

FONKSİYONLAR

Fonksiyon Nedir?

Yapısal Programlama mantığında çözülecek problem genel olarak fonksiyonların birbirini çağırmasıyla yapıldığı anlatılmıştı. Yani kodlaması yapılacak işi birbirini çağıran fonksiyonlar yazarak ana fonksiyondan bu fonksiyonları çağırarak yaparız.

Ayrıca yazılacak programda birbiriyle ilgili fonksiyonlar bir araya toplanarak ayrı bir dosyada tutulur. Bu durum *Modüler Programlama* başlığı altında anlatılmıştı. Bu bölüme kadar gördüğümüz kodlarda çözümü oluşturan tüm tasarım parçaları ana fonksiyon olan `main()` fonksiyonu içerisine çözülmüştür. Bir bilgisayar programı, genel olarak, tek başına tüm görevleri yerine getiren tek bir main fonksiyonundan oluşmaz. Bu durumda ana problemin büyüklüğüne göre ana fonksiyon bloğumuz uzar, okunabilirliği azalır, müdahale etmek zorlaşır. Bu tip büyük programları kendi içerisinde, her biri özel bir işi çözmek için tasarlanmış parçacıklarından oluşturmak daha mantıklı olacaktır. Yani **böl ve yönet** (**divide and conquer**) tekniği uygulanır. Çünkü büyük bir problemin toptan çözülmesi, problemin parçalar halinde çözülmesinden her zaman daha yorucu ve zordur.

Yazılım geliştirmede problemi büyük ana parçalara ayırırız. Daha sonra bu büyük parçaları daha küçük parçalara ayırarak çözeriz. Buna **yukarıdan aşağıya** (**top down**) yaklaşım adı verilir.

C dilinde fonksiyonlar;

- Fonksiyonlar, belirli bir görevi gerçekleştiren bir kod bloğudur.
- Parametrelili alabilir, girdilerle ya da parametresiz bir şeyler yapar ve ardından cevabı verebilir. Kısaca bilgiyi fonksiyona argümanlar ile aktarırız.
- Her fonksiyonun bir kimliği vardır. (*Değişken Kimliklendirme Kuralları* burada da geçerlidir.)
- Bir fonksiyon program içerisinde istenildiği kadar farklı yerlerden ulaşılarak çalıştırılabilir ya da çağrılabilir. Bir başka deyişle tekrar kullanılabilir.
- Bir fonksiyon, çağıran koda bir değer döndürebilir.
- Bir fonksiyon, bir başka fonksiyondan çağrılabilir.

C dilinde iki tip fonksiyon bulunmaktadır; başlık dosyalarında bize sunulan hazır fonksiyonlar ve programcının tanımlayacağı **kullanıcı tanımlı fonksiyonlar** (**user defined function**).

Hazır Fonksiyonlar

C geliştiricileri, programcılarının kullanmaları için, önceden yazılmış olan hazır fonksiyonlar hazırlamışlardır. Hazır fonksiyonlar teknik olarak C dilinin parçası değildir. Yalnızca standart hale getirilmişlerdir.

Programcı, kullanmak istediği hazır fonksiyonları; hangi modülde ise o modüle ilişkin **başlık** (**header**) dosyasını **#include ön işlemci yönergesi** (**preprocessor directive**) ile kendi projesine dahil eder. Bunu *C PROGRAMLAMA DİLİNE GİRİŞ* başlığı altında **stdio.h** başlığı için incelemiştik.

Aslında bu başlık dosyalarında;

- Hazır fonksiyonların sadece **prototipleri** (**prototype**) bulunur.
- Fonksiyonların kendileri yani derlenmiş halleri her işletim sistemi için ayrı oluşturulmuş **kütüphane** (**library**) dosyaları içerisinde bulunur.
- Derleme işlemi sırasında **bağlayıcı** (**linker**) tarafından bu fonksiyonlar **icra edilebilir dosyaya** (**executable file**) dahil edilir.

Math.h Başlık Dosyası

Matematiksel iş ve işlemleri gerçekleştirmek için standart hale getirilmiş bir başlık dosyasıdır.

Fonksiyon prototipi	Açıklama	Örnek
<code>double sqrt(double);</code>	\sqrt{x}	<code>sqrt(900.0)=30.0</code>
<code>double exp(double);</code>	e^x	<code>exp(1.0)=2.718282</code>

Fonksiyon prototipi	Açıklama	Örnek
<code>double log(double);</code>	$\ln(x)$	$\log(2.718282)=1.0$
<code>double log10(double);</code>	$\log(x)$	$\log_{10}(1.0)=0.0$
<code>double fabs(double);</code>	$ x $	$\text{fabs}(-5.0)=5.0$
<code>double ceil(double);</code>	x'i kendisinden büyük en küçük kesirsiz sayıya yuvarlar.	$\text{ceil}(-9.8)=-9.0$ $\text{ceil}(9.2)=10.0$
<code>double floor(double);</code>	x'i kendisinden küçük en büyük kesirsiz sayıya yuvarlar.	$\text{floor}(-9.8)=-10.0$ $\text{floor}(9.2)=9.0$
<code>double pow(double x, double y);</code>	x^y	$\text{pow}(2.0,3.0)=8.0$
<code>double fmod(double x, double y);</code>	Kayan noktalı sayılarda x'in y'ye bölümünden kalanı verir.	$\text{fmod}(13.657,2.333)=1.1992$
<code>double sin(double);</code>	Radyan cinsinden x'in sinüsü	$\sin(0.0)=0.0$
<code>double cos(double);</code>	Radyan cinsinden x'in kosinüsü	$\cos(0.0)=1.0$
<code>double tan(double);</code>	Radyan cinsinden x'in tanjantı	$\tan(0.0)=0.0$

