00000

A'DAN Z'YE

GİTKİLAVUZU

Pusula **23**
"Kim Korkar..." dizisi **15**
A'dan Z'ye C Kılavuzu
Kaan Aslan
1. basım, Mart 1997
2. basım, Ocak 1998
3. basım, Kasım 1999
4. basım, Temmuz 2000
5. basım, Ocak 2001
6. basım, Ağustos 2001
7. basım, Ekim 2001
8. basım, Ocak 2002
Yayın yönetmeni: Mustafa Arslantunalı
Kapak tasarımı: Mustafa Arslantunalı
Düzelti: Alper Zorlu
İkonlar: Ragıp İncesağır
Sayfa düzeni: Gülnur Özkarabacak
Baskı: Acar Matbaası Tel: (0212) 422 18 00
Dağıtım sorumlusu: Hüseyin Üstünel (0212-252 42 80)

"Kim Korkar..." dizisi, Pusula Ltd. tarafından yayınlanmaktadır.

Pusula Yayıncılık ve İletişim Ltd.
Büyükparmakkapı Sok. No: 1/2 Beyoğlu/İstanbul
Tel.: 0212 252 42 80 (faks) 0212 293 15 44e-posta:
pusula@pusula.com web site: www.pusula.com

Pusula 23
"Kim Korkar..." dizisi **15**
A'dan Z'ye C Kılavuzu
Kaan Aslan
1. basım, Mart 1997
2. basım, Ocak 1998
3. basım, Kasım 1999
4. basım, Temmuz 2000
5. basım, Ocak 2001
6. basım, Ağustos 2001
7. basım, Ekim 2001
8. basım, Ocak 2002
Yayın yönetmeni: Mustafa Arslantunalı
Kapak tasarımı: Mustafa Arslantunalı
Düzelti: Alper Zorlu
İkonlar: Ragıp İncesağır
Sayfa düzeni: Gülnur Özkarabacak
Baskı: Acar Matbaası Tel: (0212) 422 18 00
Dağıtım sorumlusu: Hüseyin Üstünel (0212-252 42 80)

"Kim Korkar..." dizisi, Pusula Ltd. tarafından yayınlanmaktadır.

Pusula Yayıncılık ve İletişim Ltd.
Büyükparmakkapı Sok. No: 1/2 Beyoğlu/İstanbul
Tel.: 0212 252 42 80 (faks) 0212 293 15 44e-posta:
pusula@pusula.com web site: www.pusula.com

ÖNSÖZ

Son yıllarda bilgisayar alanındaki gelişmeler çoğumuzun kafasını karıştıracak nitelikte. Mikroişlemcilerin göreli olarak hızlanması, bellek fiyatlarının düşmesi, depolama birimlerinin genişlemesiyle yazılımda yeni bir döneme girdik. Sistem kaynaklarını cömertçe kullanan ayrıntılı yazılımlar eskilerinin yerlerini aldılar. Yazılımlardaki kod büyümesi, kullanılan araçların da seviyesini yükseltti. Görsellik en çok konuşulan kavramlardan birisi haline geldi. Nesne yönelimli tasarımlar programalama dillerini ve araçlarını etkisi altına aldı. Yine kanımca son yılların en önemli gelişmesi, kişisel bilgisayarlarda DOS'un yavaş yavaş bırakılması ve Windows gibi görsel tarafı kuvvetli işletim sistemlerinin yaygınlaşmasıdır. Büyük ve ayrıntılı yazılımların üstesinden gelebilmek, bunlara uygun yüksek seviyeli araçlar tasarlamak, programlama faaliyetini uzmanların elinden alarak herkese sunmak önemli hedefler arasındaymış gibi görünüyor...

Ancak yazılımın hızlı bir biçimde ilerlemesine ve programlama seviyesinin yükselmesine karşın C programlama dilinin yıldızı hiç sönmedi. Tersine, gün geçtikçe daha da parladı. Sistem programlama konusundaki vazgeçilmezliği onaylandı ve kabul edildi.

Beni bir C kitabı hazırlamaya iten nedenler nelerdir? Bir kere ülkemizdeki yazılım eğitiminin yeterli düzeyde olmadığına inanıyorum. İleri yazılım konusundaki Türkçe yayınların çok az ve nitelik bakımından da zayıf olduğunu düşünüyorum. Piyasadaki C kitapları gereksinimleri iyi bir biçimde karşılamıyor. Biz bu kitabı kurs notu olarak C ve Sistem Programcıları Derneği'nde zaten üç yılı aşkın bir süredir kullanıyoruz. Kitabımda uygulanan yöntemle C eğitimi konusunda önemli başarılar elde ettik. Bir kitap olarak çıkmasının pek çok kişiyi rahatlatacağını düşündüm. Bir C'ci olarak başka ne yapabilirdim ki?..

Peki, bu kitabı diğerlerinden ayıran özellikler nelerdir?

- Bilimsel bir yaklaşım. Yazılım terimlerini herkesin anlayabileceği biçimde, doğru ve yerinde kullanmaya özen gösterdim. (Beni Türkçe kitaplardaki terim karmaşası çok rahatsız etmiştir. Yazarların çoğu, yazılım terimlerini eksik bir biçimde ve yanlış yerlerde kullanıyor.)

- Karmaşık konular açık fakat eksiksiz bir biçimde ele alınmıştır. Bir konunun açık bir biçimde ele alınmasıyla kolaylaştırılması farklıdır. Kolaylaştırma sırasında yazar zor konuları atlar, onları yok sayar ya da yetersiz bir biçimde ele alır. Fakat ben kaçamak yaklaşımlar yerine, C eğitiminde zorlanılan kör noktaların üzerine gittim; bunların birçoğunu konu başlıkları haline getirdim.

- Kitap, sistem programlama konusundaki köklü çalışmaların ve eğitmenliğin getirdiği bir deneyimle ele alınmıştır. Bu yüzden, bir kişinin kendi başına çalışabileceği güçlü eğitimsel özelliklere de sahiptir. C programlama dilini yardım almadan kendi başına öğrenmenin zor olduğunu söylerim hep. Ancak kitabım okuyucuya bu şansı tanıyor. Örneğin, bölümler içerisinde sistem programlama alanında çalışacaklar için yazılmış "80X86 Sembolik Ma-

kine Dili Programcısına Not"lar göreceksiniz. Bu küçük açıklamaları aşağı seviyeli bilgilendirmeler için koydum. Ayrıca diğer programlama dillerinde çalışmış olanlar için düşülen notların da çok yararlı olduğu kanısındayım. Güçlü eğitimsel özellikleri nedeniyle, kitap C programlama dili eğitiminin verildiği fakültelerde ve yüksek lisans programlarında Türkçe ders kitabı olarak da kullanılabilir.

- Düzgün bir Türkçe. Bilgisayar kitaplarının çoğu Türkçe yanlışlarıyla dolu, bozuk, özensiz bir anlatıma sahip. Programlama diline gösterilen bu özen neden dilimize de gösterilmiyor?.. Bu kitabın düzgün Türkçe kullanımına da örnek olacağına inanıyorum.

Nesne yönelimli programlama tekniğinin yaygınlaştığı bir dönemde C++'a değil de standart C'ye ilişkin bir kitap yazmış olmam da yadırganabilir. Nesne yönelimli programlama ve C++ için önce standart C'nin eksiksiz bir biçimde bilinmesi gerekir. Deneyimlerimle gördüm ki, iki konunun birlikte yürütülmesi pek iyi sonuç vermiyor. Zaten C++'ın yaygınlaşması standart C'ye olan gereksinimi azaltmış da değil... Ayrıca bu kitabın devamı olacak biçimde bir C++ kitabı üzerindeki çalışmalarımı da bitirmek üzereyim. Kısa süre içerisinde basıma hazır hale geleceğini umuyorum.

Kitabın yazılması uzun bir zaman dilimi içerisinde, sürekli güncelleştirilerek gerçekleşti. Süreç içerisinde o kadar çok kişi doğrudan ya da dolaylı olarak yardım etti ki... Herkese isimleriyle burada tek tek teşekkür etmem olanaksız. Teknik düzeltmeleri Gürbüz Aslan yaptı. Ali Vefa Serçe kitabı defalarca benimle birlikte okudu ve hataların ayıklanmasında yardımcı oldu. Yücel Gündüz eksikliklerin giderilmesine çalıştı. C ve Sistem Programcıları Derneği Yönetim Kurulu üyeleri Cihat Atsever, Kazım Köroğlu ve Veli Duran öneri getirdiler. Dr. Turgay Seymen isim bulma konusunda yardımcı oldu. Genel Sekreter Yardımcımız Güray Sönmez uzun süre kitabın fotokopi çoğaltmaları ile ilgilendi ve kendisini bu eziyetten kurtarmam için bana sürekli ricada bulundu. Mustafa Arslantunalı ve Pusula Yayıncılık'ta çalışanlar kitabın basılmasını sağladılar.

Öğrencilerimin çoğu çeşitli biçimlerde yardımcı olmuştur.

Emeği geçen herkese teşekkür ederim.

Kitap hakkındaki görüşlerinizi C ve Sistem Programcıları Derneği yoluyla bana iletebilirsiniz.

Annemin anısına armağan ediyorum...

Kaan Aslan
C ve Sistem Programcıları Derneği
2. Taşocağı Cad. Oğuz Sok.
Barbaros Apt. No: 5/4
80700 Mecidiyeköy/İSTANBUL
Tel.: (0212) 274 63 60-274 99 89
275 88 97- 288 35 20 - 267 27 77
E-Posta: csystem@superonline.com
www.csystem.org

İÇİNDEKİLER

1 GİRİŞ

Bu bölüm yazılım mühendisliği ile ilgili temel bilgileri içermektedir. Yazılım kavramı, programlama dillerinin sınıflandırılması ve değerleme ölçütleri gibi konuların ele alınması *C* dilinin özelliklerini daha iyi anlamanıza yardımcı olacaktır.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Programlama dillerindeki seviye kavramı ne anlama gelmektedir?
2) Programlama dillerini birbirlerinden ayıran özellikler nelerdir?
3) Yapısal programlama tekniğinin temel özellikleri nelerdir?
4) Sistem programcılığı nedir?
5) *C*'nin temel özellikleri nelerdir; hangi özellikler onu *COBOL, PASCAL, FOXPRO, VISUAL BASIC* gibi dillerden ayırmaktadır?
6) *C* hangi yıllarda tasarlanmıştır; yaygınlaşması nasıl olmuştur?

1.1 YAZILIM NEDİR?

Yazılım, progr_mlamayı ve bu konuyla ilgili dökümantasyonları içeren genel bir terimdir. Yazılım deyince ilk olarak aklımıza programlama dilleri, bu diller kullanılarak yazılmış kaynak programlar ve çeşitli amaçlar için oluşturulmuş dosyalar gelir.
Yazılımı uygulama alanlarına göre 5 gruba ayırabiliriz.

1.2 YAZILIMIN SINIFLANDIRILMASI

1) Mesleki ve Ticari Yazılımlar
Çeşitli mesleklerde çalışan kişilerin işlerini kolaylaştırmak için hazırlanan yazılımlardır. Bu tür yazılımlar verilerin yaratılması, işlenmesi ve dosyalarda saklanması ile karakterize olurlar. İşlenen veri miktarları göreli olarak büyüktür. Dikkate değer bir veritabanı uygulaması içeren bu yazılımlarda zamanın büyük kısmı *giriş/çıkış* işlemlerinde harcanır. Muhasebe programları, adres etiket programları, stok kontrol programları, hasta takip programları bu gruba örnek olarak verilebilir.

2) Bilimsel ve Mühendislik Yazılımlar
Bilimsel ve mühendislik problemlerinin çözümünde kullanılan yazılımlardır. Yoğun olarak sayı ve sayı dizileriyle uğraşılır. Matematiksel ve istatistiksel algoritmaların ağırlıklı olarak kullanıldığı bu tür yazılımlarda işlenecek veri miktarı göreli olarak düşüktür. Bilimsel ve mühendislik yazılımların matematiksel karmaşıklığı en önemli karakteristikleridir. Elektronik devrelerin çözümünü yapan programları, simülasyon programlarını, bir binanın kiriş ve kolon hesaplamalarını yapan statik programlarını, istatistik analiz paketlerini örnek olarak verebiliriz. Bu tür yazılımlarda ağırlıklı olarak bilgisayarın *Merkezi İşlem Birimi (CPU)* kullanılmaktadır.

3) Yapay Zeka Yazılımları
İnsan davranışını taklit etmeyi amaçlayan yazılımlardır. Bunlara örnek olarak, satranç oynayan programları, uzman sistemleri, doğal dilleri anlama programlarını ve robot programlarını verebiliriz.

4) Görüntüsel Yazılımlar
Görüntü işlemlerinin ve algoritmalarının yoğun olduğu yazılımlardır. Oyun programları, animasyon programları bu çeşit yazılımlardır. Bu tür yazılımlarda ağırlıklı olarak bilgisayarın grafik arabirimi kullanılır.

5) Sistem Yazılımları
Bilgisayar donanımı ile arabirim oluşturan uygulama programlarına çeşitli yönlerden hizmet veren yazılımlardır. İşletim sistemleri, derleyiciler, editörler, haberleşme programları sistem yazılımlarına örnek olarak verilebilir. Sistem yazılımları, bilgisayar donanımına can veren yazılımlardır. Uygulama programlarına göre daha düşük seviyeli işlemler yaparlar.

1.3 PROGRAMLAMA DİLLERİNİN SINIFLANDIRILMASI

Programlama dillerini çeşitli yönlerden sınıflandırabiliriz. En sık kullanılan sınıflandırma biçimleri "seviyelerine göre" yapılan sınıflandırmalar ve "uygulama alanlarına göre" yapılan sınıflandırmalardır.

1.3.1 Programlama Dillerinin Seviyelerine Göre Sınıflandırılması

Önce seviye kavramının ne anlama geldiğini açıklamak istiyoruz. **Seviye, bir programlama dilinin insan algılamasına olan yakınlığının bir ölçüsüdür.** Yüksek seviyeli diller insan algılayışına daha yakın, alçak seviyeli diller de bilgisayarın doğal çalışmasına daha yakın olan dillerdir. Dillerdeki seviye yükseldikçe programcının işi de kolaylaşır. Öyle ki, çok yüksek seviyeli programlama dillerinde artık bir işin nasıl yapılacağına ilişkin değil, ne yapılacağına ilişkin komutlar bulunur. Seviye-

nin yükselmesi programcıya kolaylık sağlamakla birlikte genel olarak verimliliği ve esnekliği de azaltır. **Esneklik (flexibility)** ve **verimlilik (efficiency)** kavramları **1.5**'te ayrıntılı bir biçimde ele alınmaktadır.

Aşağıdaki şekli inceleyiniz:

Görsel programlama dilleri program kodunun kısmen ya da tamamen görsel biçimde çeşitli araçlar tarafından üretildiği dillerdir. Görselliğin ileri uçlarında program kodu bile bulunmayabilir. Windows sistemlerinde VISUAL BASIC, ACCESS gibi görsel diller yaygın bir biçimde kullanılmaktadır. Çok yüksek seviyeli dillere **dekleratif diller** de denir. Veritabanlarının yönetimlerinde kullandığımız dilleri bu gruba sokabiliriz. Yüksek seviyeli programlama dilleri daha algoritmik dillerdir. Bu dillerde önce işlerin nasıl yapılacağına ilişkin algoritmalar tasarlanır. Daha sonra bu algoritmalar program koduna çevrilir. BASIC, PASCAL, FORTRAN gibi dilleri bu grup içerisinde ele alabiliriz.

O halde bir soru : Algoritma nedir?

Algoritma, bizi bir problemin çözümüne adım adım götüren genel yöntemlerdir. Yazılıma ilişkin problemleri çözen programlar, yöntem aynı kalmak üzere çok çeşitli dillerde kodlanabilirler. Şöyle bir örnekle açıklayabiliriz: *Türkçe, İngilizce* ve *Almanca* bilen bir arkadaşınız olsun. Ondan bir mektubu postaya atmasını bu üç dilde de rica edebilirsiniz. Fakat, postanenin yerini bilmeniz ve tarif edebilmeniz gerekir. Ancak algoritmayı dilden tamamen bağımsız olarak düşünmek de doğru değildir. Kullandığımız dilin özelliklerini algoritmaya yansıtarak daha etkin tasarımlar geliştirmek isteyebilirsiniz.

Örneğin çok yüksek seviyeli bir dil olarak ele aldığımız DBASE ya da FOXPRO'da :

SORT ON ADI TO SADI

komutu ile bir veri tabanını **ADI** alanına göre sıraya dizilerek yeni bir **SADI** dos-

yası yaratılır. Oysa, seviyesi daha düşük olan dillerde bu işlemi yapmak için bir algoritma tasarlamak ve onu program kodu biçiminde yazmak gerekir.

Orta seviyeli dillere gelince; bu diller hem kullanıcıya hem de bilgisayara yakın olan yapılar içerirler. Orta seviyeli diller, yüksek seviyeli dillerin kolaylıkları ile aşağı seviyeli dillerin esnekliğini ve doğallığını kullanırlar. İşte *C* tipik bir orta seviyeli dildir. Orta seviyeli diller özellikle sistem programlarının yazımında tercih edilirler.

Alçak seviyeli diller ise sembolik makina dillerini ve doğal makina dillerini kapsar. Sembolik makina dillerinde (assembly languages) mikroişlemci komutlarının daha algılanabilir olan sembolik biçimleri kullanılmaktadır. Ne var ki, bu dillerde bir programın tamamını yazmak oldukça zahmetlidir. Doğal makina dilleri ise yalnızca *0* ve *1* rakamlarının dizilimlerinden oluşan komutları içerirler. Zaten sayısal bilgisayarlarda her şey 2'lik sistemde 1'ler ve 0'lardan oluşan sayılardan ibarettir. Daha şimdiden *C* öğrencisinin şunu bilmesi gerekiyor:

Bilgisayarda her şey 2'lik sistemdeki sayılardan oluşur. Bu sayıların kim tarafından ve nasıl yorumlanacağı önemlidir. Bellekteki bir sayı:

- Bir makina komutu olarak yorumlanabilir.
- Bir karakter olarak yorumlanabilir. O zaman, 1 byte bilginin hangi karaktere karşılık geleceğini gösteren bir dönüştürücü tabloya ihtiyaç duyulacaktır.. (ASCII, EBCDIC tabloları gibi....)
- Bir sayı olarak yorumlanabilir.

1.3.2 Programlama Dillerinin Uygulama Alanlarına Göre Sınıflandırılması

1) Bilimsel ve Mühendislik Diller : Bu diller daha çok bilimsel ve mühendislik problemlerinin çözümünde tercih edilirler. *PASCAL* ve *C* dillerini, bir de geleceği pek parlak olmayan ve hala ısrarla kullanılan 90 canlı, dünyanın ilk yüksek seviyeli dili *FORTRAN*'ı buna örnek verebiliriz.

2) Veritabanı Programlama Dilleri: Bu diller veritabanlarının genel olarak yönetiminde kullanılan dillerdir: *DBASE, PARADOX, FOXPRO, SQL*,.. kişisel bilgisayarlarda yaygın olarak kullanılanlardan bazıları.

3) Yapay Zeka Dilleri: Bu diller insan davranışını taklit etmeye yönelik yapay zeka içeren programların yazımında kullanılan mantıksal dillerdir. En ünlüleri: *LISP* ve *PROLOG*.

4) Genel Amaçlı Diller: Çok çeşitli konularda uygulama geliştirmek amacıyla kullanılan dillerdir. *C* ve *PASCAL*'ı yine örnek olarak vereceğiz.

5) Sistem Programlama Dilleri: Sistem programlarının yazımında kullanılan dillerdir. *C*'yi ve sembolik makina dillerini bu grup içinde ele alabiliriz.

1.4 SİSTEM PROGRAMLAMA

Sistem programları, bilgisayar donanımı ile arabirim oluşturan, uygulama programlarına çeşitli yönlerden hizmet veren programlardır. Sistem programlarını uygulama programlarından ayıran iki önemli özellik vardır. Birincisi, bu tür programların bilgisayar donanımlarına olan yakınlığıdır. Sistem programları bilgisayar donanımları ile doğrudan etkileşim halindedir. Çoğu zaman donanımı yöneten ya da düzenleyen işlevlere sahiptirler. Çeşitli donanım birimlerini hedef alır, onları programlar ve belirli amaçlara yöneltirler. Bu nedenle sistem programcıları belli seviyelerde bilgisayar donanımını da bilmek zorundadırlar. Sistem programlarını uygulama programlarından ayıran diğer bir özellik, sistem programlarının uygulama programlarına hizmet vermesidir. Örneğin editörler, derleyiciler gibi sistem programları olmasaydı uygulama programlarını geliştirebilir miydik?

Sistem programlarının hızlı çalışması istenir . Bu yüzden etkin ve verimli yazılmaları gerekir. Bu tür programların kendine özgü bir algoritmik yapıya sahip olduklarını söyleyebiliriz. Sistemdeki değişmelere ve rassal etkenlere karşı sürekliliğini sağlama zorunluluğu içinde bol kontrollü ve algoritmik yapıları vardır. Bu özellikleri ile sistem programcılığı yazılımın en araştırmacı ve atılımcı kanadını oluşturur.

1.5 PROGRAMLAMA DİLLERİNİN DEĞERLEMESİ

Programlama dillerini birbirlerinden ayıran çeşitli özellikler vardır. Acaba bir programlama dilinin diğerine karşı daha fazla tercih edilmesinin nedenleri ne olabilir? Kuşkusuz bütün dillerin yapısı hemen hemen diğerlerinin aynısı olsaydı, bu kadar sayıda dil belki de hiç ortaya çıkmayacaktı. Yazılımcıları yeni bir dil tasarlamaya iten en önemli etken, onun belli konularda sağlayacağı avantajlardan kaynaklanmaktadır.

Programlama dillerini incelemek üç grup için çok önemlidir.

1) Dil tasarımcıları
2) Derleyicileri yazanlar
3) Profesyonel programcılar

Dil tasarımcılığı biraz bulanık bir kavramdır. Çünkü bazı diller bir ya da birkaç kişi tarafından tasarlandığı halde, bazıları yıllarca çok değişik kişiler tarafından tasarlanmış ve geliştirilmişlerdir. Eskiden dilleri tasarlayanlar aynı zamanda bu dilin derleyicilerini de yazarlardı. Fakat bugün dil tasarımcılarıyla derleyici yazarları tamamen değişik kişiler de olabilmektedir. Profesyonel programcılar ise mesleği program geliştirmek olan kişilerdir. Profesyonel programcılar, programlama dillerinin inceliklerini, onlardan en yüksek düzeyde faydalanmak için iyi bilmek zorundadırlar.

Programlama dillerinin değerlemesine ilişkin ölçütler 12 başlık halinde incelenebilir.

1) İfade Gücü (Expressivity)

Algoritmayı tasarlayan kişinin niyetlerini açık bir biçimde yansıtma yeteneğidir. İfade gücü yüksek bir dil, tasarlandığı alanda kullanılan notasyonu da içerir. Örneğin, bir matematikçi algoritmasını matematikte kullandığı sembollerle ifade et-

mek ister. İfade gücü okunabilirlik kavramıyla da sıkı bir ilişki içindedir. *C*, *PASCAL* gibi yapısal diller aynı zamanda ifade gücü yüksek olan dillerdir.

2) Veri Türleri ve Yapıları (Data types and structures)
Çeşitli veri türlerini (tamsayı, gerçek sayı, karakter,...) ve veri yapılarını (diziler, kayıtlar,...) destekleme yeteneğidir. Veri yapıları, veri türlerinin oluşturduğu mantıksal birliklerdir. Örneğin *C* ve *PASCAL* veri yapıları açısından oldukça zengindir.

3) Giriş/Çıkış Kolaylığı (Input/Output facilities)
Sıralı, indeksli ve rastgele dosyalara erişme, veritabanı kayıtlarını geri alma, güncelleştirme ve sorgulama yeteneğidir. Veritabanı programlama dillerinin (*DBASE*, *PARADOX*, vs) bu yetenekleri diğerlerinden daha üstündür ve bu dillerin en tipik özelliklerini oluşturur. Fakat *C*, *giriş/çıkış* kolaylığı kuvvetli olmayan bir dildir. *C*'de veritabanlarının yönetimi için özel kütüphanelerin kullanılması gerekir.

4) Taşınabilirlik (Portability)
Bir programlama dilinde yazılmış kaynak kodun başka sistemlerde de sorunsuz derlenerek çalışabilmesi anlamına gelir. Yüksek seviyeli dillerin en önemli özelliklerinden birisi olan taşınabilirlik, ortak bir tasarımı ve standardizasyonu gerektirir. Taşınabilirlik kaynak kod için kullanılan bir terimdir. Bir dilin taşınabilir olması ancak bütün sistemlerde derleyicileri yazanların önceden belirlenmiş ortak tanımlamalara uyması ile mümkün olabilir. Dillerin taşınabilirliği genel olarak seviye düştükçe azalmaktadır. Bu durumda en az taşınabilir dillerin makina dilleri olduğunu söyleyebiliriz. Orta seviyeli olmasına karşın *C* programlama dili taşınabilirlik açısından diğer dillere kıyasla en önemli yeri tutmaktadır. Örneğin BASIC derleyicileri arasında büyük farklar görebilirsiniz. Bir sistemdeki BASIC derleyicisinde olan komutlar başka sistemlerde olmayabilir. Bu durumda BASIC dilinin taşınabilir olmasından söz edebilir miyiz?

Mükemmel bir taşınabilirliğin hiçbir dil için mümkün olmadığını da belirtelim. Derleyici paketlerinin yeni uyarlamalarının çoğu birtakım yenilikleri de beraberinde getirmektedir. Bu durumda programcının hangi komutların ve yapıların taşınabilir olduğunu bilmesi büyük önem taşır.

5) Altprogramlama Yeteneği (Modularity)
Kaynak programların altprogramlara ayrılarak parçalanabilme özelliğine denir. Altprogram kullanımının pek çok faydası vardır. Bunları tek tek gözden geçirelim.

- **Altprogramlama kodu küçültür.** Çok tekrarlanan işlemlerin altprogramlar kullanılarak yazılması çalışabilir programın kodunu küçültür. Çünkü altprogramlar bir kere çalışabilir kod içerisine yazılırlar, ancak program kodunun buraya atlatılarak bu bölgenin defalarca çalıştırılması mümkündür.

- **Altprogramlama algılamayı kolaylaştırır.** Altprogramlama *soyutlamayı (abstraction)* artırarak algılamayı kolaylaştırmaktadır. Nesne yönelimli programlama tekniğinin de anahtar terimlerinden birisi olan soyutlama, ayrıntıların gözardı

edilmesi anlamına gelir. Program kodunu inceleyen bir kişi altprogramın nasıl yazıldığından çok ne yaptığıyla ilgilenmek ister.

- Altprogramlama test olanaklarını artırır. Hata araştırılırken hatanın hangi altprogramlarda olduğunun belirlenmesi ve yalnızca o altprogramın tekrar gözden geçirilmesi yoluyla test olanakları artırılmış olur. Bunun dışında, altprogramların yalnız başlarına test edilebilmesi de önemli bir avantajdır.

- Altprogramlama kaynak kodun güncelleştirilebilirliğini ve yeniden kullanılabilirliğini artırır. Kaynak kodda istenilen değişiklikler, altprogramlara yansıtılarak daha kolay yapılabilir. Bunun dışında, daha önceden yazılmış altprogramlar başka başka projelerde defalarca kullanılabilir. Program kodlarının yeniden kullanılması (reusability) başka projelerin tasarım zamanını da azaltmaktadır.

Altprogramlama, yapısal programlama tekniğinin de belkemiğini oluşturur. C dili altprogramların yoğun olarak kullanıldığı - C de altprogramlara fonksiyon denir- atomik bir dildir.

6) Verimlilik (Efficiency)

Bir dilde yazıldıktan sonra derlenerek amaç koda dönüştürülmüş programların hızlı çalışabilmesidir. Her ne kadar amaç kodun hızlı çalışması aslında derleyicilerin kendi verimliliğine bağlıysa da, dilin seviyesinin ve genel yapısının çalışma hızı üzerinde önemli bir etkisi olduğu da yadsınamaz. Örneğin C programlarının hızlı çalıştığı ve az yer kapladığı bilinir. Çalışabilir kodun küçüklüğü ile çalışma hızı arasında da çoğunlukla doğru bir orantı vardır.

7) Okunabilirlik (Readability)

Okunabilirlik, kaynak kodun çabuk ve kuvvetli bir biçimde algılanabilmesi anlamına gelen bir terimdir. Kaynak kodun okunabilirliğinde sorumluluk büyük ölçüde programı yazanın omuzlarındadır. Fakat, yine de -verimlilikte olduğu gibi- dillerin bir kısmında okunabilirliği güçlendiren yapılar ve mekanizmalar bulunduğu için bu özellik *"bir ölçüde dile de bağlıdır"*, diyebiliriz. En iyi program kodu, sanıldığı gibi *"en zekice yazılmış fakat çok az kişinin anlayabileceği kod"* değildir. Birçok durumda iyi programcılar okunabilirliği hiçbir şeye feda etmek istemezler. Çünkü okunabilir bir program kolay algılanabilme özelliğinden dolayı seneler sonra bile güncelleştirmeye olanak sağlar. Birçok kişinin ortak kodlar üzerinde çalıştığı projelerde okunabilirlik daha da önem kazanmaktadır.

C'de okunabilirlik en fazla vurgulanan kavramlardan birisidir. Biz de kitabımızda bu kavramı bıkmadan defalarca vurgulamak niyetindeyiz. Öğrencilerimize sıklıkla şunları söyleriz:

"Yetkin bir C programcısıyla deneyimsiz bir C programcısını ayıran en önemli özelliklerden birisi okunabilirliktir. Bir C programcısının yetkinliği 10 satırlık kaynak koddan bile anlaşılabilir."

8) Esneklik (Flexibility)

Esneklik, programlama dilinin programcıyı kısıtlamaması anlamına gelir. Esnek bir dilde derleme hataları daha azdır; birçok işlem, hata riskine karşın programcı

için serbest bırakılmıştır. Programcı bu serbestlikten ancak iyi bir programcıysa kazanç sağlayabilir. Fakat deneyimsiz bir programcıysa zarar da görebilir. Esnek dillerde programcıyı olası hatalardan uzak tutmak için kısıtlamalara ve yasaklamalara gidilmez. Örneğin *C* çok esnek bir dildir, karakter türüyle tamsayı türü karşılıklı birbirine atanabilir. Programcı bu özelliği yerinde kullanarak bundan algoritmik bir kazanç sağlayabilir.

9) Öğrenme Kolaylığı (Pedagogy)

Her programlama dilini öğrenmenin zorluğu aynı değildir. Yüksek seviyeli dillerin eğitimi, düşük seviyeli dillerin eğitiminden daha kolaydır. Örneğin *FOXPRO*, *BASIC* gibi dillerin bu derece sevilmesinin nedenlerinden bir de çabuk öğrenilebilmesidir. *C* maalesef eğitimi zor ve zahmetli olan bir dildir. Birçok eğitim kurumu *C* öğrencileri için önkoşullar aramaktadır. (Örneğin *C* ve *Sistem Programcıları Derneği*'nde *C* öğrencileri, diğer programlama dillerinden birinde uygulamalı çalışma yapmış olanlardan seçilir.)

10) Genellik (Generality)

Programlama dilinin çok çeşitli uygulamalarda etkin olarak kullanılabilmesidir. Örneğin, *COBOL* bilimsel ve mühendislik yazılımlarda tercih edilmez; *CLIPPER* ya da *FOXPRO* bir veritabanı dilidir. Oysa *C*, *PASCAL*, *BASIC* daha genel amaçlı dillerdir.

11) Yapısallık (Support for structural programming)

Yapısal programlama, bir programlama tekniğinin ismidir. Yapısal programlamayı destekleyen diller de yapısal diller olarak adlandırılırlar. Peki yapısal programlama nedir?

Yapısal programlamada bloklar halinde yazım ön plandadır. Program akışında atlamalar yapılması okunabilirliği ve algılamayı güçleştirdiği için istenmez. Yapısal programlamada altprogramların kullanımı önemli bir yer tutar. Yapısal diller aynı zamanda modüler dillerdir. Modüler tasarımla, programlar küçük parçalara ayrılarak soyutlama sağlanır. İfade gücü, okunabilirlik, taşınabilirlik gibi özellikler de yapısal programlamayı desteklemektedir.

12) Nesne Yönelimlilik (Object orientation)

Son yıllarda adından oldukça sık söz edilen *"nesne yönelimlilik"* de *"yapısallık"* gibi bir programlama tekniğidir. Bugün, programlama dillerinin büyük bölümünün nesne yönelimli uyarlamaları yazılmıştır. Nesne kavramını ve nesne yönelimliliğin ne olduğunu açıklayamayacağız. *C* programlama dilinin nesne yönelimli programlama tekniğini destekleyen uyarlamasına *C++* (*Si plas plas* diye okuyunuz) denilmektedir.

1.6 C NASIL BİR DİL?..

Yukarıdaki tüm bilgilerden *C* için şu sonuçları çıkarabiliriz:

- **Orta seviyeli bir dildir.** Yazılan *C* kodu ile makina kodu arasında bağlantı kolaylıkla kurulabilir.
- **Sistem programlama dilidir.** Bugün işletim sistemleri, derleyiciler, editörler gibi sistem programlarının hemen hepsi yoğun olarak *C* kodu içermektedir. Ancak sistem programlamanın dışında da birçok uygulamada *C* verimli olarak kullanılabilir.
- **Algoritmik bir dildir.** Yalnızca dilin sintaks ve semantik yapısını bilmek yetmez. Problemleri çözebilecek bir algoritma bilgisine de ihtiyaç duyulur.
- **Diğer diller arasında taşınabilirliği en fazla olanlardan biridir.**
- **İfade gücü yüksek ve okunabilirlik özelliği kuvvetli bir dildir.**
- **Çok esnektir.** Bu yüzden programcının hata yapmayacak bir bilgiye ve deneyime sahip olması gerekir.
- **Atomik bir dildir.** *C*'de altprogramlama tekniği ileri düzeyde kullanılmaktadır.
- **Güçlü bir dildir.** Tasarım özellikleri çok iyidir. *C*'ye ilişkin yapıların ve operatörlerin bir kısmı daha sonra diğer diller tarafından da benimsenmiştir.
- **Verimli bir dildir.** *C* programları seviyesi dolayısıyla daha hızlı çalışır.
- **Doğal bir dildir.** *C* bilgisayar sisteminin çalışma biçimiyle uyum içindedir.
- **Eğitimi zor bir dildir.** Öğrenebilmek için diğer dillerden daha fazla çaba gerekir.
- **Yapısal bir dildir.**
- ***C++* ile nesne yönelimlilik özelliğine de sahip olmuştur.**

1.7 C PROGRAMLAMA DİLİNİN TARİHİ

C, *AT&T* laboratuarlarında 70'li yılların başında **Dennis Ritchie** tarafından tasarlanmış ve yazılmıştır. Ritchie o yıllarda **B** adlı programlama dilininin tasarımcısı olan **Ken Thomson** ile birlikte *UNIX* işletim sistemi üzerinde çalışıyordu. O zamanın yapısal programlama dilleri olan *ALGOL 60*, *CPL*, *BPCL* sistem programlama açısından oldukça yetersizdi. Ritchie böylece Thomson tarafından tasarlanan *B* diline yeni fikirler getirerek *C* dilinin temellerini atmış oldu. *UNIX* işletim sisteminin sonraki uyarlamalarında yoğun olarak hep *C* kullanılmıştır.

C önceleri geniş kitleler tarafından tanınmıyordu. *C*'nin bütün dünyada tanınması ve yıldızının parlaması 1978'de Dennis Ritchie ve Brian Kernighan tarafından yazılan *"The C programming Language"* kitabıyla birlikte oldu. Bu kitap aynı zamanda yazılım konusunda bugüne kadar yazılmış en iyi eserler arasında yer almaktadır. Önceden yalnızca *UNIX* tabanlı sistemleri çağrıştıran *C*, kişisel bilgisayarların 1980'li yıllarda yaygınlaşmasıyla en fazla tercih edilen programlama dilleri arasına girmiştir. Bugün artık hemen her tür sistemde *C* derleyicilerine rastlamak mümkündür.

C, 1983 yılında *ANSI* tarafından standardize edildikten sonra yüksek oranda

taşınabilir bir sistem programlama dili olmuştur. Bugün işletim sistemleri, derleyiciler, editörler gibi sistem programlarının yazım dili C'dir.

SORAMADIKLARINIZ...

S1) İyi bir programcı olmak için hangi programlama dillerini bilmek gerekir?

C1) Programlama dilleri yalnızca birer araçtır. Çok dil bilmenin iyi programcılıkla doğrudan bir bağlantısı yoktur. Önemli olan programcının üzerinde çalıştığı yazılımı etkin bir biçimde gerçekleştirebilmesidir. Zaten uygulamada birkaç dille aynı anda çalışan programcılara pek rastlanmaz. İyi bir *C* programcısının -zorunluluk olmadıktan sonra -örneğin *PASCAL, BASIC* gibi dilleri kullanmasına da gerek yoktur.

Kişiyi iyi bir programcı yapan özelliğin *"birden fazla programlama dilini bilmek"* olduğunu düşünmüyoruz. (Bir programlama dilini bilmek ne anlama geliyor, orası da biraz bulanık!). İyi bir programcının, uygulama konusu hakkında bilgiye, güçlü bir algoritma bilgisine, deneyime, problemleri analiz etme ve çözüm bulabilme gibi yeteneklere de sahip olması gerekir. Fakat yine de, bir alçak seviyeli (assembly dili), bir orta seviyeli (C), bir de çok yüksek seviyeli programlama dilinde uygulama geliştirebilecek düzeyde deneyim sahibi olunmasını tavsiye ederiz.

S2) Yüksek seviyeli diller insana daha yakın olduğuna göre, bunları kullanmak daha iyi değil mi? Seviyesi düşük dillerin ne avantajları var?

C2) Yüksek seviyeli dillerde birçok konuda program geliştirmek daha kolay, bu doğru. Fakat bu dillerde de düşük seviyeli işlemleri gerçekleştirmek oldukça zordur. Örneğin *CLIPPER* ile bir haberleşme programı yazılabilir mi? Yazılsa bile verimli olur mu? Yüksek seviyeli dillerde yazılan kaynak kod ile makina kodu arasında ilişki kurmak çok güçtür. Oysa örneğin, *C*'de yazılan bir programın sembolik makina kodları programcı tarafından kolaylıkla tahmin edilebilir. Seviyesi düşük dillerle bilgisayarı daha iyi kontrol edebilirsiniz. İşlerin tam istediğiniz gibi yapılmasını sağlayabilirsiniz. Elinizde daha fazla yetki ve daha fazla sorumluluk vardır.

S3) *C* taşınabilir bir dil olduğuna göre, örneğin *DOS*'ta yazıp derleyerek çalışabilir hale getirilen bir *.EXE* kodu *UNIX*'de de çalışabilir mi?

C3) Kaynak programın taşınabilirliği söz konusudur. En az taşınabilir diller makina dilleridir. *C*'de yazdığınız kaynak kodu, *UNIX* altında çalışan başka bir *C* derleyicisiyle tekrar derlemeniz gerekir.

S4) Yapısal programlama dillerinde **goto** gibi program akışını değiştiren deyimlerden kaçınmak gerektiği söyleniyor, bunun nedeni nedir?

C4) Yapısal programlama tekniğinde programın kolay algılanabilir olması önemlidir. Program akışının sık sık değiştirilmesi tasarım, test, program sonrası bakım gibi işlemleri zorlaştırır. Yapısal dillerde bu yüzden bloklu bir yazım biçimi öngörülmüştür. Fakat bütün bunlar hiç **goto** kullanılmaması gerektiği anlamına da gelmiyor. Bazı özel durumlarda **goto** kullanmak programı daha algılanabilir bir hale bile getirebilir. Körü körüne bir **goto** düşmanı olmamak gerekir.

GENEL KAVRAMLAR

2

Bu bölüm işletim sistemleri, derleyiciler, sayı sistemleri gibi temel kavramlara ayrılmıştır. C'yi öğrenirken faydalanacağımız bu temel kavramların çok iyi anlaşılması gerekir.

Bu bölümü inceledikten sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) İşletim sistemlerinin görevleri nelerdir?

2) Derleyicilerin görevleri nelerdir ve yorumlayıcılarla aralarında ne farklılıklar vardır?

3) **80X86** mikroişlemcilerinin adresleyebildiği belleğin organizasyonu nasıldır?

4) Pozitif ve negatif sayılar nasıl elde edilirler?

5) 2'lik, 8'lik ve 16'lık sistemlerin bilgisayarlar için önemi ve anlamı nelerdir?

6) Algoritmanın karmaşıklığı ne anlama gelmektedir?

2.1 İŞLETİM SİSTEMİ (Operating System)

İşletim sistemi bilgisayar donanımını yöneten, kullanıcıyla bilgisayar sistemi arasında ilişki kuran aşağı seviyeli bir sistem programıdır. Çoğu kişi işletim sistemini yalnızca birtakım komutlara karşılık veren bir program olarak düşünse de aslında komutları yorumlamak ve çalıştırmak işletim sisteminin görevlerinden yalnızca biridir. Bir işletim sistemini işlevsel olarak *"kabuk"* ve *"çekirdek"* olmak üzere iki bölüme ayırabiliriz.

İşletim sisteminin donanımı yöneten kısmına **çekirdek (kernel)**, kullanıcı ile arabirim oluşturan kısmına ise **kabuk (shell)** ya da **komut yorumlayıcı (command interpreter)** denilmektedir. Çekirdek kısmının tasarımı kabuk kısmının tasarımından çok daha zordur.

İşletim sistemlerini birer kaynak yöneticisi olarak da tanımlayabiliriz. Kaynak yöneticisi olarak bir işletim sistemi şu görevleri yerine getirir:

1) Kaynakları izleyerek onların ne durumda olduğunu belirler.

2) Kimin, hangi kaynakları ne zaman, ne kadar süre ile kullanacağına karar verir.

3) Kaynakları belirlenen işlemler için tahsis eder.

4) İşlemler bitince kaynakları serbest bırakır.

İşletim sistemlerinin yönetmekle görevli olduğu sistem kaynakları, *Merkezi İşlem Birimi, Bellekler, Dosyalar* ve *Aygıtlardan* oluşur.

1) Merkezi İşlem Birimi (CPU)

Mikroişlemci de denilen bu birim, bilgisayar sisteminin beynini oluşturmaktadır. Örneğin, aynı anda birden fazla programın çalıştığı işletim sistemlerinde programların mikroişlemciye ne zaman ve ne kadar süre ile atanacağı gibi kararlar işletim sistemi tarafından verilir.

2) Bellek (Memory)

Programları belleğe yüklemek, çalışması bitince belleği boşaltmak gibi işlemler de işletim sistemleri tarafından yapılır.

3) Dosya (File)

Dosyalara erişim, dosyaların kontrolü ve yönetilmesi gibi işlemler, işletim sistemlerinin denetimi altındadır.

4) Aygıt (Device)

Disk, yazıcı gibi aygıtların kontrolünden de işletim sistemleri sorumludur. Örneğin, çok kullanıcılı bir işletim sisteminde iki kişi aynı anda yazıcıdan çıktı almak istediğinde olası çatışma durumu işletim sistemi tarafından çözülmek zorundadır.

