

STANDART

C

PROGRAMLAMA DİLİ

Fedon Kadifeli
A. C. Cem Say
M. Ufuk Çağlayan

Özgün metin © 1990, 1988 M. U. Çağlayan, F. Kadifeli ve A. C. C. Say.

Genişletilmiş Türkçe baskı © 2000, 1993 F. Kadifeli, A. C. C. Say ve M. U. Çağlayan.

Her hakkı mahfuzdur. Bu yayının hiçbir bölümü, yazarların izni olmadan fotokopi, teksir veya başka bir yolla çoğaltılamaz, saklanamaz veya yayınlanamaz.

Kitabın İngilizcesi ilk defa Türkiye’de 1990 Ağustos’unda yayınlanmıştır.

Türkçeye uyarlayan ve güncelleyen Fedon Kadifeli.

Apple, Macintosh, MacWrite, MacDraw ve MacPaint, Apple Computer, Inc.’in onaylı markalarıdır.

Courier, Smith-Corona Corporation’in onaylı bir markasıdır.

IBM, International Business Machines Corp.’in onaylı bir markasıdır.

Microsoft, MS, MS-DOS, CodeView, QuickC ve Word, Microsoft Corp.’in onaylı markalarıdır.

OS/2, Microsoft’a lisanslı onaylı bir markadır.

PDP, Digital Equipment Corp.’in ticari bir markasıdır.

Times ve Times Roman, Linotype AG ve/veya şubelerinin onaylı markalarıdır.

UNIX, AT&T Bell Laboratories’in onaylı bir markasıdır.

Windows, Microsoft QuickBasic ve Visual C++, Microsoft Corp.’in ticari markalarıdır.

Ailelerimize

SADECE KİŞİSEL KULLANIM İÇİNDİR

ÖNSÖZ

C programlama dili, günümüzde en yaygın kullanılan dillerden biridir. C dili, COBOL, FORTRAN, PL/I gibi eski programlama dilleri grubuna dahil edilemeyeceği gibi, Ada, Modula-2 gibi yeni sayılabilecek dil grubuna da girmez.

C dili, klasik programlama dilleri kavramlarının, yeni, basit ve kolay kullanılır bir şekilde sağlandığı pratik bir dildir. C'nin popülaritesi daha çok UNIX'inki ile ilişkilidir, çünkü C, ilk olarak, UNIX dünyasının programlama dili olarak ortaya çıkmıştır. Bu popülarite, kişisel bilgisayarların çoğalması ve bunlar üzerindeki güçlü derleyici ile programlama ortamlarının kullanılabilir hale gelmesinden sonra daha çok artmıştır.

Bu ders kitabı, programlama dilleri üzerine Türkiye'de hazırlanan ders kitapları açığını kapatma amacını gütmektedir. Bilgisayar sistemlerine giriş, sistem programlama, sistem yazılımı ve işletim sistemleri gibi, C dilinin programlama projeleri geliştirilmesinde öğretildiği ve kullanıldığı dersler için tasarlanmıştır.

Kitap, üniversite birinci ve ikinci sınıfında okuyan öğrenciler için hedeflenmiştir, ancak daha yüksek düzeyde de bir başvuru kitabı olarak kullanılabilir. Bu kitabı okuyacakların, en az bir yapısal programlama dilini—örneğin Pascal, QuickBasic veya COBOL—bilmeleri ve veri yapıları konusunda temel bilgilere sahip olmaları beklenmektedir. Bu kitap, özellikle yazılım geliştirme projelerinde C dilini kullanmayı düşündükleri için, programlama dili bilgilerini genişletmek isteyen bilgi işlem uzmanları gibi kişilerin kendi başlarına okuyabilecekleri şekilde de hazırlanmıştır.

Bu ders kitabı, C programlama dilinin bütün yönlerini kapsaması açısından tamdır. Önemli miktarda örnek program ve alıştırma verilmiştir. Okuyucuların, en azından bir IBM kişisel bilgisayarını veya uyumlusunu ve Microsoft C Derleyicisi Uyarlama 5.00 veya daha yukarısını kullanabilecekleri beklenmektedir. Ancak, ekler dışında, kitapta verilen bilgiler değişik C programlama ortamları için de geçerlidir.

Bu kitabın Boğaziçi Üniversitesi Yayınları arasında İngilizce olarak yapılan ilk basımı halen Boğaziçi Üniversitesi Bilgisayar Mühendisliği Bölümü'nde İşletim Sistemleri dersinde yardımcı ders kitabı olarak kullanılmaktadır. Kitap şu anda tamamen

güncelleştirilmiştir ve ANSI Standardının tüm özelliklerini kapsamaya çalışmaktadır. Bu kitap sayesinde, yeni C derleyicileri tarafından desteklenen, Standart C'nin hem yeni hem de eski C programcıları tarafından kullanılmaya başlanacağını umuyoruz.

Bu kitabın hazırlanması ve yayınlanmasında değerli önerileri, düzeltmeleri, destekleri ve katkıları olan Rasim Mahmutoğulları, Sema Akgün, Oğuz Sinanoğlu, Ülku Karadağ, Ahmet Demirhan, Mustafa Elbir, Hasan Gültekin, Nezihe Bahar ve adını sayamayacağımız daha birçok kişiye teşekkürlerimizi borç biliriz.

Yük. Müh. F. Kadifeli
Y. Doç. Dr. A. C. C. Say
Doç. Dr. M. U. Çağlayan

İstanbul
Ekim 1993

İÇİNDEKİLER

ÖNSÖZ.....	vii
İÇİNDEKİLER.....	ix
ŞEKİLLER VE ÇİZELGELER LİSTESİ.....	xv
BÖLÜM 0: GİRİŞ	1
0.1. C Dilinin Kısa Bir Tarihi.....	1
0.2. C Dilinin Üstünlükleri.....	2
0.3. Kullanılan Sistem.....	3
0.4. Kitabın Yapısı Ve Kullanılan Kurallar.....	4
Problemler.....	4
BÖLÜM 1: TEMEL KAVRAMLAR VE GENEL BİR BAKIŞ	5
1.1. Açıklamalar, Tanıtıcı Sözcükler Ve Anahtar Sözcükler.....	5
1.2. Değişmezler.....	7
1.3. Temel Veri Tipleri Ve Tanımlar.....	9
1.4. #define Ve #include Önışlemci Emirleri.....	12
1.5. İşleçler, İfadeler Ve Atama Deyimleri.....	13
1.6. Tip Dönüşümü Ve Kalıplar.....	16

1.7. Basit Girdi/Çıktı	17
1.8. C Deyimleri Ve Program Gövdesi.....	21
1.9. Bir C Programı Nasıl İşletilir.....	22
1.10. Örnek Programlar	24
Problemler	26
BÖLÜM 2: DEYİMLER VE KONTROL AKIŞI.....	29
2.1. C Dilinin Deyimleri	29
2.2. Bağıntısıl Ve Mantıksal İşleçler.....	30
2.3. Doğruluk-Değerli İfadeler	32
2.4. if Deyimi Ve Koşullu İşleç	33
2.5. while Deyimi	35
2.6. do Deyimi	36
2.7. for Deyimi Ve Virgül İşleci.....	37
2.8. continue Deyimi.....	38
2.9. break Deyimi	39
2.10. goto Deyimi Ve Etiketler.....	39
2.11. switch Deyimi.....	40
2.12. Bir Örnek—Sayı Sıralama.....	42
Problemler	43
BÖLÜM 3: GÖSTERGELER VE BİT İŞLEME.....	45
3.1. Gösterge Değişkenleri Ve İşlemleri.....	45
3.1.1. & Ve * İşleçleri	46
3.1.2. Gösterge Değişkenleri Bildirimleri.....	47
3.1.3. Gösterge Aritmetiği	47
3.2. Göstergeler Ve Diziler.....	49
3.3. Karakter Dizileri	51
3.4. Bitsel İşleçler.....	53
3.5. İşleç Önceliği Ve Birleşme	57
Problemler	58

BÖLÜM 4: FONKSİYONLAR VE PROGRAM YAPISI	59
4.1. Fonksiyon Tanımlama	60
4.2. Fonksiyon Çağrılarını	62
4.2.1. Değer İle Çağrı	64
4.2.2. Referans İle Çağrı	65
4.2.3. main Fonksiyonunun Parametreleri	67
4.3. Bellek Sınıfları	69
4.3.1. auto Değişkenler	70
4.3.2. register Değişkenler	70
4.3.3. static Değişkenler Ve Fonksiyonlar	71
4.3.4. Fonksiyonlar Ve extern Değişkenler	72
4.3.5. İkleme	73
4.4. Özçağrı	75
4.5. Fonksiyonlara Göstergeler	78
4.6. Bir Örnek—8 Vezir Problemi	80
Problemler	84
 BÖLÜM 5: TÜRETİLMİŞ TIPLER VE VERİ YAPILARI.....	87
5.1. Sayım Tipleri	87
5.2. Yapılar	89
5.3. Yeni Tip Tanımlama	92
5.4. sizeof İşleci	94
5.5. Birlikler	95
5.6. Alanlar	96
5.7. Bellek Ayırma	98
5.8. Karmaşık Tipler	98
5.8.1. Dizi Dizileri	99
5.8.2. Dizilere Göstergeler	100
5.8.3. Gösterge Dizileri	101
5.8.4. Göstergelere Göstergeler	102
5.9. Bir Örnek—Dosya Sıralama	103

Problemler	107
BÖLÜM 6: ÖNİŞLEMCİ	109
6.1. #define Ve #undef Emirleri	110
6.2. #include Emri	112
6.3. Koşullu Derleme	113
6.4. Diğer Emirler	115
6.5. Önceden Tanımlanmış İsimler	115
6.6. Bir Örnek—ctype.h Başlık Dosyası	116
Problemler	117
BÖLÜM 7: DOSYALAR VE GİRDİ/ÇIKTI	119
7.1. Dosya Esasları	119
7.2. Dosya Erişimi—Başka Yöntemler	121
7.3. Rastgele Erişim	123
7.4. Dosyalarla İlgili Başka Bilgiler	124
7.5. Sistem İle İlgili Fonksiyonlar	125
7.6. Dosya Tanımlayıcıları Ve İlgili Fonksiyonlar	126
7.7. Bir Örnek—Öğrenci Veritabanı	128
Problemler	132
EK A: KARAKTER KODLARI ÇİZELGESİ.....	135
EK B: MICROSOFT C DERLEYİCİSİ HAKKINDA TEMEL BİLGİLER.....	145
B.1. Bellek Modelleri	145
B.1. QC Kütüphanesi	146
B.3. CL Eniyileştirici Derleyicisi	147
EK C: MICROSOFT CODEVIEW HATA DÜZELTİCİSİNE GENEL BİR BAKIŞ	151
EK D: MICROSOFT LIB VE NMAKE YARDIMCI PROGRAMLARINA GENEL BİR BAKIŞ.....	155

D.1. LIB Yardımcı Programı	155
D.2. NMAKE Yardımcı Programı	156
EK E: DİLLERARASI ÇAĞRILAR	159
E.1. Birleştirici İle Bağlayıcının Kullanılması	159
E.2. Satır içi Birleştiricisinin Kullanılması	162
E.3. Bir Örnek—Disket Saklama	163
EK F: STANDART C PROGRAMLAMA DİLİNİN DİĞER ÖZELLİKLERİ	169
F.1. C Dünyanın Her Yerinde—Yöre ler	169
F.2. Geniş Karakterler Ve Çokbaytlı Karakterler	170
F.3. Üçlü Karakterler	170
F.4. Zaman Fonksiyonları	171
F.5. Standart Başlık Dosyaları	173
F.6. Çevirme Sınırları	181
EK G: SEÇİLMİŞ PROBLEMLERE YANITLAR	183
EK H: TÜRKÇE-İNGİLİZCE VE İNGİLİZCE-TÜRKÇE TERİMLER SÖZLÜĞÜ	187
H.1. Türkçe-İngilizce Sözlük	187
H.2. İngilizce-Türkçe Sözlük	198
BİBLİYOGRAFYA	209
DİZİN	213

ŞEKİLLER VE ÇİZELGELER LİSTESİ

ÇİZELGE 2.1 C dilinin deyimleri.....	30
ÇİZELGE 2.2 C işleç önceliği ve birleşme.....	32
ŞEKİL 2.1 while ve do deyimleri için akış çizenekleri.....	36
ÇİZELGE 3.1 C işleç önceliği ve birleşme.....	57
ŞEKİL 4.1 main fonksiyonuna geçirilen komut satırı argümanları.....	69
ŞEKİL 5.1 Bir örnek ikili ağaç.....	104

BÖLÜM 0: GİRİŞ

```
/* İlk program */
#include <stdio.h>

void main (void)
{
    int kar;
    if ((kar=getchar())!='\n')
        main();
    putchar(kar);
}
```

Eğer bir C programının nasıl görüldüğünü merak ettiyseniz, yukarıda, “yararlı ve anlamlı” bir iş yapan tam bir C programının durduğunu öğrenmek sizi şaşırtabilir. Bir C derleyiciniz varsa ve nasıl kullanacağınızı biliyorsanız, bu programı yazın, derleyin ve çalıştırın. Fakat önce, programın ne yapabileceği konusunda çılgın tahminlerde bulunmaktan da çekinmeyin.

...

Eğer tahmininiz doğru çıkmadıysa, üzülmeyin. Bu, sadece C hakkında ilginizi çekmek içindi ve umarız öyle oldu! Bu basit gibi görünen, ancak C’nin birtakım ileri özelliklerini kullanan programı, kitabın yarısını bitirinceye kadar anlamamanızı beklemiyoruz ve bu kitabı bitirdiğinizde çok daha karmaşık programları bile anlayabileceğinizi ve yazabileceğinizi bekliyoruz.

0.1. C Dilinin Kısa Bir Tarihi

UNIX işletim sistemi ile C programlama dili birbirleriyle yakından ilişkilidir. Tarihleri 70’lerin başında başlar. İlginç olan şey de, AT&T Bell Laboratuvarları’ndan Ken Thompson tarafından yazılan bir bilgisayar oyun programından kaynaklanmalarıdır.

Thompson, programını bir PDP-7 bilgisayarına uyarlamak istediğinde, bu küçük makina hakkında çok şey öğrendi, ancak işletim sistemini pek beğenmedi. Bunun üzerine, o zamanlar daha büyük bir makinada kullanılan MULTICS işletim sisteminin basitleştirilmiş ve değiştirilmiş bir uyarlamasını yazmaya karar verdi. Daha sonra, Dennis M. Ritchie de ona katıldı ve Brian W. Kernighan tarafından UNICS (Uniplexed Information and Computing Service—Birleştirilmiş Bilgi ve Hesaplama Hizmeti) adı verilen işletim sisteminin ilk uyarlaması doğdu. Bu üç kişi C ve UNIX'in tarihinde en önemli rolü oynadılar. Başlangıçta, Thompson, daha önceleri 1967 civarında geliştirilen BCPL adlı “tipsiz” dilden de büyük ölçüde etkilenecek B dilini tasarımıladı. Bundan sonra, Ritchie UNIX'i daha kolay bir şekilde yazma amacıyla C adında yeni bir dil tasarımıladı. 1973 yılında ise Ritchie ve Thompson C'yi kullanarak UNIX'i yeni baştan yazdılar. Sonuç o kadar iyiydi ki, 1983'te ACM'in Turing Ödülü'nü almaya hak kazandılar.

O zamandan beri, C çok değişmedi. Dilde yapılan bazı küçük genişletmeler, Dennis M. Ritchie tarafından hazırlanan *The C Programming Language—Reference Manual* (C Programlama Dili—Başvuru Elkitabı) adlı, 1983 basımlı, Bell Laboratuvarları yayınında anlatılmaktadır. ANSI'nin (American National Standards Institute—Amerikan Ulusal Standartlar Enstitüsü) X3J11 komitesi tarafından 1988 Ekim'inde sunulan, C Standardının son taslağı bu değişiklikleri resmileştirmekte ve kendi başına yenilerini eklemektedir. Yapılan düzenlemelerle, birtakım programcı hatalarını azaltmak için derleyici kontrolleri artırılmış ve dile yararlı birkaç özellik daha katılmıştır. Ancak, tasarımcılarının felsefesine aykırı olduğu için dilin daha fazla genişletilmesi beklenmemelidir. Bunun yerine, bu dilden yeni diller ortaya çıkmaktadır. Bir örnek, C++'dır. Bu dil, nesneye yönelik programlama ve veri soyutlama teknikleri sağlarken, Standart C ile uyumlu kalmaya çaba göstermektedir. Bu özellikleri, dili daha kapsamlı bir uygulama programlama dili yapmaktadır. Yakın zamanda ise C++'nın C'nin yerini alması beklenmektedir, ancak yine de C++ öğrenecek birisinin önce Standart C'yi bilmesi gerekmektedir.

0.2. C Dilinin Üstünlükleri

Önceki kısımdan, C'nin pratik gereksinimlerden ortaya çıktığı sonucuna varabiliriz; yani belirli bir sistem için yapay hazırlanmış bir dil değildir. C, birleştirici dilinin sorunlarını kısmen çözmek için, düşük düzeyli programlamayı destekleyici kolaylıkları olan yüksek düzeyli bir dil olarak tasarımılanmıştır. Örneğin, dilde girdi/çıkı deyimleri yoktur. Kullanıcı, girdi/çıkı yapmak için “getchar” ve “putchar” gibi bazı *fonksiyonları* çağırır. Derleyici bu fonksiyonların anlamı hakkında hiçbir şey bilmez. Sadece, C kütüphanesinde tanımlı olan bu fonksiyonlara çağrılar üretir. Bu da, iyi bir kütüphanenin C'ye çok şey kazandıracağı anlamına gelir.

C'nin diğer dillere göre bazı avantajları vardır. Bunlar aşağıda özetlenmektedir:

C, kısa, özlü, verimli, esnek ve ifadeli bir dildir. Az sayıda anahtar sözcüğe sahiptir, fakat doğru kontrol yapıları, güçlü işlemleri (diğer adıyla, işlem operatörleri) ve kolayca birleştirilen veri tipleri vardır. Bu da, dili öğrenmenin ve bir C derleyicisinin yazılmasının kolay olduğu ve bu dilde yazılan programların kısa, fakat bazen izlemesi zor olduğu

anlamına gelir. Bazı işleçler diğer dillerde yoktur, fakat bunlar kolayca makine diline çevrilebilirler, bu da C dilinde yazılan programların diğer dillerde yazılanlara göre daha verimli çalışmalarının nedenini açıklar. Bundan dolayı, bazı sistemlerde C, birleştirici dilinin yerini almıştır.

C, popüler bir işletim sistemi olan UNIX'in temel dilidir. Bu da, en azından bu işletim sisteminde, bu dili vazgeçilmez kılmaktadır. Buna rağmen, C başka sistemlerde de kullanılmaya başlanmıştır ve, taşınabilirlik özelliğinden dolayı, bir sistem için yazılmış programlar kolayca başka sistemlere de aktarılabilen ve orada bazı ufak tefek değişikliklerden sonra derlenip doğru bir şekilde çalıştırılabilmektedir. Bu durum dilin amaçlarından biridir: programcının makine bağımlılıklarını bir tarafta ayırması ve gereksinim duyduğunda programı yeni ortamlara kolayca uyarlayabilmesi. C önilemcisi bu konuda önemli bir rol üstlenmektedir.

C modüler programlamayı teşvik etmektedir. Diğer çağdaş programlama dilleri kadar yaygın olmamasına rağmen, programcının bunu sağlaması için bazı seçimler sunar. Çeşitli bellek sınıfları çeşitli düzeylerde gizlilik ve modülerite sağlar. Dildeki tek modül yapısı olan fonksiyon tanımı için C tek bir düzeye izin verir; bütün fonksiyonlar dışsaldır. Programcı, kolayca, kullanıcı tarafından tanımlanmış kaynak veya amaç kütüphaneleri yaratabilir ve bu yolla çok büyük programlar hazırlayabilir.

C'nin dezavantajları da vardır. C dilinde yazılan programların izlenmesi bir miktar zor olabilir, çünkü zengin işleç kümesi program okunaklılığını azaltır. C katı-tiplenmiş bir dil değildir; bir dizinin sınırları dışında indisleme yapmaya çalışmak gibi, bazı programcı hataları için yürütme zamanı desteği sağlamaz; bazı durumlarda, derleyici, ifadeler içindeki alt-ifadelerin veya argüman listeleri içindeki ifadelerin hesaplanma sıralarını değiştirebilir; aynı simgelerin birden fazla amaca hizmet etmesi bazı programlama hatalarına yol açabilir—eşitlik testi ve atama işleçlerinin karıştırılması gibi; bazı yapılar—örneğin **switch** deyimi—daha iyi tasarlanabilirdi. Bu sorunların bazıları, C'nin ANSI Standardını izleyen bazı yeni derleyicilerde iyileştirilmiştir. Ayrıca, C'ye dayanarak geliştirilen C++ dili, bazı sorunları çözmüş ve birçok yeni özellikler katmıştır. Gerçek bir C programcısının, C'nin dezavantajları ile birlikte yaşamayı öğrenmesi gerektiğine inanıyoruz.

0.3. Kullanılan Sistem

C “taşınabilir bir dil” olmasına rağmen—ki, bu da, belirli bir sistem için yazılmış olan bir C programının başka bir sisteme aktarıldığında orada başarılı bir şekilde derlenip yürütülebileceği anlamına gelir—sistemler arasında bazı farklılıklar olabilmektedir. Bu kitapta verilen program veya program parçaları Microsoft QuickC Uyarlama 1.01 (1988), Microsoft C Derleyicisi Uyarlama 5.00 (1987) ve Microsoft C/C++ Eniyileştirici Derleyicisi Uyarlama 8.00 (1993) kullanılarak denenmiştir. Kullandığımız işletim sistemi IBM uyumlu bir PC'de çalışan MS-DOS Uyarlama 5.00 (1991) veya daha yukarısidir. Bir sonraki bölümde bu ortamların nasıl kullanılabileceği konusunda daha detaylı bazı bilgiler verilmiştir. Ayrıca Ek B'ye bakınız.

0.4. Kitabın Yapısı Ve Kullanılan Kurallar

Bu kitabın İngilizce olan ilk uyarlaması bir Apple Macintosh Plus'ta Word ve MacDraw isimli yazılımlar kullanılarak hazırlanmıştır. Kitap en son olarak, IBM uyumlu bir PC'de çalışan Windows için Word Uyarlama 2.0c kullanılarak güncelleştirilmiş ve Türkçe baskıya hazır hale getirilmiştir. Temel yazı tipi Times Roman'dır. Sözdizimsel gösterimde, sözdizimsel sınıflar *italik* yazı stili ile gösterilmiştir. İsteğe bağlı bölümler, arkalarında satır altına yazılan *opt* simgesiyle gösterilmiştir. Program bölümleri, ekran veya yazıcı çıktısına benzetilmek için, Courier yazı tipiyle yazılmıştır. C anahtar sözcükleri **koyu**, program açıklamaları ise *italik* ile yazılmıştır. Bilgisayar girdi/çıktısı da Courier yazı tipiyle gösterilmiştir; çıktı, girdiden ayırt edilmesi için, alt çizgili yazılmıştır.

Kitabın geri kalan kısmında 7 bölüm, 8 ek ve C programlama diliyle ilgili bazı kaynakların liste halinde verildiği bir bibliyografya vardır. Bölüm 1'de dile bir giriş yapılmakta ve çok basit programlar yazmak için gerekli olan genel bilgiler verilmektedir. Bölüm 2, C dilindeki kontrol deyimlerini ve doğruluk-değerli, yani mantıksal, ifadeleri anlatmaktadır. Bölüm 3 göstergeler ve bit işlemleri hakkındadır. Bölüm 4'te fonksiyonlar ve değişkenlerle fonksiyonlara uygulanabilecek çeşitli bellek sınıfları anlatılmaktadır. Bölüm 5, karmaşık veri yapıları tanımlamada anahtar olan, bütün türetilmiş veri tiplerini kapsamaktadır. Bölüm 6, C önişlemcisine ayrılmıştır ve bütün önişlemci emirlerini anlatmaktadır. Bölüm 7'de, diğer bazı fonksiyonlarla beraber, Standart C kütüphanelerinde tipik olarak rastlanan girdi/çıkıtı işlemleri anlatılmaktadır. Ekler ise Microsoft C Derleyicisi ve çevresinin bazı özelliklerini anlatmakta, bazı problemlerin yanıtlarını vermekte ve kitapta kullanılan terimlerin İngilizce karşılıklarını bulmak için bir Türkçe-İngilizce ve İngilizce-Türkçe terimler sözlüğü içermektedir. Kitapta, olanaklar dahilinde, yaygın olarak kullanılan Türkçe terimler tercih edildiği için, İngilizce bilgisayar terimleri hakkında bilgi sahibi olan okuyucu, bu kitapta kullanılan Türkçe terimleri anlamak için sık sık Ek H'de verilen bu sözlüklere başvuracaktır.

Problemler

1. Bu bölümün başında verilen programı sisteminizde çalıştırın. Eğer herhangi bir zorlukla karşılaşırsanız sisteminizin yardım özellikleri, elkitabları veya deneyimli programcılara başvurun. Bu programı çalıştırmak için kaç komuta gereksininiz var?
2. Bir önceki alıştırmadaki programı tekrar yazın (veya değiştirin), fakat bu kez, iki satırı, tek satırda, arada bir boşluk karakteri bırakarak yazın. # ile başlayan satırın arkasına bir şey yazmamaya dikkat edin. Derleyicinin aynı şekilde kabul etmesi gerekir. Şimdi, program içinde, rastgele yerlere boşluklar, satır başları veya duraklar (tab) ekleyin. Derlerken ortaya çıkacak hata mesajları varsa, bunları anlamaya ve düzeltmeye çalışın.

5

Program açıklamaları bu kitapta *italik* olarak yazılmış ve sistemler arasındaki uyumsuzluklardan kaçınmak için Türkçe karakterler kullanılmamıştır.

Tanıtıcı Sözcükler

Değişkenler, deyim etiketleri, tip isimleri, fonksiyon isimleri gibi, programcı tarafından oluşturulan bütün nesneleri isimlendirmek için *tanıtıcı sözcükler*, bir diğer adıyla, *program isimleri* kullanılır. Tanıtıcı sözcükler için şu kurallar geçerlidir:

1. Herhangi bir sayıda karakterlerden oluşur, ancak ilk 31 karakter dikkate alınır.
2. İlk karakter bir harf veya altçizgi (_) olmalıdır.
3. Geri kalan bütün karakterler bir harf, rakam veya altçizgi olabilir.

Örnek olarak, A12, a12, sayfa_basi değişken ismi olarak kullanılabilir. Küçük ve büyük harflerin farklı olduğuna dikkat ediniz, yani A12 ile a12 farklı tanıtıcı sözcüklerdir. Ancak, bazı ortamlarda küçük büyük harf ayırımı yapılmayabilir. Bundan dolayı, aynı program içinde, harf ayırımı dışında, birbirine benzeyen iki farklı isim kullanmaktan kaçının. Ayrıca, bir tanıtıcı sözcükte, ilk karakter olarak altçizgiden kaçınılması önerilir, çünkü bu tip isimler derleyiciye özgü bazı anahtar sözcükler veya diğer isimler için kullanılmaktadır.

Anahtar Sözcükler

C dilinde 32 adet *anahtar sözcük* vardır; hepsi küçük harfle yazılır. Anahtar sözcükler tanıtıcı sözcük olarak kullanılamazlar; kendilerine özgü kullanım alanları vardır. C dilindeki bütün anahtar sözcüklerin sınıflandırılmış bir listesi aşağıda verilmiştir. Program içinde kullanacağınız isimlerin aşağıdaki listede olmamasına dikkat edin.

<u>veri tipi</u>	<u>bellek sınıfı</u>	<u>deyim</u>	<u>işlec</u>
char	auto	break	sizeof
const	extern	case	
double	register	continue	
enum	static	default	
float	typedef	do	
int		else	
long		for	
short		goto	
signed		if	
struct		return	
union		switch	
unsigned		while	
void			
volatile			

Kullanılan makine ve derleyiciye bağlı olarak, C dilinin özel durumundan dolayı başka anahtar sözcükler de olabilir. Bunlar genelde altçizgi karakteriyle başlarlar.

1.2. Değişmezler

C dilinde, tamsayı, kayan noktalı, yani gerçek sayı, karakter ve karakter dizisi değişmezleri bulunur.

Tamsayı Değişmezleri

Tamsayı değişmezleri, program içinde, ondalık, sekizli veya onaltılı sayılar şeklinde belirtilebilirler ve derleyicinin tipine göre, 16 bit veya 32 bit şeklinde saklanırlar. Aşağıdaki örneklerde bazı tamsayı değişmezleri görülmektedir:

123	ondalık 123
0123	sekizli 123 = ondalık 83
083	geçersiz bir sayı
0x123	onaltılı 123 = ondalık 291
0XFF	onaltılı FF = ondalık 255

Sayının önündeki bir sıfır rakamının, geri kalan rakamların sekizli (yani, 0-7) ve sayı önündeki bir sıfırla onun arkasından gelen küçük veya büyük x harfinin, geri kalan rakamların onaltılı (yani, 0-9, A-F veya a-f) olması gerektiğini gösterdiğine dikkat edin. Sayının program içindeki gösterimi ne olursa olsun, makine içinde her zaman bitler halinde ikili sistemde saklanır; ancak bu durum genelde programcıyı ilgilendirmez.

Bir tamsayı değişmezi, eğer değeri 16 bite sığıyorsa, kısa formda (16 bit) saklanır, aksi takdirde uzun formda (32 bit) saklanır. Tamsayı bir değişmezin uzun formda saklanmasını zorlamak için değişmezin arkasına l veya L harfi eklenmelidir. l sayısı ile karıştırılmaması için, küçük l yerine büyük L harfinin kullanılması önerilir.

123	16 bite saklanır
123l	32 bite saklanır
123L	32 bite saklanır
077	000077 şeklinde, 16 bite saklanır
077L	0...077 şeklinde, 32 bite saklanır
0xFFFF	16 bite saklanır
0xFFFFL	0000FFFF şeklinde, 32 bite saklanır
0xFFFFF	000FFFFF şeklinde, 32 bite saklanır

Bir sekizli veya onaltılı tamsayı değişmezi, daha büyük bir bellek alanına yerleştirildiği zaman, soluna sıfır konur. Yani böyle değişmezlerin işaretersiz olduğu varsayılır. Ondalık bir tamsayı değişmezinin işaretersiz olarak işlem görmesini sağlamak için arkasına u veya U eki konulmalıdır, örneğin 65000u.

Kayan Noktalı Sayı Değişmezleri

Kayan noktalı sayı değişmezleri ya tamsayı ile kesir kısmı arasına nokta konarak yada bilimsel gösterimde belirtilirler. İkinci yöntem, genelde, çok büyük veya çok küçük sayılar için kullanılır. Aşağıda bazı örnekler vardır:

1.123	
1.23E20	= 1.23×10^{20}
1.23e20	= 1.23×10^{20} (büyük veya küçük harf olabilir)
123E18	= 1.23×10^{20}
1.23E-20	= 1.23×10^{-20}

Değişmezin içinde herhangi bir boşluğun olmaması gerektiğine dikkat edin. Normalde, kayan noktalı değişmezler 8 baytta, yani **double** (çift) duyarlılıkta, saklanır. Eğer kayan noktalı değişmezde **f** veya **F** eki bulunuyorsa, o zaman 4 baytta, yani tek duyarlılıkta; eğer **l** veya **L** eki kullanılırsa, o zaman **long double** (uzun çift veya dörtlü) duyarlılıkta saklanır.

Karakter Değişmezleri

Bir *karakter değişmezi*, 'A', 'a', '%' gibi tırnak işaretleri arasına konulan tek bir karakter veya tek bir karaktere eşdeğer olan bir kaçış sırasıdır. Bir *kaçış sırası* bir ters bölü işareti ile bir harf veya rakamlardan oluşur. C dilinde sadece şu kaçış sıraları kullanılır:

\n	yeni satır
\b	geri alma
\r	satırbaşı
\t	durak (tab)
\f	sayfa ilerletme
\v	dikey durak
\a	zil
\'	tek tırnak
\"	çift tırnak
\?	soru işareti
\\	ters bölü
\ddd	sekizli kodu ddd olan karakter
\xhhh	onaltılı kodu hhh olan karakter (Ek A'ya bakınız)

Ek olarak, aşağıda bazı karakter değişmezleri örnekleri vardır:

'\0'	boş karakter
'\''	bir karakter değişmezi olarak tek tırnak
'\"'	bir karakter değişmezi olarak çift tırnak
'\'	bir karakter değişmezi olarak ters bölü
'\101'	sekizli kodu 101 olan karakter (ASCII sisteminde 'A' harfi)
'\x041'	onaltılı kodu 41 olan karakter (ASCII sisteminde 'A' harfi)

Bir karakter değişmezinin tipi **int**'tir. Karakterler ise tipik olarak bir baytta (8 bit) saklanır ve tamsayı gibi işlem görürler. Ancak **signed** (işaretli) veya **unsigned** (işaretsiz) oldukları veya kullanılan kodlama sistemi C Standardında belirtilmemiştir. Bizim sistemimizde, ASCII kodlama sistemi kullanılır (Ek A'ya bakınız). Normal karakterlerin (örneğin, sistemimizdeki 7 bitlik ASCII karakterlerinin) işaretsiz oldukları

garanti edilmiştir. Gerektiğinde, bir karakter değişkeninin işaretli olup olmadığı programcı tarafından açıkça belirtilmelidir.

Karakter Dizisi Değişmezleri

Bir *karakter dizisi değişmezi* çift tırnaklar arasında yazılmış herhangi bir sayıda karakter veya yukarıda listesi verilmiş kaçış sırasından oluşur. İşte bazı örnekler. Son örnekteki, yeni satır (\n) ve tek tırnak (\') kaçış sıralarına dikkat edin.

""	boş karakter dizisi
"Merhaba"	7 karakterlik bir karakter dizisi
"İsminizi girin,\nveya ENTER\'a basın"	iki satırdan oluşan bir karakter dizisi

Bir karakter dizisi değişmezi bellekte saklandığı zaman, dizinin sonuna otomatik olarak *boş karakter* (\0) eklenir; bundan dolayı diziyi saklamak için kullanılan bayt sayısı dizinin uzunluğundan bir fazladır. Bir program içinde, aralarına hiçbir işaret koymadan peşpeşe yazılan karakter dizisi değişmezleri birleştirilir ve tek bir dizi değişmezi olarak alınırlar. Diziler, bazı sistemlerde, salt-okunur belleğe yerleştirilebilirler, bu da onları değiştirilemez kılabılır. Ayrıca, birbirine tıpatıp benzeyen iki veya daha fazla karakter dizisi aynı bellek bölgesinde saklanabilir.

Sistemler arasındaki uyumsuzluklardan kaçınmak için, bu kitaptaki karakter dizisi değişmez örneklerinde Türkçe karakterler kullanılmamıştır.

Bir Sonraki Satıra Devam Etme

Bir deyim veya karakter dizisi değişmezi programın tek bir satırına sığmıyorsa ve bir sonraki satıra devam etmek gerekiyorsa, satırın sonuna ters bölü (\) işareti konup bir sonraki satıra devam edilebilir. Ayrıca, uzun bir karakter dizisi değişmezi iki satırda iki ayrı karakter dizisi şeklinde de yazılabilir.

1.3. Temel Veri Tipleri Ve Tanımlar

Bir değişken ismi, değişkenin alacağı değerlerin türü ve işlevini yansıtacak şekilde dikkatlice seçilen bir tanıttıcı sözcüktür. Genelde, `kalan_gunler` veya `kalanGunler` şeklinde bir değişken ismi `x132` gibi bir isme tercih edilmelidir. Değişken isimlerinde büyük harfler yerine küçük harflerin kullanılması alışlagelmiştir.

Bir C programında, kullanılmadan önce, tüm değişken ve fonksiyonların tanımı veya bildirimi yapılmalıdır. Temel veri tiplerinin, yani tamsayı, kayan noktalı sayılar ve karakterlerin bildiriminde kullanılan anahtar sözcükler şunlardır:

int	tamsayı
signed, unsigned	işaretili veya işaretsiz tamsayılar
short, long	kısa veya uzun tamsayılar
float, double	tek veya çift duyarlılıklı kayan noktalı sayılar
char	karakter

Bir tanım, bir tip ismi ile tanımlanmakta olan nesnelerin virgülle ayrılmış listesinden oluşur ve bir noktalı virgül ile sona erer. Aşağıda birkaç örnek tanım gösterilmiştir:

```
int    x;
int    x1, y1, z1;
long   d, d1;
char   c;
char   c1, c2, c3;
float  a;
float  a1, a2, a3;
int    u[3];
float  k[10*20];
```

Son iki tanımda, tek boyutlu ve üç elemanlı bir tamsayı dizisi ile tek boyutlu ve 200 elemanlı kayan noktalı sayılardan oluşan bir dizi tanımlanmıştır. Dizinin boyunun derleme esnasında hesaplanabilen *değişmez bir ifade* olduğuna dikkat edin.

Aşağıda görüldüğü gibi, değişkenler tanımlandıkları zaman ilk değerleri de verilebilir:

```
int    x = 0;
int    x1 = 10, x2 = 20, x3 = 30,
        x4 = 60 * 60; /* ilk deger degismez bir ifadedir */
char   c1 = 'a', c2 = 'z';
```

Buna *ilkeme* diyoruz. İlk değer olarak bir ifadenin de yazılabileceğine dikkat edin. Tanımlanan değişkenlerin *ilklenmesi* iyi bir alışkanlıktır.

İsimlendirilmiş *değişmezler*, yani değerleri değiştirilemeyecek olan değişkenler, **const** tip *niteleyicisi* kullanılarak tanımlanırlar:

```
int const x = 100;
char const ys = '\n';
float const a = 123.45;
```

Bu tip *değişmez* “değişken”lerin **const** ile tanımlanmasının en azından iki avantajı vardır:

1. Programcı yanlışlıkla bu tip bir değişkene atama yapmaya kalkar veya değerini değiştirebilecek bir şekilde kullanmaya kalkarsa, derleyici onu uyaracaktır.
2. Çok kullanıcıli sistemlerde bu tip değişkenlerin ortak ve değiştirilemez bir bellek kesimine yüklenmesi sağlanabilir.

Aynı anda başka bir süreç (program) tarafından kullanılan veya değiştirilebilen değişkenlerin, derleyicinin olası bir eniyileme yapmasını engellemek için, **volatile** tip

niteleyicisi kullanılarak tanımlanması gerekmektedir. Örneğin, çok kullanıcı ortamlarında iki değişik süreç tarafından ortak kullanılan bir değişkenin **volatile** tanımlanması gerekir.

Tanımlanmış bir değişken, bir deyim içinde ismi verilerek anılır. Dizi elemanlarına ulaşmak için çeşitli yollar olmasına rağmen, çoğu zaman dizi isminin arkasına köşeli parantezler içinde indis belirtilerek kullanılırlar. Eleman sayısı **BOY** olan bir dizi için indisin alabileceği değerler 0'dan **BOY-1**'e kadar değişir. Yani, yukarıda tanımlanmış olan **u** tamsayı dizisinin elemanları **u[0]**, **u[1]** veya **u[2]** şeklinde kullanılabilir. İndisin her zaman değişmez bir tamsayı olmasına gerek yoktur; genelde bir ifade olur. Yani, **u[x+5]** geçerli bir kullanımdır. Doğal olarak, eğer "**x+5**" ifadesinin değeri 0, 1 veya 2 değilse, beklenmeyen sonuçlar elde edilebilir. Bu tip kullanımları kontrol altında tutmak tamamen programcının sorumluluğundadır.

Değişik tiplerde tanımlanmış değişkenler için, *mevcut* derleyiciler tarafından ayrılan bellek miktarı ile alt ve üst limit değerleri şöyledir:

<u>tip</u>	<u>anlamı</u>	<u>bayt sayısı</u>	<u>limitler</u>
char	?	1	?
--	signed char	1	-127..127
--	unsigned char	1	0..255
short	signed short int	2	(1)
int	signed int	2 veya 4	(1) veya (2)
long	signed long int	4	(2)
signed short	signed short int	2	(1)
signed	signed int	2 veya 4	(1) veya (2)
signed long	signed long int	4	(2)
unsigned short	unsigned short int	2	(3)
unsigned	unsigned int	2 veya 4	(3) veya (4)
unsigned long	unsigned long int	4	(4)
float	--	4	(5)
double	--	8	(6)
long double	--	8 veya 10	(6) veya (7)

(1) $-2^{15} \dots 2^{15}-1 = -32\,768 \dots 32\,767$. (16 bitlik bilgisayarlar için)

(2) $-2^{31} \dots 2^{31}-1 = -2\,147\,483\,648 \dots 2\,147\,483\,647$. (32 bitlik bilgisayarlar için)

(3) $0 \dots 2^{16}-1 = 0 \dots 65\,535$. (16 bitlik bilgisayarlar için)

(4) $0 \dots 2^{32}-1 = 0 \dots 4\,294\,967\,295$. (32 bitlik bilgisayarlar için)

(5) $-10^{38} \dots -10^{-38}, 0, 10^{-38} \dots 10^{38}$.

(6) $-10^{308} \dots -10^{-308}, 0, 10^{-308} \dots 10^{308}$.

(7) $-10^{4932} \dots -10^{-4932}, 0, 10^{-4932} \dots 10^{4932}$.

Not: 16 bitlik bilgisayarlar veya derleyicilerde (**signed** veya **unsigned**) **int** 16 bittir; 32 bitliklerde ise 32 bittir.

Tam sayılar ve kayan noktalı sayılar için her bilgisayarda farklı olabilecek bu özellikler, tam olarak `limits.h` ve `float.h` adlı başlık dosyalarında tanımlanmıştır. Birtakım varsayımlar yapmak yerine, bu başlık dosyalarında tanımlanmış bulunan değişmezlerin kullanılması özellikle önerilir. (Başlık dosyaları için bir sonraki kısma ve ayrıca Kısım F.5'e bakınız.)

1.4. #define Ve #include Önışlemci Emirleri

Hemen bütün C derleyicileri, özel önışlemci emirlerini tanıyan bir önışlemciyi yapılarında bulundurlar. Önışlemci emirleri, bir bakıma, C derleyicisinin girdisini, yani kaynak kodu, kontrol etmede kullanılır. Bir önışlemci emrinin ilk karakteri her zaman numara işaretidir (#) ve kaynak programda satırın ilk karakteri olmalıdır. Normalde, önışlemci emirlerinin çoğu, kaynak programın başına, bildirimlerden önce yazılır.

C programlarında çokça kullanıldıkları için, bu bölümde sadece `#define` ve `#include` önışlemci emirleri anlatılmaktadır. Diğer önışlemci emirlerinin daha detaylı bir anlatımı Bölüm 6'da verilmektedir.

`#define` emri şu şekildedir:

```
#define tanıtıcı_sözcük karakter_dizisi
```

Bu tür bir `#define` emri, emirden sonra gelen program deyimlerinde *tanıtıcı_sözcük* bulunan her yerde, onun yerine *karakter_dizisinin* konulacağını gösterir. Örneğin,

```
#define XYZ 100
```

emri, daha sonra, XYZ'nin her rastlandığı yerde 100 konulmasını sağlar. Program içinde kullanılacak değişmezleri tanımlamanın yaygın bir yolu da budur. `#define` emirleriyle tanımlanan tanıtıcı sözcüklerin büyük harfle yazılması alışılmalıdır.

```
#include emri ya
```

```
#include "dosya_adi"
```

yada

```
#include <dosya_adi>
```

şeklinde olur ve önışlemciye `#include` satırının yerini, belirtilen dosyanın alacağını gösterir. Eğer *dosya_adi* çift tırnak içine alınmışsa, o zaman önışlemci, dosyayı kaynak program dosyasının saklandığı alt dizinde arar. Eğer *dosya_adi* açılı parantezler içinde ise, o zaman dosya, böyle `#include` emirleri için aramaların yapıldığı "standart" alt dizin(ler)de aranır. Normalde, bu `\include` alt dizini olur.

`#include` emri, daha önceden hazırlanan, standart veya kullanıcı tarafından tanımlanan dosyalarda saklanan, sık kullanılan veri ve fonksiyon bildirimlerini programa dahil etmede kullanılır. Böyle dosyalara *başlık dosyaları* adı verilir ve isimleri *dosyaadi.h* şeklinde olur.

1.5. İşleçler, İfadeler Ve Atama Deyimleri

C dili, sağladığı işleçler (işlem operatörleri) açısından çok zengindir. İşletilebilir C deyimlerinin çoğu bir ifade şeklinde yazılır. Bu *ifade* belirli kurallara uygun olarak oluşturulmuş bir işlenenler ve işleçler karışımıdır. Diğer programlama dillerine karşılık, C dilinde bir atama bile özel bir çeşit ifadedir.

Bir ifade içinde, bir *işlenen* ya bir değişmez, ya bir değişkenin kullanılması, ya bir fonksiyon çağırısı yada başka bir (alt)ifade şeklinde karşımıza çıkar. *İşleçler* (işlem operatörleri), aritmetik işlemler, atama, bağıntısal ve mantıksal karşılaştırmalar, bit işlemleri, adreslerle ilgili işlemler ve başka işlerde kullanılırlar. C’de kullanılan aritmetik işleçler ve anlamları liste şeklinde aşağıda verilmiştir:

+	toplama veya tekli artı
-	çıkarma veya olumsuzlama
*	çarpma
/	bölme
%	kalan (5%2 ifadesinin değeri, 5/2’den arta kalan, yani 1’dir. İşlenenler olumlu tamsayılar olmalıdır.)
++	anlamı, işlecin, işlenenin önünde veya arkasında olmasına bağlı olarak değişir; ancak sonuçta işlenenin değeri bir artırılır
--	anlamı, işlecin, işlenenin önünde veya arkasında olmasına bağlı olarak değişir; ancak sonuçta işlenenin değeri bir azaltılır

Basit bir atama deyimini şu şekildedir:

değişken = *ifade*;

ve *ifadenin* değerinin hesaplandıktan sonra, *değişkenin* değerinin buna eşitleneceği anlamına gelir. *İfadeden* sonraki noktalı virgüle dikkat edin. Bu noktalı virgül deyimini sonuçlandırır.

İşte bazı örnekler. Son iki örneğin, aynı anlamı taşıdığına dikkat edin.

```
a = b + 10;
c = d + c * e - f / g + h % j;
x = y * sin(z - 3.14);
z = u[2] * u[0];
u[1] = -123.45;
x = 10;
(x) = 10;
```

C dilinde, diğer programlama dillerine karşılık, atamanın kendisi de bir ifadedir, bundan dolayı bir değere sahiptir. Bir atama ifadesinin değeri, değeri hesaplanarak sol taraftaki değişkene atanan ifadenin değeridir. Atama ifadesinin tipi ise soldaki işlenenin tipiyle aynıdır. Bu olgu, C dilinde çok kullanılır. Örneğin, bir ifadenin değeri aynı anda birden fazla değişkene, şu şekilde, atanabilir:

```
a = b = c = 0;
```

c değişkeni 0'ın değerini, b c'nin değerini ve a b'nin değerini alır; böylece bütün değişkenler 0'a eşitlenir.

Aşağıda ++ ve -- işleçlerini açıklamak için bazı örnekler verilmiştir:

$x = y++;$	y'nin değeri önce x'e atanır, sonra da bir artırılır. Bu şuna eşdeğerdir: $x = y;$ $y = y+1;$
$x = ++y;$	y'nin değeri önce bir artırılır, sonra da x'e atanır. Bu şuna eşdeğerdir: $y = y+1;$ $x = y;$
$x = y--;$	y'nin değeri önce x'e atanır, sonra da bir azaltılır. Bu şuna eşdeğerdir: $x = y;$ $y = y-1;$
$x = --y;$	y'nin değeri önce bir azaltılır, sonra da x'e atanır. Bu şuna eşdeğerdir: $y = y-1;$ $x = y;$

++ ve -- işleçleri *yan etkisi* olan işleçlerdir, çünkü tek bir atama deyimi ile birden fazla değişkenin değeri değiştirilebilir. Programcıların bu işleçleri karmaşık ifadeler içinde kullanmaları uygun değildir, çünkü böyle bir kullanım nispeten daha zor anlaşılabilir ve hataların daha zor düzeltildiği programların ortaya çıkmasına neden olur.

Basit atama işleci (=) dışında, atamayı aritmetik işlemlerle birleştiren atama işleçleri de vardır. Bunlar +=, -=, *=, /= ve %='dir. İşte bazı örnekler:

$x += y;$	x'e y eklenir, bu da $x = x + (y);$ anlamına gelir.
$x -= y;$	x'ten y çıkarılır, bu da $x = x - (y);$ anlamına gelir.
$x *= y;$	x y ile çarpılır, bu da $x = x * (y);$ anlamına gelir.
$x /= y;$	x y'ye bölünür, bu da $x = x / (y);$ anlamına gelir.
$x \% = y;$	x'e x'in y'e bölümünden artı kalan atanır, bu da $x = x \% (y);$ anlamına gelir.

Tabii ki, yukarıdaki deyimlerin herhangi birinde y değişkeninin yerine genel bir ifade konulabilir. Sol taraftaki x değişkeni yerine de bellekteki belli bir konuma karşılık gelen daha karmaşık bir ifade konulabilir; örneğin $u[a+b]$. Bu durumda bu ifade bir kez hesaplanır. Diğer bazı işlemlerle atamayı birleştiren atama işleçleri daha sonraki bölümlerde tartışılacaktır.

Öncelik Ve Birleşme Kuralları

Bir ifade içinde, işleçlerin işlenenleri ya

1. C dili tarafından belirlenmiş bulunan işleçlerin öncelik ve birleşme özelliklerine, yada
2. işlenen ifadelerini parantez içine alarak belirlenir.

Parantezler, bir işlecin işlenenlerini, o işlecin önceliğine bağlı olmadan belirtmek veya işleçlerin işlenenlerini daha açıklayıcı olacak şekilde yazmak için kullanılırlar. İkinci neden kolayca anlaşılır ifadeler yazmak için önemlidir, çünkü C dilinde öncelik ve birleşme özellikleri kolayca anımsanamayacak kadar çok işleç bulunmaktadır.

Örneğin

$$a = b + c * d$$

ifadesinde işleçler ve işlenenler şöyledir:

işleç

*

+

=

işlenenler

c ve d

b ve (c * d) 'nin değeri

a ve (b + (c * d)) 'nin değeri

Eğer ifade

$$a = b + (c * d)$$

şeklinde yazılırsa, işlenenler aynıdır, ancak parantezler işleçlerin işlenenlerini daha açık hale getirirler.

Fakat, ifade

$$a = (b + c) * d$$

şeklinde yazılırsa, işleçler ve işlenenler şöyle olur:

işleç

+

*

=

işlenenler

b ve c

(b + c) 'nin değeri ve d

a ve ((b + c) * d) 'nin değeri

Yukarıdaki örneklerden de görülebileceği gibi, bir işlecin öncelik düzeyi işleçlere atanacak olan işlenenlerin sırasını belirlemeye yarar. Daha yüksek önceliği olan bir işlecin işlenenleri daha düşük olan bir işleçten önce atanacaktır. Eğer işleçlerin öncelik düzeyleri aynıysa, o zaman birleşme kuralı işlenenlerin soldan sağa mı yoksa sağdan sola mı atanacağını belirtir. Aşağıdaki çizelgede, şimdiye kadar anlatılmış bulunan işleçlerin öncelik düzeyleri ve birleşme özellikleri verilmiştir:

<u>işleç</u>	<u>öncelik</u>	<u>birleşme</u>
[]	indisleme	→
+, -, ++, --	tekli	←
*, /, %	çarpma	→
+, -	toplama	→
=, +=, vs	atama	←

Burada, “←” sağdan sola ve “→” soldan sağa birleşmeyi gösterir. Örneğin

a/b/c ile (a/b) / c aynı anlama gelir

ve

a=b=c ile a=(b=c) aynı anlama gelir

Bir başka örnek olarak

$$c = d + c * e - f / g + h \% j$$

ifadesi

$$c = ((d + (c * e)) - (f / g)) + (h \% j))$$

şeklinde, işleçlerin işlenenlerini ve ifadenin anlamını değiştirmeden, parantezlenebilir.

Dikkat: Bir işlecin işlenenlerinden hangisinin önce hesaplanacağını belirtmemiş olduğuna dikkat edin. Örneğin

$$a = ++b - b$$

ifadesinin değeri, derleyiciye bağlı olarak, ya 0 yada 1 olabilir; bu yüzden bu tür, yan etkisi olan, ifadelerden kaçınılması gerekir. Ayrıca, + ve * gibi, birleşme ve değişme özelliği gösteren işleçlerde, parantezlerin bulunmasına rağmen, birleşme kuralı bazı derleyiciler tarafından dikkate alınmayabilir.

Değişmez İfadeler

Bazı işleçler—örneğin, atama ve ++ ile -- işleçleri dışında, şimdiye kadar öğrenmiş olduğumuz işleçler—değişmezlere (veya değişmez ifadelere) uygulandıklarında *değişmez ifadeler* oluştururlar. Değişmez ifadelerin avantajı derleme sırasında hesaplanabilmeleridir. Derleyici değişmez ifadenin değerini hesaplayıp yerine değişmez değeri koyar. Değişmez ifadeler bir değişimin beklendiği yerlerde kullanılabilirler. Örneğin, dizi boyları değişmez olmalıdır, bu durumda

```
int a [MAXL*5+4];
```

tanımı, eğer MAXL ön işlemci tarafından (#define ile) tanımlanmış bir değişmez ise kabul edilebilecek, fakat eğer MAXL bir değişken ise reddedilecektir. Programın anlaşılabilirliğini geliştirebileceği için değişmez ifadelerin kullanımından kaçınmamak gerekir.

1.6. Tip Dönüşümü Ve Kalıplar

Bir işleç, işlenen(ler)ine uygulandığı zaman, belirli bir tipten bir değer oluşturur. Meydana çıkan değerın tipi, işlecin işlenen(ler)ine, işlecin kendisine ve tip dönüşümü kurallarına bağlıdır. Örneğin, $x+y$ ifadesinin değeri hesaplandığı zaman, x ve y tamsayı ise ortaya çıkan değerin tipi de tamsayı olur. C dilinde işleçlerin çoğu, değişik tiplerde işlenenleri kabul ettikleri için genel olma özelliğini taşır. Örneğin, yukarıdaki ifadede eğer y **double** olsaydı, ortaya çıkacak değer de **double** tipinde olacaktı.

Değişik tipte işlenenlerin bulunduğu ifadelere *karışık-tip ifadeler* denir. Karışık-tip ifadelerin değerleri hesaplandığında, ara ve/veya sonuç değerlerin tipleri, ya dolaylı olarak otomatik tip dönüşümüyle belirlenir yada açık olarak *kalıp* kullanılarak kontrol edilir.

Değişik tipten iki işlenen varsa, C dili kurallarına göre, *otomatik tip dönüşümü* uygulanır. Bu kurallar şöyledir:

1. Önce, **char** ve **short** olan işlenenleri **int**'e veya gerekiyorsa **unsigned**'a dönüştür.
2. Sonra, aşağıdaki listeye uygun bir şekilde, düşük tipten olan işlenenin tipini daha yüksek tipten olanına dönüştür:

<u>tip</u>	<u>düzey</u>	
long double	en yüksek	
double	:	
float	:	
unsigned long	:	
long	:	unsigned 'ın tüm olası değerlerinin long tarafından içerildiği varsayılsa
unsigned	:	
int	en düşük	

Tip dönüşümü üzerine birkaç örnek. *i*, *j*'nin **int**, *x*'in **float**, *d*'nin **double** ve *c*'nin **char** olduğunu kabul edin.

```
i = j + x + d;
c = c + 'A' - 'a';
```

İlk ifadede, *j* **float**'a dönüştürülür, *j*+*x*'in değeri **double**'a dönüştürülür ve *j*+*x*+*d*'nin değeri **double**'dan **int**'e çevrilir. İkinci ifadede, *c*'deki küçük harfi büyüğe çevirmek için tamsayı aritmetiği kullanılır.

Otomatik tip dönüşümü, programlarda önemli bir hata kaynağı olduğu için, tip dönüşümünün, açıkça kalıplar kullanılarak, kontrol edilmesi önerilir. Bir *kalıp*, basitçe, ifadenin önüne konulan parantez içine alınmış bir tiptir; yani,

(*tip*) *ifade*

ifadenin tipinin parantez içine alınmış olan tipe dönüştürüleceğini gösterir. Gerçekte, parantez içine alınmış olan *tip*, ifadenin değerini amaçlanan tipe dönüştüren özel bir *işleç*tir. Bu işlecin önceliği diğer tekli işleçlerin önceliği ile aynıdır. İşte bazı örnekler. *x*, *y*, *z*'nin **float**, *i*, *j*'nin **int** olduğunu varsayın. *y* ve *z*'nin hesaplanmasındaki farka dikkat edin.

```
x = (float) i;
y = (float) (i / j);           /* int bolme islemi */
z = (float) i / (float) j;     /* float bolme islemi */
i = (int) (x * (float) j);
```

1.7. Basit Girdi/Çıktı

Girdi ve çıktı deyimleri gerçekte C dilinin bir parçası değildir. Yani, diğer programlama dillerinin tersine, C dilinin içine konmuş girdi/çıkıtı deyimleri yoktur. Girdi/çıkıtı işlemleri, her zaman, fonksiyonlar çağırılarak yapılır. Tabii ki, girdi/çıkıtı yapmak için kullanılan

fonksiyonların programcı tarafından yazılmasına gerek yoktur. Hemen hemen bütün C ortamlarında girdi/çıkı fonksiyonları içeren standart kütüphaneler bulunmaktadır. Bu kütüphanelerde tanımlanmış bulunan fonksiyonlar (ile alabilecekleri argümanlar) ve ilgili birtakım değişkenlerin bildirisi ise bir başlık dosyasına konur. `stdio.h` böyle bir başlık dosyasıdır ve herhangi bir standart girdi/çıkı fonksiyonu çağrılmadan veya değişkenleri kullanılmadan önce

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

yazılarak kaynak programın içine kopyalanması gerekir.

Kullanıcının girdi/çıkı yapması için, üç girdi/çıkı ara dosyasının tanımı önceden yapılmıştır. Bunlar şunlardır:

<code>stdin</code>	standart girdi
<code>stdout</code>	standart çıkı
<code>stderr</code>	standart hata çıkı

ve normal olarak kullanıcının klavye ve ekranına bağlanmıştır.

Programcının basit girdi/çıkı işlemleri yapması için, sadece gerekli olan `printf`, `scanf`, `getchar`, `putchar`, `_getch` ve `_getche` fonksiyonları bu kısımda anlatılacaktır. Diğer girdi/çıkı fonksiyonları için girdi/çıkı ile ilgili bölüme bakınız.

`printf(kontrol_karakter_dizisi, argüman_listesi_opt)` Fonksiyonu

Bu fonksiyon `stdout`'a yapılacak biçimli çıkı içindir. *Kontrol_karakter_dizisi*, *argüman_listesi*ndeki argümanların değerlerinin çıkısını denetleyen sıfır veya daha fazla dönüşüm tanımlamaları sağlar. *Argüman_listesi* virgüllerle ayrılmış argümanlardan oluşur ve bulunması zorunlu değildir.

En basit dönüşüm tanımlaması şu şekildedir:

```
%z
```

Burada *z*'nin yerine aşağıdaki dönüşüm karakterlerinden biri gelmelidir:

dönüşüm karakteri

c
s
d veya i
u
x, X
o
f
e, E
g, G
p
n

çıkı

işaretsiz bir karakter
karakter dizisi (karakter göstergesi)
işaretli bir ondalık tamsayı
işaretsiz bir ondalık tamsayı
işaretsiz bir onaltılı tamsayı
işaretsiz bir sekizli tamsayı
double
e veya E gösteriminde **double**
e (E) veya f'nin en kısası
bir göstergenin değeri
şimdiye kadar yazılmış olan karakterlerin sayısı

Genelde, bir dönüşüm tanımlaması şu şekildedir:

`%f0w.plz`

Burada *f* isteğe bağlı olarak aşağıdaki bayraklardan biridir:

<u>bayrak</u>	<u>anlamı</u>
-	çıkıtı alanında sola yanaştır
+	öne bir işaret koy
#	kayan noktalı sayılar için mutlaka nokta konmasını sağla; sekizli için öne 0, onaltılı için öne 0x (0X) koy
boşluk karakteri	eğer işaret yoksa bir boşluk koy

İsteğe bağlı olan 0, sayının solunu sıfırla doldurmak içindir. İsteğe bağlı olan *w* sayısı, çıkıtı değerini alacak olan çıkıtı alanının genişliğini belirtir. İsteğe bağlı olan *p* sayısı, kayan noktalı bir sayı için kesir kısmındaki rakamların sayısını, bir karakter dizisi için yazılacak en fazla karakter sayısını veya bir tamsayı için yazılacak en az rakam sayısını gösterir. İsteğe bağlı olan *l* ise argümanın kısa (h), uzun (L) bir tamsayı veya uzun (L) bir **double** olduğunu gösterir.

float için bir dönüşüm tanımlaması olmadığına dikkat edin; herhangi bir **float** ifadeyi bir kalıp kullanarak **double**'a dönüştürün. *w* ve *p*'nin yerine konulacak bir *, genişliğin *argüman_listesindeki* bir **int** argümandan alınacağını gösterir. Normalde, değerler çıkıtı alanlarında sağa yanaştırılırlar.

