1. Programlamaya Giriş

Amaç :

1. Bilgisayar ve bilgisayar yazılımlarının tanımlanması
2. Programlama dilleri ve program geliştirme tekniklerinin tanıtılması
3. Temel problem çözme tekniklerinden olan olan yukarıdan aşağı tasarım (top-down design) tekniğinin tanıtılması ve bu teknikle algoritmaların yazılması.
4. Kullanacağımız C derleyici ortamında, C pogramlarının nasıl hazırlanacağının gösterilmesi.

1.1. Bilgisayar Nedir?

Bilgisayar, mantıksal ve aritmetiksel işlemleri çok hızlı biçimde yapan bir araçtır. Günümüzün bilgisayarları, saniyede milyonlarca aritmetiksel işlemi yapabilecek kapasitededir.

İlk üretilen bilgisayarlar, oldukça büyük ve yavaştı. Ama gün geçtikçe küçüldüler ve güçleri de hızla arttı. Örneğin, yirmi-otuz yıl önceki ancak bir odaya sığabilecek bir bilgisayarın yaptığı işlerden çok daha fazlasını, bugünkü kişisel bilgisayarlar, daha hızlı bir şekilde yapabilmektedir. Bu yüzden, bilgisayarlar günümüzde çok değişik alanlarda ve ortamlarda kullanılmaktadır. Hemen hemen her evde ve işyerinde artık bir bilgisayar görmek olasıdır.

Bir bilgisayar sisteminin fiziksel yapısını oluşturan, klavye, ekran, işlemci, disk, bellek gibi parçaların hepsi birden bilgisayar donanımı olarak adlandırılmaktadır. Bilgisayarın çalışmasını denetleyen birer komutlar kümesi olan programlar da bilgisayar yazılımı olarak adlandırılır. Bir bilgisayar sisteminin, bu iki parçaya da ihtiyacı vardır. Her biri, diğeri olmadan hiçbir işe yaramaz. Günümüzde, donanım fiyatları hızla düşmekte, ama yazılım maliyetleri tam tersine artmaktadır. Bu dersteki amacımız bilgisayar yazılımı geliştirmek olduğundan, biz yalnızca bilgisayar yazılımı ile ilgileneceğiz.

1.2. Bilgisayar Yazılımı

Bilgisayar yazılımı (program), bilgisayara bir işi yaptırmak için verdiğimiz komutlar kümesidir. Yazılımları genel olarak iki kategoride sınıflandırabiliriz :
İşletim sistemleri :
Kullanıcı ile bilgisayar arasındaki iletişimi sağlayan programlardır. Bilgisayar sisteminin bütün hareketlerini denetler ve sistemdeki MİB ve ana bellek gibi kaynakların yönetimi ile ilgilenirler. Örneğin, sizin kullandığınız Windows'95 ya da Windows NT birer işletim sistemidir. Diğer bazı işletim sistemleri şunlardır : UNIX, DOS, Solaris, MacOS.
Uygulama Programları :
İşletim sistemi dışında kalan diğer programlara verilen genel bir addır. Örneğin, MSWord, havayolları rezervasyon sistemi, bilgisayar oyunları, programlama dillerinin derleyicileri, bizim yazacağımız C programları birer uygulama programıdır.

1.3. Programlama Dilleri

Bir problemi çözmek için gerekli olan komutlar çok değişik programlama dilleri kullanılarak yazılabilir. Bazı programlama dillerinde yazılan programlar, doğrudan bilgisayarın merkezi işlem birimi (MİB) tarafından anlaşılabilir. Diğer bazı dillerde yazılan programlar ise o dilin derleyicisi tarafından merkezi işlem biriminin anlayacağı dile çevrilir.

Programlama dillerini üç genel kategoriye ayırabiliriz :

- Makina dilleri
- Assembly dilleri
- Yüksek seviyeli programlama dilleri (high-level languages)

Makina dili :
Her bilgisayar MİB'inin anlayacağı kendi makina dili vardır. Örneğin, Pentium işlemcinin makina dili, Sparc işlemciden farklıdır. Makina dilindeki bir komut, bit (0 ya da 1) gruplarından oluşur. Örneğin, makina dilinin bir komutu iki değerin toplanması ya da bir değerin ana belleğin bir bölgesinde saklanması gibi basit işlemleri içerir. İnsanların makina dilinde program yazması hemen hemen olanaksızdır. Örneğin, bir makina dilinde iki sayının toplanması,

010001100111010001111010100000100101011010000 10

biçiminde ifade edilebilir.

Assembly dili :
Bir assembly dili, makina dilindeki komutları İngilizce benzeri bir dille ifade eder. Genelde assembly dilindeki her komut makina dilindeki bir komuta denk düşer. Assembly dilindeki bir program *assembler* diye bilinen bir yazılım tarafından makina diline çevrilir. Assembly dilinde program yazmak da makina dilinde olduğu gibi zor ve zaman alıcıdır. Assembly dilinde iki sayının toplanması,

```
LOAD A
ADD B
STORE C
```

biçiminde ifade edilir.

Yüksek Seviyeli Programlama Dilleri (YSPD) :
Makina dilinde çok sayıda komutla yapılabilecek bir iş, yüksek seviyeli programlama dilinin bir komutu ile yapılabilir. Bu yüksek seviyeli diller, İngilizce ya da Türkçe gibi doğal bir dil değil, kendi yapısı ve dilbilgisi kuralları olan, insanlar için anlaşılması kolay olan dillerdir. Bu derste öğreneceğimiz C, bir yüksek seviyeli programlama dilidir.

Yüksek seviyeli programlama dilinde yazılan bir program, derleyici (compiler) dediğimiz yazılımlar tarafından, makina dilindeki denk bir programa dönüştürülür. Bilgisayar tarafından çalıştırılacak olan program, makina dilindeki programdır. Her programlama dili için ayrı bir derleyici vardır. C'de yazacağınız programlar C derleyicisi ile makina diline çevirilecek ve bu makina dilindeki program çalıştırılacaktır.

C gibi diğer bazı yüksek seviyeli programlama dilleri arasında Pascal, Fortran, Basic, Java, C++, Cobol, Lisp ve Prolog sayılabilir. Bu dillerin birkaçında, iki sayının toplanması şöyle ifade edilir:

```
C:=A+B;   (Pascal)
C=A+B;    (C ve Java)
C=A+B     (Fortran)
```

Bazen yüksek seviyeli programlama dilinde yazılmış bir program, makina diline çevrilmek yerine başka bir ara dile çevrilebilir. Bu ara dildeki komutlar, hiçbir makinaya bağımlı değildir. Bu ara dildeki programlar, o dilin yorumlayıcısı (interpreter) olarak bilinen yazılımlar tarafından çalıştırılır. Bunun amacı, ara dildeki programların değişik bilgisayarlarda hiçbir değişikliğe uğramadan kullanılabilmesidir. Örneğin Java derleyicisi

Java programlarını böyle ara bir dile çevirir ve bu dildeki programlar bir Java yorumlayıcısı tarafından çalıştırılır.

1.4. Yazılım Geliştirme

Bir program, bir problemi çözmek için yazılan komutlar kümesidir. Bu komutlar, İngilizce ya da Türkçe gibi dillerde değil, bilgisayar sisteminin anlayacağı bir dildedir. Bu derste, biz, komutlarımızı C dilini kullanarak yazacağız.

Bir program yazmadan önce, ilk yapacağımız iş, o problemi nasıl çözeceğimize karar vermek ve çözümün adımlarını belirlemek olmalıdır. Bu işte bize yardımcı olabilecek çeşitli problem çözme teknikleri vardır. Bu tekniklerden yararlandığımızda, yarattığımız çözümü bir programa dönüştürmek kolay olacaktır. En temel problem çözme tekniği, yukarıdan-aşağı tasarım yöntemidir. Bu yöntemde ilk verilen problem, küçük alt problemlere bölünür. Bu alt problemler, gerekirse, daha küçük (basit) alt problemlere bölünürler. En sonunda elde edilen alt problemler, ayrı ayrı çözülürler ve bu çözümler ana problemin çözümünü elde etmek için birleştirilirler. Bu yöntem, algoritma yazımından söz ederken örneklerle açıklanacaktır.

Bir problemi çözmek için yazacağımız programı oluştururken, genelde, aşağıdaki yazılım geliştirme aşamalarından geçmek zorundayız:

1.4.1. Problemin Anlaşılması ve Çözümlenmesi :

İlk yapacağımız iş, problemin tam olarak ne olduğunu anlamaktır. Bu nedenle, problemin çözümünden neler beklediğimizi ve yaratacağımız çözümün girdi ve çıktılarının neler olacağını kesin olarak belirlemeliyiz.

1.4.2. Tasarım :

Tasarım aşaması programı oluşturmadaki en önemli adımlardan biridir ve çok kolay değildir. Tasarım için ne kadar çok vakit harcarsak, programı yazma ve çalıştırma aşamalarında o kadar az sorun yaşarız ve programı o kadar çabuk çalıştırırız. Bu aşamada, problemi çözmek için kullanılacak çözüm adımlarını iyice düşünmemiz ve bu çözüm adımlarını gösteren bir liste yapmamız gerekir. Bir problemin çözüm adımlarını gösteren bu listeye *algoritma* denir. Böyle bir liste tasarlamak için geliştirilmiş pek çok yöntem vardır. Bunlardan biri de yukarıdan-aşağı tasarım tekniğidir. Bu teknikle önce problemin ana adımları çıkarılır; daha sonra, her adım için, gerekiyorsa, daha detaylı biz çözüm tasarlanır. Böylece verilen büyük problem daha küçük ve çözümü daha kolay küçük problemlere bölünmüş olur. Bu çözümlerin ardarda sıralanmasıyla problem için bir algoritma geliştirmiş oluruz.

Oluşturulan algoritma *pseudocode* denilen, Türkçe ve İngilizce gibi doğal ama sınırlı yapısı olan bir dilde yazılabilir. Bir programlama diline çevirmeden önce, bu algoritmanın doğru bir çözüm olduğu kanısında olmalıyız. Bu aşamayı atlayarak doğrudan programlamaya geçmek, büyük programlar yazarken işimizi çok zorlaştıracaktır.

1.4.3. Kodlama :

Bu aşamada, kağıt üzerinde geliştirdiğimiz algoritmayı bir programlama diline çeviririz ve programı bilgisayara gireriz. Biz C dilini bu aşamada kullanacağız. Bu yüzden, C programlama dilinin yapısını bilmemiz gerekecektir. Algoritmamızın her adımı, ona karşılık gelen C program parçasına çevrilecektir. Oluşturulan program büyük ihtimalle ilk denemede çalışmayacak, çeşitli hatalar verecektir. Kodlama aşamasında bu hataları bulmak ve düzeltmek gerekir.

1.4.4. Sınama ve Kanıtlama (Testing and Verifying) :
Geliştirdiğimiz programın doğruluğunu sınayarak istediğimiz gibi çalışıp çalışmadığını görmek zorundayız. Bu amaçla, program üzerinde çeşitli testler yaparak, programın o testlere nasıl karşılık verdiğine bakarız. Bunu sağlamak için program değişik girdilerle çalıştırılır ve ürettiği sonuçlara bakılır. Sonuçlar beklendiği gibiyse, programın doğru çalıştığı kanıtlanmış olur; değilse doğru çalışmayan parçaları bularak düzeltmemiz gerekir. Bu bazen tasarımda düzeltmeler yapmayı bile gerektirebilir.

1.5. Basit bir problem

Basit bir problemle program tasarlama işlemini anlatmaya çalışalım. Bir grup öğrenciye anket yapacağımızı ve onlara ülkenin kuzeyinden mi, güneyinden mi, doğusundan mı yoksa batısından mı geldiklerini soracağımızı varsayalım. Anketin sonuçlarına göre her yönden gelen öğrencilerin sayılarını ayrı ayrı bulalım.

Problemin Anlaşılması ve Çözümlenmesi:

Kuzey, güney, doğu, batı olmak üzere dört yön var. Her yön için ayrı sayaç tutarsak, her öğrencinin cevabına göre doğru sayacın değerini bir arttırırız. Öğrenciler bitince bu dört sayacın değerini bastırırız.
Girdiler: öğrencilerin cevapları
Çıktılar: 4 tane sayılan değer

Tasarım:
Bu problemi yukarıdan-aşağı tasarım yöntemini kullanarak çözeceğiz. Çözüm adımlarını bir algoritma gibi göstereceğiz. İlk önce problemimizi aşağıdaki üç ana adıma ayıralım.

1. Anket için hazırlık yap.
2. Anketi başlat: Öğrencileri sıraya diz ve nereden geldiklerini tek tek sor.
3. Sonuçları bastır.

Bu üç adımı biraz daha ayrıntılı vermemiz gerekir. Bunu da aşağıdaki şekilde yapabiliriz.

1. Anket için hazırlık yap.

1.1. Bir kağıda dört kutu çiz ve bunları K, G, D, ve B olarak adlandır.
1.2. Her kutunun içine ilk değer olarak sıfır yaz.

2. Anketi başlat.
2.1. Öğrencilere sıraya girmelerini söyle.
2.2. Sırada öğrenci olduğu sürece aşağıdaki işlemleri yap.
2.2.1. Sıradaki öğrenciye nereden geldiğini sor.
2.2.2. Doğru kutudaki değere bir ekle.
3. Her kutunun içindeki değeri bastır.

Bu örnekte görüldüğü gibi problemin çözümü için bazı adımların birkaç kez yinelenmesi gerekebilir. (2.2.1. ve 2.2.2. adımları sırada öğrenci olduğu sürece tekrar tekrar yapılmalıdır. 3. adıma ancak sırada başka öğrenci kalmamışsa geçilebilir. Bunun ve diğer işlemlerin C'da nasıl yapılacağını ilerideki derslerimizde göreceğiz)

Kodlama ve ***Sınama ve kanıtlama:***
Bu son iki aşamayı C'yi öğrendikten sonra siz yapabilirsiniz.

Açıkladığımız yazılım geliştirme aşamalarının ilk iki adımı birkaç örnekle daha incelemek isterseniz *örnekler*'i tıklayınız.

1.6. C Programlarının Hazırlanması

Algoritmayı tasarladıktan sonra onu bir programlama dili kullanarak bir program haline getirmek (kodlamak) gerekir. Biz bu amaçla C öğreneceğiz. C programı yazabilmek için önce onun 'gramer'ini öğrenmek gerekir. Bir programlama dilinin gramerine sözdizim (syntax) denir. Bunu ileriki derslerimizde detaylı olarak göreceğiz. Şimdi burada bir C programını yazarken ve çalıştırırken neler yapmamız gerektiğini özetleyeceğiz.

Herhangi bir C programını hazırlamak ve çalıştırmak için aşağıdaki aşamalardan geçmemiz gerekecektir:

- Kurgulama (Editing)
- Derleme (Compiling)
- Çalıştırma (Executing)

Bu aşamaları bir işletim sistemi ortamında (DOS ya da UNIX gibi) ya da kullandığımız dilin sağladığı tümleşik geliştirme ortamında (integrated development environment-IDE) yapabiliriz. Biz bu derste Borland Turbo C++ tümleşik geliştirme ortamını kullanacağız. Bu ortamı sitemizden burayı tıklayarak download edebilirsiniz. Bilgisayarınızda başka C derleyicisi olsa bile dersimizi takip ederken bu derleme ortamını kullanmanızı tavsiye ediyoruz. Çünkü ders notlarındaki programlar yalnızca bu ortamda test edilmiştir. Farklı ortamlarda çalışmayabilirler.

Şimdi bu program hazırlama aşamalarını kısaca açıklayalım.

1.6.1. Kurgulama:

Yazacağımız C programını bir editör yardımı ile bir kütüğün içine koymalıyız. Kullanacağımız editör vi, emacs, MS editör ya da notepad gibi bir kütüğü ASCII text olarak yaratabilecek olan bir editör olabilir (MS-Word, WordPerfect gibi yazılımlar *kullanılmamalıdır*). Ya da editörü ile beraber gelen bir paket derleyici ortamında yaratılabilir. Borland Turbo C++ ortamındaki editör yardımıyla yaratılan kütüğün adı **.cpp** eki ile biter (bu isimlendirme sisteme bağlıdır; bazı sistemler .c eki koyabilir). Bu ek o kütüğün bir C programı sakladığını gösterir.
Örneğin, **`Test1.cpp`** ve **`Test2.cpp`** birer C programı saklayan kütüklerin adları olabilir.

1.6.2. Derleme:

Bir editör yardımı ile bir C programını saklayan `Test1.cpp` kütüğünü yaratmış olduğumuzu varsayalım. İkinci adım, bu kütüğü C derleyicisiyle derlemektir. Bir başka deyişle, yüksek seviyeli dille yazılan programımızı, makina diline çevirmemiz gerekmektedir. Eğer yazdığımız programda hiçbir yazım hatası yoksa, C derleyicisi makina dilinde bir program üretecektir. Bu programı saklayan kütüğün adı `Test1.obj` olabilir (bu isimlendirme de sisteme bağlı). Bu kütüğün içindekileri ekranda göremez, bastıramaz ve bir editörde değiştiremezsiniz. Çünkü bu kütüğün formatı farklıdır. Eğer programımızda yazım ya da gramer hataları varsa, `Test1.obj` kütüğü üretilmeyecek ve C derleyicisi programımızdaki hataları, hata mesajları ile bize bildirecektir. Bu durumda programımızdaki hataları bulup hataya neden olan yerleri yine editör yardımıyla düzeltmeliyiz. Düzeltmeleri yaptıktan sonra, programımızı yeniden derlemeliyiz. Bu işe programımızdaki bütün yazım hataları temizlenene kadar devam

etmeliyiz. Hatasız programlar için yaratılan `Test1.obj` kütüğü yazdığımız C programının makina dili türünden gösterimidir. Bu dilin komutları makinaya bağlıdır ve yalnızca çalıştığınızın aynısı olan ortamlarda ve makinalarda çalıştırılabilir.

C ve benzeri yüksek seviyeli programlama dilleri programcıların işine yarayacak birçok hazır program parçacıkları sunarlar. Bunlar program yazarken bazı işlemleri yapmada büyük kolaylık sağlarlar. Bağlayıcı (linker) dediğimiz bir program derleyicinin yarattığı makina dilindeki programı bu hazır program parçacıklarından gerekli olanları ile birleştirir ve çalıştırılmaya hazır bir bütün program oluşturur.

1.6.3. Çalıştırma:

Çalıştıracağımız program, yaratılan `Test1.obj` kütüğündeki program olacaktır. Bunu çalıştımak için yükleyici (loader) dediğimiz yazılım programı belleğe yükler ve MİB'den programı başlatmasını ister. Bazı sistemler bu işlemin işletim sistemi ortamında bir komutla yapılmasına izin verirler. Tümleşik geliştirme ortamında ise editör, derleyici, bağlayıcı ve yükleyici hep birlikte bir paket olarak sunulur. Bu tip ortamlarda programcı programını editörde yazdıktan sonra menüler yardımı ile derleyip çalıştırabilir.

Biz bu derste kullanacağımız Borland Turbo C++ tümleşik geliştirme ortamınında da kurgulama, derleme ve çalıştırma aşamalarını menüler yardımıyla yapacağız. Gelecek haftadan itibaren C programlarının yapısını incelemeye ve basit programlar yazmaya başlayacağız. Ders notlarını anlamanız için bu hafta bu yazılımı mutlaka bilgisayarınıza yüklemiş ve sitede verilen örnek programları çalıştırmış olmanız gerekiyor. Borland Turbo C++'yi yüklemek ve kullanmak için yapmanız gerekenleri burayı tıklayarak öğrenebilirsiniz.

2. C Programlama Diline Giriş

Amaç:

- Basit C program örnekleri ile C programlarının yapısını tanıtmak.
- Veri türlerini tanıtmak ve değişkenlerin bellekte nasıl gösterildiği anlatmak.
- Atama ifadesinin kullanımını göstermek.
- C programlarında nasıl basit girdi/çıktı yapılacağını öğretmek.
- Derleme hataları, çalışma hataları ve mantık hatalarının nedenlerini anlatmak.

Bu haftaki dersimize başlamadan önce C derleyicisini kendi bilgisayarınıza yüklemiş, ve sitede verdiğimiz örnek programları çalıştırmış olmanız gerekiyor. Hatırlayacağınız gibi, bir C programını çalıştırmak için onu önce *derlemek* gerekiyor. C programlarının hazırlanması ile ilgili notları tekrar gözden geçirmek isterseniz burayı tıklayınız.

Bu derste C programlarının genel yapısını, temel yapı taşlarını ve bazı çalıştırılabilir ifadelerini örneklerle inceleyeceğiz.

2.1. C Programlarının Yapısı

Bu bölümde bir C programının yapısını inceleyeceğiz. Şekil 2.1'de verilen C programı derleyicinin editöründe yazılıp, `ornek.cpp` kütüğünün içinde saklandıktan sonra, menüden`Compile` seçilerek derlenir ve bu derleme sonucunda `ornek.obj` kütüğü yaratılır. Bu kütükte saklanan makina dilindeki program, menüden `Run` seçilerek çalıştırılır.

```
  1 /* Yazar: Nihan Kesim Cicekli
  2 * Tarih: Temmuz 2000
  3 *
  4 * Basit bir C programi
  5 *
  6 * Bu program adinizi okur ve adinizi Merhaba kelimesi ile
birlikte basar.
  7 */
  8
  9 #include <stdio.h>
 10 #define UZUNLUK 10
 11
 12 int
 13 main(void)
 14 {
 15  char yourName[UZUNLUK];
 16
 17   /* Adi oku */
 18   printf("Adinizi girin ve enter tusuna basin> ");
 19   scanf("%s", yourName);
 20   /* Mesaji yaz */
 21   printf("Merhaba %s! \n", yourName);
```

```
22
23  return(0);
24 }
```

Şekil 2.1: Basit Bir C Programı

Satır numaraları açıklama yapmak için kullanılmıştır.
Bu programı kendi bilgisayarınıza yüklemek için buraya tıklayınız.

Şekil 2.1'deki programda '`/*`' ile başlayıp '`*/`' biten kısımlardaki herşey açıklamalardır. Programlama açısından hiçbir önemi olmayan bu satırlar yalnızca programı açıklamak için kullandığımız bilgileri kapsar. C derleyicisi bu bölümleri açıklama olarak görür ve dikkate almaz.

Programımızın içinde, kodun okunmasını kolaylaştırmak için, istediğimiz kadar boş satır kullanabiliriz. C derleyicisi bu boş satırları gözönüne almaz.

Şimdi bu örnek programı kullanarak C programlarının genel yapısını inceleyelim. Bu C programı iki kısımdan oluşur:*`Önişlemci bildirimleri`* ve *`main işlevi.`*

2.1.1. Önişlemci bildirimleri

Önişlemci bildirimleri # ile başlayan komutlardır. Bunlar C'nin önişlemcisi tarafından kullanılır. C'nin önişlemcisi program derlenmeden önce programın metininde bu komutları kullanarak bazı değişiklikler yapar ve program ondan sonra derlenir. `#include` ve `#define` C'de en çok kullanılan önşlemci bildirimleridir.

C dili program yazmada yararlı olabilecek birçok hazır işlevi programcının kullanımına sunar. Bu hazır işlevler C'nin değişik *kütüphanelerinde* saklanır. Her kütüphanede adı `.h`uzantısıyla biten bir üstbilgi kütüğü vardır. `#include` bildirimi programın C'nin kütüphanelerinden birine erişimini sağlar. Bu bildirim ile C önişlemcisi bir kütüphanenin üstbilgi kütüğündeki bazı tanımları programa aktarır ve program bu değişikliklerden sonra derlenir. Örneğin şekildeki programda kullanılan

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

önişlemciye bu programda kullanılan bazı işlev isimlerinin (`printf` ve `scanf` gibi) `stdio.h` adlı üstbilgi kütüğünde olduğunu bildirir.

Diğer önişlemci bildirimi

```
#define UZUNLUK 10
```

C önişlemcisinin, program içinde UZUNLUK sözcüğünün geçtiği her yerde bu sözcüğü 10 rakamı ile değiştirmesini sağlar. Örnek programımızda UZUNLUK yalnızca
`char yourName[UZUNLUK];`
satırında kullanılmış. Bu durumda C önişlemcisi bu satırı
`char yourName[10]`
olarak değiştirecektir. `#define` bildiriminde programımızda kullanacağımız, değeri program çalıştığı sürece değişmeyecek olan sabit değerleri tanımlayabiliriz. Bu programda UZUNLUK 10 olarak belirtildiği için programımız sadece 10 harfe kadar olan isimleri okuyabilir.

2.1.2. main işlevi

Her C programının bir `main` işlevi içermesi gerekir. Bu işlevin başlangıcı

```
int main(void)
```

ya da

```
int main()
```

sözcükleriyle belirtilir ve { ile } arasında kalan kısım işlevin gövdesidir. Bu programı çalıştırdığımızda ilk olarak bu `main` işlevine girilecek ve programın komutları o noktadan itibaren sırayla çalışmaya başlayacaktır.

C'de her işlevin gövdesi iki kısımdan oluşur: **tanımlamalar kısmı** ve **çalıştırılabilir ifadeler**. Tanımlamalar kısmında işlevin bellekte kullanacağı alanların isimleri tanımlanır (`yourName` gibi) ve bu bilgiler derleyici tarafından kullanılır. Çalıştırılabilir ifadeler algoritma basamaklarının C'deki yazılımlarıdır ve makina koduna bunlar çevirilip çalıştırılırlar.

`main işlevi` noktalama işaretleri ve bazı özel semboller (`*,=` gibi) içerir. Örneğin noktalı virgül (`;`) işareti her çalıştırılabilir ifadenin sonunu gösterir. { ve } işaretleri işlevin başlangıç ve bitiş yerlerini belirtir. Genelde her işlevde bir `return` ifadesi olmak zorundadır.

2.1.3. Özel sözcükler

Programımızda açıklamaların dışındaki satırlarda görülen bütün sözcükler ya özel sözcüklerdir ya da tanımlayıcılardır. Özel sözcüklerin hepsi küçük harflerle yazılırlar ve C'de hepsinin bir anlamı vardır. Bu sözcükler bilinen anlamlarından farklı bir amaçla kullanılamazlar. Programda geçen özel sözcükler ve anlamları aşağıdaki tabloda özetlenmiştir. C'deki özel sözcüklerin tamamını kitabımızın Appendix E kısmında ya da burada bulabilirsiniz.

Özel Sözcük	Programdaki Anlamı
`int`	integer(tam sayı)
`void`	main işlevi işletim sisteminden hiç veri almıyor
`char`	karakter veri türü
`return`	kontrol işlevden işletim sistemine geçiyor

2.1.4. Tanımlayıcılar

İşlev isimleri, bellek bölgelerinin isimleri (değişken ya da sabit isimleri) tanımlayıcılar olarak

adlandırılır. İki türlü tanımlayıcı vardır: **standart tanımlayıcılar** ve **kullanıcının tanımladığı**tanımlayıcılar. Standart tanımlayıcıların da özel sözcükler gibi özel anlamları vardır. Örneğin programda `printf` ve `scanf` C'nin `stdio` (standard input/output) kütüphanesinde tanımlanmış iki işlevin ismileridir. Bunların özel sözcüklerden farkı anlamlarının programcı tarafından değiştirilebilir olmasıdır.

Programlarımızda kullandığımız ya da hesapladığımız verileri tutan bellek bölgelerine kendimiz isimler veririz. Bunlara kullanıcının tanımladığı tanımlayıcılar denir. Örnek programda`yourName` ve `UZUNLUK` bu tip tanımlayıcılardır. Tanımlayıcılara istediğimiz isimleri verebiliriz. Yalnız şu kurallara dikkat etmemiz gerekir:

- Tanımlayıcı isimleri sadece harfler, rakamlar ve _ işaretini içerebilir.
- Tanımlayıcı ismi rakamla başlayamaz.
- Özel sözcükler tanımlayıcı ismi olarak kullanılamaz.

Örnek bazı tanımlayıcılar: x, y, number1, number2, p5, newValue, birKelime

Bazı geçersiz tanımlayıcılar: 1a (rakamla başlıyor), b+1 ('+' karakteri tanımlayıcıda kullanılamaz)

C büyük harf küçük harf ayrımına duyarlı olduğundan, aynı harfin büyük harfi küçük harfinden farklıdır. Bu nedenle örneğin `test1`, `Test1` ve `TEST1` üç farklı tanımlayıcı olarak algılanır.

2.2. Veri Türleri

Tanımlayacağımız her değişken bellekte bir bölgeye denk gelecektir. Bir değişkenin türü o değişkene denk düşen bellek bölgesinin büyüklüğünü ve o bölgenin içindeki bilginin nasıl yorumlanacağını gösterir. Bir değişkene yeni bir değer verdiğimizde, verdiğimiz bu değer o değişkene denk düşen bellek bölgesine eski değeri yok ederek saklanır.

Örneğin,

```
int x;
```

tanımlaması, `x` değişkeni için 32 bitlik bir bellek bölgesinin kullanılacağını ve bu bölgenin içindeki bitlerin bir tamsayı (integer) olarak yorumlanacağını gösterir.

Aşağıdaki atama ifadesi ile `x` değişkenine 10 değerini verebiliriz:

```
x = 10;
```

Bu atama ifadesinden sonra, `x` değişkenine denk gelen bellek bölgesi 10 sayısını 32 bit'lik bölgeyi kullanarak bir tamsayı olarak tutacaktır. Bu bellek bölgesi sonradan yeni bir atama ifadesi ile değiştirilebilir. Örneğin,

```
    x = 20;
```

atama ifadesi `x` değişkenin içine 20 sayısını koyacaktır ve eski değer olan 10 buradan silinecektir.

2.2.1. İkilik Düzendeki (Binary) Sayılar:

Bilgisayardaki her değer, ikilik düzendeki sayılar ile gösterilir. İkilik düzendeki tek bir rakam (0 ya da 1) bir bit olarak adlandırılır. Her bir bit iki değişik durumu gösterir. Bit dizgileri daha büyük sayıları göstermekte kullanılır. Her bit dizgisinin kaç değişik değeri gösterebileceği o dizgideki bit sayısına bağlıdır. Eğer bir bit dizgisinde N bit varsa, o dizgi 2^N değişik değeri gösterebilir. Örneğin bit sayısına göre kaç değişik değer gösterilebileceği aşağıdaki tabloda verilmiştir:

1 bit 2^1=2 değişik değer : 0 1
2 bit 2^2=4 değişik değer : 00 01 10 11
3 bit 2^3=8 değişik değer : 000 001 010 011 100 101 110 111
:
32 bit 2^{32}=4,294,967,296 değişik değer
64 bit 2^{64}=18,446,744,073,709,551,616 değişik değer

Gördüğünüz gibi dizgiye her bit eklediğimizde, o dizginin gösterebileceği değişik değerin sayısı iki katına çıkar. Tanımlayacağımız her değişken bellekte bir bit dizgisi olarak tutulacaktır ve bu bit dizgisinin nasıl yorumlanacağı ise o değişkenin türüne bağlı olacaktır. Örneğin veri türü `int` olan bir değişken 32 bitlik bir bit dizgisi olarak tutulacaktır ve bu bit dizgisi artı ya da eksi bir tamsayı olarak yorumlanacaktır.