Görevleri aynı olan işletim sistemlerini birbirlerinden ayıran çeşitli farklılıklar vardır. Bunların en önemlisi bir işletim sisteminin *"çok işlemli (multi processing)"* olup olmamasıdır. Çok işlemli sistemlerde birden fazla program aynı zaman dilimi içinde çalışabilir. Örneğin *DOS* tek işlemli bir işletim sistemidir. Aynı anda birden çok programın çalışması için tasarlanmamıştır. *UNIX, WINDOWS, OS/2* gibi işletim sistemleri ise çok işlemlidir. Bilgisayar donanımına en yakın sistem programları olan işletim sistemlerinin ayrıntılı fonksiyonlarını bilmek sistem programcılığı için de çok önemlidir. Çünkü işletim sisteminin seviye olarak yukarısında çalışacak olan her sistem programı sonuçta işletim sisteminden yardım almak zorundadır. Sistem programcısı için işletim sistemini bilmek demek, "hangi komutların ne yaptığını" bilmek demek değil, onları "yazabilmek ya da gerektiğinde değiştirebilmek" demektir.

2.2 ÇEVİRİCİ PROGRAMLAR

Çevirici programlar, herhangi bir programlama dilinde yazılmış olan kaynak programı girdi olarak alan ve bunu işlevleri aynı olacak biçimde başka bir dile dönüştüren programlardır.

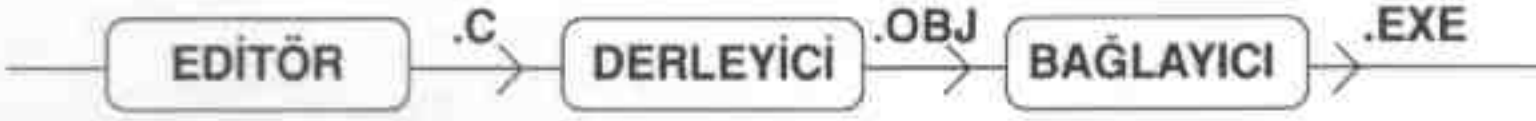
Kaynak ve amaç programların farklı programlama dilleri olduğuna dikkat etmelisiniz. Bir çevirici programda kaynak programa ilişkin programlama diline kaynak dil, amaç programa ilişkin programlama diline de amaç dil denilmektedir.

2.3 DERLEYİCİLER (Compilers)

Kaynak dili yüksek seviyeli bir dil, amaç dili alçak seviyeli bir dil (sembolik makina dili ya da makina dili) olan çevirici programlara derleyici denir. Uygulamada alçak seviyeli dil çoğunlukla makina dilidir.

Derleyicilerin çıktısı olan amaç kodun işletim sistemlerine göre anlamı değişebilir. Örneğin, 80X86 tabanlı mikroişlemcilerin kullanıldığı *DOS, WINDOWS, UNIX* işletim sistemlerinde derleme işlemi sonucunda oluşan amaç koda, **yeniden yüklenebilen amaç kod (relocatable object module)** denilmektedir. DOS işletim sisteminde *"çalışabilir bir kod"* üretmek için **bağlayıcı (linker)** denilen alçak seviyeli bir dönüştürücü programa daha ihtiyaç duyulmaktadır.

Aşağıda, *DOS* işletim sisteminde bir *C* programının hangi aşamalardan geçerek çalışabilir bir kod haline getirileceği şekilsel olarak açıklanmaktadır.



Kaynak programlar, editörler kullanılarak yazılırlar. Daha sonra bu kaynak kodlar derleyiciler tarafından amaç koda çevrilirler (relocatable object module). En sonunda ise bağlayıcı programlar ile *"çalışabilir"* hale getirilirler. Derlenen amaç kodun (.OBJ), *DOS*'ta belli bir standardı vardır. Bütün bağlayıcılar bu standarttaki dosyaları *.EXE* koduna çevirebilirler.

Derleme işlemi pekçok nedenden dolayı başarıyla sonuçlanmayabilir. Nedeni ne olursa olsun derleyiciler, derleme işlemi sırasında oluşan hataları programcıya *"hata mesajları"* biçiminde iletirler.

2.4 HATA MESAJLARI

Derleme işlemi sırasında derleyici tarafından tespit edilen hata mesajlarını 3 kısma ayırabiliriz:

1) Uyarılar (Warning)
Uyarılar, amaç kod oluşumunu engellemeyecek türde olan hatalardır. Genellikle programcının yapmış olabileceği *"olası bir yanlışlığa"* dikkati çekmek amacıyla verilirler. Her ne kadar amaç kod oluşumunu engellemiyorsa da uyarıların mutlaka programcı tarafından gözden geçirilmesi gerekir.

2) Gerçek Hatalar (Error)
Gerçek hatalar, amaç kod oluşumunu engelleyecek derecede ciddi olan hatalardır. Genel olarak sintaks ve semantik yanlışlıkları sonucununda oluşan gerçek hataların mutlaka düzeltilmeleri gerekir.

3) Ölümcül Hatalar (Fatal Error)
Bu tür hatalarda derleme işlemi dahi bitirilemez. Mesaj iletildikten sonra derleme işlemine derhal son verilir. Ölümcül hatalar genellikle sistemdeki önemli problemler nedeniyle ortaya çıkarlar. Örneğin bir derleyici, derleme işlemi sırasında diskte geçici bir dosya oluşturacakken, diskte hiç boş yer yoksa bu durumda şöyle bir ölümcül hata ile karşılaşabilirisiniz:

fatal error: disk full

Türü ne olursa olsun hata mesajlarına gereken önem verilmelidir. Deneyimsiz programcılar hata mesajlarını pek dikkate almazlar. Oysa, hangi tür hatalara karşı hangi hata mesajlarının verildiğini bilmek, tekrarlanan hataların düzeltilmesini oldukça kolaylaştırmaktadır.

2.5 YORUMLAYICILAR (Interpreters)

Derleyici ve yorumlayıcı kavramları sıklıkla birbirlerine karıştırılır. Derleyiciler, yukarıda da belirttiğimiz gibi, bir kaynak programı alarak ondan başka bir amaç program oluştururlar. Oysa yorumlayıcılar, kaynak programı kısım kısım -burada satır satır terimi de kullanılır- ele alarak doğrudan çalıştırırlar. Yorumlayıcılar standart bir amaç kod üretmezler. Yorumlama işlemi aşama aşama yapılmadığı için genellikle ilk hatanın bulunduğu yerde programın çalışması da kesilir.

Yorumlayıcılar derleyicilere göre çok daha kolay yazılabilirler. Fakat yorumlayıcıların bir programı icra zamanı (run time) derleyicilere göre oldukça uzundur. Bunun yanısıra yorumlayıcıların her zaman kaynak koda ihtiyaç duymaları da kaynak kodun gizliliği ve güvenliği açısından bir dezavantajdır.

2.6 DERLEYİCİLERE OLAN GEREKSİNİM

Mikroişlemcilerin doğrudan işleyebileceği komutlar 1'ler ve 0'lardan oluşan doğal makina dili komutlarıdır. Ancak makina dilinde program yazmak oldukça güçtür. Çünkü her şeyin 1'ler ve 0' lardan ibaret olduğu böyle bir sistemde neyin bir komut, neyin bir sayı ve neyin bir karakter bilgisi olduğu kolaylıkla algılanamaz. Bilgisayarların işleyebildiği kod doğal makina kodu olduğuna göre, yüksek seviyeli programlama dillerinde yazdığımız ve bizim kolaylıkla algıladığımız ancak bilgisayarların anlamadığı bu programları makina diline kim çevirecektir? İşte derleyiciler bu görevi yerine getirirler. Derleyiciler olmasaydı makina dilinde program yazmanın zahmetine çok az kişi katlanabilirdi.

2.7 DOS VE WINDOWS ALTINDA ÇALIŞAN C DERLEYİCİLERİ

Kişisel bilgisayarlarda kullandığımız *C* derleyicilerini yazan iki önemli şirket var: *Microsoft* ve *Borland*. Derleyici pazarındaki payları hemen hemen eşit olan bu iki şirket birbirleriyle tam bir rekabet içindeler.

DOS altında çalışan *C* derleyicilerinin iki uyarlaması vardır:

1) Tümleşik Çevreli Uyarlaması (Integrated environment version): Bu uyarlamalarda editör, derleyici ve bağlayıcı aynı çevrenin içerisindedir. Programlar editörde yazıldıktan sonra *DOS* ortamına dönmeden derlenebilir (compile), bağlanabilir (link) ve çalıştırılabilirler.

2) Komut Satırlı Uyarlaması (Command line version): Bu uyarlamalarda editör, derleyici ve bağlayıcı *DOS* imlecinden (prompt) komut satırı parametreleri alarak ayrı ayrı çalıştırılırlar. Kullanımları oldukça zahmetli olmasına karşın, derleyicilerin komut satırlı uyarlamaları ayrık programlardan oluştuğu için daha az belleğe gereksinim duyarlar. Derleyicilerin komut satırlı uyarlamaları DOS ve UNIX benzeri işletim sistemlerinde bulunur; Windows gibi görsel tarafı ağır basan işletim sistemlerinde böyle bir çalışma biçimi yoktur.

2.8 MİKROİŞLEMCİLERİN TARİHİ GELİŞİMİ VE 80X86 AİLESİ

Dünyanın ilk mikroişlemcisinin *1974* yılında *Intel* firması tarafından tasarlanan ve üretilen *8080* olduğu söylenebilir. Her ne kadar *Intel* daha önce *4040* ve *8008* gibi işlemciler tasarladıysa da bunlar bir mikroişlemcide olması gereken bütün fonksiyonlara sahip değillerdi. Bu nedenle *4040* ve *8008* yalnız elektronik hesap makinalarında kullanılmıştır. Intel'in mikroişlemci pazarındaki tekeli ise uzun sürmedi. Hemen ertesi yıl *Motorola* firması *6800*, *Zilog* firması da *Z80* mikroişlemcileri ile pazardaki yerlerini aldılar. *Intel*'den ayrılan iki mühendis tarafından tasarlanan *Z80* hem *8080* uyumlu hem de ondan iki kat daha hızlı çalışıyordu. Bunun üzerine *Intel* atağa geçerek *8080* mikroişlemcisini optimize etmiş ve *8085*'i tasarlamıştır.

8080, *6502*, *Z80* ve *8085* işlemcilerinin hepsi 8 bit mikroişlemcilerdi. 64K belleği adresleyebilen bu işlemciler o günlerde ihtiyacı büyük ölçüde karşılıyordu. Ancak *Intel* ileride 64K belleğin yetmeyeceğini diğerlerinden önce görmüş olmalı ki girişimi elde bulundurmak amacıyla -ciddi bir ihtiyaç duyulmamasına rağmen- 1978 yılında *8086*'yı piyasaya sürdü. *8086* mikroişlemcisi 1MB (1024KB) belleği adresleyebilen 16 bitlik bir mikroişlemcidir.

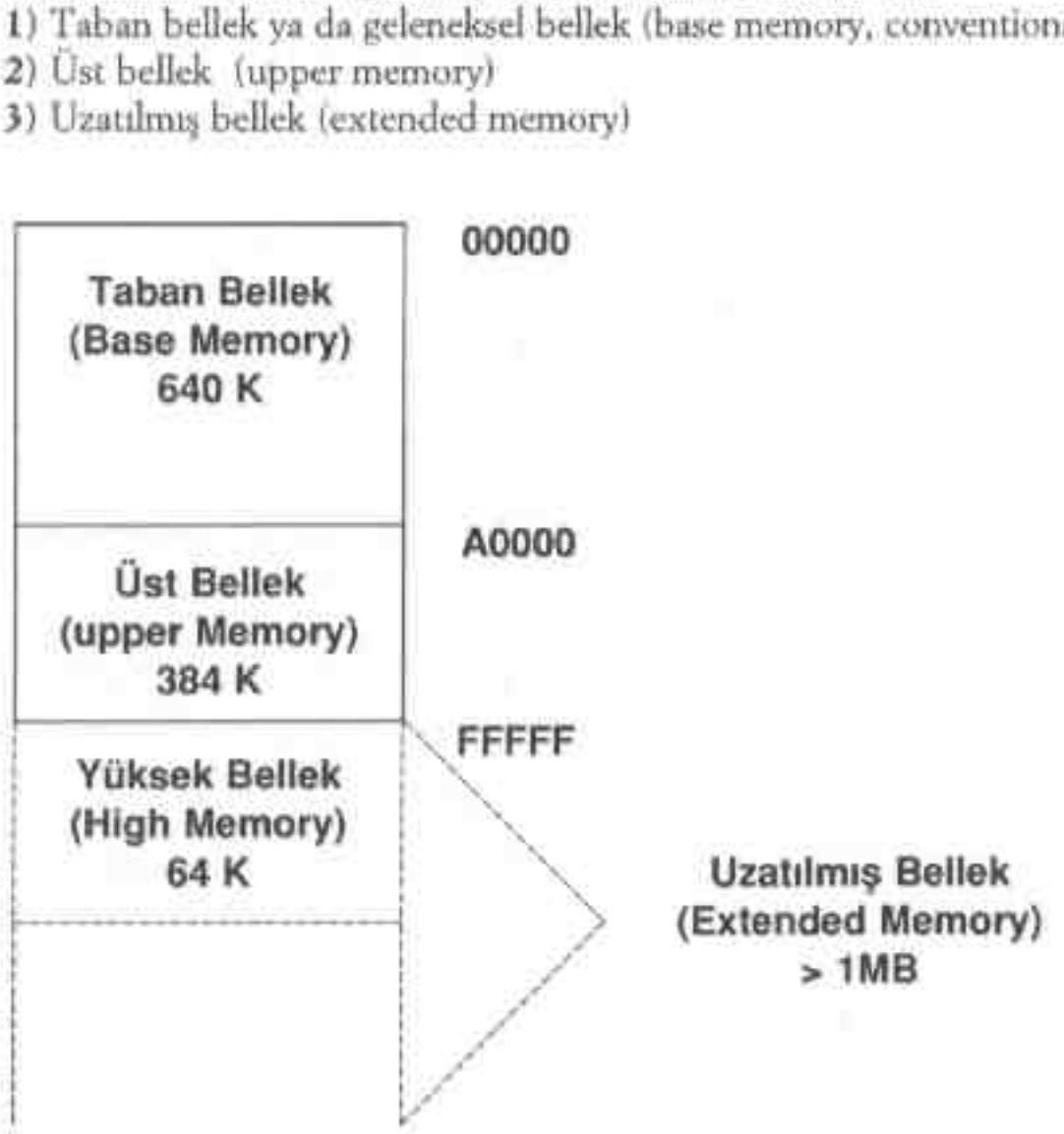
Intel'in *8086* işlemcisinin *1980* yılına kadar ciddi bir anlamda kullanılmadığını söylemek hata olmaz. *1980* yılında *IBM Intel*'in 8086 işlemcisi ile herkesin evinde kullanabileceği bir kişisel bilgisayar yapmak için kolları sıvadı. O zamana kadar 8 bit tabanlı bilgisayarlarda *CPM* denilen nispeten basit bir işletim sistemi kullanılıyordu. *IBM* yeni kişisel bilgisayarı için tamamen yeni bir işletim sistemi arayışına girişti. İşte bu sırada öyküye *Microsoft* ve *DOS* girmektedir. Böylece ilk kişisel bilgisayar 1981 yılında *Intel* firmasının *8086* mikroişlemcisi, *Microsoft* firmasının ise *DOS* işletim sistemiyle piyasaya sürülmüş oldu. Bundan sonra *Intel 8086* işlemcisinin üst modellerini çıkarttıkça, *IBM* bunları kullanarak kişisel bilgisayarları geliştirmiş ve hızlandırmıştır. *80X86* ailesinin gelişimi aşağıdaki tablo ile kronolojik sırada açıklanmıştır.

Yıl	Mikroişlemci	Fiziksel Bellek Alanı	Yazmaç Uzunluğu (bit)
1974	8080	64K	8
1977	8085	64K	8
1978	8086	1MB	16
1979	8088	1MB	16
1980	8096	8K (içsel)	16
1981	80186	1MB	16
1981	80188	1MB	16
1982	80286	16MB	16
1985	80386 DX	4GB	32
1987	80386 SL	32MB	32
1988	80386 SX	16MB	32
1989	80486 DX	4GB	32
1991	80486 SX	4GB	32
1994	Pentium	4GB	32

2.8.1 80X86 Ailesinin Bellek Organizasyonu

80X86 ailesinin bellek organizasyonu 3 bölüme ayrılmaktadır.

1) Taban bellek ya da geleneksel bellek (base memory, conventional memory)
2) Üst bellek (upper memory)
3) Uzatılmış bellek (extended memory)



Taban bellek *DOS* işletim sisteminin programları yüklemek ve çalıştırmak için kullandığı bellektir. Taban bellek 640K uzunluğundadır. Üst bellek bölgesi çeşitli donanım birimleri tarafından kullanılan ancak *DOS* tarafından kullanılmayan bellek alanıdır. 1 MB alanın üzerindeki bölge ise, *"uzatılmış bellek (extended memory)"* olarak isimlendirilir. Uzatılmış belleğin söz konusu olması için mikroişlemcinin en azından 80286 olması gerekir. Çünkü *8086* ve *8088* mikroişlemcilerinin bellek alanı zaten 1 MB'den daha büyük olamaz. Uzatılmış bellek *DOS*'ta doğrudan kullanılamaz; erişme yöntemi çeşitli kurallarla belirlenmiştir (**E**xtended **M**emory **S**pecification). Uzatılmış belleğin ilk 64K'lık kısmına yüksek bellek bölgesi denilmektedir. *80286*, *80386*, *80486* ve *Pentium* işlemcileri normal olarak gerçek modda bu bölgeyi adresleyebilirler.

Bir de *"yayılmış bellek (expanded memory)"* kavramından bahsetmenin yerinde olacağını düşünüyoruz. 640K belleğin *DOS* için yetersiz kaldığını gören *Intel*, *Microsoft ve Lotus* firmaları 1985 yılında biraraya gelerek, daha fazla belleğin kullanılmasına izin veren bir kart tasarlamış ve bu kartın nasıl kullanılacağına ilişkin kurallar belirlemişlerdir (**E**xpanded **M**emory **S**pecification). Fakat uzatılmış belleğin ortaya çıkmasıyla birlikte yayılmış bellek kavramının da geri plana itildiği söylenebilir. Ancak, uzatılmış belleğin yayılmış bellek gibi kullanılması da mümkündür.

DOS Programcılarına Not: *DOS* işletim sisteminin *5.0* uyarlamasıyla birlikte üst bellek bölgesinde donanım birimleri tarafından kullanılmayan boş bölgelere küçük programların yerleştirilmesi mümkün hale getirilmiştir. Bunun için: HIMEM.SYS ve EMM386.EXE kurulduktan sonra DOS = UMB yapılmalıdır. Ayrıca, taban belleğin büyütülmesi amacıyla *DOS*'un büyük bölümü yüksek belleğe yüklenebilir. Bunun için de *HIMEM.SYS* kurulduktan sonra DOS = HIGH yapılması gerekir. Uzatılmış belleğin yayılmış bellek olarak kullanılabilmesi için de benzer biçimde: HIMEM.SYS ve EMM386.EXE kurulmuş olmalıdır. EMM386.EXE programının /NOEMS parametresi, yayılmış bellek emülasyonunun yapılmayacağını belirlemek için kullanılır.

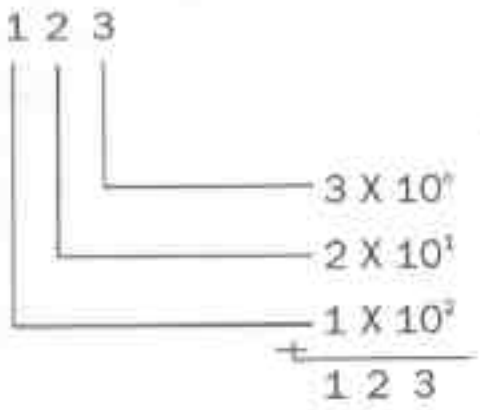
2.9 ALGORİTMANIN KARMAŞIKLIĞI

Problemlere çözüm getiren birden fazla algoritmik yöntem olabilir. Bunlardan hangisinin daha iyi olduğunun belirlenmesi için birtakım ölçütlerin tanımlanması gerekir. Örneğin, sayıların sıraya dizilmesinde (sorting) birden fazla yöntem söz konusudur. Acaba bunların hangisi diğerlerinden daha iyidir?

Öncelikle algoritmaların hız, bellek gereksinimi, açıklık gibi çok çeşitli ölçütlerle birbirleriyle kıyaslanabileceğini belirtelim. Biz burada yalnızca hız üstüne etkili olan ve adına algoritmanın karmaşıklığı denilen bir ölçütten bahsedeceğiz. Algoritmanın karmaşıklığı (complexity of algorithm), sonucun elde edilebilmesi için **en kötü olasılıkla ihtiyaç duyulan karşılaştırma sayısıdır.** Örneğin **n** tane sayı arasından en büyüğünü bulan bir algoritmayı gözönüne alalım. Tüm sayılar gözden geçirileceği için bu örnekte algoritmanın karmaşıklığı **n**'dir.

2.10 SAYI SİSTEMLERİ

Günlük yaşantımızda hepimiz 10'luk sayı sistemini kullanıyoruz. Örneğin, 123 gibi onluk sistemde yazılmış bir sayı bize kolaylıkla algılayabildiğimiz bir nicelik anlatıyor. 123 sayısını matematiksel olarak şöyle çözümleyebiliriz:

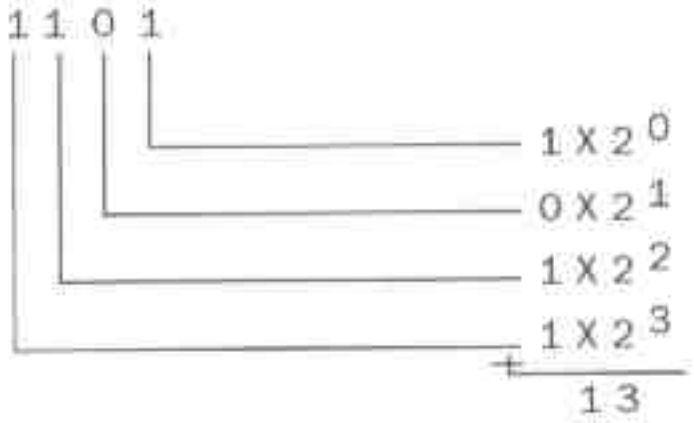


10'luk sistemde toplam 10 tane simge vardır, bunlar: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9.**

O halde 2'lik sistemde 2 tane simge olması gerekir: **0** ve **1**.

10'luk sistemdeki bütün sayıların 2'lik sistemde birer karşılıkları vardır. Burada 10'luk sistemden 2'lik sisteme ya da 2'lik sistemden 10'luk sisteme çevirme işleminin nasıl yapılacağı üzerinde durmayacağız. Eğer bunu bilmiyorsanız, başka bir kaynağa başvurmalısınız.

Örneğin 10'luk sistemde yazılmış **13** sayısının ikilik sistemdeki karşılığı **1101** dir.



2.11 BİT VE BYTE KAVRAMLARI

Bilgisayar sistemlerinde bütün bilgiler ikilik sistemde 1 ve 0 biçiminde ifade edilebilen elektriksel işaretlerle saklanır.

İkilik sistemdeki herbir basamağa bit denir.

Bu tanıma göre, ikilik sistemde yazılmış olan **10100001** sayısı **8** bittir. Fakat **bit** nicelik ifade edebilmek için uygun bir birim değildir. Temel bellek birimi olarak **byte** kullanılır.

Bir byte 8 bittir.

Yarı iletken teknolojisindeki gelişmelerle birlikte büyük bir bellek enflasyonu yaşanmıştır. Bellekler de artık byte'larla değil Kilo byte'larla, Megabyte'larla hatta Gigabyte'larla ölçülür hale gelmiştir.

- **1 Kilo byte (1 K) : 1024 byte.**
- **1 Mega byte (1 M) : 1024 Kilo byte.**
- **1 Giga byte (1 G) : 1024 Mega byte.**

2.12 POZİTİF VE NEGATİF SAYILAR

Önce şu soruya yanıt verelim. Bir byte (yani 8 bit) alan içerisinde yazabileceğimiz en küçük ve en büyük sayı nedir?

0000 0000 (0)	En küçük sayı
1111 1111 (255)	En büyük sayı

Biz bu hesaplamada sayıların hep pozitif olduğunu kabul etik. Peki ya negatif sayılar?..

İkilik sistemde negatif sayıları nasıl gösterebiliriz? Bunun için mikroişlemcilerin evrimsel gelişiminde birkaç yöntem kullanılmıştır. Biz burada *80X86* tabanlı mikroişlemcilerin (aslında modern mikroişlemcilerin hemen hepsinin) kullandığı 2'ye tümleme yönteminden bahsedeceğiz.

Fakat önce 1'e tümleme ve 2'ye tümleme kavramlarını açıklayalım.

1'e Tümleme: İkilik sistemdeki sayılarda, 1'lerin 0; 0'ların da 1 yapılmasıyla elde edilir.

Örneğin:

0101 1101 sayısının 1' e tümleyeni:

1010 0010 sayısıdır.

Burada hemen şuna dikkat ediniz: Bir sayının 1'e tümleyeninin 1'e tümleyeni sayının kendisine eşittir.

2'ye Tümleme: Sayının 1'e tümleyenine 1 ekleyerek elde edilir.

Örneğin:

1011 0001 sayısının 2'ye tümleyenini bulalım. Önce 1'e tümleyeni hesaplanmalıdır:

```
0100 1110      (1'e tümleyeni).
0000 0001      (eklenecek 1 sayısı)
+
0100 1111      Sayısını elde ederiz.
```

Bir uygulama daha:

1010 0100 sayısının ikiye tümleyenini bulmak için sayının 1'e tümleyenine 1 ekliyoruz:

```
0101 1011
        1
+
0101 1100
```

Verilen bir sayının 2'ye tümleyenini bulmanın kolay bir yolu da var:

Sayıda, sağdan sola doğru ilk 1 görülene kadar (ilk 1 dahil) aynısı yazılarak ilerlenir. Daha sonra kalanın 1'e tümleyeni yazılarak devam edilir.

Örneğin:

1010 0100 sayısının 2'ye tümleyenini bir hamlede bulmak istiyoruz. İlk 1 görene kadar sağdan sola ilerlersek 100 sayısını yazarız. Daha sonra 1'e tümleyeni ile devam edersek,

0101 1100 sayısını buluruz.

```
1010 0100      sayı
0101 1100      ikiye tümleyeni
```

Yine 1'e tümleyende olduğu gibi, 2'ye tümleyenin 2'ye tümleyeni de sayının kendisine eşittir.

Örneğin sayı 0001 0010 olsun.

```
0001 0010      sayı
```

1110 1110 2'ye tümleyeni

0001 0010 2'ye tümleyenin ikiye tümleyeni

İkilik sistemde işaretli sayılar yazmak için bitlerden birisini işaret biti olarak kullanmak zorundayız. Gerçekten de bu sistemde en soldaki bit, sayının işaretini belirlemekte kullanılan işaret bitidir. En soldaki bu bit (en yüksek anlamlı bit) 0 ise sayı pozitif, 1 ise sayı negatiftir.

Örneğin:

1001 0011 sayısı negatiftir.

0100 1010 sayısı ise pozitiftir.

Negatif ve pozitif sayılar birbirlerinin 2'ye tümleyenleridirler.

Örneğin:

0000 1010 (+10)
1111 0110 (-10)

0001 1111 (+31)
1110 0001 (-31)

Acaba 1 byte içerisinde yazılabilecek en büyük pozitif ve en küçük negatif sayılar nelerdir? En büyük pozitif sayıyı bulmak oldukça kolay:

0111 1111 (+127) En soldaki bit işaret bitidir.

Peki ya en küçük negatif sayıyı nasıl bulabiliriz? Öncelikle bu sayının 2'ye tümleyenini alarak -127 sayısını bulalım:

1000 0001 (-127)

Bu sayıdan 1 çıkartırsak sonuç:

1000 0000 (-128)

olur. Sayı hala negatiftir ve yazılabilecek en küçük sayıdır. Demek ki, 1 byte içerisinde yazılabilecek en büyük ve en küçük sayılar [-128, +127] dir. Şimdi şunu sorabilirsiniz: Neden negatif sayılar pozitif sayılardan 1 fazla?

- Yanıt basit. Toplam 256 kombinasyon vardır: (**0000 0000 , 1111 1111**) Bunlardan bir tanesi 0 olacağına göre (**0000 0000**), geriye **255** kombinasyon kalır; bu durum, ya pozitif ya da negatif sayıların diğerinden 1 fazla olacağı anlamına gelir.

Sayıların doğrudan negatiflerini yazmak ve yorumlamak güç olduğu için onların pozitiflerinden faydalanırız. Aşağıda negatif sayıların daha iyi anlaşılması için sık rastlanılan üç durum ele alınmıştır.

2.12. Negat Sayıla Değer Bulun

Sayı negatif olduğuna göre işaret biti 1' dir. İkiye tümleyenini alarak sayıyı pozitif hale getirdikten sonra değerini kolaylıkla hesaplayabilirsiniz.

Örneğin:

1111 0110 sayısı kaçtır?

İşaret biti (en soldaki bit) **1** olduğuna göre sayı negatiftir. O halde 2'ye tümleyenini alırsak, **0000 1010** sayısını elde ederiz. Bu sayı **+10** olduğuna göre, bu durumda ilk sayı da **-10**'dur.

2.12.2 Negatif Bir Sayının Yazılması

Bu durumda önce sayının pozitifini yazarız; daha sonra ikiye tümleyenini alarak negatif hale getiririz.

Örneğin, **-22** yazmak isteyelim. Önce **+22** yazarız:

0001 0110 = +22

Bunun ikiye tümleyeni **-22**'dir.

1110 1010 = -22

2.12.3 İşaretli Sayılarda Taşmalar

Pozitif sayılar sınır değerlerini aşarlarsa negatif, negatif sayılar sınır değerlerini aşarlarsa pozitif bölgeye geçerler. Çünkü bu durumlarda sayıların işaret bitleri değişir.

En büyük pozitif sayıya 1 toplayalım.

```
0111 1111     (+127)  En büyük pozitif sayı
        1
      +
-------------
1000 0000     (-128)  En küçük negatif sayı
```

Görüldüğü gibi en büyük pozitif sayıya 1 toplayınca en küçük negatif sayı elde ediliyor. Şimdi de en küçük negatif sayıdan 1 çıkartalım.

```
1000 0000     (-128)  En küçük negatif sayı
        1     (1)
      +
-------------
0111 1111             En büyük pozitif sayı
```

En küçük negatif sayıdan 1 çıkarttığımızda da en büyük pozitif sayıyı buluyoruz. Daha fazla açıklama yapmayıp, bunun nedenleri üzerine düşünmeyi size bırakalım...

Bilgisayar için anlamlı iki sayı sistemi daha vardır. Bunlar 16'lık ve 8'lik sayı sistemlerdir. Önce 16'lık sistemi inceleyeceğiz.

2.13 16'LIK (Hexadecimal) VE 8'LİK (Octal) SAYI SİSTEMLERİ

10'luk sistemde 10 simge (**0, 1, 2, 3, 4, 5, 6, 7, 8, 9**), 2'lik sistemde 2 simge (**0, 1**) vardır. O halde, 16'lık sistemde de 16 simge olmalıdır.

16'lık sistemde kullanılan simgeler şunlardır:

16'lık Sistemdeki Simgeler	10'luk Sistemdeki Sayısal Karşılıkları
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7

8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

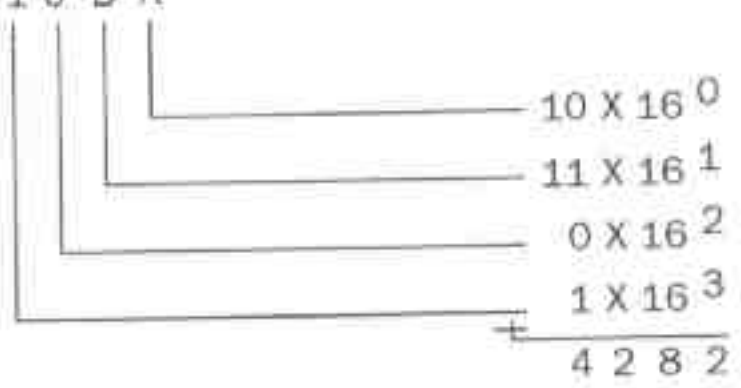
Gördüğünüz gibi, ilk 10 değişken 10'luk sistemde kullandığımız rakamlardır. Bundan sonra harfler gelir:

A ⇒ 10
B ⇒ 11
C ⇒ 12
D ⇒ 13
E ⇒ 14
F ⇒ 15

16'lık sistemden 10'luk sisteme dönüşüm benzer biçimde yapılabilir.

Örneğin:

10BA sayısını 10'luk sisteme dönüştürelim.

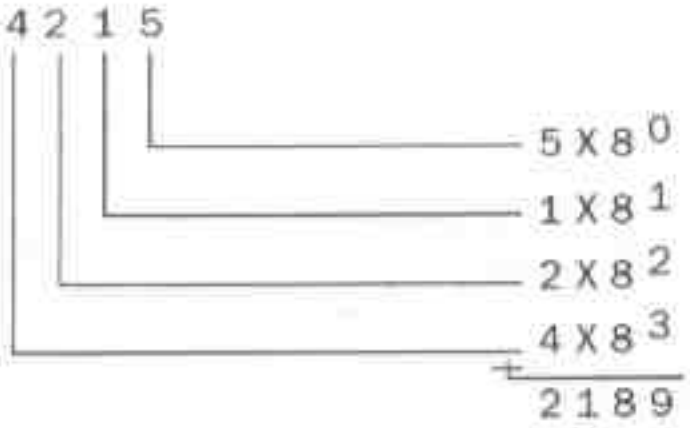


16'lık sistemde yazılan sayıların yanında **H** harfi görürseniz şaşırmayın! Bu harf, *İngilizce* 16'lık sistem anlamına gelen **Hexadecimal** sözcüğünün başharfidir ve 10'luk sistemle herhangi bir karışıklığa neden olmamak için sayının sağına yazılmaktadır.

8'lik sistemde (octal) ise toplam 8 tane simge vardır. Bunlar:

0
1
2
3
4
5
6
7

8'lik sistemde yazılmış olan **4215** gibi bir sayı 10'luk sisteme dönüştürülmek istenirse benzer işlemler yapılır:



10'luk sistem bizim kullandığımız sistemdir. 2'lik sistemin ise bilgisayar tarafından kullanıldığını söyledik. Peki ya 16'lık ve 8'lik sistemleri kimler, niçin kullanırlar?

2.14 16'LIK VE 8'LİK SİSTEMLERİN KULLANILMA NEDENLERİ

İkilik sistemde sayıların yazılması ve yorumlanması oldukça zordur. Üstelik ikilik sistemdeki sayılar çok yer kaplarlar. 10'luk sistem ile 2'lik sistem arasındaki dönüştürme işleminin zorluğu da ortadadır. İşte 16'lık ve 8'lik sistemler, 2'lik sistemin yoğun bir gösterimi olarak kullanılırlar. Çünkü 16'lık ve 8'lik sistemler ile 2'lik sistem arasında dönüşüm yapmak oldukça kolaydır.

2'lik Sistem	16'lık Sistem
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

2'lik Sistem	8'lik Sistem
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Yukarıda gördüğünüz gibi 16'lık sistemin her bir basamağı 2'lik sistemde 4 bit ile, 8'lik sistemin her bir basamağı ise 3 bit ile ifade edilebiliyor. 2'lik sistemdeki sayıları 16'lık ve 8'lik sistemlerde kodlayarak onları daha yoğun bir biçimde ifade edebiliriz. Örneğin:

0101 1010 bir byte uzunluğunda bir sayıdır. Bu sayıyı sağdan sola dörder dörder ayırarak 16'lık sistemde ifade edebiliriz.

0101 = 5
1010 = A

Bu durumda sayı **5AH** olur. **0101 1010 = 5AH** Birkaç örnek daha verelim:

```
0011 0110 1110 0100   =      36E4H
0100 1100 0101 1101   =      4C5DH
0100 0101 0100 0000   =      4540H
```

16'lık sistemde yazılmış bir sayının 2'lik sisteme dönüştürülmesi üzerine de birkaç örnek vermek istiyoruz.

```
3F0BH = 0011 1111 0000 1011
          3    F    0    B

5A8CH = 0101 1010 1000 1100
          5    A    8    C
```

Aynı biçimde, 8'lik sistem için de örnekler verebiliriz.

8'lik Sistem	2'lik Sistem
456	100 101 110
123	001 010 011
756	111 101 110

SORAMADIKLARINIZ...

S1) Çokişlemli (multiprocessing) işletim sistemlerinde birden fazla program aynı anda nasıl çalışıyor? Bir tek mikroişlemci bulunan sistemlerde böyle birşey mümkün olabilir mi?

C1) Birden fazla programın aynı anda çalışması demekle aynı anda çalışıyor olarak gözükmesi anlatılmaktadır. Yoksa, tek bir mikroişlemci varken birden fazla program gerçekten aynı anda çalışamaz. Çokişlemli işletim sistemleri bu durumda programları **zaman paylaşımlı (time sharing)** olarak , biraz ondan biraz bundan biçiminde çalıştırırlar. Ancak programların sahibi olan kullanıcılar hepsinin aynı anda çalıştığını sanabilirler.

S2) Kilo 1000 anlamına geldiğine göre, neden 1 Kilobyte 1000 byte değil de 1024 byte'tır?

C2) Bilgisayar sistemleri için bellek miktarlarının ikinin katları olması anlamlıdır. Bu yüzden **n** kilobyte, **n * 1024** byte'tır.

$$1K = 2^{10} = 1024 \text{ byte}$$
$$2K = 2^{11} = 2048 \text{ byte}$$

$$4K = 2^{12} = 4096 \text{ byte}$$
$$\vdots$$
$$64K = 2^{16} = 65536 \text{ byte}$$
$$1024K = 2^{20} = 1048576 \text{ byte}$$

Aynı tanım mega byte'lar için de verilebilir:

$$1MB = 2^{10}K = 1024K$$
$$2MB = 2^{11}K = 2048K$$
$$4MB = 2^{12}K = 4096K$$
...

S3) *Borland* derleyicileri mi daha iyidir, *Microsoft* derleyicileri mi?

C3) Bu iki derleyici grubu da birbirlerinden mutlak anlamda üstün değildir. Farklı özellikleri ve avantajları vardır. Örneğin, *Borland* derleyicileri derleme süresinin kısa olmasıyla ün yapmışlardır. *Microsoft* derleyicileri ise daha ayrıntılı ve profesyonel özelliklere sahiptir. *Borland* derleyicilerine uyum sağlamanın daha kolay olduğunu söyleyebiliriz.

S4) Negatif sayılarda ikiye tümleme yönteminin kullanılması mikroişlemciye mi bağlıdır, yoksa programcının bir varsayımı mıdır?

C4) Negatif sayılarda ikiye tümleme yönteminin kullanılması mikroişlemcinin tasarımına bağlıdır. Çünkü, işaretli sayılar üzerinde işlem yapan makina komutları vardır.

S5) *DOS*'ta derleyicilerin çıktısı neden doğrudan çalışabilir *.EXE* kodu değil? Bağlayıcı programlara ayrıca neden ihtiyaç duyuluyor?

C5) Bağlayıcı programların işlevleri konular ilerledikçe anlaşılacaktır. Ancak yine de şimdiden birkaç şey söyleyelim: Bağlayıcı programların (linkers) en önemli işlevleri, ayrı ayrı derlenmiş olan modüllerin birleştirilmesinde ortaya çıkmaktadır. *DOS*'ta farklı *.OBJ* dosyalar bağlama (link) aşamasında birleştirilerek tek bir .EXE oluşturabilir. Böylece büyük projelerin parça parça yazılmaları ve derlenmeleri mümkün kılınmıştır.

S6) **DOS** neden 640K bellek alanını kullanabiliyor. 640K sayısı nasıl bulunmuştur?

C6) Şöyle de düşünebiliriz: *"640K bellek DOS tarafından kullanıldığına göre, geri kalan 384K başka birimler tarafından kullanılmaktadır"*. Gerçekten de video kartı, EPROM gibi birimler üst bellek bölgesini kullanırlar. Ancak, yine de 640K değeri iyi hesaplanmış bir değer değildir. Çünkü, üst bellek bölgesi içerisinde hiç kullanılmayan birtakım boş bölgeler vardır. Bu durumda 640K bir zorunluluk değildir. Yani *DOS* pekala bir 64K daha büyük bellek alanı kullanabilirdi...

PROGRAMLAMANIN TEMEL KAVRAMLARI VE C'YE GİRİŞ

3

Bu bölümde hem ileride kullanacak olduğumuz çeşitli terimlerin ve kavramların açıklamasını bulacaksınız hem de *C*'ye bir giriş yapacaksınız. *C*'yi bu terimleri ve kavramları kullanarak öğrenmek çok daha kolay olacak. Açıklayacağımız bu kavramları daha önce hiç duymamış da olabilirsiniz. Bu nedenle size biraz karmaşık -ve belki de biraz gereksiz- gelebilir. Fakat ne olursa olsun sonuna kadar sabırla okumanızı salık veririz.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz ve kavramları açıklayabilmeniz gerekir:

Sorular:
1) Atom nedir ve kaç gruba ayrılır?
2) Boşluk karakterleri hangileridir?
3) *C*'deki yazım kuralı nasıldır?

Kavramlar:
1) Blok
2) Fonksiyon
3) Fonksiyon tanımlama ve fonksiyon çağırma
4) Nesne
5) İfade
6) Sol taraf değeri
7) Sağ taraf değeri

3.1 PROGRAMLAMADA KULLANDIĞIMIZ ÖZEL KARAKTERLER

Klavyede bulunan özel karakterlerin bazıları birbirleriyle karıştırılmaktadır. Olası bir karışıklığı gidermek için bütün bu özel karakterlerin *Türkçe* ve *İngilizce*'deki isimlerini liste halinde vermek istiyoruz:

Karakter	Türkçe	İngilizce
"	İki tırnak	Double quote
'	Tek tırnak	Single quote

!	Ünlem işareti	Exclamation mark
^	Şapka işareti	Caret
#	Diyez işareti	Number sign
+	Artı	Plus
-	Tire (eksi)	Hyphen (minus sign)
$	Dolar işareti	Dollar sign
%	Yüzde	Percent
&	Ve işareti	Ampersand (and)
{ }	Küme işareti	Brace
( )	Parantez	Parenthesis
[ ]	Köşeli parantez	Square bracket
=	Eşit işareti	Equal sign
?	Soru işareti	Question mark
*	Yıldız	Asterisk
,	Virgül	Comma
;	Noktalı virgül	Semicolon
:	İki nokta üstüste	Colon
.	Nokta	Period
@	Salyangoz	At sign
~	Tırtıl	Tilde
>	Büyüktür işareti	Greater than sign
<	Küçüktür işareti	Less than sign
<...>	Açısal parantez	Angular bracket
/	Bölü işareti	Slash
\	Ters bölü işareti	Back slash
|	Çubuk	Pipe

3.2 ATOM KAVRAMI (Token)

Bir programlama dili için anlam taşıyan en küçük birime atom (token) denir. Atomlar programlama dillerinin daha fazla anlamlı parçaya bölünemeyen en yalın elemanlarıdır. Yazdığımız kaynak kod derleyici tarafından derleme işleminin ilk aşamasında çözümleme amacıyla atomlarına ayrılmaktadır.

