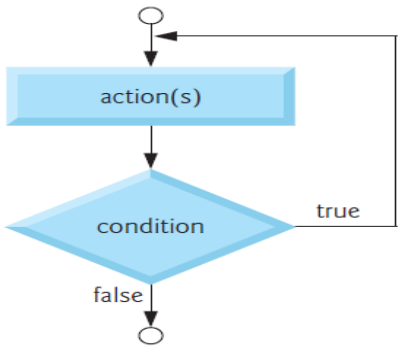
Program Kontrolü-Yineleme (İterasyon)

do…while Yineleme İfadesi

do…while yineleme ifadesi, while ifadesine benzer. while ifadesi, döngü gövdesini yürütmeden önce döngü devam koşulunu test eder. **do…while** deyimi, döngü gövdesini çalıştırdıktan sonra döngü devam koşulunu test eder, böylece döngü gövdesi her zaman en az bir kez yürütülür. Bir **do…while** sona erdiğinde, yürütme while yan tümcesinden sonraki ifadeyle devam eder.
Şekil 4.6, 1'den 5'e kadar olan sayıları görüntülemek için bir **do…while** ifadesi kullanmaktadır. Döngü devam testinde (satır 10) kontrol değişkeni sayacını ön artırma seçilmiştir.

```
1 // fig04_06.c
2 // Using the do...while iteration statement.
3 #include <stdio.h>
4
5 int main(void) {
6 int counter = 1; // initialize counter
7
8       do {
9               printf("%d ", counter);
10      } while (++counter <= 5);
11 }
```

Aşağıdaki **do…while** deyimi akış şeması, döngü devam koşulunun, döngü eylemi ilk defa gerçekleştirilene kadar yürütülmeyecektir. **while** döngüsünde ise önce koşul testi sonra yapılacak eylem gerçekleştirilir.



break ve continue deyimleri

break ve **continue** deyimleri, kontrol akışını değiştirmek için kullanılır. Bölüm 4.6, bir **switch** deyiminde karşılaşılan bir **break** ifadesinin, **switch**'in yürütülmesini sonlandırdığını göstermişti. Bu bölümde, bir yineleme ifadesinde **break**'in nasıl kullanılacağı anlatılmaktadır.

break Deyimi
break deyimi, **while**, **for**, **do…while** veya **switch** deyimi içinde yürütüldüğünde, o deyimden anında çıkışa neden olur. Program yürütme bir sonraki **while**, **for**, **do…while** veya **switch** ifadesiyle devam eder. **break**'in yaygın kullanımları, bir döngüden erken çıkmak veya bir **switch** ifadesinin geri kalanını atlamaktır (Geçen ders yaptığımız Şekil 4.5'teki gibi). Şekil 4.7, bir **for** iterasyon ifadesindeki **break** ifadesini (satır 12) göstermektedir.

```c
// fig04_07.c
// Using the break statement in a for statement.
#include <stdio.h>

int main(void) {
   int x = 1; // declared here so it can be used after loop

   // loop 10 times
   for (; x <= 10; ++x) {
      // if x is 5, terminate loop
      if (x == 5) {
         break; // break loop only if x is 5
      }

      printf("%d ", x);
   }

   printf("\nBroke out of loop at x == %d\n", x);
}
```

“if” ifadesi, x'in 5 olduğunu algıladığında, break yürütülür. Bu, for deyimini sonlandırır ve program for'dan sonra printf ile devam eder. Döngü tam olarak yalnızca dört kez yürütülür. Kontrol değişkenini bir for döngüsünün başlatma ifadesinde bildirdiğinizde, değişkenin döngü sona erdikten sonra artık mevcut olmadığını hatırlayın. Bu örnekte x'i döngüden önce tanımladık ve başlattık, böylece döngü sona erdikten sonra son değerini kullanabiliriz. Bu nedenle, for başlığının başlatma bölümü (ilk noktalı virgülden önce) boştur.

continue İfadesi

Bir continue ifadesi, **for** veya **do…while** deyimi içinde yürütüldüğünde, söz konusu kontrol deyiminin gövdesinde kalan ifadeleri atlar ve döngünün bir sonraki yinelemesini gerçekleştirir. **while** ve **do…while** deyimlerinde, **continue** deyimi çalıştırıldıktan sonra döngü başa döner ve **continue** deyiminden sonraki döngü sonuna kadar olan komutları atlar yani çalıştırmaz ve bir sonraki döngüye geçer.
Şekil 4.8 de, x değişkeni 5 olduğunda **printf** deyimini atlamak ve döngünün bir sonraki yinelemesini başlatmak için **for** ifadesinde **continue** (10. satır) kullanılmıştır.