C'deki her veri türü bir değerler kümesinden ve onların üzerinde yapılabilecek işlemlerden oluşur. Her veri türünün `int`, `double`, `char` gibi bir adı olacaktır. Bellekte tutulan her değer belirli bir veri türü kullanılarak yorumlanacaktır. C'deki her değişkenin bir türü olacaktır ve bir değişkenin türünün ne olacağını tanımlama (declaration) ifadesi ile belirtiriz. Örneğin,

```
int x, y;
int z;
double d;
```

tanımlama ifadeleri `x, y, z` değişkenlerinin türlerinin tamsayıların gösteriminde kullanılan bir veri türü olan `int` olacağını ve `d` değişkeninin türünün ise reel sayıların gösteriminde kullanılan bir veri türü olan `double` olacağını belirtir. C'de kullanılan her literalin (sabit sayı) de bir veri türü vardır. Örneğin, 5 bir tamsayıyı (yani `int`) ve 5.0 bir reel sayıyı (yani`double`) gösteren birer literaldir. Bellekte 5 ve 5.0 gösteren bit dizgileri tamamen farklı iki dizgi olacaklardır.

C'de standart veri türleri `char,double,int` gibi önceden tanımlanmış veri türleridir. `double` ve `int` reel ve tamsayıları programlarımızda göstermek için kullanılır. `char` ise her türlü karakter bilgisini saklamak için kullanılan bir veri türüdür. Standart veri türleri ile tanımlanan değişkenler için gerekli bellek, tanımlama (declaration) sırasında otomatik olarak ayrılır. C 'de standart veri türleri dışında programcı kendisi değişik veri türleri de tanımlayabilir (enumerated types). Ayrıca dinamik veri türleri de vardır (pointers). Dinamik veri türü olarak tanımlanan bir değişken için gerekli bellek bölgesi tanımlama anında değil de programın çalışma zamanında ayırılır.

Biz bu derste sadece standart veri türlerini anlatacağız. Önce sayısal veri türlerini, daha sonra da karakter veri türünü tanıtacağız.

2.2.2. Standart Veri Türleri:

C programlama dilinde aşağıdaki anahtar sözcüklerle ifade edilen 10 standart veri türü vardır:

`short, unsigned short, int, unsigned, long, unsigned long, float, double, long double, char`

Tamsayılar için standart veri türleri:

Tamsayıların gösteriminde altı değişik veri türü kullanabiliriz. Bu veri türleri şunlardır:

- `short`
- `unsigned short`
- `int`
- `unsigned`
- `long`
- `unsigned long`

Bu altı veri türünden üçü(unsigned long, unsigned, unsigned short) sadece pozitif sayıların gösteriminde, diğer üçü (short, int, long) ise hem pozitif hem de negatif sayıların gösteriminde kullanılabilir.Bu veri türlerinde farklı olan diğer bir nokta da değişik miktarda bellek kullanmalarıdır. Bu durumda gösterebilecekleri tamsayıların büyüklükleri de farklı olacaktır. Bu veri türlerinden biri ile tanımlanan değişkene denk düşen bellekteki bit dizgisi bir tamsayı (artı ya da eksi) olarak yorumlanacaktır. Her bir veri türü, yalnızca biri eksi (en küçük değer) diğeri artı (en büyük değer) olan belirli iki tam sayı arasındaki değerleri gösterebilir. Aşağıdaki tablo her veri türünün kaç bitlik bellek bölgesi kullandığını ve bu veri türü ile gösterilebilen en küçük ve en büyük tamsayıları gösterir.

Veri Türü	Bellek Kapasitesi	En Küçük Değer	En Büyük Değer
short	16 bit	-2^{15} = -32,768	$2^{15}-1$ = 32,767
unsigned short	16 bit	0	$2^{16} - 1$ = 65,535
int	16 bit	-2^{15} = -32,768	$2^{15}-1$ = 32,767
long	32 bit	-2^{31} = -2,147,483,648	$2^{31}-1$ = 2,147,483,647
unsigned long	32 bit	0	$2^{32} - 1$ = 4,294,967,295

C programlama dilinde tamsayıları aşağıdakiler gibi yazabiliriz:
`5, 3344, -89`
C'de `int` veri türü değerleri aynı zamanda mantıksal değerler olan doğru ve yanlışı ifade etmek için de kullanılır. Bir `int` değer pozitif ise doğru; sıfır ya da negatif ise yanlış olan bir durumu gösterebilir. Örneğin, `int` bir değer bir lambanın açık ya da kapalı olması gibi iki değişik değeri olan durumları göstermekte kullanılabilir. Pozıtif ise lamba açık; değilse kapalı olarak kabul edilebilir. `int` değerlerinin bu tip kullanımını daha sonraki derslerimizde detaylı olarak inceleyeceğiz.

Reel sayılar için standart veri türleri:

Reel sayılar için kullanılan üç standart veri türü vardır:

- `float`
- `double`
- `long double`

Bu üç veri türü de reel sayıların gösteriminde kullanılır. Bir reel sayı, bilimsel notasyonda `mantissa *` 2^{power} formülüyle gösterilebilir; böylece bir reel sayıyı bilgisayarda göstermek için kullanılan bit dizgisinin bir bölümü sayının rakamlarını gösteren `mantissa` için ayrılmış ve diğer bir bölümü de üs (`power`) göstermek için ayrılmıştır. `float,double` ve `long double` veri türlerinin kullandıkları bellek büyüklüğü ve `mantissa` ve `power` için ayırdıkları bit miktarı farklıdır. Aşağıdaki tablo bu veri türlerinin kaç bitlik bir alan kullandıklarını ve yaklaşık olarak gösterebilecekleri en küçük ve en büyük sayıları gösterir.

Veri Türü	Bellek Kapasitesi	En Küçük Değer	En Büyük Değer
float	32 bit	$-3.4x10^{-37}$	$3.4x10^{38}$
double	64 bit	$-1.7x10^{308}$	$1.7x10^{308}$
long double	80 bit	$-3.4x10^{4932}$	$1.1x10^{4932}$

`float` veri türü sayının 6 rakamını, `double` veri türü 15 rakamını, `long double` veri türü ise 19 rakamını saklayabilir.

Reel sayıları yazmak için noktalı notasyonu ya da bilimsel notasyon olarak bilinen e-notasyonunu kullanabiliriz. Bazı reel sayıları aşağıdaki gibi yazabiliriz:

```
5.0 7.34 0.5e+12 -15.35e-15
```

e-notasyonunda yazılan `0.5e+12` sayısı $0.5x10^{12}$ sayısına, `-15.35e-15` sayısı $-15.35x10^{-15}$ sayısına eşittir.

`char` standart veri türü:

`char` veri türü, harf, rakam ya da noktalama işareti gibi herhangi bir karakter değeri temsil eder. Her karakter sabiti iki tırnak işareti (`''`) arasına yazılır. Örneğin, bazı karakter sabitleri aşağıdaki gibi yazılabilir:

```
'A'   'B'   'a'   'b'   '0'   '1'   '*'    ':'   '"'    ' '
```

Sondan ikinci karakter çift tırnak işaretini.("), sonuncu ise boşluk karakterini göstermek için kullanılır (iki tırnak içine bir boşluk yazılır). Karakter değerleri char veri türü ile tanımlanmış bir değişkenin içinde saklanır. C değişik karakter değerlerinin karşılaştırılması ya da üzerlerinde aritmetik işlem yapılması gibi işlemler yapmaya izin verir.

2.2.3. Değişkenlerin Tanımlanması

Bir değişken belirli bir veri türünü tutan bir bellek bölgesini gösteren bir tanımlayıcıdır. Bütün değişkenler kullanılmadan önce tanımlanmalıdır. Bir tanımlama ifadesi aynı veri türüne sahip olacak bir ya da daha fazla değişkenin tanımlanmasında kullanılır. Bir tanımlama ifadesinin yapısı aşağıdakiler gibi olmalıdır:

```
veri-türü değişken-adı;
veri-türü değişken-adı1,değişken-adı2, ... ,değişken-adın;
```

Örneğin,

```
int sum;
int x,y;
double z;
```

Birinci tanımlama ifadesi `sum` değişkeninin türünün `int` olacağını, ikinci tanımlama ifadesi `x` ve `y` değişkenlerinin türünün `int` olacağını ve üçüncü ifade de `z` değişkeninin türünün`double` olacağını gösterir. Bir tanımlama ifadesinde birden fazla değişken tanımlanacaksa, bu değişkenler virgüllerle ayrılır.

Değişkenlere ilk değerleri istenirse tanımlanma anında verilebilir. Örneğin,

```
int sum = 0;
int x=0, y=10;
```

tanımlama ifadeleri aynı zamanda `sum`, `x` ve `y` değişkenlerine ilk değerlerini verirler.

Sayfa Başına

2.3. Atama ifadesi

Atama ifadesi, bir değişkene yeni bir değer vermek için kullanılır. Genelde aritmetik hesapların yapılmasında kullanılır. Bir atama ifadesinin yapısı aşağıdaki gibidir:

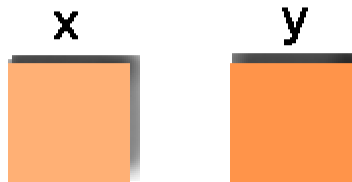
`değişken = deyim;`

C'de = işareti atama işlecidir. Sağdaki deyimin (expression) değeri bulunur ve bulunan değer soldaki değişkenin içine konur.

Deyim sadece bir değişken ya da sabit de olabilir. Örneğin,

```
int x, y;
x = 5;
y = x;
x = 6;
```

komutları ilk önce `x` değişkenin içine 5 değerini koyar; daha sonra `x` değişkenin içindeki değeri `y` değişkenin içine kopyalar ve en son olarak da `x` değişkenin içine 6 değerini, içindeki eski değeri silerek koyar.



Genel olarak bir deyim, işleçler (operators) ve işlenenlerden (operands) oluşan bir dizgidir. C'deki bazı aritmetik işleçler şunlardır:

- `+ toplama`
- `- çıkartma`
- `* çarpma`

- `/ bölme`
- `% kalan (mod operator)`

İşlenen (operand) ise bir değişken, bir sabit ya da diğer tek bir değeri gösteren yapılar olabilir. Örneğin,

```
x = y + 1;
y = x * z;
```

atama ifadelerinde, önce `y+1` deyimini hesaplamak için `y`'nin içindeki değer 1 ile toplanır ve sonuç `x` değişkeni içinde saklanır; sonra `x*z` deyimini hesaplamak için de `x`'in içindeki değer `z`'nin içindeki değer ile çarpılır ve sonuç `y` değişkeninin içinde saklanır.

2.3.1. İşleçlerin Öncelikleri (Operator Precedence)

Eğer bir deyimde birden fazla işleç varsa bu işleçleri hangi sırada yapacağımız işleçlerin önceliğine göre belirlenir. Örneğin,

```
x = y + z * 5;
```

ifadesinde ilk önce çarpma işlemini mi, yoksa toplama işlemini mi yapacağız?

Bu iki değişik sıra deyim sonucunun tümüyle iki farklı değerde olması demektir. C programlama dilinde çarpma işlecinin önceliği toplama işlecininkinden daha yüksek olduğu için ilk önce çarpma işlemi yapılacaktır. Yani ilk önce `z` ve `5` çarpılacak ve elde edilen sonuç ile `y` toplanacaktır.

Bazen bir deyimde ardarda gelen iki işlecin önceliği aynı olabilir. Bu durumda, işleçlerin hangi sırada yapılacağını gösteren birleşme (associativity) kuralına bakılarak işlem sırası belirlenir. Örneğin C'de toplama ve çıkartma aynı önceliğe sahiptir. Aşağıdaki ifadede toplama işlemleri soldan sağa doğru yapılacaktır:

```
x = y + z + 5;
```

Yani ilk önce `x` ve `y` toplanacak ve o toplamın sonucu 5 ile toplanacaktır. Bu örnekte toplamaların hangi sırada yapılacağı sonucu etkilemeyebilir; ama aşağıdaki örnekte etkileyecektir:

```
x = y - z + 5;
```

Yukarıdaki örnekte `-` ve `+` işleçleri aynı önceliğe sahip olduklarından, bu işlemleri soldan sağa doğru yaparız (ilk önce `-` işlemi, sonra `+` işlemi).

İşleç öncelikleri parantezler kullanılarak değiştirilebilir. Örneğin,

```
x = (y+z) * 5;
```

komutunda parantezleri kullanarak `+` işleci, `*` işlecinden daha önce yapılmaya zorlanmıştır.

Ders kitabımızın 2.5. kısmında C'deki işleçlerin öncelik sırası anlatılmıştır.

Örnekler:

Deyim	*Deyimin Değeri*
$\underbrace{\underbrace{5*6}_{1}-\underbrace{2*4}_{2}}_{3}$	22
$\underbrace{\underbrace{7-2}_{2}+\underbrace{5*4}_{1}}_{3}$	25
$\underbrace{\underbrace{\underbrace{(5*6)}_{1}/2}_{2}*3}_{3}$	45
$\underbrace{\underbrace{(\underbrace{\underbrace{(5+2)}_{1}*\underbrace{(7-2)}_{2}}_{3}+4)}_{4}*3}_{5}$	117

2.3.2. Atama İfadesinde Türlerin Uyuşması

Normal olarak bir atama ifadesinin solundaki değişkenin türü ile sağındaki deyimin türü aynı olmak zorundadır. Eğer sol taraftaki değişken türü `int` ise, sağ taraftaki deyimin türü de`int` olmak zorundadır. Örneğin,

```
int x;
double y;
x = y;
```

atama ifadesiyle bir `double` değeri, bir `int` değişkeni içine saklanmak istendiğinden, C derleyicisi bu atama ifadesine hata mesajı verecektir. Bunun nedeni herhangi bir `double`değeri, bir bilgi kaybı olmadan bir `int` değeri olarak saklanamaz.

Ancak bazı durumlarda C derleyicisi, değişkenin türü ile deyim türünün aynı olmasında ısrar etmez. Örneğin,

```
int x;
double y;
y = x;
```

komutuna C derleyicisi hata vermez. `x` değişkenin içindeki `int` değeri otomatik olarak bir `double` değere çevrilir ve bu `double` değer, `y` değişkenin içinde saklanır. Bu, otomatik tür değiştirme (automatic type casting) olarak bilinir. Örneğin, bütün tamsayı veri türlerindeki değerler otomatik olarak `double` değerine dönüştürülebilir. Bir tamsayı veri türündeki değer, o veri türünden daha fazla bellek kullanan tamsayı veri türüne otomatik olarak dönüştürülebilir. Örneğin, bir `short` değeri otomatik olarak `int` ya da `long` değerine dönüştürülebilir.

Deyimlerin hesaplanması anında da otomatik tür değiştirme meydana gelebilir. Örneğin,

```
5 + 3.2
```

deyiminin değeri hesaplanırken, 5 ilk önce `double` bir değere dönüştürülür ve bu `double` değerle 3.2 `double` değeri toplanır. Diğer bir deyişle, bu deyimdeki toplama

işlemi,`double` değerlerin toplama işlemidir. Bu deyimin sonucunun türü de `double` olacaktır.

Sayfa Başına

2.4. Basit Girdi/Çıktı

Hemen hemen bütün programlar dışarıdan girdi aygıtları vasıtasıyla veri alırlar ve hesapladıkları verileri kullanıcıya ekran, yazıcı gibi çıktı aygıtları vasıtasıyla sunarlar. Programın dış dünyadan veri almasına girdi işlemi; hesapladığı sonuçları sunmasına da çıktı işlemi denir.

C'de bütün girdi ve çıktı işlemleri **girdi/çıktı işlevleri** dediğimiz özel programlar tarafından yapılır. En çok kullanılan girdi/çıktı işlevleri C'nin standart girdi/çıktı kütüphanesinin bir parçası olarak sunulmuştur. Bu programlara

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

önişlemci yönlendirmesiyle erişebiliriz. Bu derste en temel girdi ve çıktı işlevleri olan `scanf` ve `printf`'in nasıl kullanıldıklarını göstereceğiz.

2.4.1. Basit Çıktı

Bir programın sonuçlarını görebilmek için, sonuçları tutan değişkenlerin değerlerini ya da mesajları bir çıktı aygıtında gösterebilmek gerekir. Bu amaçla `printf` işlevini kullanırız. Örneğin,

```
printf("bir dizgi.\n");
```

işlev çağırma komutu, bilgisayarın ekranına çift tırnak (") sembolleri arasında kalan kısmı yazar ve imleç bir sonraki satıra geçer. Yani, ekranda,

```
bir dizgi.
```

yazısını görürüz. Burada kullanılan **n** satır değiştirme damgasıdır (newline escape sequence). Bu karakter dizisi o andaki çıktı satırını sonlandırır ve imleci yeni satıra geçirir. Bir sonraki `printf` işlevini çağırma komutu verileri yeni satıra yazar. Eğer çıktımızı ekrana yazdırırken bir sonraki çıktı için alt satıra gitmek istemezsek, \n kullanmamalıyız. Örneğin,

```
printf("bir dizgi.");
```

işlev çağırma komutu, ekrana bir dizgi yazdıktan sonra, imleç bir alt satıra gitmeyecek o yazıdan hemen sonraki noktada bulunacaktır. Örneğin,

```
printf("bir dizgi.");
printf("diger dizgi");
```

işlev çağırma komutları ekrana

```
bir dizgi.diger dizgi
```

yazısını yazacaktır. Ama,

```
printf("bir dizgi.\n");
printf("diger dizgi");
```

komutları ekrana

```
bir dizgi.
diger dizgi
```

yazılarını yazacaktır.

`printf` işlevini çağırma ifadesinin genel yapısı aşağıdaki gibidir:

`printf(`*`biçimleme_dizgisi, çıktı_listesi`*`)`

Biçimleme dizgisi çift tırnak içinde yazılır, çıktı listesinde de değişkenler virgüllerle ayrılarak sıralanır. Örneğin

```
printf("Toplam uzunluk %f kilometredir. \n",kms);
```

ifadesi çalıştığında, eğer `kms` değişkeninin değeri 10.0 ise, ekranda

```
Toplam uzunluk 10.0 kilometredir.
```

yazısı görülecektir. Burada **`%f`** damgası double veri türünde bir değişken değeri için yer tutma görevini yapar. `printf` biçimleme dizgisini ekrana yazarken `%f` yerine çıktı listesindeki `double` değişkenin değerini yazar.

printf işlevi değişkenlerin değerlerini yer tutucu damgalar vasıtasıyla yazar. Bütün yer tutucular `%` işareti ile başlar. Her veri türü için ayrı bir yer tutucu damga vardır. Aşağdaki tablo`char`, `int` ve `double` veri türleri için kullanılan yer tutucu damgaları göstermektedir. Diğer yer tutucu damgaları ve değiştirme damgalarını kitabımızın 598-599. sayfalarında bulabilirsiniz.

Yer tutucu	Veri türü	Kullanıldığı işlev
%c	char	printf/scanf
%d	int	printf/scanf
%f	double	printf
%lf	double	scanf

Biçimleme dizgisi birden fazla yer tutucu damga içerebilir. Eğer çıktı listesinde birden fazla değişken ya da sabit varsa biçimleme dizgisi aynı sayıda yer tutucu damga içermelidir. C bunlarla değişkenleri soldan sağa olmak üzere sırasıyla eşleştirir. Örneğin, i ve j'nin `int` veri türünde değişkenler olduklarını ve değerlerinin sırasıyla 5 ve 9 olduğunu varsayarsak,

```
printf("%d %c %d esittir %d", i, '+', j, i+j);
```

ifadesi,

```
5 + 9 eşittir 14
```

yazacaktır.Bu ifadedeki biçimleme dizgisi ekrana yazılırken ilk %d i 'nin değeri ile, %c '+' ile, ikinci %d j'nin değeri ve üçüncü %d i+j değeri ile yer değiştirmiştir.

2.4.2. Basit Girdi

Örnek programımızda gördüğünüz

```
scanf("%s", yourName);
```

ifadesi scanf işlevini standart girdi aygıtından yourName değişkenine veri kopyalamak için çağırır. Standart girdi aygıtı çoğu kez klavyedir. Dolayısıyla bilgisayar kullanıcı klavyeden ne veri girerse onu yourName değişkeninin içinde saklamaya çalışacaktır.

scanf işlevini çağırma ifadesinin genel yapısı printf işlevini çağırma ifadesininkine benzer:

scanf(*biçimleme_dizgisi*, girdi_*listesi*)

Biçimleme dizgisi scanf'e ne tür bir veri girileceğini söyler. Örnekte görülen "%s" yer tutucu damgası yourName değişkeni için bir karakter dizisi beklendiğini belirtir. Tamsayı okuyacak olsaydık biçimleme dizgisine "%d", reel sayı için "%lf", karakter veri için ise "%c" yazacaktık. Örneğin,

```
scanf("%c%c", &harf1, &harf2);
```

ifadesi veri türü char olduğunu varsaydığınız harf1 ve harf2 değişkenlerine klavyeden girilen iki karakter değerini kopyalayacaktır.

```
scanf("%d%lf", &age, &average);
```

ifadesi ise int veri türündeki age değişkeni ile double veri türündeki average değişkenine değerlerini klavyeden okur.

Bu örneklerde programdaki ifadeden farklı olarak değişkenlerin başına & işareti koyduk. & işareti önüne koyulduğu değişkenin bellekteki adresini döndüren bir işlemcidir. Bu işaretichar, int ve double veri türündeki değişkenleri okurken koymamız gerekir. Ancak örnek programımızdaki tanımlama kısmında yaptığımız gibi bir dizi(array) tanımlamışsak veya bir değişkeni tanımlarken *kullanıldıysa bu işarete gerek yoktur. Bunun neden böyle olduğunu daha sonraki derslerimizde açıklayacağız!

Klavyeden kaç tane veri alınacağı biçimleme dizgisindeki yer tutucu damgaların sayısıyla belirtilir. Verileri okuma işleminde verilerin türleri çok önemlidir. C int ya da reel sayıları okurken sayıların başındaki boşlukları atlar ve sayının bir parçası olamayacak ilk karaktere kadar (boşluk ya da baska herhangi bir karakter) olan karakterleri sayı olarak okur. charveri türü okunurken ise her boşluk ayrı bir karakter olarak görülür. char veri türü okurken de boşlukları atlamak istersek, kaç boşluk atlamak gerekiyorsa o kadar boşluk karakterinin biçimleme dizgisine yazmamız gerekir.

Bir programa klavyeden veri girerken biçimleme dizgisinde belirtilenden daha fazla veri girersek fazladan girilen veriler bellekte saklanır ve programda (varsa) bir sonraki scanfifadesi tarafından okunur. Daha az veri girersek, program biz gerekli veri bilgisini tamamlayana kadar bekler.

2.4.3. Çıktıların biçimlendirilmesi

Bu bölümde programımızın çıktılarının ekrandaki görüntülerini nasıl kontrol edeceğimizi göstereceğiz.
int veri türündeki değerlerin görünüşünü düzenlemek için, değeri yazmak için kaç kolonluk yer ayırmak gerektiğini "%d" damgasına ekleriz. Örneğin

```
printf("   Sonuc = %3d kilo %4d gramdır.\n", k, g);
```

ifadesi, eğer `k` değişkeninin değeri 21, `g` değişkeninin değeri 50 ise

```
   Sonuc =  21 kilo   50 gramdır.
```

çıktısını verecektir. Burada 21 değeri 3 karakterlik bir alana, 50 değeri de 4 karakterlik bir alana yazılmıştır.

Reel sayıların görünüşünü ayarlamak için hem kaplayacağı toplam kolon sayısını hem de noktadan sonra kaç basamak basılacağı bilgisini vermek gerekir. Toplam alanı belirtirken noktanın kaplayacağı bir kolonu da saymak gerekir. Örneğin 3.14159 değerini taşıyan `pi` değişkenini

```
printf(" PI =  %5.2f \n", pi);
```

ifadesiyle yazmak istersek, ekranda

```
 PI =   3.14
```

görülecektir. "`%5.2f`" damgasıyla `pi` değişkenini toplam 5 kolonluk alana noktadan sonra 2 basamak bırakarak yazmak istediğimizi söylüyoruz. Çıktıda = işaretinden sonra 2 boşluk vardır. Biri biçimleme dizgisinde yazdığımız boşluktur, diğeri ise `pi`'yi yazmak için kullandığımız toplam 5 kolonluk alandan kalan boşluktur.

2.4.4. Örnek Program

Aşağıdaki program girdi/çıktı işlemlerini örneklemek amacıyla verilmiştir.

```
 /* Yazar : Nihan Kesim Cicekli
 * Tarih : Temmuz 2000
 *
 * Bu program, iki tamsayiyi klavyeden okur ve
 * bu iki sayinin toplamini ve birinci ile ikinci sayi arasindaki
 * farki bularak ekrana bastirir.
 */

#include <stdio.h>

int
main (void) {
   int i,j; /* okunan iki sayi */
   int toplam, fark; /* bulunacak toplam ve fark */

   /* Tamsayilari bildirim basarak oku*/
   printf("Birinci sayiyi girin:");
   scanf("%d", &i);

   printf("Ikinci sayiyi girin:");
   scanf("%d", &j);

   /* Toplami bul */
   toplam = i+j;

   /* Farki bul */
```

```
    fark = i-j;

    /*Toplami ve farki ekrana yazdir */
    printf("Toplam = %3d\n", toplam);
    printf("Fark   = %3d\n", fark);

    return(0);
}
```

Bu programı kendi bilgisayarınıza yüklemek için buraya tıklayınız.

Bu program, her sayıyı okumadan önce, ekrana, `printf` işlevi ile kullanıcıya ne yapacağını söyleyen bildirimi bastırır ve kullanıcının yazdığı sayı okunarak değişkenin içinde saklanır.

Sayılar okunup değişkenler içinde saklandıktan sonra, toplamları ve farkları atama ifadeleri ile bulunur. Bulunan sonuçlar, `printf` işleviyle ekrana gönderilir.

Bu örnek programı kendi bilgisayarınızda çalıştırın. Daha sonra biçimleme dizgisini değiştirerek sonuçların ekranda nasıl gösterildiğini inceleyin.

Sayfa Başına

2.5. Hata Mesajları

Bu haftadan itibaren kendi programlarımızı yazmaya başlayacağız. İdeal bir durumda, yazdığımız program bilgisayarda ilk denememizde hiç hata vermeden tam istediğimiz gibi çalışabilir. Ama bu büyük ihtimalle hiç olmayacağı için karşılaşacağımız hata çeşitlerini bilmemizde yarar vardır. Üç hata çeşidi ile karşılaşabiliriz: Derleme sırasındaki hatalar, çalıştırma sırasındaki hatalar ve mantık (ya da tasarım) hataları.

2.5.1.Derleme hataları

Derleme hataları program derlenirken ortaya çıkan hatalardır. Bunlar aynı zamanda *syntax hataları* olarak da bilinir. Programınızı yazarken yazım hataları yaparsanız (imla hatası gibi), C'nin kurallarına uymazsanız (örneğin ; ya da { ve } gibi işaretleri yanlış yerlerde kullanmak gibi) derleme hataları alırsınız. Bunlar bulması ve düzeltmesi en kolay hatalardır. C'de en sık yapılan yazım hatalarından birkaçını sıralayalım.

C'de komut içeren her ifade ';' işareti ile bitirilmelidir. Bu C'nin ifadeleri birbirinden ayırabilmesi için gereklidir. Eğer ';' unutulursa C derleyicisi hata mesajı verecektir.

Açıklamalar mutlaka '`/*`' ile başlamalı ve '`*/`' ile bitmelidir.

Programların bölümlerinin başlangıç ve sonlarını belirleyen '{' ve '}' işaretlerini unutmak ya da yanlış kullanmak da en yaygın hatalardır.

Programda kullanılan bir değişkenin tanımlanması unutulursa bu da bir derleme hatasına

sebep olur.

2.5.2. **Çalıştırma hataları**

Çalıştırma hataları program derlendikten sonra çalıştırılırken alınan hatalardır. Programınızı syntax hatalarından temizledikten sonra çalıştırdığınızda programın çıktısı yerine hata mesajı alırsanız ya da programınız hiç sonuç vermeden ekran kilitlenmiş gibi boş kalırsa çalıştırmada bir hata var demektir. Bu tip hataların nedeni daha zor bulunur, çünkü nedenler programın içeriğine göre çok çeşitlidir. Örneğin programınızda bir bölme işlemi vardır ve programın çalışma sırasında bölen değer sıfır oluyordur. İşte bu durumda sıfıra bölme işlemi çalıştırma hatası verecektir. Bu tip hataları programlama yeteneğiniz arttıkça bulmanız ve düzeltmeniz kolaylaşacaktır.

2.5.3. Mantık hataları

Mantık hataları da algoritma tasarımı sırasında ya da algoritmayı programa kodlarken yapılan bir hatadan kaynaklanır. Mantık hataları genelde derleme ve çalıştırma hatalarından kurtulduktan sonra ortaya çıkar. Bu aşamada programınız çalışır ve bir çıktı verir. Ancak çıktıyı incelediğinizde bunun sizin istediğiniz çıktı olmadığını görürsünüz. Örneğin çıktılar düzgün bir sırada alt alta çıkmamıştır, ya da rakamlar hatalı hesaplanmıştır ya da program eksik çıktı vermiştir. Bu tip hataları düzeltmek için programın bazı kısımlarını yeniden yazmak hatta bazen algoritmayı değiştirmek gerekebilir.

3. Karar Verme Yapıları

Amaç:

Bu hafta seçeneğe dayalı işlemlerin nasıl yapılacağı gösterilecek.