Tablo 20. Math.h Başlık Dosyası Fonksiyonları

Örnek olarak kenarları verilen bir üçgenin alanını hesaplayan bir program aşağıda verilmiştir.

```

/*AREA OF A TRIANGLE - HERON'S FORMULA */
#include <stdio.h>
#include <math.h>
int main()
{
    int kenar1,kenar2,kenar3;
    double alan,kenarlarToplamininYarisi;
    printf("Üçgenin Kenarlarını Giriniz:");
    scanf("%d%d%d",&kenar1,&kenar2,&kenar3);

    if ((kenar1+kenar2 <= kenar3) ||
        (kenar2+kenar3 <= kenar1) ||
        (kenar1+kenar3 <= kenar2) ) {
        printf("Böyle bir üçgen olamaz.\n");
        return 1;
    }

    kenarlarToplamininYarisi=(kenar1+kenar2+kenar3)/2.0;
    alan=sqrt( kenarlarToplamininYarisi*
               (kenarlarToplamininYarisi-kenar1)*
               (kenarlarToplamininYarisi-kenar2)*
               (kenarlarToplamininYarisi-kenar3) );
    printf("Uçgenin alanı: %.2f",alan );
    return 0;
}
/*Program Çıktısı:
Üçgenin Kenarlarını Giriniz:3 10 12
Uçgenin alanı: 12.18

...Program finished with exit code 0
*/

```

Stdlib.h Başlık Dosyası

Bu kütüphane içerisinde; Alfa sayısal metinleri ilkel veri tiplerine veya bunun tersini yapan tür dönüşüm fonksiyonları (`atoi`, `itoa`, `atof`, `strtod`, ... ileride anlatılacaktır.), Çalıştırma anında bellek tahsisi ve tahsisli

belleğin serbest bırakılmasını sağlayan hafıza yerleşim fonksiyonları (**malloc**, **free**... ileride anlatılacaktır.) Rastgele sayı üretme fonksiyonları (**rand**, **srand**) bulunur. Şimdilik rastgele sayı üretme üzerinde durulacaktır.

Fonksiyon	Açıklama
int rand();	0 ile RAND_MAX (32767) arasında rastgele bir sayı üretir.
int srand(unsigned int);	rand() fonksiyonu belli bir başlangıç değerinden itibaren bir dizi matematiksel işlem sonucu rastgele bir değer üretir. srand() fonksiyonu, rand() fonksiyonunun başlangıç değerini farklı belirlemek için kullanılır.

Tablo 21. Stdlib.h Başlık Dosyası Fonksiyonları

Aşağıda bir zar için rastgele sayı üreten bir program verilmiştir.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int rastgeleZar;
    rastgeleZar=1 + rand() %6;
    /*
        rand() fonksiyonunun üreteceği sayıyı kalan işleci ile
        6 sayısına böleriz ki 0 ile 5 arasında bir sayı elde edelim.
        Buna da 1 ekler isek zarın rakamları ortaya çıkar.
    */
    printf("Atılan Zar:%d",rastgeleZar);
    return 0;
}
```

Fakat programın her çalışmasında **rand()** aynı başlangıç değerini kullandığından aynı rastgele değerleri üretir. Bu nedenle aynı zar rakamı gelmiş olur. Bunu değiştirmek için **srand()** fonksiyonu kullanılır.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int rastgeleZar;
    int randBaslangicDegeri;

    printf("rand() için başlangıç değeri olabilecek bir sayı giriniz:");
    scanf("%d",&randBaslangicDegeri);
    srand(randBaslangicDegeri);

    rastgeleZar=1 + rand() %6;
    printf("Atılan Zar:%d",rastgeleZar);
    return 0;
}
```

Rastgele sayı her seferinde kullanıcıdan alınan sayıya göre hesaplandığından her seferinde farklı bir zar rakamı üretilmiş olacaktır. Ancak başlangıç değerinin her seferinde kullanıcıdan alınması bir başka problemdir. Bunun üstesinden gelmek için bilgisayarın saatini sayı olarak veren **time.h** başlık dosyasındaki **time()** fonksiyonu **NULL** parametresiyle kullanılabilir.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
    int rastgeleZar;
    int randBaslangicDegeri=time(NULL);
    srand(randBaslangicDegeri); // srand(time(NULL)); olarak da kullanılabilir.
    rastgeleZar=1 + rand() %6;

