CENG 218 Programlama Dilleri

Bölüm 7: Temel Semantik

Öğr.Gör. Şevket Umut Çakır

Pamukkale Üniversitesi

Hafta 10

Hedefler

- Öznitelikleri, bağlamayı ve semantik fonksiyonları anlamak
- Bildirimleri, blokları ve kapsamı anlamak
- Bir sembol tablosunun nasıl oluşturulacağını öğrenmek
- Ad çözümlemesini ve aşırı yüklemeyi anlamak
- Tahsisi, yaşam sürelerini ve ortamı anlamak
- Değişkenler ve sabitlerle çalışmak
- Takma adları, sarkan referansları ve çöpleri nasıl idare edeceğinizi öğrenmek
- TinyAda'nın ilk statik semantik analizini gerçekleştirmek

Giriş

- Sözdizimi(Syntax): dil yapıları neye benziyor
- Semantik: dil yapılarının gerçekte ne yaptığı
- Semantik belirtmek sözdizimini belirtmekten daha zordur
- Sematik belirtmenin birkaç yolu vardır:
  - Dil başvuru kılavuzu(Language reference manual)
  - Bir çevirmen tanımlama
  - Resmi tanımlama

Giriş

- Dil başvuru kılavuzu:
  - Semantik belirtmenin en yaygın yolu
  - Daha net ve daha kesin başvuru kılavuzları sağlar
  - Doğal dil tanımlarının doğasında bulunan hassasiyet eksikliğinden muzdariptir
  - Eksiklikler ve belirsizlikler olabilir

Giriş

- Bir çevirmen tanımlama:
  - Bir dil hakkındaki sorular deneysel olarak cevaplanabilir.
  - Program davranışıyla ilgili sorular önceden yanıtlanamaz
  - Tercümandaki hatalar ve makine bağımlılıkları, muhtemelen kasıtsız olarak dil semantiğinin bir parçası haline gelebilir
  - Tüm makinelere taşınabilir olmayabilir
  - Genel olarak mevcut(elde edilebilir) olmayabilir

Giriş

- Resmi tanımlama:
  - Biçimsel matematiksel yöntemler: kesin, ancak aynı zamanda karmaşık ve soyuttur
  - Anlamak için çalışma gerektirir
  - Tanımsal semantik(Denotational semantics): programların çevrilmesi ve yürütülmesi için muhtemelen en iyi biçimsel yöntem
    - Bir dizi işlevi kullanarak semantiği açıklar
- Bu kurs, gösterimsel tanımlamalarda kullanılan basitleştirilmiş fonksiyonlarla birlikte gayri resmi tanımlamanın bir melezini kullanacaktır

Öznitelikler, Bağlama ve Anlamsal Fonksiyonlar

- **İsimler(Names)** (veya **tanımlayıcılar(identifiers)**): dil varlıklarını veya yapılarını belirtmek için kullanılan temel bir soyutlama mekanizması
- Sematiğin tanımlanmasındaki temel adım, tanımlayıcılar için adlandırma kurallarını tanımlamaktır.
- Çoğu dil ayrıca konum ve değer kavramlarını da içerir
  - **Değer(Value)**: herhangi bir depolanabilir miktar
  - **Konum(Location)**: değerin saklanabileceği yer; genellikle göreceli bir konum

Öznitelikler, Bağlama ve Anlamsal Fonksiyonlar

- **Öznitelikler(Attributes)**: ilişkili oldukları adın anlamını belirleyen özellikler
- C örneği: `const int n = 5;`
  - Değişkenler ve sabitler için öznitelikler, veri türünü ve değerini içerir
- C örneği:

```
double f(int n){
//...
}
```

  - Öznitelikler arasında "fonksiyon", sayı, parametrelerin adları ve veri türü, dönüş değeri veri türü, yürütülecek kod gövdesi bulunur

Öznitelikler, Bağlama ve Anlamsal Fonksiyonlar

- Atama ifadeleri, öznitelikleri adlarla ilişkilendirir
- Örnek: `x = 2;`
  - "2 değeri" özelliğini x değişkeniyle ilişkilendirir
- C++ Örneği:

```
int * y;
y = new int;
```

  - Bellek ayırır (konumu y ile ilişkilendirir)
  - Değerle ilişkilendirir

Öznitelikler, Bağlama ve Anlamsal Fonksiyonlar

- **Bağlama(Binding)**: bir özniteliği bir adla ilişkilendirme işlemi
- **Bağlanma zamanı(Binding time)**: bir özniteliğin hesaplandığı ve bir isme bağlandığı zaman
- İki bağlama kategorisi:
  - **Statik bağlama(Static binding)**: yürütmeden önce gerçekleşir
  - **Dinamik bağlama(Dynamic binding)**: yürütme sırasında gerçekleşir
- Statik öznitelik: statik olarak bağlanan bir öznitelik
- Dinamik öznitelik: dinamik olarak bağlanan bir öznitelik

Öznitelikler, Bağlama ve Anlamsal Fonksiyonlar

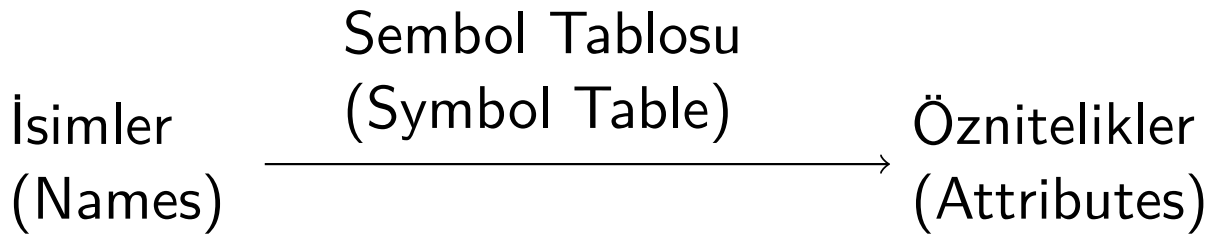
- Diller, özniteliklerin statik veya dinamik olarak bağlanma şekline göre büyük ölçüde farklılık gösterir
  - Fonksiyonel diller, zorunlu dillerden daha dinamik bağlara sahip olma eğilimindedir
- Statik öznitelikler çeviri(translation) sırasında, bağlanma(linking) sırasında veya programın yüklenmesi(loading) sırasında bağlanabilir
- Dinamik özellikler, bir prosedürden veya programdan giriş veya çıkış gibi yürütme sırasında farklı zamanlarda bağlanabilir

Öznitelikler, Bağlama ve Anlamsal Fonksiyonlar

- Bazı öznitelikler çeviri zamanından önce bağlanır
  - Önceden tanımlanmış tanımlayıcılar(Predefined identifiers): dil tanımıyla belirtilir
  - Boolean veri türüne bağlı doğru / yanlış değerler
  - Dil tanımı ve uygulaması ile belirtilen `maxint`
- Yürütme zamanı dışındaki tüm bağlama zamanları statik bağlamadır

Öznitelikler, Bağlama ve Anlamsal Fonksiyonlar

- Bir çevirmen, bağlamaları tutmak için bir veri yapısı oluşturur
  - Özniteliklerin isimlere bağlanmasını ifade eden bir işlev olarak düşünülebilir
- **Sembol tablosu(Symbol table)**: isimlerden özniteliklere bir fonksiyon

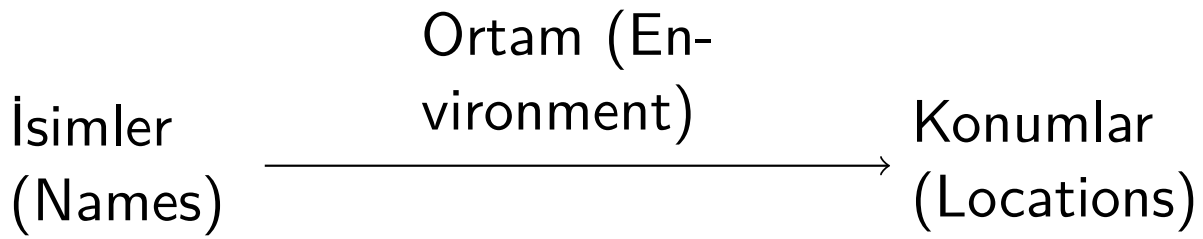


Şekil: İsimleri, özniteliklere sembol tablosu ile iz düşürme

Öznitelikler, Bağlama ve Anlamsal Fonksiyonlar

- Çevirinin ayrıştırma aşaması üç tür analiz içerir:
  - Sözcüksel analiz(Lexical analyses): bir karakter dizisinin bir sembolü(token) temsil edip etmediğini belirler
  - Sözdizimi analizi(Syntax analysis): bir sembol dizisinin bağlamdan bağımsız dilbilgisinde bir ifadeyi temsil edip etmediğini belirler
  - Statik anlam analizi(Static semantic analysis): bildirimlerdeki isimlerin özniteliklerini belirler ve bu isimlerin kullanımının bildirilen özniteliklerine uygun olmasını sağlar
- Yürütme sırasında öznitelikler de korunur

Öznitelikler, Bağlama ve Anlamsal Fonksiyonlar



Şekil: İsimleri, konumlara ortam ile iz düşürme

Şekil: Konumları, değerlere bellek ile iz düşürme

Bildirim, Blok ve Kapsam

- Bağlamalar **örtülü(implicit)** veya **açık(explicit)** olabilir
- Örnek: `int x;`
  - Veri türü açıkça bağlıdır; x'in konumu örtülü olarak bağlıdır
- Tüm bildirimin kendisi, yalnızca değişken adının kullanılmasının bildirilmesine neden olduğu dillerde örtülü olabilir
- **Tanım(Definition)**: C ve C ++ 'da, tüm potansiyel öznitelikleri bağlayan bir bildirim
- **Prototip(Prototype)**: veri türünü belirten ancak onu gerçekleştirecek kodu belirtmeyen fonksiyon bildirimi

Bildirim, Blok ve Kapsam

- **Blok(Block)**: bir dizi bildirim ve ardından bir dizi ifade
- **Bileşik ifadeler(Compund statements)**: C'deki işlevlerin gövdesi olarak veya sıradan bir program ifadesinin görünebileceği herhangi bir yerde görünen bloklar
- **Yerel bildirimler(Local declarations)**: bir blokla ilişkili
- **Yerel olmayan bildirimler(Nonlocal declarations)**: çevreleyen bloklarla ilişkili
- Blok yapılı diller, blokların yuvalanmasına ve iç içe bloklar içinde adların yeniden bildirilmesine izin verir

Bildirim, Blok ve Kapsam

- Bildirilen her isim, bir **seviye numarası(level number)** ve bir **ofset/öteleme(offset)** içeren bir **sözcük adresine(lexical address)** sahiptir.
  - Seviye numarası 0'dan başlar ve her iç içe geçmiş bloğa doğru artar

  Diğer bildirim kaynakları şunları içerir:
  - Yerel(üye) bildirimlerden oluşan bir `struct` tanımı
  - Nesne yönelimli dillerde bir sınıf
- Bildirimler paketler(Ada), modüller(ML, Haskell, Python) ve ad alanları(namespace)(C ++) olarak toplanabilir

Bildirim, Blok ve Kapsam

- **Bağlamanın kapsamı(Scope of binding)**: Bağlamanın sürdürüldüğü program bölgesi
- **Sözcük kapsamı(Lexical scope)**: Blok yapılı dillerde kapsam, ilgili bildiriminin göründüğü blokla (ve içinde bulunan diğer bloklarla) sınırlıdır.
- **Kullanım öncesi bildirim kuralı(Declaration before use rule)**: C'de bir bildirimin kapsamı, bildirim noktasından içinde bulunduğu bloğun sonuna kadar uzanır.