İşte bazı örnekler. *c*'nin **int**, toplam'ın da **double** olduğunu varsayın. Yeni satır karakterlerine dikkat edin.

```
printf("Merhaba\n");
printf("%s\n", "Merhaba"); /* yukarıdakiyle aynı çıktı */
printf("\nSayı=%d adet", c); /* c'nin değeri gereken */
/* genişlikte bir tamsayı olarak yazılır */
printf("\nSayı=%4d adet", c);
/* yukarıdakiyle aynı, ancak alan genişliği en az 4 */
printf("\nToplam=%5.2f", toplam);
printf("\nToplam=%*.f", 5, 2, toplam);
/* toplam'ın değeri dd.dd biçiminde yazılır */
printf("\nToplam=%5.2f Sayı=%4d", toplam, c);
```

Eğer %'den sonraki karakter geçerli bir karakter değilse, sonucun ne olacağı belirsizdir; %% tek bir yüzde işareti basar. `printf` fonksiyonu hakkında daha fazla bilgi için, ilgili C derleyicisinin girdi/çıkıtı kütüphane fonksiyonları el kitabına veya çevrimiçi yardım kolaylıklarına bakılması önerilir.

`scanf(kontrol_karakter_dizisi, argüman_listesi_opt)` Fonksiyonu

Bu fonksiyon `stdin`'den biçimli girdi yapmak içindir. *Kontrol_karakter_dizisi*, *argüman_listesindeki* argümanlara verilecek değerleri kontrol eden sıfır veya daha fazla dönüşüm tanımlamaları sağlar. *Kontrol_karakter_dizisindeki* dönüşüm karakterleri

dışındaki karakterlerin, girdi akımında karşılık gelen karakterlerle aynı olması beklenir. *Argüman_listesi* girdi değerleri için hedef olan değişkenlerin *adreslerinin*, birbirinden virgülle ayrılarak, yazılmasından oluşur. Yapılmış olan başarılı dönüşümlerin sayısı, fonksiyonun değeri olarak geri döndürülür.

Dönüşüm tanımlamaları şu şekildedir:

`%*wlz`

Burada *** değerin argümana atanmasını engeller, *w* en büyük alan genişliğini belirtir, *l* ise argümanın büyüklüğünü gösterir. Üçü de isteğe bağlıdır. *z* dönüşüm karakteri genelde `printf`'teki gibidir. Girdi akımında, bir alan, *beyaz* (boşluk, tab, yeni satır vs) olmayan bir karakterle başlayıp, ilk gelen beyaz karakterle veya belirtilen alan uzunluğu sonunda biter.

İşte bazı örnekler. *c1*, *c2*'nin **char**, *i*'nin **int**, *y*'nin **float**, *d*'nin **double** ve *s*'nin **char** dizisi olduğunu varsayın. `%lf` bir sayının okunmasını ve **double** olarak depolanmasını sağlar. **long double** için `%Lf` kullanın. & işlecinin, fonksiyona değişkenin *değerinin* değil de *adresinin* iletilmesi için kullanıldığına dikkat edin. Bu işleç Bölüm 3'te anlatılacaktır.

```
scanf("%d", &i);
scanf("%4d", &i);
scanf("%c%c*3s%d%f%lf", &c1, &c2, &i, &y, &d);
scanf("%[^.].", s);
scanf("%[ABC]", s);
```

Son iki örneğin ilki, (beyaz boşluk karakterlerinden biriyle değil de) bir nokta ile sonlandırılmış bir karakter dizisinin okunarak noktanın atlanmasını sağlar. Son örnekte, sadece A, B veya C karakterlerini içeren bir karakter dizisi okunmaktadır. `scanf` fonksiyonunun kullanımı hakkında daha fazla bilgi için, derleyicinizin elkitaplarına veya çevrimiçi yardım kolaylıklarına başvurunuz.

getchar() Ve putchar() Fonksiyonları

Bu fonksiyonlar `stdin`'den veya `stdout`'a bir karakterin girilmesini veya çıktısının yapılmasını sağlarlar. Tipik bir kullanım şöyledir. *c*'nin **char** veya **int** olduğunu farzedin.

```
c = getchar();
putchar(c);
```

Eğer klavyeden girilen bir karakter dizisi arka arkaya `getchar` fonksiyonunun çağırılmasıyla okunursa, `getchar()`'ın vereceği son karakter satır ilerletme karakteri ('\n') olacaktır, çünkü satırbaşı karakteri ('\r') standart girdi/çıkış yordamları tarafından elenmektedir. Aynı şekilde, `putchar()` kullanıldığında, satır sonuna '\n' yazılması yeterlidir; '\r' karakteri otomatik olarak '\n'nin önüne eklenecektir.

_getch() Ve _getche() Fonksiyonları

Standart olmayan bu fonksiyonlar, klavyedeki bir tuşa yapılan tek bir vuruştan ortaya çıkan karakteri verirler. Yani programın bilgi alabilmesi için, `getchar()` gibi, satırın ENTER'la bitirilmesini beklemezler. `_getche()`'de girilen karakter ekrana yansıtılır, `_getch()`'de bu olmaz. *Not:* Bu fonksiyonlar standart olmadıkları için, isimlerinin önünde altçizgi (`_`) karakteri bulunmaktadır; ancak bazı sistemlerde, örneğin Microsoft C derleyicisinin eski uyarlamalarında, altçizgi karakteri olmadan kullanılmaları gerekebilir. Verilen örnek programlarda bunu dikkate alarak, kendi sisteminiz için gerekli düzenlemeleri yapabilirsiniz.

Girdide Dosya Sonu Kontrolü

`stdio.h` başlık dosyasında `#define` ile tanımlanmış bulunan EOF ismi dosya sonunu kontrol etmek için kullanılabilir. Sistemimizde, dosya sonu ASCII ondalık kodu 26 olan `CONTROL+Z` (klavyedeki `CONTROL` tuşu basılı iken `Z`'ye basılarak oluşturulur) karakterinin alınması şeklinde tanımlanmıştır. Ancak C kütüphanesindeki standart bir fonksiyon tarafından döndürülen dosya sonu işareti EOF'tur. **if** deyiminin henüz anlatılmamasına rağmen, aşağıdaki örnek `stdin`'den dosya sonunun nasıl anlaşılabileceğini gösterir. *Dikkat:* `c`'nin tipi en azından **int** olmalıdır.

```
c = getchar();
if (c == EOF) { /* eger dosya sonu ise */
    ...
    dosya sonu işlemleri yapan deyimler
    ...
}
```

1.8. C Deyimleri Ve Program Gövdesi

C programları deyimlerden oluşur. Yazılabilecek en basit deyim bir *ifade deyimidir*. Bir ifade deyimi, arkasına noktalı virgül konmuş herhangi bir ifadedir. Örneğin,

```
a + b * c;
i++;
```

iki ayrı ifade deyimidir; ancak bunlardan ilki pek bir işe yaramaz, oysa ikincisi, yan etkisinden dolayı, bir işe yarar: `i`'nin değerinin bir artırılmasına neden olur.

Bileşik bir deyim dışında her deyim noktalı virgülle sona erer.

Bir *bileşik deyim* (ayrıca, bildirimler içerdiği zaman, “blok” da denir), aşağıdaki gibi çengelli parantezler arasına alınmış bir deyimler sırasındır. Bileşik deyimden sonra noktalı virgül olmadığına dikkat edin.

```
{ bildirimleropt deyim1opt deyim2opt ... deyimn opt }
```


MS-QC, uygun bir şekilde, hard diskinize yerleştirilmişse veya dağıtım disketi disket sürücünüze yerleştirilmişse, işletim sistemi iletisine QC yazın. İsterseniz, C programınızı içeren kaynak dosya adını da belirtebilirsiniz. Örneğin,

C>QC mprog.c

MS-QC, tepesinde birtakım menü maddelerinin bulunduğu ve komut satırında belirtilen kaynak dosyadan satırlarla doldurulan bir görüntüleme bölgesinden oluşan bir ekran sunacaktır. Eğer kaynak dosya belirtilmemişse görüntüleme bölgesi boş olacaktır. Hemen yeni C programınızı yazmaya başlayabilirsiniz veya aşağıdaki düzenleme tuşlarını kullanarak eski bir programda değişiklikler yapabilirsiniz:

Yukarı, aşağı, sola, sağa oklar,	
PAGE UP (sayfa yukarı),	
PAGE DOWN (sayfa aşağı)	imleç hareketi
INSERT	araya ekleme/üste yazma
DELETE	imlecin arkasındaki karakteri silme

Diğer klavye tuşlarının düzenleme işlevlerini ortaya çıkarmak için deneme yanılma yöntemini kullanın. MS-QC editörünün kullanımı temelde MS-DOS'daki EDIT komutuna benzemektedir.

Bir C programı yazıldıktan sonra, bir menü maddesi seçilerek derlenebilir, çalıştırılabilir, hataları düzeltilebilir veya bir dosyada saklanabilir. Eğer fareniz varsa, seçime doğru sürün ve tuşlayın. Yoksa, menü maddesinin ilk harfinin x olduğunu varsayarsak, ALT+ x 'e basın. Aynı şekilde, ALT'a basıp, ok tuşlarını kullanarak menü maddesini seçip ENTER'a da basabilirsiniz. MS-QC menü maddesinde altı çizilmiş harfe basarak ilgili seçenek seçilebilir. Örneğin,

ALT+F: Dosya menüsünü aç
 X: MS-DOS'a çık
 S: Programı sakla
 vs.
 ALT+R: Geçiş menüsünü aç
 C: Derle
 S: Derlenmiş programı yürütmeye başla
 vs.
 vs.
 ESC: Alt menü penceresini kapa

Bir menü maddesinin seçilmesi, diyalog penceresi denilen, ya ek seçimler yada bilgi veren mesajlar sunan başka bir pencerenin görüntülenmesine yol açabilir. Bu tip menü maddelerinin sonunda üç nokta bulunur.

Sözdizimsel yönden hatalı bir program derlendiğinde, MS-QC bütün sözdizim hatalarını bulup ilk hatalı deyimi görüntüleyecektir. İmleç, hatanın üzerinde duracak ve ekranın alt tarafındaki diyalog kutusunda bir hata mesajı görüntülenecektir. Bir önceki

veya sonraki hata bölgesi SHIFT+F3 veya SHIFT+F4'e basılarak görüntülenebilir. Bu şekilde, tekrar derlemeden önce, bütün hataları bulup düzeltebilirsiniz.

Bir C programı ayrıca, basitçe SHIFT+F5'e basılarak veya RUN altmenüsünden START seçilerek derlenip çalıştırılabilir. Sözdizimsel hatalar az önceki gibi bildirilecektir, fakat uyarı dışında başka hata yoksa, programın yürütülmesine geçilecektir. Programın çıktısı otomatik olarak çıktı ekranında görüntülenir. MS-QC görüntüsüne geri dönmek için ENTER'a basın. Çıktı ekranını tekrar görüntülemek için F4'e basın.

Şimdi de hata bulma konusunda bir iki söz. Aşağıdaki yöntemleri kullanmak için programınızın DEBUG modunda derlenmesi gerekmektedir. Bunun için, önce ALT+R'ye sonra C'ye basın. Gelen pencerenin sağ tarafında "Debug" yazısının önünde X işareti olup olmadığını kontrol edin. Eğer yoksa ALT+D'ye basın. Daha sonra ENTER'a basıp programı derleyin.

Programınızı çalıştırmaya başlamadan önce, deyme bir kesilme noktası (*breakpoint*) koyarak, yürütmenin o deyme gelince beklemesini sağlayabilirsiniz. Sadece imleci deyimin üstüne götürüp F9'a basmanız yeterlidir. Daha sonra, kesilme noktasını kaldırmak isterseniz aynı işlemi tekrarlayın.

Programın çalışması devam ettiği esnada bir değişkenin ve/veya ifadenin değerini görüntüleyebilirsiniz. DEBUG altmenüsündeki ADD WATCH menü maddesini seçin ve sürekli görüntülemek istediğiniz değişkeni veya ifadeyi yazın.

Çalıştırılmakta olan deyimleri görüntülemek için bir izleme (*trace*) kolaylığı da mevcuttur. Bir fonksiyon çağırısı esnasında fonksiyon içindeki deyimleri izlemek için F8'e basın, fonksiyon içindeki deyimlerin izlenmesini atlamak için F10'a basın.

QuickC'nin, C programlamasını kolay hale getiren başka birtakım kolaylıkları da vardır. Kullanıcının ilgili elkitaplarına bakması ve/veya menü seçimlerini deneyerek bu kolaylıkları ortaya çıkarması önerilir. *Microsoft Visual C++ Development System for Windows* paketi içindeki *Visual Workbench* programı kullanım bakımından MS-QC'ye uyumlu, ancak profesyonel kullanıcılara yönelik çok daha gelişmiş bir ortam sağlar. Diğer etkileşimli C ortamlarında da benzer olanaklar sunulmaktadır.

1.10. Örnek Programlar

Bu kısımdaki örnek programlar, ne işe yaradıkları hakkında kendi kendilerini açıklayacak şekilde yazılmışlardır, ancak henüz anlatılmamış—**if** ve **while** gibi—bazı deyimler ve işlemler içerirler. Bunların anlamlarını ortaya çıkarmak için, okuyucu, eski programlama deneyimlerine dayanarak bir kestirimde bulunabilir.

Örnek Program 1

Bu programda kullanılan `_getche` fonksiyonu, bir standart girdi fonksiyonu olmadığı için, dosya sonu kontrolü MS-DOS standardına uygun olarak CONTROL+Z karakteri

kullanılarak yapılmaktadır. *Not:* `_getche` fonksiyonunun kullanımından dolayı, bu program ANSI Standardına uygun değildir.

```

1.  /* * * * * * * * * * * * * * * * * * * * * */
2.  *
3.  * Bu program, CONTROL+Z girilinceye
4.  * kadar, klavyeden girilen her karak-
5.  * terin ASCII kodunu verir.
6.  *
7.  /* * * * * * * * * * * * * * * * * * * * */
8.
9.  #include <stdio.h>
10. #include <conio.h>
11. void main (void)
12. {
13.     int c;
14.     char ctrlz = 26; /* ASCII ond. kodu 26 (CONTROL+Z) */
15.
16.     printf("\nBazi kontrol karakterleri disinda,
17.           \"\ngirilen karakterin kodu goruntulenecek.\");
18.     while (1) {
19.         /* Ekrana yansitarak bir klavye tusunu oku.
20.          * CONTROL+Z'ye basilmissa programi bitir.
21.          */
22.         printf("\nBir karakter girin: ");
23.         if ((c = _getche()) == ctrlz)
24.             break;
25.         /* Girilen karakterle ayni satirda ondalik, onaltili
26.          * ve sekizli olarak ASCII kodunu goruntule.
27.          */
28.         printf(", ondalik: %d, onaltili: %x, sekizli: %o", c, c, c);
29.     } /* while */
30. } /* main */

```

Örnek Program 2

```

1.  /* * * * * * * * * * * * * * * * * * * * * */
2.  *
3.  * Bu program, -1 girilinceye kadar, klavyeden
4.  * girilen ondalik sayilari kabul eder, daha sonra,
5.  * girilen sayilarin agirlikli toplamini, ortala-
6.  * masini ve varyansini hesaplayip goruntuler.
7.  *
8.  * Uyarı: Çok buyuk sayilarin ve rakam olmayan
9.  * karakterlerin girilmesi sorun yaratabilir.
10. *
11. /* * * * * * * * * * * * * * * * * * * * */
12.
13. #include <stdio.h>
14. #define AGIRLIK 1.0
15. /* Eger veri degerleri agirlikli olarak hesaplanacaksa,
16.  * agirliga farkli bir deger verin.
17.  */
18.

```

```

19. void main (void)
20. {
21.     int sayi;
22.     int adet = 0;
23.     double kn_sayi, toplam=0.0, karelerin_toplami=0.0;
24.     double ortalama=0.0, varyans=0.0;
25.
26.     printf("\nToplam, ortalama ve varyans hesaplayan program.")
27.     "\n\nLutfen ilk sayiyi (veya cikmak icin -1) girin: ");
28.     scanf("%d", &sayi);
29.     /* Girilen sayi olumsuzsa programi bitir. */
30.     while (sayi >= 0) {
31.         adet++; /* Sayaci artir */
32.         /* Sayiyi double'a cevirip AGIRLIK'la carp. */
33.         kn_sayi = (double)sayi * AGIRLIK;
34.         /* Yeni toplami ve karelerin toplamini hesapla. */
35.         toplam += kn_sayi;
36.         karelerin_toplami += kn_sayi * kn_sayi;
37.         /* Yeni ortalama ve varyansi hesapla. */
38.         ortalama = toplam / (double)adet;
39.         varyans = ortalama*ortalama-karelerin_toplami/(double)adet;
40.         if (varyans < 0.0)
41.             varyans = -varyans;
42.         /* Hesaplanan degerleri goruntule. */
43.         printf("\nAdet = %d", adet);
44.         printf("\nToplam = %.2f", toplam);
45.         printf("\nOrtalama = %.2f", ortalama);
46.         printf("\nVaryans = %.2f", varyans);
47.         /* Bir sonraki sayiyi al. */
48.         printf("\nLutfen bir sonraki sayiyi "
49.             "(veya cikmak icin -1) girin: ");
50.         scanf("%d", &sayi);
51.     } /* while */
52.
53.     /* En son hesaplanan degerleri bir daha ve degisik
54.     * bir bicimde goruntule.
55.     */
56.     printf("\nVerilerin adedi: %12d", adet);
57.     printf("\nToplam : %15.2f", toplam);
58.     printf("\nOrtalama : %15.2f", ortalama);
59.     printf("\nVaryans : %15.2f", varyans);
60. } /* main */

```

Problemler

1. Kernighan ve Ritchie'nin (1978'de yayınlanan) *The C Programming Language* adlı kitaplarının ilk baskısında **entry** adı verilen bir anahtar sözcükten bahsedilmekte idi. Şu anki Standartta ve gördüğümüz herhangi bir derleyicide mevcut değildir; ileride kullanılmak amacıyla ayrıldığı söylenmekte idi. Derleyicinizde **entry**'nin bir anahtar sözcük olup olmadığını belirlemede size yardımcı olacak bir program deneyin.
2. Soyağacınızı basan bir program yazın. (Üç düzey yeterlidir.)
3. Aşağıdaki deyimın sonucunu açıklamaya çalışın:

```
printf("%d", printf("C iyi bir dildir.\n"));
```

4. Aşağıdaki geçerli bir deyim midir?

```
((x) = (5));
```

5. Aşağıdaki geçerli bir C deyimi midir?

```
{{{}}}
```

6. Aşağıdaki programın, girdi olarak verilen bir gerçek sayının tamsayı kısmını görüntüleyeceğine inanılmaktadır:

```
#include <stdio.h>
void main (void)
{
    float x;
    scanf("%f", x);
    printf("%d", x);
}
```

Fakat çalışır gibi gözükmemektedir. Düzeltin. *İpucu:* Derleyicinin bulamayacağı iki ayrı hata bulunmaktadır.

BÖLÜM 2: DEYİMLER VE KONTROL AKIŞI

Düşünüleceği gibi, her deyim, bilgisayara ne yapacağını söyleyen, temel bir program adımıdır. Bu bölüme, tüm C deyimlerinin bir listesini vermekle başlayacağız. Ondan sonra, kontrol akışı deyimleri içindeki ifadelerde sıkça kullanılan işleç grubundan bahsedeceğiz. Bölümün geri kalanında, C program deyimlerinin yürütülme sırasını kontrol eden deyimler detaylı olarak anlatılacaktır.

Bir C programı `main` fonksiyonunun ilk deyiminden çalışmaya başlar. Eğer kontrol akışı başka yöne yönlendirilmezse, deyimler, program içindeki sıralarında ard arda işletilir ve yürütme `main`'in son deyiminden sonra durur. Kontrol akışını yönlendirmenin bir yolu da bir fonksiyon çağırısı yapmaktır. Bu bölümde ayrıntılı olarak anlatılacak deyimler ise kontrol akışını kendilerine özgü yollarla etkilerler.

Başlamadan önce, Pascal-severlere önemli bir uyarımız var: C dilinde, noktalı virgül, deyimleri ayırmak için değil, bazı deyimleri bitirmek için kullanılır. Yani, noktalı virgül deyimün bir parçasıdır. Gerek olmadığını sandığınız bazı yerlere noktalı virgül koymanız gerekebilir. Örnekleri gördükçe, kuralların daha açık olacağını umuyoruz.

2.1. C Dilinin Deyimleri

Çizelge 2.1'de bütün C deyimleri liste halinde verilmiştir. Bu çizelgenin ilk iki maddesi önceki bölümde zaten anlatılmıştı. Geri kalanlar ise temelde kontrol akışı deyimleridir ve, **return** dışında, hepsi bu bölümde anlatılacaktır.

ÇİZELGE 2.1 C dilinin deyimleri

deyim	işlev
ifade	çeşitli (atama, fonksiyon çağırma vs)
bileşik	birden fazla deyimden tek bir deyim oluşturur
if	koşullu yürütme
switch	(herhangi bir sayıda seçeneklerle) koşullu yürütme
while	(devam testinin her yinelemeden önce olduğu) döngü
for	while gibi, fakat bir “ilkleme” bölümü var
do	(devam testinin her yinelemeden sonra olduğu) döngü
break	bulunulan bloktan dışarı atlanması
continue	mevcut yinelemenin geri kalanının atlanması
goto	(bulunulan fonksiyon içinde) herhangi bir bölgeye atlanması
etiketli	goto ’nun hedefi
return	(olası bir değerle) fonksiyondan dönüş
boş	hiç (bazı sözdizimsel kullanımları vardır)

2.2. Bağıntısal Ve Mantıksal İşleçler

C dilinde dokuz işleç, *doğru* veya *yanlış* diye yorumlanabilecek değerler verir. C’nin özel bir Boolean (yani mantıksal) veri tipi yoktur ve bu işleçlerle oluşturulmuş bir ifadenin değeri, eğer ifade doğru ise, 1, yanlışsa 0’dır. Aslında, sıfırdan farklı her sayı C dilinde “doğru” anlamını taşır. Böyle bir sonucun tipi her zaman **int**’tir. Aşağıda, bu işleçler, azalan öncelik sıralarına göre anlatılmaktadır. Bütün doğruluk-değerli işleçler, **struct** ve türevleri dışında, her tipten işlenen kabul eder. Sağdan sola doğru birleşen ! tekli işleci dışında, burada anlatılan tüm işleçler soldan sağa birleşirler. Bağıntısal ve mantıksal işleçler değişmez ifadelerde de kullanılabilirler.

Mantıksal Olumsuzlama İşleci !

! (“değil”) işleci işlenenin mantıksal değerini olumsuzlar; yani eğer işlenen 0’sa 1 verir, eğer işlenen sıfırdan farklı ise 0 verir. Örnekler:

```
/* Bu ve bir sonraki kısımda; a=150, b=45.33, c=-132, d=0 */
!a           değeri 0’dır
!c           değeri 0’dır
!d           değeri 1’dır
!(b+c)       değeri 0’dır
!(!c)        değeri 1’dır
```

Bağıntısal İşleçler <, >, <=, Ve >=

Bu işleçlerin isimleri şöyledir:

islec

<

>

<=

>=

isim

küçüktür

büyüktür

küçük veya eşittir

büyük veya eşittir

Bağıntısal işleçler işlenenlerini karşılaştırır ve duruma göre doğru veya yanlış bir değer verirler. Örnekler:

a<b	değeri 0'dır
b>=d	değeri 1'dir
a>b>c	(a>b) >c'ye eşdeğerdir ve değeri 1'dir
a<c<b	(a<c) <b'ye eşdeğerdir ve değeri 1'dir

Eşitlik İşleçleri == Ve !=

== işleci, eğer işlenenleri eşitse 1, yoksa 0 verir. != işleci ise tam tersini yapar. Sık yapılan bir yanlış, == işleci yerine atama işlecini (=) kullanmaktır. Buna dikkat edin. Örnekler:

a==b	değeri 0'dır
a=b	değeri b'dir ve bir atama ifadesidir

Bizim sistemde,

```
x = 5.0/3.0; /* x'in tipi float'tir */
```

deyiminden sonra,

```
5.0==3.0*x
```

ifadesi 0 (yani yanlış!) verir. Bunun nedeni hiçbir bilgisayarın, 1.666666... sayısındaki sonsuz sayıda 6'yı saklayamamasıdır. Aynı sorun 3.0 yerine 25.0 de kullansanız ortaya çıkabilir. Genelde, kayan noktalı sayıların kesin olmasını bekleyemezsiniz, bundan dolayı (derleyici tarafından izin verilmesine rağmen) <=, >= ve eşitlik işleçlerinin işlenenleri olarak **float** tipinde ifadeler kullanmak tehlikelidir.

Mantıksal VE İşleci &&

&& işleci işlenenlerinin mantıksal VE'sini verir, yani eğer her iki işlenen doğru ise sonuç 1 olur, aksi takdirde 0'dır. Bu tanımdan, eğer işlenenlerden biri 0'sa sonucun kesinlikle sıfır olacağı ve diğer işlenenin değerinin hesaplanmasına gerek olmadığı açıktır. C, bu gerçeğe dayanarak, diğer bazı dillerin tersine, ilk işlenenin değeri 0'sa ikincisini hesaplamaz. Örnekler:

```
/* fn() 'nin 657 veren bir fonksiyon oldugunu varsayın */
a&&b           değeri 1'dir
d&&fn()        değeri 0'dır ve fn() çağrılmaz
fn()&d         değeri 0'dır ve fn() çağrılır
```

Mantıksal VEYA İşleci ||

|| işleci işlenenlerinin mantıksal VEYA'sını verir, yani eğer her iki işlenen yanlış ise sonuç 0 olur, aksi takdirde 1'dir. Gördüğünüz gibi, eğer işlenenlerden biri sıfırdan farklı ise, öteki işlenenin değeri ne olursa olsun sonuç 1'dir. Yukarıda anlatılan yararlı özellik burada da geçerli olur; eğer ilk işlenen sıfırdan farklı ise, ikinci işlenen hesaplanmaz. Örnekler:

a d	! (!a && !d) 'e eşdeğerdir ve değeri 1'dir
a b	değeri 1'dir
c d	değeri 1'dir
d (!c)	değeri 0'dır

2.3. Doğruluk-Değerli İfadeler

Birçok yeni işleç öğrendiğimize göre, Bölüm 1'de verilmiş olan işleç önceliği ve birleşme çizelgesinin genişletilmiş bir uyarlamasını verelim. Bu Çizelge 2.2'de gösterilmektedir.

ÇİZELGE 2.2 C işleç önceliği ve birleşme

[]						→
!	++	--	+	-	(tip)	←
* / %						→
+ -						→
< <= > >=						→
== !=						→
&&						→
						→
=	*=	/=	%=	+=	-=	←

Bir önceki kısımdaki işleçler kullanılarak yazılmış bazı doğruluk-değerli ifade örnekleri şöyledir:

! (a<=c)	a>c'ye eşdeğerdir ve değeri 1'dir
!a>c	0>c'ye eşdeğerdir ve değeri 1'dir
! (a==b)	a!=b'ye eşdeğerdir ve değeri 1'dir
a<b==c>d	0==0'a eşdeğerdir ve değeri 1'dir
!a<=b!=c&&d	değeri 0'dır ve bütün ifadeler hesaplanır
!a d	değeri 0'dır
a!=b fn()	değeri 1'dir ve fn() çağrılmaz

2.4. if Deyimi Ve Koşullu İşleç

if deyimi, **if** anahtar sözcüğünün arkasına parantez içine yazılmış bir ifade ve peşinden gelen bir deyimden oluşur. Tüm bunların arkasında, isteğe bağlı olarak, bir **else** anahtar sözcüğü ve bir deyim bulunabilir. **if**'in bu iki şekli aşağıdaki gösterimde özetlenmektedir:

```
if (ifade)
    deyim
```

ve

```
if (ifade)
    deyim
else
    deyim
```

İşte **else**'i olmayan bir **if** deyimi örneği:

```
if (yas > 65) {
    yasli_kisiler++;
    printf("Yasli bir kisisiniz.\n");
}
```

Eğer *yas* 65'ten büyükse, blok içindeki deyimler çalıştırılır, aksi takdirde, kontrol doğrudan bir sonraki deyme geçer. Bu deyimi, her yaştan kişiye hitap edecek şekilde değiştirelim:

```
if (yas > 65) {
    yasli_kisiler++;
    printf("Yasli bir kisisiniz.\n");
} else
    printf("Henuz yasli bir kisi degilsiniz.\n");
```

Şimdi, deyiminiz, *eğer* *yas* 65'in üstünde ise, az önceki deyimle aynı şekilde davranacaktır, aksi takdirde yeni mesaj görüntülenecektir. Genelde, parantez içine alınmış ifadenin değeri doğru (sıfırdan farklı) ise, ifadeden sonra gelen deyim işletilir; ifadenin değeri yanlış (sıfır) ise, **else** anahtar sözcüğünden sonra gelen deyim (varsa) işletilir. Örneklerimizde de görüldüğü gibi, bu deyimler bileşik deyimler olabilir, bu da bileşik deyimlerin daha önce belirttiğimiz iyi özelliğidir. Birçok deyimden tek bir deyim oluştururlar. Ayrıca, bileşik deyim başlatan "{" karakterinin ifade ile aynı satırda, onun karşılığı olan "}" karakterinin de anahtar sözcükle aynı sütuna yazıldığına dikkat ediniz. Bu karakterlerin tam yeri—bloğu doğru bir şekilde ayırdıkları sürece—derleyiciyi ilgilendirmez, ancak bu şekilde yazmanın programları anlamayı kolaylaştıracığı inancındayız ve bunu kitap içinde uygulamaya devam edeceğiz.

Şüphesiz, içiçe **if** deyimleri yazabilirsiniz. Tipik bir örnek şöyle olabilir:

```

if (yas > 65) {
    yasli_kisiler++;
    printf("Yasli bir kisisiniz.\n");
} else if (yas > 40) {
    orta_yastaki_kisiler++;
    printf("Orta yastasiniz.\n");
} else if (yas > 17) {
    genc_kisiler++;
    printf("Gencsiniz.\n");
} else
    printf("Cocuksunuz.\n");

```

Bunun tek bir deyim olduğuna dikkat ediniz. Şimdi şunun üzerinde düşünün:

```

if (yas < 65)
    if (yas > 40) printf("Orta yastasiniz.\n");
else
    printf("Cok yaslisiniz.\n");

```

Eğer yas 65'ten büyük veya eşitse ne olacaktır? Diğer bir deyişle, **else** hangi **if**'e aittir? Mesajlar ve içerlek yazma şekli, bu deymi yazan kişinin ilk **if**'e ait olduğunu düşündüğü izlenimini uyandırıyor, fakat bu doğru değil! Kural şöyledir: *Her **else** en yakın **else**'i olmayan **if**'e aittir.* Bundan dolayı, yukarıda yapılmak istenen işi doğru şekilde yapan **if** şudur:

```

if (yas >= 65)
    printf("Cok yaslisiniz.\n");
else if (yas > 40)
    printf("Orta yastasiniz.\n");

```

veya,

```

if (yas < 65)
    if (yas > 40)
        printf("Orta yastasiniz.\n");
    else
        ;
else
    printf("Cok yaslisiniz.\n");

```

else'ten sonraki noktalı virgül bir *boş deyim* oluşturur. Gördüğünüz gibi, sözdizimin bir deyim gerektirdiği, fakat yapılacak bir şey olmadığı yerde boş deyim kullanılabilir. Başka bir yol olarak, ikinci **if**'i bir blok içine alırsak, boş deyimli **else** kullanmak zorunda kalmayız.

```

if (ifade_1)
    degişken = ifade_2;
else
    degişken = ifade_3;

```

şeklindeki bir **if** deymi, *koşullu işleç* kullanılarak

değişken = *ifade_1* ? *ifade_2* : *ifade_3*;

şeklinde basitçe yazılabilir. Genelde, C dilindeki tek üç-işlenenli (üçlü) işlec olan “?:” işleci, önce *ifade_1*’i hesaplar ve eğer *ifade_1* doğru ise *ifade_2*’nin değerini, aksi takdirde *ifade_3*’ün değerini verir. Son iki ifadeden sadece biri hesaplanır. Sonucun tipi, önceki bölümde anlatılan dönüşüm kurallarına göre *ifade_2* ve *ifade_3* birlikte dikkate alınarak belirlenir. “?:” işlecinin sağdan sola doğru birleşme özelliği vardır ve önceliği Kısım 2.2’de görülen tüm işleçlerden düşük, atama işlecinden ise yüksektir.

“?:” işlecinin işleri nasıl kolaylaştırdığına bir örnek:

```
printf(yas>65 ? "Yasli birisiniz.\n"
       : "Henuz yasli degilsiniz.\n");
```

Gördüğünüz gibi, hem *ifade_2* hem de *ifade_3* karakter dizileri ise, sonuç bir karakter dizisidir. “?:” işleci değişmez ifadelerde de kullanılabilir.

2.5. while Deyimi

Diğer programlama dillerinde olduğu gibi, **while** deyimi döngüler oluşturmak için kullanılır. Sözdizimi şöyledir:

while (*ifade*)
deyim

Önce *ifade* hesaplanır. Eğer doğru ise, *deyim* yürütülür, sonra da *ifade* tekrar hesaplanır. *İfadenin* hesaplanmasında yanlış (0) sonucu elde edilinceye kadar bu devam eder; bu durumda kontrol bir sonraki deyim'e geçer. Eğer başlangıçta *ifade* yanlışsa, tekrarlanan *deyim* hiç çalıştırılmaz. Şekil 2.1’deki akış çizeneği **while** deyiminin yürütülme şeklini gösterir.

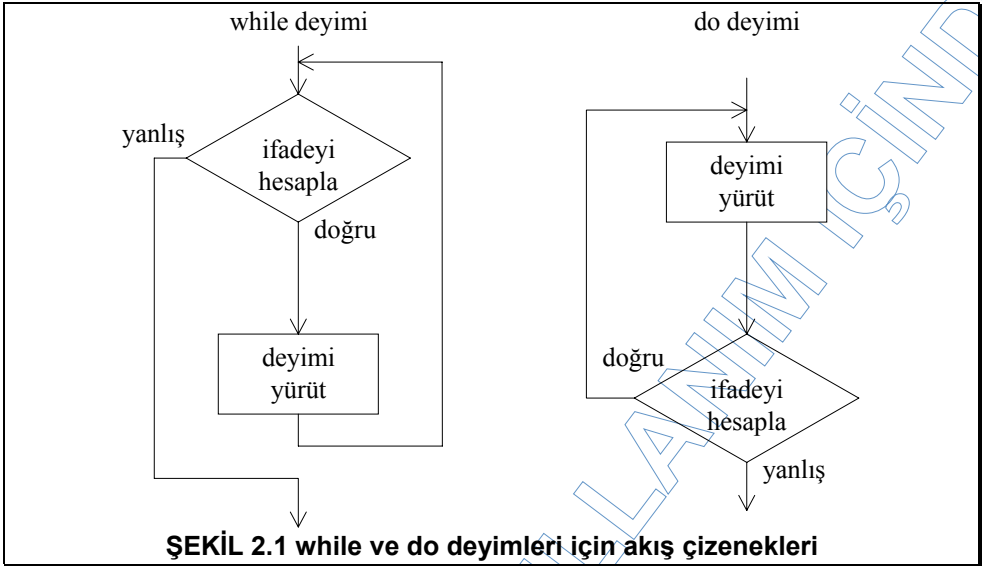
İşte bir **while** deyimi:

```
while (yas<35) {
    genc kisiler++;
    scanf("%d",&yas);
}
```

Eğer *yas*, bu deyimden önce, 35 veya daha büyük bir sayı ise blok hiç yürütülmeyecektir. Aksi takdirde, 35’ten büyük veya eşit bir sayı girilinceye kadar defalarca yürütülecektir. **while** kullanarak sonsuz döngüler yazmanın olası olduğuna dikkat edin: *İfadenin* hiçbir zaman yanlış olmayacağını temin etmeniz yeterlidir. Örneğin,

```
while (1)
    printf("C'yi cok seviyorum!\n");
```

deyimi ile C hakkındaki duyularınızı sonsuza dek yazabilirsiniz.



2.6. do Deyimi

do deyimi **while** deyiminin yakın bir akrabasıdır; o kadar ki sözdiziminde **while** anahtar sözcüğünü içerir:

```
do
    deyim
while (ifade);
```

do'nun **while**'dan tek farkı *ifadeyi*, tekrarlanan *deyim*in yürütölmesinden *sonra* hesaplamasıdır. Bunun anlamı, ne olursa olsun, **do** deyiminde en az bir defa döngüye girildiğidir. Şekil 2.1'de verilen akış çizimciklerindeki farklılıklara dikkat edin.

Birisinin yaşını sorup, olumlu bir sayı girilinceye kadar sormaya devam eden, örnek bir **do** deyimi şöyledir:

```
do {
    printf("Lutfen yasinizi giriniz.\n");
    scanf("%d", &yas);
} while (yas <= 0);
```

Normal olarak, her **do** deyiminde, burada da olduğı gibi, bir blok şeklinde yazılan içteki deyim en az bir defa yürütölür.

2.7. for Deyimi Ve Virgül İşleci

for deyimi, **for** anahtar sözcüğü, arkasında parantez içinde iki noktalı virgülle ayrılmış üç ifade ve ondan sonra gelen bir deyimden oluşur. İfadelerden herhangi biri var olmayabilir, ancak noktalı virgüller bulunmalıdır. **for** deyiminin nasıl işlediğini anlatmanın en iyi yolu, eşdeğer bir **while** deyimi ile onu ifade etmektir. Şimdi, eğer

```
for (ifade_1_opt ; ifade_2_opt ; ifade_3_opt)
    deyim
```

şeklinde bir deyimimiz varsa,

```
ifade_1;
while (ifade_2) {
    deyim
    ifade_3;
}
```

deyimi, **for** deyimi ile aynı işi yapacaktır. Eğer **for** deyimindeki *ifade_2* boşsa, sürekli olarak doğru olduğu şeklinde yorumlanacak ve sonsuz bir döngünüz olacaktır. *ifade_1*'in döngüyü kontrol eden değişkenleri ilkleme için, *ifade_2*'nin döngüyü durdurmak için gereken koşulları belirlemede; *ifade_3*'ün de döngü içindeki bazı değişkenlerin değerlerini değiştirmede kullanılacağına dikkat edin. Örnek:

```
for (i=0; i<n; i++) {
    dizi[i] = 0;
    printf("Dizinin %d nolu elemanına 0 atanmistir.\n", i);
}
```

Bu, basitçe, dizi dizisinin ilk *n* elemanına sıfır atar ve *n* sayıda satır görüntüler. **for** deyimine bir başka örnek ise şöyledir:

```
for (bosluklar=0; getchar()==' '; bosluklar++)
    ;
```

Bu deyim girdi akışındaki boşlukları sayar. Bu durumda, parantez içine alınmış olan ifadelerin gereken her şeyi yaptığına ve tekrarlanan deyim *boş deyim* olduğuna dikkat ediniz. **for**'un deyiminin boş olduğunu vurgulamak için, noktalı virgül ikinci satıra yazılmıştır. Sık yapılan hatalardan bir tanesi de **for** deyiminin sağ parantezinin hemen arkasına noktalı virgül koymak, böylece *deyimi* tek bir seferde yürütülecek normal bir deyim haline getirmektir.

Bazen, **for** deyiminde *ifade_1* ve *ifade_3* adını verdiğimiz yerlerde birden fazla ifadenin bulunması daha uygun olabilir. Örneğin, iki tane sayacımızın bulunduğunu ve döngüden herhangi birisinin sıfır olduğu zaman çıkmamızın gerektiği bir durumu düşünün. Bunu yapmanın başka yöntemleri de vardır, ama aşağıdaki deyim en iyilerindendir:

```
for ( ; sayac_1 && sayac_2; sayac_1--, sayac_2--) {
    ...
}
```

Bu, *virgül işlecini* en çok kullanıldığı yerlerden biridir. Gördüğünüz gibi, virgül işlecisi iki ifadeden tek bir ifade oluşturmaya yarar. Her zaman ikinci işlenenden önce ilk işleneni hesaplar ve soldan sağa birleşir. Sonucu ise sadece ikinci ifadenin değeridir. Sonucun tipi de ikinci ifadenin tipiyle aynıdır. Virgül işlecisi, C'nin işlemleri arasında en düşük önceliğe sahiptir.

Virgül karakterinin, C dilinde başka amaçlar için de kullanıldığına dikkat ediniz. Bildirimlerde değişken isimlerini ayırmak buna bir örnektir. Bu durumlarda, söz konusu olanın bir ayırıcı değil de, bir işleç olduğunu belirtmek için virgül ifadesi parantezler içine alınmalıdır.

2.8. continue Deyimi

continue deyimi sadece bir döngünün gövdesi içinde, yani **while**, **do** veya **for**'un tekrarlanan deyiminde, geçerlidir. Bu deyim yürütüldüğünde, döngü gövdesinin içindeki geri kalan deyimler atlanır ve hemen döngünün devamlılık testine geçilir. Örneğin,

```
for (a=10; a>0; a--) {
    b = -a;
    if (b==5)
        continue;
    printf("%d sayısının olumsuzluğu %d sayısidir.\n", a, b);
}
```

deyiminde (5 hariç) 10'dan 1'e kadar bütün tamsayılar, olumsuzlukları ile birlikte, yazılırlar. Diğer bir örnek olarak,

```
a = 0;
while (a<100) {
    a++;
    if (a%4)
        continue;
    printf("%d\n", a);
}
```

deyimi 100'e kadar 4'ün katlarını görüntüler.

continue, kontrolü her zaman içinde kaldığı en küçük döngünün sonuna aktarır. Yukarıdaki gibi birçok durumda, **if**'teki test koşulunu ters çevirerek ve **continue** yerine döngüde geri kalan deyimleri bir blok içine koyarak **continue**'dan kurtulabiliriz.

Önceki kısımda, **for** deyiminin **while** deyimi şeklindeki yazımını anımsayın. Bu kural, **for**'un tekrarlanan deyiminde bir **continue** olduğu zaman geçerli olmaz, çünkü

ifade_3 her yinelemenin sonunda baştan hesaplanır. Böylece, bu kısımda verilen **for** deyiminin **while**'la yazılmış eşdeğeri şöyledir:

```
a=10;
while (a>0) {
    b=-a;
    if (b==5) {
        a--;
        continue;
    }
    printf("%d sayisinin olumsuzu %d sayisidir.\n", a, b);
    a--;
}
```

2.9. break Deyimi

break deyimi **continue** ile yakından ilintilidir. **continue** deyimi yeni bir yinelemenin başlaması amacıyla döngünün sonuna atlarken, **break**, içinde kaldığı en küçük döngünün dışına atlar ve döngü deyiminin tamamen sona ermesine yol açar. Başka durumlar dışında, **break** “sonsuz” döngülerden çıkış için kullanılır. Örneğin:

```
while (1) {
    scanf("%d", &yas);
    if (yas<=0)
        break;
    printf("Bir sonraki kişi %d yasındadır.\n", yas);
}
```

Burada, olumlu olmayan bir sayı girildiğinde, **break** deyimi kontrolü **while**'dan sonraki deyime aktarır. İçinde **break** bulunan bir program parçası **break** olmadan da yazılabileceğine göre, **break** kullanmak bir tercih sorunudur.

break sadece **switch** (Kısım 2.11), **do**, **while** ve **for** deyimlerinden çıkış için kullanılabilir. Başka yerlerde kullanılamaz.

2.10. goto Deyimi Ve Etiketler

Her C deyimine, deyimden önce bir tanıttıcı sözcük ve iki nokta üst üste koyarak bir *etiket* iliştilerilebilir. Bir deyimi etiketlemek suretiyle, gerektiğinde o deyime ulaşmak için kullanılabilecek bir “adres” verilir. Bir fonksiyon içinde birden fazla deyime aynı isim, etiket olarak, verilemez. Bir etiket ve onun arkasına bir deyim yeni bir deyim oluşturur. Bu tür bir deyime *etiketli deyim* denir:

tanıttıcı_sözcük : deyim

goto deyimi, kontrolü doğrudan etiketli deyime aktarır. Sözdizimi şöyledir:

goto *tanıtıcı_sözcük*;

Tanıtıcı_sözcük aynı fonksiyon içinde var olan bir etiket olmalıdır. **goto** kullanarak, fonksiyon içinde (dışında değil) herhangi bir yere atlanabilir; ileri veya geriye doğru, istenildiği kadar içiçe geçmiş döngü veya blokların içine veya dışına, istediğiniz bir yere gidebilirsiniz. **goto**'nun sorumsuzca kullanımı, anlaşılması ve bakımı yapılması olanaksız programlar yaratabilir. Kuramsal olarak, anlambilimsel hiçbir yitime uğratılmadan, **goto** bir programdan çıkarılıp yerine döngü ve **if** deyimleri konulabilir. Buna rağmen, hata işleme gibi, bazı durumlarda işleri kolaylaştırabilir. Örneğin:

```
while (bir_kosul) {
    ...
    do {
        ...
        for (...; ...; ...) {
            ...
            if (guuum)
                goto felaket;
            ...
        }
    } while (baska_kosul);
    ...
}
...
felaket: hata işlemleri
```

Bu durumda bile, **goto** deyimi kaldırılabilir. *Hata işlemlerinin*, bazı temizlik işleri yapıp, olası bir hata kodu ile, çağırın fonksiyona dönüş yaptığını varsayın. O zaman, niye bu işleri **goto** deyiminin bulunduğu yerde yapıp bir **return** kullanmayalım? (**return** deyimi bir fonksiyondan dönmek için kullanılır ve Bölüm 4'te işlenmektedir.) **break**, **continue** ve **return** gibi daha uygun yapıların bulunduğu, C gibi bir dilde, **goto**'lardan kaçınılabilir ve kaçınılmalıdır. İyi bir programcı **goto**'yu ve hatta **continue**'yu hemen hiç bir zaman kullanmaz. **switch** dışında da, **break** deyimini çok seyrek kullanır. Bu arada, **goto** kullanmadan bile, kontrol akışı anlaşılmayan programlar yazmak olasıdır. Dikkatli tasarım, ayrıntılı belgeleme ve makul içerlek yazma alışkanlığı, bundan kaçınmanın bazı yollarıdır.

2.11. switch Deyimi

switch (*ifade*)
deyim

switch deyimi birçok şıktan bir tanesini seçmede kullanılır. İşte kullanıcıya “menüler” sunan bir programda bulunabilecek bir **switch**:


```
switch (_getche()) {
    case 'r':
    case 'R':
        rezervasyon_yapma();
        break;
    case 'l':
    case 'L':
        yolcu_listeleme();
        break;
    case 'i':
    case 'I':
        rezervasyon iptali();
        break;
    case 'c':
    case 'C':
        cikis();
        break;
    default:
        printf("Yanlis secenek... Tekrar deneyin.\n");
}
```

Bu örnek **switch**'in tüm önemli özelliklerini göstermektedir. Parantez içine yazılmış tamsayı tipindeki—bu örnekte olduğu gibi, bir karakter de olabilir—sayı ifadesi hesaplanır, önüne **case** anahtar sözcüğü getirilmiş olan değişmez ifadelerden birinin değeri buna uyuyorsa kontrol bu **case** etiketli deyim aktarılır ve buradan sıralı olarak devam eder. Eğer hiçbir **case** etiketindeki değer ifadenin değerine uymuyorsa, iki olasılık vardır: *Deyim* içinde bir **default** etiketi varsa, kontrol buraya aktarılır, aksi takdirde **switch** deyiminden çıkarılır.

Yukarıdaki deyimde hiçbir **break** konulmadığı takdirde, *r* veya *R* girildiğinde, bütün fonksiyonlar (*rezervasyon_yapma*, *yolcu_listeleme* vs) çağrılacak ve hata mesajı da basılacaktır. Bundan dolayı hemen hemen bütün **switch** deyimlerinde **break**'ler bulunur. Örnekte olduğu gibi, **switch**'teki parantez içindeki ifadeden sonra gelen deyim genelde bir bloktur.

case ve **default** etiketleri **switch** deyimleri dışında kullanılamazlar. Ayrıca, bir **switch** deyiminde birden fazla **default** etiketi ve aynı **switch** içinde aynı değere sahip birden fazla **case** etiketi olamaz.

switch'in öyküsünü başka bir ilginç örnekle bitiriyoruz:

```
switch (yas)
    case 20:
    case 40:
    case 60:
        printf("20, 40 yada 60 yasindasınız.\n");
```

2.12. Bir Örnek—Sayı Sıralama

```

1.  #include <stdio.h>
2.  #define GOZCU 0
3.  #define LIMIT 25
4.
5.  void main (void)
6.  {
7.      int i, j, k;
8.      int sayi, gecici;
9.      int sayilar[LIMIT];
10.
11.     /* girdi */
12.     i=0;
13.     while (i<LIMIT) {
14.         scanf("%d", &sayi);
15.         if (sayi==GOZCU)
16.             break; /* girdi sonu */
17.         sayilar[i++]=sayi;
18.     }
19.
20.     /* siralama */
21.     i--;
22.     for (j=0; j<i; j++)
23.         for (k=j+1; k<=i; k++)
24.             if (sayilar[j]>sayilar[k]) {
25.                 gecici=sayilar[k];
26.                 sayilar[k]=sayilar[j];
27.                 sayilar[j]=gecici;
28.             }
29.
30.     /* cikti */
31.     k=0;
32.     do
33.         printf("%d ", sayilar[k++]);
34.     while (k<=i);
35. } /* main */

```

Girilen tamsayıları artan sıraya göre sıralayan, yukarıdaki program, **if**, **while**, **do**, **for** ve **break** deyimlerini örneklemek için verilmiştir. En başta tanımlanmış olan iki değişmez, sırasıyla, girdi için “gözcü” bir değer ve içinde sayıların saklanacağı dizinin boyudur. *Gözcü*, girdi akışında karşımıza çıktığında, daha fazla girdi olmayacağını belirten bir değerdir. Örnekteki durumda, sıfır rastlandığında program girdi okumayı bitirecektir.

Program, sırasıyla, girdi, sıralama ve çıktı yapan üç bölümden oluşmuştur. İlk bölüm (Satır 12-18) klavyeden tamsayılar okur ve bunları sıfırcı elemandan başlayarak sayılar adlı diziye yerleştirir. **while** deyimindeki (i<LIMIT) ifadesi en fazla LIMIT sayıda tamsayının okunup sayilar dizisine konulmasını temin eder. Sıralamak istediğimiz sıfırdan farklı tamsayıların sayısı LIMIT’ten azsa, girdi sonunda GOZCU değerini, yani 0, veririz. **while** bloğunun içindeki **if** deyimi, sayilar dizisine koymadan önce girilen değeri kontrol eder ve eğer sıfırla karşılaşırsa, **break**’le **while** döngüsünün dışına atlar. İlk bölümü, **if** ve **break** kullanmadan, bir **while** ile yazmaya çalışın.

İlk bölümün sonunda, `i`'de, `sayilar` dizisinin son dolu elemanının indisinin bir fazlası bulunur. Bundan dolayı, ikinci bölüm (Satır 21-28) `i`'yi bir azaltarak başlar. Arkadan gelen `for` deyimi, sıralamayı sağlamak için, `i` ile beraber `j` ve `k` adında iki sayaç kullanır. Dıştaki (yukarıdaki) `for`, `sayilar` dizisinin ilk `i-1` elemanına bakar. Dıştaki `for` tarafından `sayilar` dizisinin bakılan her `j`'inci eleman için, içteki `for`, `j+1`'den `i`'ye kadar indislenmiş elemanlara bakar. Yani, `if` deyiminin yürütüldüğü her sefer `sayilar` dizisinin iki farklı elemanı, `sayilar[j]` ve `sayilar[k]`, karşılaştırılır ve `j < k`'dir. İşimiz bittiğinde `sayilar` dizisinde girilmiş sayıların artan sıraya göre sıralanmasını istediğimize göre, eğer `sayilar[j] > sayilar[k]` ise iki elemanın değerlerini değiş tokuş ederiz. Bu işlemin yapıldığı yer Satır 25-27'dir.

Üçüncü bölüm (Satır 31-34), bir `do` kullanarak, (artık sıralanmış sayıları içeren) dizideki ilk `i+1` elemanı basar. Eğer, en başta, ilk (ve son) değer olarak sıfır girersek, bu bölüm gereksiz ve ilgisiz bir sayı basacaktır. (Bunun nedenini söyleyebilmemiz gerekir.) Bu, en kısa ve basit gibi görünen programların bile, en beklenmeyen zamanda kendini gösteren, hatalar içerebileceğini gösterir.

Problemler

1. `do` deyimini başka kontrol akışı deyimleri kullanarak “formüle” edin.

2. Aşağıdaki `if`'in görevini yapan bir `switch` yazın.

```
if (yas == 16)
    printf("Onumuzdeki yıl "
           "surucu belgesi alabilirsiniz!\n");
```

3. `while` deyimini başka kontrol akışı deyimleri kullanarak yazın.

4. Bir tamsayı okuduktan sonra, 1'den girilen `sayıya` kadar olan bütün tamsayıların kareleri ile karşıt ($1/sayı$) değerlerini basan bir program yazın.

5. Sıfırla sonlandırılmış bir tamsayı serisini okuyup sonunda sayıların en küçük, en büyük ve ortalamasını veren bir program yazın.

6. Bir karakter sırasını okuyup arka arkaya tam 3 defa rastladığı karakterleri basan bir program yazın. Örneğin

```
bcdāaaefghijjjkkkop
```

girdisi için, çıktı

```
a k
```

olmalıdır.

7. Tedavülde (şimdilik) 5, 10, 20, 50, 100, 250, 500 bin ve bir milyonluk banknotlar bulunmaktadır. Fiyatı ve bu cinsten gösterilmiş para miktarını kabul edip geriye verilecek bozuk parayı (yani fiyat ile verilmiş para arasındaki farkı) liste şeklinde

veren bir program yazın. Bu durumda, programın kabul edebileceği para miktarı 5 binin katı olmalıdır.

8. Kısım 2.12'deki programı, her çeşit bilgi için doğru çalışacak şekilde düzeltin.
9. Bir x sayısının b tabanına göre logaritması şu şekilde hesaplanabilir:

Sayının $1 \leq x < b$ şeklinde olduğunu varsayın. (Eğer öyle değilse, kolayca o hale getirebilirsiniz.) Bu durumda $\log_b x = 0.d_1 d_2 \dots d_n$. Burada n sistemdeki kayan noktalı sayılar için anlamlı rakam sayısıdır; yani **double** sayılar için DBL_DIG. d_i rakamları şu döngü kullanılarak hesaplanabilir:

$i \leftarrow 1$ 'den n 'ye kadar aşağıdaki döngüyü yap

$$x \leftarrow x^{10}$$

$1 \leq \frac{x}{b^{d_i}} < b$ olacak şekilde d_i 'yi hesapla (d_i 0'dan 9'a kadar değişen bir tamsayı)

$$x \leftarrow \frac{x}{b^{d_i}}$$

Önce, bazı sayıların logaritmalarını elle hesaplayarak algoritmayı anlamaya çalışın. Örneğin, $\log_{10} 2$, $\log_{16} 2$ vs'yi hesaplayın. Daha sonra, 0.01 artımlarla 1.00 ile 10.00 arasındaki x sayılarının $b=10$ tabanına göre logaritmalarının çizelgesini $n=5$ ondalık basamağa kadar basacak bir C programı yazın.

10. Aşağıdaki oyunu oynayacak tam bir program yazın: Kullanıcı 1 ile 1000 arasında bir sayı seçer, bilgisayar da sayıyı tahmin etmeye çalışır. Her tahminden sonra, eğer tahmin daha büyükse kullanıcı B girer, daha küçükse K, doğru sayı ise D girer. Programın örnek bir çıktı ve girdisi şöyle olabilir: (Bu durumda, seçilmiş olan sayı 375'tir.)

```
500? B
250? K
375? D
```

Program akılcı kestirimlerde bulunmalıdır. Program en fazla kaç denemede sayıyı bulmalıdır?

BÖLÜM 3: GÖSTERGELER VE BİT İŞLEME

Bu bölümde iki konu anlatılacaktır: göstergeler ve bit işlemleri. Bunların ortak bir yönü vardır. C'nin düşük düzeyli destek sağlayan yüksek düzeyli bir dil olmasından dolayı, popüler bir sistem programlama dili olduğu gerçeğini gösterirler. Göstergeler, kullanıcının bellek adreslerine ulaşmasına ve bunlarla işlem yapmasına izin verirler; bit işlemleri ise tipik olarak birleştirici dillerde rastlanan düşük düzeyli işlemlerdir.

3.1. Gösterge Değişkenleri Ve İşlemleri

Göstergeler ve gösterge işlemleri C'nin çok önemli bir yönünü oluştururlar. Göstergeler, başka değişkenlerin adreslerini veya daha genel konuşursak, bellek konumlarını depolamaya yarayan değişkenlerdir. Göstergeler kullanmak suretiyle daha kısa ve hızlı çalışan bir kod yazabiliriz, ancak bu kodu anlamak daha zor olabilir, ayrıca hata yapma olasılığı da artabilir. Örneğin, bir `int` dizisini “temizlemeye” yarayan

```
int z[N], i;  
...  
for (i=0; i<N; i++)  
    z[i] = 0;
```

şeklindeki klasik `for` deyiminin yerine

```
int z[N], *g;  
...  
for (g=&z[0]; g<&z[N]; g++)  
    *g = 0;
```

kullanılabilir; bu da daha verimli (hızlı) olduğu için bazı programcılar tarafından tercih edilebilir.

Burada iki yeni işleç görmekteyiz: * ve &. Bunlar tekli işleçlerdir, ve * ile & ikili işleçleriyle karıştırılmamalıdır. Bir sonraki altkısımda bunlar anlatılmaktadır.

3.1.1. & Ve * İşleçleri

Zaman zaman, bir değişkenin depolandığı bellek bölgesinin adresini elde etmek gerekebilir. Bunu yapmak için, *adres alma* (&) işlecini kullanırız. Diğer tekli işleçlerle aynı önceliğe sahip olan bu tekli işleç, bellekte belli bir konumda bulunan işlenenlere uygulanabilir. Örneğin, bu işlenen basit bir değişken (°isken) veya bir dizi elemanı (&dizi[indis]) olabilir. & işleci işlenenin adresini (daha doğrusu, işlenen tarafından tutulan belleğin ilk baytının adresini) verir.

Bunun ne anlama geldiğine bakalım. Her değişken (ana) bellekte bir yerde saklanır. Bilgisayar belleği sınırlıdır; herbiri bir sayı verilerek adreslenen belirli sayıda hücrelerden oluşur. Her hücre belirli bir miktarda bilgi içerir. En küçük bilgi birimi *bittir* (binary digit—ikili rakam) ve 0/1, doğru/yanlış, açık/kapalı gibi, iki olası değerden bir tanesini saklamaya yarar. Bir bit, rahat olarak işlenmesi için çok küçük bir bilgi birimidir. Bundan dolayı, bitler, genelde 8 bitten oluşan bir *bayt* veya makineden makineye 8 ile 64 bit arasında değişen sayıda bitten oluşan bir *sözcük* şeklinde gruplandırılır.

Bizim sistemde, bir bayt 8, bir sözcük de 16 bittir (“16 bitlik” derleyiciler için); bilgisayar belleği bayt şeklinde adreslenir; arka arkaya gelen baytlar için arka arkaya gelen adresler kullanılır; normal boyda bir tamsayı değişkeni bir sözcük yani iki bayt kaplar. Böylece, iki tamsayıdan oluşan bir dizi (**int** z[2];) dört bayt kaplar. Eğer ilk elemanın adresi, örneğin, 140’sa, ikinci elemanın 142 olacaktır. *Adres alma* işleci bu sayıları verecektir. Diğer bir değişle, &z[0] 140’a eşitse, &z[1] de 142’ye eşittir.

Adres elimizde olduğu zaman, bu sefer o adreste saklanmış olan değeri elde etmek için bir yömeme gerek duyarız. Bu, temelde & işlecinin aksi olan, *dolaylama* (*) işleci ile yapılır. Bu işleç çarpım işleci ile karıştırılmamalıdır, çünkü bu *tekli*, oysa çarpım işleci *ikili* bir işleçtir. Dolaylama işleci diğer tekli işleçlerle aynı önceliğe sahiptir ve bütün tekli işleçler gibi sağdan sola birleşir. Bu da, eğer yan yana iki tekli işleç varsa, önce sağdakinin çalışacağını gösterir. Örneğin, *&x ile *(&x) aynı anlama gelir, ve her ikisi de, aşağıda açıklanacağı gibi x’e eşdeğerdir.

Dolaylama işleci, işlenen olarak geçerli bir adres bekler ve bu adreste saklanmış bulunan değeri verir. Ancak bunu yapması için hangi veri tipinde bilgi vermesi gerektiğini bilmek zorundadır; **int**, **double**, **char** yoksa başka bir şey mi? Bu bilgi işleneninde gizlidir. Örneğin, &x, x değişkeninin *tipindeki* bir bilgiyi tutan bellek bölgesinin adresidir (teknik terimi ile, bu adrese bir *göstergedir*). Yani, eğer x bir **char** değişkeni ise, &x bir **char** göstergesidir, böylece *&x, &x ile gösterilen bölgede saklanmış bulunan **char** değerini verir.