    printf("Atılan Zar:%d",rastgeleZar);
    return 0;
}
```

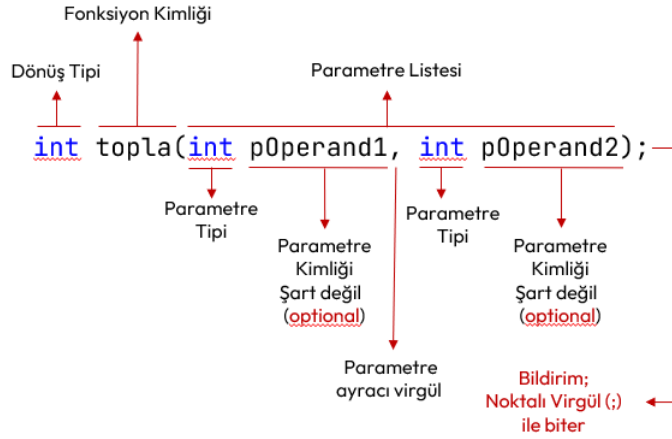
}

Kullanıcı Tanımlı Fonksiyonlar

Programcılar kendi fonksiyonlarını oluşturarak kodlarını daha kullanışlı hale getirirler. Bu tür fonksiyonlara **kullanıcı tanımlı fonksiyonlar** (**user defined function**) denilir. Kullanıcı tanımlı fonksiyonu oluşturmak ve kullanmak için bu üç unsuru bilmemiz gerekir;

- **Fonksiyon Bildirimi:** Bir fonksiyonu çalıştıran koda, çağıran program adı verilir. Çağıran program kullanılacak fonksiyonu bilmeli. Bunun için çağıran program öncesinde **fonksiyon bildirimi** (**function declaration**) yapılmalı veya prototipi (**function prototype**) tanımlanmalıdır. Bildirim derleyiciye böyle bir fonksiyon var demenin yoludur.
- **Fonksiyon Tanımı:** Fonksiyon tanımı (**function definition**), bir fonksiyonun girdileri, yaptığı işlem ve döndüreceği değeri içerecek şekilde başlık ve gövdesinin kodlanmasından oluşur. Bildirimi yapılan fonksiyonun derleme yapılabilmesi için eksiksiz bir şekilde tamamlar. Bildirime benzer başlık ve tırnaklı parantez { } kod bloğu ve arasında yer alan kodlardan oluşur. Bildirim yapılmadan da fonksiyon tanımlanabilir.
- **Fonksiyon Çağırma:** Fonksiyonu çağırmak (**function call/invoke function**) için fonksiyonun kimliğini ve ardından parantez içindeki argüman listesini yazmanız yeterlidir.

Fonksiyon Bildirimi Nasıl Yapılır?



Şekil 18. Fonksiyon Bildirimi Örneği

Yukarıda örneği verilen **fonksiyon bildiriminde** (**function declaration**); Dönüş tipi, işlevin döndürdüğü değerin veri tipidir. Bazı işlevler, herhangi bir değer döndürmeden istenen işlemleri gerçekleştirir. Bu durumda dönüş tipi **void** anahtar sözcüğüdür ve bu fonksiyonlar **yordam** (**procedure**) ya da alt **rutin** (**subroutine**) olarak adlandırılır.

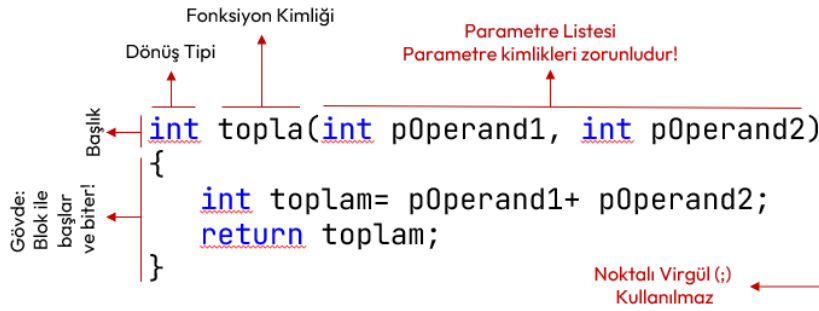
```
/* Fonksiyon Bildirimleri */
int ikiSayiyiTopla(int,int); /* ikiSayiyiTopla kimlikli bir fonksiyon olduğu ve
bu fonksiyonun iki tamsayı parametre aldığı ve
bir tamsayı geri döndürdüğü derleyiciye bildiriliyor. */
void sayiyiKonsolaYaz(int); /* sayiyiKonsolaYaz kimlikli bir fonksiyon olduğu ve
bu fonksiyonun bir tamsayı parametre aldığı ve
geri değer döndürmediği derleyiciye bildiriliyor. */
int konsoldanTamsayiOku(); /* konsoldanTamsayiOku kimlikli bir fonksiyon olduğu ve
bu fonksiyonun bir parametre almadığı ve
bir tamsayı geri döndürdüğü derleyiciye bildiriliyor. */

int main() {
    sayiyiKonsolaYaz(13);
    int toplam= ikiSayiyiTopla (10,20);
    //...
    return 0;
}
```

Fonksiyon adı, fonksiyonun benzersiz kimliğidir ve kimliklendirme, değişken kimliklendirme kuralları burada da geçerlidir. Bağımsız değişkenler parametre olarak adlandırılır. Bir fonksiyon çağrıldığı anda parametreye gerçek parametre ya da argüman adı verilir. Parametre listesi, bir işlevin parametrelerinin veri tipini, sırasını ve sayısını belirtir. Parametreler isteğe bağlıdır; yani bir fonksiyon hiçbir parametre içermeyebilir.

Fonksiyona ilişkin bir bildirim yapıldığında bir yerlerde böyle bir fonksiyon olduğu derleyiciye iletilmiş olur. Bildirim sonrasında artık onu çağırabiliriz. Tanımı daha sonradan yapılabilir. Bildirim yapılmaz ise fonksiyon tanımlanmalıdır.

Fonksiyon Tanımı Nasıl Yapılır?



Şekil 19. Fonksiyon Tanım Örneği

Bir fonksiyon bir değer döndürebilir. Dönüş tipi, işlevin döndürdüğü değer için veri tipidir.

Fonksiyon adı, fonksiyonun benzersiz kimliğidir ve kimliklendirmede, değişken kimliklendirme kuralları geçerlidir. Eğer daha önce **fonksiyon bildirimi** (**function declaration**) yapılmış ise aynı kimlik kullanılır.

Bağımsız değişkenler parametre olarak adlandırılır. Parametre listesi, bir işlevin parametrelerinin veri tipini, sırasını ve sayısını belirtir. Parametreler isteğe bağlıdır; yani bir fonksiyon hiçbir parametre içermeyebilir. Bildirimden farklı olarak her bir parametreye kimlik verilmelidir.

Fonksiyon Gövdesi, fonksiyonun ne yaptığını tanımlayan talimatları içerir. Yapısal programlama yapıldığından ilk önce değişkenler tanımlanmalı daha sonra bu değişkenleri işleyecek talimatlar yazılmalıdır. Fonksiyonda istenilen sonuca ulaşmak için kullanacağı adımlara karar verilir yani algoritması belirlenir.

Dönüş tipine uygun olarak bir değer, return talimatı ile geri döndürülmelidir. Return talimatı çoğunlukla fonksiyondaki son talimat olarak görünür.

Return talimatı, bir fonksiyonun yürütülmesini sonlandırır ve kontrolü çağrıldığı fonksiyona geri verir. Yani programın icrasına devam etmek için çağrıldığı yere dönlür. Bir fonksiyonun tanımında yer alan dönüş tipi ile return talimatına ilişkin ifade sonucu aynı veri tipinde olmalıdır.

Fonksiyonun çağrıldığı yerde, fonksiyonun döndürdüğü değer için tipi ile eşleşen bir değişkene dönüş değeri atanabilir.

C dilinde bir fonksiyon gövdesinde bir başka fonksiyon tanımlanamaz!

Kullanıcı Tanımlı Fonksiyon Örnekleri

Aşağıda toplam adlı bir kullanıcı tanımlı fonksiyona ilişkin program verilmiştir.

