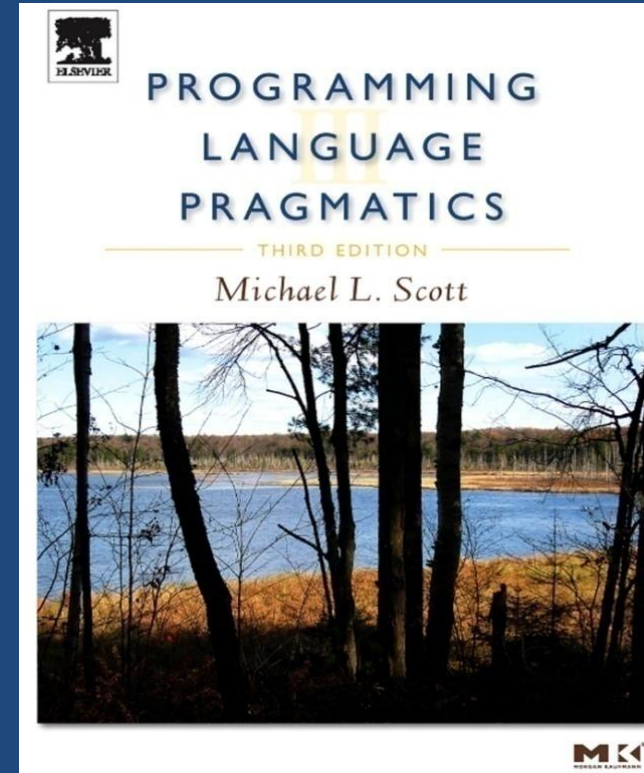
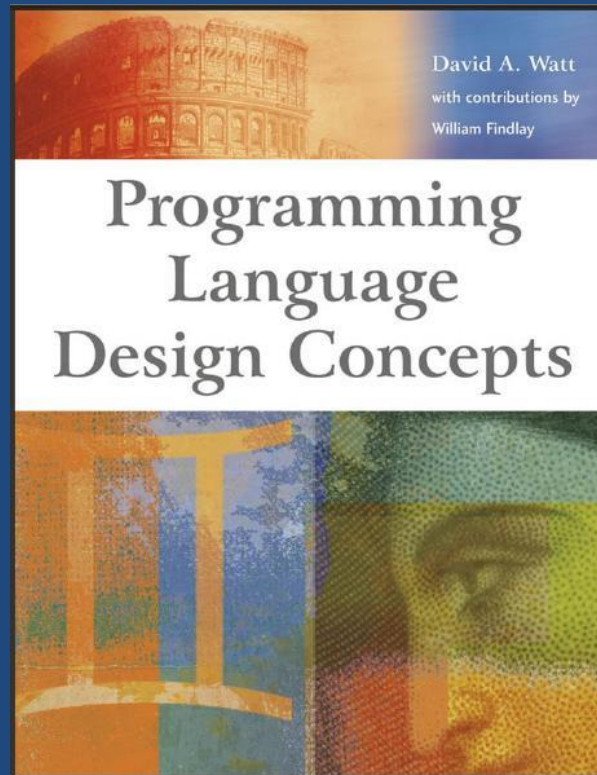
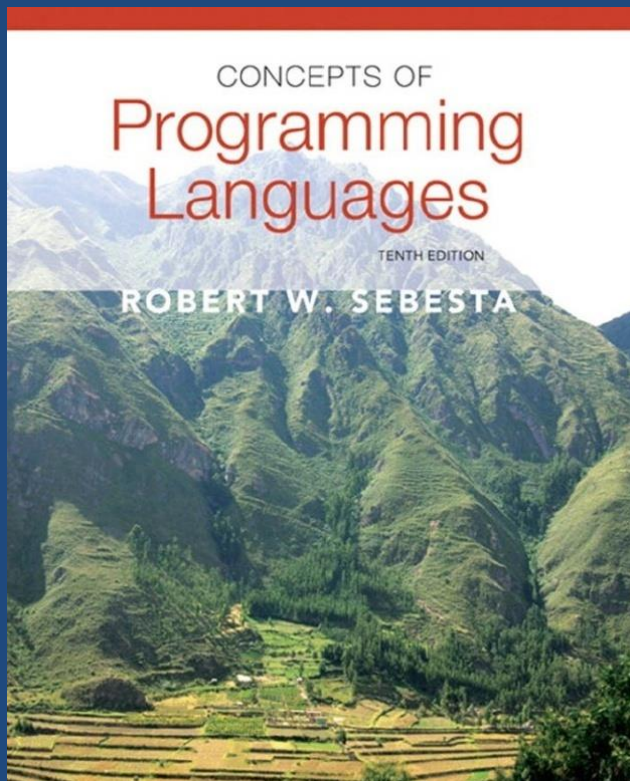


# Bölüm 9: Alt Programları Uygulamak



# Bölüm'ün Başlıkları

2

- Arama ve geri dönüşlerin genel anlamı
- “Basit” alt programların uygulaması
- Alt programların yığın dinamiği bölgesel değişkenleri ile uygulanması
- İç içe alt programlar
- Bloklar
- Dinamik kapsam uygulaması



Bilgisayar mühendisleri neden cerrah olamaz

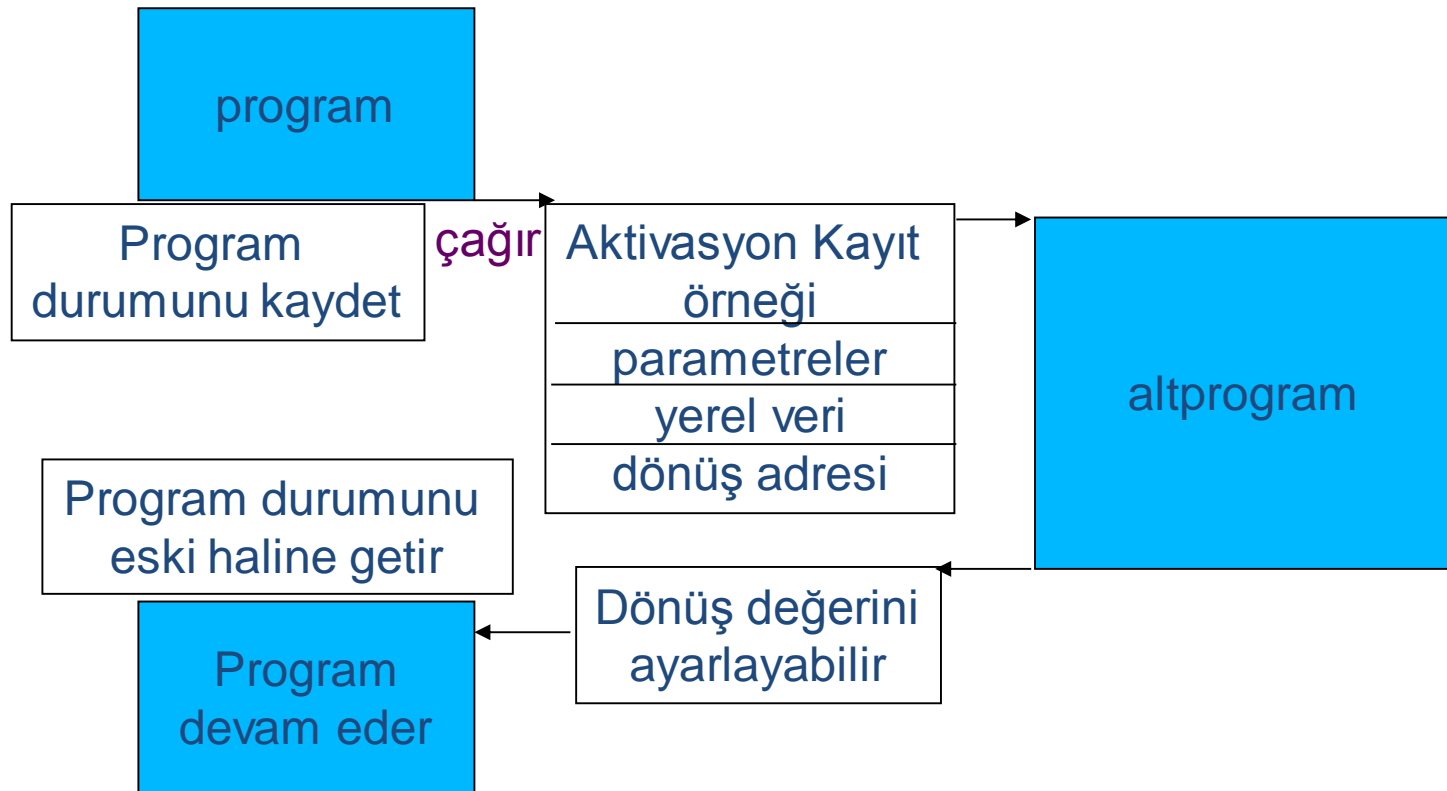
# Çağrılar ve döndükleri değerlerin genel anlam analizi

4

- Bir dilde altprogramların çağırılması ve dönmesi işlemlerine altprogram bağlanması (subprogram linkage) denir.
- Parametrelerin nasıl alt programa geçileceği belirlenmelidir.
- Çağıranın değerleri korunmalıdır.
- Alt program yerel değişkenleri statik değilse yerel olarak saklanmalıdır.
- Dönüşte çağıranın doğru noktasına, doğru değerlerle dönülmesi temin edilmelidir.
- İç içe altprogramlar varsa, yerel olmayan değişkenlere erişim sağlanmalıdır.

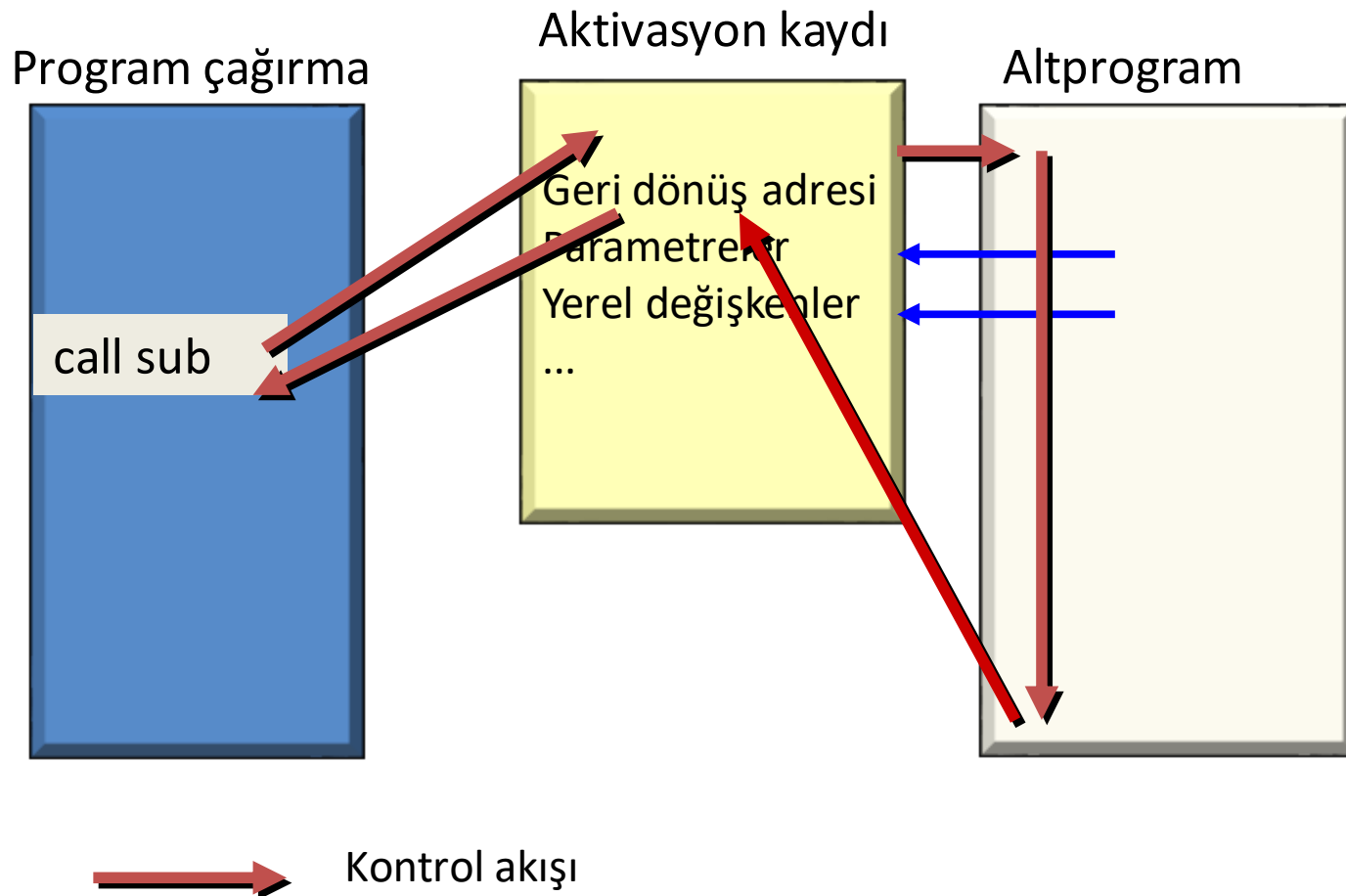
# Genel Tasarım

5



# Genel Tasarım

6



# "Basit" altprogramların gerçekleştirilmesi

7

- Basit altprogramdan kastımız, bütün yerel değişkenleri statik olan ve iç içe çağrılmayan altprogramlar (örn. Fortran I).
- Çağrı anlam analizi:
  1. Çağırان programın çalışma durumunu sakla (Save the execution status of the caller).
  2. Parametre geçme işlemlerini yap (Carry out the parameter-passing process).
  3. Dönüş adresini çağrılana geç (Pass the return address to the callee)
  4. Kontrolü çağrılana ver (Transfer control to the callee).