Bildirim, Blok ve Kapsam

```
int x;
void p(){
    char y;
    //...
} /* p */
void q(){
    double z;
    //...
} /* q */
main(){
    int w[10];
    //...
}
```

Bildirim, Blok ve Kapsam

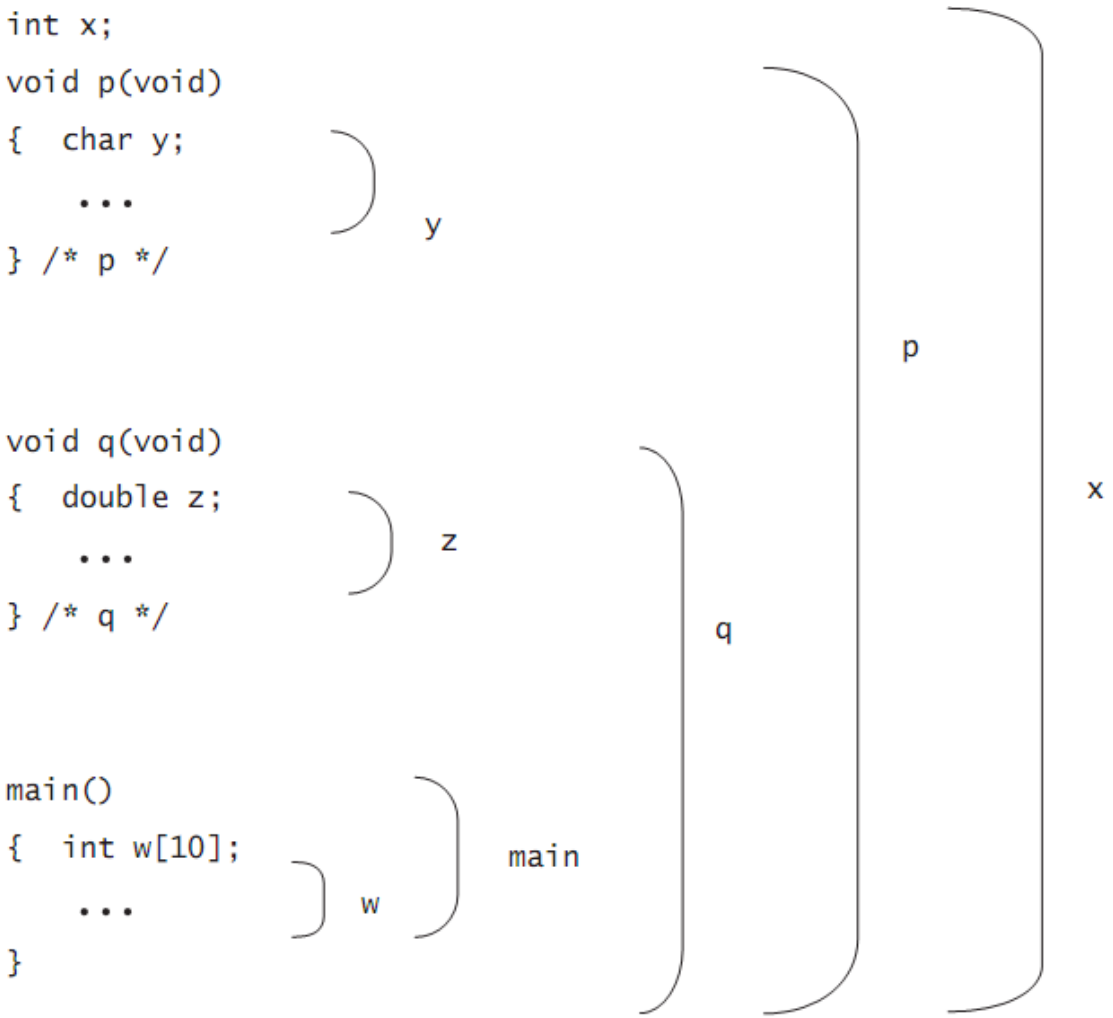


Figure 7.5 C Program from Figure 7.4 with brackets indicating scope

Bildirim, Blok ve Kapsam

- İç içe geçmiş bloklardaki bildirimler önceki bildirimlere göre önceliklidir
- Global bir değişkenin, aynı ada sahip yerel bir bildirimi içeren bir blokta bir **kapsam deliğine(scope hole)** sahip olduğu söylenir
  - Global değişkene erişmek için C++ 'da **kapsam çözümleme operatörü(scope resolution operator)** : : kullanılır
- Yerel bildirimin global bildirimi **gölgelediği(shadow)** söyleniyor
- **Görünürlük(Visibility)**: yalnızca bir bildirimin bağlamalarının geçerli olduğu bölgeleri içerir

Bildirim, Blok ve Kapsam

```
int x;
void p(){
    char x;
    x = 'a'; //char x'e atama yapar
    ::x = 42; //global x'e atama yapar
}
main(){
    x = 2; //global x'e atama yapar
    //...
}
```

Bildirim, Blok ve Kapsam

- Kapsam kurallarının, anlamlı olduklarında yinelemeli (kendine referanslı) bildirimlerin mümkün olacağı şekilde yapılandırılması gerekir.
  - Örnek: fonksiyonların özyinelemeli olmasına izin verilmelidir, bu nedenle fonksiyon adı, fonksiyon gövdesi bloğundan önce başlayan kapsamlara sahip olmalıdır.

```
int factorial(int n){
    /*
        factorial kapsamı burada başlar
        factorial buradan çağrılabilir
    */
}
```

Sembol Tablosu

- Sembol tablosu:
  - Bildirimlerdeki bağlamaları temsil eden, ilişkili özniteliklere sahip adların eklenmesini, aranmasını ve silinmesini desteklemelidir
- Sözcük kapsamlı(lexically scoped), blok yapılı bir dil, **kapsam analizi(scope analysis)** gerçekleştirmek için yığın benzeri bir veri yapısı gerektirir:
  - Blok girişinde, bu bloğun tüm bildirimleri işlenir ve sembol tablosuna bağlamalar eklenir.
  - Blok çıkışında bağlamalar kaldırılır ve var olan önceki bağlamalar geri yüklenir.

Sembol Tablosu

```
1  int x;
2  char y;
3  void p(){
4      double x;
5      //...
6      {   int y[10];
7          //...
8      }
9      //...
10 }
11 void q(){
12     int y;
13     //...
14 }
15 main(){
16     char x;
17     //...
18 }
```