```
#include <stdio.h>

int toplam(int,int); /* kullanıcı tanımlı toplam fonksiyonu bildirimi
                       (declaration)/prototipi
                       Bu bildirimde int geri döndüren ve
                       int tipinde iki farklı parametre alan
                       toplam fonksiyonu derleyiciye bildirildi.*/

int main () {
    int sonuc;
```

```

/* ÇAĞRI ORTAMI:
   ana fonksiyon içinde topla fonksiyonu çağrılacak.
*/
topla(1,2); /* topla fonksiyonu çağrılıyor ama geri
              döndürdüğü değer kullanılmıyor. */
printf("toplam: %d\n", topla(2,3)); /* topla fonksiyonu çağrılıyor.
                                      Geri döndürdüğü değer printf
                                      fonksiyonuna
                                      argüman olarak giriyor. */
sonuc= topla(3,4); /* topla fonksiyonu çağrılıyor.
                   Geri döndürdüğü değer
                   sonuc değişkenine atanıyor. */
printf("toplam: %d\n",sonuc);
sonuc= topla(1,2)+topla(3,4); /* topla fonksiyonu iki kez çağrılıyor.
                              Her bir çağrıdaki geri dönüş değeri
                              geçici bir yerde saklanıp toplamı
                              sonuc değişkenine atanıyor. */

printf("toplam: %d\n",sonuc);
sonuc= topla(topla(1,2),3)+topla(4,5); /* ... */
printf("toplam: %d\n",sonuc);
return 0;
}
/*
Kullanıcı tanımlı topla fonksiyonu burada tanımlanıyor (definition):
*/
int topla(int p0operand1,int p0operand2) //Fonksiyon başlığı
{ //Fonksiyon bloğu başlangıcı
    int toplam=p0operand1+ p0operand2;
    return p0operand1+ p0operand2;
} //Fonksiyon bloğu bitişi
/*Program Çıktısı:
toplama: 5
toplama: 7
toplama: 10
toplama: 15

...Program finished with exit code 0
*/

```

Aşağıda pozitif sayı girilmesini garanti ettikten sonra sayının 10 sayısına bölünebilirliğini bulan program verilmiştir.

```

#include <stdio.h>

int pozitifSayi0ku(); /* parametre almayan bir fonksiyon bildirimi (declaration).
                      Bu fonksiyon geri int değer döndürüyor. */
int onaBolunurMu(int); /* int veri tipinde parametre alan
                        bir başka fonksiyon bildirimi (declaration)
                        bu fonksiyonda int tipinde veri geri döndürüyor. */

int main () {

    int birPozitifSayi=pozitifSayi0ku();
    if (onaBolunurMu(birPozitifSayi))
        printf("Girilen Pozitif Sayı 10 ile bölünebilir.");
    else
        printf("Girilen Pozitif Sayı 10 ile tam bölünmez.");
    return 0;
}
/* Bildirimi yapılan "pozitifSayi0ku" fonksiyonunun tanımı (definition): */
int pozitifSayi0ku()
{
    int okunan;

```

```

do {
    printf("Lütfen pozitif sayı giriniz: ");
    scanf("%d",&okunan);
    if (okunan<=0)
        printf("HATA:Pozitif Sayı GİRMEDİNİZ!\n");
} while (okunan <= 0);
return okunan;
}
/* Bildirimi yapılan "onaBolunurMu" fonksiyonunun tanımı (definition): */
int onaBolunurMu(int p)
{
    return (p%10)==0;
}
/*Program Çıktısı:
Lütfen pozitif sayı giriniz: 30
Girilen Pozitif Sayı 10 ile bölünebilir.

...Program finished with exit code 0
*/

```

Örnek programda görüldüğü üzere yapısal programa kapsamında bir programın iskeleti oluşmuştur. Ana program içinde problemin çözümü, çeşitli fonksiyonların birbirini çağırması ile yapılmıştır. Her bir fonksiyon içinde işlenecek veriye ilişkin değişkenler (en basit veri yapısı) tanımlanmış ve bu veriler ile bu veri yapılarını işleyen kontrol işlemleri birbirinden ayrılmıştır. Okunaklılık çok yüksek bir kod elde edilmiştir.

Bunların dışında hiçbir değer geri döndürmeyen fonksiyonlar da tanımlayabiliriz. Bunlara diğer dillerde **prosedür** (**procedure**) adı da verilir. Bu durumda değeri olmayan veri tipi olan **void** anahtar kelimesi kullanılır. Bu tür fonksiyonların da geri dönüş talimatı yalnızca **return;** şekilde yazılır.

```

#include <stdio.h>

void printMenu(); /* değer geri döndürmeyen bir fonksiyon bildirimi (declaration)
                    aynı zamanda parametre de almıyor. */

int main() {
    int secim;
    do {
        printMenu();
        scanf("%d",&secim);
        switch(secim) {
            case 1:
                printf("Verileri Sakladım.\n");
                break;
            case 2:
                printf("Verileri Okudum.\n");
                break;
            case 0:
                printf("Programdan Çıkılıyor...\n");
                break;
            default:
                printf("Tekrar Seçim Yapınız!\n");
        }
    } while (secim!=0);
    return 0;
}
/* Bildirimi yapılan "printMenu" fonksiyonunun tanımı (definition): */
void printMenu() {
    printf("\n");
    printf("1-Verileri Yaz.\n");
    printf("2-Verileri Oku.\n");
    printf("0-ÇIKIŞ.\n");
    return;
}

```

}

Çağrı Kuralı

Çağrı kuralı (**calling convention**), bir fonksiyon çağrısıyla karşılaşıldığında argümanların fonksiyona hangi sıraya göre iletileceğini belirtir. İki olasılık vardır; Birincisi argümanlar soldan başlanarak sağa doğru fonksiyona geçirilir. İkincisi ise C dilinde kullanılır ve argümanlar sağdan başlanarak sola doğru fonksiyona geçirilir.