Aşağıda aynı işlevlere sahip olan *BASIC* ve *C* programları örnek olarak verilmiştir. Her iki programda da klavyeden bir sayı alınıyor; 1'den başlayarak klavyeden girilen sayıya kadar olan tüm tamsayılar ekrana yazdırılıyor, inceleyiniz:

BASIC

```
10   PRINT    "LÜTFEN BİR SAYI GİRİNİZ"
20   INPUT  N
30   FOR  K = 1 TO N
40   PRINT  K
50   NEXT  K
60   END
```

C

```
#include <stdio.h>

main()
{
     int n, k;

     printf("LÜTFEN BİR SAYI GİRİNİZ\n");
     scanf("%d", &n);
     for (k =  1; k <= n; ++k)
          printf("%d\n", k);
}
```

Örnek BASIC programındaki atomlar şunlardır.

```
10
PRINT
"LÜTFEN BİR SAYI GİRİNİZ"
20
INPUT
N
30
FOR
K
=
1
TO
N
40
PRINT
K
50
NEXT
K
60
END
```

Örnek C Programındaki atomlar ise aşağıdakilerden oluşmaktadır:

```
#       include     <       stdio.h  >     main    (       )       {
int     n           ,       k        ;     printf  (
"LÜTFEN BİR SAYI GİRİNİZ\n"     )        ;     scanf   (       "%d"    ,
&       n           )       ;        for   (       k       =       1
;       k           <=      n        ;     ++      k       )
printf  (           "%d\n"  ,        k     )       ;       }
```

Bir kaynak kodu atomlarına ayırırken, o kaynak kodu yazdığımız programlama dili hakkında da bilgi sahibi olmak zorundayız. İşlevleri aynı olmasına karşın örnek olarak verdiğimiz *BASIC* ve *C* programlarının atomları birbirlerinden farklıdır.

Atomları da kendi aralarında gruplara ayırabiliriz.

1) Anahtar sözcükler (keywords, reserved words)

Bunlar dil için belli bir anlam taşıyan, değişken olarak kullanılması yasaklanmış olan sözcüklerdir.

Örnek BASIC programında:

```
PRINT
INPUT
FOR
TO
NEXT
END
```

Örnek C programında:

```
include
int
for
```

atomları bu diller için birer anahtar sözcüktür. Dilin semantik yapısının belirlenmesinde anahtar sözcükler en önemli rolü oynarlar. Bildiğiniz gibi *BASIC*'te anahtar sözcüklerin büyük ya da küçük harfle yazılması farketmemektedir. Oysa:

C' de bütün anahtar sözcükler küçük harflerden oluşur.

2) Değişkenler (Identifiers, Variables)

Değişkenler, önceden belirlenmiş belirli kurallara uymak suretiyle ismini istediğimiz gibi verebildiğimiz atomlardır. Bu atomlar genellikle bellekte bir yer belirtirler.

Örnek BASIC programında:

```
K
N
```

Örnek C programında:

```
main
n
k
printf
```

atomları birer değişkendir.

3) Operatörler (Operators)

Operatörler önceden tanımlanmış birtakım işlemleri yapan atomlardır. +, - , *, / , <, >, birer operatördür.

Örnek BASIC programındaki tek operatör:

=

atama operatörüdür. Atama operatörü sağ taraftaki değerin sol taraftaki değişken ile belirtilen bellek bölgesine yazılmasını sağlayan bir operatördür.

Örnek C programındaki operatörler:

(...)	fonksiyon çağırma operatörü
&	gösterici operatörü
=	atama operatörü
<=	ilişkisel operatör
++	artırma operatörü

4) Sabitler (Constants)

Doğrudan işleme sokulan, değişken bilgi içermeyen atomlardır.

`C = A + B` ifadesinde `A` ve `B` değişkenlerinin içlerindeki sayılar toplanarak C'ye atanmaktadır. Oysa,

`C = A + 10` ifadesinde `10` sabiti doğrudan `A` ile toplanmaktadır.

Örnek BASIC ve C Programında bir tek sabit atom vardır:

1

5) Stringler (String literal)

Bunlar, iki tırnak içerisindeki karakterlerden oluşan atomlardır . Stringler programlama dillerinin çoğunda tek bir atom olarak ele alınırlar; daha fazla parçaya bölünemezler.

Örnek BASIC programında:

"LÜTFEN BİR SAYI GİRİNİZ" tek bir atomdur.

Örnek C programında:

`"LÜTFEN BİR SAYI GİRİNİZ"`
`"%d"`
`"%d\n"` ifadeleri birer string atomdur.

6) Ayıraçlar ya da Noktalama İşaretleri (Separators, Punctuators)

Yukarıdaki atom sınıflarının dışında kalan ayraç ve sonlandırıcı olarak kullanılan atomlardır.

Örnek BASIC programında bu anlamdaki atomlar satırları birbirinden ayırmak için kullanılan sayılardır:

```
10
20
30
40
50
```

Örnek C programındaki ayıraçlar da aşağıda listelenmiştir:

```
{
;
,
)
```

3.3 NESNE (Object)

Bellekte yer kaplayan ve içeriklerine erişilebilen alanlara nesne denir. Bir ifadenin nesne olabilmesi için bellekte yer belirtmesi gerekir. Programlama dillerinde nesnelere isimlerini kullanarak erişiriz.

Aşağıdaki program parçasını inceleyiniz. (Bu kodların hangi dile ait olduğunun bizim için şu aşamada bir önemi yok.)

```
a = b + c;
d = 100;
e[1] = 500;
e[2] = 600;
```

ifadelerinde **a, b, c, d, e[1], e[2]** birer nesnedir. Çünkü bunların hepsi bellekte bir yer göstermektedir ve istenildiğinde bu isimler kullanılarak bu bölgelere erişilebilir. **e[1]** ve **e[2]**, **e** dizisinin 1. ve 2 . indis elemanı terimi olarak kullanılmıştır. Bu iki ifadenin de bellekte farklı yerleri belirttiğini biliyorsunuz.

Nesnelerin çeşitli özellikleri vardır. Aşağıdaki şekli inceleyiniz.

Nesne özellikleri üzerinde biraz daha ayrıntılı olarak durmanın faydalı olacağını düşünüyoruz.

İsim (name): Nesneyi temsil eden karakterlerdir. Programlama dillerinde nesnelere isim vermenin belli kuralları vardır.

```
SAYI = 200
```

örneğinde **SAYI** bir nesne ismidir.

Değer (value): Nesnelerin içlerinde tuttukları bilgilerdir. Bu bilgiler duruma göre istenildiği zaman değiştirilebilirler ya da bir kere değer verildikten sonra bir daha değiştirilemezler.

Tür (type): Nesnenin türü, onun işlemlere sokulduğunda derleyici tarafından nasıl yorumlanacağını anlatan bir özelliğidir. Programlama dillerinin çoğunda:

char (karakter)

integer (tamsayı)
float ya da real (gerçek sayı)
...

gibi nesne türleri vardır. Bir nesnenin türü onun bellekteki uzunluğu hakkında da bilgi verir. Her türün bellekte ne kadar uzunlukta bir yer kapladığı programlama dillerinde önceden belirlenmiştir.

Faaliyet alanı ve ömür (scope and duration): Bu kavramların ne anlama geldiğini 8. Bölümde ayrıntılı olarak açıklayacağız.

Not: Burada kullandığımız **nesne (object)** teriminin **Nesne Yönelimli Programlama (Object Oriented Programming)** tekniğindeki nesne terimiyle hiçbir ilişkisi yoktur.

3.4 İFADE (Expression)

Değişken, operatör ve sabitlerin kombinasyonlarına ifade denir.

Örneğin:

```
a + b / c
d * b[i] - 2
c = a + b * 10
...
```

birer ifadedir.

3.5 SOL TARAF DEĞERİ (Left Value)

Nesne gösteren ifadelere denir. Bir ifadenin sol taraf değeri olabilmesi için bellekte bir yer göstermesi gerekir. Zaten, tipik olarak atama operatörünün sol tarafında bulunduklarından bu ifadelere sol taraf değeri denilmiştir.

Örneğin:

a ve **b** nesneleri tek başlarına sol taraf değeridir, fakat **a + b** ifadesi bir sol taraf değeri değildir. Çünkü bu ifade bir nesne göstermez; **a** ve **b** nesnelerinin içerisindeki değerlerin toplamını gösteren bir sayı belirtir. **a + b = c** yazabilir miyiz?

b[i] ifadesi bir sol taraf değeridir. Çünkü, **b** dizisinin i. indisli terimi olan **b[i]** ifadesi bellekte bir yer gösterir. **b[i] = c** yazabiliriz, değil mi?

Sol taraf değerleri şu tür ifadelerden oluşabilir:

- **Değişkenler her zaman sol taraf değeri olarak kullanılabilirler.**
  `a, b, adet, fiyat,...` gibi.

- **Dizi elemanları sol taraf değeridir.**
  `a[i], b[j]...` gibi.

- **Gösterici ifadeleri** (Henüz bundan bir şey anlamamanız gerekiyor).
  `*p, **p...` gibi.

3.6 SAĞ TARAF DEĞERİ (Right Value)

Nesne göstermeyen ifadelerdir. Atama operatörünün sağında bulunabilmesinden dolayı bunlara sağ taraf değeri denilmektedir.

Örneğin:

```
a + b
200
...
```

Sağ taraf değerleri de şu ifadelerden oluşabilir:

- Sabitler her zaman sağ taraf değeridirler.

Örneğin :

100, 200,.. gibi sabitler her zaman atama operatörünün sağ tarafında bulunabilirler. **100 = a** olabilir mi?

- Operatörlerin büyük bir kısmı sağ taraf değeri oluşturur:

`a + b, c / d - 3...` gibi

3.7 C'YE MERHABA

Aşağıdaki *C* programını inceleyiniz:

```
#include <stdio.h>

main()
{
     printf("Merhaba C\n");
}
```

Bu program ekrana

Merhaba C

yazısını basan yalın bir *C* programıdır.

Bu programın ilk satırı olan

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

ifadesi, **stdio.h** isimli dosyanın derleme işlemine dahil edileceğini anlatır. Bu dosyanın neden derleme işlemine dahil edildiğini ve içerisinde nelerin olduğunu şu anda açıklayamıyoruz. Çünkü bu dosyanın içerisindeki ifadeleri yorumlayabilmek için çok çeşitli bilgilere sahip olmak gerekir. Uzantısı **.h** olan bu tür dosyaların içeriklerini öğrenme süreci içinde yavaş yavaş çözümleyeceğiz. Bu konuda acele etmeyin!..

İncelememize devam edelim:

```
main ()
{
     printf("Merhaba C\n");
}
```

`main` bir fonksiyondur. *C*'de altprogramlara fonksiyon denir. *C*'deki fonksiyonlar, *FORTRAN*'ın subroutine' leri, *PASCAL* ve *CLIPPER*'ın procedure'leri gibi altprogramlardır. **main**'in bir fonksiyon olduğunu onu takip eden fonksiyon operatöründen anlıyoruz.

(...) **Fonksiyon operatörüdür . Bu operatörün solundaki ifadeler C derleyicileri tarafından birer fonksiyon olarak yorumlanırlar.**

O halde `printf` de bir fonksiyondur. (`printf` sözcüğünü `print-ef` biçiminde okuyunuz. Buradaki **f**, *İngilizce* function sözcüğünden değil **format** sözcüğünden gelmektedir).

```
printf("Merhaba C\n");
```

`main` ile `printf` fonksiyonlarını birbirleriyle karşılaştırın. Ne görüyorsunuz?

```
#include <stdio.h>
main()
{
     printf("Merhaba C");
}
```

İlk görebildiğiniz sanıyorum, fonksiyon operatörü diye adlandırdığımız parantezlerden **main**'e ait olanının içerisinde hiçbir şey olmadığı halde, **printf**'inkinde iki tırnak içerisinde bir yazının olduğudur. String ifadelerinin tek bir atom olduğunu anımsayınız.

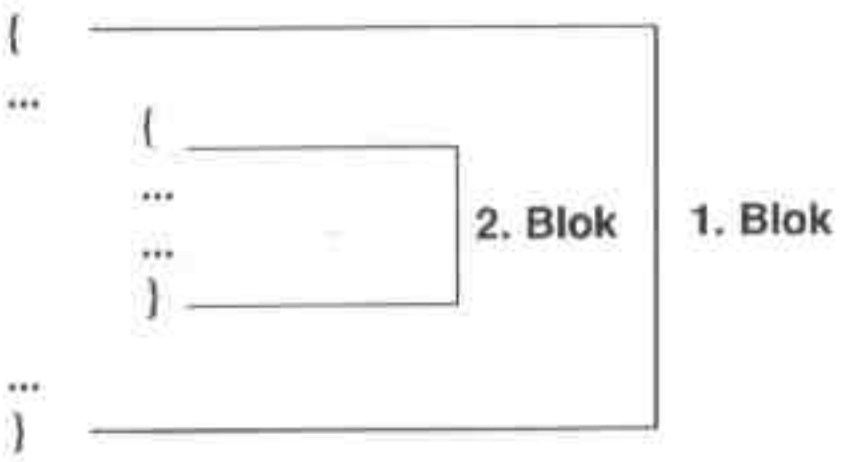
Fonksiyon operatörlerinin içindeki ifadelere parametre denir.

Buradan örneğimizdeki `main` fonksiyonunun parametresiz, `printf` fonksiyonunun ise parametreli olduğu sonucunu çıkarabilirsiniz.

`main` fonksiyonundan sonra açılan ve kapanan küme parantezlerini izleyiniz.

C'de küme parantezleri arasındaki bölgeye blok denir.

Örneğin aşağıdaki şekilde toplam iki blok vardır. 2. blok 1. bloğun içerisindedir. Şekili inceleyiniz.



Bloklama amacıyla kullandığımız küme parantezlerini (brace) fonksiyon operatörü olan normal parantezlerle (parenthesis) karıştırmayınız. Verdiğimiz örneğe dönersek şöyle diyebiliriz:

`main` fonksiyonu bir blok içermektedir ve `printf` bu bloğun içerisindedir.

`main` şu biçimde devam ederken,

`main()`

```
{
    ...
}
```

`printf` ise şöyle devam ediyor:

```
printf("Merhaba C\n");
```

`main` ve `printf`, ikisi de birer fonksiyondur. Ancak `main` tanımlanmış `printf` ise çağrılmıştır. `main`'in tanımlandığını fonksiyon operatörlerinden sonra gelen blok işleminden `printf`'in çağırıldığını ise fonksiyon operatörlerinden sonra gelen noktalı virgül'den (;) anlıyoruz. `main` programın çalışmaya başladığı fonksiyondur. Diğer fonksiyonlardan başkaca bir farkı yoktur:

C programları `main` fonksiyonundan çalışmaya başlar.

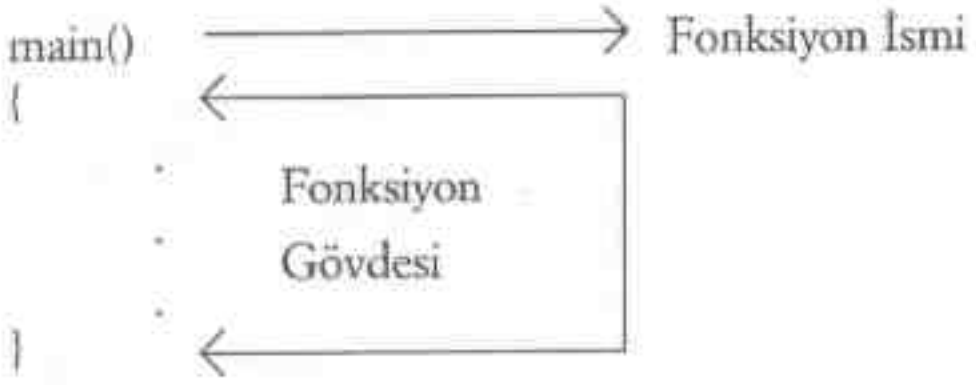
(Bazı durumlarda noktadan sonra bilerek küçük harfle başlıyoruz. Çünkü C' de büyük ve küçük harfler diğer dillerin aksine farklı karakterler olarak ele alınırlar. Yani C büyük harf ,küçük harf duyarlılığı olan bir dildir (case sensitive). Eğer biz noktadan sonra, örneğin Main diye başlasaydık dilbilgisi hatasından çok daha büyük bir C hatası yapmış olacaktık. C'de hata yapmamayı tercih ediyoruz. Dilbilimciler bizi affetsin!)

3.8 FONKSİYONLARIN TANIMLANMASI VE ÇAĞRILMASI

Tanımlanmış fonksiyonlar bizim tarafımızdan yazılmış olan fonksiyonlardır. Oysa, fonksiyonların çağrılması onların icraya davet edilmesi anlamına gelir. Kuşkusuz, bir fonksiyonun çağrılabilmesi için onun daha önce birisi tarafından tanımlanmış olması gerekir.

Fonksiyonlar her zaman tanımlanmış fonksiyonların içerisinde çağrılabilirler.

Tanımlanmış fonksiyonların bir ismi ve içlerinde ne yapacaklarına ilişkin kodların yazılı olduğu bir de gövdesi vardır:



Aklınıza şöyle bir soru gelebilir; **main** fonksiyonunu biz yazdık ve bunun içerisinde **printf** fonksiyonunu çağırdık. Peki, **printf** fonksiyonunu biz yazmadığımıza göre kim yazdı?..

Biraz sabrederseniz bu sorunuzun yanıtını ileride fazlasıyla bulacaksınız.

3.9 C'NİN GENEL YAZIM KURALI

Örnek olarak verdiğimiz C programını atomlarına ayırdıktan sonra "boşluk karakterleri"ni tanıtacağız.

```
#
include
<
stdio.h
>
main
(
)
{
printf
(
"Merhaba C\n"
)
;
}
```

3.9.1 Boşluk Karakterleri (Whitespace)

Klavyeyle yazarken boşluk bırakmak için kullanılan karakterlere boşluk karakterleri diyoruz. Boşluk karakterleri üç tanedir.

1) Ara tuşuna basarak elde ettiğimiz boşluk karakteri **(SPACE)**: Hex: 20h, Desimal: 32

2) Tab tuşuna basarak elde ettiğimiz boşluk karakteri **(TAB)**: Hex: 09h, Desimal: 9

3) Metin editörlerinde Enter tuşuna basarak elde ettiğimiz boşluk karakterleri **(ENTER)**: *DOS* altında metin editörlerinde Enter tuşuna basıldığında metin dosyasına iki karakter yazılmaktadır: CR (Carrige Return) Hex: 0D, Desimal: 13 ve LF (Line Feed): Hex: 0A, Desimal: 10

Boşluk karakterlerini özetle: **SPACE + TAB + ENTER (CR/LF)** biçiminde belirtebiliriz.

C'nin genel yazım kuralı: Atomlar arasında istenildiği kadar boşluk karakteri bulunabilir ve yan yana bulunan anahtar sözcükler ile değişkenler dışında tüm atomlar istenildiği kadar bitişik yazılabilirler. # içeren satırlar bu kurala dahil değildir.

Bu durumda yukarıdaki C programı, örneğin abartılı bir biçimde aşağıdaki gibi yazılabilirdi.

```
#include <stdio.h>

main        (
)
{
    printf
(
    "Merhaba C\n"
```

```
)
        ;
}
```

Ya da yine abartılı olarak:

```
#include<stdio.h>
main(){printf("Merhaba C\n");}
```

yazılabilir. Bu iki yazım biçimi de yukarıdaki kurala uymaktadır. # içeren satırlar diğer satırlarla birlikte yazılamazlar; fakat bunlar dışındaki bütün atomlar istenildiği kadar bitişik yazılabilirler. Programlama dillerinde atomların asla bölünemeyeceğine dikkat ediniz. Gerçekten de C derleyicileri kaynak kodu atomlarına ayırırken boşluk karakterlerini **atom ayıraçları (token delimeters)** olarak kullanırlar.

C'de bu kadar serbest bir yazım kuralı olmasına karşın en çok vurgulanan temalardan bir tanesi "*kaynak kodun okunabilirliği (readability)*" dir. Programlama dillerinin değerleme ölçütlerini ele aldığımız 1. Bölümde açıklamıştık: Okunabilirlik, kaynak koda bakılınca ne yapılmak istendiğinin kolaylıkla algılanabilmesidir.

İyi bir C programcısı ile deneyimsiz bir C programcısını ayıran en önemli özelliklerden bir tanesi okunabilirliktir. Ünlü bir sistem programcısının dediği gibi:

"En iyi C kodu birkaç sene sonra bakıldığında hemen anlaşılabilendir."

Okunabilirlik (readability) konusunun peşini hiç bırakmayacağız. Kitabımızda ilerledikçe bu konuda çeşitli uyarılar ve tavsiyeler göreceksiniz.

SORAMADIKLARINIZ...

S1) Yukarıdaki örneklerde `printf`, `scanf` gibi Standart C Fonksiyonlarının değişken atomlar olduğu belirtildi. Hemen her C programında rastlayabildiğimiz bu atomların aslında anahtar sözcükler olması gerekmiyor mu?

C1) `printf` ve `scanf` gibi Standart C Fonksiyonları anahtar sözcük değildir. Nitekim, örneklerimizde de bunları değişken atomlar olarak ele aldık. Programlama dillerinde anahtar sözcükler derleme zamanında (compile time) anlamlandırılırlar. Oysa `printf`, `scanf` gibi fonksiyonlar bağlama zamanında (link time) işleme sokulmaktadır. Yani, derleyici için `printf` ve `scanf` bizim tanımladığımız fonksiyon isimleri gibi sıradan birer isimdir. Bu fonksiyonların bütün sistemlerde hep aynı isimlerle bulunması anahtar sözcük izlenimini uyandırmaktadır.

S2) Her C programının başında

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

satırını görüyoruz. Bu satırın mutlaka yazılması gerekir mi? Eğer her zaman yazılması gerekiyorsa, neden derleyici bunu yazılmış varsayarak bizleri bu zahmetten kurtarmıyor?

C2) `stdio.h` dosyasının içerisinde neler olduğunu bilsek bu soruya yanıt vermiş olacağız. Şimdilik yalnızca şunları söyleyebiliriz: Bu satırın yazılması, yani

`stdio.h` dosyasının derleme işlemine dahil edilmesi bazı durumlarda mutlak zorunlu olmasa da çoğu zaman gereklidir.

S3) Her *C* programında `main` fonksiyonunun bulunması gerekir mi? Bir C programında birden fazla `main` fonksiyonu olabilir mi?

C3) Evet, her *C* programında bir `main` fonksiyonu bulunması gerekir (Windows altında çalışan *C* programlarında `main` fonksiyonu yerine `WinMain` fonksiyonu vardır). Çünkü `main` fonksiyonu programın çalışmaya başladığı yeri belirtmektedir. Standart *C*'de aynı isimli birden fazla fonksiyon olamaz (Fakat *C++*'da farklı parametrelere sahip aynı isimli birden fazla fonksiyon olabilmektedir).

S4) Nesne ile değişken kavramları arasında ne fark var?

C4) Nesne bellekte erişilebilir olan bir bölgedir; değişkenden daha genel bir anlamı vardır. Oysa değişkenler isimlerini bizim verdiğimiz nesnelerdir. Birçok durumda iki terim eş anlamlıymış gibi kullanılmaktadır. Her değişken bir nesnedir; fakat her nesne bir değişken değildir. Göstericiler konusundan sonra anlamlandırabileceğiniz bir örnek verebiliriz:

```
* (char *) 0X1FC0
```

ifadesi bir nesnedir, ancak değişken değildir.

4 VERİ VE NESNE TÜRLERİ

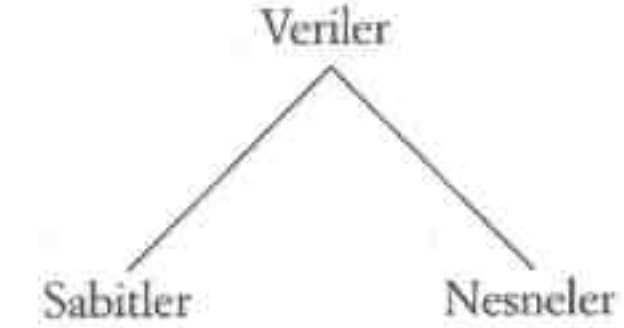
Bu bölümde C'deki veri ve nesne türleri hakkında bilgiler bulacaksınız. Tür bir nesnenin en önemli özelliğidir. Çünkü derleyiciler nesneleri türlerine bakarak yorumlarlar. Nesnelerin uzunlukları da türlerine bağlı olarak değişmektedir. Bölüm içerisinde tür ve uzunluk arasındaki ilişkiler çok çeşitli donanımlar gözönüne alınarak ayrıntılı bir biçimde incelenmektedir.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) *C*'deki veri türleri nelerdir?
2) Veri türlerinin *DOS* ve *UNIX* sistemlerindeki uzunlukları ne kadardır?
3) İşaretli türler ile işaretsiz türler arasındaki farklar nelerdir?
4) İşaretli türlerin taşması ne anlama gelir ve sonuçları nelerdir?

4.1 TÜR KAVRAMI

"*Veri türleri*" tamlaması "*nesne türleri*" tamlamasından daha genel bir anlam içermektedir. Çünkü, "veri" terimi bellekte yer göstersin ya da göstermesin bütün bilgileri kapsar. Yani hem sabitler hem de nesneler "veri" olarak yorumlanabilirler.



Tür bilgisinin bir nesne özelliği olduğunu önceki bölümden anımsıyorsunuz. Bir nesnenin türü deyince, o nesne içerisinde tutulan bilginin derleyici tarafından yorumlanış biçimi anlaşılmalıdır. Nesnelerin türleri, bize aynı zamanda onların bellekte kapladığı alan hakkında da bilgi vermektedir.

Aşağıdaki tabloyu inceleyiniz.

Anahtar sözcük	Uzunluk (Bit) (DOS/UNIX)	Sınır Değerleri
[signed] char	8	[-128, +127]
[signed] short [int]	16/16	[-32768, +32767]
[signed] int	16/32	[-32768, +32767]
long [int]	32/32	[-2147483648, +2147483647]
float	32	[3.4E-38, 3.4E+38]
double	64	[1.7E-308, 1.7E+308]
long double	80	[Taşınabilir değil]
unsigned char	8	[0, +255]
unsigned short [int]	16/16	[0, +65535]
unsigned [int]	16/32	[0, +65535]
unsigned long [int]	32/32	[0, +4294967295]

Yukarıdaki tabloyu açıklayalım. Birinci sütunda bildirim yaparken kullandığımız anahtar sözcüklerin listesini görüyorsunuz. (Bildirim kavramı sonraki bölümde ele alınmaktadır.) Köşeli parantez içerisinde sözcükler isteğe bağlı olarak belirlemelerde kullanılmayabilirler. Yani örneğin **signed long int** ile **long** aynı anlama gelir. Biz kitabımızda en kısa biçimlerini kullanacağız. İkinci sütun ise bu türlere ilişkin verilerin bit cinsinden kapladıkları bellek miktarlarını gösteriyor. Bölü işaretinin (/) sol tarafındaki değerler *DOS*, sağ tarafındaki değerler ise *UNIX, Windows 95* ve *NT* tabanlı sistemler için geçerlidir. Son sütunda da bu verilerin alabileceği alt ve üst sınırlar verilmiştir.

Şimdi bu türleri tek tek tanıyalım:

int: Nokta içermeyen sayılardır. Pozitif ve negatif olabilirler. Örneğin:
-124, 30123, 3000, -14000, ... gibi.

int türünün uzunluğu her sistemde aynı değildir. Bu türün donanım bağımlı olduğunu söyleyebiliriz.

Tamsayı türünün uzunluğu çalıştığımız sisteme ilişkin mikroişlemcinin bir kelimesi kadardır.

Örneğin, 16 bitlik mikroişlemcilerin kullanıldığı sistemlerde **int** türü 16 bit, 32 bit mikroişlemcilerin kullanıldığı sistemlerde ise 32 bit uzunluğundadır.

Belli başlı donanımlar için **int** türünün uzunluğu aşağıda verilmiştir.

DONANIM	int TÜRÜNÜN UZUNLUĞU (BİT)
INTEL 80X86 (DOS, WINDOWS 3.1)	16
INTEL 80X86 (UNIX, XENIX, WINDOWS 95)	32
3B	32
DEC PDP-11	16
DEC VAX	32
HONEYWELL 6000	36

IBM 360/370	16
INTERDATA 8/32	16
MOTOROLA 68000	32
NSC 16000	32
ZILOG 8000	16

int türü "pozitif ya da negatif olabilir" demiştik. Tabloda *DOS* için işaretli **int** sınırının **[-32768, +32767]** biçiminde olduğunu görüyorsunuz. Bu sınır değerler nereden geliyor dersiniz? İşaretli sayılar hakkında II. Bölümde yaptığımız açıklamaları anımsayınız.

16 bit içerisinde yazılabilecek en büyük pozitif sayıyı araştıralım:

İkilik sistemde	**0111 1111 1111 1111**
Onaltılık sistemde	**7 F F F**
Onluk sistemde	**+32767**

En soldaki bit, işaretin pozitif olduğunu gösteren işaret bitidir. (Temel kavramlar bölümünden anımsayınız.)

Ya en küçük negatif sayı? Yukarıdaki sayının ikiye tümleyeninden bir çıkartarak elde edebiliriz:

En büyük pozitif sayı:	0111 1111 1111 1111	+32767
İkiye tümleyeni:	1000 0000 0000 0001	-32767
Bir eksiği:	1000 0000 0000 0000	-32768

Bu durumda en küçük negatif sayının 2'lik, 16'lık ve 10'luk sistemlerdeki görünümü aşağıdaki gibi olacaktır:

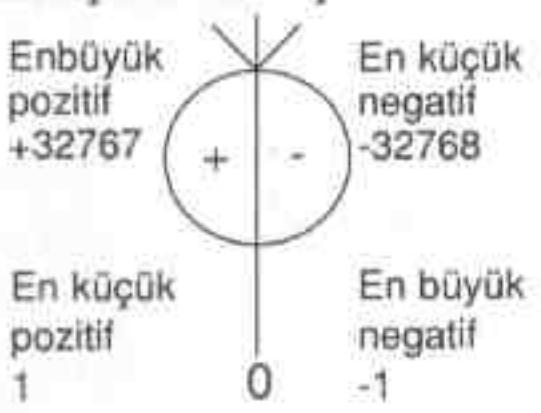
İkilik sistemde	**1000 0000 0000 0000**
Onaltılık sistemde	**8 0 0 0**
Onluk sistemde	**-32768**

Şimdi ilginç bir deneme yapalım; en büyük pozitif tamsayıya bir toplayalım:

```
0111 1111 1111 1111   (7FFF)
0000 0000 0000 0001   (0001)
---------------------
1000 0000 0000 0000   (8000)
```

Ne görüyoruz? Bu en küçük negatif sayı değil mi? O halde şu sonucu çıkarabiliriz:

Sınırı aşan pozitif sayılar negatif bölgeye geçerler. C'de olduğu gibi, derleyici tarafından sınır kontrolünün yapılmadığı dillerde beklenmedik negatif bir sayıyla karşılaşırsanız sınır taşmasından kuşkulanmalısınız!..



short: Bu tür diğer tamsayı türlerinde olduğu gibi nokta içermeyen pozitif ve negatif sayılardan oluşur. Uzunluğu dışında **short** türü ile **int** türü arasında işlevsel hiçbir farklılık yoktur. Bu türün uzunluğu da **int** türünde olduğu gibi sistemden sisteme değişebilir. Ancak şu kural her zaman geçerlidir:

short <= int

Örneğin, *DOS*'ta **short** ve **int** türleri birbirine eşittir (her ikisi de 16 bit). Oysa *UNIX* tabanlı sistemlerde 16 bit olan **short** türü, 32 bit olan **long** türünden daha küçüktür. Özetle şöyle diyebiliriz:

DOS 'ta short ve int türleri arasında işlevsel hiçbir fark yoktur. Ancak UNIX tabanlı sistemlerde short (16 bit), int türünün (32 bit) yarısı kadardır.

Fakat, biz yine de belli başlı donanımlar için **short** türünün uzunluğunu vermek istiyoruz.

DONANIM	short TÜRÜNÜN UZUNLUĞU (BİT)
INTEL 80X86 (DOS, WINDOWS 3.1)	16
INTEL 80X86 (UNIX, XENIX, WINDOWS 95)	16
3B	16
DEC PDP-11	16
DEC VAX	16
HONEYWELL 6000	36
IBM 360/370	16
INTERDATA 8/32	16
MOTOROLA 68000	16
NSC 16000	16
ZILOG 8000	16

Gördüğünüz gibi, **short** türü, biri dışında geri kalan sistemlerin hepsinde 16 bittir.

long: Bu tür de **int** türünden daha uzun olabilen bir türdür. Uzunluğundan başka **int**, **short** ve **long** türleri arasında işlevsel hiç bir farklılık yoktur. Yine **long** türü de sistemden sisteme değişebilir. Ancak, aşağıdaki kural her zaman geçerlidir:

long int >= int

DOS altında çalışan derleyicilerde **long** türü, **int** türünden iki kat daha uzundur. Fakat *UNIX* tabanlı sistemlerde **long** ile **int** türlerinin ikisi de birbirine eşit-

tir. Özetle şöyle diyebiliriz:

UNIX tabanlı sistemlerde int türü ile long türü arasında işlevsel hiçbir farklılık yoktur. Ancak DOS' ta long (32 bit) int türünden (16 bit) iki kat daha uzundur.

Aşağıda **long** türünün belli başlı donanımlardaki uzunluğu verilmiştir.

DONANIM	long TÜRÜNÜN UZUNLUĞU (BİT)
INTEL 80X86 (DOS, WINDOWS 3.1)	32
INTEL 80X86 (UNIX, XENIX, WINDOWS 95)	32
3B	32
DEC PDP-11	32
DEC VAX	32
HONEYWELL 6000	36
IBM 360/370	32
INTERDATA 8/32	32
MOTOROLA 68000	32
NSC 16000	32
ZILOG 8000	32

Bütün bu anlatılanlar üzerine şöyle bir soru aklınıza gelebilir: "Önceki bölümlerde C'nin çok taşınabilir bir dil olduğunu söylediniz. Peki nasıl oluyor da daha temel veri türlerinin uzunlukları bile her sistemde aynı olmuyor ve standardize edilmemiş oluyor? Bu durum taşınabilirliğe engel oluşturmuyor mu?"

- Gerçekten de C öğrencileri işin başındayken haklı olarak bu soruyu hep sorarlar. Oysa, biz bu soruyu ancak sonraki bölümlerde birtakım kavramları öğrendikten sonra tam anlamıyla yanıtlayabiliriz. Şimdilik şunu söylemekle yetineceğiz:

C'de, veri türlerinin uzunluklarından dolayı oluşan taşınabilirlik problemlerini giderici mekanizmalar da vardır.

char: Uzunluğu en kısa olan türdür. Sistemlerin hemen hepsinde **char** türünün uzunluğu 8 bittir.

C'de char türüyle int, short ve long türleri arasında uzunluklarından başka işlevsel hiçbir fark yoktur. Diğer dillerin aksine C'de char türü de diğer türlerle aritmetik, mantıksal, vs. işlemlere sokulabilir.

char türüyle **int**, **short** ve **long** türleri arasında uzunluklarından başka hiçbir fark olmaması sizi biraz şaşırtabilir. Çünkü yüksek seviyeli dillerin hemen hepsinde bu türler farklı özelliklere sahiptir. Örneğin, karakter nesneleri yalnız **char** bil-

gilerini saklamak için kullanılırlar. Şöyle bir soru da sorabilirsiniz: "Madem aralarında bir fark yok, peki o zaman neden bu türün ismini **char** (karakter) koymuşlar?"

- İsminin **char** olması bu türün uzunluğunun sistemlerde kullanılan karakter uzunluğuna eşit olmasından kaynaklanmaktadır. Gerçekten de bu tür, karakter bilgilerini ifade edebilmek için en ekonomik olan türdür.

char türünün uzunluğu sistemlerin hemen hepsinde 1 byte yani 8 bit olduğuna göre:

1 byte içinde yazılabilecek en büyük pozitif tamsayı:

İkilik sistemde	**0111**	**1111**
Onaltılık sistemde	**7**	**F**
Onluk sistemde	**+127**	
En küçük negatif sayı:		
İkilik sistemde	**1000**	**0000**
Onaltılık sistemde	**8**	**0**
Onluk sistemde	**-128**	

biçimindedir.

float: Bu tür gerçek sayıları belirtmekte kullanılır. Pozitif ve negatif olabilirler. Örneğin:

```
-12.786950, 304.747322, 3.141592...
```

float türü istisna birkaç sistem dışında hep 4 byte uzunluğundadır ve özellikle kesir kısmı için kullanılırlar ve bellekte tamsayılardan farklı bir biçimde tutulurlar.

double: Bu tür **float** türünde olduğu gibi pozitif ve negatif kesirli sayıları ifade etmek için kullanılır. **float** türünden iki kat daha duyarlıklıdır. Sistemlerin hemen hepsinde 64 bit (8 byte) uzunluğundadır. Arzu ettiğiniz hesaplama gerçek sayı duyarlılığından daha fazla bir duyarlılık gerektiriyorsa bu türü kullanabilirsiniz.

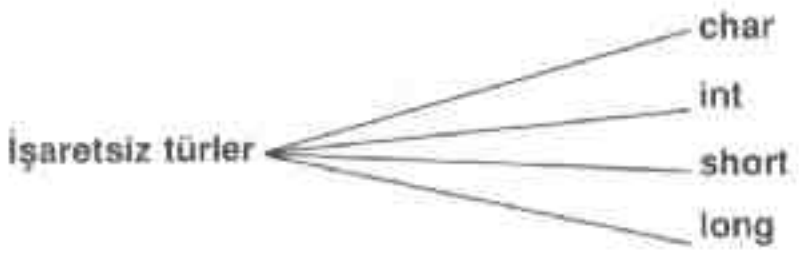
long double: double türünden daha yüksek duyarlılığa sahiptir. Bu tür sistemlerin hemen hepsinde 10 byte uzunluğundadır. Ancak **long double** türü her C derleyicisinin desteklediği taşınabilir bir tür değildir.

Not: Gerçek sayıların bellekte tutulmalarına ilişkin ANSI / IEEE 754 standardı "Ekler -A-1" bölümünde ele alınmaktadır.

4.2 İŞARETLİ VE İŞARETSİZ TÜRLER

Yukarıda incelediğimiz türlerin hepsi pozitif ve negatif olabilen **işaretli türlerdir.** Bunların yanısıra dört veri türünün bir de işaretsiz olan biçimleri vardır. İşaretsiz türlerde sayıların en solunda bulunan bitler **işaret bitleri** olarak ele alınmazlar; bu bitler diğer bitler gibi yorumlanırlar. İşaretsiz türlerin hep pozitif oldukları varsayılır.

İşaretsiz türler işaretli olan benzerleriyle aynı uzunluğa sahiptirler; ancak pozitif sınırları iki kat daha büyüktür. Aşağıda "işaretsiz biçimleri olabilen" 4 türü görüyorsunuz:



Aşağıdaki tabloda ise işaretsiz türlerin sınır değerleri verilmiştir. İşaretli olan biçimleriyle karşılaştırınız.

Tür Belirten Anahtar Sözcük	Sınır Değerleri
[signed] char	[-128, +127]
unsigned char	[0, +255]
[signed] int	[-32768, +32767]
unsigned [int]	[0, 65535]
[signed] long [int]	[-2147483648, +247483647]
unsigned long [int]	[0, +4294967295]

Örneğin:

1111 1010 sayısı işaretli olarak −6, işaretsiz olarak +250 biçiminde yorumlanır.

Son olarak şimdiye kadar görmüş olduğumuz veri/nesne türlerini iki gruba ayırmak istiyoruz. Köşeli parantezler içerisindeki anahtar sözcükler yazılı ya da sözlü anlatımlarda kullanılmayabilirler.

1) Tamsayı türleri

```
[signed] char
unsigned char
short [int]
unsigned short [int]
[signed] int
unsigned [int]
[signed] long [int]
unsigned long [int]
```

2) Gerçek Sayı Türleri

```
float
double
long double
```

SORAMADIKLARINIZ...

S1) *C*'de yazılabilecek en büyük tamsayı olan işaretsiz **unsigned long**'un sınır değeri *4,294,967,295*'tir. Peki bu durumda daha büyük sayılar nasıl ifade edilecekler? Enflasyon da malumunuz. Birkaç yıl içinde milyar *TL*'ler günlük yaşantımıza daha fazla girecek. O zaman alacak verecekleri hangi türlerle ifade edebileceğiz?

C1) **long** sınırlarını aşan tamsayıları gerçek sayı formatında **double** türü ile ifade edebilirsiniz. Abartılı büyüklükteki sayılar ise ancak programcı tarafından tanımlanan yapılar içerisinde tutulabilirler . Konular ilerledikçe bu sorunların üstesinden nasıl gelineceğini kendiniz bulacaksınız...

S2) **int** türü neden makinanın donanımına bağlıdır? Örneğin, neden "hep 2 byte ya da 4 byte uzunluğundadır" biçiminde standardize edilmemiştir?

C2) **int** türü *C* programlarında en fazla işlem gören türdür. *C* çok doğal bir dildir; **int** türünün mikroişlemcinin bir kelimesi kadar olması da bu doğallık ile uyum içindedir. Derleyicileri yazanlar ve uygulama programcıları - farkında olmasalar bile - bundan yarar sağlarlar. Daha ayrıntılı bir yanıtı içinde bulunduğumuz duruma uygun görmediğimizden bu kadarıyla yetineceğiz.

S3) *DOS*'ta **short** ile **int**, *UNIX*'te de **long** ile **int** türleri arasında gerçekten hiçbir fark yok mudur; bunlar tamamen eşdeğer türler midir?

C3) *DOS*'ta **short** ile **int**, *UNIX*'te de **long** ile **int** işlevsel olarak tamamen aynı türlerdir; aralarında hiç fark yoktur. Ancak sizin kaynak kodunuzda bunlardan hangisinin olacağı taşınabilirlik açısından önemli olabilir. Örneğin siz, DOS'ta da 2 byte UNIX'te de 2 byte uzunlukta bir tamsayı türü kullanmak istiyorsanız o zaman **short** türünü tercih etmelisiniz.

S4) Bilgisayarda herşey 1'ler ve 0' lardan oluştuğuna göre noktalı sayıların bellekte saklanmasına ilişkin bir kural var mıdır? **float** ve **double** türleri bellekte nasıl tutuluyorlar?

C4) Noktalı sayıların bellekte saklanma biçimleri tam olarak standardize edilmemiştir. Noktalı sayıların saklanmasında 2 önemli standart vardır. Bunlardan birincisi - ki en yaygın olarak kullanılanıdır- **IEEE (The Institute of Electrical and Electronics Engineers)** standardıdır. Örneğin, *80X87* matematik işlemcileri de bu standardı kullanır. İkincisi ise daha az kabul gören **Microsoft Binary** standardıdır. *Microsoft* bu formatı eski *BASIC* derleyicilerinde kullanıyordu. Ancak daha sonra kendisi bile bu formattan vazgeçmiştir. ANSII / IEEE 754 gerçek sayı formatları "Ekler -1-A" bölümünde ayrıntılı bir biçimde açıklanmaktadır.

S5) Gerçek sayı türleri neden tamsayı türlerine göre daha uzun tasarlanmışlardır? **double** ya da **long double** gibi yüksek bir duyarlılık içeren türlere gerek var mıdır?

