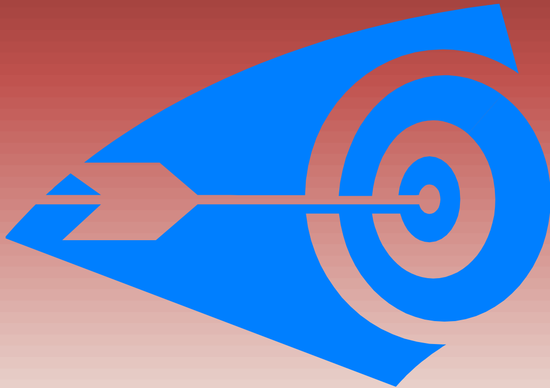
İstanbul Üniversitesi
Elektrik Elektronik Mühendisliği

BİLİNİRLİK ALANI ve ÖMÜR, KONTROL DEYİMLERİ



Kaynak: C ve Sistem Programcıları Derneği Kurs notu

Öğr.Gör.Dr. Mahmut YALÇIN

Bilinirlik Alanı

Bilinirlik alanı (*scope), bir ismin tanınabildiği program aralığıdır. Derleyiciye bildirilen* isimler, derleyici tarafından her yerde bilinmez. Her isim derleyici tarafından ancak "o ismin bilinirlik alanı" içinde tanınabilir. Bilinirlik alanı doğrudan kaynak kod ile ilgili bir kavramdır, dolayısıyla derleme zamanına ilişkindir. C dilinde derleyici, bildirimleri yapılan değişkenlere kaynak kodun ancak belirli bölümlerinde ulaşılabilir. Yani bir değişkenin tanımlanması, o değişkene kaynak dosyanın her yerinden ulaşılabilmesi anlamına gelmez. Bilinirlik alanları C standartları tarafından 4 ayrı grupta toplanmıştır:

i. Dosya Bilinirlik Alanı (*File scope) : Bir ismin bildirildikten sonra tüm kaynak dosya* içinde, yani tanımlanan tüm işlevlerin hepsinin içinde bilinmesidir.

ii. Blok Bilinirlik Alanı (*Block scope): Bir ismin bildirildikten sonra yalnızca bir blok içinde,* bilinmesidir.

iii. İşlev Bilinirlik Alanı (*Function Scope): Bir ismin, bildirildikten sonra yalnızca bir fonksiyon* içinde bilinmesidir. Yalnızca *goto etiketlerini kapsayan özel bir tanımdır.*

iv. İşlev Bildirimi Bilinirlik Alanı (*Function Prototype Scope): İşlev bildirimlerindeki, işlev* parametre ayracı içinde kullanılan isimlerin tanınabilirliğini kapsayan bir tanımdır.

Bir kaynak dosya içinde tanımlanan değişkenler, bilinirlik alanlarına göre "yerel" ve "global" olmak üzere ikiye ayrılabilir:

Yerel Değişkenler

Blokların içinde ya da işlevlerin parametre ayraçları içinde tanımlanan değişkenlere, yerel değişkenler (*local variables) denir. C dilinde blokların içinde tanımlanan değişkenlerin* tanımlama işlemlerinin, bloğun en başında yapılması gerektiğini biliyorsunuz. Yerel değişkenler, blok içinde tanımlanan değişkenlerdir, bir işlevin ana bloğu içinde ya da içsel bir blok içinde bildirilmiş olabilirler. Yerel değişkenlerin bilinirlik alanı, blok bilinirlik alanıdır. Yani yerel değişkenlere yalnızca tanımlandıkları blok içinde ulaşılabilir. Tanımlandıkları bloğun daha dışındaki bir blok içinde bu değişkenlere erişilemez.

Aşağıdaki programda tanımlanan değişkenlerin hepsi yereldir. Çünkü *x, y, z isimli* değişkenler blokların içinde tanımlanıyor. Bu değişkenler yalnızca tanımlanmış oldukları blok içinde kullanılabilir. Tanımlandıkları blok dışında bunların kullanılması geçersizdir.

