1

Fonksiyonlar

1.1 Fonksiyon Nedir?

C dili öteki dillere göre küçüktür; çünkü programın yapacağı işler fonksiyonlarla yapılır. Onlar C dilinin birer parçası değildir. Şimdiye dek, *main()* ana fonksiyonu ile *giriş/çıkış* işlerini yapan *scanf()* ve *printf()* fonksiyonlarını çok kullandık. Onların `stdio.h` başlık fonksiyonuna ait olduğunu biliyoruz.

C dili küçüktür. Çünkü, genel olarak, C dilinde her iş için ayrı bir fonksiyon yazılır.

Her C derleyicisi çok kullanılan fonksiyonları içeren zengin bir kütüphaneye sahiptir. Programcının, kütüphanede olan fonksiyonları yeniden yazması önerilmez; onun yerine, kullanacağı fonksiyonları içeren başlık dosyalarını programa çağırması istenir. Tabii, programcı, gerekseme duyduğu zaman kendi fonksiyonunu her zaman yazabilir. Ama varsa o fonksiyonu kütüphaneden çağırmak daha iyidir. Çünkü onları yazmak, çağırmaktan çok daha fazla zaman alır. Üstelik, kütüphnedeki fonksiyonlar çok denenmiştir; hata yapmazlar.

C dilinde fonksiyonlar programın yapı taşlarıdır. Belirli bir işi yapmak için uygun biçimde bir araya getirilmiş C deyimlerinden oluşur. Böyle olduğu için bir programa özgü olarak yazılan bir C fonksiyonu aynı işlevi görmek üzere bir başka C programına çağrılabilir ve orada kullanılabilir. Bu olguya taşınabilme ya da tekrar kullanılabilme (reusability) diyoruz. C dilinde her iş bir fonksiyonla yapılır. *main()* adını alan ana fonksiyon da bir C fonksiyonudur.

Programcılıkta uyulması gereken bir ilke vardır:

Tekeri yeniden keşfetme!

Bunun anlamı açıktır. Bir işi yapan deyimleri bir kez yazdıysan, her gerekişinde onları yeniden yazma; önceden yazdıklarını tekrar kullan!

C dilinde fonksiyon kavramı, tekeri yeniden keşfetmeyi önlemek için vardır. Bir işi yapan deyimleri bir kez yazmışsak, onları bir araya getirip bir dosya olarak kaydederek saklarız. Sonra, her gerektiğinde, onu kaydedildiği yerden çağırıp kullanırız.

Her bilgisayar dili değişken ve fonksiyon kavramlarına sahiptir. Bu tesadüfen değil, işin doğasından kaynaklanan bir durumdur. Değişkenler, bilgisayar programının işleyeceği ham maddelerdir. Bu ham maddelerin işlenip ürün haline getirilmesi için kullanılan araçlar fonksiyonlardır.

Matematikte fonksiyon *girdi*'yi *çıktı*'ya dönüştüren eylemdir. Matematikte, girdi'ye *değişken* değeri, çıktıya *fonksiyon* değeri denilir. Örneğin,

Liste 1.1.

$$f(x) = x^2$$

bağıntısı bir fonksiyon tanımlar; x değişkeni yerine konulacak sayının karesini hesaplar. Burada x değişkeni yerine, tanım bölgesinde olan herhangi sabit bir sayı koyabiliriz. x değişkeni kullanmadan istediğimiz sayının karesini bulan bağıntılar yazabiliriz:

Liste 1.2.

```
f(2) = 4
f(3)= 9
f(4) = 16
...
```

Ancak bu listeye bütün sayıları koyamayız. Çünkü sonsuz çoklukta sayı vardır. Hepsinin karesini veren bir liste oluşturmak olanaksızdır. Onun yerine Liste 1.1 fonksiyonunu tanımlarsak, Liste 1.2 deki gibi imkansız bir liste yapmaya uğraşmak yerine, x değişkeni yerine konulacak her sayının karesini bulan bir araca sahip oluruz. Bu araca *fonksiyon* diyoruz.

Her fonksiyonun bir adı (başlığı) vardır. Fonksiyonun bir ya da birden çok değişkeni olabilir. Örneğin,

Liste 1.3.

```
h(x,y)= x*y
```

verilecek iki sayının çarpımını bulan bir fonksiyondur.

Matematikten bildiğiniz gibi, bir $f : A \to B$ fonksiyonunun tanımlı olması için fonksiyon *adı*'nın, A *tanım* kümesinin, B *değer* kümesinin belirli olması ve tanım kümesindeki her a öğesini değer kümesinde bir (tek) b değerine götürmesi gerekir. Her $a \in A$ öğesini $b \in B$ öğesine eşleyen $a \to b$ dönüşümünü yapan eylemler topluluğu f fonksiyonunun gövdesidir. Birden çok değişken olduğunda da aynı kural işler. Örneğin, Liste 1.3 için *fonksiyonun gövdesi* $x * y$ ifadesidir. Fonksiyon adı h 'dır. Fonksiyonun değişkenleri x ile y 'dir.

Şimdi, Liste 1.1 fonksiyonunu bilgisayar diliyle ifade edelim. Liste 1.4 de verilecek bir tamsayının karesini hesaplar.

Liste 1.4.

```
int KaresiniBul(int x)
{
    int y ;
    y = x * x ;
    return y ;
}
```

Liste 1.5.

```
\emph{Çözümleme:}

Bu basit fonksiyonu kullanarak, bir C fonksiyonunun yapısını
 inceleyebiliriz.

\begin{description}

  \item[başlık (prototype) : ]

1.satırdaki

\begin{lstlisting}
          int KaresiniBul(int x)
```

ya da daha genel olarak, fonksiyonun *veri_tipini*, *adını* ve *parametrelerinin veri_tiplerini* (parametre adı yazılmayabilir) bildiren

Liste 1.6.

```
    veri_tipi   fonksiyonun_adı (parametrelerinin veri tipleri)
```

satırına fonksiyonun *başlığı* (function prototype) diyeceğiz. Parametre sayısı birden çoksa, birbirlerinden virgül ile ayrılırlar. Fonksiyon başlığı, fonksiyon hakkında bize gerekli temel bilgileri verir. Önce başlığın bize verdiği bilgilere bakalım:

Fonksiyonun veri_tipi: En baştaki *veri_tipi* (örneğimizde *int*'dir), fonksiyonun bize vereceği (döndürdüğü) değerin veri_tipini belirtir. C dilinde *int* tipi öntanımlıdır; yani bir değişken ya da fonksiyonun veri_tipi belirtilmezse, C derleyicisi onu *int* sayar. Ama her değişkenin ve her fonksiyonun veri_tipini belirtmeyi alışkanlık edinmeliyiz.

Bazen fonksiyon bir iş yapar ama bir değer vermeyebilir. Değer vermeyecek fonksiyonların veri_tipini ya hiç yazmayız ya da `void` (boş, hiç) olarak belirtebiliriz.