NOT: Bu programda 7. Satırda (*int x = 1;*) tanımı for döngüsünün dışına çıkartılmalıdır. Kitaptaki program mevcut hali ile çalışmayabilir.

```c
// fig04_08.c
// Using the continue statement in a for statement.
#include <stdio.h>

int main(void) {
   // loop 10 times
   for (int x = 1; x <= 10; ++x) {
      // if x is 5, continue with next iteration of loop
      if (x == 5) {
         continue; // skip remaining code in loop body
      }

      printf("%d ", x);
   }

   puts("\nUsed continue to skip printing the value 5");
}
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

break ve continue Notları

Bazı programcılar, break ve continue deyimlerinin yapılandırılmış programlama normlarını ihlal ettiğini iddia ediyor ve bu yüzden onları kullanmıyorlar. Bu ifadelerin etkileri, yakında tartışacağımız yapılandırılmış programlama teknikleriyle elde edilebilir, ancak **break** ve **continue** ifadeleri daha hızlı çalışır. Kaliteli yazılım mühendisliğine ulaşmak ile en iyi performans gösteren yazılıma ulaşmak arasında bir tezat vardır; biri diğerinin pahasına elde edilir. En yoğun performans gerektiren durumlar dışındaki tüm durumlar için ilk olarak, kodunuzu basit ve doğru yapın; ancak yalnızca gerekliyse daha sonra hızlı ve küçük hale getirin.

Mantıksal operatörler

Şimdiye kadar **counter <= 10**, **total > 1000** ve **grade != -1** gibi basit koşullar kullandık. Bu koşulları ilişkisel operatörler (>, <, >= ve <=) ve eşitlik operatörleri (== ve !=) cinsinden ifade ettik. Her karar tam olarak bir koşulu test etti. Karar verme sürecinde birden çok koşulu test etmek için, bu testleri ayrı ifadelerde veya iç içe if veya if…else ifadelerinde gerçekleştirmemiz gerekiyordu.
C, basit koşulları birleştirerek daha karmaşık koşullar oluşturmak için kullanılabilecek mantıksal operatörler sağlar. Mantıksal operatörler && (mantıksal VE), || (mantıksal VEYA) ve ! (mantıksal DEĞİL). Bu operatörlerin her birinin örneklerini ele alalım.

Belirli bir yürütme yolunu seçmeden önce iki koşulun da doğru olduğundan emin olmak istediğimizi varsayalım. Bu durumda, && mantıksal operatörünü aşağıdaki gibi kullanabiliriz:

```
if (gender == 1 && age >= 65) {
++seniorFemales;
}
```

Bu if ifadesi iki basit koşul içerir. Örneğin, gender == 1 koşulu, bir kişinin kadın olup olmadığını belirleyebilir. age >= 65 koşulu, bir kişinin yaşlı olup olmadığını belirler. Önce iki basit koşul değerlendirilir, çünkü == ve >= her biri &&'den daha yüksek önceliğe sahiptir. if ifadesi daha sonra gender == 1 && age >= 65 birleşik koşulunu dikkate alır; bu ancak ve ancak basit koşulların her ikisi de doğruysa doğrudur. Son olarak, bu birleştirilmiş koşul doğruysa, o zaman önceki **if** deyimi seniorFemales'i 1 artırır. Basit koşullardan biri veya her ikisi de yanlışsa, program **if**'nin gövdesini atlar ve sırayla bir sonraki ifadeye geçer.

Aşağıdaki tabloda && işleci özetlenmektedir:

ifade1	ifade2	ifade1 && ifade2
0	0	0
0	değer	0
değer	0	0
değer	değer	1

Şimdi || (mantıksal VEYA) operatörü. Belirli bir yürütme yolunu seçmeden önce, bir programın bir noktasında iki koşuldan birinin veya her ikisinin de doğru olduğundan emin olmak istediğimizi varsayalım. Bu durumda, || operatör, aşağıdaki program bölümünde olduğu gibi kullanılabilir:

```
if (semesterAverage >= 90 || finalExam >= 90) {
puts("Student grade is A");
}:
```

Bu ifade iki basit koşul içerir. sömestrAverage >= 90 koşulu, öğrencinin sömestr boyunca gösterdiği sağlam performanstan dolayı “A” almayı hak edip etmediğini belirler. finalExam >= 90 koşulu, öğrencinin final sınavındaki üstün performansı nedeniyle “A” notunu hak edip etmediğini belirler. **if** ifadesi daha sonra birleştirilmiş koşulu dikkate alır ve basit koşullardan herhangi biri veya her ikisi de doğruysa öğrenciye bir "A" verir. Her iki basit koşul da yanlış (sıfır) olmadığı sürece “Öğrenci notu A” mesajı yazdırılır. Aşağıda, mantıksal VEYA operatörü (||) için bir doğruluk tablosu verilmiştir:

Aşağıdaki tabloda || işleci özetlenmektedir:

ifade1	ifade2	ifade1 || ifade2
0	0	0
0	değer	1
değer	0	1
değer	değer	1

Mantıksal Olumsuzlama (!) Operatörü

C, tekli ! (mantıksal olumsuzlama) operatörü bir koşulun anlamını "tersine çevirmenizi" sağlar. Mantıksal olumsuzlama operatörü, işlenen olarak tek bir koşula sahiptir. Aşağıdaki program parçasında olduğu gibi, işlenen koşulu yanlışsa bir yürütme yolu seçmek istediğinizde bunu kullanırsınız:

```
if (!(grade == sentinelValue)) {
printf("The next grade is %f\n", grade);
}
```

Mantıksal olumsuzlama operatörü eşitlik operatöründen daha yüksek önceliğe sahip olduğundan, grade == sentinelValue koşulu etrafındaki parantezler gereklidir.

Çoğu durumda, koşulu farklı şekilde ifade ederek mantıksal olumsuzlama kullanmaktan kaçınabilirsiniz. Örneğin, önceki ifade şu şekilde de yazılabilir:

```
if (grade != sentinelValue) {
printf("The next grade is %f\n", grade);
}
```

Eşitlik (==) ve Atama (=) Operatörlerini Karıştırmak

Ne kadar deneyimli olursa olsun, C programcılarının o kadar sık yapma eğiliminde oldukları bir tür hata vardır ki, ayrı bir bölüme değer. Bu hata yanlışlıkla == (eşitlik) ve = (atama) operatörlerini değiştirmektir. Bu değiş tokuşları bu kadar zararlı yapan şey, normalde derleme hatalarına neden olmamalarıdır. Bunun yerine, bu hataları içeren ifadeler normalde doğru bir şekilde derlenir ve programların, çalışma zamanı mantık hataları yoluyla muhtemelen yanlış sonuçlar üretirken tamamlanmasına izin verir.

Örneğin, şunu yazmaya niyetlendiğimizi varsayalım;

```
if (payCode == 4) {
printf("%s", "You get a bonus!");
}
```

Fakat hata ile şunu yazmış olalım;

