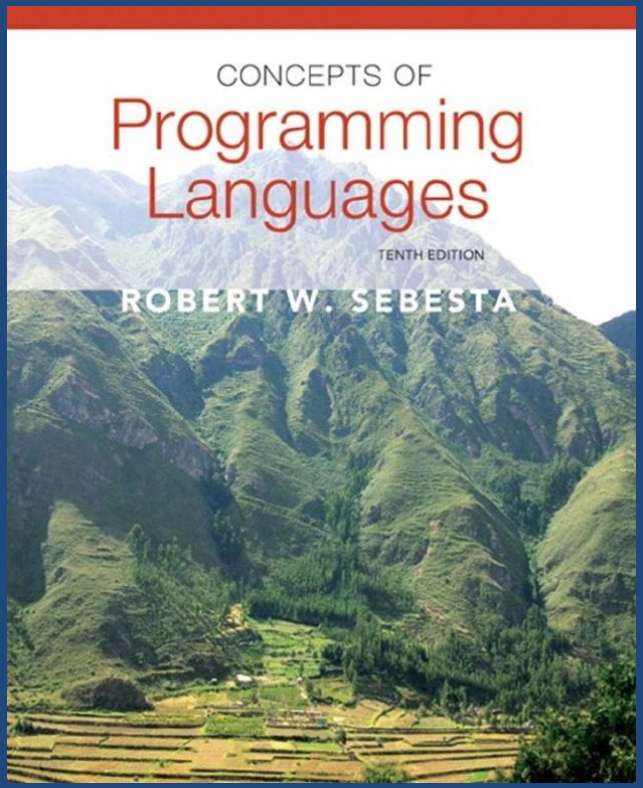
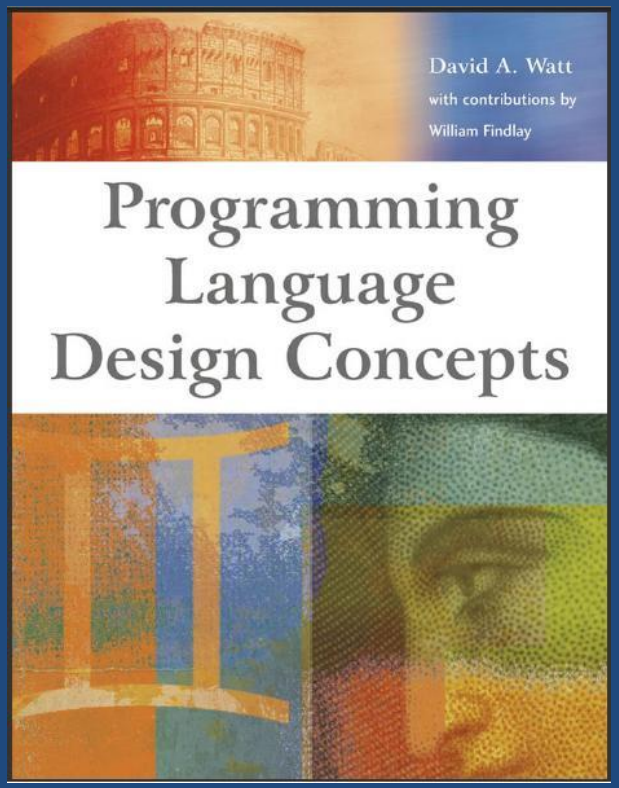
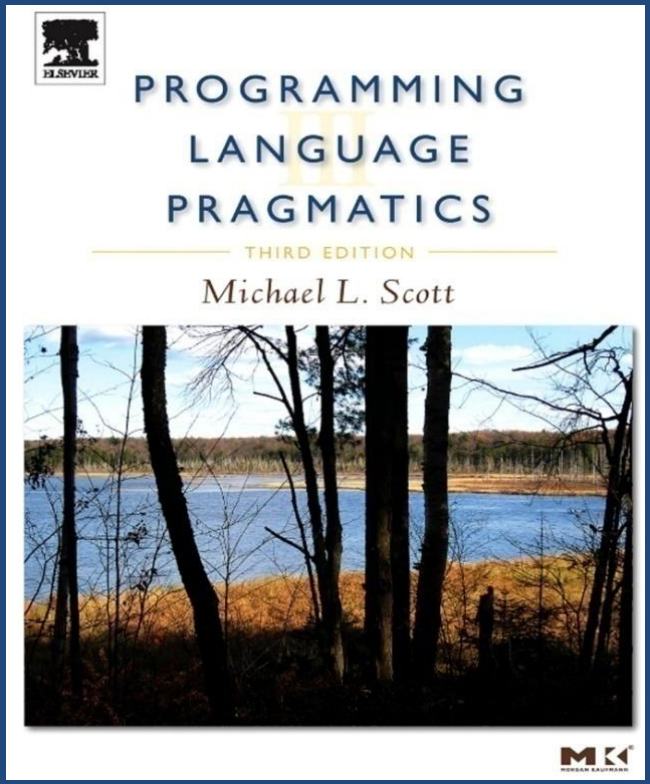
BÖLÜM 6: VERİ TİPLERİ



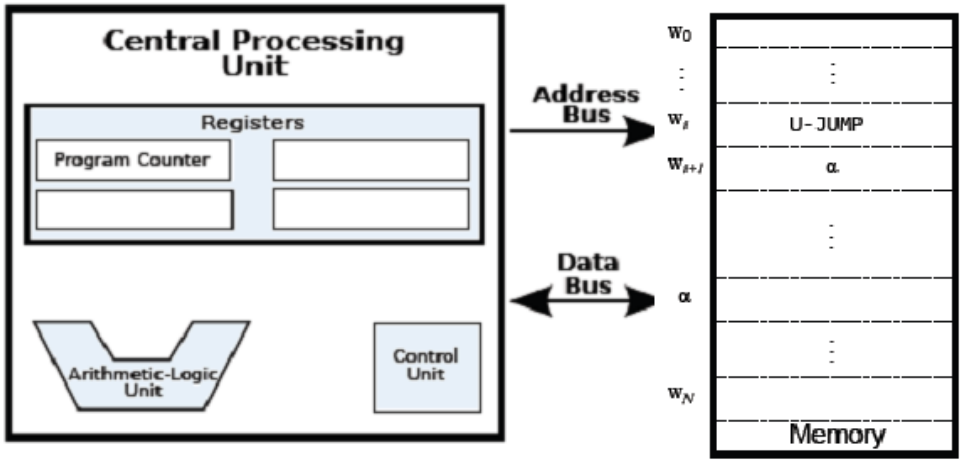




BÖLÜM 6- Konular

- Giriş
- İlkel (Primitive) Veri Tipleri
- Karakter (Character) String Tipleri
- Kullanıcı-tanımlı Sıra (Ordinal) Tipleri
- Dizi (Array) Tipleri
- Kayıt (Tutanak) (Record) Tipleri
- Bileşim (Union) Tipleri
- İşaretçi (Pointer) Tipleri

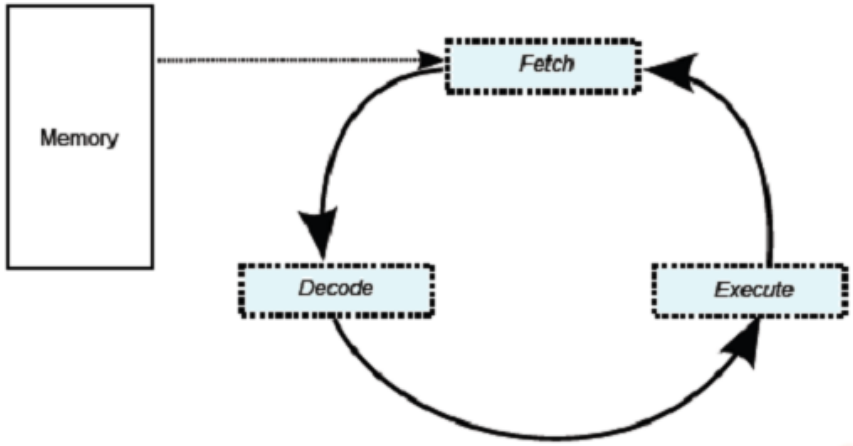
6.1.GİRİŞ



Von Neumann mimarisi

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    alice(%rip), %edx
        movl    bob(%rip), %eax
        imull   %edx, %eax
        movl    %eax, carol(%rip)
        movl    $0, %eax
        leave
        ret
alice:
        .long   123
bob:
        .long   456
```

Çevirici dili (assembly language)
İki ayrı bellek noktasındaki tam sayıları çarpıp sonucu başka bir bellek noktasına koyan program



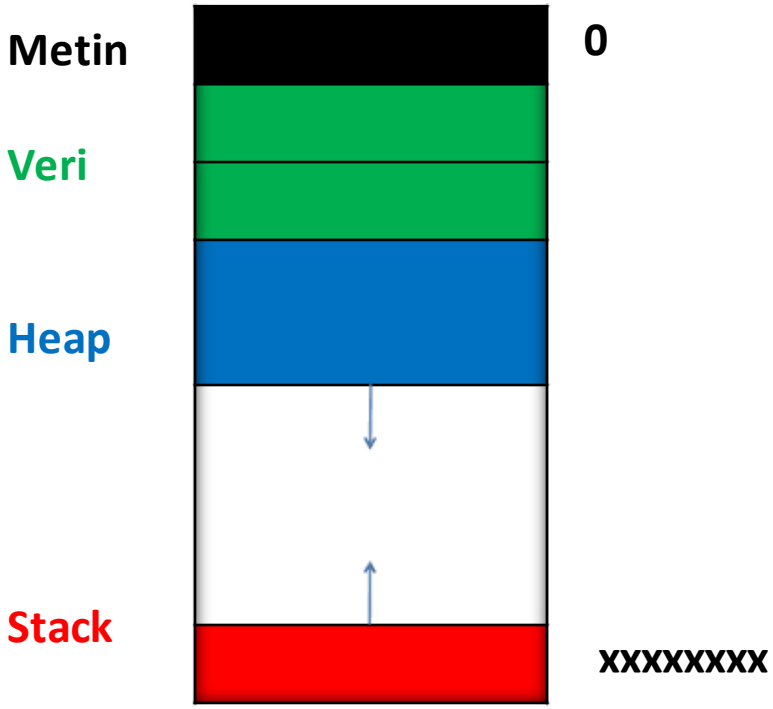
Getir (Fetch), kodu çöz (decode), yap (execute)

```
01010101 01001000 10001001 11100101 10001011 00010101 10110010 00000011
00100000 00000000 10001011 00000101 10110000 00000011 00100000 00000000
00001111 10101111 11000010 10001001 00000101 10111011 00000011 00100000
00000000 10111000 00000000 00000000 00000000 00000000 11001001 11000011
...
11001000 00000001 00000000 00000000  00000000 00000000
```

Yukarıdaki programın ikili sayı sisteminde bellekteki durumu.

Hafıza Düzeni

- **Metin**:
  - Kod
  - sabit veri
- **Veri**:
  - başlatılan global & statik değişkenler
  - global & statik değişkenler
  - 0 başlatılan ya da başlatılmayan (silinmiş, boşluk)
- **Heap**
  - dinamik hafıza
- **Stack**
  - Dinamik
  - yerel değişkenler
  - altprogram bilgisi



6.1.GİRİŞ

- Von Neumann makinesinin yapısının programlama dilleri üzerindeki etkileri önemlidir.
- Von Neumann mimarisi CPU ile belleği belirgin bir şekilde ayırır.
- Bellek içeriği oldukça karışıktır. Bellekte:
  - Her türlü CPU komutları: yazmaç (register) ↔ bellek transferleri, aritmetik işlemler, karşılaştırma, vs.
  - Bazı işlemleri yapabilmek için ek bilgi: transfer etmek için adresler, vs.
  - İşlenecek değerler, adres bilgileri gibi **veriler** (data).
- Belleğe erişim tamamen adrese göredir.
- Von Neumann makinesinde işlenecek bütün bilgiler ikili sayısal sisteme dönüştürülmek zorundadır.

6.1.GİRİŞ

İlkel ve yapısal veri tipleri arasındaki en önemli fark, ilkel veri tiplerinin başka veri tiplerini içermesidir.

- Bir problemin çözümü esnasında bilgisayarda tutulan, CPU komutu olmayan her türlü bilgiye **veri** denir.
- Bir **veri tipi**, bir değerler kümesini ve bu değerler üzerindeki işlemleri tanımlar.
- Bir programlama dilindeki veri tipleri, programlardaki ifade yeteneğini ve programların güvenilirliğini doğrudan etkiledikleri için, bir programlama dilinin değerlendirilmesinde önemli bir yer tutarlar.
- İlkel (temel) ve yapısal veri tipleri arasındaki en önemli fark, ilkel veri tiplerinin başka veri tiplerini içermemesidir.

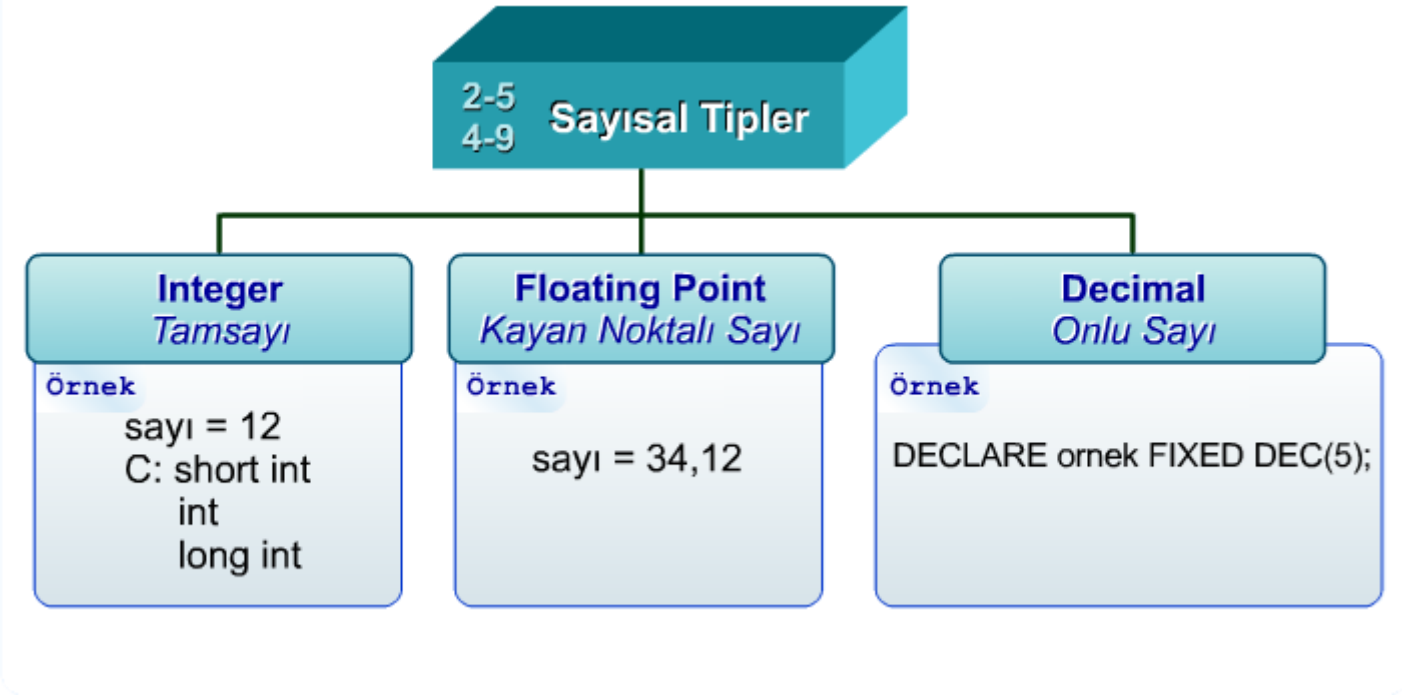
6.2. İLKEL VERİ TİPLERİ

- Başka veri tipleri aracılığıyla tanımlanmayan veri tiplerine **ilkel** (*primitive*) **veri tipleri** denir.

- Önceleri programlama dillerinde sadece sayısal ilkel veri tipleri tanımlanmışken, günümüzde popüler olan programlama dillerinde, karakter, mantıksal, karakter dizgi, kullanıcı tanımlı sıralı tipler gibi çeşitli ilkel veri tipleri bulunmaktadır.

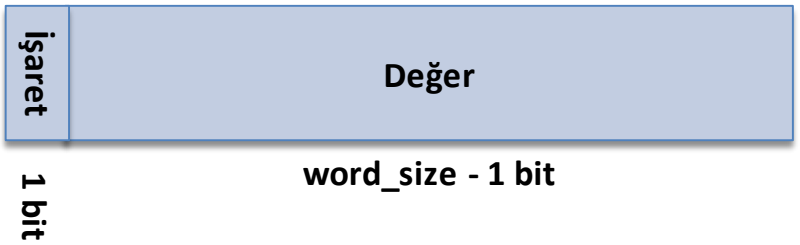
6.2.1. Sayısal Tipler

- İlkel sayısal veri tipleri; tamsayı (integer), kayan noktalı (floating point) ve onlu (decimal) veri tipleridir.



6.2.1. Sayısal Tipler

- **Integer (Tamsayı)**
- En bilinen ilkel veri tipi **tamsayı** (*integer*) dır.
- Bir tamsayı değer, bellekte en sol bit işaret biti olmak üzere bir dizi ikili (*bit*) ile gösterilir.
- Temel tamsayı veri tipine ek olarak Ada, C ve Java programlama dillerinde, (*short int, int, long int* gibi) üç ayrı büyüklükte tamsayı tipi tanımlanmıştır. Ayrıca C, C++, ve C#' ta işaretsiz tamsayı (*unsigned int*) veri tipi de bulunmaktadır.



6.2.1. Sayısal Tipler

- **Floating point (Kayan Noktalı)**
- **Kayan noktalı** (*floating point*) veri tipleri, gerçel sayıları modellerler.
- Kayan noktalı sayılar, kesirler ve üsler olarak iki bölümde ifade edilirler.
- Kayan noktalı tipler, duyarlılık (*precision*) ve alan (*range*) açısından tanımlanırlar. Duyarlılık, değerin kesir bölümünün tamlığıdır. Alan ise, kesirlerin ve üslerin birleşmesidir.
- Aşağıdaki şekilde görüldüğü gibi kayan noktalı veri tipi, **gerçel** (*real*) ve **çift-duyarlılık** (*double-precision*) olmak üzere iki tiple gösterilebilirler.

6.2.1. Sayısal Tipler

NaN : not a number

6.2.1. Sayısal Tipler

- **Decimal (Onlu)**
- **Onlu** (*decimal*) veri tipi, ondalık noktanın sabit bir yerde bulunduğu sabit sayıda onlu basamak içeren bir veri tipidir.
- Ticari uygulamalar için kullanılır (para)
  - COBOL temellidir.
  - C# dili decimal veri tipi sunar.
- Bu veri tipi, az sayıda programlama dilinde (Örneğin; PL/I, Cobol ve C#) tanımlanmıştır.
- Onlu veri tipi, onlu değerleri tam olarak saklayabilirse de, üsler bulunmadığı için gösterilebilecek değer alanı sınırlıdır. Her basamak için bir sekizli (byte) gerekli olması nedeniyle, belleği etkin olarak kullanmaz.

6.2.1. Sayısal Tipler

- **Kompleks**
- Bazı dilleri bu veri tipini destekler, Örneğin: C99, Fortran ve Python
- Her değer iki kısımdan oluşur, birisi reel değer diğer kısım ise imajiner değerdir.
- Python'daki formu aşağıdaki örnekte verilmiştir:

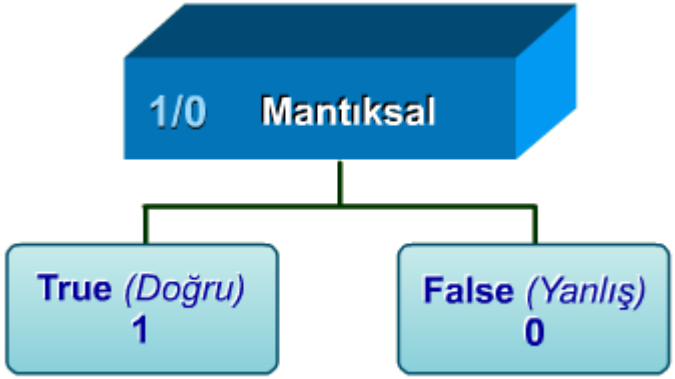
(7 + 3j), 7 sayının reel değeri, 3 ise imajiner değeridir.

6.2.1. Sayısal Tipler

- **Boolean (İkili)**
- En basit veri tipidir.
- Değer aralığında yalnızca iki değer bulunmaktadır. Bunlar true (doğru) ve false (yanlış)'tır.
- Bitler olarak uygulanabilir, fakat çoğu zaman byte kullanılır.
  - Avantaj: Okunabilirlik

6.2.2. Boolean (Mantıksal)

- **Mantıksal** (*boolean*) veri tipi, ilk olarak ALGOL 60 tarafından tanıtılmış ve daha sonra çoğu programlama dilinde yer almıştır.
- Mantıksal veri tipi, sadece *doğru (**true**)* veya *yanlış (**false**)* şeklinde ifade edilen iki değer alabilir. Bir mantıksal değer bellekte bir ikili (bit) ile gösterilebilirse de, çoğu bilgisayarda bellekteki tek bir ikiliye etkin olarak erişim güç olduğu için, bir sekizlide (byte) saklanırlar.



6.2.2. Boolean (Mantıksal)

- İlişkisel işlemciler, mantıksal tipte bir değer döndürdükleri için ve seçimli deyimler gibi programlamadaki birçok yapı, mantıksal tipte bir ifade üzerinde çalıştığı için mantıksal veri tipinin dilde yer almasının önemi büyüktür.

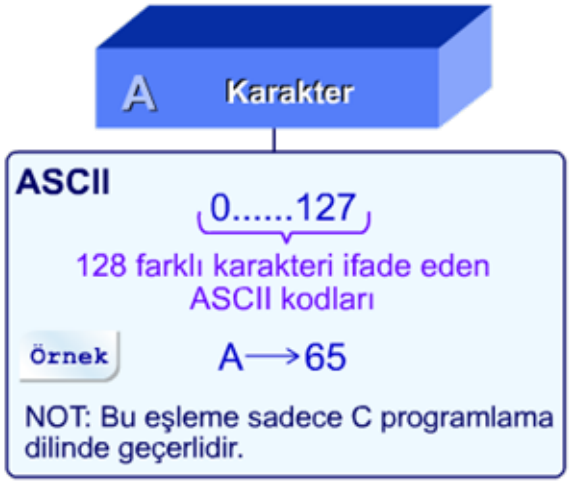
- ALGOL 60'dan sonraki çoğu dilde yer alan mantıksal veri tipinin yer almadığı bir programlama dili C dilidir. C'de ilişkisel işlemciler, ifadenin sonucu doğru ise 1, değilse 0 değeri döndürürler. C'de *if* deyimi, sıfır değeri için yanlış bölümünü, diğer durumlarda ise doğru bölümünü işler.

6.2.3. Character (Karakter)