3.1.2. Gösterge Değişkenleri Bildirimleri

Gösterge bildirimleri basittir.

```
int i, j;
```

bildirimi nasıl tamsayı değişkenleri tanımlarsa,

```
int *ig, *jg;
```

bildirimi de tamsayı tipindeki değişkenlere göstergeler tanımlayacaktır. *ig* ve *jg* gibi gösterge değişkenleri tamsayı değişkenlerinin adreslerini içerecektir. Örneğin:

```
ig = &i;  
jg = &j;
```

anlamli atamalardır. Ancak

```
ig = j;
```

şeklindeki atama, *j* “adresini” *ig*’ye koyacaktır ve ancak *j* değişkeni içindeki tamsayı değerinin bir adres olarak kullanılması gerektiği durumlarda işe yarayabilir. C Standardına uygun derleyiciler böyle atamalar olduğunda sizi uyaracaktır, çünkü değişkenlerin tipleri uymamaktadır.

Aynı şekilde,

```
double *dg;
```

double’a bir gösterge tanımlar. Hem *ig* hem de *dg* adres saklarlar. Fakat neyin adresi? Derleyicinin bunu bilmesi gerekir. Bundan dolayı farklı tanımlar yapmak zorundayız. Örneğin,

```
dg = &i;
```

derleyici tarafından kabul edilebilir, ancak anlamsızdır, çünkü *dg*, bir **double** yerine bir tamsayıyı gösterecektir. Derleyiciler bu tip yanlış atamalarda sizi uyaracaktır. Lütfen bu uyarıları dikkate alın.

3.1.3. Gösterge Aritmetiği

Şimdi, bu kısım başında verilen örneğe geri dönelim ve ne olduğuna bakalım. Dizi bildirimi, derleyicinin dizi için, ana bellekte yer ayırmasını sağlar. Bir **int** değişkeninin 2 bayt tuttuğunu ve *N*’nin 10 olarak `#define` ile tanımlandığını varsayın. Bu durumda *z* dizisi 20 bayt tutacaktır. İlk eleman, adresi `&z[0]` ifadesi ile elde edilen yerde saklanır. (Tanım gereği `&z[0]` ile *z* aynı şeydir ve buna dizinin *temel adresi* denir. Diğer bir deyişle, bir ifade içinde indissiz kullanılan dizi isimleri belirli bellek bölgelerini gösterirler, yani göstergedirler.)

Bir dizinin *i*’inci elemanını elde etmek için (örneğin `z[i]` yazıldığında), derleyici şunu yapar: Dizinin temel adresini (yani ilk elemanının adresini) alır (örneğin 8566), sonra indisin değerini alır (örneğin 3) ve eğer dizinin eleman tipi *n* tane bayt kullanıyorsa

(örneğin, bizim sistemde tamsayılar için bu 2 bayttır) i 'inci elemanın adresini bulmak için $TemelAdres+n \times i$ hesaplar. Şimdi dizi indislerinin neden sıfırdan başladığı daha iyi anlaşılıyor; eğer i sıfıra eşitse, o zaman ikinci terim yok olur ve, olması gerektiği gibi, dizinin ilk elemanının adresinin gerçekte dizinin temel adresi olduğu ortaya çıkar.

for deyimlerimize geri dönersek, ilkinde bu hesaplamanın her yinelemede tekrarlandığını görürüz. İkinci durumda ise, açıkça bir gösterge kullanılmıştır. Tamamen

```
for (g=z; g<&z[N]; g++)
    *g = 0;
```

deyimiyle eşdeğer olan bu deyim, önce g **int** göstergesine z dizisinin temel adresini koyar (örneğin 8566); daha sonra g ile işaretlenmiş bölgeye sıfır koyar. Bu adreste (yani 8566'da) saklanacak olan değer tipinin derleyici tarafından bilinmesi gerekir, **int**, **char**, **double**, yoksa başka bir şey mi? Her tipin bellekte farklı sayıda bayt kapladığını anımsayın. Bunu göstergenin tipinden belirler. Bu bir tamsayı göstergesi olduğuna göre, bu gösterge tarafından işaret edilen bölgede bir tamsayı değişkeni bulunmalıdır. Bundan dolayı 8566 ve 8567 adresli baytlara sıfır yerleştirir.

Bundan sonra gösterge artırılır. Şimdi yine bir sorumuz var demektir, çünkü sayılar için artırma (veya azaltma) elimizdeki değere bir sayısı eklenerek (veya çıkarılarak) yapılır. Aynı şey göstergelere uygulanamaz. g 'nin gösterdiği adresin değerine bir eklenmesi, göstergenin dizideki bir sonraki elemana (yani 8568 adresine) işaret etmesini sağlayamayacaktır; onun yerine, ilk elemanı içeren "sözcüğün" ikinci yarısını (yani 8567 adresini) gösterecektir. Gösterge ile gösterilen elemanın boyuna eşit bir sayı (yani bizim sistemimizde tamsayılar için 2) eklemeliyiz. Bu, derleyici tarafından bizim için yapılır. Yani, **int** göstergesi, **++** işleci ile artırıldıktan sonra 8568 adresini göstermeye başlayacaktır. Değişik tipteki göstergelerin uyumlu olmamalarının nedenleri işte bu farklılıklardır.

Gösterge aritmetiğinin diğer şekillerinde de benzer bir sorun ortaya çıkar. Örneğin, eğer g 'de 8566 ($\&z[0]$) adresi varsa, $g+3$ ifadesi nereyi göstermelidir? 8569 adresini mi? Hayır! Bu, $z[1]$ 'i içeren sözcüğün ikinci yarısının adresi demektir ki fazla anlamlı bir şey değildir. Daha iyisi $\&z[3]$ (yani 8572) adresine işaret etmesi olacaktır. Yine, derleyici bize bunu sağlar: Bir adrese (göstergeye), z , bir tamsayı, i , eklediğimizde (veya çıkardığımızda), derleyici z tarafından işaret edilen değişken tipinin boyunu i ile çarpar ve sonra toplama (veya çıkarmayı) yapar. Bunun nedeni adreslerin her zaman baytlara işaret etmelerine karşılık, göstergelerin değişik büyüklüklere sahip tipte nesneleri göstermeleridir.

Bu anlatılanların anafikri *göstergelerin tamsayı olmadıkları* ve bazı işleçlerin (**++**, **--**, **+**, **-**, **+=**, **-=**) göstergeler için farklı uygulandıklarıdır. $*g$ ile gösterilen değişkenin n bayt kapladığını varsayın. O zaman,

```
g++
g--
g+tamsayi_ifade
g-tamsayi_ifade
```


gösterge ifadeleri sırasıyla $g+n$, $g-n$, $g+n \times \text{tamsayı_ifade}$, $g-n \times \text{tamsayı_ifade}$ şeklinde hesaplanır. Ayrıca $p1$ ve $p2$ aynı tipten göstergelerse, değeri bir tamsayı olan

$p1-p2$

ifadesi $\frac{p1-p2}{n}$ şeklinde hesaplanır. Örneğin,

```
p1 = &z[3];
p2 = &z[0];
printf("%d\n", (int)(p1-p2));
```

(z dizisinin eleman tipine bağlı olmaksızın) çıktı olarak 3 verecektir. İki göstergiyi birbirinden çıkardığımızda, kullanılan sistemdeki bellek boyuna bağlı olan bir tipte bir ifade oluştuğuna dikkat edin. İşaretli olan bu tip `stddef.h` adlı başlık dosyasında uygun bir şekilde `ptrdiff_t` adıyla tanımlanmıştır.

İki uyumlu gösterge ($<$, $>$, $<=$, $>=$, $=$ ve $!=$ kullanarak) karşılaştırılabilir, ancak farklı dizilerdeki nesnelere işaret eden iki göstergiyi karşılaştırmak anlamlı (en azından taşınabilir bir özellik) değildir. Doğal olarak, bunun nedeni iki dizinin ana bellekteki birbirine göre durumları konusunda emin olamayacağınızdır. Bu kısımdaki ikinci `for`'da kullanılan "`g<&z[N]`" testi uygundur, çünkü g ve $\&z[N]$ aynı dizi içindeki (olası değişik) yerlere işaret ederler. $\&z[N]$ adresinin z dizisi içinde olmadığını söyleyerek, buna karşı çıkabilirsiniz, zira dizinin son elemanı $z[N-1]$ 'dir. Haklısınız, ancak g 'nin bu "geçersiz" adrese işaret ettiği zaman $*g$ 'ye birşey atamamak için, $<=$ yerine $<$ kullanacak kadar dikkatli olduğumuzu da gözlemleyin. C Standardı ise, bir nesnenin sınırları dışında adres üretmeyi—o adresteki bölge değiştirilsin değiştirilmesin—açıkça yasaklamaktadır. Tek istisna olarak, bir nesnenin son baytından bir sonraki baytı gösteren adresi üretmeğe izin verir. İşte bu özellik "`g<&z[N]`"de kullanılmıştır.

Bir gösterge ($=$ veya $!=$ kullanılarak) 0 tamsayı değişmeziyle karşılaştırılabilir. Geleneksel olarak 0 değerinin hiç bir şeyi göstermediği varsayılmıştır. Bu tamsayı değişmezi programlarda o kadar çok kullanılmaktadır ki `stdio.h` başlık dosyasında

```
#define NULL 0
```

şeklinde bir tanımlama yapılmıştır. Böylece, eğer 0 yerine `NULL` kullanılacak olursa "göstergeler tamsayı değildir" kuralı bozulmamış gibi görünecektir. Bir göstergenin herhangi bir yeri göstermesini istemiyorsanız

```
g = NULL;
```

atama deyimini kullanın. `NULL`'un her tip gösterge ile uyumlu olduğuna dikkat edin. Bu özel bir durumdur.

3.2. Göstergeler Ve Diziler

Şimdi, gelin aşağıdakini inceleyelim

```

int    z[N], i;
...
for (i=0; i<N; i++)
    *(z+i) = 0;

```

Buradaki atama bir önceki kısmın başında verilenle tamamen aynıdır. Neden? z ile $\&z[0]$ aynı şey olduğunu zaten biliyorsunuz. Böylece $*(z+i)$ ile $*(&z[0]+i)$ aynı şeydir. Yukarıda gösterge toplama işlemleri ile ilgili anlatılanlara göre, aynı zamanda $*(&z[i])$ ile eşdeğerdir. Şimdi, $*$ ile $\&$ birbirinin aksi işlemleri olduğuna göre, birbirlerini götürürler. Böylece ifade $z[i]$ olur. Aslında, derleyici $z[i]$ şeklindeki bir ifadeyi hemen $*(z+i)$ şekline dönüştürür, yani ilk şekil programcının rahatlığı için dilde sağlanan bir kısaltma gibi düşünülebilir. Hatta $z[i]$ yerine, biraz garip olan, $i[z]$ ifadesini de kullanabilirsiniz.

Dizi isimlerinin gerçekte gösterge olduğunu bildiğinize göre,

```

float  z[N];

```

ile

```

float  *g;

```

arasındaki fark nedir, diye sorabilirsiniz.

```

g = z

```

ataması bir yapıldı mı,

```

g,    &z[0],    ve    z,

```

kendi aralarında,

```

g+i,    &z[i],    ve    z+i,

```

ve

```

*(g+i),    z[i],    ve    *(z+i)

```

de, aynı şekilde, kendi aralarında eşdeğerdir.

Fakat önemli bir fark var:

```

z = g;

```

yazamazsınız çünkü g bir *değişken gösterge*, oysa z bir *değişmez göstergedir*. Şimdi değişken tanımlamalarına geri dönelim.

```

float  z[N];

```

derleyicinin, ana bellekte N **float** değişken için gerekli yer ayırmasına yol açacaktır. (Bizim sistemimizde $4 \times N$ bayt.) Bu bellek öbeğinin başlangıç adresi z *değişmezinin* (gösterge) değeri olur. z bir değişken değildir, çünkü bu (değişmez) adresi saklamak için bellekte ayrıca yer ayrılmamıştır. Diğer yandan,

```

float  *g;

```

bir adres tutacak genişlikte bir yerin ayrılmasını sağlar. (Bizim sistemimizde 2 bayt.) Programın yürütülmesi esnasında, bu *g değişkeni* değişik zamanlarda değişik değerler tutabilir. Başlangıçta ise, şimdiye kadar gördüğümüz değişken türleri için, değeri belirsizdir.

3.3. Karakter Dizileri

C dilinde özel bir “karakter dizisi” tipi yoktur. Ancak karakter dizisi gerektiren durumlar için bazı kolaylıklar bulunmaktadır. Karakterlerden oluşan normal bir dizi veya bir karakter göstergesi bir karakter dizisi olarak düşünülebilir.

```
#include <stdio.h>

void main (void)
{
    char *dd;
    dd = "dunya";
    printf("Merhaba, %s.\n", dd);
}
```

Bildiğiniz gibi, `printf`’in ilk argümanı bir karakter göstergesi olmalıdır; ikinci argüman da, `%s` dönüşüm tanımlamasının gerektirdiği şekilde bir karakter göstergesidir. Yukarıdaki atama ifadesinde olduğu gibi, bir ifade içinde kullanılan bir karakter dizisi değişmezi bir karakter göstergesidir ve dizinin ilk karakterine işaret eder. Karakter dizisi, derleyici tarafından, bir tarafta saklanır. Böylece, yukarıdaki atama deyiminde sadece bir gösterge ataması yapılmaktadır; bir karakter dizisi aktarması söz konusu değildir. Eğer bir karakter dizisi aktarması yapmak istiyorsak bunu kendimiz yapmalıyız. Örneğin,

```
char d[10];
d = "bir dizi";
```

geçersizdir, çünkü derleyici gösterge ataması yapmaya çalışacaktır. Ancak “=” işaretinin sol tarafında bir *değişmez* gösterge bulunduğundan, bu olanaksızdır. Karakterleri tek tek, bir döngü kullanarak, atamak zorundayız. Böyle işlemler çok sık kullanıldığı için, C kütüphanesi bazı *karakter dizisi işleme fonksiyonları* sağlamaktadır. Örneğin,

```
strcpy(d, "bir dizi");
```

bizim işimizi görecektir. Bu fonksiyon, ikinci argüman olarak geçirilen adresteki bütün karakterleri, en sonda gelen boş karakteri (`\0`) ile birlikte, `d` tarafından gösterilen yere aktaracaktır. Bu durumda, bu 9 karakterdir. Bu fonksiyon, hedef dizinin bu kadar çok karakteri alacak genişlikte olup olmadığını kontrol etmez. (Zaten edemez.)

Benzer bir durum karakter dizileri karşılaştırması yaparken ortaya çıkar.

```
if (s=="Bu dizi") ...
```

testi her zaman yanlış sonucunu verecektir. Kuşkusuz bunun nedeni gösterge eşitliğini test etmemizdir. `s` ya bir gösterge yada bir dizi ismi olabilir (her iki durumda da bir gösterge

ifadesidir); değişmez karakter dizisi ise onu saklamak için kullanılan bellek öbeğinin başına işaret eden bir göstergedir. Doğal olarak, bunlar farklı adreslerdir, böylece bu test hiçbir zaman “doğru” sonucunu vermeyecektir. Bunun yerine, `s1` ve `s2` şeklinde verilen iki karakter dizisini karşılaştırıp, sözlük sıralamasına göre `s1`’in `s2`’den küçük, `s2`’ye eşit veya `s2`’den büyük olmasına bağlı olarak, sırasıyla olumsuz bir tamsayı, sıfır veya olumlu bir tamsayı veren `strcmp(s1, s2)` fonksiyonunu kullanmalıyız.

`strcat(d, s)` fonksiyonu `s` dizisini `d`’nin arkasına aktarır. Kuşkusuz `d` dizisinin hem eski hem de yeni karakter dizisini kapsayacak kadar geniş olması gerekir.

`strlen(s)` `s` karakter dizisinin uzunluğunu verir; bunu yaparken en sondaki boş karakteri saymaz. `strchr(s, c)` `c` karakterinin `s` karakter dizisinin içinde ilk ortaya çıktığı yerin adresini bir karakter göstergesi olarak döndürür; `c` dizi içinde yoksa `NULL` verir. `strrchr(s, c)` `strchr(s, c)` gibidir, sadece tarama işlemini sağdan sola doğru yapılır.

Yukarıdaki fonksiyonlardan bazıları için işlemi sınırlayan başka bir argüman, `n`, kabul eden bazı uyarlamalar bulunmaktadır. `strncmp(s1, s2, n)` en fazla `n` karakteri karşılaştırır; `strncat(s1, s2, n)` en fazla `n` boş olmayan karakter ekler; `strncpy(s1, s2, n)` tam olarak `n` karakter aktarır (eğer `s2`’nin uzunluğu daha kısa ise, boş karakterler eklenir).

Yararlı bir değer döndürmüyor gibi gözüken fonksiyonlar aslında karakter göstergesi döndürmektedir. Genelde, geri döndürülen gösterge hedef dizisinin başına işaret eder.

Bir sonraki bölümde de göreceğimiz gibi, fonksiyonları kullanmadan önce bildirimlerini yapmak zorundayız. Bu kısımda anlatılan fonksiyonların bildirimi `string.h` başlık dosyasında bulunmaktadır.

Diğer birçok fonksiyon yanında, C kütüphanesinde bazı *veri dönüşüm fonksiyonları* da mevcuttur. Bunlardan bazılarının bildirimi `stdlib.h` başlık dosyasında bulunmaktadır ve aşağıda açıklanmıştır. Ayrıca fonksiyon ve argüman tipleri de gösterilmiştir.

double `atof(char *)`

karakter dizisi argümanını çift duyarlıklı kayan noktalı sayıya çevirir; hata durumunda 0.0 döndürür.

int `atoi(char *)`

karakter dizisi argümanını tamsayıya çevirir; hata durumunda 0 döndürür.

long `atol(char *)`

karakter dizisi argümanını uzun tamsayıya çevirir; hata durumunda 0L döndürür.

char * `itoa(int, char *, int)`

ilk tamsayı argümanını karakter dizisine çevirir, sonucu ikinci argüman olarak verilen adrese koyar; üçüncü argüman tabanı belirtir.

char * `ltoa(long, char *, int)`

ilk uzun tamsayı argümanını karakter dizisine çevirir, sonucu ikinci argüman olarak verilen adrese koyar; üçüncü argüman tabanı belirtir.

`int toupper(int)` ve `int tolower(int)`

sırasıyla karakter argümanının büyük harf ve küçük harf karşılığını döndürür.

3.4. Bitisel İşleçler

İşleçler C'nin önemli bir konusudur. Ancak, hepsinin birlikte anlatılacakları tek bir yer yoktur. En iyisi gerek duyulduğunda onları anlatmaktır. Bu bölümde, göstergelerle beraber kullanıldıkları için, iki tekli işleç (& ve *) anlatılmıştı.

Bitisel işleçler başka şeylerle pek ilgisi olmayan bir grup oluşturdıklarından, hepsini bu kısımda anlatmaya karar verdik. İşleçlerin azalan önceliklerine göre anlatılmaları açısından bu kısım Kısım 2.2'ye benzemektedir. Bire tümler ve atama işleçleri sağdan sola doğru, diğerleri soldan sağa doğru birleşirler. Bu kısımda anlatılan tüm işleçler sadece (çeşitli boylarda olabilen) tamsayı işlenenleri kabul ederler ve işlenenler üzerinde işlem yapmadan önce Kısım 1.6'da anlatılan tip dönüşümlerini gerçekleştirirler. Atama ile ilgili olanlar hariç, eğer işlenenler değişmez ifadeler ise, değişmez ifadeler oluştururlar. Bu işleçlerin isimleri ve kullanılan simgeler şöyledir:

<u>işleç</u>	<u>isim</u>
~	bire tümler
<<	sola kaydırma
>>	sağa kaydırma
&	bitisel VE
^	bitisel dışlayan VEYA
	bitisel VEYA
<<=	sola kaydırma ve atama
>>=	sağa kaydırma ve atama
&=	bitisel VE ve atama
^=	bitisel dışlayan VEYA ve atama
=	bitisel VEYA ve atama

Detaya girmeden önce, bitisel işleçlerin mantıksal olarak ne yaptığını gösteren bir çizelge sunalım. Aşağıda y ve z tek bitlerdir:

y	z	$\sim y$	$y \& z$	$y \wedge z$	$y z$
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	0	1

Bitisel bir işleç işlenen(ler)in her bitine yukarıda anlatılan şekilde uygulanır.

Bire Tümler İşleci ~

~ işleci işleneninin 1 olan bitlerini 0'a, 0 olan bitlerini de 1'e çevirir. Sonucun tipi işlenenin tipiyle aynıdır.

Kaydırma İşleçleri << Ve >>

Bir kaydırma ifadesi

ifade_1 << *ifade_2*

veya

ifade_1 >> *ifade_2*

şeklinde. Normalde *ifade_1* bir değişken veya dizi elemanıdır. << işleci durumunda, tip dönüşümlerinden sonra, *ifade_2* **int**'e çevrilir ve ilk işlenenin bitleri *ifade_2*'nin değeri kadar sola kaydırılır. İşlenenin solundan "düşen" bitler kaybedilir. Sola kayan bitlerden arta kalan sağdaki boş yerler sıfırla doldurulur. Sonucun tipi soldaki işlenenin tipidir.

>> işleci yukarıda anlatılanların sağa doğru olan şeklini yerine getirir, tek bir farkla: Eğer soldaki ifade **signed** (işaretli) ise boş kalan yerler, 0 yerine, işaret bitiyle *doldurulabilir*. Buna aritmetik kaydırma denir ve sistemimiz bunu yapar. Eğer ilk ifade **unsigned** (işaretsiz) ise solda boşalan bitlerin sıfırla doldurulacağı garanti edilir ve bir mantıksal kaydırma söz konusudur.

Bitsel VE İşleci &

İkili & işleci işlenenlerinin bitlerinin VEsini verir. Yani eğer her iki işlenenin *i*'inci bitleri 1 ise, sonucun *i*'inci biti de 1 olur. Aksi takdirde 0'dır. Tabii ki,

(a & b) & c

ve

a & (b & c)

aynı değerlere sahiptirler (yani & işlecinin birleşme özelliği vardır). C bu gerçeği kullanır ve derleyici, daha verimli olacağı gerekçesiyle, parantez içine alınmış olsa dahi, böyle ifadeleri (yani, * ve + gibi birleşme ve değişme özelliği gösteren işleçleri içeren ifadeleri) istediği şekilde yeniden düzenleyebilir.

Bitsel Dışlayan VEYA İşleci ^

^ işleci işlenenlerinin dışlayan VEYAsını verir. Yani eğer işlenenlerin *i*'inci bitleri farklı ise, sonucun *i*'inci biti 1 olur. Aksi takdirde 0'dır. ^ birleşme özelliği gösteren bir işleçtir ve bu işleci içeren ifadeler, önceki altkısımda anlatıldığı gibi, yeniden düzenlenebilir.

Bitsel VEYA İşleci |

| işleci işlenenlerinin kapsayan VEYAsını verir. Yani eğer her iki işlenenin i 'inci bitleri 0 ise, sonucun i 'inci biti de 0 olur. Aksi takdirde 1'dir. | da birleşme özelliği gösteren bir işleçtir ve yeniden düzenleme söz konusu olabilir.

Atama İşleçleri <=<, >>=, &=, ^= Ve |=

Yukarıda anlatılan işleçler belirli ifadeler oluşturmalarına rağmen, işlenenlerinin değerlerini değiştirmezler. Şimdi anlatacağımız atama işleçleri ise işlemin sonucunu soldaki işlenenlerine aktarırlar.

Bölüm 1'deki +=, -=, *= vs işleçlerini anımsayın. Her bir ikili bitsel işlecin karşılığında bir atama işleci mevcuttur, << için <=<=, >> için >>=, & için &=, ^ için ^= ve | için |=. Bu işleçlerin işlevselliği diğer atama işleçlerine benzer.

Bitsel İşleçler—Örnekler

short, **long** gibi tiplerin uzunlukları derleyiciden derleyiciye değiştiğine göre, buradaki örnekler sadece bizim sistemimiz için geçerlidir. Aşağıdaki bütün örneklerde ifadenin değeri sekizli tabanda verilecektir. Sonucun tipi de ayrıca belirtilecektir. Bütün değişkenlerin ilk değeri 5'tir. Tipler ise şöyledir:

```
/* signed */ char c; short s; int i; long l;
```

<u>ifade</u>	<u>değer</u>	<u>tip</u>
~c	0372	char
~s	0177772	int
~i	0177772	int
~l	0377777772	long
c << 3	050	char
s >> 2	01	int
i << 1	0240	int
c & s	05	int
-i & ~l	0377777772	long
c ^ 3	06	int
~s ^ 2	0177770	int
i ^ 1 & i	0	long
~c 3	0177773	int
s s >> 1	07	int
i 1	05	long
c <=<= s	0240	char
l >>= i	0	long
s &= -i	01	short
i ^= 1+1	03	int
c = 16	025	char

Bir Örnek—Sayı Paketleme

Bitsel işleçlerin kullanımını göstermek için, üç sayıyı bir tamsayıya “paketleyen”, daha sonra ise “açan” aşağıdaki programı inceleyelim.

```

1.  #include <stdio.h>
2.  #define GUN_UZ      5
3.  #define AY_UZ       4
4.  #define TABAN_YIL   1900
5.  #define TAVAN_YIL   (TABAN_YIL+(1<<16-GUN_UZ-AY_UZ)-1)
6.  void main (void)
7.  {
8.      unsigned const ay_ortu  = ~(~0<<AY_UZ);
9.      unsigned const gun_ortu = ~(~0<<GUN_UZ);
10.     unsigned yil, ay, gun, tarih;
11.
12.     do {
13.         scanf("%u%u%u", &yil, &ay, &gun);
14.     } while (yil<TABAN_YIL || yil>TAVAN_YIL ||
15.             ay<1 || ay>12 || gun<1 || gun>31);
16.     yil -= TABAN_YIL;
17.     tarih = (yil<<AY_UZ|ay)<<GUN_UZ | gun;
18.     printf("Paketleme tamamlandı.\n");
19.
20.     yil = ay = gun = 0; /* degiskenleri "temizle" */
21.     gun = tarih & gun_ortu;
22.     ay = (tarih>>GUN_UZ) & ay_ortu;
23.     yil = tarih>>(GUN_UZ+AY_UZ);
24.     printf("Acma tamamlandı.\nTarih: %u %u %u",
25.           yil+TABAN_YIL, ay, gun);
26. } /* main */

```

Program başladığında 1900 ile 2027 yılları arasında bir tarih kabul eder. (Satır 12-15.) Sadece bazı basit kontroller yapılmıştır; 13’üncü ayın 54’üncü günü giremezsiniz, ancak Şubat’ın 31’ini girebilirsiniz. Sadece 1900 yılından beri geçen yılları sakladığımız için, Satır 16’daki `--` işlemi yapılır. Paketleme Satır 17’de yapılır: `yil` içindeki geçen yıllar `AY_UZ` (yani 4) bit sola kaydırılır, `ay` yeni boşalan yere `VEYAlanır`, sonra bunun tamamı `GUN_UZ` (5) bit sola kaydırılır ve `gun` içeriye `VEYAlanır`. Paketleme sonunda, tarihin en solundaki 7 bitinde geçen yıllar, en sağdaki 5 bitte gün ve geri kalanında ay bulunur. Bizim sistemimizde **short** bir tamsayı 16 bit kapladığına göre, tarihteki bütün bitler kullanılmış olur. **do** deyimindeki `erim` (yayılma aralığı) kontrolleri, bu sayıların hiçbirinin kendileri için ayrılan miktardan daha fazla bir yer isteğinde bulunmayacaklarını sağlamak içindir.

Satır 20 açma işleminin gerçekten çalışıp çalışmadığı kuşkusunu bertaraf etmek içindir ve programın davranışında herhangi bir değişikliğe yol açmadan çıkarılabilir.

Satır 21 tarihin gün bitlerini 1’le, geri kalanını da 0’la `VEleyerek` günü alır. Buna, belli nedenden dolayı, “örtme” denir. Satır 10’da ilklenen, `gun_ortu` değişkeni gerekli bit kalıbını içerir. (`~0`’ın bütün bitleri 1’dir.) Satır 22, önce ayın en sağdaki 4 bite geleceği şekilde tarihi kaydırır, sonra da ay dışındaki bitleri örterek ayı elde eder. Satır 23’te, tarih, sonuçta geçen yılların elde edileceği şekilde, sağa kaydırılır. tarih **unsigned** olduğuna göre en solda hep 0 vardır. Program yeni açılan tarihi yazarak sona erer.

Bu arada, daha ileriki bir bölümde aynı paketlemeyi yapacak başka bir yöntem göreceğiz.

3.5. İşleç Önceliği Ve Birleşme

İşleç önceliği ve birleşme çizelgesinin en son durumu Çizelge 3.1’de gösterilmiştir. Bu çizelge yeni tanıtılan işleçlerle daha ileriki bölümlerde tanıtılacak olan işleçlerin tamamını içerir.

ÇİZELGE 3.1 C işleç önceliği ve birleşme

() [] -> .	→
~ ! ++ -- sizeof (tip)	
+(tekli) -(tekli)	←
*(dolaylama) &(adres alma)	
* / %	→
+ -	→
<< >>	→
< <= > >=	→
== !=	→
&	→
^	→
↓	→
&&	→
	→
? :	←
= *= /= %= += -=	←
<<= >>= &= ^= =	
,	→

Şimdi bu çizelgenin ne anlama geldiğini bir daha gözden geçirelim. Bu çizelge, C dilindeki *bütün* işleçleri, azalan öncelik sıralarına göre liste halinde vermektedir. Aynı satırda verilmiş olan işleçler aynı *önceliğe* sahiptirler. Eğer aynı önceliğe sahip iki işleçimiz varsa, o zaman çizelgenin sağ sütununda gösterilen *birleşmeyi* kullanırız. Sağa doğru ok (→) soldan sağa birleşmeyi ifade eder, örneğin “x/y/z” ile “(x/y)/z” aynı anlamı taşırlar. Sola doğru ok (←) sağdan sola birleşmeyi belirtir, örneğin “x=y=z” ile “x=(y=z)” eşdeğer ifadelerdir. Tüm tekli işleçler, üçlü (koşullu) işleç ve tüm atama işleçleri sağdan sola doğru gruplanırlar; geri kalanların soldan sağa doğru birleşme özelliği vardır. Bu çizelgede gösterilen bazı işleçler henüz anlatılmamıştır. Bunlar: (), ., -> ve **sizeof**’tur.

Problemler

1. Bizim sistemimizde

```
#include <stdio.h>
void main (void)
{
    int x;
    printf("%d %d %d\n", (int)&x, (int) (&x+1), (int) &x+1);
}
```

programı

8580 8582 8581

sonucunu görüntüler. Tahmin edebileceğiniz gibi bunlar bellek adresleridir. Neden ilk durumda fark 2, ikinci durumda ise 1'dir?

2. Eğer makinenizde (işaretli) bir tamsayı üzerinde yapılan bir sağa kaydırma (>>) işlemi esnasında boşalan yerler işaret bitiyle dolduruluyorsa isaret-doldurma, aksi takdirde sifir-doldurma mesajını verecek bir program yazınız. Sisteminiz hakkında (sözcük uzunluğu gibi) herhangi bir varsayım yapmayın.

3. Girilen bir tamsayıda 1 olan bitleri sayan bir program yazınız.

4. Aşağıdaki deyimler ne yapar?

```
a ^= b;
b ^= a;
a ^= b;
```

Açıklayınız. *İpucu:* Başlangıçta a'nın 0077, b'nin de 0707 içerdiğini varsayın. Son değerleri ne olacaktır?

5. Zaman da bir sözcük içine aşağıdaki gibi paketlenabilir: 5 bit saat, 6 bit dakika, 5 bit saniye. Saniyeleri 5 bite sığdırmak için değeri 2 ile bölün, yani saniyenin en sağındaki biti atın. Bunu yerine getirecek bir program yazın.

6. Aşağıdaki kodun ne işe yaradığını açıklayın:

```
int i;
for (i=0; i<10; i++)
    printf("%d:%c\n", i, "QWERTYUIOP"[i]);
```

BÖLÜM 4: FONKSİYONLAR VE PROGRAM YAPISI

Bir program bir görevi yerine getirmek için yazılır. Eğer yapılacak iş pek kolay değilse, program oldukça uzun olabilir. Bazı programlar onbinlerce satır uzunluğunda olabilir. Böyle durumlarda, esas görevi daha küçük ve kolay idare edilebilir altgörevlere ayırmadan yerine getirmek hemen hemen olanaksızdır.

C, böyle altgörevleri ifade etmek ve birbirinden ayırmak için bir yöntem öngörmektedir. Bu yöntem sayesinde “bilgi saklama” ilkeleri kullanılabilir. C sadece bir tek altprogram çeşidi sağlamaktadır, bu da *fonksiyondur*. Matematiksel olarak; bir fonksiyon, argümanlarını döndürdüğü değere ilişkilendiren bir “kara kutu” gibidir. Aynı şey C için de geçerlidir, ancak bir fonksiyon birden fazla değer döndürecek, küresel veri yapılarında yan etkiler yapacak veya girdi/çıkı sağlayacak bir şekilde de tasarımılanabilir. Örneğin, `printf` fonksiyonu, argümanlarının değerlerinin çıktısını yapmak için kullanılır. Geri döndürdüğü değer, yazılan karakterlerin sayısıdır, ancak bu fazla ilgi çekici değildir.

Çok “basit” bir fonksiyon tanımı aşağıda verilmiştir:

```
void f (void)
{ }
```

Bu fonksiyon hiçbir şey yapmamasına rağmen, program geliştirmesi esnasında yer tutucu olarak kullanılabilir. Bu bölümde fonksiyon tanımı ve kullanımı hakkında daha fazla şeyler öğreneceğiz.

Kitabın başından beri görmekte olduğumuz `main` tanıtıcı sözcüğü, programın yerine getirmesi gereken görevi ifade eden *fonksiyonun* ismidir. Her makul C programı içinde `main` adı verilen bir fonksiyonun bulunması gerekir, çünkü program yürütülmeye başlandığında, program içinde bağlanmış bulunan bir başlangıç yordamı çalışmaya başlar, bu da kontrolü sonuçta `main` fonksiyonuna geçirir. Eğer bağlama esnasında, bağlayıcı (*linker*) `main` adı verilen bir fonksiyon bulamazsa, sizin için yürütülebilir bir kod

oluşturamayacaktır. `main`'in bir anahtar sözcük olmadığına dikkat edin, yani isterseniz, `main` adında bir değişken de tanımlayabilirsiniz.

Fonksiyonların gücü, bizim tekrarlanan kod yazmamızdan kaçınmamızı sağlamalarında yatar. Herhangi bir kodu, bir fonksiyon şeklinde, bir defa belirtiriz, ona bir isim veririz ve daha sonra, bu kodu çalıştırma gereksinimi duyduğumuzda, bu fonksiyonu “çağırırız”. Bu iyi, fakat yeterli değildir. Birçok durumda bu kod o anki gereksinimlerimize “uyarlanmalıdır”. Örneğin, bir diziyi sıraya sokan bir kodumuz varsa, bu kodu her çağırdığımızda, değişik sayıda elemandan oluşan değişik bir dizi belirtecek şekilde bunu hazırlamamız daha iyi olur. Bu ise, *fonksiyon argümanları* kullanılarak yapılabilir.

Her fonksiyon diğerlerinden bağımsız olduğuna, yani bir fonksiyon içinde tanımlanmış değişkenler başka bir fonksiyonda kullanılmadığına göre, fonksiyonların birbirleriyle haberleşmelerini sağlayacak yöntemler geliştirilmiştir. Bir yöntem fonksiyonlara argüman geçirmektir. Başka bir yöntem *küresel değişkenler* kullanmaktır.

Şimdiye kadar olan örneklerde gördüğümüz bütün değişkenler fonksiyonlar (`main` fonksiyonu) içinde tanımlanmışlardı. Bu bölümde göreceğimiz gibi; değişken bildirimleri veya tanımlamalarını fonksiyonlar dışında yapmamız da olasıdır. Böyle değişkenlere *küresel* adını veririz, çünkü fonksiyon içinde tanımlanan *yerel değişkenlerin* aksine, bunlar, tanımlamadan sonra gelen program metni içindeki bütün fonksiyonlar tarafından kullanılabilirler.

4.1. Fonksiyon Tanımlama

Aslında bir fonksiyonu, yani `main` fonksiyonunu, nasıl tanımlayacağımızı zaten biliyoruz. Şimdi, bir fonksiyonun daha genel bir şeklini görelim:

*dönüş_tipi*_{opt} *fonksiyon_ismi* (*parametre_bildirimleri*_{opt})
blok

Dönüş_tipi fonksiyon tarafından döndürülen değerin tipini belirtir. Bu, **int**, **double** gibi bir aritmetik tip veya gösterge olabilir. Bir dizi, ve bazı eski derleyicilerde bir yapı veya birlik olamayabilir, ancak bunlara bir gösterge olabilir. Bir fonksiyon için varsayılan dönüş tipi **int**'tir, bu da geniş bir alanı kapsar, çünkü genelde mantıksal değerler (doğru/yanlış değerleri) ve karakterler için de **int** kullanırız.

Dönüş_tipi olarak **void** anahtar sözcüğü kullanıldığında, fonksiyondan hiçbir değer döndürülmeyeceği belirtilmiş olur. **void** fonksiyonlar FORTRAN'daki SUBROUTINE'ler veya Pascal'daki **procedure**'ların karşılığıdır.

Fonksiyon_ismi bir tanıtıcı sözcüktür. Eğer fonksiyon, ayrıca derlenen başka bir kaynak dosyadan çağrılacaksa, fonksiyon isminin sadece ilk 8 karakteri anlamlıdır, bu da bağlayıcı tarafından konulan sınırlamadan dolayıdır.

Parametre_bildirimleri virgülle ayrılmış ve önüne tipleri yazılmış *biçimsel argüman* veya *parametre* isimleri listesidir. Diğer bir fonksiyon tarafından çağrıldığında bu fonksiyona geçirilecek olan argümanların sayısını, sırasını ve tipini belirtir. Aynı zamanda, parametrelere, sadece fonksiyonun gövdesini oluşturan *blok*'un içinde anlamlı olan, isimler verir. Hiç parametresi olmayan bir fonksiyon da olabilir. Bu durum, parantezler içine **void** yazılarak açıkça gösterilir. Örneğin,

```
rand (void)
{
    fonksiyonun gövdesi
}
```

Blok, her biri isteğe bağlı olan, iki kısımdan oluşan bileşik bir deyimdir: Bir değişken veya tip tanımlama veya bildirim kısmı ile onu izleyen deyimler. Kısım 1.8'e bakınız. Bildirim kısmı, fonksiyon gövdesi içinde yerel olarak kullanılacak olan nesnelerin tanımını veya bildirimini içerir. Deyimler, fonksiyon gövdesinin yürütülebilir kısmını oluştururlar.

Bir fonksiyon başka bir fonksiyon içinde tanımlanamaz. Diğer bir deyişle bir fonksiyon tanımı diğer bir fonksiyon tanımının bitmesinden sonra başlar. Fonksiyon tanımlamalarının sırası önemsizdir, ancak bu bölümde daha ileride bahsedeceğimiz dikkat edilmesi gereken bazı küçük noktalar vardır.

Şimdi, bir örnek. C'nin "mutlak değer alma işleci" yoktur. Tamsayılar üzerinde çalışan bir fonksiyon tanımlayalım.

```
abs (int n)
{
    if (n >= 0)
        return n;
    else
        return -n;
}
```

Burada **return** adında yeni bir deyim görmekteyiz. Bu genelde,

```
return ;
```

veya

```
return ifade;
```

şeklinde olur. Kontrolün çağırana geri dönmesini sağlar. **return** deyiminden sonra gelen deyimler yerine getirilmez ve fonksiyon hemen çağırıldığı yere "geri döner". İlk şekilde kontrol geri döner, ancak yararlı hiç bir değer döndürülmez. Geri döndürülen değer tanımsız olduğu için, çağırana fonksiyon bunu kullanmaya çalışmamalıdır. İkinci şekilde *ifadenin* değeri fonksiyonun değeri olarak geri döndürülür. Eğer *ifadenin* değerinin tipi fonksiyonun dönüş tipiyle aynı değilse, otomatik tip dönüşümü yapılır. "Dönüş tipi" **void** olan fonksiyonlar için ikinci şeklin kullanılamayacağı açıktır.

Sağ çengelli parantezin önünde bir **return** olduğu varsayılır; bundan dolayı belli bir değer döndürmek istemiyorsanız **return** yazmayabilirsiniz. Son deyim yerine getirildikten sonra, yararlı bir değer olmadan kontrol çağırان fonksiyona döndürülecektir.

4.2. Fonksiyon Çağrıları

Bir fonksiyonu çağırarak için fonksiyon ismini ve virgülle ayrılıp parantez içine alınmış *argüman*ların bir listesini belirtin. Fonksiyonun argümanları olmasa dahi parantezler bulunmalıdır, böylece derleyici ismin bir değişkeni değil de bir fonksiyonu gösterdiğini anlayabilecektir. Fonksiyon tarafından geri döndürülen değer kullanılabilir veya kullanılmayabilir. Örneğin,

```
abs(-10);
```

bir fonksiyon çağrısıdır. Bu noktada kontrol, çağırان fonksiyondan çağırılan fonksiyona aktarılır. -10 *değeri* abs fonksiyonunun n parametresine atanır. Çağırılan fonksiyon, ya bir **return** deyiminin yerine getirilmesi ya da fonksiyon gövdesinin sağ çengelli parantezine ulaşılması sonucu bittiğinde, kontrol çağırان fonksiyondaki kaldığı yere geri döner. Fonksiyon tarafından bir değer döndürülüyorsa, bu değer çağırının yerini alır. Örneğin,

```
x = abs(-127);
```

x'in değerini 127 yapacaktır; oysa daha yukarıda aynı fonksiyona yapılan ilk çağrıda geri döndürülen değer (10) kullanılmamıştı, bu yüzden bu değer kaybolmuştu. “Dönüş tipi” **void** olan bir fonksiyonun bir ifade içinde kullanılmaması gerektiği açıktır.

Bir argüman istenildiği kadar karmaşık bir ifade olabilir. Argüman değerinin tipi fonksiyon tanımındaki karşılık gelen parametre için beklenen tipe uymalıdır, aksi takdirde, aşağıda açıklanacağı gibi, bulunması zor olan bazı hatalar meydana çıkabilir.

Dikkat: Fonksiyon-argümanlarının hesaplanma sırası belirtilmemiştir. Örneğin, iki **int** argümanının toplamını veren topla isminde bir fonksiyonumuzun olduğunu varsayalım. Ayrıca, değeri 5 olan, i adında bir değişkenimiz olsun.

```
toplam = topla(i--, i);
```

yazdığımızda toplam'ın değeri ne olacaktır? 9 mu 10 mu? Bu, argümanların hesaplanma sırasına bağlıdır. Değişik derleyiciler değişik şekilde yapabilirler. Bizim sistemimizde, argümanlar sağdan sola hesaplanır, böylece toplam'ın değeri 10 olacaktır. Bu tür şeylerden kaçınılması gerektiğini söylemeye gerek yok tabii.

Fonksiyonlar kullanılmadan önce bildirilmelidirler. Bu kural bazı hataların önlenmesi için titizlikle uygulanmalıdır. C derleyicisi, bildirimi yapılmayan bir fonksiyon kullanımı ile karşılaştığında dönüş tipinin **int** olduğunu varsayar. Örneğin, aşağıdaki gibi dabs adında bir fonksiyon tanımladığımızı

```
double dabs (double n)
{
    return (n >= 0.0) ? n : -n;
}
```

ve onu

```
deneme (void)
{
    double d;
    ...
    d = dabs (-3.14159);
    ...
}
```

şeklinde çağırdığımızı düşünün. Üç olasılık vardır:

1. dabs'ın tanımı aynı kaynak dosyada deneme'nin tanımından *önce* gelir. Bu durumda dabs'ın tanımı etkisini deneme'nin içinde de sürdüreceği için hiç bir sorun çıkmayacaktır. Derleyici, yukarıdaki ifadede dabs tarafından beklenen ve geri döndürülen değerin tipinin **double** olduğunu bilecek ve gerekli tip dönüşümlerini yapacaktır.
2. dabs'ın tanımı deneme'nin tanımından *sonra* gelir. Derleyici dabs'ın bir fonksiyon çağırısı olduğunu tanıyacak, ancak, henüz dönüş tipinin ne olduğunu bilemeyeceği için, **int** döndüren bir fonksiyon olduğunu varsayacaktır. Daha sonra, dabs'ın gerçek tanımıyla karşılaştığında, bir sorun ortaya çıkacaktır, çünkü **double** döndüren bir fonksiyon olarak tekrar tanımlamaya çalışacaktır. Bu durumda bir hata mesajı verilecektir.
3. En kötü durum, dabs'ın tanımı ayrı bir kaynak dosyada verildiği zaman ortaya çıkacaktır. Ayrı ayrı derlemeden dolayı, derleyici uyumsuzluğu bulamayacaktır. Hatta bağlayıcı bile birşey yapamayacaktır. Programın yürütülmesi esnasında anlamsız sonuçlar elde edilecektir. Bunun nedeni, yine, derleyicinin tanımı veya bildirimi yapılmamış olan fonksiyonun **int** döndürdüğünü varsayması, böylece de dabs fonksiyonu tarafından döndürülen **double** değerin deneme tarafından **int** gibi yorumlanmasıdır. Birçok derleyici yapılan bu tip varsayımları, bir uyarı mesajı şeklinde kullanıcıya bildirirler. Bu tip uyarıları mutlaka dikkate alın, çünkü varsayımlar her zaman sizin düşündüklerinizle uyuşmayabilir.

Yukarıdaki 2 ve 3'teki sorunları çözmek için aşağıdaki şekil önerilmektedir:

```
deneme (void)
{
    double d, dabs(double);
    ...
    d = dabs (-3.14159);
    ...
}
```

Yukarıdaki *bildirim* (buna ayrıca *fonksiyon prototipi* de denir) deneme fonksiyonun tanımının dışında ve önünde de yazılabilir. Böyle fonksiyon prototiplerinin tamamını programın başlangıcında veya böyle fonksiyonları kullanan her kaynak dosya tarafından `#include` kullanılarak içerilecek başlık dosyalarına yazmak sıkça kullanılan bir yöntemdir. Standart kütüphane fonksiyonlarının bildirimlerini yapan standart başlık dosyaları buna iyi bir örnek oluştururlar. Bunlar, derleyicilerin çeşitli hataları yakalamalarını veya gerektiği zaman tip dönüşümleri yapmalarını sağlar. Örneğin, yukarıdaki bildirimden sonra, derleyici,

```
d = dabs(5);
```

ifadesindeki 5 tamsayısını `dabs`'a geçirmeden önce **double**'a dönüştürecektir. Ancak

```
d = dabs("bes");
```

için bir hata mesajı verecektir, çünkü bu durumdaki karakter göstergesi **double**'a dönüştürülemez.

Argümanlar çağıran fonksiyondan çağrılan fonksiyona geçirildiğinde, “doğal” tiplerine (örneğin **short**'tan **int**'e) dönüştürülür. Eğer fonksiyon kısa bir tip bekliorsa, bu argümanın kısa tipe dönüştürülmesi gerekecektir. Böylece, eğer özel bir neden yoksa, fonksiyon parametrelerinin bu doğal tiplerden olması daha iyi olacaktır. Genelde, dönüş tipleri için de aynı şey sözkonusudur.

4.2.1. Değer İle Çağrı

FORTRAN'da ve bazı diğer dillerde argümanlar *referansla* geçirilir. Yani, fonksiyona argümanın *değeri* yerine *adres*i verilir. Bu yolla, çağrılan fonksiyon çağıran fonksiyon tarafından argümanın saklandığı bölgeye erişir ve değerini değiştirebilir. Böylece, bütün argümanlar hem fonksiyona bilgi iletirler, hem de fonksiyondan bilgi geri getirirler, yani, istemesek bile, fonksiyona hem girdi hem de çıktı için kullanılırlar. Eğer gerçek argüman, bir değişken değil de bir ifade veya değişmez ise ne olur? Bu durumda, çağırılan fonksiyon ifadeyi hesaplar, değerini bir yerde saklar, sonra da bu adresi geçirir. Eğer çağrılan fonksiyon, mantıksal olarak yapmaması gerektiği halde, argümanının değerini değiştirirse, bazı garip şeyler ortaya çıkabilir. Bir yordamın **F** FORTRAN fonksiyonunu bir değişmez olan 4 argümanı ile çağırdığını (yani “**F(4)**”), ancak **F** fonksiyonu içinde argümanın değerinin 5 yapıldığını varsayın. Bundan sonra, program içindeki 4 “değişmez”inin her kullanıldığı yerde (örneğin “**PRINT *,4)**” 4'ün 5 olduğu “gerçeği” ortaya çıkarılacaktır. Diğer bir deyişle, yukarıdaki deyimden 5 elde edilecektir!

Yukarıdaki tartışma, argüman geçirilmesi için başka bir yöntemin gerekli olduğunu ortaya çıkarmaktadır. Bu *değer ile çağrı*dır ve C tarafından desteklenmektedir. Bu durumda, argümanın *adres*i yerine *değeri* fonksiyona geçirilir. Bu değer, özel bir bölgede, tıpkı normal bir yerel değişken gibi saklanır. Biçimsel argümana yapılan her türlü değişiklik yerel kopyasında yapılacaktır ve çağırılan fonksiyondaki argümanlara herhangi bir etkisi olmayacaktır.

Bazı diller argüman geçişinin her iki yöntemine de izin verirler. Örneğin, Pascal'da, önüne **var** anahtar sözcüğü yazılarak tanımlanan parametreler için referansla çağrı yapılır, **var** yoksa değerle çağrı yapılır. Normalde C değerle çağrıyı desteklediği halde, daha ileride göreceğimiz gibi, mantıklı bir adres ve gösterge kaynaşması ile referansla çağrı yapılabilir. İlk önce değerle çağrıya bir örnek verelim:

```
fakt (int n)
{
    int i = n;
    while (--n)
        i *= n;
    return i;
}
```

Burada, önce *i*'ye *n* atanır, daha sonra *n* sıfır oluncaya kadar aşağı doğru sayılır, her sefer *i* yeni *n* ile çarpılır. Gördüğümüz gibi *n*'nin değeri değişir, ancak çağırın fonksiyon tarafından geçirilen gerçek argüman üzerinde bunun bir etkisi yoktur. Örneğin,

```
sayi = 4;
printf("%d! = %d\n", sayi, fakt(sayi));
```

deyimlerinden sonra *sayi*'nin değeri (0 değil) 4 olmaya devam edecektir. *fakt*'taki *n* parametresinin *i* gibi yerel bir değişken olduğuna dikkat edin.

4.2.2. Referans İle Çağrı

Çağrıyı yapana bilgi döndürmek istediğimizde ne yapmamız gerekir? Bir yöntem fonksiyonun dönüş değerini kullanmaktır. Şimdiye kadar olan örneklerimizde bu çok sık kullanılmıştı. Fakat, birden fazla döndürülecek değer varsa o zaman ne yapacağız? Bu durumda, referansla argüman geçirmenin bir yolunu bulmamız gerekecektir. Aslında bunu nasıl yapacağımızı da biliyoruz. Daha önceki bölümlerde programa değer girmek için *scanf* fonksiyonunu kullanmıştık. Bu fonksiyon argümanlarına yeni değerler atar. Bu fonksiyonu ilgilendirmedikçe göre, argümanların eski değerlerini geçirmek yerine, *adreslerini* geçiririz. Bu amaçla da, *adres alma* (&) işlecini kullanırız.

Şimdi *referans ile çağrı* yöntemini göstermek için bir örnek vermenin zamanıdır. İki **int** argümanının değerlerini değiş tokuş eden bir fonksiyon yazmak istediğimizi varsayın.

```
void degis (int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
```

fonksiyonu bu işi yapacaktır, ancak sadece yerel olarak! Parametrelerin yeni değerleri çağırın fonksiyona geri iletilmeyecektir. Değerleri geçirmek yerine, yukarıda anlatıldığı

gibi değişkenlerin adreslerini geçirmemiz gerekir. Böylece, a ve b'nin **int** değişken olduğu

```
degis(&a, &b)
```

şeklinde bir çağrı gerekecektir. Ancak *degis* fonksiyonu da bu göstergeleri kabul edecek şekilde yeniden düzenlenmelidir. Herşeyden önce, parametre bildirimleri değiştirilmelidir; yeni parametre değişkenleri *değerler* yerine *adresleri* saklamalıdır. Bildirim şöyledir:

```
int *xg, *yg;
```

ve, örneğin, *xg* değişkeninin bir **int** göstergesi olduğu anlamına gelir. Şimdi bütün *x* ve *y*'leri sıra ile **xg* ve **yg* ile değiştirirsek *degis*'in doğru tanımını elde ederiz:

```
void degis (int *xg, int *yg)
{
    int t;
    t = *xg;
    *xg = *yg;
    *yg = t;
}
```

Unutulmaması gereken bir nokta, dizi argümanlarının her zaman referans ile geçirildiğidir. Örneğin, argüman olarak verilen diziyi sıraya sokan *sirala* adında bir fonksiyonumuz olduğunu varsayalım. Fonsiyonun bildirimi şöyle bir şey olacaktır:

```
void sirala (int a[], int n) /* n dizinin uzunludur */
{
    siralama işlemi
}
```

Bu durumda,

```
sirala(dizi, eleman_sayisi);
```

şeklinde bir çağrı, dizinin taban adresini fonksiyona geçirecektir. *sirala*'daki *a* geçirilen dizinin yerel bir kopyası değildir. *a* “dizisi”nin uzunluğunun belirtilmemesinden, dizi için bir yer ayrılmadığını anlayabilirsiniz. *sirala* fonksiyonu içinde *a*'ya yapılan bütün referanslar aslında *dizi*'ye yapılmaktadır, bundan dolayı değişiklikler de orada yapılmaktadır. Bunun nedeni *dizi* ile *&dizi[0]*'ın tamamen aynı şey olmasıdır. Bu iyi bir özelliktir, çünkü aksi takdirde bütün dizinin yerel bir diziye aktarılması gerekecekti; bu da hem bilgisayar zamanı hem de bellek harcayacaktı.

Aslında, *parametre bildirimlerinde* “**int** *a*[]” ile “**int** **a*” aynı şeydir, yani *a* bir göstergedir. Örneğin, bir dizinin elemanlarını toplayan bir fonksiyonumuz var diyelim. Bunu şu şekilde tanımlayabiliriz:

```

topla (int a[], int n)
{
    int s = 0;
    while (n--)
        s += a[n];
    return s;
}

```

ve dizi'nin bir tamsayı dizisi, uzunluk'un da toplanacak elemanların sayısı olduğu

```
topla(dizi, uzunluk)
```

şeklinde bir çağrı yapabiliriz. dizi isminin bir gösterge olduğu, bundan dolayı topla'nın a'da elde ettiği şeyin bir gösterge olduğuna dikkat edin. a'nın "**int *a**" şeklinde bildirilmesi daha iyi olacaktı. Aslında derleyici de aynı şeyi düşünmekte ve fonksiyon parametreleri için iki bildirimi tamamen eşdeğer kabul etmektedir. Bunu görmek için aşağıdakini karşılaştırm:

```

topla (int a[], int n)
{
    int s = 0;
    while (n--)
        s += *a++;
    return s;
}

```

Atama deyiminde, `*a++`, `*(a++)` anlamına gelir, bu da `(*a)++`'dan tamamen farklıdır, çünkü bu sonuncusu dizinin ilk elemanını sürekli olarak artırmaya yarar, yani `a[0]++`. topla'nın bu yeni şeklinin ilk verilenle aynı işi yaptığını, ancak daha verimli olduğunu gösterin.

Ayrıca, topla'ya dizinin taban adresi yerine başka bir adresin de verilebileceğini akıldan çıkarmayın. Örneğin

```
topla(&dizi[2], uzunluk-2)
```

ve

```
topla(dizi+2, uzunluk-2)
```

topla'nın dizinin üçüncü elemanından başlayarak, yani ilk iki elemanı atlayarak, işlemi yapmasını sağlayan iki eşdeğer çağrıdır.

Bir fonksiyon parametresinin bildirimi yapılırken **const** tip niteleyicisi de kullanılabilir. Bir dizinin fonksiyon gövdesi içinde değiştirilmeyeceğini göstermesi açısından, dizi parametrelerinin bildirimlerinde yararlanılabilir.

4.2.3. main Fonksiyonunun Parametreleri

main de bir fonksiyondur, fakat ilk bakışta sanki herhangi biri onu çağırıyor gibi gözükmemektedir. Bu doğru değildir, çünkü birisi onu gerçekten çağırılmaktadır. İşletim

sistemi, programı çalıştırmak için onu yükledikten sonra, bir başlatma yordamı kontrolü eline alır; bu da sonuçta main adında bir fonksiyonu çağırır—bundan dolayı bu isimde bir tane fonksiyon bulunması gerekir—ve ona üç tane argüman geçirir. En sık kullanılanlar ilk iki argümandır, bunlar da main fonksiyonunun *komut satırı argümanlarına* ulaşmasını sağlar.

main'in şu şekilde tanımlanmış olduğunu varsayın:

```
#include <stdio.h>
void main (int argc, char *argv[])
{
    int i;
    printf("Merhaba, benim ismim %s.\n", argv[0]);
    printf("Argumanlarım sunlar");
    for (i=1; i<argc; i++)
        printf(", %s", argv[i]);
    printf(".\n");
}
```

Program derlenip bağlandıktan sonra işletilebilir bir dosya ortaya çıkmış olacaktır. Bu dosyayı, isminin arkasına, isteğe bağlı olarak verilen, birkaç argüman yazarak çalıştırabilirsiniz. Bizim sistemimizde DNM.EXE adlı bir dosya oluştuğunu varsayalım. Aşağıdaki komut kullanılabilir:

A>DNM BİR İKİ ÜÇ

Çıktı ise şöyle olacaktır:

```
Merhaba, benim ismim A:\DNM.EXE.
Argumanlarım sunlar, BİR, İKİ, ÜÇ.
```

main'in ilk parametresi (argc) bir sayıdır ve programı çalıştırmak için verilen komut satırındaki toplam isim sayısına eşittir. Yukarıdaki denemede 4 idi: Bir tanesi .EXE dosya adı için, 3 tanesi de onun argümanları için. main'in ikinci parametresi (argv) **char**'a göstergelerden oluşan bir dizidir. Yine, main onun için yer ayırmadığından dolayı uzunluğu belirtilmemiştir. "**char ***" bir *karakter dizisini* tanımlar. Bu tipten olan bir değişken, printf'in kontrol karakter dizisindeki %s dönüşüm tanımlaması kullanılarak basılabilir. argv bir karakter dizileri dizisidir ve argv[0], argv[1] elemanları, karakter dizilerini gösterir. Böylece yukarıdaki program argv'nin elemanlarını karakter dizileri olarak basar.

Şekil 4.1'deki şema başlatma yordamının main için hazırladığı belleğin durumunu gösterir.

4.3.1. auto Değişkenler

Şimdiye kadar karşılaşmış olduğumuz bütün değişkenlerin bellek sınıfı otomatiktir. Öğrenmiş olduğunuz gibi, otomatik değişkenler, içinde bulundukları blok yürütüldüğü sürece “yaşarlar” yani bloktan önce veya sonra mevcut olmazlar. Değişkenlerin otomatik bellek sınıfından olduğunu belirtmek için tanımlarından önce **auto** anahtar sözcüğünü yazabilirsiniz. Örneğin,

```
auto int i;  
auto char kar, x;
```

(Genelde, bellek sınıfı belirten bütün anahtar sözcükler bu şekilde kullanılırlar; tanım veya bildirimin önüne yazılırlar.) **auto** kullanmanın aslında gerçek bir anlamı yoktur, çünkü bloklar içinde tanımlanmış değişkenler için bellek sınıfı belirtilmediği zaman, otomatik oldukları varsayılır. Yani, C’nin bile gereksiz bir anahtar sözcüğü vardır.

Doğal olarak küresel değişkenler için **auto** kullanılamaz. Ayrıca, fonksiyonlar da otomatik olamaz. Fonksiyon parametrelerinin de otomatik olduğu varsayılır ve değerleri çağırın fonksiyon tarafından ilklenir.

4.3.2. register Değişkenler

register bellek sınıfı **auto** bellek sınıfı ile yakından ilgilidir. Bu iki sınıf için etki alanı ve yaşam süresi kuralları aynıdır, ve **register**, tıpkı **auto** gibi, fonksiyonlar dışında anlamsızdır. Eğer bir değişken **register** sınıfından tanımlanmışsa, derleyici onu makinenin hızlı bellek yazmaçlarına yerleştirmeye çalışır. Bu tür yazmaçlardan sınırlı sayıda olduğu için, sadece ilk birkaç **register** değişkeni gerçekten yazmaçlarda saklanır, diğerleri otomatik değişkenler gibi işlem görürler. Yani, bir **register** bildirimi derleyiciye sadece bir öneri özelliği taşır. Böylece, eğer bir önceki alt kısımda tanımlanmış değişkenlere gerçekten hızlı erişmeyi istiyorsak

```
register int i;  
register char kar, x;
```

yazarız.