Fonksiyonun adı: Fonksiyonun adı '*KaresiniBul*' bitişik iki sözcüktür. C dilindeki adlandırma kuralına uyar. İstersek *karesini_bul* gibi de yazabiliriz. Matematikte fonksiyonları, genellikle tek harfle gösteririz. Aynı şeyi bilgisayar fonksiyonları için de yapabiliriz. Ama bilgisayarda fonksiyon adlarını, yaptıkları işlevi çağrıştıracak biçimde vermek iyi olur. Uzun bir programda yer alan bir fonksiyon adına bakan kişi, onun ne iş yaptığını sezinlemelidir. Böyle olması, büyük programlarda kaynak programın okur tarafından algılamasını kolaylaştırır.

() parantezi: Her fonksiyonun adının sonuna () parantezinin konulması zorunludur. Böylece, derleyici onun bir fonksiyon olduğunu anlar ve fonksiyon için yapması gereken işleri yapmaya başlar. Parantez yazılmazsa, derleyici sözkonusu adın bir *değişken* ya da *sabit* adı olduğunu sanır; ama değişken listesinde o adı bulamaz. Kaynak programı sözdizimi hatası nedeniyle derleyemez ve hata olduğu uyarısını verir.

Parametreler: C fonksiyonları, işlevleri farklı olan iki tür değişken kullanır. Fonksiyon adını izleyen () parantezi içine yazılanlara *parametre* ya da *argüman* (parameter, argument) denilir. Parametreler birer değişkendir. Hiç parametre olmayabileceği gibi bir ya da daha çok parametre olabilir. Parametre yoksa () prantezi içine hiç bir şey yazılmaz ya da *void* yazılır.

Parantez içine parametrelerin veri tipleri ve adları yazılır (bkz. Program 1.1). Parametrelerin veri tipleri yazılmazsa, derleyici onları int tipinden sayar. Birden çok parametre olduğunda, her birinin veri tipi ve adı yazılır, aralarına virgül (,) konulur.

Parametre olarak, programda daha önce bildirimi yapılan global değişkenlerin adları da kullanılabilir. O durumda bile, parametrenin veri tiplerini tekrar bildirmek gerekir. (bkz. Program 1.2). Çünkü, aynı adı taşısalar bile, derleyici global değişkenler ile parametreleri farklı değişkenler olarak görür ve onlara ana bellekte farklı adresler ayırır. Böyle

olması, global değişkenlerin değerlerinin fonksiyonların parametreleri tarafından değiştirilmesini önler (bkz. Program 1.3).

fonksiyonun gövdesi : Fonksiyon başlığından sonra { } parantezleri içine, fonksiyonun yapacağı işleri yaptıran deyimler yazılır. { } parantezine fonksiyonun *gövdesi* ya da *fonksiyon bloku* denilir.

yerel değişkenler : Gövde içinde fonksiyon için gerekli değişken bildirimleri yapılır. Fonksiyonun gövdesi içinde bildirimi yapılan değişkenlere, o fonksiyonun *yerel* değişkenleri denilir.

Genel olarak, bir { } bloku içinde bildirilen değişkenler, o blokun yerel değişkenleridir. Daha öce yaptığımız gibi, yönlendirmeler ve döngüler için kullandığımız { } blokları içinde bildirimi yapılan değişkenler, o bloklar için yerel değişkendir.

Yerel değişken bildirimi, değişken bildirimi genel kuralına uyar. Liste 1.4'ün 3.satırındaki *int y* bildirimi bir yerel değişken bildirimidir. Yerel değişkenlere ancak ait oldukları blok içinden erişilebilir. Dolayısıyla, fonksiyonun yerel değişkenlerine fonksiyon gövdesi dışından erişilemez.

return : *return* deyimi fonksiyonun bize vereceği değeri bildirir. *return* deyimi, daima gövdedeki son deyim olmalıdır.

1.2 Fonksiyon Başlığını main()'e Önceden Bildir

Daha önce de söylediğimiz gibi, C programlarının çalışması iki temel ilkeye uyar:

1. Yönlendirme ve döngü yoksa, program akışı doğrusaldır; yani deyimler kaynak programa yazıldığı sırayla işler. Buna *doğrusal akış* diyoruz. Yönlendirme ya da döngü olduğunda bile, deyimler yazılış sırasıyla işleme girerler.

2. Bir öğe (değişken, sabit, fonksiyon vb) kullanılmadan önce bildirilmiş olmalıdır.

İkinci koşulu biraz özelleştirerek söylersek, bir değişkenin bildirimi kullanılmadan önce yapılmalıdır. Bir fonksiyonun çağrısında yalnızca fonksiyonun *başlığı* gereklidir. Dolayısıyla, fonksiyon başlığı, çağrının yapıldığı bloktan önce yazılmış olmalıdır.

Fonksiyonun *main()* tarafından çağrıldığını düşünelim. Eğer fonksiyon tanımı *main()* 'den önce yapılmışsa; yani programda *main()*'den önce yazılmışsa, derleyici onu bulur ve gereğini yapar. Ama fonksiyon tanımı *main()*'den sonra yapılmışsa, derleyici onu bulup çağıramaz. O nedenle, fonksiyon tanımı main()'den sonra yapılmış olsa bile, fonksiyon başlığını *main()*'den önce yazmak gerekir. Bunu genelleştirirsek, şu kuralı koyabiliriz:

Kural 1.1. *Çağrılan bir fonksiyon başlığı, çağrının yapıldığı bloktan önce yazılmalıdır.*

Uzun programlarda, fonksiyonların bildirimi, genellikle, *main()*'den sonra yapılır. Bu durumda, çağrılacak fonksiyonların başlığı (prototype) *main()*'den önce yazılır. Böylece, programcı, çağıracağı fonksiyonları derli toplu halde görebilir.

Örnek 1.1.

Bir programda,

Liste 1.7.

```
double carpim(double x, double y){
   return x * y;
}
```

fonksiyonun bildirimi *main()* den sonra yapılmış olsun. Bunu main()'e tanıtmak için, main()'den önce

```
  double carpim(double x, double y);
```

ya da

```
  double carpim(double , double );
```

deyimlerinden birisinin yazılması zorunludur. Aksi halde, derleme hatası doğar. Bunlara fonksiyon başlığı (function prototype) denilir. Birincide x, y parametre adları yazıldığı halde, ikincide yazılmamıştır. Aslında, derleyici başlıktaki parametre adlarını kullanmaz, yalnızca onların veri tipini ister. Ancak, büyük programlarda, fonksiyonların ve parametrelerinin, başkaları için açıklamalarının (documentation) yazılması gerekebilir. O zaman başlık birincide olduğu gibi parametre adlarıyla yazılmalıdır. Değilse, derleyici için ikinci başlık yeterlidir. Birden çok parametre olduğunda, aralarına virgül konulduğuna dikkat ediniz.

1.3 Global değişken, parametre ve yerel değişken

Aşağıdaki örnekler fonksiyon başlıklarının, global değişkenlerin , parametrelerin ve yerel değişkenlerin bildiriminde dikkat edilmesi gereken durumları gösteriyor.

Program 1.1, *hipotenus_bul()* fonksiyonunu, *main()*'den önce tanımlıyor. Bu tanım başlığın (prototype) işlevini üstleniyor. Dolayısıyla, ayrıca fonksiyon başlığını yazmaya gerek kalmıyor. Parametre ve yerel değişken bildirimlerinde değişkenlerin veri tipleri belirtiliyor.

