C Programlama Dili İkinci Tiraj

Dennis M. Ritchie, Brian M. Kernighan
Çevirmen Editörler: Serhan Ekmekçi, Eylül Elif Beşdallı

İçindekiler

Önsöz

Brian W. Kernighan
Dennis M. Ritchie

İlk Tiraja Önsöz

Brian W. Kernighan
Dennis M. Ritchie

Giriş

Bölüm 1 | Öğretici Kısıma Giriş

C'ye hızlı bir girişle başlayalım. Ana odağımız dilin gerekli elemanlarını detaylara, ayrıntılara, kurallara ve istisnalara girmeden gerçek programlar aracılığıyla göstermek. Bu aşamada tamamlayıcı ve kusursuz olmaya çalışmıyoruz. Sizi, olabilecek en kısa sürede işe yarar programlar yazabilecek hale getirmeye çalışıyoruz. Ve bunu yapabilmek için temellere (değişkenler (**variables**), sabitler (**constants**), aritmetikler, kontrol akışı (**control flow**), fonksiyonlar (**functions**), girdi (**input**) ve çıktıların (**output**) esasları) odaklanmamız gerekmekte. Bilerek bu bölümde C'nin büyük programlar yazmak için önemli olan özelliklerine değinmiyoruz. Bu işaretleyiciler (**pointers**), yapılar (**structers**), C'nin zengin operatör setinin çoğunu, bazı kontrol akışı (**control-flow**) ifadelerini (**statement**) ve temel C standart kütüphanesini (**library**) içeriyor.

Bu amacın kendine has sakıncaları bulunmakta. En öne çıkanı ise belirli bir dilin baştan sona hikayesinin burada bulunmaması. Aynı zamanda, öğretici kısım özetleyici olduğu gibi yanıltıcı da olabilir. Ve örnekler öğretim amacıyla C'nin tüm gücünden faydalanmadığından, olabilecek en öz ve zarif şekilde değiller. Bu etkileri hafifletebileceğimiz kadar hafiflettik fakat dikkatli olmanız gerekmekte. Bir diğer sakınca ise bundan sonraki bölümlerin bu bölümün bir kısmının tekrarını içeriyor olması. Umuyoruz ki bu tekrarlama size rahatsızlıktan çok yardım sağlar.

Herhangi bir durumda, deneyimli programcılar kendi programcılık ihtiyaçları doğrultusunda verilen materyalden faydalanacaklardır. Yeni başlayanlar ise bunu kendi küçük benzer programlarını yazarak sağlayabilirler. İki grup da bunu çatı (**framework**) olarak Bölüm 2'de başlayan daha detaylı açıklamaların üstüne kullanabilirler.

1.1 Başlamak

Yeni bir programlama dili öğrenmenin tek yolu o dille programlar yazmaktır. Yazılacak ilk program tüm diller için aynıdır.

```
Kelimeleri Bastır
   hello, world
```

Bu büyük bir engel; aşılması için programı metin olarak bir yerde oluşturmak, ardından başarılı bir şekilde derlemek, yüklemek, çalıştırmak, ve çıktının (**output**) nereye gittiğini bilmek gerekiyor. Bu mekaniksel detaylarda ustalaştıktan sonra, geri kalan her şey görece daha kolay.

C'de "hello, world" bastırma programı

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Bu programı çalıştırmak kullandığınız sisteme göre değişiklik gösterir. Spesifik bir örnek vermek gerekirse, UNIX işletim sisteminde programı hello.c gibi ".c" uzantılı bir dosyanın içine kaydedip, ardından bu komut ile derlemeniz gerekir.

```
cc hello.c
```

Eğer bir karakter yazmayı unutmak veya bir şeyi yanlış yazmak gibi bir hata yapmadıysanız, derleme sorunsuz ve sessiz bir şekilde gerçekleşmeli, ve a.out isimli bir çalıştırılabilir (**executable**) dosya oluşmalı. Eğer bu komut ile a.out dosyasını çalıştırırsanız

```
a.out
```

bunu bastıracaktır

```
hello, world
```

Diğer sistemlerde kurallar farklı olacaktır, sisteminizle ilgili yerel bir uzmana danışın. Şimdi programın kendisi hakkında biraz açıklama. Bir C programı, boyutu ne olursa olsun, fonksiyonlardan(**functions**) ve değişkenlerden(**variables**) oluşur. Bir fonksiyon (**function**), yapılacak bilgisayarsal operasyonları belirten ifadeler (**statements**) ve çalışma sırasında kullanılacak değerleri (**values**) taşıyan değişkenler (**variables**) içerir. C fonksiyonları (**functions**) daha çok altprogramlara (**subroutines**) ya da Fortran dilini bilenlerin aşina olduğu Fortran fonksiyonlarına (**function**), Pascal prosedürlerine (**procedures**)