Ana bellekte depolanmayabilecekleri için, **register** değişkenlerine tekli & işleci uygulanamaz. Ayrıca, diziler gibi “karmaşık” tipten bazı veriler **register** olamazlar; ancak bu sistemden sisteme geçişebilir.

register bellek sınıfından en iyi şekilde yararlanmak için, sadece birkaç değişkeni bu tipten tanımlamalısınız. Bunlar en çok kullanılanlar olmalıdır. (Örneğin döngü sayacıları.) Mümkün olduğu kadar kullanıldıkları program koduna yakın tanımlanabilmeleri için de blok içine alın. Fonksiyon parametreleri de **register** bellek sınıfından tanımlanabilirler. Bu bölümün sonundaki örneğe bakınız.

4.3.3. static Değişkenler Ve Fonksiyonlar

Kullanıldığı yere bağlı olarak **static** anahtar sözcüğünün iki değişik anlamı vardır. Blok içinde, **static** “kalıcı” anlamına gelir. Yani, blok içinde değişkenlerin tanımlarının önüne **static** anahtar sözcüğü kullanılırsa, bu değişkenlerin değerleri blok sona erdiğinde yok olmazlar. Blok dışına çıkıldığında değişkenler erişilmez hale gelir, fakat blok tekrar işletilirse, blok sona erdiği zamanki değerleriyle programın yürütülmesine katılırlar. Diğer bir deyişle, bu bağlamda **static auto**’nun tersidir. Fonksiyon parametreleri olarak kullanılan değişkenler **static** olamaz. Örnek:

```
void fark (void)
{
    static int  deger;
    int        yeni_deger;

    scanf("%d", &yeni_deger);
    printf("Bu deger son seferden %d birim farklidir.\n",
        deger - yeni_deger);
    deger = yeni_deger;
}
```

Bu fonksiyon girilen değerle daha önce en son girilen değer arasındaki farkı yazar. İlk çağrıda eski değerın sıfır olduğunu varsayar. (Bütün **static** değişkenlerin sıfıra ilklendiği varsayılır.) `deger` adı verilen değişken, kontrol başka fonksiyonlara dönüp tekrar buraya geldiğinde değerini sürdürür.

Bir fonksiyon veya küresel bir değişken tanımının önüne yazıldığı zaman, **static** “gizli” anlamına gelir. Yani, bu fonksiyon veya değişken (ayrıca derlenmiş yada derlenecek) başka dosyalara tamamen yabancı olacaktır; diğer bir deyişle, bu dosyalar **static** değişken veya fonksiyona erişemeyeceklerdir. Bu kısmın başındaki örnekte, `z` ve `funk_2`’yi **static** yapalım:

```
fonk_1 (...)
{  fonk_1'in gövdesi }

static double z;

static fonk_2 (...)
{  fonk_2'in gövdesi }

main (...)
{  main'in gövdesi }
```

Şimdi, eğer isterlerse, bu dosyadaki fonksiyonlar `z` ve `funk_2`’ye erişebilirler, ancak başka kaynak dosyalardaki fonksiyonlar için bu olanaksız olacaktır.

4.3.4. Fonksiyonlar Ve extern Değişkenler

Daha önce de bahsettiğimiz gibi, küresel bir değişkenin etki alanı normalde tanımlanmış olduğu kaynak dosyanın sonuna kadar olan kısımdır. Ancak, başka dosyalarda veya aynı dosyada fakat daha ileride tanımlanmış bulunan küresel değişkenleri kullanmanın bir yolu vardır: Anlamlı bir yerde **extern** anahtar sözcüğünü ve arkasına değişkenin bildirimini yazmanız gerekir. Bundan sonra, sanki orada değişkeni tanımlamışsınız gibi onu istediğiniz gibi kullanabilirsiniz. Ancak, diğer dosyalardan değişkenlere bu tür bir erişim sağlamak için, değişkenin özgün tanımının **static** olmaması gerekir.

Yukarıda anlatılan yöntemin kullanıldığı bir dosya topluluğunda **static** olmayan her küresel değişkenin tam bir tane **extern** olmayan tanımı ve birtakım **extern** bildirimleri vardır. **extern** kullanılmamış (“özgün”) tanım, değişken için bellekten bir bölgenin ayrıldığı tek yerdir. Diğer bildirimler ise, bu değişkenin tanımının başka bir yerde bulunduğunu, tipinin ise belirtildiği gibi olduğunu derleyiciye anlatmak içindir. Diğer bir deyişle, **extern** bildirimi değişken için yer ayrılmasına neden olmaz.

Doğal olarak, bir değişkenin özgün tanımıyla **extern** bildirimlerinde farklı tiplerin belirtilmesi istenmeyen şeylerin oluşmasına neden olabilir, onun için dikkatli davranmak gerekir. Eğer hem tanım hem de bildirim aynı dosyada ise derleyici hatayı bulabilecektir, fakat eğer farklı dosyalarda ise hata bulunamayacaktır.

Fonksiyonlar kendi dosyalarında daha yukarıda kalan bölümler veya (eğer **static** değillerse) başka dosyalar tarafından kullanılabilirler. Böyle bir durumda, kullanılmadan önce, kullanıldığı bloğun (içinde veya) dışında fonksiyonun bildirimi yapılmalıdır. Böyle bir bildirimde, fonksiyonun dönüş tipi, adı, parantez içine alınarak virgülle ayrılmış parametre tipleri ve noktalı virgül bulunur. Fonksiyonun bloğu yazılmaz. Örnek:

```
#include <stdio.h>

double cikar(double,double);
/* Bu fonksiyon daha ileride bir */
/* yerde tanımlanmaktadır.      */

void main (void)
{
    double a;
    a = 3.2;
    printf("%f\n", cikar(a,2));
}

double cikar (double x, double y)
{
    return x-y;
}
```

Yukarıdaki programda yazılan “**double cikar(double,double);**” bildirimi, başka bir dosyada veya bu dosyanın daha ileriki bölümlerinde iki **double** argüman kabul eden ve dönüş tipi **double** olan **cikar** isminde bir fonksiyonun bulunduğunu ve bu fonksiyon kullanılırken bu bilginin dikkate alınması gerektiğini derleyiciye anlatır. Eğer

bu satır yazılmamış olsaydı, derleyici `cikar`'ın gerçek tanımına ulaştığında biri **int** biri de **double** olan *iki* `cikar` isminde fonksiyon bulunduğunu düşünecek, bu da büyük bir hata olacaktı. Ayrıca, bu bilgiyi kullanarak, derleyici 2 tamsayı argümanını fonksiyona geçirirken **double**'a dönüştürür.

4.3.5. İlkleme

Bu altkısmı başlamadan önce, gelin bellek sınıflarını bir daha gözden geçirelim: *Otomatik* sınıfından olan değişkenler ait oldukları blokla beraber ortaya çıkarlar ve sonra yok olurlar. *Yazmaç* sınıfı değişkenler otomatikler gibidir, sadece yazmaçlara yerleştirilebilirler. *Dural (static) yerel* değişkenler değerlerini asla yitirmezler, oysa aynı sınıftan olan küresel değişkenler ve fonksiyonlar kendi dosyalarında gizli kalırlar. **static olmayan küresel** değişkenler ve fonksiyonlar dördüncü ve son bellek sınıfını oluştururlar, bunlara *dışsal* adı verilir ve programın herhangi bir yerinden kullanılabilirler.

Programcı tarafından, aksi belirtilmediği sürece *dışsal* ve **static** bellek sınıfından olan değişkenler sıfıra ilklenir. **auto** ve **register** sınıfından olan değişkenlerin ilk değerleri belirsizdir.

Bildiğiniz gibi, tanım esnasında bir ilkleme yapmak için, değişkenin arkasına = işaretini koyup (çengelli parantez içine alarak veya almayarak) bir ifade yazarız. Böyle ifadeler için kurallar bellek sınıfına bağlı olarak değişir.

Dışsal veya **static** değişkenler durumunda, bu bir *değişmez ifade* olmalıdır ve şimdiye kadar gördüğümüz *değişmez ifadelerden* farklı olarak (adreslerini belirtmek için) dizi ve fonksiyon isimleri ve *dışsal* ile **static** değişkenler veya dizi elemanlarına uygulanan tekli & işleciyle oluşturulmuş ifadeler de içerebilir. Bütün bunların anlamı, böyle bir ifadenin değerinin (*a*) bir adres artı veya eksi bir *değişmez* veya (*b*) sadece bir *değişmez* olabileceğidir. Örnek:

```
float t = (900.9-888.1)*1.1;
```

Eğer ilklenecek değişken **auto** veya **register** ise ilkleme ifadesi daha önce tanımlanmış sözcükler içerebilen her çeşit geçerli C ifadesi olabilir. Örnek:

```
{
    double v = x/a()%77;           /* x ve a bu blok */
                                   /* icinde bilinmektedir. */
    bloğun geri kalanı
}
```

Gelin bu kuralların arkasındaki nedeni araştıralım:

Dışsal ve **static** değişkenler derleme sırasında hesaplanabilecek ilkleme ifadeleri isterler, çünkü bu esnada ilklenmektedirler. Program çalışmaya başlamadan, bu değişkenler ilk değerlerini almış olurlar. Bunun için, tanımında bir ilkleme olan yerel bir **static** değişken bloğa her girişte değil de sadece bir defa ilklenir (bu da doğru bir şeydir, çünkü **static** bir değişken böyle davranmalıdır).

Diğer taraftan, derleyici, bir **auto** veya **register** değişkeni için bir ilkleyen gördüğünde, bunu o bloğun tanım listesinden hemen sonra gelen ve ilgili değişkenlere atama yapan bir deyim gibi görür. Bu atamalar, yürütme sırasında, bloğun başına ulaşıldığı her sefer tekrarlanır. Bundan dolayı, **auto** ve **register** değişkenleri için ilkeme ifadeleri, ilgili değişken bir sol işlenen olarak bir atama deyiminde kullanıldığı zaman sağ tarafta gelebilecek her tür geçerli ifade olabilir. Bu da, diğer dillerde rastlanması zor olan, C'nin güzel özelliklerinden biridir.

Artık, **static** olmayan değişkenlerin bir **switch** bloğunda ilklenmelerinin neden anlamsız olduğu anlaşılıyor: Her durumda, kontrol bu ilklemelerin üstünden geçer.

İklenecek değişken bir dizi ise o zaman ne olur? Artan indis sırasına göre çengelli parantezler içine ve virgülle ayırarak her dizi üyesinin değerini belirten ifadeler yazılır. Bu listede, dizinin boyundan daha fazla sayıda ifadenin bulunması hata olur. Fakat eğer daha az sayıda ifade varsa, kalan dizi elemanları sıfıra iklenir. Örneğin:

```
int g[5] = { 1, -2, 0, 3 };
```

dizinin ilk dört elemanına belirtilen değerleri atar, beşinciye de sıfır yerleştirir.

Dizi iklenmesinde bazı kestirmeler söz konusu olabilir. Eğer dizinin tanımlanması esnasında boyunu belirtmezseniz, derleyici (mutlaka var olması gereken) ilkeme listesine bakar ve listedeki eleman sayısına eşit boyda bir dizi yaratır. Örneğin;

```
float f[] = { 2.2, 0.3, 99.9, 1.1 };
```

dört elemanlı bir dizinin ayrılmasını ve yukarıdaki değerlerle iklenmesini sağlar.

Karakter dizileri liste yerine değişmez bir karakter dizisi yazarak da iklenebilirler:

```
char gercek[] = "C iyidir.";
```

ile

```
char gercek[] = {
    'C', ' ', 'i', 'y', 'i', 'd', 'i', 'r', '.', '\0'
};
```

aynı anlama gelirler. İlk durumda belirtilmemiş olmasına rağmen derleyici tarafından otomatik olarak eklenen boş karaktere dikkat edin. Eğer böyle bir durumda dizinin uzunluğu belirtilmiş olsa ve ilkleyen karakter dizisinin uzunluğuna (yukarıdaki örnekte 9'a) eşit olsaydı, bitirici boş karakter diziye dahil edilmeyecekti.

Göstergeleri de ilkleyebilirsiniz. Otomatik veya **register** göstergeler için ilkleyen herhangi bir geçerli gösterge ifadesi olabilir. **static** veya dışsal göstergeler için değişmez bir ifade olmalıdır. Aşağıdaki örnekte,

```
#define N 10
test (int i)
{
    int a[N], *ag = a+i;
    static b[N], *bg = &b[N],
        *bg1 = &b[0]+i, /* hata */
        *bg2 = a;      /* hata */
    ...
}
```

`ag` ve `bg` için ilklemeler geçerlidir. Fakat `bg1` için ilkleme geçersizdir, çünkü `i` bir değişmez değildir. Ayrıca `bg2`'de de bir sorun vardır: `a` otomatik bir dizidir; temel adresi yürütme esnasında belirlenir, oysa (`ag`'den farklı olarak) `bg2`'nin ilklemesi derleme esnasında yapılır, ancak bu olanaksızdır.

Dizilerle göstergeler arasında ilgi çekici bir karşılaştırma aşağıda görülebilir:

```
char const msj1[] = "Sonraki lutfen? ";
char const *msj2 = "Sonraki lutfen? ";
```

Hem `msj1` hem de `msj2` karakter göstergeleri olarak değerlendirilebilirler ve birçok bağlamda birbirinin yerine kullanılabilirler. Fakat derleyici açısından bir fark söz konusudur. İlk tanımda, dizi için 17 bayt kullanılır, bu da `msj1` dizisi için ayrılan bellek miktarıdır. İkincisinde ise, dizinin derleyici tarafından, başka bir yerde saklanması (yine 17 bayt) ve bunun başlangıç adresinin `msj2` için ayrılan yere (bizim sistemde 2 bayt) depolanması sağlanır. Yani ikinci tanım daha fazla yer kaplar. Bu *değişken* bir gösterge için ödememiz gereken bedeldir; oysa programın yürütülmesi esnasında belki de bu göstergenin değerini değiştirmeyeceğiz. Bu tür program değişmezleri için ilk seçeneği (yani *değişmez* bir göstergeyi) kullanmamız daha akıllıca olacaktır.

4.4. Özçağrı

Şimdiye kadar başka fonksiyonlar çağıran fonksiyonlara örnekler vermiş bulunuyoruz. Peki, kendini çağıran bir fonksiyona ne dersiniz? Daha önce böyle bir şey görmeyenler için, bu oldukça garip gözükebilir. Herşeyden önce, bir fonksiyon niye kendisini çağırmak zorunda kalsın? İkinci olarak, sonsuz böyle özçağrı sonucu ortaya çıkacak bir döngüden nasıl kaçınabiliriz? Üçüncü olarak da, aynı fonksiyondan birden fazla kopyanın işlek durumda olacağı için, fonksiyonun yerel değişkenlerine ne olur?

Kendini çağıran fonksiyonlara *özçağrılı* adı verilir. Özçağrılı bir fonksiyonun kendini dolaylı veya dolaysız olarak çağırabileceğine dikkat edin. İkinci durumda, örneğin, `f` adında bir fonksiyonun `g` adında başka bir fonksiyonu çağırması, onun da `f`'yi tekrar çağırması söz konusudur. Her durumda, bir fonksiyonun bir önceki etkinleştirilmesi sona ermeden aynı fonksiyonun tekrar çağırılması söz konusudur. Bundan dolayı, özçağrılı fonksiyonlar özel işlem gerektirirler. Özçağrılı fonksiyonları desteklemek için derleyicinin özel bir bellek düzeni kullanması; özçağrılı fonksiyonlar yazmak için ise programcının biraz farklı bir düşünme tarzına sahip olması gerekir.

Yine de, özçağrılı fonksiyonlar o kadar garip değildir; ve bazı durumlarda, eşdeğer özçağrılı olmayan fonksiyon yerine özçağrılı olanı kodlamak daha kolaydır. İşte basit ve klasik bir örnek:

Matematikte, faktöriyel (!) tanımı şöyledir:

$$n! = n \times (n-1) \cdots 2 \times 1.$$

Bu tanım, faktöriyel fonksiyonunun algoritmasını açıkça gösterir:

```
fakt (int n)
{
    int i = 1;
    while (n)
        i *= n--;
    return i;
}
```

Faktöriyel başka bir eşdeğer tanımı da şöyledir:

$$0! = 1$$

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

Bu bir *özçağrılı* tanımdır. Özçağrılı bir fonksiyonun tanımının temel unsurları şunlardır:

1. Fonksiyonun, bazı argümanlar için, değerini veren bir *temel* (veya “aksiyomlar” veya “sınır koşulları”). Yukarıdaki tanımda ilk deyim buna bir örnektir.
2. Bilinen değerlerden fonksiyonun başka değerlerinin nasıl elde edileceğini gösteren *özçağrılı yapı kuralı*. Bu da yukarıdaki örneğin ikinci deyiminde gösterilmektedir.

Özçağrılı tanımın, özçağrılı yapı kuralının temel fonksiyon değerleri üzerinde *sınırlı* sayıda uygulama sonucu sona eren bir yöntem tarif ettiğine dikkat etmemiz gerekir. Her $n \geq 0$ için yukarıdaki tanımın doğru olduğu gösterilebilir. Örneğin,

$$3! = 3 \times 2! = 3 \times (2 \times 1!) = 3 \times (2 \times (1 \times 0!)) = 3 \times (2 \times (1 \times 1)) = 6$$

Bu kadar matematik yeter. Şimdi de bilgisayarlı gerçek yaşama dönelim. Faktöriyel fonksiyonunun özçağrılı uyarlaması şöyledir:

```
fakt (int n)
{
    return (n==0) ? 1 : n*fakt(n-1);
}
```

Bu da tamamen matematik dilinden bilgisayar diline bir çeviridir. Koşullu işlece dikkat edin. Argümanın, temel deyimin gereğine uyup uymadığı, yani sıfır olup olmadığı, kontrol edilir. Eğer öyle ise, temel fonksiyon değeri, yani 1, döndürülür. Aksi takdirde, özçağrılı yapı kuralı uygulanır. Bu kural (3! örneğinde olduğu gibi) sınırlı sayıda tekrarlanarak doğru değer hesaplanır. Böylece bu kısmın başında sorulan ikinci soruyu yanıtlamış bulunuyoruz: Temel deyim sonsuz sayıda özçağrıya engel olur.

İlk soruya gelince: Bu bir zevk ve verimlilik sorunudur. Bazı programcılar `fakt`'ın ilk tanımını, bazıları ise ikinci tanımını beğenebilir. Ancak, verimlilik programcılar için

önemli (ve nesnel) bir parametredir. Göreceğimiz gibi, özçağrılı tanımlar özçağrılı olmayan (yani yinelemeli olan) tanımlardan daha az verimlidir. Buna rağmen, gerçek programlarda (örneğin bir derleyicinin kendisinde) özçağrılı fonksiyonlar kullanılır, çünkü özçağrılı bir tanım bazen daha zariftir ve anlaşılıp izlenmesi daha kolay olur.

Son sorumuz özçağrılı fonksiyonların nasıl çalıştığı ile ilgili idi. Aşağıdaki atama deyiminin neyle sonuçlanacağını izleyelim.

```
f2 = fakt(2);
```

(fakt'ın ikinci uyarlamasını kullanmaktayız.) Aşağıda adım adım ne olduğu verilmektedir:

```
fakt 2 argümanıyl a çağrılır.....0
  n = 2; .....1
  fakt n-1 (=1) argümanıyl a çağrılır .....1
    n = 1; .....2
    fakt n-1 (=0) argümanıyl a çağrılır .....2
      n = 0; .....3
      return 1; .....3
    return n * 1; (=1 × 1 = 1) .....2
  return n * 1; (=2 × 1 = 2) .....1
f2 = 2; .....0
```

Sağda yazılan sayılar fakt fonksiyonunun kaç tane işlek kopyasının bulunduğunu gösterir. Üçüncü seferki işleme esnasında n yerel değişkeninin (parametrenin) değeri 0'dır, daha sonra ikinci işlemeye döndüğümüzde, n'nin değeri fonksiyonun o işleme başladığı zamanki değer olur. Bu örnekten, üç n'nin aslında farklı değişken olduğu ortaya çıkar.

Bunu sağlamak için, derleyici programa bir yığıt kullanır. Bütün **auto** ve **register** değişkenleri yığıtta yer alır. Yığıt, bir bayt dizisi şeklinde bitişik bir bellek bölümü ve bununla birlikte bir yığıt göstergesi şeklinde düşünülebilir. Yığıt göstergesi başlangıçta yığıtın başına işaret eder; yığıtın geri kalanı serbest (yani boş) kabul edilir. Özçağrılı olsun, olmasın, herhangi bir fonksiyona giriş yapıldığında, o fonksiyonun (**static** ve dışsal olanlar hariç) yerel değişkenlerinin tamamını tutacak miktarda bayt için yığıtta yer ayrılır; yığıt göstergesi yığıtın geri kalan boşluğunun başını göstereceği şekilde ileriye götürülür. Fonksiyonun bütün ("dinamik") yerel değişkenlerine yapılan referanslar yığıtın ayrılmış bu kesimine yapılır. Fonksiyon bitip geri döndüğünde, yerel değişkenler yok olur, çünkü yığıt göstergesi fonksiyon çağrılmadan önce gösterdiği yeri tekrar gösterecek şekilde geriye kaydırılır. Eğer aynı fonksiyondan iki çağrı yapılırsa, yerel değişkenler için iki defa yer ayrılır; üç defa çağrılırsa, yığıtta yerel değişkenlerin üç kopyası yaratılır vs. Aynı fonksiyona birden fazla çağrı yapılmışsa, yerel değişkenlerin sadece son kopyası erişilir olmaktadır. Fonksiyon geri döndüğünde eski değişkenler etkili olmaya başlarlar vs. Bir fonksiyon çağrısı olduğu esnada yürütmenin sürdürdüğü yer olan *dönüş adresi* de yığıta "itilir". Özçağrılı fonksiyonları desteklemek için bütün bunlar gereklidir. FORTRAN gibi, bazı diller bunu yapmazlar, bundan dolayı bu dillerde

özçağrılı fonksiyonlara izin verilmez. Diğer taraftan, C ve Pascal gibi bazı diller özçağrılı destekler.

Özçağrılı fonksiyonlar yığıttan çok yer harcarlar. Bir **int** değişkenin iki bayt, dönüş adresinin de dört bayt kapladığını varsayalım. Bu durumda, "**fakt(i)**" şeklinde bir çağrı yığıtta $6 \times (i+1)$ bayt kullanacaktır. Örneğin, "**fakt(6)**" için 42 bayt gereksinimimiz olacaktır. Diğer taraftan, **fakt**'ın yinelemeli (yani ilk verilen) uyarlaması yığıttan 4 (dönüş adresi için) + 2 (n için) + 2 (i için) = 8 bayt kullanacaktır. Özçağrılı fonksiyonun bellek üzerinde daha fazla talepte bulunduğu açıktır. Ayrıca, yerel değişkenlerin ve dönüş adresinin yığıta itilip geri alınması işlemci zamanı da harcar.

Yinelemenin özçağrıdan daha verimli olduğu sonucunu elde edebiliriz. Verimlilikteki bu farkın o kadar fazla olmadığı gerçek örnekler de verebiliriz, bundan dolayı eşdeğer bir yinelemeli algoritma bulamadığınız zaman özçağrı kullanmaya tereddüt etmeyin.

4.5. Fonksiyonlara Göstergeler

C dilinde, tıpkı bir değişkenin adresi gibi, bir fonksiyonun adresini de bir gösterge değişkeninde depolayabilir veya başka bir fonksiyona argüman olarak geçirebiliriz. Bu kısımda, diğer fonksiyonlara fonksiyon göstergelerini nasıl geçirebileceğimiz konusyla ilgileneceğiz.

Derleyici tarafından, bir tanıtıcı sözcüğün bir fonksiyonu gösterdiği, ya daha önce bir fonksiyon olarak tanımlandığı veya bildirimi yapıldığı için, yada tanıtıcı sözcüğün arkasında sol parantez olduğu için (yani bağlamdan) derleyici tarafından yapılan otomatik bildirim sonucu bilinir. Eğer bu tanıtıcı sözcük daha sonra bir parantezle birlikte kullanılmazsa, derleyici bu fonksiyonu çağırmaya kalkmayacak, onun yerine fonksiyonun adresini kullanacaktır. (C dilinde sık yapılan bir yanlış ta argümanı olmayan bir fonksiyonu parantezleri belirtmeden çağırmaya çalışmaktır. Yani

f () ;

yerine

f ;

şeklinde bir deyim kullanmaktır. İkinci deyim, "**f**" yerine fonksiyonun adresini koyma dışında bir şey yapmayacaktır.)

Bazı değerler ile bu değerler üzerinde uygulanacak *herhangi* bir fonksiyondan geri döndürülen değerlerin bir çizelgesini çıkaracak bir fonksiyon yazmak istediğimizi düşünün. Bu, örneğin, bir *sinüs*, *karekök* veya *doğal logaritma* çizelgesi olabilir:

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

void cizelge (double al, double ul, double art,
             double (*fg) (double), char *fi)
{
    double x;

    printf("\n   x\t%s(x)\n\n", fi);
    for (x = al; x <= ul; x += art)
        printf("%7.2f\t%7.2f\n", x, (*fg) (x));
}
```

Buradaki

```
double (*fg) (double);
```

tanımına dikkat edin. Bunun ne olduğunu anlamak için işleçlerin uygulanış sırasını göz önünde tutmamız gerekir. Bu bir **double** döndürüp argümanı da **double** olan fonksiyona bir göstergedir. İleride, bir ifade içinde “(*fg) (...)” yazdığımızda böyle bir fonksiyona çağrı yapmış oluyoruz. Şimdiye kadar incelediğimiz diğer gösterge değişkenlerinin, örneğin **int** göstergelerinin, hem tanım hem de kullanımındaki benzerliğe dikkat edin. fg fonksiyona bir göstergedir, (*fg) ise fonksiyonun kendisidir. Gösterge ifadesini çevreleyen parantezler gereklidir, çünkü aksi takdirde

```
double *fg(double);
```

ifadesi

```
double *(fg(double));
```

anlamına gelecekti, bu da **double** gösterge döndüren bir fonksiyon demektir ki bu bildirim parametre tanımlarında kullanılamayacak çok farklı bir şeydir. Bunun nedeni fg'den sonra gelen parantezlerin *'dan daha yüksek önceliğe sahip bir işleç olmasıdır.

Aşağıdaki program cizelge fonksiyonuna geçerli çağrılar yapmaktadır:

```
#include <math.h>

void main (void)
{
    double sin(double), sqrt(double), log(double);
    cizelge(0.0, 3.14, 0.1, sin, "sin");
    cizelge(1.0, 100.0, 1.0, sqrt, "k.kok");
    cizelge(1.0, 2.718, 0.05, log, "log");
}
```

Dördüncü argümanın bir değişken değil de bir fonksiyon ismi olduğuna dikkat edin. Fonksiyon tarafımızdan tanımlanmış olabileceği gibi bir *kütüphane fonksiyonu* da olabilir. Bu noktada bu fonksiyona bir çağrı söz konusu değildir, çünkü argümanlar verilmemiştir, hatta parantezler bile yoktur. Derleyici, sin, sqrt ve log'un fonksiyon olduğunu bilir, çünkü yukarıda bildirimleri yapılmıştır; böylece bu fonksiyonların başlangıç adreslerini alıp cizelge fonksiyonuna geçirir. Dizilerde olduğu gibi burada da *adres alma* (&)

işlecini kullanmamıza gerek yoktur. Eğer bir dizi isminin arkasında “[]” varsa, o zaman dizinin belirtilen elemanının değeri kullanılır; aksi takdirde dizinin *temel adresi* kullanılır. Aynı şekilde, bir fonksiyon isminden sonra eğer “()” varsa (çağrı sonucu) fonksiyondan döndürülen değer kullanılır; aksi takdirde ifadenin değeri olarak fonksiyonun *başlangıç adresi* kullanılır.

cizelge fonksiyonu, ona geçirilen adresi kullanarak asıl çağırıcı yapar. cizelge’ye geçirilen bütün fonksiyonların “benzer” olması gerekir, yani aynı sayıda ve tipte argüman kabul edip aynı tipi döndürmelidirler. Parametrenin tanımlanması esnasında bütün bunlar belirtilir. main içinde verilen fonksiyon bildirimleri gerekli değildir, çünkü #include edilen standart başlık dosyası math.h’de bu bildirimler zaten mevcuttur.

4.6. Bir Örnek—8 Vezir Problemi

Bu kısımda, fonksiyonlar, bitset işlemler, küresel değişkenler, **register** bellek sınıfı ve özçağrı gibi kavramları kullanan örnek bir program vereceğiz.

Bu program bize klasikleşmiş bir problemin çözümlerini bulacaktır: 8 çarpy 8’lik bir satranç tahtasına birbirini almayacak şekilde 8 vezirin yerleştirilmesi. İki vezir aynı sıra, sütun veya çaprazda olduğunda birbirini alabilir. Yani bir çözümde aynı sıra, sütun veya çapraz hatta iki vezir bulunmamalıdır. Örneğin,

.	V
.	.	.	V
V
.	.	V
.	V	.	.
.	V
.	V	.
.	.	.	.	V	.	.	.

bir çözümdür. Burada “**V**” bir veziri “.” ise boş bir kareyi gösterir.

Bu problemi çözmeye ilk yaklaşım sistemli bir şekilde tüm yerleştirmeleri inceleyip sadece geçerli olanları listelemektir. Bir bilgisayarımız olduğuna göre, bunu çabuk bir şekilde yapabiliriz. Aslında, o kadar da değil! Olası tüm değişik yerleştirmelerin sayısı

$$\frac{(8 \times 8)!}{(8 \times 8 - 8)! \times 8!} = 4,426,165,368 \text{’dir.}$$

Her yerleştirme için bir milisaniye harcasak bile, hepsini teker teker denemek elli günden daha uzun bir süre olacaktır. Araştırma bölgemizi daraltmamız gerekir. Aşağıdaki gözlemler yararlı olacaktır:

1. Eğer iki veziri yerleştirdiğimizde bunların birbirini aldığını görürsek, diğerlerini de dikkate alan bütün olası yerleştirmeleri denememize gerek yoktur. Bu, bütün vezirleri satranç tahtasına yerleştirdikten sonra değil, bir vezir yerleştirir yerleştirmeyi uyumsuzlukları kontrol etmemiz gerektiği anlamına gelir.
2. Aynı sıraya iki vezir yerleştirmeye çalışmamalıyız. Her veziri bir sıraya “atayarak” bunu gerçekleştirebiliriz. Her vezir sadece kendi sırasındaki 8 kareden birisine yerleştirilecektir.

Bu iki gözlem araştırma bölgemizi çok daraltacaktır. Algoritma şöyledir:

- En üst sıradan başlayarak, her veziri kendi sırasına, en sağdaki sütundan itibaren, yerleştirin.
- Bir vezir yerleştirdiğinizde, daha önce yerleştirilmiş vezirlerle bir uyumsuzluğun olup olmadığını kontrol edin. Eğer varsa, veziri bir sonraki (yani hemen solundaki) kareye kaydırın; eğer yoksa, bir sonraki (yani hemen altındaki) sırayı değerlendirin. Eğer son vezir de başarılı bir şekilde yerleştirilirse, bu bir çözümdür.
- Eğer bir vezir, sırasındaki sekizinci sütunu da geçerse, o zaman veziri satranç tahtasından alın, bir önceki sıraya *geriye dönüş* yapın ve bu sıradaki veziri bir sonraki (yani hemen solundaki) kareye kaydırın.
- İlk sıradaki vezir son sütundaki kareden dışarı çıkıncaya kadar buna devam edin.

Algoritmayı belirledikten sonra, bu sefer işlenecek bilgiyi saklayacak *veri yapısı* için karar vermemiz gerekecektir. Verimlilik en önemli tasamız olmalıdır. Veri yapısı fazla miktarda bilgisayar belleği harcamamalı ve kolay, hızlı erişim sağlamalıdır.

Satranç tahtasını nasıl göstermemiz gerekir? En azından iki seçenek söz konusudur:

1. Her kare bir bitle gösterilebilir. Eğer bit 1 ise bu karede bir vezir var, 0 ise, boş demektir. Böylece, mantıksal olarak 8’e 8’lik bir kare şeklinde düzenlenmiş 64 bite gereksinimimiz vardır.
2. Her sırada en fazla bir vezir olduğuna göre; her elemanında o sıradaki vezir tarafından işgal edilen sütun sayısının saklandığı doğrusal bir dizi tutabiliriz.

Bu örnekte ilk seçeneği kullanacağız. Her sıra 8 bite gereksinim duyar. Birçok bilgisayarda, bir `int` değişken en az 16 bittir. Böylece iki sıra tek bir `int` değişkenin içinde saklanabilir. Ama, her sıra için bir tamsayı ayırıp sadece sağdaki (düşük) bitlerin kullanılması daha kolay olacaktır. Soldaki bitler ise kullanılmayabilir. Ayrıca, programı daha genel bir problem (yani n çarpı n ’lik bir satranç tahtasına n adet vezirin yerleştirilmesi) için tasarlırsak, aynı programı kullanarak, örneğin, 15 vezir problemini de çözebiliriz.

Programda kullanılan veri yapısı (aşağıdaki program listesine bakın) şöyledir:

```
int tahta[8];
```

(Bu arada, `int` **unsigned** olarak `#define` edilmiştir, böylece sağa kaydırma işlemlerinin aritmetik, yani işaret doldurma, değil de mantıksal, yani sıfır doldurma,

olması temin edilmiştir.) tahta ve sayı küresel değişkenler olarak tanımlanmışlardır, böylece bütün fonksiyonlar bunlara erişebilir. tahta[0] satranç tahtasının en üstteki sırasını, tahta[7] ise en alttaki sırayı gösterir. Bir dizi elemanının en düşük (yani sağdaki) biti sıranın en sağdaki sütununu gösterir. Altıncı sıranın en sağdaki sütununa bir vezir koymak istediğimizde

```
tahta[5] = 1;
```

atama deyimini; bu veziri bir sola kaydırmak istediğimizde

```
tahta[5] <= 1;
```

atama deyimini; en soldaki sütundan ileriye bir vezir geçip geçmediğini anlamak için

```
tahta[sira] >= 1<<8
```

testini; iki farklı sıradaki iki vezirin aynı sütunda olup olmadığını anlamak için

```
tahta[sira2] == tahta[sira1]
```

testini; aynı çapraz hatta olup olmadığını anlamak için

```
tahta[sira2] == tahta[sira1]<<sira1-sira2 ||  
tahta[sira2] == tahta[sira1]>>sira1-sira2
```

testini kullanabiliriz. Yukarıdaki testleri kullanarak, tahtaTamam fonksiyonunu tasarlayabiliriz. Bu fonksiyon, sıra'ya yeni bir vezir yerleştirildiğinde tahtanın tamam olup olmadığını kontrol eder. Bunu yapmak için 0'dan sıra'ya kadar (sıra hariç) olan bütün sıralarla sıra arasındaki uyumsuzlukları kontrol eder. Eğer sıra 0 ise o zaman **for** döngüsüne hiç girilmeyecek tahtaTamam fonksiyonu hemen 1 (doğru) döndürecektir. İlk sıra için satranç tahtası her zaman "tamam"dır.

Bir çözüm bulunduğunda çözüm yaz fonksiyonu çağrılır. sayı (çözüm sayısı) değişkeni bir artırılır ve tahta'daki çözüm yazılır. Dış döngü sıralar, iç döngü de sütunlar içindir.

Programın kalbi yerlestir fonksiyonudur. Algoritmadan da tahmin edebileceğiniz gibi, bu özçağrılı bir fonksiyon olarak tanımlanabilir. Bu fonksiyon, başta 0 (ilk sıra) argümanı ile ana programdan çağrılır. **for** döngüsü belirtilen sıra'nın her sütununa veziri yerleştirir. Her yerleştirmeden sonra tahta kontrol edilir; eğer tamamsa, bu sıra için yerleştirme işlemi geçici olarak bekletilir ve bir sonraki sıra denir. Bir sonraki sıra'yı kim deneyecektir? Tabii ki, aynı fonksiyon! Sonsuz özçağrılara engel olmak için fonksiyonun başında bir test yapılır. Eğer yerlestir 8 argümanı ile çağırılmışsa, bu, satranç tahtasına (başarılı bir şekilde) 8 vezirin yerleştirildiği anlamına gelir. Bu durumda yerlestir fonksiyonu çözümü yazdırır ve hemen döner. yerlestir fonksiyonunun her yeni çağrısında sıra adı verilen yeni bir (yerel) değişken yaratılır. Bir önceki çağrıya geri döndüğümüzde bir önceki sıra değişkeni tekrar geri gelir; değeri ise bir eksiği olur.

```

1.  #include <stdio.h>
2.  #define VEZIRLER 8          /* vezir sayisi ve tahta boyu */
3.  #define int unsigned       /* isaretsiz tamsayi kullan */
4.
5.  int sayi = 0;               /* cozum sayisi */
6.  int tahta [VEZIRLER];      /* her eleman bir sirayi gosterir */
7.
8.  int tahtaTamam              /* tahtanın gecerliliginin kontrol et */
9.      (register int siray)
10. {
11.     register int r;
12.
13.     for (r = 0; r < siray; r++) /* onceki tum siralari kontrol et */
14.         if (tahta[siray] == tahta[r] ||
15.             tahta[siray] == tahta[r] << siray-r ||
16.             tahta[siray] == tahta[r] >> siray-r)
17.             return 0;        /* uyusmazlik varsa */
18.     return 1;                /* uyusmazlik yoksa */
19. }
20.
21. void cozumyaz (void)         /* cozumu goster; sayiyi artır */
22. {
23.     register int t, r;
24.
25.     printf("\n\n\tCOZUM %u\n\n", ++sayi);
26.     for (r = 0; r < VEZIRLER; r++) { /* siray */
27.         for (t = 1<<VEZIRLER-1; t > 0; t >>= 1)
28.             printf(" %c", tahta[r] == t ? 'V' : '.');
29.         printf("\n");
30.     }
31. }
32.
33. void yerlestir (int siray) /* bir sonraki siraya yerlestir */
34. {
35.     if (siray == VEZIRLER) /* tum siralar dolu ve kontrol edilmis */
36.         cozumyaz();
37.     else
38.         for (tahta[siray]=1; tahta[siray]<1<<VEZIRLER; tahta[siray]<<=1)
39.             if (tahtaTamam(siray))
40.                 yerlestir(siray+1); /* bir sonraki sirayi dene */
41. }
42.
43. void main (void)
44. {
45.     yerlestir(0);            /* ilk siradan basla */
46.     printf("\n\n%d vezir probleminin %u cozumu vardir.\n",
47.         VEZIRLER, sayi);
48. }

```

Bu örnekte, özçağrı adımı yerine bir döngü ve siray değişkeninin uygun şekilde artırılıp azaltılması kullanılabilir.

Aşağıda VEZIRLER değişmezine değişik değerler verildiğinde yukarıdaki programla elde edilen sonuçları görüyorsunuz:

VEZİRLER

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Cözüm sayısı:

1
0
0
2
10
4
40
92
352
724
2 680
14 200
73 712
365 596
2 279 184

Problemler

1. Altkısım 4.2.1'in sonundaki printf deyiminin çıktısı

4! = 24

şeklinde olur. Bir an için, fakt'ın sayı değişkeninin değerini *gerçekten* değiştirdiğini varsayın; bu durumda da çıktının aynı olacağını söyleyebilir miyiz?

2. C dilinde bir fonksiyon çağrısı esnasında verilen argümanların sayısı ve tipinin bildirimdeki ile aynı olmasına gerek yoktur. Ancak, çok gerekmedikçe bu özelliği kullanmak iyi değildir. printf kütüphane fonksiyonu buna bir örnektir. Burada, sizden (sayısı belli olmayan) argümanların enbüyüğünü döndüren maks adında bir fonksiyon yazmanızı istiyoruz. Listenin sonunu belirtmek için çağrıdaki son argüman her zaman sıfır olacaktır. Örnekler: maks(15, -62, 21, 57, 43, 0) == 57; maks(0) == 0; maks(-10, 0) == -10. *İpucu:* Microsoft C derleyicisinde (ve muhtemelen diğer birçok C derleyicilerinde) argümanlar bellekte arka arkaya olan yerlerde geçirilirler; yani, ikinci argüman ilk argümanın hemen arkasında saklanır vs. Yazacağınız fonksiyonun tanımında tek bir parametre belirtmeniz yeterli olacaktır. Arkadan gelen argümanları *adres alma* ve *dolaylama* işleçlerini kullanarak elde edebilirsiniz. Bu fonksiyonun taşınabilir *olmadığına* dikkat edin.

3. Sistemimizde

```
#include <stdio.h>
void main (void)
{
    { int y = 1993; }
    {
        int i;
        printf("%d\n", i);
    }
}
```

programı

1993

yazar. Bunu açıklayabilir misiniz?

4. Sistemimizde

```
#include <stdio.h>
void main (void)
{
    { int a1=111, a2=222, a3=333, a4=444; }
    {
        register r1, r2, r3, r4;
        printf("%d %d %d %d\n", r1, r2, r3, r4);
    }
}
```

programı

131 3982 333 444

yazar. Bu size **register** değişken tanımlaması konusunda neyi düşündürür? r3 değişkeni bir yazmaçta mı saklanmaktadır?

5. Bölüm 0'ın başında verilen programın ne yaptığını anlatın.
6. Bir önceki bölümde anlatılan ve C kütüphanesinde bulunan strcpy, strcmp, strcat ve strlen fonksiyonlarının işlevsel benzerlerini yazın.
7. 8 vezir programını, yerleştirebilir özçığrılı fonksiyonu içindeki sıra değişkeninin küresel bir değişken olacağı şekilde tekrar düzenleyin. Şimdi program daha az yığıt yeri kullanacaktır.
8. 8 vezir programını özçığrılı olmayana (yani yinelemeli olana) çevirin.

BÖLÜM 5: TÜRETİLMİŞ TIPLER VE VERİ YAPILARI

C dilinin yapısı içinde var olan veri tipleri arasında **int**, **float**, **double**, **char** gibi basit tipler, diziler, göstergeler, sayım tipleri, yapılar ve birlikler bulunmaktadır. Veri tipleri özçağrılı bir şekilde tanımlanabilirler, yani dizilerden oluşan diziler, yapılar içeren yapılar, göstergelere göstergeler vs sözkonusu olabilir. Böylece, gerçekte, tanımlanabilecek veri tipleri sonsuz sayıdadır.

Bu bölümde, yeni veri tiplerini ve var olan veri tiplerinden nasıl yeni tipler tanımlayacağımızı göreceğiz. Şimdiye kadar, göstergeler ve dizileri tanımış olduk, ancak, sayım tipleri, yapılar ve birlikler nedir? Gelin, bunları tek tek inceleyelim.

5.1. Sayım Tipleri

C dilindeki temel veri tipleri arasında basamaklı sayı tipleri vardır (işaretli veya işaretsiz çeşitli boylarda tamsayılar). Bütün basamaklı tipler sınırlı ve doğrusal olarak sıralanmış bir değerler kümesi oluştururlar. C dili tarafından sağlananlar dışında, kullanıcının kendi basamaklı sayı tiplerini tanımlama olanağı vardır, bunlara *sayım* (veya *sayılı*) tipler denir. Böyle bir tipin tanımı, sırasıyla, bütün olası değerleri belirtir. Bunlar tanıtıcı sözcüklerle temsil edilir ve yeni tipin değişmezleri olurlar.

Sayım tiplerini tanımlamanın genel sözdizimi aşağıda verilmiştir:

```
enum tanıtıcı_sözcükopt { sayım_listesi }opt bild_belirteçleriopt ;
```

Tanıtıcı_sözcük'e *sayım künyesi* denir ve tanımlanmakta olan tipe verilen isteğe bağlı bir isimdir. Daha ileride, aynı tipten başka değişken veya fonksiyon tanımlamaları yapmak gerekirse, bu isim kullanılabilir.

Sayım_listesi virgülle ayrılmış tanıttıcı sözcüklerden oluşan bir listedir. Bunlara *sayıcı* adı verilir.

Bild_belirteçleri, olası ilkleyenlerle izlenen, değişkenlerden oluşan bir listedir.

Birkaç örnek:

```
enum islecler { arti, eksi, carpi, bolu }
    isl1 = arti, isl2 = carpi, isl3;
enum gunler { pa, pt, sa, ca, pe, cu, ct };
enum gunler ilk_gun = pa, bir_gun, son_gun = ct;
enum { erkek, kadın }
    cinsiyetim = erkek, cinsiyetler[MAKSK];
```

İlk örnekte “**enum islecler**” tipini ve bu tipten *isl1*, *isl2*, *isl3* değişkenlerini tanımlamaktayız. Bu değişkenler herhangi bir zamanda *arti*, *eksi*, *carpi* ve *bolu* değerlerinin sadece bir tanesini alabilirler. Ayrıca, *isl1* ve *isl2* değişkenleri sırasıyla *arti* ve *carpi* değerleriyle ilklendirir. İlkleme işlemi tamsayı değişken ilklemesine çok benzer, kuralları da aynıdır. Üçüncü değişken, *isl3*, ilklennmemiştir, fakat, tıpkı diğer değişkenler gibi yürütme esnasında buna da yukarıdakilerden bir değer atanabilir.

İkinci ve üçüncü satırlarda da buna benzer bir tanımlama yapılmakta, ancak bu sefer iki adımda olmaktadır. Önce “**enum gunler**” tipini tanımlıyoruz—henüz değişken tanımı yok—sonra da *ilk_gun* gibi değişkenleri bu tipten tanımlıyoruz.

Dördüncü satırda künyeyi atlamış bulunuyoruz, böylece bundan sonra bu tipten yeni değişken veya fonksiyon tanımlamak istesek *sayım_listesi*ni tekrar belirtmek zorunda kalacağız. Bu örnekte ayrıca *sayım* tipinden elemanlardan oluşmuş bir dizi tanımlamasını da görmekteyiz.

Bu tanımlardan sonra, değişkenleri aşağıdaki gibi deyimlerde kullanabiliriz:

```
cinsiyetim = kadın;
bir_gun = ca;
isl3 = isl1;
if (isl1==bolu) ...
while (bir_gun!=ct) ...
printf("%d %d %d\n", cinsiyetim, bolu, son_gun);
```

cinsiyetim değişkeninin *kadın*, *son_gun*’ün de *ct* olduğunu varsayarsak, son deyim

1 3 6

yazacaktır. Bunun nedeni, *sayım* tipinden olan değişkenlerin aslında tamsayı değerleri sakladıkları ve değişmezlerin de aslında tamsayı olduklarıdır. Örneğin, *arti*, *pa* ve *erkek*’in değerleri 0’dır, *eksi* 1, *carpi* 2, *bolu* 3’tür. Normalde *sayım* değişmezlerine sıfırdan başlayarak sırayla değerler verilir, ancak C dilinde programcının istediği değişmez tamsayıyı bir sayıcıya vermesine izin verilir. Bundan sonra gelen sayıcılar da sonraki değerleri alırlar. Örneğin,


```
enum renk {
    siyah, mavi, yesil, kirmizi=4, sari=14, beyaz
};
```

Burada, *siyah*'ın değeri 0, *mavi*'nin 1, *yesil*'in 2 ve *beyaz*'ın 15'tir. Bir **enum** tanımında, iki sayıcı aynı değeri alacak şekilde tanımlanabilirler. Bir sayıcı eksi bir değere sahip olabilir.

Sayım tipinden olan değişkenler fonksiyonlara argüman olarak geçirilebilir ve bir fonksiyonun dönüş tipi sayım tipinden olabilir. Buna örnek bir fonksiyon aşağıda verilmiştir:

```
enum gunler gun_ekle (enum gunler g, int n)
{
    return (enum gunler) (((int) g + n) % 7);
}
```

Bu fonksiyon bir gün kabul edip *n* gün sonraki günü verir.

```
bir_gun = gun_ekle(pe, 31);
```

şeklindeki bir çağrı *bir_gun*'e *pa* atayacaktır. Sayım tiplerini kalıplarda da kullanabileceğimize dikkat ediniz. Bu örnekte iki tane kalıp kullanmaktayız: Bir tanesi, hesap yapmak için günü tamsayıya dönüştürmede, diğeri de sonucu tekrar "**enum gunler**" tipine dönüştürmede kullanılır.

Sistemimizde, sayım tiplerinden olan değişkenlerin tamsayı olduğunu düşünebilirsiniz, örneğin **enum** değerlerinin girdi ve çıktısını tamsayı biçiminde yapabilirsiniz. Tıpkı **int** değişkenleri gibi, sayıcıların (ve sonuçta sayım tipinden olan değişkenlerin) alacakları değerler 16 bit içinde kalmalıdır.

5.2. Yapılar

Yapılar ve birlikler C programları için karmaşık veri yapılarının tanımlanmasında anahtar rolü oynarlar. Bu kısımda yapılar anlatılmaktadır. Daha sonraki bir kısım ise birlikleri kapsamaktadır.

Pascal'la tanışıklığı olan okuyucularımız, herhalde C dilindeki yapıların Pascal'daki **record**'lara karşılık olduğunu fark edeceklerdir. Aynı tipte belirli sayıdaki değişkenleri saklamak için bir *dizi* kullanılabilir. Diğer taraftan, bir *yapı*, genelde *değişik* tipte olan bir veya daha fazla değişkeni toplamak için kullanılabilir. Niye, ayrı ayrı değişkenler kullanmak yerine bunları bir yapı içinde toplamaya gerek duyarız? Temel neden kolaylıktır. Örneğin, bir fonksiyona çok sayıda argüman geçirmek yerine, bazılarını yapılar içinde gruplayıp yapıları geçirebiliriz.

Şimdi yapıları nasıl tanımlayabileceğimizi ve kullanacağımızı görelim. Yapı tanımlamasının genel şekli şöyledir:

```
struct tanıtıcı_sözcük_opt { yapı_bild_listesi }_opt bild_belirteçleri_opt ;
```

struct anahtar sözcüğü bir yapının tanımlanmakta olduğunu göstermektedir. Her zaman olduğu gibi, tanımlanmakta olan değişken(ler)in bellek sınıfını göstermek için bu anahtar sözcüğün önüne uygun herhangi bir bellek sınıfı belirteci (örneğin, **static**, **extern**) konulabilir.

İsteğe bağlı olan *tanıtıcı sözcük*'e *yapı künyesi* adı verilir. Tanımlanmakta olan yapı tipine isim verir ve daha sonra benzer yapı değişkenleri tanımlamak için bir kısaltma işlevi görür. Eğer bu yapı tipi bütün program içinde bir defa kullanılıyorsa, daha sonra kullanılmayacağı için yapı künyesini atlayabiliriz.

Yapı_bild_listesi her bildirimin bir noktalı virgülle bitirildiği bir dizi bildirim içerir. Bu bildirimler (ilkleyen içermemeleri dışında) normal değişken tanımlamalarına benzer; fakat bu durumda değişkenler bildirimi yapılan yapı altında gruplanacaklar ve yapının üyeleri olacaklardır. Bir yapının üyeleri diziler veya yapılar gibi karmaşık tipler olabileceği için, C dilinde istediğimiz kadar karmaşık veri yapıları tanımlayabiliriz.

Bild_belirteçleri belirtilen yapı tipinden değişkenler tanımlayan isteğe bağlı bir listedir. Bunlar basit (yapı) değişkenleri, yapılara göstergeler, yapılardan oluşan diziler, yapı döndüren fonksiyonlar veya bunların herhangi bir birleşimi olabilir.

Bir örnek:

```
struct yk { int i; float f; } yd;
```

Burada, yk yapı künyesini ve “**struct** yk” tipinden yd değişkenini tanımlamış bulunuyoruz. “**struct** yk” tipi iki üye içerir, bir tamsayı, bir de gerçek sayı. yk yapı künyesi daha sonra aynı tipten başka değişkenlerin bildirimini yapmak için kullanılabilir. Örneğin,

```
struct yk yd1, *yg;
```

(yd'ye benzeyen) bir yd1 değişkenini ve “**struct** yk” tipindeki bir yapının saklandığı bir yere bir gösterge tanımlayacaktır.

Gelin birkaç örneğe daha bakalım. Bu kısmın başında verilen sözdizimde üç tane isteğe bağlı bölüm görmekteyiz. Herhangi bir zamanda bunlardan en çok bir tanesi belirtilmeyebilir. Böylece,

```
struct { int i; float f; } yd2;
```

tanımlamasında bir yapı değişkeni, yd2, tanımlanmakta, fakat yapı künyesi tanımlanmamaktadır;

```
struct yk { int i; float f; };
```

tanımlamasında ise bir yapı künyesi, yk, tanımlanmakta, fakat değişken tanımlamaları bulunmamaktadır; yani sadece bir tip bildirimi vardır. Daha sonra, bu ismi kullanarak, aşağıda yapıldığı gibi değişkenler tanımlayabiliriz:

```
struct yk ydz[5], yf(float), (*ydgf(int))[8];
```

Burada *yapı_bild_listesi*'ni tekrar belirtmeye gerek yoktur. ydz böyle 5 tane yapıdan oluşan bir dizedir; yf bir **float** argüman alan ve böyle bir yapı döndüren bir

fonksiyondur (bazı eski derleyiciler böyle bir bildirimi kabul etmeyebilirler, çünkü, eski “Standarda” göre, fonksiyonlar dizi veya yapı gibi “karmaşık” tipler döndürmezler); ydgf böyle 8 yapıdan oluşan bir diziyeye bir gösterge döndüren ve bir **int** argüman alan bir fonksiyondur. Son örneği anlamak için, işleçleri, uygulandıkları sırayı dikkate alarak (tersten) okuyun: (), *, []; yani yapı dizisi göstergesi fonksiyonu.

Şimdi de daha karmaşık bir örnek:

```
struct kisi {
    char *isim;
    int dogumtarihi;
    enum cinsiyet_tipi { erkek, kadın } cinsiyet;
    struct kisi *baba, *ana, *es,
        *cocuklar[MAKSCCK];
} kisiler[MAKSKS];
```

Yukarıdaki tanımda gördüğünüz kadarıyla, tıpkı diğer dillerde olduğu gibi, C dilinde de “karmaşık” veri yapıları tanımlanabilir. Burada, basit değişkenler, *göstergeler*, *sayım tipinden değişkenler*, *yapılara göstergelerden oluşan diziler* vs olan *üyeleri* içeren bir *yapı dizisi* görmekteyiz. Yapının bazı üyelerini tanımlarken yapının kendi tanımını da kullanmakta olduğumuza dikkat edin; tanımlamakta olduğumuz üyeler *gösterge* oldukları için bu herhangi bir problem oluşturmaz. Aslında, gösterge tanımlarken daha ileride tanımlanmış yapıları da kullanabiliriz. Yukarıda tanımlanan *kisiler* değişkeni, kolayca erişilip işlenebilen en fazla MAKSKS kadar kişinin bilgisini saklayabilir.

Bir yapının üyelerine ulaşmak için, bir nokta (“.”) ile gösterilen, *yapı üyesi işlecini* kullanırız. Bu işleç, yapı değişkeni ile üyesi arasına konur. Örneğin,

```
y.d.i = (int) y.d.f;
```

y.d yapı değişkeninin f üyesi içindeki tamsayıyı i üyesine atayacaktır. “.” işleci en yüksek önceliği olan işleçlerden biridir, böylece, örneğin y.d.i genelde sanki tek bir değişkenmiş gibi, etrafına parantez koymaya gerek duymadan kullanılabilir.

Şimdi, bir yapı göstergesi değişkeninin üyesine erişirken karşılaşılabileceğimiz küçük bir problemi inceleyelim. Örneğin, yg bir yapıya bir göstergedir; *yg yapıya erişmek için kullanılır, ancak onun üyesine erişmek için *yg.i kullanılamaz. Bunun nedeni önceliklerdir, “.”nin önceliği *’dan yüksektir, böylece *yg.i aslında *(yg.i) anlamına gelir, bu da yg’nin bir üyesi olan i’nin bir gösterge olduğunu varsayar. Doğru sonucu elde etmek için parantez kullanmak zorundayız: (*yg).i. Ancak bu da sıkıntılı olabilir. Bunun üstesinden gelmek için, C’ye başka bir işleç (->) eklenmiştir. yg->i tamamen (*yg).i ile eşdeğerdir ve daha kısa ve anlamlıdır.

Dört işleç, (), [], . ve ->, öncelik tablosunda en yüksek düzeyi oluştururlar ve referans oluşturmak için kullanıldıklarından diğer işleçlerden farklıdır.

Bazı eski C uygulamalarında, yapılarla yapabileceğimiz şeyler, & kullanarak bir yapının adresini almak ve üyelerinden birisine erişmekle sınırlıdır. Örneğin, yapılar fonksiyonlara geçirilip geri döndürülemez;

```
yapil = yapil2;
```

şeklinde her iki tarafında yapı değişkeni olan bir atama gerçekleştirilemez. Bu sınırlamaların üstesinden gelmek için göstergeler kullanılabilir. Buna rağmen, ANSI Standardını izleyen yeni C derleyicilerinde böyle sınırlamalar yoktur. Örneğin, bizim sistemde aşağıdaki program kabul edilmektedir:

```
#include <stdio.h>

struct kunye {
    int i;
    char *c;
} y1 = { 1, "bir" }; /* y1 yapil degiskeninin */
/* tanimlanmasi ve ilklenmesi */

struct kunye f /* parametresi ve donus tipi */
(struct kunye y) /* struct kunye olan fonksiyon */
{
    struct kunye z;
    z.i = y.i * 2; /* uyelere degerler ata */
    z.c = "iki kere";
    return z; /* bir yapil dondur */
} /* f */

void main (void)
{
    struct kunye y2, f(struct kunye);
    y2 = f(y1); /* yapili baska bir yapila ata */
    printf("%d %s\n", y2.i, y2.c);
} /* main */
```

ve çıktı şöyledir

```
2 iki kere
```

Burada yapılarla yapılacak üç şeyi görmekteyiz: Bir yapıyı bir argüman olarak geçirmek, fonksiyondan bir yapı döndürmek ve bir yapıyı başka bir yapıya atamak. Yapı ataması, kaynak yapının her üyesinin hedef yapının ilgili üyesine atanması demektir ve yapının boyuna göre bir miktar zaman alabilir.

Yukarıdaki örnekte, yapı değişkenlerini ilkleme yöntemini de görmekteyiz. Sözdizimi diziler için kullanılabilmektedir. İlklenecek değişken, = atama simgesinden sonra virgülle ayrılıp çengelli parantezler içine alınmış değişmez ifadelerden oluşan bir liste ile izlenir. Her üye ilgili ifadenin değerine ilklenir. Eğer ifadelerden daha fazla üye varsa, geri kalan üyeler C dilindeki varsayılan değere, yani sıfıra ilklenir.

5.3. Yeni Tip Tanımlama

Türetilmiş veri tipleri üzerindeki tartışmamıza devam etmeden önce, gelin, dilin içindeki basit tipler gibi görünen, ancak karmaşık tip olan yeni, yani kullanıcı tarafından tanımlanmış, tiplere ve böyle tiplerin tanımlanması için kullanılan yönteme bakalım. Bu

amaçla, C dilinde, bir tipe bir tanıttıcı sözcük vererek yeni bir tip tanımlamaya yarayan **typedef** bildirimi bulunmaktadır. Bildirim, anlambilim açısından oldukça farklı olmasına rağmen, sözdizimsel olarak bir değişken tanımlaıdır. Sadece başına **typedef** anahtar sözcüğü getirilir. **typedef** anahtar sözcüğünün bir *bellek sınıfı belirteci* olduđu kabul edilir, ve C dili bir bildirimde en fazla bir tane bellek sınıfı belirtecine izin verdiđi için **typedef**'in arkasına (veya önüne) **auto**, **register**, **static** veya **extern** konulamaz. Gelin birkaç örneđe bakalım:

```
typedef unsigned short int kısa;
typedef char *kdizisi, *kdfonk(char *, char *);
typedef struct { float ger, im; } kompleks;
typedef int vektor[6], matriks[8][6];
typedef enum { yanlis, dogru } mantiksal;
```

(kisa, kdizisi, kdfonk, kompleks, vektor, matriks ve mantiksal gibi) yeni tanımlanan tanıttıcı sözcüklerin kendileri değişken veya fonksiyon değillerdir (eđer **typedef** anahtar sözcükleri olmasaydı öyle olacaktı), ancak bu tür değişken veya fonksiyon tanımlamaya yarayacak tiplerdir. Böyle tip isimleri kalıplarda ve **sizeof**'un işlenenleri olarak kullanılabilir. (**sizeof** bir sonraki kısımda anlatılmaktadır.) Örneđin, aşağıdakiler birbirlerine eşdeğerdir:

kisa i, j, k;	ile	unsigned short int i, j, k;
kdizisi s1, s2;	ile	char *s1, *s2;
kdfonk strcpy;	ile	char *strcpy(char *, char *);
kompleks z[5];	ile	struct { float ger, im;} z[5];
vektor a[8];	veya	
matriks a;	ile	int a[8][6];
mantiksal m1,	ile	enum { yanlis, dogru } m1,
md=dogru, my=yanlis;		md=dogru, my=yanlis;

typedef kullanmanın temel nedeni uzun bildirimleri kısaltmasıdır. Fakat, mantiksal olarak, programcı C dilinin sağladığı tiplerin yanına sanki *yeni* bir tip ekliyormuş gibi düşünebilir. Bu kolaylık programlama çabasını azaltır, programın okunaklılığını geliştirir ve programlarımızı daha “yapısal” hale getirir. Ayrıca, önemli ve karmaşık tip bildirimleri programın belirli bir yerinde toplandığında, programı belgelemek ve daha sonra onu anlayıp değiştirmek daha kolay olacaktır.

typedef bildirimlerinin etki alanı değişkenlerde olduđu gibidir. Bir blok içinde tanımlanmış bir tip dış bloklara erişilebilir değildir, fakat bir dış blokta tanımlanmış bir tip, tekrar tanımlanmadığı sürece, bir iç blokta kullanılabilir. Dışsal düzeyde yapılmış bir **typedef** bildirimi kaynak dosyasının sonuna kadar etkisini sürdürecektir, fakat ayrı derlenen dosyalarda değil. **struct**, **union** ve sayım tipi bildirimlerinin etki alanı da aynıdır.

5.4. sizeof İşleci

C dilinin bir diğer anahtar sözcüğü de **sizeof**'tur. **sizeof** C dilinde bir anahtar sözcük ile ifade edilen tek işleçtir. Bir fonksiyon değildir; diğer tekli işleçler ile aynı önceliğe sahip olan ve sağdan sola birleşme özelliği gösteren bir tekli işleçtir. Genelde, ya bir değişkene yada bir tipe uygulanır. Bu işleç işlenenin tipinden olan bir değişkenin bellekte tutacağı bayt sayısını (bir değişmez) döndürür. Bir önceki kısmın değişken tanımlamalarını kullanan örnekler aşağıda verilmiştir:

```
printf("%ld %ld %ld %ld %ld\n", (long)sizeof i,
      (long)sizeof a, (long)sizeof s1, (long)sizeof z[3],
      (long)sizeof 1.0);

printf("%ld %ld %ld %ld %ld\n", (long)sizeof(int),
      (long)sizeof(float), (long)sizeof(double),
      (long)sizeof(int *), (long)sizeof(kompleks[3]));

printf("%ld %ld %ld %ld %ld\n", (long)sizeof(vektor),
      (long)sizeof(long[4]),
      (long)sizeof(struct { char *c; vektor v[5]; }),
      (long)sizeof *s1, (long)sizeof(a[1]));
```

Bu deyimler farklı sistemlerde farklı çıktılar görüntüleyecektir. Sistemimizde çıktı şöyledir:

```
2 96 2 8 8
2 4 8 2 24
12 16 62 1 12
```

Bunun anlamı, **int** değişkenlerinin 2 bayt, **float** değişkenlerinin 4 bayt, **double** değişkenlerinin 8 bayt ve gösterge değişkenlerinin 2 bayt tuttuğudur. Her gerçekleştirmede **sizeof(char)**'ın 1 olduğu tanımlanmıştır. Diğer sayıları kendiniz doğrulamaya çalışın. Bir **long** kaç bayttır?

sizeof'un değerinin derleme zamanında belirlendiğine dikkat edin, böylece **sizeof** bir *değişmez ifade*nin kullanılmasına izin verilen yerlerde kullanılabilir. Standarda göre **sizeof** işlecinin verdiği sayının **int** olması beklenmemelidir; gerçekleştirmeye bağlı olarak **long** veya herhangi bir uygun tamsayı tipi olabilir. Bu (**unsigned** olan) tip standart başlık dosyası `stddef.h`'de `size_t` ismi ile tanımlanmıştır. Bu sayıyı `printf`'e geçirmeden önce bir kalıpla **long**'a çevirecek ve **long** için olan dönüşüm tanımlamasını kullanacak kadar dikkatli davrandık.

sizeof ile ilgili bir sözdizimsel ayrıntı da, bir ifade yerine bir tipe uygulandığında tipin parantezler içine alınması *gerektiğidir*.

5.5. Birlikler

Birlikler bir sözdizimsel, bir de anlambilimsel fark dışında yapılara çok benzerler. Bir birliğin sözdizimi ayırdır, sadece **struct** yerine **union** anahtar sözcüğünü kullanırız. Yapılarda, üyeler belleğe birbiri ardınca yerleştirilirler (yani bir üyenin değerinin üstüne sarkması söz konusu değildir), böylece her üyeyi diğerlerinden bağımsız işleyebiliriz. Birliklerde üyeler aynı bellek bölgesini paylaşırlar. Böylece, herhangi bir zamanda, üyelerden sadece bir tanesi ile ilgilenebiliriz. Derleyici, en büyük üyeyi içine alabilecek kadar geniş bellek ayırır. Örneğin,

```
union cifdg {
    char c;
    int i;
    float f;
    double d;
    int * g;
} bd = { 'A' };
```

5 yerpaylaşımlı üyeden oluşan **bd** birlik değişkenini tanımlar. **sizeof** **bd**’dir, bu da en büyük üye olan **double** değişkeninin boyudur. Üyelere yapılarda olduğu gibi aynı yolla ulaşılır (yani “.” veya “->” işlecini kullanarak). Örneğin,

```
bd . d = 9.8696044011;
printf("%c\n", bd.c);
```

Bu şekilde bu iki deyimi arka arkaya kullanmanın yanlış olduğuna dikkat edin, çünkü bir taraftan **double** bir değer koyup, diğer taraftan bir **char** almaya çalışmaktayız. Ayrıca, her üye için bir ilkleynin bulunduğu yapılara karşılık, birliklerde en fazla bir tane ilkleynin olabildiğine dikkat edin. İlkleyen ilk üyeye uygulanır. Örneğin, yukarıdaki tanımda **bd.c**’ye ‘A’ atanmaktadır; daha sonra, yukarıdaki iki deyim yerine getirildikten sonra, **bd.d** çıktısını elde ederiz, bu da anlamsız bir değerdir, çünkü **bd.d**’ye atanan değer **bd.c**’deki değeri bozar.

Herhangi bir zamanda üyelerinden en fazla bir tanesinin bir değer tutacağını bildiğimiz zamanlarda, bellekten tasarruf etmek amacıyla genelde birlikler kullanılır. Bu özellik, daha önce verilmiş olan bir örneğin genişletilmiş uyarlaması olan, aşağıdaki örnekte kullanılmıştır:

```
struct kisi {
    char *isim;
    int dogumtarihi;
    enum cinsiyet_tipi { erkek, kadın } cinsiyet;
    union {
        int ahbtarihi;
        char *ksoyadi;
    } cbil;
    struct kisi *baba, *ana, *es,
        *cocuklar[MAKSCCK];
} kisiler[MAKSKS];
```

Burada kişinin cinsiyetine bağlı olan iki adet bilgi vardır: Erkekler için askerlik hizmetini bitirme, yani terhis tarihi (ahbtarihi), kadınlar için ise kızlık, yani evlenmeden önceki soyadı (ksoyadi). Aynı zamanda her iki üyenin anlamlı bir değer saklamayacaklarını bildiğimiz için, bellekten tasarruf etmek amacıyla, bunları bir birliğin içine alıyoruz. Hangi üyeyi (kisiler[birisi].cbil.ahbtarihi veya kisiler[birisi].cbil.ksoyadi) kullanmamız gerektiğini belirlemek için kisiler[birisi].cinsiyet içindeki bilgiyi kullanabiliriz.

5.6. Alanlar

Bazı durumlarda, dar bir yer içine (örneğin bir tamsayı boyunda bir bölgeye) birçok bilgi sığdırmamız gerekebilir. Bunun nedeni bellek tasarrufu olabildiği gibi, program dışındaki bazı durumlardan (örneğin donanım aygıtlarına olan arabirimlerden) kaynaklanabilir. Bunu yapmak için bir yapının üyelerini bit uzunluklarını da vererek tanımlarız. Bu tür üyelere *alan* adı verilir. Doğal olarak, uzunluklar sıfırdan küçük olmayan değişmez ifadelerdir. Tanım yapılarında olduğu gibidir; sadece her üyenin arkasına iki nokta ile uzunluk verilebilir. Erişim de yapılarında olduğu gibi “.” veya “->” işleçleri ile yapılır. Bir örnek:

```
struct tarih {
    unsigned yıl : 7;
    unsigned ay : 4;
    unsigned gun : 5;
};
```

Burada, tarihin iki bayt (`sizeof(struct tarih)` 2’dir) içine sığdırılabileceği bir yapı tanımlanmaktadır. yıl 7 bit kullanır (işaretsiz olduğu için, değeri 0’dan $2^7-1=127$ ’e kadar değişebilir ve örneğin 1900 ile 2027 arasındaki yılları ifade etmek için kullanılabilir), ay 4 bit (0’dan $2^4-1=15$ ’e kadar değişir), gun ise 5 bittir (0’dan $2^5-1=31$ ’e kadar değişir). Eğer uzunlukları belirtmemiş olsaydık, her üye 2 bayt kullanacak, böylece tüm yapı 6 bayt kaplayacaktı.

Alanlarla uğraşan birisinin dikkat etmesi gereken kurallar şöyle özetlenebilir:

1. Alan tipi **unsigned** bir tamsayı olmalıdır. Standart, en azından **unsigned int**’in desteklenmesini gerektirir.
2. Bitlerin alanlara atanma tarzı (soldan sağa veya sağdan sola) derleyiciye bağlıdır.
3. Eğer bir alan *sözcük sınırlarını* (sistemimizde, **short** ve **int** için 16, **long** için 32 ve **char** için 8 bitlik sınırları) kesecek şekilde tanımlanırsa bir sonraki sözcükten başlatılır; kullanılmakta olan sözcükteki bitler kullanılmaz. Örneğin,


```
struct dnm {
    unsigned a1 : 12;
    unsigned a2 : 5;
    unsigned a3 : 12;
};
```

aşağıdaki biçimde 6 bayt kaplar:

12 bit a1, 4 bit kullanılmıyor (ilk sözcük, yani iki bayt),

5 bit a2, 11 bit kullanılmıyor (ikinci sözcük),

12 bit a3, 4 bit kullanılmıyor (üçüncü sözcük).

Eğer **unsigned** yerine **unsigned long** kullanmış olsaydık, yapı aşağıdaki biçimde 4 bayt kaplayacaktı:

12 bit a1, 5 bit a2, 12 bit a3, 3 bit kullanılmıyor (bir **long** sözcük, yani 4 bayt).

Eğer **unsigned char** kullanmış olsaydık, tanım kabul edilmeyecekti, çünkü alan a1 12 bit uzunluğundadır, oysa bir **char** en fazla 8 bit tutabilir. Bu, bir alanın kendi tipinden daha geniş olamayacağı anlamına gelir. Eğer bir alan için bit sayısı belirtilmemişse, o tip için en büyük değer kullanılır.

4. Alanlardan oluşan diziler tanımlanamaz.
5. *Adres alma* işleci (&) alanlara uygulanamaz, böylece alanlar gösterge kullanarak doğrudan doğruya adreslenemezler.
6. Alan isimleri isteğe bağlıdır. Eğer bir alanı kullanmayı düşünmüyorsanız—örneğin yukarıdaki örnekte a2'yi—tipi ve alan ismini atlayın. Örneğin,

```
struct dnm {
    unsigned a1 : 12;
    : 5;
    unsigned a3 : 12;
};
```

7. Eğer bir sonraki sözcüğün başına atlamak istiyorsanız, araya 0 genişliğinde bir boş alan koyun. Örneğin, sistemimizde

```
struct dnm {
    unsigned a1 : 12;
    : 0;
    unsigned a3 : 12;
};
```

ile

```
struct dnm {
    unsigned a1 : 12;
    : 4;
    unsigned a3 : 12;
};
```

eşdeğerdir. Neden?

8. Alanlar da tıpkı yapı üyeleri gibi ilklenebilirler, fakat alan için çok büyük olan bir değer vermemeye dikkat etmek gerekir.

5.7. Bellek Ayırma

Bazı durumlarda, programcı program için gereken ana bellek miktarını önceden belirleyemez. Örneğin, bu, büyük oranda program girdisine bağlı olabilir. Bir dosyayı sıraya sokan bir program, bütün dosyayı alacak kadar ana belleğe gereksinim duyabilir. Bir dosya istenildiği kadar uzun olabildiğine göre, yürütme esnasında, sistemin bize sağlayabileceği kadar daha fazla bellek elde edebilme yöntemine gereksinimimiz vardır.

Standart kütüphane, buna izin verecek bir fonksiyon, `calloc(n, boy)` içerir. İki argümanı `size_t` tipinden tamsayılar olmalıdır. `boy` boyunda `n` tane değişken tutabilecek sığara ilklenmiş bir bellek bloğuna bir gösterge döndürür.

Daha basit başka bir fonksiyon `malloc(bayts)`'tır ve `bayts` kadar bayttan oluşan bir bellek bloğuna bir gösterge döndürür. Argümanın tipi yine `size_t`'dir. `size_t` gerçekleştirmeye bağlı olan ve en büyük bellek boyunu saklayabilecek kadar büyük olan bir tamsayıdır.

`free(g)` fonksiyonu daha önce `calloc` veya `malloc` ile ayrılmış bulunan ve `g` ile gösterilen bellek bloğunu serbest bırakacaktır.

`malloc` ve `calloc`'un dönüş tipi ve `free`'nin argüman tipi genel gösterge tipi olan `"void *"`dir. `"void *"` tipi bekleyen bir fonksiyona bir gösterge argümanı geçirdiğinizde, bir kalıp kullanarak, bunu `"void *"` tipine dönüştürmeniz gerekir. Aynı şekilde, bir fonksiyon `"void *"` döndürüyorsa, bunu uygun gösterge tipine dönüştürmeniz gerekir. Her tip gösterge, herhangi bir bilgi yitimine uğramadan, `"void *"` tipine ve daha sonra eski tipine dönüştürülebilir. `stdlib.h` başlık dosyası bu fonksiyonlar için gerekli bildirimleri içerir ve bu bölümün sonunda bu fonksiyonları kullanan örnek bir program bulunmaktadır.

5.8. Karmaşık Tipler

Bu kısımda, dizi dizileri, dizilere göstergeler, gösterge dizileri ve göstergelere göstergeler gibi daha karmaşık veri tiplerini inceleyeceğiz. Bir tanıımı istediğimiz kadar karmaşıktırabileceğimize dikkat edin; örneğin, tamsayı dizilerine göstergeler içeren diziye bir gösterge. Biz ise burada yalnızca, yukarıda sıraladığımız gibi, "iki-düzeyli" karmaşık veri tiplerine bakacağız.

5.8.1. Dizi Dizileri

Bir *dizi* benzer elemanlardan oluşan bir topluluktur. Eleman sayısı derleme sırasında belirlenir, böylece C dilinde devingen dizilere izin verilmez. Bir dizinin elemanları (alan veya fonksiyon dışında) herşey olabilir; ayrıca dizi de olabilir, böylece çok-boyutlu diziler tanımlanabilir. Bir dizinin elemanlarına ulaşmak için indisler kullanılır. İndisler tamsayı ifadelerdir ve, dizinin n tane elemanı varsa, değerleri 0 ile $n-1$ arasında değişebilir.

Tek-boyutlu dizileri nasıl kullanacağımızı biliyoruz. Şimdi çok boyutlu bir dizi tanımına bakalım:

```
int a[3][4] = {
    {1, 2, 3, 2+2},
    {5, 6, 7, 1<<3},
    {3*3, 10, 11, 0xF&~03}
};
```

Burada çok-boyutlu bir dizinin (bir dizi dizisinin) tanımının bütün boyutlarda boyların ayrı ayrı köşeli parantezler içine alınarak verildiği görüyoruz. Yukarıdaki tanımda dizinin, her biri 4 elemanlı 3 “vektör” şeklinde düzenlenmiş 12 elemanı vardır. Diğer bir deyişle, diziler satır şeklinde saklanırlar, yani son indis en hızlı değişendir. İlkeme beklendiği gibi yapılır, dizinin her elemanı için üç madde bulunmaktadır, bunlar da virgülle ayrılmış vektörlerdir. Bu maddelerin her biri birbirinden virgülle ayrılıp çengelli parantezler içine alınmış 4 *değişmez ifade*den oluşurlar. Bütün (12) değerleri de belirtmemize gerek yoktur. Örneğin,

```
int b[3][4] = {
    {1, 2, 3},
    {5, 6}
};
```

ile

```
int b[3][4] = {
    {1, 2, 3, 0},
    {5, 6, 0, 0},
    {0, 0, 0, 0}
};
```

aynıdır. Eğer *bütün* değerleri belirtirsek içteki çengelli parantezlere gerek yoktur:

```
int a[3][4] =
    { 1, 2, 3, 2+2, 5, 6, 7, 1<<3, 3*3, 10, 11, 0xF&~03 };
```

Eğer derleyici dizinin boyunu ilkleyenden çıkarabilirse, o zaman boyu belirtmeye gerek yoktur. Örneğin,

```
int a[][4] = {
    {1, 2, 3, 2+2},
    {5, 6, 7, 1<<3},
    {3*3, 10, 11, 0xF&~03}
};
```

a için ilk verilen tanıma eşdeğerdir. Bildirimdeki ilk (yani belirtilmeyebilen) indisin sadece dizi tarafından gerek duyulan bellek miktarını belirlemede kullanıldığına dikkat edin; indis hesaplamalarında kullanılmamaktadır.

Bir dizi elemanına ulaşmak için “[]” işlecini kullanırız. Eğer n -boyutlu bir dizimiz varsa, n tane böyle işlec kullanılmalıdır. Örneğin, $a[1][2]$ değeri 7 olan dizi elemanını göstermektedir. Pascal programlama dilinden farklı olarak $a[1,2]$ —veya $a[(1,2)]$ — $a[2]$ ’ye eşdeğerdir ve bir **int**’in beklendiği bir yerde kullanılamaz.

5.8.2. Dizilere Göstergeler

Daha önce de açıklandığı gibi, bir fonksiyon parametresi olarak tanımlanmış tek boyutlu bir dizinin boyunun belirtilmesine gerek yoktur, çünkü bu bir dizi değil, gerçekte bir göstergedir. Çok-boyutlu bir dizide ne olur? Kural şöyledir: İlki dışında bütün boyutların boyunu belirtmeniz gerekir. Örneğin,

```
f (int uc_b[][5][7])
{ ... }
```

Gelin bunun nedenine bakalım. $uc_b[0]$ ’ın 5×7 boyunda iki-boyutlu bir dizi olduğuna dikkat edin, $uc_b[1]$ ilkinden 35 **int** ileride olan bir sonraki iki-boyutlu dizidir. Yani, $uc_b[1]$ veya $*(uc_b+1)$ —bir gösterge ifadesi— $uc_b[0]$ ’ın 35 **int** ilerisini göstermelidir. Bu, uc_b ’nin 5 çarpı 7 (yani 35) elemanlık bir diziye gösterge olduğu anlamına gelir. Yukarıdaki parametre bildirimi şöyle de yazılabilir:

```
int (*uc_b)[5][7];
```

bu da yukarıdaki ile tamamen aynıdır. Parantezlere dikkat edin. Eğer yazılmasalardı,

```
int *(uc_b[5][7]);
```

anlamına gelecek, bu da **int** göstergelerden oluşan iki-boyutlu bir dizi anlamına gelecekti; bir sonraki altkısımın konusu.

Aşağıdaki örnek çok-boyutlu dizilerin kullanımını özetlemeye çalışmaktadır. Programı inceleyip, onu izleyen çıktıyı doğrulamaya çalışın.

```
#include <stdio.h>
int x[3][5][7];          /* uc-boyutlu bir dizi */

void f (int a[][5][7])    /* iki-boyutlu diziye gösterge */
{
    printf("%d\t%d\n", a[0][0][0],
           (int)sizeof a[0][0][0]);
    printf("%d\t%d\n", a[0][0], (int)sizeof a[0][0]);
    printf("%d\t%d\n", a[0], (int)sizeof a[0]);
    printf("%d\t%d\n", a, (int)sizeof a);
    printf("\n%d\t%d\n", x[0][0][0],
           (int)sizeof x[0][0][0]);
}
```

```

printf("%d\t%d\n", x[0][0], (int)sizeof x[0][0]);
printf("%d\t%d\n", x[0], (int)sizeof x[0]);
printf("%d\t%d\n", x, (int)sizeof x);
printf("\n%d\t%d\t%d", (int)a[0], (int)a[1],
      ((int)a[1]-(int)a[0]) / (int)sizeof(int));
printf("\n%d\t%d\t%d", (int)x[0], (int)x[1],
      ((int)x[1]-(int)x[0]) / (int)sizeof(int));
printf("\n%d\t%d\t%d", (int)a[0][0], (int)a[0][1],
      ((int)a[0][1]-(int)a[0][0]) / (int)sizeof(int));
printf("\n%d\t%d\t%d", (int)x[0][0], (int)x[0][1],
      ((int)x[0][1]-(int)x[0][0]) / (int)sizeof(int));
}

void main (void)
{ f(x); }

```

Çıktı şöyledir:

0	2
3384	14
3384	70
3384	2

0	2
3384	14
3384	70
3384	210

3384	3454	35
3384	3454	35
3384	3398	7
3384	3398	7

5.8.3. Gösterge Dizileri

Daha önceki bir bölümde, geleneksel olarak argv adı verilen, main'in ikinci parametresinin bir karakter göstergeleri dizisi olduğunu görmüştük. Aslında bunu anlamak ta o kadar zor değildir; eğer belirli bir sayıda göstergemiz varsa, diğer tipteki değişkenler için yaptığımız gibi, bunları da bir dizi içinde saklayabiliriz. Bir örnek görelim,

```

char *msj[] = {
    "Tamam",
    "Disk dolu",
    "Sozdizim hatasi",
    ...
    "Dosya bulunamadi"
};

```

Burada, ilkleyen içinde verilen karakter dizilerinin sayısı kadar boyda bir dizimiz vardır. Her karakter dizisinin temel adresi msj dizisi içinde arkada gelen gösterge

elemanları içinde saklanmaktadır. Bu `msj`'yi iki-boyutlu bir karakter dizisi olarak tanımlamaktan iyidir. Bellek kullanımı dikkate alarak bunun nedenini açıklamaya çalışın.

Bu diziyi kullanabilecek bir hata yordamı şöyledir:

```
void hata (int hk)
{
    printf("<%d>: %s.\n", hk, msj[hk]);
    exit(hk);
}
```

Burada `exit`, programı bitiren bir kütüphane fonksiyonudur. `stdlib.h` başlık dosyasında bildirilmiştir. `int` tipinden olan argümanı da, çağıran sürece—tipik olarak işletim sistemine—döndürülen bir durum kodudur. Geleneksel olarak, 0 normal sonuçlanmayı, sıfırdan farklı bir sayı da farklı bir durumu göstermek için kullanılır.

Başka bir örneğe bakalım:

```
int cikis(void), ekleme(void),
    silme(void), degistirme(void);

int (*fonkt[]) (void) = {
    cikis, ekleme, silme, degistirme
};
```

Bu, fonksiyonlara göstergelerden oluşan bir dizidir. Tekrar, işleç önceliklerini anımsayın. İlk olarak fonksiyon tiplerini tanımlıyoruz. Fonksiyonların hepsinin parametresiz ve dönüş tiplerinin de hep aynı olduğuna dikkat edin. Daha sonra bu fonksiyonlara göstergelerle diziyi ilklıyoruz. Bildiğiniz gibi, eğer bir fonksiyon ismi arkasında parantez bulunmuyorsa, bu o fonksiyona bir göstergedir. Bu dizideki bir fonksiyonu çağırmak için,

```
(*fonkt[secenek])()
```

kullanılır; `secenek` burada bir tamsayıdır. Eğer `secenek` 0'sa `cikis`, 1 ise `ekleme` çağrılır vs.

5.8.4. Göstergelere Göstergeler

Göstergeler değişken olduklarına göre, onlara da gösterge tanımlanabilir. Pratik bir durum, bir fonksiyona geçirilen bir gösterge dizisidir. `main`'in ikinci parametresi böyle bir örnektir, argv `char` göstergelerine bir gösterge olarak düşünülebilir. Böylece,

```
void main (int argc, char *argv[])
```

ve

```
void main (int argc, char **argv)
```

tanımları eşdeğerdir. Aşağıda, Altkısım 4.2.3'te verilen programın bir başka şekli verilmiştir:

```
#include <stdio.h>

void main (int argc, char *argv[])
{
    printf("Merhaba, benim ismim %s.\n", *argv);
    printf("Argumanlarım şunlar");
    while (*++argv)
        printf(", %s", *argv);
    printf(".\n");
}
```

argv'nin en sonunda bir NULL göstergenin bulunduğuna dikkat edin.

Bir sonraki kısımda bir göstergeye gösterge örneği vardır.

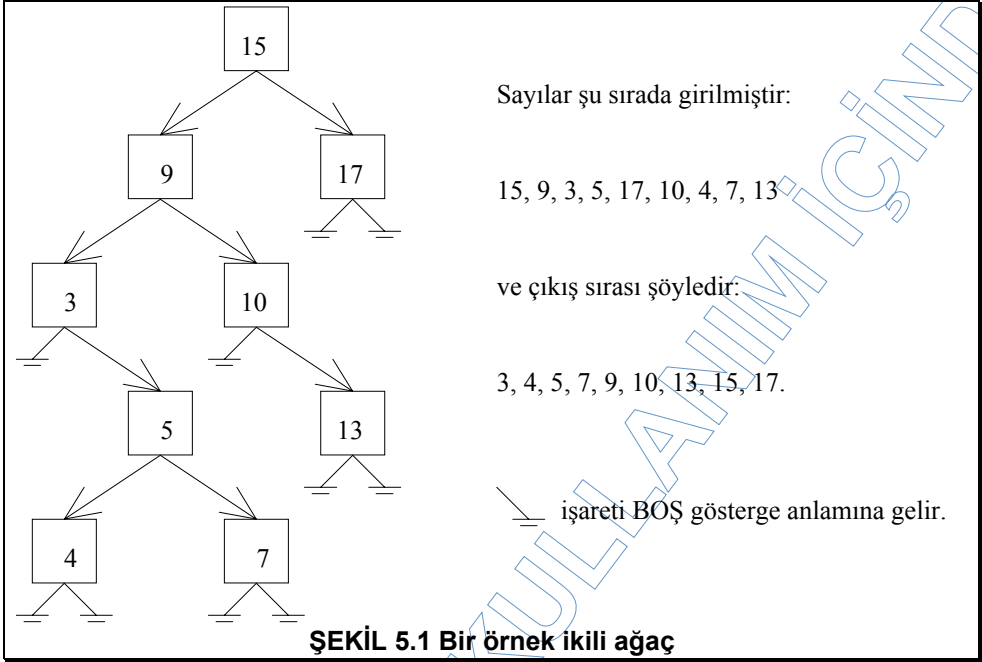
5.9. Bir Örnek—Dosya Sıralama

Bu kısımda, malloc ve free kütüphane fonksiyonlarını kullanan ve bu bölümde öğrenilmiş bazı kavramları gösteren bir örneğe bakacağız. Problem bir metin dosyasındaki satırları sıraya sokmaktır. Sıraya sokma düzeni, sistem tarafından sağlanan sıraya göre olacaktır. Bu, bizim sistem için, artan ASCII karakterleri sırasıdır. (Ek A'ya bakınız.) Bu program MS-DOS'un SORT komutuna benzemektedir.

Yöntemimiz ikili bir ağaç kullanmak olacaktır. Ağacın her düğümünde bir bilgi alanı ve iki gösterge vardır: biri sola, biri sağa. Algoritma şöyledir:

- Boş bir ağaçla başla.
- Bir satır oku; bir düğüm oluştur; bilgi bölümünü satırla doldurup iki göstergeye BOŞ (NULL) koy. Bu, kök düğümü olacaktır.
- Bir sonraki satırı oku; bu satır için yukarıdaki gibi bir düğüm oluştur. Şimdi bu düğümün ağaç içine konulacak yeri araştır. Kök düğümüne olan göstergeden başlayarak ve BOŞ bir gösterge elde edilmediği sürece göstergeleri şu şekilde izle: Eğer yeni satır, eldeki gösterge tarafından işaret edilen düğümdeki satırdan sözlük sırası açısından küçükse, göstergeye o düğümün sol göstergesini aktar; aksi takdirde sağ göstergesi aktar; bu işlemi tekrarla. Sonunda BOŞ bir göstergeye ulaşılabilecektir; bu göstergesi yeni düğümü gösterecek şekilde değiştir.
- Girilen bütün satırlar için önceki adımı tekrarla. Sonunda—ille de dengeli olmasına gerek olmayan—bir ikili ağaç elde edilecektir.

Sayısal değerler kullanılarak elde edilen örnek bir ikili ağaç Şekil 5.1'de gösterilmiştir.



Ağacı bastırmak için aşağıdaki özçeğrili algoritmayı kullanacağız: Kök düğüme olan göstergeden başla. Eğer gösterge BOŞ değilse, soldaki altağacı bastır; eldeki düğümün bilgisini bastır; sağdaki altağacı bastır. “Altağaç”ların basım işi aynı tümcede anlatılan algoritma kullanılarak yapılacaktır. Bu algoritma eninde sonunda bitecektir; çünkü tümce başında “eğer” koşulu vardır. Dikkat ederseniz, bir ağaçtaki göstergelerin yarısından fazlası BOŞtur. Neden?

Bu algoritmayı bir program şeklinde uygulamaya sokmak için düğüm için ne tür bir veri yapısı kullanacağımıza karar vermemiz gerekir. Doğal olarak, bu, üç üyeden oluşan bir yapı (**struct**) olacaktır: satir, sol ve sag; bu son ikisi de aynı yapıya olan gösterge tipinden olacaktır. Yani, bir öz-referanslı yapımız olacaktır. İlk üye (satir), belirli bir üst limite (örneğin 80 karaktere) kadar olabilecek bir satırı saklayabilecek büyüklükte bir karakter dizisi şeklinde tanımlanabilir. Fakat bütün satırlar bu kadar uzun olamayacaklarına göre, dizi içindeki alanın büyük bir bölümü kullanılmayabilecektir, yani bellek israfı olacaktır. Bundan dolayı satir bir karakter göstergesi şeklinde tanımlanmıştır ve satırın boyu kadar, yürütme esnasında istenen, bellek bölgesinin başına işaret edecektir.

Şimdi de program:

```
1. #include <stddef.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <string.h>
5. #define MAKSSU 80          /* en büyük satir uzunlugu */
6.
```



```

7.  typedef struct dugum {      /* struct dugum'un tanimi */
8.      char *satir;           /* bilgi */
9.      struct dugum *sol, *sag;
10. } *dugumg;                  /* dugumg tipinin tanimi*/
11.
12. void *yeni (size_t d)        /* bellek ayirma */
13. {
14.     void *g;
15.
16.     if ((g=malloc(d))!=NULL)
17.         return g;           /* tamam */
18.     fprintf(stderr, "SIRALA: Yetersiz bellek");
19.     exit(1);                 /* programi kes */
20. } /* yeni */
21.
22. size_t satiroku              /* satir uzunlugu dondur; 0 dosya sonu */
23. (char *d, int u)
24. {
25.     register char *g = d;
26.     register int c;
27.
28.     while ((c=getchar())!='\n' && c!=EOF)
29.         if (u-->1)
30.             *g++ = (char)c;   /* daha uzun satirlari kes */
31.     if (c==EOF && g==d)
32.         return 0;            /* bir sey okunamadi */
33.     *g++ = '\0';
34.     return g-d;              /* asil uzunluk arti bir ('\0' icin) */
35. } /* satiroku */
36.
37. void satirlaribas            /* agacin ozcagrili taranmasi */
38. (dugumg g)
39. {
40.     if (g!=NULL) {
41.         satirlaribas(g->sol);
42.         printf("%s\n", g->satir);
43.         satirlaribas(g->sag);
44.         free((void *)g->satir);
45.         free((void *)g);
46.     }
47. } /* satirlaribas */
48.
49. void main (void)
50. {
51.     char satir[MAKSSU+1];    /* MAKSSU arti bir ('\0' icin) */
52.     size_t su;               /* simdiki satir uzunlugu */
53.     dugumg kok = NULL, *g;
54.
55.     while ((su=satiroku(satir, (int)sizeof satir)) > 0) {
56.         g = &kok;
57.         while (*g!=NULL)
58.             g = (strcmp(satir, (*g)->satir) < 0)
59.                 ? &(*g)->sol : &(*g)->sag;
60.         *g = (dugumg) yeni(sizeof(struct dugum));
61.         strcpy((*g)->satir = (char *)yeni(su), satir);
62.         (*g)->sol = (*g)->sag = NULL;
63.     }
64.     satirlaribas(kok);
65. } /* main */

```

Programın başında, yapı (**struct** dugum) ve böyle bir yapıya bir gösterge (dugumg) için tipler tanımlamış bulunuyoruz. main yordamı, her okunan satır için bir defa yürütülen büyük bir **while** döngüsünden oluşmaktadır.

satiroku fonksiyonu satir karakter dizisinin içine bir satır okur ve okunan karakter sayısı artı bir (en sona konan '\0' için) verir. Dosya sonu (EOF) durumunda ise 0 döndürür.

main içinde iki önemli değişkenimiz vardır: kok, düğüme bir göstergedir ve her zaman kök düğümüne işaret eder, g ise bir düğüme olan göstergeye göstergedir; bir düğümün sol yada sag göstergelerine işaret eder. Bir defa bir satır okundu mu, g kök düğüm göstergesine işaret edecek şekilde getirilir.

Sonra, g tarafından işaret edilen yerde NULL (BOŞ) bir gösterge olmadığı sürece, g'nin işaret ettiği gösterge tarafından işaret edilen düğümde şimdi okunan satırdan sözlük sırasına göre daha büyük bir satırın olup olmadığını kontrol ederiz. Eğer öyle ise g'nin sol, aksi takdirde sag göstergeye işaret etmesini sağlarız. **while** döngüsü g tarafından işaret edilen (sol ve sag) gösterge bir düğüme işaret ettiği (yani NULL olmadığı) sürece tekrarlanır.

Bir NULL göstergeye ulaştığımızda, g'nin bu göstergeye işaret ettiğini bilerek, döngüden çıkarız. Şimdi yeni bir düğüm oluşturma zamanıdır. malloc kütüphane fonksiyonunu kullanan yeni fonksiyonu, bu düğüm için yeterli miktarda bellek bölümü ayırmak için çağırılır. Döndürdüğü gösterge g tarafından işaret edilen göstergeye aktarılır. Daha sonra, okunmuş olan satırı alacak büyüklükte bir bellek bölgesi ayrılır; satır bu bölgeye aktarılır ve bu bölgeye olan gösterge, g'nin işaret ettiği gösterge tarafından gösterilen düğüm, yani yeni düğüm, içindeki satir göstergesine atanır. Ayrıca, bu düğümün sol ve sag göstergelerine de NULL konur.

Yukarıda anlatılanlar, ilk bakışta, çok karmaşık gelebilir. Eğer açıklamayı tam olarak izleyemediyseniz, tamamen haklısınız. Birkaç tekrar yararlı olacaktır.

Yapılacak son şey ağacı basmaktır. Burada özçağrı bize çok yardımcı olacaktır. Ana program, kok göstergesini vererek satirlaribas adlı özçağrılı fonksiyonu çağırır. Bu fonksiyon, "Eğer gösterge NULL değilse, sol altağacı bas; şimdiki düğümün bilgisini bas; sağ altağacı bas" cümlesinde denileni yapar ve ayrıca "bu düğüm tarafından kullanılan belleği serbest bırak" işini de yapar. Bellek serbest bırakıldıktan sonra, tekrar bellek ayırma söz konusu olmadığı için free fonksiyonu çağrılarının bu program için gereksiz olduğuna dikkat edin.

Programımız girdisini standart girdi aygıtından alır ve çıktısını da standart çıktı aygıtına gönderir. Daha sonraki bir bölümde göreceğimiz gibi, programlarımızda girdi/çıkıtı için dosyalar da kullanabiliriz, ama daha basit bir yol vardır. Birçok işletim sistemi, *girdi/çıkıtının yeniden yönlendirilmesi* adı verilen ve tekrar derleme yapmadan, standart girdi ve çıktı dosyaları yerine başka dosyaların kullanılmasına izin veren bir yöntem sağlarlar. MS-DOS'ta, bu, şu şekilde yapılabilir: Programınızın SIRALA adlı yürütülebilir dosya içinde derlendiğini varsayın.

A>SIRALA <SIRASIZ >SIRALI

komutunu kullanarak programın girdiyi SIRASIZ adlı disk dosyasından almasını ve çıktığı SIRALI adlı dosyaya koymasını sağlayabilirsiniz. Bu “argümanların” bir tanesini belirtmediğinizde ne olduğunu deneyip kendiniz bulmaya çalışın. Program için bu “argümanların” tamamen görünmez olduğuna dikkat edin; yani (main fonksiyonunun ikinci parametresi olan) argv dizisinde gözükmeyeceklerdir; gerekli işlemleri işletim sistemi yapar.

Başka bir özellik *küme komut işlemidir*.

A>DIR | SIRALA

şeklindeki MS-DOS komutunu deneyip çıktığı açıklayın. Daha fazla bilgi için işletim sisteminizin elkitabına danışın.

Problemler

1. İlklenmemiş “**enum** gunler” tipinden bir değişken (*a*) eğer **static** veya dışsalsa ve (*b*) eğer **auto** veya **register** ise hangi değere sahip olacaktır.
2. Kısım 5.2’de sözü edilen yapılarla ilgili sınırlamaları olan bir derleyiciniz olduğunu varsayın. Argüman olarak aynı tipten iki yapının adreslerini alıp ikinci yapıyı ilk yapıya aktaran `yapi_ata` adında bir fonksiyon yazın.
3. Yeni yapı tipleri tanımlamanın iki yöntemi vardır: yapı künyesi kullanarak ve **typedef** kullanarak. Aşağıdakini **typedef** kullanan bir bildirime dönüştürün. aylar değişkeni için yazılmış tanımını da uygun şekilde değiştirin.

```
struct ay {
    char *ayadi;
    int ayuzunlugu;
};
...
struct ay aylar[12];
```

4. İki yerde (yani kalıplarda ve **sizeof**’un işlenenleri olarak), bir veri tipi verilmesi gerekir. Bu basit bir tip ismi olabileceği gibi daha karmaşık bir ifade de olabilir. Böyle bir bağlamda aşağıdakilerin ne anlama geldiğini bulmaya çalışın:

```
int *
int *[3]
int (*) [3]
int * (float)
int (*) (void)
```

5. Altkısım 5.8.1’in başında tanımlanmış bulunan *a* dizisinin içindekileri görüntülemek için aşağıdakini deneyin:

```
#include <stdio.h>
void main (void)
{
    int i, j;
    for (i=0; i<3; i++)
        for (j=0; j<4; j++)
            printf("%2d%c", a[i,j], j==3 ? '\n' : ' ');
}
```

Neden doğru çıktıyı elde edemiyoruz? Görüntülenen sayılar nelerdir? Beklenen çıktıyı elde edecek şekilde programı değiştirin.

6. Altkısm 5.8.2'de verilen örnekte neden **sizeof(x)** 210'dur da **sizeof(a)** 2'dir?
7. Eğer MS-DOS kullanmakta iseniz, MS-DOS elkitabında SORT komutu için belirtilenleri okuyup, bu bölümün sonunda verilen örnek programı, bunları da yapacak şekilde geliştirin. Örneğin, büyük-küçük harf ayırımı yapmadan sıralama yapmayı veya sıralama yaparken belirli bir kolondan itibaren dikkate almayı sağlayın.
8. Aşağıdaki örneklerde **const** ve **volatile** anahtar sözcüklerinin ne işe yaradığını anlamaya çalışın:

```
int const      x = 11;
int            y = 22;
int * const    a = &y;
const volatile * const b = &x;
```

BÖLÜM 6: ÖNİŞLEMCİ

```
1.  #include <conio.h>
2.  #define PI 3.14159265358979323846
3.  #define DEGIS(x,y,tip) { tip t; t = x; x = y; \
4.      y = t; }
5.  #ifdef EOF
6.      # define SON EOF
7.  #else
8.      # define SON (-1)
9.  #endif
10. #ifndef MAKS
11.      # define MAKS(p,q) ((p)>(q) ? (p) : (q))
12. #endif
13. #define OZEL static
14. ...
15. #undef OZEL
16. #if (HDUZ & 0x40) != 0
17.     printf("satir = %d\n", __LINE__);
18. #endif
```

Bu bölümde hemen her zaman C derleyicisi ile birlikte kullanılan ayrı bir “işlemci”ye göz atacağız. Bu, derlenmeden önce kaynak dosyalarımızı *metin yönünden* önışlemeye tabi tuttuğu için bu şekilde adlandırılan, C önışlemcisidir. Önışlemciyi, kaynak dosyalarımızı alıp, verilen önışlemci emirlerine göre yeni bir dosya oluşturan bir düzenleyici şeklinde düşünebiliriz. Kabaca durum şöyledir:

stdio.h
benim.h → ÖNİŞLEMCİ → C DERLEYİCİSİ → amaç kod.
prog.c

Önışlemci aslında derleyici ile bütünleşmiştir, yani genelde ayrı bir sistem programı değildir.

Bir önışlemci emrini ile sıradan bir C deyimini ne ayırır? İlk olarak, farklı sözdizimi. İkinci olarak ta, *satırın başındaki* sayı işareti (#). Genelde, emirler bir satır tutarlar; ancak bir satır, sonuna bir ters bölü işareti (\) konularak, bir sonraki satıra devam ettirilebilir. Ayrıca emrin sonunda noktalı virgül *olmadığına* dikkat edin.

Emirler hiçbir zaman derleyici tarafından görünmezler. Önışlemci, emirleri çıkarıp yerlerine istenen işlemleri yapar. Örneğin, bir `#include` emri içeren satır yerine emirde belirtilen dosyanın içeriği konur. Önışlemci emirleri herhangi bir satıra konabilirler, ancak önışlemci açıklamaları atladiğı için, açıklamaların içine konulamazlar.

Bu bölümün geri kalan kısımlarında, önışlemci emirlerini ayrıntılı olarak açıklayacağız.

6.1. #define Ve #undef Emirleri

`#define` emri belirli karakter dizileri için (“makro” adı da verilen) takma isimler tanımlamaya yarar. Sözdizimi şöyledir:

```
#define tanıtıcı_sözcük değıştirme_dizisi
```

Örneğin,

```
#define BLOKBASI {
#define BLOKSONU }
#define YORDAM    static void
```

Tanımlanmakta olan *tanıtıcı_sözcük*’ün önünde ve arkasında en az bir tane boşluk karakteri bulunmalıdır. Normal C tanıtıcı sözcüklerinden ayırt etmek için, bu tür isimlerde büyük harflerin kullanılması alışlagelmıştır. Bu isimlerin tanımı `#undef` ile iptal edilmediğı sürece, kaynak dosyanın geri kalan kısmında, tanımında ondan sonra gelen karakter dizisi (yani *değıştirme_dizisi*) ile değıştirilirler. Tanımlanmış bulunan bir *tanıtıcı_sözcük* için ikinci bir `#define` ancak *değıştirme_dizisi* aynı ise kabul edilebilecektir.

Örneğin,

```
YORDAM hepsini_goster (...)
BLOKBASI
...
BLOKSONU
```

program parçası, derlenmeden önce, önışlemci tarafından

```
static void hepsini_goster (...)
{
    ...
}
```

şeklinde değıştirilecektir. Bir istisna: Eğer isim tırnak içinde yer alıyorsa, makro değışikliği yapılmayacaktır. Örneğin,

```
printf("BU YORDAM...");
```

deyimi değıştirilmeden, aynı şekilde kalacaktır.

Tipik olarak, `#define` emri program için sayı, karakter dizisi vs değişmezleri tanımlamak için kullanılır, ancak *değiştirme_dizisi*, daha önce veya sonra tanımlanan makrolar da dahil, her şeyi içerebilir.

Daha önce tanımlanmış bir makronun tanımını kaldırmak için `#undef` emrini, arkasına makro ismini belirterek, kullanabilirsiniz. (Bu bölümün başındaki ilk örnekte Satır 13 ve 14'e bakınız.) Yani

```
#undef tanıtıcı_sözcük
```

Makrolara argüman da verilebilir. (Bölümün başında Satır 3 ve 11'e bakınız.) Genel şekil şöyledir:

```
#define tanıtıcı_sözcük(tanıtıcı_sözcük_opt, ...) değiştirme_dizisi
```

Makro ismi ile “biçimsel argüman listesi” içindeki ilk parametreden önce gelen sol parantez arasında herhangi bir boşluk bulunmamalıdır. Bir fonksiyona benzemesine rağmen, bazı farklar vardır. Makrolar “satırıcı kod” oluştururlar, yani makronun kullanıldığı her sefer, biçimsel argümanlar yerine gerçek argümanlar konularak, ilgili *değiştirme_dizisi* onun yerini alır. Makro çağrısındaki argümanların sayısı, bildirimde belirtilmiş bulunan biçimsel argümanların sayısı ile aynı olmalı ve virgüllerle ayrılmalıdırlar. Tırnak işaretleri içinde veya bir çift parantez arasında bulunan virgüller gerçek argüman ayırıcıları olarak sayılmazlar.

Örneğin, Satır 11'de tanımlanmış makroya bir çağrı içeren

```
m = MAKS(x=10, c?y:z) + 5;
```

ifadesi önışlemci tarafından,

```
m = ((x=10)>(c?y:z) ? (x=10) : (c?y:z)) + 5;
```

şeklinde “açılacaktır”. Şimdi makro tanımında parametreler etrafında neden parantez kullanmış olduğumuzu görüyorsunuz. Hepsisi gereklidir. Aksi takdirde, istenen sonuç elde edilemeyecektir. Eğer inanmıyorsanız, bir de parantezler olmadan deneyin. (İşleç önceliklerini unutmayın!) Böyle durumlar için kural şöyledir:

- İlk olarak, *değiştirme_dizisini* sol parantezle başlatıp sağ parantezle bitirin. Az önceki örnekte olduğu gibi, genelde, bir makro, ifadenin bir bölümünü oluşturur. Bu aynı zamanda Satır 8'deki tanımda da gerekli idi. [Neden? “**if** (kar SON)” durumunu inceleyin.]
- İkinci olarak, *değiştirme_dizisindeki* bütün parametreleri parantez içine alın. Çünkü gerçek argümanların karmaşık ifadeler olabileceğini unutmayın.

Bir önceki örnekte olduğu gibi, dikkat ettiyseniz, makro açılımından sonra bazı hesaplamalar birden fazla kez yapılabilir. Örneğin, “`x=10`” atama deyimi. Bu durum verimsiz olması dışında, bazı beklenmeyen sonuçların çıkmasına da neden olabilir. Örneğin `i`'nin değerinin 6 olduğunu varsayın. “`MAKS(6, i++)`”nin değeri ne olacaktır? 6 mı? Bakalım. Önışlemci “`((6)>(i++) ? (6) : (i++))`” şeklinde açacaktır. Böylece değeri 7 olacaktır. Ayrıca, `i`'nin yeni değeri 7 değil de 8 olacaktır.

Eğer *değiştirme_dizisinde* bir parametrenin önüne # konursa, makro açılımından sonra çift tırnak içine alınacaktır, yani argüman bir karakter dizisine dönüştürülecektir. Ayrıca böyle bir argümanın içinde bulunan " ve \ karakterlerinin önüne \ eklenecektir.

Eğer *değiştirme_dizisi* içinde bir ## simgesi varsa, bu işlecin önünde ve arkasında bulunan isimlerin birbirine yapıştırılacağı şekilde, ## ve etrafındaki her türlü beyaz boşluk karakteri çıkarılacaktır. Örneğin

```
#define cagir(x) yord ## x()
```

tanımını ele alalım. Bu makroyu

```
cagir(1);
```

şeklinde kullandığımızda

```
yord1();
```

şeklinde açılacaktır.

6.2. #include Emri

```
#include "dosya_adi"
```

```
#include <dosya_adi>
```

veya

```
#include karakter_dizisi
```

şeklindeki bir satır, önışlemcinin bu satır yerine belirtilen dosyanın içindekilerini koymasını sağlar. Dosyanın içinde uygun C kodunun bulunması gerektiği açıktır. İlk şekil, önışlemcinin *dosya_adi* için önce *varsayılan altdizine*, yani derlenmekte olan kaynak dosyanın bulunduğu altdizine, daha sonra “standart yerler”e bakmasını sağlayacaktır. İkinci şekil sadece standart yerlerde aramaya neden olur. “Standart yerler” derken, bu tür dosyaların bulunduğu, önışlemci tarafından bilinen, altdizinlerden söz etmekteyiz. *Dosya_adına* bazen *başlık dosyası* adı da verilir. Üçüncü şekilde ise, önışlemci tarafından #define ile tanımlanmış *karakter_dizisi* açıldığında ilk iki şekilden birisinin elde edilmesi beklenir.

Sistemimizde, standart yerler, “INCLUDE” adındaki MS-DOS ortam değişkeninde belirtilir. Örneğin,

```
SET INCLUDE=a:\ktphnm;a:\include
```

MS-DOS komutu, önışlemcinin dosyayı bulması için, belirtilen iki altdizine bakmasını sağlayacaktır. Eğer #include’un çift tırnaklı olanı kullanılırsa, önce varsayılan altdizine bakılacaktır. Ayrıca, isterseniz #include emrindeki *dosya_adında* dosya isminin önüne altdizin ismini de belirtebilirsiniz.

Son bir nokta: `#include` emirleri (8 düzeye kadar) içiçe yazılabilir; bu, `#include` edilen bir dosyanın içinde başka `#include` emirleri de bulunabilir anlamına gelir.

6.3. Koşullu Derleme

`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` ve `#endif` emirleri, bazı koşullara bağlı olarak derlemeyi kontrol etmek için kullanılırlar.

`#if değişmez_ifade`

emri, eğer *değişmez_ifadenin* değeri doğru (sıfırdan farklı) ise bir sonraki `#endif` veya `#else` (veya `#elif`) emrine kadar olan satırların derlemeye dahil edilmesini sağlar. İlk durumda, derleme `#endif`'ten sonra, normal olarak devam eder; ikinci durumda ise, `#else`'den sonra *ilgili* `#endif`'e kadar olan satırlar önışlemci tarafından atlanır (yani derleyiciye gönderilmez). Eğer *değişmez_ifadeni* değeri yanlış (0L) ise, `#if` ile `#endif` veya `#else` (hangisi önce geliyorsa) arasındaki metin atlanır. `#if` emirlerinin içiçe yazılabildiğine dikkat edin, bunun için hangi `#else`'in hangi `#if`'e ait olduğunu bilmemiz gerekir: `#else` bir `#endif` ile bağlantısı olmayan, yukarıdaki en yakın `#if`'e aittir. *Değişmez_ifade*, önışlemcinin “önışlem zamanında” hesaplayabileceği herhangi bir C ifadesidir; **sizeof** ve kalıp işleçleriyle sayım ve kayan noktalı sayı *değişmezleri* burada kullanılamazlar. Daha önce önışlemciye tanımlanmış makrolar ise kullanılabilirler; ancak, tanımsız isimlerin yerine 0L konur. *Değişmez_ifade* her zaman **long** duyarlıkta hesaplanır.

Bir emir satırında

`defined tanıtıcı_sözcük`

şeklindeki bir ifade, eğer *tanıtıcı_sözcük* önışlemci tarafından biliniyorsa 1L ile değiştirilir, aksi takdirde 0L konur.

Bir örnek olarak, iki müşteri için hazırlanmış bir programı ele alalım. Program içinde belirli bir müşteri için gerekli bölümler olabilir. Derlemeyi kontrol etmek için, şu anki derlemenin hangi müşteri için yapıldığını gösteren bir tanıtıcı sözcük tanımlayabiliriz. Örneğin,

```
#define MUSTERI 1 /* diger must. icin baska bir deger */
...
#if MUSTERI == 1
/* ilk musteri icin kod */
...
#else
/* ikinci musteri icin kod */
...
#endif
```

Bu arada

```
#ifdef tanıtıcı_sözcük
```

ve

```
#ifndef tanıtıcı_sözcük
```

emirleri, sırasıyla, *tanıtıcı_sözcük*'ün önışlemci tarafından #define ile tanımlanmış olup olmadığını test ederler. Bunlar, sırasıyla,

```
#if defined tanıtıcı_sözcük
```

ve

```
#if !defined tanıtıcı_sözcük
```

emirleriyle eşdeğerdirler.

```
#elif değişmez_ifade
```

emri

```
#else
```

```
#if değişmez_ifade
```

için bir kısaltma şeklinde düşünülebilir, ancak #elif kullanıldığında yeni bir #endif'e gerek kalmaz; önceki #if'in #endif'i kullanılır.

Başka bir örnek olarak, hataların düzeltilmekte olduğu bir program düşünün; böylece bu programda *htkntrl* adlı bir fonksiyona bazı yerlerde çağrılar olmaktadır. (Bu fonksiyon, örneğin, bazı değişkenlerin o anki değerlerini basıyor olabilir.) Ancak, program geliştirilmesi sonunda programı “normal” bir şekilde derlememiz gerekecektir. Bunu yapmanın kolay bir yolu da şöyledir:

```
...
#ifdef HATASIZ
    /* hata düzeltme yardımcı fonksiyonu */
    /* içeren başlık dosyası */
    # include "htkntrl.h"
#else
    # define htkntrl()
#endif
...
```

Eğer HATASIZ adlı tanıtıcı sözcük #define ile tanımlanmamışsa, o zaman #include emri önışlemeye girecek ve *htkntrl* *fonksiyonu* da dahil, hata düzeltme yardımcı programları içeren “*htkntrl.h*” dosyasının içindekileri derlenecektir. Ancak, eğer HATASIZ #define ile tanımlanırsa, o zaman, boş bir değiştirme dizisine sahip olan *htkntrl* *makrosu* #define ile tanımlanmış olacaktır. Bu durumda, *htkntrl*'e yapılan çağrılar, önışlemci tarafından, boşlukla değiştirilecektir. Buna benzer #if emirleri programın başka bölümlerinde de konulabilir. HATASIZ tanıtıcı sözcüğü programın başında tanımlanabileceği gibi, UNIX veya sistemimizde olduğu gibi C (önışlemcisi ve) derleyicisinin çağrıldığı komut satırında belirtilebilir. Ek B'ye bakınız.

6.4. Diğer Emirler

Pek kullanılmayan bir emir de aşağıdakidir:

```
#line değişmez "dosya_adi"opt
```

Burada *değişmez*, bir *değişmez* ifade değil, 1 ile 32 767 arasında bir *tamsayı* *değişmez*dir. *Dosya_adi* ise bir karakter dizisidir. Bu emir, derleyicinin şu anki satır sayısını unutup derlemekte olduğu satırın sayısının sanki *değişmez* ile ifade edilen sayı imiş gibi davranmasını sağlar. Yani emirden sonraki beşinci satırda bir hata olursa “*değişmez+4 nolu satırda hata var*” şeklinde bir hata mesajı ortaya çıkacaktır. İsteğe bağlı olan ikinci argüman ise, derleyicinin şu anda derlenmekte olan dosyanın adının sanki *dosya_adi* imiş gibi bir varsayımda bulunmasına neden olur. *Dosya_adn*ın içinde altdizin de belirtilmişse, bu *#include* dosyaları için taranacak olan *varsayılan altdizini* de etkileyebilir. (Kısım 6.2’deki tartışmaya bakınız.)

```
#error karakter_dizisi
```

şeklindeki bir satır, önışlemcinin *karakter_dizisi*ni de içeren bir mesaj yazmasını sağlayacaktır.

```
#pragma karakter_dizisi
```

şeklindeki bir önışlemci emri, önışlemci ve/veya derleyicinin *karakter_dizisi* içinde belirtilen bir işlem yapmasına neden olacaktır. *Karakter_dizisi* için kabul edilebilecek değerler C Standardında belirtilmemiştir. Onun için bu tamamen uygulamaya bağlı bir özelliktir ve dikkate alınmayabilir.

En son olarak; görünen tek karakteri # olan bir satır dikkate alınmaz ve atlanır.

6.5. Önceden Tanımlanmış İsimler

Önışlemci, ayrıca bazı tanıttıcı sözcükleri önceden *#define* ile tanımlamıştır. Bunların tanımının *#undef* ile kaldırılması veya *#define* ile yeniden tanımlanmaları mümkün değildir. Bunlar şunlardır:

```
__FILE__  
__LINE__  
__DATE__  
__TIME__  
__STDC__
```

Derlenmekte olan dosyanın adı (karakter dizisi),
Şu anki kaynak satır numarası (ondalık sayı),
Tarih (“Aaa gg yyyy” şeklinde karakter dizisi),
Saat (“ss:dd:ss” şeklinde karakter dizisi),
Standarda uyan derlemeler için 1 *değişmezi*.

6.6. Bir Örnek—ctype.h Başlık Dosyası

Aşağıda ctype.h başlık dosyasının basitleştirilmiş ve Türkçeleştirilmiş bir uyarlaması verilmektedir:

```

1.  /**
2.  *ctype.h - karakter donusum makrolari ve ctype makrolari
3.  *
4.  * Copyright (c) 1985-1992, Microsoft Corp. Her hakkı saklidir.
5.  *
6.  *Amac:
7.  * Karakter siniflandirmasi ve donusumu icin makrolar tanimlar.
8.  * [ANSI/Sistem V]
9.  *
10. ****/
11.
12. #ifndef _INC_CTYPE
13.
14. /* ... */
15.
16. /* Asagidaki tanim, kullanicinin, ctype.obj'da
17.  * tanimlanmis bulunan _ctype adli karakter tipi
18.  * dizisine erisimini saglar.
19.  */
20.
21. extern unsigned char /* ... */ _ctype[];
22.
23. /* olasi karakter tipleri icin bit ortulerinin tanimlanmasi */
24.
25. #define _UPPER 0x1 /* buyuk harf */
26. #define _LOWER 0x2 /* kucuk harf */
27. #define _DIGIT 0x4 /* rakam[0-9] */
28. #define _SPACE 0x8 /* durak, satir basi, yeni satir, */
29. /* dikey durak veya sayfa ilerletme */
30. #define _PUNCT 0x10 /* noktalama karakteri */
31. #define _CONTROL 0x20 /* kontrol karakteri */
32. #define _BLANK 0x40 /* bosluk karakteri */
33. #define _HEX 0x80 /* onaltili rakam */
34.
35. /* ... */
36.
37. /* karakter siniflandirilmesi ile ilgili makro tanimlari */
38.
39. #define isalpha(_c) ( (_ctype+1)[_c] & (_UPPER|_LOWER) )
40. #define isupper(_c) ( (_ctype+1)[_c] & _UPPER )
41. #define islower(_c) ( (_ctype+1)[_c] & _LOWER )
42. #define isdigit(_c) ( (_ctype+1)[_c] & _DIGIT )
43. #define isxdigit(_c) ( (_ctype+1)[_c] & _HEX )
44. #define isspace(_c) ( (_ctype+1)[_c] & _SPACE )
45. #define ispunct(_c) ( (_ctype+1)[_c] & _PUNCT )
46. #define isalnum(_c) ( (_ctype+1)[_c] & (_UPPER|_LOWER|_DIGIT) )
47. #define isprint(_c) ( (_ctype+1)[_c] & (_BLANK|_PUNCT|_UPPER|_LOWER|_DIGIT) )
48. #define isgraph(_c) ( (_ctype+1)[_c] & (_PUNCT|_UPPER|_LOWER|_DIGIT) )
49. #define iscntrl(_c) ( (_ctype+1)[_c] & _CONTROL )
50.
51. #ifndef _STDC_
52. #define toupper(_c) ( (islower(_c)) ? _toupper(_c) : (_c) )

```

```

54. #define tolower(_c) ( (isupper(_c)) ? _tolower(_c) : (_c) )
55. #endif
56. #define _tolower(_c) ( (_c) - 'A' + 'a' )
57. #define _toupper(_c) ( (_c) - 'a' + 'A' )
58. #define __isascii(_c) ( (unsigned) (_c) < 0x80 )
59. #define __toascii(_c) ( (_c) & 0x7f )
60.
61. /* genisletilmis ctype makrolari */
62.
63. #define __iscsymf(_c) (isalpha(_c) || ((_c) == '_'))
64. #define __iscsym(_c) (isalnum(_c) || ((_c) == '_'))
65.
66. /* ... */
67.
68. #define _INC_TYPE
69. #endif

```

Bu dosyanın başında (`_ctype`) **extern** dizisinin bildirimi yapılmaktadır. Bu 129 elemanlık bir dizidir. İlk elemanın değeri 0'dır, böylece ikinci elemandan başlamamız gerekir. İlk elemanı atlamak için, dizi şu şekilde indislenebilir:

```
(_ctype+1)[indis]
```

Her eleman, indis olarak kullanılan karaktere karşılık gelen bir kod içerir. Örneğin:

```
(_ctype+1) [' ']' in değeri 0x48'dir
(_ctype+1) ['A'] in değeri 0x81'dir
```

Bu sayılar, herbirinin özel bir anlamı olan, 8 bitten oluşmaktadır. Bu bitlerin anlamları 25'ten 33'e kadar olan satırlarda verilmiştir. Örneğin, 0x48 `_BLANK` (boşluk) ve `_SPACE` (görünmeyen karakter) anlamına gelir, 0x81 `_HEX` (onaltılı rakam) ve `_UPPER` (büyük harf) demektir. Bu bilgi ile, “is...” (...dır) makrolarının nasıl çalıştığını açıklamak zor olmayacaktır. `__isascii` (ASCII karakteri) olup olmadığını test eder.

Diğer makroların açıklaması kolaydır ve okuyucuya bırakılmıştır.

Problemler

1. Yeni tipler tanımlamak için en az iki yolunuz vardır. *Bir*, önışlem esnasında `#define` kullanarak; *iki*, derleme sırasında **`typedef`** anahtar sözcüğünü kullanarak. `#define`'in **`typedef`** yerine kullanılamayacağı bir örnek verin.
2. Makrolar, karakter dizileri, yani çift tırnak içinde ise açılmazlar. Ancak makro parametreleri, makro tanımında tırnak içinde olsalar bile değiştirilebilirler. Standart açıkça bunu yasaklamaktadır ve bizim önışlemcimiz de bunu yapmamaktadır. Sizin için de aynı şeyi yapıp yapmadığını deneyerek görün ve bu “özellik” kullanan yararlı bir örnek verin.
3. `_ctype` dizisini kullanmadan `ctype.h` dosyasında tanımlanmış bulunan “is...” (...dır) makrolarına eşdeğer uyarlamalar tanımlayın. Bir örnek aşağıda verilmiştir:

```
#define isalpha(_c) (((_c)>='A' && (_c)<='Z') || \
    ((_c)>='a' && (_c)<='z'))
```

4. Anahtar sözcükleri Türkçe, Almanca, İtalyanca veya başka bir dilde olan C programları yazmanızı sağlayacak #define'lar yazın.

BÖLÜM 7: DOSYALAR VE GİRDİ/ÇIKTI

Alıştığımız değişkenlerden farklı olarak, bir *dosya*, programın yürütülmesinden önce ve/veya sonra da var olan bir nesnedir. “Kalıcı olma” özelliği dosyaları çok yararlı kılar. Şimdiye kadar gördüğümüz programlarda olduğu gibi, dosyalar sayesinde, programlar kullanıcılarıyla veya birbirleriyle haberleşebilirler. C dilinde dosyalar, istenildiği kadar uzun olabilen karakter (bayt) sıraları şeklinde görülürler. Ancak daha karmaşık bir yapı, kullanıcı tarafından, uygulamaya göre, verilebilir. Bu bölümün büyük bir kısmı, dosyalarla ilgili çeşitli işlemler ve girdi/çıkıı yapmak için bize yardımcı olacak standart kütüphaneyi anlatacaktır. Bu bölümde anlatılan fonksiyonlar, birçok C ortamında sağlananların sadece küçük bir kısmıdır. Daha fazla bilgi için ilgili başvuru elkitablarına danışın.

Kullanıcı programları, *işletim sistemi* adı verilen, karmaşık bir sistem programının gözetimi altında çalıştırılırlar. Programların işletim sistemine çeşitli işlemler yaptırması için, C kütüphanesi bazı yordamlar içerir. C farklı sistemlerde uygulandığı için kütüphanenin bu bölümü pek standart değildir. Bu bölümde tartışılacak başka bir konu da, bizim sistemin C ile nasıl etkileştiğidir.

Kısım 7.1’den 7.4’e kadar anlatılan fonksiyonlar Standardın bir parçasıdır ve `stdio.h` başlık dosyasında bildirilmiştir. Kısım 7.5’teki fonksiyonların bildirimi `stdlib.h` standart başlık dosyasında bulunmaktadır. Ancak Kısım 7.6’dakiler Standardın bir parçası değildir, buna rağmen birçok ortamda desteklenmektedir.

7.1. Dosya Esasları

Program bir dosya üzerinde herhangi bir işlem yapmadan önce, dosyanın açılması gerekmektedir. `fopen` standart fonksiyonu bunu yapar ve iki karakter dizisi argüman alır: açılacak olan dosyanın adı ve erişim modu. İlk argüman herhangi geçerli bir dosya

adı içerebilir; dosya adının önüne sürücü ve alt dizinler de belirtilebilir. İkinci argümanın alabileceği değerler ve anlamları aşağıdaki çizelgede gösterilmiştir:

<u>değer</u>	<u>erişim modu</u>
"r"	okuma
"w"	yeniden yazma
"a"	sona ekleme
"r+"	okuma ve yazma (değiştirme)
"w+"	okuma ve yeniden yazma (değiştirme)
"a+"	okuma ve ekleme

Varolmayan bir dosyadan okumaya kalkmak hatadır. Yeni bir dosya yaratmanın yolu varolmayan bir dosya adı verip "a..." veya "w..." belirtmektir. Eğer erişim modunun sonuna b harfi eklenirse—örneğin "rb"—ikili (ham) modda girdi/çıkıtı yapılacaktır, aksi takdirde metin girdi/çıkıtısı yapılır. Eğer açma işlemi başarılı ise fopen açtığı dosyaya bir *dosya göstergesi* döndürür, yoksa, standart kütüphanede de bulunan, NULL yani boş gösterge döndürür.

Dosya göstergeleri şu şekilde tanımlanmalıdır:

```
FILE * dosya_gostergesi;
```

FILE bütün dosyaların tipidir; kütüphanede tanımlı bulunan bir yapıdır ve bu tanımın detayları normal kullanıcıları ilgilendirmez. Gelin bir dosya oluşturalım:

```
dosya_gostergesi = fopen("DOSYA.YEN", "w")
```

Bundan sonra, DOSYA.YEN ile ilgili işlemlerde dosya_gostergesi kullanılacaktır.

Bir program ekrana bilgi görüntülediği zaman, aslında standart çıktı dosyasına karakter yazmaktadır. Programınıza klavyeden bilgi girdiğinizde, aslında bunları standart girdi dosyasına yazmaktasınız ve program bu dosyayı okumaktadır. “Standart dosyalar” programcı tarafından fopen kullanılarak *açılmazlar*, yürütme başlamadan önce, otomatik olarak açılıp, normalde, kullandığınız ekran ve klavyeye atanırlar. Kütüphane, bu iki dosya için de dosya göstergeleri içermektedir: standart girdi dosyası için stdin, standart çıktı dosyası için de stdout.

Bazı durumlarda, hata mesajları için ayrı bir çıktı akışının olması uygun olabilir. C bir standart hata dosyası sağlamaktadır; bu dosyaya olan göstergenin adı stderr'dir. Normalde, stderr'e giden çıktı stdout'ınkiyle birlikte aynı aygıtta yapılır, ancak bunu değiştirmenin yolları vardır. stdin, stdout ve stderr *değişmez* göstergelerdir; onları kullanırken bunu unutmayın.

Şimdi, bir dosyayı nasıl açacağımızı bildiğimize göre, gelin ona nasıl yazacağımıza bakalım. dosya_gostergesi tarafından işaret edilen dosyaya kar karakterini yazmak için putchar fonksiyonunu kullanınız:

```
putc(kar, dosya_gostergesi)
```


`putc` aslında karakterleri bir tampon bölgeye koymakta ve bu bölge dolduğunda içindekileri dosyaya aktarmaktadır. Eğer dosyaya, tamponun boyundan daha az miktarda, birkaç karakter yazmak isterseniz ne olacaktır? Tamponu “boşaltmak” için bir yol gereklidir. `fclose` fonksiyonu, diğer işlerin yanında bunu yapmaktadır. `fclose`’un, dosya göstergesi olan, bir argümanı vardır.