Program 1.1.

```
#include <stdio.h>
#include <math.h>

float hipotenus_bul(int x, int y) {
  float hipotenus ;
  hipotenus = sqrt(x*x + y*y);
  return hipotenus;
}

int main(void) {
  printf("%f"  , hipotenus_bul(3,4));
  return 0;
}
```

Program 1.2 global değişkenlerin adlarının parametre olarak kullanılabileceğini gösteriyor.

Program 1.2.

```
#include <stdio.h>
#include <math.h>

int x=1;
int y=4;
float hipotenus_bul(x, y) {
  float hipotenus ;
  hipotenus = sqrt(x*x + y*y);
  printf("%d  %d \n", &x, &y);
  return hipotenus;
}

int main(void) {
  printf("%d  %d \n", x, y);
  printf("%d  %d \n", &x, &y);
  printf("%f  \n"  , hipotenus_bul(3,4));
  printf("%d  %d \n", x, y);
  return 0;
}
```

```
/**
   1  4
   4206608  4206612
   2293280  2293288
   5.000000
   1  4
  */
```

Açıklamalar:

1. 4 ve 5. satır glaobal x,y değişkenlerinin bildirimidir. 15.satır onların bellek adreslerini yazıyor.

2. 6.satırdaki x,y parametreleri global x,y değişkenleri ile aynı adları taşıyorlar. Parametrede veri tipleri belirtilmediği için, C derleyicisi onları öntanımlı *int* tipinden sayıyor. Parametrelerin adreslerini 9.satır yazıyor. global değişkenler ile parametrelerin adreslerinin farklı olduğu, çıktının 3. ve 4. satırında görülüyor.

3. Parametreler global değişkenlerin adlarını kullansa bile, onlara bellekte ayrı yerler ayrılır. Böylece parametrelere verilen değerlerin global değişkenlerin değerlerini değiştirmesi önlenir.

Program 1.3 parametrelerin global değişkenlerin adını alabileceğini, ama farklı veri tipinden olabileceğini gösteriyor.

Program 1.3.

```
#include <stdio.h>
#include <math.h>

int x=1;
int y=4;
float hipotenus_bul(float x, float y) {
  int a = x;
  int b = y;
  float hipotenus ;
  hipotenus = sqrt(a*a + b*b);
  printf("%d  %d \n", &x, &y);
  return hipotenus;
}

int main(void) {
  printf("%d  %d \n", x, y);
  printf("%d  %d \n", &x, &y);
  printf("%f  \n"  , hipotenus_bul(3.2,4.7));
  printf("%d  %d \n", x, y);
  return 0;
}
```

```
/**
   1  4
   4206608  4206612
   2293280  2293288
   5.000000
   1  4
  */
```

Açıklamalar:

1. 4 ve 5. satır global x,y değişkenlerini *int* olarak bildiriyorr. 17.satır onların bellek adreslerini yazıyor.

2. 6.satırda fonksiyonun x,y parametreleri global x,y değişkenleri ile aynı adları taşıyorlar, ama veri tipleri farklıdır. Parametrelerin adresleri 11 satır ile yazılıyor. global değişkenler ile parametrelerin adreslerinin farklı olduğu, çıktının 3. ve 4. satırında görülüyor.

3. Parametreler global değişkenlerin adlarını kullansa bile, onlar farklı veri tipinden olabilir. Onlara bellekte ayrı yerler ayrılır. Böylece parametrelere verilen değerlerin global değişkenlerin değerlerini değiştirmesi önlenir.

Şimdi parametrelerin veri tipini belirtmezsek ne olacağını Program 1.4 örneği ile gösterelim.

Program 1.4.

```
#include <stdio.h>
#include <math.h>

float x=1.2;
float y=4.3;
float hipotenus_bul(x, y) {
  float hipotenus ;
  hipotenus = sqrt(x*x + y*y);
  printf("%d  %d \n", &x, &y);
  return hipotenus;
}

int main(void) {
  printf("%.2f  %.2f \n", x, y);
  printf("%d  %d \n", &x, &y);
  printf("%f  \n"  , hipotenus_bul(3.7,4.8));
  printf("%f  %f \n", x, y);
  return 0;
}
```

```
/**
   1.20  4.30
   4206608  4206612
```

```
    2293280   2293288
    -1.#IND00
6   1.200000   4.300000

  */
```

Açıklamalar:

1. 4 ve 5. satır, float tipten global x,y değişkenlerinin bildirimidir. 15.satır onların bellek adreslerini yazıyor.

2. 6.satırdaki x,y parametreleri global x,y değişkenleri ile aynı adları taşıyorlar, ama veri tipleri belirtilmediği için, derleyici onları *int* tipinden sayıyor. Ancak fonksiyon çağrısını yapan 16.satır, parametrelere i*nt* olmayan değerler atıyor. O nedenle, *hipotenus_bul()* fonksiyonu çalışmıyor. Çalışmadığını çıktının 4.satırından anlıyoruz.

Fonksiyonun yapacağı işler, sırasıyla { } gövdesine yazılır. Gövdedeki en son deyim *return* deyimidir. Fonksiyonun vereceği değer parantez içine ya da karşısına yazılır. *return* değerin tipi, fonksiyonun veri_tipi ile aynı olmalıdır.

Fonksiyon bir değer vermeyecekse, yalnızca *return* yazılır, değer yeri boş bırakılır. Buna göre, fonksiyoun son deyimi;

```
    return değer;
2   return (değer);
    return ;
```

deyimlerinden birisi olur.

Program akışı fonksiyon gövdesine girince, gövdedeki deyimleri sırasıyla işlemeye başlar. *return* deyimini görünce, gövdeden çıkar. O nedenle, fonksiyonun *return* değeri, gövdeye yazılan son deyim olmalıdır.

Bilgisayar'da *fonksiyon* kavramı, yukarıda anlatılan matematiksel fonksiyon kavramını da içine alan daha genel bir araçtır. O nedenle bazı nesne tabanlı dillerde *fonksiyon* terimi yerine *metot* terimi kullanılır.

Matematikte her fonksiyonun bir (tek) değer döndürmesine alışığız. Ama bilgisayarda bazı fonksiyonlar bir ya da daha çok değer döndürür; bazıları da bir ya da daha çok eylemi gerçekleştirirler ancak hiç bir değer döndürmeyebilirler. Bazı dillerde değer döndürmeyen fonksiyonlara *procedure* deniliyor.

Özetlersek, C fonksiyonun değişkenleri, konumlarına göre ikiye ayrılır. Fonksiyon başlığından hemen sonra yazılan () parantezi içinde olan değişkenlere *argüman* ya da *parametre* denilir. Ayrıca fonksiyon gövdesinde

değişkenler olabilir. Onlara fonksiyonun *yerel değişkenleri* deniliyor. Derleyici her fonksiyonun paramatrelerine ve yerel değişkenlerine, o fonksiyona özgü adresler ayırır. Böylece, fonksiyona ait olan değişkenlerin, aynı adı taşısalar bile başka değişkenlerin değerlerini değiştirmesi önlenmiş olur.

Global Değişken

Programda hiçbir fonksiyonun parametresi (argüman) ya da yerel değişkeni olmayan ve hiç bir bloka ait olmayan değişkenler olabilir. Onlara *global değişkenler* denildiğini biliyoruz. Program 1.3'in 4. ve 5.satırındaki x ile y *global* değişkenlerdir.