```
char ab;
int sayı;
ab = 'C' + 3;
sayı = 'H'
ab = sayı;
```

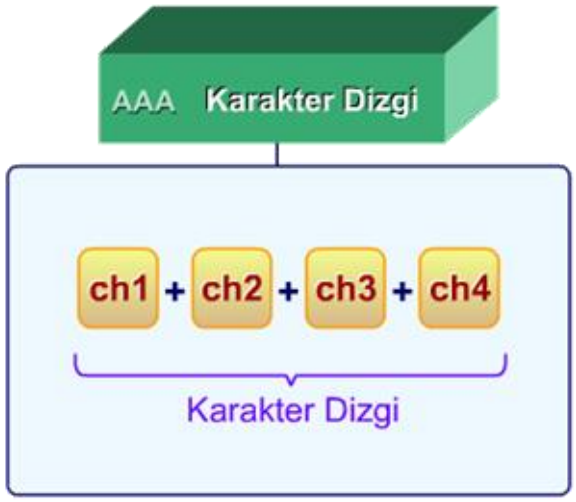
- **Karakter** veri tipi, tek bir karakterlik bilgi saklayabilen ve bilgisayarlarda sayısal kodlamalar olarak saklanan bir veri tipidir.
  - **ASCII Kodlaması:** Karakter veri tipinde en yaygın olarak kullanılan kodlamalardan biri 8 bitlik ASCII kodlamasıdır. ASCII kodlaması, 128 farklı karakteri göstermek için, 0..127 arasındaki tamsayı değerleri kullanır.
  - ASCII kodlamasıyla bağlantılı olarak bazı programlama dilleri, karakter veri tipindeki değerlerle tamsayı tipi arasında ilişki kurarlar. C'de, *char* veri tipi, eş olarak kullanılabilir.

6.2.3. Character (Karakter)



- Yandaki şekilde verilen C deyimleri bu durumu örneklemektedir. Bu örnekte görüldüğü gibi, C'de *char* ve *int* veri tipleri dönüşümlü olarak kullanılabilmektedir.
  - ISO 8859-1 başka bir karakter kodudur ve 256 karakterden oluşur.
  - Daha çok dilin karakter setini göstermek amacıyla daha sonra Unicode (UTF) geliştirilmiştir. Bu kodlamaya bütün diller eklenmiştir. Burada karakterler 1-4 bayt ile gösterilirler. ASCII kodu bu gösterimde tek bayt olarak dahil edilmiştir. Java, JavaScript ve C# bu karakter kodlarını kullanabilmektedir (UCS-2 16 bit; UCS-4 32 bit)

6.2.4. Character String (Karakter Dizgi)



- Bir **karakter dizgi** veri tipinde, nesneler karakterler dizisi olarak bulunur.
- Karakter dizgi veri tipi bazı programlama dillerinde ilkel bir veri tipi olarak, bazılarında ise özel bir karakter dizisi olarak yer almıştır.
- FORTRAN77, FORTRAN90 ve BASIC'te karakter dizgiler ilkel bir veri tipidir ve karakter dizgilerin atanması, karşılaştırılması vb. işlemler için işlemciler sağlanmıştır.
- Pascal, C, C++ ve Ada'da ise karakter dizgi veri tipi, tek karakterlerden oluşan diziler şeklinde saklanır.

6.2.4. Character String (Karakter Dizgi)

- İşlemler:
  - Atama, tanımlama (char *str = “özellikler”;)
  - Karşılaştırma (=, >, strcmp, vs.)
  - Birleştirme
  - Alt dizgiye erişim
  - Örüntü eşleme (Pattern matching)

6.2.4. Character String (Karakter Dizgi)

Örnekler:

- Pascal
  - Temel veri tipi değil; sadece atama ve karşılaştırma
- Ada, FORTRAN 90, ve BASIC
  - Temel veri.
  - Atama, karşılaştırma, birleştirme, alt dizgiye erişim
  - FORTRAN da örüntü eşleme var.
  - Örneğin (Ada)
  - N := N1 & N2 (birleştirme)
  - N(2..4) (alt dizgiye erişim)

6.2.4. Character String (Karakter Dizgi)

- C ve C++
  - Temel veri tipi değil
  - **char** dizilimleri ve kütüphane fonksiyonları kullanılır.
  - örnek: strcpy (src, dst);
  - C++'da string sınıfı kullanmak daha iyi; kontrol var.
- SNOBOL4 (bir dizgi işleme dili)
  - Temel veri tipi
  - Ayrıntılı örüntü eşleme dahil birçok operatör.

6.2.4. Character String (Karakter Dizgi)

- Perl, JavaScript, C++, Java, C#, vs
  - Örüntüler (Patterns) düzenli ifadeler (regular expressions) ile tanımlanır.
  - Çok güçlü bir özellik
  - Örneğin :/[A-Za-z][A-Za-z\d]+/ veya /\d+\.?\d*|\.\d+/
- Java
  - **String** class (karakter dizilimleri değil) statik dizgi nesneleri oluşturur. Nesneler değiştirilemez (kesin).
  - Buna karşılık **StringBuffer** sınıfı değiştirilebilir dizgi nesnelerinin sınıfıdır.

6.2.4. Character String (Karakter Dizgi)

- **Karakter Dizgilerin Uzunlukları**
- Karakter dizgilerin uzunluklarının durağan (statik) mı yoksa dinamik mi olacağı konusunda programlama dilleri farklı yaklaşımlar uygulamıştır.

Örnek

Aşağıdaki gibi bir tanımlama ile C'de son elemanı dizi sonunu gösteren 7 elemanlı bir karakter dizisi tanımlanmıştır.

```
char sehir[] = "Ankara";
```

- **Durağan Uzunluk:** Karakter dizgilerin uzunlukları için, FORTRAN 77, FORTRAN 90, COBOL, Pascal ve Ada'da durağan uzunluk kullanılmıştır ve bu uzunluğun tanımlamada belirtilmesi gereklidir. Örneğin. (FORTRAN 90) CHARACTER (LEN = 15) NAME;

6.2.4. Character String (Karakter Dizgi)

- **Karakter Dizgilerin Uzunlukları**

  - **Değişen Uzunluk (üst sınır var):** PL/I, C ve C++, karakter dizgilerin tanımlanan sabit bir uzunluğa kadar değişen uzunlukta olmasına izin verir. Bu karakter dizgiler için hem en fazla izin verilen, hem de o anki uzunluğu saklamak için çalışma zamanında bir tarifleyici (tanımlayıcı) (run-time descriptor) oluşturulur. Karakter dizgi değişkenlerin bellek bağlaması gerçekleştiği zaman, gerekli bellek yeri atanır ve en fazla uzunluk derleme zamanında sabitlenir.

  - PL/I'da değişken tanımına yapılan VARYING eklemesi ile bu durum belirtilirken, C'de karakter dizgiler, sıfırla gösterilen özel bir karakter (null) ile sona erdirilir. Böylece, tanımlanan en fazla uzunluktan daha az uzunlukta değer saklamak olasıdır.

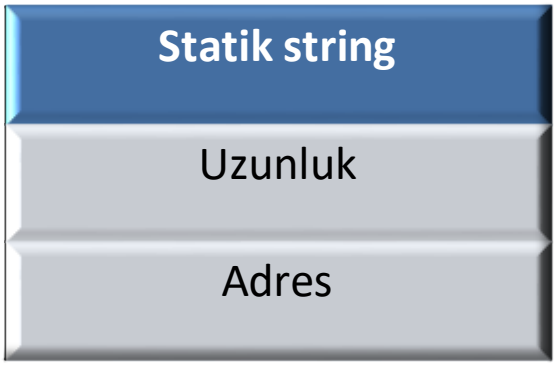
6.2.4. Character String (Karakter Dizgi)

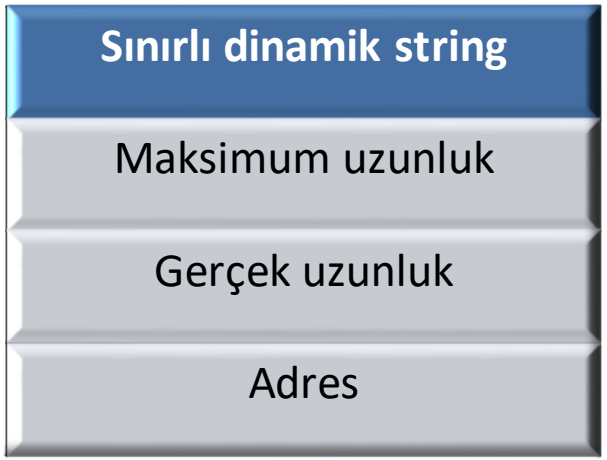
- **Değişen Uzunluk (üst sınır yok):** Karakter dizgiler için üçüncü olasılık, karakter dizgilerin belirli bir üst sınır olmaksızın değişen uzunlukta olabilmesidir. Değiştirilebilir uzunlukta karakter dizgiler oluşturulabildiği için esneklik sağlayan bu teknik, dinamik olarak bellek yönetimi gerektirir. Bu nedenle, değiştirilebilir uzunlukta karakter dizgiler için çalışma zamanında tarifleyici oluşturulur.

Durağan Uzunluk	ch1 + ch2 + ch3 + ch4
	77, FORTRAN 90, COBOL, Pascal, Ada
Değişen Uzunluk (üst sınır var)	ch1 + ch2 + ch3 +.....+ ch n
	PL/I, C, C++
Değişen Uzunluk (üst sınır yok)	ch1 + ch2 + ch3 + ch4 +.....
	Dinamik bellek yönetimi olan diller

SNOBOL4, Perl, JavaScript

Derleme ve Çalışma Zamanı Açıklayıcıları

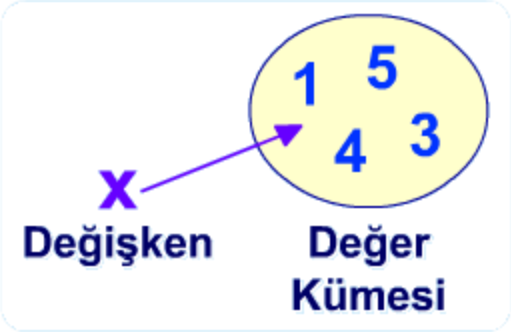




Statik stringler için derleme-zamanı açıklayıcı

Sınırlı dinamik stringler için yürütme-zamanı açıklayıcı

6.2.5. Kullanıcı Tanımlı Sıralı Tipler



- Bir sıralı (ordinal) tip, olası değerlerin pozitif tamsayılar kümesi ile ilişkilendirilebildiği veri tipidir.
- Bir çok programlama dilinde kullanıcılar, **sayılama** (*enumeration*) ve **altalan** (*subrange*) olmak üzere iki tür sıralı tip tanımlayabilir.
- Bu tip tanımlarındaki amaç, programcılara modellenen gerçek dünya nesnelerine karşı gelebilecek yeni tipler oluşturma olanağı sağlamaktır.

6.2.5. Kullanıcı Tanımlı Sıralı Tipler

- **Enumeration (Sayılama) Tipleri**
  - **Sayılama** (*enumeration*) **tipi**, gerçek hayattaki verilerin tamsayı (*integer*) veri tipine eşleştirilmesi için kullanılan veri tipidir.
  - Bir sayılama tip tanımı, parantezler arasında yazılmış belirli sayıdaki isimden oluşur.
  - Sayılama tipi, değişkenleri tamsayılarla ilişkilendirildiği için, bu tipteki değişkenler dizi indisleri olarak, ***for*** döngü değişkenleri olarak kullanılabilir. Benzer şekilde, iki sayılama tipinde değişken veya sabit ilişkisel işlemcilerle, tanımlamadaki sıralarına göre karşılaştırılabilir.

6.2. İLKEL VERİ TİPLERİ

- **Enumeration (Sayılama) Tipleri**

Örnek Pascal'da dört elemanlı, meyve adlı sayılama tipinin tanımı aşağıda görülmektedir.

```
type meyve = (elma, portakal, mandalina, muz)
              1      2         3           4
```

Bu tanımlama ile elma, 1 değeriyle, muz ise 4 değeriyle eşleşmektedir.

Örnek

```
type renkler = (kırmızı, beyaz, mavi, mor)   → renkler adlı sayılama tipi tanımlanmıştır.
var arabarengi : renkler;                    → arabarengi değişkeni renkler tipinde tanımlanmıştır.

arabarengi := beyaz                          → arabarengi := 2 değerini almıştır.
if (arabarengi > kırmızı) ....               → arabarengi değişkeninin değeri ile kırmızı sırasal
                                               öncelik açısından karşılaştırılmıştır. if (2>1)...
```

6.2.5. Kullanıcı Tanımlı Sıralı Tipler

- **Enumeration (Sayılama) Tipleri**
  - Pascal'daki type deyiminin benzeri olarak düşünülebilen C'deki enum yapısı ile yeni bir tip yaratılmaz. Bu yapının kullanımındaki amaç, bir dizi define kullanımı yerine daha anlaşılır bir yaklaşım sunmaktır.
  - İsimlendirilmiş değerlerin hatırlanması kolay olduğu için, sayılama tipi kullanımı, programların okunabilirliğini ve güvenilirliğini artırır.

Örnek

Aşağıdaki C deyiminde, sıcaklığın üç değeri, sıralarıyla birlikte tanımlanmıştır.

```
typedef enum {dusuk, orta, yuksek} sıcaklık;
```

6.2.5. Kullanıcı Tanımlı Sıralı Tipler

- **Enumeration (Sayılama) Tipleri**
  - Pascal – sabitleri tekrar kullanılamaz; Dizilim indeksi, değişken, "case" seçicisi olarak kullanılabilirler. Karşılaştırılabilirler, girdi, çıktı değeri olarak kullanılamaz.
  - Ada – sabitler tekrar kullanılabilir (overloaded literals); kullanıldıkları bağlamda veya tip adından farkları anlaşılır (birinden biri). Pascal'daki gibi kullanılabilir, girdi çıktı değeri olabilirler. Tam sayılara zorlanmazlar.
  - C ve C++ - Pascal gibi; ayrıca tam sayılar olarak girdi ve çıktı değişkeni olabilirler.
  - C#, Java: C++ gibi. Ancak tam sayılara zorlanmazlar.
  - Örnek: enum days {mon, tue, wed, thu, fri, sat, sun};

6.2.5. Kullanıcı Tanımlı Sıralı Tipler

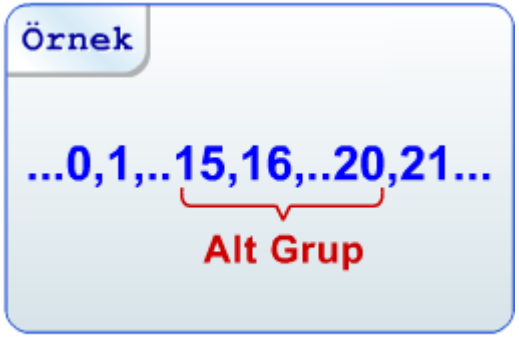
- **Enumeration (Sayılama) Tipleri**
- Tasarım Problemi:
  - Bir sembolik sabitin birden fazla tip tanımlaması içinde yer almasına izin verilmeli midir? Eğer böyle ise o sabitin ortaya çıkmasının tipi nasıl kontrol edilir?
  - Sayım listesi değerleri tamsayıya zorlanır mı?
  - Diğer bir tip bir sayma tipine zorlanır mı?

6.2.5. Kullanıcı Tanımlı Sıralı Tipler

- **Enumeration (Sayılama) Tipleri Değerlendirmesi**
- Okunabilirliğe yardım, örneğin, bir sayı olarak bir renk koduna gerek yoktur.
- Güvenilirliğe yardım, örneğin, derleyici aşağıdakileri kontrol edebilir:
  - İşlemleri (Renk eklenmesine izin vermez)
  - Sayım listesi değişkeninin dışından bir değer atanmasına izin vermez
  - Ada, C# ve Java 5.0; C++'tan daha iyi sayım listesi desteği sağlar. Çünkü bu dillerdeki sayım listesi değişkenleri tamsayı tiplere zorlanmaz.

6.2.5. Kullanıcı Tanımlı Sıralı Tipler

- **Subrange (Altalan) Tipleri**
  - Bir **altalan** (*subrange*) **tipi**, bir sıralı tipin bir alt grubudur
  - Bir altalan tipinin tanımındaki değerler, daha önceden tanımlanmış veya dilde tanımlı olan (*built-in*) sıralı tiplerle ilişkilendirir. Böylece, yeni tanımlanan tip ile alt grubu olduğu ana sınıf arasında bağ kurulur. Altalan tipinin ana sınıfına uygulanabilen tüm işlemciler, altalan tiplerine de uygulanabilmektedir.



15..20, tamsayı tipinin bir alt grubudur.

6.2.5. Kullanıcı Tanımlı Sıralı Tipler

- **Subrange (Altalan) Tipleri**
  - Altalan tipleri ilk olarak Pascal'da tanıtılmıştır.
  - Altalan tipleri değer alanını sınırladıkları için, bir altalan tipindeki bir değişkenin hangi değerleri alabileceği belirlidir. Bu durum, hem programlamların okunabilirliğini hem de güvenilirliğini artırır.
    - `Type Day is (mon, tue, wed, thu, fri, sat, sun);`
    - `subtype Weekdays is Days range Mon..Fri;`
    - `subtype Index is Integer range 1..100;`
    - `Day1 : Days;`
    - `Day2 : Weekdays;`
    - `. . .`
    - `Day2 := Day1;`

Örnek Aşağıda iki altalan tipinin Pascal'da tanımı örneklenmiştir:

```
Type
    Buyukharfler ='A'..'Z';
    Puanlar =50..100;
```

Buyukharfler, tek karakterler için dilde tanımlı olan char tipinin altalanı olarak,

Puanlar ise integer tipinin altalanı olarak tanımlanmıştır.

  - **Ada** – Altalan tipleri yeni tipler değildir. Sınırlandırılmış var olan tiplerdir. Pascal'daki gibi "case" içinde de kullanılabilir.
  - subtype POS_TYPE is INTEGER range 0..INTEGER'LAST;
  - subtype Index is Integer range 1..100;

6.2.5. Kullanıcı Tanımlı Sıralı Tipler

- **Subrange (Altalan) Tipleri Değerlendirmesi**
- Okunabilirliğe yardım
  - Okuyucuların kolayca görebileceği altalan değişkenlerini yalnızca belirli aralıkta saklayabiliriz.
- Güvenilirlik
  - Belirlenen değerler dışında altalan değişkene farklı değerler atamak hata olarak algılanır.

6.3. YAPISAL VERİ TİPLERİ

- **Yapısal** (*structured*) tipler ilkel tiplerden oluşur ve bellekte bir dizi yerleşimde saklanırlar. Diziler, kayıtlar ve göstergeler yapısal veri tiplerini oluşturmaktadır.

6.3.1. Diziler (Arrays)

- **Diziler (Arrays)**
- Bir dizi, homojen veri elemanlarının bir kümesidir, elemanlardan her biri kümedeki birinci elemana göre olan pozisyonuyla tanımlanır.
  - Örnek: C: int aa[4][3][7];
  - sum += aa[i][j][k];

6.3.1.1. Dizi İndisleri

- İndisler, indis elemanları ile dizi elemanlarını birbirlerine eşler.
- **array_name (index_value_list) → an element**
- **İndis sözdizimi (syntax):**
  - FORTRAN, PL/I, Ada ( ) parantez kullanır
  - Birçok diğer diller [ ] kullanır
- Dizi indis tipleri:
  - FORTRAN, C, C#, Java – sadece “integer”
  - Pascal – Her türlü sıralı tip (integer, boolean, char, enum)
  - Ada – integer veya enum (boolean ve char dahil)

6.3.1.1. Dizi İndisleri

- **İndisin Üst Sınırı:**

- İlk programlama dillerinin aksine, günümüzde popüler olan programlama dillerinde dizi indislerinin sayısı sınırlanmaz. Böylece çok boyutlu diziler oluşturulabilir. Ancak, genellikle programlama açısından 3'ten fazla boyutu olan dizilere gereksinim duyulmaz.

- C'de ise diğer programlama dillerinden farklı bir tasarım vardır. C'de dizilerin tek indisi olabilir. Ancak, dizilerin elemanları diziler olabildiği için çok boyutlu diziler oluşturulabilir. Her boyut için ayrı bir köşeli parantez gerektirmesi dışında, C'deki dizilerin diğer dillerdeki dizilerden bir farkı yoktur.

6.3.1.1. Dizi İndisleri

C'de 4,5 büyüklüğünde bir tablo aşağıdaki şekilde tanımlanmaktadır.

int tablo [4][5];

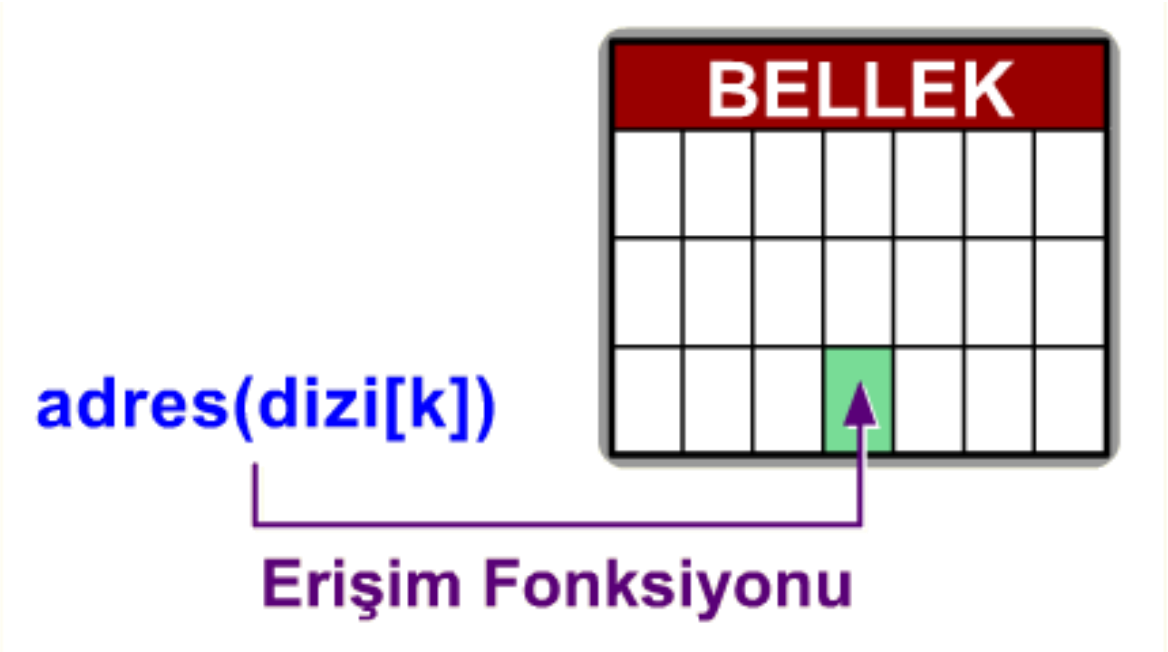
	1	2	3	4	5
1	1,1	1,2	1,3	1,4	1,5
2	2,1	2,2	2,3	2,4	2,5
3	3,1	3,2	3,3	3,4	3,5
4	4,1	4,2	4,3	4,4	4,5

6.3.1.1. Dizi İndisleri

- **İndisin Alt Sınırı:**
- İndisin alt sınırı, bazı dillerde varsayılan değerler alır. Örneğin, C ve C++'da tüm indisler için alt sınır varsayılan olarak sıfır (0), FORTRAN 77 ve FORTRAN 90'da ise varsayılan olarak bir (1) kabul edilir.
- Birçok programlama dilinde ise, indislerin alt sınır değerleri varsayılan olarak tanımlı değildir ve dizinin tanımında belirtilmelidir.

6.3.1.2. Dizi Tiplerinin Gerçekleştirimi

- Bir dizi için **erişim fonksiyonu**, dizinin taban adresini ve indis değerlerini, indis değerleriyle belirtilen bellek adreslerine eşleştirmektir.



6.3.1.2. Dizi Tiplerinin Gerçekleştirimi

- Bir dizi elemanının adresi iki bölümde hesaplanabilir.
- İlk bölüm, dizi tanımlanır tanımlanmaz hesaplanması mümkün olan bölüm, ikinci bölüm ise dizi indisinin değerine bağlı olduğu için çalışma zamanında hesaplanması gereken bölümdür.

Örnek

Tek boyutlu bir dizi için erişim fonksiyonu, alt indis sınırı 1 kabul edilerek, aşağıda belirtilmiştir:

```
adres(dizi[k]) = adres(dizi[1]) + (k-1) * eleman_uz
```

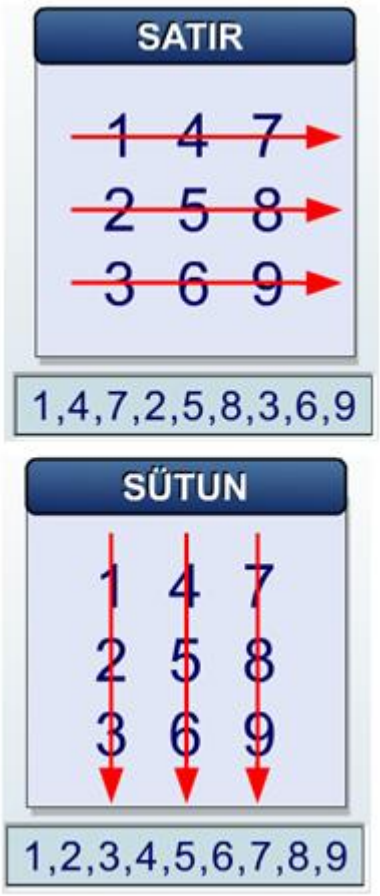
```
adres(dizi[k]) = adres(dizi[1]) + k * eleman_uz - eleman_uz
```

```
adres(dizi[k])=(adres(dizi[1])-eleman_uz)+(k* eleman_uz)
```

sabit **değişken**

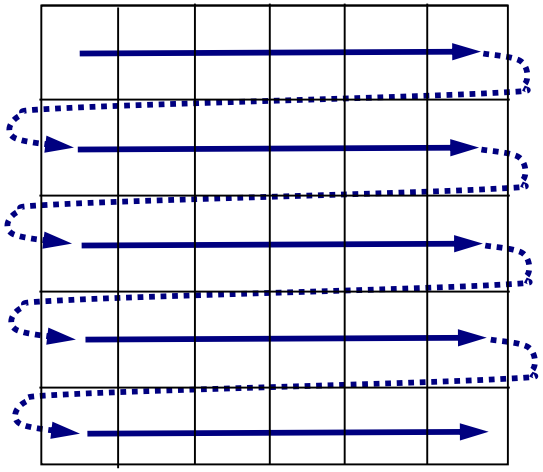
Sadeleştirme sonucu elde edilen formülde ilk bölüm sabit olurken, ikinci bölüm değişken olmaktadır. Eğer eleman tipi ve dizi için bellek bağlaması durağan olarak gerçekleşiyorsa, sabit bölüm, çalışma zamanından önce hesaplanabilir.

6.3.1.3. Dizilerin Belleğe Eşleştirilmesi

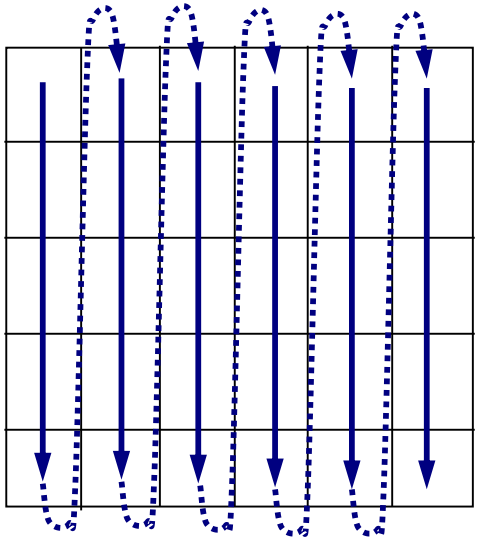


- Bellek donanımı, doğrusal olan bir dizi sekizliden (byte) oluştuğu için, iki veya daha çok boyutlu diziler, tek boyutlu belleğe eşleştirilmelidir.
- Bu durum, çok boyutlu dizilerin gerçekleştirimini tek boyutlu dizilere göre daha karmaşıklaştırır. Çok boyutlu dizilerin tek boyuta eşleştirilmesi *satır tabanlı sıra* ve *sütun tabanlı sıra* olmak üzere iki şekilde yapılabilir.
- Satır tabanlı sırada, eğer dizi bir matris ise, satırlara göre saklanır.

Satır tabanlı sıra

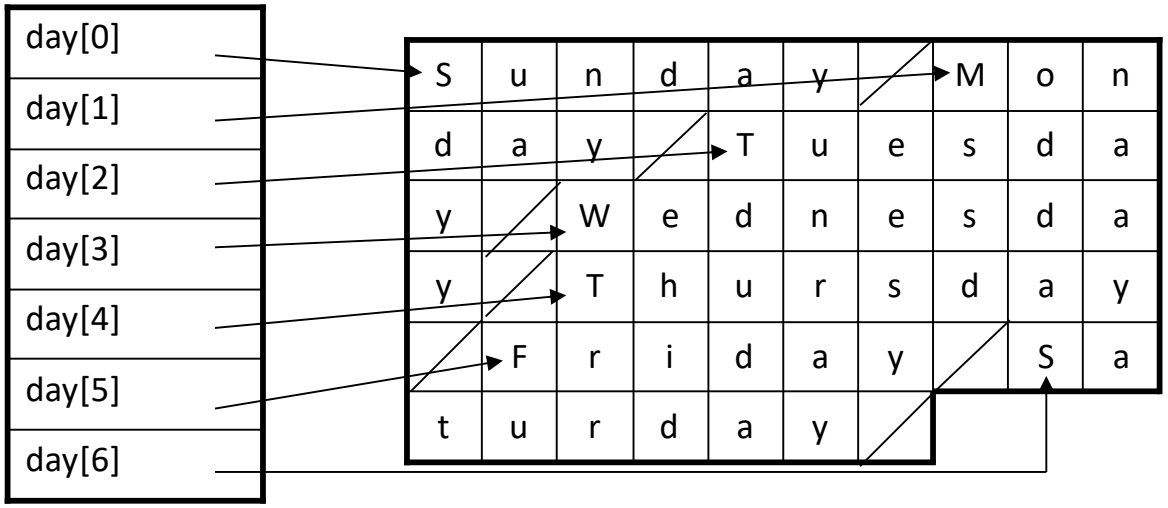


Sütun tabanlı sıra



Satır göstergeleri – Bitişik yerleştirme

- Satır göstergeleri – bir diziye göstergeler dizisi. Ayrılan hafıza dışında yeni bir boyut oluşturur.
- Hafızada boşlukları engeller.



Dizi = 57 bayt

Göstergeler = 28 bayt

Toplam Alan = 85 bayt

Satır göstergeleri – Bitişik yerleştirme

- Bitişik yerleştirme- her elemanın ayrılan yerde bir satıra sahip olduğu dizi.

- Gerçek bir çok-boyutlu dizidir.
  - Aynı zamanda düzensiz dizidir

S	u	n	d	a	y				
M	o	n	d	a	y				
T	u	e	s	d	a	y			
W	e	d	n	e	s	d	a	y	
T	h	u	r	s	d	a	y		
F	r	i	d	a	y				
S	a	t	u	r	d	a	y		

Dizi = 70 bayt

6.3.1.3. Dizilerin Belleğe Eşleştirilmesi

- Satır tabanlı sıraya göre saklanan iki boyutlu diziler için erişim fonksiyonu, taban adresine, bir elemanının uzunluğunun (kaç byte) erişilmek istenen elemandan önceki eleman sayısı ile çarpımının eklenmesi ile bulunur. İki boyutlu (*i* sıra ve *j* sütunu olan) *tablo* isimli bir dizi için erişim fonksiyonu aşağıdaki gibi olmaktadır.

adres(tablo[i,j])=(tablo[1,1]in adresi+((((i-1)*n)+(j-1))* eleman_uz)

- Burada *n*, bir satırda bulunan eleman sayısını göstermektedir.
- Yukarıdaki fonksiyon, her boyut için bir toplama ve bir çarpım komutu ekleyerek, ikiden fazla boyutlu diziler için genelleştirilebilir.

6.3.1.4. İndis bağlama ve dizi kategorileri-

- İndis bağlanmaları ve bellekteki bağlanmalarına göre dizi kategorileri

- **1. Statik (Durağan):** İndis sınırları ve dizinin bellekteki bağlanmaları çalışma zamanında önce hesaplanır, statiktir.
  - FORTRAN 77, Ada'da bazı diziler. C'de statik diziler.
  - Avantaj: Yürütme verimi (bellekten yer alma, geri verme yok)

- **2. Sabit Yığın Dinamik (Fixed Stack Dynamic):** İndis sınırlarının bağlanmaları statik fakat dizinin belleğe bağlanması çalışma zamanında gerçekleşir (yer ayırma işlemi tanımlama zamanında yapılır).
  - Örneğin "statik" olmayan çoğu Java, C yerel değişkenleri
  - Avantaj: bellek verimi

6.3.1.4. İndis bağlama ve dizi kategorileri

- **3. Yığın Dinamik (Stack-dynamic):**
- İndis sınırları ve dizinin bellekteki bağlanmaları dinamiktir fakat değişken ömrüne göre statiktir.
  - Örnek: Ada declare blokları:
    - declare
    - STUFF : array (1..N) of FLOAT;
    - begin
    - ...
    - end;
  - Avantaj: Esneklik – Dizi kullanılmaya başlanmadan önce boyutu bilinmek zorunda değildir

6.3.1.4. İndis bağlama ve dizi kategorileri-

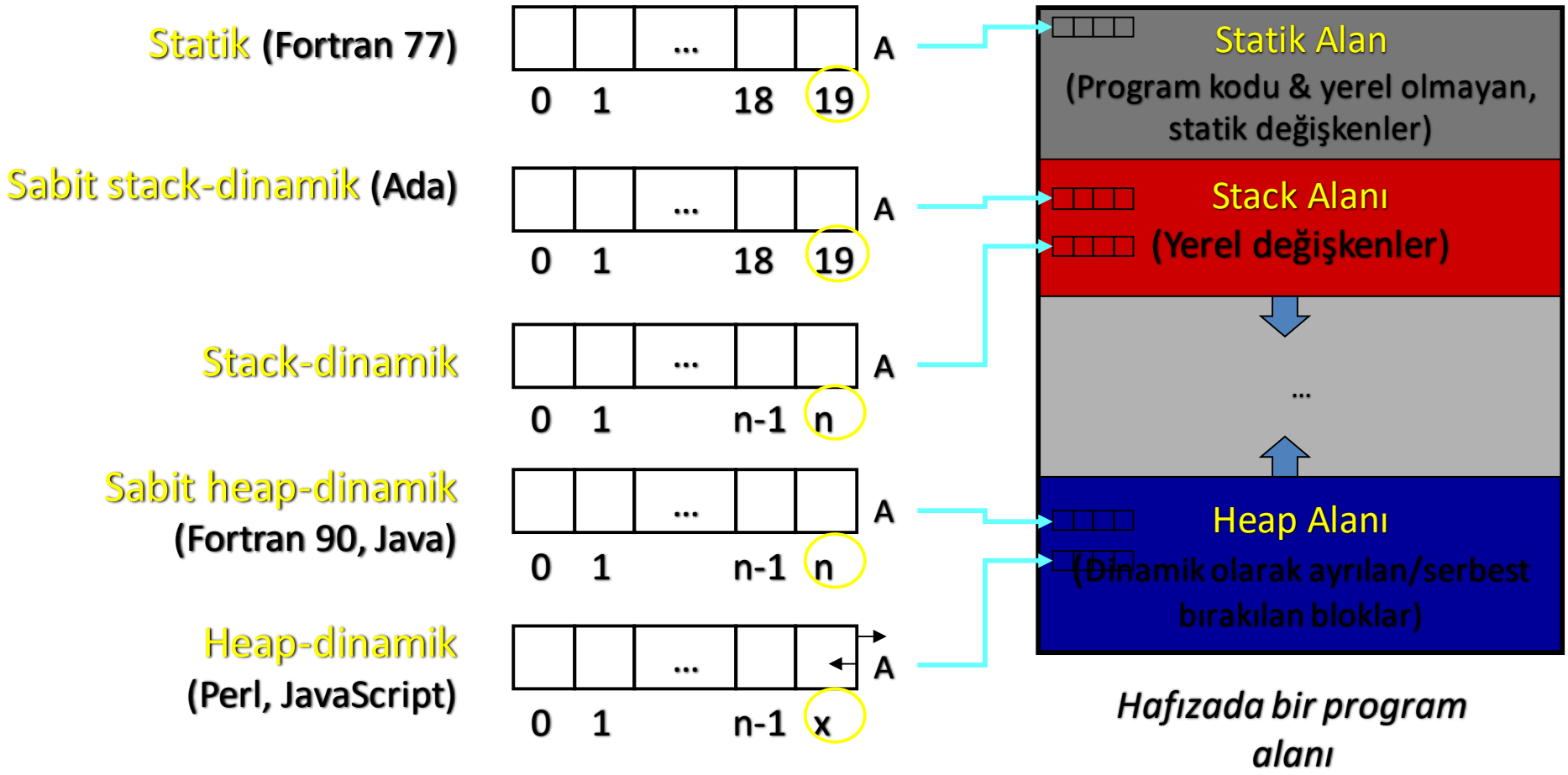
- **4. Sabit Yığın Dinamik (Fixed Heap-dynamic):**
- İndis ve bellek bağlanmaları dinamik (yer tahsisi heapten) olur, fakat bir kez belirlendikten sonra sabit. Bellek geri verilebilir.
  - FORTRAN 90
    - INTEGER, ALLOCATABLE, ARRAY (:,:) :: MAT
      (MAT'ı dinamik 2-boyutlu dizilim olarak tanımlar)
    - ALLOCATE (MAT (10,NUMBER_OF_COLS))
      (MAT'a 10 satır ve NUMBER_OF_COLS sütun ayırır)
    - DEALLOCATE MAT (MAT için ayrılmış belleği iade eder)
  - C, C++: malloc, free ...
  - C++: new, delete.
  - Java'da bütün dizilimler sabit yığın dinamiktir. C# da sabit yığın dinamiği destekler.
  - Avantaj: Esneklik – Dizi kullanılmaya başlanmadan önce boyutu bilinmek zorunda değildir

6.3.1.4. İndis bağlama ve dizi kategorileri

- **5. Yığın Dinamik (Heap-dynamic):** İndis aralığı ve yer ayırma (bellek bağlama) dinamiktir ve istenilen zamanda değiştirilebilir.
  - Avantaj: esneklik (diziler program çalışması sırasında büyüyebilir veya küçülebilir)
  - APL'de, Perl ve JavaScript, diziler (Arrays) ihtiyaca göre büyüyüp küçülebilir.
    - Perl örneği: @list = (1 , 3, 7, 10);
    - push(@list, 13 , 17); // -> (1 , 3, 7, 10 13 , 17)
    - @list = (); // belleği boşaltır ve iade eder.
  - Java'da, bütün diziler birer nesnedir (heap-dynamic)
  - C# ArrayList class, yığın dinamik dizilimleri sağlar. Bu class'ın nesneleri elemansız oluşturulur, sonra dinamik olarak elemanlar eklenir:
    - ArrayList intList = new ArrayList();
    - intList.Add(nextone);

Dizi Kategorileri: Özet

indis bağlama ve *bellek bağlama*'ya bağlı



6.3.1.5. Dizi Başlatma

- Dizi başlangıcı, genellikle dizi elemanlarının bellekte saklandığı sıra ile sıralanmış olarak konulan değerlerin listesidir.
- Bazı diller dizi başlangıcına izin verir.
  - C, C++, Java, C#
    - int list [] = {4, 5, 7, 83}
  - C ve C++ da karakter stringleri
    - char name [] = “freddie”;
  - C ve C++da pointerlar ile string dizileri
    - char *names [] = {“Bob”, “Jake”, “Joe”];
  - Java String nesneleri
    - String[] names = {“Bob”, “Jake”, “Joe”};

6.3.1.5. Dizi Başlatma

- FORTRAN - DATA deyimini kullanır veya değerleri tanımlama sırasında / ... / içine koyar.
  - Integer, Dimension (3) :: List = (/0,5,5/)
- Ada – değerler için pozisyonlar belirlenebilir
  - SCORE : array (1..14, 1..2) :=
  - (1 => (24, 10), 2 => (10, 7),
  - 3 =>(12, 30), others => (0, 0));
  - List: array(1..5) of Integer := (1, 3, 5, 77)
- Pascal dizi başlatmaya izin vermez

6.3.1.6. Heterojen Dizi

- Heterojen dizi, elemanları aynı tipten olması gerekmeyen dizilerdir.
- Perl, Python, JavaScript ve Ruby tarafından desteklenir.
- Diğer dillerde heterojen diziler yerine struct (yapılar) kullanılır, fakat yapılar heterojen diziyi tam manasıyla karşılayamazlar.

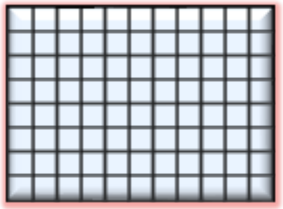
6.3.1.7. Dizi İşlemleri

- APL vektörler ve matrisler için hem en güçlü dizi işleme işlemleri hem de tekli operatör desteği (örneğin, kolon elemanları tersine çevirmek için) sağlar
- Ada yalnızca dizilerde atama, birleştirme ve ilişkisel operatör işlemlerine izin verir.
- Pyton'nun dizi atamaları yalnızca referans değişikliği işlemlerini yapmasına rağmen eleman üyelik sistemiyle Ada'nın sağladığı tüm işlemleri yapabilir.
- Ruby dizilerde atama, birleştirme ve ilişkisel operatör işlemlerine izin verir
- Fortran iki dizi arasındaki eleman işlemlerini destekler
  - Örneğin Fortrandaki + operatörü iki dizi çiftleri arasındaki elemanları toplar.

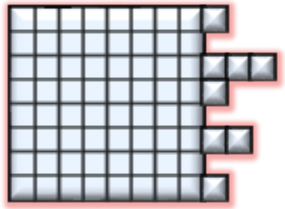
6.3.1.8. Dikdörtgen(Düzenli, Rectangular) ve Tırtıklı (Düzensiz, Jagged) Diziler

- Bir dikdörtgen dizi satırları tüm unsurlarının aynı sayıda ve tüm sütun elemanlarının aynı sayıya sahip olduğu çok boyutlu dizidir.
- Tırtıklı matrisler ise her satırında aynı sayıda eleman bulunmayan dizilerdir.
  - Olası çoklu-boyutlu zaman dizileri aslında dizinler olarak görünür
  - C, C++, ve Java tırtıklı (düzensiz) dizileri destekler
- Fortran, Ada, ve C# dikdörtgen dizileri destekler (C# aynı zamanda tırtıklı dizileri de destekler)

Rectangular Dizi

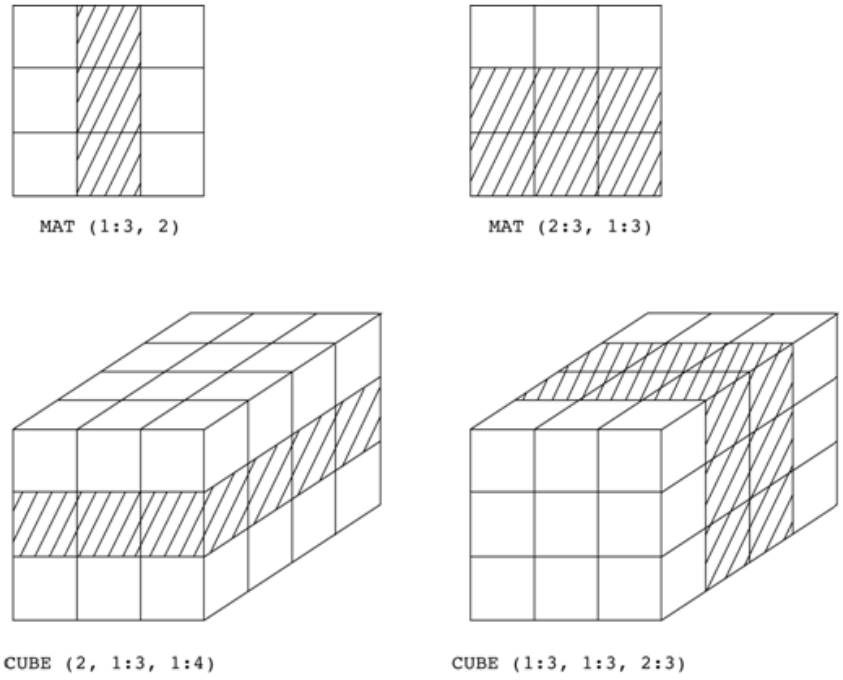


Jagged Dizi



6.3.1.9. Kesitler

- Kesitler (slices)
  - Kesit (slice), bir dizininin bir kısım altyapısıdır (substructure) ; bir referanslama mekanizmasından fazla bir şey değildir
  - Kesitler (slices) sadece dizi işlemleri olan diller için kullanışlıdır.



MAT = CUBE(:,:,2) // cube dizininin ikinci dilimini mat'a koyar.

6.3.1.9.1. Kesit Örnekleri

- Python

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`vector (3:6)` dizinin 3 elemanını temsil eder.

`mat[0][0:2]` dizinin ilk satırındaki ilk elemandan 3. elemana kadar olanı gösterir.

- Ruby slice metot olarak kesiti destekler.

`list.slice(2, 2)` örneğinde 2. elemandan 4. elemana kadar olanları kapsar.

6.3.1.10.Demet (Tuple) Tipi

- Demet veri tipi kayıt veri tipine benzeyen bir veri tipidir.
- Python, ML, F#, C# (.Net 4.0 ile birlikte)'ta kullanılır.Fonksiyonlara birden fazla değer döndürür.
  - Python
    - Listelerle yakından ilişkili ama değiştirilemez
    - Demet oluşturma

```
myTuple = (3, 5.8, 'apple')
```

İndislerini 1'den başlayarak referanslandırır.

+ operatörünü kullanır ve del komutuyla silinir.

6.3.1.10.Demet (Tuple) Tipi

- ML

```
val myTuple = (3, 5.8, 'apple');
```

- Takipçilere erişim:

`#1(myTuple)` demetin ilk elemanı

- Yeni bir demet aşağıdaki gibi tanımlanır.

```
type intReal = int * real;
```

- F#

```
let tup = (3, 5, 7)
 let a, b, c = tup
```

6.3.1.11. Liste Tipleri

- LISP ve Şema listeleri parantez ayracıyla kullanılırlar ve elemanlar arasına virgül konulmaz.

  `(A B C D)` ve `(A (B C) D)`

- Veri ve kod aynı formdadır.

  Veri, `(A B C)`

  Kod, `(A B C)` bir fonksiyonun parametreleri

- Yorumlayıcı hangi listeye ihtiyaç duyacağını bilmelidir. Buradaki karmaşıklığı ortadan kaldırmak için veri listelerinin önüne ‘ işareti konur.

  `'(A B C)` veridir

6.3.1.11. Liste Tipleri

- Şema içerisindeki Liste operatörleri
  - `CAR` listesi ilk elemanını döndürürse

    `(CAR '(A B C))` returns `A`
  - `CDR` ilk elemanı söküldükten sonra kendi listesinde parametresi kalanı verir.

    `(CDR '(A B C))` returns `(B C)`

  - `CONS` Yeni bir liste yapmak için ikinci parametre, bir liste içine ilk parametre koyar.

    `(CONS 'A (B C))` returns `(A B C)`
  - `LIST` yeni bir liste döndürür.

    `(LIST 'A 'B '(C D))` returns `(A B (C D))`

6.3.1.11. Liste Tipleri

- ML'de Liste Operatörleri
  - Listeler parantez içinde yazılır ve elemanları virgüllerle ayrılır.
  - Liste elemanları aynı veri tipinde olmalıdır.
  - `CONS` fonksiyonu ML dilinin binary operatörüdür, `::`

    `3 :: [5, 7, 9]` dönüşür `[3, 5, 7, 9]`
  - CAR ve CDL fonksiyonları burada hd ve tl olarak adlandırılır.

6.3.1.11. Liste Tipleri

- F# Listeler
  - ML dilindeki liste yapısına benzer, yalnızca elemanların ayrılmasıyla hd ve tl metotları List sınıfının içinde yer alır
- Python Listeler
  - Liste veri tipi genelde python dizileri olarak sunulur
  - Genelde LISP, ML, F# ve Python listeleri birbirine benzer.
  - Listedeki elemanlar değişik veri tiplerinden olabilir
  - Liste oluşturulması aşağıdaki gibidir.

