

3.3 * ve & Operatörleri

x değişkeninin adresinin &x ile gösterildiğini biliyoruz. Burada & bir operatördür. Bellek adresini göstermek gibi önemli bir işleve sahiptir. Ama bellek adreslerindeki veriler üzerinde işlem yapmaya elvermez. Bellek adreslerindeki verilerle işlem yapacağımız zaman * operatörünü kullanırız. p pointerinin gösterdiği adresteki değer, *p ile belirtilir. Anımsayacağınız gibi, x değişkeninin adresini p pointerine atamak için

```
p = &x;
```

deyimini kullanıyoruz. Bu atamadan sonra

```
*p = x;
```

olur. Bu nedenle & operatörü ile * operatörü birbirlerinin *tümleyen*'leridir, denilir. * operatörü, adresteki veriyi dolaylı gösteriyor. O nedenle * operatörüne dolaylı adres operatörü (indirection operator) denilir.

Bazı problemlerin çözümünde & peratörü çok etkili olur. Program 3.4 takas işleminde & operatörünün kullanılışını gösteriyor.

Program 3.4.

```c
#include <stdio.h>

void swap(int *i, int *j) {
   int t = *i;
   *i = *j;
   *j = t;
}

void main() {
   int a = 23, b = 47;
   printf("Önce a: %d, b: %d\n", a, b);
   swap(&a, &b);
   printf("Sonra a: %d, b: %d\n", a, b);
}
```

```
/**
  Önce : a = 18, b = 67
  Sonra: a = 67, b = 18
```

3.4 Operatörlerin Öncelikleri

İşlemlerde operatörlerin öncelik sıraları vardır. Kod yazarken bunlara dikkat edilmezse mantıksal hatalar doğar. Parantez kulllanarak, işlemlerde öncelik sırasını istediğimiz gibi ayarlayabiliriz.

Ana bellek adresleri artan sırada ardışık unsigned int (pozitif tam sayı) değerler alır. Dolayısıyla p pointeri bir adres gösteririyorsa, p için $++$ ve $--$ operatörlerini *önel* ya da *sonal* olarak kullanabiliriz. Böylece $++p$, $--p$, $p++$ ve $p--$ simgeleri sayısal değişkenlerde bildiğimiz anlamlarına denk olur. Adresler üzerinde

```
++, -, !, *, &
```

operatörleri işlevselsdir. Bunların öncelik sıraları sağdan sola doğrudur. Şimdiye dek gördüğümüz operatörlerin öncelik sıralarını Tablo 3.1'de olduğu gibi, topluca görmek yararlı olabilir.

3.5 Fonksiyonları Referans İle Çağırma

Fonksiyon çağrısının değerle (call by value) ve referansla (call by reference) olmak üzere iki türlü yapılabileceğini söylemiş, referansla yapılan çağrıyı incelenmeyi pointer kavramından sonraya bırakmıştık. Artık referans ile çağrıyı inceleyebiliriz.

Değerle çağrıda, parametre ya da yerel değişken olarak kullanılsalar bile, fonksiyon bloku dışındaki değişken değerlerinin değişmediğini söyle-

Operatörler	*İşlem Sırası Yönü*	*Tür*
() [ ]	soldan sağa	en yüksek öncelik
+ − ++ −− ! ∗ &.	sağdan sola	birli
∗ / %	soldan sağa	çarpımsal
+ -	solda sağa	toplamsal
< <= > >=	solda sağa	ilşkisel
== !=	soldan sağa	eşitlik
&&	sağdan sola	mantıksal VE
||	sağdan sola	mantıksal VEYA
?:	solda sağa	koşullu deyim
= += -= ∗= /= %=	solda sağa	atama
,	solda sağa	virgül, ayraç

Tablo 3.1: Operatör Öncelikleri

miştik. Ancak, bazen fonksiyon bloku dışındaki değişken değerlerini değiştirmek isteyebiliriz. Bunun için referans ile çağrı yöntemi kullanılır.

Ama öncelikle, programın & ve * operatörleri ile ne kadar kolay ve kısa yazıldığını Program 3.5 örneğinde görecek, sonra, Program 3.6 örneğinde *main()* ve *takas()* fonksiyonlarının her adımda yaptığı işleri printf() ile ekrana yazdıracağız.

Program 3.5.

```
#include <stdio.h>

void takas(int *i, int *j) {
   int t = *i;
   *i = *j;
   *j = t;
}

void main() {
   int a = 3, b = 8;
   printf("Önce  a: %d, b: %d\n", a, b);
   takas(&a, &b);
   printf("Sonra a: %d, b: %d\n", a, b);
}
```

```
/**
Önce : a = 3,  b = 8
Sonra: a = 8,  b = 3
*/
```

Referans ile çağrıyı (call by reference) iyi anlayabilmek için Program 3.6 örneğini satır satır incelemek (izleme, trace) yeterli olacaktır.