1.3.1 Neden Fonksiyon?

Fonksiyonlar veri kümeleri üzerinde yapılması istenen eylemleri tanımlayan genel araçlardır. Fonksiyonların başlıca işlevleri şunlardır:

1. Kodların tekrar tekrar yazılmasını en aza indirirken kullanılmalarını en çoğa çıkarır.
2. Genelleme yapar.
3. Programı yapısal parçalara ayırır; dolayısıyla güncellenmesini kolaylaştırır.
4. Kodların tekrar kullanılmasını (code reuse) sağlar.
5. Kodların taşınmasını sağlar.

Bunların ne anlama geldiğini örneklerle açıklayalım.

Kodların tekrar tekrar yazılmasını önleme : Bin kişilik bir fabrikanın ücret bordrosunu düşünelim. Her çalışanın brüt ücretinden gelir vergisi, sosyal güvenlik kurumu kesintileri gibi kesintiler yapıldıktan sonra emekçinin eline geçecek net ücretin bin kişi için tek tek hesaplanması gerekir. Bu işleri yapacak kodları 1000 kez yazmak mümkündür ama kod tekrarı zaman alıcı bir uğraşa dönüşür. Üstelik, diyelim ki, gelir vergisi dilimlerinde yasal değişiklikler oldu. O değişikliğin 1000 kişi için düzeltilmesi (programın güncellenmesi) gerekir.

Oysa, ücret hesabını yapan kodları bir kez yazıp bir fonksiyon gövdesi içine koysak ve 1000 işçi için o fonksiyonu çağırsak, program çok kısa

olur. Değişiklik gerektiğinde, 1000 yerde güncelleme yerine bir yerde güncelleme yetecektir.

Genelleme Tamsayılar üzerinde iki sayının toplamını düşünelim.

Program 1.5.

```
2 + 3 = 5
34 + 7 =41
...
5000000 + 2000003 = 7000003
...
```

Görüldüğü gibi sonsuz tane toplama işlemi vardır. Hepsini yazmak, listelemek olanaksızdır. Onun yerine

Program 1.6.

```
int toplam(m,n)
    {
     return (m + n);
    }
```

biçiminde genel bir kural tanımlarsak, m ve n değişkenleri yerine istenilen tamsayılar konularak her toplama işlemi yapılabilir. Böylece bütün tamsayı çiftlerinin toplamlarını gösteren imkansız bir listeyi yapma zorunluğu ortadan kalkar.

Programı yapısal parçalara ayırma Gene yukarıdaki ücret bordrosu programını ele alalım. Bu programda emekçilerin kimlikleri ile ilgili bilgilerin okunması, gelir vergisi kesintilerinin hesaplanması, sosyal güvenlik kurumu primlerinin hesaplanması, çıkan sonuçların yazdırılması gibi birbirlerinden farklı işlemler vardır. Bunlar gibi, bir programda birbirlerinden farklı işlemleri ayrı parçalar halinde yazıp, parçaları birleştirmek oldukça rasyonel bir yoldur. Bu parçalardan herhangi birisi üzerinde değişiklik öteki parçaları etkilemeyeceği için, programdaki düzeltmeler ve güncellemeler kolay olur. Genellikle, programın sözkonusu parçaları birer fonksiyon olarak yazılabilir.

Kodların tekrar kullanılması Örneğimizdeki ücret bordrosu programında gelir vergisi hesabını yapan fonksiyon 1000 kez çağrılıyor. Bu kodların tekrar tekrar kullanılmasına (code reuse) iyi bir örnektir.

Kodların taşınması A şirketi için yazılan bir ücret bordrosu programı, üzerinde çok az değişikliklerle başka bir B şirketi için de geçerli kılınabilir. Bunu yaparken, ilk programdaki fonksiyonların çoğunu ikinci programa aynen almak mümkündür. Buna kodların taşınması denilir.

Program 1.6 örneğinde *toplam* fonksiyonun adıdır, m ile n parametreleridir; m+n ise döndürdüğü değerdir (fonksiyonun değeri). Fonksiyon ve değişken adları, ilgili dilin genel adlandırma kurallarına uyulmak koşuluyla, programcı tarafından konulur.

Bilgisayar dillerinde Program 1.6 ile verilen fonksiyon bildirimi, matematikte alıştığımız fonksiyon tanımından biraz farklıdır. ama anlaşılmaları daha kolaydır.

1.3.2 Fonksiyon Bildirimi

Kullanıcı kendisine gerekli olan bir fonksiyonu, derleyiciye gömülü (built-in) ise hemen kullanabilir. Operatörlerin çoğu bu gruptandır. Unutmayalım ki, operatörler de birer fonksiyondur. Çok kullanıldıkları için derleyiciye gömülü delirler. Eğer bir fonksiyon derleyiciye gömülü değil ama kütüphanede ise, o, ait olduğu başlık dosyasından hemen çağırabilir. Bu iki yöntem öncelikli önerilir. Ama, isteniyorsa ya da gerekseme doğmuşsa, kullanıcı kendi fonksiyonlarını her zaman tanımlayabilir. Fonksiyon bildirimi yapılırken C dilinin sözdizimi (syntax) kurallarına uymak gerekir.

Fonksiyon bildirimi için standart bir biçem yoktur. Aynı işi yapan birden çok fonksiyon tanımlanabilir. Sözdizimi doğru ve amacı gerçekleştiren her bildirim geçerlidir. Örneğin, yukarıdaki Program 1.6 fonksiyonunu Program 1.7 gibi de yazabiliriz. İkisi aynı işi yapar.

Program 1.7.

```
1 int toplam(m,n){
      u = m;
      v = n;
      t = u + v;
      return t;
6     }
```

Tabii, mümkün olduğu kadar az kod yazarak amacı gerçekleştirmeye çalışmalıyız. Örnekse, Program 1.6 ile 1.7 aynı işi yapıyorlar. İkincide gereksiz yere üç tane yerel değişken tanımlanıyor.

Başlık

Fonksiyona istenilen ad verilebilir; tabii C dilinin adlandırma kuralları geçerlidir. C dilinin *anahtar* sözcükleri ad olarak kullanılmaz. Fonksiyon adının önüne, fonksiyonun bize geri vereceği değerin tipi yazılır. Buna, kısaca,

fonksiyonun veri tipi diyoruz. Fonksiyon adından hemen sonra () parantezleri mutlaka yazılmalıdır. C derleyicisi () parantezlerini görünce, önündeki adın bir fonksiyon olduğunu anlar. Fonksiyonun parametreleri (argüman, argument) varsa, onlar () parantezi içine yazılır. Birden çok parametre varsa, aralarına virgül (,) konulur. Bazı fonksiyonların parametreleri olmayabilir; ama gene de () parantezleri mutlaka yazılmalıdır.