```
myList = [3, 5.8, "grape"]
```

6.3.1.11. Liste Tipleri

- Python Listeler (devamı)
  - Liste indisi “0”dan başlar ve sonradan değiştirilebilir.

    `x = myList[1]` `x’` e `5.8` atar
  - Liste elemanları del komutuyla silinir.

    `del myList[1]`
  - Liste anlamları – küme gösterimiyle temsil edilebilir.

    `[x * x for x in range(6) if x % 3 == 0]`

    `range(12)` creates `[0, 1, 2, 3, 4, 5, 6]`

    Constructed list: `[0, 9, 36]`

6.3.1.11. Liste Tipleri

- Haskell'in Liste Anlamları
  - Orjinal

```
[n * n | n <- [1..10]]
```

- F#'in Liste Anlamları

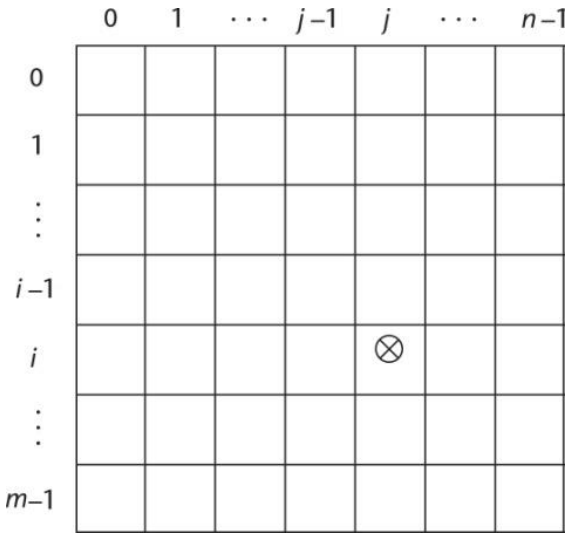
```
let myArray = [|for i in 1 .. 5 -> [i * i) |]
```

- C# ve Java dilleri de listeleri destekler. Kendi dinamik koleksiyonlarında List ve ArrayList adında sınıfları vardır.

6.3.1.12. Dizi Uygulamaları

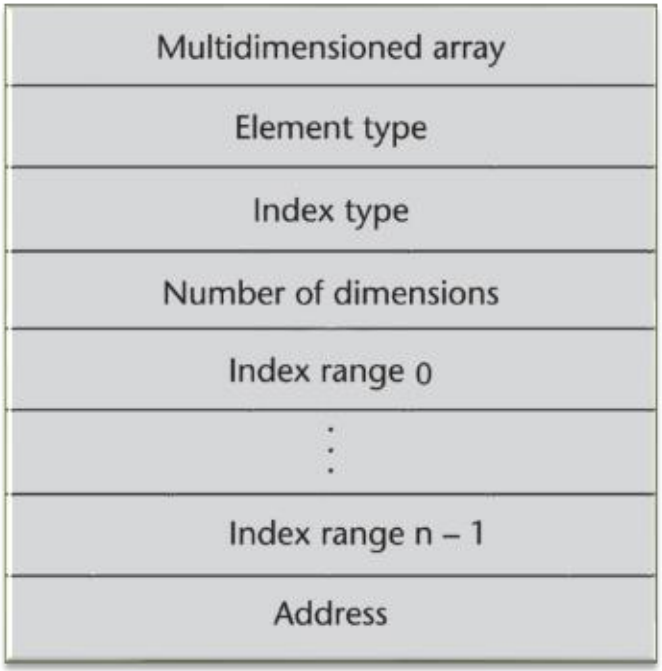
- Erişim fonksiyonları (Access function) altsimge (subscript) ifadelerini dizideki bir adrese eşler (map)
- Tek Boyutlu Dizilere Erişim fonksiyonu:

address(list[k]) = address (list[lower_bound])
+ ((k-lower_bound) * element_size)



6.3.1.12. Dizi Uygulamaları

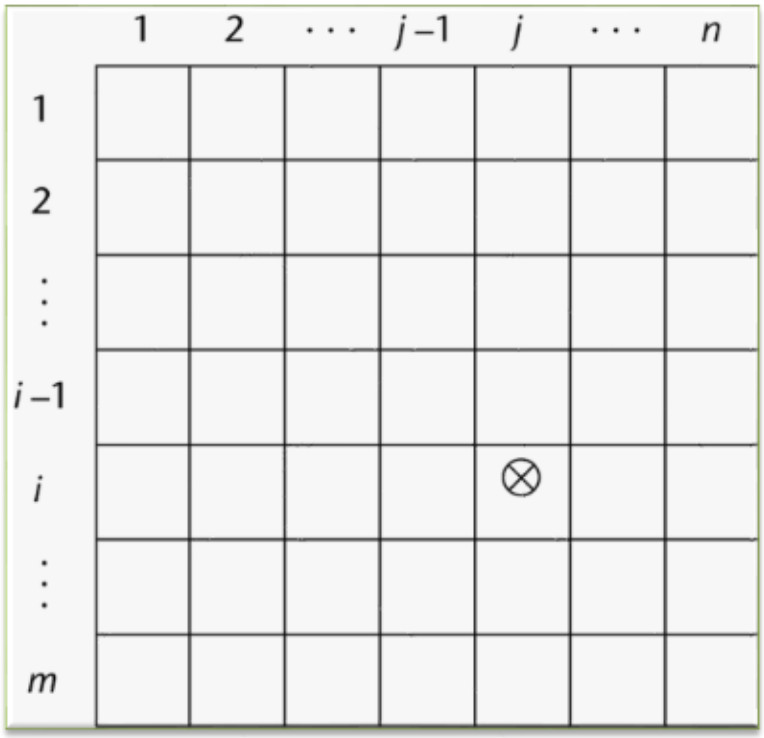
- Çok Boyutlu Dizilere Erişim için yaygın olarak kullanılan metotlar:
  - Satıra göre sıralama – Bir çok dilde kullanılan metottur
  - Sütuna göre sıralama – Fortran tarafından kullanılır
  - Çok boyutlu dizilerin derleme süreleri yan taraftaki şekilde verilmiştir.



6.3.1.12. Dizi Uygulamaları

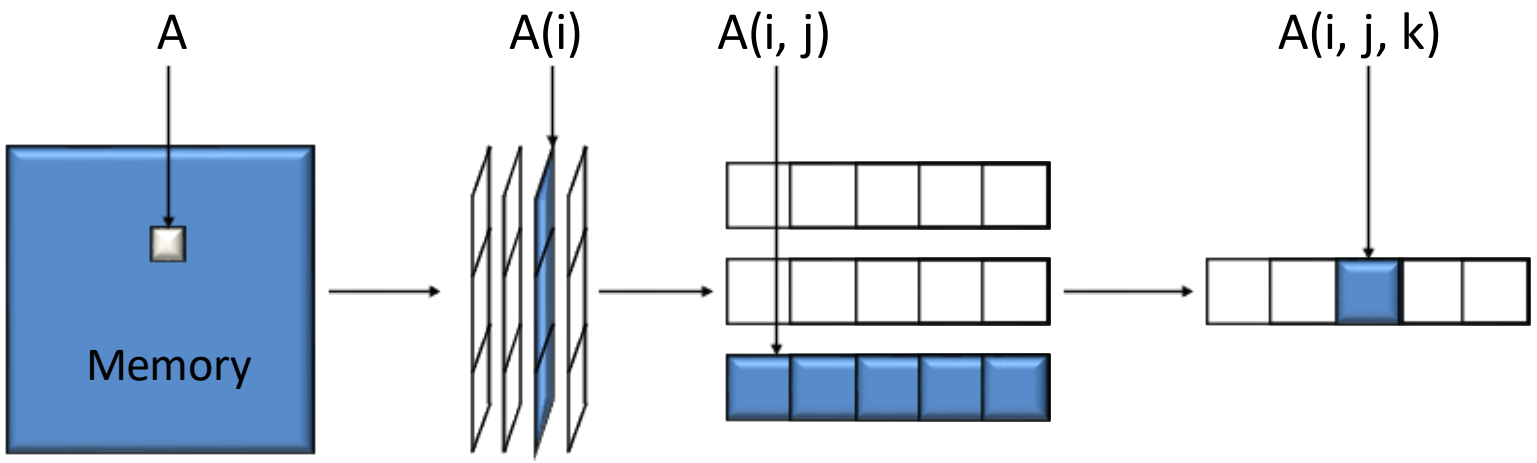
- **Çok Boyutlu Dizilerde Eleman Yerleştirme**
- Genel format

  Location (a[I,j]) = address of a [row_lb,col_lb] + (((I – row_lb) * n) + (j – col_lb)) * element_size



- 3-boyutlu A(i, j, k) dizisinin adresi:

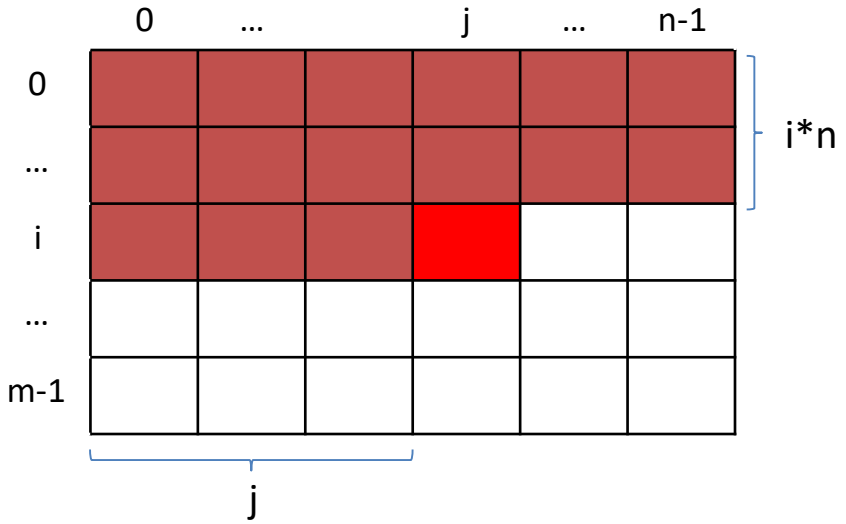
A'nın adresi
+ ((i - $L_{düzlem}$) * düzlem boyutu)
+ ((j - $L_{satır}$) * satır boyutu)
+ ((k - L_{eleman}) * eleman boyutu)



Özet: Dizilere Erişim

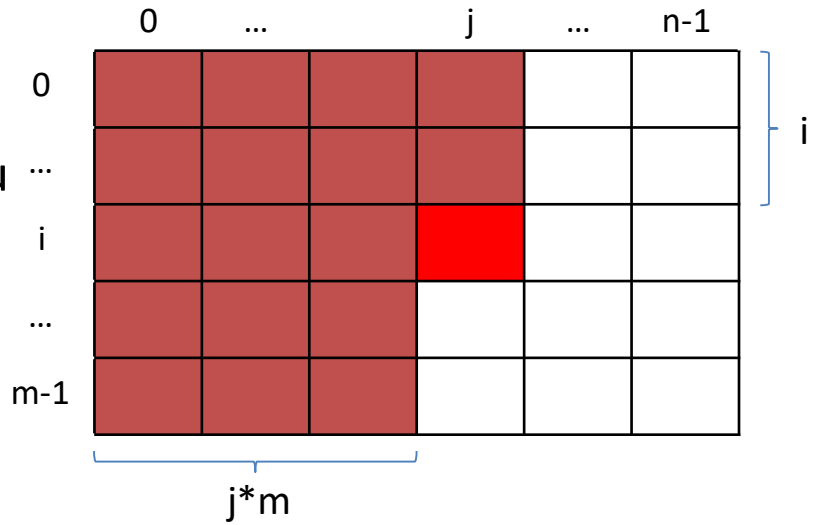
Satır öncelikli:

$A_{i,j} = A_{0,0}+(i *n+j)*eleman\_boyutu$



Sütun öncelikli:

$A_{i,j} = A_{0,0}+(j*m+i)*eleman\_boyutu$



6.3.1.12. Dizi Uygulamaları

Bir elemanın yerini bulmak

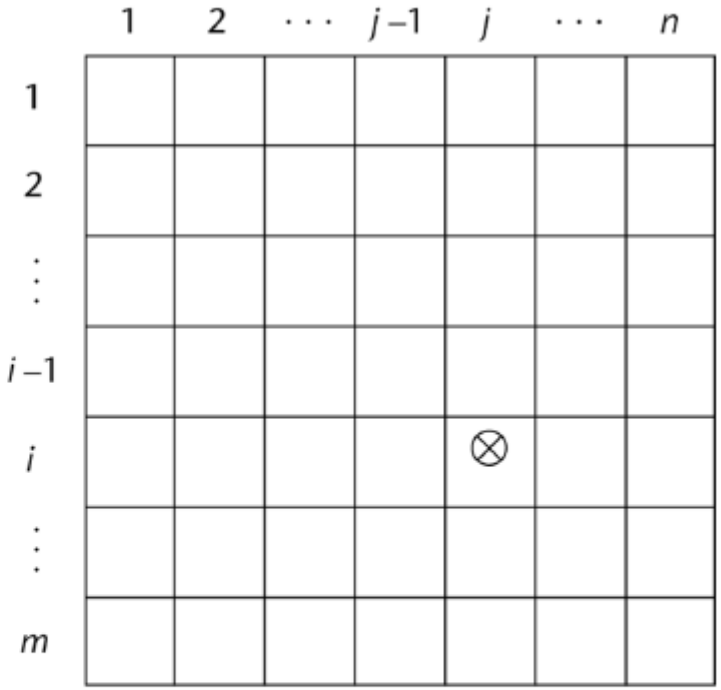
Eğer altsimgeler 1'den başlıyorsa ve satır öncelikli yerleştirme varsa:
Adres(eleman(i,j)) = Adres(eleman(1,1)) + ((i -1)*n + (j-1))*eleman_boyu

Not 1: Satır önceliklide döngüleri sütun üzerinden, sütun önceliklide satır üzerinden yapmak bellek erişimini hızlandırır.

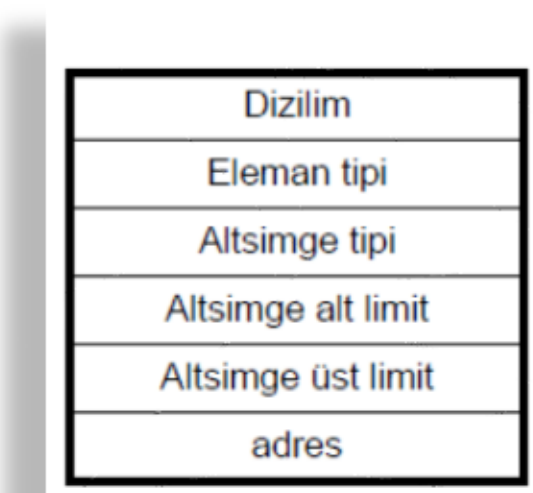
Not 2: :Çok boyutlu dizilimlerde işaretçi kullanımına dikkat.

Örnek:

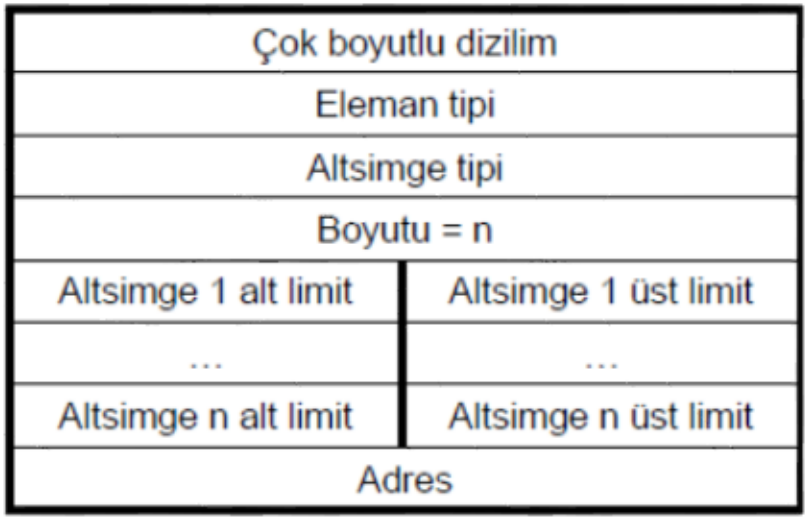
```
int * ip;
int i;
int aa[100][150];
ip = &aa;
for(i = 0; i < 100*150; i++)
    *ip++ = i;
```



6.3.1.13. Derleme Süresi Betimleyiciler



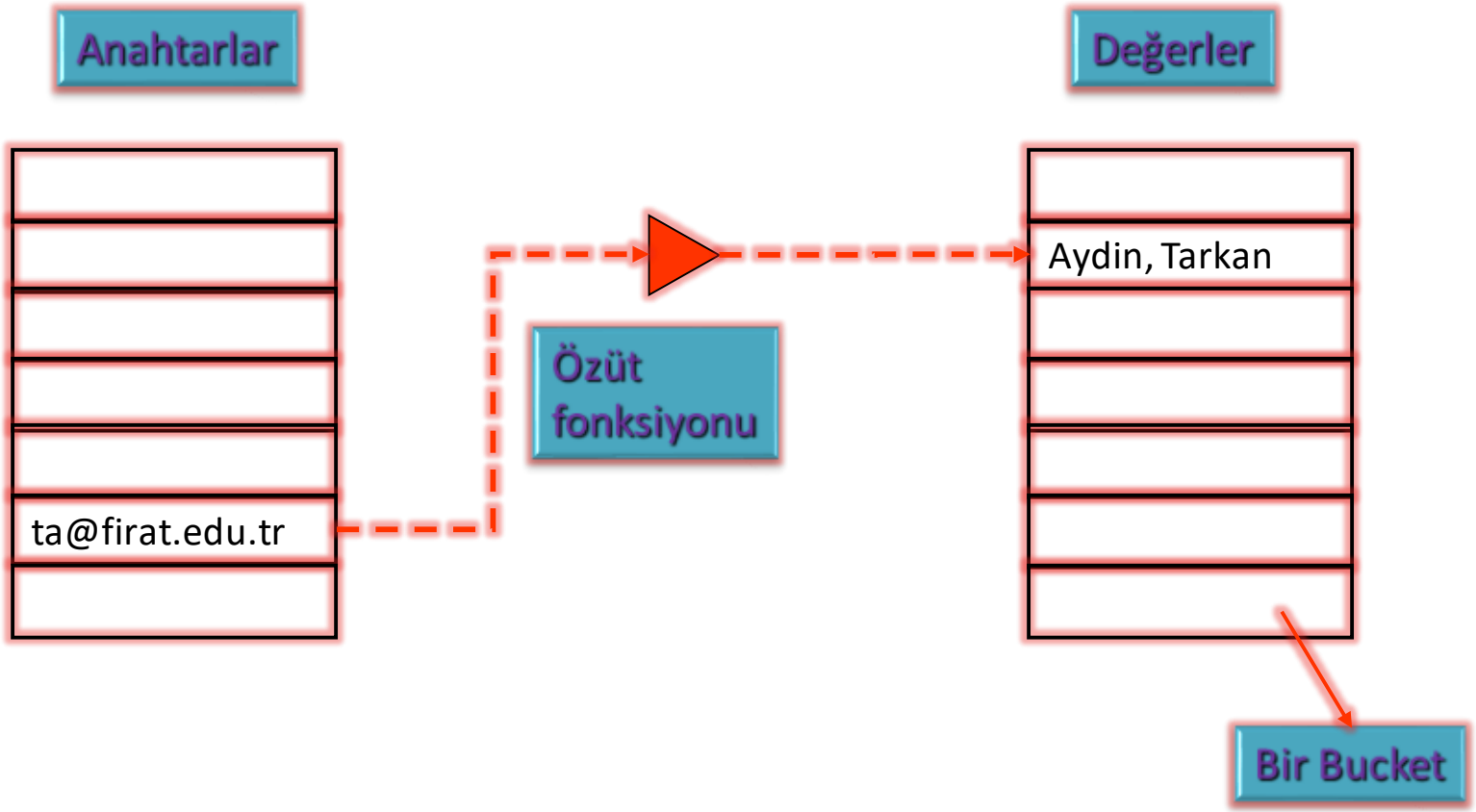
Tek Boyutlu Dizi (Dizilim)



Çok Boyutlu Dizi (Dizilim)

6.3.1.14. İlişkili Diziler

- Bir ilişkili dizi (associative array), kendisiyle aynı sayıdaki anahtar denilen değerlerle indekslenmiş rassal verilerdir.
- Anahtar değerlerinden bir kıyım (çırpı, özüt) (hash) algoritması ile değerler bulunur.
- Tasarım Problemleri:
  1. Dizi elemanlarına referansın şekli nedir?
  2. Boyut statik midir yoksa dinamik mi?


Anahtarlar
Değerler
ta@firat.edu.tr
Özüt fonksiyonu
Aydin, Tarkan
Bir Bucket

6.3.1.14. İlişkili Diziler

Perl'de İlişkili Diziler

- Değişkenler % ile başlar. Boyu değişkendir.
- Sabit değerler parantez arasına yazılır

```
%YuksekSicakliklar = ("Pazartesi" => 45, "Salı" => 49,…);
```

- İndisleme küme parantezleri ve anahtar verilerle ($) yapılır.

```
$hi_temps{"Wed"} = 83;
```

- Elemanlar delete komutuyla silinir.