```
fonksiyon(a,b,c,d,e);
```

Yukarıdaki örnek incelendiğinde argümanların hangi sırada fonksiyona geçirildiğinin bir önemi yoktur. Ancak aşağıdaki verilen örnekteki durumu inceleyelim;

```
int a = 1;
printf("%d %d %d", a, ++a, ++a);
```

Bu kodun çıktısı **1 2 3** olarak beklenir ancak çağrı kuralından ötürü çıktı **3 3 1** olur. Bunun nedeni C'nin çağrı kuralının sağdan sola olmasıdır. Bunu şöyle açıklayabiliriz;

1. Fonksiyona sağdan ilk argüman olarak a değişkeni değeri 1 geçirilir.
2. Ardından ++ ifadesi ile a değişkeni 2'ye yükseltilir.
3. Sonra ++ ifadesi ile a değişkeni 3'e yükseltilir.
4. Fonksiyona ikinci argüman olarak 3 geçirilir.
5. Fonksiyona son olarak a'nın son değeri olan 3 geçirilir.

Depolama Sınıfları

Depolama sınıfları (**storage classes**) bir değişkenin **ömrünü** (**lifetime**), **görünürlüğünü** (**visibility**), **bellek konumunu** (**memory location**) ve **başlangıç değerini** (**initial value**) belirlemek için kullanılır. Bu özellikler temel olarak bir programın çalışma süresi boyunca belirli bir değişkenin varlığını izlememize yardımcı olan **kapsam** (**scope**), **görünürlüğü** ve **yaşam süresini** içerir.

Yapısal programlama ve fonksiyon kavramının bilgisayarlarda kullanılmasıyla birlikte işlemciler de değişiklikler olmuştur. *TEMEL BİLGİSAYAR KAVRAMLARI* başlığı altında belirtilen işlemci kaydediciler dışında yeni kaydediciler de eklenmiştir;

- Verilerle icra edilen kod farklı bellek bölgesinde bulunur. Bu nedenle verileri işaret eden **veri bellek adreslerini tutan kaydedicisi** (**data memory register**).
- **Yığın bellek kaydedicisi** (**stack register**), fonksiyon çağrıları sırasında mevcut işlemci durumu ve yerel değişkenler gibi bazı işlemleri depolamak için kullanılan belleği gösteren (**stack pointer**) adresi tutan kaydedicidir.
- **Bayrak kaydedicisi** (**flag register**), işlemcinin durumunu veya **sıfır** (**zero**) bayrağı, **taşıma** (**carry**) bayrağı, **işaret** (**sign**) bayrağı, **taşma** (**overflow**) bayrağı, **eşlik** (**parity**) bayrağı, **yardımcı taşıma** (**auxiliary carry**) bayrağı ve **kesme etkinleştirme** (**interrupt enable**) bayrağı gibi çeşitli işlemlerin sonucunu tutmak için kullanılan özel bir kayıt türüdür.
- **Genel amaçla kullanılan kaydediciler** (**general purpose registers**).

Eklenen bu kaydediciler ile derlenen her bir program, dört **bellek bölgesinden** (**segment**) oluşur;

1. **Kod bellek** ya da kaynak koda ithafla **metin bellek** (**code segment / text segment**) icra edilecek emirleri içeren bellek.
2. Programın işleyeceği verileri tutan **veri belleği** (**data segment**).
3. Fonksiyonların çağırmadan (**call**) önce mevcut işlemci durumu ve yerel değişkenler gibi bazı işlemleri depolamak için ve fonksiyondan dönülünce (**return**) işlemciyi ve çağrı ortamını eski hale getirmek için kullanılan **yığın bellek** (**stack segment**). Bu bellek bu nedenle **son giren veri ilk çıkar** (**last in first out-LIFO**) mantığıyla çalışır.
4. Programın icrası sırasında dinamik olarak eklenip çıkarılan veriler için hurdalık gibi kullanılan **öbek bellek** (**heap segment**). İleride *DİNAMİK BELLEK YÖNETİMİ* başlığında anlatılacaktır.

Depolama sınıfları tanımlanan değişkenleri hangi bellek bölgesinde yer alacağını belirler. Dört tür depolama sınıfı vardır; **auto**, **extern**, **static** ve **register**.

Auto Depolama Sınıfı

Otomatik (**auto**) depolama sınıfı, bir fonksiyon veya blok içinde kimliklendirilmiş tüm değişkenler için **varsayılan** (**default**) depolama sınıfıdır. Bu nedenle, C dilinde program yazarken **auto** anahtar sözcüğü nadiren kullanılır. Otomatik değişkenlere;

- Yalnızca blok içinden erişilebilir.
- Değişkenin kapsamı içinde bulunduğu blok ile sınırlıdır.
- **Yığın** (**stack**) bellekte tutulur.

```
#include <stdio.h>
int main() {
    auto int a = 32;
    int b=20;
    /*
    int a=32; ile eşdeğerdir.
    a ve b değişkenleri bu fonksiyon bloğu ve iç bloklarda geçerlidir.
    Yani faaliyet alanları (scope) bu fonksiyon ile sınırlıdır.
    */
    if (a==32){
        int b=10; /* Ana fonksiyon bloğunda tanımlı b değişkenine
                   artık ulaşamaz. */
        int c=50;
        /*
        auto int b; ile eşdeğer
        if bloğunda ve tanımlanabilecek iç bloklarda geçerlidir.
        */
        a=16;// Bu blokta a da geçerlidir.
        printf("a:%d, b:%d c:%d\n",a,b,c); // Çıktı: a:16, b:10 c:50
    }
    /* if bloğu dışında sadece ana fonksiyon bloğunda tanımlı
       değişkenlere ulaşılabilir. */
    printf("a:%d, b:%d\n", a,b); // Çıktı: a:16, b:20
    return 0;
}
```

Static Depolama Sınıfı

Statik (**static**) depolama sınıfı yaygın olarak kullanılan statik değişkenleri kimliklendirmek için kullanılır. Statik değişkenler, **kapsam** (**scope**) dışına çıktıktan sonra bile değerlerini koruma özelliğine sahiptir! Statik değişkenler;

- Kapsamları yani geçerli olduğu yer, tanımlandıkları fonksiyonla ya da blokla sınırlıdır.
- Kapsamlarındaki son kullanımlarının değerini korur.
- Program genelinde geçerli olan **evrensel** (**global**) statik değişkenlere programın herhangi bir yerinden erişilebilir.
- **Veri bellekte** (**data segment**) yer alır. Derleme sırasında bu bellek hep sıfırla doldurulduğundan **ilk değerler** (**initial value**) hep sıfırdır.

Statik (**static**) olarak tanımlanan değişken, program başladığında veri bellekte oluşturulup program bitene kadar aynı bellek bölgesini kullanan değişkendir.