Sembol Tablosu

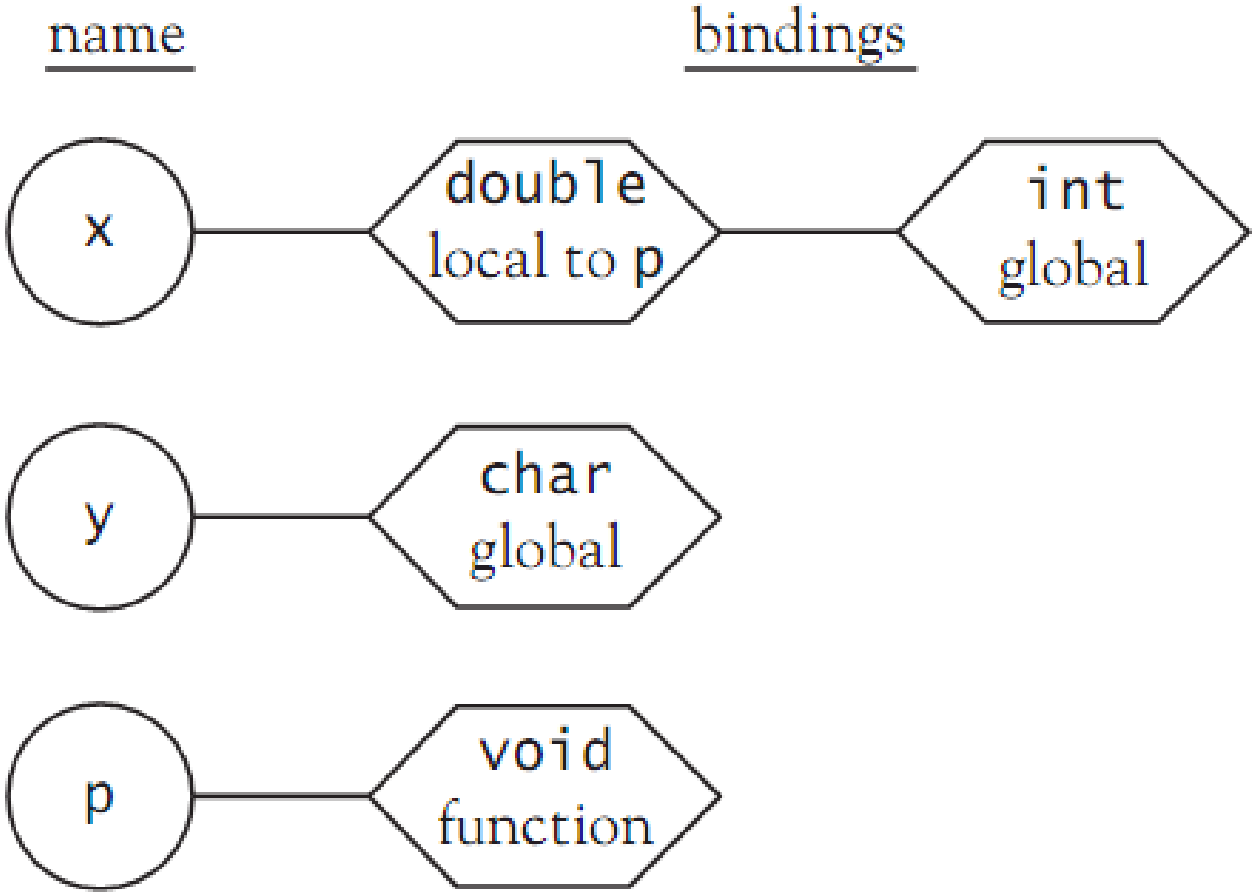


Figure 7.7 Symbol table structure at line 5 of Figure 7.6

Sembol Tablosu

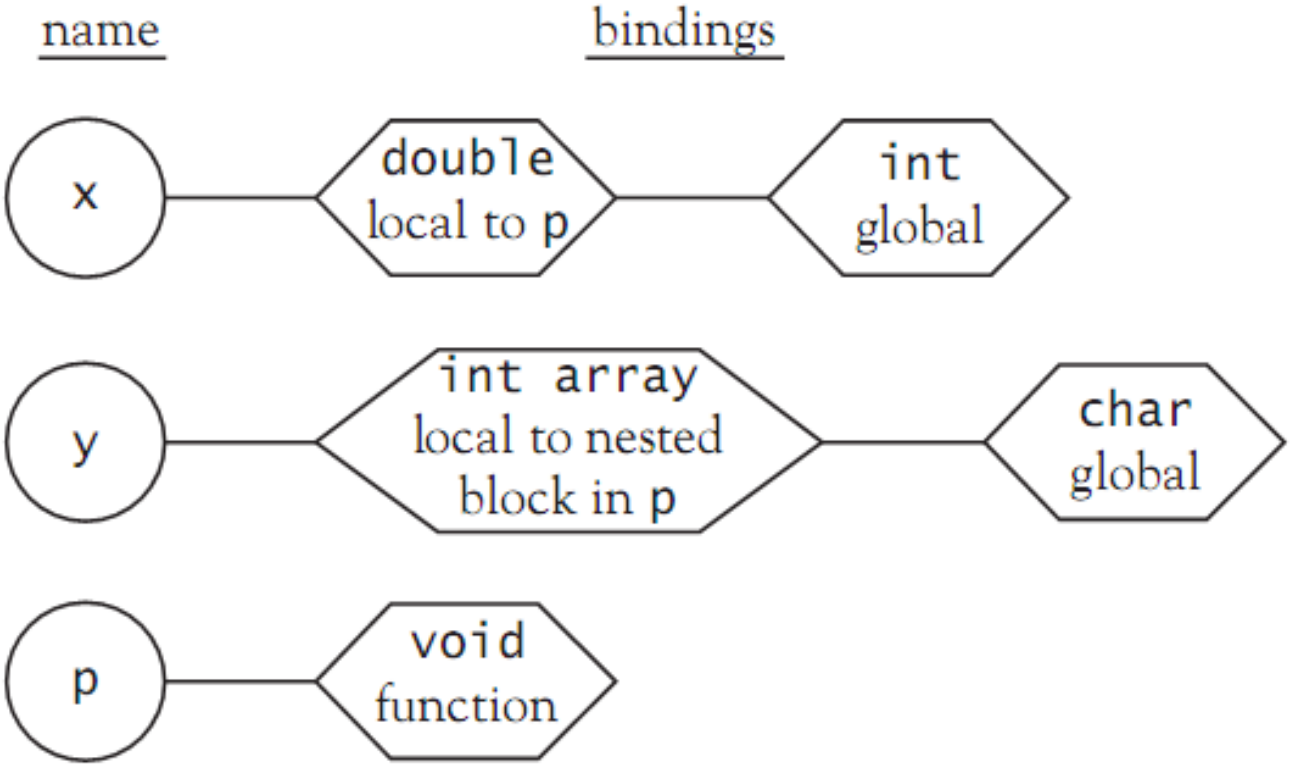


Figure 7.8 Symbol table structure at line 7 of Figure 7.6

Sembol Tablosu

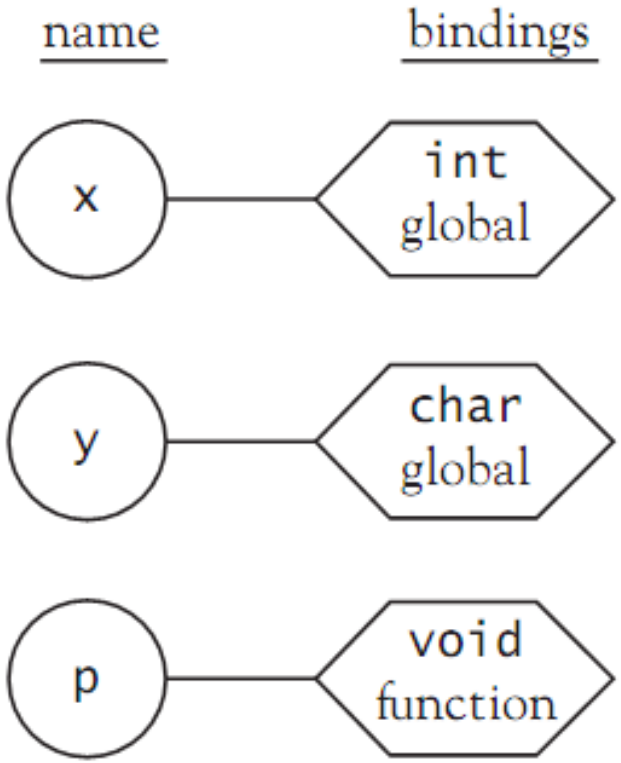


Figure 7.9 Symbol table structure at line 10 of Figure 7.6

Sembol Tablosu

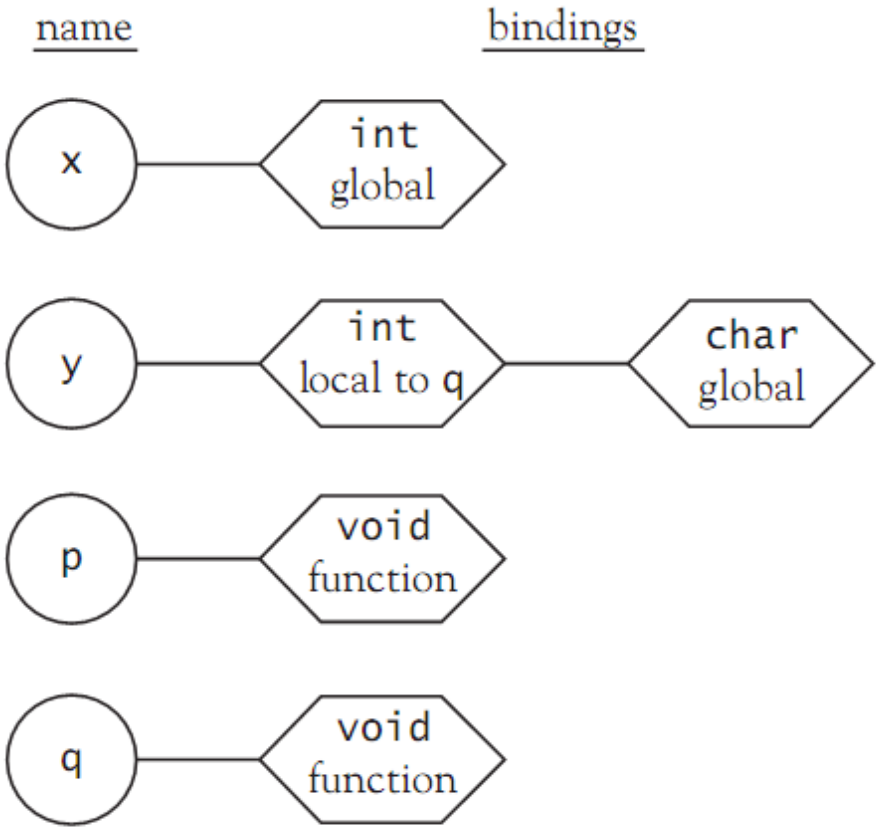


Figure 7.10 Symbol table structure at line 13 of Figure 7.6

Sembol Tablosu

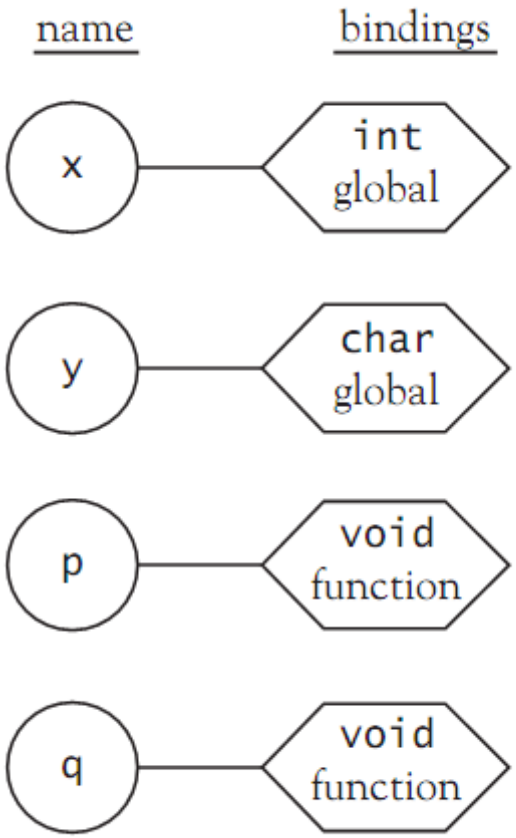


Figure 7.11 Symbol table structure at line 14 of Figure 7.6

Sembol Tablosu

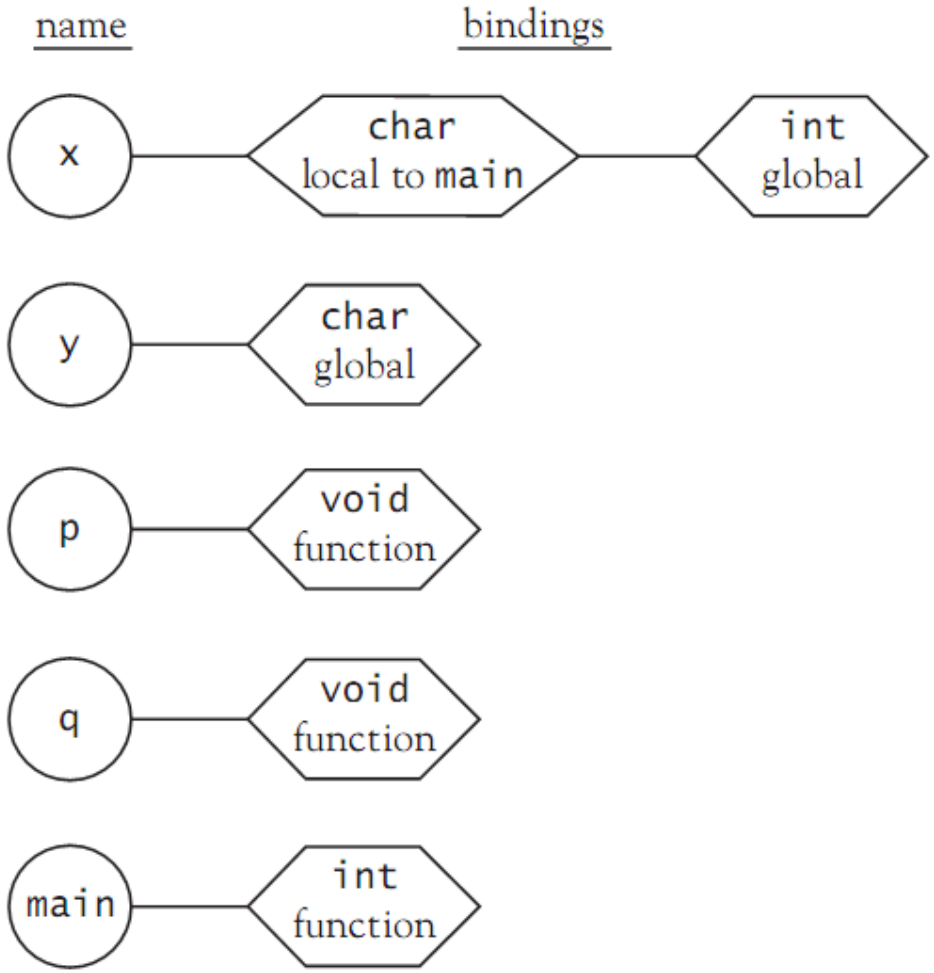


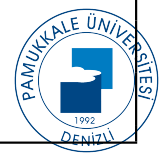
Figure 7.12 Symbol table structure at line 17 of Figure 7.6

Sembol Tablosu

- Önceki örnek, bildirimlerin statik olarak işlendiğini varsayar (yürütmeden önce)
  - Buna **statik kapsam(static scoping)** veya **sözcüksel kapsam(lexical scoping)** denir
  - Sembol tablosu bir derleyici tarafından yönetilir
  - Bildirimlerin bağlamaları statiktir
- Sembol tablosu dinamik olarak yönetiliyorsa(yürütme sırasında), bildirimler bir yürütme yolunda karşılaşıldıkça işlenir
  - Buna **dinamik kapsam(dynamic scoping)** belirleme denir

Sembol Tablosu

```c
#include<stdio.h>
int x=1;
char y='a';
void p(){
    double x=2.5;
    printf("%c\n", y);
    {
        int y[10];
    }
}
void q(){
    int y=42;
    printf("%d\n", x);
    p();
}
main(){
    char x = 'b';
    q();
    return 0;
}
```