```
delete $hi_temps{"Tue"};
```

- Diziyi tamamen boşaltmak için

```
@YuksekSicakliklar = ();
```

- PHP dizilimleri hem normal hem ilişkili dizilimlerdir.
- Eğer dizilimde arama yapılacaksa ilişkili dizilimler çok verimlidir. Sıralı erişim için normal dizilim kullanılmalıdır.

6.3.2. Record (Kayıt) Tipi

- Bir sınıftaki öğrencilerin hem numara, hem ad-soyad hem de adreslerinin tutulması için 3 ayrı diziye gereksinim vardır.
- Kayıt tipi, programlardaki bu tür gereksinimleri karşılamak için tasarlanmıştır.
- **Kayıt** (*record*) **tipi**, her elemanın ismiyle ayırt edildiği heterojen veri elemanları topluluğudur. İlk olarak COBOL'da tanıtılan kayıt tipi, daha sonra çoğu programlama dilinde yer almıştır.
- Tasarım sorunları:
  - 1. Referansların şekli nasıl olacak?
  - 2. Hangi birim işlemleri tanımlanacak?

6.3.2. Record (Kayıt) Tipi

- Diziler ve kayıtlar arasındaki en temel fark, dizilerde homojen elemanların, kayıtlarda ise heterojen elemanların bulunmasıdır.
- Bunun sonucu olarak, kayıtlardaki sahalara indislerle değil, her sahayı gösteren tanımlayıcılarla ulaşılmaktadır. (örneğin; ogrenci.adsoyad gibi).
- Kayıt sahaları, dizilerde olduğu gibi homojen olmak zorunda olmadıkları için, saha tipleri seçiminde kayıtlar, dizilerden daha fazla esneklik sunarlar.
- Dizi elemanlarına erişim kayıt alanlarına erişimden daha yavaştır, çünkü altsimgeler (subscripts) dinamiktir (alan adları (field names) statiktir).

İç içe yuvalanmış kayıtlar

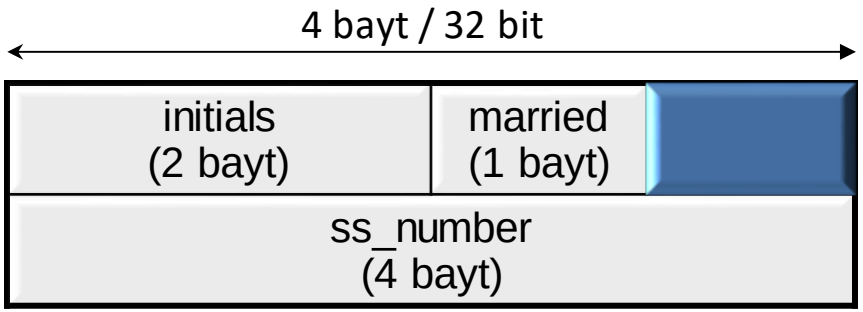
- Çoğu dil kayıtların iç içe yuvalanmasına izin verir
  - Pascal

```
type two_chars = array [1..2] of char;
type married_resident = record
  initials: two_chars;
  ss_number: integer;
  incomes: record
    husband_income: integer;
    wife_income: integer;
  end;
end;
```

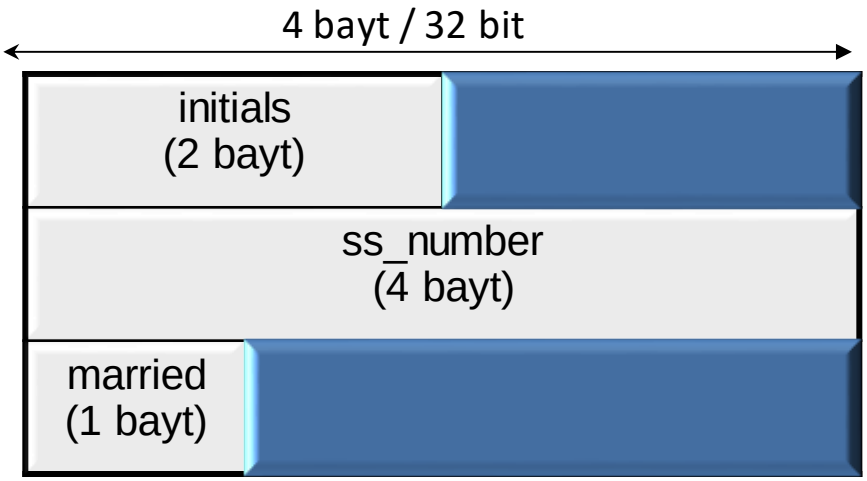
Kayıtların hafıza düzeni

- Alanlar hafızada bitişik olarak tutulur.
- Hafıza kayıtlar için alanların oluşturulma sırasına göre ayrılır.
- Değişkenler kolay referans için sıraya dizilir.

boşluk için optimize edilmiş



hafıza sırası için optimize edilmiş



6.3.2.1. Kayıt Sahalarına Başvuru

- Kayıtlardaki sahalara başvuru, çeşitli şekillerde gösterilebilir. Ortak nokta, erişilmek istenilen sahanın ve bunu içeren kaydın belirtilmesidir.
- COBOL düzey numaraları kullanır, diğerleri özyinelemeli olarak tanımlarlar.
- Aşağıdaki COBOL için verilen örnekte, *OGRENCI_KAYDI, OGRENCI_ADI* kaydından ve *OGRENCI_NO* ve *OGRENCI_ADRES* sahalarından oluşmaktadır. *01,02* ve *05* düzey numaraları olup kayıttaki hiyerarşik yapıyı göstermektedir.

```
01 OGRENCI_KAYDI.
   02 OGRENCI_ADI.
      05 ILK PICTURE IS X(15).
      05 SOYAD PICTURE IS X(15).
   02 OGRENCI_NO PICTURE IS 99999.
   02 OGRENCI_ADRES PICTURE IS X(30).
```

6.3.2.1. Kayıt Sahalarına Başvuru

- Burada kayıtlara erişim aşağıdan yukarıya veya yukarıdan aşağıya şeklinde olabilir.

SOYAD **OF** OGRENCI_ADI **OF** OGRENCI_KAYDI

OGRENCI_KAYDI.OGRENCI_ADI.SOYAD

- Eğer bir kaydın bir sahasına başvuruda, en dıştaki kayıttan başlayarak istenilen sahaya kadar aradaki tüm kayıtların isimleri yer alıyorsa buna **tam başvuru** denir.

6.3.2.1. Kayıt Sahalarına Başvuru

- COBOL gibi PL/I'da da kayıtlar düzey numaraları ile yapılandırılır.
-
```
01 EMP-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID   PIC X(10).
      05 LAST  PIC X(20).
   02 HOURLY-RATE PIC 99V99
```
- ADA dilinde ise type özelliği kullanılır, Pascal a benzerdir. Java'da kayıt özelliği yoktur.
- **ADA**

```
type IsciAdiTipi is record
   ilk: String (1..20);
   orta: String (1..10);
   soy: String (1..20);
end record;
type IsciKaydiTipi is record
   IsciAdi: IscıAdiTipi;
   SaatUcret: Float;
end record;
IsciKaydı: IsciKaydiTipi;
```

6.3.2.2. Pascal ve C' de Kayıt Sahalarına Başvuru

Pascal'da *with* yapısı kullanımı

```
with ogrenci_kaydı do
begin
   ilk := 'Mehmet';
   orta:= 'Ali';
   soyad := 'Tokcan';
end
```

C'de *struct* tanımı

```
typedef  struct{
    char ilk[15];
    char soyad[15];
    int ogr_no;
    char ogr_adres[30];
} ogrenci_kayit;
```

- **Pascal' da Kayıt Sahalarına Başvuru:**
- Pascal, kayıt sahaları için tam başvuru yerine kullanılabilecek bir yöntem sağlar. Bu yöntemde bir kaydın sahalarına başvuru için **with** yapısı kullanılır. Yandaki şekil bu kullanımı, *ilk, orta ve soyad* sahalarını bulunduran *öğrenci_kaydı* isimli kayıt üzerinde örneklemektedir.

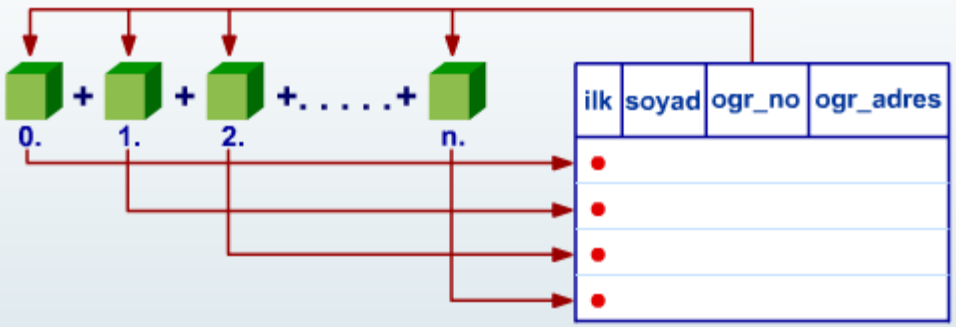
- **C'de Kayıt Sahalarına Başvuru:**
- C'de kayıtlar, *structure* olarak adlandırılır ve Pascal'daki *record* yapısı ile benzer özellikler taşır. Yandaki şekilde verilen tanım, COBOL'da verilen kayıt tanımı ile aynı yapıdadır.
- Şekilde görülen *struct* tanımından sonra, C'de bu tipte değişkenler tanımlanabilir.
- Örnek; ***ogrenci_kayit*** *ogrenci;*

6.3.2.2. Pascal ve C' de Kayıt Sahalarına Başvuru

- C'de bir kayıt, bir dizi elemanı da olabilir. Bu durumda, dizinin her elemanı kayıtta bulunan her bileşeni içerir.

Örnek Aşağıda her elemanı bir ogrenci_kayıt yapısında kayıt olan 100 elemanlı bir dizi tanımlanmaktadır.

ogrenci_kayit ogrenci[100];



Diziye erişimde öncelikle dizi elemanı, daha sonra kayıt sahası adreslenmelidir.

6.3.2.3. Kayıt Ataması

- Kayıt tipi ile ilgili işlemler açısından Pascal, C, C#' ta, iki taraftaki kayıt tipleri aynı ise, bir kayıttaki tüm sahalar karşı gelen kaydın ilgili sahalarına tek bir atama deyimi ile atanabilir. Bu işlem kayıt ataması olarak nitelendirilir.
- Aşağıda verilen, Pascal'da kayıt atamayı örneklemektedir. Bu örnekte *kayıt1* ve *kayıt2*, *ornek* tipinde iki değişken olarak tanımlandıktan sonra, bir atama deyiminde yer aldıklarında, karşı gelen sahaların değerleri, sağdan sola kopyalanmaktadır.

```
Type ornek = record
        A:real;
        B:integer;
End;
Var kayıt1, kayıt2 : ornek;
        .....
kayıt1:=kayıt2;
```

6.3.2.3. Kayıt Ataması

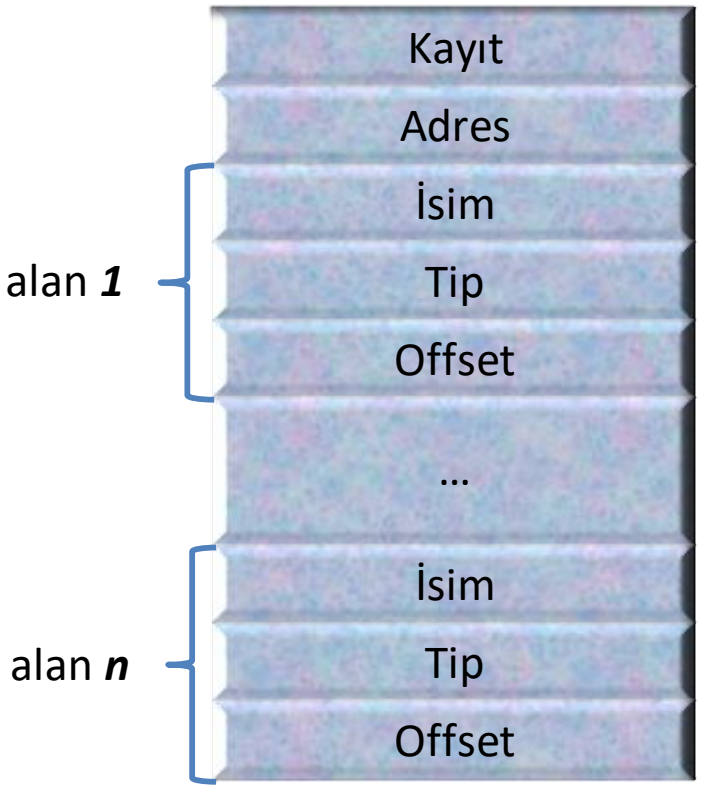
- COBOL'da bulunan *MOVE CORRESPONDING* deyimi ile, iki kayıtta bulunan aynı isimli sahaların, kaynak kayıttan hedef kayda (sağdan sola) kopyalanmasını sağlar.

- Kayıtların sahaları bellekte ardışık yerlerde saklanır. Ancak, bir kayıttaki her sahanın boyutu eşit olmadığı için, diziler için kullanılan erişim fonksiyonu kayıtlar için kullanılamaz.

- Dizi elemanlarına erişim kayıt alanlarına erişimden daha yavaştır. Bunun nedeni dizi altsimgelerinin dinamik, buna karşılık kayıt alanlarının statik olmasıdır.

6.3.2.4. Kayıt Uygulaması

Kayıtların başlangıcına göre Ofset adresi her alanı ile ilişkilidir.



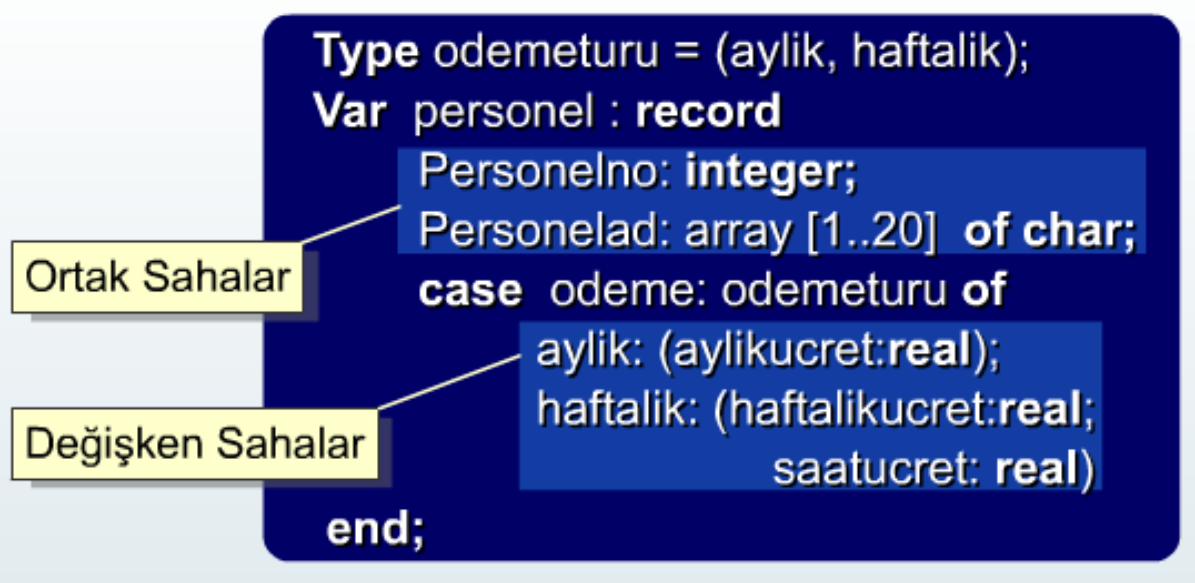
Derleme zamanı kayıt (record) açıklayıcısı. Yürütme zamanında gereksiz çünkü gerçek adresler hesaplanmış oluyor.

6.3.3. Union (Bileşim) Tipi

- Bir programın çalışması boyunca farklı değerler saklayabilen **bileşim** (*union*) tipi, kayıt tipinin özel bir şekli olarak görülebilir.
- Bir programda veya fonksiyonda değişkenlerin aynı bellek alanını paylaşması için ortaklık bildirimi bileşim (union) deyimi ile yapılır. Bu da belleğin daha verimli kullanılmasına imkan verir.
- Bileşim tipinin tasarımı ve bileşim tipinde tip denetimi, programlama dillerinde çeşitli biçimlerde işlenmiştir.
- Örneğin Pascal'da bileşim tipi, kayıt yapısı ile bütünleşmiş ve **değişken kayıt** (*record variant*) olarak adlandırılmıştır. Değişken kayıtlar, bazı ortak özellikleri olan ancak tüm özellikleri ortak olmayan nesneleri göstermeye yarar.
- Tasarım problemleri:
  - 1. Bir tip kontrolü yapılacaksa, nasıl bir kontrol yapılacak? Bu kontrolün dinamik olması zorunlu mu?
  - 2. Tutanaklarla bütünleşmeli midir?

6.3.3. Union (Bileşim) Tipi

- Aşağıda verilen şekildeki değişken kayıt tanımında, personelno ve personelad sahaları ortak sahalardır. Aynı kayıt tanımında, haftalık olarak ödeme yapılan ve aylık olarak ödeme yapılan kişileri göstermek üzere, aylık ve haftalık olarak iki tane de değişken saha bulunmaktadır.

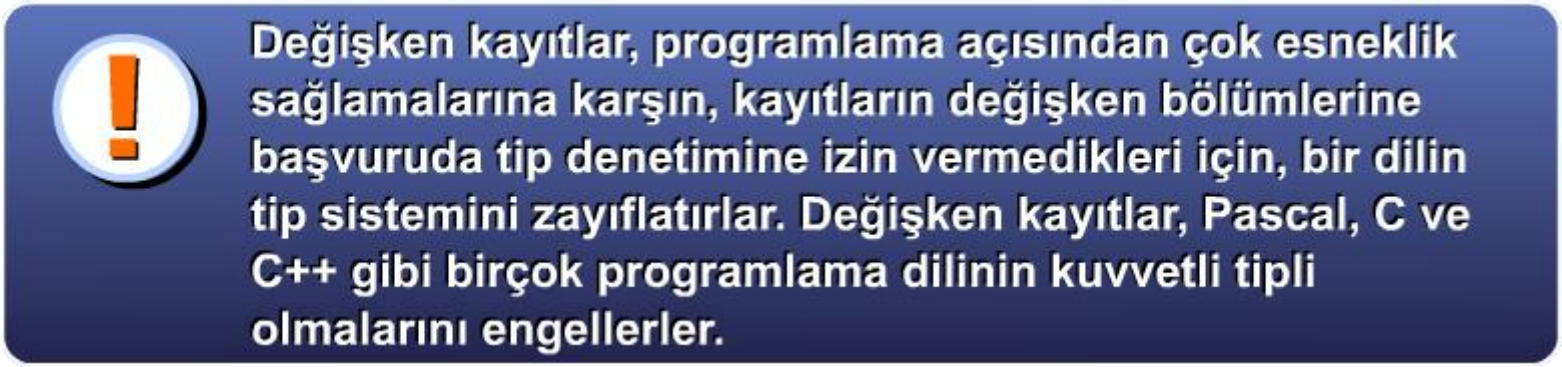


6.3.3. Union (Bileşim) Tipi

- C'de bileşim tipi, union ile yapılır. Örneğin:
  - union paylas{ double f;int i;char kr; };
- Yukarıdaki bildirim yapıldığında, değişkenler için bellekte bir yer ayrılmaz. Değişken bildirimi:
  - union paylas bir, iki;

  şeklinde yapılır.
- Üyelere erişmek aşağıdaki gibi olur:
  - bir.kr= 'A'; iki.f = 3.14; bir.i = 2000;
- C derleyicisi union yapılar için bellekte yer ayıracağı zaman, her zaman en geniş elemanın saklanabileceği kadar yer ayırır (yukarıdaki örnekte double için 8 byte)

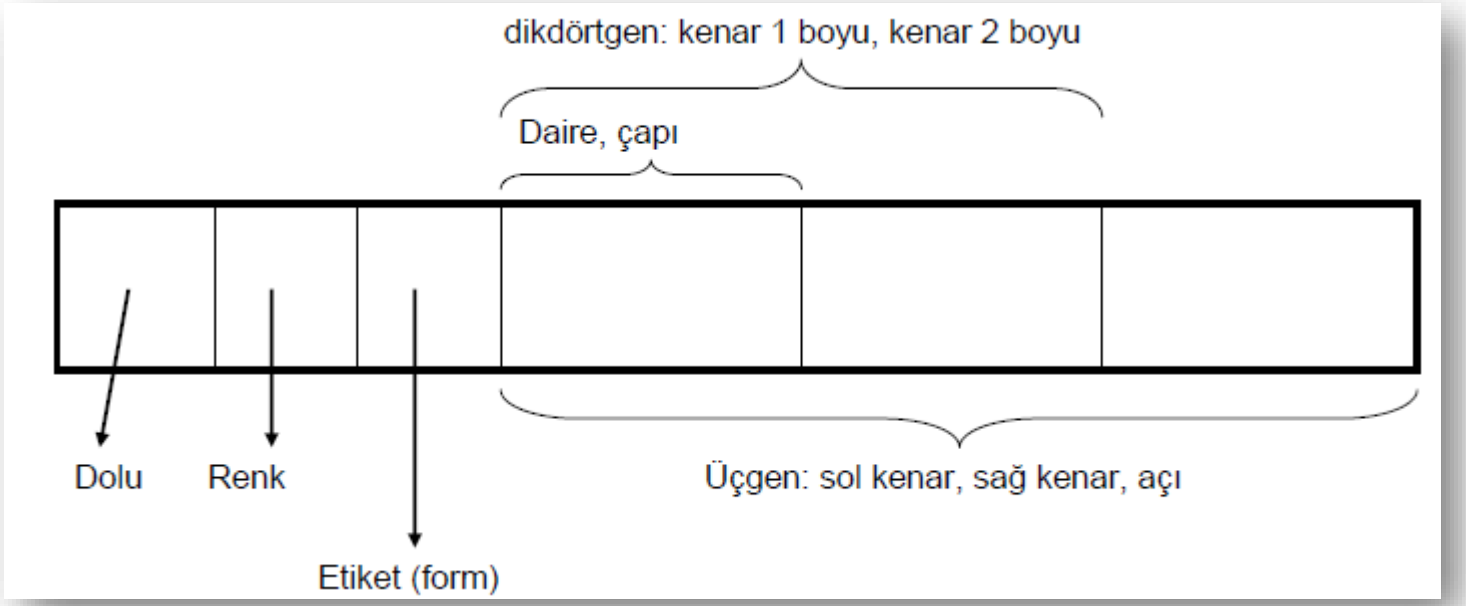
6.3.3. Union (Bileşim) Tipi



- Union tip Ada hariç diğer bütün diller için potansiyel olarak güvensiz yapılardır.
  - Tip kontrolüne izin vermezler.
- Java ve C# union ları desteklemez.
  - Artık programlama dillerinde güvenlik önemlidir.

6.3.3. Union (Bileşim) Tipi

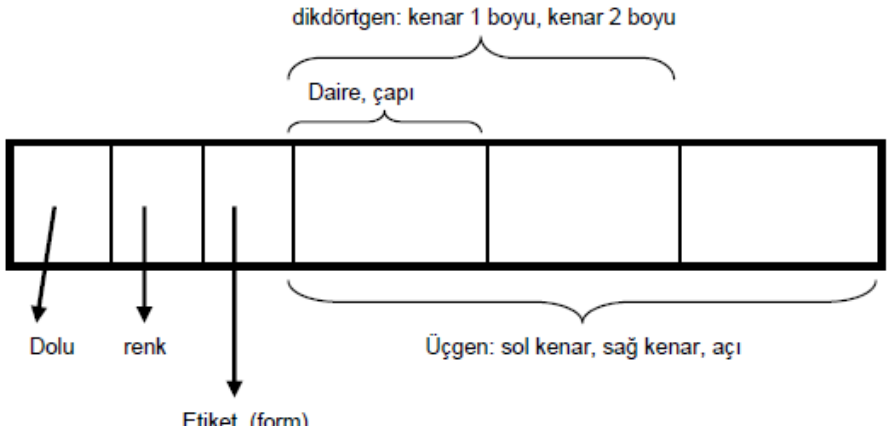
- Etiketlendirilmiş Union (Bileşim)
- Etiketlendirilmiş bileşimde (discriminated union) veri tipi verinin içine ayrıca yazılır. Tip kontrolünü olanaklı kılar.



Etiketlendirilmiş Union (Bileşim) Örneği

- ADA union tip örneği

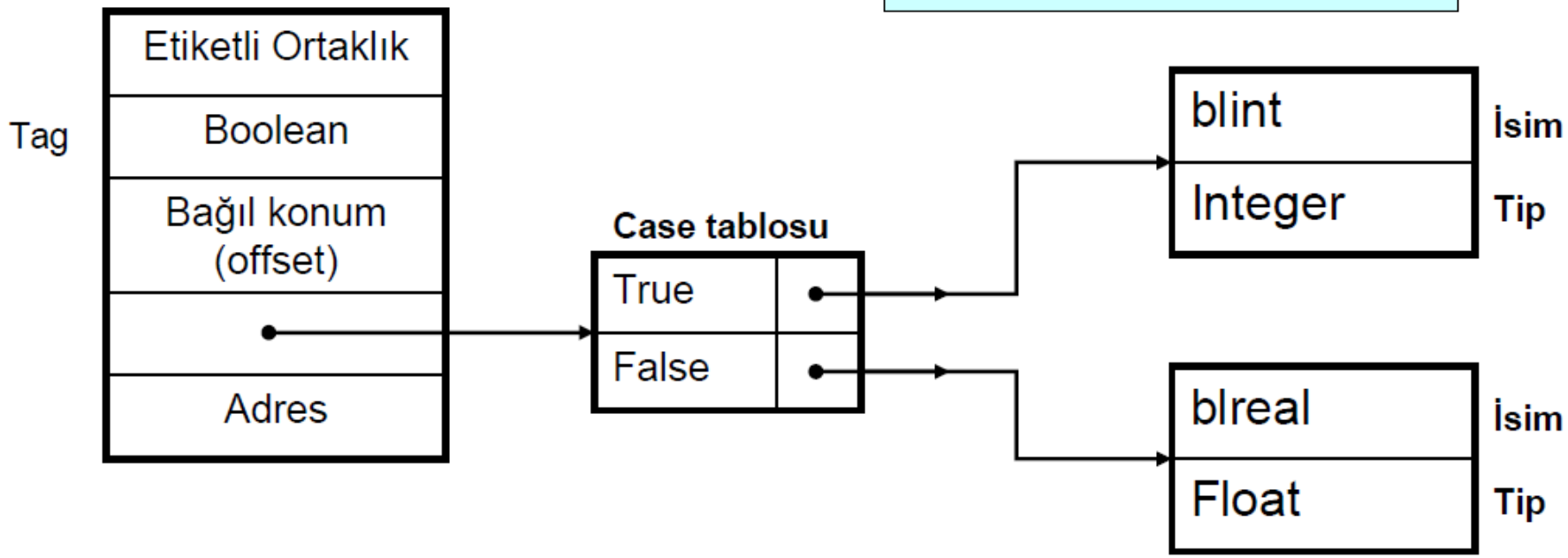
```
type Gsekil is (Daire, Ucgen, Dortgen);
type Renkler is (Kirmizi, Yesil, Mavi);
type Sekil (Form : Gsekil)
  record
     Dolu : Boolean;
     Renk : Renkler;
     case Form is
        when Daire =>
           Capi : Float;
        when Ucgen =>
           Sol_kenar : Integer;
           Sag_kenar : Integer;
           Aci: Float;
        when Dortgen =>
           Kenar_1: Integer;
           Kenar_2: Integer;
      end case;
  end record;