C5) Gerçek sayıların bellekte tutulma biçimleri, bunlarla yapılan işlemlerde yuvarlama hatalarının (rounding error) oluşmasına yol açmaktadır. Yuvarlama ha-

taları önemsiz gibi görünse de çarpma ve bölme işlemleriyle büyüyebilir. Yuvarlama hatalarını en aza indirmek, ancak gerçek sayıları daha büyük duyarlılıkla ifade etmekle mümkün olabilir. *"Ekler-1-A"* bölümünde gerçek sayı formatları ve yuvarlama hataları hakkında bilgiler bulacaksınız.

S6) Mikroişlemcilerin büyük çoğunluğunun yalnızca tamsayılar üzerinde işlemler yaptığı söyleniyor. Bu durumda gerçek sayı işlemleri nasıl yapılıyor?

C6) Oldukça teknik bir soru olduğu için, kısa bir yanıt vermekle yetineceğiz. Gerçekten de yalnızca tamsayılar üzerinde işlem yapan mikroişlemcilere gerçek sayı işlemlerini yaptırmak için makina seviyesinde altprogramlar (kesmeler ya da fonksiyonlar) kullanılır. İki gerçek sayının bu biçimde toplanması büyük bir zaman kaybı oluşturmaktadır. *80X86* ailesinde gerçek sayı işlemlerini hızlı bir biçimde yerine getirmek için *8087, 80287, 80387* gibi matematik işlemcilerden faydalanılmaktadır. *80486 DX* ve *Pentium* işlemcilerinin matematik işlemcisi kendi içindedir.

S7) Neden gerçek sayı türlerinin işaretsiz biçimleri yoktur?

C7) Gerçek sayı türleri IEEE 754 numaralı standardına göre işaretli bir biçimde tasarlanmıştır. Gerçek sayı işlemlerini yapan matematik işlemciler de sayıları hep işaretli olarak ele alırlar. Gerçek sayı türleri yeteri kadar uzun olduğundan işaretsiz olarak ele alınıp duyarlılığın artırılmasına gerek duyulmamıştır. İşaretsiz türlerle işlem yapma ek bir donanım ve yazılım maliyetini de beraberinde getirir.

5 BİLDİRİM VE TANIMLAMA

Bu bölüm nesnelerin kullanılmadan önce derleyicilere tanıtılması ile ilgilidir. Bildirim (declaration) ve tanımlama (definition) işlemlerinin nerede ve nasıl yapılacağı bölüm içerisinde ayrıntılı bir biçimde ele alınmıştır.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

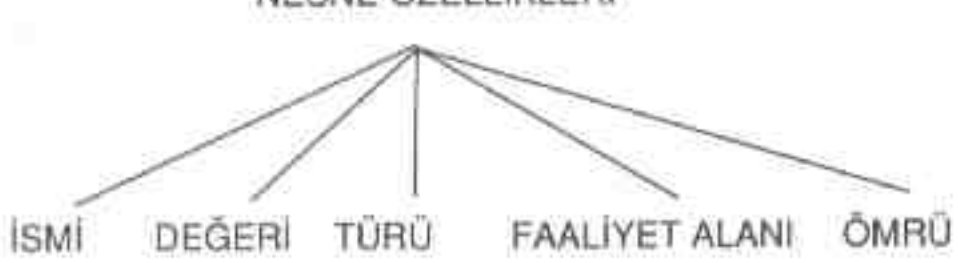
1) Bildirim ve tanımlama kavramları ne anlama gelmektedir?
2) Bildirimler nerede yapılabilirler?
3) Bildirim işleminin genel biçimi nasıldır?
4) Noktalı virgülün (;) işlevi nedir?

5.1 BİLDİRİM VE TANIMLAMA KAVRAMLARI

Modern programlama dillerinin çoğunda nesneler kullanılmadan önce derleyiciye tanıtılırlar.

Kullanılmadan önce nesnelerin özellikleri hakkında derleyiciye bilgi verilmesi işlemine bildirim (declaration) denir.

Derleyiciler bildirim işlemi sayesinde nesnelerin hangi özelliklere sahip olduklarını anlarlar. Nesnelerin sahip oldukları özellikleri tekrar anımsatmak istiyoruz.



Nesne yaratan bildirimler için **tanımlama (definition)** terimi kullanılmaktadır. **Tanımlama** ve **bildirim** kavramları aralarındaki benzerlik nedeniyle sıklıkla birbirlerine karıştırılır.

Tanımlama nesne yaratan bir bildirim işlemidir.

Derleyici bir tanımlama işlemiyle karşılaşınca, nesnenin belirtilen özelliklerine uygun olarak o nesne için bellekte bir yer tahsis eder.

5.2 BİLDİRİM İŞLEMİNİN GENEL BİÇİMİ

Bildirim işlemlerinin yapılış biçimi ve kuralları dilden dile değişir.

Genel Biçim:

```
<tür> nesne1[, [nesne2], [nesne3],...];
```

Yukarıdaki genel gösterimde **tür**, nesnenin hangi türe ait olduğunu gösteren bir anahtar sözcüktür. Nesne türlerine ilişkin anahtar sözcükleri önceki bölümde vermiştik; yeniden anımsatmak istiyoruz:

Tür Belirten Anahtar Sözcük	Uzunluk (byte) DOS WINDOWS 3.1 / UNIX WINDOWS 95
[signed] char	1
[signed] short [int]	2 / 2
[signed] int	2 / 4
[signed] long [int]	4 / 4
unsigned char	1
unsigned short [int]	2 / 2
unsigned [int]	2 / 4
unsigned long [int]	4 / 4
float	4 / 4
double	8 / 8
long double	10 / 10

Yukarıdaki gösterimde bazı anahtar sözcükler köşeli parantezler içerisinde yazılmıştır. Köşeli parantezlerin yazılması zorunlu olmayan yani isteğe bağlı olan ifadeleri gösterdiğini anımsayınız. Daha açık bir anlatımla: **signed short int** yerine yalnızca **short** ya da **signed long int** yerine yalnızca **long** yazabiliriz. C derleyicileri **short** ve **long** anahtar sözcüklerini gördüklerinde onları **signed short int** ve **signed long int** olarak kabul ederler. Burada olduğu gibi bundan sonra da genel gösterimlerdeki köşeli parantezleri, yazılıp yazılmaması "*isteğe bağlı olan (optional)*" ifadeler için kullanacağız.

Tür belirten anahtar sözcükten sonra aralarına virgül koyarak birden fazla nesnenin bildirimi aynı anda yapılabilir. Virgül (,) ileride ele alınacak olan bir ope-

ratördür. Genel biçimde kullandığımız üç nokta (...) ise *C*'ye ilişkin bir sintaks değil; ifadenin benzer biçimde devam edeceğini anlatan ve bizim tarafımızdan kullanılan bir gösterim biçimidir. Bildirim işleminin noktalı virgül (;) ile bittiğine de dikkat ediniz.

Örnek bildirimler:

```
int a, b, c;
float tutar;
char c, ch;
long int x, y, z;
short a1, a2, a3;
long  x1, y1;
double d1, d2;
...
```

Atomlar arasında istenildiği kadar boşluk karakterleri konulabileceğini anımsayınız...

```
int a, b;
```

bildirimini isterseniz, örneğin:

```
int
a   ,
b   ;
```

biçiminde de yazabilirsiniz. Fakat en okunabilir biçimi tercih etmelisiniz!..

Yukarıdaki bildirim işlemiyle, derleyiciye *nesnelerin ismi ve türü* hakkında açık bir bilgi veriyoruz. Peki derleyici diğer nesne özelliklerini nasıl anlıyor dersiniz? Bu sorunun yanıtı için biraz sabretmelisiniz!..

5.3 NOKTALI VİRGÜLÜN İŞLEVİ

C'de noktalı virgül (;) sonlandırıcı olarak görev yapmaktadır. Önceki bölümlerde noktalı virgülün ayıraç türünden bir atom olduğunu belirtmiştik. *C*'de bütün ifadeler noktalı virgül ile birbirlerinden ayrılırlar. Örneğin:

```
...; a = x + 1;     c = d - 1;
```

ifadelerinde noktalı virgüller iki ayrı ifadenin varlığını gösterir. Eğer noktalı virgül olmazsa derleyici iki ifadeyi tek bir ifade gibi ele almaya çalışacak ve buna bir anlam veremeyecektir:

```
...; a = a + 1     c = c - 1;
```

Diğer dillerde noktalı virgül yerine başka sonlandırıcı karakterlerin kullanıldığını da görebilirsiniz. Örneğin, BASIC'te ya da sembolik makina dilinde Enter (CR/LF) sonlandırıcı karakter olarak kullanılır. Bu da her ifadenin ayrı bir satır halinde yazılmasını gerektirmektedir.

5.4 DEĞİŞKEN İSİMLENDİRME KURALLARI

C'de değişken isimlendirme kuralı -büyük harf küçük harf duyarlılığı dışında- diğer dillerde olduğu gibidir. Buna göre:

1) C büyük harf küçük harf duyarlılığı olan (case sensetive) bir dildir. Büyük harflerle küçük harfler ayrı karakterler olarak algılanır. Örneğin aşağıdaki değişken isimlerinin hepsi birbirlerinden farklıdır:

```
sample
Sample
SAMPLE
...
```

2) Değişken isimleri nümerik bir karakterle başlayamaz. (Fakat alfabetik karakterlerden biriyle başlayıp nümerik karakterlerle devam edebilir.) Örneğin:

```
1nci_devre   Geçersiz
devre1       Geçerli
...
```

3) Değişken isimleri boşluk içeremez. Boşluk duygusu verebilmek için çoğunlukla alt tire karakteri kullanılmaktadır:

```
pencere yarat      Geçersiz
pencere_yarat      Geçerli
...
```

Windows programcılığında birden çok sözcükten oluşan değişken isimlerinin her sözcüğünün ilk harfinin büyük yazılması alışkanlık haline gelmiştir:

```
CreateWindow
HesaplaBordro
...
```

gibi.

4) Değişken uzunlukları sistemden sisteme değişebilir. Ancak çoğunda ilk 32 karakter dikkate alınmaktadır.

5.5 İŞARET BELİRLEYİCİLERİ VE İŞARETSİZ TÜRLERİN BİLDİRİMLERİ

Türlerin işaretleri hakkında bilgi veren iki işaret belirleyici anahtar sözcük vardır:

```
unsigned
signed
```

Bu iki işaret belirleyicisinin de yalnızca 4 tür anahtar sözcük ile kullanılabildiğini anımsayınız.

```
unsigned char
unsigned [int]
unsigned short [int]
unsigned long [int]

[signed] char
[signed] int
[signed] short [int]
[signed] long [int]
```

`char`, `int`, `short int` ve `long int` türleri zaten kendiliğinden (default olarak) işaretli türler oldukları için bunların önlerine `signed` anahtar sözcüğü getirilmeyebilir.

`signed char` ile `char`
`signed int` ile `int`
`signed short int` ile `short int`
`signed long int` ile `long int`

aynı anlamdadır. `unsigned int` yerine yalnızca `unsigned` anahtar sözcüğünü de yazabilirsiniz. Bu durumda:

`unsigned`
ile
`unsigned int`
aynı anlama gelmektedir.

Örnek Bildirimler:

```
unsigned short int x;      /* int yazılmayabilir*/
unsigned long int a;       /* int yazılmayabilir */
unsigned int max, min;     /* int yazılmayabilir */
signed long int b;         /* signed ve int yazılmayabilir */
signed char ch;            /* signed yazılmayabilir */
unsigned x, y;             /* unsigned int x, y ile aynı anlamdadır */
```

```
unsigned long a, b;
int toplam;
float a1, a2;
...
```

5.6 BİLDİRİMLERİN YAPILIŞ YERLERİ

Standart *C (ANSI C)*' de bildirimler üç yerde yapılabilir:

1) Blokların başlarında

2) Bütün blokların dışında

3) Fonksiyon parametresi olarak, fonksiyon parantezlerinin içinde ya da fonksiyon parantezlerinden sonra.

Biz bu bölümde yalnızca blokların başlarında yapılan bildirimleri inceleyeceğiz. Fakat önce blok kavramını anımsatmak istiyoruz:

Küme parantezleri arasındaki bölgelere blok denir.

```
{
    ...
    {
        ...
    }
    ...
}
```

Yukarıdaki şekilde iki tane blok vardır. İkinci blok birinci bloğun içindedir. Peki blokların başı demekle neyi anlatmak istiyoruz? Blokların başları, açılış küme parantezinden sonraki ilk ifadeleri anlatmaktadır. **C'de bildirimler blok içerisinde yapılacaksa, blokların ilk işlemi olmak zorundadır.** Bildirimlerden önce başka bir ifade bulunamaz ya da fonksiyon çağrılamaz.

Örneğin:

```
...
{
    int a, b;
    char ch;

    a = 100;
    b = 300;
    ...
    {
        float x;
```

```
            a = a * 2;
            ...
        }
        ...
}
        ...
```

Burada **a, b, ch** ve **x** nesnelerinin bildirimleri kurala uygun olarak blokların başlarında yapılmıştır. Şimdi aşağıdaki örneği inceleyiniz:

```
...
    {
        a = 3 *  b;
        int x, y;
           ...
           {
               char ch;

               deneme();
           }
           ...
    }
...
```

Birinci bloktaki **x** ve **y** nesnelerinin bildirim yeri yanlıştır. Çünkü kendisinden önce bir ifade yazılmıştır. Ancak içerideki blokta, **ch** nesnesinin bildirimi doğru yerde yapılmıştır. *C* öğrencilerinin en sık yaptığı hatalardan birisi blok içerisindeki bildirimlerden önce fonksiyon çağırmaktır. Örneğin:

```
...
{
    clrscr();
    int a, b;
    char ch;
}
...
```

a, b ve **ch** nesnelerinin bildirimleri hatalıdır.

NOT: C++ da blok içerisindeki nesnelerin bildirimleri blokların başlarında yapılmak zorunda değildir. Herhangi bir yerde yapılabilir. Bu yüzden yukarıdaki tüm kodlar C++' da hata oluşturmaz. Uzantısı -derleyiciye bağlı olarak - .CPP, .CXX olan C++ kodlarında bildirimlerin blokların başlarında yapılmadığını görürseniz şaşırmayın!..

Aynı türden birden fazla nesnenin bildirimini aynı satırda yapmak biçiminde bir zorunluluk yoktur.

```
...
{
    int a, b;
```

```
        char x, y;
        ....
}
...
```

yerine

```
...
{
        int a;
        char x;
        char y;
        int b;
        ...
}
...
```

de yazabilirdik.

SORAMADIKLARINIZ...

S1) Tanımlama (definition) işlemi ile derleyicilerin nesneler için bellekte yer ayırdıkları belirtildi. Bu yer ayırma işlemi nasıl yapılmaktadır ve belleğin neresi kullanılmaktadır?

C1) Derleyiciler -ileride ele alınacak- yer ayırma işlemlerini nesnelerin ömürlerine göre, **durağan (static)** ya da **dinamik (dynamic)** olarak yaparlar. Durağan ayırma işleminde nesnenin yeri programın çalışmak üzere yüklenmesiyle belirlenir. Oysa, dinamik ayırmada nesnenin yeri potansiyel olarak belirlenmiştir; programın çalışma zamanı sırasında (run time) gerçek yer tahsis edilir. Durağan ayırma işlemlerinde *80X86* sisteminde **data segment** bölgesi, dinamik ayırmada ise **stack segment** bölgesi kullanılmaktadır.

S2) İşaret belirleyici anahtar sözcükler kullanılmamışsa nesneler **işaretli (signed)** olarak varsayıldıklarına göre, o halde neden **signed** anahtar sözcüğüne gerek duyuluyor?

C2) Derleyicilerin çoğunda bazı türlerin varsayılan işaret özelliğini değiştirebilirsiniz. Derleyici seçenekleriyle oynayarak örneğin, yalnızca **char** anahtar sözcüğü **unsigned char** biçimine dönüştürülebilir. Bu durumda onları işaretli yapmak için özellikle **signed char** yazmak gerekir. Daha açık olarak:

1) **char** zaten **signed char** ile aynı anlamdadır.

2) Derleyici seçeneklerini değiştiriyorsunuz. **char** şimdi **unsigned char** oluyor.

3) Şimdi işaretli karakter türünü anlatmak için **signed char** bildirimini yapmak zorundasınız.

6 SABİTLER

Sabitler konusu programlama dillerinde önemli bir yer tutar. Bu bölümde C'deki sabit türleri ve bunların ifade biçimleri ele alınmaktadır.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Sabit nedir?

2) **int**, **long**, **float** ve **double** sabitleri nasıl ifade edilir?

3) **char** sabitleri kaç biçimde ifade edilir?

4) **char** sabitleri sayısal olarak ne anlam ifade etmektedir?

5) Önceden tanımlanmış ters bölü karakter sabitlerine neden ihtiyaç duyulmuştur?

6) İşaretsiz sabitler ile işaretli sabitler arasında nasıl bir ilişki vardır?

6.1 SABİT KAVRAMI

Veriler ya nesnelerin içerisinde ya da doğrudan sabit biçiminde bulunurlar. Sabitler nesne biçiminde olmayan verilerdir.

Örneğin:

```
c = a + b;
```

ifadesi bize, **a** ve **b** değişkenlerinin içerisindeki sayıların toplanacağı ve **c**'ye aktarılacağını anlatır. Oysa,

```
c = a + 100;
```

ifadesinde **a** ile **100** sayısı toplanmıştır. Burada **100** herhangi bir değişkenin içerisinde değildir, doğrudan sayı biçiminde yazılmıştır.

Nesnelerin türleri olduğu gibi sabitlerin de türleri vardır. Nesnelerin türleri bildirim yaparken belirtiliyordu. Peki sabitlerin türleri nasıl belirtiliyor dersiniz?

6.2 SABİT TÜRLERİ

C'de veri/nesne türü kadar sabit türü vardır. Bu bölümde bunları tek tek ele alıp inceleyeceğiz. Ancak hemen şunu belirtelim:

C'de bir atama ifadesinde sağ taraf ve sol taraf değerinin aynı türden olması gerekmez. Farklı türlerin farklı türlerdeki nesnelere atanması durumu 12. Bölümde ayrıntılarıyla ele alınmaktadır. Ancak bu bölüme gelene kadar örneklerimizde tür uyuşumunu tam olarak sağlayamaya özen göstereceğiz.

6.3 int SABİTLERİ

Bunlar tipik olarak tamsayı değişkenlerine atanan sabitlerdir.

C'de int sınırları içerisinde olan her sayı birer int sabiti olarak ele alınır.

Örneğin:

```
-100
 5500
-12000
```

gibi sayılar birer **int** sabitidir. C'de **int** sabitler herhangi bir ek almazlar. **int** türünün donanım bağımlı olduğunu anımsatmak istiyoruz:

Uzunluk		Sınır Değerleri
DOS, WINDOWS 3.1	2 byte	[-32768, +32767]
UNIX, WINDOWS 95	4 byte	[-2147483648, +2147483647]

DOS altında çalışıyorsak eğer, [-32768, +32767] arasındaki her tamsayı aynı zamanda bir tamsayı sabitidir. Örneklerimize devam edelim:

-1000	**int** sabitidir. İşaretli tamsayı sınırları içinde.
4500	**int** sabitidir.
-40000	**int** sabiti değildir. Sınır dışında.
30000	**int** sabitidir.
50000	**int** sabiti değildir.
32768	**int** sabiti değildir.

...

Son olarak, **int** sabitlerinin program içinde kullanılışına da bir bir örnek verelim:

```
...
{
     int a, b;              /* int türünden değişkenler */

     a = 100;               /* int sabiti atanıyor */
     b = -50;               /* int sabiti atanıyor */
     ...
}
...
```

6.4 short SABİTLERİ

short sabitleri de **int** sabitleri gibi ek almazlar. **short** sınırları içerisindeki tüm sayılar **short** sabiti olarak kabul edilirler.

short türünün uzunluğunun **DOS, WINDOWS 3.1** ve **UNIX, WINDOWS 95** tabanlı sistemlerin hepsinde 2 byte olduğunu anımsayınız:

Uzunluk		Sınır Değerleri
DOS, WINDOWS 3.1	2 byte	[-32768, +32767]
UNIX, WINDOWS 95	2 byte	[-32768, +32767]

Örneğin, *DOS* için:

```
4000
6500
-2000
```

sayılarının hepsi hem **int** hem de **short** sabitidir. Yani,

DOS'ta her short sabit aynı zamanda bir int sabit, her int sabit de aynı zamanda bir short sabittir.

UNIX tabanlı sistemler için de birkaç örnek verelim:

1500	Hem **int** hem de **short** sabiti.
56000	int sabiti ancak **short** sabiti değil.
-46000	**int** sabiti ancak **short** sabiti değil.

Not: **short** işlem öncesi en azından **int** türüne dönüştürültüğü için **short** sabit kavramının işlevsel bir anlamı yoktur. Bu nedenle ANSI notlarında **short** sabit tanımlamasına yer verilmemiştir.

Program içerisinde kullanımına ilişkin bir örnek vermek istiyoruz:

```
...
        {
                short a;

                a = 500;
                ....
        }
...
```

6.5 long SABİTLERİ

long sabitleri iki biçimde ifade edilir:

1) Sayının sonuna **L** ya da **l** yazılarak.

Bu durumda derleyiciler sayıyı **long** sabiti olarak yorumlarlar. Örneğin:

```
-1399L
10L
0L
400000l
...
```

Sayılarının hepsi **long** sabitidir.

2) **int** sınırlarını aşan **long** sınırları içerisinde kalan her tamsayı, derleyici tarafından doğrudan **long** sabiti olarak ele alınır. Tabi bu biçim, *DOS*'ta olduğu gibi **int** ve **long** türlerinin birbirlerinden farklı uzunlukta olduğu sistemler için anlamlıdır.

```
68000
-100000
-82500
13800000
```

sayıları *DOS*'ta **int** sınırlarını aştığı için derleyiciler tarafından doğrudan **long** sabit olarak ele alınırlar. Oysa:

```
12000000000
-42500000000
```

sayıları **long** sınırlarını da aştığından **long** sabit olarak ele alınmazlar.

Program içerisinde kullanımına ilişkin bir örnek verelim:

```
...
    {
            long int x;

            x = 100L;
            ...
    }
...
```

6.6 char SABİTLERİ

char sabitleri C'de üç biçimde bulunur.

1) İstenilen bir karakterin tek tırnak (single quote) içerisinde yazılmasıyla elde edilen char sabitleri:

Örneğin:

```
'a'
'X'
'q'
...
```

birer **char** sabitidir.

C'de tek tırnak içerisindeki karakter sabiti, aslında o karakterin karakter tablosundaki (ASCII tablosundaki) sıra numarasını gösteren bir sayıdır.

```
...
{
    char ch;

    ch = 'a';
    ...
}
...
```

Yukarıdaki örnekte aslında ch isimli karakter değişkenine a karakterinin *ASCII* karşılığı olan 97 sayısı aktarılmaktadır. Tek tırnak içerisindeki karakter sabitlerini görünce, aslında onların birer sayı olduğunu düşünmelisiniz. Çünkü bellekte karakter diye bir bilgi yoktur; herşey 2'lik sistemdeki 1'ler ve 0'lardan oluşan sayılardır. (2. Bölümden anımsayınız.) Bu sayıların kim tarafından ve nasıl yorumlanacağı önemlidir!

```
ch ='a';
```

ifadesinde ch değişkenine aslında 97 sayısı atanıyor. Siz şimdi ch değişkeninin içindeki sayıyı ekrana karakter olarak yazdırmak isterseniz a karakterini, 10'luk sistemde sayı olarak yazdırmak isterseniz o zaman da 97 sayısını görürsünüz.

```
ch = 'a' + 1;
```

Bu durumda da ch değişkenine 97+1 yani 98 atanmaktadır. Bu sayıya da *ASCII* tablosundaki b karakteri karşılık gelir.

2) Önceden tanımlanmış ters bölü karakter sabitleri : Ekrana basılamayan (none printable) karakterlerin sabit biçimine getirilememesi yukarıdaki yöntemin bir olumsuzluğudur. Örneğin, çan karakteri (biip sesi) 7 numaralı *ASCII* karakteridir, ancak ekrana basılamaz. İşte tek tırnak içerisinde bir ters bölü (back slash) işaretinden sonra bazı karakterler çok kullanılan ancak basılamayan karakter sabitlerinin yerini tutar.

Aşağıda bunların listesini görüyorsunuz:

'\a'	Çan sesi; 7 numaralı *ASCII* karakteri.
'\b'	Geri boşluk (backspace); 8 numaralı *ASCII* karakteri.
'\f'	Sayfa ileri (form feed); 12 numaralı *ASCII* karakteri
'\n'	Aşağı satır (new line); 10 numaralı *ASCII* karakteri.
'\r'	Satır başı karakteri (carriage return); 13 numaralı *ASCII* karakteri.
'\t'	Tab karakteri (tab); 9 numaralı *ASCII* karakteri.
'\v'	Düşey tab karakteri (vertical tab); 11 numaralı *ASCII* karakteri.
'\\'	Ters bölü karakteri.
'\"'	Çift tırnak karakteri.
'\0'	0 numaralı ASCII karakteri; (NULL karakter)

Progam içerisinde kullanımına ilişkin bir örnek:

```
...
    {
        char ch;

        ch = '\a';
        ...
    }
...
```

3) 16'lık (Hexadecimal) ya da 8'lik (Octal) sistemlerde tanımlanmış karakter sabitleri: İstediğimiz bir karakteri onun 16'lık ya da 8'lik sistemdeki *ASCII* numa-

rasıyla belirtebiliriz. (*ASCII* diyoruz ancak başka bir tablo da söz konusu olabilir.)

16'lık sistemde: '\xhh' biçimindedir. Tek tırnak içinde bir ters bölü ve sonra x ya da X sonra 16'lık sistemde yazılmış bir sayıdan oluşur.

Örneğin:

'\x41'	41H numaralı *ASCII* karakteri.
'\xff'	FFH numaralı 2 karakteri
'\x1c'	1C numaralı *ASCII* karakteri.

8'lik sistemde: '\0ooo' biçimindedir. Tek tırnak içerisinde bir ters bölü ve sonra 0 (sıfır), daha sonra da 8'lik sistemde bir sayı.

Örneğin:

'\012'	10 numaralı ASCII karakteri
'\016'	14 numaralı ASCII karakteri
'\0123'	66 numaralı ASCII karakteri

...

Program içinde kullanımına bir örnek:

```
...
    {
            char a, b;

            a = '\xff';
            b = '\0123';
            ...
    }
...
```

7numaralı *ASCII* karakteri olan çan karakterini sabit olarak 3 biçimde de yazabiliriz:

```
'\x7'
'\07'
'\a'
...
```

Acaba hangi yazım biçimi daha iyidir? Burada '\a' biçimi hem daha taşınabilir (portable) hem de daha okunabilir (readable) olduğu için tercih edilmelidir. '\a'

biçimi daha taşınabilirdir; çünkü başka karakter tablolarında çan karakterinin numarası farklı olsa da bu farklılıktan etkilenmez. Daha okunabilirdir; çünkü bu kodu gören kişi çan karakterinin 7 numaralı karakter olduğunu bilmek zorunda kalmaz.

6.7 float SABİTLERİ

Bu tür sabitler tipik olarak **float** türünden değişkenlere atanan sabitlerdir. Nokta içeren ve sonuna sonuna f ya F harfi getirilmiş sayılar **float** sabit olarak ele alınırlar. Örneğin:

```
1.31f
-34.2F
10.3F
-2.f
...
```

float sabitler üstel biçimde de ifade edilebilirler:

Örneğin:

```
2.3E+04f
1.8E-08F
-3.24e04f
...
```

Burada E ya da e 10'nun kuvveti anlamına gelmektedir.

```
1.34E-02F ile  0.0134F
-1.2E+02F ile -120.0F
```

aynı sabitlerdir.

Program içerisinde kullanımına bir örnek:

```
...
    {
            float a, b, c;

            a = 1.2f;
            b = 3.f;
            c = 1.234E-02F;
    }
...
```

6.8 double SABİTLERİ

Sonuna f ya da F eki almamış sabitler ile **float** duyarlılığını aşmış sabitler **double** sabit olarak değerlendirilirler.

Örneğin:

-12.3	**double** sabit
1.2E+02f	**float** sabit
-1.2E+200f	**float** duyarlılığını aştığı için **double** sabit.

```
...
    {
            double a, b, c;

            a = 2.3;
            b = 3.3;
            c = 2.4E-200;
            ...
    }
...
```

double sabitlerin de üstel biçimde gösterilebildiğine dikkat ediniz.

6.9 long double SABİTLERİ

Bu türden sabitler noktalı ya da üstel biçimdeki sayıların sonuna l ya da L getirilerek elde edilir. Örneğin:

```
1.34L
-3.45E+03L
10.2L
...
```

sayıları birer **long double** sabitlerdir.

6.10 İŞARETSİZ TÜRLERE İLİŞKİN SABİTLER

İşaretsiz türlerin sabitleri onların işaretli biçimlerinin sonuna **u** ya da **U** getirilerek elde edilirler.

-13760	**int** sabiti
13760U	**unsigned int** sabiti
1200L	**long** sabiti
1200LU	**unsigned long** sabiti

...

Bu durumda **unsigned int** sabitlerinin sonuna yalnızca **u** ya da **U**, **unsigned long** sabitlerinin sonuna ise **L** ya da **l** ile birlikte **u** ya da **U** yazılması gerekir. (**U** ve **L**'ler küçük ya da büyük harf olabilir. Sırası önemli değil)

Örneğin:

```
123UL
123LU
123Lu
...
```

yukarıdaki sabitlerin hepsi **unsigned long** sabitlerdir.

6.11 TAMSAYI SABİTLERİNİN 16'LIK ve 8'LİK SİSTEMLERDE GÖSTERİMİ

Tamsayı sabitleri (char, int, short int, long int) 10'luk sistemin yanısıra 16'lık ve 8'lik sistemlerde de yazılabilirler.

16'lık ve 8'lik sistemlerde yazılmış sayılar için aynı sabit kuralları geçerlidir. Çünkü bir sayıyı 16'lık ya da 8'lik sistemde yazmakla onun yalnızca görünümünü değiştirmiş oluruz. Sabit türlerinin, gösterim biçimiyle değil nicelikle ilişkili olduğunu unutmayınız.

16'lık sistemdeki gösterim:

`0XHH...` biçimindedir. Önce `0` (sıfır) daha sonra `x` ya da `X` ve daha sonra `HEX` basamaklar. Örneğin:

`0X12` sayısı 10'luk sistemde 18'e karşılık gelir. Bu bir **int** sabitidir değil mi?

Aynı biçimde:

`0x12L` onundaki **L** harfinden dolayı bir **long** sabitidir.

`0X1C205470` **long** sabitidir. Çünkü bu sayı 8 HEX basamağa sahiptir; **signed int** sınırını aşmıştır.

`0X1934U` **unsigned int** sabitidir.

`0x1c57LU` **unsigned long** sabitidir.

...

8'lik sistem 16'lık sisteme göre çok daha seyrek kullanılır. 8'lik sistemdeki gösterim:

0000... biçimindedir. 8'lik sistemde yazmak için yalnızca **0** (sıfır) ile başlamak yeterlidir.

Örneğin:

01234	**int** sabiti
0567L	**long** sabiti
0777U	**unsigned int** sabiti
04523LU	**unsigned long int** sabiti

...

6.12 TANIMLAMA SIRASINDA DEĞİŞKENLERE İLKDEĞER VERİLMESİ (INITIALIZATION)

Değişkenlere tanımlama sırasında ilkdeğer verebiliriz. Bunun için **atama operatörü** (=) kullanılır.

Örneğin:

```
...
{
      int a = 100, b = 200;
      char ch = 'A';
      float f  = 1.2f;
      ...
}
...
```

İlkdeğer verilmesi (initialization) genellikle sabitler yoluyla olsa da özel durumlar dışında böyle bir zorunluluk yoktur.

```
...
{
      int a = 100;
      int b = a;
      ...
}
...
```

6.13 printf FONKSİYONU ÜZERİNE KISACA...

C'de ilerleyebilmek için artık değişkenlerin içindeki değerleri ekrana yazdırmamız gerekiyor. Amacımız **printf** fonksiyonunun ayrıntılarını anlatmak değil; yalnızca işimize yarayacak kadar bilginin bu aşamada yeterli olduğunu düşünüyoruz.

printf birden fazla parametre alabilen bir fonksiyondur. İlk parametresi her za-

man bir string ifadesi olmak zorundadır. Bu string ifadesini oluşturan karakterler, % karakterleri dışında aynı biçimde ekrana yazılırlar. % karakterleri, diğer parametre değerlerini ekrana yazdırmak için kullanılmaktadır. Buna göre, % karakterleri ile diğer parametreler sırayla bire bir eşlenirler; parametre değerlerinin ne biçimde ekrana yazılacağı ise % karakterlerinden sonra gelen "*önceden tanımlanmış*" format karakterlerine bağlıdır.

Format karakterlerinin listesi aşağıda verilmiştir:

%d **char** ve **int** türlerini desimal sistemde
%c Karakter görüntüsü biçiminde
%x **int** türünü Hex sistemde (harfler küçük yazılır)
%X **int** türünü Hex sistemde (harfler büyük yazılır)
%o **int** türünü Octal sistemde
%u **unsigned int** türünü desimal sistemde
%ld **long** türünü desimal sistemde
%lx **long** türünü Hex sistemde
%f **float** ve **double** türlerini desimal sistemde
%e **float** ve **double** türlerini üstel biçimde
%s Stringleri karakter olarak

Eğer % karakterinden sonra gelen karakterler önceden tanımlanmış format karakterlerinden biri değilse **printf**, % karakteri ile birlikte bu karakteri aynı biçimde basar.

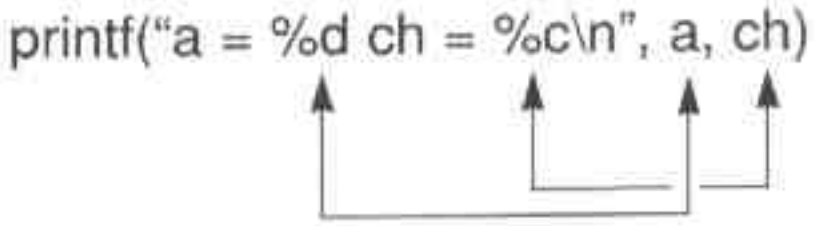
Örneğin:

```
#include <stdio.h>

main()
{
     int a;
     char ch;

     a = 100;
     ch = 'A';
     printf("a = %d ch = %c\n", a, ch);
}
```

Bu programın çıktısı:

a = 100 ch = A

biçimindedir. Bire bir eşlemenin nasıl yapıldığını inceleyiniz:



'\n' (new line) karakterinin "*imleç aşağı satırın başına geç*" anlamına geldiğini anımsayınız. Bu ifade ekrana basıldıktan sonra imleç (cursor) aşağı satırın başına geçecektir.

```
ch ='A';
```

ifadesiyle aslında **ch** değişkenine **65** karakteri atanmıştır. Eğer **printf** fonksiyonunun parametrelerini aşağıdaki gibi değiştirirsek:

```
printf("a = %d ch = %d ch = %c\n", a, ch, ch);
```

ekrandaki görüntü de aşağıdaki gibi olur:

```
a = 100 ch = 65 ch = A
```

Çünkü **%c**, **ch** değişkeninin içerisindeki değeri "karakter olarak", **%d** ise "tamsayı olarak yaz" anlamına gelmektedir. **printf** ile format karakterlerini kullanmadan ekrana yalnızca mesaj da yazdırabiliriz.

```
printf("Lütfen bir sayı giriniz:");
```

gibi...

SORAMADIKLARINIZ...

S1) Tek tırnak içindeki karakterler aslında bu karakterlerin *ASCII* tablosundaki sıra numarasını gösterdiğine göre:

```
char ch;
ch = 'A';
```

yerine;

```
ch = 65;  (A'nın ASCII karşılığı)
```

yazabilir miyiz?

C1) Bu iki ifade aslında aynı işlevlere sahiptir; ancak farkında olmamız gereken birşey var: **65** sayısı bir **int** sabitidir:

```
ch = 65;
```

yazmakla ch değişkenine bir tamsayı sabiti atamış olursunuz. C'de farklı türlerin birbirlerine atanması hata oluşturmaz; bu işlemde de bir hata yoktur. Fakat farklı türlerin birbirlerine atanmasının anlatıldığı 10. Bölüme gelene kadar böyle bir işlemden kaçınmalısınız.

S2) C'de tamsayıları 10'luk sistemin yanısıra 16'lık ve 8'lik sistemlerde de yazabiliyoruz. Acaba 2'lik sistem için de bir gösterim biçimi var mı? `printf` fonksiyonunu kullanarak bir değişkenin içeriğini 2'lik sistemde ekrana yazdırabilir miyiz?

C2) C'de tamsayı sabitlerini 2'lik sistemde belirtmenin bir yolu yok. Değişkenlerin içeriklerini `printf` fonksiyonu kullanarak da 2'lik sistemde ekrana yazdıramayız. Fakat, bunun çok çeşitli yolları olabilir. İlerideki bölümlerde somut bir uygulama bulacaksınız.

7

FONKSİYONLAR

Bu bölüm tümüyle fonksiyonlar konusuna ayrılmıştır. Fonksiyon kavramı, fonksiyonların geri dönüş değerleri, tanımlanmaları, çağrılmaları gibi temel konular bölüm içerisinde ayrıntılı bir biçimde ele alınmaktadır. Bölümün sonunda ise klavyeden karakter alan *C* fonksiyonları hakkında bilgiler bulacaksınız.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Fonksiyonların geri dönüş değerleri ne anlama gelmektedir ve neden kullanılır?
2) `return` anahtar sözcüğünün işlevi nedir?
3) `void` anahtar sözcüğünün işlevi nedir?
4) Fonksiyon parametreleri nasıl tanımlanır?
5) Bağlayıcıların (linkers) fonksiyonlarla ilgili işlevleri nelerdir?
6) Standart *C* fonksiyonları ne anlam ifade etmektedir?
7) Kütüphane nedir ve hangi amaçlarla kullanılır?

7.1 FONKSİYON NEDİR?

C'de altprogramlara fonksiyon (function) denir. Her fonksiyon faydalı birtakım işlemleri yerine getirmek için tasarlanır ve çağrılır. Fonksiyonların, onları çağıran fonksiyonlardan aldıkları **girdileri** ve yine onları çağıran fonksiyonlara gönderdikleri **çıktıları** vardır.

Fonksiyonların girdilerine parametreler ya da argümanlar diyoruz. Bir fonksiyon -icra süresince belli amaçları gerçekleştirmesinin yanı sıra- icrası sonunda bir değeri de çıktı olarak kendisini çağıran fonksiyona gönderebilmektedir.

7.2 FONKSİYONLARIN GERİ DÖNÜŞ DEĞERLERİ (Return Value)

Bir fonksiyonun çalışması sonunda onu çağıran fonksiyona dönüşünde gönderdiği değere **geri dönüş değeri (return value)** denir. Geri dönüş değeri, bir değişkene atanabilir ya da doğrudan aritmetik işlemlerde kullanılabilir. Örneğin:

```
x = fonk();
```

biçimindeki bir ifadede önce **fonk** isimli fonksiyon çalıştırılır; çalışma sonunda elde edilen geri dönüş değeri **x** değişkenine atanır. C'de fonksiyonlar, geri dönüş değerlerine sahip olmaları nedeniyle *"sağ taraf değeri (RValue)"* olarak kullanılabilirler, fakat nesne göstermediklerinden *sol taraf değeri (LValue)* olarak kullanılamazlar. Örneğin:

```
fonk() = a;
```

gibi bir ifade geçersizdir.

```
x = fonk() + b;
```

Bu örnekte de önce **fonk** isimli fonksiyon çalıştırılır, daha sonra geri dönüş değeri **b** ile toplanır ve en sonunda da **x** değişkenine atanır. Peki, fonksiyonların geri dönüş değerleri hangi amaçlarla kullanılıyor dersiniz?

1) Bazı fonksiyonlar bir tek değer elde etmek amacıyla tasarlanmışlardır. Elde ettikleri bu değerleri de kendilerini çağıran fonksiyonlara geri dönüş değeri biçiminde iletirler. Parametre olarak aldığı bir sayının karekökünü bulan **sqrt** fonksiyonunu buna örnek verebiliriz.

```
a = sqrt(x);
```

sqrt fonksiyonunun amacı **x** sayısının karekökünü bulmaktır. Sonuç geri dönüş değeri biçiminde **a** değişkenine atanmaktadır.

2) Bazı fonksiyonların geri dönüş değerleri yapılan işlemin başarısı hakkında bilgi verir. Yani bu tür fonksiyonların geri dönüş değerleri test amacıyla kullanılmaktadır. Geri dönüş değeri: *"İşlem başarılı olmuş mudur ya da neden başarısız olmuştur?..."* gibi sorulara yanıt verir.

Örneğin:

```
p = malloc(size);
```

ifadesiyle bellekte **size** byte uzunluğunda bir blok tahsis etmek isteyen programcı bu işlemin başarılı bir biçimde yerine getirilip getirilmediğini de test etmek

zorundadır. Hemen arkasından **p** değişkeninin aldığı değeri kontrol edecek ve işlemin başarısı hakkında bir karara varacaktır.

3) Kimi fonksiyonlar hem belli bir amacı gerçekleştirirler hem de buna ek olarak amaçlarını tamamlayan bir geri dönüş değeri üretirler.

Örneğin:

```
c = printf("Merhaba\n");
```

`printf` fonksiyonu ekrana merhaba yazısını yazmak için kullanılmıştır. Ancak ekrana yazdığı karakter sayısını da geri dönüş değeri olarak vermektedir.

4) Bazen geri dönüş değerlerine hiç ihtiyaç duyulmaz. Örneğin, yalnızca ekranı silme amacıyla tasarlanmış olan bir fonksiyonun herhangi bir geri dönüş değerine sahip olması gereksizdir.

```
clrscr();
```

`clrscr` fonksiyonu yalnızca ekranı siler; böyle bir fonksiyonun geri dönüş değerine gereksinimi yoktur.

Fonksiyonların geri dönüş değerlerinin de türleri söz konusudur:.

Fonksiyonların geri dönüş değerleri herhangi bir türden olabilir. Geri dönüş değerlerinin türleri fonksiyonların tanımlanması sırasında belirtilir.

7.3 FONKSİYONLARIN TANIMLANMASI

Kendi yazdığımız fonksiyonlar için **tanımlama (definition)** terimini kullanıyoruz. (Tanımlama teriminin nesne yaratan bir bildirim olduğunu anımsayınız. Fonksiyonlar da bellekte yer kaplarlar değil mi?) *C*'de fonksiyon tanımlama işleminin genel biçimi şöyledir.

```
[Geri Dönüş Değerinin Türü]  <Fonksiyonun İsmi>  ([Parametreler])
{
        .
        .
        .
}
```

Yukarıdaki gösterimde <.....> arasında belirtilen ifadeler zorunlu olarak bulunması gerekenleri; [...] arasındakiler de bulunması zorunlu olmayan, isteğe bağlı (optional) ifadeleri göstermektedir.

Tanımlanan fonksiyonlar en az bir blok içerirler.

Örneğin:

```
float fonk()
{
        .
        .
        .
}
```

Fonksiyonun ana bloğu

Buradan **fonk** isimli fonsiyonunun geri dönüş değerinin `float` olduğunu ve parametre almadığını görüyoruz.

Bir fonksiyonun parametresi ve/veya geri dönüş değeri olmayabilir.

Parametresiz fonksiyonlarda fonksiyon parantezleri içine ya hiç birşey yazılmaz ya da `void` anahtar sözcüğü yazılır.