Yorum satırları içine alınan deyimler geçersizdir. *z ve y değişkenleri bilinirlik alanlarının* dışında kullanılmıştır. Aşağıdaki örnekte değişkenlerin hepsi yerel olduğu için blok bilinirlik alanı kuralına uyar, ancak bu durum, 3 değişkenin de bilinirlik alanının tamamen aynı olduğu anlamına gelmez.

```c
#include <stdio.h>
int main()
{
int x = 10;
printf("x = %d\n", x);
{
int y = 20;
printf("y = %d\n", y);
x = 30;
{
int z = 50;
y = 60;
printf("z = %d\n", z);
printf("x = %d\n", x);
printf("y = %d\n", y);
}
z = 100; /* Geçersiz! */
y = x;
printf("x = %d\n", x);
printf("y = %d\n", y);
}
y = 500; /* Geçersiz! */
printf("x = %d\n", x);
return 0;
}
```

Bu programda *x değişkeni en geniş bilinirlik alanına* sahipken *y değişkeni daha küçük ve z değişkeni de en küçük bilinirlik alanına sahiptir.*
İşlevlerin parametre değişkenleri de *(formal parameters), blok bilinirlik alanı kuralına* uyar. Bu değişkenler işlevin ana bloğu içinde bilinir. İşlev parametre değişkeninin bilinirlik alanı, işlevin ana bloğunun kapanmasıyla sonlanır. Yani işlev parametre değişkeninin bilinirlik alanı, işlevin ana bloğudur.

```
void func (int a, double b)
{
/* a ve b bu işlevin her yerinde bilinir. */
}
```

Yukarıdaki örnekte *func işlevinin parametre değişkenleri olan a ve b isimli değişkenler, func işlevinin her yerinde kullanılabilir.*

Global Değişkenler

C dilinde blokların dışında da değişkenlerin tanımlanabileceğini biliyorsunuz. Blokların dışında tanımlanan değişkenler *"global değişkenler" (global variables) olarak* isimlendirilir.
Derleme işleminin bir yönü vardır. Bu yön kaynak kod içinde yukarıdan aşağıya doğrudur. Bir değişken yerel de olsa global de olsa, tanımlaması yapılmadan önce kullanılması geçersizdir. Global değişkenler tanımlandıkları noktadan sonra kaynak dosyanın sonuna kadar her yerde bilinir:

```
#include <stdio.h>
int g;
void func()
{
g = 10;
}
int main()
{
g = 20;
printf("g = %d\n", g); /* g = 20 */
func();
printf("g = %d\n", g); /* g = 10 */
return 0;
}
```

Yukarıdaki örnekte *g değişkeni blok dışında tanımlandığı için -ya da hiçbir işlevin içinde* tanımlanmadığı için- global değişkendir. *g değişkeninin bilinirlik alanı, dosya bilinirlik* alanıdır. Yani *g değişkeni, tanımlandıktan sonra tüm işlevlerin içinde kullanılabilir.* Yukarıdaki programda önce *g global değişkenine 20 değeri atanıyor.Daha sonra bu değer printf işleviyle ekrana yazdırılıyor. Daha sonra func işlevi çağrılıyor. func işlevi çağrılınca* kodun akışı *func işlevine geçer.*

Aynı İsimli Değişkenler

C dilinde aynı isimli birden fazla değişken tanımlanabilir. Genel kural şudur: İki değişkenin bilinirlik alanları aynı ise, bu değişkenler aynı ismi taşıyamaz. Aynı ismi taşımaları derleme zamanında hata oluşturur. İki değişkenin bilinirlik alanlarının aynı olması ne anlama gelir? İki değişkenin bilinirlik alanları, aynı kapanan küme ayracı ile sonlanıyorsa, bu değişkenlerin bilinirlik alanları aynı demektir.

```
{
float a;
int b;
double a; /* Geçersiz */
{
int c;
/*...*/
}
}
```