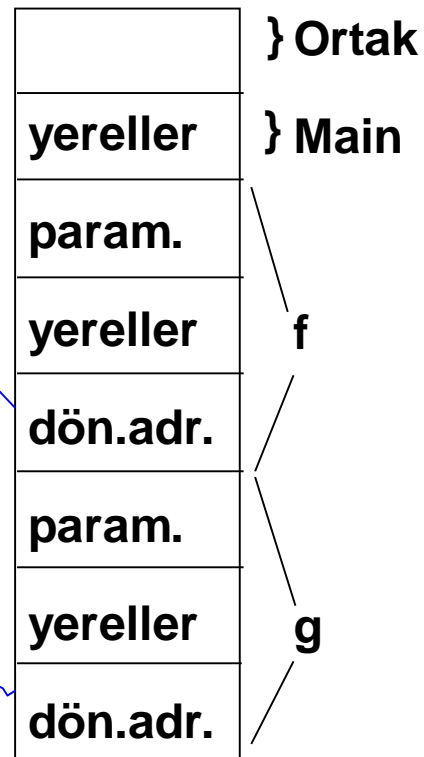
# Fortran aktivasyon bilgisi

8

- Tüm hafızaya statik olarak yerleştirebilir

main

```
...  
f (...)  
...  
proc f (...)  
...  
g (...)  
...  
proc g( )  
...  
g (...) -- !!!
```





# "Basit" altprogramların gerçekleştirilmesi

9

- Dönüş analizi (Return Semantics):
  - 1. Eğer sonucu değeri ile geç (pass-by-value-result) parametreler kullanılmışsa değerlerinin gerçek parametrelere geçir.
  - 2. Eğer dönen fonksiyonsa, döneceği değeri çağırının bulması gereken yere yerleştir.
  - 3. Çağırının çalışma ortamını yeniden yapılandır.
  - 4. Kontrolü çağırana geri ver.

# "Basit" altprogramların gerçekleştirilmesi

10

- Gerekli depolama: Çağırılan durum bilgileri, parametreleri, dönüş adresi ve döneceği değer (eğer fonksiyonsa).
- Altprogramın çalışan kod olmayan, altprogram verilerinin tutulduğu kısmının biçimi veya yerleşim planına etkinleştirme kaydı (activation record) denir. Bu kaydın sonraki sayfalarda göreceğimiz gibi belli bir formatı olur.
- Altprogram çağırıldığı zaman belli değerlerle doldurulmuş haline somutlaşan gerçekleşme kaydı denir (instance of activation record).

# "Basit" altprogramların etkinleştirme kaydı

11

Fonksiyon geri dönüş  
adresi

Yerel değişkenler

Parametreler

Geri dönüş adresi

# "Basit" altprogramların etkinleştirme kaydı ve kodu

12

- ❑ Etkinleştirme kaydı verileri statik bellekte statik olarak saklanıyor.
- ❑ Bu nedenle somutlaşmış tek etkinleştirme kaydı olabilir.
- ❑ Bu nedenle kullanım olarak kolay, erişim daha hızlı, ancak özyinelemeyi desteklemez.
- ❑ İlk Fortran derleyicileri bu tip etkinleştirme kaydı tutuyordu.

```
void sub (float total, int part) {  
    int list[4];  
    float sum;  
    ...  
}
```

**Kod Olmayan Bölüm**  
(Yerel değişkenler ve veri)

**Kod Bölümü**  
(Yürütme kodu)

### Aktivasyon Kaydı

**Yerel Değişkenler**  
(list, sum)

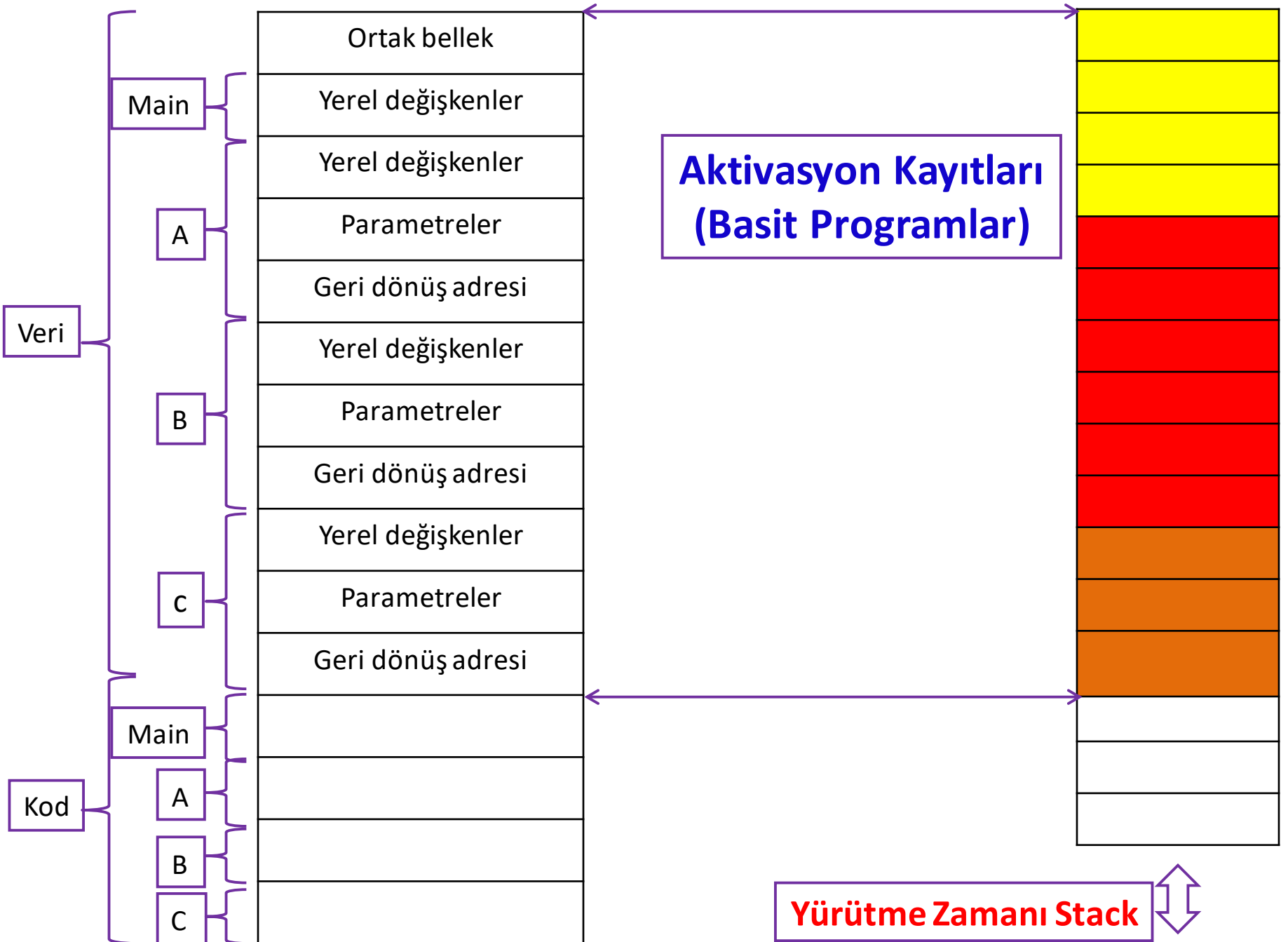
**Parametreler**  
(total, part)

**Dinamik Linkler**

**Dönüş Adresi**

*Çağırır kodda bir yer*

*Aktivasyon kayıt örneğinin başı*



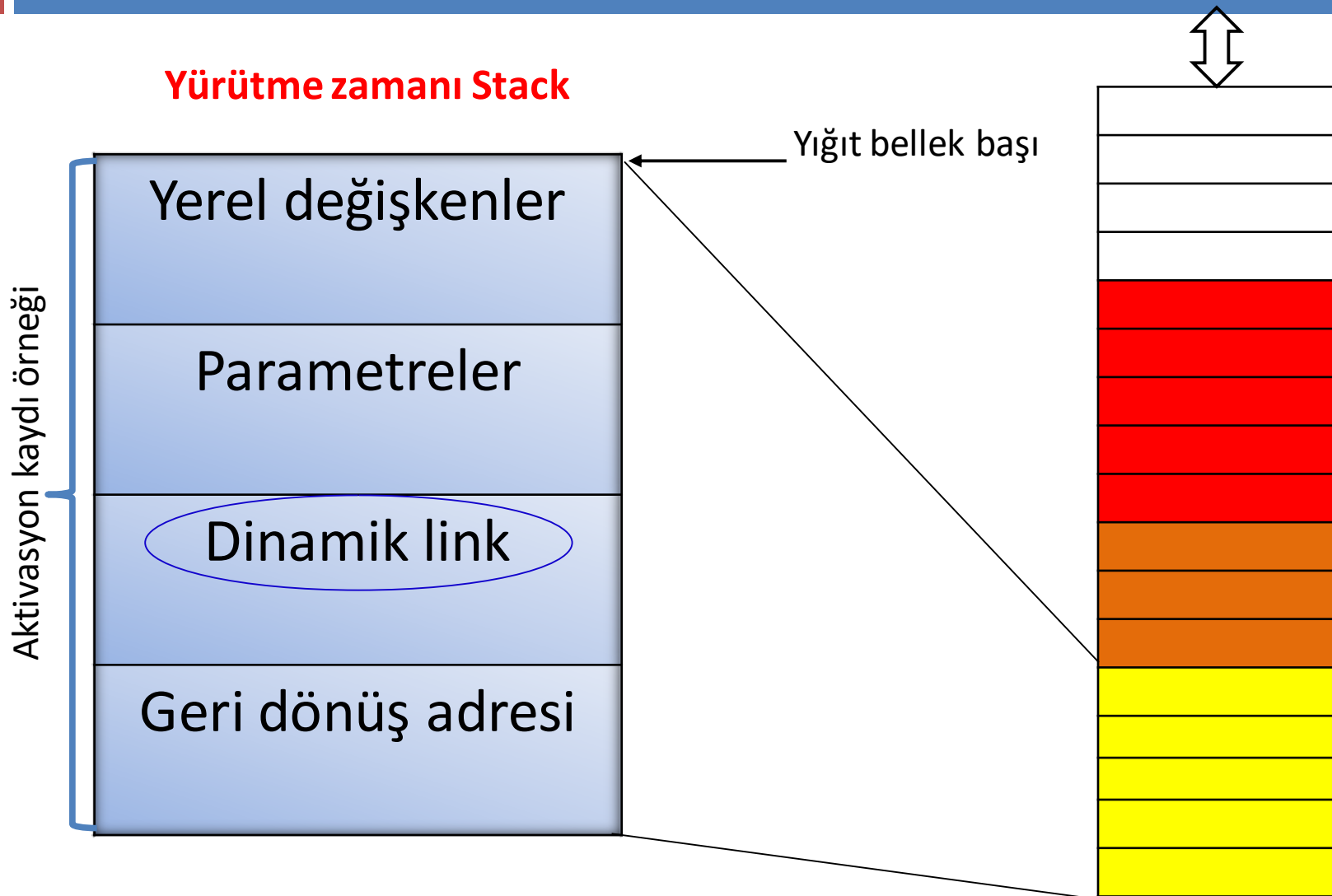
# Altprogramları yığıt dinamik (Stack-Dynamic) yerel değişkenlerle gerçekleştirmek

15

- Daha karmaşık çünkü:
  - Derleyici altprogram yerel değişkenlerine yığıt bellekte örtülü bellek tahsis ve geri alma işlemleri için kod hazırlamalıdır.
  - Özyineleme desteklenebildiğinden altprogramın aynı anda birden çok peş peşe çağırılabilmesini destekler, bu da birden çok etkinleştirme kodu somutlaşmasına neden olur.
  - Bazı dillerde yerel dizilimlerin (local arrays) boyu çağrı sırasında belirlendiğinden, etkinleştirme kodunun içinde yer alan bu tip değişkenler etkinleştirme kodunun boyunun dinamik olmasını gerektirir.

# Tipik Altprogramları yığıt dinamik yerel değişkenlerle gerçekleştirme etkinleştirme kaydı

16





# Tipik Altprogramları yığıt dinamik yerel değişkenlerle gerçekleştirme

```
void sub(float total, int part) {  
    int list[4];  
    float sum;  
    ...  
}
```

Yerel değişken	sum
Yerel değişken	list[3]
Yerel değişken	list[2]
Yerel değişken	list[1]
Yerel değişken	list[0]
Parametre	part
Parametre	total
Dinamik link	
Geri dönüş adresi	

# Altprogramları yığıt dinamik yerel değişkenlerle gerçekleştirmek

18

- Etkinleştirme kaydı formatı statik fakat boyutları dinamiktir.
- “Dinamik link” çağıran programın etkinleştirme kaydının başını gösterir.
- Etkinleştirme kaydı dinamik olarak altprogram çağırıldığında üretilir.