Sembol Tablosu

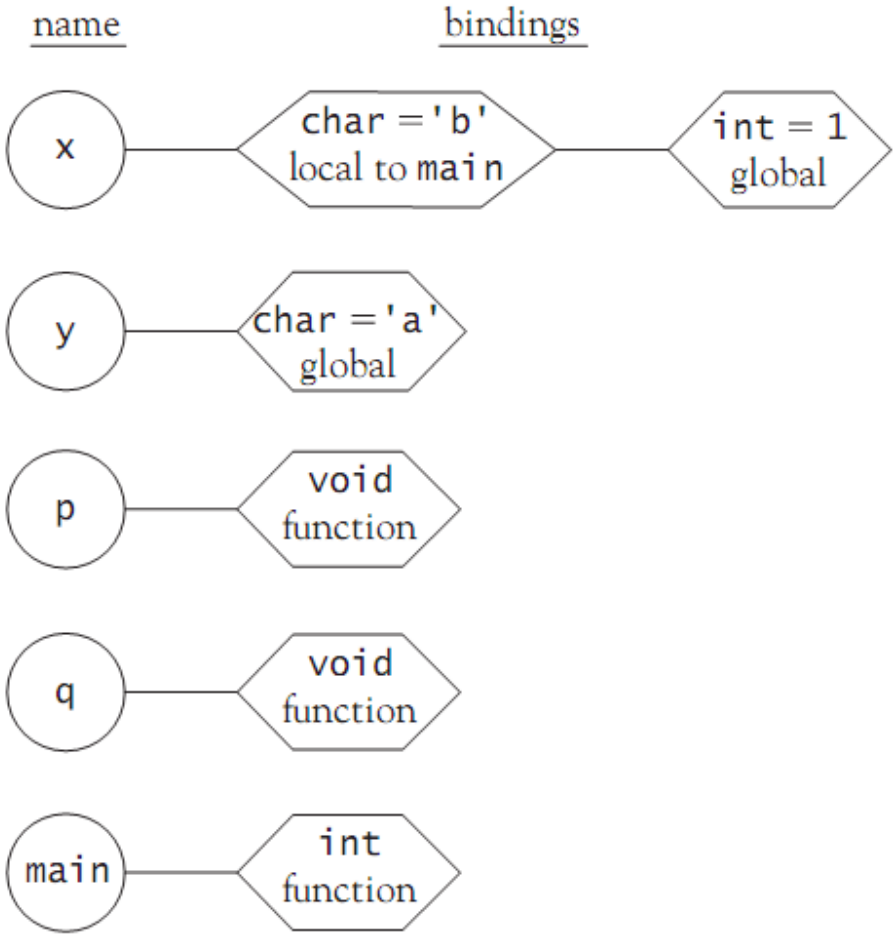


Figure 7.14 Symbol table structure at line 17 of Figure 7.13 using dynamic scope

Sembol Tablosu

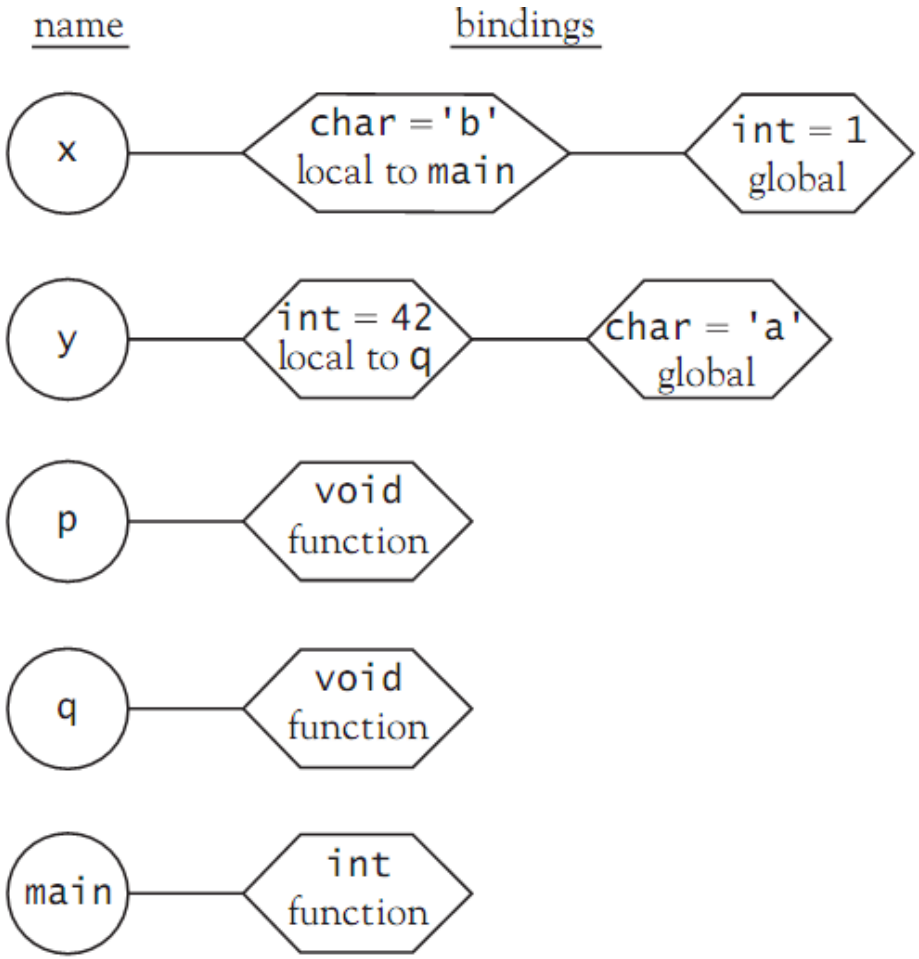


Figure 7.15 Symbol table structure at line 12 of Figure 7.13 using dynamic scope

Sembol Tablosu

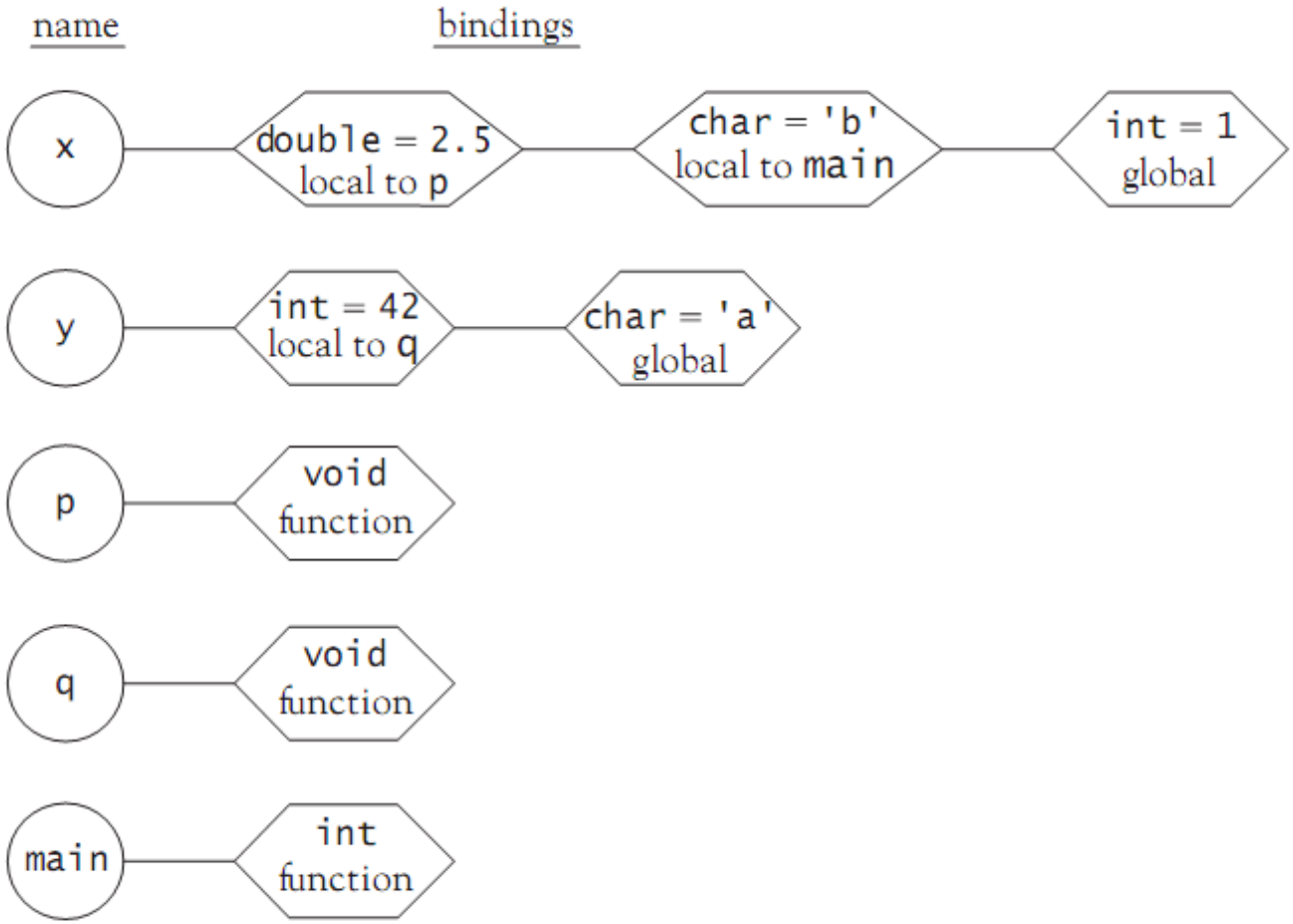


Figure 7.16 Symbol table structure at line 6 of Figure 7.13 using dynamic scope

Sembol Tablosu

- Dinamik kapsam, programın anlamını etkileyecek ve farklı çıktılar üretecektir.
- Sözcüksel kapsam kullanarak çıktı:

  1
  a
- Dinamik kapsam kullanarak çıktı:

  98
  *
- Dinamik kapsam sorunlu olabilir, bu yüzden az sayıda dil kullanır

Sembol Tablosu

- Dinamik kapsam belirleme ile ilgili sorunlar:
  - Yerel olmayan bir adın bildirimi, sadece programı okuyarak belirlenemez: yürütme yolunu bilmek için program yürütülmelidir.
  - Yerel olmayan değişken referansları yürütmeden önce tahmin edilemediğinden, veri türleri de tahmin edilemez
- Dinamik kapsam belirleme, programların çok büyük olması beklenmediğinde yüksek dinamik, yorumlamalı diller için olası bir seçenektir

Sembol Tablosu

- Bir yorumlayıcıda dinamik kapsam belirleme ile çalışma zamanı ortamı daha basittir
  - APL, Snobol, Perl ve Lisp'in erken lehçeleri dinamik olarak kapsama alındı
  - Scheme ve Common Lisp statik kapsam kullanır
- Sembol tabloları için ek karmaşıklık var
- `struct` bildirimi, içindeki veri alanlarının başka bildirimlerini içermelidir
  - Bu alanlara, `struct` değişkeni kapsam dahilinde olduğunda nokta üye gösterimi kullanılarak erişilebilir olmalıdır.

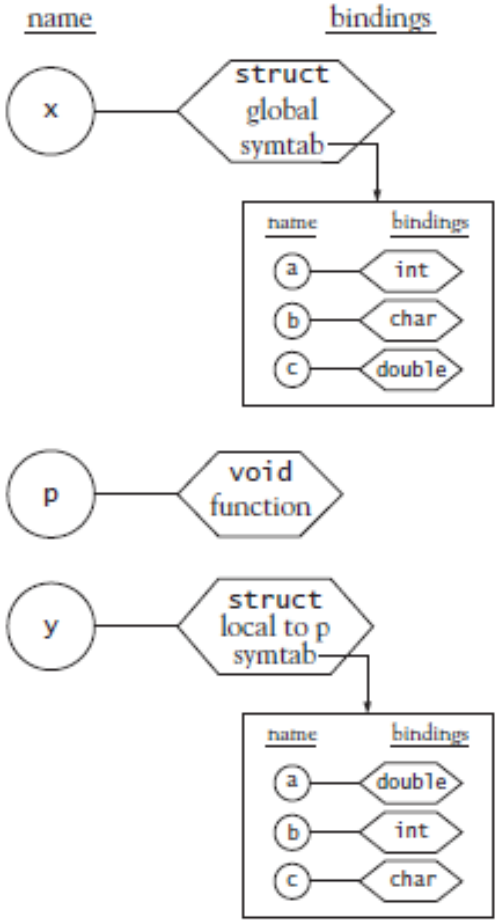
Sembol Tablosu

- `struct` değişkenleri için iki çıkarım:
  - Bir struct bildirimi aslında bir öznitelik olarak yerel bir sembol tablosunun kendisini içerir
  - Bu yerel sembol tablosu, `struct` değişkeninin kendisi programın global sembol tablosundan silinene kadar silinemez.

Sembol Tablosu

```
struct {
    int a;
    char b;
    double c;
} x = {1, 'a', 2.5};
void p(){
    struct {
        double a;
        int b;
        char c;
    } y = {1.2, 2, 'b'};
    printf("%d, %c, %g\n", x.a, x.b, x.c);
    printf("%f, %d, %c\n", y.a, y.b, y.c);
}
main(){
    p();
    return 0;
}
```

Sembol Tablosu



Şekil: 12. satırdaki sembol tablosu temsili

Sembol Tablosu

- Doğrudan referans verilebilecek herhangi bir kapsam belirleme yapısının kendi sembol tablosu da olmalıdır.
- Örnekler:
  - Ada'da isimli kapsamlar
  - C++ 'da sınıflar, yapılar ve ad alanları
  - Java'daki sınıflar ve paketler
- Tipik olarak, bir sembol tablosu yığınında her kapsam için bir tablo olacaktır.
  - Bir isme referans oluştuğunda, mevcut tabloda bir arama başlar ve bulunamazsa sonraki tabloya devam eder ve böyle devam eder

Sembol Tablosu

```
1  with Tetx_IO; use Text_IO;
2  with Ada.Integer_Text_IO; use
   ↪  Ada.Integer_Text_IO;
3  procedure ex is
4      x: integer := 1;
5      y: character := 'a';
6      procedure p is
7          x: float := 2.5;
8      begin
9          put(y); new_line;
10         A: declare
11             y: array (1..10) of integer;
12         begin
13             y(1) := 2;
14             put(y(1)); new_line;
15             put(ex.y); new_line;
16         end A;
17     end p;
18     procedure q is
19         y: integer := 42;
20     begin
21         put(x); new_line;
22         p;
23     end q;
24     begin
25         declare
26             x: character := 'b';
27         begin
28             q;
29             put(ex.x); new_line;
30         end;
31 end ex;
```

Sembol Tablosu

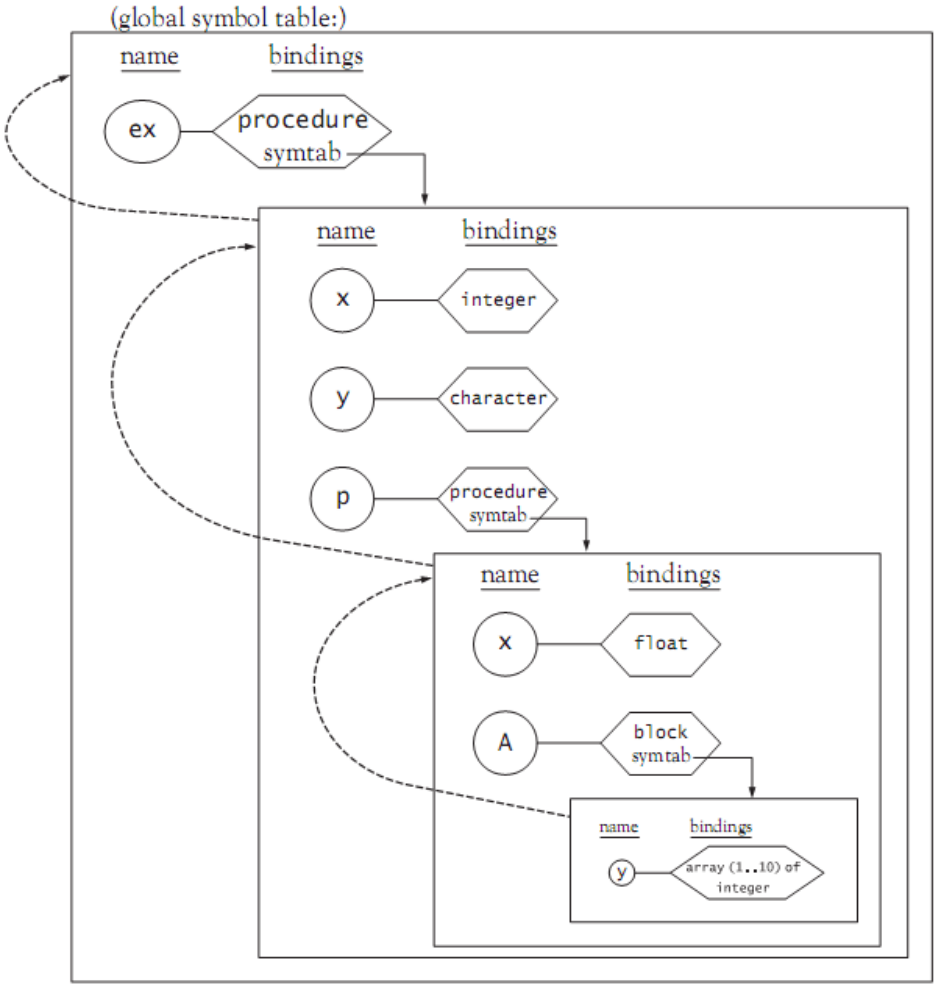


Figure 7.20 Symbol table structure at line 12 of Figure 7.19

Ad Çözümlemesi ve Aşırı Yükleme

- Toplama operatörü + aslında en az iki farklı işlemi gösterir: tamsayı toplama ve kayan nokta toplama
  - + operatörün aşırı yüklenmiş olduğu söyleniyor
- Çevirmen, hangi işlemin gösterildiğini belirlemek için her işlenenin veri türüne bakmalıdır.
- **Aşırı yükleme çözümü(Overload resolution)**: aynı ada sahip birçok fonksiyon arasından benzersiz bir fonksiyon seçme işlemi
  - Bir sembol tablosunun arama işlemi, isim artı parametrelerin sayısı ve veri tipine göre aramalıdır.