Gövde

Başlıktan sonra fonksiyonun { } bloku içine yazılan gövdesi başlar. Gövde, fonksiyonun yapacağı işleri yapan deyimleri içerir. Gövdeye gerektiği kadar blok konulur. İçiçe bloklar olabilir. Bloklarda gerektiği kadar deyim olabilir.

return

Fonksiyon bir değer veriyorsa (döndüriyorsa) o değer *return* deyimi ile belirtilir. Fonksiyonun verdiği `return değer`, fonksiyonun veri tipinden olmalıdır. Fonksiyonun veri tipi (return değeri), genellikle, parametrelerine, yerel değişkenlerine ve global değişkenlere bağlı olarak bulunan sonuçtur. Bu değer;

```
return değer;
return (değer);
```

biçimlerinden birisiyle verilir. Bazı metotlar bir iş yapar ama bir değer vermezler. O zaman *return* anahtar sözcüğü karşısına bir şey yazılmaz,

```
return ;
```

yazılır. Gövde içine yazılan bu deyimden sonra fonksiyon tanımı bitmiş olur.

Argüman, Yerel ve Global Değişkenler

Bir fonksiyonun tanımında başlıktaki ( ) parantezi içine yazılan değişkenlere *formal parametre* ya da *argüman* (argument), fonksiyonun gövdesi içine ya da bir blokun içine yazılan değişkenklere *yerel* değişken, hiçbir fonksiyonun gövdesinde ya da blokun içinde olmayan; yani yerel olmayan değişkenlere *global* değişken denildiğini daima anımsamalıyız.

Erişim

Yerel değişkenlere ait oldukları blok dışından erişilemez. Başka bir deyişle, bir fonksiyonun ya da blokun yerel değişkenlerine ancak o fonksiyon ya da

blok içinden erişilebilir. Ama fonksiyon içindeki deyimler, fonksiyonun ait olduğu programın global değişkenlerine erişebilir.

1.4 Fonksiyon Çağrısı

Bir fonksiyonun çağrılması demek, o fonksiyona, *"çalışmaya başla"* buyruğunu vermek demektir. Çağrı yapılınca, program akışı fonksiyonun gövdesine (fonksiyon bloku) girer ve oradaki deyimleri sırayla çalıştırır. Sonra gövdeden çıkar sonraki deyime geçer. Böylece fonksiyon, kendi işlevini yerine getirmiş olur. C dilinde fonksiyon çağrısı iiki türlü olabilir:

1.4.1 Değerle Çağırma

Çağrının gerçekleşebilmesi için, varsa fonksiyonun parametrelerine (arguments) birer gerçek sabit (literal) değerin atanması gerekir. Gerçek değerler doğrudan atanabileceği gibi, atanacak gerçek değeri veren ifadeler de kullanılabilir. Bu durumda bile, gene fonksiyon gerçek değerle çağrılıyor olur.

Bu tür çağrıya, fonksiyonun *değerle çağrılması (call by value)* denir.

C fonksiyonları *içiçe olamaz*; yani bir C fonksiyonunun içinde başka bir C fonksiyonu tanımlanamaz. Ama bir C fonksiyonu başka bir C fonksiyonunu çağırıp çalıştırabilir.

Bildirimi yapılan fonksiyon, programda onun kapsanma alanındaki her yerden çağrılabilir.

Örnekse, Program 1.7'deki bildirimde *toplam* fonksiyonun adıdır. m, n fonksiyonun parametreleridir (argümanlar). u, v, t fonksiyonun *yerel* değişkenleridir. t fonksiyonun döndürdüğü *int* tipinden bir değerdir.

Bu fonksiyonu, 3 ile 5 sayılarını toplatmak için çağırmak istersek,

```
toplam(3, 5) ;
```

yazarız.

Bu çağrıda m,n parametreleri yerine konulan $m = 3$ ve $n = 5$ değerlerine, m ile n parametrelerine atanan *gerçek parametre (literal)* değerleri denilir. Değerle fonksiyon çağrılırken (call by value), parametreler yerine daima buradaki gibi gerçek değerler konulur.

Fonksiyon, bildirimi kaynak program içinde değil, standart kütüphanedeki, örneğin, *aaaa.h* adlı bir başlık dosyasında ise, onu içeren *başlık* dosyası (header file), programın başında

```
#include <aaaa.h>
```

biçiminde çağrılmalıdır. Buna önişlemci buyruğu (preprocessor drective) demiştik. C derleyicisi, asıl programı deremeye geçmeden önce önişlemciye gönderilen buyruklara bakar. Onlar geçerli değilse, programı derlemeye geçmez; hata verir.

Çağrılacak fonksiyon standart kütüphane yerine, örneğin başka bir kütüphanedeki *bbbb.h* başlık dosyasında ise, o dosya

```
#include "bbbb.h"
```

biçiminde çağrılmalıdır.

Şimdiye dek çok kullandığımız *printf()* ve *scanf()* fonkiyonlarının başlıkları (prototype) *stdio.h* başlık dosyası içindedirler. O nedenle, giriş/çıkış eylemi içeren her programın başında

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

çağrısını yapıyoruz.

Değerle yapılan çağrıda, parametrelere verilen gerçek değerler, parametrenin asıl değerini değiştirmez. Bunun nedenini bir örnek üzerinde açıklayabiliriz.

Program 1.8.

```
#include <stdio.h>

int f(int);

int main(){
  int a = 1;
  printf("main() de &a  = %d"  , &a);
  printf("\nmain() de  a  = %d\n"  , a);
  //f(a);
  printf("\nf(a) = %d\n"  , f(a));
  printf("\nmain() icinde  a = %d\n"  , a);
  return 0;
}

int f(int a)
{
  printf("\nf() nin parametresinin adresi &a = %d\n" ,  &a);
  printf("\nf() nin parametresinin degeri    a = %d\n"  , a);
  a = 100;
  printf("\nf() icinde a = %d\n"  , a);
  return a;
}
```

```
/**
  main() de &a  = 2293324
```

```
3  main() de  a  = 1
   f() nin parametresinin adresi &a = 2293280
   f() nin parametresinin degeri    a = 1
8
   f() icinde a = 100
   f(a) = 100
13 main() icinde  a = 1
   */
```

Açıklamalar:

3.satır: f fonksiyonunu main()'e bildiriyor (prototype).

5.satır: int a= 1 bildirimini ve ilk değer atamasını yapıyor. a değişkeni main()'in yerel değişkenidir.

15.satır. f() fonksiyonunun bildirimidir. Parametresine a adının verilmiş olması, onun main()'deki 1 değerini mutlaka alacağı anlamına gelmez. Farklı değer atanabilir. Çünkü, C derleyicisi her ikisine farklı adresler ayırır. Gerçekten, adresleri yazan 7. ve 17.satırların çıktılarına bakarak, adreslerin farklı olduğunu görebiliriz.

18.satır: f() nin a parametresine başka değer atanabilirdi. Atanmayınca, f() fonksiyonu dıştaki bloklara bakıyor ve main()'deki a=1 değerini görüyor. Onu kullanmaya başlıyor.

19.satır: Bu satır a parametresine 100 değerini atıyor. Atanan değer f() nin a parametreine ayrılan adrese yazılıyor (main()'deki a nın adresine değil)

20.satır: f() nin a parametresine atanan 100 değerini yazıyor. 6.satırdaki atamanın yapıldığı adresdeki değer etkilenmiyor.