Sekil_1 : Sekil;
Sekil_2 : Sekil(Form => Daire);

Sekil_1 := (   Dolu => True,
               Renk => Mavi,
               Form => Dortgen,
               Kenar_1 => 15,
               Kenar_2 => 5);
```



- Sekil_1 sınırlandırılmadan tanımlandığından her değer atamasında farklı şekiller içine konulabilir. Ancak bu tanımlamanın tam olması gerekir. Aynı alanlar tüm veri tipleri için kullanılabilir. Bunun için yeterince yer ayrılır.
- Buna karşılık Sekil_2 "Daire" ile sınırlandırıldığından, sadece Daire ile ilgili veri saklanmasında kullanılabilir. Sınırlandırıldığı için, statik olarak gerektiği kadar yer ayrılır.

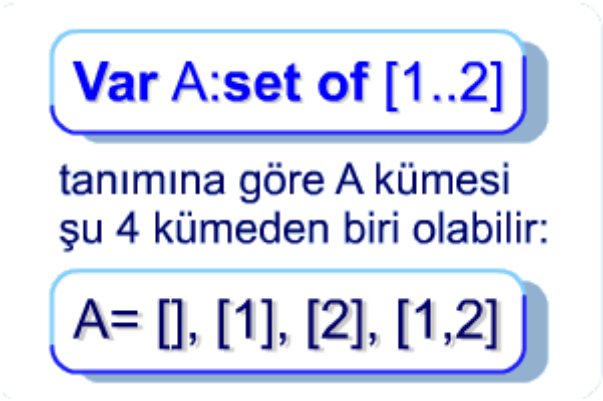
6.3.3. Union (Bileşim) Tipi


```
Type Intreal (Tag: Boolean) is
   record
   case Tag is
        when True => blint : Integer;
        when False => blreal : Float;
   end case;
   end record;
```



Bileşim (Union) Derleme Zamanı Açıklayıcısı

6.3.4. Set (Küme)Tipi



- Bir **küme** (*set*) **tipi**, sıralı bir tipin sıralı olmayan değerlerini saklayabilir. Küme tipleri, sayısal kümeleri modellemek için kullanılır. Yaygın olarak kullanılan *imperative* dillerden sadece Pascal'da kümeler, bir veri tipi olarak tanımlıdır.
- Küme elemanları tek tek veya altalan tipi olarak yazılabilir. Bir kümenin tüm elemanları aynı veri tipinde olmalıdır.
- *Set of* tip oluşturucusu ile sayılama (enum) ve altalanlardan (subrange) da küme tipleri oluşturulabilir.

6.3.4. Set (Küme)Tipi

- Aşağıdaki şekilde, otokume isimli bir küme tipi tanımı ve kume1 ve kume2 adında küme tipi değişkenlerinin tanımı ve kullanımı verilmiştir:

PASCAL'da Küme Tanımlaması

```
Type  otomobiller = (tofas, renault, opel, ford, toyota, hyundai);
          otokume =  set of otomobiller;
var  kume1, kume2 : otokume;
var arabam: otomobiller;
```

PASCAL'da Küme Tipindeki Değişkenlerin Kullanımı

```
kume1:=[tofas, opel, ford];
kume2:=[renault, opel, toyota, hyundai];
```

6.3.4. Set (Küme)Tipi

- Kümelerdeki temel işlem, üyelik sınaması olup, *in* işlemi, bir elemanın belirli bir kümede olup olmadığını sınar.

```
If arabam in kume1 then begin
...
end
```

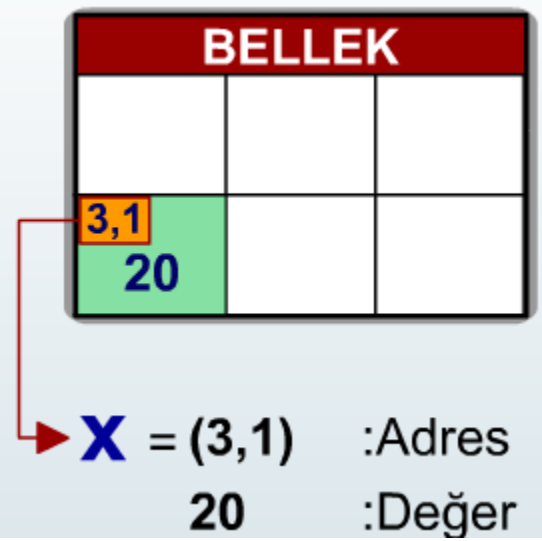
- Pascal, küme bileşimi (union), küme keşismesi (intersection), küme farkı ve küme eşitliği (equality) olmak üzere bir dizi küme işlemi içerir.

A + B → Küme bileşimi

A - B → Küme farkı

A * B → Küme kesişimi

6.3.5. Pointer (Gösterge) Tipi

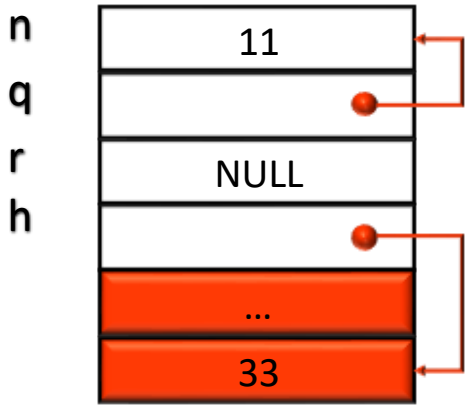


- **Gösterge** (*pointer*) tipi, belirli bir veriyi içermek yerine başka bir veriye başvuru amacıyla kullanılır. Bu nedenle, hem ilkel tiplerden hem de yapısal tiplerden farklı bir veri tipidir. Bir gösterge tipi sadece bellek adreslerinden oluşan değerler ve boş (*null*) değerini içerebilen bir tiptir.

- Gösterge tipindeki değerler, gösterdikleri veriden bağımsız olarak sabit bir büyüklüktedirler ve genellikle tek bir bellek yerine sığarlar.

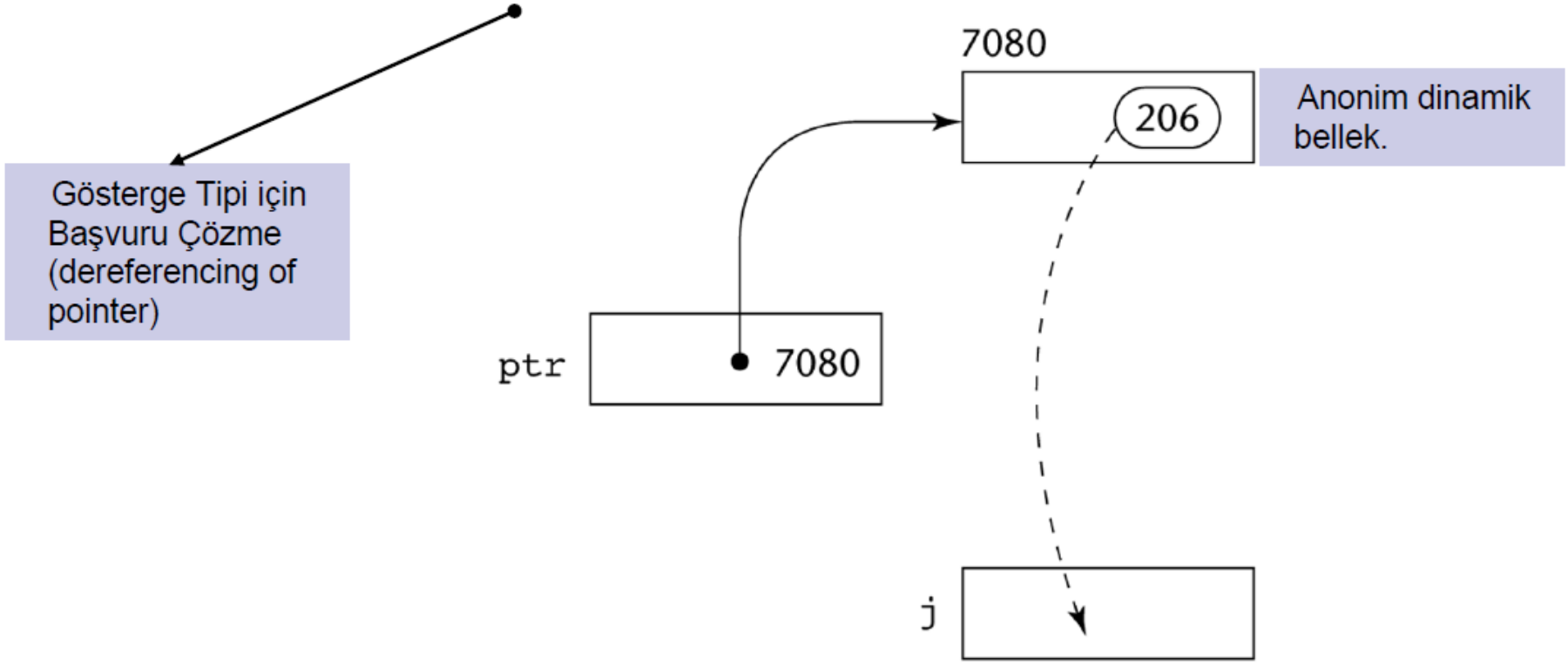
```
int n = 11;
int *q, *r, *h;
q = &n;  r = NULL;
h = (int *)malloc(sizeof(int));
*h = 33;
int m = *h;
```

Geri havale (dolaylı referans)



6.3.5. Pointer (Gösterge) Tipi

Örnek: atama işlemi: `int *ptr; j = *ptr;`



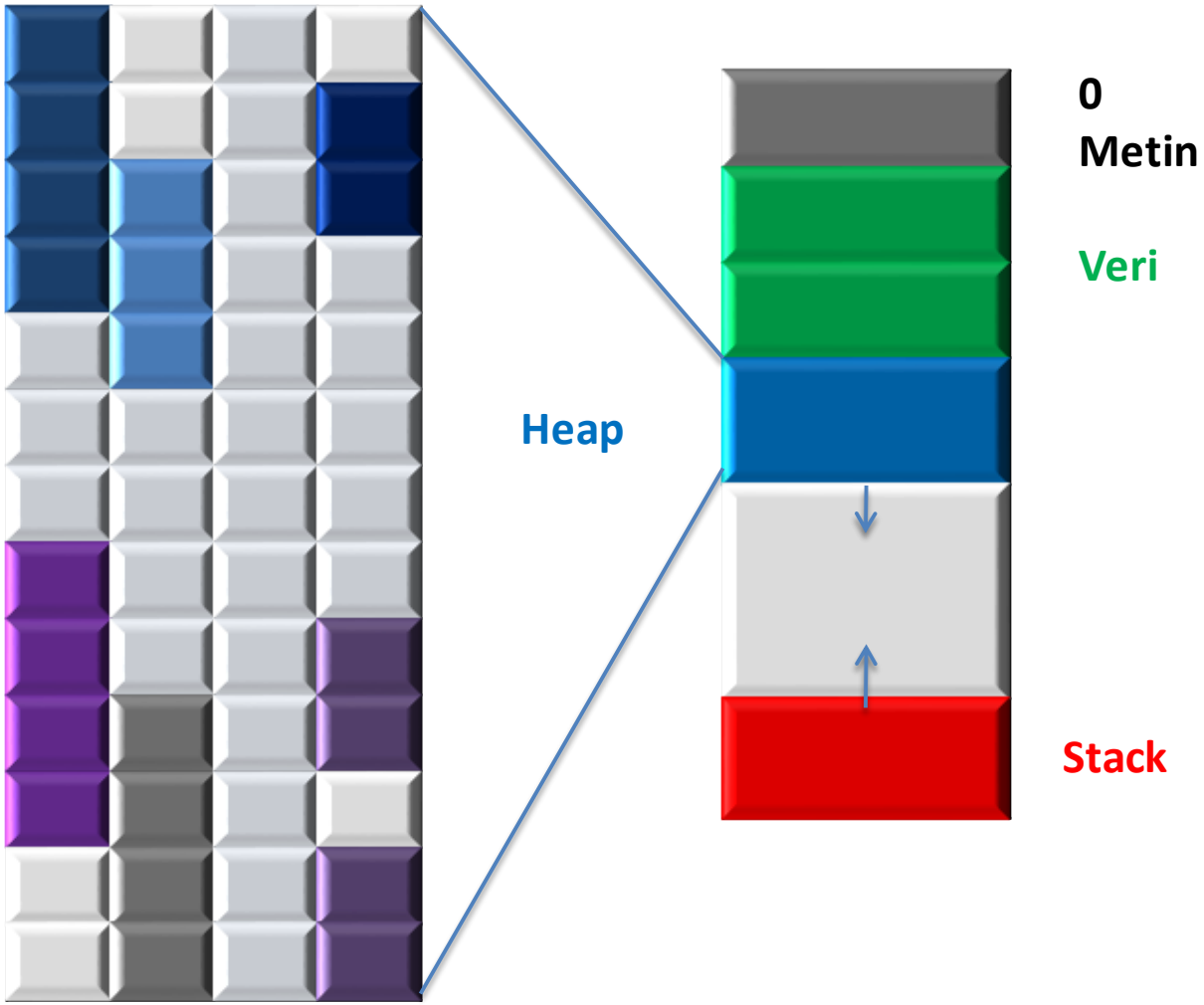
Heap Bellek

Implicit –

Otomatik

Explicit –

Programcı komutları



6.3.5.1. Genel Özellikler

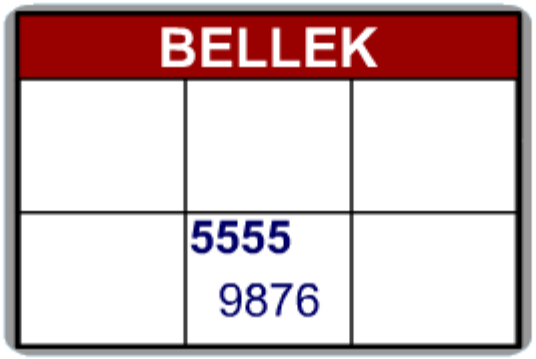
- Gösterge tipindeki değişkenler iki amaçla kullanılabilir:

- ***1. Dolaylı adresleme:*** Göstergeler, bellekteki değerlere erişim yolunu göstermek için dolaylı adresleme aracı olarak kullanılabilirler. Dolaylı adreslemeyi bir örnekle inceleyelim.
- *abc* isimli bir gösterge değişkeninin *5555* değerini içerdiğini düşünelim. Eğer, adresi *5555* olan bellek hücresinde *9876* değeri varsa, *abc* değişkenine normal başvuru *5555* değerini, dolaylı bir başvuru ise *9876* değerini verecektir.

abc = 5555

Adres = 5555

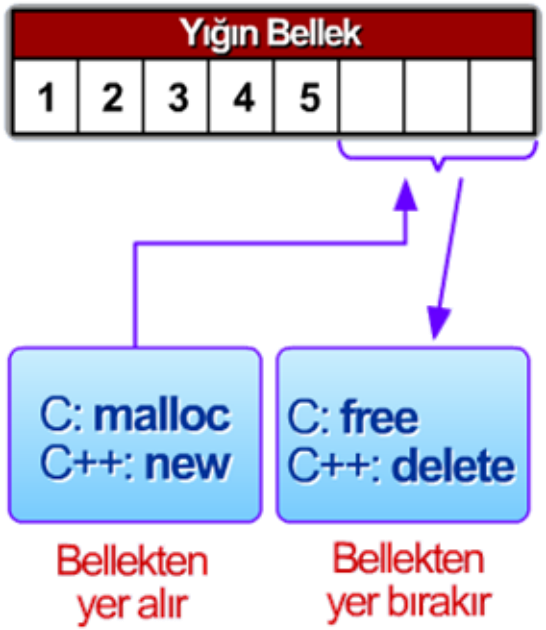
Adres Değeri = 9876



Normal Başvuru: 5555

Dolaylı Başvuru: 9876

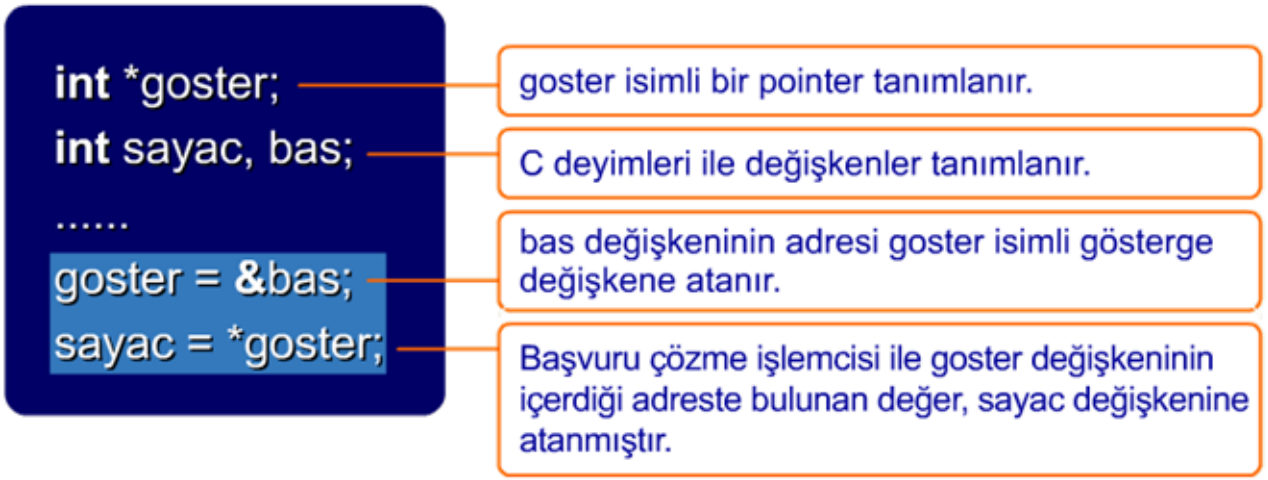
6.3.5.1. Genel Özellikler



- ***2. Dinamik bellek yönetimi:*** Programlarda çalışma zamanında büyüyen/küçülen veri yapılarını içeren yığın belleğe erişmek için gösterge tipinde değişkenler kullanılabilir. Yığın bellekte saklanan değişkenlerin tanımlayıcı isimleri olmadığı için, sadece gösterge değişkenler ile başvurulabilir.
- Yığın bellek yönetimi için gösterge kullanımını sağlayan programlama dillerinde yığın bellekten yer almak için bir işlemciye veya fonksiyona gereksinim vardır. Örneğin C'de malloc fonksiyonu, C++'da new işlemcisi, bellekten yer almak için kullanılabilir. Bazı dillerde, belleği serbest bırakmak için de ayrı bir işlemci veya fonksiyon vardır. C'de free fonksiyonu, C++'da delete işlemcisi bu amaçla kullanılabilir.

6.3.5.2. Gösterge Tipi için Atama ve Başvuru Çözme

- C'de gösterge tipi için, atama ve başvuru çözme (*dereferencing*) olmak üzere iki temel işlem tanımlanmıştır.
- **Gösterge Tipi için Atama:**
- Atama işleminde bir gösterge değişkene belirli bir nesnenin adresi verilir ve bunun için bir değişkenin adresini veren & işlemcisi kullanılır. * sembolü ise başvuru çözme işlemcisini göstermektedir.



NOT: Son iki deyim sayac = bas; deyimi ile aynı görevi görmektedir.

6.3.5.2. Gösterge Tipi için Atama ve Başvuru Çözme

C'de

```
(*gosterge).adres
gosterge    adres
```

Pascal'da

```
gosterge^.adres
```

- **Gösterge Tipi için Başvuru Çözme:**
- Gösterge değişkenler, kayıtlara başvuru için kullanıldığında çeşitli programlama dillerinde farklı sözdizimler kullanılır. C ve C++'da, bir kaydın bir sahasına bir gösterge değişkenle başvuru için iki gösterim vardır.
- Örneğin, *gosterge* isimli bir gösterge değişkenin, bir kaydın *adres* isimli bir sahasına başvuruda kullanılması için olası gösterimler yandaki şekilde görüldüğü gibidir.
- Aşağıdaki şekilde ise dinamik bellek yönetimine ve kayıtların göstergelerle kullanımına ilişkin bir örnek görülmektedir:

6.3.5.2. Gösterge Tipi için Atama ve Başvuru Çözme


```
struct aa{
int sayac;
char ad;
};
struct aa *gosterge;
```

aa tipindeki bir değeri gösterebilen gosterge değişkeni tanımlanır.