# C fonksiyonu örneği

19

```
void sub (float total,  
         int part) {  
    int list[5];  
    float sum;  
    ...  
}
```

- statik link:  
yerel olmayan  
değişkenler referansı  
için kullanılır
- dinamik link:  
çağıranın AR'si nin  
başını gösterir

Yerel	sum
Yerel	list ( 4 )
Yerel	list ( 3 )
Yerel	list ( 2 )
Yerel	list ( 1 )
Yerel	list ( 0 )
Parametre	part
Parametre	total
Dinamik link	
Statik link	
Geri dönüş adresi	

sub prosedürü için  
aktivasyon kaydı

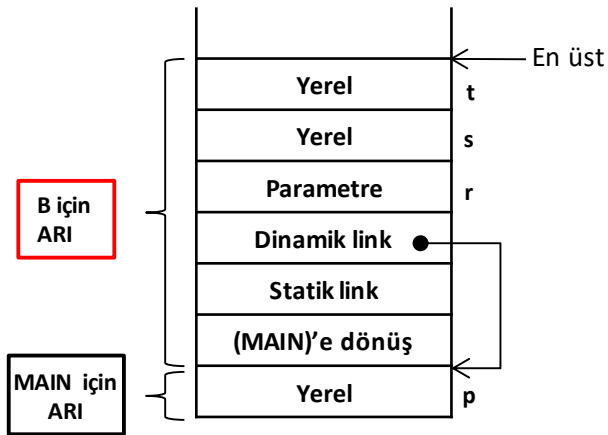
# Özyinelemesiz C programı örneği – Program için yığıt bellek içeriği

20

```
void A(int x) {  
    int y;  
    ...  
    C(y);  
    ...  
}  
void B(float r) {  
    int s, t;  
    ...  
    A(s);  
    ...  
}  
void C(int q) {  
    ...  
}  
void main() {  
    float p;  
    ...  
    B(p);  
    ...  
}
```

# MAIN, B'yi çağırdıktan sonra Stack

main calls **B**



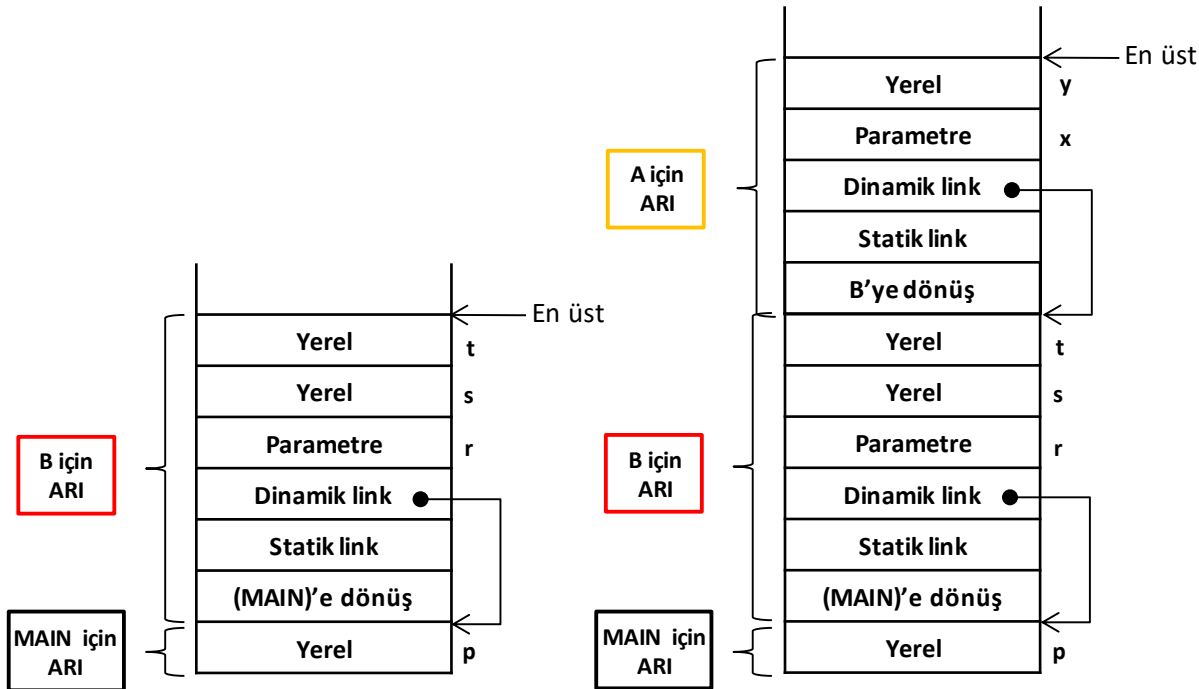
```
void main() {  
    float p; ...  
    B(p); ...  
}
```

```
void B(float r) {  
    int s, t; ...  
    A(s); ...  
}
```

ARI = Etkinleştirme kaydı (activation record instance)

# B, A'yı çağırdıktan sonra Stack

main calls **B** calls **A**



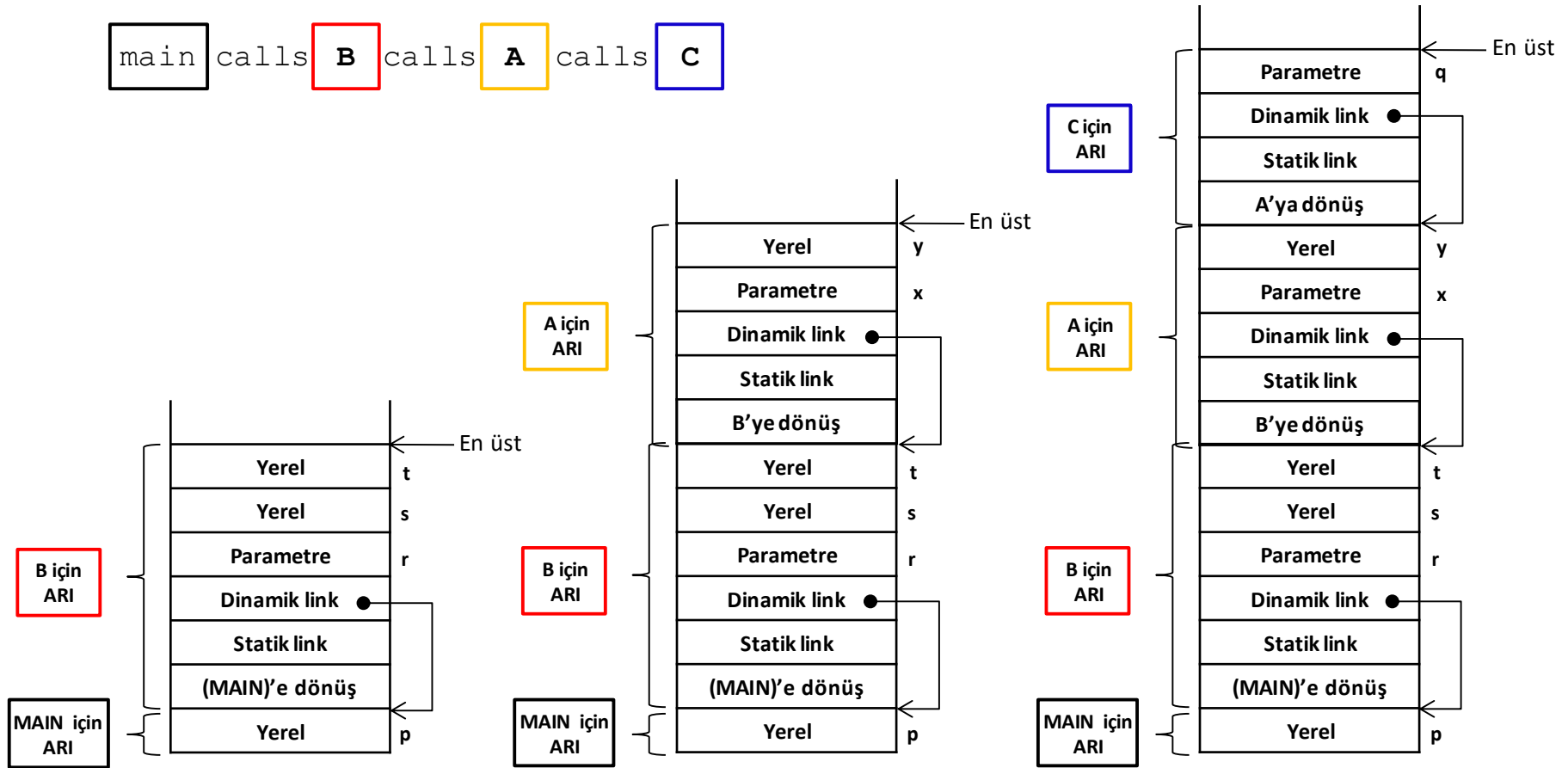
```
void main() {  
    float p; ...  
    B(p); ...  
}
```

```
void B(float r) {  
    int s, t; ...  
    A(s); ...  
}
```

```
void A(int x) {  
    int y; ...  
    C(y); ...  
}
```

# A, C'yi çağırdıktan sonra Stack

main calls **B** calls **A** calls **C**



```
void main() {  
    float p; ...  
    B(p); ...  
}
```

```
void B(float r) {  
    int s, t; ...  
    A(s); ...  
}
```

```
void A(int x) {  
    int y; ...  
    C(y); ...  
}
```

```
void C(int q) {  
    ...  
}
```

# Altprogramların gerçekleştirimi

24

- Herhangi bir anda yığıttaki dinamik linklerin toplamına çağrı zinciri (call chain) denir.
- Yerel değişkenlere etkinleştirme kaydının (activation record) başlangıcına göre bağlı konumlarından (offset) erişilir. Buna yerel bağlı konum denir (local\_offset).
- Yerel değişkenlerin yerel bağlı konumları derleyici tarafından belirlenir.



# Özyinelemede Aktivasyon Kayıtları

25

Bir önceki örnekte kullanılan etkinleştirme kaydı (activation record) özyinelemeli fonksiyonlarda da kullanılabilir

Fonksiyon değeri
Parametreler
Dinamik link
Geri dönüş adresi
<b>Main</b>

3

```
int factorial(int n) {  
    if (n <= 1)  
        return 1  
    else  
        return (n * factorial(n-1));  
}  
  
void main() {  
    int value;  
    value = factorial(3);  
}
```

# Factorial için Aktivasyon Kayıtları

26

Geri dönüş değeri
Parametreler
Dinamik link
Geri dönüş adresi
Geri dönüş değeri
Parametreler
Dinamik link
Geri dönüş adresi
<b>Main</b>

2

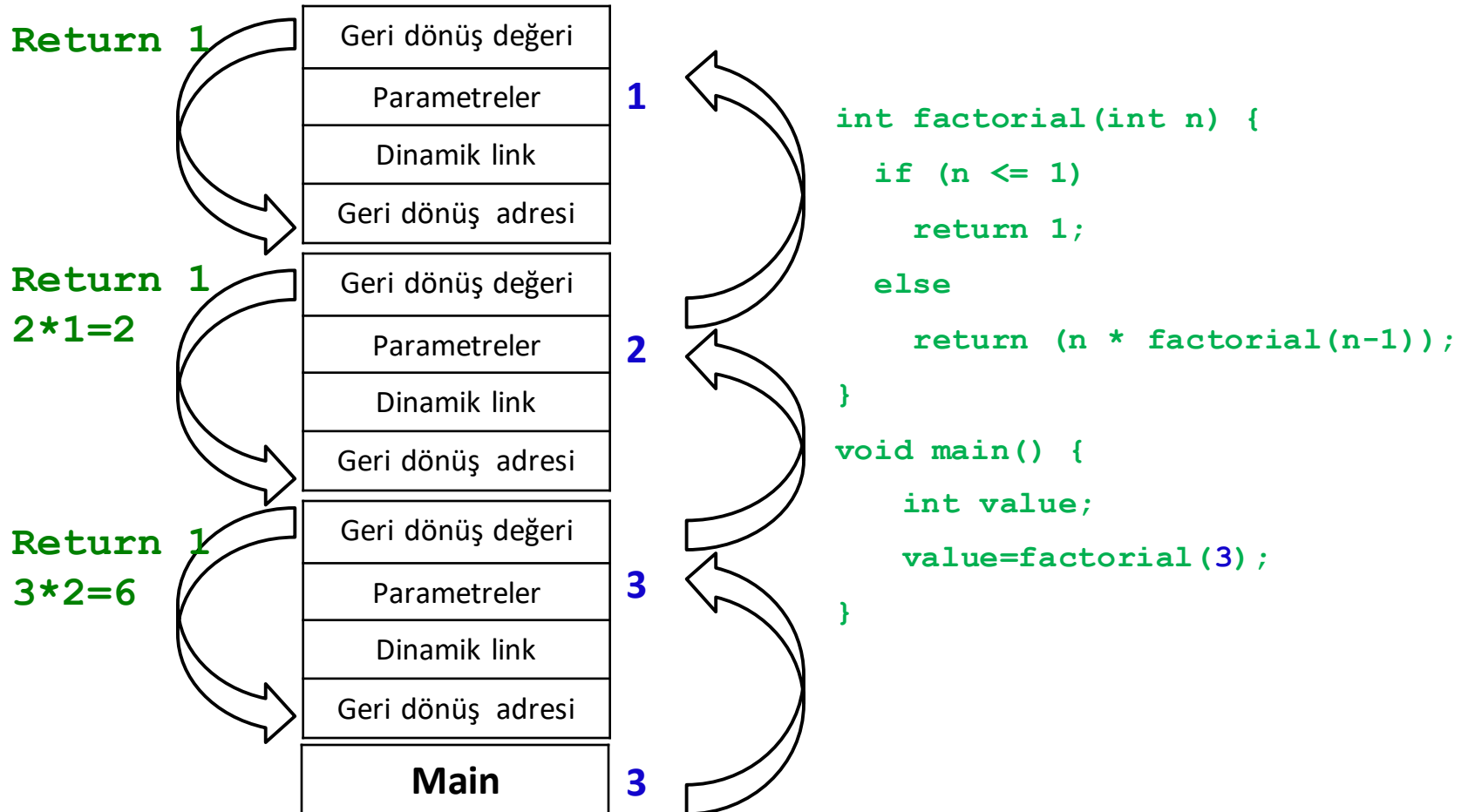
```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n * factorial(n-1));  
}
```