Sonuç: Buradan çıkan sonuç şudur: Global ya da dış bloktakilerle aynı adı taşısalar bile, fonksiyonun parametrelerine, ayrı adresler ayrılır. Bu adreslere, veri tiplerine uygun atamalar yapılabilir. (bkz. Program 1.2 ve 1.3). Bu durum fonksiyonun yerel değişkenleri için de geçerlidir. Fonksiyonun parametrelerine ya da yerel değişkenlerine hiç atama yapılmadan kullanılınca, fonksiyon dış bloklara bakmaya başlar. Dış bloklarda aynı adlı değişkenler varsa onları kullanır. Sonra parametre ya da yerel değişkenlere yeni atamalar olsa bile, o eylemler kendi adreslerinde gerçekleşir; dış bloktaki değerler etkilenmez.

1.4.2 Referans İle Çağrı

Aslında, C dilinde fonksiyon çağrısı değerle yapılır. Ama bazı hallerde, gerçek değer yerine belirli bir adresteki değeri kullanmak gerekebilir. C dili, *referansla çağırma (call by reference)* denilen o tür çağrılara da izin verir. Referans ile çağrı, bir adres işaret eden *pointer* ile yapılır. O nedenle, bu konuyu pointerleri işlerken yeniden ele alacağız. Referansla çağrıda dış bloktaki değişkenlerin adresleri kullanıldığı için, yapılan atamalar o adreslere yapılmış olur. Dolayısyla, değerle çağırmanın aksine, referansla çağırmada dış bloktaki değişken değeri değişir.

Bir İşi Yapmanın Başka Yolları Olabilir

Bir programın ya da fonksiyonun yaptığı işleri yapan başka programlar ve fonksiyonlar her zaman olabilir. Programın istenen işi eksiksiz yapması, doğru ve güvenilir olması vazgeçilemez hedeftir. Onlardan sonra sistem kaynaklarını en az kullanan, kolay güncellenebilen program tercih nedenidir. Bu açıdan bakınca, programcılıkta bir işi eksiksiz ve doğru yapan en kısa deyimler tercih edilir. Örneğin, Program 1.7 fonksiyonunun yerine, daha az kod kullanan Program 1.6 fonksiyonunu tercih etmeliyiz.

Açıklamalı Örnekler:

Daire Alanı Program 1.9 bir dairenin alanını hesaplıyor.

Program 1.9.

```
#include <stdio.h>

float daireAlani(float r) {
  float pi = 3.1459;
  float alan = pi*r*r;
  return alan;
}

int main() {
  printf("%f"  ,daireAlani(4.7));

  return 0;
}
```

```
/**
69.492928
*/
```

Açıklamalar:

1. 1.satırda *daireAlani* adlı fonksiyon bildirimi başlıyor. Fonksiyonun veri tipi *float*, parametresi *float* tipinden r dir.

2. 4. satırlardaki *pi* değişkeni float tipinden bir yerel değişkendir. pi'ye 3.14159 değeri atanıyor.

3. 5.satırdaki float tipinden olan *alan* değişkenine pi*r*r değeri ataniyor. Bu değer, dairenin formal parametre cinsinden alanıdır.

4. 6.satır, dairenin alanını döndürüyor. Bu değer fonksiyonun formal parametre cinsinden değeridir.

5. 10.satırda *main()* fonksiyonu *daireAlani()* fonksiyonunun r formal parametresi yerine 4.7 gerçek değerini (literal değer) koyarak çağırıyor. r= 4.7 için alan hesabını yapan fonksiyon 69.492928 değerini veriyor.

Şimdi aynı işi yapan başka bir fonksiyon yazalım:

Program 1.10.

```
#include <stdio.h>
2 #define PI 3.14159

float daireAlani(float r) {
  return  PI*r*r;
}
7
int main() {
  printf("%f"  ,daireAlani(4.7));

  return 0;
12 }
```

```
/**
69.397720
3 */
```

Açıklamalar:

1. 2.satırda PI sayısı sabit olarak tanımlanmıştır (Önişlemci)

2. 5 satır tek deyimle istenen alani hesaplayıp veriyor.

3. 8.satırda main() metodu fonksiyonu gerçek parametre değeri ile çağırıyor.

Son olarak, dairenin alanını bulan başka bir program yazalım. Program 1.11 aynı dairenin alanını buluyor.

Program 1.11.

```
#include <stdio.h>
#define PI 3.14159
#define  daireAlani(r) PI*r*r

int main() {
  printf("%f"  ,daireAlani(4.7));

  return 0;
}
```

```
/**
69.397723
*/
```

Açıklamalar:

1. 2. ve 3. satırlarda PI sayısı ve alan fonksiyonu önişlemciler olarak tanımlanıyor.

2. Fonksiyon tanimına gerek kalmadığı için, main() foksiyonu başlıktaki daireAlani() fonksiyonunu r= 4.7 gerçek değeri ile çağırıyor.

Aynı işi yapan bu üç programdan birisini en iyi diye seçmek doğru olmaz. Öğrenme aşamasında en iyi program, öğrencinin anlayarak yazdığı programdır.

Uyarı 1.1. *Aynı işi yapan bu üç programın çıktılarının kesir kısımlarının farklı olduğunu görebiliriz. Bundan çıkaracağımız sonuç şudur:* float *sayıların tamsayı kısmı doğru, ama ondalık kısımları daima* yaklaşık *değerlerdir.*

Celsius-Fahrenheit Program 1.12 verilen *Celsius* derecesini *Fahrenheit* derecesine dönüştürüyor.

Program 1.12.

```
#include<stdio.h>

float celsius_fahrenheit(float) ;  //fonksiyon başlığı (prototype)

int main() {
float c, f ;

printf("\nCelsius dereceyi giriniz :  ");
scanf("%f", &c);

printf("\n celsius %.2f   =   %.2f fahrenheit ", c,
 celsius_fahrenheit(c));
```

```
12
return (0);
}

float celsius_fahrenheit(float celsius){
17 float fahrenheit;
fahrenheit = (1.8 * celsius) + 32;
return fahrenheit;
}
```

```
/**
Fonksiyon 0, 40 ve 100 celsius için çağrılırsa, çıktılar şöyle olur:
  Celsius   0.00  =   32.00 Fahrenheit
  Celsius  40.00  =  104.00 Fahrenheit
5 Celsius 100.00  =  212.00 Fahrenheit
*/
```

Açıklamalar:

1. Bu program, işlemi bir fonsiyon çağrısı ile yapıyor.
2. celsius_Fahrenheit() fonksiyonu main() fonksiyonundan sonra tanımlandığı için, onu çağırmadan önce *main()* fonksiyonuna tanıtmak gerekiyor. Bu işlem 3.satırdaki fonksiyon başlığı (prototype) bildirimi ile yapılıyor.
3. Fonksiyon başlığı (protptype) bildiriminde, parametrelerin adlarını yazmak zorunlu değildir, ama veri tipleri yazılmalıdır. Bu örnekte, yalnızca, parametrenin veri tipi yazılmıştır.

Gerektiğinde varolan fonksiyonları kullanarak başka işler yapan fonksiyonlar tanımlayabiliriz.

Üs Alma Program 1.13 üs alma fonksiyonunu, C dilinin standart kütüphanesinde yer alan *math.h* başlık dosyası içindeki `pow()` fonksiyonu yardımıyla tanımlıyor. Burada yapılan iş, `pow()` fonksiyonuna `us()` adını vermekten başka bir şey değildir.