Program 3.6.

```c
#include <stdio.h>

void takas(int* , int*  );
int main() {

int a = 3, b = 8;
printf("\nTakastan once:");
printf("\na = %d, b = %d", a, b);
printf("\n&a =%d  ve &b = %d"  , &a,&b);
printf("\n");

takas(&a, &b);
printf("\n\nTakastan sonra:");
printf("\na = %d, b = %d", a, b);
printf("\n&a =%d  ve &b = %d"  , &a,&b);

}

void takas(int *x, int *y) {
  int z;
  printf("\n&x =%d  , &y = %d  , &z = %d"  , &x,&y, &z);
  printf("\n *x =%d  , *y = %d  , z= %d"  , *x,*y, z);

  printf("\n\n");
  printf("z = *x  atamasından sonra:");
  z = *x;
  printf("\n&x =%d  , &y = %d  , &z = %d"  , &x,&y, &z);
  printf("\n *x =%d  , *y = %d  , z= %d"  , *x,*y, z);

  printf("\n\n");
  printf("*x = *y  atamasından sonra:");
  *x = *y;
  printf("\n&x =%d  , &y = %d  , &z = %d"  , &x,&y, &z);
  printf("\n *x =%d  , *y = %d  , z= %d"  , *x,*y, z);


  printf("\n\n");
  printf("*y = z  atamasından sonra:");
  *y = z;
  printf("\n&x =%d  , &y = %d  , &z = %d"  , &x,&y, &z);
  printf("\n *x =%d  , *y = %d  , z= %d"  , *x,*y, z);
}
```

```
/**
Takastan once:
a = 3, b = 8
&a =2293324  ve &b = 2293320

&x =2293280  , &y = 2293288  , &z = 2293260
 *x =3  , *y = 8 , z= 0

z = *x  atamasından sonra:
&x =2293280  , &y = 2293288  , &z = 2293260
 *x =3  , *y = 8 , z= 3

*x = *y  atamasından sonra:
&x =2293280  , &y = 2293288  , &z = 2293260
 *x =8  , *y = 8 , z= 3
```

```
*y = z  atamasından sonra:
&x =2293280  , &y = 2293288  , &z = 2293260
 *x =8  , *y = 3 , z= 3

Takastan sonra:
a = 8, b = 3
&a =2293324  ve &b = 2293320
*/
```

Açıklamalar:

main() içindeki 6.satır a ve b değişkenlerine ilk değerleri atıyor. 8. ve 9. satırlar a ile b nin değerlerini ve adreslerini yazıyor. Çıktının 3. ve 4. satırları bunları gösteriyor.

main() içindeki 12. satır *takas()* fonksiyonunu çağırıyor. 3. satırda, *takas()* fonksiyonunun *prototipine* bakarsak, parametrelerin *int* gösteren *pointer* olduğunu görebiliriz. Bu demektir ki, *takas()* fonksiyonu çağrılırken, parametrelere ancak *int* tipinden değişkenlerin adresleri konulabilir. 12.satır bu işi yapıyor. Çıktının 5.-19.satırları *takas()* fonksiyonu tarafından yazılıyor. a ile b nin adresleri ile x ile y nin adreslerinin farklı olduğuna dikkat ediniz. Her fonksiyon, kendi kapsanma alanında (scope), kendi parametreleri ve yerel değişkenleri için gerekli adresleri ayırır. Değişken adları aynı olsa bile, bu adresler, fonksiyonu çağıran bloktaki adreslerden farklıdır.

takas() içindeki 26.satırda z= *x ataması yapılınca, 27. ve 28. satırların yaptığı iş, çıktının 10.ve 11.satırlarında görülüyor. x, y değişkenlerinin adreslerinin ve değerlerinin değişmediğine, ama x in değerinin z ye atandığına; dolayısıyla *x= 3, *y= 8, z=3 olduğuna dikkat ediniz. Takas işleminin ilk adımı burada atılıyor.

takas() içindeki 32.satırda *x= *y ataması yapılınca, 33. ve 34. satırların yaptığı iş, çıktının 14.ve 15.satırlarında görülüyor. x, y değişkenlerinin adreslerinin değişmediğine, ama *x= 8, *y= 8 ve z=3 olduğuna dikkat ediniz. Bu adımda, &x içindeki değer değişmiştir. Takas işleminin ikinci adımı burada atılıyor.

takas() içindeki 39.satırda *y= z ataması yapılınca, 40. ve 41. satırların yaptığı iş, çıktının 18.ve 19.satırlarında görülüyor. x, y değişkenlerinin adreslerinin değişmediğine, ama *x= 8, *y= 3 ve z=3 olduğuna dikkat ediniz. Takas işleminin üçüncü ve son adımı burada atılıyor.

takas() fonksiyonunun işlevi bu adımda bitiyor. Program akışı, *takas()* fonksiyonunu çağıran *main()*'e dönüyor ve kaldığı 12.satırdan devam ediyor.

main() içindeki 14. ve 15.satırlar, a ile b değişkenlerinin değerlerini ve

adreslerini yazdırıyor. adreslerin değişmediğini, ama içlerindeki değerlerin takas edildiğini görüyoruz. Böyle olması doğaldır; çünkü *takas()* fonksiyonu onların içindeki *x ve *y değerlerini takas etti.

Program 3.7.

```
#include <stdio.h>

int kareBul ( int* ) ;
int main( void )
 {
int number = 7;
printf ( "\nSayinin ilk degeri   :%d" , number ) ;

kareBul ( &number ) ;
printf ( "\nSayinin yeni degeri  : %d\n" , number ) ;
}

int kareBul ( int *nPtr ){
*nPtr = *nPtr * *nPtr   ;
return  *nPtr;
}

/**

*/
```