3

```
void main() {  
    int value;  
    value = factorial(3);  
}
```

# Factorial için Aktivasyon Kayıtları

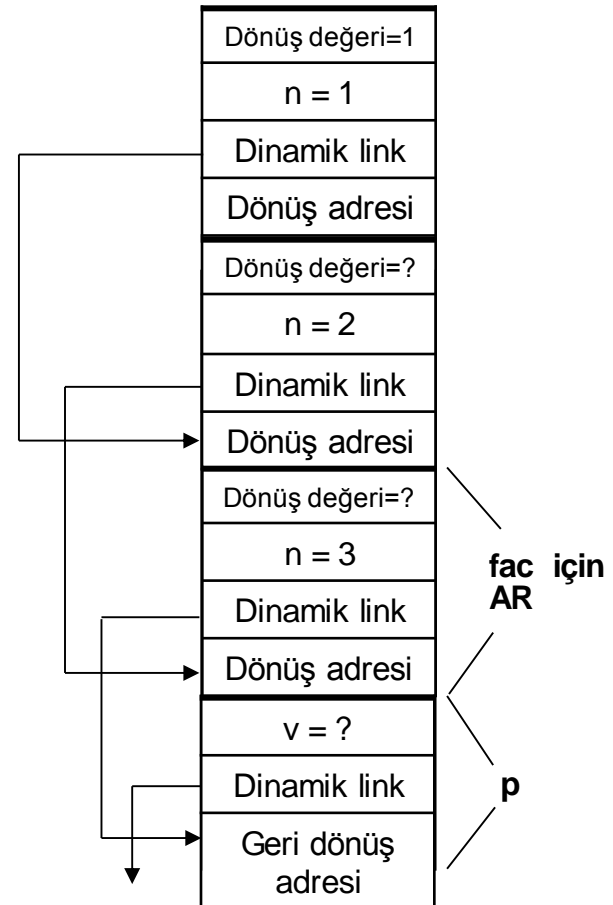
27



# Faktoriyel Programı

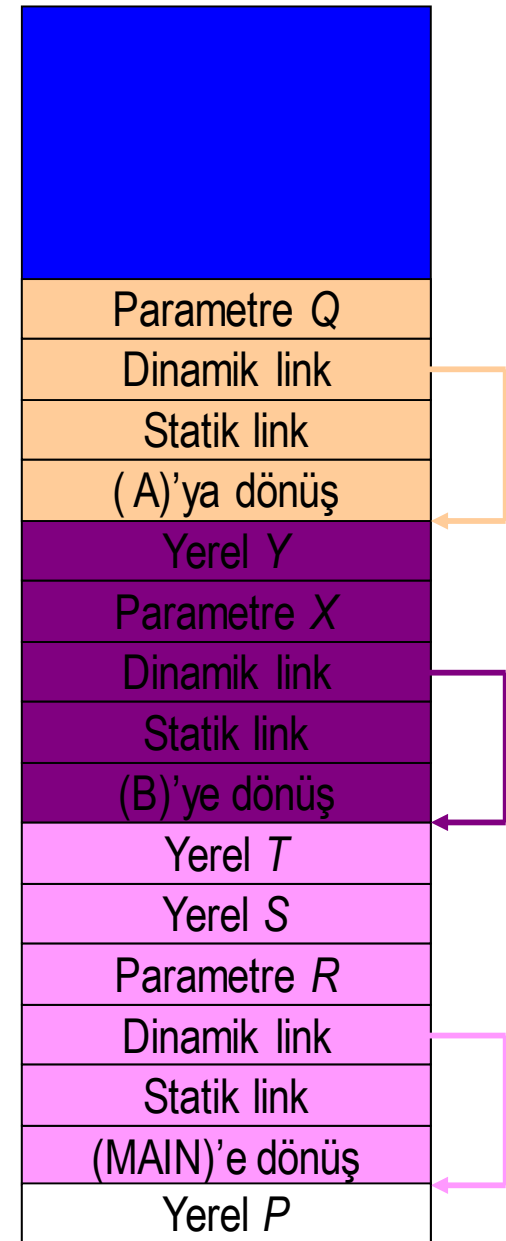
28

```
program p;  
  var v : int;  
  function fac(n: int): int;  
  begin  
    if n <= 1 then  
      fac := 1  
    else  
      fac := n * fac(n - 1);  
    end;  
  begin  
    v := fac(3);  
    print(v);  
  end.
```



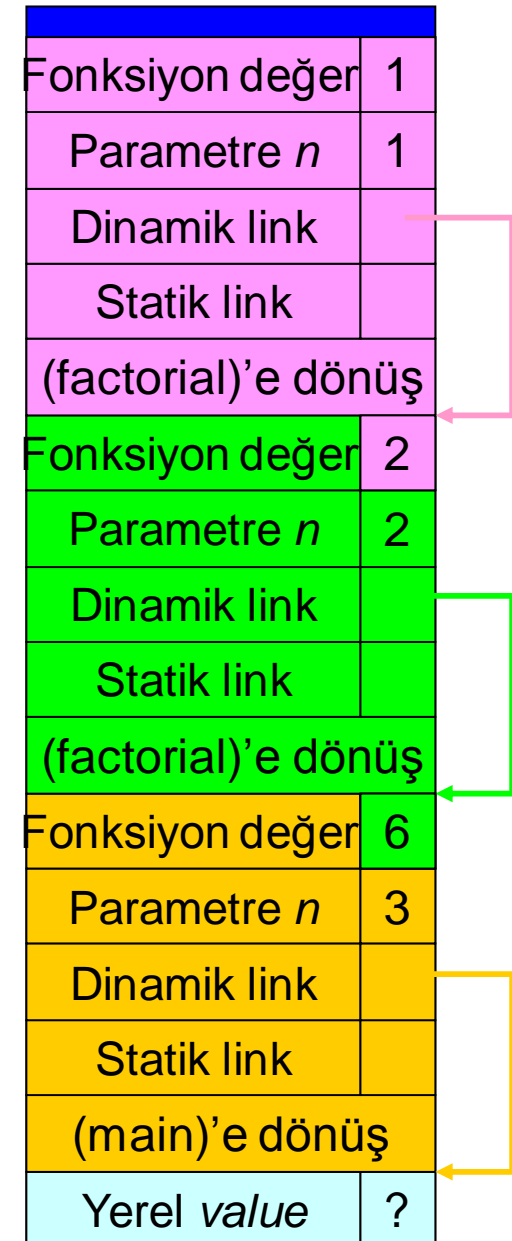
# Yerel başvuru örneği

```
Program MAIN_1;  
  var P : real;  
  procedure A (X : integer);  
    var Y : boolean;  
    procedure C (Q : boolean);  
      begin {C}  
        ...  
      end; {C}  
    begin {A}  
      ...  
      C(Y);  
      ...  
    end; {A}  
  procedure B (R : real);  
    var S, T : integer;  
    begin {B}  
      ...  
      A(S);  
      ...  
    end; {B}  
  begin {MAIN_1}  
    ...  
    B(P);  
    ...  
  end. {MAIN_1}
```



# Özyineleme örneği

```
int factorial (int n){  
    If (n <=1)  
        return 1;  
    else return (n* factorial (n-1));  
}  
void main(){  
    int value;  
    value = factorial (3);  
}
```



# İçiçe altprogramlar(Nested subprograms)

31

- Bazı programlama dilleri (Fortran 95, Ada, JavaScript) yığıt dinamik yerel değişkenler kullanır (use stack-dynamic local variables) ve içiçe altprogramlara izin verirler.
- Kural: yerel olarak erişilmeyen tüm değişkenler yığıt bellekte aktif bir etkinleştirme kaydı içinde bulunurlar.
- Yerel olmayan değişken referansı bulma işlemi:
  1. Doğru etkinleştirme kaydını bul.
  2. Etkinleştirme kaydında değişkenin bağlı konumunu bul.

# Yerel olmayan değişken referansını bulma işlemi

32

- Etkinleştirme kaydı içinde bağıl konumu bulmak kolay.
- Doğru etkinleştirme kaydını bulmak:
- Statik anlam kuralları gereği yerel olarak erişilmeyen tüm değişkenler yığıt bellekte aktif bir etkinleştirme kaydı içinde bulunurlar.



# Yerel olmayan değişken referansını bulma işlemi

33

- **1. Teknik – Statik zincir (Static Chains)**
- Belli etkinleştirme kayıtlarını birleştiren statik link zincirine statik zincir (static chain) denir.
- Bir altprogramın etkinleştirme kaydını ebeveyninin etkinleştirme kaydına bağlayan linke statik link (static link) denir.
- Statik zincir bir etkinleştirme kaydını yığıt bellekte çağıran altprogramların etkinleştirme kayıtlarına bağlar.
- Yerel olmayan değişkenin tanımlandığı yeri bulmak için:
- Statik zincir üzerinden etkinleştirme kayıtlarında değişkenin adı aranır.
- Etkinleştirme kaydının statik zincirdeki derinliğine `statik_derinlik (static_depth)` denir.

# 1. Teknik – Statik zincir (Static Chains)

34

- Yerel olmayan değişkenin tanımlandığı yeri bulmak için:
  - ▣ Statik zincir üzerinden etkinleştirme kayıtlarında değişkenin adı aranır.
- Etkinleştirme kaydının statik zincirdeki derinliğine `statik_derinlik` (`static_depth`) denir.

# 1. Teknik – Statik zincir (Static Chains)

35

```
main ----- static_derinlik = 0
┌
├   A ----- static_derinlik = 1
│   ┌
│   │   B ----- static_derinlik = 2
│   └
└
┌
├   C ----- static_derinlik = 1
└
```

# 1. Teknik – Statik zincir (Static Chains)

36

- Tanım: zincir\_bağılkonum (chain\_offset) veya içiçe\_bağılkonum (nesting\_depth ): Bir değişkenin referans verildiği altprogramın statik derinliği ile tanımlandığı altprogramın statik derinliği arasındaki fark.
- Bir referans aşağıdaki çift ile gösterilebilir:  
(zincir\_bağılkonum, yerel\_bağılkonum)  
burada yerel\_bağılkonum (local\_offset), değişkenin bulunduğu etkinleştirme kaydındaki bağıl konumudur.

# Statik zincir ile lokal olmayan referanslar (statik derinlik – iç içe derinlik)

37

```
program main          ← 0   Statik derinlik
  var x,y,w
  procedure sub1      ← 1
    var z,y
    procedure sub11   ← 2
      var z,w
      begin
        z = x + y * w  ← İç içe derinlik, offset:
      end
    begin
      sub11 ()
    end
  begin
    sub1 ()
  end
end
```

z: 0, 3  
x: 2, 3  
y: 1, 4  
w: 0, 4

# Örnek Pascal Programı

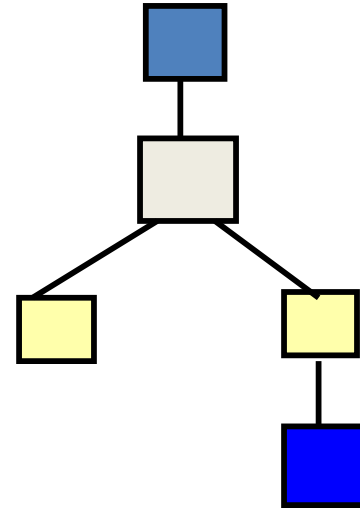
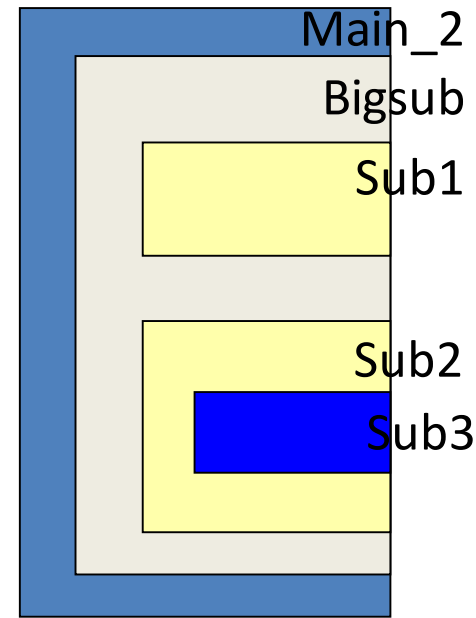
38

```
program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
        A := B + C;  <-----1
      end; { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
          SUB1;
          E := B + A;  <-----2
        end; { SUB3 }
      begin { SUB2 }
        SUB3;
        A := D + E;  <-----3
      end; { SUB2 }
    begin { BIGSUB }
      SUB2(7);
    end; { BIGSUB }
  begin
    BIGSUB;
  end. { MAIN_2 }
```

```

program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
        A := B + C;  <-----1
      end; { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
          SUB1;
          E := B + A;
        end; { SUB3 }
      begin { SUB2 }
        SUB3;
        A := D + E;
      end; { SUB2 }
    begin { BIGSUB }
      SUB2(7);
    end; { BIGSUB }
  begin
    BIGSUB;
  end. { MAIN_2 }

```



MAIN\_2 calls BIGSUB

BIGSUB calls SUB2(7)

SUB2 calls SUB3;