ve Pascal fonksiyonlarına (**functions**) oldukça benzer. Bizim örneğimiz **main** isimli fonksiyon (**function**). Normalde fonksiyonlara (**functions**) istediğiniz ismi vermekte özgürsünüz fakat "**main**" için değil. Programınız **main** fonksiyonunu (**function**) baştan aşağıya doğru çalıştırarak işe başlar. Bu, her programın herhangi bir yerinde bir **main** fonksiyonu (**function**) olmasını gerektirir. **main** genellikle başka fonksiyonları (**functions**) gerçekleştireceği işe yardım için çağırır, bazen sizin yazdıklarınız, bazen de kütüphanelerde (**libraries**) sizin için sağlanan fonksiyonları. Programınızın ilk satırı,

#**include** <**stdio.h**>

derleyiciye (**compiler**) standart girdi (**input**), çıktı (**output**) hakkında bilgilerin programa dahil edilmesini söyler. Bu satır hemen hemen her C kaynak (**source**) dosyasında bulunur. Standart kütüphane (**library**) Bölüm 7'de ve Ek B'de açıklanmıştır. Fonksiyonlar (**functions**) arasında veri aktarımı için gerekli metot, çağırılan fonksiyonlara (**functions**) argümanlar (**arguments**) adı verilen değer veya değerler vermektir. Fonksiyon adından sonraki parantezlerin arasında argüman (**argument**) veya argümanlar (**arguments**) bulunur.

```
#include <stdio.h>          standart kütüphane (library) hakkında
                            bilgileri içer

main()                      main adında hiç argümanı (argument)
                            olmayan bir fonksiyon (function) tanımla

{
                            main'in ifadeleri (statements) süslü
                            parantezler arasındadır

   printf("hello, world\n"); main bir kütüphane (library)
                             fonksiyonu (function) olan printf
                             fonksiyonunu (function) parantezlerin
                             arasında verilen argümanı ekrana
                             bastırması için çağırıyor; \n yeni bir
}                            satırı temsil eder.
```

İlk C programı.

Bu örnekte **main** fonksiyonu (**function**) hiçbir argüman (**argument**) beklemeyen bir fonksiyon (**function**) olarak tanımlandı, bu parantezler arasında hiçbir veri olmamasıyla gösterilir (). Fonksiyonun ifadeleri (**statements**) süslü parantezler içerisinde tanımlanır **{}**. **main** fonksiyonu sadece bir ifade (**statement**) içeriyor,

```
printf("hello, world\n");
```

C derleyicisi (**compiler**) hata mesajı üretecektir. **printf** asla yeni bir satırı otomatik olarak sağlamayacaktır, yani bu sayede tek bir çıktıyı (**output**) aralıksız bir biçimde printf fonksiyonunu (**function**) birden fazla kere çağırarak bastırabiliriz ve bu ilk programımız kadar iyi çalışır.

```
#include <stdio.h>

main()
{
    printf("hello, ");
    printf("world");
    printf("\n");
}
```

Bu şekilde aynı çıktıyı (**output**) bastırabiliriz. \n'nin sadece bir karakteri temsil ettiğini farkedin. \n gibi bir kaçış dizesi (**escape sequence**) bize yazılması zor veya görünmez karakteri belirtmek için genel ve genişletilebilir bir mekanik sağlıyor. C'nin diğer sağladığı kaçış dizeleri: \t tab karakteri için, \b geri tuşu (**backspace**) için, \" çift tırnak için, ve \\ ters eğik çizginin (**backslash**) kendisi için. Bunların listesinin tamamını Bölüm 2.3'te bulabilirsiniz.

Egzersiz 1-1. "hello, world" programını sisteminizde çalıştırın. Programın belli başlı parçalarını çıkartın ve alacağınız uyarı mesajlarını ve çıktıları (**outputs**) görün.

Egzersiz 1-2. printf'in argümanı (**argument**) olan karakter öbeği (**string**) burada listelenmemiş olan \c karakterini içerdiğinde ne olacağını görün.