1. Bu amaçla C'de bulunan karar verme yapılarından `if` ifadesinin seçeneğe dayalı işlemlerde nasıl kullanılacağını göreceğiz. Bu ifadenin değişik yapılarını ve nasıl çalıştığını inceleyeceğiz.
2. Karar verme bir mantıksal deyim (logical expression) sonucuna bağlı olacağından, mantıksal deyim yapıları da ayrıntılı olarak sunulacaktır.
3. Çoktan seçmeli işlemlerde kullanacağımız diğer bir karar verme yapısı olan `switch` ifadesi de tanıtılacaktır.

3.1. Koşullar

Koşullar sonucu 1 (doğru) ya da 0 (yanlış) olan mantıksal deyimlerdir. Basit bir mantıksal deyim karşılaştırma (relational) işleçleri kullanılarak yaratılabilir. Örneğin,

```
x < y
```

mantıksal deyimi, `x` ve `y` değişkenlerinin `<` karşılaştırma işleci kullanılarak birleştirilmesiyle yaratılmıştır. Eğer `x` değişkenin değeri, `y` değişkenin değerinden daha küçükse, bu mantıksal deyimin sonucu 1 (doğru) olacaktır; aksi halde bu deyimin sonucu 0 (yanlış) olacaktır.

Genelde bir mantıksal deyim bir karşılaştırma işleciyle yaratıldığında aşağıdaki yapıda olur:

```
deyim1  karşılaştırma-işleci  deyim2
```

Genelde buradaki `deyim1` ve `deyim2` birer aritmetik deyim olabilir ve onların sonuçları `karşılaştırma-işleci` kullanılarak karşılaştırılabilir. C'de kullanılan karşılaştırma işleçleri şunlardır:

<	küçük	eğer sol taraftaki değer sağ taraftaki değerden küçükse, sonuç 1, aksi halde 0
<=	küçük ya da eşit	eğer sol taraftaki değer sağ taraftaki değerden küçük ya da eşitse, sonuç 1, aksi halde 0
>	büyük	eğer sol taraftaki değer sağ taraftaki değerden büyükse, sonuç 1, aksi halde 0
>=	büyük ya da eşit	eğer sol taraftaki değer sağ taraftaki değerden büyük ya da eşitse, sonuç 1, aksi halde 0
==	eşit	eğer sol taraftaki değer sağ taraftaki değere eşitse, sonuç 1, aksi halde 0

!=	eşit değil	eğer sol taraftaki değer sağ taraftaki değere eşit değilse, sonuç 1, aksi halde 0

Örneğin, `x` değişkeni 10 değerini ve `y` değişkeni 5 değerini tutuyor olsun. Bu durumda aşağıdaki mantıksal deyimlerin değerleri şöyle olacaktır:

Mantıksal Deyim	**Sonuç**
x >1	*1 (doğru)*
x <10	*0 (yanlış)*
x >= 10	*1 (doğru)*
x >y	*1 (doğru)*
x ==y	*0 (yanlış)*
x !=y	*1 (doğru)*
(y+6) >x	*1 (doğru)*
*(y-x) >=(x*y)*	*0 (yanlış)*

Daha karmaşık mantıksal deyimler, *mantıksal işleçler* (logical operators) kullanılarak yaratılabilir. Bu mantıksal işleçlerden çoğu iki mantıksal deyimden yeni bir mantıksal deyim yaratırlar; bir tanesi ise tek mantıksal deyimden yeni bir mantıksal deyim yaratır. C programlama dilindeki mantıksal işleçlerden bazıları şunlardır:

mantıksal-deyim1 **&&** *mantıksal-deyim2*	***ve (and)***
mantıksal-deyim1 **||** *mantıksal-deyim2*	***veya (or)***
! *mantıksal-deyim*	***değil (not)***

Bu mantıksal işleçlerin sonucu bu işleçlerin doğruluk tablolarına (truth tables) göre bulunur. Bu işleçlerin doğruluk tabloları aşağıdaki gibidir:

Deyim	**E1**	**E2**	**Sonuç**
E1 && E2	sıfır değil (doğru)	sıfır değil (doğru)	1 (doğru)

	sıfır değil (doğru)	0 (yanlış)	0 (yanlış)
	0 (yanlış)	sıfır değil (doğru)	0 (yanlış)
	0 (yanlış)	0 (yanlış)	0 (yanlış)
E1 || E2	sıfır değil (doğru)	sıfır değil (doğru)	1 (doğru)
	sıfır değil (doğru)	0 (yanlış)	1 (doğru)
	0 (yanlış)	sıfır değil (doğru)	1 (doğru)
	0 (yanlış)	0 (yanlış)	0 (yanlış)
! E1	sıfır değil (doğru)		0 (yanlış)
	0 (yanlış)		1 (doğru)

Bu tabloda da görüldüğü gibi C mantıksal bir deyimin sonucunu daima 0 ya da 1 olarak hesaplar. Ancak C sıfır olmayan herhangi bir değeri de 'doğru' olarak görür. Şimdilik biz doğru değeri hep 1 tamsayısı ile göstereceğiz. Örneğin, `x` değişkeni 10 değerini ve `y` değişkeni 5 değerini tutuyor olsun. Bu durumda aşağıdaki mantıksal deyimlerin değerleri şöyle olacaktır:

Mantıksal Deyim	Sonuç
`(x>1) && (y<5)`	0 (yanlış)
`(x>1) || (y<5)`	1 (doğru)
`(x>1) ^ (y<5)`	1 (doğru)
`!(x>1)`	0 (yanlış)

Mantıksal işleçlerin de aritmetik işleçler gibi bir öncelik sırası vardır. Şimdiye kadar gördüğümüz işleçler öncelik sırasına göre şöyle sıralanır:

1. `! - (unary minus)`
2. `* / %`
3. `+ -`

4. `<  >  <=  >=`
5. `==  !=`
6. `^`
7. `&&`
8. `||`

1 numaradaki işleçler önceliği en yüksek işleçler, 8 numaradaki işleç önceliği en düşük işleçdir. Örneğin,

```
x>1 || y>2 && z>3
```

mantıksal deyiminde `&&` işlemi `||` işleminden önce yapılacaktır. Çünkü onun önceliği daha yüksektir. Bütün işleçlerin öncelik sırası kitabımızın Appendix C bölümünde bulunabilir. Bu öncelik sırasını açık hale getirmek ya da değiştirmek istersek parantezler kullanabiliriz. Örneğin yukarıdaki ifadeyi

```
(x>1 || y>2) && z>3
```

şeklinde yazarsak, `||` işlemi `&&` işleminden önce yapılacaktır.

Programlarımızda birşeyin doğru ya da yanlış olduğunu `int` veri türündeki bir değişken kullanarak gösterebiliriz. Atama ifadesiyle bu değişkene 0 değeri verirsek yanlış, sıfır olmayan bir değer verirsek doğru olarak kabul ederiz. Örneğin, `flag` `int` veri türünde bir değişken olsun ve bu değişkenle `x` değikeninin 10'dan büyük olup olmama durumunu belirtmek isteyelim. Bu durumu aşağıdaki atama ifadesiyle belirtebiliriz.

```
flag = (x>1);
```

Bu atama ifadesi eğer `x`'in değeri 10'dan büyükse `flag`'a 1 (doğru), değilse `flag'a` 0 (yanlış) değerini koyacaktır.

```
cift = (n % 2 == 0);
```

ifadesi ise eğer `n` bir cift sayıysa `cift`'e 1 değerini verecektir.

Özetleyecek olursak C'de koşul şunlardan biridir:

- bir `int` değer ya da bir `int` değişken
- yalnızca karşılaştırma işleçleri kullanılarak yaratılan basit bir mantıksal deyim
- diğer mantıksal deyimlerden mantıksal işleçler kullanılarak yaratılan bir mantıksal deyim

Kısa devre değerlendirme (Short-circuit Evaluation)

İf ifadesine geçmeden önce C'nin koşulların sonucunu hesaplama şeklini anlatmak gerekir. C mantıksal deyimlerin sadece bir bölümünün değerini hesaplar. Bütün deyim ancak gerektiği durumlarda hesaplanır. Örneğin `a || b` deyimi `a` doğruysa doğru olacaktır. Bu yüzden C bu tip bir ifadeyi çalıştırırken eğer ilk değer sıfır değilse ikinci değeri hiç hesaplamaz. Benzer şekilde `a && b` deyimi `a` yanlış ise yanlış olacaktır. Bu tip

koşullarda da C ilk değer sıfırsa ikinci değere hiç bakmadan sonucu sıfır olarak bulur. Bu değerlendirme metoduna kısa devre değerlendirme diyoruz.

3.2. IF ifadesi

Programlarımızda bir komutlar kümesinden koşullara bağlı olarak yalnızca birini çalıştırmak isteyebiliriz. Diğer bir deyişle bazı durumlarda, komutlardan yalnızca birini çalıştırır diğerlerini çalıştırmayabiliriz. C programlama dilinde koşula bağlı çalıştırma işlemini genellikle `if` ifadesini kullanarak yaparız. C'de `if` ifadesinin iki değişik yapısı vardır.

3.2.1. Bir seçenekli if ifadesi

Bu yapıyla bir komutlar dizisini koşula bağlı olarak ya çalıştırırız yada çalıştırmayız.Bir seçenekli `if` ifadesinin yapısı şöyledir:

```
if ( koşul )
   doğru-ifade;
```

Bu `if` ifadesinde ilk önce bir mantıksal deyim olan *koşul*'un değeri bulunur. Eğer koşulun değeri sıfır değilse (yani doğruysa) *`doğru-ifade`* çalıştırılır. Aksi halde *`doğru-ifade`*hiç çalıştırılmadan bir sonraki ifadeye geçilir. Buradaki `koşul` herhangi bir mantıksal deyim olabilir ve `doğru-ifade` C' deki herhangi başka bir ifade olabilir. Şimdiye kadar gördüğümüz ifade yapıları şunlardır: atama ifadesi, işlev çağırma ifadesi ve `if` ifadesi. Tabii burada kullanılacak ifadeler bunlarla sınırlı değildir; daha sonra göreceğimiz gruplama (block), `while`, `for` gibi diğer ifadeler de olabilir.

3.2.2. İki seçenekli if ifadesi

Bu ifadeyle koşula bağlı olarak iki C ifadesinden birini çalıştırabiliriz. İki seçenekli `if` ifadesinin yapısı aşağıdaki gibidir:

```
if ( koşul )
     doğru-ifade ;
else
     yanlış-ifade ;
```

Eğer koşul değeri sıfır değilse `doğru-ifade` çalıştırılır, aksi halde (yani değeri sıfırsa) `yanlış-ifade` çalıştırılır. Bu yapıda da `doğru-ifade` ve `yanlış-ifade` C 'deki herhangi bir ifade olabilir.

NOT: Daha önce algoritmalarımızda kullandığımız **eğer-değilse** türü karar verme işlemlerini C'de `if` ifadeleriyle yapabiliriz.

Örnekler:

1.

```
if (y != 0)
     x = x / y;
else
```

```
        printf("y is zero \n");
```

Bu örnekte `y`'nin değeri sıfırdan farklı ise, `x`'in içindeki değer `y`'nin içindeki değer ile bölünür ve sonuç `x`'in içinde saklanır. Aksi halde `y`'nin sıfır olduğunu gösteren mesaj ekrana yazılır.

2.

```
    if (x >= 0)
        printf("x is positive \n");
    else
        printf("x is negative \n");
```

Eğer `x`'in değeri 0'a eşit ya da büyük ise, ekrana onun pozitif olduğu, aksi halde onun negatif olduğu yazılır.

3.

```
    if (x < 0)
        sign = -1;
    else
        sign = 1;
    absulatex = sign * x;
```

Eğer `x`'in değeri 0'dan küçük ise, `sign` değişkenin içine –1 değeri, aksi halde 1 değeri konur. Bu `sign` değişkeni içindeki değer, `x` değişkenin pozitif ya da negatif olduğunu gösterir. `if` ifadesinden sonraki atama ifadesi de `x` değerini `sign` değeri ile çarparak `x`'in mutlak değerini bulur.

4.

```
    if ((x>1) && (x<10))
        x = x + 1;
```

Bu ifadedeki atama ifadesi yalnızca `x` değeri 1'den büyük ve 10'dan küçük olduğu zamanlarda çalıştırılacaktır. Aksi halde bu `if` ifadesi hiçbir şey çalıştırmadan bir sonraki ifadeye geçilecektir.

3.3. Öbek İfade (Compound Statement)

'{' ve '}' sembolleri arasında kalan komutlar kümesi öbek ifade olarak adlandırılır. Öbek ifade de C programlama dilindeki ifadelerden biridir. Diğer ifadelerin kullanıldığı her yerde bir öbek ifade de kullanılabilir. Diğer ifadelerden sonra her zaman `';'` işareti koyarız, ama öbek ifadesini kapatan '}' den sonra `';'` işareti koymamıza gerek yoktur. Daha önce öbek ifadesiyle karşılaşmıştık. Örneğin bir işlevin gövdesi bir öbek ifadedir. If ifadesi içinde öbek ifadesini `doğru-ifade` veya `yanlış-ifade` yerine kullanabiliriz. If ifadesinin koşulu doğru olduğunda (ya da else kısmında) eğer birden fazla işlem yapmamız gerekiyorsa, bu işlemleri yapan bütün C ifadelerini bir öbek ifadesi biraraya toplarız.

Örnekler:

1.

```
if (indirimOrani != 0) {
     indirim = fiyat * indirimOrani;
     fiyat = fiyat - indirim;
}
```

Bu örnekte iki atama ifadesi bir öbek ifade içinde toplanmıştır. Eğer `if` ifadesinin koşulu 1 ise bu öbek ifadedeki bütün ifadeler çalıştırılacaktır.

2.

```
if (x>y) {

      temp = x;
      x = y;
      y = temp;
 }
```

Bu örnekte, eğer `x`'in değeri `y`'nin değerinden büyük ise, üç atama ifadesini kapsayan öbek ifade çalıştırılır. Bu öbek ifadedeki atama ifadeleri `x` ve `y` değişkenlerinin içindeki değerlerin yerlerini `temp` değişkenini kullanarak değiştirir. Eğer `x'`in değeri `y'`den büyük değilse bu öbek ifades hiç çalıtırılmaz.

3.

```
if (x != y) {
    printf("x ve y farkli degerlere sahipler. \n");
    printf("x: %d  y: %d \n", x, y);
}
else {
    printf("x ve y ayni degerlere sahipler. \n");
    printf("x ve y: %d \n", x);
}
```

Bu örnekteki `if` ifadesi eğer `x`'in değeri `y`'nin değerinden farklı ise bunu gösteren mesajı bir satıra yazar ve sonra `x` ve `y`'nin değerlerini bir sonraki satıra ayrı ayrı yazar. Aksi halde, eşit oldukları mesajını yazar ve sadece `x` değerini yazar. Görüldüğü gibi bu örnekteki `if` ifadesinin hem `doğru-ifadesi`, hem de `yanlış-ifadesi` birer öbek ifadedir.

3.4. İç-İçe IF İfadeleri (Nested IF Statements)

İkiden fazla koşulu içeren karar verme yapılarında da if-ifadelerini kullanabiliriz. Eğer bir `if`-ifadesinin `doğru-ifadesi` ve/veya `yanlış-ifadesi` de başka bir `if`-ifadesi ise, böyle `if`-ifadelerini `iç-içe if`-ifadeleri olarak adlandırırız.

Örneğin,

```
if (x<10)
```

```
    printf("x bir rakamdan olusur");
else
  if (x<100)
      printf("x iki rakamdan olusur");
  else
      printf("x ikiden fazla rakamdan olusur");
```

ifadesi bir iç-içe `if`-ifadesidir. Dışarıdaki `if`-ifadesinin (kırmızı renkle gösterilen) `yanlış-ifadesi` (else kısmı) başka bir `if`-ifadesidir (yeşille gösterilen). İlk önce `x`'in 10'dan küçük olup olmadığı sınanacaktır. Eğer küçükse, `x`'in tek rakamdan oluştuğunu yazacak olan `printf` işlevi çalışacaktır. Değilse, dışardaki (kırmızı) `if`-ifadesinin `else`-bölümündeki `if` ifadesi çalışacaktır. Bu durumda (yani x 10'dan küçük değilken), `x`'in 100'den küçük olup olmadığı kontrol edilecektir. Eğer küçükse, ikinci `if` ifadesinin (yeşil olan) doğru bölümünü oluşturan ve `x`'in iki rakamdan oluştuğunu yazan `printf` işlevi çalışacaktır. Aksi halde (yani x 100'den küçük değilse) ikinci `if`-ifadesinin `else`-bölümünü oluşturan ve `x`'in ikiden fazla rakamdan oluştuğunu yazacak olan `printf` işlevi çalışacaktır.

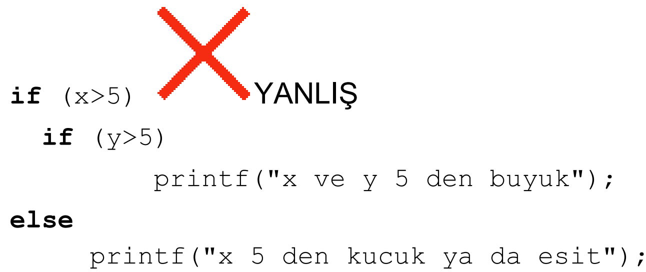
Aşağıdaki örnekte bir iç-içe `if` ifadesi görülmektedir. Bu yapı her `if`-ifadesinin (sonuncu `if`-ifadesi hariç) `else`-bölümünde yine bir `if`-ifadesi yazarak ortaya çıkmıştır. Birbiriyle eşleşen `if` ve `else`'ler aynı renkte gösterilmiştir.

```
if (grade>=90)
    printf("A");
else if (grade>=80)
    printf("B");
else if (grade>=70)
    printf("C");
else if (grade>=60)
    printf("D");
else
    printf("F");
```

Bu `if` ifadesi, eğer `grade` 90'dan büyük ya da eşitse A yazar. Değilse, `grade'`in 80'den büyük ya da eşit olup olmadığını kontol eder ve eğer öyleyse B yazar. Değilse ve`grade'`in değeri 70'ten büyük ya da eşit ise C yazar. Aksi halde, `grade'`in değerini tekrar 60 ile karşılaştırır ve bu karşılaştırmanın sonucuna göre D ya da F yazar.

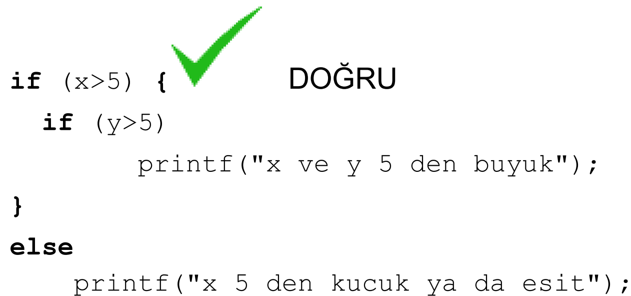
Eşleşmemiş Else Problemi (Dangling Else Problem)

C programlama dili, her zaman bir `else`-ifadesini bir önceki `if`-ifadesinin parçası olarak görür. Bu yüzden iç-içe `if`-ifadelerini kullanırken, eğer bazı `if`-ifadelerin `else`-bölümleri yoksa dikkatli olmalıyız. Örneğin,




```
if (x>5)
  if (y>5)
         printf("x ve y 5 den buyuk");
else
     printf("x 5 den kucuk ya da esit");
```

Bu iç-içe `if`-ifadesi beklediğimiz davranışı göstermeyecektir. Buradaki `else` bölümü, dışardaki `if`-ifadesi ile değil de içerideki `if`-ifadesiyle eşleşecektir (yeşil renkle gösterilmiştir). Bu yüzden `x`'in 5'ten küçük ya da eşit olduğunu yazan `printf` komutu, `x`'in 5'ten büyük ve `y`'nin 5'ten küçük ya da eşit olduğu durumlarda çalıştırılacaktır. Eğer `else`bölümünü dışardaki `if`-ifadesi (kırmızı olan) ile eşleştirmek istiyorsak, yukarıdaki örneği aşağıdaki gibi değiştirmeliyiz:




```
if (x>5) {
  if (y>5)
        printf("x ve y 5 den buyuk");
}
else
    printf("x 5 den kucuk ya da esit");
```

Böylece `x`, 5'ten büyük değilse, `else` kısmı çalıştırılacaktır. Bu yazımda dışarıdaki (kırmızı) `if`-ifadesinin doğru bölümü bir öbek ifadedir ve bu öbek ifade bir seçenekli `if` ifadesini kapsamaktadır. Gördüğünüz gibi burada tek bir ifadeyi öbek ifadenin içine koyduk. Bunu if ve else'leri doğru eşleştirmek için yaptık.

3.5. IF-ifadeleri ile Bir C Programı

MinValue.cpp

```
/*
 * Bu program uc tamsayiyi klavyeden okur ve
 * bu uc sayinin en kucugunu bularak ekrana bastirir.
 */
#include <stdio.h>;

int main () {

   int num1,num2,num3; /* okunan tamsayilar */
   int minValue;     /* bulunacak olan en kucuk deger */

   /* Ilk tamsayiyi oku */
   printf("Birinci tamsayiyi girin : ");
   scanf("%d", &num1);

   /* Ikinci tamsayiyi oku */
```

```
   printf("Ikinci tamsayiyi girin : ");
   scanf("%d", &num2);

   /* Ucuncu tamsayiyi oku */
   printf("Ucuncu tamsayiyi girin : ");
   scanf("%d", &num3);

   /* En kucuk degeri bul:
       1. ve 2. sayilarin en kucugunu bul ve minVal da sakla
       3. sayidaki ve minValue'daki degerin en kucugunu bul,
       minValue'daki deger, 3 sayinin en kucugu olacaktir.*/

   if (num1<num2)
      minValue = num1;
   else
      minValue = num2;

   if (num3<minValue)
      minValue = num3;

   /* En kucuk degeri ekrana yazdir */
   printf("En kucuk deger: %d \n", minValue);
   return(0);
}
```

Bu program üç tamsayıyı ayrı ayrı okur ve okuduğu bu sayıları `num1`, `num2` ve `num3` değişkenleri içinde saklar. Sayılar okunduktan sonra bu üç sayının en küçüğü, iki `if`-ifadesi kullanılarak bulunur. İlk `if`-ifadesi iki seçenekli bir `if` ifadesidir ve ilk iki sayının en küçüğünü `minValue` değişkenin içinde saklar. İkinci `if`-ifadesi bir seçenekli `if` ifadesidir ve eğer `num3` değişkeninin içindeki değer, `minValue` değişkeninin içindeki değerden küçükse, `num3'`teki değer `minValue`'ya aktarılır. Böylece `minValue` girilen üç sayının en küçüğünü tutacaktır. Daha sonra bu en küçük değer ekrana yazdırılır.

3.6. switch ifadesi

Çok sayıdaki seçenekler arasından bir seçim yapmak isteniyorsa, diğer bir karar verme yapısı olan `switch`-ifadesi de kullanılabilir. Seçenekler arasındaki seçim tek bir değişkenin ya da basit bir deyimin değerine dayalı olarak yapılıyorsa `switch`-ifadesi çok yararlı olabilir. Seçimin dayalı olduğu bu deyimin veri türü ya `int` ya da `char` olmalıdır. `double` veri türü olamaz. `switch` ifadesinin genel yapısı şöyledir:

`switch (` *kontrol-deyimi* **`) {`**
değer-kümesi-1
ifade(ler)
`break;`
değer-kümesi-2
ifade(ler)

```
            break;
        ...
        ...
        değer-kümesi-n
          ifade(ler)
            break;
        default:
          ifade(ler)
    }
```

Burada *kontrol-deyimi* veri türü `int` ya da `char` olan bir deyimdir; *değer-kümeleri* kontrol-deyiminin alabileceği değerlerin listesidir ve *ifade(ler)* de C'deki herhangi bir veya daha fazla ifade olabilir. Değer kümeleri bir ya da daha fazla

case *sabit* **:**

şeklindeki etiketlerden oluşur. `break` ve `default:` kısımları `switch` ifadesinde bulunmak zorunda değildir (optional).

Bir `switch`-ifadesinde ilk önce *kontrol-deyimi*nin değeri hesaplanır. Bu değer her *değer-kümesi*ndeki etiketlerin sabitleri ile tek tek karşılaştırılır. İçindeki sabit, kontrol deyiminin değerine eşit olan ilk değer-kümesi bulunana kadar bu karşılaştırmaya devam edilir. Eşitlik bulununca o `case` etiketini takip eden ifade(ler) ilk `break` ifadesine gelininceye kadar çalıştırılırlar. Bulunan ilk `break` ifadesinden sonra `switch` ifadesinin geri kalan kısmı çalıştırılmadan atlanır. Hiç `break` ifadesi yoksa çalıştırma işlemi `switch`-ifadesinin sonuna erişene kadar devam eder.

Eğer değer kümelerindeki bütün etiket sabitleri kontrol deyiminin değerinden farklıysa, `default` kısmı seçilir ve bu kısımdaki ifadeler çalıştırılır. `default` kısmı yoksa bütün`switch` ifadesinin gövdesi hiç çalıştırılmadan atlanır.

Örnek:

```
switch (x) {
    case 1: printf("x in degeri 1 \n");
            break;
    case 2: printf("x in degeri 2 \n")
    case 3:
    case 4: printf("x in degeri 3 ya da 4 \n")
            break;
    default: printf("x in degeri 1,2,3,4 degil \n");
}
```

Eğer `x`'in değeri 1 ise, ilk `case` etiketindeki ifadeler çalıştırılır. İlk `printf` işlevi çağırılır ve sonra `break` ifadesi çalıştırıldığında `switch`-ifadesinin çalışması bitmiş demektir. Böylece `switch`-ifadesinden sonraki ifadeye geçilir. Ekranda şunları görürüz:

```
x in degeri 1
```

Eğer `x`'in değeri 2 ise, ikinci `case` etiketindeki ifadelere gidilir. İlk önce `x`'in değerinin 2 olduğunu yazan olan `printf` işlevi çalışır. Bu etiketin altında hiç bir `break`-ifadesi olmadığı için üçüncü değer-kümesindeki ifadeler de çalıştırılmaya başlanır. Böylece ekrana `x`'in değerinin 3 ya da 4 olduğunu yazacak `printf` işlevi çalıştırılır. Daha sonra `break`-ifadesi çalıştırıldığında `switch`-ifadesinden çıkılır. Ekranda şunları görürüz:

```
x in degeri 2
x in degeri 3 ya da 4
```

Eğer `x`'in değeri 3 ise, üçüncü değer kümesine gidilir. Bu kümede iki `case` etiketi vardır. Buradaki ifadeler çalıştığında İlk önce `x`'in değerinin 3 ya da 4 olduğunu yazacak olan`printf` komutu çalışır. Daha sonra `break`-ifadesi çalıştırıldığında `switch`-ifadesinden çıkılır. Ekranda şunları görürüz:

```
x in degeri 3 ya da 4
```

Eğer `x`'in değeri 1,2,3 ya da 4 değilse, `default kısmına` gidilir ve oradaki ifade çalıştırılır. `switch`-ifadesinin sonuna eriştiğimizden `switch`-ifadesinden çıkarız. Ekranda şunları görürüz:

```
x in degeri 1,2,3,4 degil
```

4. Döngü Yapıları

Amaç:
Bu hafta döngü yapıları tartışılacaktır. Döngü yapılarını kullanarak bir ifade kümesi birden fazla kere tekrar ettirilebilir.

1. İlk olarak temel bir döngü yapısı olan `while` ifadesi incelenecektir.
2. Değişik döngü türlerinin `while` ifadesi ile nasıl gösterileceği incelenecektir.
3. Diğer döngü yapıları olan `for` ve `do-while` ifadeleri tanıtılacaktır.
4. İç-içe döngülerin nasıl kurulduğu anlatılacaktır.

4.1. while İfadesi

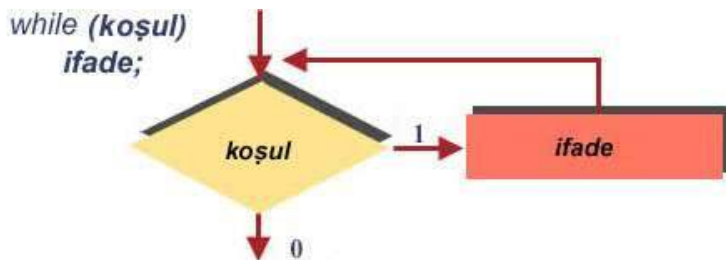
Bir ifade kümesinin tekrarlanması işlemi *döngü* (loop) olarak bilinir. Döngüler C dilinde birkaç değişik ifade ile yaratılabilir. Bunlardan en temeli `while`-ifadesidir. Diğer ifadelerle yaratılabilen bir döngü, `while`-ifadesi kullanılarak da yaratılabilir.

`while` ifadesinin yapısı şöyledir:

```
while ( koşul )
    ifade
```

Burada *koşul* bir mantıksal deyimdir ve döngünün gövdesini oluşturan *ifade* C dilindeki herhangi bir ifade olabilir. Koşulun değeri sıfır olmadığı sürece (yani koşul doğru olduğu sürece) döngünün gövdesi tekrarlanır ve koşulun değeri 0 (yani yanlış) olduğunda tekrarlama işlemi sona erecektir. Eğer `while`-ifadesine başlamadan önce koşulun değeri 0 ise, döngünün gövdesini oluşturan ifade hiç çalıştırılmayacaktır. Eğer koşulun değeri sıfırdan farklı ise ifade bir kere çalıştırıldıktan sonra, `while`'ın başına geri dönülür ve koşulun değeri tekrar hesaplanır. Döngünün tekrar çalıştırılıp çalıştırılmayacağı yine koşulun değerine bağlı olacaktır. Normal olarak döngü gövdesinde yaptığımız işlemlerin sonucuna göre, döngü gövdesi belirli bir miktarda tekrar edildikten sonra koşulun değeri 0 olur ve döngüden çıkılır. Eğer koşulun değeri hiç bir zaman 0 olmazsa, döngü sonsuz kere tekrar edilecektir ve bu olay *sonsuz döngü* (infinite loop) olarak bilinir. Döngünün gövdesi herhangi bir C ifadesi olabilir, ama genelde döngülerin gövdesi bir öbek ifade olacaktır.

`while` ifadesinin çalışma mantığı aşağıdaki grafiksel akış şemasından da anlaşılabilir.



Şimdi `while` ifadesinin kullanışını basit örnek program parçaları ile anlatacağız.