SUB3 calls SUB1

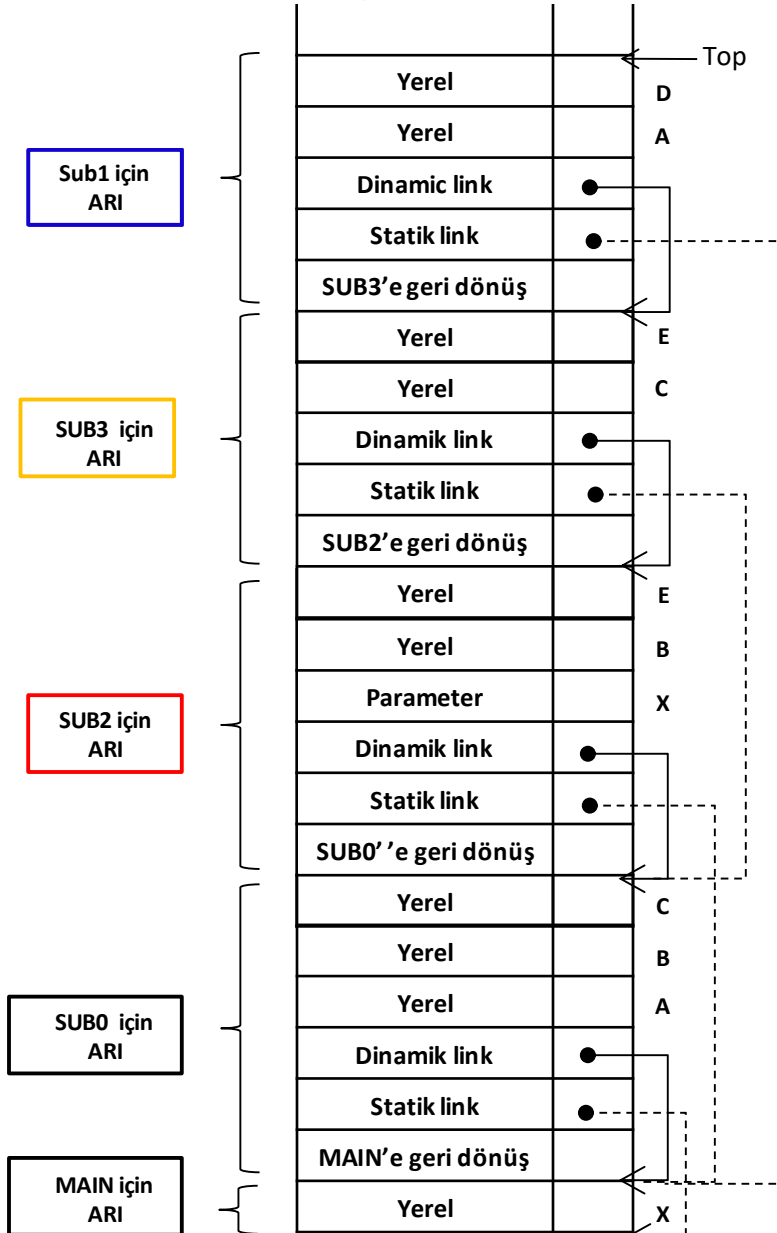
**A** - (0, 3)

**B** - (1, 4)

**C** - (1, 5)

(zincir\_bağılkonum, yerel\_bağılkonum)

# Pozisyon 1'de Stack (SUB1 içinde)



MAIN { tanımlar: X

SUB0 { tanımlar: A B C

SUB1 { tanımlar: A D; kullanır: A B C }

SUB2 { tanımlar: X B E; kullanır: A D E

SUB3 { tanımlar : C E; kullanır: A B E }

}

}

}

**MAIN**

**MAIN** calls **SUB0**

**SUB0** calls **SUB2**

**SUB2** calls **SUB3**

**SUB3** calls **SUB1**



# Örnek Pascal Programı

41

```
program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
        A := B + C;
      end; { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
          SUB1;
          E := B + A; <-----2
        end; { SUB3 }
      begin { SUB2 }
        SUB3;
        A := D + E;
      end; { SUB2 }
    begin { BIGSUB }
      SUB2(7);
    end; { BIGSUB }
  begin
    BIGSUB;
  end. { MAIN_2 }
```

in SUB3:

**E** - (0, 4)

**B** - (1, 4)

**A** - (2, 3)

```

program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
        A := B + C;
      end; { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
          SUB1;
          E := B + A;
        end; { SUB3 }
      begin { SUB2 }
        SUB3;
        A := D + E; <-----3
      end; { SUB2 }
    begin { BIGSUB }
      SUB2(7);
    end; { BIGSUB }
begin
  BIGSUB;
end. { MAIN_2 }

```

In SUB2:

**A** - (1, 3)

**D** - an error

**E** - (0, 5)

```

program MAIN;
  var X : integer;
  procedure SUB0;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
        A := B + C;
      end; { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
          SUB1;
          E := B + A;
        end; { SUB3 }
      begin { SUB2 }
        SUB3;
        A := D + E;
      end; { SUB2 }
    begin { SUB0 }
      SUB2(7);
    end; { SUB0 }
  begin
    SUB0;
  end. { MAIN }

```

<-----1

<-----2

<-----3

Pozisyon 1 SUB1:

A - (0, 3)

B - (1, 4)

C - (1, 5)

Pozisyon 2 SUB3:

E - (0, 4)

B - (1, 4)

A - (2, 3)

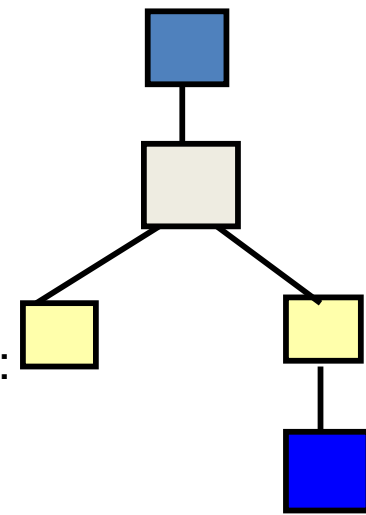
Pozisyon 3 SUB2:

A - (1, 3)

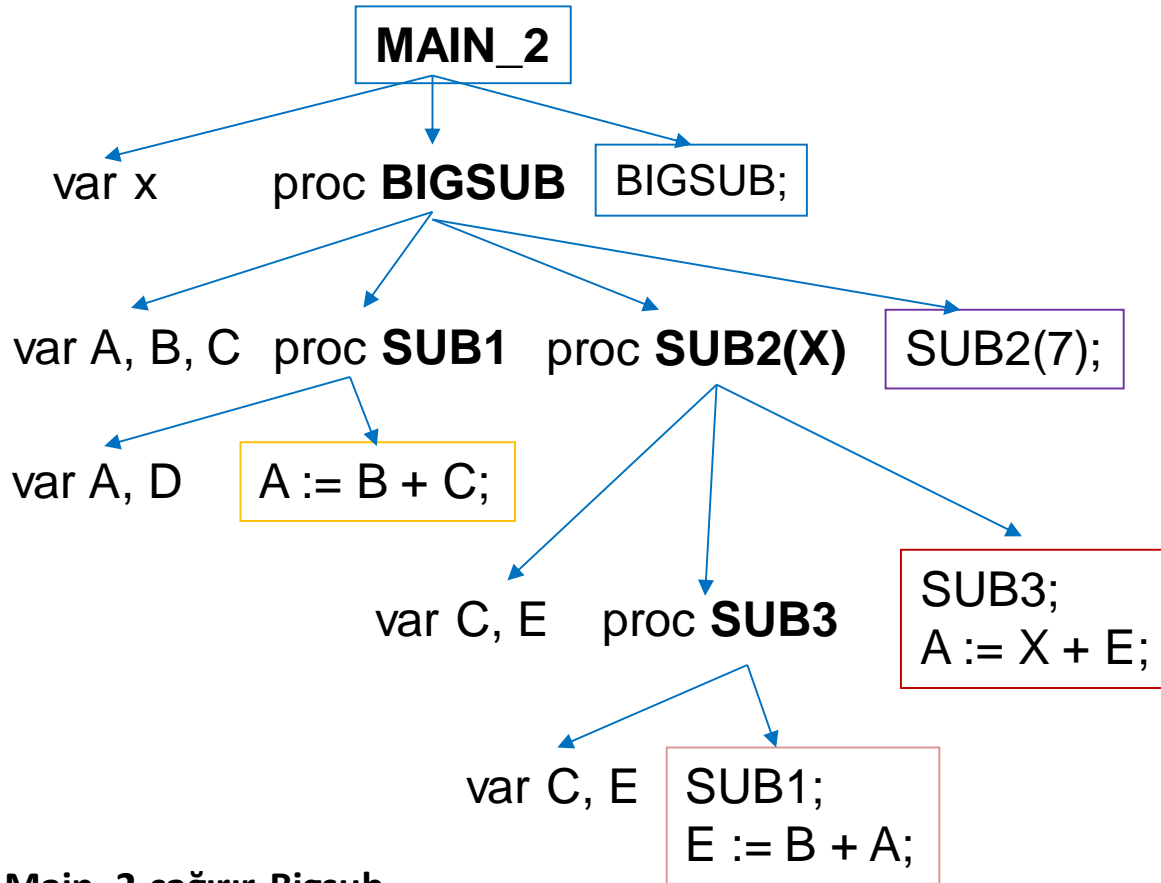
D - an error (sub1 içinde tanımlı)

E - (0, 5)

(zincir\_bağılkonum, yerel\_bağılkonum)



# Yerel olmayan referans örneği

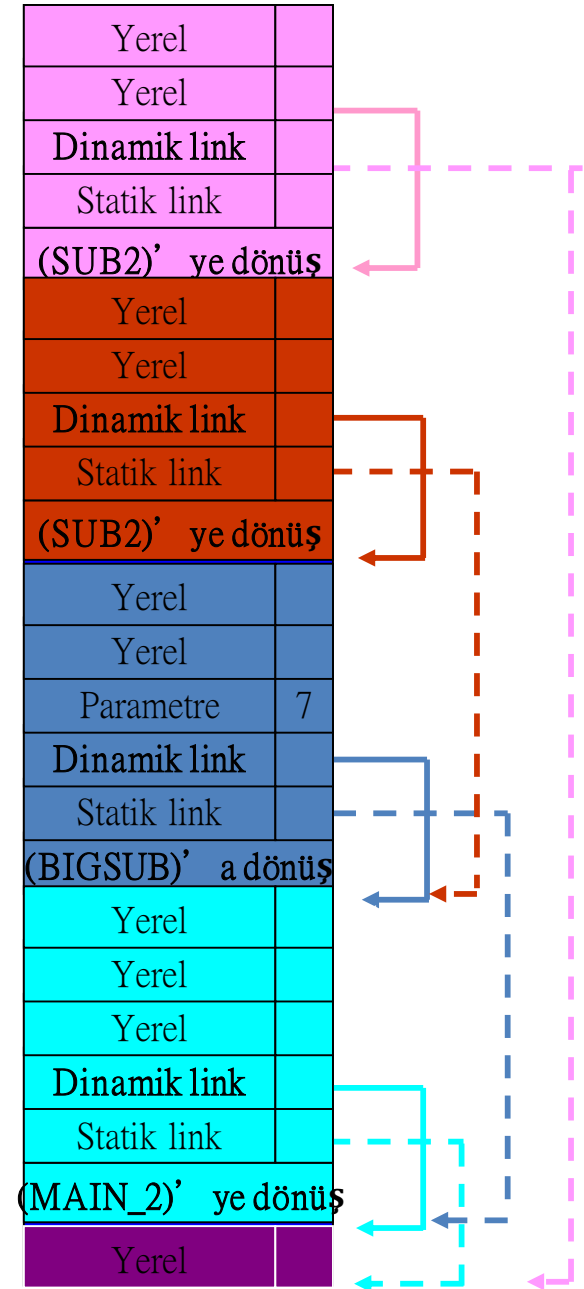


Main\_2 çağırır Bigsub

Bigsub çağırır sub2

Sub2 çağırır sub3

Sub3 çağırır sub1



# 1. Teknik – Statik zincir

45

## □ Statik zincir uygulaması

- ▣ Altprogram çağrıldığı zaman (altprogram parametre ve isimle geçilen parametre olmadığını varsayarak) :

- Gerçekleştirme kaydı somutlaşan örneğini (instance) hazırla.
- Eski yığıt bellek tepesine bir dinamik link.
- Statik link, statik ebeveynin en son yaratılmış etkinleştirme kaydını gösterir.

## ▣ İki metot:

1. Statik ebeveynin ilk etkinleştirme kaydını bulmak için dinamik zincirde ara. Kolay fakat yavaş.
2. Derleyici, derleme esnasında çağırana, çağrılan arasındaki derinliği (zincir bağıl konum) hesaplar ve daha sonra program yürütülürken kullanılmak üzere saklar.
  - Örneğin **MAIN\_2** nin yığıt içeriğine bakarsak: **SUB1**, **SUB3**'ün içinden çağırılırken derleyici **SUB3**'ün **SUB1**'in tanımlandığı altprograma (**Bigsb**) göre derinliğinin iki olduğu bilgisini programa yerleştirir. Program çalışırken bu noktada yerleştirilen bilgi kullanılarak etkinleştirme kaydına olan statik link kurulur.

# 1. Teknik – Statik zincir

46

## □ Statik zincir metodunun değerlendirilmesi

### ■ Sorunlar:

1. Yerel olmayan referanslar, özellikle derinlik fazlaysa, yavaş çalışır. Programın performansı düşer.
2. Yerel olmayan referansların kullandığı zaman eşit olmadığından, zamana hassas program yazmak zorlaştığı gibi bunların güncellenmesi de bu zamanlamayı değiştirir.

# İç içe altprogramlar:

## 2. Teknik - ekranlar

47

- 2. Teknik – ekranlar
  - ▣ Fikir: Statik linkleri "ekran" (display) denilen ayrı bir yığıtaya (stack) yerleştir. Ekrandaki veriler etkinleştirme kodlarına (activation record) göstericilerdir.
  - ▣ Referansları:  
(ekran\_bağılkonum, yerel\_bağılkonum) (display\_offset, local\_offset) olarak gösterebiliriz. Burada ekran\_bağılkonum değeri zincir\_bağılkonumla aynıdır.
- Referansların işleyişi:  
ekran\_bağılkonum'u kullanarak, aranılan değişkenin doğru etkinleştirme kaydının göstericisini bul.  
yerel\_bağılkonum değerini kullanarak, etkinleştirme kaydı içinde aranılan değişkeni bul.