```
#include <stdio.h>
void func() {
    int sayac = 0;
    for (sayac = 1; sayac < 10; sayac++)
    {
        static int statik0lan = 5;
        int statik0lmayan = 10;
        statik0lan++;
    }
}
```

```

        statik0lmayan++;
        printf("sayaç: %d, static0lan: %d\n", sayac, statik0lan);
        printf("sayaç: %d, statik0lmayan: %d\n", sayac, statik0lmayan);
    }
    //Burada statik0lan değişkene erişilemez.
}
int main() {
    func();
    return 0;
}
/*Program Çıktısı:
sayaç: 1, static0lan: 6
sayaç: 1, statik0lmayan: 11
sayaç: 2, static0lan: 7
sayaç: 2, statik0lmayan: 11
sayaç: 3, static0lan: 8
sayaç: 3, statik0lmayan: 11
sayaç: 4, static0lan: 9
sayaç: 4, statik0lmayan: 11

...Program finished with exit code 0
*/

```

Programı çalıştırdığında **statik0lan** değişken yalnızca for bloğu içinde geçerli olduğu ve her kullanımda değerini koruduğu görülmektedir.

Harici Depolama Sınıfı

Harici (**extern**) depolama sınıfı bize basitçe değişkenin kullanıldığı blokta değil başka bir yerde tanımlandığını söyler. Harici değişkenler;

- Bu değişkenlere değerler tanımlandığı yerde değil farklı bir blokta atanır ve üzerinde değişiklik yapılabilir.
- Bir global değişken, **extern** anahtar sözcüğünü kimliklendirme önüne yerleştirerek harici de yapılabilir.
- Bu değişkenler de **veri bellekte** (**data segment**) yer alır. Derleme sırasında bu bellek hep sıfırla doldurulduğundan ilk değerler hep sıfırdır.

Bir **harici** (**extern**) değişkene **bildirim** (**declaration**) yapıldığında bir başka dosyada **tanımlanmış** (**definition**) yapıldığı kabul edilir. Bunun için iki farklı kaynak kod dosyası gereklidir. Birinci kaynak kod dosyası (**xtrn.c**) aşağıda verilmiştir.

```

//XTRN.C
int externDegisken; //Diğer dosyalarda kullanılacak değişken tanımlandı.
void funcExtern() //Diğer dosyalarda kullanılacak fonksiyon tanımlandı.
{
    externDegisken++;
}

```

Bu dosyadaki değişken ve fonksiyonları kullanan bir başka kaynak kod (**main.c**) aşağıda verilmiştir;

```

#include <stdio.h>
extern void funcExtern(); // bir başka dosyada tanımlandı. Burada bildirimi yapıldı.
int main()
{
    extern int externDegisken; // bir başka dosyada tanımlandı. Burada bildirimi yapıldı.
    funcExtern();
    printf("extern: %d\n", externDegisken);
    externDegisken = 2;
    printf("extern: %d\n", externDegisken);
    return 0;
}

```

Bu iki dosyayı birlikte derlemek için `gcc xtrc.c main.c -o main.exe` komutu ile derleyiciye iki farklı dosya parametre olarak verilir.

Kaydedici Depolama Sınıfı

Kaydedici (**register**) depolama, otomatik değişkenlerle aynı işlevselliğe sahiptir. Tek fark, derleyicinin, eğer boş bir işlemci kaydedicisi mevcutsa değişken olarak o kaydedicinin kullanılmasıdır. Genel amaçlı kaydediciler bu amaçla kullanılır. Bu da bu değişkenlerin, bellekte saklanan değişkenlerden çok daha hızlı olmasını sağlar.

Eğer boş bir CPU kaydedicisi mevcut değilse, değişken yalnızca bellekte saklanır. **register** anahtar sözcüğüyle tanımlanan değişken nitelendirilir. **register** Anahtar sözcüğü ile sadece bloklar içinde tanımlanan **yeral** (**local**) değişkenler nitelendirilebilir, **evrensel** (**global**) değişkenler bu anahtar sözcük ile kullanılamaz.

Aşağıda **kaydedici** (**register**) depolama sınıfının ilişkin örnek verilmiştir. Kuradaki sayma ve toplama işlemleri uygun olan işlemci kaydedicilerinde yapılır.

```
#include <stdio.h>
int main() {
    register int k, sum;
    for(k = 1, sum = 0; k < 1000; sum += k, k++);
    //döngü bloğu olmayan for talimatı örneği sonunda ; var!
    printf("%d\n",sum); //Çıktı:499500
}
```

Öncelikle C standardı tek bir değişkenle birden fazla depolama sınıfının kullanılmasını yasaklar. Aynı değişkene hem **static** hem **extern** niteleyicisi kullanılamaz. Depolama sınıfları aşağıdaki tabloda özetlenmiştir; Tablodaki **çöp** (**garbage**) kavramı tahsis edilen bellek içeriği o anda ne ise değişkenin o değeri alması anlamındadır.

Depolama sınıfı	Yer aldığı bellek bölgesi	Başlangıç Değeri	Kapsamı (scope)	Ömür (lifetime)
auto	Yığın Bellek (Stack Segment)	Çöp	Bulunduğu blok ve bu blok içindeki bloklar	Bulunduğu blok sonunda biter
extern	Veri Bellek (Data Segment)	Sıfır	Evrensel (tüm dosyalarda)	Program sonunda biter
static	Veri Bellek (Data Segment)	Sıfır	İçinde bulunduğu blok	Program sonunda biter
register	İşlemci kaydedicileri (CPU Registers)	Çöp	Bulunduğu blok ve bu blok içindeki bloklar.	Bulunduğu blok sonunda biter

Tablo 22. Depolama Sınıfları Özet Tablosu

Evrensel ve Yerel Değişken

C dilinde bir kaynak koda ait dosya içinde bir değişkene her yerden erişilmek istenirse **evrensel** (**global**) değişken olarak tanımlanır. Bu değişkenler herhangi bir fonksiyon bloğu içinde tanımlanmazlar. Tanımlandıkları yerden sonra kaynak kodun her yerinde kullanılabilirler. Bu nedenle genellikle dosyanın başında tanımlama yapılır.