Ad Çözümlemesi ve Aşırı Yükleme

```
int max(int x, int y) { // max #1
    return x > y ? x : y;
}
double max(double x, double y) { // max #2
    return x > y ? x : y;
}
int max(int x, int y, int z) { // max #3
    return x > y ? (x > z ? x : z) : (y > z ? y : z);
}
```

Ad Çözümlemesi ve Aşırı Yükleme

- Şu fonksiyon çağrılarını düşünün:

```
max(2, 3); // max #1'i çağırır
max(2.1, 3.2); // max #2'yi çağırır
max(1, 3, 2); // max #3'ü çağırır
```

- Sembol tablosu, parametrelerin sayısına ve türüne göre uygun işlevi belirleyebilir
- **Çağrı bağlamı(Calling context)**: her aramada bulunan bilgiler
- Ancak bu **belirsiz(ambiguous)** çağrı, veri türleri arasında dönüştürme yapmak için dil kurallarına(varsa) bağlıdır:

```
max(2.1, 3); // hangi max?
```

Ad Çözümlemesi ve Aşırı Yükleme

- Bu tanımların eklenmesi, fonksiyon çağrılarını C++ ve Ada'da geçerli hale getirir ancak Java'da gereksizdir.

```
double max(int x, double y) { // max #4
    return x > y ? (double) x : y;
}
double max(double x, int y) { // max #5
    return x > y ? x : (double) y;
}
```

- C++ ve Java'da mevcut oldukları şekliyle otomatik dönüştürmeler, aşırı yükleme çözümünü önemli ölçüde karmaşıklaştırır

Ad Çözümlemesi ve Aşırı Yükleme

- Bir çağrı bağlamındaki ek bilgiler, aşırı yük çözümü için kullanılabilir:
  - Ada, dönüş türü ve parametre adlarının genel gider çözümü için kullanılmasına izin verir
  - C++ ve Java, dönüş türünü yok sayıyor
- Hem Ada hem de C++ (ancak Java değil) yerleşik operatörlerin aşırı yüklenmesine izin verir
- Yerleşik bir operatörü aşırı yüklerken, sözdizimsel özelliklerini kabul etmeliyiz
  - Örnek: + operatörünün ilişkilendirilebilirliği veya önceliği değiştirilemez

Ad Çözümlemesi ve Aşırı Yükleme

- Operatörler ve fonksiyonlar arasında anlamsal bir fark olmadığını, yalnızca sözdizimsel fark olduğunu unutmayın.
  - Operatörler infix biçiminde yazılır
  - İşlev çağrıları her zaman prefix biçiminde yazılır
- İsimler de aşırı yüklenebilir
- Bazı diller, bir tür, bir fonksiyon ve bir değişken için aynı adı sağlamak üzere ana tanım türlerinin her biri için farklı sembol tabloları kullanır
  - Örnek: Java

Ad Çözümlemesi ve Aşırı Yükleme

```
class A {
    A A(A A) {
        A:
            for(;;){
                if (A.A(A) == A) break A;
            }
            return A;
    }
}
```

Şekil: Farklı dil yapıları için aynı adın aşırı yüklenmesini gösteren bir Java sınıfı

Tahsis, Yaşam Süreleri ve Ortam

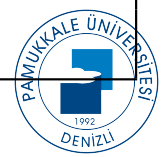
- **Ortam(Environment)**: adların konumlara bağlanmasını sağlar
  - Statik (yükleme zamanında), dinamik (yürütme zamanında) veya her ikisinin karışımı ile oluşturulabilir
- Bir programdaki tüm isimler konumlara bağlı değildir
  - Örnekler: sabitlerin ve veri türlerinin adları tamamen derleme zamanı miktarlarını temsil edebilir
- Ortam yapımında da bildirimler kullanılır
  - Hangi tahsis kodunun oluşturulması gerektiğini belirtir

Tahsis, Yaşam Süreleri ve Ortam

- Tipik olarak, blok yapılı bir dilde:
  - Global değişkenler statik olarak tahsis edilir
  - Bloğa girildiğinde yerel değişkenler dinamik olarak tahsis edilir
- Bir bloğa girildiğinde, bu blokta bildirilen değişkenler için bellek tahsis edilir
- Bir bloktan çıkıldığında, bu belleğin tahsisi kaldırılır

Tahsis, Yaşam Süreleri ve Ortam

```
1   A: {int x;
2       char y;
3       //...
4       B: {double x;
5           int a;
6           //...
7       } /* B sonu */
8       C: {char y;
9           int b;
10          //...
11          D: {int x;
12              double y;
13              //...
14          } /* D sonu */
15          //...
16      } /* C sonu */
17      //...
18  } /* A sonu */
```