# Ekranlar (Displays)

48

- Ekran uygulaması (altprogram parametre ve isimle geçilen parametre olmadığını varsayarak) :

Not: ekran\_bağılkonum'u sadece etkinleştirme kaydı türetilmekte olan altprogramın statik derinliğine bağlıdır. Bu altprogramın statik derinliğidir.

- Belli bir anda çalışan altprogramın statik derinliği  $k$  ise, ekranda  $k+1$  gösterici olur ( $k=0$  ana program).



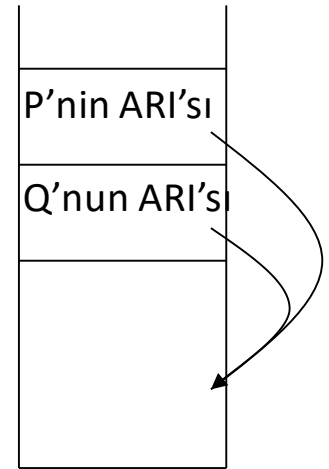
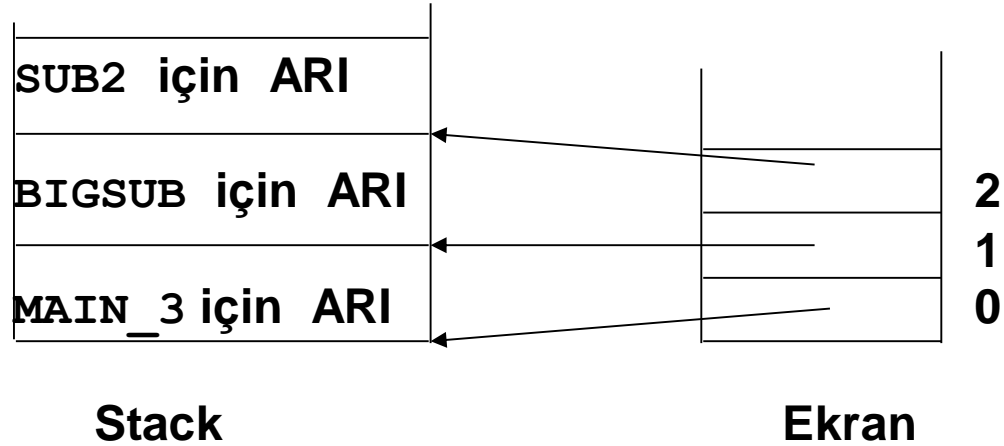
- Psd P'nin static\_depth'i ve Qsd Q'nun static\_depth'i olsun
- Q, P'yi çağırırsın
- Üç olası durum söz konusu:
  1. Qsd = Psd
  2. Qsd < Psd
  3. Qsd > Psd

Örnek iskelet program:

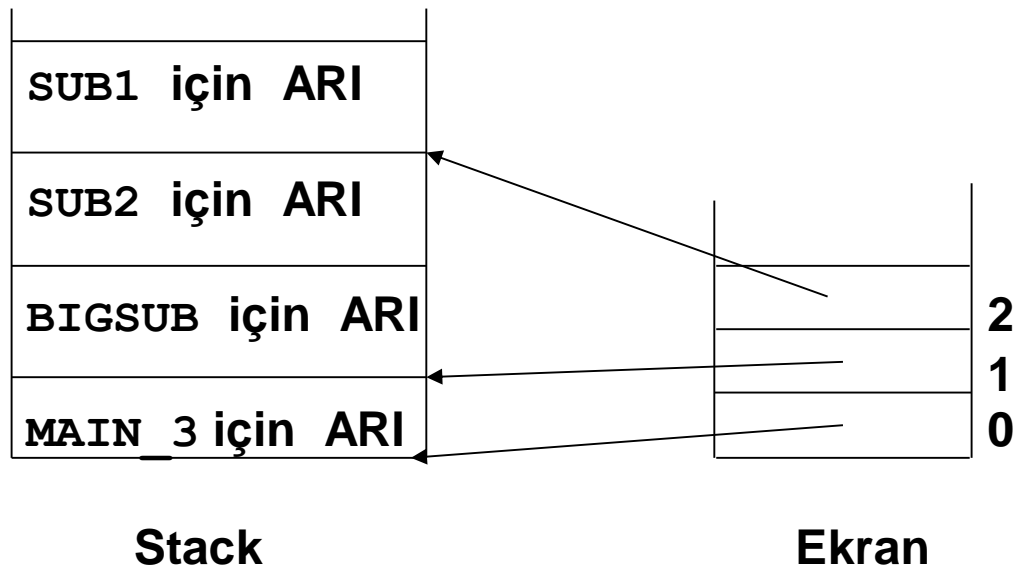
```
program MAIN_3;  
  procedure BIGSUB;  
    procedure SUB1;  
      end; { SUB1 }  
    procedure SUB2;  
      procedure SUB3;  
        end; { SUB3 }  
      end; { SUB2 }  
    end; { BIGSUB }  
  end. { MAIN_3 }
```

MAIN\_3 tüm üç durumu örnekler

**Durum 1: SUB2 , SUB1 \ i çağırır**  
**Çağırmadan önce:**

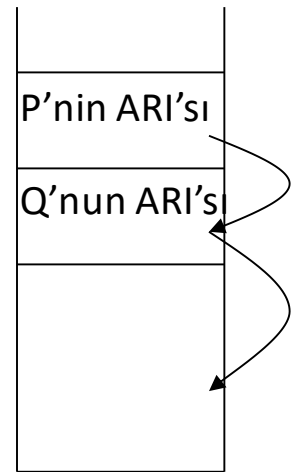
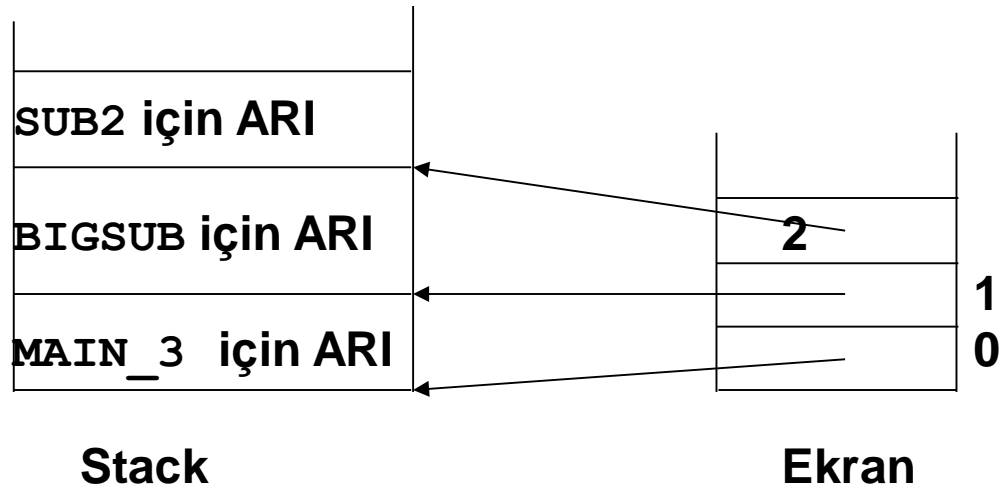


**Çağırmadan sonra:**

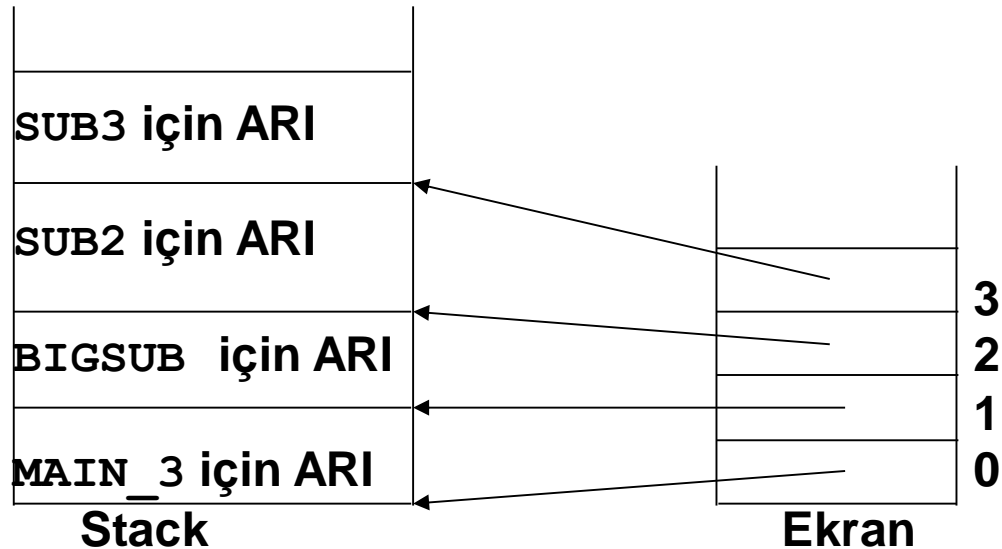


**Durum 2: SUB2, SUB3 \ i çağırır**

**Çağırmadan önce:**

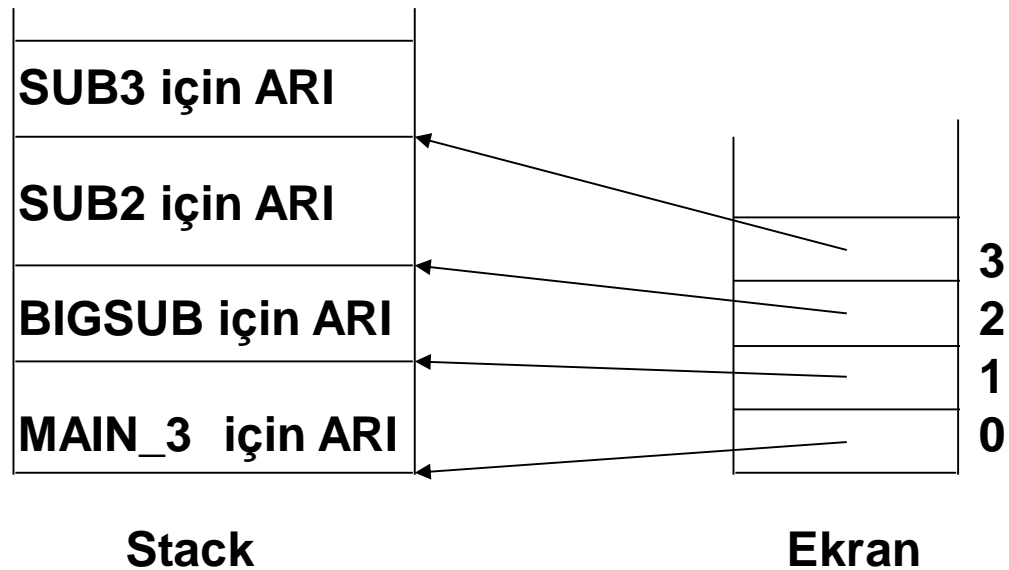


**Çağırmadan sonra :**

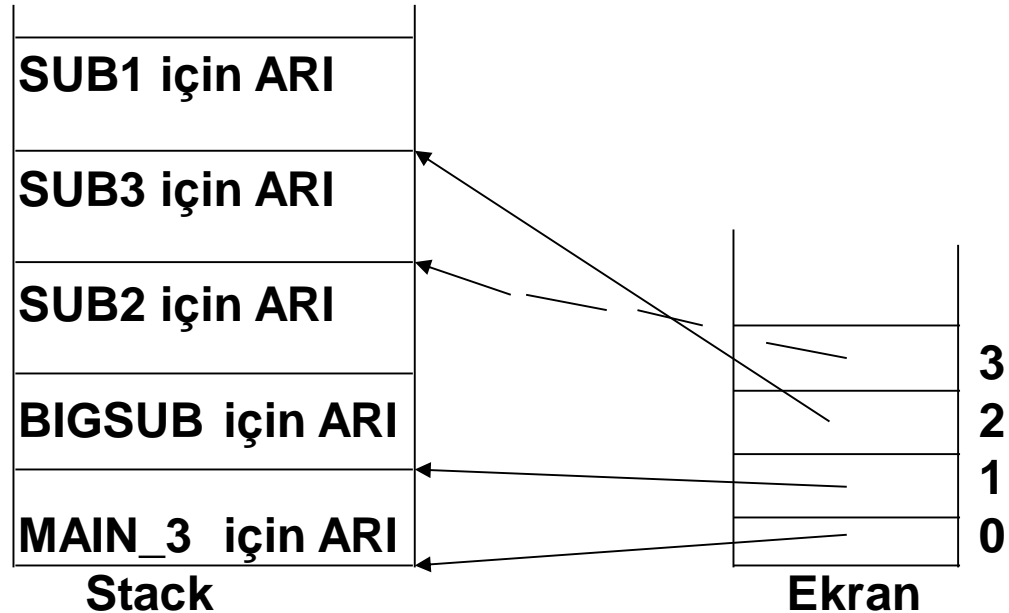


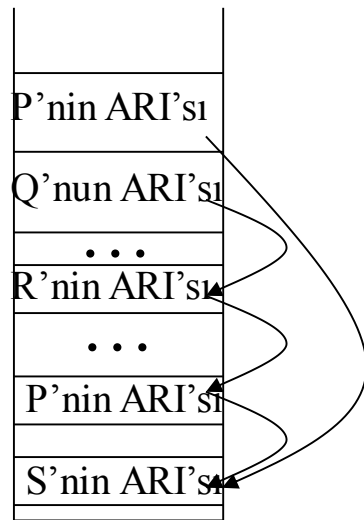
**Case 3:** SUB3, SUB1' i çağırır

**Çağırmadan önce:**

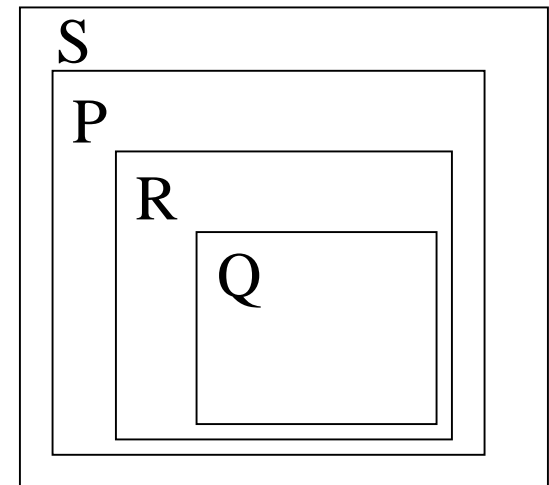


**Çağırmadan sonra :**





n 2'de olsun



# Bloklar

54

- ❑ Bloklar, değişkenler için kullanıcı odaklı yerel ölçeklerdir.