Örnek 1.

```
x = 1;
while (x < 10) {
   printf("%d \n",x);
   x = x + 1;
}
```

Bu örnek, bir atama ifadesi ve bir `while`-ifadesinden oluşur. `while`-ifadesinin koşulu mantıksal deyim `x<10` ve gövdesi iki ifadeyi kapsayan bir öbek ifadedir. İlk atama ifadesiyle`x` değişkenine 1 değeri konduktan sonra, `while`-ifadesi şöyle çalışacaktır:

- 1 değeri, 10'dan küçük olduğu için, koşulun değeri 1 olacaktır. Böylece `while`-ifadesinin gövdesini oluşturan öbek ifade çalıştırılır. Bunun sonucu olarak x değişkenin değeri ekrana yazılır ve x değişkenin değeri bir artırılır (yani değeri 2 olacaktır).
- Bundan sonra yine `while`-ifadesinin başına gidilir ve koşulun değeri yeniden hesaplanır. Burada da, 2 değeri 10'dan küçük olduğu için koşulun değeri yine 1 olacaktır. Bu yüzden gövde yeniden çalıştırılacaktır.
- Bu `while`-ifadesi 9 kere `x`'in 1'den 9'a kadar olan değerleri için tekrar edilir. Böylece ekranda, 1'den 9'a kadar değerleri görürüz. `x`'in değeri 10 olduğunda, koşulun değeri 0 olacağı için, `while` ifadesinin gövdesi bir daha çalıştırılmaz ve kontrol `while`-ifadesinden sonraki ifadeye geçer.

Örnek 2.

```
x = 1;
while (x != 10) {
   printf("%d \n",x);
   x = x + 2;
}
```

Bu örnekteki `while` ifadesi sonsuz döngüye bir örnektir. Burada amacımız 10 dan küçük tek sayıları yazdırmaktı, ama koşulumuzu yanlış seçtiğimizden dolayı sonsuz bir döngüye neden oluyoruz. Bu `while` ifadesinin koşulu mantıksal deyim `x != 10` (x 10 dan farklı ise) ve gövdesi iki ifadeyi kapsayan bir öbek ifadedir. İlk atama ifadesiyle `x` değişkenine 1 değeri konduktan sonra, `while`-ifadesi şöyle çalışacaktır:

- 1 değeri, 10'dan farklı olduğu için, koşulun değeri 1 olacaktır. Böylece `while`-ifadesinin gövdesini oluşturan öbek ifade çalıştırılır. Bunun sonucu olarak x değişkenin değeri ekrana yazılır ve x değişkenin değeri 2 artırılır (yani değeri 3

olacaktır).

- Bundan sonra yeniden `while`-ifadesinin başına gidilir ve koşulun değeri hesaplanır. Burada da, 3 değeri 10'a eşit olmadığı için koşulun değeri yine 1 olacaktır. Bu yüzden gövde yeniden çalıştırılacaktır.
- Bu `while`-ifadesinde `x`'in değeri hiç bir zaman 10 olmayacağından, `x`'in alacağı bütün değerler için bu döngü tekrarlanacaktır. Sonuç olarak çıktı olarak ekranda 1 3 5 7 9 11 13 15 ...... olarak bütün tek sayılar basılacak, bu döngü sonsuza kadar tekrar edecektir. Yani `while` ifadesinden sonraki ifadeye hiç geçilemeyecektir.

Burada sonsuz döngüden kurtulmak için şartı `x<10` olarak değiştirmek yeterli olacaktır.

Örnek 3.

```
/* x ve y değişkenleri 0 dan büyük iki pozitif tamsayı
   tutuyor olsunlar */

while ((x > 0) && (y > 0)) {
   printf("%d \n",x * y);
   x = x - 2;
   y = y - 3;
}
```

Bu örnekteki `while` ifadesin koşulu karmaşık bir mantıksal deyimdir. Bu döngü `x` ve `y` nin değeri 0 dan büyük olduğu sürece tekrarlanacaktır. Herhangi birinin (veya ikisinin de) değeri 0 dan büyük olmadığı bir anda tekrarlama işlemi duracaktır. Bu `while`-ifadesinden önce `x`'in değerinin 7 ve `y` nin değerinin 5 olduğunu varsayalım. Bu durumda `while`-ifadesi şöyle çalışacaktır:

- 7 ve 5 değerleri 0 dan büyük olduğundan, koşulun değeri 1 olacaktır. Böylece `while`-ifadesinin gövdesini oluşturan öbek ifade çalıştırılır. Bunun sonucu olarak 7 ile 5 çarpımı olan 35 değeri ekrana yazılır, ve x'in değeri 2 azaltılarak 5 ve y'nin değeri 3 azaltılarak 2 olur.
- Bundan sonra yeniden `while`-ifadesinin başına gidilir ve koşulun değeri tekrar hesaplanır. Burada da, 5 ve 2 değerleri 0 dan büyük olduğu için koşulun değeri yine 1 olacaktır. Bu yüzden gövde tekrar çalıştırılacaktır. Böylece 5 ile 2 nin çarpımı olan 10 değeri ekrana yazılır, ve x'in değeri 2 azaltılarak 3 ve y'nin değeri 3 azaltılarak -1 olur.
- Bundan sonra yeniden `while`-ifadesinin başına gidilir ve koşulun değeri tekrar hesaplanır. Burada 3 değeri 0 dan büyük ama -1 değeri 0 dan büyük değildir. Böylece koşulun değeri 0 olur ve program döngüden çıkar, ve `while`-ifadesinden

sonraki ifade çalışmaya başlar.

4.1.1. Kontrol Değerli Döngüler (Sentinel Controlled Loops)

Bazı durumlarda döngünün kaç kere çalışması gerektiğini bilebiliriz. Ama bazen de döngünün kaç kere çalışacağını, o döngüyü yazarken bilemeyebiliriz. Örneğin bir sınavın sonuçlarını klavyeden okuyup bu sonuçların ortalamasını bulmak isteyelim. Sınavda kaç kişi olduğunu bilmediğimiz durumda döngümüzü nasıl kurmalıyız? Döngümüzü nasıl durdurabiliriz?

Bu amaçla bir kontrol değeri (sentinel value) kullanırız. Bir sınav notu olamayacak bir değeri (örneğin –1), kontrol değeri olarak seçeriz. Klavyeden normal olarak sınav notlarını girerken, en son değer olarak bu kontrol değerini gireriz. Bu kontrol değeri girildikten sonra, döngümüzün çalışmasını durdururuz. Bu işi yapacak C programının parçası aşağıdaki gibi olacaktır:

```
toplam = 0;     /* notlarin toplami */
ogrenciSayisi = 0;  /* ogrenci sayisi */

/* ilk notu oku */
printf("bir not giriniz (durdurmak icin -1): ");
scanf("%d",&not);


/* not -1'den farkli oldugu surece okumaya devam et */
while (not != -1) {
     toplam = toplam + not;     /* notu toplama ekle */
     ogrenciSayisi = ogrenciSayisi + 1;    /* Öğrenci Sayısını bir artır */
     /* diger notu oku */
     printf("bir not giriniz (durdurmak icin -1): ");
     scanf("%d",&not);
}
printf(" \n\nOrtalama: %f5.2 \n",(toplam/ogrenciSayisi));
```

Bu örnek program parçasında, ilk önce notların toplamını tutacak olan `toplam` değişkenin içine 0 değeri konur ve ilk not klavyeden okunarak `not` değişkenin içine saklanır. Bundan sonraki ifademiz `while`-ifadesidir. Bu `while`-ifadesi `not` değişkeninin içindeki değer –1 değerinden farklı olduğu sürece tekrarlanacaktır. Böylece `not` değişkeninin değeri –1'den farklı olduğunda `while`-ifadesinin gövdesini oluşturan öbek ifade çalıştırılır. Bu öbek ifadede, ilk önce `not` değişkeni `toplam` değişkenine eklenerek sonuç yine `toplam` değişkeni içinde saklanır. Bundan sonra klavyeden yeni bir not değeri okunarak `not` değişkeninin içinde saklanır ve tekrar `while`-ifadesinin başına dönülür. Bu

işlem klavyeden –1 değeri girilene kadar tekrar edilecektir ve –1 değeri girildiğinde tekrarlama işlemi durdurularak `while`-ifadesinden sonraki ifadeye geçilecektir. Bu durumda `toplam` değişkenin içinde –1 değeri hariç ondan önce okunan bütün değerlerin toplamı olacaktır. Bu toplam değer öğrenci sayısına bölünerek sınavın ortalaması `while`-ifadesinden sonraki `printf` komutuyla ekrana yazdırılır.

4.1.2. Sayaç Kontrollü Döngüler (Counter-Controlled Loops)

Bazı durumlarda bir döngüyü bir sayaç kullanarak tasarlayabiliriz. Böyle sayaç kontrollü bir döngüdeki temel öğeler şunlardır:

1. Bu tür döngünün bir sayacı olur ve bu sayaç bir değişken ile ifade edilir. Bu değişken döngü sayacı (loop counter) olarak bilinir.

2. Döngü sayacına bir atama ifadesiyle ilk değer verilir.

3. Döngü sayacının değeri, bir mantıksal deyim ile kontrol edilir.

4. Döngünün gövdesinin her çalışmasından sonra, döngü sayacının değeri bir artırma (ya da eksiltme) işlemi ile değiştirilir.

Örneğin aşağıdaki `while`-ifadeli program parçası sayaç kontrollü bir döngüdür.

döngü sayaç değişkeni
döngü sayaç değişkenine ilk değerinin verilmesi
döngü sayaç değişkeninin son değerinin kontrol edilmesi

```
sayac = 1;
toplam = 0;
while ( sayac <=10 ) {
  toplam = toplam + sayac;
  sayac = sayac + 1;
}
```

döngü sayaç değişkeninin değiştirilmesi

Bu sayaç kontrollü döngü, `sayac` değişkeninin 1'den 10'a kadar olan değerleri için 10 kere tekrar edilecektir. Burada `sayac` değişkenine ilk değer bir atama ifadesiyle verilmiştir. Döngünün her tekrarında `sayac` değişkeni bir artırılacaktır. Bu `sayac` değişkenin değeri 10'u geçtiğinde tekrarlama bitecektir. Bu döngüye ilk girildiğinde `sayac` değişkeninin değeri 1 olacaktır ve döngü gövdesinin en sonundaki atama ifadesiyle bu değer bir arttırılacaktır. Böylece ikinci defa girildiğinde `sayac` değişkenin değeri iki olacaktır. Bu döngü 1'den 10'a kadar olan sayıların toplamını bulmak için kullanılabilir. Bu döngüden çıkıldığında `toplam` değişkeni, 1'den 10'a kadar olan sayıların toplamını tutacaktır.

Bir önceki bölümde eğer öğrenci sayısını bilirsek, o program parçasını sayaç kontrollü bir döngü kullanarak tekrar yazabiliriz. Bu durumda klavyeden sınav notlarını girmeye başlamadan önce, ilk olarak oğrenci sayısını gireriz. Bu işi yapacak bir C programının parçası aşağıdaki gibi olacaktır:

```
toplam = 0; /* notlarin toplami */

/* öğrenci sayısını oku */
printf("Ogrenci Sayisi: ");
scanf("%d ",&ogrenciSayisi);

/* Öğrenci sayısı kadar notu klavyeden oku */
sayac = 1;
while (sayac <= ogrenciSayisi) {
    /* notu oku */
    printf("bir not giriniz: ");
    scanf("%d ",&not);
    toplam = toplam + not; /* notu toplama ekle */
    sayac = sayac + 1;   /* sayacı bir artır */
}
printf("Ortalama: %f5.2 \n", (toplam/ogrenciSayisi));
```

Bu örnek program parçasında, ilk önce notların toplamını tutacak olan `toplam` değişkenin içine 0 değeri konur ve öğrenci sayısı klavyeden okunarak `ogrenciSayisi` değişkeni içine saklanır. `sayac` değişkeni ilk öğrenciyi belirtmek amacıyla 1'e eşitlenir. Bundan sonraki ifademiz `while`-ifadesidir. Bu `while`-ifadesi `sayac<=ogrenciSayisi` koşulu doğru olduğu sürece tekrarlanacaktır. Döngünün gövdesinde okunan not toplam değere eklendikten sonra, `sayac` değişkeni 1 artırılarak döngü başına gidilir. Öğrenci sayısı kadar not okunduktan sonra `sayac` değişkenin değeri `ogrenciSayisi` değişkeni içindeki değerden bir fazla olacaktır. Bu da `sayac<=ogrenciSayisi` koşulunu yanlış yapacak ve döngüden çıkmamıza neden olacaktır. Son olarak da toplam değer öğrenci sayısına bölünerek sınavın ortalaması `while`-ifadesinden sonraki `printf` komutuyla ekrana yazdırılır.

4.1.3. Örnek Programlar

Program 1:

Bu program verilen artı bir tamsayının faktöryel değerini hesaplar ve sonucu ekrana yazdırır. Bu örnekte döngümüzün kaç kere dönmesi gerektiği girilen tamsayıya bağlı olduğu için sayaç kontrollü bir döngü kullanmak uygun olmuştur.

Faktoryel.cpp

```
/*
 * Bu program klavyeden girilen bir arti tamsayinin faktoryel
 * degerini hesaplar, ve sonucu ekrana yazar.
 */
```

```
#include <stdio.h>
int main(){
   int num,       /* okunan tamsayi */
       factVal,       /* sayinin faktoryel degeri */
       sayac;         /* sayac degiskeni */


   /* tamsayiyi oku */
   printf("Bir arti tamsayi girin: ");
   scanf("%d",&num);
   while (num <= 0) {
       printf("Bir arti tamsayi girin: ");
       scanf("%d",&num);
   }

   /* sayinin faktoryel degerini bul */
   sayac = 2;
   factVal = 1;
   while (sayac <= num) {
       factVal = factVal * sayac;
       sayac = sayac + 1;
   }

   /* sonucu yaz */
   printf("Girilen tamsayi: %d \n",num);
   printf("Sayinin faktoryel degeri: %d",factVal);
   return(0);

}
```

Program 2:

Bu program ise biraz daha karmaşık bir program. Bu örnekte bir dizi tamsayı klavyeden okunur ve okunan sayılar üzerinde belirli işlemler yapılır. Burada kaç tane sayı girileceği belli olmadığından girilecek son sayı 0 seçilerek dizinin sonu belirlenir. Kontrol değerli bir döngü yapısı bu örnek için uygun olduğu için o tür bir döngü yapısı kullanılmıştır.

Bu örnekte göreceğiniz gibi döngü gövdesi oldukça çok ifadeyi kapsayan bir öbek ifade olabilir. Bu öbek ifadenin içinde diğer ifadelere ek olarak iç-içe bir `if` ifadesi de kullanılmıştır.

PosNegCount.cpp

```
/*
 * Bu program klavyeden girilen tamsayilar arasında arti ve
```

```
 * eksi olanların sayisini bulur. Her tamsayi ayri bir satirdan girilir
 * ve  en sondaki sayi arti veya eksi sayi olarak
 * gormedigimiz 0'dir.
 *
 * Bu program ayni zamanda, girilen sayilar arasındaki ilk arti sayinin
ilk
 * girildigi sirayi ve ikinci girildigi sirayi bulur.
 *
 * Bu program girilen arti ve eksi sayilarin toplam sayisini da
 * bastirir.
 * Girilen arti sayilardan ilkinin degerini, ilk ve ikinci girildigi
siralari (eger
 * ikinci defa girilmediyse ikinci pozisyon icin 0) basar.
 */

#include <stdio.h>
#define sentinel  0    /* Dizideki son sayi */

int
main(){
   int anInt, /* okunan tamsayi */
   numOfPos,  /* pozitif sayilarin sayisi */
   numOfNeg,  /* negatif sayilarin sayisi */
   locOfFirstPos, /* ilk pozitif sayinin pozisyonu */
   secLocOfFirstPos, /* ilk pozitif sayinin ikinci pozisyonu */
   firstPos, /* ilk pozitif sayinin degeri */
   loc; /* okunan sayinin pozisyonu */

   /* degiskenlerin ilk degerlerini ver  */
   loc = 0; numOfPos = 0; numOfNeg = 0;
   firstPos = 0; locOfFirstPos = 0; secLocOfFirstPos = 0;

   /* Ilk tamsayiyi oku */
   printf("Tamsayi girin (bitirmek icin 0): ");
   scanf("%d",&anInt);

   /* negatif ve pozitif sayilarin sayisini bul.
      ayni zamanda ilk pozitif sayinin ilk ve ikinci pozisyonunu bul. */
   while (anInt != sentinel) {
      loc = loc + 1;
      if (anInt<0)  /* negatif sayi */
         numOfNeg = numOfNeg + 1;
      else {        /* pozitif sayi */
         numOfPos = numOfPos + 1;
         if (locOfFirstPos==0) {    /* ilk pozitif sayi */
            firstPos = anInt;
            locOfFirstPos = loc;
         }
         else if((secLocOfFirstPos==0) && (firstPos==anInt))
                  /* ilk pozitif sayinin ikinci gorulusu */
```

```
                    secLocOfFirstPos = loc;
        }
        /* tamsayiyi oku */
        printf("Tamsayi girin (bitirmek icin 0): ");
        scanf("%d",&anInt);
    }
    /* sonuclari yaz */
    printf("Negatif sayilarin sayisi: %d \n", numOfNeg);
    printf("Pozitif sayilarin sayisi: %d \n", numOfPos);
    if (locOfFirstPos!=0) {
        printf("Ilk pozitif sayinin degeri: %d \n", firstPos);
        printf("Ilk pozitif sayinin pozisyonu: %d \n ", locOfFirstPos);
        printf("Ilk pozitif sayinin ikinci pozisyonu: %d \n",
secLocOfFirstPos);
    }
    return(0);
}
```

4.2. for ifadesi

`for` ifadesi döngülerin kurulmasında kullanılan başka bir yapıdır. Bu döngü yapısı özellikle sayaç kontrollü döngülerin yazılımında `while`-ifadesinden daha uygundur. `for`-ifadesinin yapısı şöyledir:

`for (` *ilklendirme* `;` *döngü koşulu* `;` *güncelleme* `)`
ifade

Burada,

- *ilklendirme* bu sayaç kontrollü döngününün döngü sayacına ilk değerini verir. Bu ifade sadece bir kere döngüye girmeden önce çalıştırılır.
- *döngü koşulu* değeri her seferinde döngüye girmeden önce bulunur ve bu değer 1 (doğru) ise döngüye girilir ve *ifade* çalıştırılır; aksi halde döngüden çıkılır. Bu mantıksal deyimin değeri genelde sayaç değişkenin değerine bağlı olacaktır.
- *güncelleme* döngünün gövdesini oluşturan *ifade* ifadesinin çalıştırılmasından sonra çalıştırılacaktır. Görevi sayaç değişkenin değerini güncellemektir.
- *ifade* döngünün gövdesini oluşturan ifadedir. Herhangi bir C ifadesi olabilir

Bir `for`-ifadesi, her zaman bir `while`-ifadesi olarak aşağıdaki gibi yeniden yazılabilir:

```
ilklendirme ;
while ( döngü koşulu ) {
   ifade ;
   güncelleme ;
}
```

Daha önce verdiğimiz aşağıdaki `while`-ifadeli sayaç kontrollü döngü

```
   toplam = 0;
   sayac = 1;
   while (sayac <= 10) {
      toplam = toplam + sayac;
      sayac = sayac + 1;
   }
```

`for`-ifadesi kullanılarak şöyle yazılabilir:

```
toplam = 0;
for (sayac = 1; sayac <= 10; sayac = sayac + 1)
   toplam = toplam + sayac;
```

Bu `for`-ifadesinde, ilk önce,

`sayac = 1`

ifadesiyle `sayac` değişkenine ilk değer verilir. Bu ifade sadece bir kere çalıştırılacaktır. Bundan sonra döngü gövdesi çalıştırılmadan önce, `sayac` değişkeninin değeri

`sayac <= 10`

koşulu ile kontrol edilecektir. Bu koşulun değeri eğer 1 ise döngü gövdesini oluşturan

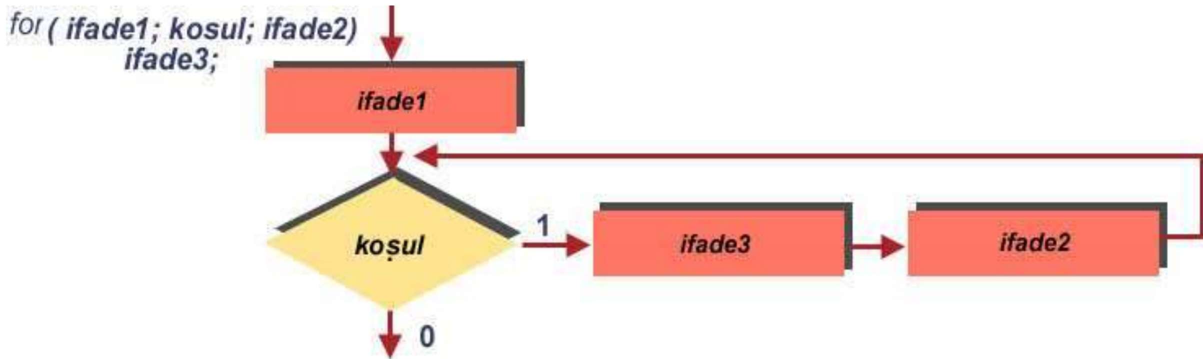
`toplam = toplam + sayac`

ifadesi çalıştırılacaktır. Gövdenin çalıştırılmasından sonra, `sayac` değişkeninin değerini değiştirmek için

`sayac = sayac + 1`

ifadesi çalıştırılacaktır. Bu işlemden sonra sayaç değerinin son değere gelip gelmediği tekrar kontrol edilir ve bu kontrolün sonucuna göre döngü tekrar edilir ya da döngüden çıkılır.

`for` ifadesinin çalışma mantığı aşağıdaki grafiksel akış şemasından da anlaşılabilir.



Şimdi `for` ifadesinin kullanılışını basit örnek program parçaları ile anlatalım.

Örnekler:

1. `for (i=1; i<=100; i=i+1)` *ifade* Burada döngünün gövdesi olan *ifade,* `i` değişkeninin 1'den 100'e kadar olan değerleri için 100 kere tekrar edilir. Gövdeye ilk girildiğinde `i` değişkenin değeri 1, ikinci girildiğinde 2, ... ve en son girildiğinde ise 100 olacaktır.
2. `for (i=100; i>=1; i=i-1)` *ifade* Burada döngünün gövdesi olan *ifade,* `i` değişkeninin 100'den 1'e kadar olan değerleri için 100 kere tekrar edilir. Gövdeye ilk girildiğinde `i` değişkenin değeri 100, ikinci girildiğinde 99, ... ve en son girildiğinde ise 1 olacaktır.
3. `for (i=2; i<=20; i=i+2)` *ifade* Burada döngünün gövdesi olan *ifade,* `i` değişkeninin 2 ve 20 arasındaki alacağı çift sayı değerleri için 10 kere tekrar edilir. Gövdeye ilk girildiğinde `i` değişkenin değeri 2, ikinci girildiğinde 4, ... ve en son girildiğinde ise 20 olacaktır.
4. `for (i=100; i>=10; i=i-10)` *ifade* Burada döngünün gövdesi olan *ifade,* `i` değişkeninin 100 ve 10 arasındaki 10'nun katları olarak alacağı değerler için 10 kere tekrar edilir. Gövdeye ilk girildiğinde `i` değişkenin değeri 100, ikinci girildiğinde 90, ... ve en son girildiğinde ise 10 olacaktır.

4.2.1 Arttırma ve Azaltma İşleçleri

Örneklerde de gördüğünüz gibi `for` ifadesiyde olsun, `while` ifadesiyle olsun sayaç kontrollu döngülerde sayaç değişkeninin değerini hep

```
sayac = sayac + 1;
```

ya da

```
sayac = sayac - 1;
```

gibi atama ifadeleri ile güncellememiz gerekiyor. Eğer sayacın değeri sadece 1 arttırılıyor ya da 1 azaltılıyorsa bu tip değiştirmeleri yapmak için C'deki arttırma (`++`) ve azaltma (`--`) işleçlerini kullanabiliriz. Her iki işleç de tek işlenenle çalışır. Yukarıdaki ilk atama ifadesini arttırma işleci ile

`sayac ++;` (*ya da* `++sayac;`)

şeklinde yazabiliriz. Bu ifade `sayac`ın değerini bir arttırır. İkinci atama ifadesini ise

`sayac --;` (*ya da* `--sayac;`)

şeklinde yazabiliriz. Bu ifade de `sayac`ın değerini bir azaltır. Buna göre yukarıda verdiğimiz ilk örnekteki for ifadesi de

```
for (i=1; i<=100; ++i)
```

şeklinde yazılabilir.

Arttırma ve azaltma işleçlerini kullanırken dikkat etmemiz gereken bir nokta vardır. Bu işleçlerin değişkenin önüne ya da arkasına yazılması bazı durumlarda farklı sonuçlar doğurabilir. Azaltma işlecinin kullanıldığı aşağıdaki iki ifadeyi inceleyelim:

`printf("%3d %3d \n", --n, n);`	`printf("%3d %3d \n", n--,n);`

Eğer `n` değişleninin değeri 5 ise ilk `printf` ifadesi ekrana

```
 4   4
```

değerlerini; ikinci `printf` ifadesi ise ekrana

```
 5   4
```

değerlerini yazacaktır. Eğer `--` işleci değişkenin önüne yazılırsa ilk önce değişkenin değeri bir azaltılır sonra bulunan değer kullanılır. Eğer `--` işleci değişkenin arkasına yazılırsa önce değişkenin değeri kullanılır sonra değeri bir azaltılır. Aynı durum `++` işleci için de sözkonusudur. Aşağıdaki iki atama ifadesini inceleyelim:

`j = ++i;`	`j = i++;`

Bu atama ifadelerinden önce `i`'nin değerinin 2 olduğunu varsayalım. Bu durumda ilk atama ifadesi önce `i`'yi bir arttırır, sonra değerini `j`'de saklar. Sonuç olarak `i`'nin değeri 3,`j'`ninki de 3 olur.
İkinci atama ifadesi ise önce `j`'ye `i'`nin değerini verir, sonra `i'`nin değerini bir arttırır. Sonuçta `i` 3, `j` 2 değerlerini saklarlar.

Eğer bir değişkenin değeri birden fazla arttırılacak ya da azaltılacaksa C'deki atama ifadelerini aşağıdaki gibi daha kısa bir şekilde yazabiliriz:

```
sayac += 10;            /* sayac = sayac + 10; */
```

ya da

```
sayac -= 5;            /* sayac = sayac - 5;  */
```

4.3. do-while ifadesi

`do-while` döngü yapısı `while`-ifadesine benzer. `while`-ifadesinde döngünün devam edip etmeyeceğini konrol eden mantıksal deyimin değeri döngü gövdesine girmeden kontrol edilir. Eğer değer hemen 0 (yanlış) olur ise, `while`-ifadesinin gövdesi hiç çalıştırılmadan döngüden hemen çıkılabilir. `do-while` ifadelerinde ise döngü koşulu döngü gövdesi çalıştırıldıktan sonra yapılır. Bu da, `do-while` ifadesinin gövdesinin en az bir kere çalıştırılacak olması demektir. Bu yüzden, `do-while` ifadesi gövdesi en azından bir kere çalıştırılması gereken döngü yapılarının tasarımında kullanılabilir.

`do-while` ifadesinin genel yapısı şöyledir:

`do`
  ifade

`while` (*koşul*) `;`

Bu yapıda gövdeyi oluşturan *ifade* çalıştırıldıktan sonra, bir mantıksal deyim olan *koşul*un değeri bulunur. Eğer değeri sıfırdan farklı (doğru) ise döngü yeniden çalıştırılır. Burada *ifade*herhangi bir C ifadesi olabilir, ama genelde bir öbek ifade olacaktır. Buradaki `do-while` ifadesinin sonundaki `while` anahtar sözcüğü, bu ifadenin sonunu belirtir. Programlarımızda bunu, `while`-ifadesinin başlangıcını gösteren `while` anahtar sözcüğüyle karıştırmamak gerekir.

Yukarıdaki gibi bir `do-while` ifadesi aşağıdaki gibi bir `while`-ifadesine eşit olacaktır:.

ifade
`while` (*koşul*)
   ifade `;`

Daha önceki örneğimiz, `do-while` ifadesi kullanılarak şöyle yazılabilir:

```
toplam = 0;
sayac = 1;
do {
   toplam = toplam + sayac;
   sayac = sayac + 1;
} while (sayac <= 10);
```

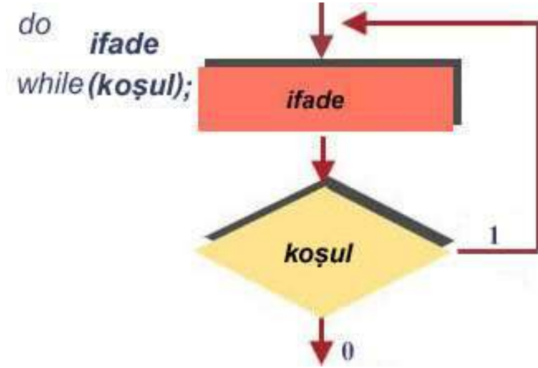
Çoğu durumda bir döngü tasarlanırken, üç döngü ifadesinden biri kullanılabilir. Diğer bir deyişle aynı işi üç değişik şekilde yapabiliriz. Ama her durum için, bu üç ifadeden birini kullanmak daha uygun olacaktır. Uygun ifadeyi kullanmak, programımızın daha kısa ve daha anlaşılır olmasını sağlayacaktır. Örneğin sayaç kontrollü döngüler için `for`-ifadesini kullanmak daha uygundur. Tabii bu demek değildir ki aynı işi `while`-ifadesiyle yapamayız. Önemli olan her durum için en uygun ifadeyi seçmektir. Yukardaki örnek Program 2'deki kontrol değerli döngüde, hem döngünün içinde hem de döngünün dışında klavyeden okuma yapmak zorunda kaldık. Bu nedenle kodumuz biraz uzun olmuştu. O örnek için `do-while` yapısı kullanmak daha uygun olabilir. Bu durumda kodumuz aşağıdaki gibi olacaktır:

```
toplam = 0; /* notlarin toplami */
ogrenciSayisi = 0;  /* Öğrenci Sayısı */

/* not -1 den farkli oldugu surece okumaya devam et */
do {
    /* notu oku */
    printf("bir not giriniz (durdurmak icin -1) ");
    scanf("%d",&not);
    /* notu toplama ekle */
    if (not != -1) {
       toplam = toplam + not;
       ogrenciSayisi = ogrenciSayisi + 1;
    }
} while (not != -1);
printf("Ortalama: %f5.2 \n ", (toplam/ogrenciSayisi));
```

Bu durumda notların okunmasını sadece bir noktadan (döngünün içinden) yaparız.

do-while ifadesinin çalışma mantığı aşağıdaki grafiksel akış şemasından da anlaşılabilir.