```
if (payCode = 4) {
printf("%s", "You get a bonus!");
}
```

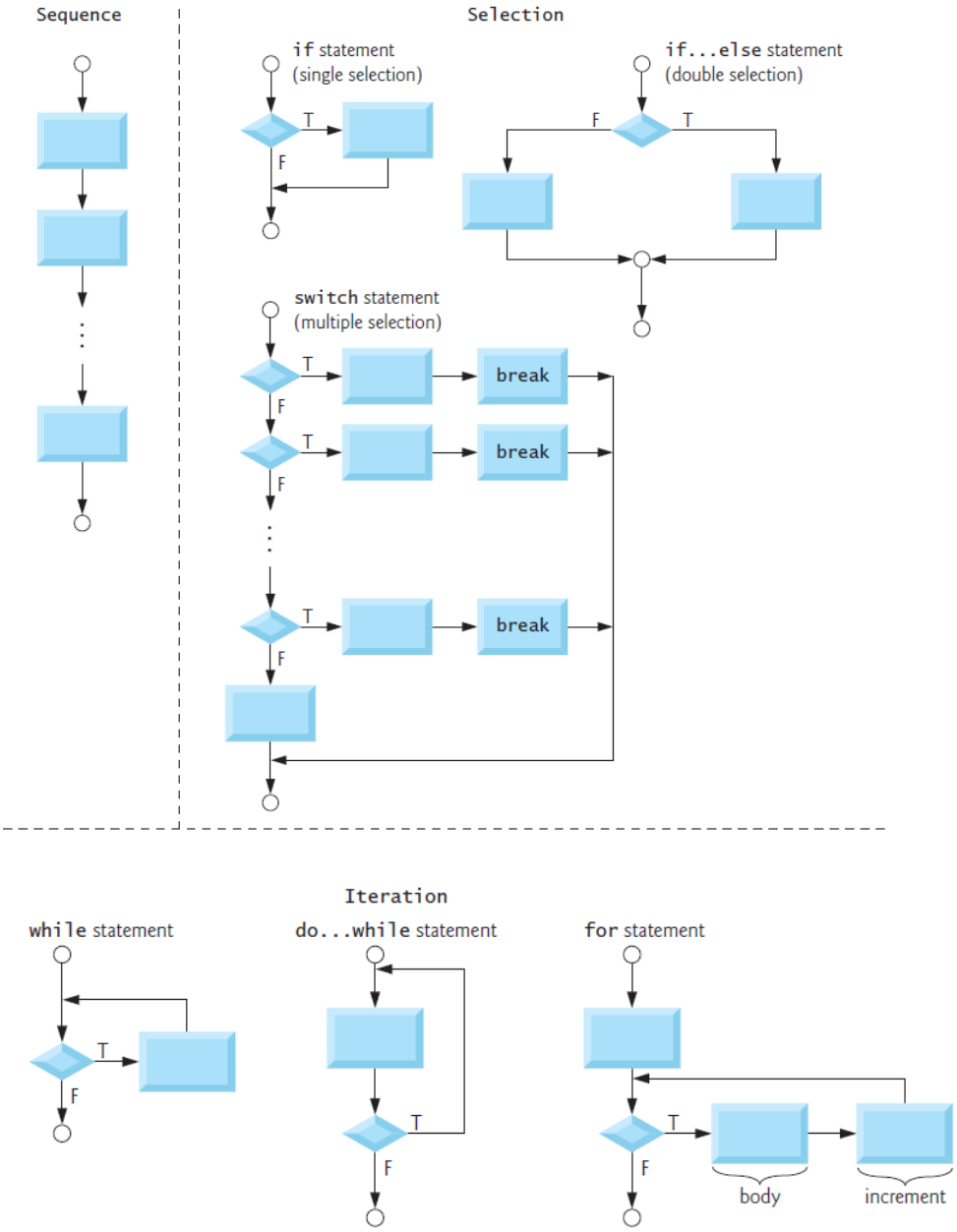
İlk if ifadesi, pay Code 4'e eşit olan kişiye uygun şekilde bir ikramiye verir. Hatalı olan ikinci if ifadesi, if koşulundaki atama ifadesini değerlendirir. Atamadan sonraki koşuldaki değer 4'tür. Sıfır olmayan herhangi bir değer doğru olduğundan, bu if ifadesindeki koşul her zaman doğrudur. payCode yanlışlıkla 4'e ayarlanmakla kalmaz, aynı zamanda gerçek payCode ne olursa olsun kişi her zaman bir bonus alır! Çünkü, C standardı yalnızca 0 veya 1 değerlerini tutabilen, **_Bool** anahtar kelimesiyle temsil edilen bir binary türü içerir. Bir koşuldaki 0 değerinin yanlış, sıfır olmayan herhangi bir değerin doğru olduğunu kabul eder. Bir **_Bool**'a sıfır dışında herhangi bir değer atamak, onu 1 olarak ayarlar. Standart C ayrıca, sırasıyla 1 ve 0'ın adlandırılmış temsillerini tanımlayan **<stdbool.h>** kütüphanesini de içerir.

Bu durumda atama için yanlışlıkla == operatörünün kullanılması ve eşitlik için yanlışlıkla = operatörünün kullanılması mantık hatasıdır.

Yapılandırılmış Programlar Oluşturma

Nasıl mimarlar mesleklerinin ortak aklını kullanarak binalar tasarlıyorsa, programcılar da mesleklerinin ortak aklını kullanarak programlar tasarlamalıdır. Belki de en önemlisi, yapılandırılmış programların matematiksel anlamda anlaşılmasının, test edilmesinin, hatalarının ayıklanmasının, değiştirilmesinin ve hatta doğruluğunun kanıtlanmasının (yapılandırılmamış programlara göre) daha kolay olduğunu görüyoruz.

3. ve 4. bölümlerde C'nin kontrol ifadeleri tartışıldı. Şimdi, bu yetenekleri özetleyelim ve yapılandırılmış programlar oluşturmak için basit bir dizi kurala dönüştürelim. Aşağıdaki diyagram, kontrol ifadelerinin akış şemalarını özetlemektedir:

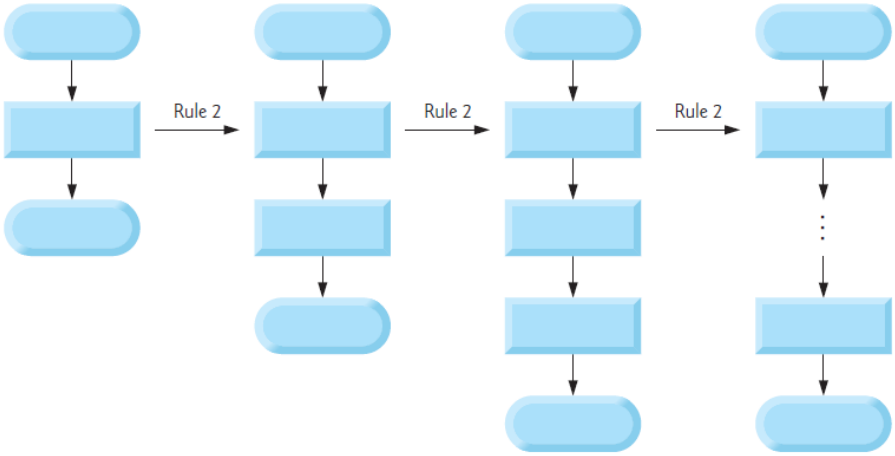


Diyagramda, küçük daireler her ifadenin tek giriş noktasını ve tek çıkış noktasını gösterir. Bireysel akış şeması sembollerini keyfi olarak bağlamak, yapılandırılmamış programlara yol açabilir. Bu nedenle, iyi bir programlama için, sınırlı sayıda kontrol ifadesi oluşturarak, akış şeması sembollerini birleştirerek kontrol ifadelerini iki basit şekilde birleştirerek düzgün yapılandırılmış programlar oluşturmak gerekecektir. Basit olması için, yalnızca tek girişli/tek çıkışlı kontrol ifadeleri kullanılır ve bunları yalnızca kontrol ifadelerini sırayla istifleyerek veya iç içe geçirerek birleştirebilirsiniz.

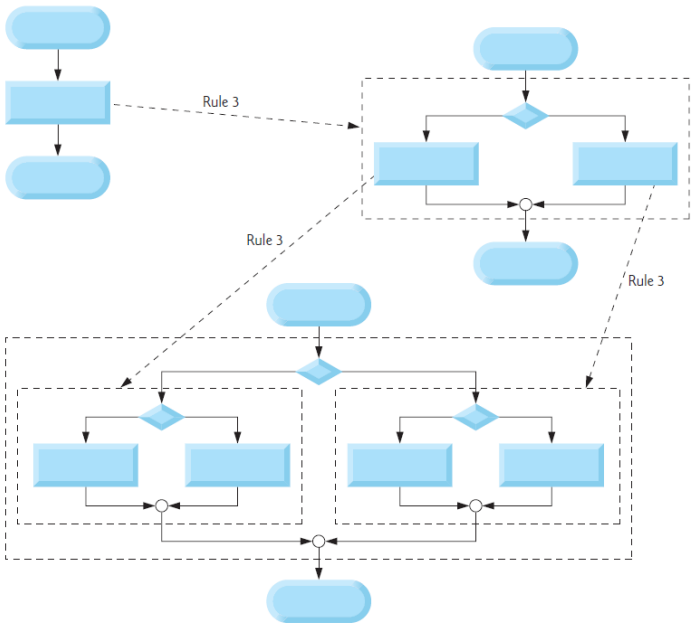
Yapılandırılmış programlar oluşturma kuralları

1. Aşağıdaki şekilde gösterilen “en basit akış şeması” ile başlayın.
2. “Yığınlama” kuralı—Herhangi bir eylem (dikdörtgen ile gösterilen), sırayla iki eylem (dikdörtgen) ile değiştirilebilir.
3. "İç içe yerleştirme" kuralı—Herhangi bir eylem (dikdörtgen), herhangi bir kontrol ifadesi ile değiştirilebilir (sırasal ifade, if, if…else, switch, while, do…while veya for, gibi).
4. Kural 2 ve 3, istediğiniz sıklıkta ve herhangi bir sırayla uygulanabilir.

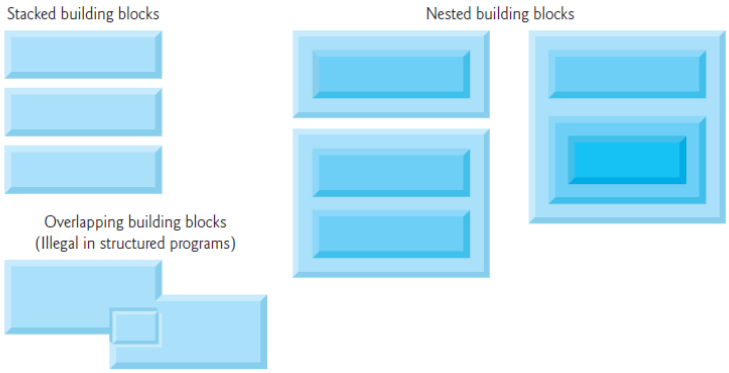
Yapılandırılmış programlar oluşturmak için kuralların uygulanması, her zaman düzgün, yapı taşı görünümüne sahip yapılandırılmış bir akış şeması oluşturulmasını sağlar. Kural 2'yi en basit akış şemasına tekrar tekrar uygulamak, aşağıdaki şemada olduğu gibi sırayla birçok dikdörtgen içeren yapılandırılmış bir akış şeması oluşturur. Kural 2, bir kontrol ifadeleri yığını oluşturur, bu nedenle Kural 2'ye yığın kuralı diyoruz.



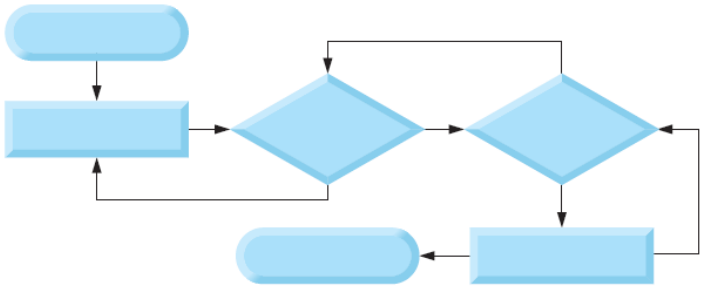