```
#include <stdio.h>

int evrenselDegisken=10; /* evrenselDeğişken bu noktadan sonra
                           her yerde kullanılabilir.*/

void fonksiyon();
int main() {
    int yerelDegisken=20; /* Bu yerel değişken sadece main bloğu
                           içinde kullanılabilir. */

    printf("evrensel:%d, yerel:%d\n",evrenselDegisken,yerelDegisken);
}
```

```

    evrenselDegisken++; // evrensel değişken değiştiriliyor.
    yerelDegisken--; // yerel değişken değiştiriliyor.
    printf("evrensel:%d, yerel:%d\n",evrenselDegisken,yerelDegisken);
    fonksiyon();

    return 0;
}
void fonksiyon() {
    int fonksiyonYerelDegiskeni=30;
    evrenselDegisken++;
    printf("evrensel:%d, fonsiyonYerel:%d\n",evrenselDegisken,fonksiyonYerelDegiskeni);
    if (fonksiyonYerelDegiskeni==30) {
        int evrenselDegisken=100; /* Artık evrensel değişkene bu blokta
                                   ulaşamaz. */
        printf("evrensel:%d, fonsiyonYerel:%d\n",
               evrenselDegisken,fonksiyonYerelDegiskeni);
    }
}
/*Program Çıktısı:
evrensel:10, yerel:20
evrensel:11, yerel:19
evrensel:12, fonsiyonYerel:30
evrensel:100, fonsiyonYerel:30

...Program finished with exit code 0
*/

```

Burada **fonksiyon** içindeki if bloğunda yer alan değişken her ne kadar **evrenselDeğişken** olarak kimliklendirilse de yerel değişkendir ve sadece bu if bloğu içinde geçerlidir. Yani tanımlamada **mantıksal hata** (logical error) yapılmıştır.

Fonksiyon Çağırma Süreci

Bir fonksiyon çağrıldığında nelerin yapıldığı aşağıdaki programdan anlaşılabilir;

```

#include <stdio.h>

int topla(int, int); /* İki tamsayıyı toplayan fonksiyon bildirimi/prototipi */

int main () {
    /* main() fonksiyonu içinde geçerli yerel (local) değişkenler*/
    int a = 10;
    int b = 20;
    int toplam = 0;
    printf("çağrı öncesi a: %d, b:%d, toplam:%d\n",a,b,toplam);
    toplam = topla(a, b);
    printf("çağrı sonrası a: %d, b:%d, toplam:%d\n",a,b,toplam);
}

int topla(int x, int y) { /* iki tamsayıyı toplayan fonksiyon tanımı*/
    /* Topla fonksiyonu yerel değişkenler */
    int temp= x+y;
    x=100; y=200;
    /*
        Burada değiştirilenler parametre değişkenleridir.
        main() içindeki yerel değişkenler değildir.
    */
    printf("çağrı içinde x: %d, y:%d\n",x,y);
    return temp;
}

/* Program çalıştırıldığında:

```

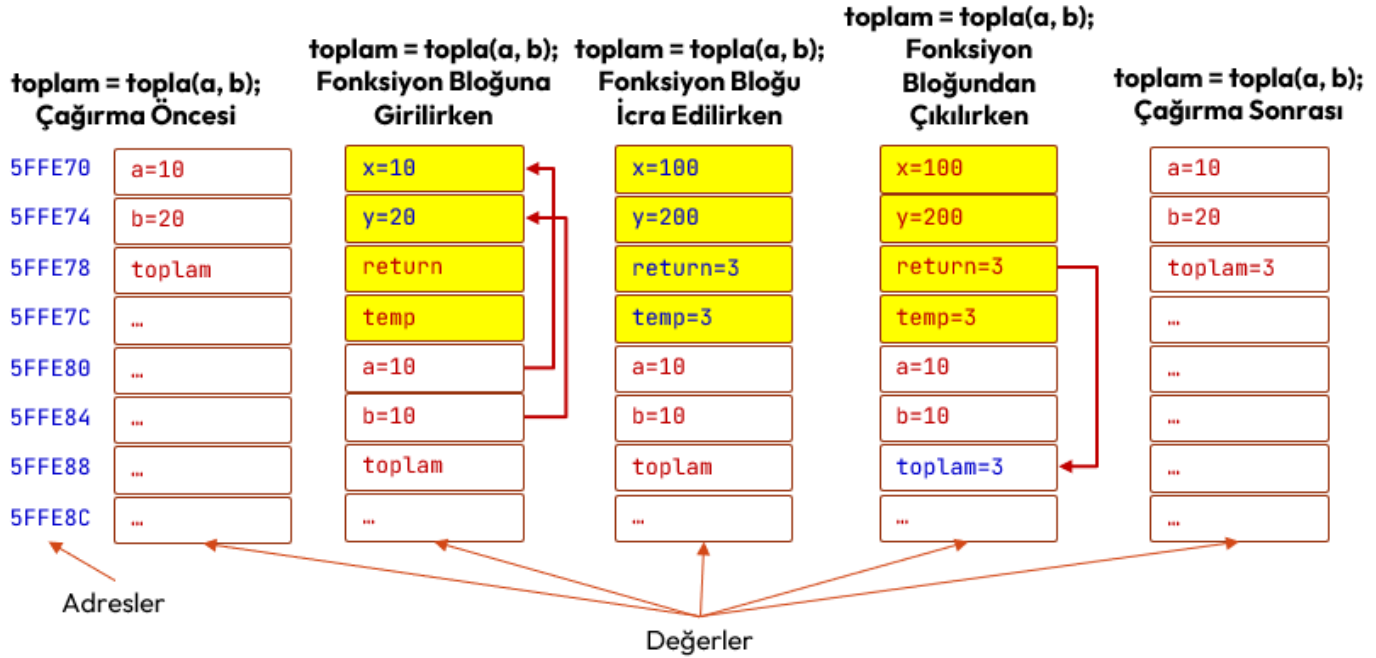
```

çağrı öncesi a:10, b:20 toplam:0
çağrı içinde x:100, y:200
çağrı sonrası a:10, b:20 toplam:30