Örneğin:

```
float a1(void)
{
        .
        .
        .
}
```

```
float a1()
{
        .
        .
        .
}
```

Yukarıdaki iki tanımlama biçimi de **a1** fonksiyonunun parametresiz olduğunu anlatmaktadır, dolayısıyla eşdeğerdir.

Geri dönüş değerine ihtiyaç duyulmadığı durumlarda da geri dönüş değerinin türü yerine yine `void` anahtar sözcüğü kullanılır.

Örneğin:

```
void x1(void)
{
        .
        .
        .
}
```

x1 fonksiyonu geri dönüş değerine de parametreye de sahip değildir.

Bu durumda, fonksiyon tanımlamalarında kullandığımız `void` anahtar sözcüğünün iki işlevi vardır:

1) Fonksiyon parantezlerinin içerisine yazılırsa fonksiyonun parametre almadığını belirtir.

2) Geri dönüş değerinin türü yerine yazılırsa geri dönüş değerine ihtiyaç duyulmadığı, dolayısıyla geri dönüş değerinin kullanılmayacağı anlamına gelir.

Fonksiyon tanımlarken geri dönüş değerinin türü yerine hiçbir şey yazılmazsa C derleyicileri geri dönüş değerinin türünün int olduğunu varsayarlar.

Örneğin:

```
y1 ()
{
        .
        .
        .
}
```

y1 fonksiyonunun parametresi yoktur; ancak geri dönüş değeri int türündendir. Bir de fonksiyon tanımlamasına ilişkin -zaten farketmişsinizdir- şu kuralı bilmeniz gerekir:

C'de fonksiyon içinde fonksiyon tanımlanamaz!

Fonksiyon içinde fonksiyon tanımlamak C öğrencilerinin en fazla yaptığı hatalardan birisidir.

```
float fonk1()
{
     ...
     int fonk2()
     {
             .
             .                  /* Hatalı tanımlama */
             .
     }
     ...
}
```

Yukarıdaki örnekte **fonk2**'nin **fonk1**'in içinde tanımlandığını görüyoruz; bu nedenle tanımlama işlemi geçerli değildir. İki fonksiyonun aşağıdaki örnekte olduğu gibi tamamen birbirinin dışında tanımlanması gerekirdi.

```
float fonk1()
{
     .
     .
     .
}
```

```
int fonk2()
{
      .
      .
      .
}
```

7.4 FONKSİYONLARIN ÇAĞRILMASI

C'de fonksiyon çağırma operatörü olarak ( ) kullanılmaktadır. Bir fonksiyon çağrıldığında programın akışı fonksiyonu çalıştırmak üzere bellekte fonksiyon kodunun bulunduğu bölgeye atlar; fonksiyonunun çalışma işlemi bittikten sonra da akış tekrar çağıran fonksiyonun kalınan yerinden devam eder.

```
#include<stdio.h>
main()                                  sample()
{                                       {
      ...                                   .
      sample();                             .
      ...                                   .
}                                       }
```

Fonksiyonlar ancak tanımlanmış fonksiyonların içerisinde çağrılabilirler. Aşağıdaki örneği inceleyiniz.

```
main()
{
      int a = 100, b = 200;

      printf("a = %d b = %d\n", a, b);
}

printf("merhaba\n");        /* Hata!.. Burada fonksiyon çağrılamaz */

fonk()
{
      int x = 10, y =  20;

      printf("x = %d y = %d\n", x, y );
}
```

Burada **printf**, **main** ve **fonk** isimli fonksiyonların içinde doğru biçimde çağrılmıştır. Ancak iki fonksiyonun arasında hiçbirinin içinde olmayacak biçimde çağrılmış olan:

```
printf("merhaba\n");
```

geçersizdir.

Çağıran fonksiyon ile çağrılan fonksiyonun her ikisi de aynı amaç kod içerisinde bulunmak zorunda değildir. Çağıran fonksiyon ile çağrılan fonksiyon farklı amaç kodlar içerisinde de bulunabilir (bir projeyi oluşturan amaç kodlara kısaca **modül** denir) . Çünkü derleme sırasında bir fonksiyonun çağrıldığını gören derleyici, **amaç kod içerisine (.obj)** yalnızca çağrılan fonksiyonun adını ve çağrılış biçimini (short, near, far, direct, indirect) yazmaktadır.

Çağıran fonksiyon ile çağrılan fonksiyon arasında bağlantı kurma işlemi, bağlama aşamasında, bağlayıcı program (linker) tarafından yapılır.

Bu nedenle tanımladığınız bir fonksiyonun içinde, olmayan bir fonksiyonu çağırsanız bile, derleme aşamasında bir hata oluşmaz. Hata, bağlama aşamasında, bağlayıcının çağrılan fonksiyonu bulamaması biçiminde ortaya çıkar. Bir deneme yapabilirsiniz. Aşağıdaki programı önce yalnızca derleyin, daha sonra ise bağlamaya çalışın.

```
#include <stdio.h>

main()
{
    xxxxx();
}
```

Ne görüyorsunuz? Hata kimin tarafından ve ne biçimde veriliyor?

Bundan böyle karşılaştığınız hataların derleme zamanına mı (compile time) yoksa bağlama zamanına mı (link time) ilişkin olduğunu tespit etmeye çalışmalısınız.

Önceki bölümde de anlatıldığı gibi, **main** fonksiyonu programın başlangıç noktasını belirtmektedir. Programın çalışmaya başladığı fonksiyon olması dışında **main** fonksiyonunun diğer fonksiyonlardan hiçbir farkı yoktur. **main** fonksiyonunun icrası bitince program da bitmiş olur. Ayrıca, bir C programının çalışabilmesi için mutlaka bir **main** fonksiyonuna sahip olması gerekir.

C programlarının çalışabilmesi için mutlaka main fonksiyonun bulunması gerekir. main fonksiyonu yoksa hata bağlama aşamasında, bağlayıcı program tarafından bildirilecektir.

7.5 STANDART C FONKSİYONLARI

Standart *C* fonksiyonları, *C* derleyicilerini yazanlar tarafından ilk elden yazılmış olan ve derleyici paketlerinin içerisinde zaten bulunan fonksiyonlardır. Bu fonksiyonların her sistemde olması "taşınabilirlik gereği" garanti altına alınmıştır. Standart *C* fonksiyonlarının **standart olmaları** dışında bizim yazdığımız fonksiyonlardan hiç farkları yoktur. Örneğin, önceki bölümde kısaca değindiğimiz **printf** bir standart *C* fonksiyonudur. Bir fonksiyonun derleyici paketinde bulunması onun standart *C* fonksiyonu olduğu anlamına gelmez. Gerçekten de derleyici paketlerinin yeni uyarlamaları programcıların işini kolaylaştırmak amacıyla çok sayıda standart olmayan fonksiyonlar da barındırmaktadır. Standart olmayan fonksiyonları kullanmak programcıların işini kolaylaştırsa da kaynak programın taşınabilirliğini azaltır.

Peki, standart *C* fonksiyonları nerede bulunuyorlar acaba? *C* öğrencilerinin çoğu onların başlık dosyalarının (uzantısı **.h** olan dosyalar) içinde bulunduğunu sanır. Oysa, standart *C* fonksiyonları kütüphanelerin içerisindedir.

7.6 KÜTÜPHANELER(Library)

Kütüphaneler object modüllerden (.obj), object modüller de derlenmiş fonksiyonlardan oluşur. *DOS*'ta kütüphane dosyalarının uzantısı **.LIB** (library), *UNIX*'te ise **.A** (archieve) biçimindedir. *WINDOWS* altında **.DLL** biçiminde dinamik kütüphaneler de bulunmaktadır.

Standart *C* fonksiyonları bağlama aşamasında, bağlayıcı tarafından çalışabilir kod içerisine (.EXE) yazılırlar. Kişisel bilgisayarlarda kullandığımız derleyicilerin tümleşik çevreli uyarlamalarında (entegrated environment version) bağlayıcılar, amaç kod içerisinde bulamadıkları fonksiyonları otomatik olarak önceden belirlenmiş olan standart kütüphanelerde de ararlar. Oysa komut satırlı uyarlamalarında (command line version) bağlayıcıların hangi kütüphanelere bakacağı komut satırında belirtilir.

7.7 return ANAHTAR SÖZCÜĞÜ

return anahtar sözcüğünün iki önemli işlevi vardır.

1) Fonksiyonların geri dönüş değerlerini oluşturur.

2) Fonksiyonları sonlandırır.

Hemen bir örnekle açıklayalım:

```
#include <stdio.h>

int fonk(void)
{
     int a = 100, b = 200;

     return a + b;       /* geri dönüş değeri a + b, yani 300 */
}

void main(void)
{
     int a;

     a = fonk();
     printf("a = %d\n", a);
}
```

fonk isimli fonksiyondaki return anahtar sözcüğü, hem fonksiyonu sonlandırır hem de geri dönüş değerini oluşturur.

return anahtar sözcüğünün kullanılması zorunlu değildir. **Eğer return anahtar sözcüğü yoksa fonksiyon, ana bloğu bitince kendiliğinden sonlanır.** return anahtar sözcüğünün fonksiyon bloğunun sonunda olması gibi bir zorunluluğun da olmadığını belirtelim.

Birkaç örnek daha vermek istiyoruz:

```
return (a +  b * c); /* parantez kullanabilirsiniz, size kalmış */
...
return 100;          /* değişkenle geri dönmek gerekmiyor */
...
return getkb();      /* önce getkb fonksiyonu çalıştırılır */
...
```

Son örnek biraz ilginç gelebilir size. Fonksiyonun geri dönüş değeri, başka bir fonksiyonun geri dönüş değeridir. Bu durumda önce ilgili fonksiyon getkb() fonksiyonunun geri dönüş değeri ile geri dönmektedir.

Geri dönüş değerine sahip olmayan fonksiyonlarda **return** yalnız başına kullanılmalıdır.

```
void x1()
{
      ...
      return;          /* fonksiyonun geri dönüş değeri yok */
      ...
}
```

return bu örnekte yalnızca fonksiyonu sonlandırmak için kullanılmıştır.

7.8 FONKSİYON PARAMETRELERİNİN TANIMLANMASI

Parametreler (ya da argümanlar) fonksiyonların kendilerini çağıran fonksiyonlardan aldıkları girdilerdir. Bir fonksiyonun sahip olduğu parametre sayısı, bunların isimleri ve türleri gibi bilgiler, fonksiyonun tanımlanması sırasında derleyiciye bildirilir.

C'de parametrelerin tanımlanmasında kullanılan 2 yöntem vardır. Bunlardan birine **eski biçim** (old style), diğerine ise **yeni biçim** (new style) diyoruz. Eski biçim -isminden de anlaşılacağı gibi- *C*'nin *ANSI* tarafından standardize edildiği 83 yılına kadar yoğun olarak kullanılıyordu. Şimdilerde ise programcılar hep yeni biçimi tercih ediyorlar.

1) Eski Biçim (Old style)

Bu biçimde, önce fonksiyon parantezinin içerisine parametre değişkenlerinin isimleri aralarına **virgül** (,) konularak yazılır; daha sonra fonksiyonun ana bloğundan önce bu değişkenlerin bildirimleri yapılır.

[Geri Dönüş Değerinin Türü] <Fonksiyon İsmi> ([Parametre Değişkenleri])

```
      <Tür> <Parametre Değişkeni>
      <Tür> <Parametre Değişkeni>
      ...
{
      .
```

```
        .
        .
}
```

Örneğin:

```
int topla (a, b)
int a, b;
{
     return a + b;
}
```

Burada **topla** fonksiyonu **a** ve **b** isimli iki tane **int** türünden parametre almıştır.

```
float px(x, y)
float x;
int y;
{
        .
        .
        .
}
```

Bu örnekte ise **px** fonksiyonu 2 parametre almaktadır. **x** parametre değişkeni **float**, **y** parametre değişkeni ise **int** türündendir.

2) Yeni Biçim (Modern style)

Yeni biçim hem yer bakımından daha ekonomiktir hem de daha okunabilirdir. Biz de kitabımızdaki bütün örneklerde hep yeni biçimi kullanacağız. Bu biçimde parametre değişkenlerinin türü ve isimleri fonksiyon parantezinden sonra birlikte yazılırlar.

[Geri Dönüş Değerinin Türü] <Fonksiyon İsmi> ([<Tür> <para1>, Tür <para2>,...])

```
{
        .
        .
        .
}
```

Örneğin:

```
int topla (int a, int b)
{
        return a + b;
}

float px (float x,  int y)
{
        .
        .
        .
}
```

Parametre değişkenleri aynı türden olsalar bile, tür belirten anahtar sözcük hepsi için yazılmak zorundadır. Örneğin:

```
int fonk(int a, b)                    /* Hata */
{
     ...
}
```

b değişkeninin bildirimi hatalıdır. Bildirim aşağıdaki biçimde olmalıdır:

```
int fonk(int a, int b)
{
     ...
}
```

7.9 KLAVYEDEN KARAKTER ALAN C FONKSİYONLARI

Sistemlerin çoğunda klavyeden karakter alan üç tür C fonksiyonu vardır. Bu fonksiyonların biri tam olarak standarttır; ancak diğer ikisi, sistemlerin çoğunda bulunmasına karşın tam anlamıyla standart değildir.

1) int getchar(void)

getchar bir standart C fonksiyonudur. Geri dönüş değeri klavyeden alınan karakterin *ASCII* tablosundaki numarasını gösteren **int** türünden bir sayıdır. **getchar** fonksiyonu ENTER tuşuna ihtiyaç duyar.

```
#include <stdio.h>
main()
{
        char ch;

        ch = getchar();
        printf("Karakter olarak ch = %c, ASCII numarası ch = %d\n", ch, ch);
}
```

Bu örnekte klavyeden alınan bir tuşun karakter ve sayısal karşılıkları ekrana yazdırılıyor. **getchar** fonksiyonunun geri dönüş değerinin basılan tuşun *ASCII* karşılığını gösteren bir tamsayı olduğunu unutmayınız. Bu fonksiyon tam olarak standardize edilmiştir; dolayısıyla standart C fonksiyonudur.

Not: getchar derleyicilerin çoğunda **stdio.h** dosyasında bir makro olarak tanımlanmıştır. Makrolar hakkında ilerideki bölümlerde ayrıntılı bilgiler bulacaksınız.

2) int getch(void)

Tıpkı **getchar** gibi bu fonksiyon da basılan tuşun *ASCII* karakter numarasıyla geri döner. **getchar** fonksiyonundan tek farkı *ENTER* tuşuna gereksinim duymamasıdır. Yani tuşa basar basmaz işlem görür; bastığımız tuş ise ekranda görünmez. Bu fonksiyon tam olarak standardize edilmemiştir. Dolayısıyla standart C fonksiyonudur, diyemiyoruz.

Yukarıdaki örnek programı **getchar** yerine **getch** yazarak tekrar çalıştırmaya deneyiniz. **getch** fonksiyonu özellikle, *"tuş bekleme ya da onaylama"* amacıyla kullanılmaktadır.

Örneğin:

```
...
printf("Devam etmek için bir tuşa basınız...\n");
getch();
...
```

Burada basılan tuşun programcı tarafından bir önemi yoktur. Bu nedenle geri dönüş değeri de kullanılmamıştır.

3) **int getche(void)**

Bu fonksiyon da basılan tuşun ASCII numarasıyla geri döner ve ENTER tuşuna gereksinim duymaz. Ancak, `getch` fonksiyonundan tek farkı basılan tuşun ekranda görünmesidir.

```
...
printf("(E)vet mi (H)ayır mı?.. ");
ch = getche();
...
```

(`getche`, *İngilizce* `get-char-echo` sözcüklerinden gelmektedir)

7.10 EKRANA KARAKTER YAZAN C FONKSİYONLARI

Sistemlerde `-printf` fonksiyonunu saymazsak- ekrana karakter yazan iki fonksiyona rastlanır: `putchar` ve `putch`.

1) **int putchar (int ch)**

`putchar` standart bir *C* fonksiyonudur; dolayısıyla bütün sistemlerde bulunmak zorundadır. Parametresi olan karakteri ekranda imlecin (cursor) bulunduğu yere yazar. Örneğin:

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

burada `getchar` ile klavyeden alınan karakter `putchar` ile tekrar ekrana yazdırılıyor. `putchar` fonksiyonu:

```
printf("%c", ch)
```

ile aynı işleve sahiptir. `putchar` fonksiyonu ile '\n' karakterini yazdırdığımızda, `printf` fonksiyonunda olduğu gibi, imleç (cursor) sonraki satırın başına geçer. `putchar` fonksiyonu ekrana yazılan karakterin ASCII karşılığı ile geri dönmektedir.

Not: `putchar` fonksiyonu da derleyicilerin çoğunda , `stdio.h` dosyasında bir makro olarak tanımlanmıştır. Makrolar konusu 30. Bölümde ayrıntılı bir biçimde ele alınmaktadır

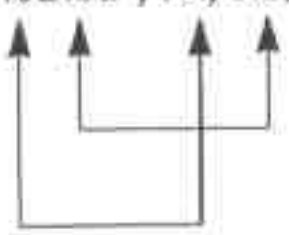
2) int putch (int ch)

`putch` bir standart *C* fonksiyonu değildir; dolayısıyla sistemlerin tümünde bulunmayabilir. Bu fonksiyonun `putchar` fonksiyonundan tek farkı "`\ n`" karakterinin yazdırılması sırasında ortaya çıkar. `putch`, '\n' karakterine karşılık yalnızca *LF (ASCII:10)* karakterini yazmaktadır. Bu durum imlecin (cursor) bulunduğu kolonu değiştirmeksizin aşağıdaki satıra geçmesine yol açar.

7.11 scanf FONKSİYONU ÜZERİNE KISACA...

scanf fonksiyonu klavyeden her türlü bilgiyi almak için kullanılan standart bir *C* fonksiyonudur. Biz **scanf** i teknik ve ayrıntılı bir biçimde değil yalın ve yüzeysel bir biçimde ele alacağız. **scanf** fonksiyonunun ilk parametresi **printf** de olduğu gibi bir string ifadesidir. Yalnız **printf** bu string ifadesi içerisindeki format bilgisini ekrana yazım biçimini belirlemek amacıyla kullanırken, **scanf** klavyeden alınacak bilginin türünü belirlemek amacıyla kullanmaktadır. **scanf** fonksiyonunun format kısmında format karakterlerinden başka bir şey olmamalıdır. **scanf** format karakterlerinin dışında buraya yazılanları ekrana basmaz; bu karakterler tamamen başka anlama gelir. Bu nedenle fonksiyonun nasıl çalıştığını ayrıntılarıyla öğrenmeden bu bölgeye yalnızca format karakterlerinden başka birşey koymayınız. Unutmayınız: Buraya konulacak bir boşluk bile farklı anlama gelmektedir.

```
int a, b;

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



Yukarıdaki örnekte **a** ve **b** sayıları için 10'luk sistemde klavyeden giriş yapılmaktadır. Giriş arasına istenildiği kadar boşluk karakteri konulabilir. Yani ilk sayıyı a için girdikten sonra ikinci sayıyı **SPACE**, **TAB** ya da **ENTER** tuşuna bastıktan sonra girebilirsiniz. Örneğin:

100 200

biçiminde bir giriş geçerli olabileceği gibi ;

100
200

biçiminde bir giriş de geçerlidir. **scanf** fonksiyonunun diğer parametrelerinin & operatörü ile kullanıldığına dikkat ediniz. (& bir gösterici operatörüdür ve 18. Bölümde ayrıntılı bir biçimde açıklanmaktadır.) Girişlerde bu operatörü değişkenin önüne koymayı unutmayınız. **printf** de belirtilen format karakterlerinin hepsini **scanf** de de kullanabilirsiniz.

Format karakterlerinin listesi aşağıda verilmiştir:

%d **int** türünü desimal sistemde görmek için
%c Karakter biçiminde girmek için
%x **int** türünü Hex sistemde (harfler küçük yazılır) girmek için
%X **int** türünü Hex sistemde (harfler büçük yazılır) girmek için
%o **int** türünü Octal sistemde girmek için
%u **unsigned int** türünü desimal sistemde girmek için
%ld **long** türünü desimal sistemde girmek için
%lx **long** türünü Hex sistemde girmek için
%f **float** türünü desimal sistemde girmek için
%lf **double** türünü desimal sistemde girmek için
%s Stringleri girmek için

printf fonksiyonunda %f hem **float** hem de **double** türünü yazdırmak için kullanıyordu. Oysa **scanf** fonksiyonunda **float** %f , **double** ise %lf formatıyla girilmektedir. Aşağıdaki örneği inceleyiniz:

```
int a;
double b;
...
scanf("%d%lf", &a, &b);
```

scanf fonksiyonunun yalnızca giriş için kullanıldığını ekrana yazmak için **printf**'i kullanmanız gerektiğini anımsatalım.

int no;

printf("Numara giriniz:");

scanf("%d", &a);

gibi...

Not: scanf fonksiyonu EK-B'de ayrıntılı bir biçimde açıklanmıştır.

SORAMADIKLARINIZ...

S1) Fonksiyonların geri dönüş değerleri mutlaka kullanılmak zorunda mıdır?

C1) Fonksiyonların geri dönüş değerlerini kullanmak zorunda değilsiniz. Bunu, yukarıda **getch** fonksiyonuyla ilgili verilen örnekte de gördünüz. Örneğin, aslında **printf** fonksiyonunun da bir geri dönüş değeri vardır; ancak nadiren kullanılır.

S2) Fonksiyonların geri dönüş değerleri bir tane midir, birden fazla olamaz mı?

C2) Fonksiyonların yalnız bir tane geri dönüş değeri olabilir. Fakat bu durum, fonksiyonların kendilerini çağıran fonksiyonlara birden fazla değer iletemeyeceği anlamına gelmez. **Göstericiler (pointers)** ve **yapılar (structures)** konularında bunun nasıl yapıldığını anlayacaksınız.

S3) Fonksiyonların geri dönüş değerleri kullanılmayabiliyor. Parametresiz fonksiyonların tanımlanmasında **void** anahtar sözcüğünün kullanılması da zorunlu olmadığına göre; peki, o halde **void** anahtar sözcüğüne neden ihtiyaç duyulmuştur?

C3) **void** anahtar sözcüğü *C*'ye sonradan girmiştir. Gerçekten de **standart C (ANSI C)** içinde **void** anahtar sözcüğünün kullanılmasının zorunlu olduğu bir durum yoktur. Ancak bu anahtar sözcüğün kullanımı **okunabilirliği (readability)** artırmaktadır. Çünkü fonksiyon tanımlamasında geri dönüş değerinin türü yerine **void** anahtar sözcüğünü gören birisi, geri dönüş değerine ihtiyaç duyulmadığını hemen anlayabilir.

Örneğin:

```
void f1()
{
    .
    .
    .
}
```

```
f1()
{
    .
    .
    .
}
```

f1 fonksiyonunun geri dönüş değerinin kullanılmadığını düşünelim. Bu durumda **void** ile tanımlanmış ilk biçim bize daha çok şey anlatmaktadır; dolayısıyla daha okunabilirdir.

void anahtar sözcüğü derleyicilerin tür ve parametre kontrolünde de etkili olur. Aşağıdaki örneği inceleyelim:

```
void f1 (void)
{
     .
     .
     .
}

int f2(void)
{
     int a, b;
     ...
     a = f1();              /* Hata, f1 void fonksiyon */
     f1(b);                 /* Hata, f1'in parametresi yok */
}
```

Burada derleyici **f2** fonksiyonunda iki hata (error) bildirecektir. Çünkü **f1** fonksiyonunun geri dönüş değeri ve parametresi **void** olduğu halde başka bir değişkene atanmış ve parametreli olarak çağrılmıştır.

Oysa **f1** fonksiyonu

```
f1()
{
     .
     .
     .
}
```

biçiminde tanımlansaydı, derleme sırasında bir hata oluşmazdı. Yani **void** anahtar sözcüğü bir kontrol mekanizmasını da sağladığı için programcının yapacağı birtakım hataları derleme zamanında önlemektedir. Şimdilik bu kadar bilgiyi yeterli görüyoruz. Çünkü konu fonksiyon prototiplerinin anlatıldığı 13. Bölümde daha ayrıntılı biçimde ele alınmaktadır.

S4) *C* programları `main` fonksiyonundan çalışmaya başlıyor ve `main` fonksiyonu bittiğinde program da bitiyor. Buna göre, `main` fonksiyonunun geri dönüş değeri olabilir mi? Olabilirse bu geri dönüş değeri kim tarafından kullanılmaktadır?

C4) `main` fonksiyonunun da geri dönüş değeri olabilir. `main` fonksiyonunun geri dönüş değeri programın çalışması bittikten sonra işletim sistemine iletilmektedir. Bir programın başka programları çalıştırdığı uygulamalar dışında böyle bir geri dönüş değerine ihtiyaç duyulmaz. Bu nedenle programcıların çoğu tanımlama sırasında `main` sözcüğünün soluna hiçbir şey yazmazlar.

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

Bu durumda geri dönüş değerinin tamsayı (int) kabul edildiğini anımsayınız. Derleyicilerin yeni uyarlamaları, geri dönüş değerlerine sahip oldukları halde belli bir değerle geri dönmeyen fonksiyonlar için `uyarı` (**warning**) mesajı verebilirler. Bu nedenle programcıların bir kısmı `main` fonksiyonunu `void` olarak tanımlarlar.

```
void main()
{
        .
        .
        .
}
```

S5) Çağıran fonksiyon ile çağrılan fonksiyon arasındaki bağlantıyı neden derleyici kurmuyor da bu iş bağlayıcı tarafından yapılıyor?

C5) Çağıran fonksiyon ile çağrılan fonksiyon aynı amaç dosyada olmak zorunda değildir. *DOS, UNIX* ve *WINDOWS* altında birden fazla `amaç dosya (.obj)` tek bir `çalışabilir program (.exe)` elde etmek amacıyla birlikte bağlanabilir.

S6) Tanımlaması yapılmamış bir fonksiyon çağrıldığında hata neden bağlama aşamasında, bağlayıcı tarafından tespit ediliyor?

C6) Çünkü derleyicinin görevi fonksiyonun var olup olmadığını araştırmak değildir. Bu işi bağlayıcı yapmaktadır. 4. soruda da belirttiğimiz gibi bir'den fazla kaynak kod ayrı ayrı derlenebilir.

NESNELERİN FAALİYET ALANLARI VE ÖMÜRLERİ

8

Nesnelerin birtakım özellikleri önceki bölümlerde incelenmişti. Şimdi sıra *"faaliyet alanı (scope)* ve *ömür (duration)"* özelliklerine geldi. Faaliyet alanı, bir nesnenin tanınabildiği *"program aralığını"*, ömür ise faaliyet gösterdiği *"zaman aralığını"* belirtmektedir. Bu iki özelliğin de *C* öğrencileri tarafından çok iyi anlaşılması gerekir. Fakat maalesef *C* öğrencilerinin çoğu bu konuları yeterli olmayan bilgilerle geçerler; bunun sonucunda da sonraki konuları anlamakta zorlanırlar.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) *C*'de kaç türlü faaliyet alanı vardır?

2) Faaliyet alanlarına göre değişkenler kaç gruba ayrılırlar?

3) Aynı isimli değişkenlerin tanımlanmasına ilişkin kural nasıldır?

4) Aynı isimli fakat farklı faaliyet alanlarına sahip değişkenlere erişme kuralı nasıldır?

5) *C*'de parametre aktarım kuralı nasıldır?

6) Ömürleri bakımından değişkenler kaç gruba ayrılırlar?

7) İlkdeğer verilmemiş değişkenlerin içerisindeki değerler hakkında ne söylenebilir?

8.1 FAALİYET ALANI (Scope)

Faaliyet alanı, bir değişkenin ömrünü sürdürdüğü ve tanınabildiği program aralığıdır. Değişkenlerin faaliyet alanları, onların program içerisindeki **tanımlanma yerleriyle** ilişkilidir. Faaliyet alanlarını üç grupta toplayabiliriz:

1) Blok Faaliyet Alanı (Block scope): Yalnızca bir blok içerisinde tanınma aralığıdır. (Küme parantezlerinin arasındaki bölgeye blok denildiğini anımsayınız.)

2) Fonksiyon Faaliyet Alanı (Function scope): Yalnızca bir fonksiyonun her yerinde tanınma aralığıdır.

3) Dosya Faaliyet Alanı (File scope): Tüm dosya içinde, yani fonksiyonların hepsi içerisinde tanınma aralığıdır.

Değişkenleri de faaliyet alanlarına göre 3 bölüme ayırabiliriz.

1) Yerel değişkenler (local variables)

2) Global değişkenler (gloabal variables)

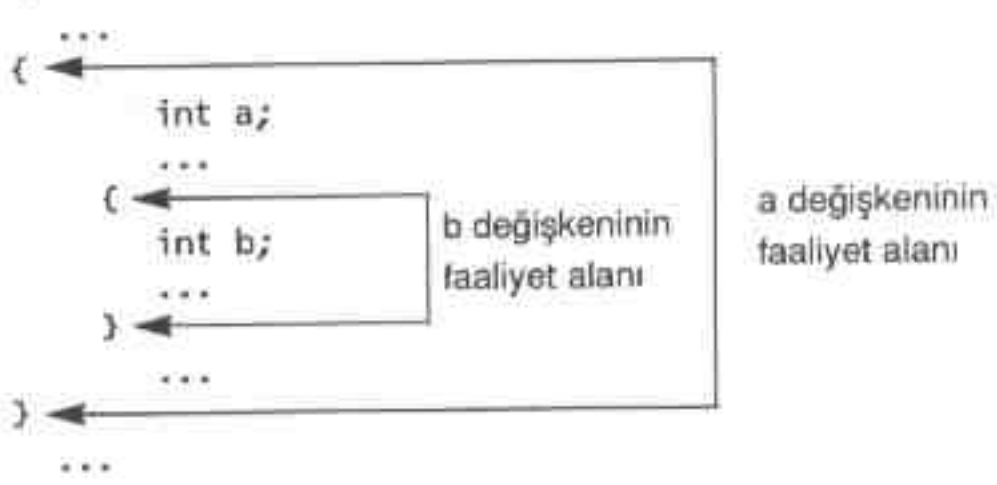
3) Parametre değişkenleri (formal parameters)

Şimdi bunları tek tek inceleyeceğiz:

8.2 YEREL DEĞİŞKENLER (Local Variables)

Blokların başlarında tanımlanan değişkenlere yerel değişkenler denir. Yerel değişkenler blok faaliyet alanı kuralına uyarlar; yani yalnızca tanımlandıkları blok içinde tanınabilir ve erişilebilirler.

Örneğin:



Burada **a** ve **b**'nin her ikisi de yerel değişkendir. Çünkü blok başlarında tanımlanmışlardır. **a** değişkeninin faaliyet alanının **b** değişkeninin faaliyet alanından daha büyük olduğunu görüyorsunuz. **a** değikeninin tanımlandığı blok, **b** değişkeninin tanımlandığı bloğu kapsıyor, değil mi? O halde şöyle diyebiliriz:

İki değişkenin aynı faaliyet alanı grubuna ait olması (blok, fonksiyon ya da dosya) faaliyet alanlarının tamamen birbiriyle aynı olduğu anlamına gelmez. Blok, fonksiyon ve dosya faaliyet alanları bir genellemedir.

Farklı bloklarda tanımlanan yerel değişkenler farklı faaliyet alanına sahiptirler. Aşağıdaki örneği inceleyiniz:

```
#include <stdio.h>

void main(void)
{
     int a = 10;

     /* Bu bölgede yalnızca a tanınabilir */

     {
          int b = 20;
          printf("a = %d b = %d\n", a, b);

          /* Bu bölgede hem a hem b tanınabilir */

     }

     b = 30;        /* Hata! b'nin faaliyet alanı bitti */

     /* Bu bölgede yalnıca a tanınabilir, b tanınamaz */
}
```

`b = 30` ifadesi b'nin tanımlandığı blok dışında olduğu için geçersizdir. Çünkü `b`, yalnızca tanımlandığı blok içerisinde faaliyet gösterebilir. Bu basit örneği deneyiniz ve derleyiciniz tarafından bildirilen hata mesajını inceleyiniz.

Bir çatışma durumunu çözmemiz gerekiyor: Acaba farklı bloklarda aynı isimli değişkenler tanımlanabilir mi?

Örneğin:

```
#include <stdio.h>

void main(void)
{
     int a = 10;

     printf("a (dış blok) = %d\n", a);  /* a = 10 olmalı */
     {
          int a = 20;

          printf("a (iç blok) = %d\n", a);    /* Hangi a? */
     }
     printf("a (dış blok) = %d\n", a);       /* a = 10 olmalı */
     }
```

C'de farklı faaliyet alanlarına sahip aynı isimli yerel değişkenler tanımlanabilir. Örneğimizdeki farklı bloklarda tanımlanmış olan iki a değişkeni birbiriyle karışmaz. Çünkü derleyiciler isimleri aynı olmasına karşın bu iki a değişkenini bellekte farklı yerlerde tutarlar. İkisi de yerel değişken olmasına karşın dış blok içerisindeki a'nın daha geniş faaliyet alanına sahip olduğuna dikkat ediniz.

Yukarıdaki örneği tekrar dikkatle inceleyiniz. İç blok içerisinde her iki a da tanınabilir durumdadır, değil mi? Peki, iç blok içerisinde a' yı kullandığımız zaman derleyici bunu hangi a olarak kabul eder dersiniz? İşte bu durumlarda iç blok içerisinde tanımlanmış olan aynı isimli değişkenler, dış blok içerisinde tanımlanmış olan aynı isimli değişkenleri maskelerler. Dolayısıyla yukarıdaki örnekte iç blok içerisinde ekrana yazdırdığınız a değişkeni, 20 olan a değişkenidir.

Anlattıklarımızı şöyle özetleyebiliriz:

C'de farklı faaliyet alanlarına sahip aynı isimli değişkenler tanımlanabilirler, fakat aynı faaliyet alanına sahip aynı isimli değişkenler tanımlanamazlar. Farklı faaliyet alanlarına sahip olan aynı isimli değişkenler için derleyiciler bellekte farklı yerler ayırırlar. Bu nedenle bunların birbirleriyle karışması söz konusu olmaz. Aynı isimli birden fazla değişkenin tanınabilir olduğu bir blokta, her zaman kendi bloğu içerisinde tanımlanmış -en dar faaliyet alanına sahip olan- değişkene erişilebilir.

Kafanızdaki soru işaretlerini giderecek (eğer hala kaldıysa (!)) birkaç örnek daha vermek istiyoruz:

```
...
{
     int x;
     char ch;
     double x;

     /* Hata! aynı faaliyet alanına sahip aynı isimli birden
     fazla değişken olamaz */
     ...
}
...
```

İki tane yerel x değişkeni tanımlanmıştır ve bunların faaliyet alanları aynıdır. Dolayısıyla ikinci x'i tanımlamak hatadır.

```
...
{
        int x;
        char ch;
        ...
    {
        int x;              /* hata değil */
        ...
    }

}
...
```

İkisi de yerel olmasına karşın farklı faaliyet alanlarına sahip iki ayrı x değişkeni söz konusudur; hata değildir.

```
void fonk1(void)
{
    int i = 10;
    ...
}

int fonk2 (void)
{
    int i = 20;
    ...
}
```

fonk1'deki yerel i değişkeni ile **fonk2**'deki yerel i değişkeni farklı faaliyet alanlarına sahiptir, dolayısıyla bellekte farklı yerlerde tutulurlar ve birbirleriyle karışmazlar.

8.3 GLOBAL DEĞİŞKENLER (Global Variables)

Global değişkenler bütün blokların dışında tanımlanmış olan değişkenlerdir. **"Bütün blokların dışı"** demekle neyi anlatmak istiyoruz acaba?

Aşağıdaki örneği inceleyiniz.

```
#include <stdio.h>

...
/* Burada global değişken tanımlanabilir */

int fonk1(void)
{
      .
      .
      .
}
```

```
/* Burada global değişken tanımlanabilir */

main(void)
{
      .
      .
      .
}

/* Burada global değişken tanımlanabilir */
...
```

Yorum satırlarının bulunduğu bölgeler global değişkenlerin tanımlanabileceği bölgeleri gösteriyor. (C'de /* ile */ arasına yazılan karakterler derleyici tarafından dikkate alınmazlar. Programcının kaynak kod ile ilgili açıklamalar yaptığı bu satırlara yorum satırları da denir.) Gördüğünüz gibi, bu bölgeler hiçbir fonksiyonun içinde değildir. Global değişkenler **dosya faaliyet alanı** kuralına uyarlar. Yani global değişkenler programın her yerinde; başka bir deyişle, tüm fonksiyonların içerisinde tanınabilirler. Aşağıdaki örneği inceleyiniz:

```
#include <stdio.h>
int x;                                  /* x global bir değişkendir */

void fonk(void)
{
      x = 200;
}

void main()
{
      x = 100;
      printf("x = %d\n", x);            /* x = 100 */
      fonk();
      printf("x = %d\n", x);            /* x = 200 */
}
```

Bu örnekte x tüm blokların dışında tanımlandığı için global bir değişkendir. x değişkeninin faaliyet alanı dosya faaliyet alanıdır; yani, x tüm fonksiyonlarda tanınır. Neler olduğunu adım adım izleyelim:

Program main fonksiyonundan çalışmaya başlayacaktır.

```
x = 100;
```

ifadesi ile x değişkenine 100 atanır. Daha sonra **fonk** isimli fonksiyon çağrılı-

yor. **fonk** fonksiyonu x değişkeninin değerini **200** olarak değiştirir. Yani programın akışı **fonk**'tan **main**' e döndüğünde **x**'in değeri de değişmiş olacaktır. Bu durumda ekranda:

```
x = 100
x = 200
```

görmemiz gerekir.

Global değişkenlere de ilkdeğer verilebilir.

Örneğin:

```
int a = 500;

int f1(void)
{
      .
      .
      .
}
```

burada **a** değişkenine tanımlanma sırasında ilkdeğer olarak **500** verilmiştir.

Bir global değişkenin her fonksiyonda tanınabilmesi ona her fonksiyondan erişebileceğimiz anlamına gelmiyor. Çünkü aynı isimli hem yerel hem de global değişkenin tanınabilir olduğu bir blokta ancak yerel değişkene erişebiliriz.

Örneğin:

```
int x = 100;                          /* Global x */

void fonk()
{
   x = 400;                           /* Global x değiştiriliyor */
   printf("Global x = %d\n", x);      /* Global x yazdırılıyor */
}

void main(void)
{
   int x;                             /* Yerel x */

   x = 300;                           /* Yerel x değiştiriliyor */
   printf("Yerel x = %d\n", x);       /* Yerel x yazdırılıyor */
   fonk();
   printf("Yerel x = %d\n", x);       /* Yerel x yazdırılıyor */
}
```

Fonksiyonların kendileri de, bütün blokların dışında tanımlandıklarına göre global nesnelerdir. Gerçekten de fonksiyonlar kaynak kodun her yerinden çağrılabilirler. Aynı faaliyet alanına ilişkin aynı isimli birden fazla değişken olamayacağına göre, aynı isme sahip birden fazla fonksiyon da olamaz.

Not: C++'da aynı isimli birden fazla fonksiyon tanımlanabilmektedir (function overloading). Bu durumda C++ derleyicileri aynı isimli fonksiyonları parametre sayılarına ve türlerine göre birbirlerinden ayırırlar.

Programcıların çoğu global değişkenleri mümkün olduğu kadar az kullanmak ister. Çünkü global değişkenleri kullanan fonksiyonlar, başka projelerde rahatlıkla kullanılamazlar. Global değişkenlerin, fonksiyonların yeniden kullanılabilirliğini azalttığını söyleyebiliriz. Yeniden kullanılabilirlik (reusability) nesne yönelimli programlamanın da anahtar fikirlerinden bir tanesidir.

8.4 PARAMETRE DEĞİŞKENLERİ (Formal Parameters)

Parametre değişkenleri, fonksiyon parametreleri olarak kullanılan değişkenlerdir. Parametre değişkenleri **fonksiyon faaliyet alanı** kuralına uyarlar. Yani, parametresi oldukları fonksiyonların her yerinde tanınabilirler.

```
fonk(int a, int b)
{
      .
      .       /* a ve b fonksiyonun her yerinde tanınabilir */
      .
}
```

Şimdi aşağıdaki örneği inceleyelim:

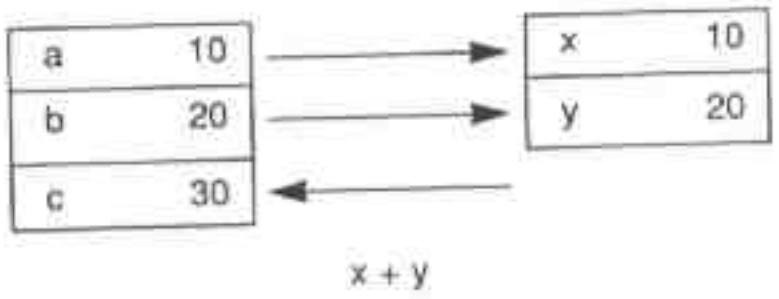
```
fx(int x)
{
      int x;                  /* Hata */
      ...

      {
            int x;            /* Hata değil */
            ...

      }
      ...
}
```

Bu örnekte fonksiyonun ana bloğunun başında tanımlanmış olan x, *"aynı faaliyet alanına sahip aynı isimli birden fazla değişken tanımlanamaz"* kuralı uyarınca geçersizdir. Biri parametre değişkeni, diğeri yerel değişken olmasına karşın, her iki x değişkeni de aynı faaliyet alanına sahiptir.

8.5 PARAMETRE AKTARIM KURALI

C'de parametreler, çağıran fonksiyonlardan çağrılan fonksiyonlara kopyalanarak aktarılırlar. Parametrelerin kopyalanarak fonksiyonlara aktarılması C'nin en önemli özelliklerinden biridir.

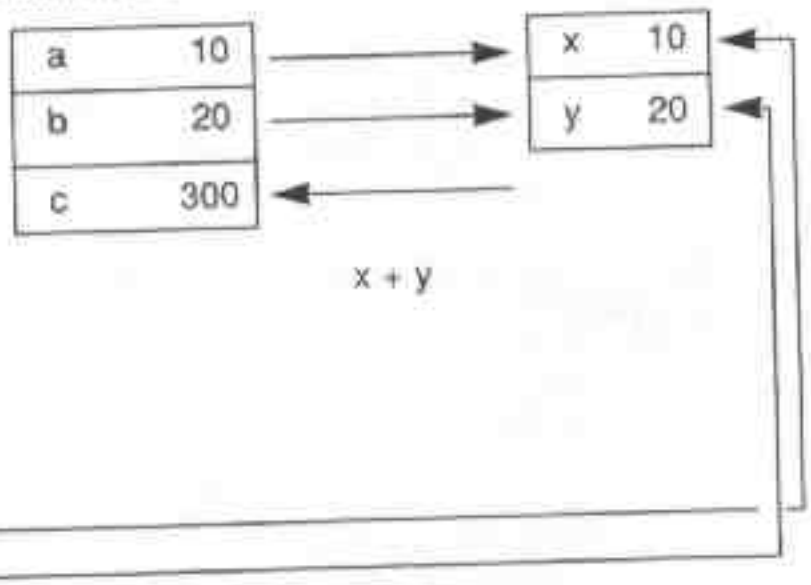
Aşağıdaki örneği inceleyiniz:




```
#include <stdio.h>
void main(void)
{
     int a = 10, b = 20, c;

     c = topla(a, b);
     printf("%d\n" ,c);
}

int topla (int x, int y)
{
     return x + y;
}
```

topla fonksiyonu çağrıldığında programın akışı bu fonksiyona geçmeden önce, a ve b değişkenlerinin içinde bulunan değerler, x ve y parametre değişkenlerine kopyalanırlar.

Şimdi size bir soru: **topla** fonksiyonunda x ve y'ye başka değerler atasak bundan **main** fonksiyonundaki a ve b etkilenir mi?..