Kural 3, iç içe geçme kuralı olarak adlandırılır. En basit akış şemasına Kural 3'ü tekrar tekrar uygulamak, düzgün bir şekilde iç içe geçmiş kontrol ifadelerine sahip bir akış şeması oluşturur. Örneğin, aşağıdaki şemada, en basit akış şemasındaki dikdörtgen, bir çift seçim (if…else) ifadesi ile değiştirilmiştir. Ardından, çift seçim ifadesindeki her iki dikdörtgene de Kural 3 tekrar uygulanır ve bu dikdörtgenlerin her biri çift seçim ifadeleriyle değiştirilir. Çift seçim deyimlerinin her birinin etrafındaki kesikli kutu, orijinal akış şemasında değiştirdiğimiz dikdörtgeni temsil eder.



Kural 4, daha büyük, daha ilgili ve daha derin iç içe geçmiş yapılar oluşturur. Yapılandırılmış programlar oluşturmak için kuralların uygulanmasından ortaya çıkan akış şemaları, tüm olası yapılandırılmış akış şemaları setini ve dolayısıyla tüm olası yapılandırılmış programların setini oluşturur.
Bunun nedeni, "**goto"** ifadesinin ortadan kaldırılmasıdır. Yapılandırılmış yaklaşımın güzelliği, yalnızca az sayıda basit tek girişli/tek çıkışlı parçalar kullanmamız ve bunları yalnızca iki basit yolla bir araya getirmemizdir. Aşağıdaki diyagram, Kural 2'nin uygulanmasıyla ortaya çıkan yığılmış yapı taşlarının türlerini ve Kural 3'ün uygulanmasının ortaya çıktığı iç içe yapı taşlarının türlerini göstermektedir.



Yapılandırılmış program oluşturma kurallarına uyulursa, aşağıdaki diyagramdaki gibi yapılandırılmamış bir akış şeması oluşturulamaz:



Belirli bir akış şemasının yapılandırılmış olup olmadığından emin değilseniz, akış şemasını en basit akış şemasına indirgemeye çalışmak için yapılandırılmış programlar oluşturma kurallarını tersten uygulayın. Başarılı olursanız, orijinal akış şeması yapılandırılır; Aksi takdirde, yapılandırılamaz.