1.2 Değişkenler ve Aritmetik Açıklamalar

Sonraki program aşağıdaki Fahrenhayt dereceleri ve onların Selsiyus (Santigrat) eşitliklerini gösteren tabloyu bastırmak için $^\circ C = (5/9)(^\circ F\text{-}32)$ formülünü kullanıyor.

```
0   -17
20  -6
40  4
60  15
80  26
100 37
120 48
140 60
160 71
180 82
200 93
220 104
240 115
260 126
280 137
300 148
```

Programın kendisi hâlen sadece **main** adında tek bir fonksiyon (**function**) içeriyor. "hello, world" bastıran programdan biraz uzun ancak korkacak bir şey yok, komplike değil. Bu program bize, yorumlar (**comments**), tanımlamalar (**declarations**), değişkenler (**variables**), aritmetik açıklamalar, döngüler (**loops**) ve formatlanmış çıktılar (**output**) gibi birkaç yeni fikri gösteriyor.

```
#include <stdio.h>

/* Fahrenhayt-Selsiyus tablosunu bastır

    for fahr = 0, 20, ... 300 */

main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0; /* ısı tablosunun en düşük değeri */
    upper = 300; /* ısı tablosunun en yüksek değeri */
    step = 20; /* adım değeri */

    fahr = lower;

    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Bu iki satır

```
/* Fahrenhayt-Selsiyus tablosunu bastır

   for fahr = 0, 20, ... 300 */
```

bu durumda programın nasıl çalıştığını, programın kaynak (**source**) kodunu okuyanlara açıklayan *yorum* (**comment**) satırlarıdır. /* ve */ karakterleri arasında olan karakterler derleyici (**compiler**) tarafından görmezden gelinir, bu satırlar programı okurken anlamayı daha kolay hale getirmek için özgürce kullanılabilir. Yorum satırları boşluk, tab veya yeni satır karakterinin koyulabileceği bütün yerlere koyulabilirler.

C'de; bütün değişkenler (**variables**) kullanılmadan önce tanımlanmalı. Genellikle fonksiyonun (**function**) başında bütün çalıştırılabilir ifadelerden (**executable statements**) önce tanımlanır. Bir tanımlama (**decleration**) değişkenlerin (**variables**) özelliklerini anons eder; bu anons değişkenlerin (**variables**) tanımlanacak türünü (**type**) ve değişkenlere (**variable**) verilecek adları içerir, örneğin

```
int fahr, celsius;
int lower, upper, step;
```

tür(**type**) **int** kendisinin ardından listelenen değişkenlerin (**variable**) birer tamsayı (**integer**) olduğunu belirtir. Kayar noktalı yani ondalıklı sayılar (**float**) ile karşılaştırırsak, **int** ve **float**'un da sahip olduğu aralık, kullandığınız makineye dayalıdır; 16-bit **int'ler**, -32768 ve +32767 arasındadır ve 16-bit **int'ler** de 32-bit **int'ler** kadar yaygındır, çoğunlukla **float** sayı 32-bit niceliğindedir, bu en az 6 farklı basamağa ve 10^{-38} ve 10^{+38} arasında bir büyüklüğe tekabül eder.

C **int** ve **float**'un yanında birkaç başka temel veri türü (**type**) daha sağlıyor:

```
char   karakter - tek byte
short  kısa tamsayı (integer)
long   uzun tamsayı (integer)
double 64-bit (double precision) ondalıklı (float)
```

Bu objelerin büyüklüğü de makineye dayalı (**machine-dependent**). Bunların yanında kurs boyunca tanışacağımız dizeler (**arrays**), yapılar (**structers**) ve birlikler (**unions**) gibi basit türler (**types**); onları işaretlemek (**pointing**) için işaretleyiciler (**pointers**) ve onları döndürmek (**return**) için fonksiyonlar (**functions**) vardır.

Isı birimi dönüşümü programında hesaplama atama (**assignment**) ifadeleriyle (**assignment statements**) başlıyor.

```
lower = 0;
upper = 300;
step = 20;
fahr = lower;
```

Bu ifadeler (**statement**) ile değişkenler (**variables**) başlangıç değerlerine tanımlanıyor. Tekli (toplu değil, bireysel) tanımlanmak istenen değişkenler (**variables**) noktalı virgül ile bitiriliyor.

Tablonun her satırı aynı şekilde hesaplanıyor, bu yüzden her çıktı (**output**) satırı için bunu tekrarlayabiliriz; işte **while** döngüsünün (**loop**) amacı budur.