Tahsis, Yaşam Süreleri ve Ortam

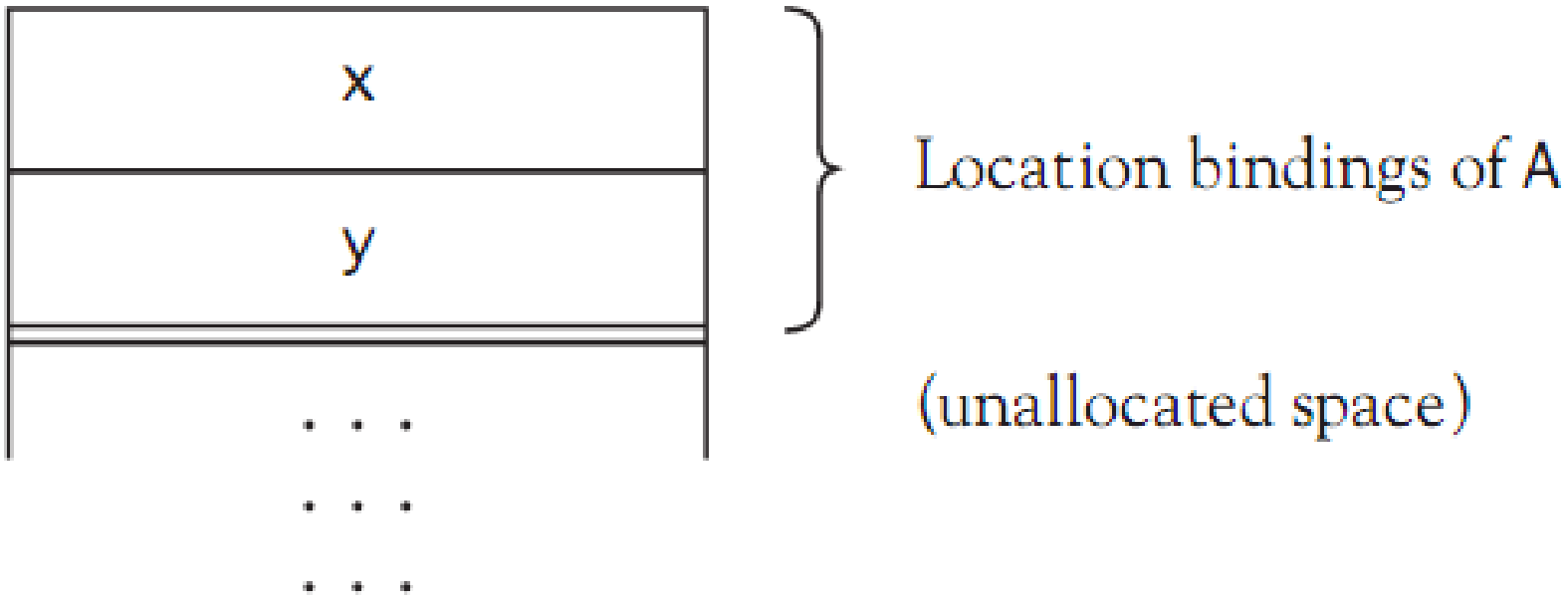


Figure 7.28 The environment at line 3 of Figure 7.27 after the entry into A

Tahsis, Yaşam Süreleri ve Ortam

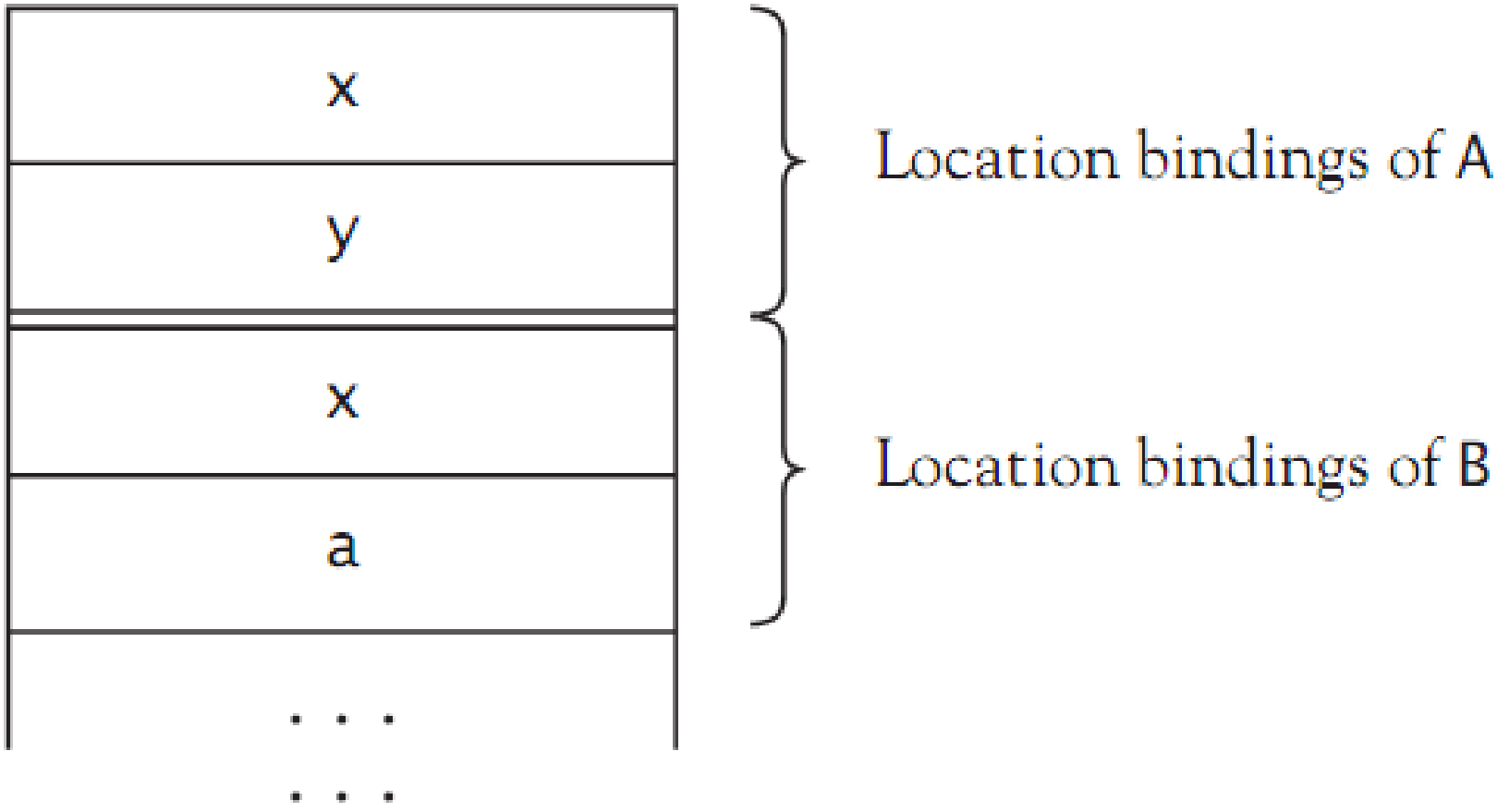


Figure 7.29 The environment at line 6 of Figure 7.27 after the entry into B

Tahsis, Yaşam Süreleri ve Ortam

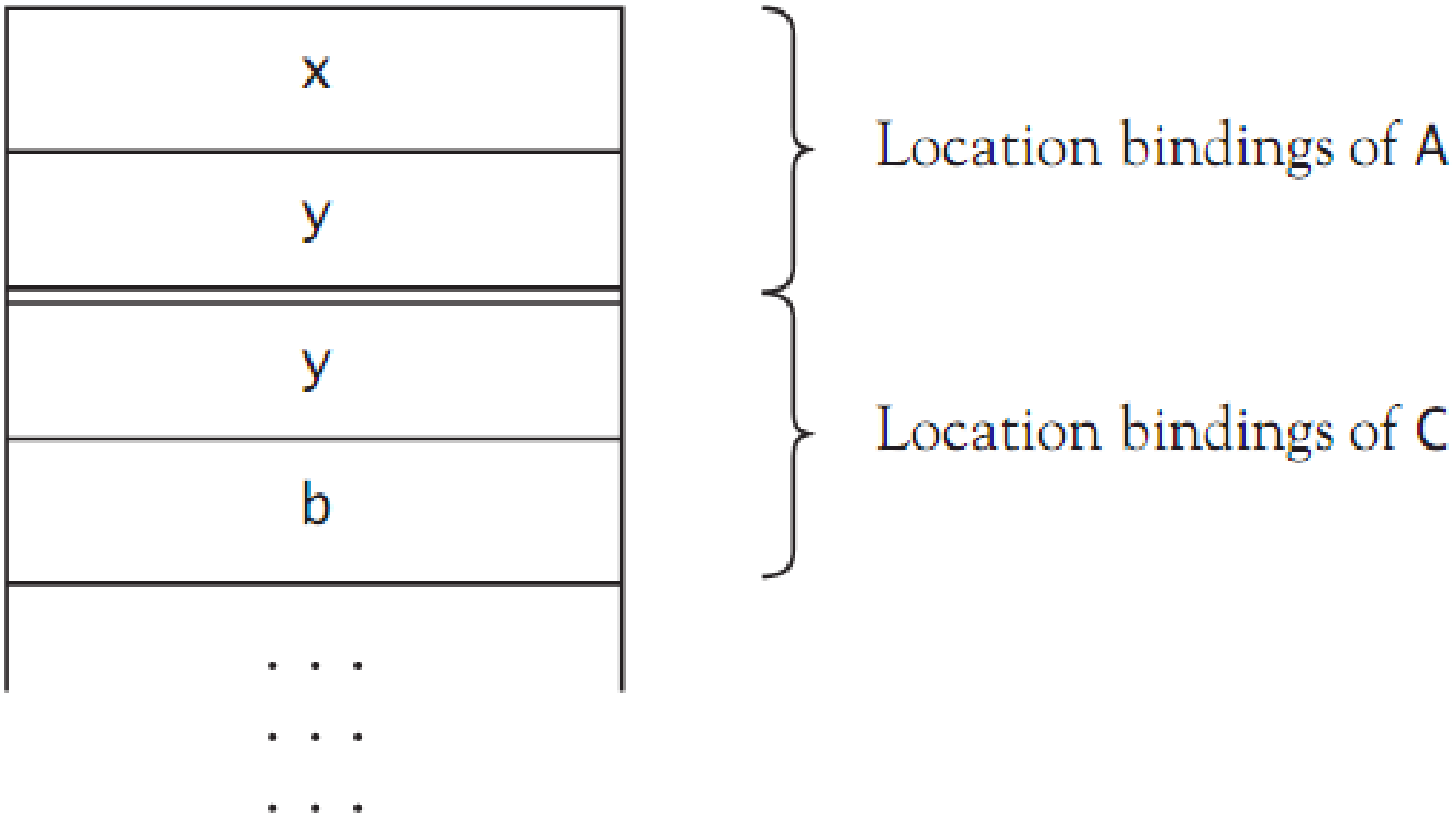


Figure 7.30 The environment at line 10 of Figure 7.27 after the entry into C

Tahsis, Yaşam Süreleri ve Ortam

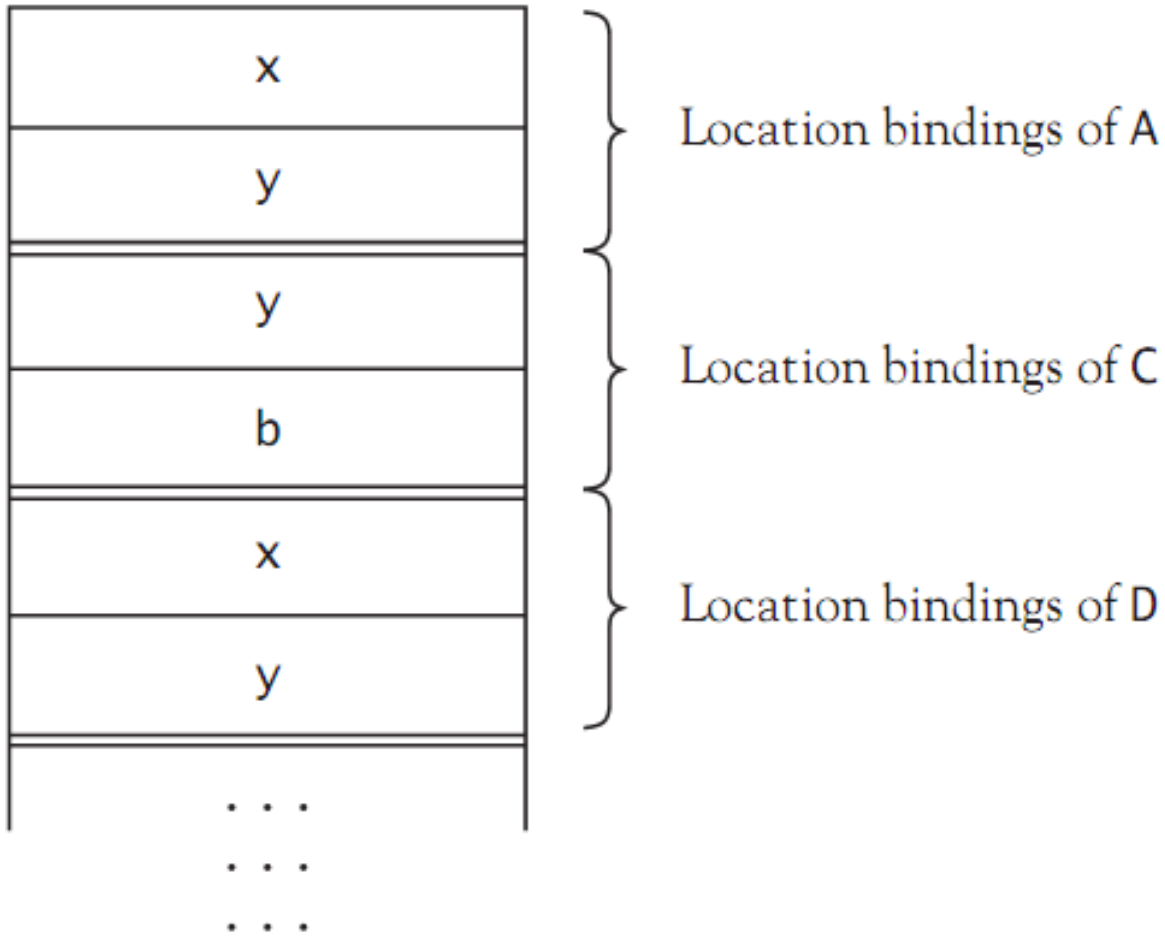


Figure 7.31 The environment at line 1 of Figure 7.27 after the entry into D

Tahsis, Yaşam Süreleri ve Ortam

- Bir fonksiyon içindeki yerel değişkenler için bellek, fonksiyon çağrılana kadar tahsis edilmeyecektir.
- **Aktivasyon(Activation)**: bir fonksiyona çağrı
- **Aktivasyon kaydı(Activation record)**: tahsis edilen hafızanın karşılık gelen bölgesi
- Sözcük kapsamı olan blok yapılı bir dilde, aynı ad farklı konumlarla ilişkilendirilebilir, ancak bunlardan yalnızca birine aynı anda erişilebilir.
- Bir nesnenin **yaşam süresi(lifetime)** (veya **kapsamı(extent)**), ortamdaki tahsis süresidir.

Tahsis, Yaşam Süreleri ve Ortam

- Bir nesnenin ömrü, erişilebildiği bir programın bölgesinin ötesine uzanabilir
  - Kullanım ömrü kapsam deliği(scope hole) ile uzar
- **İşaretçi(Pointer)**: saklanan değeri başka bir nesneye referans olan bir nesne
- C, tahsis edilmiş bir nesneye işaret etmeyen işaretçilerin başlatılmasına izin verir: `int* x = NULL;`
  - Nesneler, bir tahsis rutini kullanılarak manuel olarak tahsis edilmelidir
  - Değişkenin referansı tekli * operatörü kullanılarak kaldırılabilir

Tahsis, Yaşam Süreleri ve Ortam

- C ++, new ve delete operatörleriyle dinamik bellek tahsisini basitleştirir:

```
int* x = new int;//C++
*x = 2;
cout << *x << endl;//C++'da çıktı
delete x;
```

  - Bunlar fonksiyonlar değil, tekli operatörler olarak kullanılır.
- **Yığın(Heap)**: `new` çağrılarına yanıt olarak konumların tahsis edilebileceği bellekteki alan
- **Dinamik tahsis(Dynamic allocation)**: yığın(heap) üzerinde bellek tahsisi

Tahsis, Yaşam Süreleri ve Ortam

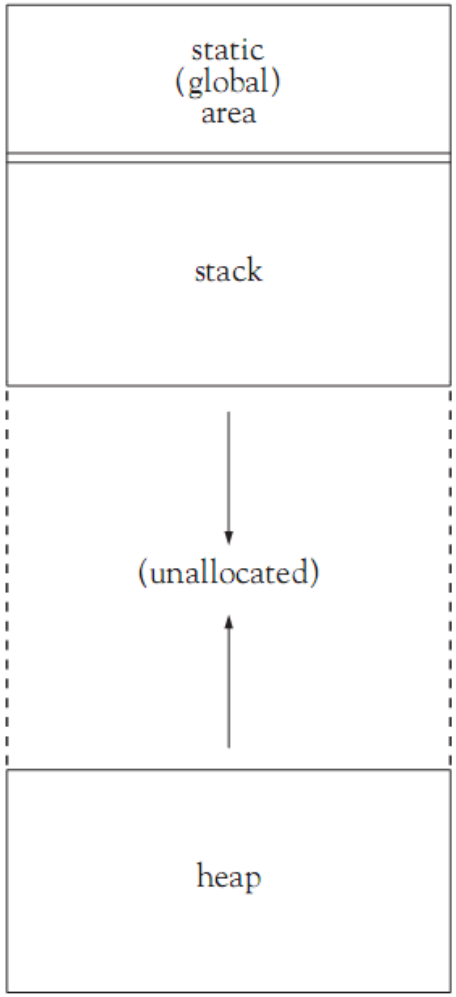


Figure 7.32 Structure of a typical environment with a stack and a heap

Tahsis, Yaşam Süreleri ve Ortam

- Birçok dil yığın(heap) tahsis kaldırmasının otomatik olarak yönetilmesini gerektirir
- Yığın(heap) tahsisi / serbest bırakma ve açık işaretçi manipülasyonu, doğası gereği güvenli olmayan işlemlerdir
  - İşletim sistemini bile tehlikeye atabilecek ciddi hatalı çalışma zamanı davranışına neden olabilir
- **Depolama sınıfı(Storage class)**: tahsis türü
  - Statik (global değişkenler için)
  - Otomatik (yerel değişkenler için)
  - Dinamik (yığın(heap) tahsisi için)

Değişkenler ve Sabitler

- Değişkenlere ve sabitlere yapılan atıflar birçok dilde aynı görünse de, rolleri ve anlamsallıkları çok farklıdır.
- Her ikisinin de temel semantiğine bakacağız

Değişkenler

- **Değişken(Variable)**: saklanan değeri yürütme sırasında değişebilen bir nesne
- Tamamen özniteliklerine göre belirtilir (ad, konum, değer, veri türü, bellek depolama boyutu)
- **Kutu ve daire diyagramı(Box and circle diagram)**: ada(isme) ve konuma odaklanır

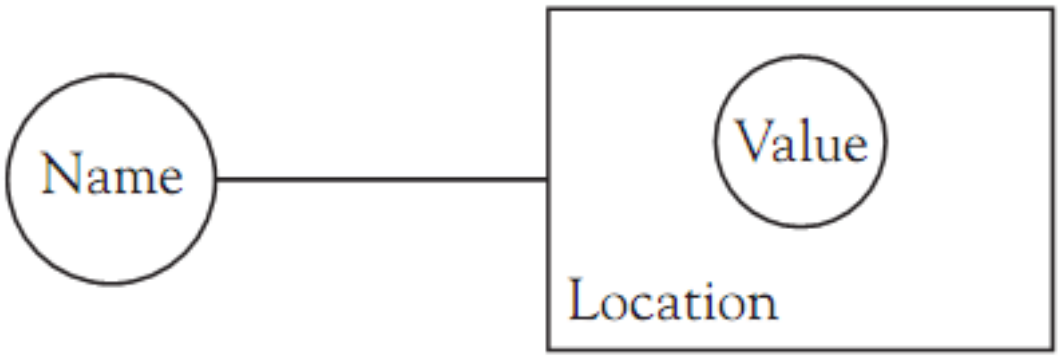


Figure 7.36 Schematic representation of a variable, its value, and its location

Değişkenler

- Atama ifadesi: bir değişkenin değerini değiştirmesinin temel yolu
- Örnek: x=*e*
  - Anlambilim: ifade e bir değer olarak değerlendirilir, ardından x'in konumuna kopyalanır
- *e*, y adında bir değişkense:

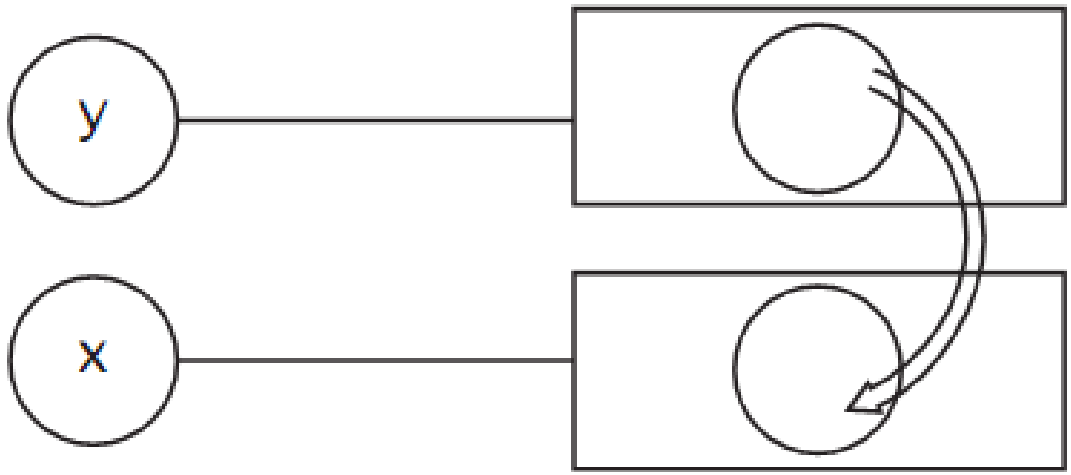


Figure 7.37 Assignment with copying of values

Değişkenler

- Atama ifadesinin sağ tarafındaki değişken bir değeri (**r-değeri(r-value)**); sol taraftaki değişken bir konum anlamına gelir (**l-değeri(l-value)**)
- C'deki **adres operatörü(address of operator)** (&): bir değişkenin adresini almak için bir referansı bir işaretçiye dönüştürür
- **Paylaşarak atama(Assignment by sharing)**: değer yerine konum kopyalanır
- **Kopyalama yoluyla atama(Assignment by cloning)**: yeni konum tahsis eder, değeri kopyalar ve yeni konuma bağlanır
- Her ikisi de bazen **işaretçi semantiği(pointer semantics)** veya **referans semantiği(reference semantics)** olarak adlandırılır