Üç Kontrol Şekli
Yapısal programlama basitliği teşvik eder. Böhm ve Jacopini, yalnızca üç kontrol biçiminin gerekli olduğunu gösterdi:
• Sıralı işlem (squence).
• Seçim (selection).
• Yineleme (iteration).

Seçim üç yoldan biriyle gerçekleştirilir:
• if ifadesi (tek seçim).
• if…else ifadesi (çift seçim).
• if…elseif….else ifadesi (birden çok seçim) veya;
• switch deyimi (birden çok seçim).

Basit if ifadesinin herhangi bir seçim biçimi sağlamak için yeterli olduğunu kanıtlamak kolaydır. if…else deyimi ve switch deyimi ile yapılabilecek her şey, bir veya daha fazla if deyimi ile gerçekleştirilebilir.

Yineleme üç yoldan biriyle uygulanır:
• while ifadesi.
• do…while deyimi.
• for ifadesi.

Ayrıca while ifadesinin herhangi bir yineleme biçimi sağlamak için yeterli olduğunu kanıtlamak da kolaydır. do…while deyimi ve for deyimi ile yapılabilecek her şey while deyimi ile yapılabilir.

Bu sonuçların birleştirilmesi, bir C programında şimdiye kadar ihtiyaç duyulan herhangi bir kontrol biçiminin yalnızca üç denetim biçimiyle ifade edilebileceğini göstermektedir:
• sequence (sıralama).
• if ifadesi (seçim).
• while ifadesi (yineleme).

Ve bu kontrol ifadeleri yalnızca iki şekilde birleştirilebilir: istifleme ve yuvalama. Aslında, yapılandırılmış programlama basitliği teşvik eder.

3. ve 4. Bölümlerde, yalnızca eylemleri ve kararları içeren kontrol ifadelerinden programların nasıl oluşturulacağını tartıştık. 5. Bölüm'de, fonksiyon adı verilen başka bir program yapılandırma birimini tanıtacağız. Sırasıyla kontrol ifadelerinden oluşabilen fonksiyonları birleştirerek büyük programlar oluşturmayı öğreneceğiz. Fonksiyonları kullanmanın yazılımın yeniden kullanılabilirliğini nasıl desteklediğini de tartışacağız.