```
#include <stdio.h>

void main(void)
{
     int a = 10, b = 20, c;

     c = topla(a, b);
     printf("%d\n" ,c);
}
int topla (int x, int y)
{
     x = 100;
     y = 200;
     return x + y;
}
```

Yanıt hayır!.. parametre değişkenleri bellekte farklı yerlerde tutulan farklı nesnelerdir. Bu yüzden örneğimizde, x ve y parametrelerini değiştirmekle çağıran fonksiyondaki a ve b'yi değiştiremeyiz. **Bu örneği mutlaka çalıştırarak deneyiniz!..**

Parametre aktarımında parametrelerin kendilerinin değil de kopyalarının altprogramlara geçirilmesi yöntemine yazılım terminolojisinde genel olarak **değerle çağırma (call by value)** denir. Diğer programlama dillerinin bir bölümünde de parametre aktarımında bu yöntem kullanılmaktadır. **Parametre aktarım işleminin aslında bir çeşit atama işlemi olduğuna da dikkat etmelisiniz.**

Pascal Programcısına Not: *Pascal*'da parametrelerin altprogramlara geçirilmesinde iki yöntem kullanılır. Birinci yöntem tıpkı C'de olduğu gibidir **(call by value)**. Yani parametreler kopyalanarak altprograma geçirilir. İkincisi ise, parametrelerin kopyalarının değil de kendilerinin geçirilmesini sağlayan bir yöntemdir **(call by reference)**. Hangi yöntemin kullanılacağı parametreleri tanımlarken parametre değişkenlerinin önüne **var** anahtar sözcüğünün getirilip getirilmediğine göre belirlenir. Göstericilerin *Pascal*'da bu kadar zayıf kullanılmasının nedeni de budur.

```
procedure X1 (var a:integer);      a'nın kendisi geçiriliyor.
procedure X2 (a:integer);          a'nın kopyası geçiriliyor.
```

80X86 Sembolik Makina Dili Programcısına Not: C'de parametrelerin fonksiyonlara kopyalanmasında `stack bölgesi` kullanılır. C derleyicileri bir fonksiyon çağrıldığında parametreleri sağdan sola doğru `stack bölgesine` yollar ve daha sonra fonksiyonu çağırırlar. Fonksiyondan çıkışta `stack ayarlaması` derleyici tarafından yapılmaktadır.

```
...
      fonk(a, b);
...
```

gibi bir ifade için derleyici aşağıdaki gibi bir kod üretir:

```
...
push b
push a
call _fonk
add sp, 4
...
```

`fonk` isimli fonksiyon bellek modeline göre, gönderilen parametreleri `bp+4` ya da `bp+6`'dan başlayarak alır. Parametrelerin ters sırada stack bölgesine yollanmasına ve `stack` bölgesinin fonksiyon çıkışında dengelenmesine *"C tarzı fonksiyon çağırma"* denilmektedir. C tarzı fonksiyon çağırma biçimi **Borland** derleyicilerinde `cdecl`, **Microsoft** derleyicilerinde ise `_cdecl` anahtar sözcükleri ile ifade edilir.

8.6 NESNELERİN ÖMÜRLERİ (Duration)

Ömür, nesnelerin faaliyet gösterdiği zaman aralığını anlatmak için kullanılan bir terimdir. Nesneler belli bir zamanda yaratılırlar ve yine belli bir zamanda faaliyetlerini bitirerek yok olurlar. Ömürleri bakımından nesneleri 2 bölüme ayırabiliriz:

1) Statik ömürlü nesneler.

2) Dinamik ömürlü nesneler.

8.6.1 Statik ömürlü nesneler (Static duration)

Statik ömürlü nesneler, programın çalışmaya başlamasıyla yaratılırlar, programın çalışması bitene kadar faaliyet gösterirler. Statik nesneler genel olarak *object modül (.OBJ)* içerisine yazılırlar.

80X86 Sembolik Makina Dili Programcısına Not: Statik ömürlü nesneler *80X86* mimarisinde `data segment` bölgesini kullanır. Bu nedenle derleyiciler statik nesneleri, derleme sırasında segment içi değer olarak amaç kodun içine yazarlar.

C'de statik ömürlü üç nesne grubu vardır:

1) Global değişkenler
2) Stringler (iki tırnak içerisindeki ifadeler)
3) Statik yerel değişkenler

Bu üç grup arasından siz şu anda yalnızca global değişkenlerin ne olduğunu biliyorsunuz. Stringler 20. Bölümde ve statik yerel değişkenler de 13. Bölümde ele alınmaktadır.

Bu durumda global değişkenlerin ömürleri konusunda şunları söyleyebiliriz: Bunlar programın çalışması süresince yaşayan statik ömürlü değişkenlerdir; programın yüklenmesiyle yaratılırlar ve program sonuna kadar bellekte kalırlar.

8.6.2 Dinamik ömürlü nesneler (Dynamic duration)

Dinamik ömürlü nesneler programın çalışmasının belli bir zamanında yaratılan ve belli bir süre faaliyet gösterdikten sonra yok olan nesnelerdir. Bu tür nesnelerin ömürleri, programın toplam çalışma süresinden kısadır.

C'de dinamik ömürlü üç nesne grubu vardır.

1) Yerel değişkenler
2) Parametre değişkenleri
3) Dinamik bellek fonksiyonları ile yaratılmış olan nesneler

İlk iki grubu oluşturan yerel ve parametre değişkenlerinin ne olduklarını biliyoruz. Dinamik bellek fonksiyonları kullanılarak oluşturulmuş nesneler 23.Bölümde ele alınmaktadır.

Yerel değişkenler ve parametre değişkenleri dinamik ömürlü nesnelerdir. Tanımlandıkları bloğun çalışması başladığında yaratılırlar; bloğun çalışması bitince yok olurlar. Faaliyet alanları kendi bloklarının uzunluğu kadar olan yerel değişkenlerin ömürleri "bloğun çalışma süresi" kadardır.

```
...
{
     int a;          /* Bellekte a için yer  ayrılıyor */
     .
     .               /* a blok içerisinde yaşıyor */
     .
}                    /* a'nın ömrü sona eriyor */
...
```

Parametre değişkenleri de benzer biçimde fonksiyon çağrıldığında yaratılırlar; fonksiyon çalışması boyunca yaşarlar, fonksiyonun çalışması bitince yok olurlar.

```
fonk(int a) /* a yaratılıyor */
{
     .
     .
     .
}                    /* a'ın ömürü bitiyor,
                        a için ayırılan yer geri boşaltılıyor */
```

80X86 Sembolik Makina Dili Programcısına Not: Yerel değişkenler ve parametre değişkenleri 80X86 sistemlerinde `stack segment` bölgesini kullanırlar. `Stack` bölgesinin yerel ve parametre değişkenleri tarafından kullanılmaları aşağıda özetlenmiştir:

```
int fonk (int x, int y)
{
     int a, b;
     ...
}
```

Bu fonksiyonun sembolik makina dili kodu aşağıdaki gibi olacaktır:

```
_fonk  proc
       push    bp
       mov     bp, sp
       sub     sp, 4        ;a ve b değişkeni için yer ayrılıyor
       ...
       mov     sp, bp
       pop     bp
_fonk  endp
```

Statik değişkenler yaratıldıklarında içlerine otomatik olarak `0 (sıfır)` değeri yerleştirilir. Yani statik ömürlü bir değişkene ilkdeğer verilmemiş olsa bile içinde `0` değeri bulunmaktadır. Oysa dinamik değişkenlere değer atanmadığı durumlarda içlerindeki değerler bilinemez. O anda değişken için ayrılan bellek bölgesinde, rastgele hangi değer varsa değişkenin değeri de o olur. Ancak, herhangi bir hatanın oluşma olasılığına karşı derleyiciler genel olarak değer atanmamış değişkenlerin kullanılması durumunda uyarı mesajı verirler.

Biraz daha somut olarak:

Değer atanmamış global bir değişkenin içinde 0 (sıfır) vardır. Oysa bir yerel değişkene değer atanmamışsa içindeki değer kestirilemez; o anda bellekte bulunan rastgele bir sayıdır.

Aşağıdaki örneği bilgisayarınızda yazarak çalıştırınız. Derleyiciniz tarafından verilen uyarı mesajını inceleyiniz.

```
#include <stdio.h>

int a;                  /* ilk değer verilmemiş */

void main(void)
{
     int b;             /* İlk değer verilmemiş */
     printf ("a = %d b = %d\n", a, b);   /* a = 0 b = ? */
}
```

SORAMADIKLARINIZ...

S1) C'de parametreler fonksiyonlara kopyalanarak geçirildiğine göre, çağrılan fonksiyonun çağıran fonksiyona ilişkin yerel değişkenlerini değiştirebilmesi mümkün müdür?

C1) Bu soru **göstericiler (pointers)** konusu ile doğrudan ilgilidir. Tabi, biz şimdilik bu soruya yanıt vermeyeceğiz. Ancak şunu da belirtmekte yarar görüyoruz: Böyle bir değişiklik fonksiyona parametre olarak değişkenin kendisinin değil de adresinin geçirilmesiyle mümkün olabilir (call by reference).

S2) Aynı isimli global ve yerel değişkenin söz konusu olduğu bir durumda yerel değişkenin olduğu blokta global değişkene erişmenin herhangi bir yolu var mıdır?

```
#include <stdio.h>

int a = 100;

void main(void)
{
     int a = 200;

     /* Global a değişkenine erişmenin yolu var mı?  */
     ...
}
```

C2) Standart C'de böyle bir yol yok; ancak C++'da **:: operatörü (scope resolution operator)** ile bu yapılabilir.

Aşağıdaki bir C++ programı olsun:

```
#include <stdio.h>

int a;

void main()
{
     int a;

     ::a = 300;              /* global a değişkenine atanıyor */
     a = 100;                /* yerel a değişkenine atanıyor */
}
```

C++'a geçmeden önce standart C'yi çok iyi öğrenmiş olmalısınız!..

9

OPERATÖRLER

Bu bölüm operatörler konusuna ayrılmıştır. Operatörlerin sınıflandırılması, işlevleri ve aralarındaki öncelik ilişkileri örneklerle ayrıntılı bir biçimde ele alınmaktadır. *C* hem operatörlerin işlevleri bakımından hem de aralarındaki öncelik ilişkisi bakımından diğer programlama dillerine göre oldukça karmaşıktır. Öyle ki, deneyimli *C* programcılarının bile operatörlerin öncelik ilişkisi konusunda çatışmaya düştüğü zamanlar olabilmektedir. Neyse ki biz bu bölümde çok açık bir anlatım biçimine başvurduk. Kafanızda oluşacak soru işaretlerini de örneklerle gidermeye çalıştık.

Bu bölümde tüm operatörler ele alınmıyor. Karmaşık konularla ilişkili operatörler, o konuların ele alındığı bölümlere bırakılmıştır. Örneğin göstericilere ilişkin operatörler göstericiler bölümünde, yapılara ilişikin operatörler de yapıların anlatıldığı bölümde ele alınmaktadır.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz ve kavramları açıklayabilmeniz gerekir:

Sorular:

1) İşlevlerine ve operand sayılarına göre operatörler nasıl sınıflandırılır?

2) *Soldan sağa* ve *sağdan sola* öncelik ilişkisi ne anlama gelmektedir?

3) Artırma (++) ve eksiltme (--) operatörlerinin işlevleri nedir?

4) İlişkisel ve mantıksal operatörler hangileridir ve ne amaçla kullanılırlar?

5) İlişkisel ve mantıksal operatörlerin aldıkları operandların türleri ve ürettikleri değerler hakkında neler söylenebilir?

6) Grup olarak ele alındığında operatörlerin öncelik ilişkileri konusunda neler söylenebilir?

7) Şüpheli kodların zararları nelerdir; neden tavsiye edilmez?

8) Bit operatörleri hangi amaçlarla kullanılır?

Kavramlar:

1) Operatör, operand

2) Önek (prefix), araek (infix), sonek (postfix)

3) Öncelik sırası

9.1 OPERATÖR NEDİR?

Operatörler, nesneler veya sabitler üzerinde önceden tanımlanmış birtakım işlemleri yapan atomlardır. Operatörler mikroişlemci tarafından bir faaliyete neden olurlar ve bu faaliyet sonunda da bir değer üretilmesini sağlarlar. Programlama dillerindeki her operatör bir ya da birden fazla makina komutuna karşılık gelir.

C'de her ifade en az bir operatör içerir:

```
c = a * b / 2;          3 operatör vardır: = * /
d = fonk();             2 operatör vardır: = ()
c = a > 4;              2 operatör vardır: = >
++x;                    1 operatör vardır: ++
...                     ...
```

Her operatörün **operandları** vardır. Operandlar operatörlerin işleme soktukları nesneler ya da sabitlerdir.

```
a + b       Operatör: +, operandları: a ve b (2 tane)
c++         Operatör: ++, operandı: c (1 tane)
b > 3       Operatör: >, operandları: b ve 3 (2 tane)
...
```

9.2 OPERATÖRLER ARASINDAKİ ÖNCELİK İLİŞKİSİ

Parantez kullanılmamışsa -C'de parantezleri de bir operatör olarak ele alıyoruz- her operatörün diğerlerine göre bir öncelik sırası vardır. Örneğin bilirsiniz, programlama dillerinin hemen hepsinde çarpma operatörünün toplama operatörüne göre bir önceliği vardır; atama operatörü ise en az önceliklidir. Örneğin:

```
c = a + b * 2;
```

ifadesi derleyiciler tarafından şu sırada ele alınır:

```
İşlem1: b * 2
İşlem2: a + İşlem1
İşlem3: c = İşlem2
```

Birkaç örnek daha vermek istiyoruz:

```
x = a * b - 2 / 3;
```

```
İşlem1: a * b
İşlem2: 2 / 3
İşlem3: İşlem1 - İşlem2
İşlem4: x = İşlem3
```

Aynı önceliğe sahip operatörler arasındaki işlem sırası iki biçimde olabilir: Soldan sağa ya da sağdan sola.

Örneğin:

```
a = x - b / 2 * c;
```

Bölme (/) ve çarpma (*) operatörleri eşit önceliğe sahiptir. Bu operatörler kural gereği C'de soldan sağa öncelikli olarak işleme sokulurlar. Şimdi soralım; yukarıdaki örnekte hangi operatör daha soldadır? Çarpma mı (*), bölme (/) mi?..

```
İ1: b / 2
İ2: İ1 * c
İ3: x - İ2
İ4: a = İ3
```

(İşlem1, İşlem2, İşlem3 yerine İ1, İ2, İ3,... kullanacağız)

C'de atama operatörü sağdan sola öncelikli bir operatördür.

```
a = b = c;
İ1: b = c
İ2: a = İ1 (yani a = b)
```

Diğer programlama dillerinde olduğu gibi C'de de, parantezler içine alınan işlemler öncelik kazanırlar. İlerledikçe göreceksiniz; C'de parantezler de aslında birer operatördür.

```
a = (b + 2) * 80

İ1: (b + 2)
İ2: İ1 * 80
İ3: a = İ2
```

Bir operatör sembolü birden fazla işleve sahip olabilir. Onun hangi işlevinin kullanıldığına derleyici, ifade içindeki durumunu gözönüne alarak karar verir. Örneğin:

```
c = fonk();
d = (a + b) * c;
```

Birinci örnekte (...) fonksiyon çağırma operatörü olarak kullanılmıştır; oysa ikinci örnekte aynı operatör işlemlerin öncelik sırasını değiştirmektedir.

Operatörleri daha iyi tanımak ve aralarındaki ilişkileri daha iyi kavramak için sınıflandırma yoluna gideceğiz.

9.3 OPERATÖRLERİN SINIFLANDIRILMASI

Operatörleri 3 biçimde sınıflandırabiliriz..

1) İşlevlerine göre

2) Operand sayılarına göre

3) Operatörün konumuna göre

C'de operatörleri işlevlerine göre 6 gruba ayırabiliriz.

9.4 OPERATÖRLERİN İŞLEVLERİNE GÖRE SINIFLANDIRILMASI

İlk 3 grup diğer programlama dillerinin hemen hepsinde vardır. Bit operatörleri ve gösterici operatörleri ise programlama dillerinin çoğunda bulunmazlar. Yine programlama dillerinin çoğu, kendi uygulama alanlarında kolaylık sağlayacak birtakım özel amaçlı operatörlere de sahip olabilir.

Aritmetik operatörler, 4 işlemle ilişkili olan operatörlerdir:

*	Çarpma
/	Bölme
%	Mod alma (bölümden elde edilen kalan)
+	Toplama
-	Çıkartma

İlişkisel operatörler iki değer arasındaki ilişkiyi sorgularlar. Bunlara "karşılaştırma (comparison) operatörleri de denir:

>	Büyük
<	Küçük
>=	Büyük ya da eşit
<=	Küçük ya da eşit

==	Eşit
!=	Eşit değil

Mantıksal operatörler mantıksal işlemler yapan operatörlerdir:

!	Değil (not)
&&	Ve (and)
||	Veya (or)

Bit operatörleri bit seviyesinde işlemler yapar:

~	Değil (bitwise not)
<<	Sola kaydırma (left shift)
>>	Sağa kaydırma (right shift)
&	Ve (bitwise and)
^	Özel veya (bitwise exor)
|	Veya (bitwise or)

Gösterici operatörleri adres işlemlerinde kullanılan operatörlerdir:

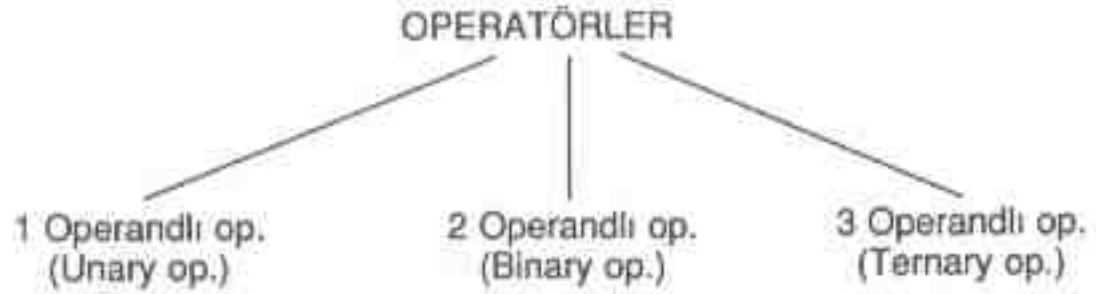
*	İçerik alma (indirection)
&	Adres alma (address of)
[]	İndeks (index)
->	Yapı gösterici (arrow)

Özel amaçlı operatörler:

`()`	Fonksiyon çağırma ve öncelik değiştirme
`.`	Yapı elemanına erişme
`(tür)`	Tür dönüştürme
`sizeof`	Uzunluk
`?:`	Koşul
`=`	Atama
`+=, *=, /=, ...,`	İşlemli atama

9.5 OPERATÖRLERİN OPERAND SAYILARINA GÖRE SINIFLANDIRILMASI

Operatörleri işleme soktukları operandların sayısına göre de sınıflayabiliriz.



Programlama dillerinde 2'den fazla operand alan operatörlere pek rastlanmaz. C'de 3 operand alan ve ismine koşul operatörü denilen istisna bir operatör vardır.

Örneğin:

`*, /, +, -, =, ...` İki operandlı operatörlerdir.

```
a = a * b + 2;
```

```
t1: a * b        (2 operand)
t2: t1 - 2       (2 operand)
t3: a = t2       (2 operand)
```

`++, —, !, ...` Tek operandlı operatörlerdir.

```
b = !++c;
```

```
t1: ++c          (1 operand)
t2: !t1          (1 operand)
```

9.6 OPERATÖRLERİN KONUMLARINA GÖRE SINIFLANDIRILMASI

2 operand alan bir operatör, C'de her zaman operandlarının arasında **(infix)** bulunur.

Sembolik makina dillerinde (assembly languages) operatörlerin operandlarının **önünde (prefix)** bulunduğunu görüyoruz:

Tek operand alan operatörlerde ise operatör ya operandının **önünde (prefix)** ya da **arkasında (postfix)** bulunur.

++, C'nin çok kullanılan tek operandlı bir operatörüdür. Operandının önünde ya da arkasında bulunabilir.

Sonuç olarak, ister iki operandlı olsun, isterse tek operandlı olsun; operatörler operandlarının önünde, ortasında ya da arkasında bulunabilirler. Biz kitabımızda bir operatörün operandlarının önünde bulunmasına **önek (prefix)**, arkasında bulunmasına **sonek (postfix)** ve ortasında bulunmasına **araek (infix)** durumu diyeceğiz. Buna göre operatörleri konumlarına göre üç gruba ayırabiliriz.

C'nin tüm iki operandlı operatörleri **araek** durumunda kullanılır. Tek operandlı operatörlerin ise bir kısmı yalnızca **önek**, bir kısmı yalnızca **sonek**, bir kısmı da hem **önek** hem de **sonek** durumunda kullanılabilirler.

Şimdi operatörleri, işlevlerine göre teker teker inceleyelim.

9.7 ARİTMETİK OPERATÖRLER

Aritmetik operatörler dört işlemle ilgili olan operatörlerdir. Aritmetik operatörlerin listesi ve kendi aralarındaki önceliği şöyledir:

++ —	**Sağdan sola**
* / %	**Soldan sağa**
+ -	**Soldan sağa**
...	
=	**Atama operatörü C'nin düşük öncelikli operatörlerinden birisidir.**

+ ve - Operatörleri

İki operandlı araek operatörlerdir. Diğer dillerde olduğu gibi + iki operandı toplamak, - ise iki operandın farkını almak için kullanılır. Operandları herhangi bir türden olabilir.

* ve / Operatörleri

İki operandlı aracek operatörlerdir. * operandları çarpmak, / ise operandları bölmek için kullanılır. Diğer dillerde olduğu gibi + ve - operatörlerinden yüksek önceliklidirler. Bu iki operatörün de operandları herhangi bir türden olabilir.

% Operatörü

İki operandlı bir operatördür. % operatörü sol tarafındaki operandın sağ tarafındaki operanda bölümünden elde edilen kalanı bulmak için kullanılır. % operatörünün İki operandı da tamsayı türlerinden (char, short, int, long) olmak zorundadır.

Örneğin:

```
c = 10 % 3;
```

10'un 3'e bölümünden elde edilen kalan 1'dir (c=1)

% operatörü, * ve / operatörleriyle soldan sağa eşit önceliğe sahiptir. + ve - operatörlerinden daha yüksek önceliklidir.

Örneğin:

```
c = 10 % 4 + 1;
i1:  10 % 4  ⟶  2
i2:  i1 + 1  ⟶  3
i3:  c = i2  ⟶  3
```

Artırma (++) ve Eksiltme (--) Operatörleri

`Artırma (++)` ve `eksiltme (—)` operatörleri *C*'nin en çok kullanılan operatörlerindendir. Tek operand alırlar; **önek** ya da **sonek** durumunda bulunabilirler. Artırma operatörü (++), operandı olan değişkenin içeriğini 1 artırmak, eksiltme operatörü (--) ise operandı olan değişkenin içeriğini 1 eksiltmek için kullanılır. Bu iki operatör de diğer aritmetik operatörlerden daha yüksek önceliğe sahiptir.

İşlevlerini 2 biçimde inceleyebiliriz:

1) Yalın olarak kullanıldıklarında (başka hiçbir operatör olmaksızın) önek ya da sonek durumları arasında hiçbir fark yoktur. **++**, "**1 artır**", **—** ise "**1 eksilt**" anlamına gelir.

++x; ile x++; ikisi de birbirinin tamamen aynısıdır ve her ikisi de `x = x + 1;` anlamına gelir.

—x; ile x—; İkisi de birbirinin tamamen aynısıdır ve her ikisi de `x = x - 1` anlamına gelir.

Örneğin:

```
...
x = 10;
++x;            /* x = 11 */
x++;            /* x = 12 */
...

x = 20;
--x;            /* x = 19 */
x--;            /* x = 18 */
...
```

2) Diğer operatörlerle birlikte kullanıldıklarında (örneğin atama operatörü ile) önek ve sonek biçimleri farklı anlamlara gelir:

Önek durumunda artırma ya da eksiltme işlemi tabloda belirtildiği gibi, diğer aritmetik operatörlerden daha yüksek öncelikli olarak yapılır.

```
x = 10;
y = ++x;

/* x = 11, y = 11 */
```

Burada `x = x + 1` işlemi yapılır; böylece `x = 11` olur. Daha sonra bu değer `y`'ye atanır. Daha açık bir biçimde:

```
y = ++x;

i1: ++x      ---> 11
i2: y = i1   ---> 11
```

++ ve -- sonek durumundaysa artırma ve eksiltme işlemi söz konusu ifadenin en son işlemi olacak biçimde yapılır.

```
x = 10;
y = x++;
/* x = 11, y = 10 */
```

Bu örnekteki, `y = x++` ifadesini ele alalım. `++` sonek durumunda olduğu için **artırım hemen değil, ifadenin son işlemi olacak biçimde, ifadenin en sonunda yapılacaktır.**

```
i1: y = x      ---> 10
i2: x = x + 1  ---> 11
```

Aşağıdaki programı yazıp deneyiniz...

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

```
void main(void)
{
     int a, b;

     a = 10;
     b = ++a;
     printf("a = %d b = %d\n", a, b);     /* a = 11, b = 11 */

     a = 10;
     b = a++;
     printf("a = %d b = %d\n", a, b);     /* a = 11, b = 10 */
}
```

Birkaç örnek daha vermek istiyoruz.

```
x = 10;
y = 5;
z = x++ % 4 * --y;
/* x = ?, y = ?, z = ? */
```

İşlem sırası:

i1: --y	⟶	4	ifade içerisindeki en öncelikli operatör
i2: x % 4	⟶	2	% ile * soldan sağa eşit önceliğe sahiptir
i3: i2 * i1	⟶	8	
i4: z = i4	⟶	8	
i5: x++	⟶	11	x = x + 1, sonek durumunda olan x ifadenin son işlemi olarak artırılıyor.

Bu durumda:

x = 11, y = 4, z = 8

Yukarıdaki örneği yalnızca "alıştırma olsun" diye verdik. Uygulamada ++ ve — operatörlerinin böyle karmaşık ifadelerin içine sokulması pek tavsiye edilmez.

Bir örnek daha:

```
c = 20;
d = 10;
a = b =  d + c--;
/* a = ?, b = ?, c = ?, d = ? */
```

İşlem sırası:

i1: d + c	⟶	30	+ operatörü = operatöründen daha öncelikli dir. (— sonek durumunda olduğu için ifadenin son işlemi olarak yapılacak)
i2: b = i1	⟶	30	Atama operatörü sağdan sola önceliklidir.
i3: a = i2	⟶	30	**a = b** anlamına geliyor.
i4: c—	⟶	19	**c = c - 1**, sonek durumundaki — ifadenin sonunda eksiltiliyor.

Bu durumda:

```
a=30,b=30,c=19,d=10
```

80X86 Sembolik Makina Dili Programcısına Not: C'nin artırma ve eksiltme operatörleri sembolik makina dillerinin **INC** ve **DEC** komutlarına karşılık gelmektedir. Örneğin:

```
...
b = a++;
...
```

C ifadesini derleyici şu biçimde makina komutlarına dönüştürebilir:

```
...
MOV   AX, word ptr [AAAA]
MOV   word ptr [BBBB], AX
INC   word ptr [AAAA]
...
```

9.8 ŞÜPHELİ KODLAR

++ ve — operatörlerinin amaç dışı ve bilinçsizce kullanılmaları taşınabilirlik konusunda problemlere neden olabilir. Bu operatörlerin bulunduğu anlaşılmaz ve tuhaf ifadeler, derleyiciler arasında yorum farklılığına yol açarak taşınabilirliği bozarlar. Böyle taşınabilir olmayan **şüpheli kodlardan** kaçınmak gerekir. Her nedense *C* öğrencilerinin çoğu, bazı çatışmalı ifadelerin sonuçlarını obsesyonel bir biçimde merak ederler. Bu şüpheli durumların nasıl olup da standardize edilmediğine şaşar kalırlar. Oysa ortada *C*'ce bir problem yoktur. Çünkü amaç dışı, çatışmaya sevkedici ve yararsız birtakım ifadelerin standardize edilmesi *C*'nin genel felsefesine de ters düşmektedir. **İyi bir C programcısı olmak istiyorsanız, programlarınızı mümkün olduğu kadar açık, anlaşılır ve taşınabilir yazmalısınız.**

Kullanılmaması salık verilen, derleyicilerden derleyicilere yorum farklılığına neden olabilecek tüm şüpheli kodlar aşağıda tek tek inceleniyor. *Unutmayın: "Bir derleyici şüpheli bir kodu hep aynı biçimde yorumlar; burada sorun, bütün derleyicilerin aynı biçimde yorumlamama olasılığıdır"*.

1) Üç + operatörü boşluk karakterleriyle ayrılmış olmaksızın yan yana getirilmemelidir.

```
a = 10;
b = 20;
c = a+++b;
/* c = ? */
```

Burada çalıştığımız derleyici **a+++b** ifadesini, **a++ + b** ya da **a + ++b** biçiminde ele alabilir? İki yorumlayış biçiminden farklı sonuçlar elde edilir, değil mi? Sahi, bu kodu yazan kişi bir boşluğu neden derleyiciden esirgemiş olabilir dersiniz?..

2) İfade içerisinde bir değişken birden fazla ++ ya da — ile kullanılmamalıdır. Örneğin:

```
a = 10;
b = ++a + ++a;
/* a = ?, b = ? */
```

İfadenin sonucunu nasıl değerlendirirsiniz? Derleyicilerin çoğu bu kodu aşağıdaki gibi ele almaz!

```
İ1: ++a          ——► 11
İ2: ++a          ——► 12
İ3: İ1 + İ2      ——► 23
İ4: b = İ3       ——► 23
```

Sonuç :

```
a = 12, b = 23
```

İşlemlerin bu biçimde yürütüleceğinin hiçbir garantisi yoktur. Başka bir örnek:

```
a = 10
b =  a++ + ++a;
/* a = ?, b = ? */
```

Burada sonek artırım önek artımdan elde edilen değer üzerine mi yapılacak? İşlemlerin aşağıdaki gibi yapılacağının bir garantisi yoktur.

```
İ1: ++a          ——►11
İ2: a + İ1       ——►22
İ3: b = İ2       ——►22
İ4: a = a + 1    ——►12
```

3) Fonksiyon çağrılırken aynı değişken birden fazla parametre içinde kullanılmışsa, hiçbiri ++ ve — operatörleriyle işleme sokulmuş olmamalıdır. Örneğin:

```
a = 10;
b = fonk(a, ++a);
```

ifadesinde fonksiyon hangi parametrelerle çağrılıyor? `fonk(10, 11)` ile mi? Yoksa `fonk(11, 11)` ile mi?.. Şimdi siz `fonk(10, 11)` ile çağrılmasının daha doğal olduğunu düşünebilirsiniz. Oysa sistemlerin çoğunda (örneğin *80X86* mimarisindeki *C* derleyicilerinde *DOS, UNIX ve Windows'ta* parametreler sağdan sola doğru ele alınırlar.)

Bir değişken parametre olarak (diğer parametrelerde kullanılmamak koşuluyla) önek ya da sonek biçiminde bulunabilir. Aşağıdaki ifadeler geçerlidir:

```
b = fonk(++a);              /* sakıncasız */
...
b = fonk(a++);              /* sakıncasız */
```

Artırma ve eksiltme operatörlerinin operandı mutlaka bir nesne olmalıdır.

Örneğin:

```
10++;              /* 10 = 10 + 1, fakat 10 nesne değil */
...
a = ++b = c;       /* a = (b = b + 1) = c  fakat ++'dan elde edilen
                   değer nesne değil*/
```

Not: ++ ve — operatörlerinin gösterici operatörleriyle kullanımı 18.Bölümde ele alınmaktadır.

9.9 İLİŞKİSEL OPERATÖRLER

İlişkisel operatörler iki değer arasında karşılaştırma yapan operatörlerdir. Bu nedenle bu operatörlere karşılaştırma operatörleri de denir. İlişkisel operatörlerin hepsi iki operand alır. Aralarındaki öncelik ilişkisi aşağıdaki gibidir:

<	>	<=	>=	**Soldan sağa**
==	!=			**Soldan sağa**

Gördüğünüz gibi, toplam 6 ilişkisel operatör 2 öncelik grubu içinde bulunuyor. İlişkisel operatörler ile aritmetik operatörler arasındaki öncelik ilişkisinin de basit bir kuralı vardır:

Bütün ilişkisel operatörler aritmetik operatörlerden düşük önceliklidir.

Programlama dillerinin çoğunda mantıksal veri türleri vardır ve ilişkisel operatörlerin ürettiği değerler de bu türlerdendir. Oysa *C*'de mantıksal veri türü yerine yine **int** türü kullanılır. Mantıksal bir veri türünün tamsayı türüyle aynı olması *C*'ye esneklik ve doğallık kazandırmıştır.

Pascal Programcısına Not: *Pascal*'da mantıksal değişkenler BOOLEAN türündendir. BOOLEAN türünden bir değişkene ancak, TRUE ya da FALSE olabilen mantıksal değerler atanabilir. Bunun doğal sonucu olarak, Pascal'da ilişkisel operatörlerden üretilen değerler de BOOLEAN türünden olacaktır.

Örneğin:

```
VAR
X, Y: INTEGER;
A: BOOLEAN;
...
A :=  X > Y;       /* İlişkisel operatörün ürettiği değer mantıksal BOOLEAN
                   türündendir */
...
```

Dbase, Clipper ve Foxpro Programcılarına Not: *Dbase, Clipper, Foxpro* ve bunlarla uyumlu olan dillerde ilişkisel operatörlerin ürettiği değer, LOGICAL türündendir. LOGICAL türünden bir değişkende ancak TRUE ya da FALSE olabilen mantıksal değerler tutulabilir. Bu dillerde ilişkisel operatörlerin ürettiği değerler de ancak LOGICAL türünden değişkenlere atanabilir.

Örneğin:

```
LOGICAL A
DECIMAL  B, C
...
A = B > C          /* İlişkisel operatörün ürettiği değer LOGICAL türündendir */
...
```

C'de ilişkisel operatörlerin ürettiği değerler, "koşul sağlanıyorsa 1, sağlanmıyorsa 0" olan **int** türünden değerlerdir.

Örneğin:

```
a = 5 > 2;          /* a = 1 */
...
b = 3 == 3;         /* b = 1 */
...
c = 4 <= 1;         /* c = 0 */
...
```

Daha karmaşık örnekler:

```
x = 10;
y = 20;
z = y <= x + 10 ;
/* z = ? */
```

İşlemlerin yapılış sırası şöyledir:

```
İ1: x + 10    ——► 20    Aritmetik + operatörü öncelikli
İ2: y <= İ1   ——► 1     20 <= 20 koşulu sağlanıyor, üretilen değer 1
İ3: z = İ3    ——► 1     Atama en az öncelikli operatör
```

Sonuç:

```
z = 1
```

```
a = 10;
b = 20;
c = a == 10 < b + 1;
/* c = ? */
```

İşlemlerin yapılış sırası şöyledir:

```
İ1: b + 1     ——► 21    Aritmetik + operatör öncelikli
İ2: 10 < İ1   ——► 1     < operatörü == operatöründen daha öncelikli.
İ3: a == İ2   ——► 0     Koşul sağlanmıyor
İ4: c = İ3    ——► 0
```

Sonuç:

```
c = 0
```

Daha karışık bir örnek verelim:

```
a = 10 * 2 + 1 >= 4 * 4 + 5 == 5 > 5 - 3 * 2;
```

İşlemlerin yapılış sırası şöyledir:

```
i1: 10 * 2      ——► 20
i2: 4 * 4       ——► 16
i3: 3 * 2       ——► 6
i4: i1 + 1      ——► 21
i5: i2 + 5      ——► 21
i6: 5 - i3      ——► -1
i7: i4 >= i5    ——► 1       Koşul sağlanıyor
i8: 5 > i6      ——► 1       > operatörünün önceliği == operatö
                            ründen yüksek
i9: i7 == i8    ——► 1       Koşul sağlanıyor.
i10: a = i9     ——► 1
```

Sonuç:
`a = 1`

Uygulamalarda birden fazla ilişkisel operatör mantıksal operatör olmaksızın biraraya gelmez. Dolayısıyla, tam bir "*operatör salatası*" olan bu örnek yalnızca alıştırma amacıyla verilmiştir.

9.10 MANTIKSAL OPERATÖRLER

Operandları üzerinde mantıksal işlem yapan operatörlerdir. Bu operatörler operandlarını **Doğru (true)** ya da **Yanlış (false)** olarak yorumladıktan sonra işleme sokarlar. C'de öncelikleri birbirlerinden farklı üç tane mantıksal operatör vardır:

`!`	`Değil (not)`	Sağdan sola
`&&`	`Ve (and)`	Soldan sağa
`||`	`Veya (or)`	Soldan sağa

`! (Değil)` operatörü `tek operand` alır ve `her zaman` `önek` durumunda bulunur; diğer iki operatör ise iki operandlıdır.

9.10.1 C'de Mantıksal Doğru ve Yanlış Değerleri

Programlama dillerinin hepsinde mantıksal işlemler iki tür veri üzerinde gerçekleştirilirler: `Doğru` ve `Yanlış`. Mantıksal operatörlerin ürettikleri değerler de yine `Doğru` ya da `Yanlış` olabilen mantıksal değerlerdir. İlişkisel operatörleri açıklarken de değinmiştik: "*C' de mantıksal veri türü olmadığından, mantıksal veriler yerine tamsayılar kullanılmaktadır*".

C'de ilişkisel bir operatörün ürettiği değer, koşulun sağlanma durumuna göre 1 (Doğru) ya da 0 (Yanlış) sayılarıdır; ancak bir sayı mantıksal olarak yorumlanacağı zaman şu kural geçerlidir:

- Sayı sıfır ise Yanlış.

- Sayı sıfır dışı bir değer ise, yani negatif ya da pozitif herhangi bir sayı ise Doğru.

Örneğin, aşağıdaki sayıların mantıksal yorumlarını inceleyelim:

-11	Doğru
0	Yanlış
+99	Doğru
...	...

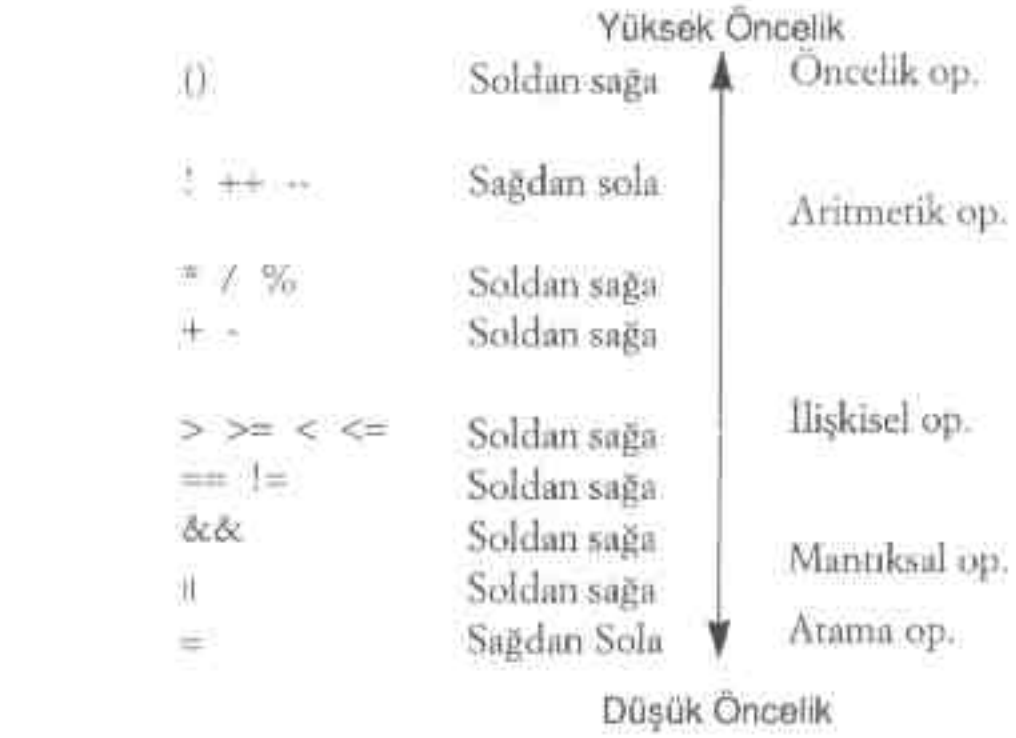
İlişkisel ve mantıksal operatörlerin işleme soktukları operandların ve işlem sonunda ürettikleri değerlerin türlerini şekilsel olarak aşağıdaki gibi özetleyebiliriz:

Bir ilişkisel operatör herhangi iki türden operand alabilir, fakat ürettiği değer **1** ya da **0** tamsayılarıdır. Bu tamsayılar mantıksal operatörler tarafından **Doğru** ya da **Yanlış** olarak yorumlanırlar. Mantıksal operatörlerin de ürettikleri değerler ilişkisel operatörlerde olduğu gibi **int** türünden **1** (Doğru) ya da **0** (Yanlış) sayılarıdır.

3.10.2 Mantıksal Operatörlerin Diğer Operatörlere Göre Öncelik Durumları

Değil operatörü (!) dışındaki iki mantıksal operatör, ilişkisel operatörlerden düşük önceliklidir. Değil operatörü ise en yüksek önceliğe sahip aritmetik operatörlerden artırma (++) ve eksiltme (—) ile **sağdan sola** eşit önceliklidir.

Şimdiye kadar incelediğimiz operatörlerin öncelik sıralarını aşağıdaki tablo ile özetleyebiliriz.



Şimdi yukarıda tanıttığımız üç mantıksal operatörü ve işlevlerini örneklerle tek tek açıklayacağız:

Değil (not) İşlemi ve ! Operatörü

Tek operand alan ve her zaman önek durumunda bulunan ! operatörü `Doğru` değerini `Yanlış` değere Yanlış değerini de Doğru değere dönüştürür.

a	Değil a
Doğru	Yanlış
Yanlış	Doğru

Örnekler:

```
a = !6;
```
⟶ 0 — 6 sıfır dışı bir değerdir, Doğru olarak yorumlanır, tersi Yanlıştır.