4.4. İç-İçe Döngüler (Nested Loops)

Eğer bir döngünün gövdesinde diğer bir döngü bulunuyorsa, bu tür döngüler iç-içe döngüler olarak bilinir. Bu durumda içteki döngü dıştaki döngünün her adımında yeniden çalıştırılacaktır. Örneğin,

```
deger = 0;
for (i=1; i<=10; i=i+1)
   for (j=1; j<=5; j=j+1)
      deger = deger + 1;
```

program parçasında iç-içe döngüler kullandık. Dıştaki döngü (kırmızı olan) bir `for`-ifadesi ve bu döngünün gövdesi diğer bir `for`-ifadesidir (yeşil olan). Dıştaki döngü 10 kere tekrarlanacaktır. İçteki döngü de, dıştaki döngünün her tekrar edilişinde 5 kere tekrar edilecektir. Böylece içteki döngünün gövdesini oluşturan atama ifadesi 50 kere tekrar edilmiş olacaktır. Bu yüzden bu `for`-ifadesinden çıktığımızda, `deger` değişkeni 50 değerini tutacaktır.

Aşağıdaki örnek, yukarıdaki örneğe benzemektedir, ama içteki döngünün tekrar ediliş sayısı her defasında farklı olacaktır:

```
deger = 0;
for (i=1; i<=10; i=i+1)
   for (j=1; j<=i; j=j+1)
      deger = deger + 1;
```

Burada dıştaki döngü 10 kere tekrarlanacaktır. İçteki döngü ise her seferinde `j`'nin 1'den `i`'nin o andaki değerine kadar olan değerleri için tekrar edilecektir. Bu durumda dıştaki döngünün ilk dönüşünde `i`'nin değeri 1 olacağından, içteki döngü sadece 1 kere tekrar edilecektir. Dıştaki döngünün ikinci tekrarında `i`'nin değeri 2 olacaktır ve içteki döngü 2 kere tekrar edilecektir. Bu demektir ki içteki döngü dıştaki döngünün her

seferinde `i` kere tekrar edilecektir. `i`'nin değeri 1'den 10'a kadar değişeceğinden, içteki döngünün gövdesini oluşturan atama ifadesi 1+2+3+...+10=55 kere tekrar edilecektir.

Şimdi iç-içe döngü yapılarına örnek olarak, basit bir program parçası vereceğiz.

Örnek.

Aşağıdaki gibi bir dik üçgeni ekrana yazdırmak istiyoruz.

```
*
**
***
****
*****
```

Bunun için for-ifadeli aşağıdaki gibi bir iç-içe döngü kullanabiliriz.

```
for (i=1; i<=5; ++i){
   for (j=1; j<=i; ++j)
       printf("*");
    printf("\n");
}
```

İstenen şekilde 5 satır yıldız vardır. İlk satırda bir yıldız, sonraki her satırda bir önceki satırdan bir fazla yıldız vardır. Dışardaki döngü 5 kere dönecektir, ve her dönüşte üçgenin bir satırı yazılacaktır. Birinci dönüşte tek yıldızlı birinci satır, ikinci dönüşte iki yıldızlı ikinci satır olmak üzere satırlar basılır. Döngünün içinde `i` değişkeninin değeri satır numarasını gösterecektir. Dış döngünün gövdesi diğer döngüyü ve `printf` işlevini çağıran ifadeyi içeren bir öbek ifadedir (başlangıç ve bitişi kırmızı renkle gösterilmiştir). İç döngü ise 1 den `i`değerine kadar gövdesini tekrarlar. Bu yüzden iç döngünün gövdesi ilk satırda bir kere, ikinci satırda iki kere tekrarlanarak çalışacaktır. İç döngünün gövdesinde çağırdığımız`printf` işlevi bir * karakteri yazar. Dış döngünün gövdesinde çağırdığımız `printf` işlevi bir sonraki satıra geçmemizi sağlar.

4.5. Örnek Programlar

Örnek 1.

İlk örnek program ekranda bir merdiven çizer. Bu işlemi yaparken, `for` ifadesi kullanarak yaratılmış sayaç kontrollü iç-içe döngüler kullanır.

Merdiven.cpp

```
/*
 * Bu program asagi dogru bir merdiven cizer.
 * Basamak sayisi programin girdisi olarak okunur.
```

```
 */

#include <stdio.h>

#define MAXBASAMAK 7    /* en fazla basamak miktari */

int
main()
{
   int  basamak_sayisi = 0;     /* basamak sayisi */
   int i,j,k,x,y;

   /* basamak sayisini oku */
   do {
      printf("Basamak sayisini girin > ");
      scanf("%d", &basamak_sayisi);
      if (basamak_sayisi > MAXBASAMAK) {
        printf("En fazla %d basamak cizilebilir \n",
MAXBASAMAK);
        basamak_sayisi = MAXBASAMAK;}
      else if (basamak_sayisi <= 0)
        printf("Sifirdan buyuk bir sayi girmeniz gerekiyordu.
\n");
   } while (basamak_sayisi <= 0);

   x=0;
   y=5;

   /* basmak sayisi kadar basamaklari ciz */
   for (i=1; i <= basamak_sayisi; ++i) {
      for (k=1; k<=x; ++k)
        printf(" ");
      x += 5;
      printf("******\n");
      for (j=1; j<=2; ++j){
         for (k=1; k<=y; ++k)
            printf(" ");
         printf("* \n");
      }
      y += 5;
   }
   return (0);
}
```

Örnek 2.

İkinci örnek program ekranda çarpım tablosunu yazar. Bu işlemi yaparken iç-içe döngüler kullanır.

CarpimTablosu.cpp

```
/*
 * Carpim tablosunu bastir
 */
#include <stdio.h>

#define MAXROW  15     /* maximum sira */
#define MAXCOL  10     /* maximum kolon */
#define CIZGI_UZUNLUGU MAXCOL*5+5

int
main()
{
   int row,col;  /* sira ve kolon sayaci */

   /* Basligi yazdir */
   printf("\n\n\n%35s\n", "CARPIM TABLOSU");

   /* Kolon basliklarini yazdir */
   printf("     ");
   for (col=1; col <= MAXCOL; ++col)
      printf("%5d", col);
   printf("\n");

   /* Kolon basliklarinin altina cizgi cek */
   for (col=1; col <= CIZGI_UZUNLUGU; ++col)
      printf("-");
   printf("\n");

   /* Carpim tablosundaki degerleri satir basliklariyla beraber
yazdir */
   for (row=1; row<=MAXROW; ++row) {
      printf("%2d | ", row);
      for (col=1; col<=MAXCOL; ++col)
         printf("%5d", row*col);
      printf("\n");
   }
   return(0);
}
```

5. İşlevler

Amaç:
Bu hafta işlevlerin yapısını inceleyeceğiz.

1. Bir işlevin nasıl tanımlandığını ve nasıl çağrıldığını öğreneceğiz.
2. İşlevlere parametre geçirme yöntemlerini tartışacağız ve işlevleri kullanarak modüler program geliştirme yöntemlerini anlatacağız.

5.1. Modüler Programlama

Büyük çapta bir programı tek parça olarak yazmak genelde çok zordur. Binlerce hatta onbinlerce satırlık bir program yazmanız gerektiğini düşünün. Bu programı tek bir işlev (`main`işlevi) olarak yazmak hemen hemen olanaksızdır. Burada yapmamız gereken, problemi alt-problemlere bölmek ve bu alt-problemleri ayrı ayrı çözmek olacaktır. Eğer bir alt-problem de karışıksa, onu daha da küçük alt-problemlere ayırmalıyız. Bu yöntem hatırlayacağınız gibi yukarıdan aşağı problem çözme tekniği (diğer adıyla böl-yönet) olarak bilinir.

C programlarımızı yazarken problemimizi parçalara ayırarak, her parçayı ayrı bir modül olarak yazarız. Genelde her modül C dilinde bir işleve denk düşecektir. Bir modülü birden fazla ve değişik amaçlar için de kullanabiliriz. C programlama dilinde bir program içinde main işlevi ile birlikte birçok işlev yer alabilir. Bu durumda her işlevi bir alt-problemi çözmek için yazılmış bir modül olarak görebiliriz. Böylece büyük çaptaki bir program her biri o programın bir modülünü ifade eden bir çok işlevden oluşacaktır. İşlevler genellikle algoritmamızın her ana basamağına karşılık gelirler. Biz bu hafta işlevlerin nasıl tanımlandığını ve onların nasıl kullanıldığını anlatacağız.

5.2. Kütüphane İşlevleri

C dilinin kütüphanesinde programlarımızda hazır olarak kullanabileceğimiz birçok işlev vardır. Bunların çoğu değişik matematiksel işlemleri yapmamıza yardımcı olacak işlevlerdir. Örneğin bir sayının karekökünü bulmak için yeni bir program yazacağımıza C'nin kütüphanesindeki `sqrt` işlevini çağırırız. Bu işlev aşağıdaki gibi çağırabilir:

```
y = sqrt(x);
```

Bu atama ifadesinde `sqrt` işlevini `x` değişkeni ile çağırıyoruz. Bu çağırma sırasında C kütüphanesindeki `sqrt` işlevi olarak yazılmış kod çalıştırılacak ve bulunan değer programımızın bu işlevi çağırdığı noktaya döndürülecektir. `x`'in değerinin 16.0 olduğunu varsayarsak `sqrt` işlevi 16'nın karekökü olarak 4.0 değerini bulacak ve bu değeri atama ifadesinde kullanacaktır. Buna göre `y'`nin değeri de 4.0 olacaktır.

Aşağıdaki tabloda C'de bulunan diğer bazı kütüphane işlevlerinden örnekler bulacaksınız. C kütüphanesinde bulunan bütün işlevlerin listesini görmek için kitabımızın Appendix B kısmına bakabilir ya da burayı tıklayabilirsiniz.

İşlev	Bulunduğu başlık	Amacı	Parametreleri	Sonuç
abs(x)	<stdlib.h>	tamsayının mutlak değerini bulur	int	int
fabs(x)	<math.h>	reel sayının mutlak değerini bulur	double	double
ceil(x)	<math.h>	x'den büyük en küçük tamsayıyı bulur	double	int
floor(x)	<math.h>	x'den küçük en büyük tamsayıyı bulur	double	int
pow(x,y)	<math.h>	x üzeri y (x^y) bulur	double,double	double

Bu tabloda ilk kolon işlevin adını, ikinci kolon işlevin bulunduğu kütüphane ismini, üçüncü kolon işlevin amacını, dördüncü kolon işlevi çağırırken kullandığımız değişkenlerin ya da sabitlerin veri türlerini ve son kolon da işlevin bulduğu sonucun veri türünü göstermektedir. Bu işlevleri programımızda kullanabilmek için bulundukları kütüphane ismini programımızın başındaki önişlemci bildirimleri ile eklemeliyiz `(#include <math.h> gibi)`

Örnekler:

1. $(x1 - x2)^2 + (y1 - y2)^2$ değerinin karekökünü `d` değişkeninde saklamak istediğimizi varsayalım. Bunun için aşağıdaki C program parçasını yazabiliriz.

```
a = x1 - x2;
b = y1 - y2;
c = pow(a,2) + pow(b,2);
d = sqrt(c);
```

Bunu yalnızca bir atama ifadesi ile de yapabilirdik:
d=sqrt(pow((x1-x2), 2) + pow((y1-y2), 2)) .

2. Bu örnekte

```
sqrt(floor(fabs(-16.8)))
```

ifadesinin değerinin nasıl hesaplandığına bakalım. İlk önce `fabs` işlevi -16.8 değeri ile çağırılır ve 16.8 bulunur. Sonra `floor` işlevi 16.8 ile çağırılır ve 16 değeri bulunur. En son olarak `sqrt` işlevi 16 ile çağırılır ve sonuç olarak 4.0 double değeri bulunur.

Örneklerden de görüleceği gibi C'deki matematiksel işlevler iç-içe yazılıp çağırılabilir.

5.3. İşlevler

C bu hazır işlevlerin yanısıra kendi işlevlerimizi tanımlamamıza da olanak sağlar. Böylece yazdığımız bazı program parçalarını işlev olarak tanımlayıp değişik yerlerde kullanabiliriz. Aynı zamanda modüler programlama tekniğini işlevler yardımıyla uygulayabiliriz.

Bu dersimizde kendi işlevlerimizi nasıl yazacağımızı öğreneceğiz. Aslında biz şu ana kadar C'deki işlevlerle karşılaştık. Örneğin programlarımızdaki `main` işlevi de kendi yazdığımız bir işlevdir. Bu işlev her C programında bulunması gereken bir işlevdir. C kütüphanesindeki bazı işlevleri de, işlev çağırma komutları ile çalıştırmıştık. Örneğin,

```
printf("abcdef");
```

`printf` işlevini çağırma komutudur. Bu komut çalıştığında,
denetim `printf` işlevine geçer ve o işlev çalışmaya başlar. O işlev çalışmasını bitirdikten sonra kontrol tekrar bu çağırma ifadesinden sonraki satıra geçer.

5.3.1. İşlevin Tanımlanması

Bir C işlevi aşağıdaki gibi tanımlanır:

işlev-türü işlev-adı (*biçimsel-parametre-listesi*) {

yerel-tanımlamalar
çalıştırılabilir-ifadeler

}

Burada

- ***işlev-adı*** bu işleve verdiğimiz bir isimdir. *işlev-adı* herhangi bir tanımlayıcı (identifier) olabilir.
- ***işlev-türü*** bu işlevin döndüreceği değerin veri türünü gösterir.
- ***biçimsel-parametre-listesi*** işlevin kaç tane parametre alacağını ve parametrelerin veri türlerini gösterir.
- ***yerel-tanımlamalar*** bu işlevin kullanacağı değişkenlerin tanımlanacağı yeri gösterir.
- ***çalıştırılabilir-ifadeler*** işlevde yapılması gereken işleri yapacak olan ifadelerdir.

Örneğin,

```
int ucuncuKuvvet ( int sayi ) {
int kup;
kup = sayi * sayi * sayi;
return kup;
}
```

verilen bir tamsayının kübünü almak için tanımlanmış bir işlevdir. Bu işlevin tanımlanmasındaki adımları aşağıdaki gibi özetleyebiliriz:

- İşlevin adını `ucuncuKuvvet` olarak seçtik. Bu işlevi çağırırken bu adı kullanacağız. Genelde bir işlev adını seçerken o işlevin fonksiyonunu yansıtan bir isim seçmek yerinde olacaktır. Bu işlev verilen sayının üçüncü kuvvetini bulacağından, adını bunu yansıtacak bir isim olan `ucuncuKuvvet` olarak seçtik.
- İşlevimizin döndüreceği değerin veri türü `int` olacaktır. Bu yüzden işlevin adının başına `int` yazdık.
- İşlevimiz tek parametre alacaktır ve bu parametrenin türü `int` olmalıdır. Bu işleve geçireceğimiz değere işlev içinde `sayi` değişkenini kullanarak erişeceğiz. Bu değişken, tanımladığımız işlevin biçimsel parametresi olarak bilinir.
- İşlev içinde `kup` adında yerel bir değişken tanımladık. Bu değişken yalnızca `ucuncuKuvvet` işlevi içinden erişilebilir. Bu işlevin dışından bu değere erişmek mümkün değildir.
- İşlevimizin ilk çalıştırılabilir ifadesi bir atama ifadesidir. Bu atama ifadesinde işleve parametre olarak geçirilen ve `sayi` değişkeni içinde bulunan `int` değerinin kübü bulunur ve `kup` değişkeni içinde saklanır.
- İşlevimizin ikinci çalıştırılabilir ifadesi `return` ifadesidir. Bu `return` ifadesi çalıştığında, `return` anahtar kelimesini izleyen deyimin (burada deyimimiz `kup`değişkenidir) değeri bu işlevin değeri olarak geri döndürülür. *Bir `return` ifadesindeki deyimin türü işlevin türü ile aynı olmak zorundadır.* Bu örneğimizde ikisinin de veri türü `int` veri türüdür.

5.3.2. İşlevin Türü ve `return` İfadesi

Bir işlevin türü, o işlevin çağırıldığı noktaya döndüreceği değerin veri türünü gösterir. Bir işlevin değeri onun içindeki bir `return`-ifadesi çalıştığında döndürülür. Bir `return`-ifadesi aşağıdaki yapıdadır:

`return` *deyim* `;`

Yani `return` özel sözcüğü bir deyim tarafından izlenir. Buradaki deyimin türü, içinde bulunduğu işlevin veri türü ile aynı olmak zorundadır.

Eğer bir işlev değer döndürmeyecekse, o işlevin türü `void` olarak tanımlanmalıdır. Bu durumda o işlev çağırıldığı noktaya bir değer

döndürmeyecektir. Eğer bir işlevin türü `void`olarak tanımlanmışsa, o işlevde `return`-ifadesi kullanmak zorunda değiliz. Ama bir işlevin veri türü `void` dışında başka herhangi bir veri türü ise, o işlev en az bir tane `return`-ifadesi içermelidir.

Veri türü `void` olarak tanımlanmış bir işlev, gövdesindeki son çalıştırılabilir ifadeden sonra sona erecek ve denetim o işlevin programda çağırıldığı noktadan sonraki ifadeye geri dönecektir. Bu tür işlevleri çağırma komutu ayrı bir ifade olarak tek başına kullanılır. Örneğin, çok kullandığımız `printf` işlevinin veri türü `void` olarak tanımlanmıştır ve o bir işlev çağırma ifadesiyle çağırılır.

Veri türü `void` olmayan bir işlev, gövdesindeki herhangi bir `return`-ifadesi çalıştığında sona erecektir ve o `return`-ifadesindeki deyimin değeri o işlevin değeri olarak işlevin çağırıldığı noktaya geri dönecektir. Bu tür işlevlerin çağırma komutları bir deyimin parçası olarak kullanılır. Örneğin, yukarıda tanımladığımız işlev başka bir işlevin içinden aşağıdaki gibi çağırılabilir:

```
n = ucuncuKuvvet(5);
```

Burada `ucuncuKuvvet(5)` çağırma komutu atama ifadesinin sağ tarafındaki deyim olarak kullanılmıştır. `ucuncuKuvvet` işlevi 5 değeri ile çağrılacak ve 125 değerini geri döndürecektir. Bu geri dönen değer de `n` değişkenin içinde saklanacaktır. Genelde veri türü `void` olmayan işlevler, bir deyimin içinden çağrılırlar. Yukarıdaki örnekte işlevin çağırılması atama ifadesinin sağındaki deyimin kendisidir.

5.3.3. Parametreler ve İşlevin Çağırılması

Bir işlev sıfır veya daha fazla parametre kabul edebilir. İşlevi tanımlarken kullandığımız biçimsel parametre listesi ile o işlevin kaç tane parametre alacağını, bu parametrelerin adlarını ve veri türlerini belirtiriz. Örneğin

```
int f(int x, double y, long z) {
...
}
```

olarak tanımlanan `f` işlevi 3 parametre alacaktır. İlk parametrenin adı `x` ve türü `int`; ikinci parametrenin adı `y` ve türü `double`; son parametrenin adı `z` ve türü `long` olacaktır.

Görüldüğü gibi, eğer bir işlev birden fazla parametre alacaksa, biçimsel parametre listesindeki parametreler virgül ile ayrılır. Her bir biçimsel parametre de, o parametrenin türünü gösteren özel sözcük ve onu izleyen adından oluşur.

Tanımlanan bir işlev, bir işlev çağırma komutuyla işleme sokulur. İşlev çağırma komutu aşağıdaki gibidir:

işlevin-adı (*asıl-parametre-listesi*)

Böylece işlev *asıl-parametre-listesi* içinde verilen para,etreler ile çağırılacaktır. Asıl parametrelerin her biri bir deyim olabilir ve bu deyimin türü o parametreye denk gelen biçimsel parametrenin türüyle aynı olmak zorundadır. Eğer listemizde birden fazla asıl parametre varsa, bu parametreler birbirlerinden virgül işareti ile ayrılır. Asıl parametre işleve geçirilirken, bellekte bu parametrenin bir kopyası yapılır ve bu parametreye denk gelen biçimsel parametrenin içine bu kopya konulur. Başka bir deyişle işlevi çağırırken kullandığımız değişkenlerin değeri çağırıldıkları pozisyona karşılık gelen biçimsel parametrelerin değeri olarak bellekte kopyalanır.

Örnekler:

1. Yukarıda tanımlanan `ucuncuKuvvet` işlevi aşağıdaki program parçasında olduğu gibi çağırılabilir.

```
:
x = 5;
y = ucuncuKuvvet(x);
z = ucuncuKuvvet(2);
:
```

Burada işlevimiz iki kere çağırılmıştır.

- İlkinde asıl parametremiz `x` değişkeninden oluşan bir deyimdir. Bu durumda `x` değişkeninin değeri olan 5, biçimsel parametre `sayi` içine kopyalanacaktır ve`ucuncuKuvvet` işlevi çalışmaya başlayacaktır. Bu işlev çalışmaya başladığında `sayi` biçimsel parametresinin içinde 5 değeri olacaktır. İşlevin ifadeleri çalıştıktan sonra, işlev içindeki `return`-ifadesi çalıştığında 125 değerini çağırılan noktaya geri döndürecektir. Bu değer `y` değişkeni içinde saklanacaktır.
- İkinci çağırılışta, asıl parametre bir tamsayı olan 2 değeridir. Yine bu tamsayının değeri `sayi` biçimsel parametresinin içine kopyalanacak ve işlev bu sefer bu değerle çalışacaktır. Geri döndürülecek olan 8 değeri `z` değişkenin içinde saklanacaktır.

Burada da görüldüğü gibi, bu işlev için asıl parametre türü `int` olan herhangi bir deyim olabilir.

2. Aşağıdaki gibi bir işlev tanımlayalım:

```
double kuvvet(double n, int m) {
/* yerel tanimlamalar */
int i;
```

```
double sonuc;
/* calistirilabilir ifadeler */
sonuc = 1.0;
for (i=1; i <= m; i=i+1)
   sonuc = sonuc * n;

return (sonuc);

}
```

Bu işlev verilen bir `n` sayısının (bir `double` değer) `m'`inci (pozitif bir `int` değer) kuvvetini bulur. `kuvvet` işlevi iki parametre alır: Birincisi bir `double` değer, ikincisi bir `int` değer olmalıdır. Bu işlev tarafından döndürülen değerin türü de `double` bir değerdir. İşlevde iki tane de yerel değişken (`i` ve `sonuc`) tanımlanmıştır. Verilen `n` değeri kendisiyle `m` kez bir`for`-ifadesi kullanılarak çarpılır. Bu `for`-ifadesinin bitiminde `sonuc` değişkeni `n'`nin `m`'ninci kuvvetini tutacaktır. Bu değer `return`-ifadesi ile geri döndürülür.

Bu işlev aşağıdaki gibi çağrılabilir:

```
double x,y;
x = kuvvet(3.0,2);
y = kuvvet(x,6);
```

Burada da görüldüğü gibi ilk gerçek parametrenin türü `double`, ikincisininki ise `int` olmak zorundadır.

5.4. İşlev Tanımlamalarına Örnekler

Bu kısımda şimdiye kadar anlattıklarımızı örneklerle daha da açıklayacağız. Örneklerimizi parametresiz ve parametreli işlevler olarak ikiye ayırdık.

5.4.1. Parametresiz İşlevler

Bazı işlevler hiç parametre almaz. Böyle bir işlevi

işlev_türü işlev_adı `(void) {`
`...`
`}`

şeklinde ya da hiç `void` yazmadan
işlev_türü işlev_adı `() {`
`...`
`}`
şeklinde tanımlayabiliriz. Örneğin

```
void kutu_ciz() {
   printf("**********\n");
   printf("*        *\n");
   printf("*        *\n");
```

```
    printf("*        *\n");
    printf("**********\n");
}
```

işlevi ekrana bir kutu çizer. Bu işlevi programımızın `main` işlevinden (ya da başka bir işlevinden)
`kutu_ciz ();`
şeklinde çağırırız. Bu işlevin veri türü `void` olduğu için çağırıldığı noktaya hiçbir değer döndürmeyecektir. O yüzden işlevin gövdesinde `return` ifadesi kullanmak zorunda değiliz.

Başka bir örnek:

```
int tamsayi_oku () {
    int sayi;
    printf("Bir tamsayi girin : ");
    scanf("%d" &sayi);
    return (sayi)
}
```

Bu işlev klavyeden bir sayı okur ve bunu çağırıldığı noktaya döndürür. Bu işlevi çağırırken döndüreceği değeri dikkate almamız gerekir. Örneğin,

```
yas = tamsayi_oku();
```

şeklinde bir atama ifadesinin içinde çağırabiliriz. (`yas` da `int` veri türünde olmalıdır)

5.4.2. Parametreli İşlevler

Parametrelerin kullanımı programlama açısından çok önemlidir. Parametreli bir işlev aynı işi değişik veriler üzerinde yapabilir. Parametreler bir işlev ile onu çağıran işlev arasındaki bilgi alışverişini sağlarlar. Bir işleve onu çağıran işlevden gönderilen değerleri tutan parametrelere *girdi parametreleri*; işlevin onu çağıran işleve döndürdüğü değerleri tutan parametrelere de *çıktı parametreleri* denir. Eğer işlevimiz onu çağıran programa sadece bir değer döndürecekse o zaman bu değeri ayrı bir parametre kullanmak yerine `return`ifadesiyle döndürebiliriz (şimdiye kadar yaptığımız gibi). Bir işlevin birden fazla çıktı parametresi olması gerekiyorsa bu durumda adresleme işleçlerini (* ve &) kullanmamız gerekir. Bu konuyu ileriki derslerimize bırakalım ve parametreli işlevlere örnekler verelim.

İlk örnek olarak daha önce verdiğimiz `kutu_ciz` işlevine benzer parametreli bir işlev yazalım. Bu işlev yine aynı boyda bir kutu çizecek ama kutunun ortasına işlevin girdi parametresi olarak verilen reel sayıyı da yazacak.

```
void sayili_kutu_ciz(double sayi) {
   printf("**********\n");
   printf("*        *\n");
   printf("* %6.2f *\n", sayi);
   printf("*        *\n");
   printf("**********\n");
}
```

Bu işlevin veri türü `void` olduğu için hiçbir değer döndürmez. Bu işlevin bir biçimsel parametresi vardır: `double sayi`. İşlevi

```
sayili_kutu_ciz(730.75);
```

şeklinde çağırabiliriz. Bu durumda işlevin biçimsel parametresi `sayi`'ya asıl parametre 730.75 değeri kopyalanacaktır ve ekranda kutu içinde 730.75 değeri görülecektir.

İkinci örneğimizdeki işlevin iki girdi parametresi ve bir çıktı parametresi var. Bu işlev verilen iki tamsayının karelerinin ortalamasını hesaplıyor. Bu işlevin tek çıktısı (hesaplanan`averaj`) olduğu için bu değeri `return` ifadesi ile çağıran işleve döndürüyor.

```
double karelerin_ortalaması (int x, int y) {
   double averaj;

   averaj = (pow(x,2)+pow(y,2))/2;
   return (averaj);
}
```

Bu işlevin veri türü `double` olduğu için döndürdüğü değer bir reel sayı olacaktır. Bu işlevi

```
a = 4;
b = 5;
ortalama = karelerin_ortalaması(a,b);
```

ile çağırdığımızı varsayalım. Asıl parametrelerin değeri 4 ve 5 olduğu için işlevin sonucu 20.5 olacak ve bu değer `ortalama` değişkeninde saklanacaktır. Burada asıl parametreler `a`ve `b`, sırasıyla, biçimsel parametreler `x` ve `y` ile eşleşmiştir. Başka bir deyişle `x`'e `a`'nın; `y`'ye `b`'nin değeri kopyalanmıştır.

Birden fazla parametreli bir işlev tanımlarken biçimsel ve asıl parametrelerin sayılarının eşit olmasına, eşleşecek parametrelerin sıralarının ve veri türlerinin aynı olmasına çok dikkat etmeliyiz.

Bu kısmı bitirmeden önce bir işlev çağırılıp çalışmaya başladığında bellekte neler olduğunu bilmemizde fayda var. Bir işlev her çağırılışında o işlev için

bellekte ayrı bir bölge ayrılır. Bu bellek bölgesinde o işlevin kullandığı veriler saklanır. İşlevin kullandığı veriler o işlevin yerel tanımlamalarında bulunan değişkenler ve biçimsel parametreleri için ayrılan yerlerdir. Asıl parametre değerlerinin biçimsel parametre değerlerine kopyalanması dediğimizde yapılan şey, işlev için ayrılan bu bellek bölgesine asıl parametre değerlerinin taşınmasıdır. *Bu bellek bölgelerinin içi işlevin çalışması biter bitmez silinir.* (yani işlev sona erdiğinde biçimsel parametrelerin değeri yok olur)

5.5. İşlevlerin Programdaki Yerleşimleri

Bir C programı, bir dizi işlevlerden ve değişkenlerden oluşur. Değişkenler gibi işlevler de program içinde kullanılmadan önce tanımlanmalıdır. İşlevleri tanımlamak için *işlev prototiplerini* `main` işlevinden önce programda yazarız. İşlev prototipinde işlevin veri türünü, ismini ve parametreleri ile ilgili bilgileri belirtiriz. Örneğin,

```
void kutu_ciz ();
```

daha önce tanımladığımız `kutu_ciz` işlevinin prototipidir. İşlevin tanımından farkı gövdesinin yerine sadece `';'` işareti konmuş olmasıdır. Prototipte sadece işlevin adı (`kutu_ciz`), veri türü (`void`) ve parametre listesi tanımlanır. Bu işlevin parametre listesi boş olduğundan bu işlev çağırılırken hiç parametre almayacaktır. Başka bir örnek :

```
double ortalama_bul (int x, int y);
```

Bu işlev prototipi de bu programda veri türü `double` olan `ortalama_bul` isimli ve `int` veri türünde iki parametresi olan bir işlev olduğunu belirtir.

İşlev prototipleri işlevin sadece başlığını tanımlar ama ne iş yaptığını tanımlamaz. İşlevin yaptığı işi daha önce anlattığımız gibi (bknz. 5.4.1) gövdesindeki yerel tanımlamalar ve çalıştırılabilir ifadelerle belirtiriz.