```
while (fahr <= upper) {
    ...
}
```

while döngüsü (**loop**) şöyle işliyor: Parantezler arasındaki koşul test edilir, eğer koşul doğru ise (**fahr** küçüktür veya eşittir **upper**'a), döngünün (**loop**) gövdesi (süslü parantezler arasındaki bölüm) çalıştırılır. Ardından koşul tekrardan test edilir ve eğer doğru ise, gövde tekrardan çalıştırılır. Koşul testi sırasında yanlış sonuç ortaya çıkarsa (**fahr** büyüktür **upper**'dan) gövde çalıştırılmaz ve döngü (**loop**) sona erer. Ve programın çalışması döngüden (**loop**) sonrası için olan ifadelerle(**statement**) devam eder. Bu programda döngüden sonra bir ifade (**statement**) yoktu ve bu yüzden program sona erdi.

while'ın gövdesi süslü parantezler arasında bir veya birden çok ifade (**statement**) içerebilir veya süslü parantezler olmadan tek bir ifade (**statement**) içerebilir, örneğin;

```
while (i < j)
    i = 2 * i;
```

Her koşulda, her zaman ifadeleri (**statements**) bir tab (4 boşluk) ile girintileyeceğiz. Bu şekilde sizler göz gezdirirken hangi ifadelerin (**statement**) döngünün (**loop**) içinde olduğunu rahatlıkla görebileceksiniz. Girintileme programın mantıksal yapısını öne çıkartır. Buna karşın C derleyicisi (**compiler**) programın nasıl gözüktüğünü umursamaz, düzgün girintileme ve boşluklama sadece programların insanlar için daha okunabilir olması içindir. Buna karşın insanların girintileme için bazı tutkulu düşünceleri var. Biz popüler tarzlar arasından birini seçtik. Size uygun olanı seçmekten çekinmeyin ve sürekli kullanın.

İşin çoğu döngünün (**loop**) gövde kısmında gerçekleşiyor. Celcius ısı birimi hesaplanıyor ve **celcius** adında bir değişkene (**variable**) ifade (**statement**) tarafından atanıyor.

```
celsius = 5 * (fahr-32) / 9;
```

Sadece 5/9 ile çarpmak yerine 5 ile çarpılıp ardından 9'a bölünmesinin sebebi C'de ve diğer bir çok dilde, tam sayıların (**integer**) bölümlerinin kesirli (ondalıklı) kısımlarının budanması. 5 ve 9 tamsayı (**integer**) olduğundan, 5/9'un sonucunun kesirli kısmı budanırdı, geriye sadece 0 kalırdı ve tüm Selsiyus ısıları 0 olarak raporlanırdı.

Bu örnek **printf**'in nasıl çalıştığını biraz daha gösteriyor. **printf** Bölüm 7'de detaylı açıklayacağımız genel amaçlı çıktı (**output**) formatlama fonksiyonudur (**function**). İlk argümanı, (**argument**) bastırılacak karakter öbeğidir (**string**), her % karakterinin yeri, fonksiyona (**function**) girilen bir diğer argüman (**argument**) tarafından yeri doldurulur; (ikinci, üçüncü, ...) argümanlar (**arguments**) % karakterlerinin yerine geçerler. Örneğin, **%d** bir tamsayı (**integer**) argümanı (**argument**) belirtir. Böylelikle ifade

```
printf("%d\t%d\n", fahr, celsius);
```

iki tam sayının (**integer**) değerlerin (**values**) yani **fahr** ve **celcius**'un aralarında tab(\t) karakteri ile bastırılmasını sağlar.

Her % karakteri **printf**'in ilk argümanından (**argument**) sonraki uyan ilk argümanıyla (**argument**) eşleşir. Argümanlar (**arguments**) ve % karakterleri birbiriyle tür (**type**) olarak ve sıralama olarak eşleşmeli, yoksa yanlış çıktı alırsınız.

Bu arada **printf** C dilinin bir parçası değil: C'nin kendisinde girdi (**input**) ve çıktı (**output**) tanımlı değil. **printf** sadece standart girdi (**input**) çıktı (**output**) kütüphanesinde (**library**) bulunan yararlı bir fonksiyon (**function**). **printf**'in davranışı ANSI standartında belirlenmiştir, böylelikle özellikleri her derleyiciyle (**compiler**) ve kütüphaneyle (**library**) standarta uyar.

C'nin kendisine odaklanmak için, girdi (**input**) ve çıktı (**output**) hakkında Bölüm 7'den önce çok konuşmayacağız. Özellikle, o zamana dek formatlanmış girdiye (**input**) değinmeyeceğiz. Eğer numaraları girdilemeniz (**input**) gerekiyorsa, **scanf** fonksiyonu (**function**) üzerine olan Bölüm 7.4'ü okuyunuz. **scanf**, **printf** gibi, sadece çıktı yazmak yerine girdi okuyor.

Isı birimi dönüştürme programında birtakım problemler var. Bu problemlerden daha basit olanı çıktının (**output**) pek güzel olmaması, çünkü sayılar tam yerinde değil. Bunu düzeltmesi kolay; her % karakterini **printf** ifadesinde (**statement**) bir genişlik ile yazarsak, sayılar alanlarında olması gereken yerlerinde olacaktır. Örneğin