Değişkenler

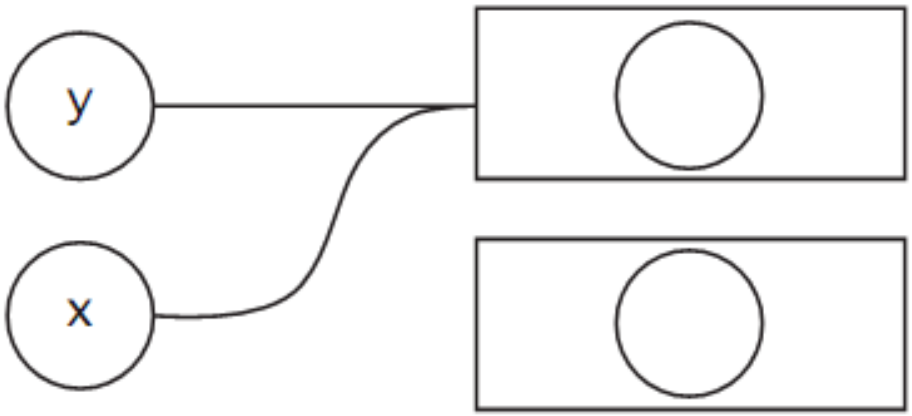


Figure 7.38 The result of assignment by sharing

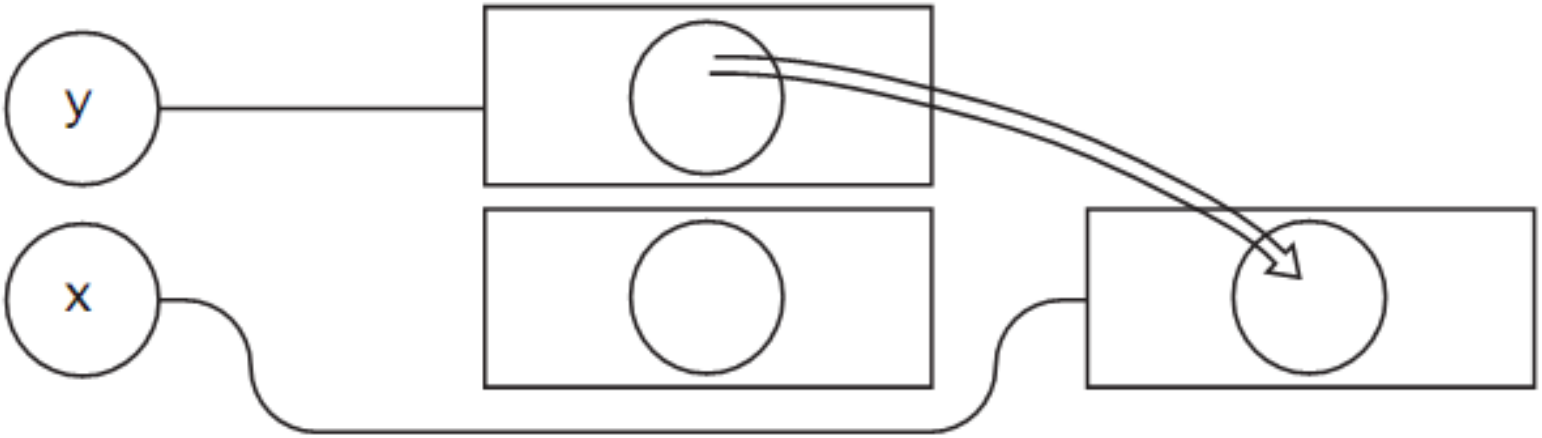


Figure 7.39 The result of assignment by cloning

Değişkenler

- **Depolama semantiği(Storage semantics)** veya **değer semantiği(value semantics)** standart atamaya atıfta bulunur
- Paylaşım yoluyla standart atama uygulaması, işaretçiler ve örtük referans kaldırma kullanır

Değişkenler

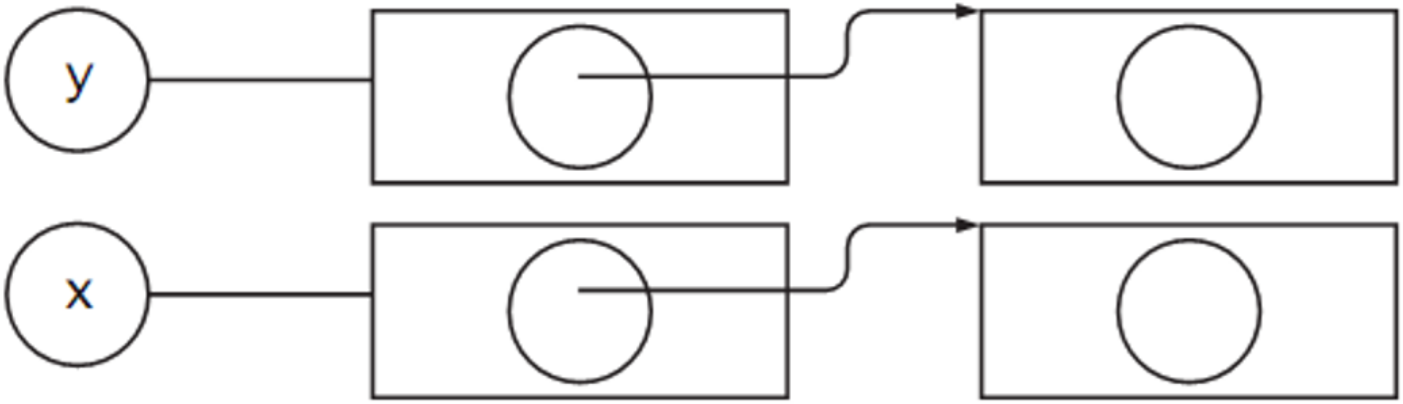


Figure 7.40 Variables as implicit references to objects

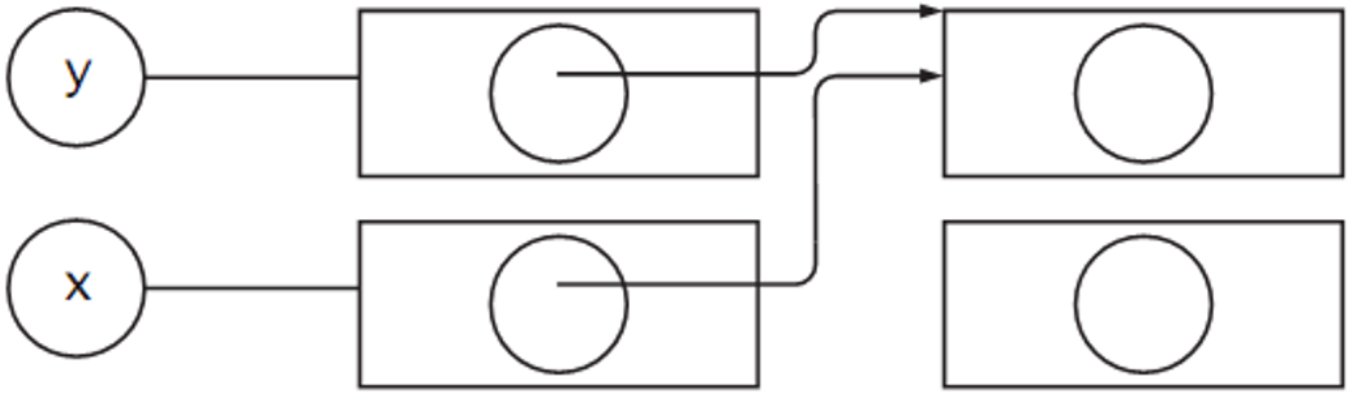


Figure 7.41 Assignment by sharing of references

Sabitler

- **Sabit(Constant)**: bir programda var olduğu süre boyunca sabit bir değere sahip bir varlık
  - Değişken gibi, ancak konum özniteliği yok
  - Bazen bir sabitin değer semantiğine sahip olduğunu söyleyin
- **Değişmez(Literal)**: karakterlerin veya rakamların temsili
- **Derleme zamanı sabiti(Compile-time constant)**: değeri derleme sırasında hesaplanabilir
- **Statik sabit(Static constant)**: değeri yükleme anında hesaplanabilir

Sabitler

- **Manifest sabiti(Maifest constant)**: bir değişmez(literal) için bir isim
- **Dinamik sabit(Dynamic constant)**: değeri yürütme sırasında hesaplanmalıdır
- Hemen hemen tüm dillerdeki fonksiyon tanımları, değerleri fonksiyon olan sabitlerin tanımlarıdır.
  - Bu, C'deki bir işaretçi olarak tanımlanması gereken bir fonksiyon değişkeninden farklıdır.

baslik

- a ve b derleme zamanı sabitleridir
- a bir manifest sabittir
- c statik (yükleme zamanı sabiti)
- d dinamik bir sabittir

```c
#include <stdio.h>
#include <time.h>

const int a=2;
const int b=27+2*2;
/* uyarı: geçersiz C kodu*/
const int c = (int) time(0);

int f(int x){
    const int d = x + 1;
    return b + c;
}
```

Takma Adlar, Sarkan Referanslar ve Çöp

- Programlama dillerinin, özellikle C, C++ ve Ada'nın, adlandırma ve dinamik tahsis kurallarıyla ilgili çeşitli sorunlar vardır.
- Bir programcı olarak, bu sorunlu durumlardan kaçınmayı öğrenebilirsiniz.
- Bir dil tasarımcısı olarak kendi dilinize çözümler geliştirebilirsiniz

Takma Adlar

- **Takma ad(Alias)**: aynı nesne aynı anda iki farklı ada bağlı olduğunda ortaya çıkar
- Prosedür çağrısı sırasında, işaretçi değişkenlerinin kullanılmasıyla veya paylaşım yoluyla atama yoluyla gerçekleşebilir

```
int *x, *y;
x = (int *) malloc(sizeof(int));
*x = 1;
y = x; /* x ve y takma adlardır*/
*y = 2;
printf("%d\n", *x);
```