```
...
b = 10 + 4 > 5 + !2;
/* b = ? */
```

İşlem sırası:

```
!1: !2
```
⟶ 0 — En öncelikli ! operatörüdür

```
!2: 10 + 4
```
⟶ 14

```
!3: 5 + !1
```
⟶ 5

```
!4: !2 > !3
```
⟶ 1

```
!6: b = !4
```
⟶ 1 — En az öncelikli operatördür

Sonuç:
```
b = 1
```

```
...
a = 10
b = !++a > 10 != 5;
/* b = ? */
```

İşlem sırası:

```
İ1: ++a        → 11
İ2: !İ1        → 0
İ3: İ2 > 10    → 0
İ4: İ3 != 5    → 1
İ5: b = İ4     → 1
```

! ve ++ sağdan sola eşit önceliklidir

!İ1

Sonuç:
```
b = 1
```

```
...
a = 0;
x = !!a;
```

İşlem sırası:

```
İ1: !a         1
İ2: !İ1        → 0
İ3: x = İ2     → 0
               →
```

Yanlışın değili

Değilin değili yine kendisidir.

Sonuç:
```
x = 0
```

Ve (and) İşlemi ve && Operatörü

Bu operatör, ilişkisel operatörlerin hepsinden düşük önceliklidir. Operandlarının `İkisi de Doğru` ise `Doğru (1)` değerini, operandlardan bir tanesi `Yanlış` ise `Yanlış (0)` değerini üretir. Tablo biçiminde özetlersek:

a	b	a Ve b
Yanlış	Yanlış	Yanlış
Yanlış	Doğru	Yanlış
Doğru	Yanlış	Yanlış
Doğru	Doğru	Doğru

```
a = 4 && 0          /* a = 0*/
...
b = 10 && -4        /* b = 1 */
...
c = 0 && 0          /* c = 0 */
```

&& operatörünün önce sol tarafındaki işlemler öncelik sırasına göre tam olarak yapılır. Eğer bu işlemlerde elde edilen sayısal değer **0** ise sağ tarafındaki işlemler yapılmaz.

Örneğin;

```
a = 5;
b = !a > 10 && fonk() == 10;
/* b = ? */
```

İşlem Sırası:

```
İ1: !a          ——► 0
İ2: a + 5       ——► 10
İ3: İ1 > 10     ——► 0
```

Artık sağ tarafındaki işlemlerin yapılmasına gerek yoktur. Dolayısıyla **fonk** fonksiyonu da çağrılmayacaktır.

Sonuç:
`b = 0`

Uygulamada mantıksal operatörler, ilişkisel operatörlerle birlikte kullanılırlar:

```
...
a = 15;
x = a >= 10 && a <= 20;
/* x = ? */
```

İfadesinde eğer `a`, `10` ile `20` arasındaysa `Doğru (1)`, değilse `Yanlış (0)` değeri üretilecektir. Çünkü sonucun `Doğru` olabilmesi için `Ve` operatörünün iki operandının da `Doğru` olması gerekir ki, bu da ilişkisel operatörlere ilişkin koşulların sağlanmasıyla mümkün olabilir.

İşlem sırası:

```
İ1: a >= 15     ——► 1
İ2: a<= 20      ——► 1
İ4: İ1 && İ2    ——► 1
İ5: x = İ4      ——► 1
```

Sonuç:
`x = 1`

Tek tırnak (single quote) içerisindeki karakterlerin, aslında *ASCII* tablosunda sıra numarası gösteren sayılar olduklarını anımsayınız.

```
...
ch = 'c';
z = ch >= 'a' && ch <= 'z';
/* z = ? */
```

İfadesi, ch değişkenin içindeki karakterin küçük harf olup olmadığını test etmek amacıyla kullanılmıştır. Çünkü ch değişkenin içindeki sayı (c harfinin *ASCII* sıra numarası) a harfinin *ASCII* karşılığından büyük ya da eşit ve z harfinin *ASCII* karşılığından küçük ya da eşit ise o zaman && operatörü `Doğru (1)` sonucunu üretecektir.

```
İ1: ch >= 'a'   ──►1      99 (c'nin ASCII karşılığı) >= 97
                          ('a' nın ASCII karşılığı)
İ2: ch <= 'z'   ──►1      99 (c'nin ASCII karşılığı) <= 122
                          ('z' nin ASCII karşılığı)
İ3: İ1 && İ2    ──►1      ch küçük harftir.
İ4: z = İ3      ──►1
```

Sonuç:
`z = 1`

Bu ifade *Türkçe* karakterler için geçerli değildir. Çünkü *Türkçe* karakterlerin *ASCII* tablosundaki yerleri ikinci yarı bölgededir (128 - 255).

Veya (or) İşlemi ve || Operatörü

Önceliği en az olan mantıksal operatördür. İki operandının biri `Doğru` ise `Doğru`, ikisi de `Yanlış` ise `Yanlış` değerini üretir. Doğruluk tablosu aşağıdaki gibidir:

a	b	a Veya b
Yanlış	Yanlış	Yanlış
Yanlış	Doğru	Doğru
Doğru	Yanlış	Doğru
Doğru	Doğru	Doğru

```
a = 3 || 0          /* a = 1 */
...
b = 0 || -30        /* b = 1 */
...
c = 0 || 0          /* c = 0 */
```

|| operatörünün önce sol tarafındaki işlemler öncelik sırasına göre tam olarak yapılır. Eğer bu işlemlerden elde edilen sayısal değer **1** ise sağ tarafındaki işlemler yapılmaz.

Veya operatörünün diğer operatörlerle birlikte kullanılmasına ilişkin birkaç örnek veriyoruz:

```
...
a = 20;
b = 10;
y = a + b >= 20  || a - b <= 10;
```

Yukarıdaki ifadede a ile b değişkenlerinin toplamı 20'den büyükse ya da farkları 10'dan küçükse Doğru değeri üretilir.

İşlem sırası:

```
i1: a + b      ——► 30
i2: i1 >= 20   ——► 1
```

Artık sağ taraftaki işlemlerin yapılmasına gerek yoktur. Dolayısıyla a - b <= 10 işlemi yapılmayacaktır.

Sonuç:
```
y = 1;
```

Aşağıdaki ifade ise, **ch** değişkenin içinde bulunan karakterin, "Türkçe karakterler de dahil olmak üzere" küçük harf olup olmadığını bulur.

```
...
ch = 'ş';
a = ch >= 'a' && ch <= 'z' || ch == 'ı' || ch == 'ü' ||
ch == 'ğ' || ch == 'ç' || ch == 'ö'|| ch == 'ş';
```

İfadenin Doğru olabilmesi için yalnızca bir Doğru yeter...

İşlem sırası:

`i1: ch >= 'a'`	1	**158** (ş'nin *ASCII* karşılığı) 97'den (a'nın ASCII karşılığı) büyüktür.
`i2: ch <= 'z'`	0	**158** (ş'nin ASCII karşılığı) 122'den (z'nin ASCII karşılığı) küçük değildir.
`i3: ch == 'ı'`	0	
`i4: ch == 'ü'`	0	
...		
`in: ch == 'ş'`	1	`ch = 'ş', 158 == 158.`
...		

Sonuç
```
a = 1
```

9.11 BİT OPERATÖRLERİ

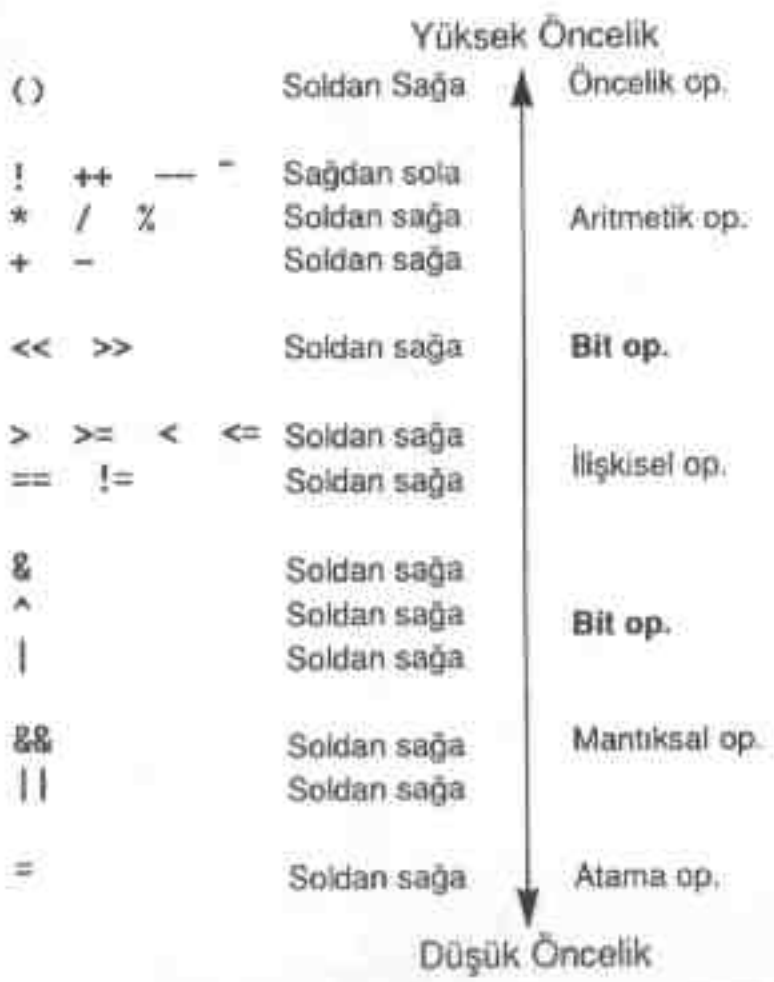
Bit operatörleri bit düzeyinde işlem yapan operatörlerdir; yani sayıları bir bütün olarak değil, bit bit ele alarak işleme sokarlar. Bit operatörlerinin aldığı operandların tamsayı türlerinden olması gerekir (char, short, int, long). **float** ya da **double** türleri bu operatörlerle işleme sokulamaz. Bit operatörlerinin listesi, "kendi aralarındaki öncelik sırası gözönüne alınarak" aşağıda verilmiştir:

~	**Bit Değil (bitwise not)**
<< >>	**Sola ve sağa öteleme (bitwise left, bitwise right shift)**
&	**Bit Ve (bitwise and)**
^	**Bit Özel Veya (bitwise exor)**
|	**Bit Veya (bitwise or)**

`Bit Değil` **(bitwise not)** operatörü (~) tek operandlıdır ve yalnızca önek durumunda bulunur. Bunun dışındaki tüm bit operatörleri iki operand almaktadır. Bit operatörleri uygulamalarda genellikle *"sayıların bit durumlarını öğrenmek"* veya *"diğer bitlere dokunmadan çeşitli bitlerin değerlerini değiştirmek" amacıyla kullanılır.*

9.11.1 Bit Operatörlerinin Diğer Operatörlere Göre Öncelik Durumları

Bit operatörleri öncelik tablosunda sıralı bir grup olarak bulunmazlar. Bu nedenle diğer operatörlere göre öncelik durumlarını *"şimdiye kadar gördüğümüz operatörleri dikkate alarak"* yine bir tablo yardımıyla vereceğiz:



Tabloda da gördüğünüz gibi, bit operatörlerinin bir kısmı aritmetik operatörlerle ilişkisel operatörler arasında, bir kısmı ise ilişkisel operatörlerle mantıksal operatörler arasında bulunuyorlar. **Mantıksal Değil (!)** operatöründe olduğu gibi, **Bit Değil (~)** operatörünün de önceliği hepsinden fazladır.

Bit Değil (~) Operatörü

Tek operand alan ve her zaman önek durumunda olan bu operatör, 1'e tümleme işlemi yapar. Yani operandı olan sayıdaki **1** olan bitleri **0**; **0** olan bitleri de **1**'e dönüştürür.

Örneğin:

```
unsigned int x = 0x1834;
unsigned int y;
...
y = ~x;
/* y = ? */
```

Önce x içerisindeki sayıyı bit bit ifade edelim:

```
x    =        0001 1000 0011 0100
```

1'leri **0**; **0**'ları **1** yaparsak:

```
~x = 1110 0111 1100 1011
       E    7    C    B
```

sayısını elde ederiz.

80X86 Sembolik Makina Dili Programcısına Not: Bit Değil işlemi `NOT {reg, r/m}` makina komutuna karşılık gelmektedir.

Bit Ve (&) Operatörü

Karşılıklı bitleri **Ve** işlemine tabi tutar. **Ve** işleminin doğruluk tablosunu mantıksal operatörleri anlatırken vermiştik.

```
x = 0xB8;
y = 0x1C;
...
z = x & y;
/* z = ? */
```

x ve y değişkenlerinin bit durumlarını inceleyelim:

```
x  = 1011 1000
       B    8

y = 0001 1100
      1    C
```

Karşılıklı bitler üzerinde **Ve** işlemi yapılırsa:

```
1011 1000     x
0001 1100     y
---------------
0001 1000     x & y
  1    8
```

elde edilir.

1 -tıpkı çarpma işleminde olduğu gibi- **Ve** işleminde etkisiz elemandır. **x** bir tamsayı olsun:

```
x = 0x...;
y = x & 0xFFFF;
```

işlemi **y = x** ile aynı anlamdadır.

0 - tıpkı çarpma işleminde olduğu gibi- **Ve** işleminde yutan elemandır. **a** bir tamsayı olsun:

```
a = 0x...;
y = a & 0;
```

işlemiyle **y** değişkenine **0** atanır.

80X86 Sembolik Makina Dili Programcısına Not: Bit Ve işlemi `AND {reg} {r/m}` makina komutuna karşılık gelmektedir.

Bit Veya (|) Operatörü

Karşılıklı bitler üzerinde **Veya** işlemi uygular. **Veya** işleminin de doğruluk tablosunu ilişkisel operatörlerin anlatıldığı bölümde vermiştik.

```
x = 0x1FC0;
y = 0x153A;
...
z = x | y;
```

x ve **y** değişkenlerini bitlerine ayrıştıralım:

```
x = 0001 1111 1100 0000
      1    F    C    0

y = 0001 0101 0011 1010
      1    5    3    A
```

Karşılıklı bitler Veya işlemine tabi tutulursa:

```
0001 1111 1100 0000     x
0001 0101 0011 1010     y
-------------------------
0001 1111 1111 1010     x | y
 1    F    F    A
```

0 -tıpkı toplama işleminde olduğu gibi- **Veya** işleminde de etkisiz elemandır.

80X86 Sembolik Makina Dili Programcısına Notu: Bit Veya operatörü `OR (reg) (r/m)` makina komutuna karşılık gelmektedir.

Bir sayının **"diğer bitlerine dokunmadan bazı bitlerinin istenilen değerlerle değiştirilmesi"** bit operatörlerinin en önemli uygulama alanıdır. Örneğin, a bir tamsayı olmak üzere 3 ve 4 numaralı bitleri sırasıyla 1, 0 ile değiştirmek isteyelim.

Böyle bir işlemi 2 adımda gerçekleştirebiliriz:

1. Adım: a'nın içerisindeki sayıyı bilmiyoruz. Önce diğer bitlere dokunmadan yalnızca 3 ve 4 numaralı bitleri 0 yaparız. Bunun için a değişkenini uygun bir sayıyla `Bit Ve` işlemine tabi tutmak gerekir.

a

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

&

0xFFE7

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1
F				F				E				7			

=

a & 0xFFE7

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	0	0	x	x	x

a değişkenini `0xFFE7` ile `Bit Ve` işlemine tabi tutuğumuzda, 3. ve 4. bitler 0 olur, fakat diğer bitler değerlerini değiştirmez.

2. Adım: Elde ettiğimiz sayıyı, 3 ve 4 numaralı bitleri istediğimiz gibi değiştirebilecek bir sayı ile bu kez `Bit Veya` işlemine sokmamız gerekir.

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0				0				0				8			

Bu iki adımı aynı ifade içerisinde yazarsak:

```
a = a & 0xFFE7 | 0x0008;
```

Bu örnekte anlaşılması en zor olan nokta 1. Adımın gerekliliğidir. Eğer bir tereddütünüz varsa, kalem ve kağıt ile bunu gidermeye çalışmalısınız.

ASCII karakter tablosunu dikkatle inceleyin; büyük harflerden sonra hemen küçük harflerin gelmediğini göreceksiniz! İyi ama neden?..

Karakter	Hex (Desimal)	İkilik Sistem
A	41 (65)	0100 0001
B	42 (66)	0100 0010
C	43 (67)	0100 0011
...	...	...
Z	5A (90)	0101 1010
[	5B (91)	0101 1011
\	5C (92)	0101 1100
...	...	...
a	61 (97)	0110 0001
b	62 (98)	0110 0010
c	63 (99)	0110 0011
...	...	...

`'a' - 'A' = 32` olması bir rastlantı değil, tasarımın bir özelliğidir. Büyük ve küçük harfler arasında yalnızca 5 numaralı bit farklıdır: Küçük harf ya da büyük harf olduğunu bildiğimiz karakterlerin 5. bitinin değerini değiştirirsek, büyük harfi küçük harfe, küçük harfi de büyük harfe dönüştürmüş oluruz.

```
a = 'A';
ch = a + 32;          /* ch = 'a' */
```

ifadesi ile

```
a = 'A';
ch = a | 32;          /* ch = 'a' */
```

ifadesi aynı anlama gelir.

Bit Özel Veya (^) Operatörü

Bu operatör de iki operand alır. Operandları aynı ise **0**, farklı ise **1** değerini üretir. Doğruluk tablosunu inceleyiniz:

a	b	a Özel Veya b
Yanlış	Yanlış	Yanlış
Yanlış	Doğru	Doğru
Doğru	Yanlış	Doğru
Doğru	Doğru	Yanlış

Özel Veya operatörü farklı olan bitleri belirlemek için kullanılabilir.

```
a = 0x15C4;
b = 0x12C5;
...
c = a ^ b;
```

a bilgisi değişerek **b** biçimine dönüşmüş olsun; **a**'nın acaba hangi bitleri değişmiştir?

```
0001 0101 1100 0100     a
0001 0010 1100 0101     b
-------------------------------
0000 0111 0000 0001     a ^ b
```

1'lerin bulunduğu yerler, değişen bitleri gösteriyor. Bir değişkeni kendisi ile `Özel Veya` işlemine sokarsanız ne olur? Yanıt koca bir sıfır.

```
a = a ^ a;                    /* a = 0 */
```

80X86 Sembolik Makina Dili Programcısına Not: Özel Veya işlemi `XOR (reg, r/m)` makina komutuna karşılık gelmektedir.

9.12 ÖTELEME İŞLEMLERİ

C'de sola ve sağa öteleme yapmak için iki ayrı bit operatörü bulunur. Öteleme işlemleri özellikle bitlerin sayı içerisindeki pozisyonlarını değiştirmek amacıyla kullanılmaktadır. *C*'deki iki öteleme operatörü de iki operand alır. Diğer bit işlemlerinde olduğu gibi öteleme işlemlerinde de işleme giren operandlar değişmezler, işlem sonucunun başka bir değişkene atanması gerekir.

Sola Öteleme (<<) Operatörü

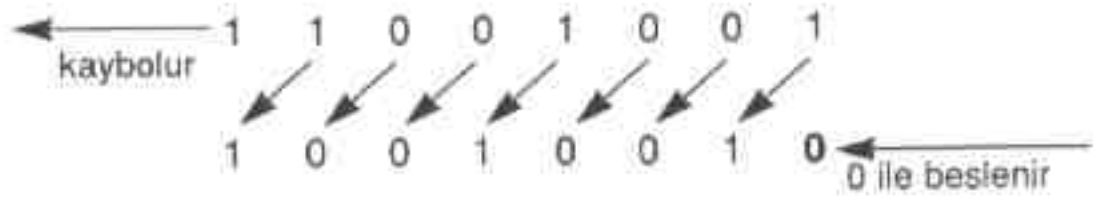
Sola öteleme operatörünün sol tarafındaki operand öteleme işleminin yapılacağı değişkeni ya da sabiti, sağ tarafındaki operand ise öteleme sayısını göstermektedir.

```
değişken << öteleme sayısı
```

Sola öteleme işleminde bütün bitler öteleme sayısı kadar sola kaydırılır. Kaydırma işlemi sonunda en sağdaki bit sürekli sıfırla beslenir, en soldaki bit ise menzil dışına çıktığı için kaybolur. Örneğin:

```
a = 0xC9;                     /* 1100 1001 */
...
b = a << 1;
```

gibi bir ifade sonucunda `C9` sayısı bir kere sola ötelenir:



Sola bir kez öteleme sayıyı iki ile çarpma anlamına gelir:

```
a = 2;                  /* 0000 0010 = 2 */
...
a = a << 1;             /* 0000 0100 = 4 */
```

Bir'den fazla öteleme, aynı işlemin bir'den fazla yineleneceği anlamına gelir. Örneğin:

```
a = 10 olsun;

a                 ──► 0000 1010
a << 1            ──► 0001 0100
a << 2            ──► 0010 1000
a << 3            ──► 0101 0000
...                   ...
```

Sağa Öteleme (>>) Operatörü

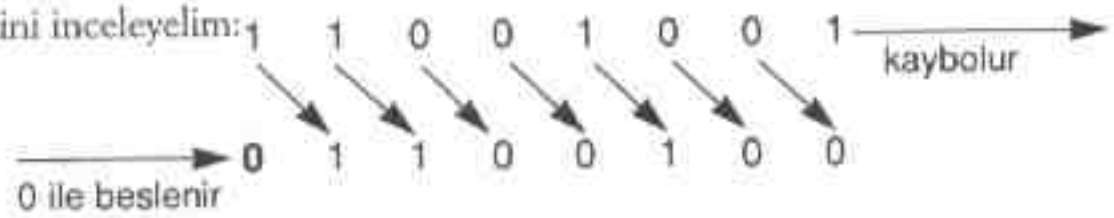
Sağa ötelemenin sola ötelemeden farkı yönünün ters olmasıdır. Sağa ötelemede sayı içerisindeki bütün bitler sağa kaydırılır.

```
değişken >> öteleme sayısı
```

Sağa 1 kez öteleme sonucunda sayının en sağındaki bit kaybolur, en solundaki bit ise 0 ile beslenir. Örneğin:

```
0xC9 >> 1
```

işlemini inceleyelim:



Sağa 1 kez öteleme, 2'ye bölme anlamına gelir.

```
a = 10;
...
a = a >> 1; /* a = 5 */
```

İşlemi sonucunda a değişkenin içindeki sayı 2'ye bölünmektedir (Tabi, burada tam bölme kastedilmektedir).

```
a = 10              0000 1010      ──► 10
...                 ...
a = a >> 1          0000 0101      ──► 5
```

bir kez daha ötelersek:

```
a = a >> 1          0000 0010      ──► 2
```

Sağa ve sola öteleme işlemleri uygulamalarda, **belirli bitlerin pozisyonunu değiştirmek** amacıyla `Bit Ve` ya da `Bit Veya` operatörleriyle birlikte kullanılır. Örneğin, bir tamsayının 5. ve 6. bitlerinin sayı değerini bulmak isteyelim. Bu durumda en anlaşılır yol, sayıyı önce 5 kez sağa ötelemek ve sonra da uygun sayıyla `Bit Ve` işlemine tabi tutmaktır.

```
int a = 0x157C;
...
b = a >> 5 & 3;
```

a içerisindeki sayının bit durumları:

```
a = 0001  0101  0111  1100
      1     5     7     C
```

İşlemlerin yapılış sırası adım adım aşağıda verilmiştir:

```
İ1: a >> 5    0000 0000 1010 1011
İ2: İ1 & 3    0000 0000 1010 1011 & 0000 0000 0000 0011
= 0000 0000 0000 0011
İ3: b = İ2    0000 0000 0000 0011
```

Öteleme operatörlerinin `Bit Ve` ile `Bit Veya` operatörlerinden daha yüksek öncelikli olduğunu unutmayınız.

```
a = b & c >> 2;
```

ifadesinde önce `c >> 2` işlemi yapılır.

```
İ1: c >> 2
İ2: b & İ1
İ3: a = İ2
```

Önceliği değiştirmek için `(...)` operatörünü kullanabilirsiniz.

```
a = (b & c) >> 2;

İ1: b & c
İ2: İ1 >> 2
İ3: a = İ2
```

80X86 Sembolik Makina Dili Programcısına Not: Sola öteleme işlemi `SAL (r/m) (1 ya da cl)` ve `SHL (r/m) (1 ya da cl)` makina komutlarına karşılık gelir. `SAL (sola aritmetik öteleme)` ve `SHL (sola mantıksal öteleme)` komutlarının işlem kodları aynıdır; dolayısıyla aynı komutlardır. Öteleme işlemleri 1 defadan fazla yapılacaksa, operand olarak `cl` yazmacı kullanılmalıdır. Sola öteleme, ikinin katlarıyla çarpım işlemlerinde daha hızlı olduğu için `cl` komutuna karşı tercih edilmektedir. Sağa öteleme komutları olan `SHR (r/m) (1 ya da cl)` ve `SAR (r/m) (1 ya da cl)` ise birbirlerinden farklıdır. `SHR (sağa mantıksal öteleme)` sayıyı işaretsiz olarak ele alır; oysa `SAR (sağa aritmetik öteleme)` işaret bitini koruyacak biçimde, yani işaretli öteme yapmaktadır. Yine bu iki komut da `DIV` makina komutundan daha hızlı olduğundan ikinin katlarına bölme durumlarında tercih edilir.

9.13 GÖSTERİCİ OPERATÖRLERİ

Adresler üzerinde işlemler yapan operatörlerdir. Gösterici operatörleri olarak sınıflandırdığımız `*`, `&`, `[]` operatörleri *"göstericiler"*, `->` operatörü ise yapılar konusunda ele alınmaktadır.

9.14 ÖZEL AMAÇLI OPERATÖRLER

Bu bölümde özel amaçlı operatörlerden atama, işlemli atama, öncelik ve virgül operatörlerini inceleyeceğiz. Özel amaçlı operatörlerin çoğu -gösterici operatörlerinde olduğu gibi- karmaşık konularla ilişkilidir. Bu nedenle diğer özel amaçlı operatörlerin burada ele alınmasının faydalı olmayacağını düşünüyoruz.

9.14.1 Atama (=) Operatörü

Örneklerimizde sıkça kullandığımız atama operatörü, C'nin en düşük öncelikli ikinci operatörüdür. Virgül operatörü dışındaki tüm operatörler atama operatöründen daha yüksek önceliğe sahiptir. Her operatörün bir değer ürettiğini biliyorsunuz. Atama operatörü de bir değer üretmektedir. **Atama operatörünün ürettiği değer sağ taraf değerinin kendisidir.** Örneğin:

```
if (a = 10) {
...
}
```

Burada sırasıyla şunlar olur:

1) 10 sayısı `a` değişkenine atanır.

2) Bu işlemden 10 sayısı üretilir.

3) 10 sayısı sıfır dışı bir değer olduğu için `if` deyimi Doğru olarak değerlendirilir. (`if` için `a == 10` olması daha normal gözüküyor.)

Atama operatörünün ürettiği değer nesne değildir. Eğer nesne olsaydı aşağıdaki ifade geçerli olurdu!

```
(b = c) = a;                    /* Hata! sol taraf değeri yok */
```

`b = c` atamasından elde edilen değer `c` nesnesinin kendisi değildir. `c`'nin sayısal değeridir.

Atama operatörü kendi arasında sağdan sola öncelikli bir operatördür. Örneğin:

```
a = b = c = 10;
```

işlemi ile `a`, `b` ve `c` değişkenlerinin hepsine 10 sayısı atanır.

```
İ1: c = 10 ──►10
İ2: b = İ1 ──►10
İ3: a = İ2 ──►10
```

9.14.2 İşlemli Atama Operatörleri

Bir işlemin operandı ile işlem sonucunda atanan nesne aynı ise işlemli atama operatörleri kullanılabilir.

`<nesne1> = <nesne1> işlem <nesne2>` ile `<nesne1> işlem= <nesne2>` aynı anlamdadır.

İşlemli atama operatörleri atama operatörüyle sağdan sola eşit önceliğe sahiptir.

```
=  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=        Sağdan sola
```

İşlemli atama operatörleri hem okunabilirlik için hem de daha kısa yazım için tercih edilirler. Aşağıdaki ifadeler eşdeğerdir:

```
a += 2;          a = a + 2;
b -= 10;         b = b - 10;
c *= 10;         c = c * 10;
```

```
scrp += row * 160 + col * 2;
```

ifadesinde işlem sırası aşağıdaki gibidir:

```
İ1: row * 160
İ2: col * 2
İ3: İ1 + İ2
İ4: scrp += İ3      scrp = scrp + İ3
```

Aşağıdaki ifade hakkında ne düşünürsünüz?

```
a += b += 2;
```

Bu ifade:

```
a = a + (b = b + 2);
```

anlamına gelmiyor mu? İfade geçerlidir.

9.14.3 Virgül (,) Operatörü

İki ayrı ifadeyi tek bir ifade olarak birleştiren virgül operatörü C'nin en düşük öncelikli operatörüdür.

```
ifade1;
ifade2;
```

ile

```
ifade1, ifade2;
```

```
...
}
```

birbirine benzeyen farklı ifadeleri tek bir ifade altında birleş

```
...
```

virgül operatörünü birbirine benzeyen farklı ifadeleri tek bir ifade altında birleştirmek için kullanabilirsiniz. Örneğin:

```
if (ifade) {
     x1 = 1;
     x2 = 2;
     x3 = 3;
}
```

ile

```
if (ifade)
     x1 = 1, x2 = 2, x3 = 3;
```

eşdeğer işlevlere sahiptir.

9.14.4 Öncelik Operatörü

Diğer programlama dillerinden tanıdığınız `öncelik operatörü (...)`, bir ifadenin önceliğini yükseltmek amacıyla kullanılmaktadır.

```
a = (b + c) * d;
```

`b + c` ifadesi öncelik operatörüyle kullanıldığı için ilk işlem olarak yapılacaktır. Öncelik operatörünün kendisi aynı zamanda **"C'nin en yüksek önceliğe sahip operatörleri"** arasındadır. Öncelik operatörü de kendi arasında soldan sağa öncelik kuralına uyar. Örneğin:

```
a = (x + 2) / ((y + 3) * (z + 2) - 1);
```

ifadesinde işlem sırası şöyledir:

```
i1: x + 2
i2: y + 3
i3: z + 2
i4: i2 * i3
i5: i4 - 1
i6: i1 / i5
i7: a = i6
```

9.15 İFADELERİN OKUNABİLİRLİĞİ ÜZERİNE...

C'de operatörler arasındaki öncelik ilişkileri oldukça karmaşıktır. Bu nedenle, özellikle farklı işlevlere sahip operatörlerin birarada kullanıldığı ifadelerde okunabilirlik **(readability)** azalmaktadır. Böylesi durumlarda, hiç gerekmese bile ifadelerin okunabilirliğini artırmak amacıyla, öncelik operatörünün kullanılması salık verilir. Örneğin size soralım:

```
x = a + b < c + d || a - b > x + 10;
```

ifadesi mi daha kolay algılanıyor? Yoksa:

```
x = (a + b) < (c + d) || (a - b) > (x + 10);
```

ifadesi mi? Aslında ikinci ifadedeki parantezlere hiç gerek yoktur. Başka bir örnek:

```
x = a & (b << 4);
```

Bu ifadede de öncelik operatörünün kullanılmasına gerek yoktur; çünkü **sola öteleme** işlemi, `Bit Ve` işleminden zaten daha önceliklidir.

Bazı programcılar mümkün olduğunca az parantez (öncelik operatörü) kullanmak isterler (?). Nedendir bilinmez! (Operatörler arasındaki öncelik ilişkilerini çok iyi bildiklerini göstermek istiyor da olabilirler). Oysa ne fazla, ne az; kararını bilmek gerekir. Çabuk algılanabilen, ikilemde bırakmayan ve taşınabilir kodları tercih etmelisiniz.

SORAMADIKLARINIZ...

S1) Neden mantıksal ya da bit düzeyindeki `Ve (and)` işlemi, `Veya (or)` işleminden daha önceliklidir?

C1) `Ve` işlemi çarpma `Veya` işlemi toplama gibidir. `0` yanlış, `1` doğru olsun:

a	b	a & b	a | b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1 (2)

Çarpma işleminin toplama işlemine göre bir önceliği vardır, değil mi?

S2) `<<, !=, ==,...` gibi iki sembolden oluşmuş operatörlerin arasına boşluk karakterleri koyabilir miyiz? Örneğin:

```
b = a ! = 2;
```

olabilir mi?

C2) `<<, !=, ==,...` gibi operatörler tek bir atomdur. Eğer arasına boşluk karakterlerinden birisini koyarsanız (white space), o zaman derleyici bu iki sembolü iki ayrı atom olarak ele alır:

b = a ! = 2;

Bu ifade, iki ayrı atom olarak ele alındığında ise yazım hatası (syntax error) olarak değerlendirilecektir. Çünkü ! tek operand alan önek bir operatördür. Boşluk karakterlerinin atom ayıracı (token delimiter) olarak kullanıldığını unutmayınız.

S3) *C*'de **üs alma** operatörü var mıdır?

C3) *C*'de **üs alma** diye bir operatör yoktur. Üstel işlemler bir operatör ile değil, fonksiyon yardımıyla yapılır. Bu iş için standart *C* kütüphanelerinde bulunan `pow` fonksiyonu kullanabilirsiniz.

```
double pow(double base, double exp);
```

`pow` fonksiyonun geri dönüş değeri, $base^{exp}$ işleminin sonucudur.

10

if DEYİMİ

Bu bölümde program akışını yönlendirmekte kullanılan `if` deyimi ele alınmaktadır. `if` deyiminin hemen hemen tüm varyasyonları bölüm içerisinde ayrıntılı bir biçimde gözden geçirilmiş, bölümün sonunda da birkaç uygulama üzerinde durulmuştur. Uygulama konularını standart *C* fonksiyonları arasından seçmeye özen gösterdik. Böylece, uygulama yaparken aynı zamanda standart *C* fonksiyonlarının işlevlerini de daha iyi anlayacağınızı düşünüyoruz.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Deyim nedir?
2) Deyimler kaç gruba ayrılırlar?
3) `if` deyimi nasıl çalışır?
4) Karakter test fonksiyonları nelerdir ve ne amaçla kullanılırlar?

10.1 DEYİM NEDİR?

III. Bölümde ifade (expression) kavramını açıklamıştık. İfadenin tanımını anımsayalım:

Operatörlerin, değişkenlerin ve sabitlerin kombinasyonlarına ifade denir.

C'de ifadeler birbirleriyle noktalı virgül ile ayrılırlar. İfadeleri bitirmesi nedeniyle, noktalı virgüle **sonlandırıcı (terminator)** da denilmektedir. Biz de kitabımızın bundan sonraki bölümlerinde noktalı virgül yerine **sonlandırıcı** terimini kullanacağız.

Deyim (statement) ifade ile sonlandırıcıdan oluşan atom grubudur.

Örneğin:

```
c = a * b - d
```

bir ifadedir; ancak

```
c = a * b - d;
```

bir deyimdir. Bir deyimi şöyle de belirtebiliriz:

```
ifade;
```

C'de deyimler 4 bölüme ayrılır:

1) Yalın deyimler

Bunlar bir ifadenin sonuna **sonlandırıcı** (;) getirilerek oluşturulan deyimlerdir. Yalın deyimler:

```
ifade;
```

biçimindedir. Örneğin:

```
a = b + 3;
c++;
c = getch();
...
```

gibi...

2) Bileşik deyimler

Bileşik deyimler birden fazla deyimin bir blok içinde toplanmasıyla oluşur. C'de bloklar aynı zamanda bileşik deyimlerdir.

```
{
        deyim1
        deyim2
        deyim3
        ¼..
}
```

deyim1, deyim2, deyim3,... bileşik deyimi oluşturan deyimlerdir. Daha somut bir örnek verebiliriz:

```
...
{
        a = b * c - d;
        b++;
}
...
```

3) Bildirim deyimleri

Bildirim amacıyla oluşturulmuş deyimlerdir:

```
int a, b, c;
...
char x;
...
float kilo, boy;
...
```

gibi...

4) Kontrol deyimleri

Program akışını kontrol etmek için kullanılan deyimlerdir. Kontrol deyimleri en az bir anahtar sözcük içerir; bir ya da birden fazla blokdan oluşabilir.

```
if (ifade) {
        deyim1
        deyim2
}
else {
        deyim3
        deyim4
}
...

switch(a)
{
        case 1: deyim11,... break;
        case 2: deyim21,... break;
        case 3: deyim31,... break;
        ....
        default: deyimN ;
}
...
```

Solunda bir ifade olmaksızın yalnızca bir sonlandırıcıdan oluşan deyime, **boş deyim (null statement)** denir. Derleyiciler boş deyimler için hiçbir işlem yapmazlar.

Örneğin:

```
a = b * 2;              /* yalın deyim */
;                       /* boş deyim */
c = a + 4;              /* yalın deyim */
```

10.2 if DEYİMİ

C'de program akışını kontrol etmeye yönelik en önemli deyim, `if` deyimidir. `if` deyiminin genel biçimi aşağıda verilmiştir.

```
if (ifade)
        deyim1
else
        deyim2
...
```

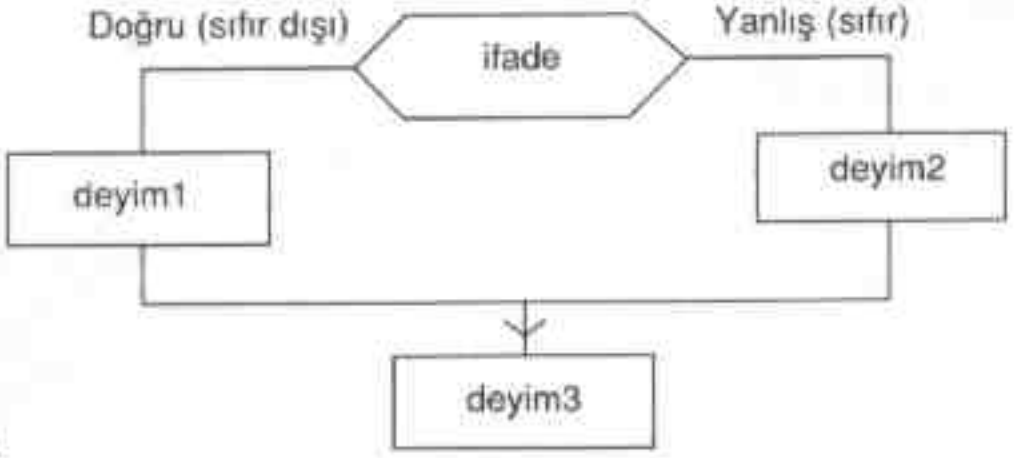
deyim1 ve **deyim2,** yalın ya da bileşik olabileceği gibi başka bir kontrol deyimi de olabilir.

`if` deyiminin icrasında, önce derleyici `if` parantezinin içindeki ifadenin sayısal değerini hesaplar. Hesapladığı bu sayısal değeri mantıksal Doğru ya da Yanlış olarak yorumlar (0 ise Yanlış, 0 dışında bir değer ise Doğru). Eğer ifadenin sonucu Doğru ise, `else` anahtar sözcüğüne kadar olan kısım; Yanlış ise, `else` anahtar sözcüğünden sonraki kısım icra edilir.

Akış diyagamı ile gösterirsek:

```
if (ifade)
        deyim1
else
        deyim2
deyim3
...
```

deyim3, `if` deyiminin dışında bulunan ilk deyimdir:



Örneğin:

```
a = 8;
...
if (a * 5 < 50)
        deyim1
else
        deyim2
deyim3
...
```

1. Adım: `if` parantezinin içindeki ifadenin sayısal değeri hesaplanır.

```
i1: a * 5   ──► 40
i2: i1 < 50 ──► 1
```

Sonuç: DOĞRU

2. Adım: İfade Doğru olduğuna göre, **`else`** anahtar sözcüğüne kadar olan kısım (deyim1) yapılır.

3. Adım: `else` kısmı atlanarak **deyim3** yapılır.

`if` parantezi içindeki ifadeler daha karmaşık olabilirler:

```
ch = 'A';
...
if (ch >= 'a' && ch <= 'z')
        deyim1
else
        deyim2
deyim3
```

Yukarıdaki `if` deyiminde `ch` karakterinin küçük harf olup olmadığı test edilmektedir. Eğer `ch` küçük harf ise **deyim1**, küçük harf değilse **deyim2** yapılır; daha sonra akış **deyim3** ile devam eder.

1. Adım: `if` parantezinin içindeki ifadenin sayısal değeri bulunur.

```
i1: ch >= 'a' ──► 0
i2: ch <= 'z' ──► 0
i3: i1 && i2  ──► 0
```

Sonuç: YANLIŞ

2. Adım: Sonuç yanlış olduğu için `else` kısmı (deyim2) yapılır.

3. Adım: **Deyim3** ile akışa devam edilir.

`if` deyiminin Doğru ve/veya Yanlış kısmı birden fazla deyimden oluşuyorsa (bileşik deyimse) bloklama yapılmalıdır.

```
if (a < b - 4) {
        deyim1
        deyim2
}
else {
        deyim3
        deyim4
}
```

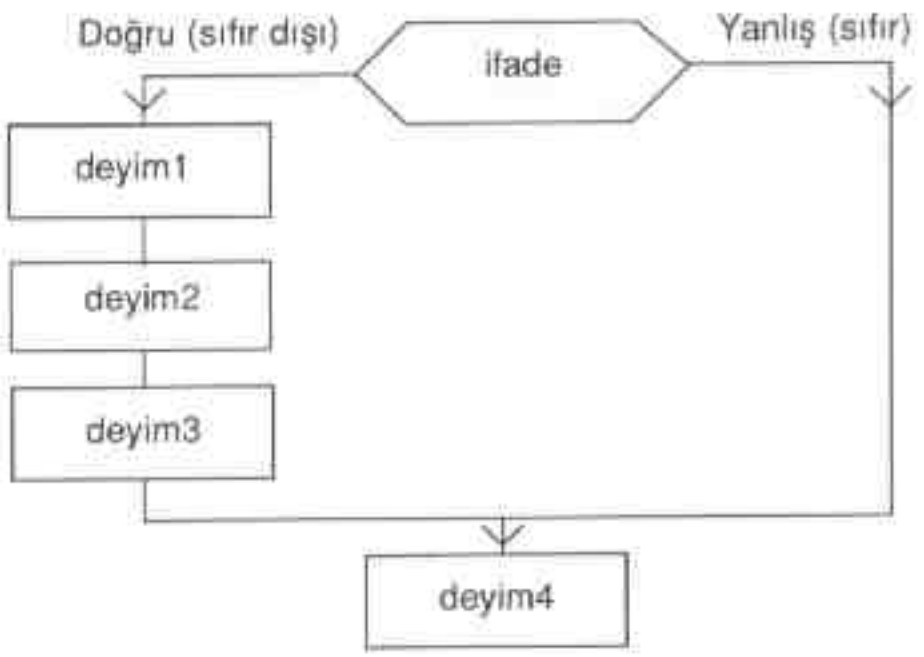
Bu örnekde `a < b - 4` ifadesi Doğruysa **deyim1** ve **deyim2**, Yanlışsa **deyim3** ve **deyim4** yapılır.

Bir `if` deyiminin **else** kısmı olmayabilir.

Örneğin:

```
if (ifade) {
        deyim1
        deyim2
        deyim3
}
deyim4
...
```

`if` parantezinin içerisindeki ifade doğruysa **deyim1, deyim2** ve **deyim3** yapılır; daha sonra programın akışı **deyim4** ile devam eder. Eğer ifade yanlış ise blok atlanarak doğrudan **deyim4** yapılır.



Bir `if` deyimi yalnızca else kısmına sahip olamaz. Bu durumda en uygun yol koşul ifadesini değiştirmektir.

```
if (ifade)
else
        deyim1                  /* Hata */
```

Yukarıdaki hatalı olan `if` deyiminin yerine aşağıdaki eşdeğeri kullanılabilir:

```
if (!ifade)
        deyim1
```

Aynı şey, `if` deyiminin Doğru kısmına boş deyim yerleştirilerek de yapılabilirdi.

```
if (ifade)
        ;
else
        deyim1
```

if parantezinin içerisindeki ifadede değişken olması gibi bir zorunluluk yoktur.

```
if (10)
        deyim
...
if (-1)
        deyim
...
if (0)
        deyim
....
```

Yukarıdaki ilk iki if deyimi her zaman doğru, ikincisi ise her zaman yanlıştır. Böyle if deyimlerinin kullanılmasının bir amacı olabilir mi?