Yukarıdaki kod geçersizdir. Çünkü her iki *a değişkeninin de bilinirlik alanı aynıdır.* Farklı bilinirlik alanlarına sahip birden fazla aynı isimli değişken tanımlanabilir.

```
#include <stdio.h>
int main()
{
int x = 100;
printf("%d\n", x);
{
int x = 200;
printf("%d\n", x);
{
int x = 300;
printf("%d\n", x);
}
}
return 0;
}
```

Yukarıdaki program parçasında bir hata bulunmuyor. Çünkü her üç *x değişkeninin de* bilinirlik alanları birbirlerinden farklıdır. Peki yukarıdaki örnekte iç bloklarda *x ismi* kullanıldığında derleyici bunu hangi *x değişkeni ile ilişkilendirir?* Bir kaynak kod noktası, aynı isimli birden fazla değişkenin bilinirlik alanı içinde ise, bu noktada değişkenlerden hangisine erişilir?

Derleyici, bir ismin kullanımı ile karşılaştığında bu ismin hangi yazılımsal varlığa ait olduğunu bulmaya çalışır. Bu işleme "isim arama" (*name lookup) denir. İsim arama, dar* bilinirlik alanından geniş bilinirlik alanına doğru yapılır. Yani derleyici söz konusu ismi önce kendi bloğunda arar. Eğer isim, bu blok içinde tanımlanmamış ise bu kez isim kapsayan bloklarda aranır. İsim, kapsayan bloklarda da bulunamaz ise bu kez global isim alanında aranır. Dar bilinirlik alanına sahip isim, daha geniş bilinirlik alanında yer alan aynı ismi maskeler, onun görünmesini engeller. Aşağıdaki programı inceleyin:

```
void func1()
{
int k;
/***/
}
void func2()
{
int k;
/***/
}
void func3()
{
int k;
 /***/
}
```

Aynı isimli hem bir global hem de bir yerel değişkene erişilebilen bir noktada, erişilen yerel değişken olur. Çünkü aynı bilinirlik alanında, birden fazla aynı isimli değişken olması durumunda, o alan içinde en dar bilinirlik alanına sahip olanına erişilebilir. Aşağıdaki kodu inceleyin:

```
#include <stdio.h>
int g = 20; /* g global değişken */
void func()
{
/* global g değişkenine atama yapılıyor. */
g = 100;
/* global g değikeninin değeri yazdırılıyor. */
printf("global g = %d\n", g);
}
int main()
{
int g; /* g yerel değişken */
/* yerel g değişkenine atama yapılıyor */
g = 200;
/* yerel g yazdırılıyor. */
printf("yerel g = %d\n", g);
func();
/* yerel g yazdırılıyor. */
printf("yerel g = %d\n", g);
return 0;
}
```

Nesnelerin Ömürleri

Ömür *(storage duration / lifespan), nesnelerin, programın çalışma zamanı içinde bellekte* yer kapladığı süreyi anlatmak için kullanılan bir terimdir. Bir kaynak kod içinde tanımlanmış değişkenlerin hepsi, program çalışmaya başladığında aynı zamanda yaratılmaz. Programlarda kullanılan varlıklar, ömürleri bakımından üç gruba ayrılabilir:

1. Statik ömürlü varlıklar
2. Otomatik ömürlü varlıklar
3. Dinamik Ömürlü varlıklar

i. Statik Ömürlü Varlıklar

Statik ömürlü varlıklar *(static duration – static storage class), programın çalışmaya* başlamasıyla bellekte yerlerini alır, programın çalışması bitene kadar varlıklarını sürdürür, yani bellekte yer kaplar. Statik ömürlü varlıklar, genellikle amaç kod (*.obj) içine* yazılır.

ii. Otomatik Ömürlü Varlıklar

Otomatik ömürlü nesneler programın çalışmasının belli bir zamanında yaratılan, belli süre etkinlik gösterdikten sonra yok olan, yani ömürlerini tamamlayan nesnelerdir. Bu tür nesnelerin ömürleri, programın toplam çalışma süresinden kısadır.