```
gosterge = (struct aa *) malloc (sizeof (struct aa));
gosterge->sayac =101;
gosterge->ad = 'ABC';
```

aa kaydı için malloc fonksiyonu ile bellekten yer alınır ve aa tipindeki bir değeri gösterebilen gosterge değişkeni ile aa kaydının sahalarına atama yapılır.

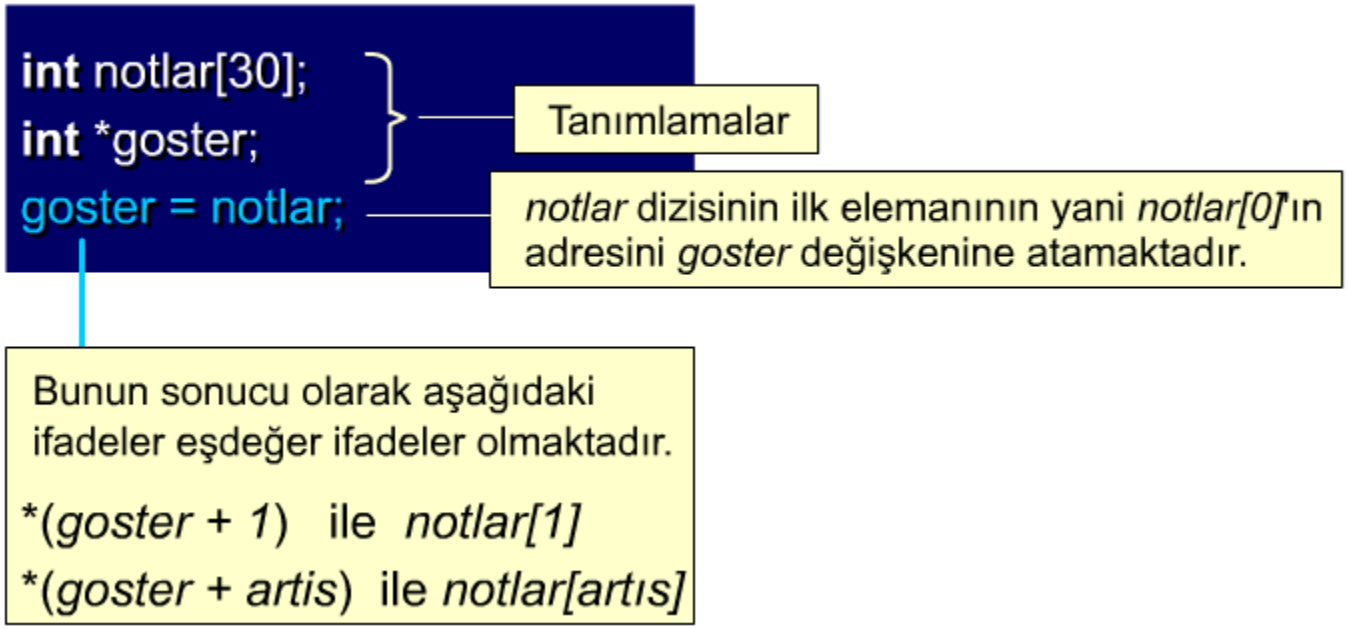
6.3.5.3. Gösterge Aritmetiği


```
int *goster;
goster + artis
```

- C ve C++'da gösterge değişkenler ile **gösterge aritmetiği** yapılabilir.
- Yandaki örnekte verilen deyiminin anlamı, *goster* değişkenine *artıs* değişkeninin değerinin eklenmesi değil, *artıs* değişkeninin değerinin, *goster* in işaret ettiği adresteki elemanın bellekte kapladığı yer kadar ölçeklenmesi ve daha sonra *goster* in değerinin artırılmasıdır.
- Örneğin, *goster*, bellekte iki sekizli (byte) kaplayan bir nesneyi işaret ediyorsa, *artıs* ın değeri *2* ile carpılır ve bu değer, *goster* in değerine eklenir.

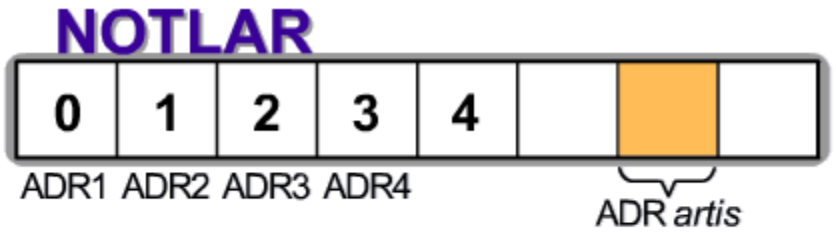
6.3.5.3. Gösterge Aritmetiği

- Gösterge aritmetiğinin gerçekleştirilebilmesi, özellikle dizilerle ilgili işlemlerde yararlıdır. C ve C++'da, dizilerin indislerinin alts ınırı her zaman sıfırdır. Bir dizi ismi, indisler kullanılmadan programda yer aldığında, dizinin ilk elemanının adresini belirtir.



6.3.5.3. Gösterge Aritmetiği


```
int notlar[30];
int *goster;
goster = notlar;
```



ADR1: *goster = notlar*

ADR2: (*goster+1*) ⟷ *notlar[1]*

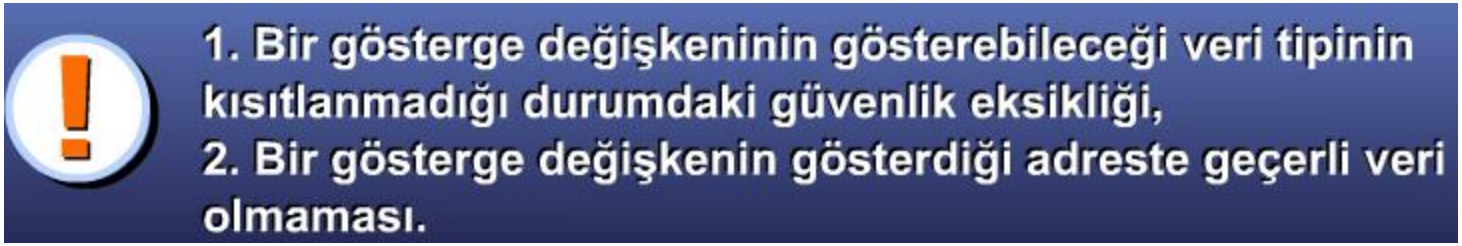
ADR *artis*: (*goster+artis*) ⟷ *notlar[artis]*

*(*goster + 1*) ile *notlar[1]*

*(*goster + artis*) ile *notlar[artıs]*

6.3.5.4. Gösterge Veri Tipi Sorunları

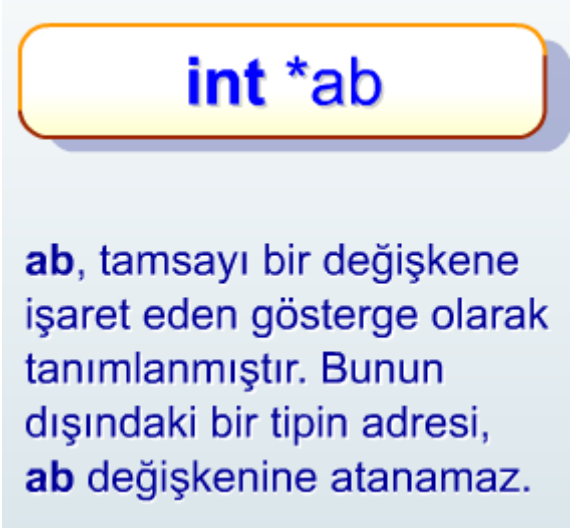
- Gösterge veri tipinin sorunları iki başlık altında toplanabilir:



- Gösterge değişkenlerini içeren ilk yüksek düzeyli programlama dili PL/I'dır. PL/I'da, bir gösterge değişkenin gösterebileceği elemanların veri tipi sınırlanmamıştır. Bu durumda, gösterge kullanımlarında durağan tip denetimi yapılamadığı için, programların güvenilirliği azalmıştır. Bu nedenle, sonraki programlama dillerinde gösterge değişkenlerin gösterebildikleri veri tipleri tek bir veri tipi ile sınırlanmıştır.

6.3.5.4. Gösterge Veri Tipi Sorunları

- Örneğin C'de bir gösterge değişken tanımlanırken, adresini tutabileceği değişken tipi de belirtilmelidir.



6.3.5.4.1. *Dangling Pointer* (Sallanan Gösterge)

- Bir gösterge değişkenin gösterdiği adreste geçerli veri olmaması durumu, göstergenin serbest bırakılmış bir dinamik yığın değişkene işaret etmesi ile oluşur. Serbest bırakılmış bir bellek adresini gösteren değişkene **sallanan gösterge** (*dangling pointer)* denir. Bir göstericinin gösterdiği belleğin bir şekilde sisteme iade edildiği durumdur.

- **Pascal, C, C++ ve Java'da Sallanan Gösterge:**
- Pascal'da, yığın değiştirilebilir değişkenler ***new*** ile oluşturulur ve ***dispose*** ile yok edilirler. Pascal'da gösterge değişkenler, sadece değiştirilebilir yığın değişkenlere erişmek için kullanılırlar. Ancak Pascal'da *dispose* deyimi bir değişkeni gösteren tüm gösterge değişkenleri düzenlenmediği için sallanan göstergeler oluşabilir.

6.3.5.4.1. *Dangling Pointer* (Sallanan Gösterge)


```
int *a, *b;
...
a =malloc(sizeof (int));
b =a;
free(a);
```

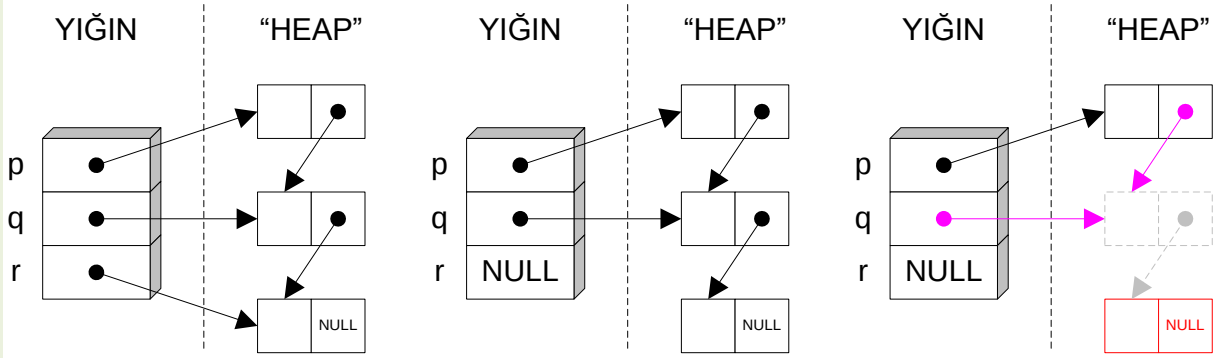
- Benzer şekilde C ve C++'da da sallanan gösterge sorunu vardır. C'de sallanan gösterge oluşumu yandaki şekilde görülmektedir.
- Başka bir örnek verelim:
  - **int* f () { int fv = 42; return &fv; }**
- Aşağıdaki kod gösterici dönen f'i çağırır, sonucu bir gösterici değişkene koyar, sonra bunu değiştirmeye çalışır.
  - **int* p = f(); *p = 0;**
- Fakat göstericinin gösterdiği fv f'nin içinde tanımlıdır, f'nin yaşamı bitince onun bellekte kullandığı yer de iade edilmiştir, bu şekilde kullanılması beklenmedik sonuçlar doğurur.
- Java'da ise gösterge veri tipine dilde yer verilmeyerek, güvenilirlik problemlerinin önlenmesi amaçlanmıştır.

6.3.5.4.2. Bellek Sızıntısı

Kayıp yığın dinamik (Heap-Dynamic) değişkenler.

- Yığın dinamik değişken herhangi bir program göstericisi tarafından gösterilmemektedir.
- Örnek:
  a) Gösterici p1 önce bir yığın değişkeni gösterir:
  int *p1;
  ```
  p1 =(int *)malloc(sizeof(int));
  ```
  b) Daha sonra, yeni yaratılmış başka birini:
  ```
  p1 =(int *)malloc(sizeof(int));
  ```
- Bu şekildeki kayıplara bellek kaçağı veya sızıntısı denir.

```
class Node {
  int  value;
  Node *next;
};

...

Node *p = new Node();
Node *q = new Node();
Node *r = new Node();

p->next = q;
q->next = r;
                // a
r = NULL;
                // b
delete q;
                // c
...
```



(a) (b) (c)

Şekil: Verilen program parçası için belleğin durumu.
(a) *a* satırındaykenki durum.
(b) *b* satırındaykenki durum.
(c) *c* satırındaykenki durum.

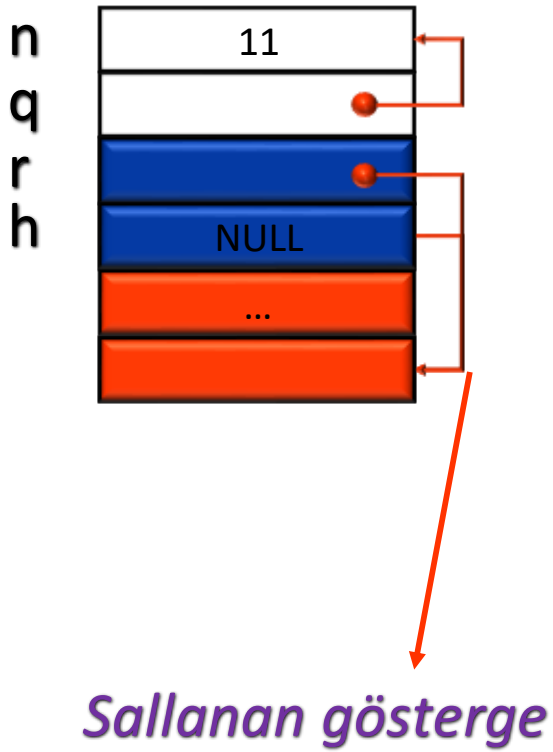
Yukarıda verilen program parçasının *a* satırına kadar işletilmiş olduğunu düşünelim. Buna göre belleğin şu andaki durumu Şekil a'daki gibi olacaktır. Bu durumda tüm nesnelere programın bağlamındaki değişkenlerden (statik ya da yığındaki değişkenlerden) ulaşmak mümkündür.

b satırına gelindiğinde r değişkenine NULL atanarak r ile işaret ettiği nesne arasındaki bağlantı koparılır. Son olarak *c* satırında q'nun işaret ettiği nesnenin kapladığı alan bellek yöneticisine geri verilir. Bu anda iki sorun ortaya çıkar:

1. Artık q'nun ve p->next'in işaret ettiği bellek bölgesi bellek yöneticisine geri verildiği için geçersizdir. Bu yüzden q'ya ve p->next'e *geçersiz işaretçiler* deriz.
2. Eskiden r'nin işaret ettiği nesneye artık bağlamdan ulaşmanın bir yolu kalmamıştır. Bu tür nesnelere *çöp* denir. Çöp nesnelerin bellekte gereksiz alan kaplamasına da *bellek sızıntısı* denir.

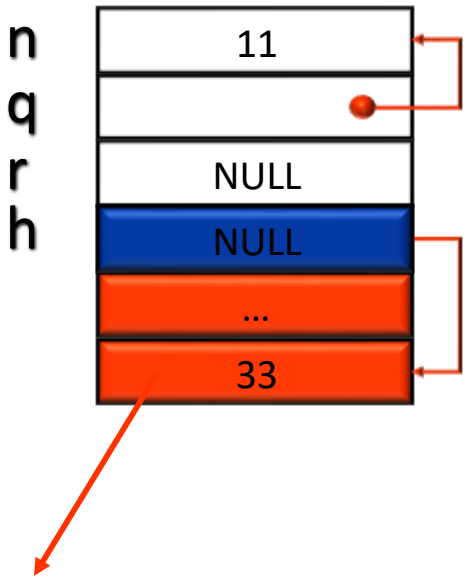
Pointer Problemleri – Sallanan Gösterge


```
int n = 11;
int *q, *r, *h;
q = &n;  r = NULL;
h = (int *)malloc(sizeof(int));
*h = 33;
r = h;
free(h);
```



Pointer Problemleri – Kayıp heap-dinamik değişkenler


```
int n = 11;
int *q, *r, *h;
q = &n;  r = NULL;
h = (int *)malloc(sizeof(int));
*h = 33;
h = NULL;
```



Kayıp heap-dinamik değişken

Çöp, Bellek Sızıntısı

6.3.5.5. Dillerde göstericiler

- Pascal: sadece dinamik bellek yönetimi için kullanılır.
  - Açıkça gösterici çözme var (postfix **^).**
  - Sallanan göstericiler var (**dispose komutu nedeniyle).**
- Ada: Pascal'dan biraz daha iyi
  - Dinamik değişkenler göstericinin tip yaşam döngüsü bittiğinde otomatik olarak serbest bırakıldığından, bazı sallanan göstericilere izin verilmiyor.
  - Bütün göstericilerin otomatik başlangıç değeri null.
  - Benzer sallanan değişkenler problemi (fakat açıkça serbest bırakma nadir olduğundan daha az oluyor).

6.3.5.5. Dillerde göstericiler

C ve C++

- Dinamik bellek ve adres yönetimi için kullanılır.
- Açık başvuru çözme, adres kaldırma işlemleri.
- Alan tipinin sabitleştirilmesi şart değil (**`void *`**).
- **`void *`** - herşeyi gösterebilir ve tip kontrolü yapılabilir (başvuru çözme işlemi yapılamaz)
- Sınırlı olarak adres aritmetiği yapılır, örneğin:

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` ➔ `stuff[5]` ve `p[5]`

`*(p+i)` ➔ `stuff[i]` ve `p[i]`

(örtülü ölçekleme)

6.3.5.6. Göstericilerin Değerlendirilmesi

- Göstericilerin Değerlendirilmesi:

1) Sallanan göstericiler, sallanan nesneler, yığın bellek yönetimi ile birlikte problemler.

2) Göstericiler programlama dillerinde “go to” gibidirler:
   - “go to” programda bir sonraki işlemde gidilebilecek noktayı genişletir;
   - “pointer” da aynı şekilde erişilebilecek bellek yerini genişletir.

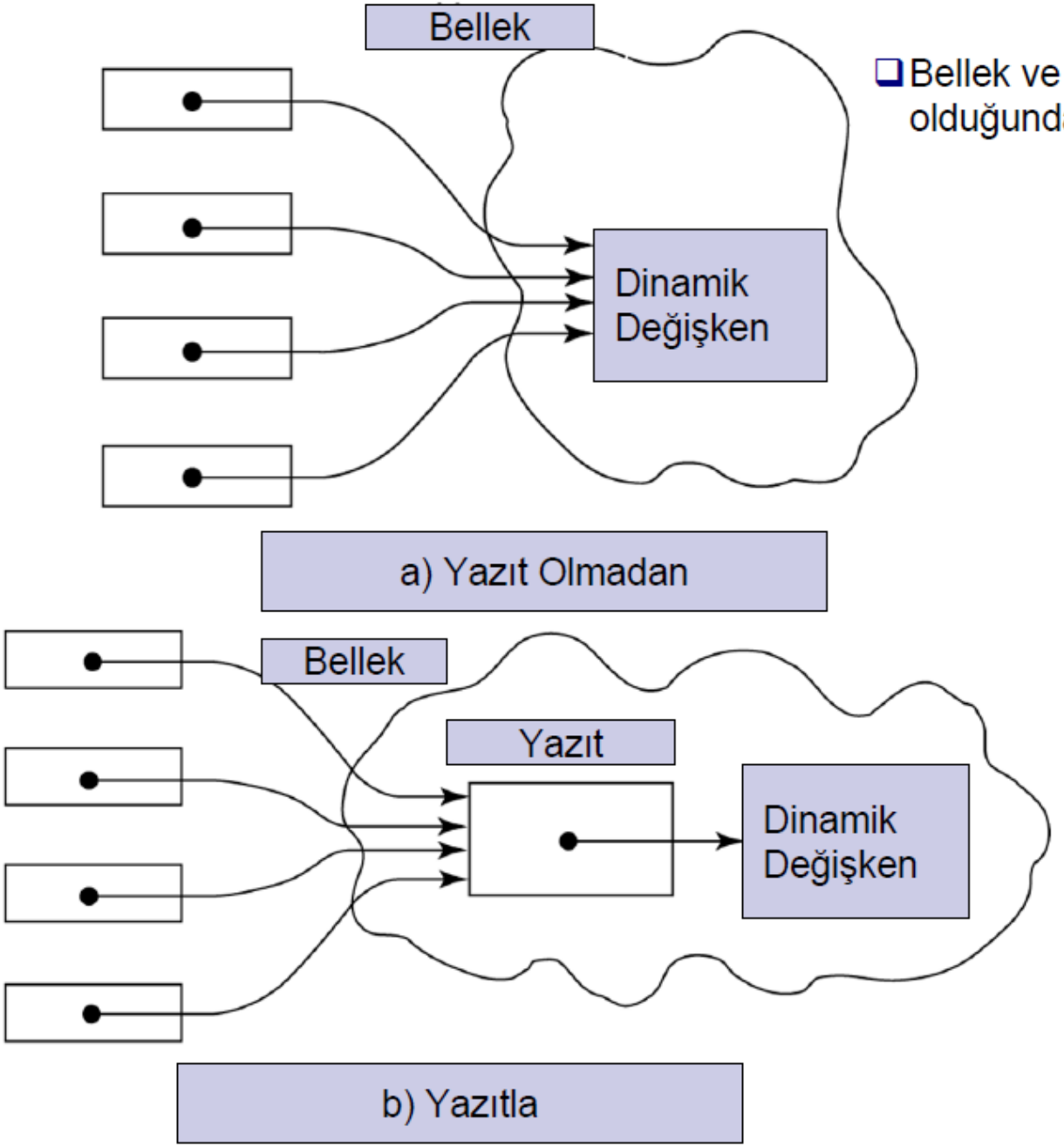
3) Dinamik veri yapıları için gerektiğinden onlardan vazgeçemeyiz

Gösterici sorunlarına çözüm

1. Yazıt (Tombstone):

- Yığın dinamik bellek değişkenini gösteren ek bellek hücresi (Lomet 1975).
- Gerçek gösterici yazıt üzerindeki belleği gösterir; o da gerçek yığın dinamik değişkeni.
- Yığın dinamik değişken sisteme geri verildiğinde yazıt kalır ancak içeriği “nil” olur. Böylelikle bu değişkenin artık olmadığını bildirir.

Sallanan gösterici problemine çözüm



- Bellek ve zaman kaybına neden olduğundan kullanılmadı.

Gösterici sorunlarına çözüm

2. Kilit ve Anahtar (Locks and keys) (Fisher ve LeBlanc, 1977, 1980)

- Göstericiler [anahtar, adres] çiftlerinden oluşturulur.
- Yığın dinamik değişken ise [kilit, değişken] çiftinden oluşur.
- Yığın dinamik değişken yaratıldığında kilit ve anahtara aynı (tam sayı) değer atanır. Bu değişkene referans verildiğinde anahtar ve kilit değerlerinin eşit olması kontrolü yapılır.
- Değişken silindiğinde kilit değeri de kullanılmayan bir değer yapılır. Böylelikle başka bir gösterici aynı değişkeni daha sonra gösterdiğinde anahtar ve kilit değerler tutmayacağından hata mesajı verilir.

- Sallanan gösterici probleminin esas çözümü programcıdan bellek iade yetkisini almaktır. Java ve C# referans değişkenleri ve Lisp bu yetkiyi vermeyip, kullanılmayan belleği daha sonra kendisi belirleyerek sisteme iade etme/yeniden kullanma yöntemini kullanmaktadır.

- Yığın bellek yönetimi (Heap management)
  - Dilin tasarım problemi olmaktan çok, dilin uygulanması problemi.
  - Sabit boylu hücre (cells) – Değişken boylu hücre
    - Sabit boyuta örnek Lisp.
- Kullanılmayan belleği toplamanın iki yöntemi vardır: Referans sayıcıları (Reference counters) (aceleci yaklaşım) ve çöp toplama (garbage collection) (tembel yaklaşım)

1. Referans sayıcıları: dinamik yığın bellekteki her değişken için bir sayıcı verilir ve kaç gösterici tarafından değişken gösteriliyorsa bu bilgi saklanır. Sayıcı sıfır olduğu zaman bellek kullanılmıyor demektir, sisteme iade edilir.
  - Dezavantajı: bellek alanı ve işlem zamanı kullanılır (özellikle hücreler küçükse). Dairesel tanımlanmış göstericiler için problem.

Değişken boylu hücreler

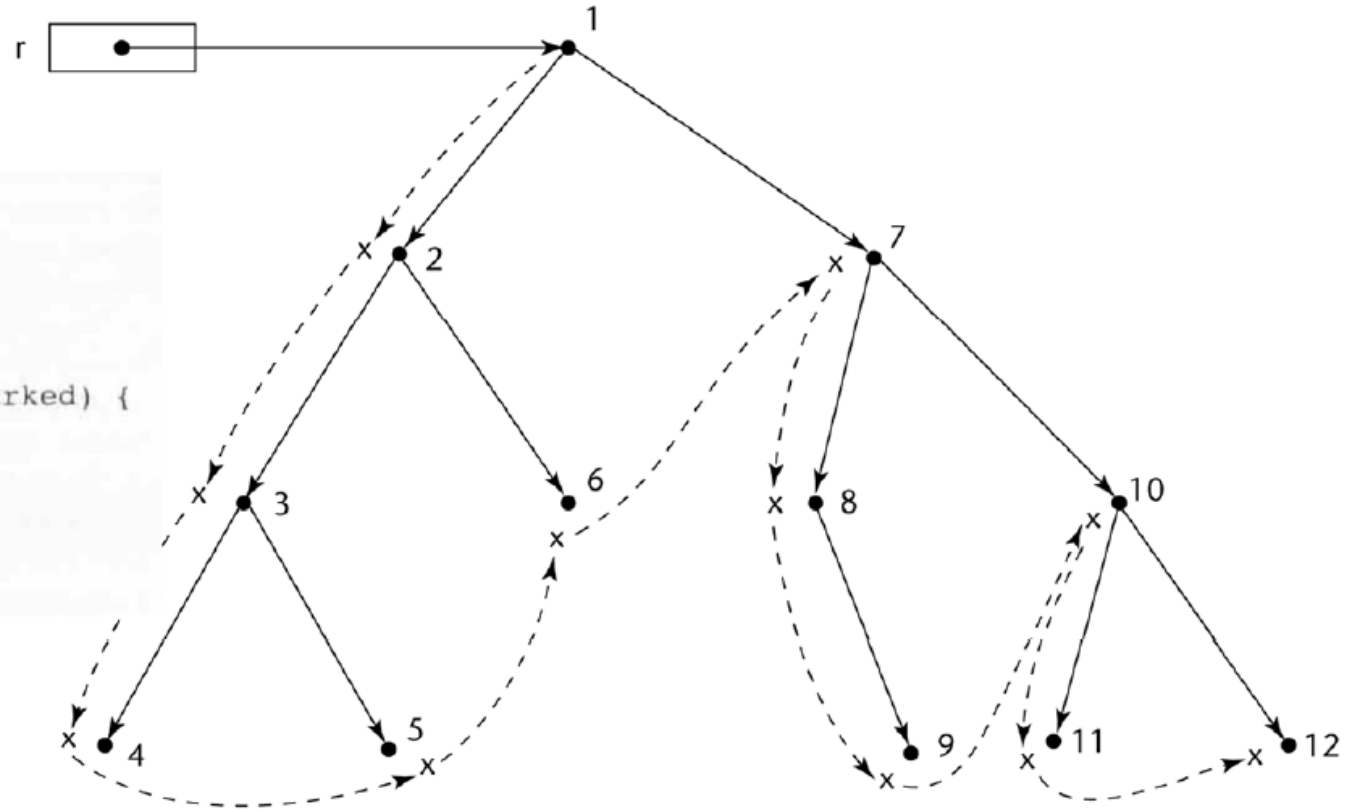
- Hücre boyları değişkense bahsedilen güçlüklere ilaveten başka güçlükler de gözlenir.
- Eğer çöp toplama yöntemi kullanılıyorsa ek olarak aşağıdaki zorluklar gözlenir:
  - Hücrelerin boyları sabit olmadığından işaretlenen hücrenin boyutları bilinemez. Çözüm için her hücrenin başına hücrenin boyu yazılabilir.
  - Kullanılan hücrelerin zincirini takip etmek de zor. Çünkü her hücre farklı ve bir sonraki hücreyi gösteren gösterici farklı yerde olabilir.
  - Kullanılabilir alanın listesinin tutulması da hücre boyları ve yapıları farklı olunca daha zor.
- Eğer referans sayıcı yöntemi kullanılırsa ilk iki problem ortadan kalksa da kullanılabilir alanların yönetim zorlukları kalır.

2. Çöp toplama: bellekte yer olduğu sürece yeni bellek al ve kullan, işin bitince terk et; bellek tükenince çöpleri topla.

- Her hücrenin işaretleme için kullanılan bir ek bit'i olur.
- Bütün dinamik yığın bellekten kullanılan hücreler çöp olarak işaretlenir.
- Bütün göstericiler taranır ve yığın bellekte bu şekilde gösterilebilen hücreler "çöp değil" şeklinde işaretlenir.
- Sonuçta çöp olarak işaretli kalan hücreler, kullanılabilir hücrelere eklenir.
- Dezavantajı: en çok ihtiyacınız olduğunda, en yavaş çalışır. Programın bellek ihtiyacı artınca, büyük bellekte çöp toplama daha uzun sürdüğünden yavaşlar.

İşaretleme algoritması