Program 1.13.

```
#include <stdio.h>
#include <math.h>

4 float us(float,float); //fonksiyon başlığı (prototype)

int main(){
  float a, b;

9 printf("Tabanı ve üssü giriniz");
```

```
  scanf("%f%f"  , &a,&b);

  printf("%.3f"  ,us(a,b));

  return 0;
}

float us(float x, float y){
  return pow(x,y);
}
```

```
/**
Fonksiyon çağrısı:
 us(2,3);
8.000
*/
```

Açıklamalar:

Fonksiyon tanımı, *main()*'den sonra yapıldığı için, 4.satırdai fonksiyon başlığı gereklidir.

Yer Değiştirme (swap)

```
#include <stdio.h>

 void swap(int i, int j){
  printf("Önce     i: %d, j: %d\n", i, j);
      int t = i;
      i = j;
      j = t;
      printf("Sonra    i: %d, j: %d\n\n", i, j);
 }


int main() {
   int t, i = 23, j = 47;
   printf("Önce    i: %d, j: %d\n\n", i, j);
   swap(i,j);
   printf("Sonra   i: %d, j: %d\n", i, j);
   return 0;
}
```

```
/**
Önce    i: 23, j: 47

Önce     i: 23, j: 47
Sonra    i: 47, j: 23

Sonra   i: 23, j: 47
```

Açıklamalar:

1. *takas()* fonksiyonu parametrelerini takas ediyor (yerlerini değiştiriyor).

2. *takas()* fonksiyonu main()'den önce tanımlandığı için, fonksiyon başlığı (prototype) işlevini de görüyor.

3. *takas()* fonksiyonunun i, j parametreleri, main() fonksiyonunun i,j değişenlei ile aynı adı taşıyor. Ama derleyici onlara farklı adresler ayırıyor.

4. 15.satırda takas() fonksiyonu 23 ve 47 gerçek parametrelei ile çağrılıyor. Fonksiyon kendi parametrelerini takas ediyor, ama main()'in değişkenlerine dokunmuyor.

5. Böyle olduğunu görmek için çıktıyı incelemek yetecektir. Çıktının 2.satırı main()'in 14.satırı tarafından yazılıyor. 15.satır takas() fonksiyonunun çağırınca, 4. ve 5. satırlar yazılıyor. Bu satırlarda 23 ve 47 gerçel parametre değerlerinin takas edildiğini görüyoruz. main()'in 16.satırı çıktının 7.satırını yazıyor. 2.ve 7.satırlar, main()'in 23 ve 47 değerlerini alan i, j değişkenleridir. Onların kendi yerlerinde kaldıklarını görüyoruz.

6. Burada aklımıza takılması gereken soru şudur: *main()* fonksiyonuna ait olan ya da global olan iki değişkenin değerleini takas etmek mümkün değil mi? Elbette mümkündür. Bunun için farklı yöntemler izlenebilir. Birincisi takas işlemini yapan kodları main()'in gövdesine yazmaktır (bkz. **??**). İkinci yol, takas işlemini fonksiyon ile yapacaksak, ileride göreceğimiz *pointerleri* kullanabiliriz. (bkz. **??**).

Üç Sayının En Büyüğünü Bulma: Program 1.14, birisi ötekini çağıran iki fonksiyon yardımıyla üç sayının en büyüğünü bulmaktadır.

Program 1.14.

```
#include <stdio.h>

int enBuyuk(int,int,int);

int main() {
  enBuyuk (10,20,30);
    return 0;
}
```

```
  int enBuyuk(a,b,c){
   int m1,m2;
   printf("Sayılar    a= %d, b= %d c = %d\n", a,b,c);
13   m1 = max(a,b) ;
     m2 = max(m1,c);
     return   printf("En büyük sayı %d \n" ,m2);
   }

18 int max(x,y){
   int max;
   if (x < y) max = y;
   else max = x;
   return max;
23 }
```

```
1 /**
    enBuyuk(10, 20, 30);
    30
*/
```

Açıklamalar:

1. Program 1.14, main() dışında *max()* ve *enBuyuk()* adlı iki fonksiyon tanımlıyor.

2. 6.satırda *main()* fonksiyonu *enBuyuk()* adlı fonksiyonu çağırıyor.

3. 13. ve 14. satırlarda *enBuyuk()* fonksiyonu *max()* fonksiyonunu iki kez çağırıyor. Her çağrıda *max()* fonksiyonuna farklı gerçek parametre değerleri veriyor. Bir fonksiyonun başka bir fonksiyonu çağırabileceğini biliyoruz.

4. *Uyarı:* Üç sayı arasınad en büyüğünü bulmak için çok daha kısa kodlar yazabiliriz. Ama burada yapılmak istenen şey, fonksiyonun başka bir fonksiyonu çağırmasını göstermektir.

Üçlemeyi (koşullu operatör, ternary operator) kullanarak önceki programı daha tıkız duruma getirebiliriz. Program 1.15 o işi yapıyor. Deyimleri inceleyerek, metodun isteneni nasıl yaptığını açıklayınız.

Program 1.15.

```
1 #include<stdio.h>


int buyuk(int a, int b, int c){
  int max;
6 max =(a>b&&a>c?a:b>c?b:c);
  return max;
}
```

```
int main(){
int a,b,c ;
printf("\n3 sayi giriniz:");
scanf("%d %d %d",&a,&b,&c);
printf("\nGirilen sayilarin en buyugu: %d", buyuk(a,b,c));

return 0;
}
```

```
/**
   buyuk(4,-7,-9);
   4
*/
```

1.5 C Dilinin Fonksiyonları

Her dilde olduğu gibi, *C dili*'nde de fonksiyonları üç kaynaktan elde edebiliriz: [**?**]

1. Derleyiciye gömülü olan (built-in) fonksiyonlar (operatörler)
2. Bir dış kütüphaneden çağrılan fonksiyonlar
3. Kullanıcının tanımladığı fonksiyonlar

Standart C kütphanesinde çok sayıda fonksiyon vardır. Bunlar konularına göre gruplanarak ayrı başlık dosyalarına (header files) konulmuşlardır. Programcıların çok kullndığı fonksiyonlar standart kütüphanede vardır. Ayrıca, üçüncü kişi ya da şirketlerin yazdığı ama standart kütüphane dışında kalan çok fonksiyon vardır. Bütün bunlar yetmediği zaman programcı, her zaman kendi fonksiyonunu yazabilir. İsterse onları ayrı bir başlık dosyası haline getirebilir ve her istediğinde kullanabilir. C dilinin satandart kütüphanesinde olan önemli başlık dosyalarını ve onların içerdiği başlıca fonksiyonları sonraki bölümde ele alacağız. Bu kesimde, yukarıda listelenen türlere birer örnek vermekle yetineceğiz.

Program 1.16 verilen iki sayının toplamını buluyor. + operatörü derleyiciye gömülüdür. Onu çağırmaya ya da yeniden tanımlamaya gerek yoktur.

Program 1.16.

```
#include <stdio.h>

int main(){
  int a = -25, b= 29;
  printf("%d" , a + b);

  return 0;
}
```

Buradai (+) operatörü iki sayıyı toplayan bir fonksiyondur. Bu fonksiyon derleyiciye gömülü olduğu için, onu başka yerden çağırmaya gerek olmaksızın, programın istediğimiz yerinde kullanabiliriz.