```
    printf("%3d %6d\n", fahr, celsius);
```

ilk sayıyı 3 basamak genişliğinde, ikinci sayıyı da 6 basamak genişliğinde üstteki gibi bastırırsak:

```
  0     -17
 20      -6
 40       4
 60      15
 80      26
100      37
120      48
140      60
160      71
180      82
200      93
220     104
240     115
260     126
280     137
300     148
```

Daha ciddi olan problem ise tam sayı (**integer**) aritmetiği kullandığımızdan dolayı Selsiyus ısıları tam olarak doğru değil; örneğin, 0°F aslında -17.8°C, -17 değil. Daha doğru çıktıları (**output**) almamız için tamsayı (**integer**) yerine ondalıklı (**float**) aritmetik kullanmalıyız. Bu, programda bazı değişiklikleri gerektiriyor. İşte programın ikinci versiyonu:

```
#include <stdio.h>

/* Fahrenhayt-Selsiyus tablosunu bastır

    for fahr = 0, 20, ... 300 */

main()
{
    float fahr, celsius;
    int lower, upper, step;

    lower = 0; /* ısı tablosunun en düşük değeri */
    upper = 300; /* ısı tablosunun en yüksek değeri */
    step = 20; /* adım değeri */

    fahr = lower;

    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0)
        printf("%3.0f\t%6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Bu öncekine çok benziyor, tek farkı **fahr**'ın ve **celsius**'un ondalıklı sayı (**float**) olarak tanımlanmış olması ve dönüşüm için kullanılan formülün daha doğal şekliyle yazılmış olması. 5/9'u önceki versiyonda kullanamıyorduk çünkü iki tam sayının (**integer**) bölümü ondalıklı kısmını budayarak sonucun 0 çıkmasını sağlıyordu. Bir sabitteki (**constant**) ondalık noktası o sabitin türünün (**type**) ondalıklı sayı (**float**) olduğunu belirtir, yani 5.0/9.0 budanmadı çünkü işlem iki ondalıklı sayı (**float**) ile yapıldı.

Eğer aritmetik operatörün tam sayı (**integer**) işlenenleri (**operands**) olursa tam sayılarla (**integers**) gerçekleşen bir işlem yapılırdı. Eğer aritmetik operatörün bir ondalıklı sayı (**float**) ve bir tam sayı (**integer**) işleneni (**operand**) var ise bir şey farketmez ve tam sayı (**integer**) işlemden önce ondalıklı sayıya (**float**) dönüştürülür ve ardından hesaplama yapılırdı. Eğer **fahr-32** yazmış olsaydık, **32** otomatikman ondalıklı sayıya (**float**) dönüştürülürdü. Yine de ondalıklı sayı (**float**) olan sabitleri tam (**integral**) olmalarına rağmen ondalık noktasını kullanarak yazmamız, onların ondalıklı sayı (**float**) doğasını insan okuyucular için vurgular. Tam sayıların (**integer**) ondalıklı sayılara (**float**) dönüşümlerinin detaylarını Bölüm 2'de göreceğiz.

Şuanlık şu atamayı (**assignment**) farkedin

```
    fahr = lower;
```

ve test edin.

```
    while (fahr <= upper)
```

İkisi de çalışıyor, **int** operasyondan önce **float**'a dönüştürülüyor ve ardından hesaplama yapılıyor.

printf dönüşümü belirteçi **%3.0f** ondalıklı sayının (**floating-point number**) (burada fahr) en az 3 karakter genişliğinde, ondalık noktası ve kesir basamağı olmadan bastırılmasını belirtiyor. **%6.1f** diğer bir sayı olan **celcius**'un en az 6 karakter genişliğinde ve ondalık noktasından sonra sadece bir kesir basamağı olacak şekilde bastırılmasını belirtiyor. Çıktı (**output**) şöyle gözüküyor:

```
  0     -17.8
 20      -6.7
 40       4.4
 60      15.6
 80      26.7
100      37.8
120      48.9
140      60.0
160      71.1
180      82.2
200      93.3
220     104.4
240     115.6
260     126.7
280     137.8
300     148.9
```