```
for every pointer r do
    mark(r)

void mark(void * ptr) {
    if (ptr != 0)
      if (*ptr.tag is not marked) {
        set *ptr.tag
        mark(*ptr.llink)
        mark(*ptr.rlink)
      }
}
```



Kesikli çizgi işaretlemenin sırasını gösteriyor

Çöp Toplama Yöntemleri

- Başvuru sayma yöntemi (reference counting): Nesnelere yapılan başvuru sayısının takibinin yapılmasına dayanır. Bunun için her nesne için bir başvuru sayısı tutulur. Yeni yaratılan nesnelerin başvuru sayısı her zaman 1'dir. Bir nesneye yeni bir başvuru yapıldığında bu sayı bir artırılır, başvurulardan biri silindiğinde ise bir azaltılır. Azaltma sonucunda başvuru sayısı 0 olan nesneler bellek yöneticisine geri verilir
- İşaretle-süpür yöntemi (mark-sweep): Çöp nesneleri belirlemek için ilk akla gelen yöntemi kullanır. Nesneler bağlamdaki değişkenlerden başlanarak taranır. Ulaşılabilenler işaretlenir. Ulaşılamayanlar bellek yöneticisine geri verilir.
- Kopyalama yöntemi (copying): Bellek uzayını iki alt uzaya ayırır: Ekin uzay ve eskimiş uzay. Çöp toplama zamanı geldiğinde toplayıcı, etkin uzay ile eskimiş uzayın görevlerini takas eder ve eski etkin uzaydaki ulaşılabilir nesneleri yeni etkin uzayın başına kopyalar. Toplama işlemi bittiğinde tüm ulaşılabilir nesneler yeni etkin uzayın en başına taşınmış olur
- Kuşak yaklaşımı: Nesneler hayatta kalma sürelerine göre kuşaklara ayrılmıştır. Genç kuşak her toplama çevriminde dolaşılırken, yaşlı kuşaklar daha seyrek dolaşılır. Ayrıca birkaç dolaşma boyunca hayatta kalan genç kuşaklar daha yaşlı kuşaklara aktarılırlar
- Azar azar toplama yaklaşımı: Programın arkasında onunla zaman paylaşımlı olarak çalışan bir çöp toplayıcı, programı belirgin kesintilere uğratmadan otomatik bellek yönetimi sağlanabilir.

6.3.5.5. Referans Tipi

- C + + formal parametreleri için öncelikle kullanılan bir başvuru türü denilen gösterge türü özel bir tür içerir.
  - Avantajı: Hem referans değerini hem de veri değerini verebilir.
- Java C++'in referans değerini uzatarak göstergelerin sadece referans değeri tutmasını sağlar.
  - Referanslar yerine adresleri olmaktan çok, nesnelere başvurular vardır.
- Örtülü olarak başvuru çözülen sabit göstericilerdir.
- Sabit olduklarından bir değişkenin adresiyle başlatılmaları gerekir ve daha sonra da bu adres değişemez.
- Referans tip değişken çağıran fonksiyonda tanımlanınca, çağıran ve çağrılan arasında iki yönlü haberleşme olanağı sağlar. Aynı amaçla göstericiler de kullanılabilir ama bunların başvuru çözümlerinin açıkça yapılması gerekir ki programın okunabilirliğini azaltır.
- C# hem Java'nın nesne modelini hem de C++'nın referans modelini kullanmaktadır.
- Referans tip değişkenler "ve" işareti ile başlar:

```
int result;
int &ref_result = result;
…
ref_result = 100;
```

- Java – Sadece referans tip göstericiler bulunur.
  - C++ referans tiplerinden geliştirilmiştir.
  - Sınıf nesnelerini gösterirler (yığın bellekte duran).
  - Bu nedenle gösterici aritmetiği yoktur.
  - Kullanılmayan belleği sisteme iade eden açık bir komut yoktur, bunun yerine çöp toplama kullanılır (garbage collection).
  - Bu nedenle sallanan referanslar da yoktur.
  - Referans çözme her zaman örtülüdür (implicit).
  - C++'dan farklı olarak gösterdiği nesneler değişebilir.
  - Örnekte "String" standart Java sınıfı (class):

```
String str1;
…
str1 = "Bu bir dizgidir";
```

İlk satırda str1 bir dizgi nesnesine referans tip gösterici olarak tanımlanıyor. Başlangıç değeri "null". Daha sonra "Bu bir dizgidir" nesnesini gösterecek atama yapılıyor.

- C# – Java referans tip göstericiler, C++ göstericilerle birlikte bulunur.
  - Ancak kullanılmaları istenmez.
  - Kullanan “metot”ların “unsafe” olarak tanımlanması gerekir.
  - Esas olarak C ve C++ kodları ile birlikte çalışabilmeleri için konulmuştur.

6.4. KUVVETLİ TİPLEME

- **Kuvvetli tipleme** (strong typing), farklı veri tiplerinin farklı soyutlamaları göstermeleri nedeniyle etkileşimlerinin kısıtlanmasıdır.

- Bir programlama dilinin kuvvetli tipli (strongly typed) bir dil olarak nitelenmesi için, dilin tüm tip hatalarını yakalaması gereklidir.

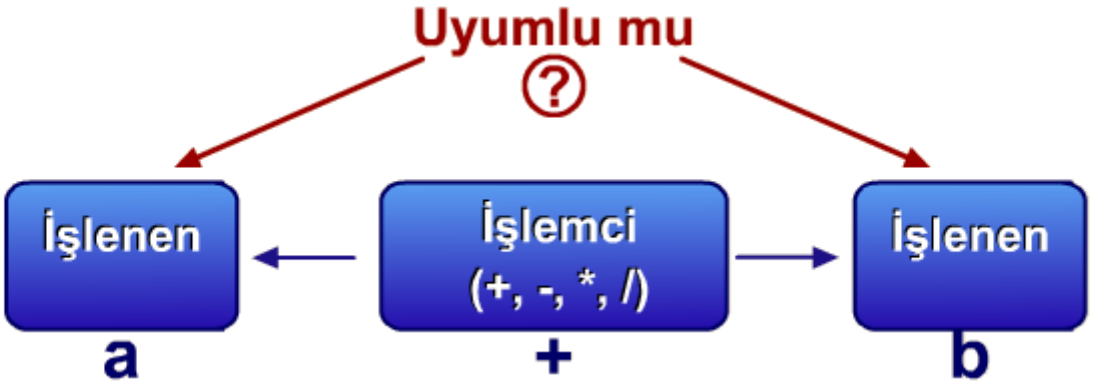
- Kuvvetli tiplemenin sağlandığı bir programlama dilinde derleyici, her değişken ve her ifadenin tipinin belirlenebilmesi için kurallar içerir. Bu kurala göre, kuvvetli tipleme, eş olmayan tipleri birbirlerine atamaya ve yordam çağırımlarında gerçek-resmi parametre bağlamalarına izin vermez. Az sayıdaki zorunlu dönüşümler (coercion), istisnai durumlar olarak nitelendirilir.

6.4. KUVVETLİ TİPLEME

- Popüler programlama dillerinin kuvvetli tipleme açısından özellikleri aşağıda özetlenmiştir:
  - C ve C++, kuvvetli tiplemeyi gerçekleştirmemektedirler.
  - Programlama dilleri alanına, kuvvetli tiplemeyi tanıtan dil Pascal'dır. Pascal, kuvvetli tiplemeyi sağlamaya yakın bir dil olmakla birlikte, tam olarak kuvvetli tipli bir dil olarak kabul edilmemektedir.

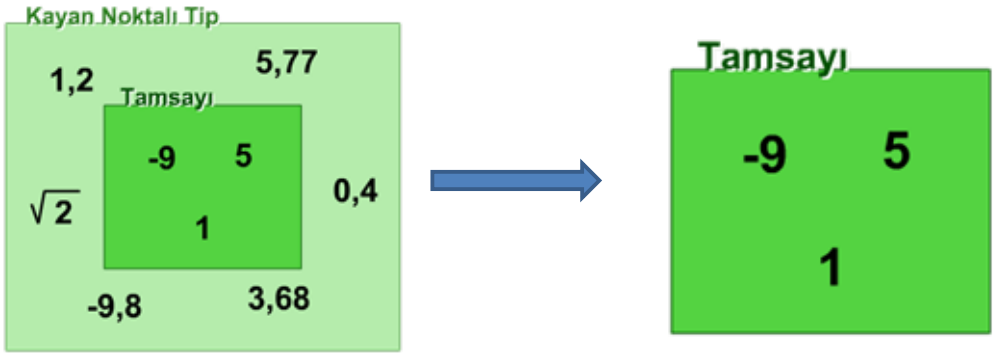
6.5. TİP DENETİMİ

- Bir işlemcinin işlenenlerinin birbirleriyle uyumlu tipler olduğunun denetlenmesi, tip denetimi olarak adlandırılmaktadır.
- Bir programlama dilinin güvenilirliğini önemli derecede etkileyen tip denetimi, dildeki tüm değişkenler için durağan tip bağlama uygulanıyorsa, durağan olarak gerçekleştirilebilir. Eğer programlama dilinde dinamik tip bağlama gerçekleştiriliyorsa, tip denetimi çalışma zamanında yapılmalıdır..



6.5.1. Tip Dönüşümleri

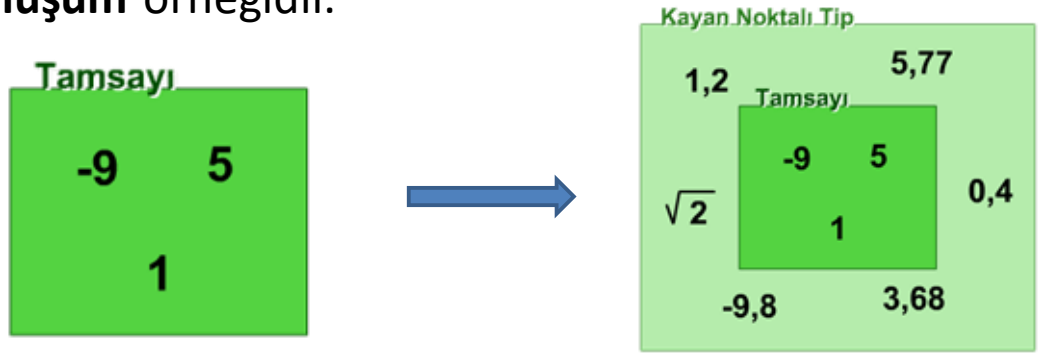
- Bir işlemcinin farklı tiplerde işlenenlerinin olabildiği sayısal ifadeler, **karışık tipli ifadeler** (*mixed mode expression*) olarak nitelendirilir.
- Farklı tiplerdeki işlenenler için ayrı işlemcilerin bulunmadığı ve karışık tipli ifadelerin olası olduğu dillerde, **tip dönüşümleri** gereklidir.
- Tip dönüşümleri daralan veya genişleyen türde olabilir.
- Eğer bir nesne, kendi tipindeki tüm değerleri içermeyen bir tipe dönüştürülüyorsa **daralan dönüşüm** gerçekleşmektedir. Örneğin, kayan noktalı tipten tamsayıya dönüşüm, daralan dönüşümdür.



Çoğu programlama dili daralan dönüşümlere izin vermezler

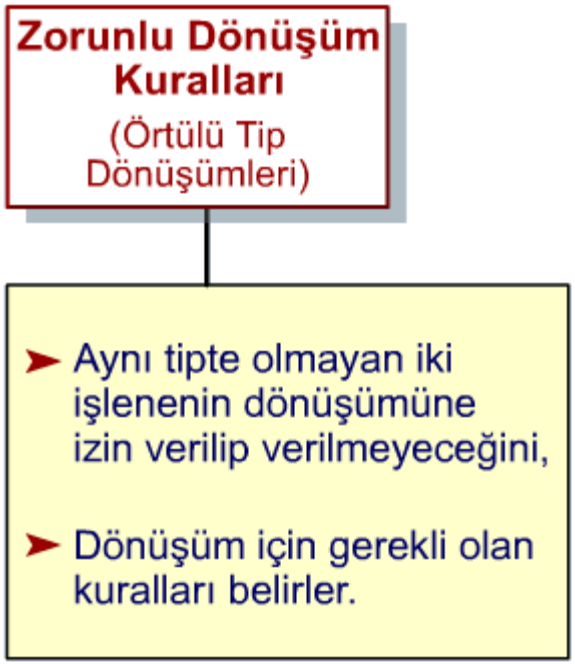
6.5.1. Tip Dönüşümleri

- Öte yandan, bir nesnenin kendi tipinin tüm değerlerini içeren bir tipe dönüşümü, genişleyen dönüşüm olmaktadır.
- Bir tamsayı değişkenin kayan noktalı tipe dönüşümü, **genişleyen dönüşüm** örneğidir.



- Görüldüğü gibi, genişleyen dönüşümler güvenli iken, daralan dönüşümler hatalara neden olabilirler. Bu nedenle çoğu programlama dili daralan dönüşümlere izin vermezler.
- Bir programlama dilinin tip dönüşüm kuralları, dildeki tip denetimini etkileyen önemli bir etmendir. Tip dönüşümleri, **dışsal** (*explicit*) veya **örtülü** (*implicit*) olarak iki şekilde gerçekleştirilebilir.

6.5.1.1. Örtülü Tip Dönüşümleri



- Örtülü tip dönüşümleri, derleyici tarafından gerçekleştirilir ve **zorunlu dönüşüm** (*coercion*) olarak adlandırılır.
- Zorunlu dönüşüm kuralları, dil gerçekleştirim zamanında (derleyici tasarımı zamanında) belirlenir. Bu kurallar, bir işlemcinin iki işleneni aynı tipte değilse, dönüşüme izin verilip verilmeyeceğini ve bir dönüşüm gerçekleşecekse bunun için gerekli olan kuralları belirlerler. Bu amaçla, derleyici dönüşüm için bir işleneni taban seçmeli ve diğer işlenen üzerinde gerekli dönüşümü yapmalıdır.

6.5.1.1. Örtülü Tip Dönüşümleri

- Çoğu popüler yüksek seviyeli programlama dilinde, tipler arasındaki zorunlu dönüşüm genişleyen dönüşüm olarak gerçekleşir. Örneğin Pascal ve C gibi birçok programlama dilinde bir tamsayı ve bir gerçel sayı toplanmak istendiğinde, tamsayının değeri gerçel sayıya dönüştürülür ve işlem gerçel sayılar üzerinde yapılır.
- Bu durum çoğunlukla programcı tarafından da istenen bir dönüşüm olmasına karşın, hataların fark edilmesini engelleyebilir. Bu nedenle zorunlu dönüşümün gerçekleştiği dillerde, tip denetimi kısıtlandığı için güvenilirlik azalmaktadır. Zorunlu dönüşümün en az gerçekleştiği dillerden birisi Ada'dır.

6.5.1.1. Örtülü Tip Dönüşümleri

- Zorunlu dönüşümün neden olabileceği bir hatayı aşağıdaki şekilde verilen C örneğinde inceleyelim.

- Eğer dilde zorunlu dönüşüm olmasa ve karışık tipli ifadelere dilde izin verilmeseydi, bu yazım hatası bir tip uyuşmazlığı olarak derleyici tarafından yakalanacaktı.

C (hata vermez)

```
int a;
float b,c,d;

…
d=b*a;
```

Ada (hata verir)

```
a: Integer;
b,c,d: Float;

…
d:=b*a;  //bu satırda "c"
         // yanlışlıkla "a" yazılmış.
```

- Güvenilirlik olumsuz etkileniyor.
- Hataların fark edilmesini engelleyebilir.
- Zorunlu dönüşümün gerçekleştiği dillerde, tip denetimi kısıtlanmakta

Aşağıdaki programda ***a=b*d*** *ifadesindeki d'nin yanlışlıkla* yazıldığını düşünelim

```
void main(){
int a,b,c;
float d;
a=b*d;
}
```

Derleyici, *b* yi *float* tipine dönüştürecek ve çarpım işlemi *float* olarak gerçekleşecektir. Zorunlu dönüşüm, tip uyuşmazlığı hatasını engellemiştir.

6.5.1.2. Dışsal Tip Dönüşümleri

- Bazı dillerde örtülü olarak gerçekleşen tip dönüşümlerine ek olarak, hem daralan hem de genişleyen özellikte dışsal tip dönüşümleri gerçekleşebilir. C'de dışsal tip dönüşümlerine ***cast*** adı verilir. *Cast* işleminde, işlenenden önce parantez içinde değerin dönüştürülmesi istenen tip yazılır.

Örnek

```
sonuc = (int) toplam + 99
```

toplam değişkenin değeri önce int tipine dönüştürülür.

Daha sonra 99 ile toplanarak sonuc değişkenine atanır.

```
#include <stdio.h>
void main()
{
 double x;
 x=4/3;
 printf("sonuç =%f\n",x);
}
```

```
#include <stdio.h>
void main()
{
 double x;
 x=4.0/3;
 printf("sonuç =%f \n",x);          // sonuç = 1.333333
 x=(double) 4/3;
 printf("sonuç =%f \n",x);          // sonuç = 1.333333
 x=(double) (4/3);
 printf("sonuç =%f \n",x);          // sonuç = 1.000000
}
```

Burada x değişkeni double olarak tanımlanmıştır. Double veri tipi kesirli ve çok büyük sayıları tutma özelliğine sahiptir. x=4/3; işlemi ile 4 sayısı 3'e bölünmekte ve sonuç x değişkenine aktarılmaktadır ve printf komutu ile ekrana sonuç =1.000000 olarak yazdırılmaktadır. Bunun sebebi 4 ve 3 sabit bilgilerinin int veri tipinde birer bilgi olmasındandır.

Çoğu programlama dili daralan dönüşümlere izin vermez

Ödev

- 1- İlkel ve Yapısal Veri Tiplerinin imperative, fonksiyonel, mantıksal ve nesneye yönelik diller için kullanımlarına birer örnek program araştırınız. (Birçok özellik aynı program içinde verilecek ise satırlar arasında tanımlamalar belirtilecek)
- 2- Mevcut diller hangi veri türlerini destekliyor/desteklemiyor araştırınız.
- 3. Java dili için kullanılan veri tipleri ile ilgili bir program yazınız.
  - Örnek program: Öğrencilerin kimlik ve not bilgilerinin girilip hesaplanması.

Özet

- Bu bölümde programlama dilinin kullanışlılığını belirleyen en büyük parçası olan ve dillerde çok önemli bir yer tutan veri tipi kavramı ve çeşitli veri tipleri incelenmiştir.
- Bu kapsamda; İlkel ve Yapısal Veri Tipi kavramları açıklanmıştır.
- İlkel veri tiplerinde, Sayısal, Mantıksal, Karakter, Karakter String ve Kullanıcı Tanımlı Sıralı Tipler; yapısal veri tiplerinde ise Diziler, Record (Kayıt) Tipi, Union (Bileşim) Tipi, Set (Küme)Tipi ile Pointer (Gösterge) Tipi anlatılmıştır.
- Ayrıca; Kuvvetli Tipleme, Tip Denetimi ve Tip Dönüşümleri kavramları incelenmiştir.

Kaynaklar

- Roberto Sebesta, Concepts Of Programming Languages, International 10th Edition 2013
- David Watt, Programming Language Design Concepts, 2004
- Michael Scott, Programming Languages Pragmatics, Third Edition, 2009
- Zeynep Orhan, Programlama Dilleri Ders Notları
- Mustafa Şahin, Programlama Dilleri Ders Notları
- Ahmet Yesevi Üniversitesi, Programlama Dilleri Uzaktan Eğitim Notları
- Erkan Tanyıldızı, Programlama Dilleri Ders Notları
- Tuğrul Yılmaz, Programlama Dilleri Ders Notları
- David Evans, Programming Languages Lecture Notes
- Oscar Nierstrasz, Programming Languages Lecture Notes