Öte yandan, bir sayının kare kökünü bulan *sqrt()* fonksiyonu, derleyiciye gömülü değil, *math.h* başlık dosyası içindedir. Onu kullanabilmek için, proram *math.h* dosyasını çağırmak gerekir:

Program 1.17 verilen bir sayının kare kökünü buluyor. Bu fonksiyon math modülü içindedir. Onu kullanmak için *math.h* başlık dosyası çağrılır.

Program 1.17.

```
#include <stdio.h>
#include <math.h>

int main(){
  int a = 49;
  printf("%.2f" , sqrt(a));

  return 0;
}
```

```
/*
7.00
*/
```

Program 1.18 verilen bir sayının kaç hanesi (sayak) olduğunu buluyor. Bu fonksiyon derleyiciye gömülü olmadığı gibi, bir dış kütüphane de yoktur. Böyle durumlarda, kullanıcı gerekli gördüğü fonksiyonu tanımlayabilir:

Program 1.18.

```
#include<stdio.h>

int sayak(int) ;

int main(){
  int n ;
       printf("Bir sayı giriniz: ");
     scanf("%d",&n);
  sayak(n) ;
```

```
  printf("Toplam sayak sayısı: %d",sayak(n));

  return 0;
}

int sayak(int n){
    int sayac=0;

   while(n){
       n=n/10;
       sayac++;
     }
  return sayac;
  printf("Toplam sayak sayısı: %d",sayac);
  return 0;
}
```

```
/**
   sayak(98765);
   5
*/
```

1.6 Kapsanma Alanı (scope)

Programda tanımlanan bir öğenin (değişken, fonksiyon, vb) *kapsanma alanı (scope)*, ona erişilebildiği bloklardır. Fonksiyon bildirimi yapıldıktan sonra, onun kapsanma alanından istenen sayıda çağrı yapılabilir. Her çağrıda fonksiyonun formal parametrelerine gerçek (literal) değerler atanır. Tabii, farklı çağrılarda farklı değerler atanabilir. Program 1.19 örneğinde, kareBul fonksiyonu $x = 3$ ve $x = 7.2$ literal değerleri için iki kez ayrı ayrı çağrılıyor.

Program 1.19.

```
#include <stdio.h>

float kareBul (float);  //prototype

int main() {
  printf("%.2f"  ,kareBul(7.2));
}

float kareBul(float x) {
  return x * x;
}
```

```
/**
  kareBul(3);
  9
  kareBul(7.2)
  51.84
*/
```

Program 1.20 fonksiyonunun birden çok parametresi var. Return değeri yoktur. Fonksiyon bir iş yapıyor, ama *return* değeri vermiyor.

Fonksiyon çağrısı yapılırken, parametrelerin sırasının karışmaması gerekir. Çağrı yapılırken parametrelerin sırasına ve veri tipine kesinlikle uymak gerekir.

Program 1.20.

```
#include <stdio.h>

 void  liste_yaz(int no, int yas, float puan);  //prototype

int main() {
  liste_yaz (12345, 21,85);
}


 void  liste_yaz(int no, int yas, float puan){

        printf( "\nOgrenci No:  %d", no );
        printf( "\nYas        : %d", yas );
        printf( "\nPuan       : %d", puan );
  return ;
}
```

```
/**
   Ogrenci no: 12345
   Yas       : 21
   Puan      : 85.70
*/
```

Program 1.20 fonksiyonunu liste_yaz(85.7, 12345, 21) diye çağırmayı deneyiniz. C derleyicisi veri tipi denetimi yapmadığı için, çıktı aşağıdaki gibi olur.

```
Ogrenci No:  85
Yas       : 12345
Puan      : 21.00
```

Tabii, bunun ciddi bir kullanım hatası olduğu açıktır. Derlendiği halde, mantıksal hatalar yapan program hataları en tehlikeli hatalardan sayılır. Çünkü, programın mantıksal hata yaptığı anlaşılmayabilir ve kullananlara zarar verebilir.

Bazı diller veri tipi denetimi yapar. Hatalı veri girildiğinde program derlenmez ve koşmaz. Dolayısıyla yukarıdaki gibi ölümcül hatalar oluşmaz.

1.7 *global* Değişken Örnekleri

Yerel olmayan değişkene *global* değişken demiştik.

Program 1.21.

```
#include<stdio.h>

int a,b;     // global değişken

int topla() {
  return a + b;
}

int main() {
  int sonuc; // yerel değişken
  a = 4;
  b = 8;
  sonuc = topla();
  printf("%d\n" ,sonuc);
  return 0;
}
```

```
/**
  12
*/
```

Program 1.21 ile tanımlanan *topla()* fonksiyonu, global a ve b değişkenlerini kullanıyor.

Program 1.22.

```
#include <stdio.h>
#include <stdlib.h>

int varGlobal;

void function(int i) {
  static int a=0;
  a++;
  int t=i;
  i=varGlobal;
  varGlobal=t;
  printf("Cagri #%d:\ni=%d\nvarGlobal=%d\n\n",a,i,varGlobal,t);
}

int main() {
  function(2);
  function(7);
  function(12);
  return 0;
}
```

```
/**
Cagri #1:
i=0
varGlobal=2

Cagri #2:
```

```
i=2
varGlobal=7

Cagri #3:
i=7
varGlobal=12
*/
```

func2() fonksiyonunun x adlı yerel değişkeni olmadığı için doğrudan global x değişkenine erişiyor ve onu kullanıyor.

Bazen, bir fonksiyonun yerel değişkeni ile aduzayındaki değişken aynı adı taşıyabilir. Fonksiyon, normal durumda kendi yerel değişkenlerini kullanır. Ama, fonksiyonun kendi yerel değişkenini değil, aduzayındaki global değişkeni kullanması isteniyorsa, değişkenin önüne *global* anahtar terimini koymak gerekir.

Program 1.23.

```
#include <stdio.h>
#include <stdlib.h>

int x,y, max;

/* İki sayının maksimumunu bul */
 int maksimum( ){
    if (x >= y)
        max = x;
    else
        max = y;
    return max;
}

int main(){
    x = printf("İlk sayiyi giriniz     :"); scanf("%d"  ,&x);
    y = printf("ikinci sayiyi giriniz :" ); scanf("%d"  ,&y);
    max = maksimum(x,y);
    printf("%d ile %d sayilarinin maximumu : %d"  ,x,y, max);
}
```

```
/**
İlk sayyıyı giriniz     :-25
ikinci sayıyı giriniz   :54
-45 67  sayılarının maximumu : 54 dir
*/
```

Program 1.24.

```
#include <stdio.h>
#include <stdlib.h>

char ch;
float puan;
```

```
float r;

char notVer(float puan) {
  /*Bu fonksiyon puanları harf notuna dönüştürür.*/
  if (puan >= 90.0)
    printf("A");
  else if (puan >= 80.0)
    printf("B");
  else if (puan >= 70.0)
    printf("C");
  else if (puan >= 60.0)
    printf("D");
  else
    printf("F");
  return;
}


int main() {

  printf("Puanı giriniz : ");
  scanf("%f"  ,&r);
  printf("Aldığınız not : %c" , notVer(r));
}
```

```
/**
Puanı giriniz :  87.3
Aldığınız not :  B
*/
```