...Program finished with exit code 0
*/

```

Aşağıda örneği verilen programda fonksiyon çağırma sürecindeki **yığın (stack)** bellek değişimi gösterilmektedir.



Tablo 23. Fonksiyon çağırma sürecinde yığın bellek

Yukarıdaki kod incelendiğinde;

- İcra sırası `topla(a, b)` fonksiyonuna geldiğinde fonksiyon çağrılmadan önce main fonksiyonu yerel değişkenleri `a`, `b` ve `toplam` değişkenleri yığın (stack) belleğe itilir (push).
- `topla(a, b)` fonksiyonu çağrıldığı anda fonksiyon bloğuna başlamadan parametrelere sırasıyla `a` ve `b` değişkenlerinin değerleri kopyalanarak fonksiyona geçirilir.
- `topla(a, b)` fonksiyonu icra edilirken fonksiyon yerelindeki `temp` ve parametre değişkenleri `x`, ve `y` istenildiği gibi değiştirilebilir. Geri dönüş değeri de belirlenir.
- `toplam=topla(a, b)` fonksiyon bloğundan çıkılırken geri dönüş değeri çağrı ortamındaki `toplam` değişkenine atanır ve fonksiyon için yığın (stack) belleğe itilen değerler geri çekilir.

Özyinelemeli Fonksiyonlar

Özyineleme (recursion), bir fonksiyonun kendi çağrısını yapma tekniğidir. Bu teknik, karmaşık sorunları çözülmesi daha kolay basit sorunlara ayırmanın bir yolunu sağlar. Özyineleme, **yineleme (iteration)**, **döngüler (loop)** veya tekrar yerine geçebilecek çok güçlü bir tekniktir.

Özyinelemeli (recursive) algoritmalarda, tekrarlar fonksiyonun kendi kendisini kopyalayarak çağırması ile elde edilir. Bu kopyalar işlerini bitirdikçe kaybolur. Bir problemi özyineleme ile çözmek için problem iki ana parçaya ayrılır.

1. Cevabı kesin olarak bildiğimiz **temel durum (base case)**
2. Cevabı bilinmeyen ancak cevabı yine problemin kendisi kullanılarak bulunabilecek durum.

Faktöriyel örneğini inceleyelim; Matematikte, negatif olmayan bir tam sayı olan n 'nin faktöriyeli, $n!$ ile gösterilir ve n 'den küçük veya ona eşit olan tüm pozitif tam sayıların çarpımıdır.

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 3 \cdot 2 \cdot 1$$

Negatif olmayan bir tam sayı olan n 'nin faktöriyeli aynı zamanda n 'nin bir sonraki daha küçük faktöriyelle çarpımına eşittir. Yani problemin tanımı yine kendisini içerir:

$$n! = n \cdot (n - 1)!$$

Örneğin;

$$5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

Aşağıda faktöriyel için girilen sayıdan sonra tüm sayıların çarpımıyla hesaplayan bir fonksiyon yazılmıştır.

```
#include <stdio.h>

unsigned int faktoriyel(unsigned int n)
{
    int sonuc=1,indis;
    for (indis=n;indis>0;indis--)
        sonuc=sonuc*indis;
    return sonuc;
}

int main() {
    unsigned int sayi,sonuc; //Pozitif tamsayı yanımı yapılmış değişkenler.
    printf("Pozitif Bir Sayı Giriniz:");
    scanf("%d",&sayi);
    sonuc=faktoriyel(sayi);
    printf("%d!=%d\n",sayi,sonuc);
    return 0;
}
```

Sıfır sayısı da pozitif bir sayıdır ama sonucum sıfır çıkmaması için 0 sayısının faktöriyeli 1 kabul edilir. Bu aslında faktöriyel problemi için **temel durumdur** (base case) ve **boş ürün** (empty product) kuralı olarak bilinir. Kısaca faktöriyel fonksiyonu aşağıdaki şekilde gösterilir;

$$n \in \mathbb{N} \text{ olmak üzere } f(n) = n! = \begin{cases} 1, & n = 0 \\ n \cdot f(n-1), & n > 0 \end{cases}$$

Özyinelemeli (recursive) olarak ise bu fonksiyon aşağıdaki şekilde kodlanabilir;

```
unsigned int faktoriyel(unsigned int n)
{
    if (n==0) return 1;
    else return n*faktoriyel(n-1);
}
```

Girilen taban ve üs ile kuvvet hesaplama fonksiyonu özyinelemeli olarak aşağıda verilmiştir;

```
#include <stdio.h>

int kuvvet(int taban, int us) {
    if (taban == 0) { return 0; } //Taban 0 ise 0 dön
    if (us == 0) { return 1; } // Üs 0 ise 1 dön
    return taban*kuvvet(taban, us - 1);
}

int main(){
    int taban,us;
    printf("Tabanı Giriniz:");
    scanf("%d",&taban);
    printf("Üssü Giriniz:");
    scanf("%d",&us);
    printf("%d üzeri %d = %d", taban,us, kuvvet(taban,us) );
    return 0;
}
```

```

/*
Tabanı Giriniz:3
Üssü Giriniz:6
3 üzeri 6 = 729

...Program finished with exit code 0
*/

```

Satır İçi Fonksiyonlar

Fonksiyon Çağırma Süreci başlığında anlatıldığı üzere fonksiyonlar çağrılırken sürekli **yığın** (*stack*) belleğe it-çek yapılır. Bazı durumlarda bu işlem performans gereği istenmez. Bu durumda fonksiyonlar, **satır içi fonksiyon** (*inline function*) olarak tanımlanır. 2017 C sürümü ile gelen bu özellik, fonksiyona **inline sıfatı** (*inline specifier*) kullanılır.

```

#include <stdio.h>
inline int topla (int op1, int op2) {
    int toplam= op1+op2;
    return toplam;
}
int main() {
    int x=10, y=15, sonuc;
    sonuc = topla(x,y);
    printf("%d+%d=%d",x,y,sonuc);
    return 0;
}

```

Bu kod aşağıdaki şekle çevrilmiş gibi derlenir;

```

#include <stdio.h>
int main() {
    int x=10, y=15, sonuc;
    {
        int toplam= op1+op2;
        sonuc = toplam;
    }
    printf("%d+%d=%d",x,y,sonuc);
    return 0;
}

```

Bir fonksiyon aşağıda belirtilen durumlarda satır içi fonksiyon olarak derlenmez;

- Bir döngü içeriyorsa!
- Statik değişkenler içeriyorsa!
- Özyinelemeli isel!
- Dönüş türü **void** haricinde olup return ifadesi işlev gövdesinde mevcut değilse!
- Switch veya **goto** talimatı içeriyorsa!

Bu tip fonksiyon tanımlamadaki amacımız, fonksiyon çağırma yükünün azaltılması için işlemci kaydedicilerinin daha çok kullanılmasıdır. Bu da icra hızını yükseltir.