Şimdi işlev prototiplerinin, işlev tanımlamalarının ve diğer öğelerin tamamlanmış bir C programında nasıl yerleştirildiklerini inceleyelim. Genelde tamamlanmış bir C programın yapısı şöyledir:

önişlemci bildirimleri

işlev prototipleri

```
int
main ()
{
   ...
}
```

işlev-1

:

:
işlev-n

Önişlemci bildirimlerinden (`#include` ya da `#define` bildirimleri) sonra işlev prototipleri tanımlanır. Sonra `main` işlevi yer alır. Sonra da prototipi verilmiş bütün işlevlerin tanımı yapılır. `main` işlevinden sonra yapılan bu tanımlamaların sırası hiç önemli değildir. Bu işlevlerin hangi sırada çalıştırılacağını `main` işlevindeki çağırma sırası belirler. Bu genel yapıyı aşağıda verdiğimiz örnek programlarda inceleyebilirsiniz.

5.6. Örnek Programlar

Örnek 1.

İlk örnek programımız main işlevinin yanısıra üç işlev tanımlıyoruz.

`CombPerm.cpp`

```
/*
 * Bu program klavyeden girilen n ve r tamsayiları icin kombinasyon ve
 * permutasyon degerlerini hesaplar.
 *
 * Bu amacla comb, perm ve fact islevleri yazilmistir.
 */

#include <stdio.h>

/* islev prototipleri */
int fact(int n);
int comb(int n, int r);
int perm(int n, int r);

int
main()
{
  int n,r,combVal,permVal;
  /* n ve r degerlerini oku */
  printf("n degerini girin > ");
  scanf("%d", &n);
  printf("r degerini girin > ");
  scanf("%d", &r);
  /* Kombinasyon ve permutasyon degerlerini hesapla */
```

```
  combVal = comb(n,r);
  permVal = perm(n,r);
  /* Hesaplanan kombinasyon ve permutasyon degerlerini bastir */
  printf("Kombinasyon C(%d,%d): %d \n", n,r,combVal);
  printf("Permutasyon P(%d,%d): %d \n",n,r,permVal);

}

/* Bu islev verilen n ve r degerleri icin
 * C(n,r) = n! / (r! * (n-r)!)
 * formulunu kullarak kombinasyon degerini hesaplar.
 * Bunun icin fact (faktoryel) islevini cağırır.
 */

int comb(int n, int r) {
  return fact(n)/(fact(r)*fact(n-r));
}
/* Bu islev verilen n ve r degerleri icin
 * P(n,r) = n! / (n-r)!
 * formulunu kullarak permutasyon degerini hesaplar
 */
int perm(int n, int r) {
  return fact(n)/fact(n-r);
}
/* Bu islev verilen bir pozitif degerin faktoriyel degerini
 * hesaplar.
 */
int fact(int n) {
  int i,val;
  val = 1;
  for (i=1; i<=n; i=i+1);
    val = val*i;
  return val;

}
```

Bu programı çalıştırdığımızda ilk önce `main` işlevi çalışmaya başlar. Bu işlev klavyeden iki tamsayı (`n` ve `r`) okur; onların kombinasyon ve permutasyon değerlerini `comb` ve `perm`işlevlerini çağırarak bulur ve sonuçları ekrana yazdırır. Bu programda `main` işlevinin dışında üç işlev tanımladık:`comb`, `perm` ve `fact`. İlk ikisini `main` işlevi doğrudan çağırıyor; `fact`işlevi ise `comb` ve `perm` işlevleri çalıştığında çağırılıyor.

`main` işlevi `comb` işlevini çağırdığında denetim bu işleve geçer. `comb` işlevi de `fact` işlevini üç kez üç değişik değerle çağırır ve geri dönen değerleri kullanarak kombinasyon değerini hesaplayar ve `main` işlevine döndürür. Dönen değer `combVal` değişkeninde saklanır. `main` işlevinde bir sonraki atama ifadesinde `perm` işlevi çağırılır. Bu sefer denetim `perm` işlevine geçer. O

da `fact` işlevini iki kez iki farklı değerle çağırır. İlk çağırışın sonucunu ikinci çağırışın sonucuna böler ve bulunan değeri kendi sonucu olarak `main` işlevine döndürür. `main` işlevi dönen değeri `permVal` değişkeninde saklar ve bir sonraki ifadeyi (`printf`) çalıştırır.

Örnek 2.

İkinci örnek programımızda geçen haftaki dersimizde yazdığımız Merdiven.cpp programını işlevli hale getirdik. Programın bu hali ilkine göre daha okunabilir ve modüler oldu.

Basamaklar.cpp

```
/*
 * Bu program asagi dogru bir merdiven cizer.
 * Basamak sayisi programin girdisi olarak okunur.
 */

#include <stdio.h>

#define MAXBASAMAK 7    /* en fazla basamak miktari */

/* işlev prototipleri */
int basamak_oku();
void bir_basamak_ciz(int x);

int
main()
{
   int  basamak_sayisi;     /* basamak sayisi */
   int i, bosluk;

   /* basamak sayisini oku */
   basamak_sayisi = basamak_oku();

   bosluk = 0;
   /* basamak sayisi kadar basamak ciz */
   for (i=1; i <= basamak_sayisi; ++i){
      bir_basamak_ciz(bosluk);
      bosluk = bosluk + 5;
   }
   return(0);
}

int basamak_oku()
```

```
{
   int bs;

   do {
      printf("Basamak sayisini girin > ");
      scanf("%d", &bs);
      if (bs > MAXBASAMAK) {
        printf("En fazla %d basamak cizilebilir \n", MAXBASAMAK);
        bs = MAXBASAMAK;}
      else if (bs <= 0)
        printf("Sifirdan buyuk bir sayi girmeniz gerekiyordu. \n");
   } while (bs <= 0);
   return(bs);
}

void bir_basamak_ciz(int x)
{
   int i, j, k;

   /* x kadar boşluk bırak */
   for (i=1; i<=x; ++i)
     printf(" ");

   /* basamağı çiz */
   printf("******\n");
   x += 5;
   for (j=1; j<=2; ++j){
      for (k=1; k<=x; ++k)
         printf(" ");
      printf("* \n");
   }
}
```

Bu program basamak sayısını `basamak_oku` işlevini çağırarak okur. Bu işlev okunan basamak sayısını `main` işlevine döndürür ve bu değer `basamak_sayisi` değişkeninin içinde saklanır. `bir_basamak_ciz` işlevi tek parametreli ve bir değer döndürmeyen bir işlevdir. Bu işlevin görevi ekrana yalnızca bir basamak çizmektir. Girdi parametresi bu basamağı çizmeden önce ne kadar boşluk bırakılacağını gösterir. Bu parametrenin asıl değeri `main` işlevinde bu işlevin her çağırılışından önce arttırılır. Böylece her basamak biraz daha sağa doğru çizilir. Sonuçta merdiven ortaya çıkar.

6. Diziler (Arrays)

Amaç:

C programlama dilinde dizilerin nasıl tanımlanacağı, nasıl kullanılacağı bu haftaki dersin konusudur.

1. Diziler üzerinde arama ve sıralama algoritmaları ve
2. Tek boyutlu dizilere ek olarak çok boyutlu diziler de tartışılacaktır.

6.1. Dizilerin Yapısı

Bir dizi, aynı veri türündeki değerlerin sıralanmış bir halidir. Diziler çok fazla miktarda değer tutabilecek bir bölgenin tek bir değişken adı ile tanımlanmasında kullanılır. Bu bölge çok sayıda aynı türden değer tutabilir ve o bölgedeki her değere yalnızca bir değişken adı kullanarak erişebiliriz. Her dizinin saklayacağı en fazla değer miktarı, o dizinin tanımlanması sırasında belirtilir. Bir dizinin saklayacağı en fazla değer miktarı, o dizinin uzunluğu olarak bilinir. Dizi içindeki her bir değere indis (indeks) kullanılarak erişilir.

Örneğin,

bir dizi : x	5	-2	8	12	22	92	3	7
	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]

`x` değişkeni 8 tane tamsayı tutan bir dizidir. Bu bölgenin her bir noktasına `x` değişkeni kullanılarak erişilebilir. `x` dizisindeki ilk elemana `x[0]` ile erişilebilir. Burada köşeli parantezler içindeki tamsayı o elemanın indisidir. İlk değerin indisi 0, ikinci değerin indisi 1'dir. Bir dizide en fazla n değer tutulabiliyorsa, o dizinin elemanlarına erişmek için kullanılan indis değerleri 0 ile n-1 arasındadır. İlk değerin indisi 0 ve en son değerin indisi ise n-1 dir.

6.1.1. Dizi Elemanlarına Erişilmesi

dizi-adı `[` *indis* `]`	burada *indis*, bu dizi için geçerli bir indis değeri olacak olan bir tamsayı deyimidir.

Örnekler:

`x[0] = 22;`	x dizisinin ilk elemanının (0 indisi ile gösterilen) içine 22 değerini koyar.
`i = 5;`	`i+1` deyiminin değeri olan 6 indisi ile gösterilen (yani 7. eleman) içine 33

`x[i+1] = 33;`	değerini koyar.
`x[i] = x[i+1];`	`i=5` iken 6 indisi ile gösterilen (7.) elemanın içinde saklanan değeri 5 indisi ile gösterilen (6.) elemanın içine kopyalar.

6.1.2. Dizi Uzunluğu

Her dizinin saklayabileceği eleman sayısının bir sınırı vardır. Bu miktar dizinin yaratılışı sırasında belirtilir.

6.1.3. Dizilerin Tanımlanması

veri-türü *dizi-adı*[*dizi-uzunlugu*];

tanımlaması ile bir dizi tanımlanır. Burada veri türü ile dizi adını köşeli parantezler içinde dizinin uzunluğu izler. Köşeli parantezler, tanımlanan değişkenin(dizi-adı) bir dizi olacağını gösterir. Örneğin,

```
int x[10];
double y[50];
```

tanımlamaları `x` ve `y` dizilerini tanımlar. `x` dizisinin her bir elemanının türü `int` ve uzunlugu 10; `y` dizisinin her bir elemanının türü `double` ve uzunlugu 50 olacaktır.

Bir dizi tanımlandığı zaman derleyici bellekte dizi uzunluğu kadar verilen veri türünde değer tutabilecek yer ayırır. Ayırılan bu bellek bölgesinin başlangıç adresi *dizi-adı* değişkenin içinde saklanır. Başka bir deyişle dizi adları aslında bildiğimiz anlamda birer değişken değil, dizilerin bellekteki başlangıç adreslerini gösteren sabitlerdir. Dizi icin ayırılan bellek bölgesinin her elemanına dizi adı (yani dizi başlangıç adresi) ve indis kullanarak erişebiliriz.Bu durumda indis dizinin bir elemanının dizinin başlangıç adresine goreceli olarak bellekteki adresini gösterir.

6.1.4. Dizilere İlk Değerlerinin Verilmesi

Bir dizi yukarıdaki gibi tanımlandığında, elemanlarının değerleri belirli değildir. Eğer bir dizinin elemanlarına tanımlama anında ilk değerlerini vermek istersek aşağıdaki yapıyı kullanabiliriz.

veri-türü dizi-adı `[] = {` degerler`}` `;`

Dizinin elemanlarının ilk değerlerini bu şekilde veriyorsak dizi uzunluğunu ayrıca belirtmemize gerek yoktur. Örneğin,

```
int y[] = { 1, 3, 7, 8 }
```

ifadesi ile dört elemanlı (indis aralığı 0 ile 3 olan) bir `int` dizi tanımlar ve bu dizinin elemanlarına listedeki sayıları ilk değerleri olarak veririz. Böylece

tanımlanan `y` değişkeni bu diziyi gösterir.

Örnekler:

1. Bir dizinin değerlerinin okunması:

```
int x[100];
for (i=0; i < 100; i=i+1) {
   printf("bir int degeri > ");
   scanf("%d",&n);
   x[i] = n;
}
```

Bu program parçası 100 tane tamsayıyı klavyeden okur ve bu okunan değerleri `x` dizisinin içinde saklar. İlk okunan değer 0 indisi ile gösterilen ilk dizi elemanının içinde saklanacaktır. Burada `x`' in uzunlugu 100 olduğundan, bu döngü `i` değişkeninin 0 ile 99 değerleri için tekrarlanacaktır.

2. Bir dizi içindeki değerlerin sola kaydırılması:

```
// ilk elemanın değerini kaybetmemek için başka bir değişkende
saklıyoruz:
temp = x[0];

for (i=0; i < 99; i=i+1)
   x[i] = x[i+1];
//sakladığımız ilk değer son elemanın değeri oluyor:
x[99] = temp;
```

Bu program parçası `x` dizisinin içindeki her değeri bir sol tarafa kaydırır. Bu program parçası çalıştıktan sonra, 1. eleman 2. elemanın değerini, 2.eleman 3. elemanın değerini tutacaktır. Son eleman da 1. elemanın değerini tutacaktır. `x` dizisinin uzunluğu 100 olduğu icin, buradaki döngü 99 kere tekrar edilecektir.

3. 50 elemanlı bir tamsayı dizisinde saklanan sayıların en büyüğünün yerinin bulunması:

```
loc = 0;
for (i=1; i<50; i=i+1)
   if (x[i] > x[loc])
      loc = i;
```

Burada bir tamsayı dizisini baştan sona tarayarak, en büyük değeri saklayan elemanın indisini buluruz. Bu program parçası çalıştıktan sonra `loc` değişkeni en büyük sayının saklandığı elemanın indisini tutacaktır. Döngüye başlamadan önce en büyük değerin 0. elemanda olduğunu varsayarak başlarız. Döngü içinde, eğer baktığımız değer, şimdiye

kadar bulduğumuz değerden daha büyükse en büyük değerin yerini gösteren `loc` değişkenini değiştiririz.

4. Verilen bir değerin 100 elemanlı bir dizi içindeki yerinin bulunması:

```
i = 0;
while ((i<100) && (x[i] != item))
   i = i + 1;

if (i == 100)
   loc = -1; // bulunamadı
else
   loc = i;  // bulundu
```

Bu program parçası, `item` değişkeni içinde tutulan değerin `x` dizisi içinde ilk bulunduğu yeri bulur. Bu amaçla `x` dizisinin her elemanı 0. elemandan başlanmak üzere`item` içindeki değerle karşılaştırılır. Bu değer dizi içinde bulunana kadar ya da dizinin sonu erişilene kadar dizi içindeki her değerle teker teker karşılaştırılır. Döngüden çıktığımızda, eğer `i`'nin değeri, `x` dizisindeki eleman sayısına eşit ise, aranan değer dizi içinde bulunamamış demektir. Bu yüzden `loc` değişkeninin içine bunu göstermek üzere -1 değerini koyarız. Aksi halde bulduğumuz yeri `loc` değişkeni içinde saklarız.

5. Verilen 50 elemanlı bir tamsayı dizisindeki bütün sayıların toplamının bulunması:

```
sum = 0;
for (i=0; i<50; i=i+1)
   sum = sum + x[i];
```

6.2 Dizilerin İşlevlere Parametre Olarak Geçirilmesi

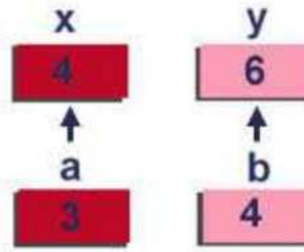
call-by-value:

- Normal olarak basit veri türlerindeki değerler (`int, short, double` gibi) işlevlere parametre olarak geçirildiğinde *call-by-value* olarak bilinen parametre geçirme yöntemi kullanılır.
- Bu durumda asıl parametrenin değeri biçimsel parametrenin içine kopyalanacaktır.
- Eğer kullandığımız asıl parametre bir değişken ise, bu değişkenin değeri işlev içinde değiştirilemez.

Bu şimdiye kadar gördüğümüz parametre geçirme yöntemiydi. Örneğin,

```
void m 1 (int x, double y) {
  x = x + 1;
  y = y + 2;
}

// main içinde
   int a = 3;
double b = 4;
   m1 (a, b);
```

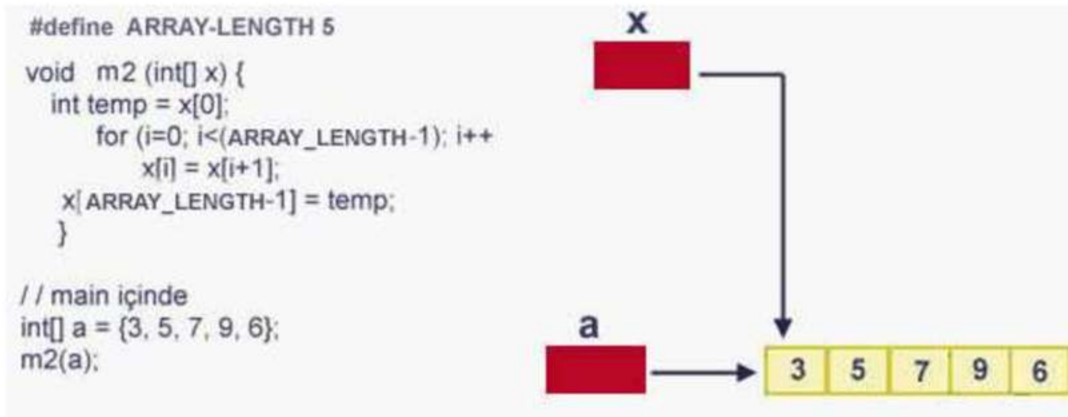


bu program parçası çalıştığında, `main` içinde tanımlanan `a` ve `b` değişkenlerinin değerleri, `m1` işlevi çağrıldığında `x` ve `y` biçimsel parametrelerinin içine kopyalanır. `m1` işlevinin içinde`x` ve `y` değişkenlerinin değerlerini değiştirdiğimizde, `main`'deki `a` ve `b` değişkenlerinin değerleri değişmez. Değişen yalnızca `m1` işlevi içinde, `x` ve `y` biçimsel parametreleri ile gösterilen bellek bölgeleridir. `m1` işlevinin sonunda `x` ve `y` değişkenleri 4 ve 6 değerlerini tutacaktır ve `m1` işlevinden `main`'e geri dönüldüğünde `a` ve `b` değişkenleri hala 3 ve 4 değerlerini tutacaktır.

call-by-reference:

- Diziler *call-by-reference* diye bilinen bir parametre geçirme yöntemiyle işlevlere parametre olarak geçirilir.
- Bir dizi bir işleve parametre olarak geçirilirken, o dizideki elemanlar işlevin biçimsel parametresinin içine kopyalanmaz. Gerçekte bu diziyi gösteren bir işaretçi (referans) işleve geçirilir. Bu işaretçi kullanılarak işleve parametre olarak geçirilen dizinin elemanlarına erişilir.
- Bu yüzden parametre olarak geçirilen bir dizinin elamanları, işlev içinde değiştirilebilir.

Örneğin,



program parçasında, main işlevi içinde 5 elemanlı bir a dizisi yaratılır ve bu dizi m2 işlevine parametre olarak geçirilir. Gerçekte m2 işlevine parametre olarak geçirilen değer bu diziye bir işaretçidir (dizinin başlangıç adresi)
. m2 işlevindeki x ve main işlevindeki a değişkenleri aynı dizinin bellekteki adresini tutacaklardır. Bu dizinin elemanları her iki değişkeni kullanarak da değiştirilebilir. m2 işlevinde x dizisinin elemanlarının değerlerini değiştirdiğimizde, gerçekte main işlevindeki a dizisinin elemanlarının değerlerini de değiştiriyoruz demektir. m2 işlevinden geri dönüldüğünde, a dizisinin içindeki değerler şöyle olacaktır:

5	7	9	6	3

NOT: Diziler işlevlere parametre olarak geçirildiğinde biçimsel parametre ile asıl parametre yazılış şeklinde dikkat ediniz. Bir biçimsel parametrenin dizi olduğunu dizi adının yanına '[]'koyarak belirtiyoruz. Dizinin uzunluğunu vermiyoruz. İşlevi çağırırken asıl parametre olan dizinin ise yalnızca adını veriyoruz.

Örnek:
Asal Sayıların Bulunması

```
// İlk 100 asal sayıyı bul ve bastır
//
#include<stdio.h>
void findprimes(int p[], int n);
int isprime(int v, int p[], int lastprimeloc);
int main() {
      int i;
      int primes[100];
      findprimes(primes, 100);
      printf("Ilk Yuz Asal Sayi: \n");

      for ( i=0; i<100; i=i+1)
          printf("%d \n", primes[i]);
      return 0;


}

// Bu işlev ilk n asal sayıyı bulacak, ve bu bulunan asal
// sayılar ilk biçimsel parametreye karşılık gelen dizi içinde
// saklanacaktır.
```

```
void findprimes(int p[], int n) {

      int val, i;
      p[0] = 2; // ilk asal sayı
      val = 3;
      i = 1;
      while (i<n) {
            if (isprime(val, p, i)) {
                  p[i] = val;
                  i = i + 1;
            }
            val = val + 1;
      }
}
// verilen v değerinin asal sayı olup olmadığını, daha önceden bulunan
// ve parametre olarak gecen p dizisinde bulunan asal sayılari kullanarak
// karar verir. Bir sayı asal sayı değilse, kendinden küçük bir asal sayıya
// bölünebilinir. Bu işlevde verilen v değerinin kendisinden küçük asal
// sayılara bölünüp bölünmediğini kontrol eder. Eğer bölünüyorsa 0 (yanlış);
//  bölünmüyorsa 1 (doğru) değerini döndürür. 1 döndürmesi v değerinin
// asal sayı olduğunu gösterir.
int isprime(int v, int p[], int lastprimeloc) {
      int primeflag = 1;
      int i = 0;
      while (primeflag && (i<lastprimeloc))
            if (v%p[i] == 0)
                  primeflag = 0;
            else
                  i = i + 1;
      return primeflag;
}
```

Burada `main` işlevinde 100 elemanlı bir dizi tanımlanır. Bu dizi ve 100 sayısı `findprimes` işlevine parametre olarak geçirilir. `findprimes` işlevi içinde asal sayılar bulundukça bu işlevdeki `p` dizisine (dolayısıyla `main` işlevinde ona karşılık `primes` dizisine) yerleştirilecektir.

6.3. Sıralama

Bu bölümde dizilerin yaygın olarak kullanıldığı bir uygulama alanı olan sıralama ile ilgili bir örnek vereceğiz. Sıralama algoritmaların amacı verilen bir listedeki elemanları küçükten büyüğe (ya da büyükten küçüğe) sıralamaktır. Yani bir sıralama algoritmasının girdisi bir dizi, çıktısı ise onun sıralı hali olacaktır. Örneğin

9 7 3 1 2 5 dizisinin sıralanmış hali ==> 1 2 3 5 7 9

Soldaki değerleri kapsayan bir dizi verilirse, sıralama algoritması sağdaki diziyi üretecektir.

Çok değişik sıralama algoritmaları vardır. Biz burada yalnızca seçerek-sıralama (`selection-sort`) algoritması için bir işlev vereceğiz ve onun nasıl çalıştığını açıklayacağız.

Bir tamsayı dizisi olduğunu varsayalım ve bu dizinin içindeki eleman sayısı da `n` olsun. Bu `n` sayısı dizisinin uzunluğu da olabilir ya da dizi uzunluğundan küçük bir sayı da olabilir. Bu durumda dizinin yalnızca ilk `n` elemanı bir değer tutuyor demektir. Aşağıdaki `selectionSort` işlevi bir `int` dizisi ve içindeki eleman sayısını gösteren bir `n` değeri alır. Bu`selectionSort` işlevinin çalışması bittiğinde dizi sıralanmış olacaktır:

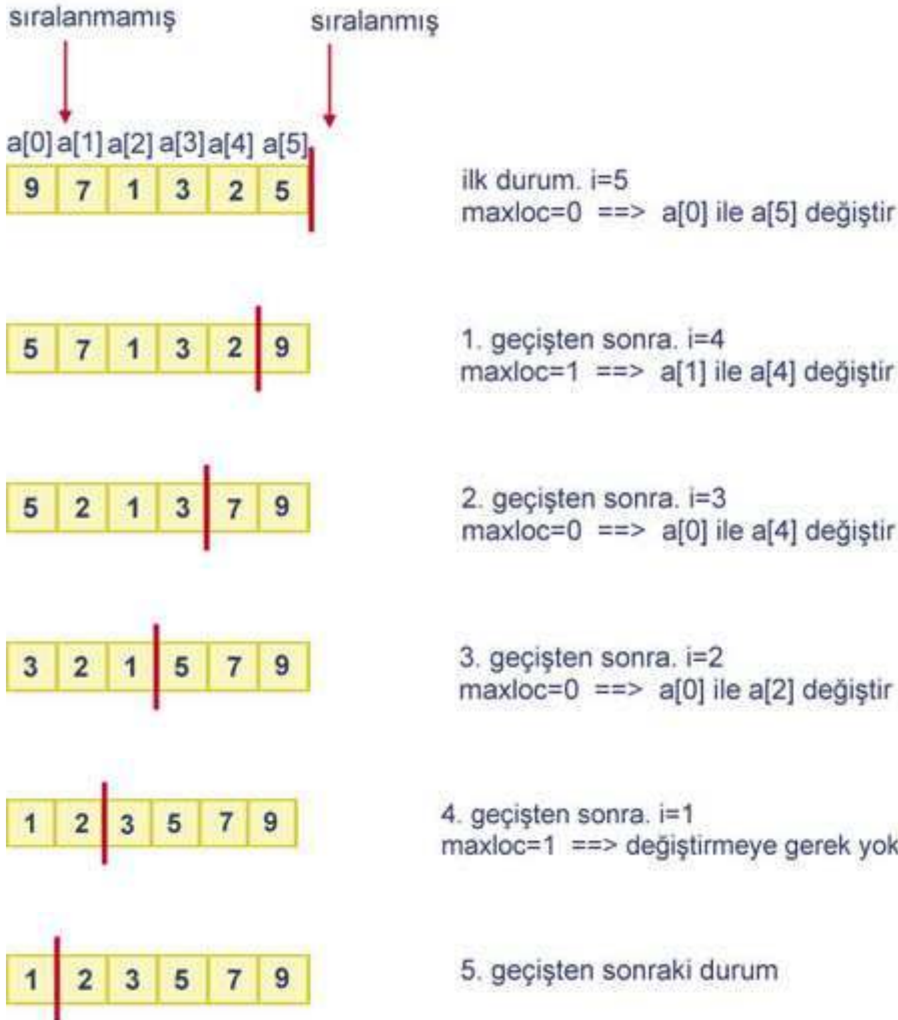
```
void selectionSort(int a[], int n) {
      int i,j,temp,maxloc;
      // Dizi iki bölgeye ayrılır: sıralanmamış ve sıralanmış.
      // Sıralanmış bölge i+1 ile n-1 arasındaki bölgedir ve
      // dizideki en büyuü değerler bu bölgede sıralı
      // olarak durur.
      // Aşağıdaki döngünün her dönüşünde sıralanmış bölgenin
      // uzunluğu bir artacak (başta 0), ve sıralanmamış bölgenin
      // uzunluğu bir azalacaktır (başta n)
      for (i=n-1; i>=1; i=i-1) {
            // sıralanmamış kısımdaki en büyük değerin dizi
            // içindeki yerini bul
            maxloc=0;
            for (j=1; j<=i; j=j+1)
                  if(a[j] > a[maxloc])
                        maxloc = j;
            // eğer en büyük elemanın yeri dizinin sıralanmamış
            // kısmındaki en son yer değilse, en büyük elemanla
            // bu en son elemanın yerlerini değistir.
            if (maxloc != i) {
                  temp = a[maxloc];
                  a[maxloc] = a[i];
                  a[i] = temp;
            }
      }
}
```

Bu sıralama algoritması şöyle çalışır:

- Dizi iki ayrı bölgeye ayrılmış olarak düşünülür. İlk bölge daha sıralanmamış elemanlardan oluşur. İlk başta bütün dizi sıralanmamış olarak görülecektir. Dizinin ikinci kısmında sıralanmış elemanlar olacaktır. İlk başta hiç sıralanmış eleman bulunmayacaktır.
- Sıralanmamış bölgedeki bütün elemanlara bakarak onların en büyüğünün yerini buluruz. İlk geçişte bütün elemanlar sıralanmamış bölgede olduğundan, gerçekte

dizideki en büyük elemanın yerini bulmuş olacağız. Daha sonra yer değiştirme işlemi ile bu en büyük elemanı sıralanmış bolgede kendi yeri olacak olan yere (sıralanmamış bölgenin en sonuna) koyarız ve bu yerdeki değeri de bulunan en büyük elemanın yerine kaydırırız. Böylece sıralanmış bölgenin boyu bir artacak ve sıralanmamış bölgenin boyu bir kısalacaktır. Bu işlem n-1 kere tekrar edildiğinde bütün dizi sıralanmış olacaktır.

Bu algoritma 9,7,1,3,2,5 dizisi için aşağıdaki gibi çalışacaktır:



Görüldüğü gibi 5 geçişten sonra dizimiz sıralanmış oldu.

6.4. İki-Boyutlu Diziler

Diziler yalnızca tek boyutlu değil 2, 3 ya da daha fazla boyutlu da olabilir. İki boyutlu diziler tablolar, matrisler gibi verilerin gösterilmesinde kullanılabilir. İki boyutlu dizi aşağıdaki gibi tanımlanıp yaratılabilir:

veri-türü dizi-adı `[`*sıra_sayısı*`]` `[`*kolon_sayısı*`]` `;`

Bu ifade iki boyutlu bir dizi yaratacaktır. Yaratılan dizide *sıra_sayısı* kadar sıra (indis 0'dan *sıra_sayısı*-1'e kadar) ve *kolon-sayısı* kadar da kolon (indis 0'dan *kolon_sayısı*-1'e kadar) olacaktır.

Örneğin,

```
int a[5][6];
```

ifadesi 5 sıralı ve 6 kolonlu iki boyutlu bir dizi yaratacaktır. Bu dizi toplam 5*6=30 eleman tutacaktır. Bu dizideki elemanlara iki indis kullanarak erişebiliriz. Yaratılan bu dizi şekilsel olarak aşağıdaki gibi gösterilebilir:

İlk elemana

```
a[0][0]
```

ifadesiyle erişebiliriz. Genelde iki boyutlu bir dizi elemanına erişebilmek için aşağıdaki ifadeyi kullanırız:

dizi-adı[*sıra-indisi*][*kolon-indisi*]

İki boyutlu bir dizinin elemanlarına da tanımlama zamanında ilk değerleri verilebilir. Örneğin,

```
int a[][]  = {{3,5,7},{8,6,9}};
```

ifadesi 2 sıralı ve 3 kolonlu iki boyutlu bir diziyi yukarıda verilen ilk değerler ile yaratır. (3, 5, 7 ilk sıradaki elemanlar, 8, 6, 9 ikinci sıradaki elamanlar olacaktır.)