```
fclose(dosya_gostergesi)
```

çalıştırıldıktan sonra, dosya kapatılır ve `dosya_gostergesi` artık ona işaret etmez. `fclose` `fopen`’in “tersi”dir. `fopen` ile açtığınız her dosyayı `fclose` ile kapatmanız gerekir. Sistemden sisteme fark etmekle beraber, herhangi bir zamanda açık olan dosya sayısında bir sınırlama olduğu için, bu ayrıca gerekli de olabilir. Program normal olarak sona erdiğinde, bütün açık dosyalar `fclose` ile kapatılır.

Okuma için açılmış ve `dosya_gostergesi` tarafından işaret edilen bir dosyadan bir karakter okumak için `getc` kullanılır:

```
kar = getc(dosya_gostergesi)
```

Eğer okuyacağı karakter kalmazsa `getc` ne yapar? Dosya sonuna ulaştığını göstermek için özel bir değer döndürmesi gerektiği açıktır. Bu değer bir karakter *olmamalıdır*, çünkü aksi takdirde, `getc`’nin dosyadan bir karakter mi okuduğunu, yoksa dosya sonunu mu işaret etmek istediğini anılamayacağız. Böyle bir durumda, `getc`, EOF değişimini döndürür. Kütüphane, genelde EOF’u (-1) olarak tanımlar. Bu değeri de tutmak için, yukarıda kullanılan `kar` “karakter”nin bir **char** değil de, bir **int** olarak tanımlanması gerektiğine dikkat edin. Böylece, her seferinde dosyadan bir karakter okuyup, o karakter için bir şey yapan ve dosyada karakter kalmadığı zaman sona eren bir program parçası şu şekilde olacaktır:

```
if ((kar=getc(dosya_gostergesi))!=EOF) {
    kar'a göre işlem yap
} else
    programı bitir
```

7.2. Dosya Erişimi—Başka Yöntemler

`fopen`, `fclose`, `putc` ve `getc` ile çok yararlı şeyler yapılabilir, fakat C, kolaylık için, başka, “daha gelişmiş” dosya erişim yöntemleri de sağlar. Zaten bildiğiniz, `getchar` ve `putchar` böyle iki örnektir. Sistemimizde, `stdio.h` dosyasında şu şekilde tanımlanmışlardır:

```
#define getchar()    getc(stdin)
#define putchar(c)   putc((c), stdout)
```

Doğal olarak, programcılar, bir seferde birden fazla karakter yazabilen fonksiyonlara gereksinim duyarlar. İşte bunu yapan `fputs` fonksiyonu:

```
fputs(kar_dizisi, dosya_gostergesi)
```

Bu, `kar_dizisi` adlı karakter dizisinin `dosya_gostergesi` tarafından işaret edilen dosyaya yazılmasını sağlar. Dizinin sonundaki boş karakter yazılmaz. Bu fonksiyonun da `puts` adında “standart dosya” uyarlaması bulunmaktadır.

```
puts(kar_dizisi)
```

`kar_dizisi`’ni `stdout`’a yazar ve `fputs`’tan farklı olarak yeni satır karakteri yazıp yeni bir satıra geçer.

Kütüphanede, ayrıca, bir seferde bir satır okuyan `fgets` adında bir fonksiyon bulunmaktadır.

```
fgets(kar_dizisi, maks_uzunluk, dosya_gostergesi)
```

`dosya_gostergesi` tarafından işaret edilen dosyanın bir sonraki satırından en fazla `maks_uzunluk-1` sayıda karakteri `kar_dizisi` içine okuyup, eğer dosya sonu değilse, `kar_dizisi`’ni (bir gösterge), aksi takdirde `NULL` döndürür. (Bir *satır* sadece sonunda yeni satır karakterinin bulunduğu bir karakter sırasındır.) `kar_dizisi` içine aktarılan satırın sonuna boş karakter eklenir.

Standart girdi fonksiyonu `gets` biraz `fgets`’e benzer.

```
gets(kar_dizisi)
```

`stdin`’den bir satır okuyup `kar_dizisi` ile başlayan yere koyar. `fgets`’ten farklı olarak, `gets` `kar_dizisi`’ne yeni satır karakterini koymaz.

Kolaylık isteyen programcılar, karakter dizisi-sayı dönüşümlerinin fonksiyon tarafından yerine getirildiği, herhangi bir tipten sayıların dosyaya yazılmasını veya oradan okunmasını isterler. Bu gereksinimler, `fprintf` ve `fscanf` fonksiyonları tarafından yerine getirilirler.

```
fprintf(dosya_gostergesi, kontrol_karakter_dizisi, argüman_listesi)
```

çağrısı `kontrol_karakter_dizisi` içindeki sıradan karakterlerle, `argüman_listesi`’nden aldığı argümanları uygun bir şekilde biçimlendirerek `dosya_gostergesi` tarafından işaret edilen dosyaya yazar. `Kontrol_karakter_dizisi` ve `argüman_listesi` için kurallar `printf`’teki gibidir.

```
fscanf(dosya_gostergesi, kontrol_karakter_dizisi, argüman_listesi)
```

çağrısı `dosya_gostergesi` tarafından işaret edilen dosyadan bir karakter dizisi okur, `kontrol_karakter_dizisine` göre dönüşümleri yapıp, değerleri, `argüman_listesindeki` argümanlar tarafından gösterilen yerlere koyar. Kurallar `scanf`’teki gibidir. Argümanlar için doğru biçimleri ve doğru sayıda argüman sayısını vermek kullanıcının sorumluluğundadır.

`printf` ve `scanf` fonksiyonları `fprintf` ve `fscanf`’in özel uyarlamalarıdır. İlk argüman olarak dosya göstergesi yoktur; bunlar, sırasıyla, `stdout` ve `stdin`’i kullanırlar.

Bu fonksiyon ailesinin başka iki üyesi de `sprintf` ve `sscanf`'tir. Bunlar dosyaya erişmezler; onun yerine, ilk argüman olarak belirtilen bir karakter dizisi üstünde çalışırlar.

`sprintf(kar_dizisi, kontrol_karakter_dizisi, argüman_listesi)`

çağrısı, tıpkı `fprintf` gibi, `kontrol_karakter_dizisi` ve `argüman_listesi` ile normal biçim dönüşümlerini yapar; sonucu `fprintf`'ten farklı olarak `kar_dizisi`'ne koyar. Buna benzer olarak,

`sscanf(kar_dizisi, kontrol_karakter_dizisi, argüman_listesi)`

çağrısı, `scanf`'in `stdin`'i taradığı şekilde `kar_dizisi`'nin taranıp değerlerin, normalde olduğu gibi, `argüman_listesi`'nin elemanları tarafından gösterilen yerlere konmasını sağlar. Örneğin

`sprintf(kar_dizisi, "abc%dcba", 100);`

deyimi `kar_dizisi`'nin içine "abc100cba" koyar; eğer ondan sonra

`sscanf(kar_dizisi, "abc%d", &int_deg);`

deyimi işletilirse, beklendiği gibi `int_deg`'in içine 100 konur.

7.3. Rastgele Erişim

Normalde, bir dosya açıldıktan sonra, ilk okuma veya yazma işlemi dosyanın hemen başından yapılır. Bir sonraki işlem ise, en son işlem tarafından etkilenen son baytı izleyen bayttan başlayarak yapılır. Bundan dolayı, buna *sıralı erişim* denir. Bunu yapmadan, yani önceki baytları okuyup veya yazmadan, istediğimiz bir bayta ulaşmak (yani *rastgele erişim* yapmak) için `fseek` kullanırız:

`fseek(dosya_gostergesi, uzun_sayi, nereden)`

Yukarıdaki çağrıda ilk iki argümanın tipleri isimlerinden anlaşılabilir. `nereden` ise ya `SEEK_SET` (dosyanın başından), ya `SEEK_CUR` (şu anki konumundan) ya da `SEEK_END` (dosyanın sonundan) olabilir. `fseek` işletildikten sonra `dosya_gostergesi` tarafından işaret edilen dosya üzerindeki okuma veya yazma `nereden` ile belirtilmiş yerden `uzun_sayi` mesafedeki bayttan başlayacaktır. Eğer herhangi bir hata olursa, `fseek` sıfırdan farklı bir sayı döndürür, aksi takdirde 0 verir. Bir *metin* dosyasında bir karakterin konumunun hesaplanmasının anlamlı yada en azından taşınabilir bir özellik olmadığına dikkat edin; `fseek` tamamen *ikili* (ham) girdi/çıkı içindir.

Eğer, birtakım işlemlerden sonra, dosyanın başından kaç bayt ileride olduğunuzu öğrenmek isterseniz `ftell` kullanın:

`uzun_sayi = ftell(dosya_gostergesi)`

`uzun_sayi`'ya miktarı koyacaktır; eğer bir hata olursa -1L verecektir.

Sık sık dosyayı “geri sarmak”, yani başına `fseek` yapmak isteyebilirsiniz.

```
rewind(dosya_gostergesi)
```

bunu yapmanın kısa bir yoludur.

7.4. Dosyalarla İlgili Başka Bilgiler

“Okuma” fonksiyonları veya makrolarının okuduğu şey `ungetc` tarafından etkilenebilir.

```
ungetc(kar, dosya_gostergesi)
```

`çarısı`, `dosya_gostergesi` ile işaret edilen dosyadan yapılacak bir sonraki okuma işlemi (`getc`, `fgetc`, `fscanf` vs) tarafından dosyada var olan karakterlerden önce `kar`’da verilen karakterin okunmasını sağlar. `ungetc` ile okuma işlemi arasında `fseek` yapılırsa, `ungetc` ile saklanan karakter unutulacaktır. `ungetc` isminin aslında biraz yanıltıcı olmasına rağmen, `ungetc` genelde okunmaması gerekirken okunmuş bir karakterin sanki okunmamış gibi geri alınmasını sağlar. Her dosya için, en fazla bir karakter bu şekilde geri konabilir. `ungetc` işlemi başarısız olursa EOF döndürülür.

Dosyaları oluşturduğunuz gibi onları silebilirsiniz de. Dosyanın ismi `dosya_adi` ise, o zaman

```
remove(dosya_adi)
```

yazın. Dosya bu altdizinden silinmiş olacaktır. Eğer dosyanın ait olduğu tek altdizin bu ise dosya sistemden tamamen silinecektir. (Bir *altdizin* başka altdizin veya dosya isimleri içerebilen bir rehber olarak düşünülebilir.) Eğer işlem başarılı ise `remove` 0 döndürecektir.

Buna benzer olarak bir dosyanın adını değiştirmek istiyorsanız.

```
rename(eski_dosya_adi, yeni_dosya_adi)
```

kullanın.

Bilgi öbekleri okumak veya yazmak için iki yararlı fonksiyon vardır:

```
fread(gosterge, boy, sayi, dosya_gostergesi)
```

ve

```
fwrite(gosterge, boy, sayi, dosya_gostergesi)
```

`fread`, `dosya_gostergesi` tarafından işaret edilen dosyadan `gosterge` ile işaret edilen diziye `boy` uzunluğunda `sayi` kadar nesne aktaracaktır. `fwrite` ise bu işlemi ters yönde yapacaktır. Bu fonksiyonlar aktarılan nesne sayısını döndürürlər; eğer bir hata olmuşsa, bu sayı `sayi`’dan az olacaktır.

Çok girdi/çıkı, örneğin bir aktarma işlemi yapan bir program dosyalar için büyük tampon bölgeleri kullanırsa kazançlı çıkacaktır. Tamponlu girdi/çıkı yapmak için standart kütüphane iki fonksiyon sağlamaktadır. Bunlar `fopen`’dan hemen sonra çağrılmalıdır.

```
setbuf(dosya_gostergesi, tampon)
```

fonksiyonu, normalde

```
char tampon [BUFSIZ];
```

olarak tanımlanması gereken tampon dizisini dosya için tampon bölge haline getirir. BUFSIZ stdio.h dosyasında tanımlanmış olan uygun bir değişmezdir. Eğer tampon yerine NULL verirsek, tamponlama ortadan kalkar.

Tampon boyunun kullanıcı tarafından belirlendiği daha iyi bir fonksiyon ise şöyledir:

```
setvbuf(dosya_gostergesi, tampon, mod, tampon_boyu)
```

Burada, mod ya _IOFBF (tam tamponlama), ya _IOLBF (metin dosyaları için satır tamponlama) yada _IONBF (tamponlamanın kaldırılması) olabilir. Eğer tampon NULL ise tampon_boyu boyundan bir tampon bölgesi setvbuf tarafından ayrılacaktır. İşlem başarılı ise, bu fonksiyon 0 döndürür.

Son olarak,

```
fflush(dosya_gostergesi)
```

fonksiyonu, tamponda bilgi varsa, bunun dosyaya aktarılmasını sağlar. Sadece bir çıktı dosyası için kullanılabilir ve bir hata durumunda EOF, aksi durumda 0 döndürür.

7.5. Sistem İle İlgili Fonksiyonlar

İlginç bir fonksiyon olan system, bir C programı içinden, işletim sisteminizin kabul edebileceği herhangi bir komutu çalıştırmanızı sağlar.

```
system(kar_dizisi)
```

çağrısı yapıldığında, kar dizisi içindeki işletim sistemi komutu çalıştırılır; bu komut tamamlandıktan sonra ise program normal bir şekilde devam eder. Örneğin,

```
system("DIR C:\\MSVC\\INCLUDE\\*.H")
```

ifadesi c:\msvc\include alt dizisini içinde dosya tipi h olan bütün dosyaların ekranda görüntülenmesini sağlar.

Şimdiye kadar görmüş olduğumuz programlar (herhangi bir sorun çıkmazsa), kontrol main bloğunun sonuna ulaştığında biterler. exit fonksiyonu ise, konulduğu yerde programın sona ermesini sağlar.

```
exit(durum)
```

çağrısı, açık olan tüm dosyalar için fclose'un, daha sonra da _exit kütüphane fonksiyonunun çağrılmasını sağlar. _exit başka bir şey yapmadan programı bitirir. (Programlarınızda kütüphanede tanımlanmış bulunan ve "_" ile başlayan bir şey kullanmanız tavsiye edilmez; normal programcılar için bunlar biraz düşük düzeyde kalırlar ve Standarda girmezler.) Bir tamsayı olan durum'un değeri, bu programın

çalışmasını sağlayan sürece geçirilir ve bu süreç tarafından bazı kararlar vermek için kullanılabilir. Geleneksel olarak, sıfırdan farklı durum değerleri çeşitli hatalı durumları gösterirler, 0 ise başarılı bir sonuçlanmayı gösterir. `stdlib.h` başlık dosyasında tanımlanmış `EXIT_SUCCESS` (başarılı sonuçlanma) ve `EXIT_FAILURE` (hatalı sonuçlanma) değerleri bu kodları standartlaştırmaktadırlar.

Birbirleriyle ilişkili iki fonksiyon da şunlardır:

```
rand()
```

ve

```
srand(tohum)
```

İlki 0 ile `RAND_MAX` (≥ 32767) arasında bir sözde rastgele tamsayı verir. İkincisi ise, `rand` tarafından üretilecek yeni bir sözde rastgele sayı sırası için tohumu (işaretsiz bir tamsayı) belirtmek için kullanılır. Eğer `srand` çağrılmazsa, program başlarken tohum'un değeri 1'dir.

7.6. Dosya Tanımlayıcıları Ve İlgili Fonksiyonlar

Dosya göstergelerine bir alternatif de dosya tanımlayıcılarıdır. Her *dosya tanımlayıcısı* (sistemimizde *tutamak* da denir), programın yürütülmesi esnasında dosyayı gösteren, sıfırdan küçük olmayan bir tamsayıdır. Kısım 7.1'den 7.4'e kadar anlatılan dosya erişim fonksiyonlarının dosya tanımlayıcıları için ayrı uyarlamaları bulunmaktadır.

Dikkat: Önceki kısımdakilerden farklı olarak, bu kısımda anlatılan fonksiyonlar standartlaşmış değildir; sadece bizim sistem tarafından bunların desteklendiğini söyleyebiliriz. Bundan başka, dosya tanımlayıcısı fonksiyonları ile dosya gösterge fonksiyonları birbirinden tamamen farklı ve uyumsuzdurlar; bir program içinde ikisi birlikte *kullanılmamalıdır*. Aslında, bu kısımda anlatılan fonksiyonları hiç kullanmamak en iyi yoldur! Bu kısımda anlatılanlarla ilgili bilgiler, Standarda dahil olmayan, `fcntl.h`, `sys/types.h`, `sys/stat.h` ve `io.h` adlı başlık dosyalarında bulunur. Aşağıda anlatılan isimlerin önünde, standart olmadıklarını göstermek için, altçizgi (`_`) karakteri bulunmaktadır; birçok sistemde bu karakter bulunmayabilir.

Standart girdi, çıktı ve hata dosyalarının sırasıyla 0, 1 ve 2 şeklinde dosya tanımlayıcıları vardır.

Bir dosya oluşturma yolu

```
dosya_tanimlayicisi = _creat(kar_dizisi, izin)
```

yazmaktır. `_creat` ile ilgili yeni bir şey de her dosya için bir izin belirtebilmeyi sağlamasıdır. `kar_dizisi` adında bir dosya önceden bulunmuyorsa, `_creat` böyle bir dosya yaratır ve yazma için açar. Daha sonra, dosya ilk defa kapatıldığında, `_creat`'in ikinci argümanı olarak belirtilen izin dosyaya atanır. `izin`, `_S_IWRITE` ve `_S_IREAD` kütüphane değişmezlerinin bir tanesine eşit olması gereken bir tamsayı ifadesidir.

`_S_IREAD` sadece okumaya izin verilmiştir, `_S_IWRITE` yazılmaya izin verilmiştir anlamına gelir. Hem yazma hem de okumaya izin vermek için `_S_IREAD|_S_IWRITE` kullanın.

Eğer `kar_dizisi` adında bir dosya daha önceden bulunuyorsa, `_creat` içindekileri silip, yazma için açmaya çalışır. Dosyanın eski izin durumu etkisini sürdürür.

Normalde, `_creat` dosya için bir tanımlayıcı döndürür. Eğer başarısızsa, örneğin dosya daha önceden var olan salt okunabilen bir dosya ise veya çok fazla açık dosya varsa, `-1` döndürür.

`kar_dizisi` adında bir dosyayı açmak için aşağıdaki yazılır:

```
dosya_tanimlayicisi = _open(kar_dizisi, bayrak, izin)
```

Eğer her şey iyi giderse `dosya_tanimlayicisi`'nin içine bir tutamak değeri konur; aksi takdirde `-1` döndürülür. `bayrak` dosya ile ilgili yapılacak işlemi gösterir; örneğin dosyayı okumak için `_O_RDONLY` kullanın. izin yukarıdaki gibi `_S_IWRITE` ve `_S_IREAD` kütüphane değişmezlerinden biri olmalıdır ve sadece `bayrak` için `_O_CREAT` (dosyayı yarat) kullanıldığında anlam taşır.

```
_close fonksiyonu daha önce açık bulunan bir dosyayı kapatmak için kullanılır ve  
_close(dosya_tanimlayicisi)
```

şeklinde. Dosyayı kapatabilirse `0`, aksi takdirde `-1` döndürür.

Dosya tanımlayıcıları kullanarak girdi-çıkı yapmak istiyorsanız, okunacak veya yazılacak baytlar için önceden yeterli büyüklükte bir (tampon) bölge tanımlamanız gerekir. Aşağıda, tampon'un yeterince büyük bir **char** dizisi olduğunu varsayın.

`dosya_tanimlayicisi` tarafından belirtilmiş bulunan dosyaya `sayi` kadar bayt yazmak isterseniz

```
s = _write(dosya_tanimlayicisi, tampon, sayi)
```

kullanın. Yazmayı düşündüğünüz baytları, `tampon` ile başlayan bölgeye daha önceden yerleştirmiş olmanız gerekir. `s`'ye ya `-1` (hatayı göstermek için) veya gerçekte yazılmış bulunan bayt sayısı konur. `_write` işleminden sonra eğer `s != sayi` ise bir hata olmuş demektir; fonksiyonu kullandıktan sonra bu testi yapmayı unutmayın.

```
s = _read(dosya_tanimlayicisi, tampon, sayi)
```

ifadesi `dosya_tanimlayicisi` ile belirtilen dosyanın okunup `tampon` ile başlayan bölgeye baytların konulmasını sağlar. Dosya sonuna ulaşıldığında okuma biter. Her durumda en fazla `sayi` kadar bayt okunur. `s` okunmuş bulunan bayt sayısını veya okuma yapılamazsa `-1` içerecektir.

`fseek`'in dosya tanımlayıcısı karşılığı `_lseek`'tir.

```
uzun_yeni_yer = _lseek(dosya_tanimlayicisi, uzun_sayi,  
nere)
```

çağrısında son iki argüman *fseek*'te olduğu gibi aynı anlam taşırlar ve aynı iş yapılmış olur. *_lseek*, dosyanın başından şimdiki konuma kadar olan uzaklığı bayt cinsinden döndürür.

7.7. Bir Örnek—Öğrenci Veritabanı

Öğrenci kayıtlarını tutup bunlar üzerinde ekleme, silme ve görüntüleme işlemlerini yapan bir program yazalım. Program kullanıcıya bir menü sunacak ve kullanıcının seçimine göre ilgili seçeneği çalıştıracaktır. C'nin bu tür uygulama için en uygun dil olduğunu veya bunun en iyi ve kısa program olduğunu savunmuyoruz; amacımız sadece, bu bölümde anlatılmış olan C ile ilgili bazı konuları göstermektir. Programın listesinden sonra, kodun detaylı bir açıklaması verilmektedir. Devam etmeden önce, programı bilgisayara girip çalıştırmanız önerilir.

```

1.  #include <conio.h>
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.
6.  /* Degismezler */
7.  char const DOSYA_ADI[] = "OGRENCI.DAT";
8.  char const BOS[] = "\r\r\r\r\r\r\r\r\r\r";
9.  #define ANHTR_UZ 7
10. char const AKD[] = "%7s";
11. #define ISIM_UZ 15
12. #define KAYIT_BOYU ((long) (ANHTR_UZ+ISIM_UZ+4))
13. char const KKD[] = "%7s%15s%4d";
14.
15. struct kayit {
16.     char anhtr[ANHTR_UZ+1], isim[ISIM_UZ+1];
17.     int yil;
18. };
19.
20. /* Her fonksiyonda bastan tanımlamak yerine bu
21.  * degiskenler burada tanımlanmışlardır.
22.  */
23. char satir[128];
24. FILE *dg;
25.
26. void dizioku /* d icine en fazla u karakter oku */
27. (char d[], int u)
28. {
29.     gets(satir);
30.     strncpy(d, satir, u);
31.     d[u] = '\0';
32. } /* dizioku */
33.
34. long bul /* Dosya icindeki ANHTR'in yerini; */
35. (char const anhtr[]) /* aksi takdirde -1L dondur. */
36. {
37.     int drm, kar = 1;
38.     long dns;
39.     char dl[ANHTR_UZ+1], d2[ANHTR_UZ+1];
40.

```



```

41.     sprintf(d2, AKD, anhtr); /* soldan bosluk doldur */
42.     dg = fopen(DOSYA_ADI, "rb");
43.     if (dg == NULL)
44.         return -1L;
45.     drm = fseek(dg, -KAYIT_BOYU, SEEK_END);
46.     while (drm == 0 && fgets(d1, ANHTR_UZ+1, dg) != NULL &&
47.           (kar=strcmp(d2,d1)) != 0)
48.         drm = fseek(dg, -KAYIT_BOYU-ANHTR_UZ, SEEK_CUR);
49.     if (kar == 0)
50.         dns = ftell(dg)-ANHTR_UZ;
51.     else
52.         dns = -1L;
53.     fclose(dg);
54.     return dns;
55. } /* bul */
56.
57. void ekleme (void)
58. {
59.     long konum;
60.     struct kayit k;
61.     int scd;
62.
63.     puts("Eklenecek ogrenci numarasini girin:");
64.     dizioku(k.anhtr, ANHTR_UZ);
65.     if (bul(k.anhtr) == -1L) {
66.         puts("Ogrenci adini girin:");
67.         dizioku(k.isim, ISIM_UZ);
68.         do {
69.             puts("Kayit yilini girin:");
70.             scd = scanf("%d", &k.yil);
71.             gets(satir);
72.         } while (1!=scd || k.yil<1980 || k.yil>9999);
73.         konum = bul(BOS);
74.         dg = fopen(DOSYA_ADI, "ab"); /* dosyayi olustur */
75.         fclose(dg);
76.         dg = fopen(DOSYA_ADI, "r+b");
77.         if (konum == -1L)
78.             fseek(dg, 0L, SEEK_END); /* dosya sonuna git */
79.         else
80.             fseek(dg, konum, SEEK_SET); /* bos kayida git */
81.         fprintf(dg, KKD, k.anhtr, k.isim, k.yil);
82.         fclose(dg);
83.         puts("Ekleme islemi tamamlandi.");
84.     }
85.     else
86.         fprintf(stderr, "Cift anahtar!\n\a");
87. } /* ekleme */
88.
89. void silme (void)
90. {
91.     long konum;
92.     char anhtr[ANHTR_UZ+1];
93.

```

```

94.     puts("Silinecek ogrenci numarasini girin:");
95.     dizioku(anhtr, ANHTR_UZ);
96.     if ((konum = bul(anhtr)) != -1L) {
97.         dg = fopen(DOSYA_ADI, "r+b");
98.         fseek(dg, konum, SEEK_SET);
99.         fprintf(dg, AKD, BOS);
100.        fclose(dg);
101.        puts("Silme islemi tamamlandi.");
102.    }
103.    else
104.        fprintf(stderr, "Boyle anahtar yok!\n\a");
105. } /* silme */
106.
107. void görüntuleme (void)
108. {
109.     long konum;
110.     struct kayıt k;
111.
112.     puts("Goruntulenecek ogrenci numarasini girin:");
113.     dizioku(k.anhtr, ANHTR_UZ);
114.     if ((konum = bul(k.anhtr)) != -1L) {
115.         dg = fopen(DOSYA_ADI, "rb");
116.         fseek(dg, konum+ANHTR_UZ, SEEK_SET);
117.         printf("Ogrenci adi: %s\n", fgets(k.isim, ISIM_UZ+1, dg));
118.         fscanf(dg, "%d", &k.yil);
119.         fclose(dg);
120.         printf("Kayit yili : %d\n", k.yil);
121.     }
122.     else
123.         fprintf(stderr, "Boyle anahtar yok!\n\a");
124. } /* görüntuleme */
125.
126. int menu (void)
127. {
128.     int secenek;
129.
130.     do {
131.         puts("\n\t1.EKLEME\n\t2.SILME\n\t3.GORUNTULEME\n\t0.CIKIS");
132.         printf("Seciminizi yapin:");
133.     } while ((secenek=_getche()) < '0' || secenek > '3');
134.     putchar('\n');
135.     return secenek - '0';
136. } /* menu */
137.
138. void main (void)
139. {
140.     static void (*fd[])(void) =
141.     { (void (*)(void)) exit, ekleme, silme, görüntuleme };
142.
143.     while (1) (*fd[menu()])();
144. } /* main */

```

Programda, her öğrenci için bir sayı, isim ve kayıt yılı saklanır. Bu bilgiyi birleştirmek için Satır 15'te kayıt yapısı tanımlanmıştır. Öğrenci numarası “anahtar”dır, yani her öğrencinin ayrı numarası vardır ve öğrencinin bilgisine ulaşmak için bu numaray kullanırız.

menu fonksiyonu (Satır 126-136) kullanıcıya seçimleri sunar ve neyi seçtiğini gösteren bir sayı (aslında bir **char**) girmesini ister. 0'dan küçük veya 3'ten büyük

seçimler kabul edilmez. Geçerli bir seçenek girildiğinde, fonksiyon tarafından seçenek değişkeninin tamsayı eşdeğeri döndürülür. Bu fonksiyonda puts kullanımına dikkat edin. Herhangi bir değişkenin değerinin basılmasına gerek olmadığından, printf kullanmanıza gerek yoktur. Programda, böyle durumlar için puts kullandık.

main'deki **while** deyimi biraz karmaşık gözükebilir. Satır 140-141'de exit, ekleme, silme ve görüntüleme fonksiyonlarına göstergeler içerecek şekilde ilklenmiş bulunan, fd dizisinin menu'ncü elemanını çağırır. Gördüğünüz gibi, fonksiyonlara göstergeler için bir dizi gibi, karmaşık veri yapılarının kullanımı küçük bir yerde çok şey yapabilecek kod yazmamızı sağlar. *Dikkat:* stdlib.h'de tanımlanmış olan exit fonksiyonu bir **int** argüman beklemektedir; diğer fonksiyonlara uyması için, bir kalıp aracılığı ile, bu fonksiyonu argümansız hale çevirdik.

ekleme, silme ve görüntüleme fonksiyonları, argüman olarak verilmiş anahtarları içeren kaydı taramak için bul adı verilen bir fonksiyonu çağırırlar. bul (Satır 34-55) kaydın dosyanın başına göre konumunu bir **long** sayı şeklinde verir; herhangi bir hata olursa -1L döndürür. bul'un başındaki sprintf, anhtır'ın sağa dayanmış bir uyarlamasını d2'ye koyar. Daha sonra, öğrenci dosyasını okumak için dosyayı açmaya çalışırız. Eğer açılmazsa, hata döndürülür. Eğer dosya açılırsa, dosyadaki son kaydın başına konumlanırız. Satır 46-48'deki **while** şu anda konumlandırılan kaydın anahtarını d2 ile karşılaştırır ve eğer eşitseler veya dosyanın başına ulaşırsak, sona erer. Satır 48 dosyayı bir önceki kaydın başına konumlandırır. Dosya taramasını geriye doğru yapmamızın nedeni, fseek fonksiyonunun, dosyanın sonundan ileri doğru gitmeyi bir hata şeklinde değerlendirmemesinden kaynaklanır. Satır 53'te dosyayı kapattıktan sonra, bul'un sonunda, (Satır 50'de ftell kullanılarak uygun değer verilen) dns döndürülür.

Üç “büyük” fonksiyon tarafından çokça kullanılan başka bir fonksiyon da dizioku'dur. Bu “kolaylık” fonksiyonu iki argüman kabul eder: İçine bir şey okunacak karakter dizisi ile okunacak dizinin uzunluğunu gösteren bir tamsayı. Baştaki gets girdiden bir satır okur. Geri kalan deyimler, gerekli sayıda karakteri d'ye aktarır sonuna bir boş karakter koyarlar.

ekleme fonksiyonu dosyaya yeni bir öğrenci kaydı yerleştirmeye çalışır. Yeni öğrencinin numarasını okur ve bu anahtarla bul'u çağırır. Eğer bul -1L'den farklı bir şey döndürürse, dosya içinde bu numaradan bir öğrenci kaydı vardır demektir, onun için ekleme uygun bir uyarı mesajı ile sona erer. Eğer bu numara dosyada yeni ise, öğrencinin ismi ve kayıt yılı sorulur. Yıl için dört rakamlı bir sayı girinceye kadar devam etmeyi engelleyen **do** deyimine dikkat ediniz. scanf'in gerçekleştirilen başarılı dönüşüm sayısını döndürdüğünü anımsayın.

Öğrenci hakkında bütün bilgileri elde ettiğimize göre, onu dosyaya yazmamız gerekir. Satır 73'te BOS anahtarları içeren bir kayıt ararız. BOS, bir öğrenci numarası olarak, kullanıcı ne kadar çalışırsa çalışsın, giremeyeceği özel bir karakter dizisidir. (Satır 8'deki BOS'un tanımına bakıp bunun neden böyle olduğunu söyleyiniz.) BOS anahtarları daha önce eklenmiş, fakat sonra silinmiş kayıtları gösterir. Satır 74-75, dosya yoksa, yeni bir dosya açar; aksi takdirde bir değişiklik yapmaz. Satır 76, okuma-yazma erişimi için dosyayı

açar. Eğer BOS bir kayıt yoksa, yeni kayıt dosyanın sonuna eklenir (Satır 78). Aksi takdirde, adresi konum içinde bulunan BOS kayda yazılır (Satır 80). Asıl yazma işlemi Satır 81'de `fprintf` ile yapılır. Dosyayı kapattıktan sonra ekleme sona erer.

`silme` silinecek öğrenci numarasını okuyarak başlar. Böyle bir öğrencinin dosya içinde olup olmadığını araştırmak için `bul` çağrılır. Eğer kayıt varsa, anahtarına BOS yazılır ve `silme` biter. Eğer yoksa, dönmeden önce, uygun bir hata mesajı basılır. Hata mesajlarının `stderr`'e gönderildiğinde dikkat edin.

`goruntuleme`, öğrenci numarasını okur ve `silme`'ye benzer bir şekilde kaydın bulunup bulunmadığını kontrol eder. Kayıt bulunursa, anahtar dışındaki bilgiler görüntülenir.

Problemler

1. `stdio.h` dosyasında `getc()` ve `putc()`'nin tanımlarına bakıp ne olduğunu açıklamaya çalışın.
2. Kısım 7.7'deki örnek programa `degistirme` adında yeni bir seçenek ekleyin. Seçildiğinde, öğrenci numarası okunur ve öğrencinin şu anki bilgileri görüntülenir. Kullanıcı, isterse, anahtar dışındaki bilgileri değiştirebilir ve yeni bilgiler eski bilgilerin yerini dosyada alır. Varlık testleri ve uygun hata mesajları bulunmalıdır.
3. Sözdizimsel açıdan doğru olan bir C programından bütün açıklama ve görünmeyen gereksiz karakterleri çıkaran bir program yazın.
4. Komut satırında argüman olarak verilen dosyaları birleştirip `stdout`'a yazan (ve UNIX komutu `cat`'e benzeyen) bir program yazın.
5. Büyük bir dosyayı birkaç küçük dosyaya ayıran bir program yazın. İlk argüman dosya adını; ikincisi (eğer verilmezse 1000 varsayılacak) parça başına satır sayısını gösterir. Parçaların dosya isimleri `parca.1`, `parca.2` vs olacaktır.
6. Argüman olarak verilen iki dosyayı karşılaştırıp farklı olan baytları sekizli gösterimde görüntüleyecek (ve UNIX komutu `cmp`'e benzeyen) bir program yazın.
7. Bir satır editörü yazın. Çalışma dosyasının adı komut satırı argümanı olarak verilecektir. Program aşağıdaki komutların herhangi birini girdi olarak kabul edebilecektir:

E *numara*

Bu komuttan sonra, standart girdiden bir satır okunup *numara*'ıncı satırdan önce yerleştirilir.

S *numara*

Numara'ıncı satır silinir.

G *numara*

Numara'ncı satır görüntülenir.

C

Program sona erer.

8. Bir metin dosyasındaki satırları tersten basacak bir program yazın; yani son satır ilk olarak basılacak vs. Dosya boyuna herhangi bir sınırlama getirmeyin.
9. Ekranı temizleyen bir fonksiyon yazın.
10. Sözdizimsel açıdan doğru ve tam olan bir C programını içeren bir dosyayı alıp, aynı programı daha iyi bir biçimde başka bir dosyaya yazan bir program hazırlayın. “Daha iyi bir biçim” derken, şunu anlatmak istiyoruz: Bütün fonksiyon başlıkları aynı sütunda başlar ve başka deyimlerin parçaları olan blokların sol ve sağ çengelli parantezlerin yeri Bölüm 2’de önerilen şekildedir.
11. Metin okuyup aşağıdaki şekilde biçimlendirilmiş olarak bir dosyaya koyan bir program yazın: Solda 7 karakterlik boşluk bulunur, doküman her biri 55 satır olan sayfalardan oluşur, her satır en fazla 65 karakterdir, her sayfanın son satırında sayfa numarası bulunur ve noktadan sonra gelen ilk harf büyüğe dönüştürülür. Program, tek bir yeni satır ('\\n') karakterini boşluğa çevirecek; arka arkaya iki tane yeni satır karakteri varsa, bunlar paragraf sonu olarak değerlendirilecek ve aynen bırakılacaktır. Gereksinimlerinize göre programı geliştirin. (Bazı biçimleme komutları ekleyin: Örneğin, özel bir karakterle başlayan satırlar emir olarak değerlendirilebilir.)

EK A: KARAKTER KODLARI ÇİZELGESİ

Bizim kullandığımız sistemde olduğu gibi, birçok sistemde karakterleri göstermek için ASCII (*American Standard Code for Information Interchange*—Bilgi Değişimi için Standart Amerikan Kodu) karakter kodları kullanılır. ASCII karakter kümesi 128 tane (7 bitlik) koddan oluşur ve 33 kontrol karakteri, bir boşluk ve 94 görüntülenebilen (grafik) karakteri tanımlar. Bir kontrol karakteri çıktı aygıtında değişik etkilere neden olabildiği gibi, bazen grafik bir şekil de oluşturabilir.

Bir karakterlik bilginin saklanması için 8 bitlik baytların kullanıldığı makineler genelde 128 tane daha karakter içeren “genişletilmiş” bir karakter kümesi sağlarlar. Bu karakterlerin kodları 128’den 255’e kadardır ve aygıttan aygıtı veya sistemden sisteme değişebilir.

Bundan sonraki dört sayfa, standart ASCII karakter kümesinin bir listesini vermektedir. Bu listede kontrol karakterleri *^tuş* şeklinde gösterilirler; bu da CONTROL tuşu basılı iken *tuş*a basarak bu karakterin elde edilebileceği anlamına gelmektedir. IBM uyumlu kişisel bilgisayarlarda (PC’lerde) ALT tuşu basılı iken klavyenin sağındaki sayısal tuşlar kullanılarak girilecek olan ASCII ondalık kod aracılığıyla ilgili karakter oluşturulabilir. Örneğin, ‘A’ harfini elde etmek için ALT tuşu basılı iken, sırayla önce 6’ya sonra da 5’e basın.

Daha sonraki dört sayfada ise 128’den 255’e kadar kodlara sahip olan karakterler için PC’lerde kullanılan iki ayrı karakter standardı gösterilmiştir:

- ASCII (genişletilmiş): Bu, daha çok MS-DOS ortamında kullanılmaktadır. MS-DOS ortamında, bu karakterlerin klavyeden girilmesi için ALT tuşu basılı iken klavyenin sağındaki sayısal tuşlar kullanılarak, ilgili karakterin ondalık kodu girilmelidir. Örneğin, ‘Ç’ için ALT+128. Windows ortamında da kullanılan yöntem aynıdır, ancak bu işlemi yaparken NUM LOCK ışığının yakılı olması gerekmektedir.

- ANSI: Windows ortamında kullanılmaktadır. Bu karakterlerin klavyeden girilmesi için yukarıdaki yöntem kullanılır; sadece girilen kodun ANSI olduğunu belirtmek için kodun önüne '0' eklenir. Örneğin, 'Ç' için ALT+0199.

Türkçe karakterler için en az üç ayrı standart bulunmaktadır. Herbirinde, belirli karakter kodları için, uluslararası tanımlanmış karakterin yerini Türkçe karakter almaktadır. Bunlar, listede, yerini aldıkları karakterlerin yanında, parantez içinde verilmiştir.

- 7 bitlik ASCII standardı: Bu sadece 128 değişik karakter kodunu destekleyen aygıtlarda kullanılmaktadır. Günümüzde 256 değişik karakter kullanan PC'lerin yaygınlaşmasıyla, bu standart daha az kullanılmaya başlanmıştır.
- 8 bitlik genişletilmiş ASCII standardı: Genelde MS-DOS ortamında kullanılmaktadır.
- 8 bitlik ANSI standardı: Windows gibi grafik ortamlarda kullanılmaktadır.

<u>Ond.</u>	<u>Sekizli</u>	<u>Onaltılı</u>	<u>Karakter</u>	<u>Anlamı</u>
0	0000	0x00	^@ NUL	Boş—zaman doldurmak için kull. tamamı sıfır karakter
1	0001	0x01	^A SOH	Başlık başı
2	0002	0x02	^B STX	Metin başı
3	0003	0x03	^C ETX	Metin sonu
4	0004	0x04	^D EOT	İletim sonu
5	0005	0x05	^E ENQ	Sorgu—“Kimsiniz?”
6	0006	0x06	^F ACK	Olumlu yanıt—“Evet”
7	0007	0x07	^G BEL	Zil—İnsan dikkati gerekiyor
8	0010	0x08	^H BS	Geriye alma (biçim etkileyicisi)
9	0011	0x09	^I HT	Yatay durak (biçim etkileyicisi)
10	0012	0x0A	^J LF	Satır ilerletme (biçim etkileyicisi)
11	0013	0x0B	^K VT	Dikey durak (biçim etkileyicisi)
12	0014	0x0C	^L FF	Sayfa ilerletme (biçim etkileyicisi)
13	0015	0x0D	^M CR	Satırbaşı (biçim etkileyicisi)
14	0016	0x0E	^N SO	Dışarı kayma—standart olmayan kod geliyor
15	0017	0x0F	^O SI	İçeri kayma—standart koda geri dönüş
16	0020	0x10	^P DLE	Veri bağl. kaçışı—sınırlı veri iletişimi kontr. değişikliği
17	0021	0x11	^Q DC1	Yardımcı aygıtları açıp kapamak için aygıt kontrolü
18	0022	0x12	^R DC2	Yardımcı aygıtları açıp kapamak için aygıt kontrolü
19	0023	0x13	^S DC3	Yardımcı aygıtları açıp kapamak için aygıt kontrolü
20	0024	0x14	^T DC4	Yardımcı aygıtları açıp kapamak için aygıt kontrolü
21	0025	0x15	^U NAK	Olumsuz yanıt—“Hayır”
22	0026	0x16	^V SYN	Eşzamanlı boşa işleme—eşzamanlama sağlamak için
23	0027	0x17	^W ETB	İletim öbeği sonu—fiziksel iletişim öbekleri ile ilintili
24	0030	0x18	^X CAN	Önceki bilginin iptali
25	0031	0x19	^Y EM	Ortam sonu—kullanılan/istenen bilgi bölümünün sonu
26	0032	0x1A	^Z SUB	Hatalı karakterin yerine gelen karakter
27	0033	0x1B	^[ESC	Kaçış—kod genişlemesi için
28	0034	0x1C	^ \ FS	Dosya ayırıcısı
29	0035	0x1D	^] GS	Grup ayırıcısı
30	0036	0x1E	^ ^ RS	Kayıt ayırıcısı
31	0037	0x1F	^ _ US	Birim ayırıcısı

<u>Ond.</u>	<u>Sekizli</u>	<u>Onaltılı</u>	<u>Karakter</u>	<u>Anlamı</u>
32	0040	0x20	BOSLUK	Basılmayan karakter—sözcük ayırıcısı
33	0041	0x21	!	Ünlem işareti
34	0042	0x22	"	(Çift) tırnak işareti
35	0043	0x23	# (Ğ)	Sayı işareti
36	0044	0x24	\$	Dolar
37	0045	0x25	%	Yüzde
38	0046	0x26	&	Ve işareti
39	0047	0x27	'	Kesme işareti (tek tırnak)
40	0050	0x28	(Sol parantez
41	0051	0x29)	Sağ parantez
42	0052	0x2A	*	Yıldız
43	0053	0x2B	+	Artı
44	0054	0x2C	,	Virgül
45	0055	0x2D	—	Çizgi işareti
46	0056	0x2E	.	Nokta
47	0057	0x2F	/	Bölu
48	0060	0x30	0	
49	0061	0x31	1	
50	0062	0x32	2	
51	0063	0x33	3	
52	0064	0x34	4	
53	0065	0x35	5	
54	0066	0x36	6	
55	0067	0x37	7	
56	0070	0x38	8	
57	0071	0x39	9	
58	0072	0x3A	:	İki nokta
59	0073	0x3B	;	Noktalı virgül
60	0074	0x3C	<	Küçüktür
61	0075	0x3D	=	Eşittir
62	0076	0x3E	>	Büyüktür
63	0077	0x3F	?	Soru işareti

<u>Ond.</u>	<u>Sekizli</u>	<u>Onaltılı</u>	<u>Karakter</u>	<u>Anlamı</u>
64	0100	0x40	@ (Ç)	-de işareti
65	0101	0x41	A	
66	0102	0x42	B	
67	0103	0x43	C	
68	0104	0x44	D	
69	0105	0x45	E	
70	0106	0x46	F	
71	0107	0x47	G	
72	0110	0x48	H	
73	0111	0x49	I	
74	0112	0x4A	J	
75	0113	0x4B	K	
76	0114	0x4C	L	
77	0115	0x4D	M	
78	0116	0x4E	N	
79	0117	0x4F	O	
80	0120	0x50	P	
81	0121	0x51	Q	
82	0122	0x52	R	
83	0123	0x53	S	
84	0124	0x54	T	
85	0125	0x55	U	
86	0126	0x56	V	
87	0127	0x57	W	
88	0130	0x58	X	
89	0131	0x59	Y	
90	0132	0x5A	Z	
91	0133	0x5B	[(Ş)	Sol köşeli parantez
92	0134	0x5C	\ (İ)	Ters bölü
93	0135	0x5D] (Ö)	Sağ köşeli parantez
94	0136	0x5E	^ (Ü)	Uzatma işareti
95	0137	0x5F	_	Alt çizgi

<u>Ond.</u>	<u>Sekizli</u>	<u>Onaltılı</u>	<u>Karakter</u>	<u>Anlamı</u>
96	0140	0x60	` (ç)	Ağır vurgu
97	0141	0x61	a	
98	0142	0x62	b	
99	0143	0x63	c	
100	0144	0x64	d	
101	0145	0x65	e	
102	0146	0x66	f	
103	0147	0x67	g	
104	0150	0x68	h	
105	0151	0x69	i	
106	0152	0x6A	j	
107	0153	0x6B	k	
108	0154	0x6C	l	
109	0155	0x6D	m	
110	0156	0x6E	n	
111	0157	0x6F	o	
112	0160	0x70	p	
113	0161	0x71	q (ğ)	
114	0162	0x72	r	
115	0163	0x73	s	
116	0164	0x74	t	
117	0165	0x75	u	
118	0166	0x76	v	
119	0167	0x77	w	
120	0170	0x78	x	
121	0171	0x79	y	
122	0172	0x7A	z	
123	0173	0x7B	{ (ş)	Sol çengelli parantez
124	0174	0x7C	(ı)	Dikey çizgi
125	0175	0x7D	} (ö)	Sağ çengelli parantez
126	0176	0x7E	~ (ü)	İnceltme işareti
127	0177	0x7F	DEL	Ortam doldurma

<u>Ond.</u>	<u>Sekizli</u>	<u>Onaltılı</u>	<u>ASCII karakter</u>	<u>ANSI karakter</u>
128	0200	0x80	Ç	€
129	0201	0x81	ü	□
130	0202	0x82	é	,
131	0203	0x83	â	f
132	0204	0x84	ä	"
133	0205	0x85	à	...
134	0206	0x86	â	†
135	0207	0x87	Ç	‡
136	0210	0x88	ê	^
137	0211	0x89	ë	%
138	0212	0x8A	è	Š
139	0213	0x8B	ï	<
140	0214	0x8C	î	€
141	0215	0x8D	ì (1)	□
142	0216	0x8E	Ä	□
143	0217	0x8F	Å	□
144	0220	0x90	É	□
145	0221	0x91	æ	`
146	0222	0x92	Æ	'
147	0223	0x93	ô	"
148	0224	0x94	ö	"
149	0225	0x95	ò	•
150	0226	0x96	û	—
151	0227	0x97	ù	—
152	0230	0x98	ÿ (İ)	~
153	0231	0x99	Ö	™
154	0232	0x9A	Ü	Š
155	0233	0x9B	ç	>
156	0234	0x9C	£	œ
157	0235	0x9D	¥	□
158	0236	0x9E	₺ (₺)	□
159	0237	0x9F	f (₺)	Ÿ

<u>Ond.</u>	<u>Sekizli</u>	<u>Onaltılı</u>	<u>ASCII karakter</u>	<u>ANSI karakter</u>
160	0240	0xA0	á	
161	0241	0xA1	í	ı
162	0242	0xA2	ó	ç
163	0243	0xA3	ú	ü
164	0244	0xA4	ñ	ı
165	0245	0xA5	Ñ	¥
166	0246	0xA6	ª (ğ)	ı
167	0247	0xA7	º (ğ)	Ş
168	0250	0xA8	č	..
169	0251	0xA9	—	©
170	0252	0xAA	┐	a
171	0253	0xAB	½	«
172	0254	0xAC	¼	┐
173	0255	0xAD	ı	—
174	0256	0xAE	«	®
175	0257	0xAF	»	—
176	0260	0xB0	░	o
177	0261	0xB1	▒	±
178	0262	0xB2	▓	2
179	0263	0xB3		3
180	0264	0xB4	└	˘
181	0265	0xB5	┘	µ
182	0266	0xB6	┌	¶
183	0267	0xB7	┐	•
184	0270	0xB8	┘	˘
185	0271	0xB9	┌	1
186	0272	0xBA		o
187	0273	0xBB	┐	»
188	0274	0xBC	┘	¼
189	0275	0xBD	┌	½
190	0276	0xBE	┘	¾
191	0277	0xBF	┐	č

<u>Ond.</u>	<u>Sekizli</u>	<u>Onaltılı</u>	<u>ASCII karakter</u>	<u>ANSI karakter</u>
192	0300	0xC0	L	À
193	0301	0xC1	⊥	Á
194	0302	0xC2	T	Â
195	0303	0xC3	┆	Ã
196	0304	0xC4	—	Ä
197	0305	0xC5	†	Å
198	0306	0xC6	‡	Æ
199	0307	0xC7	‡	Ç
200	0310	0xC8	ℒ	È
201	0311	0xC9	℞	É
202	0312	0xCA	⊥	Ê
203	0313	0xCB	‡	Ë
204	0314	0xCC	‡	Ì
205	0315	0xCD	=	Í
206	0316	0xCE	‡	Î
207	0317	0xCF	⊥	Ï
208	0320	0xD0	⊥	Ð (Ğ)
209	0321	0xD1	‡	Ñ
210	0322	0xD2	‡	Ò
211	0323	0xD3	ℒ	Ó
212	0324	0xD4	ℒ	Ô
213	0325	0xD5	F	Õ
214	0326	0xD6	‡	Ö
215	0327	0xD7	‡	×
216	0330	0xD8	‡	Ø
217	0331	0xD9	┘	Ù
218	0332	0xDA	┘	Ú
219	0333	0xDB	■	Û
220	0334	0xDC	■	Ü
221	0335	0xDD	■	Ý (İ)
222	0336	0xDE	■	Þ (Ş)
223	0337	0xDF	■	ß

<u>Ond.</u>	<u>Sekizli</u>	<u>Onaltılı</u>	<u>ASCII karakter</u>	<u>ANSI karakter</u>
224	0340	0xE0	α	à
225	0341	0xE1	β	á
226	0342	0xE2	Γ	â
227	0343	0xE3	π	ã
228	0344	0xE4	Σ	ä
229	0345	0xE5	σ	å
230	0346	0xE6	μ	æ
231	0347	0xE7	τ	ç
232	0350	0xE8	Φ	è
233	0351	0xE9	θ	é
234	0352	0xEA	Ω	ê
235	0353	0xEB	δ	ë
236	0354	0xEC	∞	ì
237	0355	0xED	Ø	í
238	0356	0xEE	€	î
239	0357	0xEF	∩	ï
240	0360	0xF0	≡	ð (ğ)
241	0361	0xF1	±	ñ
242	0362	0xF2	≥	ò
243	0363	0xF3	≤	ó
244	0364	0xF4		ô
245	0365	0xF5		õ
246	0366	0xF6	÷	ö
247	0367	0xF7	≈	÷
248	0370	0xF8	°	ø
249	0371	0xF9	•	ù
250	0372	0xFA	•	ú
251	0373	0xFB	√	û
252	0374	0xFC	ⁿ	ü
253	0375	0xFD	²	ý (ı)
254	0376	0xFE	•	þ (ş)
255	0377	0xFF		ÿ

EK B: MICROSOFT C DERLEYİCİSİ HAKKINDA TEMEL BİLGİLER

Bu ekte, Microsoft C derleyicileri hakkında bazı bilgiler verilecektir. Bir C programının Microsoft C Eniyileştirici Derleyicisi kullanılarak nasıl derleneceğine geçmeden önce, C bellek modellerini bilmek ve anlamak gerekir.

B.1. Bellek Modelleri

80x88 ve 80x86 işlemcilerinin yapısından dolayı, bir bellek adresi 16 bitlik kesim adresi ile kesim içinde 16 bitlik bir uzaklıktan oluşur. Böyle bir adres normalde *kesim:uzaklık* (*segment:offset*) şeklinde gösterilir. Verimlilik açısından, bir programın aynı kesim içinde kalan adresler üretmesi daha iyidir, çünkü adresleme sadece uzaklık oluşturularak sağlanabilir. Sadece uzaklık bölümünden oluşan bir adrese *yakın adres* adı verilir. Bazen, program, başka kesimlerde bulunan nesneleri adreslemek durumunda kalabilir; böyle adreslere *uzak adres* adı verilir. Uzak bir adres 32 bittir ve kesim temel adresi ile uzaklıktan oluşur. Bir kesimden, yani 64 Kbayttan daha büyük olan bir veri yapısı, örneğin bir dizi, uzak adres kullanılarak adreslenemez. Bu amaç için, *dev adres* adı verilen başka bir adres tipi kullanılır. Dev adreslerin uzak adreslerden farkı göstergelerde ortaya çıkar; uzak adreslerde göstergeler 16 bit olmalarına rağmen, dev adreslerde 32 bittir.

Bir C programı derlendikten sonra, en az bir *veri kesimi* ve bir *kod kesiminden* oluşur. Bu, olası en küçük C programı için bile geçerlidir. Büyük C programları birkaç veri kesimi, yani çok fazla ve büyük veri yapıları ve/veya birkaç kod kesimi, yani birçok yürütülebilir deyim şeklinde oluşabilirler. Büyük programların doğru bir şekilde derlenip

yürütülebilmeleri için, Microsoft C'ye *bellek modelleri* konmuştur. Aşağıdaki çizelge bellek modelleri arasındaki farkları göstermektedir:

Model	Kod Kesimi Sayısı	Veri Kesimi Sayısı	Kullanılan Kütüphane
minik (tiny)	kod ile veri toplam 1 kesim	1 kesim	SLIBCE.LIB
küçük (small)	1	1	SLIBCE.LIB
kısa (compact)	1	çok	CLIBCE.LIB
orta (medium)	çok	1	MLIBCE.LIB
büyük (large)	çok	çok	LLIBCE.LIB
dev (huge)	çok	çok	LLIBCE.LIB

Microsoft C Eniyileştirici Derleyicisi (MS-CL) için varsayılan model küçük olandır; Microsoft QuickC (MS-QC) ise her zaman orta modelde çalışır.

B.1. QC Kütüphanesi

MS-QC derleyici ortamı içinde bulunan fonksiyonlar çekirdek kütüphaneyi oluştururlar; ancak QuickC çekirdek kütüphanesi Standart C kütüphanesi içindeki bütün fonksiyonları içermez. Örneğin, `rand` standart fonksiyonu QC ile birlikte yüklenmez. Bu fonksiyonu çağıran bir program çalıştırmaya kalkarsanız, “çözümlememiş dışsal referans” şeklinde bir hata mesajı alırsınız. Bir amaç dosya oluşturmanız ve daha sonra yürütülebilir bir dosya elde etmek için bunu `LINK` ile bağlamanız gerekecektir. Programınızı her değiştirdiğinizde, bunu yapmaktan kurtulmak için, bir QuickC kütüphanesi oluşturmanız ve QC'yi çalıştırdığınızda bu kütüphaneyi de yüklemeniz gerekir. Bunu yapmak için aşağıdaki yöntemi kullanın:

- Gerektiği kadar dışsal referans yapan, örneğin `q1.c` adında, bir program yazın. Derleme esnasındaki uyarı mesajlarını dikkate almayın. Örneğin:

```
#include <stdlib.h>
void main (void)
{
    rand();
    srand();
    bsearch();
    qsort();
}
```

- QC'yi kullanarak bir amaç dosya, örneğin `q1.obj`, oluşturun.
- Bu dosyayı MS-QC tarafından sağlanan `QUICKLIB.OBJ` adlı amaç modülle beraber bağlayıp bir Quick kütüphane oluşturun. Örneğin:

```
LINK /Q QUICKLIB.OBJ+q1.obj,q1.lib;
```

- QC'yi çalıştırırken bu kütüphaneyi belirtin. Örneğin:

```
QC /1 q1
```

Dikkat: QC kütüphanelerinin içerilmesi QC çalışırken bellek-içi derleme için ayrılmış bulunan serbest bellek miktarının azalmasına yol açar. En iyisi, birden fazla küçük Quick kütüphane oluşturup, gerektiği zaman bunları yüklemektir.

B.3. CL Eniyileştirici Derleyicisi

Microsoft C Eniyileştirici Derleyicisinin (MS-CL) 5.10 uyarlaması, 80x86 işlemcileri ile MS-DOS ve OS/2 işletim sistemleri için amaç kod üretmede kullanılan gelişmiş bir eniyileştirici derleyicidir. 5.10 uyarlamasının 5.00 uyarlamasından tek farkı 80386 işlemcisi ile OS/2 işletim sistemini desteklemesidir. Microsoft Visual C++ paketi içinde bulunan Microsoft C Eniyileştirici Derleyicisinin 8.00 uyarlaması ise Windows ortamı ve C++ için ek olanaklar tanır.

Ancak, temel olarak, CL derleyici/bağlayıcısının kullanımı, komut satırından birtakım seçim anahtarları ile dosya isimleri belirtilmek suretiyle CL komutunun çağrılması şeklindedir. Seçeneklerin bir özeti ve çağrı örnekleri şöyledir:

C>CL /HELP

CL seçeneklerinin bir özeti görüntülenir.

C>CL F1.C

F1.C C kaynak dosyası derlenip SLIBCE.LIB adlı varsayılan kütüphane ile bağlanır. F1.OBJ amaç dosyası ve F1.EXE yürütülebilir program oluşturulur.

C>CL F1.C F2.C

Yukarıda olduğu gibi, fakat ayrıca F2.C dosyası F2.OBJ içine derlenir. Yürütülebilir program F1.EXE içinde bağlanır.

C>CL /c F1.C

Derleme yapılır, ama bağlama yapılmaz.

C>CL F1.OBJ

Daha önce derlenmiş bulunan F1.OBJ dosyası F1.EXE içine bağlanır.

C>CL F1

Varsayılan dosya tipi .OBJ olduğu için, yukarıdaki ile aynı.

C>CL F1.C F2.C F3.OBJ F4

F1.C ve F2.C derlenip, F1.OBJ, F2.OBJ, F3.OBJ ve F4.OBJ bağlandıktan sonra F1.EXE oluşturulur.

C>CL *.C

Bulunulan altdizin içindeki tüm C kaynak dosyaları derlenip tek bir program şeklinde bağlanır.

C>CL *

Bulunulan altdizin içindeki tüm .OBJ amaç dosyaları bağlanır.

C>CL /zi /Od F1.C

Eniyilemenin engellendiği ve Codeview hata düzelticisinin kullanılabileceği şekilde F1.EXE oluşturulur.

Aşağıda derleyici seçeneklerinden bazılarının özet bir listesi bulunmaktadır:

Bellek Modeli Seçenekleri

/AS	Küçük bellek modeli (varsayılan).
/AC	Kısa bellek modeli.
/AM	Orta bellek modeli.
/AL	Büyük bellek modeli.
/AH	Dev bellek modeli.
/AT	Minik bellek modeli (.COM dosyası oluştur).

Eniyileme Seçenekleri

/O	Eniyilemeye izin ver (/Ot ile aynı).
/Oa	Başka ad vermeyi dikkate alma.
/Od	Eniyilemeleri engelle (varsayılan).
/Oi	Yapııcı fonksiyonların kullanılmasına izin ver.
/Ol	Döngü eniyilemelerine izin ver.
/On	“Güvenilmez” eniyilemeleri engelle.
/Op	Duyarlık eniyilemelerine izin ver.
/Or	Satırıcı dönüşleri engelle.
/Os	Kod için eniyileme yap.
/Ot	Hız için eniyileme yap.
/Ox	En yüksek eniyileme (/Oailt /Gs).

Kod Oluşturma Seçenekleri

/G0	8088/8086 kodu oluştur (varsayılan).
/G1	186 kodu oluştur.
/G2	286 kodu oluştur.
/G3	386 kodu oluştur.
/Gc	Pascal tarzında fonksiyon çağrılarını oluştur.
/Gs	Yığıt kontrolü yapma.
/Gtsayı	Veri boyu eşiği.

Listeleme Seçenekleri

/Fadosyaadı _{opt}	MASM için girdi olarak kullanılabilecek birleştirici dil listesi (.ASM).
/Fcdosyaadı _{opt}	Birleştirilmiş kaynak ve birleştirici dil listesi (.COD).
/Fedosyaadı	Yürütülebilir dosya adı (.EXE).
/Fldosyaadı _{opt}	Amaç ve birleştirici dil listesi (.COD).
/Fmdosyaadı _{opt}	Bağlayıcı planı (.MAP).
/Fodosyaadı	Amaç dosya adı (.OBJ).
/Fsdosyaadı _{opt}	Kaynak listesi (.LST).

/Sl *satirgenişliği*

Listenin genişliğini *satirgenişliğine* ayarla; varsayılan değer 79'dur.

/Sp *sayfaboyu*

Listede sayfa başına satır sayısı; varsayılan değer 63'tür.

/Ss *"altbaşlık"*

Bir liste altbaşlığı görüntüle.

/St *"başlık"*

Bir liste başlığı görüntüle.

Önişlemci Seçenekleri

/C

Açıklamaları çıkarma.

/Disim=*metin*_{opt}

Kaynak programda "#define *isim* *metin*"e eşdeğerdir.

/E

Önişlemci çıktısını stdout'a gönder.

/EP

Önişlemci çıktısını stdout'a, "#line" emirleri olmadan, gönder.

/Iisim

#include işlemi için ek altdizin.

/P

Önişlemci çıktısını dosyaya gönder; dosya tipi .I'dir.

/Uisim

Önceden tanımlanmış makronun tanımını kaldır.

/u

Önceden tanımlanmış bütün makroların tanımını kaldır.

/X

"Standart yerleri" dikkate alma.

Dil Seçenekleri

/Za

Dildeki genişletmeleri engelle (sadece ANSI Standardını kullan).

/Zd

Satır numarası bilgisi.

/Ze

Dildeki genişletmelere izin ver (varsayılan).

/Zg

Fonksiyon prototipleri oluştur.

/Zi

Simgesel hata düzeltme bilgileri.

/Zl

.OBJ içindeki varsayılan kütüphane bilgisini kaldır.

/Zpn

Yapıları *n*-baytlık sınırlar içine paketlenir.

/Zs

Sadece sözdizim kontrolü yap.

Bağlama Seçenekleri

/Fonaltılı_sayı

Yığıt boyu (onaltılı gösterimde bayt sayısı).

/linkbağ_seç_ve_kütüp

Bağlayıcı seçenekleri belirle.

Kayan Noktalı İşlemler İçin Kod Üretme Seçenekleri

/FPa

Karşılıklı matematik kodu için çağrılar oluştur.

/FPc

80x87 benzetme kodu için çağrılar oluştur.

/FPc87

80x87 kodu için çağrılar oluştur.

/FPi
/FPi87

Satırıcı 80x87 benzetme kodu üret (varsayılan).
80x87 komutları üret.

Çeşitli Seçenekler

/c
/Huzunluk
/J
/Tcdosya
/Vkar_dizisi
/Wdüzey

Sadece derle, bağlama.
Dışsal isim uzunluğu.
Varsayılan karakter tipi **unsigned** olsun.
.C'si olmayan dosyayı derle.
Uyarlama *kar_dizisi*ni ver.
Uyarı düzeyi ($0 < \text{düzey} \leq 4$).

EK C: MICROSOFT CODEVIEW HATA DÜZELTİCİSİNE GENEL BİR BAKIŞ

Microsoft Codeview; C, Pascal, FORTRAN ve birleştirici dil programlarının yürütme esnasındaki hatalarını düzeltmek için kullanılan, kullanımı kolay, etkileşimli bir hata düzelticisidir. C programcıları için, kaynak dosyaların /Zi ve /Od seçenekleri ile derlenip /CO seçeneğiyle bağlanmaları gerekir. Bir örnek şöyledir:

```
C>CL /c /Zi /Od F1.C  
C>LINK /CO F1
```

veya,

```
C>CL /Zi /Od F1.C
```

Bu yolla F1 .EXE dosyası oluşturulduktan sonra, hata düzelticisini çağırmak için

```
C>CV seçenekler exedosyası argümanlar
```

yazmak gerekir. Burada, *exedosyası* derlenip bağlanmış olan programı içeren dosyanın adı, *argümanlar* hataları düzeltilecek olan programa geçirilecek olan argümanlar ve *seçenekler* de Codeview seçeneklerinden oluşan bir listedir. Bu seçeneklerden en önemli birkaç tanesi şöyledir:

- /2 İki monitör kullan; bir tanesi hata düzelticisi çıktısı diğeri de program çıktısı için.
- /B Renkli bir adaptör ve monitörde siyah/beyaz renkler kullan.
- /M Olşa bile, fareyi kullanma.

Codeview'a girildiğinde, seçilebilir menü maddeleri, bir görüntüleme penceresi, bir diyalog penceresi ve isteğe bağlı olarak bir yazmaç içerikleri görüntüleme bölgesinden oluşan bir ekran kullanıcıya sunulur. Hatası düzeltilecek programın ilk birkaç satırı otomatik olarak görüntüleme penceresine konur. Codeview komutları fare ve/veya klavye aracılığıyla girilebilir. Girdilerin çoğu için fare kullanımı kolay olduğu için, sadece klavye

komutları burada anlatılmaktadır. Bazı klavye komutları tek tuşa dayanır; diyalog komutları gibi, bazılarının ise diyalog penceresinden girilmesi gerekir. Aşağıda bazı komutların özet bir listesi verilmiştir. PAGE UP, PAGE DOWN, oklar vs gibi düzenleme tuşları her zamanki anlamlarını taşırlar.

CONTROL+HOME	Programın ilk satırını görüntüle.
CONTROL+END	Programın son satırını görüntüle.
F1	Yardım.
F2	Yazmaç penceresi görüntüleme anahtarı.
F3	Kaynak kodu, birleştirici kodu veya her ikisini görüntüle.
F4	Program çıktı ekranını görüntüle.
F5	Eğer varsa, bir sonraki kesilme noktasına kadar, programı yürüt.
F6	Değişik pencereler arasında dolaşma anahtarı.
F7	İmlecin bulunduğu yere kadar programı çalıştır.
F8	Bir izleme (<i>trace</i>) komutu çalıştır.
F9	Kesilme noktası (<i>breakpoint</i>) ver/sil.
F10	Bir sonraki kaynak satırını çalıştır (program adım [<i>step</i>] modu).

Diyalog komutları “*komut işlenenler*” şeklindedir. *İşlenenler* olarak her tür geçerli C ifadesi kullanılabilir. Diyalog komutlarında kullanılan diğer işlenenler aşağıda örneklerle tanıtılmaktadır:

.100	100 numaralı satır. (Öndeki noktaya dikkat edin.)
.X.C:100	X.C dosyasındaki 100 numaralı satır.
1000:2000	<i>kesim:uzaklık</i> şeklinde bir adres. Kesim isteğe bağlıdır. Rakam gösterimi o andaki sayı sistemine bağlıdır, fakat varsayılan sistem onaltılıdır.
AH BH CH DH	Yazmaçların yüksek baytları.
AL BL CL DL	Yazmaçların düşük baytları.
AX BX CX DX	16-bit yazmaçlar.
CS DS SS ES	16-bit kesim yazmaçları.
SP BP IP	16-bit gösterge yazmaçları.
SI DI	16-bit indis yazmaçları.
EAX EBX ECX EDX	32-bit yazmaçlar.
ESP EBP	32-bit gösterge yazmaçları.
ESI EDI	32-bit indis yazmaçları.
<i>adr1 adr2</i>	<i>Adr1</i> 'den <i>adr2</i> 'ye bir adres aralığı.
BY <i>adres</i>	<i>Adresteki</i> bayt.
WO <i>adres</i>	<i>Adresteki</i> sözcük.
DW <i>adres</i>	<i>Adresteki</i> çift sözcük.

Sıkça kullanılan diyalog komutları, komutun genel şeklinden sonra gelen örneklerle birlikte aşağıda liste şeklinde verilmiştir:

V .100	Satır 100'ü göster.
?ifade, biçim	Belirtilen <i>biçim</i> de <i>ifadenin</i> değerini göster.
?A+B, d	Ondalık olarak A+B'nin değerini göster.
?WO SP, x	SP ile gösterilen sözcüğü onaltılı gösterimde göster.
XP <i>simge</i>	<i>Simgenin</i> adresini göster.
X*	Bütün simgelerin adreslerini göster.
R	Bütün yazmaçları göster.
MD <i>biçim adres</i>	<i>Biçim</i> seçimine göre <i>adresten</i> başlayarak 128 bayt dök.
MD <i>biçim adr1 adr2</i>	<i>Biçim</i> seçimine göre <i>adr1</i> 'den <i>adr2</i> 'ye kadar dök. <i>Biçim</i> seçimi şöyledir:
A	ASCII karakter
B	Bayt
C	Program kodu
I	Tamsayı (2 bayt)
IU	İşaretsiz tamsayı
L	Uzun tamsayı (4 bayt)
R	Kısa kayan noktalı sayı (4 bayt)
RL	Uzun kayan noktalı sayı (8 bayt)
RT	10-baytlık kayan noktalı sayı
MC <i>adr1 adr2 adr3</i>	<i>Adr1</i> 'den <i>adr2</i> 'ye kadar olan baytları <i>adr3</i> 'ten başlayan baytlarla karşılaştırıp farkları göster.
MS <i>adr1 adr2 değer</i>	Adres aralığı içinde <i>değeri</i> araştır.
MS 100 1000 "isim"	100-1000 aralığı içinde "isim"i araştır.
MS 100 1000 0A	100-1000 aralığı içinde onaltılı gösterimde 0A'yı araştır.
BP <i>adr_listesi</i>	Kesilme noktası koy.
BC <i>adr_listesi</i>	Kesilme noktasını sil.
BP .29	Satır 29'a kesilme noktası koy.
BC 0 8 22	0, 8 ve 22 adreslerindeki kesilme noktalarını sil.
BC *	Tüm kesilme noktalarını sil.
BL	Tüm kesilme noktalarını göster.
W?ifade, <i>biçim</i>	<i>Ifadenin</i> değerini gözle ve <i>biçim</i> de göster.
W	Tüm gözetlemeleri göster.
WC *	Tüm gözetlemeleri sil.

EK D: MICROSOFT LIB VE NMAKE YARDIMCI PROGRAMLARINA GENEL BİR BAKIŞ

Bu ekte iki Microsoft yardımcı programlarına kısaca göz atılacaktır. Daha kesin bilgi için kullandığınız paketin elkitablarına danışın.

D.1. LIB Yardımcı Programı

LIB, amaç program kütüphaneleri oluşturmak ve bakımını yapmak için kullanılan yardımcı bir programdır. LIB'i çağırarak için kullanılacak komutun genel şekli şöyledir:

`C>LIB eskikütüphane komutlaropt ,listedosyasıopt ,yenikütüphaneopt`

Burada,

eskikütüphane

Yaratılacak yeni kütüphane veya değiştirilecek eski kütüphanenin dosya adı.

komutlar

LIB'e komutlar:

+*amaçdosya*

Eski kütüphaneye yeni amaç modülü ekle.

+*kütüphanedosyası*

*Kütüphanedosyası*ndaki tüm amaç modüllerini eski kütüphaneye ekle. (Kütüphaneleri birleştir.)

-+*amaçdosya*

Eski kütüphane içindeki amaç modülün yerine yenisini koy.

**modülismi* Kütüphanedeki bir modülü (*modülismi*.OBJ adlı) dosyaya aktar.

*-*modülismi* Kütüphanedeki bir modülü bir dosyaya taşı, yani aktarıp kütüphaneden sil.

listedosyası Çapraz referans için liste.

yenikütüphane Değişikliklerden sonra oluşturulan yeni kütüphanenin dosya ismi. Eğer belirtilmezse, eski kütüphanenin tipi .BAK'a dönüştürülüp, *eskikütüphane* isminde saklanır.

Örnekler:

LIB KH1 +MOD1; MOD1.OBJ'daki amaç modülü KH1.LIB kütüphanesine ekle.

LIB KH1 -+MOD1; KH1.LIB kütüphanesindeki amaç modülü MOD1.OBJ'daki amaç modülle değiştir.

LIB komutunda ayrıca birtakım seçenekler de bulunmaktadır.

D.2. NMAKE Yardımcı Programı

NMAKE—veya MAKE—farklı kaynak dosyalarda bulunan çok sayıda modülden oluşan büyük programların otomatik olarak derlenmesi ve bağlanması için kullanılan bir yardımcı programdır. NMAKE programını çalıştıracak komutun genel şekli şöyledir:

C>NMAKE *seçimler*_{opt} *makrotanımları*_{opt}

seçimler Aşağıdakiler olabilir:

/Fdosyaadi NMAKE yardımcı programı komutları içeren tanımlayıcı dosyanın adı. (Dosya tipi .MAK'tır.)

/D NMAKE tarafından işlenen her dosyanın son değiştirilme tarihini görüntüle.

/I Tanımlayıcı dosyadaki programlardan döndürülen kodları dikkate alma ve daha sonraki satırları yürütmeyi sürdür.

/N Tanımlayıcı dosyadaki komutları işletme, sadece görüntüle.

/S Sessiz mod, hiçbir şey görüntüleme.

/Xdosyaadi Hata mesajlarını bir dosyaya yönlendir.

/X- Hata mesajlarını standart çıktı aygıtına yönlendir.

makrotanımları İsteğe bağlı olarak, *isim=metin* şeklinde, bir veya daha fazla sayıda karakter dizisi.

Bir *tanımlayıcı dosya* bir veya daha fazla sayıda *tanımlayıcı bloktan* oluşur. Boş satırlar tanımlayıcı blokları ayırır. Bir tanımlayıcı bloktaki maddelerin sözdizimi ve anlambilimi aşağıda özetlenmiştir:

çıkdosya : girdosya ... #açıklama
#açıklama
komutlar

Çıkdosya iki noktadan sonra gelen dosyalara bağımlıdır. *Komutlar*, satırı *çıkdosya* üzerinde yapılacak işlemleri belirtir. Örneğin, XYZ.MAK tanımlayıcı dosyasında

```
XYZ.EXE : XYZ.C F1.C F2.C  
# XYZ'yi derleyip bağla.  
CL XYZ.C F1.C F2.C
```

yazılı ise

```
C>NMAKE /F XYZ
```

şeklinde bir çağrı, eğer XYZ.C, F1.C ve F2.C dosyalarından herhangi biri XYZ.EXE oluşturulduktan sonra değiştirilmişse, bu dosyaların tekrar derlenmesini sağlayacaktır.

EK E: DİLLERARASI ÇAĞRILAR

Yazılım ayrı ayrı modüller şeklinde geliştirildiğinde, modülleri ayrı programlama dilleri kullanarak gerçekleştirmek olasıdır. Bu, hem o modül için belli bir dilin daha uygun olmasından, hem de programcının o dili daha iyi bilmesinden dolayı belli bir dili seçmiş olmasından kaynaklanabilir. Örneğin, C’de yazılmış bir programın bazı modülleri için sık sık birleştirici dil kullanılır. Bu ekte bir C programından bir birleştirici dil yordamı çağırmak veya bir birleştirici dil programından bir C fonksiyonu çağırmak için çağırma ve argüman geçirme yöntemlerini kısaca inceleyeceğiz. BASIC, Pascal ve FORTRAN’dan C yordamları veya C’den bu dillerde yazılmış yordamları nasıl çağırabileceğinizi öğrenmek için *Microsoft C Version 5.00 Mixed Language Programming Guide* (Karma Dil Programlama Kılavuzu) veya *Microsoft Visual C++ Programming Techniques* (Programlama Teknikleri) elkitablarına bakınız.

E.1. Birleştirici İle Bağlayıcının Kullanılması

Bir C programında, fonksiyon isimleri dışsal simgelermiş gibi işlem görürler ve bağlayıcı tarafından kullanılmaları için, amaç program dosyasına kaydedilirler. Fakat bir farkla: C derleyicisi her fonksiyon isminin önüne bir altçizgi ekler. Örneğin:

```
xyz (...)  
{  
  ...  
}
```

fonksiyon tanımında, fonksiyon ismi amaç programda `_xyz` şeklinde kaydedilir. Standarda göre, dışsal tanıtıcı sözcüklerde, anlamlı karakter sayısı en az 8’dir. Ayrıca, küçük büyük harf ayırımı yapılmayabilir.

Bir C fonksiyonu başka bir C fonksiyonunu çağırdığında, çağrıdaki argümanlar ve dönüş adresi, *yığıt* adı verilen bir veri yapısı içinde çağrılan fonksiyona geçirilir ve

çağrılan fonksiyon değişiklikleri bu yığıt içinde yapar. Bundan dolayı, çağırıcı ile çağrılan fonksiyonun bu yığıtı nasıl değiştirdiğini anlamak çok önemlidir.

Yığıt, belleğin büyük adreslerinden küçük adreslerine doğru büyür. C derleyicisi, çağrıdaki argümanları argüman listesindeki sıralarına göre fakat sağdan başlayarak yığıta yerleştirir. Böylece, yığıta en son itilen ve bundan dolayı en düşük adreste bulunan argüman en soldaki, yani argüman listesindeki ilk argümandır. Çağırıcı yordam tarafından yığıta en son itilen şey ise dönüş adresidir. Çağrılan yordam Taban Yazmaç BP'nin içeriğini saklayarak, yerel verileri için yer ayırarak ve, SI ile DI gibi, içerikleri değişebilecek olan yazmaçların içeriklerini saklayarak yığıtta değişiklikler yapar. Aşağıdaki çizelge yığıtın içeriğini göstermektedir.

	<i>n</i> 'inci argüman	(büyük adres)
çağırıcı fonksiyon tarafından değiştirilen yığıt bölümü	<i>n</i> -1'inci argüman ... birinci argüman dönüş adresi	
çağırılmış fonksiyon tarafından değiştirilen yığıt bölümü	saklanmış BP yerel veriler saklanmış yazmaçlar	← BP ← SP (küçük adres)

Her argüman için kullanılan bayt sayısı argümanın tipi ve, değerle veya referansla çağrı gibi, argümanı geçirmek için kullanılan yöntemle bağlıdır. Referansla çağrı durumunda, bir adres geçirildiğine göre, bu sayı adresin yakın/uzak niteliğine de bağlıdır. Aynı durum dönüş adresi için de sözkonusudur. Her biri için kullanılan bayt sayısı şöyledir:

Yöntem	Tip	Bayt
değerle	short, int	2
değerle	long	4
değerle	float	4
değerle	double	8
referansla	yakın	2
referansla	uzak	4
dönüş adresi	yakın	2
dönüş adresi	uzak	4

Program minik, küçük veya kısa bellek modelinde derlenmişse dönüş adresi yakın, aksi takdirde uzaktır. Uzak adresler *kesim : uzaklık* şeklinde olurlar.

C'de dizi dışındaki tüm veri yapılarının değerle, dizilerin ise referansla geçirildiğini anımsayın. Eğer bir dizinin değerle geçirilmesi isteniyorsa, o zaman değerle geçirilen bir yapının tek üyesi olarak tanımlanmalıdır.

Çağrılan yordam, işini bitirdikten sonra, hesapladığı değeri şu şekilde döndürür:

Döndürülen**Değer**

1 bayt
2 bayt
4 bayt
> 4 bayt

Yazmaç

AL
AX
yüksek baytlar DX'te, düşük baytlar AX'te
değerin adresi DX:AX'te

Aşağıdaki örnekte, birleştirici dilde yazılmış xyz adındaki fonksiyonu çağıran bir C main fonksiyonu gösterilmektedir:

```
extern int xyz (int, int);
```

```
main (...
```

```
{
```

```
...
```

```
xyz (3,5);
```

```
...
```

```
}
```

```
.MODEL SMALL
```

```
.CODE
```

```
    PUBLIC _xyz
```

```
    PROC
```

```
_xyz    push    bp                ; Taban gösterge yazmacini sakla
        mov     bp,sp            ; Yigitta ... boyunda yeni
        sub     sp,...           ; bir taban bolge ac
        push    si                ; SI ve DI yazmaclarini sakla
        push    di
        pushf                    ; Bayrak yazmacini sakla
        mov     ax,[bp+4]        ; Ilk argumani AX'e tasi
        mov     cx,[bp+6]        ; Ikinci argumani CX'e tasi
        ...                     ; xyz'nin hesaplamasi
        ...                     ; gereken seyi hesapla
        mov     ax,...           ; Donus degerini koy
        popf                    ; Bayrak yazmacini geri al
        pop     di                ; DI ve SI yazmaclarini geri al
        pop     si
        mov     sp,bp            ; Yigiti temizle
        pop     bp                ; Eski BP'yi geri al
        ret                     ; Cagirana don
xyz     ENDP
        END
```

Programları ayrı ayrı amaç dosyaları içine derleyin ve birleştirin; daha sonra bu iki dosyayı tek bir yürütülebilir dosya şeklinde bağlayın.

E.2. Satırıcı Birleştiricisinin Kullanılması

Microsoft C Derleyicisinin 8.00 uyarlamasını kullanıyorsanız, bir önceki kısımda anlatıldığı şekilde MASM gibi ayrı bir birleştirici kullanmanıza gerek kalmayabilir. Derleyici içine dahil edilmiş *satırıcı birleştiricisini* kullanabilirsiniz. Bu durumda, amaç dosyaların LINK işlemiyle de uğraşmanıza gerek kalmaz.

Satırıcı birleştiricisinin birçok avantajları vardır: tek bir kaynak dosya içinde hem C hem de birleştirici kodunu yazma; C değişmez, değişken, fonksiyon ve etiket isimlerine birleştirici kod içinden ulaşma; C deyimleri arasına istenilen yere birleştirici kodu ekleme gibi. Ancak, birleştirici kodun kullanılması taşınabilirlik sorunları ortaya çıkarabilir; bunun için birleştirici kodlarının, kaynak dosya içinde, ayrı bir yerde tutulması yararlı olur.

Satırıcı birleştiricisini kullanmak için bir C fonksiyonu içinde uygun yerlere

```
__asm {
    ...
}
```

şeklinde özel bir deyim kullanmanız gerekir. Örneğin

```
__asm {
    mov ah, 2
    mov dl, 7
    int 21h
}
```

deyimi ekrana zil karakterinin yazılmasını sağlar.

Satırıcı birleştiricisinde birçok MASM ifadesi ve 80286 ile 80287 işlemci komutları kullanılabilir. İfadeler içinde kaynak dosyada tanımlanmış C isimleri de kullanılabilir. Açıklamalar, normal olarak satırın sonuna noktalı virgülün arkasına yazılır. Birleştirici kodu içinde `_emit` sözde komutu kullanılarak tek bir bayt tanımlanabilir. Örneğin:

```
_emit 0x66
_emit 0x98
```

sözde komut sırası, satırıcı birleştiricisinin anlamadığı, 80386 CWDE komutunun yerine kullanılabilir.

Satırıcı birleştiricisi kodu içinde AX, BX, CX, DX, ES ve bayrak yazmaçlarını istediğiniz gibi kullanabilirsiniz. Ancak, DI, SI, DS, SS, SP ve BP yazmaçlarıyla (STD ve CLD komutuyla değiştirilebilen) yön bayrağını eski halinde bırakmanız gerekir.

E.3. Bir Örnek—Disket Saklama

Bu kısımda, bir disketi olduğu gibi diske kopyalayıp geri alabilen veya disketle disk dosyasını karşılaştırabilen bir programı inceleyeceğiz. Program bir disketi blok blok okuyup diskteki tek bir dosya içine kopyalamakta veya bu işin tam tersini yapmaktadır.

Bu amaçla diskette istediğimiz bir sektörü okuyup yazabilen bir yönteme gereksinimimiz vardır. MS-DOS kesintilerinden 0x25 ve 0x26 bu işi yaparlar, ancak bir sorunları vardır: Olası bir hata durumunu bayrak yazmacının elde (*carry*) bitinde, hata kodunu ise AX yazmacında döndürürler. Bu arada, bayrak yazmacının eski durumunu da yığıtta saklarlar. Bu durumda, bu kesintilerden döndüğü zaman yığıtta fazla bir sözcük bulunmaktadır. Programa devam etmeden önce bunun geri alınması gerekir. Bu yüzden, bu kesintileri `dos.h` başlık dosyasında tanımlanmış `_int86` fonksiyonunu kullanarak çağıramayız; birleştirici kodu yazmamız gerekir. Aşağıdaki programda `mdisk` fonksiyonu bu işi yapar.

```

1.  /*      Disk üzerinde dosya şeklinde disket saklama. Uyarlama 2.00.
2.  *      (C) Fedon Kadifeli 1989-1993.
3.  *      Derleyici: Microsoft C. Uyarlama 8.00.
4.  */
5.
6.  #include <stdio.h>
7.  #include <stdlib.h>
8.  #include <conio.h>      /* MS-CL */
9.
10. #define M_OKU    (0x25)    /* mutlak disk okuma kesinti kodu */
11. #define M_YAZ    (0x26)    /* mutlak disk yazma kesinti kodu */
12. #define SEKTOR   ((size_t)512) /* sektor boyu */
13. #define IOS      ((size_t)48)  /* bellek tampon boyu (sektor) */
14. #define BBS      (1024/SEKTOR) /* 1K'lik bloklar */
15. #define MAKS_DB  (66)        /* en uzun dosya ismi boyu */
16.
17. typedef unsigned char BAYT; /* 8 bit */
18.
19. BAYT tam1 [IOS*SEKTOR], tam2 [IOS*SEKTOR];
20.
21. /* karsilastirma islemi */
22. int esit (register BAYT * a, register BAYT * b, register size_t u)
23. {
24.     while (u-->0)
25.         if (*a++ != *b++)
26.             return 0;    /* uyusmuyor */
27.     return 1;           /* uyusuyor */
28. } /* esit */
29.

```

```

30.  /* mutlak disk girdi/cikti */
31.  size_t mdisk (BAYT kesno, BAYT surno, size_t seksay, size_t sekno,
32.  BAYT * aktadr)
33.  {
34.  size_t dondeg;
35.  __asm {
36.      push    di          ; yazmaclari
37.      push    si          ; sakla
38.      push    bp
39.      mov     ah,kesno    ; kesinti numarası (M_OKU, M_YAZ)
40.      mov     al,surno    ; disket surucu numarası
41.      sub     al,'A'      ; 'A' -> 0, 'B' -> 1
42.      mov     cx,seksay   ; sektor sayisi
43.      mov     dx,sekno    ; sektor numarası
44.      mov     bx,aktadr   ; aktarma adresi
45.      cmp     ah,M_OKU
46.      jne     else        ; eger M_YAZ ise
47.      int     M_OKU
48.      jmp     SHORT devam
49.  else: int     M_YAZ
50.  devam: pushf
51.      pop     ax          ; bayraklari al
52.      and     ax,1        ; hata biti (kesinti donus degeri)
53.      popf
54.      pop     bp          ; bayraklari geri al
55.      pop     si          ; yazmaclari
56.      pop     di          ; geri al
57.      mov     dondeg,ax
58.  }
59.  return !dondeg;         /* 0 hata; 1 tamam */
60.  } /* mdisk */
61.
62.  /* menu seceneklerini sor */
63.  int mensor (void)
64.  {
65.  int sec;
66.  static char * scnkler [] = {
67.      "\n",
68.      " 1. Sakla\tDisket --> Disk",
69.      " 2. Yukle\tDisket <-- Disk",
70.      " 3. Karşilaştır\tDisket <-> Disk" };
71.
72.  printf("\n\n\t%s\n\t%s\n\t%s\n\nSecenek:",
73.      scnkler[1], scnkler[2], scnkler[3]);
74.  if ((sec = _getch()) == '1' || sec == 'S' || sec == 's')
75.      sec = 1;
76.  else if (sec == '2' || sec == 'Y' || sec == 'y')
77.      sec = 2;
78.  else if (sec == '3' || sec == 'K' || sec == 'k')
79.      sec = 3;
80.  else
81.      sec = 0;
82.  puts(scnkler[sec]);
83.  return sec;
84.  } /* mensor */
85.
86.  /* kullanılacak disket surucusunu sor */
87.  int sursor (void)
88.  {
89.  int sur;
90.

```

```

1. printf("Disket surucusu (A/B): ");
2. if ((sur = toupper(_getch())) != 'A' && sur != 'B')
3.     sur = 0;
4. printf(sur ? "%c:\n" : "\n", sur);
5. return sur;
6. } /* sursor */
7.
8. /* kullanılacak disk dosyasinin adini sor */
9. char * dossor (void)
10. {
11.     static char da [MAKS_DB+3] = { MAKS_DB+1 };
12.
13.     printf("Disk dosya adi: ");
14.     _cgets(da);
15.     putchar('\n');
16.     return da[2] ? da+2 : NULL;
17. } /* dossor */
18.
19. int main (void)
20. {
21.     int secenek, surucu;
22.     char * da;
23.     FILE * dg;
24.     int b;
25.     size_t i, d1, d2;
26.
27.     printf("Disk üzerinde dosya seklinde disket saklama."
28.           " Uyarlama 2.00.\n(C) Fedon Kadifeli 1989-1993.\n");
29.     while ((secenek = mensor()) != 0)
30.         while ((surucu = sursor()) != 0)
31.             while ((da = dossor()) != NULL) {
32.                 i = 0;
33.                 d1 = 1;
34.                 b = 0;
35.
36.                 /* 1 nolu secenek : Disket --> Disk */
37.                 if (secenek == 1) {
38.                     if ((dg = fopen(da, "rb")) != NULL) {
39.                         fclose(dg);
40.                         printf("%s' dosyasi zaten var!\n", da);
41.                         continue;
42.                     }
43.                     if ((dg = fopen(da, "wb")) == NULL) {
44.                         printf("%s' dosyasi acilamiyor!\n", da);
45.                         continue;
46.                     }
47.                     while (mdisk(M OKU, (BAYT)surucu, IOS, i, tam1) &&
48.                           (d1=fwrite((void *)tam1, SEKTOR, IOS, dg)==IOS) != 0)
49.                         printf("%r%d", (i+=IOS)/BBS);
50.                     printf(d1 ? (i?" adet blok '%s' dosyasina aktarildi.\n"
51.                                     : ". Disket okunamadi!\a\n")
52.                             : ". '%s' dosyasinda yazma hatasi!\a\n", da);
53.                 } else {
54.                     /* 2 veya 3 nolu secenek */
55.                     if ((dg = fopen(da, "rb")) == NULL) {
56.                         printf("%s' dosyasi acilamiyor!\n", da);
57.                         continue;
58.                     }
59.                 }
60.             }
61.         }
62.     }
63. }

```

```

148.      /* 2 nolu secenek : Disket <-- Disk */
149.      if (secenek == 2) {
150.          while (fread((void *)tam1, SEKTOR, IOS, dg) == IOS &&
151.                  (dl=mdisk(M_YAZ, (BAYT)surucu, IOS, i, tam1)) != 0)
152.              printf("\r%d", (i+=IOS)/BBS);
153.          printf(dl ? " adet blok diskete aktarıldı.\n"
154.                  : ". Diskette yazma hatası!\a\n");
155.      } else {
156.          /* 3 nolu secenek : Disket <-> Disk */
157.          while ((dl=mdisk(M_OKU, (BAYT)surucu, IOS, i, tam1)) &
158.                  (d2=fread((void *)tam2, SEKTOR, IOS, dg)==IOS) &&
159.                  (b=esit(tam1, tam2, IOS*SEKTOR)) != 0)
160.              printf("\r%d", (i+=IOS)/BBS);
161.          printf(
162.              !b ? ". Karsilastirma hatası!\a\n" :
163.              dl ? ". Dosya disketten kısa!\a\n" :
164.              d2 ? ". Disket dosyadan kısa!\a\n" :
165.              " adet blok uyusuyor.\n");
166.      }
167.      } /* if */
168.      fclose(dg);
169.      } /* while */
170.      printf("\nİşlem tamamlandı\n");
171.      return 0;
172.  } /* main */

```

esit fonksiyonu (Satır 22-28) iki tampon bölgeyi karşılaştırır, eşitse 1 döndürür, aksi takdirde 0 döndürür.

mdisk fonksiyonu (Satır 31-60) surnoda verilen sürücüdeki—bu programda, surno 'A' veya 'B' değerlerini alabilmektedir—sekno başlangıç sektör (kesim) numarasından seksay sektör sayısı kadar sektör üzerinde işlem yapar. kesno M_OKU'ya eşitse disketten bu sektörleri okuyup aktadır tampon bölgesine aktarır; kesno M_YAZ'a eşitse aktadır tampon bölgesindeki baytları disketteki ilgili sektörlerle yazdır. Bu fonksiyon, yukarıda anlatıldığı gibi ilgili kesintiye çağırıldıktan sonra, yığıtta saklanmış bulunan bayrakları bayrak yazmacına geri alır (Satır 53).

Menu seçeneklerini kullanıcıya mensör fonksiyonu (Satır 63-84) gösterir ve istenilen seçeneği kullanıcıdan alır. *Bir numaralı seçenek*, programın disketi sektör sektör okuyup, adı kullanıcıdan alınan, bir disk dosyası içine kopyalamasını sağlar. *İki numaralı seçenek* bu işin tersini yapar; daha önce, bir numaralı seçenek kullanılarak, disk dosyası içine aktarılmış olan bir disket görüntüsünün sektör sektör okunup formatlanmış bir diskete aktarılmasını sağlar. *Üç nolu seçenek* diskteki dosya ile disketi karşılaştırır; bu seçenek, bir veya iki nolu seçenekler kullanılarak yapılan bir kopyalama işleminden sonra, işlemin doğruluğunu sınamak için kullanılabilir.

Kullanılacak disket sürücüsü ("A:" veya "B:") sursor fonksiyonu (Satır 87-96) aracılığıyla kullanıcıdan alınır. Kullanılacak disk dosyasının adı ise dossor fonksiyonu (Satır 99-107) tarafından alınır. Burada, dosya adını okumak için standart olmayan _cgets fonksiyonu kullanılmıştır. _cgets fonksiyonu kontrollü bir şekilde klavyeden bir satır okumak için kullanılabilir.

Ana fonksiyon içiçe üç döngüden oluşur; bunlar, sırasıyla, kullanıcıdan menü seçeneğini, kullanılacak sürücüyü ve dosya adını alırlar. Her bir menü seçeneği için

yapılan işlem temelde aynıdır: Bir döngü içinde disk dosyası ve/veya disketten bilgi okunup gerekli işlem yapılır. (Karşılaştırma seçeneği kısmında, satır 157'nin sonunda neden && değil de & kullanılmıştır?)

Not: Bu program kullanılırken dikkat edilmesi gereken bir nokta disket kapasitesidir. İşlenen bilgi miktarı kilobayt cinsinden ekranda görüntülenir. Örneğin 1.44 megabaytlık bir disket ile işlem yapıldığında ekrandaki sayının son durumu 1440 olmalıdır. Programda kullanılan kesintinin gerçek disket kapasitesini anlaması olası değildir; MS-DOS ortamında iken en son yapılan disket işlemindeki disket kapasitesi kullanılır. Bu yüzden, herhangi bir sorun çıkmaması için, program kullanılmadan önce, kullanılacak disket tipinden bir disket sürücüyü takılıp, örneğin “DIR A:” veya “CHKDSK A:” şeklinde bir MS-DOS komutu çalıştırılmalıdır. Bundan sonra, program yürütülmesi esnasında, kullanılan disketlerin kapasitesi değişmediği sürece, herhangi bir sorun çıkmayacaktır. Dikkat edilmesi gereken bir diğer nokta da, kullanılan disketlerin formatlı olması ve bozuk sektörlerin bulunmamasıdır.

EK F: STANDART C PROGRAMLAMA DİLİNİN DİĞER ÖZELLİKLERİ

Bu ekte, ANSI Standardında olmasına rağmen, henüz yaygın olarak bilinmeyen ve pek kullanılmayan bazı özelliklerden söz edilmektedir. Ayrıca kitapta sözü edilmiş standart kütüphane fonksiyonları hakkında (dönüş tipi ve argümanlarının tipi gibi) bazı detaylar için son kısma başvurulabilir.

F.1. C Dünyanın Her Yerinde—Yöreler

Bilginin gösterim yolları ülkeden ülkeye değişir. Örneğin, tarih yazılırken, Türkçe’de önce gün, sonra ay yazılır. Bazı batı ülkelerinde ise bunun tersi yapılır. İngilizce’de ondalık sayıları ayırmak için nokta kullanılırken, başka ülkelerde virgül kullanılır. Ve, en önemlisi de, her ülkenin kendine özgü harfleri vardır ve bütün dillerin bütün karakterlerini ifade edebilecek tek bir karakter takımı yoktur.

Bu konuya bir çözüm getirmek için *yöreler* kavramı getirilmiştir. Bir yöre, belirli bir ülke veya dil için yukarıdaki paragrafta sözü edilen türden bilgileri içeren bir bilgi paketidir. Yöreler, ctype fonksiyonlarını, yazı yazma yönünü, saat-tarih biçimlerini, karakterlerin sırasını ve ondalıkları ayırmak için kullanılan karakteri belirlerler. Yeni bir yöre belirterek, printf gibi, birçok standart fonksiyonu etkilemek olasıdır.

Kullanılmakta olan yöre ile ilgili bilgileri lconv yapısı tutar ve buna bir gösterge elde etmek için localeconv adlı fonksiyonu çağırmak gerekir. Yürütme esnasında yöreyi belirtmek için, şu iki argümanı olan setlocale adlı fonksiyonu çağırırsınız: Değiştirmek istediğiniz özelliklerden oluşan grup (LC_COLLATE [karakter sırası], LC_CTYPE [karakter tipi], LC_MONETARY [para], LC_NUMERIC [sayılar], LC_TIME [zaman], LC_ALL [hepsi]) ve yeni özelliklerin alınacağı yörenin adı. Eğer bütün diller

için yöreler tanımlanmışsa, programın başında kullanıcının milliyetini sorup ondan sonra bu kişinin alıştığı şekilde çalışan bir program yazılabilir. Örneğin,

```
setlocale(LC_ALL, turkish)
```

Standart sadece “C” yöresinin gerçekleştirilmesini gerektirmektedir. Bu da, kitap boyunca anlatılan özelliklerden oluşmaktadır.

F.2. Geniş Karakterler Ve Çokbaytlı Karakterler

char tipinin 256 farklı değer alabilmesi temin edilmiştir. Japonca veya Çince gibi, bazı diller 256’dan fazla simge gerektirmektedir. Böyle dillerle çalışabilmek için, geniş karakterler ve çokbaytlı karakterler kavramı geliştirilmiştir.

Bir *geniş karakter* `wchar_t` adlı standart tipe aittir. Bu tamsayı tipi herhangi bir karakter takımındaki bütün değerleri gösterebilecek kadar büyük olan, normalde 16 bitlik bir tiptir. Geniş karakterlerden oluşan diziler, başlangıçtaki “”’ın hemen önüne bir `L` harfi konarak belirtilir. Karakter değişmezleri için de, benzer şekilde, “”’ın önüne `L` konur.

```
wchar_t gkd[] = L"Bir geniş karakter dizisi";  
wchar_t gk = L'h';
```

Bir *çokbaytlı karakter*, çevresel bir aygıt yazılabilecek, bir veya birden fazla bayttan oluşan, sıradan bir karakter sırasıdır. Çokbaytlı bir karakter dizisinin uzunluğunu elde etmek için `mblen` fonksiyonu kullanılır.

Bu iki karakter türü birbirleriyle yakın ilişkili oldukları için, bunlar arasında dönüştürme yapmayı isteriz. Aşağıdaki fonksiyonlar bu işi yaparlar:

<code>wctomb</code>	bir geniş karakteri bir çokbaytlı karaktere,
<code>mbtowc</code>	bir çokbaytlı karakteri bir geniş karaktere,
<code>wcstombs</code>	bir geniş karakter dizisini bir çokbaytlı karakter dizisine,
<code>mbstowcs</code>	bir çokbaytlı karakter dizisini bir geniş karakter dizisine dönüştürür.

F.3. Üçlü Karakterler

Bazı klavyelerde C programlarında kullanılan bazı önemli karakterler yazılamayabilir. Böyle dokuz karakteri ifade etmek için dokuz özel karakter sırası kullanılır. Bütün bu sıralar üç karakterden oluştuğu için bunlara *üçlü karakter* denir. Önilemci bir üçlü karakter ile karşılaştığında yerine onun ifade ettiği karakteri koyar. Üçlü karakterler ve onların eşdeğerleri şunlardır:

Üçlü Karakter

??=

?? (

??/

??)

??'

??<

??!

??>

??-

Karakter

(sayı işareti)

[(sol köşeli parantez)

\ (ters bölü)

] (sağ köşeli parantez)

^ (uzatma işareti)

{ (sol çengelli parantez)

| (dikey çizgi)

} (sağ çengelli parantez)

~ (inceltme işareti)

Programın *her yerinde* üçlü karakter yerine ilgili karakterin konulduğuna dikkat edin. Yani,

```
printf("Devam?? (e/h) ")
```

fonksiyon çağrısı

```
Devam[e/h]
```

basacaktır.

F.4. Zaman Fonksiyonları

Zamanla ilgili dokuz tane standart fonksiyon bulunmaktadır. Gerekli bildirimler `time.h` adlı dosyada bulunmaktadır.

Şu anki takvim zamanını elde etmek için `time` fonksiyonunu çağırın. Sonuç `time_t` tipindedir. Eğer zaman elde edilebilir değilse, bu fonksiyon `-1` döndürür.

`difftime` fonksiyonu iki `time_t` argümanı alır ve bunlar arasındaki zaman farkının saniye cinsinden verildiği bir **double** döndürür.

Takvim zamanını yerel zamana dönüştürmek için, `localtime` fonksiyonunu kullanın.

`gmtime` fonksiyonu takvim zamanını (eskiden Greenwich Ortalama Saati olarak bilinen) Düzenlenmiş Evrensel Saate çevirmek için kullanılır.

Zamanı görüntülemek için önce bir karakter dizisine çevirmeniz gerekir. Bu `asctime` fonksiyonu ile yerine getirilir. Argümanı tarafından işaret edilen yapı içindeki zamanı bir karakter dizisine dönüştürür.

```
ctime(zg)
```

çağrısı

```
asctime(localtime(zg))
```

çağrısına eşdeğerdir. Yani yerel zamanı bir karakter dizisine dönüştürür.

`mktime` fonksiyonu argümanı tarafından işaret edilen yapı içindeki yerel zamanı takvim zamanına dönüştürür ve sonucu `time_t` şeklinde döndürür. Eğer sonuç ifade edilemezse, `-1` döndürülür.

Program çalışmaya başladığından beri ne kadar işlemci zamanı kullandığını öğrenmek için, `clock_t` tipinden sonuç döndüren, `clock` fonksiyonunu çağırın.

En son olarak, şu anki yöreye uygun olarak tarih ve saatler içeren karakter dizileri hazırlamak için `strftime` fonksiyonu kullanılabilir. Dört tane argümanı vardır: Sonuçta uygun formatlanmış karakter dizisini içerecek bölgeye bir karakter göstergesi; bu karakter dizisine konulacak en fazla karakter sayısını gösteren `size_t` tipinde bir sayı; `printf`'tekine benzeyen bir kontrol karakter dizisi ve zaman bilgisini içeren yapıya bir gösterge. Normalde olduğu gibi önüne `%` konan, kontrol karakter dizisindeki dönüşüm tanımlamaları zaman unsurlarını belirtirler ve sonuçtaki karakter dizisinde kullanılmakta olan yöreye uygun olarak biçimlendirilmiş ilgili zaman bilgileri ile değiştirilirler. Diğer karakterler, tıpkı `printf` gibi, değiştirilmeden bırakılırlar. Dönüşüm tanımlamaları ve anlamları şöyledir:

Dönüşüm Karakteri

Anlamı

a	kısaltılmış gün adı
A	tam gün adı
b	kısaltılmış ay adı
B	tam ay adı
c	yerel tarih ve saat gösterimi
d	ay içindeki gün
H	saat (24 saatlik)
I	saat (12 saatlik)
j	yıl içindeki gün
m	ay
M	dakika
p	AM (öğleden önce) veya PM (öğleden sonra) ifadelerinin yerel eşdeğeri
S	saniye
U	(Pazar gününden başlamak koşuluyla) yıl içindeki hafta sayısı
w	hafta içindeki gün
W	(Pazartesi gününden başlamak koşuluyla) yıl içindeki hafta sayısı
x	yerel tarih gösterimi
X	yerel saat gösterimi
Y	yılın son iki rakamı
Y	yıl
Z	saat dilimi adı
%	%

`strftime`, en sondaki `\0` hariç, oluşturulan karakter sayısını döndürür. Eğer bu sayı ikinci argümandan büyükse, 0 döndürülür.

F.5. Standart Başlık Dosyaları

ANSI Standardı, değişmez, makro ve tip tanımlayan; değişken bildiren ve tüm standart kütüphane fonksiyonlarının prototiplerini içeren 15 tane başlık dosyası tanımlar. Bir ANSI fonksiyonu çağırmadan önce ilgili başlık dosyası `#include` emri ile programa dahil edilmelidir. Bazı gerçekleştirmelerde, bazı “fonksiyonlar” önışlemci makroları şeklinde tanımlanmışlardır. Kullanıcı açısından bir fonksiyonun gerçekten fonksiyon şeklinde mi yoksa makro şeklinde mi tanımlanmış olduğu, bir istisna dışında, farketmez: Makro argümanları, parametre makro içinde kaç defa kullanılırsa, o kadar kere hesaplanırlar. Eğer argümanlarda yan etkili işleç veya fonksiyonlar kullanılırsa, bu sorun yaratabilir. Örneğin, bazı uygulamalarda `tolower` ve `toupper` makro şeklinde `ctype.h` içinde tanımlanırken, `stdlib.h` içinde ilgili fonksiyonların bildirimi yapılmaktadır. Eğer yan etkisi olmayan işleçler içeren argümanlar verecekseniz, daha hızlı olacağı için, makroları kullanmak daha akılcıdır; ancak argümanlar olası yan etkileri olan karmaşık ifadeler ise, argümanların bir defa hesaplanacakları ilgili fonksiyonları çağırın.

assert.h—Diyagnostikler

Bu dosyada, yürütme esnasında bir savı (mantıksal ifadeyi) kontrol edip, eğer yanlışsa bir diyagnostik görüntüleyen bir “fonksiyon” tanımı vardır:

```
void                assert(int)
```

Eğer bu dosya programa içerilmeden önce `NDEBUG` `#define` ile tanımlanmışsa, `assert` çağırısı dikkate alınmaz; aksi takdirde aşağıdakine benzer bir durum gerçekleşir:

```
#define assert(exp) { \
if (!(exp)) { \
    fprintf(stderr, "Assertion failed: %s, file %s, line %d\n", \
        #exp, __FILE__, __LINE__); \
    abort(); \
} \
}
```

ctype.h—Karakter Kontrolleri Ve Dönüşümleri

Bu dosya karakterleri kontrol eden veya değiştiren fonksiyonlar ve/veya makrolar bildirir. Test fonksiyonları veya makroları, doğru için sıfırdan farklı herhangi bir tamsayı, aksi takdirde sıfır döndürürler.

```
int                isalnum(int)
int                isalpha(int)
int                isctrl(int)
```

int	isdigit (int)
int	isgraph (int)
int	islower (int)
int	isprint (int)
int	ispunct (int)
int	isspace (int)
int	isupper (int)
int	isxdigit (int)
int	tolower (int)
int	toupper (int)

errno.h—Hatalar

Bu dosya **errno** değişkeni için bir bildirim içerir. Ayrıca çeşitli kütüphane fonksiyonları tarafından bir hata durumunu göstermek için, **EDOM**, **ERANGE** gibi, bu değışkene verilebilecek bazı sıfırdan farklı değışmezler tanımlar.

float.h—Kayan Noktalı Değerler İçin Sınırlar

Bu dosya, kayan noktalı (yani **float**, **double** ve **long double**) değerler için bazı değışmezler tanımlar. Standart tarafından kabul edilebilen en küçük değerler parantez içinde gösterilmiştir.

FLT_DIG	duyarlı ondalık rakam sayısı (6)
FLT_EPSILON	$1.0 + \text{FLT_EPSILON} \neq 1.0$ şekilde en küçük sayı ($1E-5$)
FLT_MANT_DIG	FLT_RADIX tabanına göre mantisteki rakam sayısı
FLT_MAX	en büyük değer ($1E+37$)
FLT_MAX_10_EXP	en büyük ondalık üs
FLT_MAX_EXP	en büyük üs
FLT_MIN	en küçük düzgünlenmiş pozitif değer ($1E-37$)
FLT_MIN_10_EXP	en küçük ondalık üs
FLT_MIN_EXP	en küçük üs
FLT_RADIX	üssün taban rakamı (2)
FLT_ROUNDS	toplamada yuvarlama
DBL_DIG	duyarlı ondalık rakam sayısı (10)
DBL_EPSILON	$1.0 + \text{DBL_EPSILON} \neq 1.0$ şekilde en küçük sayı ($1E-9$)
DBL_MANT_DIG	DBL_RADIX tabanına göre mantisteki rakam sayısı
DBL_MAX	en büyük değer ($1E+37$)
DBL_MAX_10_EXP	en büyük ondalık üs
DBL_MAX_EXP	en büyük üs
DBL_MIN	en küçük düzgünlenmiş pozitif değer ($1E-37$)
DBL_MIN_10_EXP	en küçük ondalık üs
DBL_MIN_EXP	en küçük üs
DBL_RADIX	üssün taban rakamı
DBL_ROUNDS	toplamada yuvarlama

LDBL_DIG	duyarlı ondalık rakam sayısı
LDBL_EPSILON	$1.0 + \text{LDBL_EPSILON}! = 1.0$ şekilde en küçük sayı
LDBL_MANT_DIG	LDBL_RADIX tabanına göre mantisteki rakam sayısı
LDBL_MAX	en büyük değer
LDBL_MAX_10_EXP	en büyük ondalık üs
LDBL_MAX_EXP	en büyük üs
LDBL_MIN	en küçük düzgünlenmiş pozitif değer
LDBL_MIN_10_EXP	en küçük ondalık üs
LDBL_MIN_EXP	en küçük üs
LDBL_RADIX	üssün taban rakamı
LDBL_ROUNDS	toplamada yuvarlama

limits.h—Tamsayı Değerleri İçin Sınırlar

Bu dosya, çeşitli boylardaki tamsayılar için bazı üst ve alt sınırlarla ilgili değişmezler tanımlar. Kabul edilebilir en düşük değerler parantez içinde gösterilmiştir. Yani, bir derleyici için INT_MAX 32767'den büyük olabilir, fakat Standarda uygun hiçbir derleyicinin bu sayıdan küçük bir INT_MAX sağlamasına izin verilmez.

CHAR_BIT	bir char 'daki bit sayısı (8)
CHAR_MAX	en büyük char değeri (UCHAR_MAX veya SCHAR_MAX)
CHAR_MIN	en küçük char değeri (0 veya SCHAR_MIN)
INT_MAX	en büyük (signed) int değeri (+32767)
INT_MIN	en küçük (signed) int değeri (-32767)
LONG_MAX	en büyük (signed) long değeri (+2147483647)
LONG_MIN	en küçük (signed) long değeri (-2147483647)
SCHAR_MAX	en büyük signed char değeri (+127)
SCHAR_MIN	en küçük signed char değeri (-127)
SHRT_MAX	en büyük (signed) short değeri (+32767)
SHRT_MIN	en küçük (signed) short değeri (-32767)
UCHAR_MAX	en büyük unsigned char değeri (255U)
UINT_MAX	en büyük unsigned int değeri (0xFFFF)
ULONG_MAX	en büyük unsigned long değeri (0xFFFFFFFF)
USHRT_MAX	en büyük unsigned short değeri (0xFFFF)
MB_LEN_MAX	bir çokbaytlı karakter içindeki en büyük bayt sayısı

locale.h—Yöre Fonksiyonları

Bu dosya, “**struct lconv**” için bir tip tanımlaması, LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME değişmezlerinin tanımlarını ve aşağıdaki fonksiyon prototiplerini içerir:

```
struct lconv * localeconv(void)
char * setlocale(int, const char *)
```

math.h—Matematiksel Yordamlar

Bu dosya, birtakım kayan noktalı matematik yordamları tarafından hata durumunda döndürülen bir değeri tutmak için HUGE_VAL adlı **double** değişkenini bildirir. Ayrıca, alan hatası veya taşma durumlarını göstermek için sırasıyla EDOM veya ERANGE değeri errno değişkenine atanır. Fonksiyon prototipleri şöyledir:

```
double      acos(double)
double      asin(double)
double      atan(double)
double      atan2(double, double)
double      ceil(double)
double      cos(double)
double      cosh(double)
double      exp(double)
double      fabs(double)
double      floor(double)
double      fmod(double, double)
double      frexp(double, int *)
double      ldexp(double, int)
double      log(double)
double      log10(double)
double      modf(double, double *)
double      pow(double, double)
double      sin(double)
double      sinh(double)
double      sqrt(double)
double      tan(double)
double      tanh(double)
```

setjmp.h—Yerel Olmayan Atlamalar

Bu dosya, program durumunu saklamak ve tekrar yüklemek için setjmp makrosu ile longjmp fonksiyonunun kullandığı jmp_buf adlı, makineye bağlı, bir tampon bölge için bir tip tanımlar ve ayrıca bu yordamların bildirimini içerir.

```
void      longjmp(jmp_buf, int)
int      setjmp(jmp_buf)
```

signal.h—Sinyaller

Bu dosya, yürütme esnasında kesintileri veya hataları işlemek için kullanılan fonksiyonlar bildirir.

Sinyal tipleri:

SIGABRT	abort çağrısı ile tetiklenen düzensiz sonuçlanma
SIGFPE	kayan noktalı işlem hatası
SIGILL	hatalı komut—geçersiz fonksiyon imgesi
SIGINT	CONTROL+C kesintisi—etkileşimli dikkat
SIGSEGV	bellek sınırları dışında erişim
SIGTERM	öldürme sonucu yazılım sonlandırma sinyali

Sinyal işleyiş kodları:

SIG_DFL	varsayılan signal işleyişi
SIG_IGN	dikkate almama

Hata durumunda signal çağrısı tarafından döndürülen sinyal hata değeri:

SIG_ERR	signal hata değeri
---------	--------------------

Fonksiyon prototipleri:

```
int      raise(int)
void     (* signal(int, void (*)(int))) (int)
```

stdarg.h—Değişken Argüman Listeleri

Bu dosya, değişken sayıda argüman alabilen fonksiyonların argümanlarına erişebilmesi için standart yöntemler tanımlar. `va_list` tipi bir gösterge olarak tanımlanmıştır.