Yerel değişkenler, otomatik ömürlüdür. Programın çalışma zamanında tanımlandıkları bloğun çalışması başladığında yaratılırlar, bloğun çalışması bitince yok olurlar, yani ömürleri sona erer.

```
void func(int a, int b)
{
int result;
/***/
}
```

Yukarıdaki *func işlevinin ana bloğu içinde result isimli bir yerel değişken tanımlanıyor.* Programın çalışması sırasında *func işlevinin koduna girildiğinde result değişkeni yaratılır.* Programın akışı *func işlevinden çıktığında, result değişkeninin ömrü sona erer.*

Statik ömürlü değişkenlerle otomatik ömürlü değişkenler arasında ilkdeğer verme *(initialization) açısından da fark vardır. Statik ömürlü olan global değişkenlere de yerel* değişkenlerde olduğu gibi ilkdeğer verilebilir.

İlkdeğer verilmemiş ya da bir atama yapılmamış bir yerel değişkenin içinde bir çöp değer bulunur. Bu değer o an bellekte o değişken için ayrılmış yerde bulunan *1 ve 0 bitlerinin* oluşturduğu değerdir.

İlkdeğer verilmemiş statik ömürlü değişkenlerin *0 değeri ile başlatılması güvence* altındadır. İlk değer verilmemiş ya da bir atama yapılmamış global değişkenler içinde her zaman *0 değeri vardır.*

iii. Dinamik Ömürlü Varlıklar

Dinamik bellek işlevleri ile yerleri ayrılmış nesneler, dinamik ömürlüdür.

Global ve Yerel Değişkenlerin Karşılaştırılması

Bir programda bir değişken gereksinimi durumunda, global ya da yerel değişken kullanılması bazı avantajlar ya da dezavantajlar getirebilir. Ancak genel olarak global değişkenlerin bazı sakıncalarından söz edilebilir. Özel bir durum söz konusu değil ise, yerel değişkenler global değişkenlere tercih edilmeli, global değişkenler ancak zorunlu durumlarda kullanılmalıdır. Global değişkenler aşağıdaki sakıncalara neden olabilir:

1.Global değişkenler statik ömürlü olduklarından programın sonuna kadar bellekte yerlerini korur. Bu nedenle belleğin daha verimsiz olarak kullanılmalarına neden olurlar.

2. Global değişkenler tüm işlevler tarafından ortaklaşa paylaşıldığından, global değişkenlerin çokça kullanıldığı kaynak dosyaları okumak daha zordur.

3. Global değişkenlerin sıkça kullanıldığı bir kaynak dosyada, hata arama maliyeti daha yüksektir. Global değişkene ilişkin bir hata söz konusu ise, bu hatayı bulmak için tüm işlevler araştırılmalıdır. Tüm işlevlerin global değişkenlere ulaşabilmesi, bir işlevin global bir değişkeni yanlışlıkla değiştirebilmesi riskini de doğurur.

4. Global değişkenlerin kullanıldığı bir kaynak dosyada, değişiklik yapmak da daha fazla çaba gerektirir. Kaynak kodun çeşitli bölümleri, birbirine global değişken kullanımlarıyla sıkı bir şekilde bağlanmış olur. Bu durumda kaynak kod içinde bir yerde değişiklik yapılması durumunda başka yerlerde de değişiklik yapmak gerekir.

5. Programcıların çoğu, global değişkenleri mümkün olduğu kadar az kullanmak ister. Çünkü global değişkenleri kullanan işlevler, başka projelerde kolaylıkla kullanılamaz. Kullanıldıkları projelerde de aynı global değişkenlerin tanımlanmış olması gerekir. Dolayısıyla global değişkenlere dayanılarak yazılan işlevlerin yeniden kullanılabilirliği azalır.