Genişlik ve nicelik bir belirteçle görmezden gelinebilir: **%6f** sayının en az 6 karakter genişliğinde olmasını belirtiyor; **%.2f** ondalık noktasından sonraki iki karakteri temsil ediyor fakat genişlik zoraki değil ve **%f** sayıyı ondalıklı sayı (**float**) olarak bastırmasını belirtiyor.

```
%d          tamsayı(integer) olarak bastır

%6d         tamsayı(integer) olarak bastır, 6 karakter genişliğinde

%f          ondalıklı sayı(float) olarak bastır

%6f         ondalıklı sayı(float) olarak bastır, 6 karakter
            genişliğinde

%.2f        ondalıklı sayı(float) olarak bastır, ondalık
            noktasından sonra sadece iki karakter

%6.2f       ondalıklı sayı(float) olarak bastır, en az 6 karakter
            genişliğinde ve ondalık noktasından sonra iki karakter
```

Diğerlerininin yanında, **printf** ayrıca **%o** belirtecini sekizlikler(**octal**), **%x** onaltılıklar(**hexademical**) için, **%c**'yi karakterler için, **%s**'yi karakter öbekleri(**string**) için, ve **%%**'yi %'ün kendisi için kullanıyor.

Egersiz 1-3. Isı birimi dönüşüm programını tablonun başındaki yorum satırlarındaki başlığı bastıracak şekilde düzenleyin.

Egzersiz 1-4. Tablodakileri bastıracak benzer bir programı yardım almadan kendiniz yazın.

1.3 For Döngüsü (loop)

Bir programı belirli bir görevi gerçekleştirmesi için çok farklı şekillerde yazabiliriz. Haydi ısı dönüşüm programının bir varyasyonunu oluşturalım.

```
#include <stdio.h>

/* Fahrenhayt-Selsiyus tablosu bastır */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Bu da aynı çıktıyı (**output**) üretecektir, fakat kesinlikle farklı gözüküyor.

En büyük değişiklik birçok değişkenin (**variables**) saf dışı bırakılması; sadece **fahr** kaldı ve onu bir **int** yaptık. **lower** ve **upper** limitleri ve adım büyüklüğü sadece **for** ifadesi (**statement**) içinde sabit (**constant**) olarak karşımızda, artık yeni bir inşâ ve Selsiyus sıcaklığını hesaplayan açıklama ayrı bir ifade (**statement**) yerine artık **printf**'nin üçüncü argümanı (**argument**) olarak karşımızda.

Bu son değişiklik bize bir türün (**type**) değişkeni (**variable**) yerine doğrudan değer (**değer**) kullanmamıza izin verilmesi durumuna bir örnek, bu durumlarda o türün (**type**) daha komplike açıklamaları kullanılabilir. **printf**'in üçüncü argümanı **%6.1f** ile eşleşecek olan bir ondalıklı sayı (**float**) olması gerektiğinden herhangi bir ondalıklı sayı (**float**) oraya konulabilir.

for ifadesi (**statement**) bir döngü (**loop**), **while**'ın bir genellemesi. Eğer erken (**earlier**) **while** ile karşılaştırırsak, işleyişi açıkça görebiliriz. Parantezler içerisinde noktalı virgüller ile ayrılan üç ayrı bölüm mevcut. İlk bölüm, tanımlama:

```
fahr = 0
```

Döngü (**loop**) başlamadan önce sadece bir kere yapılır. İkinci bölüm ise test veya döngüyü (**loop**) kontrol edecek olan koşul bölümüdür:

```
fahr <= 300
```

Bu koşul değerlendirilir; eğer doğru ise, döngünün (**loop**) gövde (**body**) kısmı (burada sadece **printf**) çalıştırılır. Ardından arttırma bölümü

```
fahr = fahr + 20
```

çalıştırılır ve koşul tekrardan test edilir. Döngü (**loop**) eğer koşul yanlış (**false**) olursa sonlanır. **while**'da olduğu gibi gövde (**body**) kısmı tek bir ifadeden (**statement**) veya süslü parantezler arasındaki ifade (**statements**) gruplarından oluşabilir. Tanımlama, koşul ve arttırma bölümleri herhangi bir açıklamadan (**expression**) oluşabilir.

while ve **for** arasındaki seçim isteğe bağlı, hangisi gereken durum için daha temiz görünüyorsa onu kullanın. **while**'dan daha kompakt olduğundan ve döngü (**loop**) kontrol ifadelerini (**statements**) beraber tek bir yerde tuttuğundan, **for** genellikle tanımlama ve arttırmanın tek bir ifade olduğu ve mantıksal olarak ilişkili olduğu döngüler (**loop**) için daha uygundur.

Egzersiz 1-5. Isı dönüşüm programını tabloyu tersine bastıracak şekilde düzenleyin, 300 dereceden 0'a doğru.

1.4 Sembolik Sabitler (constants)

Isı dönüşümünü sonsuza dek bir kenara bırakmadan önce son bir gözlem yapalım. Programa 300 ve 20 gibi "sihirli sayılar" gömmek kötü; bu sayılar bu programı daha sonra okuyacak bir kişinin kolayca görmek ve değiştirmek istediği değerler (**values**) taşıyorlar, bu sayıları bu şekilde gömmek onları sistematik yolla değiştirmeyi zorlaştırdığı gibi hangi sayının neye ait olduğunu da bilmemizi zorlaştırıyor. Bu sihirli sayılarla ilgilenmenin bir yolu onlara anlamlı isimler vermek. #**define** satırı belirli bir karakter öbeğini bir sembolik isim veya sembolik sabit (**constant**) olarak tanımlar:

```
#define isim(name) karakter öbeği
```

Daha sonra programda herhangi bir yerde bu isim (**name**) geçtiğinde, isim (**name**) ile eşleşen karakter öbeği (**string**) ismin yazıldığı yere geçer. İsmin (**name**) formu değişkenlerin (**variables**) adlarının formuyla aynıdır: bir harf ile başlayan, sayı ve harflerden oluşan dizi. Eşleşen karakter öbeği her türlü karakterden oluşabilir, sadece sayılarla sınırlı değildir.

```
#include <stdio.h>

#define   LOWER   0         /* tablonun en düşük limiti */
#define   STEP    20        /* adım büyüklüğü */
#define   UPPER   300       /* en yüksek limiti */

/* Fahrenhayt-Selsiyus tablosunu bastır */
int main () {
    for(int fahr = LOWER; fahr <= UPPER; fahr += STEP) {
        printf("%3d\t%6.1f\n", fahr, (5.0/9.0)*(fahr-32));
    }

}
```

LOWER, **UPPER** ve **STEP** nicelikleri birer sembolik sabittir (**constant**), değişken (**variables**) değil; bu yüzden tanımlamalarla ortaya çıkmıyorlar. Sembolik sabitler (**constant**) genellikle büyük harflerle yazılır böylece kolayca küçük harfli değişken (**variable**) adlarından ayırt edilebilirler. #**define** satırının sonunda noktalı virgül olmadığını farkedin.

1.5 Karakter Girdisi (input) ve Çıktısı (output)

Artık karakter verisi işleyecek, birbiriyle alakalı bir program ailesini konuşacağız. Birçok programın, sadece buradaki prototiplerin genişletilmiş versiyonları olduğunu farkedeceksiniz.

Standart girdi (**input**) çıktı (**output**) kütüphanesinin (**library**) desteklediği girdi (**input**) çıktı (**output**) modeli oldukça basit. Metin girdisi (**input**) ve çıktısı (**output**); nereden kaynaklandığı ve nereye gittiği farketmeksizin, karakter akıntıları (**stream of charachters**) ile yapılır. *Metin akıntısı* (**text stream**) satırlara bölünmüş bir karakter dizisidir; her satır sıfır veya daha fazla karakterden ve onları takiben gelen bir yeni satır karakterinden oluşur. Her girdi (**input**) ve çıktı (**output**) akışını bu modele uydurmak kütüphanenin (**library**) sorumluluğunda; C programcısı kütüphaneyi satırların program dışında nasıl temsil edildiği hakkında endişelenmemek için kullanıyor.

Standart kütüphane (**library**) bize bir kerede bir adet karakter yazdırmak için birçok fonksiyon sağlıyor, **getchar** ve **putchar** en basitleri. Her çağrıldığında **getchar** bir sonraki girdi (**input**) karakterini metin akışından okuyor ve onu kendi değeri (**value**) olarak döndürüyor (**return**). Değişken (**variable**) c

```
c = getchar()
```

bir sonraki girdi (**input**) karakterini içerir. Karakterler genelde klavyeden gelir; dosyalardan girdileri (**input**) Bölüm 7'de konuştuk. **putchar** fonksiyonu (**function**) her çağrıldığında bir karakteri bastırıyor:

```
putchar(c)
```

Tam sayı olan değişken (**variable**) c'yi karakter olarak bastırıyor, genellikle ekrana. **putchar** ve **printf** çağrıları belki iç içe geçmiş olabilir, çıktı (**output**) yapılan çağrıya göre ortaya çıkacaktır.

1.5.1 Dosya Kopyalama

Bilinen **getchar** ve **putchar** ile, girdi (**input**) ve çıktı (**output**) dışında başka bir şey bilmeden şaşırtıcı derecede fazla yararlı kod yazabiliriz. En basit örnek girdiyi (**input**) çıktıya (**output**) bir kerede bir karakter olarak kopyalayan program:

```
karakteri oku
while (karakter dosya-sonu göstergesi değil)
    okuduğun karakteri çıktıla(output)
    karakter oku
```

Bunu C'ye dönüştürürsek:

```
#include <stdio.h>

/* girdiyi (input) çıktıya (output) kopyala; 1. versiyon */
main()
{
    int c;

    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

Mantıksal bir operatör olan **!=** "eşit değil" anlamına gelir.