```
tip      va_arg(va_list, tip)
void     va_end(va_list)
void     va_start(va_list, son_argüman)
```

stddef.h—Standart Sistem Tanımları

Bu dosya, NULL gösterge değeri, `errno` dışsal değişkenine bir referans ve uygulamaya bağlı olarak değişen `ptrdiff_t` (iki göstergenin farkı alındığında elde edilen *işaretli* değerin tipi), `size_t` (`sizeof` işleci tarafından döndürülen *işaretsiz* değerin tipi) ve `wchar_t` (geniş bir karakterin tipi) için tanımlar içermektedir.

stdio.h—Standart Girdi Ve Çıktı

Bu dosya, standart girdi/çıktı yordamları tarafından kullanılan yapılar, değerler, makrolar ve fonksiyonlar tanımlar. `size_t`, `va_list`, `fpos_t` ve `FILE` adlı tipler; `BUFSIZ`, `EOF`, `NULL`, `FILENAME_MAX` (dosya isimlerinde en büyük karakter sayısı), `FOPEN_MAX` (aynı anda açık olabilecek en çok dosya sayısı), `P_tmpnam` (geçici dosyaların açılabilceği altdizin), `L_tmpnam` (geçici dosya isimlerinin uzunluğu), `TMP_MAX` (açılacak en çok farklı geçici dosya isimleri sayısı), `SEEK_CUR`, `SEEK_END`, `SEEK_SET`, `stdin`, `stdout`, `stderr`, `_IOFBF`, `_IOLBF` ve `_IONBF` değişmezleri bu dosyada tanımlanmışlardır. Fonksiyon prototipleri:

```

void          clearerr(FILE *)
int           fclose(FILE *)
int           feof(FILE *)
int           ferror(FILE *)
int           fflush(FILE *)
int           fgetc(FILE *)
int           fgetpos(FILE *, fpos_t *)
char *        fgets(char *, int, FILE *)
FILE *        fopen(const char *, const char *)
int           fprintf(FILE *, const char *, ...)
int           fputc(int, FILE *)
int           fputs(const char *, FILE *)
size_t        fread(void *, size_t, size_t, FILE *)
FILE *        freopen(const char *, const char *,
FILE *)

int           fscanf(FILE *, const char *, ...)
int           fseek(FILE *, long, int)
int           fsetpos(FILE *, const fpos_t *)
long          ftell(FILE *)
size_t        fwrite(const void *, size_t, size_t,
FILE *)

int           getc(FILE *)
int           getchar(void)
char *        gets(char *)
void          perror(const char *)
int           printf(const char *, ...)
int           putc(int, FILE *)
int           putchar(int)
int           puts(const char *)
int           remove(const char *)
int           rename(const char *, const char *)
void          rewind(FILE *)
int           scanf(const char *, ...)
void          setbuf(FILE *, char *)
int           setvbuf(FILE *, char *, int, size_t)
int           sprintf(char *, const char *, ...)
int           sscanf(const char *, const char *, ...)
FILE *        tmpfile(void)
char *        tmpnam(char *)
int           ungetc(int, FILE *)
int           vfprintf(FILE *, const char *, va_list)
int           vprintf(const char *, va_list)
int           vsprintf(char *, const char *, va_list)

```

stdlib.h—Sıkça Kullanılan Kütüphane Fonksiyonları

Bu dosyada, size_t, wchar_t, div_t (div'den döndürülen yapı) ve ldiv_t (ldiv'den döndürülen yapı) tipleri ve NULL, RAND_MAX (rand fonksiyonundan

döndürülen en büyük değer), EXIT_SUCCESS (programın başarılı exit kodu) ve EXIT_FAILURE (programın başarısız exit kodu) değişmezleri için tanımlarla MB_CUR_MAX (şu anki yöreye göre bir çokbaytlı karakter içindeki en büyük bayt sayısı) ve errno değişkenleri için bildirimler bulunmaktadır. Fonksiyon prototipleri:

```

void          abort(void)
int           abs(int)
int           atexit(void (*)(void))
double        atof(const char *)
int           atoi(const char *)
long          atol(const char *)
void *        bsearch(const void *, const void *,
                      size_t, size_t, int (*)(const void *,
                      const void *))
void *        calloc(size_t, size_t)
div_t         div(int, int)
void          exit(int)
void          free(void *)
char *        getenv(const char *)
long          labs(long)
ldiv_t        ldiv(long, long)
void *        malloc(size_t)
int           mblen(const char *, size_t)
size_t        mbstowcs(wchar_t *, const char *, size_t)
int           mbtowc(wchar_t *, const char *, size_t)
void          perror(const char *)
void          qsort(void *, size_t, size_t,
                    int (*)(const void *, const void *))
int           rand(void)
void *        realloc(void *, size_t)
void          srand(unsigned int)
double        strtod(const char *, char **)
long          strtol(const char *, char **, int)
unsigned long strtoul(const char *, char **, int)
int           system(const char *)
size_t        wcstombs(char *, const wchar_t *, size_t)
int           wctomb(char *, wchar_t)

```

Eğer tolower makrosu #define ile tanımlanmamışsa,

```
int          tolower(int)
```

Eğer toupper makrosu #define ile tanımlanmamışsa,

```
int          toupper(int)
```

string.h—Karakter Dizileri İşleyen Fonksiyonlar

Bu dosya NULL değişmezi ile size_t için bir tip tanımı ve aşağıdaki fonksiyon prototiplerini içerir:

```

void *      memchr(const void *, int, size_t)
int         memcmp(const void *, const void *, size_t)
void *      memcpy(void *, const void *, size_t)
void *      memmove(void *, const void *, size_t)
void *      memset(void *, int, size_t)
char *      strcat(char *, const char *)
char *      strchr(const char *, int)
int         strcmp(const char *, const char *)
int         strcoll(const char *, const char *)
char *      strcpy(char *, const char *)
size_t      strcspn(const char *, const char *)
char *      strerror(int)
size_t      strlen(const char *)
char *      strncat(char *, const char *, size_t)
int         strncmp(const char *, const char *,
size_t)
char *      strncpy(char *, const char *, size_t)
char *      strpbrk(const char *, const char *)
char *      strrchr(const char *, int)
size_t      strspn(const char *, const char *)
char *      strstr(const char *, const char *)
char *      strtok(char *, const char *)
size_t      strxfrm(char *, const char *, size_t)

```

time.h—Tarih Ve Saat Fonksiyonları

Bu dosya, zaman yordamları için bildirimler ile, `size_t` ve sırasıyla `clock` ve `time` fonksiyonları tarafından döndürülen, uygulamaya bağlı `clock_t` ve `time_t` tiplerinin tanımlarını içerir. Ayrıca, `localtime` ve `gmtime` yordamları tarafından döndürülüp `asctime` tarafından kullanılan yapı şöyle tanımlanmaktadır:

```

struct tm {
    int tm_sec;      /* dakikadan sonraki saniyeler: [0,60] */
    int tm_min;      /* saatten sonraki dakikalar: [0,59] */
    int tm_hour;     /* geceyarısından beri saatler: [0,23] */
    int tm_mday;     /* ayın günü: [1,31] */
    int tm_mon;      /* Ocak'tan beri aylar: [0,11] */
    int tm_year;     /* 1900'dan beri yıllar */
    int tm_wday;     /* Pazar'dan beri günler: [0,6] */
    int tm_yday;     /* Ocak 1'den beri günler: [0,365] */
    int tm_isdst;    /* Yaz Saati Uygulaması bayrağı */
}

```

`CLOCKS_PER_SEC` makrosu, “`clock()/CLOCKS_PER_SEC`” ifadesi aracılığıyla, saniye cinsinden işlemci zamanını elde etmek için kullanılabilir.

Bu dosyada, ayrıca, `ctime` ailesinden yordamlar tarafından kullanılan küresel değişkenler için **extern** bildirimleri de bulunmaktadır:

```

int _daylight; /* eger Yaz Saati Uygulaması kullanı-
               * liyorsa, sifirdan farklı bir deger */
long _timezone; /* saniye cinsinden Düzenlenmiş Evrensel
               * Saat ile yerel saat arasındaki fark */
char * _tzname[2]; /* Standart/Yaz Saati dilim isimleri */

```

Fonksiyon prototipleri:

```

char *      asctime(const struct tm *)
clock_t     clock(void)
char *      ctime(const time_t *)
double      difftime(time_t, time_t)
struct tm * gmtime(const time_t *)
struct tm * localtime(const time_t *)
time_t      mktime(struct tm *)
size_t      strftime(char *, size_t, const char *,
                    const struct tm *)
time_t      time(time_t *)

```

F.6. Çevirme Sınırları

ANSI Standardı, Standart C uyumlu her derleyici tarafından karşılanması gereken birtakım sınırlar tanımlamıştır. Örneğin, tanımlanan bir nesne için izin verilen en büyük boy 32 767 bayttır; ancak belli bir derleyici, 40 000 karakterlik bir dizi gibi daha büyük bir nesneye izin verebilir. Her durumda, taşınabilirliği garantilemek için programınızın aşağıda belirtilmiş “en küçük büyük” sınırlar içinde kalmasını sağlayın:

Bir fonksiyon çağrısındaki argümanlar	31
Bir nesnedeki baytlar	32 767
Bir switch deyimindeki case etiketleri	257
Mantıksal bir kaynak kod satırındaki karakterler	509
Bir karakter dizisi değişmezindeki karakterler	509
Bileşik deyimler, iç içe yazma	15
bir enum ’daki değişmezler	127
do , for ve while , iç içe yazma	15
Tam bir ifade içindeki ifadeler, iç içe yazma	32
Bir çevirme birimi içindeki dışsal tanıttıcı sözcükler	511
Başlık dosyaları, iç içe yazma	8
Bir blok içindeki, etki alanı blok olan, tanıttıcı sözcükler	127
if ve switch deyimleri, iç içe yazma	15
#if , #ifdef ve #ifndef , iç içe yazma	8
Bir çevirme birimi içindeki makro tanıttıcı sözcükleri	1 024
Bir struct veya union içindeki üyeler	127
Bir fonksiyon tanımındaki parametreler	31
Bir makro içindeki parametreler	31
Bir dışsal tanıttıcı sözcük içindeki anlamlı karakterler	8

Bir içsel tanıttıcı sözcük içindeki anlamlı karakterler.....	31
struct veya union tanımları, içiçe yazma	15

EK G: SEÇİLMİŞ PROBLEMLERE YANITLAR

Bu ekte, her bölümün sonunda verilen problemlerden bazıları için yanıtlar verilmiştir. Bazı problemler için değişik çözümler de sözkonusu olabilir. Burada ilk sayı bölümü, ikinci sayı ise problemi belirtir.

1.1. Şunu deneyin

```
void main (void)
{
    int entry;
}
```

1.4. Evet, $x=5$; deyimine eşdeğerdir.

1.5. Evet, boş bir blok deyimidir.

```
1.6. ...
scanf("%f", &x);
printf("%d", (int)x);
...
```

```
2.2. switch (yas)
    case 16 : printf("...");
```

```
2.4. #include <stdio.h>
void main (void)
{
    unsigned i, n;
    scanf("%u", &n);
    for (i=1; i<=n; i++)
        printf("%u\t%g\n", i*i, 1.0/i);
}
```

```

2.6. #include <stdio.h>
void main (void)
{
    int p, c, i;
    c = p = getchar();
    i = 1;
    while (c!=EOF) {
        c = getchar();
        if (c==p)
            i++;
        else {
            if (i==3)
                printf("%c ", p);
            p = c;
            i = 1;
        } /* else */
    } /* while */
}

```

2.8. Satır 32'deki **do**'yu uygun bir şekilde **while**'a çevirin.

3.1. Gösterge aritmetiğine karşılık tamsayı aritmetiği.

```

3.2. #include <stdio.h>
void main (void)
{
    printf("%s-doldurma\n", (-10>>1<0)? "isaret": "sifir");
}

```

```

3.3. #include <stdio.h>
void main (void)
{
    unsigned n, b;
    scanf("%d", &n);
    for (b=0; n!=0; n>>=1)
        b += n&1;
    printf("%d bit\n", b);
}

```

3.4. Eğer a ve b aynı tipten ise içeriklerini değiştirir.

4.1. Hayır! Bundan emin olamayız, çünkü bu, fonksiyon argümanlarının hesaplanma sıralarına bağlıdır.

```

4.2. int maks (int x, ...)
{
    int maks, *xg;
    for (maks = *(xg=&x); *xg != 0; xg++)
        if (maks < *xg)
            maks = *xg;
    return maks;
}

```


Tanımda “...” (üç nokta) geri kalan parametrelerin sayılarının (ve tiplerinin) belirlenmediği anlamına gelir. Ancak, böyle değişken sayıda argümanlı fonksiyon yazmanın taşınabilir yolu bu değildir. Bunun yerine, aşağıda gösterildiği gibi, `stdarg.h` standart başlık dosyasında tanımlanmış bulunan `va_start`, `va_arg` ve `va_end` makroları kullanılmalıdır.

```
#include <stdarg.h>
int maks (int x, ...)
{
    int m;
    va_list xg;          /* geri kalan argümanlara gösterge */
    va_start(xg, x);     /* ilk isimlendirilmemiş argümana
                          * isaret et */
    for (m = x; x != 0; x = va_arg(xg, int))
        if (x > m)
            m = x;       /* yeni enbüyük */
    va_end(xg);          /* donmeden önce temizlik yap */
    return m;
}
```

4.3. İlklenmemiş otomatik değişkenler belirsiz değerler taşır. Bu, daha önce aynı adreste saklanmış olan bir otomatik değişkenin değeri olabilir.

4.4. Sadece ilk iki **register** bildirimi dikkate alınır, geri kalanlar **auto** gibi işlem görürler. Aynı zamanda bir önceki alıştırmaya bakın.

4.5. Girilen satırdaki karakterleri tersten basar.

5.1. (a) sıfır, yani ilk sayıcı ve (b) belirsiz bir değer.

5.4. **int *** tamsayıya gösterge,
int *[3] tamsayıya 3 göstergeden oluşan dizi,
int (*)[3] 3 tamsayıdan oluşan diziye gösterge,
int *(float) argümanı **float** olan ve **int**'e gösterge döndüren fonksiyon,
int (*) (void) argümanı olmayan ve **int** döndüren fonksiyona gösterge.

5.5. `a[i, j]` ile `a[(i, j)]` eşdeğerdir, bu da `a[j]`'ye, yani bir göstergeye eşdeğerdir.

5.6. `x` dizi olmasına karşın, `a` bir göstergedir.

5.8. `a` değişmez bir göstergedir; bundan sonra hep `&y`'yi göstermek zorundadır. Fakat gösterdiği yer, yani `y` içindeki bilgi değiştirilebilir.

`b` de değişmez bir göstergedir, bunun yanında gösterdiği yer, yani `x` de program tarafından değiştirilemez. Buna rağmen, bu yer *program dışındaki* bir süreç tarafından değiştirilmektedir. Yani `b`'nin gösterdiği yeri program değiştiremez, sadece okuyabilir.

6.1. **typedef int *** `gosterge`;

ile

```
#define gosterge int *
```

birbirine benzemektedir, ama

```
typedef int dizi[N];
```

#define ile ifade edilemez, onun için **typedef** tercih edilir.

6.2. #define dok(x) printf("x = %d\n", x)

Eğer Standarda uygun bir önışlemciniz varsa, o zaman aşağıdakini kullanın:

```
#define dok(x) printf(#x " = %d\n", x)
```

7.9. #include <stdlib.h>

```
void temizle (void)
```

```
{  
    system("CLS");  
}
```

EK H: TÜRKÇE-İNGİLİZCE VE İNGİLİZCE-TÜRKÇE TERİMLER SÖZLÜĞÜ

Bu kitapta, olanaklar ölçüsünde, İngilizce terimlerden kaçınılmaya çalışılmıştır. Onun yerine Türkçe’de yaygın olarak kullanılan bilgisayar terimleri veya—C’nin özel terimleri için—yeni Türkçe karşılıklar bulunmuştur. Okuyucuların terminoloji konusunda farklı düşüncelerde olması doğaldır. Ancak bir metnin akıcılığını yitirmemesi ve hiç İngilizce bilmeyen kişilerin de zorlanmadan bilgisayar konusundaki kitapları okuyabilmeleri için ulusal bir bilgisayar terminolojisinin oluşturulması ve geliştirilmesi gerekmektedir. Bu terminoloji içinde Türkçe’ye girip türkçeleşmiş sözcükler pekala bulunabilir; ancak belli bir standardın bulunması esastır. Örneğin, “çevirici” sözcüğünden birisinin *translator*, başkasının *assembler* anlamaması gerekir.

Bu ek böyle bir düşüncenin sonucunda ortaya çıkan ve bu kitapta da kullanılan bir çalışmanın sonucunu vermektedir. İlk kısımda, okuyucularımızın kitap içinde karşılaştıkları Türkçe terimlerin İngilizce karşılıklarını bulabilecekleri bir Türkçe-İngilizce sözlük bulunmaktadır. İkinci kısımdaki İngilizce-Türkçe sözlük ise, İngilizce terimlerin karşılığı olarak hangi Türkçe terimlerin kullanıldığını göstermesi açısından okuyucularımıza yardımcı olacaktır.

H.1. Türkçe-İngilizce Sözlük

açıklama.....	comment
açılı parantez, ◇	angular bracket
açma.....	unpack
adres	address
adres alma.....	address of
adresleme.....	addressing

ağ.....	network
ağa köle.....	master slave
ağaç.....	tree
akım.....	current
akış çizeneği.....	flowchart
aktarma.....	copy, transfer
alan.....	field; domain, range
alfasayısal.....	alphanumeric
algoritma.....	algorithm
altçizgi, _.....	underscore
altdizin.....	directory
altprogram.....	subprogram
altaşma.....	underflow
altyordam.....	subroutine
amaç program.....	object program
ana bellek.....	main memory
ana bilgisayar.....	mainframe, host computer
anahtar.....	key, switch, toggle
anahtar sözcük.....	keyword
anlambilim.....	semantics
anlambilimsel.....	semantic
anlamlı rakam.....	significant digit
anlamsal.....	semantic
arabirim.....	interface
arama.....	search, seek
argüman.....	argument
aritmetik mantık birimi.....	arithmetic and logic unit (ALU)
artım.....	increment
artırma.....	increment
atama deyimi.....	assignment statement
aygıt.....	device
ayırıcı.....	separator
ayırma.....	allocation; partitioning
ayraç.....	parenthesis
ayrılmış sözcük.....	reserved word
ayrıştırıcı.....	parser
azaltma.....	decrement
bağıntısal.....	relational
bağlam.....	context
bağlama.....	link
bağlayıcı.....	linker
bakım.....	maintenance
basamaklı.....	scalar
basılmayan karakter.....	nonprint character
başlık.....	header

bayrak	flag
bayt	byte
belgeleme	documentation
belirsiz	undefined
belirteç	specifier
bellek	memory
bellek ayırma	storage allocation
bellek sınıfı	storage class
bellenir	mnemonic
benzetim	simulation
benzetme	emulation
biçim(lendirme)	format
biçimsel	formal
bildirim	declaration
bildirme	declare
bileşik deyim	compound statement
bilgi	information
bilgisayar	computer
bilgisayar destekli tasarım	computer aided design (CAD)
birikeç	accumulator
birim	unit
birleşme	associativity
birleştirici	assembler
birleştirici dil	assembly language
birleştirme	merge
birlik	union
bit	bit
bitişme	join
bitsel	bitwise
blok	block
Boole cebiri	Boolean algebra
boş	blank
boş dizgi	null string
boş karakter	null character
boşluk karakteri	space character
boy	size
böyüt	dimension
bölü	slash
buluşsal	heuristic
çağırma	call
çağrı	call
çengelli parantez, {}	brace
çevirici	translator
çevresel	peripheral
çevrimdışı	offline

çevrimiçi	online
çıktı	output
çift duyarlık	double precision
çizelge	table
çizici	plotter
çok iş düzeni	multitasking
çok kullanıcı	multiuser
çoklama	multiplexing
çoklu	multiple
çözümleme	analysis
çözünürlük	resolution
dal	branch
dallanma	branching
değer ile çağrı	call by value
değil	not
değişken	variable
değişmez	constant
denetim	control
denetleme	verify
derleme	compile
derleyici	compiler
devingen	dynamic
devre	circuit
deyim	statement
dışsal	external
dikey	vertical
dil	language
disk	disk
dizgi	string
dizi	array
dizin	directory; index
doğrudan erişim	direct access
doğrulama	validate
doğruluk tablosu	truth table
doğruluk-değerli	truth-valued
dolaylama işleci	indirection operator
dolaylı adresleme	indirect addressing
donanım	hardware
dosya	file
dosya sonu	end of file (EOF)
dosya tipi	extension
döküm	dump
döngü	loop
dönüş	return
dönüştürme	convert

dönüşüm	conversion
dönüşüm tanımlaması	conversion specification
durağan	static
durak	tab
dural	static
durma	halt
duyarlık	precision
düğüm	node
düşük düzeyli dil	low level language
düşürme	abort
düzen	order, scheme
ekran	screen
elkitabı	manual
emir	directive
enbüyük	maximum
eniye	optimal
eniyeleştirici	optimizing
enküçük	minimum
erim	range
erişim	access, retrieval
erişim modu	access mode
eş(leşme)	match
eşanlamlı	synonym
eşik	threshold
eşitlik	equality
eşlik	parity
eşmerkezli	coaxial
eşzamanlı	synchronous
etiket	label
etki alanı	scope
etkileşimli	interactive
fare	mouse
fonksiyon	function
geçiş	pass, run
geçiş süresi	run time
genişleme	extension
gerçek adres	actual address
gerçek zaman	real time
gerçekleştirme	implementation
geri alma	backspace
geriye dönüş (yapma)	backtrack
getirme	fetch
girdi	input
girdi/çıkı (G/Ç)	input/output (I/O)
görelî adres	relative address

görev	task
görüntü bellek	virtual memory
görüntü noktası	pixel
görünüm	configuration
gösterge	pointer
gösterim	notation
güncelleştirme	update
güvenilirlik	reliability
hareket	transaction
hata	bug, error
hata düzelticisi	debugger
hata düzeltme	debug
hedef	destination
hesaplama	evaluate
içerik	content
içiçe	nested
içsel	internal
ifade	expression
ikil	bit
ikili	binary
ikili düğümlemiş onlu yazım	binary coded decimal (BCD)
ikiye tümler	twos complement
ileti	prompt
iletişim	communication
ilişki	relation
ilişkisel	relational
ilk değer atama	initialize
ilkleme	initialize
ilkleyen	initializer
imleç	cursor
indis	subscript; index
indisleme	indexing
iptal	cancel
isteğe bağlı	optional
iş	job
işaret	mark; sign
işaret biti	sign bit
işaretsiz	unsigned
işleç	operator
işleklik	activity
işlem	operation
işlem kodu	opcode
işlem operatörü	operator
işlemci	operator
işleme	processing

işlenen.....	operand
işletilebilir.....	executable
işletim sistemi.....	operating system
işletme.....	execution
iz.....	track
izleme.....	trace
kabuk.....	shell
kaçış sırası.....	escape sequence
kaçış tuşu.....	escape key
kalıp.....	cast; pattern
kalıtım.....	inheritance
karakter.....	character
karakter dizisi.....	string
karakter sırası.....	character sequence
karar.....	decision
kayan noktalı.....	floating point
kaydırma.....	shift
kayıt.....	record
kayma.....	scroll
kaynak program.....	source program
kesilme noktası.....	breakpoint
kesim.....	sector; segment
kesinti.....	interrupt
kısaa.....	acronym
kısım.....	section
kişisel bilgisayar.....	personal computer (PC)
kitaplık.....	library
klavye.....	keyboard
kod.....	code
kod çözme.....	decode
kodlama.....	coding, encode
komut.....	command, instruction
kontrol.....	control
kontrol akışı.....	control flow
kontrol karakteri.....	control character
konum.....	position; offset
kopyalama.....	copy
koşullu dallanma.....	conditional branch
koşulsuz dallanma.....	unconditional branch
kök.....	radix
köşeli parantez, [].....	bracket
kullanıcı.....	user
kullanıcı-tanımlı.....	user-defined
kullanma.....	reference
kurgu.....	setup

kuruluş	installation
kuşak	generation
kuyruk	queue, tail
küme	set
küme komut işleme	pipelining, piping
künye	tag
küresel	global
kütüphane	library
liste	list
makine dili	machine language
makro	macro
mantık	logic
mantıksal kaydırma	logical shift
menü	menu
merkezi işlem birimi (MİB)	central processing unit (CPU)
mesaj	message
metin	text
metin düzenleme	editing
metin düzenleyici	editor
mikroişlemci	microprocessor
mimari	architecture
mod	mode
modüler	modular
monitör	monitor
mutlak adres	absolute address
nesne	object
nesneye dayalı	object oriented
niteleyici	qualifier
nitelik	attribute
okunaklılık	readability
olumlu sayı	positive number
olumsuz sayı	negative number
olumsuzlama	negation
onaltılı gösterim	hexadecimal notation
ondalık	decimal
ortam	environment, medium
otomasyon	automation
otomatik	automatic
öbek	block
ön bellek	cache memory
öncelik	priority, precedence
önek	prefix
önişlemci	preprocessor
önyükleme	bootstrapping
örneksel	analog

örtü	mask
özçağrı	recursion
özçağrılı	recursive
özdevimli	automatic
özdevinim	automation
özel karakter	special character
özellik	property
paketleme	packing
parametre	parameter
parantez	parenthesis
pencere	window
plan	schema
program	program
program sayacı	program counter
programcı	programmer
programlama dili	programming language
rakam	digit
rastgele erişimli bellek	random access memory (RAM)
rastgele sayı	random number
referans	reference
referans ile çağrı	call by reference
saat	clock
sabit	constant
sağa yanaştırma	right justify
saklama	save, store
salt okunur bellek	read only memory (ROM)
satır ilerletme	line feed
satırbaşı	carriage return
satır içi	inline
sayaç	counter
sayfa ilerletme	form feed
sayı gösterimi	number representation
sayıcı	enumerator
sayılı	enumerated
sayım	enumeration
sayısal	digital, numeric
sayısal tuşlar	numerical keypad
seçme	select
sekizli	octal
sıfır doldurma	zerofill
sıfırlama	reset
sıfırların kaldırılması	zero suppression
sığa	capacity
sıkıştırma	compression
sinama	test

sınıf.....	class
sınıflandırma.....	taxonomy
sınırlayıcı.....	delimiter
sıra.....	order, sequence
sıralama.....	sort
sıralı erişim.....	sequential access
silme.....	clear
simge.....	symbol
sistem.....	system
sola yanaşık.....	left justified
sonek.....	postfix
sorgu(lama).....	query
soyut.....	abstract
sözcük.....	word
sözcük işleme.....	word processing
sözcük uzunluğu.....	word length
sözde kod.....	pseudo code
sözde komut.....	pseudo instruction
sözdizim.....	syntax
sözdizimsel.....	syntactic
süreç.....	process
sürücü.....	driver; drive
şifre.....	password
şimdiki.....	current
taban.....	base
tahteravalli.....	flip flop
tampon.....	buffer
tamsayı.....	integer
tanım.....	definition
tanımlama.....	define
tanımlayıcı.....	descriptor
tanımsız.....	undefined
tanıtıcı sözcük.....	identifier
tarama.....	scan
tasarım.....	design
taşınabilir.....	portable
taşma.....	overflow
tekli.....	unary
temel adres.....	base address
ters bölü.....	backslash
tez sıralama.....	quick sort
tikel.....	partial
tip.....	type
toplama.....	add
tutamak.....	handle

tuzak	trap
tümleşik	integrated
türetilmiş tip	derived type
uç	terminal
uyarı mesajı	warning message
uyarlama	version
uygulama	application
uyumlu	compatible
uzaklık	offset
üç nokta,	ellipsis
üçlü	ternary
üçlü harf	trigraph
üs	exponent; superscript
üye	member
varsayılan değer	default value
vazgeçme	abort
ve	and
veri	data
veritabanı	database
versiyon	version
veya	or
yama	patch
yan etki	side-effect
yanıştırma	justify
yanıt süresi	response time
yapay anlayış	artificial intelligence
yapı	structure
yapı adı	structure tag
yapıçı	intrinsic
yapısal	structured
yardımcı program	utility
yayılma aralığı	range
yazıcı	printer
yazılım	software
yazım	notation
yazma-koruma	write-protect
yazmaç	register
yedek(leme)	backup
yeni satır	newline
yeniden yönlendirme	redirection
yeniden yükleme	restore
yerdeğiştir	relocatable
yerel	local
yerleşen (-şik)	resident
yerpaylaşım	overlay

yığıt.....	stack
yıldız.....	asterisk
yineleme.....	iteration
yinelemeli.....	iterative
yol.....	path
yonga.....	chip
yordam.....	procedure, routine
yorumlayıcı.....	interpreter
yöntem.....	method
yöre.....	locale
yumuşak disk.....	floppy disk, diskette
yükleme.....	load
yükleyici.....	loader
yüksek düzeyli dil.....	high level language
yürütme.....	execution
yürütülebilir.....	executable
zaman uyumsuz.....	asynchronous
zamanpaylaşım.....	timesharing
zil.....	bell

H.2. İngilizce-Türkçe Sözlük

abort.....	düşürme, vazgeçme
absolute address.....	mutlak adres
abstract.....	soyut
access.....	erişim
access mode.....	erişim modu
accumulator.....	birikeç
acronym.....	kısaad
activity.....	işleklik
actual address.....	gerçek adres
add.....	toplama
address.....	adres
address of.....	adres alma
addressing.....	adresleme
algorithm.....	algoritma
allocation.....	ayırma
alphanumeric.....	alfasayısal
analog.....	örneksel
analysis.....	çözümleme
and.....	ve
angular bracket, \angle	açılı parantez
application.....	uygulama
argument.....	argüman

architecture	mimari
arithmetic and logic unit (ALU)	aritmetik mantık birimi
array	dizi
artificial intelligence	yapay anlayış
assembler	birleştirici
assembly language	birleştirici dil
assignment statement	atama deyiimi
associativity	birleşme
asterisk	yıldız
asynchronous	zaman uyumsuz
attribute	nitelik
automatic	özdevimli, otomatik
automation	özdevinim, otomasyon
backslash	ters bölü
backspace	geri alma
backtrack	geriye dönüş (yapma)
backup	yedek(leme)
base	taban
base address	temel adres
bell	zıl
binary	ikili
binary coded decimal (BCD)	ikili düğümlemiş onlu yazım
bit	bit, ikil
bitwise	bitsel
blank	boş
block	blok, öbek
Boolean algebra	Boole cebiri
bootstrapping	önyükleme
brace, { }	çengelli parantez
bracket, []	köşeli parantez
branch	dal
branching	dallanma
breakpoint	kesilme noktası
buffer	tampon
bug	hata
byte	bayt
cache memory	ön bellek
call	çağırma, çağrı
call by reference	referans ile çağrı
call by value	değer ile çağrı
cancel	iptal
capacity	sığa
carriage return	satırbaşı
cast	kalıp
central processing unit (CPU)	merkezi işlem birimi (MİB)

character	karakter
character sequence	karakter sırası
chip	yonga
circuit	devre
class	sınıf
clear	silme
clock	saat
coaxial	eşmerkezli
code	kod
coding	kodlama
command	komut
comment	açıklama
communication	iletişim
compatible	uyumlu
compile	derleme
compiler	derleyici
compound statement	bileşik deyim
compression	sıkıştırma
computer	bilgisayar
computer aided design (CAD)	bilgisayar destekli tasarım
conditional branch	koşullu dallanma
configuration	görünüm
constant	değişmez, sabit
content	içerik
context	bağlam
control	kontrol, denetim
control character	kontrol karakteri
control flow	kontrol akışı
conversion	dönüşüm
conversion specification	dönüşüm tanımlaması
convert	dönüştürme
copy	aktarma, kopyalama
counter	sayaç
current	şimdiki; akım
cursor	imleç
data	veri
database	veritabanı
debug	hata düzeltme
debugger	hata düzelticisi
decimal	ondalık
decision	karar
declaration	bildirim
declare	bildirme
decode	kod çözme
decrement	azaltma

default value	varsayılan değer
define	tanımlama
definition	tanım
delimiter	sınırlayıcı
derived type	türetilmiş tip
descriptor	tanımlayıcı
design	tasarım
destination	hedef
device	aygıt
digit	rakam
digital	sayısal
dimension	boyut
direct access	doğrudan erişim
directive	emir
directory	(alt)dizin
disk	disk
documentation	belgeleme
domain	alan
double precision	çift duyarlık
drive	sürücü
driver	sürücü
dump	döküm
dynamic	devingen
editing	metin düzenleme
editor	metin düzenleyici
ellipsis,	üç nokta
emulation	benzetme
encode	kodlama
end of file (EOF)	dosya sonu
enumerated	sayılı
enumeration	sayım
enumerator	sayıcı
environment	ortam
equality	eşitlik
error	hata
escape key	kaçış tuşu
escape sequence	kaçış sırası
evaluate	hesaplama
executable	işletilebilir, yürütülebilir
execution	işletme, yürütme
exponent	üs
expression	ifade
extension	genişleme; dosya tipi
external	dışsal
fetch	getirme

field.....	alan
file.....	dosya
flag.....	bayrak
flip flop.....	tahteravalli
floating point.....	kayan noktalı
floppy disk.....	yumuşak disk (disket)
flowchart.....	akış çizeneği
form feed.....	sayfa ilerletme
formal.....	biçimsel
format.....	biçim(lendirme)
function.....	fonksiyon
generation.....	kuşak
global.....	küresel
halt.....	durma
handle.....	tutamak
hardware.....	donanım
header.....	başlık
heuristic.....	buluşsal
hexadecimal notation.....	onaltılı gösterim
high level language.....	yüksek düzeyli dil
host computer.....	ana bilgisayar
identifier.....	tanıtıcı sözcük
implementation.....	gerçekleştirme
increment.....	artım, artırma
index.....	indis; dizin
indexing.....	indisleme
indirect addressing.....	dolaylı adresleme
indirection operator.....	dolaylama işleci
information.....	bilgi
inheritance.....	kalıtım
initialize.....	ikleme, ilk değer atama
initializer.....	ilkleyen
inline.....	satırıçi
input.....	girdi
input/output (I/O).....	girdi/çıkı (G/Ç)
installation.....	kuruluş
instruction.....	komut
integer.....	tamsayı
integrated.....	tümleşik
interactive.....	etkileşimli
interface.....	arabirim
internal.....	içsel
interpreter.....	yorumlayıcı
interrupt.....	kesinti
intrinsic.....	yapııçi

iteration.....	yineleme
iterative.....	yinelemeli
job.....	iş
join.....	bitişme
justify.....	yanıştırma
key.....	anahtar
keyboard.....	klavye
keyword.....	anahtar sözcük
label.....	etiket
language.....	dil
left justified.....	sola yanaşık
library.....	kütüphane, kitaplık
line feed.....	satır ilerletme
link.....	bağlama
linker.....	bağlayıcı
list.....	liste
load.....	yükleme
loader.....	yükleyici
local.....	yerel
locale.....	yöre
logic.....	mantık
logical shift.....	mantıksal kaydırma
loop.....	döngü
low level language.....	düşük düzeyli dil
machine language.....	makine dili
macro.....	makro
main memory.....	ana bellek
mainframe.....	ana bilgisayar
maintenance.....	bakım
manual.....	elkitabı
mark.....	işaret
mask.....	örtü
master slave.....	ağa köle
match.....	eş(leşme)
maximum.....	enbüyük
medium.....	ortam
member.....	üye
memory.....	bellek
menu.....	menü
merge.....	birleştirme
message.....	mesaj
method.....	yöntem
microprocessor.....	mikroişlemci
minimum.....	enküçük
mnemonic.....	bellendir

mode	mod
modular	modüler
monitor	monitör
mouse	fare
multiple	çoklu
multiplexing	çoklama
multiuser	çok kullanıcı
multitasking	çok iş düzeni
negation	olumsuzlama
negative number	olumsuz sayı
nested	iç içe
network	ağ
newline	yeni satır
node	düğüm
nonprint character	basılmayan karakter
not	değil
notation	gösterim, yazım
null character	boş karakter
null string	boş dizgi
number representation	sayı gösterimi
numeric	sayısal
numerical keypad	sayısal tuşlar
object	nesne
object oriented	nesneye dayalı
object program	amaç program
octal	sekizli
offline	çevrimdışı
offset	konum, uzaklık
online	çevrimiçi
opcode	işlem kodu
operand	işlenen
operating system	işletim sistemi
operation	işlem
operator	işleç, işlemci, işlem operatörü
optimal	eniyi
optimizing	eniyileştirici
optional	isteğe bağlı
or	veya
order	düzen, sıra
output	çıkış
overflow	taşma
overlay	yerpaylaşım
packing	paketleme
parameter	parametre
parenthesis	parantez, araç

parity.....	eşlik
parser.....	ayrıştırıcı
partial.....	tikel
partitioning.....	ayırma
pass.....	geçiş
password.....	şifre
patch.....	yama
path.....	yol
pattern.....	kalıp
peripheral.....	çevresel
personal computer (PC).....	kişisel bilgisayar
pipelining.....	küme komut işleme
piping.....	küme komut işleme
pixel.....	görüntü noktası
plotter.....	çizici
pointer.....	gösterge
portable.....	taşınabilir
position.....	konum
positive number.....	olumlu sayı
postfix.....	sonek
precedence.....	öncelik
precision.....	duyarlık
prefix.....	önek
preprocessor.....	önişlemci
printer.....	yazıcı
priority.....	öncelik
procedure.....	yordam
process.....	süreç
processing.....	işleme
program.....	program
program counter.....	program sayacı
programmer.....	programcı
programming language.....	programlama dili
prompt.....	ileti
property.....	özellik
pseudo code.....	sözde kod
pseudo instruction.....	sözde komut
qualifier.....	niteleyici
query.....	sorgu(lama)
queue.....	kuyruk
quick sort.....	tez sıralama
radix.....	kök
random access memory (RAM).....	rastgele erişimli bellek
random number.....	rastgele sayı
range.....	erim, alan, yayılma aralığı

read only memory (ROM)	salt okunur bellek
readability	okunaklılık
real time	gerçek zaman
record	kayıt
recursion	özçağrı
recursive	özçağrılı
redirection	yeniden yönlendirme
reference	referans, kullanma
register	yazmaç
relation	ilişki
relational	bağıntısal, ilişkisel
relative address	görelî adres
reliability	güvenilirlik
relocatable	yerdeğişir
reserved word	ayrılmış sözcük
reset	sıfırlama
resident	yerleşen, yerleşik
resolution	çözünürlük
response time	yanıt süresi
restore	yeniden yükleme
retrieval	erişim
return	dönüş
right justify	sağa yanaştırma
routine	yordam
run	geçiş
run time	geçiş süresi
save	saklama
scalar	basamaklı
scan	tarama
schema	plan
scheme	düzen
scope	etki alanı
screen	ekran
scroll	kayma
search	arama
section	kısım
sector	kesim
seek	arama
segment	kesim
select	seçme
semantic	anlambilimsel, anlamsal
semantics	anlambilim
separator	ayırıcı
sequence	sıra
sequential access	sıralı erişim

set	küme
setup	kurgu
shell	kabuk
shift	kaydırma
side-effect	yan etki
sign	işaret
sign bit	işaret biti
significant digit	anlamli rakam
simulation	benzetim
size	boy
slash	bölü
software	yazılım
sort	sıralama
source program	kaynak program
space character	boşluk karakteri
special character	özel karakter
specifier	belirteç
stack	yığıt
statement	deyim
static	dural, durağan
storage allocation	bellek ayırma
storage class	bellek sınıfı
store	saklama
string	karakter dizisi, dizgi
structure	yapı
structure tag	yapı adı
structured	yapısal
subprogram	altprogram
subroutine	altyordam
subscript	indis
superscript	üs
switch	anahtar
symbol	simge
synchronous	eşzamanlı
synonym	eşanlamli
syntactic	sözdizimsel
syntax	sözdizim
system	sistem
tab	durak
table	çizelge
tag	künye
tail	kuyruk
task	görev
taxonomy	sınıflandırma
terminal	uç

ternary.....	üçlü
test.....	sınama
text.....	metin
threshold.....	eşik
timesharing.....	zamanpaylaşım
toggle.....	anahtar
trace.....	izleme
track.....	iz
transaction.....	hareket
transfer.....	aktarma
translator.....	çevirici
trap.....	tuzak
tree.....	ağaç
trigraph.....	üçlü harf
truth table.....	doğruluk tablosu
truth-valued.....	doğruluk-değerli
twos complement.....	ikiye tümler
type.....	tip
unary.....	tekli
unconditional branch.....	koşulsuz dallanma
undefined.....	belirsiz, tanımsız
underflow.....	alttaşma
underscore, _.....	altçizgi
union.....	birlik
unit.....	birim
unpack.....	açma
unsigned.....	işaretsiz
update.....	güncelleştirme
user.....	kullanıcı
user-defined.....	kullanıcı-tanımlı
utility.....	yardımcı program
validate.....	doğrulama
variable.....	değişken
verify.....	denetleme
version.....	uyarlama, versiyon
vertical.....	dikey
virtual memory.....	görüntü bellek
warning message.....	uyarı mesajı
window.....	pencere
word.....	sözcük
word length.....	sözcük uzunluğu
word processing.....	sözcük işleme
write-protect.....	yazma-koruma
zero suppression.....	sıfırların kaldırılması
zerofill.....	sıfır doldurma

BİBLİYOGRAFYA

- Bach, Maurice J. *The Design of the UNIX Operating System*. Englewood Cliffs, NJ: Prentice-Hall, 1986.
- Bolon, C. *Mastering C*. Berkeley, CA: Sybex, 1986.
- Bolsky, Morris I. *The C Programmer's Handbook*. Englewood Cliffs, NJ: Prentice-Hall, 1985.
- Brown, D. L. *From Pascal to C*. Belmont, CA: Wadsworth, 1985.
- Butzen, Fred, "Porting to ANSI C," *UNIX World*, May-June 1989.
- BYTE*. McGraw-Hill Inc., August 1983.
- BYTE*. McGraw-Hill Inc., August 1988.
- Campbell, Joe. *Crafting C Tools for the IBM PCs*. Englewood Cliffs, NJ: Prentice-Hall, 1986.
- Costales, Bryan. *C: From A to Z*. Englewood Cliffs, NJ: Prentice-Hall, 1985.
- Curry, David A. *Using C on the UNIX System: A Guide to System Programming*. Sebastopol, CA: O'Reilly & Associates, 1991.
- Feuer, Alan R. *The C Puzzle Book*. Englewood Cliffs, NJ: Prentice-Hall, 1982.
- Gehani, N. *Advanced C: Food for the Educated Palate*. Rockville, Md.: Computer Science Press, 1985.
- Hancock, Les and Morris Krieger. *The C Primer*. 2nd ed. New York, NY: McGraw-Hill, 1986.
- Harbison, Samuel P. and Guy L. Steele, Jr. *C: A Reference Manual*. Englewood Cliffs, NJ: Prentice-Hall, 1984.
- Holub, Allen. *The C Companion*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

- Holzner, Steven. *PS/2-PC Assembly Language*. New York, NY: Brady, 1989.
- Jaeschke, Rex, "Standard C: A status report," *Dr. Dobbs's Journal*, August 1991.
- Kelley, A. and I. Pohl. *An Introduction to Programming in C*. Menlo Park, CA: Benjamin/Cummings, 1984.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- Ladd, Scott Robert. *C++ Techniques and Applications*. Redwood City, CA: M&T Books, 1990.
- Lippman, Stanley B. *C++ Primer*. 2nd ed. Reading, MA: Addison-Wesley, 1993.
- Microsoft® *C for the MS-DOS® Operating System, Version 5.00*, Run-time Library Reference. Microsoft Corporation, 1987.
- Microsoft® *C for the MS-DOS® Operating System, Version 5.00*, Mixed Language Programming Guide. Microsoft Corporation, 1987.
- Microsoft® *Macro Assembler for the MS-DOS® Operating System*, User's Guide. Microsoft Corporation, 1985.
- Microsoft® *QuickC™ Compiler for IBM® Personal Computers and Compatibles*, Microsoft Corporation, 1987.
- Microsoft® *Visual C++™ Development System for Windows™*, Introduction (Presenting Visual C++). Redmond, WA: Microsoft Corporation, 1993.
- Microsoft® *Visual C++™ Development System for Windows™*, User's Guides (Visual Workbench User's Guide, App Studio User's Guide). Redmond, WA: Microsoft Corporation, 1993.
- Microsoft® *Visual C++™ Development System for Windows™*, Programmer's Guides (C++ Tutorial, Class Library User's Guide, Programming Techniques). Redmond, WA: Microsoft Corporation, 1993.
- Microsoft® *Visual C++™ Development System for Windows™*, Professional Tools User's Guides (Programming Tools for Windows, CodeView Debugger User's Guide, Command-line Utilities User's Guide, Source Profiler User's Guide). Redmond, WA: Microsoft Corporation, 1993.
- Microsoft® *Visual C++™ Development System for Windows™*, Reference, Vol. 1-3 (Class Library Reference; C Language Reference, C++ Language Reference; Run-time Library Reference, iostream Class Library Reference). Redmond, WA: Microsoft Corporation, 1993.
- Microsoft® *Visual C++™ Development System for Windows™*, Comprehensive Index. Redmond, WA: Microsoft Corporation, 1993.

- Microsoft® Visual C++™ Development System for Windows™, C/C++ Version 7.0 Update. Redmond, WA: Microsoft Corporation, 1993.
- Oualline, Steve. *Practical C Programming*. Sebastopol, CA: O'Reilly & Associates, 1993.
- Plauger, P. J. and Jim Brodie. *Standard C*.
- Plum, Thomas. *C Programming Guidelines*. Englewood Cliffs, NJ: Prentice-Hall, 1984.
- Plum, Thomas. *Learning to Program in C*. Englewood Cliffs, NJ: Prentice-Hall, 1983.
- Plum, Thomas. *Notes on the Draft C Standard*. Cardiff, NJ: 1988.
- Prosser, David F. Draft proposed American National Standard for Information Systems—Programming Language C X3 Secretariat, CBEMA, Washington.
- Schildt, Herbert. *Windows NT Programming Handbook*. Berkeley, CA: Osborne McGraw-Hill, 1993.
- Stroustrup, Bjarne. *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1986.
- Stroustrup, Bjarne. *The C++ Programming Language*. 2nd ed. Reading, MA: Addison-Wesley, 1993.
- Tanenbaum, Aaron M., Yedidyah Langsam, and Moshe J. Augenstein. *Data Structures Using C*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
- Tanenbaum, Andrew S. *Operating Systems: Design and Implementation*. Englewood Cliffs, NJ: Prentice-Hall, 1987.
- The Waite Group. *Advanced C Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1986.
- Tondo, Clovis L. and Scott E. Gimpel. *The C Answer Book*. 1st ed. Englewood Cliffs, NJ: Prentice-Hall, 1985.
- Tondo, Clovis L. and Scott E. Gimpel. *The C Answer Book*. 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- Varhol, Peter D. *Object-Oriented Programming: The Software Development Revolution*. Charleston, South Carolina: Computer Technology Research Corp., 1993.
- Waite, M., S. Prata, and D. Martin. *C Primer Plus*. Indianapolis, IN: Howard W. Sams & Co., 1984.
- Wiener, Richard S. and Lewis J. Pinson. *An Introduction to Object-Oriented Programming and C++*. Reading, MA: Addison-Wesley, 1988.
- Williams, Mark. *ANSI C Lexical Guide*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- Wortman, Leon A. and Thomas O. Sidebottom. *The C Programming Tutor*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

DİZİN

#

#define, 12, 110
#elif, 113
#else, 113
#endif, 113
#error, 115
#if, 113
#ifdef, 113
#ifndef, 113
#include, 12, 112
#line, 115
#pragma, 115
#undef, 111

_

_asm, 162
DATE, 115
FILE, 115
LINE, 115
STDC, 115
TIME, 115
_cgets, 166
_close, 127
_creat, 126
_daylight, 181
_emit, 162
_exit, 125
_getch, 21
_getche, 21
_IOBF, 125, 177
_IOLBF, 125, 177

_IONBF, 125, 177
_int86, 163
_lseek, 127
_O_CREAT, 127
_O_RDONLY, 127
_open, 127
_read, 127
_S_IREAD, 126
_S_IWRITE, 126
_timezone, 181
_tzname, 181
_write, 127

A

abort, 177, 179
abs, 179
acos, 176
açıklama, 5
adım modu, 152
alan, 96
altdizin, 124
anahtar, 130
anahtar sözcük, 6
ANSI, 2
ANSI karakter kümesi, 136
ANSI Standardı, 3, 25, 92, 149, 169, 173, 181
argc, 68
argüman, 62
argv, 68
aritmetik kaydırma, 54
ASCII karakter kümesi, 135
asctime, 171, 181

asin, 176
 assert, 173
 assert.h, 173
 atama, 13
 atan, 176
 atan2, 176
 atexit, 179
 atof, 52, 179
 atoi, 52, 179
 atol, 52, 179
 auto, 70

B

bağlama, 59, 68, 146, 149
 bağlayıcı, 59, 60, 63, 148, 159
 başlangıç adresi, 80
 başlık dosyaları, 12
 başlık dosyası, 112
 bayt, 46
 bellek modeli, 146
 bellek sınıfı, 69
 bellek sınıfı belirteci, 93
 beyaz karakter, 20
 biçimsel argüman, 61, 111
 bildirim, 72
 bileşik deyim, 21
 bire tümler işleci, 54
 birleşme, 14, 57
 birleştirici dil, 159
 birlik, 95
 bit, 46
 bitsel işleç, 53
 blok, 21, 61
 boş deyim, 30, 34, 37
 boş karakter, 8, 9
 break, 39
 bsearch, 179
 BUFSIZ, 125, 177
 büyük model, 146

C

C, 1
 C++, 2
 calloc, 98, 179
 case, 41
 ceil, 176
 char, 10
 CHAR_BIT, 175
 CHAR_MAX, 175

CHAR_MIN, 175
 clearerr, 178
 clock, 172, 181
 clock_t, 172, 180
 CLOCKS_PER_SEC, 180
 Codeview, 151
 const, 10, 67
 continue, 38
 cos, 176
 cosh, 176
 ctime, 171, 181
 ctype.h, 116, 173

çağrı

değer ile, 64, 160
 referans ile, 64, 160
 çevirme sınırları, 181
 çıktı, 17
 çokbaytlı karakter, 170

D

DBL_DIG, 44, 174
 DBL_EPSILON, 174
 DBL_MANT_DIG, 174
 DBL_MAX, 174
 DBL_MAX_10_EXP, 174
 DBL_MAX_EXP, 174
 DBL_MIN, 174
 DBL_MIN_10_EXP, 174
 DBL_MIN_EXP, 174
 DBL_RADIX, 174
 DBL_ROUND, 174
 default, 41
 defined, 113
 değer ile çağrı, 64, 160
 değişken, 9
 dışsal, 72
 dural, 71
 otomatik, 70
 yazmaç, 70
 değişken gösterge, 50, 75
 değişmez, 7
 çift duyarlılık, 8
 dörtlü duyarlılık, 8
 karakter, 8
 karakter dizisi, 9
 kayan noktalı, 7
 kısa tamsayı, 7

onaltılı, 7
 sekizli, 7
 tamsayı, 7
 tek duyarlıklılı, 8
 uzun çift duyarlıklılı, 8
 uzun tamsayı, 7
 değişmez gösterge, 50, 75
 değişmez ifade, 35, 53, 73, 92, 94, 99, 113
 dev adres, 145
 dev model, 146
 devam etme, satır, 9, 109
 devingen dizi, 99
 deyim, 29
 difftime, 171, 181
 div, 179
 div_t, 178
 dizi, 10, 99
 do, 36
 dos.h, 163
 dosya, 119
 dosya göstergesi, 120
 dosya sonu, 21
 dosya tanımlayıcısı, 126
 dosya tutamağı, 126
double, 10
 dönüş adresi, 77
 dönüşüm tanımlaması, 18

E

EDOM, 174, 176
 else, 33
 entry, 26
enum, 87
 envp, 69
 EOF, 21, 121, 124, 177
 ERANGE, 174, 176
 errno, 174, 176, 177, 179
 errno.h, 174
 etiket, 39
 etiketli deyim, 39
 etki alanı
 değişken, 69
 tip bildirimi, 93
 EXIT_FAILURE, 126, 179
 EXIT_SUCCESS, 126, 179
 exit, 102, 125, 179
 exp, 176
 extern, 72

F

fabs, 176
 fclose, 121, 125, 178
 fcntl.h, 126
 feof, 178
 ferror, 178
 fflush, 125, 178
 fgetc, 178
 fgetpos, 178
 fgets, 122, 178
 FILE, 120, 177
 FILENAME_MAX, 177
float, 10
 float.h, 12, 174
 floor, 176
 FLT_DIG, 174
 FLT_EPSILON, 174
 FLT_MANT_DIG, 174
 FLT_MAX, 174
 FLT_MAX_10_EXP, 174
 FLT_MAX_EXP, 174
 FLT_MIN, 174
 FLT_MIN_10_EXP, 174
 FLT_MIN_EXP, 174
 FLT_RADIX, 174
 FLT_ROUNDS, 174
 fmod, 176
 fonksiyon, 59
 fonksiyon argümanı, 60
 fonksiyon bildirimi, 64
 fonksiyon göstergesi, 78
 fonksiyon prototipi, 64
 fonksiyon tanımı, 60
 fopen, 119, 178
 FOPEN_MAX, 177
 for, 37
 fpos_t, 177
 fprintf, 122, 178
 fputc, 178
 fputs, 121, 178
 fread, 124, 178
 free, 98, 103, 179
 freopen, 178
 frexp, 176
 fscanf, 122, 178
 fseek, 123, 178
 fsetpos, 178
 ftell, 123, 178
 fwrite, 124, 178

G

genel gösterge, 98
 geniş karakter, 170
 genişletilmiş ASCII karakter kümesi, 135
 gerçek argüman, 62
 geriye dönüş, 81
 getc, 121, 178
 getchar, 20, 121, 178
 getenv, 179
 gets, 122, 178
 girdi, 17
 girdi/çıkıtının yeniden yönlendirilmesi, 106
 gmtime, 171, 181
 goto, 39
 gösterge, 45
 gösterge aritmetiği, 48
 gözcü, 42

H

harf ayırımı, 6
 HUGE_VAL, 176

I

INT_MAX, 175
 INT_MIN, 175
 if, 33
 ifade, 13
 değişmez, 10, 16
 doğruluk-değerli, 32
 karışık-tip, 16
 ifade deyimi, 21
 ifadelerin yeniden düzenlenmesi, 16, 54
 ikili ağaç, 103
 ikili girdi/çıkıtı, 120
 ikili işleç, 46
 ikili sistem, 7
 ilk değer atama, 10
 ilkleme, 10, 73
 int, 10
 io.h, 126
 isalnum, 173
 isalpha, 173
 iscntrl, 173
 isdigit, 174
 isgraph, 174
 isim, 6
 islower, 174
 isprint, 174
 ispunct, 174

isspace, 174
 isupper, 174
 isxdigit, 174
 işleç, 13
 adres alma, 46
 adres alma, 20, 65, 79, 97
 aritmetik, 13
 artırma, 14
 atama, 14, 55
 azaltma, 14
 bağıntısal, 30
 bitisel dışlayan VEYA, 54
 bitisel VE, 54
 bitisel VEYA, 55
 doğruluk-değerli, 30
 dolaylama, 46
 eşitlik, 31
 kaydırma, 54
 koşullu, 34
 mantıksal olumsuzlama, 30
 mantıksal VE, 31
 mantıksal VEYA, 32
 üçlü, 35
 virgöl, 38
 yapı üyesi, 91
 işlem operatörü, 13
 işlenen, 13
 işletim sistemi, 119
 itoa, 52
 izleme, 24, 152

J

jmp_buf, 176

K

kaçış sırası, 8
 kalıp, 17, 89, 93
 karakter dizisi, 51
 karakter dizisi işleme fonksiyonu, 51
 kesilme, 152
 kesilme noktası, 24
 kesim, 145, 152, 160
 kesinti, 163, 176
 kısa model, 146
 kod kesimi, 145
 komut satırı argümanı, 68
 kullanım, 11
 küçük büyük harf ayırımı, 6
 küçük model, 146
 küme komut işleme, 107

küresel değişken, 60, 69
kütüphane, 119
kütüphane fonksiyonu, 79

L

L_tmpnam, 177
labs, 179
LC_ALL, 169, 175
LC_COLLATE, 169, 175
LC_CTYPE, 169, 175
LC_MONETARY, 169, 175
LC_NUMERIC, 169, 175
LC_TIME, 169, 175
lconv, 169, 175
LDBL_DIG, 175
LDBL_EPSILON, 175
LDBL_MANT_DIG, 175
LDBL_MAX, 175
LDBL_MAX_10_EXP, 175
LDBL_MAX_EXP, 175
LDBL_MIN, 175
LDBL_MIN_10_EXP, 175
LDBL_MIN_EXP, 175
LDBL_RADIX, 175
LDBL_ROUNDS, 175
ldexp, 176
ldiv, 179
ldiv_t, 178
LIB, 155
limits.h, 12, 175
locale.h, 175
localeconv, 169, 175
localtime, 171, 181
log, 176
log10, 176
long, 10
LONG_MAX, 175
LONG_MIN, 175
longjmp, 176
ltoa, 52

M

MAKE, 156
makro, 110
malloc, 98, 103, 179
mantıksal kaydırma, 54
math.h, 80, 176
MB_CUR_MAX, 179
MB_LEN_MAX, 175

mblen, 170, 179
mbstowcs, 170, 179
mbtowc, 170, 179
memchr, 180
memcpy, 180
memcpy, 180
memmove, 180
memset, 180
Microsoft C Derleyicisi, 147
minik model, 146
mktime, 172, 181
modf, 176

N

NDEBUG, 173
NMAKE, 156
NULL, 49, 120, 177, 178, 179

O

orta model, 146
otomatik tip dönüşümü, 16, 61, 73

Ö

önceden tanımlanmış isimler, 115
öncelik, 14, 57
önişlemci, 12, 109
örtme, 56
özçağrı, 75, 82, 106
özçağrılı fonksiyon, 75
öz-referanslı yapı, 104

P

P_tmpnam, 177
parametre, 61
perror, 178, 179
pow, 176
printf, 18, 59, 122, 178
program açıklaması, 5
ptrdiff_t, 49, 177
putc, 120, 178
putchar, 20, 121, 178
puts, 122, 178

Q

qsort, 179
QuickC çekirdek kütüphanesi, 146
QuickC kütüphanesi, 146

R

raise, 177
 rand, 126, 179
 RAND_MAX, 126, 178
 rastgele erişim, 123
 realloc, 179
 referans, 11
 referans ile çağrı, 64, 160
 register, 70
 remove, 124, 178
 rename, 124, 178
 return, 61
 rewind, 124, 178

S

satır devam etme, 9, 109
 satırıçı birleştiricisi, 162
 satırıçı kod, 111
 sayıcı, 88
 sayım künyesi, 87
 sayım tipi, 87
 scanf, 19, 122, 178
 SCHAR_MAX, 175
 SCHAR_MIN, 175
 SEEK_CUR, 123, 177
 SEEK_END, 123, 177
 SEEK_SET, 123, 177
 setbuf, 125, 178
 setjmp, 176
 setjmp.h, 176
 setlocale, 169, 175
 setvbuf, 125, 178
short, 10
 SHRT_MAX, 175
 SHRT_MIN, 175
 SIG_DFL, 177
 SIG_ERR, 177
 SIG_IGN, 177
 SIGABRT, 177
 SIGFPE, 177
 SIGILL, 177
 SIGINT, 177
 SIGSEGV, 177
 SIGTERM, 177
 sıralı erişim, 123
 signal, 177
 signal.h, 176
signed, 10
 sin, 176

sinh, 176
 size_t, 94, 98, 177, 178, 179, 180
 sizeof, 93, 94
 sözcük, 46
 sözcük sınırı, 96
 sprintf, 123, 178
 sqrt, 176
 srand, 126, 179
 sscanf, 123, 178
 static, 71
 stdarg.h, 177, 185
 stddef.h, 49, 94, 177
 stderr, 18, 120, 177
 stdin, 18, 120, 177
 stdio.h, 18, 119, 121, 177
 stdlib.h, 52, 98, 102, 119, 178
 stdout, 18, 120, 177
 strcat, 52, 180
 strchr, 52, 180
 strcmp, 52, 180
 strcoll, 180
 strcpy, 51, 180
 strcspn, 180
 strerror, 180
 strftime, 172, 181
 string.h, 52, 179
 strlen, 52, 180
 strncat, 52, 180
 strncmp, 52, 180
 strncpy, 52, 180
 strpbrk, 180
 strrchr, 52, 180
 strspn, 180
 strstr, 180
 strtod, 179
 strtok, 180
 strtol, 179
 strtoul, 179
struct, 89
 struct tm, 180
 strxfrm, 180
 switch, 40
 sys\\stat.h, 126
 sys\\types.h, 126
 system, 125, 179

T

tan, 176
 tanh, 176

tanım, 72
tanıtıcı sözcük, 6
taşınabilirlik, 3, 84
tekli işleç, 46
temel adres, 47, 80
time, 171, 181
time.h, 171, 180
time_t, 172, 180
tip dönüşümü, 16, 53
 otomatik, 16, 61, 73
tip niteleyicisi, 10, 67
tm, 180
TMP_MAX, 177
tmpfile, 178
tmpnam, 178
tolower, 53, 174, 179
toupper, 53, 174, 179
tutamak, 126
typedef, 93

U

UCHAR_MAX, 175
UINT_MAX, 175
ULONG_MAX, 175
ungetc, 124, 178
UNIX, 1
unsigned, 10
USHRT_MAX, 175
uzak adres, 145, 160
uzaklık, 145, 152, 160

Ü

üç nokta, 185
üçlü karakter, 170
üye, 90

V

va_arg, 177, 185
va_end, 177, 185
va_list, 177, 185
va_start, 177, 185
varsayılan alt dizin, 112, 115
veri dönüşüm fonksiyonu, 52
veri kesimi, 145
veri yapısı, 81, 104
vfprintf, 178
void, 60
void *, 98
volatile, 10
vprintf, 178
vsprintf, 178

W

wchar_t, 170, 177, 178
wcstombs, 170, 179
wctomb, 170, 179
while, 35

Y

yakın adres, 145, 160
yan etkisi, 14
yapı, 89
yapı künyesi, 90
yeniden yönlendirme, 106
yerel değişken, 60
yığıt, 77, 159
yığıt göstergesi, 77
yineleme, 78
yön bayrağı, 162
yöre, 169