İşlevlerin Geri Dönüş Değerlerini Tutan Nesneler

İşlevler geri dönüş değerlerini, geçici bir nesne yardımıyla kendilerini çağıran işlevlere iletir. Aşağıdaki programı inceleyin:

```
#include <stdio.h>
int add(int x, int y)
{
return x + y;
}
int main()
{
int a, b, sum;
printf("iki sayı girin: ");
scanf("%d%d", &a, &b);
sum = add(a, b);
printf("toplam = %d\n", sum);
return 0;
}
```

Bir işlevin geri dönüş değerinin türü aslında, işlevin geri dönüş değerini içinde taşıyacak geçici nesnenin türü demektir. Yukarıda tanımı verilen *add isimli işlevin main işlevi* içinden çağrıldığını görüyorsunuz. Programın akışı, *add işlevi içinde return deyimine* geldiğinde, geçici bir nesne yaratılır. Bu geçici nesne, *return ifadesiyle ilkdeğerini alır.*

KONTROL DEYİMLERİ

C dilinde yazılmış bir programın cümlelerine deyim(*statement) denir.*

Bazı deyimler, yalnızca derleyici programa bilgi verir. Bu deyimler derleyicinin işlem yapan bir kod üretmesine neden olmaz. Böyle deyimlere "bildirim deyimi" (*declaration statement) denir.*

Bazı deyimler derleyicinin işlem yapan bir kod üretmesine neden olur. Böyle deyimlere "yürütülebilir deyim" (*executable statement) denir.*

Yürütülebilir deyimler de farklı gruplara ayrılabilir:

Yalın Deyim:

Bir ifadenin, sonlandırıcı atom ile sonlandırılmasıyla oluşan deyimlere yalın deyim *(simple statement) denir;*

```
x = 10;
y++;
func();
```

Yukarıda 3 ayrı yalın deyim yazılmıştır.

Boş Deyim:

C dilinde tek başına bulunan bir sonlandırıcı atom ';', kendi başına bir deyim oluşturur. Bu deyime boş deyim *(null statement) denir. Boş bir blok da boş deyim oluşturur:*

```
;
{}
```

Yukarıdaki her iki deyim de boş deyimdir.

Bileşik Deyim:

Bir blok içine alınmış bir ya da birden fazla deyimin oluşturduğu yapıya, bileşik deyim *(compound statement) denir. Aşağıda bir bileşik deyim görülüyor.*

```
{
x = 10;
y++;
func();
}
```

Kontrol deyimi:

Kontrol deyimleri, programın akış yönünü değiştirebilen deyimlerdir. Kontrol deyimleri ile programın akışı farklı noktalara yönlendirilebilir. Bunlar, C dilinin önceden belirlenmiş bazı sözdizimi kurallarına uyar, kendi sözdizimleri içinde en az bir anahtar sözcük içerir. C dilinde aşağıdaki kontrol deyimleri vardır:

if deyimi
while döngü deyimi
do while döngü deyimi
for döngü deyimi
break deyimi
continue deyimi
switch deyimi
goto deyimi
return deyimi

if DEYİMİ

C dilinde program akışını denetlemeye yönelik en önemli deyim *if deyimidir.*
En yalın biçimiyle *if deyiminin genel sözdizimi aşağıdaki gibidir:*

```
if (ifade)
deyim;
```

if ayracı içindeki ifadeye koşul ifadesi (conditional expression) denir.
if ayracını izleyen deyime, if deyiminin doğru kısmı (true path) denir.
if deyiminin doğru kısmını oluşturan deyim, bir yalın deyim (simple statement) olabileceği gibi, bir boş deyim (*null statement), bir bileşik deyim (compound statement) ya da* başka bir kontrol deyimi de *(control statement) olabilir.*
Yalın *if deyiminin yürütülmesi aşağıdaki gibi olur:*
Önce koşul ifadesinin sayısal değerini hesaplar. Hesaplanan sayısal değer, mantıksal *DOĞRU ya da YANLIŞ olarak yorumlanır. Koşul ifadesinin hesaplanan değeri 0 ise yanlış, 0'dan farklı bir değer ise doğru olarak yorumlanır.*

```
int main()
{
int x;
printf("bir sayi girin : ");
scanf("%d", &x);
if (x > 10)
printf("if deyiminin doğru kısmı!\n");
return 0;
}
```