Örnekler:

1. İki boyutlu bir dizinin elemanlarının değerlerinin klavyeden okunması:

```
int a[10][20];
for (row=0; row<10; row=row+1)
   for(col=0; col<20; col=col+1) {
      printf("bir sayı girin > ");
      scanf("%d",&n);
      a[row][col] = n;
```

```
    }
```

Bu program parçası klavyeden 200 tamsayı okur, ve yukarıda tanımlanan iki boyutlu diziyi bu tamsayılar ile doldurur. Birinci sayı `a[0][0]` elemanının içine, ikinci sayı `a[0][1]`elemanının içine, ..., yirminci sayı `a[0][19]` elemanının içine, yirmibirinci sayı `a[1][0]` elemanının içine, ..., 200. sayı `a[9][19]` elemanının içine konacaktır.

2. Matris çarpımı:

Burada `NxM` olan bir matrisi `MxK` olan bir matrisle çarparak onların çarpımını tutan bir `NxK` matrisi yaratacağız. Bu matris çarpma işlemini bir işlev olarak yazacağız. Bu işlevin 6 tane parametresi olacaktır. İlk üçü matrisler, diğer üç değer de `N, M` ve `K` değerleridir. İlk iki matris çarpılacak olan matrisler ve üçüncüsü de sonucun içinde döndürüleceği matris olacaktır.

```
void matrixMult(int a[][], int b[][],
                              int c[][],int n,
                              int m, int k) {
   int i,j,h;

   for(i=0; i<n; i=i+1)
      for (j=0; j<k; j=j+1) {
         c[i][j] = 0;
         for (h=0; h<m; h=h+1)
            c[i][j] = c[i][j] + a[i][h] * b[h][j];
      }
}
```

7. Karakterler ve Dizgiler

Amaç: Bu hafta karakter verilerin ve dizgilerin ne olduğunu ve onlar üzerinde yapabileceğimiz işlemleri tartışacağız.

1. Bu kapsamda karakter verilerin nasıl okunup bastırıldığını, ve C kütüphanesindeki karakterler için tanımlanmış işlevlerinden bazılarını göstereceğiz.
2. Dizgilerin (karakter dizilerinin) bildirimlerinin nasıl yapıldığını, bellekte nasıl saklandıklarını ve C kütüphanesindeki en çok kullanılan dizgi işlevlerini açıklayacağız.

7.1. Karakterler

Klavyeden her tuşa dokunduğumuzda bir karakter yazmış oluruz. Karakterler C programlama dilinde `char` veri türü ile tanımlanır. Örneğin

```
char ch1, ch2;
```

iki tane `char` değişkeni tanımlar. Bir karakter sabiti tek tırnak içine yazılan bir karakterdir. Örneğin,

```
ch1 = 'A';
ch2 = 'b';
```

atama ifadeleri `ch1` değişkenin içine `A` karakterini, ve `ch2` değişkeninin içine de `b` karakterini koyar.

Boşluk da bir karakterdir. Boşluk karakteri `' '` karakter sabiti ile ifade edilebilir. Her bir rakam da bir karakter olabilir. '0','1', '2',...,'9' rakamların karakter olarak gösterimidir. Rakam karakterleri normal tamsayı rakamlarından tamamen farklıdır. Onlar bilgisayar tarafından tamamen iki farklı değer olarak yorumlanır. Örneğin, '1' ile gösterilen bir rakamının karakter hali tamsayı 1 den farklıdır. Birincisi bir karakter, ikincisi ise bir tamsayı olarak C tarafından yorumlanır. Aşağıdaki gibi bir karakter değişkene bir tamsayı koymaya kalkarsak C derleyicisi hata vermese de `ch1` değişkeni istediğimiz '5' karakteri yerine ASCII tablosundaki 5. karaktere denk gelen karakteri içerecektir.

```
ch1 = 5;
```

Bütün bastırılabilir karakterlere ek olarak bastırılamaz karakterler de vardır. Bu özel karakterlerin sabitleri bir escape karakter kullanılarak verilir. Kullanılan escape karakteri `'\'`karakteridir. Bazı özel karakterler ve onların sabitleri aşağıda verilmiştir:

`'\n'`	`return` ya da satır sonunu gösteren karakter. Klavyeden `return` ya da `enter` tuşuna basıldığında gönderilen karakterdir. Bu karakteri ekrana

	yazdırmaya kalktığımızda, bir sonraki satıra geçilecektir.
`'\t'`	Klavyeden `tab` tuşuna basıldığında gönderilen karakterdir. Eğer bu karakteri ekrana yazmaya kalkarsak belirli miktarda boşluk bırakacaktır.
`'\''`	Tek tırnak karakteri ′.
`'\\'`	Ters yatay çizgi karakteri \.

C programlama dilinde bütün karakterlerin 0 ile 255 değeri arasında değişen bir kodu vardır. Diğer bir deyişle C programlama dilinde 256 değişik karakter olabilir. C programlama dilinde kullanılan bu kodlama sistemi ASCII kodlama sistemidir. ASCII kodlama sisteminde kullanılan ilk 128 karakter İngilizce harfler, rakamlar, ve çok kullanılan diğer bazı karakterlerden oluşur (Kitabımızın Appendix A kısmına bakınız). Bu kodlama sistemi ile, aynı zamanda bütün karakterler bir sıraya konmuş olurlar. Bir karakterin ASCII kodunubulabilmek için o karakteri `int` veri türüne çeviririz. Örneğin,

`n1 = (int) 'A';`	// n1'in değeri 65 olacaktır.
`n1 = (int) 'Z';`	// n1'in değeri 90 olacaktır.
`n1 = (int) 'a';`	// n1'in değeri 97 olacaktır.
`n1 = (int) 'z';`	// n1'in değeri 122 olacaktır.
`n1 = (int) '0';`	// n1'in değeri 48 olacaktır.
`n1 = (int) '9';`	// n1'in değeri 57 olacaktır.
`n1 = (int) ' ';`	// n1'in değeri 32 olacaktır.

(n1 int veri türünde tanımlanmış olmalıdır.)

Bu tablodan da görüldüğü gibi, bu kodlama sisteminde 'A' karakteri 'Z' den önce gelmektedir (65 < 90) ve bütün büyük harfler sıralamada küçük harflerden önce gelmektedir. Rakamlar da büyük harflerden önce gelmektedirler.

İki karakter değeri karşılaştırılırken, onların ASCII kodlarına bakılır. Eğer ASCII kodları birbirine eşit ise, o iki karakter birbirine eşittir. Örneğin, aşağıdaki mantıksal deyimlerin değerleri yanlarında açıklamaları ile birlikte verilmiştir:

`'A' == 'A'`	doğru	aynı koda sahipler (=65)
`'A' == 'a'`	yanlış	farklı koda sahipler (65 ve 97)
`'A' < 'a'`	doğru	65 < 97
`'0' < '1'`	doğru	48 < 49
`'9' < 'A'`	doğru	57 < 65

7.1.1. Karakterlerle Girdi/Çıktı

C dilinin `stdio.h` kütüphanesinde karakter verilerin girdi ve çıktısı için gereken işlevler bulunur. Karakter analiz ve değişimleri için de `ctype.h` kütüphanesinde işlevler vardır. Bunlara programımızın başında `#include <ctype.h>` diyerek erişebiliriz.

Karakterler klavyeden `stdio.h` kütüphanesinde bulunan `getchar` işlevi kullanılarak okunabilir. Bu işlev bir parametre almaz, okuduğu değeri çağıran programa geri döndürür. Örneğin,

```
    char ch;
    ch = getchar();
```

program parçası ile, `getchar()` işlevi klavyeden bir karakter okur ve okuduğu değeri atama ifadesine döndürür. Böylece `ch` değişkeni bu okunan değeri saklayacaktır.

Karakterleri `scanf` ve `%c` damgası kullanarak da okuyabiliriz. Örneğin

```
scanf("%c", &ch);
```

ifadesi de klavyeden girilen karakteri okur.

Karakter verileri bastırmak için `putchar` işlevi kullanılabilir. Örneğin

```
putchar(ch);
putchar('a');
```

bu işlevin kullanımını göstermektedir. `printf` işlevi de `%c` damgası ile birlikte karakter verilerin çıktısı için kullanılabilir.

7.1.2 Karakter İşlevleri

Karakter verileri işleyen programlarda sık sık bir karakterin rakam mı, harf mi, boşluk mu vs. olduğunu belirlemek gerekir. Bu tür işlemleri yapabilmek için C'nin `ctype.h`kütüphanesindeki işlevleri kullanırız. Bu kütüphanede ayrıca büyük harfleri küçük harflere, küçük harfleri büyük harflere dönüştürme gibi değiştirme işlevleri de vardır. Aşağıdaki tabloda bu işlevlerden bazılarını bulacaksınız. Kitabımızın 458. sayfasında bunlara ek olarak başka işlevler de gösterilmiştir.

İşlev	Sonuç
`isalpha`	parametresi harf ise sıfırdan farklı bir değer (doğru) döndürür.
`isdigit`	parametresi rakam ise sıfırdan farklı bir değer (doğru) döndürür.
`islower`	parametresi küçük harf ise sıfırdan farklı bir değer (doğru) döndürür.
`toupper`	parametresinin büyük harfe çevrilmiş halini döndürür

Aşağıdaki örnekler bu işlevlerin kullanımını göstermektedir.

Örnekler:

Örnek-1: Aşağıdaki program parçası klavyeden bir karakter okur. Bu karakter küçük harf ise onu büyük harfe çevirir ve bastırır.

```
char ch;

ch = getchar();
if (islower(ch)){
   ch = toupper(ch);
   putchar(ch);
}
```

Örnek-2: Aşağıdaki program parçası bir satırdaki karakterleri nokta ('.') ya da return karakterine ('\n') kadar okur, kaç karakter okuduğunu bastırır.

```
int i=0;
char ch;

for (ch = getchar(); ch != '.' && ch != '\n'; ch = getchar())
   i += 1;
printf("%d tane karakter okundu",i);
```

7.2. Dizgiler (Strings)

Dizgiler karakterlerden oluşan dizilerdir. Karakter dizilerinin her bir elemanı bir karakterdir. Diğer dizilerde olduğu gibi bir dizginin de her bir elemanına ayrı ayrı erişebiliriz. Bir dizginin parçalarına erişebiliriz, bir karakterin bir dizgi içindeki yerini bulabiliriz, ve buna benzer bir çok işlemi dizgiler üzerinde yapabiliriz.

7.2.1. Dizgilerin Tanımlanması

C dilinde dizgileri bir dizi tanımlar gibi tanımlarız. Örneğin

```
char isim[20];
```

bildirimi ile `isim` dizgisi tanımlanıyor. Bu dizgi, indisleri 0 ile 19 arasında değişen herhangi bir karakter dizisini saklayabilir. Bu tanımlama ile derleyici bellekte toplam 20 karakterlik bir bölge ayıracaktır. `isim` olarak adlandırdığımız değişken de bu ayırılan bellek bölgesinin başlangıç adresini tutacaktır. Bu bölgedeki ilk karaktere 0, ikinci karaktere 1, üçüncü karaktere 2, ... indisleri ile erişebiliriz. Bu bölgede en çok 19 karakterlik bir dizgi tutabiliriz. Çünkü C'de her dizginin sonunu belirtmek için dizginin en son karakterinden sonra '\0' (boş karakter) konur. Bu karakter de bellekte bir elemanlık yer kaplar.

Dizgilerin kullanım alanları çok geniş olduğu için C'de diğer veri türündeki dizilerden farklı olarak dizgilerin ilk değerleri tanımlama sırasında kısaca iki çift tırnak içinde belirtilebilir (`"abc"` gibi). Örneğin

```
char str[15] = "bir dizgi";
```

ifadesi ile en çok 14 karakter uzunluğunda `str` adlı bir dizgi tanımlayıp ilk değer olarak "bir dizgi" sözcüklerini verebiliriz. Bu durumda `str` dizgisinin bellekteki görünümü aşağıdaki gibi olacaktır.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
str	b	i	r		d	i	z	g	i	'\0'	?	?	?	?	?

Bu şekilde de görüldüğü gibi `str` dizgisinin 9 indisli 10. elemanı '\0' yani boş karakterdir (null character) . C'de bütün dizgilerin sonu bu özel karakter ile belirtilir. C'deki dizgilerle ilgili bütün işlevler boş karakterden sonraki elemanlarda olan değerleri hiç dikkate almazlar.

`str` dizgisinin her elemanına indisini kullanarak erişebiliriz. Örneğin `ch` `char` veri tütünde bir değişken ise

```
ch = str[2];
```

atama ifadesiyle değerini `'r'` yapabiliriz.

C'de dizgileri tanımlamanın başka bir yolu da *karakter işaretçisi* kullanmaktır. Genel olarak işaretçi dediğimiz değişkenler bellekte bir adres gösteren değişkenlerdir. Bu gösterilen adresin veri türüne göre işaretçiyi de karakter işaretçisi, tamsayı işaretçisi vs. diye adlandırırız. Her veri türü için işaretçi tanımlamak mümkündür. Biz burada yalnızca karakter işaretçisinden bahsedeceğiz. (Diğer veri türleri için olan işaretçiler ve onlarla yapılabilen işlemler *Veri Yapıları ve Algoritmalar* dersinde anlatılacak).

Karakter işaretçileri `char *` ifadesiyle belirtilir. Örneğin

```
char *str1;
```

bildirimi ile `str1` değişkeni bir karakter işaretçisi olarak tanımlanır.
Yani `str1` değişkeni ile `char` veri türünde bir bellek bölgesine erişilebilir. Bu aynı zamanda o adreste başlayan bir dizginin de ismi olacaktır. Bu tanımlama anında derleyici `str1` değişkeninin gösterdiği bu dizgi için bellekte hiç yer ayırmaz. Çünkü bu dizginin uzunluğu hakkında hiç bilgi yoktur. Bu dizgiye yer ancak ona bir dizgi sabiti ya da değişkeni kopyalandığı zaman ayrılacaktır. Bu şekilde tanımlanan dizgilerin elemanlarına da yine indis kullanarak erişmek mümkündür.

İki boyutlu karakter dizileri tanımlayarak dizgi dizileri de tanımlanabilir. Örneğin

```
char gunler[7][10] =
{"Pazartesi","Sali","Carsamba","Persembe",
                      "Cuma", "Cumartesi", "Pazar"}
```

her elemanı en cok 10 karakter uzunluğunda bir dizgi olan 7 elemanlı bir dizi tanımlar ve bu diziye ilk değerlerini verir.

7.2.2.Dizgilerle girdi/çıktı

Dizgileri okumak ve bastırmak için `scanf` ve `printf` işlevlerini `%s` biçimleme damgası ile kullanırız.

```
printf("İsim : %s \n", str);
```

ifadesiyle `str` dizgisinin değeri bastırılır. `printf` işlevi dizginin sonundaki boş karaktere kadar (`'\0'`) bütün karakterleri bastırır. Eğer dizginin sonunda `'\0'` karakteri yoksa`printf` bellekte ilk boş karakter bulduğu yere kadar ya da bulamazsa programın bellekteki bölgesinin sonuna kadar olan bütün değerleri karakter olarak bastırır. Programın bellekteki bölgesini aşınca da hata vererek programı durdurur. Bu yüzden programlarımızda dizgi yaratırken dizginin sonuna bu boş karakterin konduğundan emin olmalıyız. Sabit dizgiler ("abcd" gibi) için boş karakter C tarafından eklenir.

`printf` işlevinde `%s` damgasını uzunluk vererek de kullanabiliriz.

```
printf("***İsim:%8s*Soyadi:%3s*** \n","Emre", "Kesim");
```

ifadesinde isim 8 karakterlik bölgeye *sağa dayalı* olarak yazılacaktır. Soyadı ise verilen uzunluktan daha uzun olduğu için 3 yerine 5 karakterlik bölgeye yazılacaktır. Ekranda

```
***İsim:    Emre*Soyadi:Kesim***
```

görülecektir. Dizgileri *sola dayalı* yazmak istersek %s damgasının önüne eksi işaretiyle birlikte rakam yazmamız gerekir:

```
printf("***%-10s*** \n", "Emre");
```

ekrana

```
***Emre      ***
```

yazacaktır.

Dizgileri `scanf` ile okurken adresleme işlecini (&) kullanmamıza gerek yoktur. Çünkü dizgi adları aslında zaten dizginin ilk elemanının bellekteki adresini gösterir. Örneğin,

```
scanf("%d%s%d%s", &gun,ay,&yil,gun_adi);
```

ifadesi ile ekrandan iki tamsayı ve iki dizgi okuyoruz (burada `gun` ve `yil` `int` veri türünde, `ay` ve `gun_adi` ise birer dizgidir). `scanf` dizgileri de tamsayıları okuduğuna benzer şekilde okur. Yani baştaki beyaz karakterleri (boşlukları, sekme tuşlarını, satırbaşlarını) atladıktan sonra okuduğu karakterleri ardışık bellek bölgelerine sırayla kopyalar. Tekrar bir beyaz karaktere ya da satır sonuna geldiğinde dizginin sonuna '\0' karakterini ekler. Örneğin, yukarıdaki `scanf` ifadesi

```
  11         Eylul
       2000              Pazartesi
```

verisi ile çalıştığında her bir değişken istenilen değeri saklar.

7.2.3. Dizgi İşlevleri

C programlama dilinde dizgiler üzerinde tanımlanmış işlevler `string.h` kütüphanesinde bulunmaktadır. Bu kütüphanede bulunan bazı işlevlerin prototipleri şunlardır (Daha büyük bir liste için kitabımızın Appendix B kısmına bakınız):

- `char *strcpy(char *s1, char *s2)`
- `char *strncpy(char * s1, char *s2, int n)`
- `char *strcat(char *s1, char *s2)`
- `char *strncat(char *s1, char *s2, int n)`
- `int strcmp(char *s1, char *s2)`
- `int strncmp(char *s1, char *s2, int n)`
- `int strlen(char *s)`

Bu listedeki işlevlerin dizgi olan parametreleri karakter işaretçisi (`char *`) olarak belirtilmiştir. İlk dört işlev de sonuç olarak bir dizgi döndürmektedir. Eğer işlevlere dizgileri parametre olarak geçiriyorsak ya da işlev bir dizgi döndürüyorsa karakter işaretçilerini bu şekilde kullanabiliriz.

strcpy ve strncpy:

C'de atama işleci ile bir dizginin değerini değiştiremeyiz. Çünkü daha önce de söylediğimiz gibi bir dizgi adı aslında karakter dizisinin ilk elemanının bellekteki adresidir.

```
char str[10];
str = "deneme";
```

şeklinde yazdığımız program parçası, C derleyicisi tarafından `str` dizgisinin değerini değil, onun bellekteki başlangıç adresini değiştirmeye teşebbüs olarak görülür ve hata mesajına sebep olur. Bu yüzden program içinde `str` dizgisine herhangi bir değeri vermek istersek `strcpy` ya da `strncpy` işlevlerinden birini

kullanmamız gerekir. Bu işlevler bir dizgiye diğer bir dizginin değerini kopyalarlar. Farkları parametre sayılarının aynı olmamasıdır. `strcpy` iki parametrelidir. İkinci parametresi olan dizgiyi ilk parametresi olan dizgiye tamamen kopyalar (başından '\0' karakterine kadar olan kısmı). `strncpy` ise üç parametrelidir. İkinci parametresi olan dizginin başından, bir tamsayı olan üçüncü parametresinde belirtilen sayı kadar karakterini birinci dizgiye kopyalar. Dizginin sonunu gösteren '\0' karakteri kopyalanan karakterler arasında yoksa onu ayrıca eklemek gerekir (varsa ayrıca eklememiz gerekmez). Örneğin,

```
strcpy(str, "deneme");
```

işlev çağırma komutuyla `str` dizgisinin değeri "deneme" olarak değiştirilir.

```
strncpy(str, "deneme", 4);
str[4] = '\0';
```

ifadeleri ise `str` dizgisine yalnızca "dene" değerini kopyalayacaktır.

Bu işlevleri kullanırken dikkat etmemiz gereken şey, değeri değişen dizginin sonuna mutlaka '\0' karakterinin eklenmiş olmasıdır. Yukarıdaki örnekte `str` dizgisinin uzunluğu yeterince büyük olduğundan '\0' karakteri `str` dizgisinin sonuna doğru bir şekilde eklenmiştir. Ancak,

```
strcpy(str, "cok uzun bir dizgi");
```

ifadesi ise `str` dizgisi için ayrılan 10 karakterlik yeri taşıracaktır. Taşan "ir dizgi" karakterleri '\0' karakteri ile birlikte bellekte başka değişkenler için ayrılan yerlere yazılacaktır. Bu da çeşitli hatalara sebep olacaktır. O yüzden bu işlevleri kullanırken dikkatli olmalıyız.

`strcpy` ve `strncpy` işlevleri sonuç olarak bir dizgi döndürür (prototiplerine bakınız). Bu döndürülen dizgi değeri değişen ilk parametrenin kendisidir.

Aşağıdaki örnek program parçası `strcpy` ve `strncpy` işlevlerini kullanarak bir dizginin parçaları ile yeni dizgi değerleri yaratmayı gösteriyor.

```
char sonuc1[10], sonuc2[10];
tarih[15]="11 Eylul 2000";
strncpy(sonuc1, &tarih[3],5);
sonuc1[5]='\0';
strcpy(sonuc2,&tarih[9]);
```

Bu program parçası çalıştığında `sonuc1`'in değeri `"Eylul"`, `sonuc2`'nin değeri de `"2000"` olacaktır. Burada işlevlere `tarih` dizgisinin belli elemanlarının adreslerini parametre olarak verdik. Bu durumda `strcpy` ve `strncpy` tarih dizgisinin bu elemanlardan itibaren olan kısımlarını bir dizgi olarak görmüş oldu.

strcat ve strncat:

Bu işlevler ilk dizgi parametrelerinin değerini, ikinci dizgi parametresinin değerinin tamamını ya da bir kısmını birincinin sonuna ekleyerek değiştirirler.

Her iki işlev de birinci parametre olan dizginin uzunluğunun eklenecek karakterler için yeteceğini varsayarlar. İşlevlerin döndürdüğü değer yine değeri değişen ilk parametredeki dizgidir. Örneğin,

```
    char s[8]="abcd";
    strcat(s,"FGH"); // s abcdFGH karakter dizisi tutar.
```

program parçasında `s` dizgisine ilk değer olarak "`abcd`" verdik, sonra bu dizginin değerini `strcat` işlevi ile `abcdFGH` olarak değiştirdik. Burada `strcat` `s` değişkeninin gösterdiği `"abcd"` dizgisinin sonuna `"FGH"` dizgisini kopyalar ve `s` değişkeni yeni `"abcdFGH"` dizgisini gösterir. Bu yeni dizgi değeri '\0' karakteri ile birlikte toplam 8 karakter uzunluğundadır ve `s` dizgisini tam olarak doldurur.

Aşağıdaki örnekte ise `t` dizgisinin `strcat` ile yeni yaratılan değeri toplam uzunluğunu aşmaktadır:

```
char t[10]="abcdef";
strcat(t,"GHIJKLM");
```

Bu durumda bellekte diğer değişkenler için ayrılan bölgelerin değeri değişmiş olabilir. Bu tür hatalara sebep olmamak için `strcat` yerine `strncat` işlevini kullanabiliriz:

```
strncat(t, "GHIJKLM",3);
t[9] = '\0';
```

Burada `strncat` işlevi `t` dizgisinin sonuna "`GHIJKLM`" dizgisinin sadece ilk üç karakterini kopyalar, `'\0'` karakterini ayrıca eklemez. `strncat` işlevi de `strncpy` işlevi gibi sadece belirtilen sayıda karakteri dikkate alır. Bu karakterler arasında '\0' karakteri yoksa onu ayrıca eklememiz gerekir.

strcmp ve strncmp:

Bu iki işlev iki dizgi değerini karşılaştırmak için kullanılır. Sayısal ve karakter değerleri =, > ya da < işleçleri ile karşılaştırmak kolaydır. Örneğin, 'A' < 'B' ya da 50 > 10 doğru ifadelerdir. Ama dizgiler için karşılaştırma yapmak bu kadar kolay değildir. C dilinde "abc" < "abcd" gibi bir ifade yazamayız. Çünkü dizgilerin bellekte saklanma şekilleri standart veri türlerine göre farklıdır.

Örneğin s1 ve s2 iki dizgi değişkeni olsun. Daha önce de belirttiğimiz gibi, s1 ve s2 değişkenleri bu dizgilerin bellekteki başlangıç adreslerini tutarlar. Eğer s1 < s2 gibi bir şart yazarsak C bunu farklı şekilde yorumlar ve s1 dizgisinin bellekteki adresi s2 dizgisinin bellekteki adresinden önce mi diye kontrol eder. Bu yüzden, iki dizginin değerlerini alfabetik olarak karşılaştırmak istiyorsak C'nin `string.h` kütüphanesindeki `strcmp` ya da `strncmp` işlevlerini kullanmalıyız.

strcmp işlevi verilen iki dizgi değerini karşılaştırdıktan sonra sonucu aşağıdaki üç şekilden biriyle ifade eder (Karşılaştırılan dizgiler s1 ve s2 olsun) :

- Eğer s1 alfabetik sırada s2'den önce geliyorsa strcmp eksi bir tamsayı döndürür.
- Eğer s1 ve s2 aynı ise strcmp sıfır döndürür.
- Eğer s1 alfabetik sırada s2'den sonra geliyorsa strcmp artı bir tamsayı döndürür.

Örneğin
strcmp("abcd", "abcde")   -----> eksi bir tamsayı
strcmp("xyz", "xyz") ---------> sıfır
strcmp("xyz", "abc") -------> artı bir tamsayı
İki dizginin alfabetik karşılaştırılmasında her iki dizginin karakterlerinin ASCII kodlarının tek tek karşılaştırılması esas alınır. Karşılaştırmaya ilk karakterlerden başlanır ve ilk farklı karaktere kadar devam edilir. İki dizgi arasında bulunan ilk farklı karakterlerin karşılaştırma sonucuna göre artı ya da eksi tamsayı döndürülür. Hiç farklılık bulunamazsa sonuç sıfır olarak döndürülür. Bu ASCII kod karşılaştırmasına göre küçük harf, büyük harf ayrımı da karşılaştırmanın sonucunu etkileyecektir. Örneğin
strcmp("ABC", "abc") ------> eksi bir tamsayı döndürecektir.
Diğer karşılaştırma işlevi strncmp ise iki dizginin başından üçüncü parametresinde belirtilen değer kadar karakterini karşılaştırır. Karşılaştırma sonucunu strcmp işlevi gibi döndürür. Örneğin:
strncmp("abcde", "abcDEF", 3) -----> sıfır
strncmp("abcde", "abcDEF", 4) -----> artı bir tamsayı
döndürür.

strlen:

Bir programda kullandığımız dizgilerin uzunluğunu çoğu zaman bilemeyiz. C kütüphanesinde dizgilerin uzunluğunu bulmamıza yarayacak strlen işlevi vardır. Bu işlev parametresi olarak verilen dizginin sonundaki '\0' karakteri hariç kaç tane karakteri olduğunu döndürür. Örneğin

```
char s[10] = "abcde";
```

dizgisini tanımlamış olalım. Bu dizgi için bellekte ayrılan 10 karakterlik yerin hepsi kullanılmamıştır. Bu dizginin uzunluğu sadece 5 karakterdir. Bu durumda
printf("%d \n", strlen(s));
ifadesi ile ekranda 5 sayısını görürüz.

7.3. Program Örnekleri

Örnek 1:

İlk örneğimiz bir karakter dizisinin palindrom olup olmadığını kontrol eden bir programdır. Bir karakter dizisi soldan sağa, ve sağdan sola aynı okunuyorsa palindrom'dur. Bu programda isPalindrome adlı bir işlev tanımladık. Bu işlev bir dizgiyi parametre olarak alır. O dizgi palindrom ise 1, değilse 0 döndürür. Bu işlevi yazarken parametrenin dizgi olduğunu karakter işaretçisi kullanmak yerine herhangi bir dizi parametresi gibi [ve] kullanarak belirttik.

Palindrome.cpp

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

#define UZUNLUK 20

 // Verilen en cok 20 karakter uzunlugundaki bir dizginin palindrom
 // olup olmadigini kontrol et.
 // Eger bir karakter dizisi soldan saga, ve sagdan sola ayni okunuyorsa
 // palindrom dur. Bazi palindrom kelimeler:
 //     tat
 //     ana
 //     abba
 //     madam

 int isPalindrome(char s[]);

 int main()
 {
    char str[UZUNLUK];

    // dizgiyi oku
    printf("Bir dizgi girin > ");
    scanf("%s", str);

    // okunan karakter dizisinin palindrom olup olmadigini kontrol et.
    if (isPalindrome(str))
       printf("%s bir palindromdur.\n", str);
    else
       printf("%s bir palindrom degildir.\n", str);
    return 0;
 }

int isPalindrome(char s[])
{
    int i, j, flag;
    i = 0;  // ilk karakterin pozisyonu
    j = strlen(s) - 1;  // son karakterin pozisyonu
    flag = 1;
    while ((i<j) && flag){
       // i. ve j. karakterleri karsilastir
```

```
        if (s[i] != s[j])
           flag = 0;  // ayni degillerse, kelime palindrom olamaz
        else {i++; j--; } // bir sonraki karakterlere ilerle
    }
    return flag;
}
```

Örnek 2:

İkinci örneğimiz artı bir tamsayı okur ve bu tamsayıyı ikili sistemdeki (binary sayı) karşılığına çevirir. Bu programda kullanılan toBinary işlevi bir tamsayıyı parametre olarak alır ve bir dizgi döndürür. Bu işlevin içinde `strcat` işlevini `strcpy` işlevinin parametresi olarak çağırdık. Başka bir deyişle `strcpy` işlevi `strcat` işlevinin döndürdüğü dizgi üzerinde işlem yapıyor. Bu işlevde `sb` değişkeninin gösterdiği ikili sayıyı tutan dizgi en sağ basamaktan sola doğru yaratılıyor. Döngünün her dönüşünde en sol basamağa ya 0 ya da 1 ekleniyor. Dizginin böyle sol baştan büyümesini gerçekleştirmek için s0 ve s1 adlı "0" ve "1" dizilerini kullandık. Bu programı almak için DecToBinary.cpp yi tıklayınız.

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

 // Bir pozitif decimal sayiyi, binary sayiya (ikili sistemde) cevirir.
 // Sonuc bir dizgi olarak geri dondurulur.
 // Ornegin onluk sistemde 12, ikili sistemde 1100 olur.

char *toBinary(int decVal);

int
main()
{
    int num;

    // arti bir tamsayi oku
    printf("Bir arti tamsayi girin > ");
    scanf("%d",&num);

    // okunan tamsayiyi ikili sistemdeki sayiya cevir ve ekrana yaz.
    printf("%d sayisinin ikili sistemdeki karsiligi: %s",num,
            toBinary(num));

}
```

```
char *toBinary(int decVal) {


    // bos dizgi  yarat
    char *sb = "               ";
    char *s0, *s1;


       strcpy(sb,"");
    if (decVal == 0)
       strcat(sb,"0");  // eger sayi 0 ise, yanit 0 dir
    else  // sayi sifirdan farkli ise, sayiyi binary sayiya cevir
       while (decVal != 0) {
          strcpy(s0,"0");
       strcpy(s1,"1");
          if (decVal%2 == 0)
             strcpy(sb,strcat(s0,sb)); //sayinin son karakteri 0
          else
             strcpy(sb,strcat(s1,sb));  // sayinin son  karakteri 1
          decVal = decVal / 2;  /* bir sonraki karakteri bulmak icin
                                   ilerle */
       }
    return sb;
}
```

8. Sırasal Erişimli Kütükler

Amaç:
Bu hafta standart girdi (klavye) ve standart çıktı (ekran) detaylarını tekrar gözden geçirdikten sonra programlarda girdi ve çıktıların disklerde saklanan dosyalar aracılığıyla nasıl yapılabileceğini göstereceğiz. Bu amaçla

1. Temel dosya işlemlerini anlatacağız.
2. C'deki işlevlerle metin dosyaları ve ikili dosyaların yaratılmasını ve işlenmesini göstereceğiz.