Tabiki ekrandaki yada klavyedeki karakter, diğer her şey gibi, bit olarak içeride saklanıyor. Tür (**type**) **char** özellikle karakter verisi taşıması için oluşturuldu fakat herhangi bir tam sayı (**integer**) değer kullanılabilir. Biz ince ve önemli bir sebeple **int** kullandık.

Problem, girdinin (**input**) sonunun geçerli veriden ayırılması. Çözüm ise **getchar**'ın girdi (**input**) kalmadığında herhangi gerçek bir karakterle karıştırılamayacak belirgin bir değer dönmesi (**return**). Bu değer **EOF** olarak adlandırılıyor, "dosya sonu" (**end of file**) anlamına geliyor. Biz c'yi **getchar**'ın döndürebileceği kadar büyük bir değer taşıyan bir türde tanımlamamız gerekiyor. c, lazım olduğunda EOF'yi taşıyabileceği kadar büyük olması gerektiğinden **char** kullanamazdık. Bu nedenle **int** kullandık.

EOF <**stdio.h**>'da bir tam sayı (**integer**) olarak tanımlanmıştır, fakat EOF'nin sayı değeri hakkında **char** değeri ile aynı olmadığından dolayı endişelenmemiz gerekmiyor. Sembolik sabiti (**Symbolic Constant**) kullanarak, programdaki hiçbir şeyin belirli bir sayı değerine dayalı olmadığından emin olduk.

Kopyalama programı tecrübeli C programcıları tarafından daha farklı yazılabilir. C'de herhangi bir atama (**assignment**), örneğin

```
c = getchar()
```

bir açıklamadır (**expression**) ve değeri vardır, eşitliğin sol tarafindekinin değeri atamadan (**assignment**) sonra belirlenir. Böylece bir atama (**assignment**) daha geniş bir açıklamanın bir parçası olabilir. Eğer c'nin bir karaktere atanması **while** döngüsünün test kısmının içerisine konulursa, kopyalama programı bu şekilde yazılabilir:

```
#include <stdio.h>

/* girdiyi (input) çıktıya (output) kopyala; 2. versiyon */

main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

while karakteri alıyor, c'ye atıyor, ve karakterin dosya-sonu (**end-of-file**) sinyali olmadığı koşulunu test ediyor. Eğer değilse, **while**'ın gövde (**gövde**) kısmı çalıştırılıyor ve karakter bastırılıyor. Sonra **while** tekrar ediyor. Ve girdinin sonuna ulaşıldığında, **while** sona eriyor ve ardından tabiki **main** de.

Bu versiyon girdiyi (**input**) merkezleştiriyor - artık sadece tek **getchar**'a tek bir referans var - ve programı kısaltıyor. Son program çok daha kompakt, ve bu kullanımda ustalaştıkça, okumak da daha kolay. Bu tarzı çok sık göreceksiniz. (bunu biraz abartmak ve anlaşılmaz kodlar yazmak mümkün, bu kendimizi tutmamız gereken bir eğilim.)

Atamanın etrafındaki koşulun içindeki parantezler gerekli. !='in *önceliği* (**precedence**)=='den yüksek, bu yüzden parantezler olmasaydı mantıksal test olan != atama (**assignment**) olan=='den önce yapılırdı.

Bu yüzden ifade (**statement**)

```
c = getchar() != EOF
```

eşittir

```
c = (getchar() != EOF)
```

Bunun **getchar**'ın dosya-sonu (**end-of-file**) sinyaline ulaşıp ulaşmadığına göre c'yi ya 0'a yada 1' atamak (**assignment**) gibi bir istenmeyen etkisi var. (Bölüm 2'de bununla ilgili fazlası var.)

Egzersiz 1-6. Açıklama (**expression**) getchar() != EOF'nin 0'a yada 1'e eşit olduğunu doğrulayın.

Egzersiz 1.7. EOF'in değerini bastıran bir program yazın.

1.5.2 Karakter Sayma