- ❑ C deki bir örnek

```
{int temp;  
  temp = list [upper];  
  list [upper] = list [lower];  
  list [lower] = temp  
}
```

- ❑ Temp in yukarıdaki örnekteki ömrü kontrolün bloku girmesi ile beraber başlar.
- ❑ Temp gibi bir yerel değişkeni kullanmanın bir avantajı şudur: aynı isimli başka bir değişkenle çakışmaz

# Blokların Uygulanması

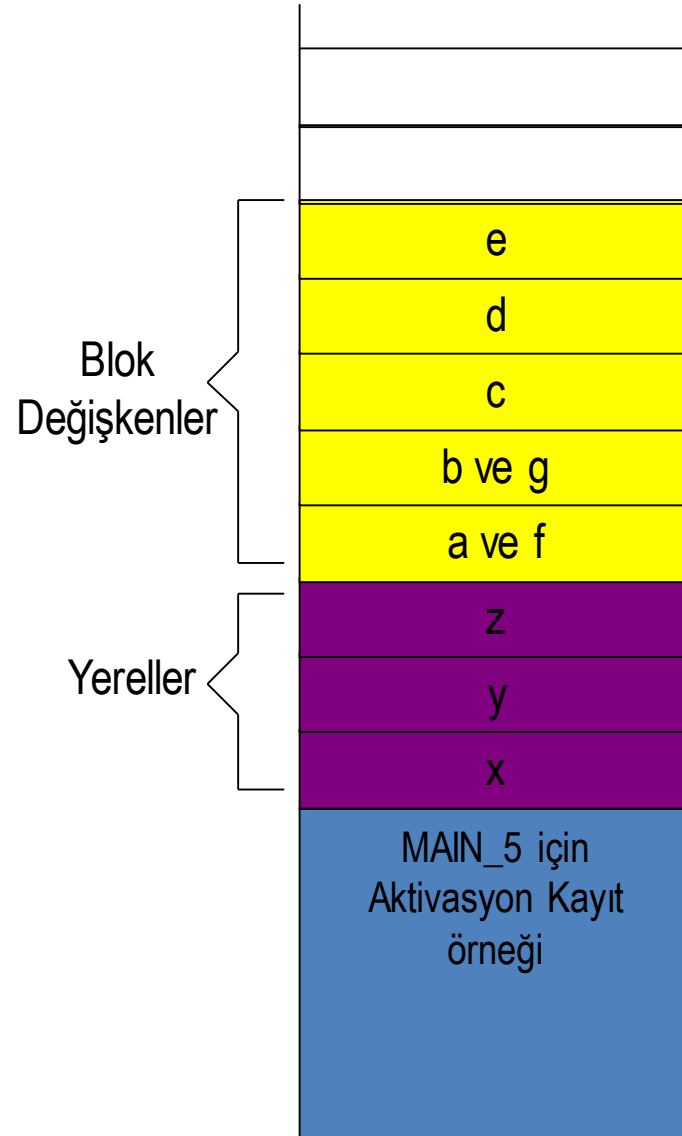
55

## □ İKİ YÖNTEM

1. Bloklara daima aynı bölgeden çağrılan alt programlardan ziyade bir parametre olarak davranın
  - her blokun bir aktivasyon kaydı vardır: blok her çalıştığında bunun bir örneğini oluşturur.
2. Bir blok için gereken maksimum depo statik olarak tespit edilebildiği için, bu gereken alan aktivasyon kaydındaki yerel değişkenlerden sonra ayrılır.

# Bloklar

```
MAIN_5() {  
    int x, y, z;  
    while ( ... ) {  
        int a, b, c;  
        ...  
        while ( ... ) {  
            int d, e;  
        }  
    }  
    while ( ... ) {  
        int f, g;  
    }  
}
```





# Dinamik Kapsamın Uygulanması

57

\* *Derin Erişim*: yerel olmayan referanslar dinamik zincirdeki aktivasyon kayıt örneklerinin araştırılmasıyla bulunur.

- zincirin uzunluğu statik olarak belirlenemez.
- her aktivasyon kayıt örneği değişken isimlere sahip olmalıdır.

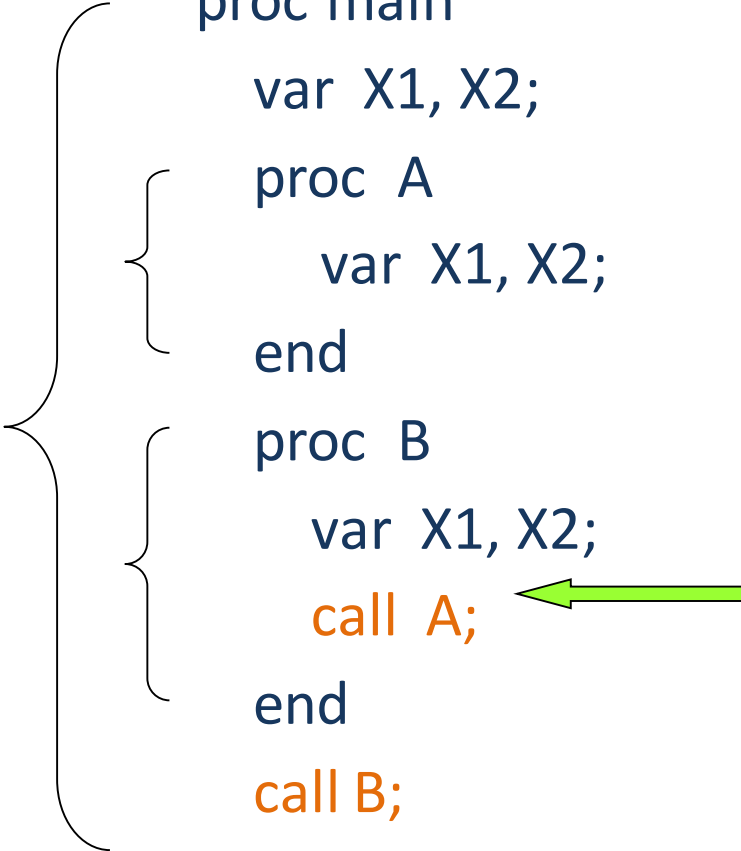
\* *Sığ Erişim*: yerelleri merkezi bir yere koy

- her bir değişken için bir yığın
- her bir değişken isim için içinde girdi bulunan merkezi bir çizelge

# Kapsam Örneği

58

```
proc main
  var X1, X2;
  {
    proc A
      var X1, X2;
    end
  }
  {
    proc B
      var X1, X2;
      call A;
    end
  }
  call B;
end
```



Proc A referans  
çevresi

Statik kapsam:

A → main

Dinamik kapsam:

A → B → main

# Derin Erişim

```

Procedure C;
  integer x, z;
  begin
    x := u + v;

```

...

```

  end;

```

```

Procedure B;
  integer w, x;
  begin

```

...

```

  end;

```

```

Procedure A;
  integer v, w;
  begin

```

...

```

  end;

```

```

Program MAIN_6
  integer v, u;
  begin

```

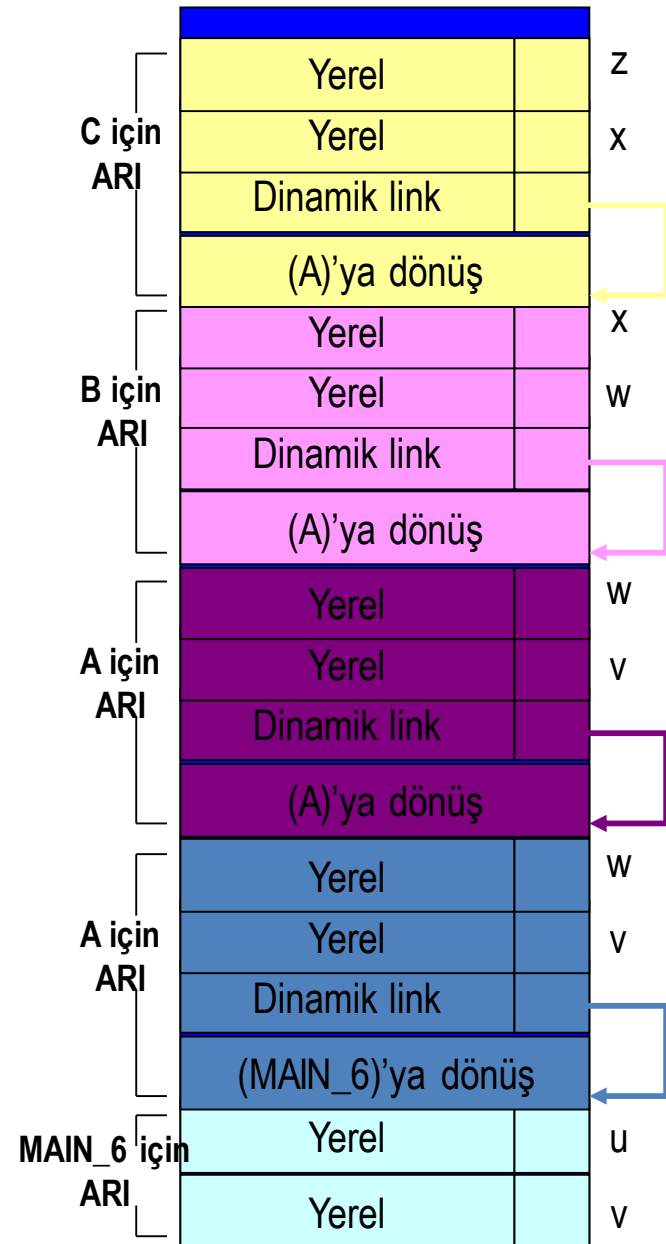
...

```

  end;

```

*MAIN\_6 calls A*  
*A calls A*  
*A calls B*  
*B calls C*



# Sığ Erişim

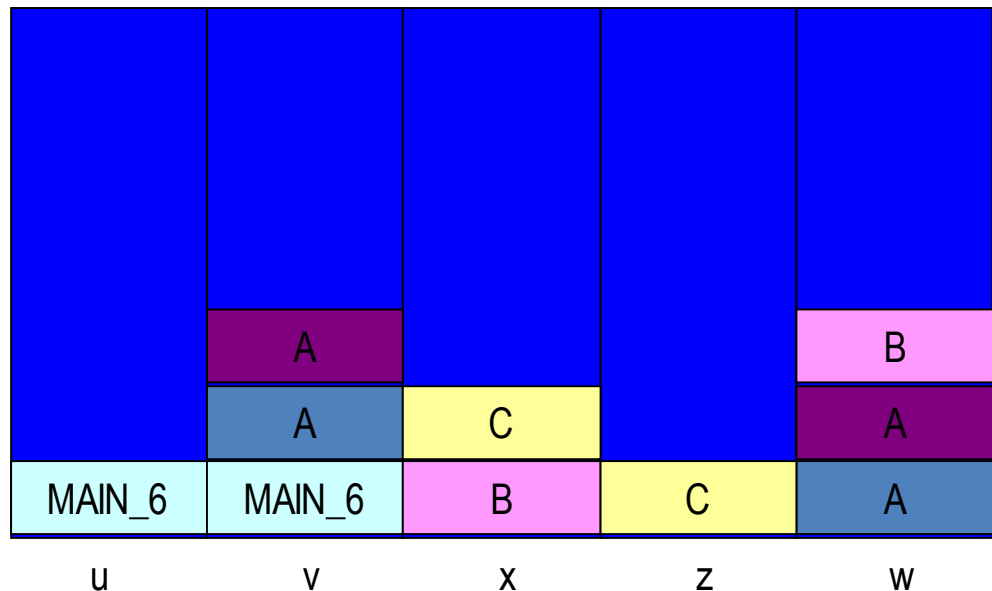
```
Procedure C;  
  integer x, z;  
  begin  
    x := u + v;  
    ...  
  end;
```

```
Procedure B;  
  integer w, x;  
  begin  
    ...  
  end;
```

```
Procedure A;  
  integer v, w;  
  begin  
    ...  
  end;
```

```
Program MAIN_6  
  integer v, u;  
  begin  
    ...  
  end;
```