Takma Adlar

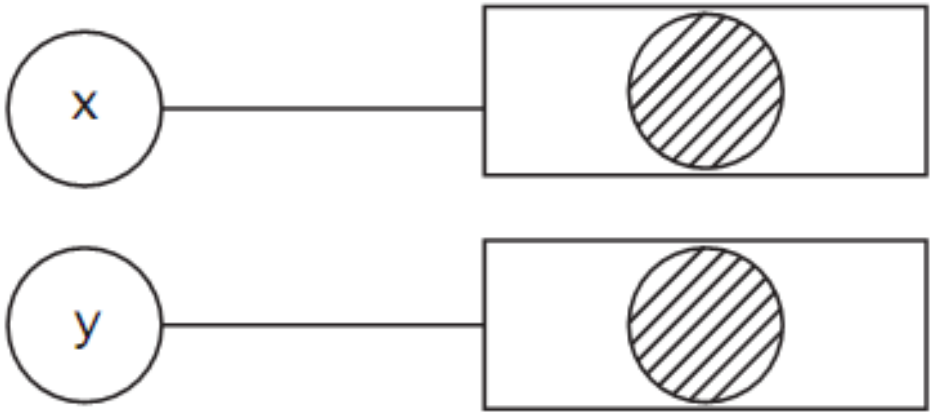


Figure 7.45 Allocation of storage for pointers x and y

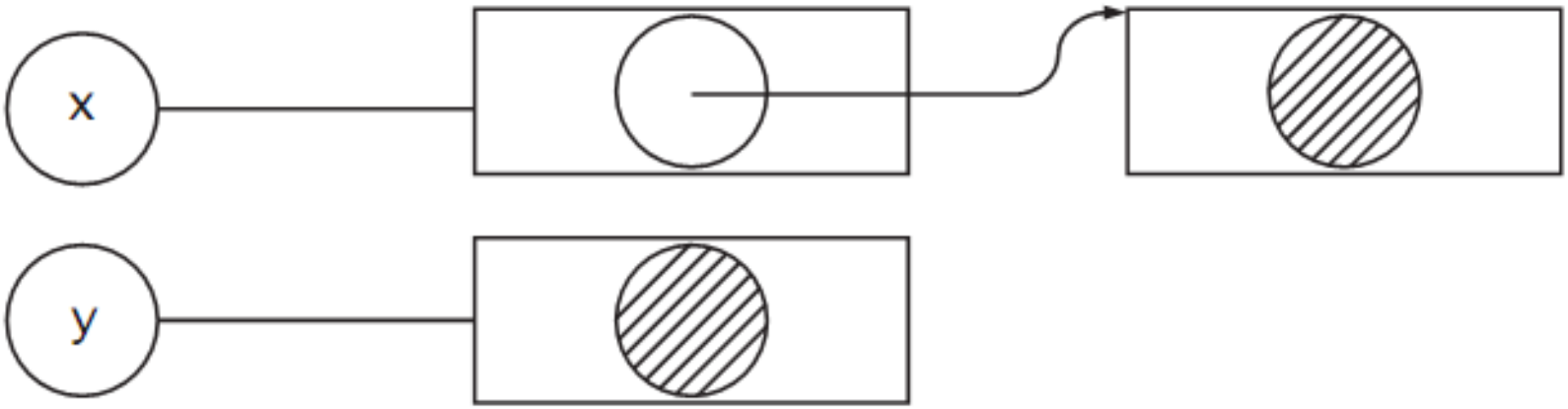


Figure 7.46 Allocation of storage for *x

Takma Adlar

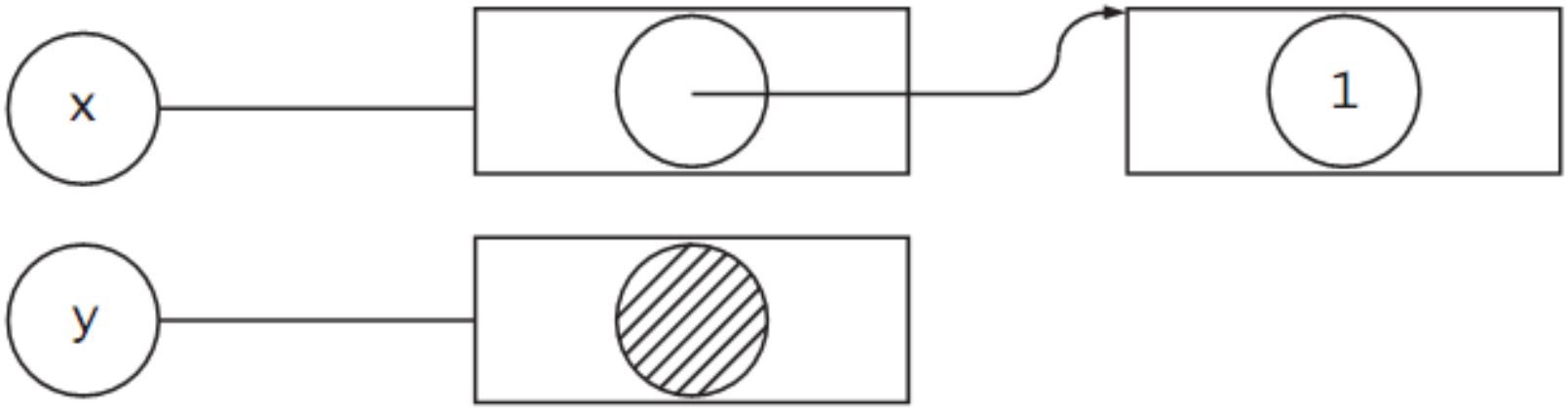


Figure 7.47 Result of *x = 1

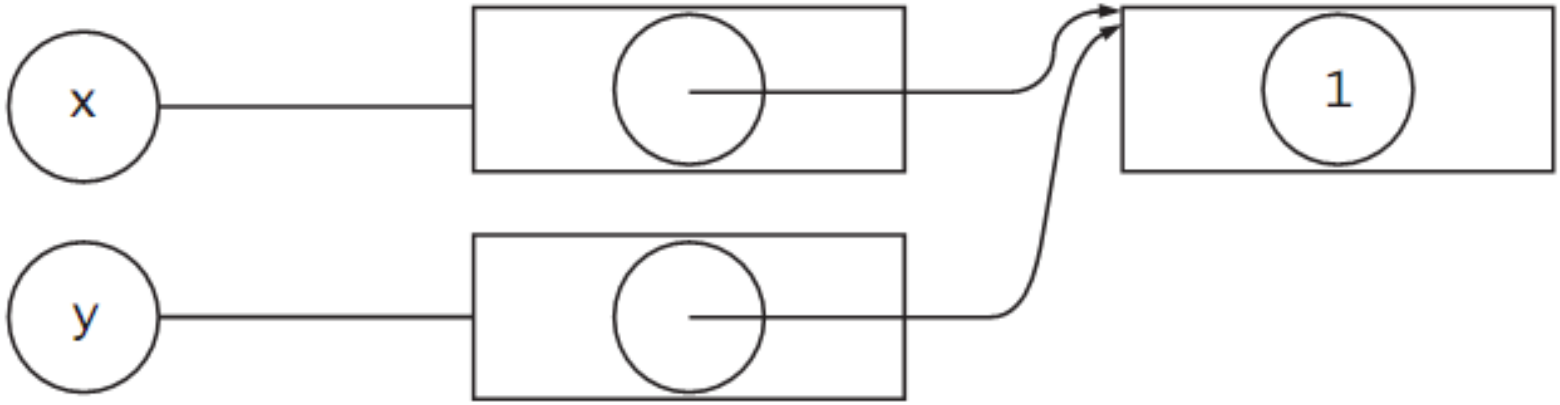


Figure 7.48 Result of y = x

Takma Adlar

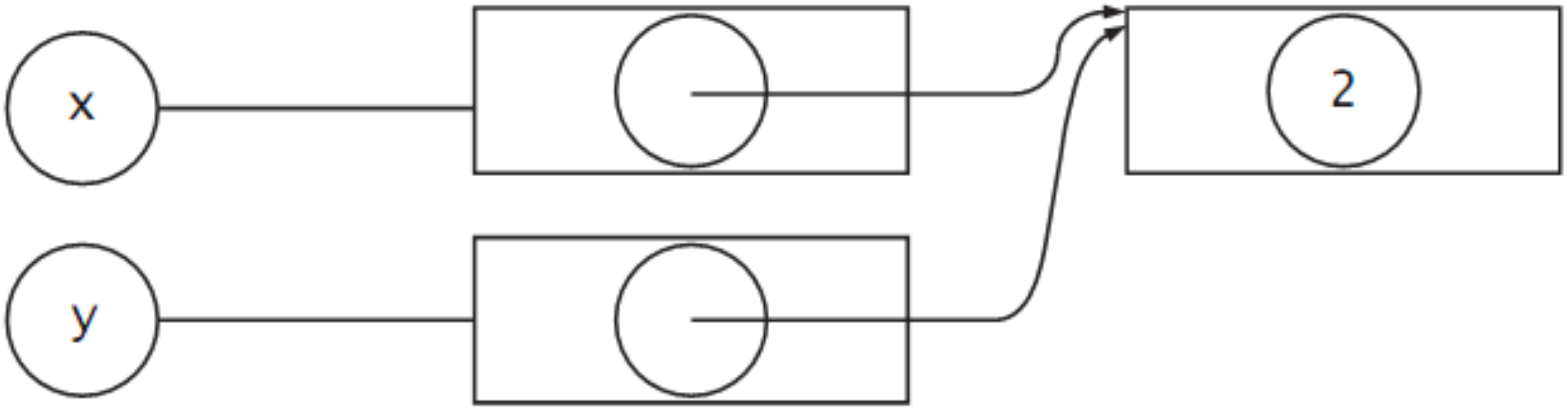


Figure 7.49 Result of *y = 2

Takma Adlar

- Takma adlar potansiyel olarak zararlı yan etkilere neden olabilir
- **Yan etki(Side effect)**: bir değişkenin değerinde, ifadenin yürütülmesinin ötesinde devam eden herhangi bir değişiklik
- Tüm yan etkiler zararlı değildir; bir atama ifadesi kasıtlı olarak değişiklik yapar
- İsimleri doğrudan ifadede görünmeyen değişkenleri değiştiren yan etkiler potansiyel olarak zararlıdır
  - Yazılı koddan belirlenemez
- İşaretçi ataması nedeniyle takma adın kontrol edilmesi zordur

Takma Adlar

- Paylaşarak atama, dolaylı olarak işaretçiler kullanır
- Java, bir nesneyi açıkça klonlamak için bir mekanizmaya sahiptir, böylece takma adlar atama ile oluşturulmaz

```
class ArrTest {
    public static void main(String[] args) {
        int[] x = {1, 2, 3};
        int[] y = x;
        x[0] = 42;
        System.out.println(y[0]);
    }
}
```

Sarkan Referanslar

- **Sarkan referans(Dangling reference)**: ortamdan tahsisi kaldırılmış ancak yine de bir program tarafından erişilebilen bir konum
  - Bir işaretçi ayrılmamış bir nesneyi işaret ettiğinde gerçekleşir

```
int *x, *y;
//...
x = (int *) malloc(sizeof(int));
//...
*x = 2;
//...
y = x; /* *x ve *y takma adlardır */
free(x); /* *y şu an sarkan bir referans */
//...
printf("%d\n", *y); /* geçersiz referans */
```

Sarkan Referanslar

- C'deki adres operatörü ile bir bloktan çıkışta yerel değişkenlerin otomatik olarak serbest bırakılmasından da kaynaklanabilir

```
{
    int *x;
    {
        int y;
        y = 2;
        x = &y;
    }
    /* *x bir sarkan referans */
}
```

Sarkan Referanslar

- Java, sarkan referanslara kesinlikle izin vermez, çünkü:
  - Açık bir işaretçi mekanizması yok
  - Adres operatörü yok
  - `free` veya `delete` gibi bellek serbest bırakma operatörleri yoktur.

Çöp

- **Çöp(Garbage)**: ortamda ayrılmış ancak mevcut anda programda erişilemeyen bellek
- Bir işaretçi değişkenini yeniden atamadan önce `free` çağrılamadığında C'de gerçekleşebilir

```
int * x;
//...
x = (int *) malloc(sizeof(int));
x = 0;
```

- Dahili olarak doğru olan ancak çöp üreten bir programın belleği yetersiz olabilir

Çöp

- Sarkan referanslara sahip bir program şunları yapabilir:
  - Yanlış sonuçlar üretir
  - Bellekteki diğer programları bozar
  - Bulunması zor olan çalışma zamanı hatalarına neden olur
- Bu nedenle, belleği programcıdan açıkça tahsis kaldırma ihtiyacını ortadan kaldırmak yararlıdır.
- **Çöp toplama(Garbage collection)**: çöpü otomatik olarak geri alma süreci
- Dil tasarımı, programların doğru yürütülmesi için gerekli olan çalıştırma ortamı türünde önemli bir faktördür

Örnek Olay: TinyAda'nın İlk Statik Semantik Analizi

- Bölüm 6, TinyAda için bir sözdizimi çözümleyicisi tanıttı
  - Bir sözdizimi hatası tespit edilene kadar bir tarayıcıdan jetonları çeken basit bir ayrıştırma kabuğu
- Burada, bazı anlamsal analizler yapmak için ayrıştırma kabuğunu genişletiyoruz
  - Kapsam analizi ve tanımlayıcıların kullanımını kısıtlama araçlarına odaklanılacaktır.
- Bir tanımlayıcının iki özelliğine odaklanılmalıdır:
  - Ad
  - Oynadığı rol (sabit, değişken, tür veya prosedür)

Kapsam Analizi

- TinyAda, aşağıdaki kapsam kuralları ile sözcüksel olarak kapsamlıdır:
  - Tüm tanımlayıcılar kullanımdan önce beyan edilmelidir
  - Tek bir blokta belirli bir tanımlayıcı için en fazla bir bildirim
  - Yeni bir blok, bir prosedürün biçimsel parametre spesifikasyonlarıyla başlar ve ayrılmış kelime sonuna kadar uzanır.
  - Bildirilen bir tanımlayıcının görünürlüğü, bu blokta yeniden bildirilmediği sürece iç içe geçmiş bloklara kadar uzanır.
  - Tanımlayıcılar büyük / küçük harfe duyarlı değildir

Kapsam Analizi

- TinyAda'da beş yerleşik (önceden tanımlanmış) tanımlayıcı bulunur:
  - Veri türü adları `integer`, `char`, `boolean`
  - Boole sabitleri `true` ve `false`
- Bir kaynak program ayrıştırılmadan önce bu tanımlayıcıların üst düzey kapsamda görünmesi gerekir *Bu kapsamın statik iç içe geçme düzeyi 0'dır*
- İç içe yerleştirme seviyesi 1'deki kapsam, prosedürün resmi parametrelerini (varsa) ve prosedürlerin temel bildirimlerinde sunulan tanımlayıcıları içerir.
- İç içe yordamlardaki isimler bu kalıbı izler

Kapsam Analizi

- TinyAda'nın ayrıştırıcısı bir dizi sembol tablosu kullanır
  - Her yeni kapsam girildiğinde, yığına yeni bir tablo itilir
  - Bir kapsamdan çıkıldığında, yığının en üstündeki tablo yığından çıkar.
- Kapsam analizini desteklemek için iki sınıf tanımlanmıştır:
  - SymbolEntry: bir tanımlayıcı hakkındaki bilgileri tutar
  - SymbolTable: kapsam yığınını yönetir

Kapsam Analizi

Table 7.1 The interface for the `SymbolTable` class

`SymbolTable` Method	What It Does
`SymbolTable(Chario c)`	Creates an empty stack of tables, with a reference to a `Chario` object for the output of error messages.
`void enterScope()`	Pushes a new table onto the stack.
`void exitScope();`	Pops a table from the stack and prints its contents.
`SymbolEntry enterSymbol(String name);`	If `name` is not already present, inserts an entry for it into the table and returns that entry; otherwise, prints an error message and returns an empty entry.
`SymbolEntry findSymbol(String name);`	If `name` is already present, returns its entry; otherwise, prints an error message and returns an empty entry.

Tanımlayıcı Rol Analizi

- Bir tanımlayıcı, değişken, sabit veya tüm veri türü gibi bir varlığı adlandırır
  - Bir tanımlayıcının bu özelliğine rolü denir
- Bir tanımlayıcının rolü, kullanımına belirli kısıtlamalar getirir
- Örnekler:
  - Bir atama ifadesinin sol tarafında yalnızca bir değişken veya parametre tanımlayıcı görünebilir
  - Bir dizinin öğe türü olarak yalnızca bir tür tanımlayıcı görünebilir

Tanımlayıcı Rol Analizi

- Tanımlayıcı, bildirimindeki rolünü kazanır
  - Rol, ileride kullanılmak üzere sembol tablosuna kaydedilir
- Rol analizi, aksi takdirde bağımsız ayrıştırma yöntemleri arasında tanımlayıcılarla ilgili bağlamsal bilgileri paylaşmak için sembol tablosunu kullanır.