Yanlış Kısmı Olan if Deyimi
if kontrol deyimi, else anahtar sözcüğünü de içerebilir. Böyle if deyimine, yanlış kısmı olan *if deyimi denir. Yanlış kısmı olan if deyiminin genel biçimi aşağıdaki gibidir:*

```
if (ifade)
deyim1;
else
deyim2;
```

Bu kez *if deyiminin doğru kısmını izleyen deyimden sonra else anahtar sözcüğünün, daha* sonra ise bir başka deyimin yer aldığını görüyorsunuz. Genel biçimdeki *deyim2'ye if* deyiminin yanlış kısmı *(false path) denir.*
if deyiminin koşul ifadesi, mantıksal olarak DOĞRU ya da YANLIŞ olarak yorumlanır. Bu kez koşul ifadesinin *DOĞRU olması durumunda deyim1, YANLIŞ olarak yorumlanması* durumunda *deyim2 yapılır.* Yanlış kısmı olan *if deyimi, bir koşul ifadesinin doğru ya da* yanlış olmasına göre iki ayrı deyimden birinin yapılmasına yöneliktir. Yani ifade doğru ise bir iş, yanlış ise başka bir iş yapılır.

```c
#include <stdio.h>
int main()
{
char ch;
printf("bir karakter girin : ");
ch = getchar();
if (ch >= 'a' && ch <= 'z')
printf("%c kucuk harf!\n", ch);
else
printf("%c kucuk harf degil!\n", ch);
return 0;
}
```

Yukarıdaki *main işlevinde standart getchar işlevi kullanılarak klavyeden bir karakter* alınıyor. Alınan karakterin sıra numarası, *ch isimli değişkene atanıyor. Koşul ifadesinin* doğru ya da yanlış olması durumuna göre, klavyeden alınan karakterin küçük harf olup olmadığı bilgisi ekrana yazdırılıyor. Koşul ifadesine bakalım:

```c
ch >= 'a' && ch <= 'z'
```

Bu ifadenin doğru olması için "mantıksal ve *(&&)" işlecinin her iki teriminin de doğru* olması gerekir. Bu da ancak, *ch karakterinin küçük harf karakteri olması ile mümkündür.*

if deyiminin doğru ve/veya yanlış kısmı bir bileşik deyim olabilir. Bu durumda, koşul ifadesinin doğru ya da yanlış olmasına göre, birden fazla yalın deyimin yürütülmesi sağlanabilir. Aşağıdaki örneği inceleyin:

```
/***/
if (x > 0) {
y = x * 2 + 3;
z = func(y);
result = z + x;
}
else {
y = x * 5 - 2;
z = func(y - 2);
result = z + x - y;
}
/***/
```

Yukarıdaki *if deyiminde, x > 0 ifadesinin doğru olup olmasına göre, result değişkeninin* değeri farklı işlemlerle hesaplanıyor. *if deyiminin hem doğru hem de yanlış kısımlarını* bileşik deyimler oluşturuyor.