8.1. Dosyalara İlişkin Temel Kavramlar

Dosyalar (ya da kütükler) disket, disk ya da CD gibi ortamlarda tanımlanmış alanlardır. Dosyalara onlara vermiş olduğumuz isimleri kullanarak erişebiliriz. Dosyaların içindeki veriler üzerinde işlem yapılabilmesi için *dosyanın açılması* ve içindekilerin belleğe kopyalanması (okunması) gerekir. Dosyanın açılması dosyayla ilgili bazı başlangıç işlemlerinin yapılması demektir. Açılan her dosyanın bilgileri (ismi, tipi, diskteki yeri vs.) işletim sistemi tarafından bellekteki bazı tablolarda saklanır. Bu dosyalarla bellek arasındaki veri alışverişi için bellekte tampon bölgeler oluşturulur. Bir dosya üzerindeki işlemler bitince o *dosyanın kapatılması* gerekir. Dosyanın kapatılması ile işletim sistemi bu dosyanın içeriğinin güvenilir bir şekilde diskte saklanmasını sağlar.

Programlarımızda açtığımız dosyalara erişim için dosya işaretçilerinden yararlanırız. Dosya işaretçisi o anda dosyanın kaçıncı byte offset'i üzerinde işlem yapılabileceğini gösteren bir sayıdır. Girdi ve çıktı işlemleri bu dosya işaretçisinin gösterdiği yerden itibaren yapılır. Bu işaretçinin konumunu değiştirmek C'deki bazı işlevlerle mümkündür.

8.1.1. Dosya Tipleri

C'de iki tür dosya üzerinde işlem yapabiliriz: Metin dosyaları (text files) ve ikili dosyalar (binary files). Metin dosyaları birbiri ardına gelen karakterlerden oluşan dosyalardır. Örneğin editör yardımıyla yarattığımız dosyalar birer metin dosyalarıdır. Metin dosyalarında karakterlerin ASCII kodları saklanır. İkili dosyalarda ise verilerin değeri ASCII koda çevrilmeden bellekteki gösterim biçimleri ile saklanır. İkili dosyalar editör yardımıyla yaratılamazlar ve içerikleri ekranda okunur bir şekilde görülemez. Bu tip dosyalar ancak programlar tarafından yaratılır ve okunabilirler. Şimdilik ikili dosyaları bu dersimizin son kısmına bırakıp metin dosyalarını daha detaylı inceleyelim.

Metin dosyalarının uzunluğu sabit değildir. Dosyanın sonunu belirtmek için, bilgisayar (işletim sistemi) dosyanın sonuna en son karakterden sonra özel bir karakter ekler. Bu özel karaktere dosya-sonu (end-of-file) karakteri denir (bunu <eof> ile belirteceğiz). Metin dosyalarında satır sonları da özel bir karakterle belirtilir. Bu karaktere de yeni-satır (newline) karakteri denir. C'de bu karakter '\n' ile gösterilir. Örneğin bir editörde her <enter> tuşuna bastığınızda bu yeni satır karakteri dosyanıza eklenecektir.

Aşağıdaki satırları editör yardımıyla yazmış olduğumuzu varsayalım:

```
Bu birinci satir. <enter>
Bu da ikinci satir. <enter><eof>
```

Bu yazdığımız karakterler ekranda iki satır olarak gözükse de diskte birbiri ardına gelen karakterler dizisi olarak saklanacaklardır:
`Bu birinci satir.` <enter>`Bu da ikinci satir.` <enter><eof>
Bütün girdi ve çıktı verileri aslında böyle sürekli biçimde ardarda gelen karakter akışları olarak görüldüğünden veri kaynaklarına ***girdi akışı***, veri hedeflerine de ***çıktı akışı*** denir. Bu terimler dosyalar için kullanıldığı gibi klavye'den gelen veriler ve ekrana giden veriler için de kullanılır.

8.1.2. Klavye ve Ekran

Etkileşimli programlamada C klavye ve ekranı özel isimlerle özdeşleştirir. Klavyeden gelen girdi akışı **`stdin`** ismi ile temsil edilir. Programın beklenen normal çıktılarını temsil eden**`stdout`** akışı ile hata mesajlarının yönlendirildiği **`stderr`** akışı ekran ile özdeşleşmiştir. Bu üç akış verileri birbiri ardına gelen karakterler olarak gördüğünden birer metin dosyası gibi görülebilirler.

Klavyeden veri girerken normal olarak satır sonlarını <enter> tuşu ile belirtiriz. Bu tuşa bastığımızda newline karakteri `stdio` akışına eklenir. Girdimizin sonunu ya bir sentinel değeri ile belirtebiliriz, ya da <eof> karakterini kullanabiliriz. <eof> karakteri sistemden sisteme değişik olabilir. Örneğin Unix ortamında <control-d> tuşlarına beraber basınca <eof> karakteri girmiş oluruz. Başka sistemlerde <control-z> kullanılıyor olabilir.

Sistemdeki `stdout` ve `stderr` akışlarına birşey yazıldığında bunu ekranda görürüz. Daha önceki derslerimizde de gördüğümüz gibi ekranda satır atlamak için `printf`'in biçimleme dizgisinde '`\n`' kullanıyorduk. Bundan başka sıklıkla kullanılan damgalardan bazıları şunlardır: '`\t`' tab damgası, '`\f`' yeni sayfaya atlama damgası, '`\r`' üstünde bulunulan çıktı satırının başına gitmek için kullanılan damga, '`\b`' backspace damgası.

C bu damgalar ile <eof> karakterini farklı şekilde kullanır. Örneğin yeni satır damgası diğer herhangi bir karakter gibi işlem görür: `scanf` kullanılarak %c damgası ile okunur; '`\n`' karakterine eşit mi diye karşılaştırma yapılabilir; `printf` ile bastırılabilir. <eof> karakteri ise bu şekilde işlenemez. Okuma işlemi yapılırken <eof> karakterine rastlanırsa bu bir hata olarak yorumlanır ve girdi işini yapmaya çalışan işlev (örneğin `scanf`) eksi bir sayı döndürür. Bu eksi sayı EOF isimli bir sistem sabitinin değeridir. Programlarımızda veri sonu <eof> karakteri ile belirlenmişse, okuma işlemini aşağıdakine benzer bir döngü ile yürütebiliriz:

```
status= scanf("%d", &num);
while (status !=EOF) {
   //okunan sayiyi işle
   process(num);
   status = scanf("%d", &num);
}
```

8.1.3. Dosya İşaretçileri

Etkileşimli programlar genelde girdilerini klavyeden alırlar ve çıktılarını ekrana yazarlar. Ancak bazen programın girdisi ya da çıktısı (ya da her ikisi) bir dosya olabilir. Eğer bir program bir dosyadaki verileri işleyecekse o programda o dosyaya erişimi sağlayacak bir dosya işaretçisi tanımlamak gerekir. Dosya işaretçisi olarak tanımlanan değişkenlerin veri türü `FILE *` olmalıdır. Örneğin,

```
FILE *girdi_dosyasi,
     *cikti_dosyasi;
```

bildirim ifadesi ile isimleri `girdi_dosyasi` ve `cikti_dosyasi` olan iki dosya işaretçisi tanımlanır. Daha önce de belirttiğimiz gibi bir dosyanın işlenebilmesi için önce o dosyanın açılması gerekir. C'de bu işlemi `fopen` işlevi ile yapabiliriz. `fopen` işlevinin prototipi aşağıdaki gibidir:
`FILE *fopen(char *`*dosya_adı*`, char *`*açış_tarzı*`);`
`fopen` işlevi, veri türü FILE * olan bir dosya işaretçisi geri döndürür. İşlevin ilk parametresi açmak istediğimiz dosyanın diskteki ismidir. Bu isim çift tırnak içinde gerekirse dizin yolu da eklenerek yazılır. İkinci parametre ise dosyanın hangi işlem için açılacağını belirtir. Çeşitli açış tarzları olabilir. Aşağıdaki tablo bunlardan bazılarını özetlemektedir.

Açış Tarzı	**İşlem Türü**
"r"	Yalnız okuma. Bu tarzda açılmış dosyaya yazma yapılamaz. Dosyanın daha önce yaratılmış olması gerekir.
"w"	Dosya diskte olsa da olmasa da yeni baştan yaratılır. Varolan bir dosya bu tarzda açılırsa eski içeriği tamamen silinir.
"a"	Varolan bir dosyanın sonuna ekleme yapmak için açılır. Bu tarzda açılan dosyadan okuma yapılamaz.

Buradan da görüleceği gibi açış tarzına göre `fopen` ile diskte hazır bulunan bir dosyayı da açabiliriz, yeni bir dosyayı da en baştan yaratabiliriz. Buna tamamen açış tarzı ile karar verilir.

Aşağıdaki ifadeler `fopen` işlevinin kullanımına örnektir:

```
girdi_dosyasi = fopen("veriler.txt", "r");
cikti_dosyasi = fopen("sonuclar.txt", "w");
```

Bu satırları içeren bir programda `girdi_dosyasi` programla aynı dizindeki `veriler.txt` isimli dosyaya erişmek için, `cikti_dosyasi` da yine aynı dizinde `sonuclar.txt`isimli dosyayı yaratmak için kullanılacaktır. Bu arada belirtmekte fayda var: `stdin`, `stdout` ve `stderr` de `FILE *` veri türünde değişkenler olup C programının başında sistem tarafından ilk değerleri verilir.

Eğer `fopen` işlevi dosyayı açmayı başaramazsa (örneğin, açılmaya çalışılan dosya diskte bulunamayabilir), NULL dosya işaretçisini geri döndürür. Örneğin `veriler.txt` dosyası diskte yoksa `fopen` bu değeri döndürecektir ve kullanıcı da bu durumdan aşağıdaki gibi bir ifade ile haberdar edilebilir:

```
if (girdi_dosyasi == NULL)
   printf("Hata: Dosya bulunamadı. \n");
```

Programda açılan her dosya, üzerindeki işlemler bittiğinde kapatılmalıdır. Dosyanın kapatılmaması özellikle yazım işlemi yapıldıysa problemlere yol açabilir. C'de dosya kapatmak için`fclose` işlevi kullanılır. Bu işlevin prototipi aşağıdaki gibidir:

```
int fclose(FILE *d);
```

İşlevin tek parametresi kapatılmak istenen dosyanın işaretçisidir. Kapatma işlemi başarılıysa 0 değerini, başarısızsa EOF değerini geri döndürür.
Örneğin `girdi_dosyasi` ile erişilen `veriler.txt` dosyası
```
fclose(girdi_dosyasi);
```
ifadesiyle kapatılır.

8.2. Metin Dosyaları

C'de dosya işlemleri işlevlerle yapılır. Dosyaların açılması, kapatılması, içerisine bilgilerin yazılması, okunması gibi işlemler için standart C işlevleri vardır. Bu kısımda bu işlevlerden metin dosyaları üzerinde en çok kullanılanları tanıtacağız. Metin dosyalarının C'de nasıl açılıp kapatıldığını bir önceki kısımda gördük. Şimdi metin dosyalarından nasıl girdi ve çıktı yapıldığını inceleyelim.

8.2.1.Girdi/Çıktı İşlevleri

C'de dosyalardan girdi ve çıktı yapabilmek için, `scanf` ve `printf` işlevlerine eşdeğer **`fscanf`** ve **`fprintf`** işlevleri tanımlıdır. Bir dosyadan veri okumak için kullanılan `fscanf`işlevinin ilk parametresi okuma yapılan dosyanın dosya işaretçisi olmalıdır. İşlevin geri kalan kısmı aynen `scanf` gibidir: yani o da bir biçimleme dizgisi ve değeri okunacak değişkenler listesi içerir. `scanf` klavyeden gelen verileri okurken `fscanf` de dosya işaretçisinin gösterdiği dosyadan gelen verileri okur. `fscanf` de `scanf` gibi başarıyla okuyabildiği değerlerin sayısını döndürür. Eğer okuma sırasında <eof> karakterine rastlanırsa `fscanf` de eksi EOF değerini döndürür.

Dosyaya yazma işi için kullanılan `fprintf` işlevinin de `printf` işlevinden tek farkı ilk parametresi olarak yazılan dosyanın dosya işaretçisini istemesidir. `printf` ekrana nasıl yazıyorsa `fprintf` de dosyaya aynı biçimde yazar. Yazdığınız bir programda `printf` işlevlerinin yerine `fprintf` kullanırsanız ekran yerine dosyaya yazma yapmış olursunuz.

Aşağıdaki örneklerle bu işlevlerin kullanımını inceleyelim:

```
scanf("%d", &sayi);
```
ifadesi ile ekrandan bir tamsayı okuyabiliriz.
```
fscanf(girdi_dosyasi, "%d", &sayı);
```
ifadesi ile de tamsayıyı `girdi_dosyasi`'nın gösterdiği "`veriler.txt`" dosyasından okuyabiliriz. Burada da okuma işleminde boşluk karakterleri atlanır ve bir sonraki ilk boşluğa kadar okunan ASCII rakamlar tamsayıya çevirilip `sayi` değişkeninin değeri olarak bellekte saklanırlar.

Ekrana bir sayı yazdırmak için

```
printf("Sayi =  %d \n", sayi);
```
ifadesini kullanabiliriz. Aynı sayıyı "`sonuclar.txt`" dosyasına yazdırmak için ise
```
fprintf(cikti_dosyasi, "Sayı = %d \n", sayi);
```
yazmalıyız.

Gördüğünüz gibi ilk parametre olarak dosya işaretçisinin verilmesi dışında bir fark yok.

Daha önce her veri türü için tanımladığımız biçimleme damgaları `fscanf` ve `fprintf` için de geçerli.

Bunların dışında dosyadan bir karakter okuyan ve dosyaya bir karakter yazan **`getc`** ve **`putc`** işlevleri vardır. Bunlar da işlevsel olarak `getchar` ve `putchar` işlevlerine eşdeğerdir. Farkları yine dosya işaretçisini parametre olarak almalarıdır. Örneğin

```
ch= getc(girdi_dosyasi);
```

ile "`veriler.txt`" dosyasından bir karakter okuruz.

```
putc(ch, cikti_dosyasi);
```

ile "`sonuclar.txt`" dosyasına bir karakter yazarız. (`putc` dosya işaretçisini ikinci parametresi olarak alır).

8.2.2. Dosya sonunun tespit edilmesi

Genelde dosya işlemlerinde dosya baştan sona taranır. Bir dosyadan veri okurken dosyanın sonuna gelinip gelinmediği değişik şekillerde kontrol edilebilir. Eğer `fscanf` ile okuma yapılıyorsa `fscanf`'in döndürdüğü değer kontrol edilebilir:

```
for (status= fscanf(fp, "%d", &x);
      status !=EOF;
      status= fscanf(fp, "%d", &x) ) {
   ...
}
```

Başka bir yol da **feof** işlevini kullanmaktır. Bu işlevin tek parametresi vardır ve o da bir dosya işaretçisidir. Dosya işaretçisi dosya sonunu gösterdiğinde bu işlev 1 değerini; henüz dosya sonuna gelinmediyse 0 değerini döndürür. Bu işlevi kullanarak dosya okuma döngümüzü

```
while (!feof(fp)){
...
}
```

şeklinde kurabiliriz.

8.2.3. Örnek Program

Bu program, içindeki sayılar küçükten büyüğe sıralı olan iki dosyayı okur ve bu dosyalardan gelen bütün sayıları sıralı bir şekilde birleştirerek tek bir dosya yaratır. (merge.cpp)

```
/* Bu program iki dosyadaki sayilari okur, butun sayilari kucukten
 * buyuge sirali bir sekilde ucuncu bir dosyaya yazar. Ilk iki
 * dosyadaki sayilar da ilk basta siralidir. Uc dosya da birer
 * metin dosyasidir.
 */
#include<stdio.h>

int main()
```

```
{
   FILE *dosyaA, /* okunan birinci dosya */
        *dosyaB, /* okunan ikinci dosya */
        *dosyaC; /* yaratilan dosya */
   int not1,  /* birinci dosyadan okunan sayi */
       not2;  /* ikinci dosyadan okunan sayi */
   int f1, f2;

   /* Dosyalari ac */
   dosyaA = fopen("sinif1.txt","r");
   dosyaB = fopen("sinif2.txt","r");
   dosyaC = fopen("sinif.txt","w");

   /* Her iki dosyada da sayi oldugu muddetce okuma ve karsilastirma
      islemini yap, kucuk olan sayiyi ucuncu dosyaya yaz. Kucuk sayiyi
      okudugun dosyadan bir sayi daha oku */

   f1 = fscanf(dosyaA, "%d", &not1);
   f2 = fscanf(dosyaB, "%d", &not2);

   while ((f1!=EOF) && (f2!=EOF)){
      if (not1 < not2){   /* birinci dosyadan okunan sayi daha kucuk */
         fprintf(dosyaC,"%d\n", not1);
         f1 = fscanf(dosyaA, "%d", &not1);
      }
      else if (not2 < not1) { /* ikinci dosyadan okunan sayi daha kucuk*/
         fprintf(dosyaC,"%d\n", not2);
         f2 = fscanf(dosyaB, "%d", &not2);
      }
      else { /* iki sayi esit. Her iki dosyadan da oku */
         fprintf(dosyaC,"%d\n", not1);
         f1 = fscanf(dosyaA, "%d", &not1);
         f2 = fscanf(dosyaB, "%d", &not2);
      }
   }

   while (f1!=EOF){  /*ikinci dosya bittiyse, birincide geri kalan
                       sayilari oku ve ucuncu dosyaya yaz */
      fprintf(dosyaC,"%d\n", not1);
      f1 = fscanf(dosyaA, "%d", &not1);

   }
   while (f2!=EOF){ /* birinci dosya bittiyse, ikincide geri kalan
                       sayilari oku ve ucuncu dosyaya yaz */
      fprintf(dosyaC,"%d\n", not2);
      f2 = fscanf(dosyaB, "%d", &not2);

   }

   /* dosyalari kapat */
   fclose(dosyaA);
   fclose(dosyaB);
```

```
    fclose(dosyaC);
    return 0;
}
```

8.3. İkili Dosyalar

Daha önce de belirttiğimiz gibi ikili dosyalarda veriler bellekteki gösterim biçimleriyle saklanırlar. Yani, örneğin `int` veri türündeki bir tamsayı ASCII koduna çevirilmeden bellekte saklandığı gibi iki byte'lık bir veri olarak dosyaya yazılır. `double` veri türündeki bir reel sayı da yine bellekteki gibi mantissa ve exponent bölgeleri şeklinde dosyaya kopyalanır. İkili dosyaların içeriği ekranda okunamaz ve editör yardımıyla yaratılamaz. İkili dosyalar ancak programlar tarafından yaratılırlar ve yine ancak programlar tarafından okunurlar. Metin dosyalarında veriler ASCII koduna çevirilip saklandığı için, her okuma işleminde her verinin gösteriminin ASCII kodundan veri türünün bellekteki gösterim biçimine çevirilmesi için zaman harcamak gerekir. Aynı şekilde bir veriyi metin dosyasına yazarken de bellekteki gösterim biçiminden ASCII koduna çevrilmesi için zaman harcanır. (Bu çevirimler `scanf` ve`printf` gibi işlevler tarafından yapılır). Oysa ikili dosyalarla girdi çıktı yaparken bu çevirim için ayrıca bir zaman harcamaya gerek yoktur. Bu kısımda ikili dosyalar üzerindeki işlemler için gerekli bazı işlevleri anlatacağız.

8.3.1. Girdi/Çıktı İşlevleri

İkili dosyaları da metin dosyalarını tanımladığımız gibi tanımlayabiliriz. Örneğin,

```
FILE *bir_dosya;
```

ifadesi `bir_dosya` isimli bir dosya işaretçisi tanımlar. Bunun bir ikili dosya olduğunu dosyayı açarken belirtiriz. İkili dosyalar da yine `fopen` ve `fclose` işlevleri ile açılıp kapatılırlar ama bu işlevlerin ikinci parametreleri (açış tarzı) metin dosyalarından farklıdır. Açış tarzı "rb" ya da "wb" olabilir. Örneğin

```
bir_dosya = fopen("sayilar.bin", "rb");
```

ifadesi ile `bir_dosya`'yı yalnız okumak için açıyoruz. "`rb`" (`r`ead `b`inary) açış tarzı bunun ikili dosya olduğunu belirtiyor.

```
baska_dosya = fopen("output.bin", "wb");
```

ifadesi ile de `baska_dosya`'yı yaratmak için açıyoruz. ("`wb`" (`w`rite `b`inary) açış tarzı)

İkili dosyalarla girdi/çıktı yapmak için C'deki `fread` ve `fwrite` işlevlerini kullanmalıyız.

fread işlevi

`fread` işlevinin dört parametresi vardır:

1. okunacak verinin bellekte saklanacağı bölgenin adresi
2. verinin byte cinsinden uzunluğu
3. kaç tane veri okunacağı
4. okunacak dosyanın işaretçisi (dosya "rb" modunda açılmış olmalıdır)

`fread` dosya işaretçisinin gösterdiği yerden ikinci ve üçüncü parametresinin çarpımı kadar byte'ı ilk parametresinde belirtilen adresten başlayarak belleğe okur. `fread` okuyabildiği kadar veriyi okur ve okuyabildiği veri sayısını geri döndürür. Örneğin `bir_dosya'`nın

gösterdiği dosyadan bir tamsayı (`i`) okumak için aşağıdaki ifadeyi yazabiliriz:

```
fread(&i, sizeof(int), 1, bir_dosya);
```

Burada adresleme işlecini (&) kullanarak `i` değişkeninin adresini `fread` işlevine ilk parametre olarak verdik. Okuyacağımız veri bir tamsayı (`int` veri türü) olduğu için verinin uzunluğunu `sizeof(int)` ifadesi ile belirterek ikinci parametre olarak verdik. (`sizeof` C'de bulunan ve her veri türünün byte cinsinden uzunluğunu döndüren bir işlevdir). Üçüncü parametre olarak 1 yazdığımız için bu uzunlukta sadece bir veri okuyacağımızı belirtmiş olduk. Dördüncü parametre de `bir_dosya` olduğu için bu okuma işlemi "`sayilar.bin`" dosyasından yapılacaktır.

Burada akla 3. parametrenin gereksiz olduğu gelebilir. Ancak bu parametre birden fazla aynı türde veri okunacaksa çok işe yarar. Örneğin 10 elemanlı bir dizinin değerleri tek`fread` ifadesiyle okunabilir. Okuma işleminin başarıyla tamamlanıp tamamlanmadığı işlevin döndürdüğü değer ve bu üçüncü parametre değerinin karşılaştırılması ile kontrol edilebilir. Aşağıdaki örneği inceleyelim:

```
double dizi[10];
FILE *d;
int flag;
...
flag = fread(dizi, sizeof(double), 10, d);
if (flag != 10 )
   printf("Hata ! \n");
```

Burada `fread` işlevinin ilk parametresi bir dizi adı olduğu için adresleme (&) işlecini kullanmamıza gerek yoktur. (Hatırlayacağınız gibi dizi adları dizilerin bellekteki başlama adreslerini gösterirler).

fwrite işlevi

`fwrite` işlevinin de `fread` gibi dört parametresi vardır:

1. yazılacak verinin bellekteki adresi
2. verinin byte cinsinden uzunluğu
3. kaç tane veri yazılacağı
4. dosya işaretçisi

Örneğin `baska_dosya` ile gösterilen `output.bin` dosyasına bir tamsayı (`i`) yazmak için aşağıdaki `fwrite` ifadesi kullanılabilir:

```
fwrite(&i, sizeof(int), 1, baska_dosya);
```

Burada `i` değişkeninin değeri hiç ASCII koda çevrilmeden bellekte gösterildiği iki byte'lık gösterim biçimiyle dosyaya kopyalanacaktır. Örneğin `i`'nin değeri 183 olsaydı`0000000010110111` değeri dosyaya yazılacaktı. Oysa bu değeri bir metin dosyasına yazsaydık `i'`nin değeri 1, 8, 3 karakterlerine çevirilip yazılacaktı.

`fwrite` işlevini kullanarak bir dizinin elemanlarını ikili dosyaya yazabiliriz. Örneğin yukarıdaki örnekte tanımladığımız `dizi`'nin elemanları

```
fwrite(dizi, sizeof(double), 10, baska_dosya);
```

ifadesi ile `output.bin` dosyasına yazılabilir.

8.3.2. Dosya işaretçisinin konumunu değiştirmek

Şimdiye kadar gördüğümüz girdi çıktı yapan işlevlerin hepsi dosya işaretçisinin dosya üzerindeki konumunu okunan ya da yazılan sayıda byte değeri kadar ilerletir. Dosya işaretçisinin konumu girdi çıktı işlevlerinden başka bir de `fseek` işlevi ile direk değiştirilebilir. `fseek` işlevinin üç parametresi vardır:

1. dosya işaretçisi
2. kaydırma miktarı
3. başlangıç noktası

Üçüncü parametre konumlandırma işleminin nereden başlanarak yapılacağını belirtir. Bu parametre üç değer alabilir: 0,1 ya da 2:
0: dosyanın başından itibaren
1: bulunulan yerden itibaren
2: dosyanın sonundan itibaren
demektir.

Örneğin

```
fseek(d, 100, 1);
```

ifadesi ile `d` dosya göstericisi o andaki pozisyonundan 100 byte ileriye konumlandırılır.

```
fseek(d, -100, 1);
```

ifadesi ise o andaki pozisyondan 100 byte geriye konumlandırır.

```
fseek(d,0,0);
```

ifadesi de dosya işaretçisini tekrar dosyanın başına alır.

8.3.3. Örnek Programlar

Örnek 1.
Bu program klavyeden okuduğu sayıları yeni yarattığı bir ikili dosyada saklar.
Bu programın text haline buradan ulaşabilirsiniz.

```
#include<stdio.h>
#define UZUNLUK 20

int
main ()
{
     char dosya_ismi[UZUNLUK];
     FILE *cikti;
     int sayi, durum;

     /* Yaratilacak dosyanin ismini oku */
     printf("Cikti dosyasinin ismini girin: ");
     for (scanf("%s", dosya_ismi);
          (cikti = fopen(dosya_ismi,"wb")) == NULL;
          scanf("%s", dosya_ismi)) {
```

```
        printf("Hata: %s acilamadi \n", dosya_ismi);
        printf("Dosya ismini tekrar girin: ");
     }

     /* Klavyeden girilen sayilari EOF girilene kadar oku ve dosyaya yaz */
     printf("Sayilari girin:\n");
     for (durum = scanf("%d", &sayi);
          durum != EOF;
          durum = scanf("%d", &sayi))
        fwrite(&sayi,sizeof(int), 1, cikti);

     fclose(cikti);
     printf("%s dosyasi yaratildi. \n", dosya_ismi);

     return 0;
}
```

Örnek 2.

İkinci örnek programımız, içinde tamsayılar bulunan bir ikili dosyada kullanıcının klavyeden girdiği sayı(ları) arar. Sayının bulunup bulunmadığı mesajını bastırdıktan sonra başka sayı aranıp aranmayacağını sorar. Kullanıcı devam etmek isterse, dosya işaretçisinin konumunu dosya başına getirerek yeni girilen sayıyı arar. Bu programın text haline buradanulaşabilirsiniz.

```
#include<stdio.h>
#define UZUNLUK 20

int
main ()
{
     char dosya_ismi[UZUNLUK];
     FILE *girdi;
     int sayi,num, durum, bulundu;
     char c;

     /* Girdi dosyasinin ismini oku */
     printf("Girdi dosyasinin ismini girin: ");
     for (scanf("%s", dosya_ismi);
          (girdi = fopen(dosya_ismi,"rb")) == NULL;
          scanf("%s", dosya_ismi)) {
        printf("Hata: %s acilamadi \n", dosya_ismi);
        printf("Dosya ismini tekrar girin: ");
     }
```

```
    do
    {
        printf("Aradiginiz sayiyi girin: ");
        scanf("%d", &sayi);
        durum = fread(&num, sizeof(int), 1, girdi);
        bulundu = 0;
        while(!bulundu && (durum == 1)){
           if (sayi == num)
              bulundu = 1;
           durum = fread(&num, sizeof(int), 1, girdi);
        }
        if (bulundu)
           printf("Aradiginiz sayi dosyada bulundu.\n\n\n");
        else printf("Aradiginiz sayi dosyada bulunamadi.\n\n\n");
        printf("Baska sayi aramak ister misiniz? (E/H) ");
        scanf(" %c",&c);
        fseek(girdi,0,0);
    } while ((c == 'E')|| (c == 'e'));
    fclose(girdi);

    printf("%s dosyasi kapatildi \n", dosya_ismi);

    return 0;
}
```