*MAIN\_6 calls A*  
*A calls A*  
*A calls B*  
*B calls C*

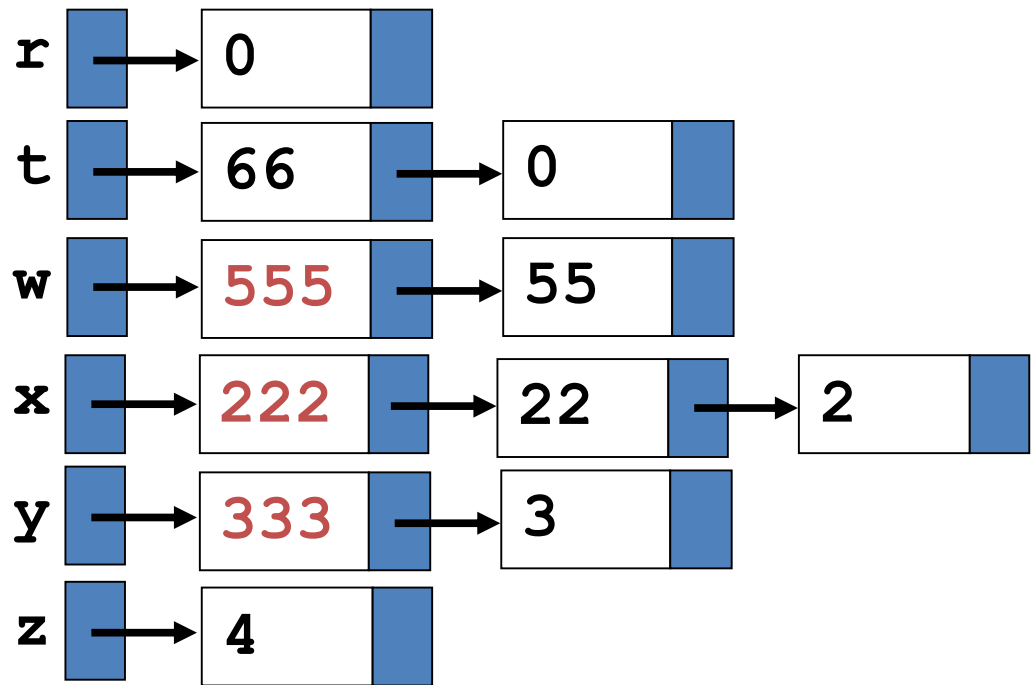


# Değişken Stack'larla Sığ Erişim

61

```
proc A
  var x=2, y=3, z=4
begin
  call B
end
proc B
  var x=22, w=55, t=66
begin
  call C
end
proc C
  var x=222, y=333, w=555
begin
  call B
end
main
  var r=0, t=0
begin
  call A
end
```

C'de while:



# Arabellek aşımı saldırısı

62

- Bir bilgisayarda işlem yaptığınızda bilgisayar söz konusu işlemi kimi zaman direkt olarak Sabit Disk'e (HDD) ya da ilgili depolama birimine yazmaz. Bunun yerine Bellek (RAM) üzerinde geçici olarak konumlandığı ve ileride topluca yazılacağı bir alanda tutar. Burası ara bir bellektir. İşte bu alana "buffer" ya da tampon yahut arabellek diyoruz.
- Arabellek belirli bir alana yani boyuta sahiptir. Ancak bu alan bazen programsal hata (bug) sonucunda haddinden fazla bilgi ile yüklenir. Bazen de kötü niyetli bir saldırgan söz konusu yol ile bu alana haddinden fazla büyüklükte bilgi yükleyebilir. Bu haddinden fazla yükleme söz konusu alanda taşkına sebep olur.
- İşleyişi;  
Bir saldırgan bu sorunu kendi çıkarına kullanarak söz konusu alana alabileceğinden fazla uzunlukta bir değer yükler bunun sonucunda artan kısım yani taşan kısım direk olarak çalıştırılabilir hale gelir. Bu taşan kısma saldırgan çalıştırabilir bir kod yüklediğinde amacına erişmiş ve Arabellek Aşımını kullanıp içeride (uzaktaki bir PC'de ya da sunucuda) kötü niyetli bir kod çalıştırmış olur.

# Arabellek aşımı saldırısı

63

- Arabellek aşımı saldırısı exploits a common çoğu programdaki genel bir problemi kötüye kullanır.
- C gibi çoğu yüksek seviye programlama dilinde, “sınır testi”, yani kopyaladığınız değişkenin uzunluğunun beklediğiniz gibi olduğunu görmeyi test etme, yapılmaz.

```
void main(){  
    char bufferA[256];  
    myFunction(bufferA);  
}
```

```
void myFunction(char *str)  
{  
    char bufferB[16];  
    strcpy(bufferB, str);  
}
```

# Arabellek aşımı saldırısı

64

```
void main(){  
    char bufferA[256];  
    myFunction(bufferA);  
}
```

```
void myFunction(char *str)  
{  
    char bufferB[16];  
    strcpy(bufferB, str);  
}
```

- main(), 256 baytlık bir diziye myFunction()'a parametre olarak gönderir, ve myFunction() fonksiyonu içinde ise dizilerin sınırlarını kontrol etmeden, strcpy() ile uzun dizinin tamamını 16 baytlık dizi olup tasana kadar kopyalar!
- bufferB'nin yeterince büyük olup olmadığı ile ilgili bir test olmadığından, ekstra veri bellekte diğer bilinmeyen diğer yerlerin üstüne yazar.
- Bu savunmasızlık arabellek aşımı saldırısının temelidir
  - Sistem stack'ı değiştirilir



# Arabellek aşımı saldırısı

65

```
void main() {  
    char bufferA[256];  
    myFunction(bufferA);  
}
```

Stack içeriği



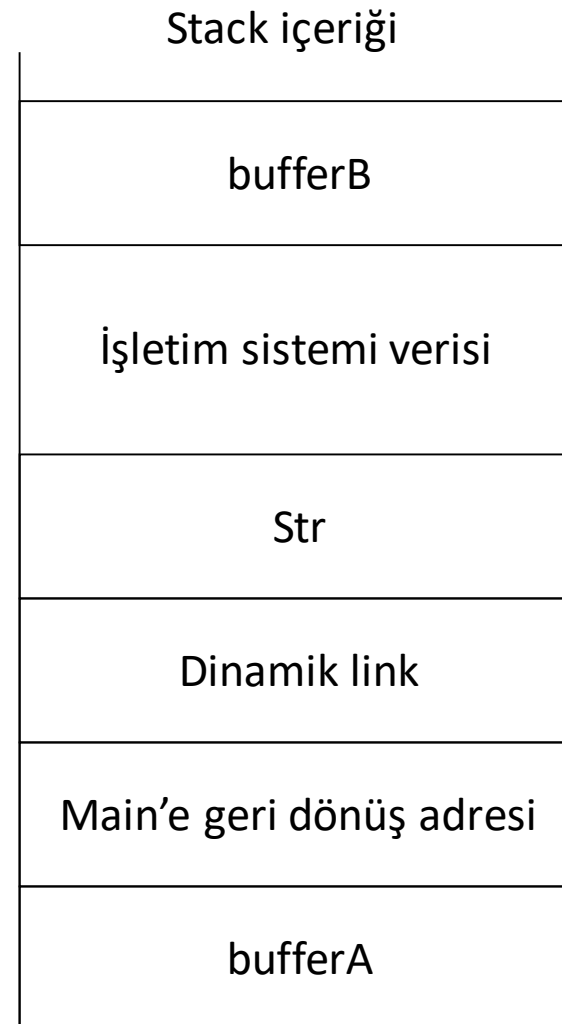
bufferA

# Arabellek aşımı saldırısı

66

```
void main() {  
    char bufferA[256];  
    myFunction(bufferA);  
}
```

```
void myFunction(char *str) {  
    char bufferB[16];  
    strcpy(bufferB, str);  
}
```



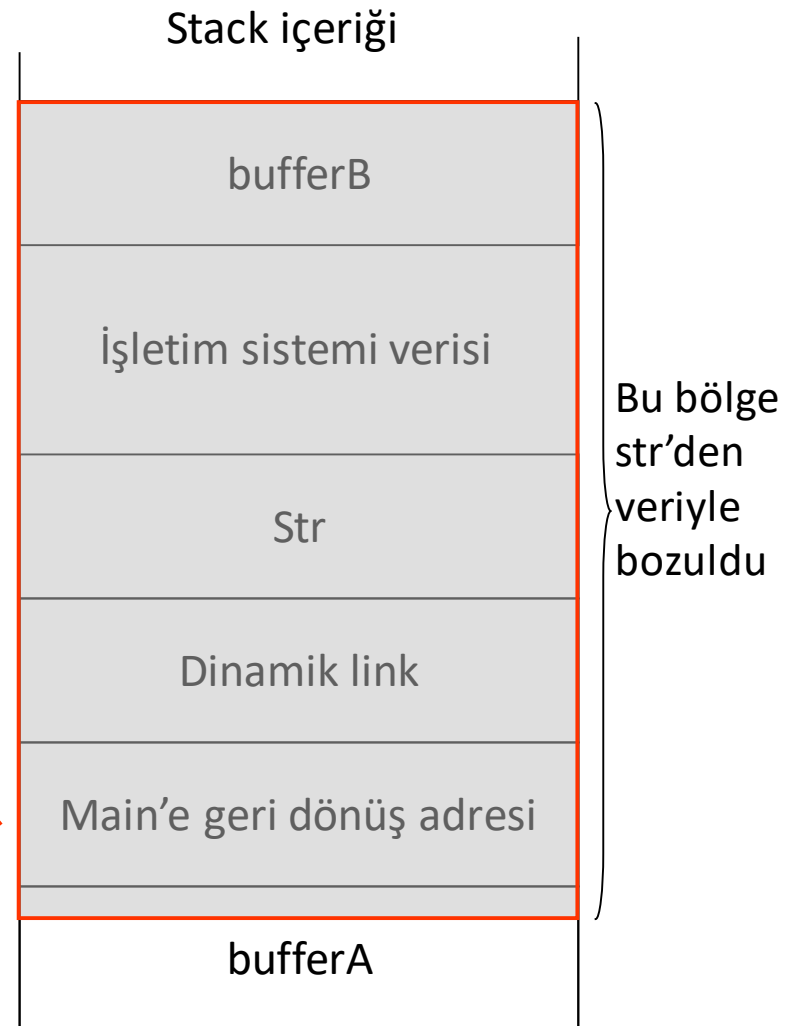
# Arabellek aşımı saldırısı

67

```
void main() {  
    char bufferA[256];  
    myFunction(bufferA);  
}
```

```
void myFunction(char *str) {  
    char bufferB[16];  
    strcpy(bufferB, str);  
}
```

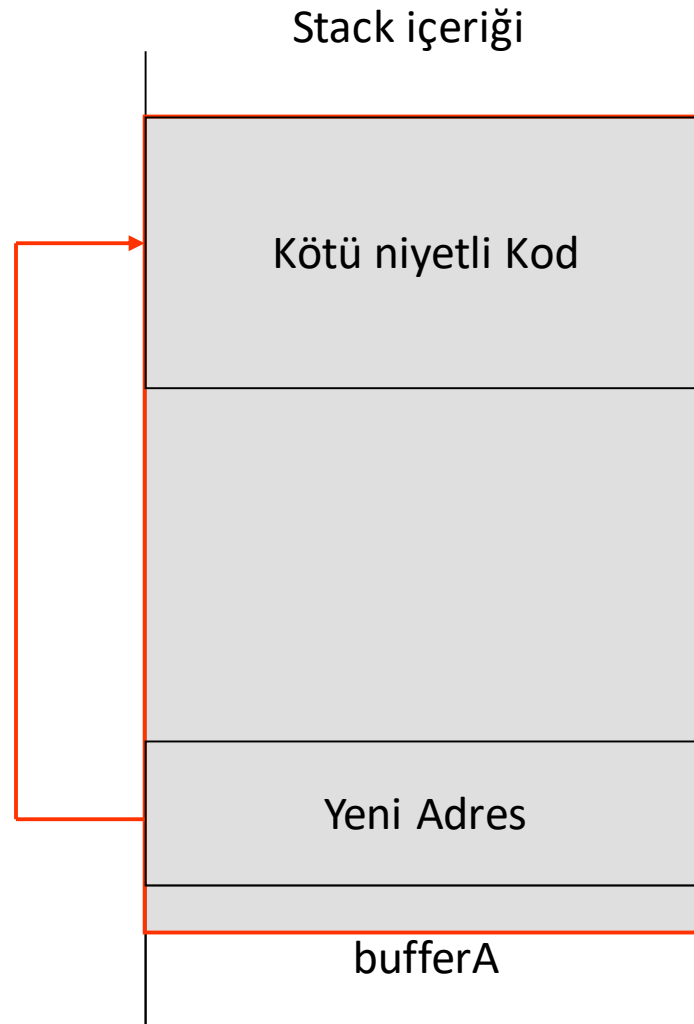
Geri dönüş adresi  
üzerine yazabilir!!



# Arabellek aşımı saldırısı

68

- str içeriği dikkatlice seçilirse, yazmış olduğumuz kodun bir kısmına dönüş adresini gösterebiliriz
- Sistem fonksiyon çağrısından döndüğünde zararlı kodu çalıştırarak başlayacaktır



# Muhtemel bir çözüm

69

```
void main() {  
    char bufferA[256];  
    myFunction(bufferA,  
256);  
}
```

```
void myFunction(char *str,  
int len)  
{  
    char bufferB[16];  
    if (len <= 16)  
        strcpy(bufferB,  
str);  
}
```

# Kaynaklar

70

- 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, Uzaktan Eğitim Notları
- Erkan Tanyıldızı, Programlama Dilleri Ders Notları
- David Evans, Programming Languages Lecture Notes
- O. Nierstrasz, Programming Languages Lecture Notes
- Giuseppe Attardi, Programming Languages Lecture Notes
- John Mitchell, Programming Languages Lecture Notes