Bir *if deyiminin yanlış kısmı olmak zorunda değildir. Ancak bir if deyimi yalnızca else* kısmına sahip olamaz. Bu durumda *if deyiminin doğru kısmına boş deyim ya da boş* bileşik deyim yerleştirilmelidir:

```
if (ifade)
;
else
deyim1;
```

ya da

```
if (ifade)
{ }
else
deyim1;
```

Yalnızca yanlış kısmı olan, doğru kısmı bir boş deyim olan bir *if deyimi, okunabilirlik* açısından iyi bir seçenek değildir. Böyle durumlarda daha iyi bir teknik, koşul ifadesinin mantıksal tersini alıp, *if deyiminin yanlış kısmını ortadan kaldırmaktır:*

```
if (!ifade)
deyim1;
```

Aşağıdaki kod parçasını inceleyin:

```
/***/
if (x > 5)
;
else {
func1(x);
func2(x);
}
/***/
```

Yukarıdaki *if deyiminde, x değişkeninin değeri 5'ten büyükse bir şey yapılmıyor, aksi* halde *func1 ve func2 işlevleri x değişkeninin değeri ile çağrılıyor. Koşul ifadesi ters* çevrilerek *if deyimi yeniden yazılırsa:*

```
/***/
if (x <= 5) {
func1(x);
func2(x);
}
/***/
```

if ayracının içinde, ifade tanımına uygun herhangi bir ifade bulunabilir:

```
if (10)
deyim1;
if (-1)
deyim2;
```

Yukarıdaki koşul ifadelerinin değeri, her zaman doğru olarak yorumlanır. Çünkü ifadeler, sıfırdan farklı değere sahiptir.

Aşağıdaki koşul ifadesi ise her zaman yanlış olarak yorumlanacağından *if deyiminin doğru* kısmı hiçbir zaman yürütülmez:

```
if (0)
deyim1;
```

Aşağıdaki *if deyiminde ise, x değişkeninin değerinin 0 olup olmamasına göre, deyim1 ve deyim2 yürütülür:*

```
if (x) {
deyim1;
deyim2;
/***/
}
```

Yukarıdaki yapıyla aşağıdaki yapı eşdeğerdir:

```
if (x != 0) {
deyim1;
deyim2;
}
```

```
if (!x) {
deyim1;
deyim2;
}
```

Bu *if deyiminde ise ancak x değişkeninin değerinin 0 olması durumunda deyim1 ve deyim2 yürütülür.*

Yine yukarıdaki yapıyla aşağıdaki yapı eşdeğerdir:

```
if (x == 0) {
deyim1;
deyim2;
}
```

İç İçe if Deyimleri

if deyiminin doğru ya da yanlış kısmını, başka bir if deyimi de oluşturabilir:

```
if (ifade1)
if (ifade2) {
deyim1;
deyim2;
deyim3;
}
deyim4;
```

Bu örnekte ikinci *if deyimi birinci if deyiminin doğru kısmını oluşturur. Birinci ve ikinci if* deyimlerinin yanlış kısımları yoktur.

İç içe *if deyimlerinde, son if anahtar sözcüğünden sonra gelen else anahtar sözcüğü, en* içteki *if deyimine ait olur:*

```
if (ifade1)
if (ifade2)
deyim1;
else
deyim2;
```

Yukarıdaki örnekte, yazım biçimi nedeniyle *else kısmının birinci if deyimine ait olması* gerektiği gibi bir görüntü verilmiş olsa da, *else kısmı ikinci if deyimine aittir. else anahtar* sözcüğü, bu gibi durumlarda, kendisine yakın olan *if deyimine ait olur (dangling else). else anahtar sözcüğünün birinci if deyimine ait olması isteniyorsa, birinci if deyiminin* doğru kısmı bloklanmalıdır:

```
if (ifade1) {
if (ifade2)
deyim1;
}
else
deyim2;
```

Yukarıdaki örnekte *else kısmı birinci if deyimine aittir.*

```
if (ifade1) {
if (ifade2)
deyim1;
else {
deyim2;
deyim3;
}
deyim4;
}
else
deyim5;
```

Yukarıdaki örnekte birinci *if deyiminin doğru kısmı, birden fazla deyimden oluştuğu için -* bu deyimlerden birisi de yine başka bir *if deyimidir- bloklama yapılıyor. deyim5, birinci if* deyiminin yanlış kısmını oluşturur.

TO BE CONTINUED...