```
if (a) {
        deyim1
        deyim2
        ...
}
```

Bu if deyimi ise a sayısının 0 olup olmadığına göre **Doğru** ya da **Yanlış** biçiminde değerlendirilecektir. Biraz daha karmaşık bir örnek verelim:

```
if ((ch = getchar()) == 'q')
        deyim1
else
        deyim2
```

Burada klavyeden girilen karakterin, 'q' karakteri olup olmadığı test ediliyor. İfade içindeki parantezlerin atama işlemine öncelik kazandırmak için kullanıldığına dikkat ediniz.

Fonksiyonların geri dönüş değerlerinin if deyimi içerisinde test amacıyla doğrudan kullanıldığına da sıklıkla rastlanır.

```
if (isalpha(ch))
        deyim1
else
        deyim2
...
```

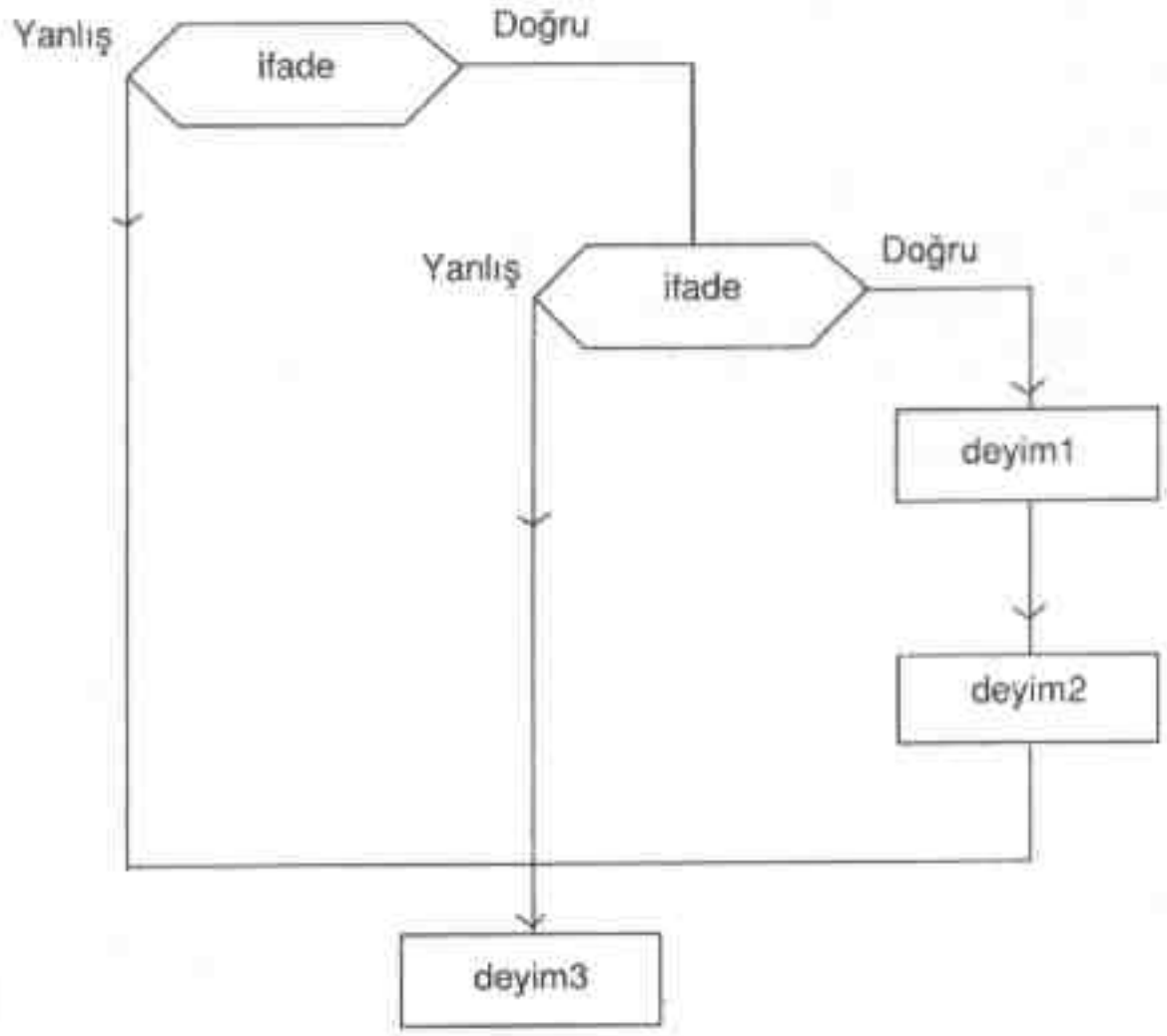
if deyiminin **Doğru** ya da **Yanlış** kısmı başka bir kontrol deyimi de olabilir.

```
if (ifade1)
        if (ifade2) {
            deyim1
            deyim2
        }
deyim3
```

Birinci `if` deyiminin Doğru kısmı 2. `if` deyim

Birinci `if` deyeminin dışındaki ilk deyim

Bu örnekte ikinci `if` deyimi birinci `if` deyimin **Doğru** kısmını oluşturuyor. Birinci ve ikinci `if` deyiminin **Yanlış** kısımları yoktur. Akış diyagramını aşağıda veriyoruz.



İç içe `if` deyimlerinde tereddüt edeceğiniz bir nokta olabilir.

```
if (ifade1)
        if (ifade2)
                deyim1
        else
                deyim2          /* hangi if deyiminin Yanlış kısmı? */
...
```

Deyim2'nin hangi `if` deyiminin **Yanlış** kısmı olduğu sizi çatışmaya düşürebilir. Evet, `else` ikinci `if` deyimine aittir. Eğer birinci `if` deyimine ait olmasını istiyorsanız, bloklama yapmalısınız!..

```
if (ifade1) {
    if (ifade2)
        deyim1
}
else
    deyim2
```

Bu örnekte **else** birinci **if** deyimine aittir; çünkü bloklama yapılmıştır. Aşağıdaki örneği inceleyelim.

```
if (ifade1) {
    if (ifade2)
        deyim1
    else {
        deyim2
        deyim3
    }
    deyim4
}
else
    deyim5
```

Birinci **if** deyiminin **Doğru** kısmı birden fazla deyimden oluştuğu için bloklama yapılmıştır. **Deyim5**, birinci **if** deyiminin **Yanlış** kısmını oluşturuyor. Birinci **if** deyiminin **Doğru** kısmının, ikinci **if** deyimi ile **deyim4**'den oluştuğuna dikkat ediniz.

else if merdivenlerine de çok rastlanır.

```
if (ifade1)
    deyim1
else
    if (ifade2)
        deyim2
    else
        if(ifade3)
            deyim3
        ...
```

10.3 BİRAZ DA UYGULAMA...

isalpha standart bir *C* fonksiyonudur (aslında bir makrodur!). Parametresi olan karakter, eğer alfabetik bir karakterse (yani büyük ya da küçük harf ise) **Doğru (sıfır dışı bir değere)**, alfabetik bir karakter değilse **Yanlış (sıfır değerine)** değerine geri döner. Bu fonksiyonu **myisalfa** ismiyle biz yazalım.

```
#include <stdio.h>
int myisalpha (char ch)
{
    if  (ch >= 'a' && ch <= 'z' || ch >='A' && ch <='Z')
        return ch;
    else
        return 0;
}

void main(void)
{
    char ch;
    ch = getchar();
    if (myisplaha(ch))
        printf("Alfabetik karakter!\n");
    else
        printf("Alfabetik karakter değil!\n");
}
```

Bu örnekte, sizi aşağıdaki if deyimi zorlayabilir.

```
if (ch >= 'a' && ch <= 'z' || ch >='A' && ch <='Z')
    return ch;
else
    return 0;
```

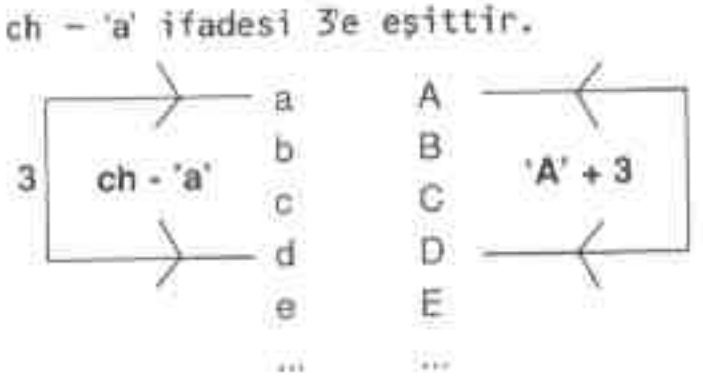
&& operatörünün || operatörüne karşı önceliği olduğunu anımsayınız. Bu durumda ch, küçük harf ya da büyük harf ise if deyimi **Doğru** olarak ele alınacaktır. Madem, C'de **Doğru** sıfır dışı bir değerdir, o halde doğrudan ch karakterinin sayısal değeriyle geri dönülebilir. Farketmişsinizdir; burada aslında else anahtar sözcüğüne hiç ihtiyaç yok; çünkü ifade **Doğru** ise zaten fonksiyon sonlandırılıyor.

toupper standart bir **C** fonksiyonudur (aslında bir makrodur). Parametresi olan karakter, eğer küçük harf ise, onun büyük harf karşılığıyla geri döner. toupper küçük harf olmayan karakterlere hiç dokunmaz; onları değiştirmeksizin geri dönüş değeri olarak iade eder. Bu fonksiyonu **mytouppper** ismiyle biz yazalım.

```
#include <stdio.h>

int mytoupper (int ch)
{
    if (ch >= 'a' && ch <='z')
        return ch - 'a' + 'A';
    return ch;
}
```

```
void main(void)
{
    char ch;

    ch = getchar();
    printf("%c\n", mytoupper(ch));
}
```

Küçük harfi büyük harfe nasıl çeviriyoruz? Büyük ve küçük harfler *ASCII* tablosunda peşpeşe bulunurlar, değil mi? Örneğin, **getchar** ile girdiğimiz karakter 'd' olsun.



`ch - 'a'` ifadesi **ch** karakterinin 'a' karakterinden uzaklığını gösteriyor. Bu uzaklığa 'A' karakterinin *ASCII* numarasını eklersek büyük harf **ch** karakterini buluruz. **Parametreleri ve geri dönüş değerleri karakter türünden olan fonksiyonların tamsayı olarak tanımlandıklarını görürseniz, şaşırmayın; bunun bir nedeni var!..** Örneğin, yukarıda **toupper** fonksiyonunu;

```
char mytoupper (char ch)
{
    ...
}
```

yerine:

```
int mytoupper(int ch)
{
    ...
}
```

olarak tanımladık. Dökümantasyonlarda da genellikle, fonksiyonların parametreleri ve geri dönüş değerleri **char** olsa bile, `int` biçiminde gösterilir. Bunun nedenini ancak sembolik makina dili programcılarının anlayabileceği bir biçimde açıklayabiliriz.

80X86 Sembolik Makina Dili Programcısına Not: C'de parametrelerin fonksiyonlara geçirilmesinde `stack` bölgesi kullanılmaktadır. Sistemlerin hemen hepsinde `stack` bölgesine mikroişlemcinin bir kelimesi kadar bilgiyi aktaran makina komutları vardır (PUSH, POP komutları gibi). C'de `int` veri türü de mikroişlemcinin bir kelimesi kadardır. Yani biz, bir parametreyi `char` türünden tanımlasak bile, derleyiciler onu `stack` bölgesine yollamak için `int` gibi ele alırlar. Bu durumda doğallığın korunması için parametrelerin `int` olarak tanımlanması daha anlamlı görünüyor. Aynı açıklama geri dönüş değerleri için de geçerlidir.

Küçük harfi büyük harfe çeviren fonksiyon olur da, büyük harfi küçük harfe çeviren fonksiyon olmaz mı? İşte, `tolower` fonksiyonu:

```
int tolower(int ch)
{
    if (ch >= 'A' && ch <= 'Z')
        return ch - 'A' + 'a';
    return ch;
}
```

Tıpkı `mytoupper` fonksiyonunda olduğu gibi önce büyük harf testi yapılıyor. Büyük harf ise aynı yöntem kullanılarak küçük harfe çevriliyor, değilse olduğu gibi bırakılıyor.

Yukarıda açıkladığımız `isalpha, toupper, tolower` fonksiyonlarının hepsi de yalnızca İngilizce harfleri dikkate almaktadır. Bunların Türkçe uyarlamalarını siz**ler** yazmalısınız. Türkçe uyarlamalarının isimlerini örneğin, **isalpha_trk, toupper_trk, tolower_trk** verebilirsiniz. Şu ana kadarki bilgilerimizle bu fonksiyonların yazılması mümkündür. Bunun için önce bütün Türkçe karakterleri kontrol etmelisiniz. Örneğin **toupper_trk** için kontrol hiç üşenmeden tek tek yapılmalıdır.

```
int toupper_trk (char ch)
{
    if (ch == 'ş')
       return 'Ş';
    if (ch == 'ı')
       return 'I';
    ....
    if (ch >='a' && ch <= 'z')
       return ch - 'a' + 'A';
}
```

10.4 KARAKTER TEST FONKSİYONLARI

Karakterler hakkında bilgi edinmek için kullandığımız fonksiyonlara karakter test fonksiyonları diyoruz. Bu fonksiyonlar, `ctype.h` başlık dosyasında **makro** biçiminde bulunurlar. Bu nedenle, karakter test fonksiyonları kullanılmadan önce mutlaka `ctype.h` dosyası kaynak koda dahil edilmelidir.

```
#include <ctype.h>
```

Karakter test fonksiyonlarının gerçekte birer makro olduklarını söyledik. Makrolar konusunda yeterli açıklamaları **önişlemci (preprocessor)** konusunun anlatıldığı 30. Bölümde bulacaksınız.

C'deki bütün standart karakter test fonksiyonlarının listesini aşağıda veriyoruz. Bu fonksiyonların hepsi parametre olarak bir karakter alır. Geri dönüş değerleri koşulun sağlanması durumuna göre, **Doğru (sıfır dışı herhangi bir değer)** ya da **Yanlış (sıfır)**'dır.

Not: Karakter test fonksiyonlarının orijinalleri parametrelerinin ilk 7 bitini dikkate alırlar. Yani ASCII tablosunun ikinci yarısındaki karakterleri bu fonksiyonlarla test etmemelisiniz. Bu nedenle, yukarıda yazdığımız `isalpha` fonksiyonu da -karakteri bir bütün olarak ele aldığı için- oriinali ile tamamen özdeş değildir. Bu fonksiyonları kendiniz tasarlayıp yazarak denemelisiniz!..

Fonksiyon	Geri Dönüş değeri
`isalpha`	Alfabetik bir karakterse Doğru, değilse Yanlış.
`isupper`	Büyük harf ise Doğru, değilse Yanlış.
`islower`	Küçük harf ise Doğru, değilse Yanlış.
`isdigit`	Sayısal bir karakterse ('0'-'9' arası) Doğru, değilse Yanlış.
`isxdigit`	Hex sayıları göstermekte kullanılan bir karakterse ('0'-'9' ya da 'a'-'f' ya da 'A'-'F') Doğru, değilse Yanlış.
`isalnum`	Alfabetik ya da nümerik bir karakterse ('0'-'9' ya da 'a'-'z' ya da 'A'-'Z') Doğru, değilse Yanlış.
`isspace`	Boşluk karakterlerinden biriyse (Space, Carriage Return, New Line, Vertical Tab, Form Feed) Doğru, değilse Yanlış.
`ispunct`	Noktalama karakterlerinden biriyse (Kontrol karakterleri, alfabümerik karakterler ve boşluk karakterlerinin dışındaki karakterler) Doğru, değilse Yanlış.
`isprint`	Ekranda görülebilen (printable) bir karakterse (space dahil) Doğru, değilse Yanlış.

`isgraph`	Ekranda görülebilen bir karakterse (space dahil değil) Doğru, değilse Yanlış.
`iscntrl`	Kontrol karakteri ya da silme karakteri ise (ilk 32 karakter veya 127 nolu karakter) Doğru, değilse Yanlış.
`isascii`	*ASCII* tablosunun standart kısmı olan ilk 128 karakterden birisiyse Doğru, değilse Yanlış.

SORAMADIKLARINIZ...

S1) Bir fonksiyonun geri dönüş değeri başka bir fonksiyonun parametresi olabilir mi?

C1) Olabilir; yukarıda `mytoupper` fonksiyonunun geri dönüş değeri karakter biçiminde ekrana yazdırılırken böyle bir yol izlenmiştir:

```
printf("%c\n", mytoupper(ch));
```

S2) Karakter test fonksiyonları kullanılırken, neden `ctype.h` başlık dosyası kaynak koda dahil edilmek zorundadır?

C2) Başlık dosyalarının niçin kaynak koda dahil edildikleri konusunda henüz hiçbir şey bilmiyoruz. Ancak karakter test fonksiyonlarının gerçekte birer **makro** olması bunu zorunlu hale getiriyor.

S3) Standart C fonksiyonları da birbirlerini çağırır mı?

C3) Bu sorunun yanıtı evet ya da hayır olabilir. Tasarıma bağlıdır. Genel olarak birbirlerini çağıracak biçimde tasarlandıklarını söyleyebiliriz.

11 FONKSİYON PROTOTİPLERİ

Fonksiyon prototiplerini kavramsal karmaşıklığı nedeniyle ayrı bir bölüm olarak ele almayı uygun gördük. Prototip bildirimlerinin yapılış biçimleri, nerede ve neden kullanıldıkları bölüm içerisinde açık bir biçimde ele alınmaktadır. Bu bölüm size ilk kez olarak başlık dosyalarının içeriği hakkında da küçük bir fikir verecektir.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Fonksiyon prototiplerine neden ihtiyaç duyulmaktadır?
2) Prototip bildirimlerinin genel biçimi nasıldır?
3) Prototip bildirimleri ile parametre kontrolü arasında nasıl bir ilişki vardır?
4) Prototip bildirimleri yapılmamış ve programcı tarafından tanımlanmamış olan fonksiyonlar, *C* derleyicileri tarafından nasıl ele alınırlar?
5) Prototip bildirimlerini yapmamanın doğuracağı sakıncalar nelerdir?

11.1 PROTOTİP KAVRAMI

Bir fonksiyonun çağrılması durumunda derleyiciler, çağrılan fonksiyonun geri dönüş değerinin türünü bilmek zorundadır. Derleyiciler, çağrılan fonksiyonlarla ilgili bilgileri çeşitli biçimlerde edinebilirler. Eğer, çağrılan fonksiyon çağıran fonksiyondan daha önce tanımlanmışsa; bu durumda derleyici, derleme akışı içerisinde çağrılan fonksiyonu daha önce göreceği için, geri dönüş değerinin türünü de doğal olarak öğrenecektir. **Çünkü derleme işleminin bir yönü vardır ve bu yön yukarıdan aşağıya doğrudur.**

```
#include <studio.h>
double div(double a, double b)
{
    return a / b;
}
void main(void)
{
    double a;
    a = div (10.2, 2.3)
    printf("a = %f\n", a);
}
```

Derleme işleminin yönü yukarıdan aşağıya doğrudur.

Derleyici bu sırada div fonksiyonunun geri dönüş değerinin double türünde olduğunu biliyor.

Bu örnekte **div** fonksiyonu kendisini çağıran **main** fonksiyonundan daha önce (daha yukarıda) tanımlanmıştır. **div** fonksiyonunun **main** içerisinde çağrıldığını gören derleyici, geri dönüş değerinin **double** türünden olduğunu belirleyebilecektir.

Eğer çağrılan fonksiyon çağıran fonksiyondan sonra tanımlanmışsa, bu durumda derleyici çağırılan fonksiyonun geri dönüş değerinin türünü bilemez. Burada problemli bir durum söz konusudur.

```
#include<stdio.h>

void main(void)
{
    double a;
    a = div (10.2, 2.3)
    printf("a = %f\n", a);
}

double div(double a, double b)
{
    return a / b;
}
```

Derleme işleminin yönü yukarıdan aşağıya doğrudur.

Derleyici bu sırada div fonksiyonunun geri dönüş değerinin double türünden olduğunu bilmez.

C derleyicileri çağrılma işleminin yapıldığı yere kadar, haklarında bilgi edinemedikleri fonksiyonların int geri dönüş değerine sahip olarak tanımlandıklarını varsayarlar.

Bu durumda yukarıdaki örnekte **div** fonksiyonunun:

```
int div()
```

biçiminde tanımlandığı varsayılır. Bundan sonra derleyici, derleme akışı içinde **div** fonksiyonunun geri dönüş değerinin **double** türünden olduğunu anladığı za-

man ise iş işten geçmiş olur. Derleyiciler, amaç kod üretilmesini engelleyecek derecede ciddi olan bu çatışma durumunu bir hata mesajı (error) ile programcıya bildirirler. Örneğin, bu tür durumlarda eğer **Borland** derleyicileri ile çalışıyorsanız:

```
Type mismatch in redeclaration of 'div'
```

ya da eğer **Microsoft** derleyicilerinde çalışıyorsanız:

```
'div' : redefinition
```

biçiminde bir hata mesajı ile karşılaşırsınız. Bu mesajların yorumunu size bırakarak bir noktaya dikkatinizi çekmek istiyoruz: *"Eğer* `div` *fonksiyonunun geri dönüş değeri* `int` *türünden olsaydı, bu durumda derleyicinin varsaydığı tür ile tanımlanan tür birbirinin aynısı olacağından herhangi bir problem de ortaya çıkmayacaktı."*

Anlattıklarımızdan bir özet çıkarmak istersek, şöyle diyebiliriz: Derleyiciler çağrılan fonksiyonların geri dönüş değerlerinin hangi türden olduklarını bilmek isterler. Bunu sağlamanın en doğal yolu, çağrılan fonksiyonu çağıran fonksiyondan önce tanımlamaktır. Çağrılma işleminin yapıldığı yere kadar, haklarında hiçbir bilgi edinilmemiş fonksiyonların geri dönüş değerleri ise *C* derleyicileri tarafından `int` olarak varsayılır.

Fakat çağrılan fonksiyonu her zaman çağıran fonksiyonun yukarısında tanımlamak mümkün olmayabilir. Dahası, *"standart C fonksiyonları"* bağlama aşamasında kütüphanelerden alınarak çalışabilen kod içerisine (**.EXE**) yerleştirildiklerine göre, ortada bu fonksiyonlar için bir tanımlama da yoktur. İşte **fonksiyon prototipleri** bu durumlarda derleyicileri bilgilendirmek amacıyla kullanılırlar.

Çağrılana kadar tanımlanmamış olan fonksiyonlar hakkında derleyicilerin bilgilendirilmesi işlemi fonsksiyon prototipleri ile yapılır.

Genel biçim:

```
[geri dönüş değerinin türü] <fonksiyon ismi> ([tür1], [tür2],...);
```

Örneğin `div` fonksiyonu için yazacağımız prototip şöyledir:

```
double div (double, double);
```

Derleyici böyle bir prototip bildiriminden, `div` fonksiyonun geri dönüş değerinin ve iki parametresinin `double` türünden olduğu bilgisini edinecektir. Birkaç örnek daha verelim:

```
int topla (int, int);
int getch (int);
```

```
void clrscr(void);
...
```

Fonksiyon prototip bildiriminin, yalnızca derleyiciyi bilgilendirmek amacıyla kullanıldığına dikkat etmelisiniz. Fonksiyon prototip bildirimleri sonucunda derleyici bellekte bir yer ayırmaz; dolayısıyla bir tanımlama işlemi değildir.

Prototip bildirimlerinde paramete türlerinden sonra parametre isimleri de yazılabilir.

```
double div(double a, double b);
int topla(int a, int b);
double pow(double base, double exp);
...
```

Prototiplerdeki parametre isimlerinin faaliyet alanları parametre parantezi ile sınırlıdır. Prototiplerdeki parametre isimlerinin aynısının tanımlamada kullanılması biçiminde bir zorunluluk yoktur.

11.2 FONKSİYON PROTOTİPLERİNİN BİLDİRİM YERLERİ

Fonksiyon prototiplerinin bildirimi programın herhangi bir yerinde yapılabilir. Nerede bildirilmiş olurlarsa olsunlar, bildirildikleri yerlerden dosyanın sonuna kadar olan bölgede geçerlidirler. Burada önemli olan nokta, fonksiyonu çağırmadan önce prototip bildiriminin yapılmış olmasıdır. **Fonksiyon prototiplerinin, programın en yukarısında ya da programcının tanımladığı başlık dosyalarının birinin içinde bildirilmesi tavsiye edilir.**

Örneğin:

```
#include <stdio.h>

float div(double, double);

void main(void)
{
    double a;

    a = div(10.2, 2.3);
    printf("a = %f\n", a);
}

float div(double a, double b)
{
    return a / b;
}
```

11.3 STANDART C FONKSİYONLARININ PROTOTİPLERİ

Standart C fonksiyonları bizim tarafımızdan yazılmadığına göre, derleyici bu fonksiyonların geri dönüş değerlerinin türlerini akış yönü içerisinde belirleme şansına sahip olamaz. Bu durumda standart C fonksiyonlarının geri dönüş değerlerinin türleri fonksiyon protipleri ile belirtilmek zorundadır. Pototiplerin programcı tarafından doğru bir biçimde yazılması oldukça zahmetli olacağı için bunlar standart başlık dosyalarının içerisine yerleştirilmiştir. Yani programcı kullanacağı standart C fonksiyonunun prototipini yazmak yerine onun bulunduğu standart başlık dosyasını kaynak koda dahil eder. C'ye yeni başlayanların çoğu, fonksiyonların başlık dosyalarının (`*.h`) içerisinde bulunduğunu sanır. Oysa fonksiyonlar kütüphanelerde, prototipleri başlık dosyalarının içerisindedir. Örneğin, bütün **"giriş/çıkış"** fonksiyonlarının prototipleri `stdio.h` dosyasının içerisindedir. Bu dosyanın kaynak koda dahil edilmesiyle bütün *giriş/çıkış* fonksiyonları için prototip bildirimi yapmış oluruz.

Standart C fonksiyonlarının prototipinin belirtilmediği bazı durumlarda hata ortaya çıkmayabilir.

C derleyicilerinin bu durumlarda çağrılan fonksiyonların geri dönüş değerlerinin türlerini int olarak varsaydığını anımsayınız. Hata ancak, geri dönüş değeri `int` türünden farklı olan ve geri dönüş değerinin kullanıldığı durumlarda ve çalışma zamanı sırasında ortaya çıkar.

Not: C++'da bir fonksiyon int geri dönüş değerine sahip olsa bile prototipinin yazılması zorunludur.

11.4 FONKSİYON PROTOTİPLERİ VE PARAMETRE KONTROLÜ

Fonksiyon prototipleri derleyiciye parametre kontrolü yaptırmak amacıyla da kullanılabilir. Eğer bir prototip bildiriminde parametre türleri belirtilmişse, bu durumda derleyici çağrılan fonksiyonun parametrelerini prototipte bildirilenler ile **sayı ve tür** bakımından karşılaştıracaktır. Örneğin:

```
double div(double, double);
```

biçiminde bir fonksiyon prototipi yazılmışsa, derleyici parametre kontrolü yapar. Bu durumda `div` fonksiyonunun eksik ya da fazla parametreyle çağrılması derleme hatası oluşmasına neden olur.

```
a = div(3.2);                  /* Derleme hatası! 1 parametreyle çağırılmış*/
...
a = div(10.2, 4.0, 5.3);       /* Derleme hatası! 3 parametreyle çağırılmış*/
```

Parametre kontrolü yapılması istenmiyorsa, prototip içerisinde parametre türleri

hiç yazılmamalıdır. Örneğin:

```
double div();
```

gibi bir prototip bildiriminde derleyici parametre kontrolü yapmaz. Bu prototip bildiriminin, **div** fonksiyonun parametresiz olduğunu belirtmek için değil, parametre kontrolünü engellemek için yazıldığını vurgulayalım. Bu durumda **div** fonksiyonu kaç parametreyle çağrılmış olursa olsun bir derleme hatası oluşmayacaktır.

```
double div();
...
a= div(10.2, 5.6, 6.8);        /* Derleme hatası oluşmaz */
...
a = div(2.4);                  /* Derleme hatası oluşmaz */
...
```

Prototip bildiriminde, bir fonksiyonun parametresiz olduğunu belirtmek için parametre yerine **void** anahtar sözcüğü yazılmalıdır.

```
double fonk(void);
```

Yukarıdaki prototip ifadesi, **fonk** isimli fonksiyonun parametresiz olduğunu belirtmektedir. Bu durumda;

```
a = fonk(10);                  /* Derleme hatası! */
```

yukarıdaki ifade derleme hatasına neden olur.

11.5 FONKSİYON PROTOTİPLERİ VE TÜR DÖNÜŞÜMLERİ

Prototip bildirimi yoluyla yapılan parametre kontrolü otomatik tür dönüşümlerine de neden olmaktadır. Örneğin:

```
int sample(long int);
```

biçiminde yapılmış bir prototip bildiriminde derleyici **sample** isimli fonksiyonun parametresinin **long int** olduğunu tespit ettikten sonra, bu fonksiyonun çağrılması sırasında parametre uyuşumunu sağlamak için otomatik dönüşüm yapacaktır. Örneğin:

```
a = sample(10);
```

gibi bir çağırmada, **10** bir **int** sabitidir. Derleyici, 10 **int** sabitini, long **int** türüne dönüştürdükten sonra fonksiyona yollar.

Bu kadar bilgiyi şimdilik yeterli görüyoruz. Tür dönüşümleri sonraki bölümde ayrıntılı bir biçimde ele alınmaktadır.

SORAMADIKLARINIZ...

S1) Bir fonksiyonun eksik ya da fazla parametreyle çağrılması sakınca doğurur mu? Örneğin, iki parametreli `div` gibi bir fonksiyonu bir ya da üç parametreyle çağırırsak ne olur?

C1) Öncelikle şunu ifade etmeliyiz: Fonksiyonun prototipi parametre kontrolünü mümkün kılacak biçimde yazılmışsa ya da fonksiyon çağrılmadan önce tanımlanmışsa bu durumda zaten derleyici parametre kontrolü yapacağı için eksik ya da fazla parametreyle çağırmak mümkün olmaz. Ancak prototip içinde parametre kontrolü yapılması istenmemişse ya da çağrılan fonksiyonun prototipi hiç yoksa böyle bir durum mümkün olabilir.

Eksik parametreyle çağırmak, kestirilemeyen sonuçların doğmasına neden olabilir. Çünkü, yazılmayan diğer parametrelerin alacağı değerler tıpkı yerel değişkenlerde olduğu gibi, o onda parametre için tahsis edilen bellek bölgesinde bulunan rastgele değerlerdir.

```
...
a = div(10.2);
...

double div(double a, double b)
{
   return a / b;
}
...
```

Fazla parametre ile çağırma **80X86** mimarisinde çalışan derleyicilerde herhangi bir probleme neden olmaz. Fakat şüphesiz, bu durumdan da kaçınmak gerekir!..

80X86 Sembolik Makina Dili Programcısına Not: Fazla parametre ile çağırma, **80X86** sistemlerinde herhangi bir probleme yol açmaz. *C* derleyicileri gönderilen parametre sayısına göre fonksiyon çıkışında yığını düzenlerler. Örneğin `div` fonksiyonu iki parametreye sahip olduğu halde biz onu dört parametre ile çağırmış olalım:

```
...
x = div(a, b, c, d);
...
```

C derleyicileri yığına parametreleri sağdan sola gönderdikten sonra fonksiyonu çağırırlar. Fonksiyon çıkışında derleyici ne kadar parametre gönderdiyse, `SP` yazmacını da o kadar ilerletir.

```
...
push d
push c
push b
push a
call _div
add sp, 8
...
```

div fonksiyonu içeride yine ilk iki parametreyi kullanacaktır (a ve b; [bp+4], [bp+6] near modeller için, [bp+6], [bp+8], far modeller için)

S2) Derleyiciler niçin çağrılan fonksiyonların geri dönüş değerlerinin hangi türden olduklarını bilmek zorundadırlar?

C2) Bu sorunun yanıtını ancak sembolik makina dili programcılarının anlayabileceği bir biçimde verebiliriz.

80X86 Sembolik Makina Dili Programcısına Not: *C* Derleyicileri, geri dönüş değerlerini *CPU* yazmaçlarının içerisinden alırlar. Geri dönüş değerlerinin türü bir anlamda geri dönüş değerlerinin hangi yazmaçlar içerisinde olduğunu anlatmaktadır. **80X86** ailesini kullanan 16 bit işletim sistemlerinde iki byte geri dönüş değerleri AX yazmacından, dört byte geri dönüş değerleri ise DX:AX yazmaçlarından alınır.

S3) Fonksiyon prototiplerinde parametre isimleri yazmanın faydası ne olabilir?

C3) Prototip bildirimlerinde parametre isimleri yazmak okunabilirliği artırır; bunun dışında başkaca bir faydası yoktur. Örneğin, üs alma işlemi yapan **pow** fonksiyonunun hangi prototip bildirimi daha okunabilirdir?

```
double pow (double, double);
...
double pow (double taban, double us);
```

Birinci prototip ifadesinde, fonksiyon parametrelerinin ne amaçla kullanıldığı tam olarak anlaşılmıyor. Oysa ikinci örneğe bakan bir kişi bu fonksiyonun **taban**$^{\mathbf{us}}$ işlemini yaptığını kolaylıkla anlayabilir.

S4) Hangi standart *C* fonksiyonunu kullanacaksak onun bulunduğu başlık dosyasını kaynak koda dahil etmemiz gerekiyor. Peki bunun bir sırası var mıdır? Örneğin **pow** fonksiyonunun prototipi **math.h**, **printf** fonksiyonun prototipi **stdio.h** dosyasının içinde olduğuna göre:

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

ile

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

sıralamaları arasında bir ayrım var mıdır?

C4) Standart başlık dosyaları için zorunlu bir sıralama biçimi yoktur. Programcılar kendilerine göre bir sıra izleyebilirler. Ancak geleneksel olarak **stdio.h** dosyası programcılar tarafından birinci dosya olarak kaynak koda dahil edilirler.

S5) Bir fonksiyonun birden fazla yerde prototip bildirimi yapılabilir mi?

C5) Her defasında aynı olmak koşuluyla bir fonksiyonun kaynak kod içerisinde birden fazla yerde prototip bildirimi yapılabilir; ancak böyle bir duruma ihtiyaç duyulmaz.

12 TÜR DÖNÜŞÜMLERİ

Bu bölüm tümüyle tür dönüşümlerine ayrılmıştır. C'de farklı türlerdeki nesneler ve sabitler "tür uyuşumunu sağlamak amacıyla" otomatik dönüşümlere tabi tutulduktan sonra birbirleriyle işleme sokulurlar. Her programcı derleyiciler tarafından yapılan otomatik tür dönüşümünün kurallarını, "işlemlerin sonuçlarını kestirilebilmek için" bilmek zorundadır. Tür dönüşümleri yalnız otomatik olarak değil, programcının isteği doğrultusunda **bilinçli olarak** da yapılabilir. Fakat nasıl yapılırsa yapılsın tür dönüşümlerinin açık ve taşınabilir kuralları vardır. Biz de bölüm içerisinde öncelikle bu kuralları açıklamayı uygun gördük. Daha sonra farklı türlerin birbirlerine atanması, işlem öncesi otomatik tür dönüşümleri ve nihayet bilinçli tür dönüşümleri konularını inceledik.

Bölüm içerisinde verdiğimiz örnekler 16 bit *DOS* ya da *Windows 3.1* sistemleri gözönüne alınarak düzenlenmiştir. Farklı sistemlerde çalışan okuyucularımız daha çok örneklerden çıkan sonuçlara dikkat etmelidirler.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Tür dönüşüm kuralları nasıldır?

2) Hangi dönüştürme işlemlerinde "bilgi kaybı" söz konusudur?

3) Farklı türlerin birbirlerine atanması durumunda tür dönüşümleri nasıl yapılır?

4) İşlem öncesi otomatik tür dönüşümünün kuralları nelerdir?

5) Tür dönüştürme operatörü nasıl kullanılır ve diğer operatörlere göre öncelik sırası nasıldır?

12.1 DÖNÜŞÜM KURALLARI

Tür dönüşümlerine ilişkin karşılaşabilecek tüm durumları dört gruba ayırabiliriz.

1) Küçük tamsayı türünün büyük tamsayı türüne dönüştürülmesi

2) Büyük tamsayı türünün küçük tamsayı türüne dönüştürülmesi

3) Tamsayı türleri ile *(char, short int, int, long int)* gerçek sayı türleri *(float, double, long double)* arasındaki dönüşümler

4) Gösterici türleri ile ilgili dönüşümler

Bunlardan ilk üçü bölüm içerisinde ele alınmaktadır. *"Gösterici türleri ile ilgili dönüşümler"* ise göstericilerin anlatıldığı bölüme bırakılmıştır. Tür dönüşümleri diğer dillerde olduğu gibi C'de de *"bir işlemlik"*, yani geçici olarak yapılır; zaten tanımlama ile belirlenen nesnelerin türü hiçbir biçimde kalıcı olarak değiştirilemez.

12.1.1 Küçük Tamsayı Türünün Büyük Tamsayı Türüne Dönüştürülmesi

char ► int, int ► long, char ► long ,... gibi küçük türlerin büyük türlere dönüştürülmesi durumunda bilgi kaybı söz konusu değildir. Kısa türün **işareti korunarak** dönüştürme yapılır. Aşağıda 16 bit sistemler göz önünde bulundurularak çeşitli örnekler verilmiştir.

Tür	DESİMAL	HEX	Dönüştürülecek Tür	HEX	DESİMAL
int	256	0x0100	long int	0x00000100L	256L
char	'a' = 97	0x61	int	0x0061	97
int	-10	0xFFF6	long int	0xFFFFFFF6L	-10L
char	'\x85' = -123	0x85	int	0xFF85	-123
unsigned int	50000	0xC350	long int	0x0000C350L	50000L
char	'A' = 65	0x41	long int	0x00000041L	65L
...	...	...	...	...	...

Küçük türün büyük türe dönüştürülmesinde en çok tereddüt edilen nokta, negatif bir sayının dönüştürme işlemine tabi tutulması durumunda ortaya çıkmaktadır. Üçüncü örnekteki `0xFFF6 (-10)` sayısının, **long int** türüne `0xFFFFFFF6` biçiminde dönüştürüldüğünü görüyorsunuz. Belki siz de sayının `0x0000FFF6` olması gerektiğini düşüneceksiniz. Oysa sayı negatiftir ve yüksek anlamlı iki byte negatifliği korumak için `FFFF` değerini almak zorundadır. Genel olarak şunları söyleyebiliriz: Küçük türün büyük türe atanması durumunda sayı pozitifse yüksek anlamlı byte değerleri **00** ile, sayı negatif ise **FF** ile doldurulur.

12.1.2 Büyük Tamsayı Türünün Küçük Tamsayı Türüne Dönüştürülmesi

Büyük türün küçük türe dönüştürülmesi durumunda bilgi kaybı söz konusu olabilir. Kural oldukça yalındır: **Yüksek anlamlı byte değerleri kaybedilecek biçimde dönüştürme yapılır.** Aşağıdaki örnekleri inceleyiniz.

Tür	DESİMAL	HEX	Dönüştürülecek Tür	HEX	DESİMAL
int	500	0x01F4	char	0xF4	-12
long int	1000000L	0x000F4240L	int	0x4240	16960
long int	1000000L	0x000F4240L	char	0x40	64
unsigned int	50000U	0xC350U	char	0x50	80
int	-250	0xFF06	char	0x06	6
long int	-1000L	0xFFFFFC18L	int	0xFC18	-1000
...	...	...	...	...	...

Örneklerden de gördüğünüz gibi, dönüştürme işlemi sonucunda sayıların yüksek anlamlı byte değerleri kayboluyor. Buradaki en önemli nokta şudur: Büyük türdeki bir sayı, yüksek anlamlı byte değerlerini kaybedince ilkiyle alakasız, tamamen farklı bir sayı haline gelebiliyor. Fakat büyük türdeki sayı, dönüştürülecek olan küçük türün sınırları içinde kalıyorsa bu durumda bir bilgi kaybı oluşmuyor.

12.1.3 Tamsayı Türleri ile Gerçek Sayı Türleri Arasındaki Dönüşümler

Tamsayı türleri ile (`char`, `short int`, `int`, `long int`) gerçek sayı türleri (`float`, `double`, `lonf double`) arasındaki dönüşümleri iki kısma ayırarak inceleyebiliriz.

a) **Gerçek sayı türlerinden tamsayı türlerine yapılan dönüştürmeler.** Bu durumda dönüştürme sonucunda gerçek sayının noktadan sonraki kısmı kaybolur. Yani gerçek sayı, gerçek sayı formatından çıkarılarak tamsayı formatıyla ifade edilir. Örneğin:

Tür	DESİMAL	Dönüştürülecek Tür	DESİMAL
float	102.5f	int	102
float	13.902f	int	13
float	12.58f	long int	12L
double	1328.35	int	1328
float	183.68f (183 = 0xB7)	char	-73
double	5238.2781	long int	5328L
...	...	...	...

`183.68f` float sayısının `char` türüne dönüştürüldüğü beşinci örneğimiz için küçük bir açıklama yapmayı uygun görüyoruz: `183.68f` sayısının kesir kısmı atıldığında elde edilen **183** sayısı `int` türündendir. Bu türün tekrar `char` türüne dönüştürülmesi sonucunda **-73** sayısı bulunmuştur.

b) Tamsayı türlerinden gerçek sayı türlerine yapılan dönüştürmeler. Bu durumda sayı noktalı bir biçim kazanır. (Yani gerçek sayı formatına dönüştürülür.) **Dönüştürme sonucunda kesir kısmı 0 olan gerçek sayı elde edilir.** Örneğin:

Tür	DESİMAL	Dönüştürülecek Tür	DESİMAL
int	100	float	100.0f
char	'\x52'	float	82.0f
long int	4530L	double	4530.0
unsigned int	52500	float	52500.0f
...	...	...	...

12.1.4 Gösterici Türleri ile İlgili Dönüşümler

Bu çeşit dönüşümler göstericilerin anlatıldığı 18. Bölümde ele alınmaktadır.

12.2 FARKLI TÜRLERİN BİRBİRLERİNE ATANMASI

Bu bölüme gelene kadar verdiğimiz tüm örneklerde aynı türlerin birbirlerine atanmasına büyük özen gösterdik. Gösterdiğimiz bu özen "farklı türlerin birbirlerine atanmasının yasak olmasından" kaynaklanmıyor. Çünkü C'de atama işlemlerinde sağ taraf ve sol taraf değerlerinin aynı türden olması gibi bir zorunluluk yoktur. Ancak programcının aynı türden olmayan atama işlemlerinin sonuçlarını önceden kestirebilmesi gerekir. Farklı türlerin birbirlerine atanmasında yalın bir kural vardır.

Sağ taraf değeri, sol taraf değerinin türüne dönüştürüldükten sonra atama yapılır.

Pascal, Dbase, Clipper Programcılarına Not: Bu dillerde farklı türlerin birbirlerine atanması yasaklanmıştır. Örneğin *Pascal*'da karakter türünden bir değişken doğrudan tamsayı türüne atanamaz.

```
a: char;
b: integer;
...
a := 'x';
b := a;                    /* Geçersiz */
```

Farklı türlerin birbirlerine atanması durumlarını yine tür dönüşümlerinde olduğu gibi ayrı ayrı gruplar halinde incelemeyi uygun görüyoruz.

12.2.1 Küçük Tamsayı Türünün Büyük Tamsayı Türüne Atanması

Bu durumda küçük tür, büyük türe dönüştürüldükten sonra atama yapılır. Anımsayacağınız gibi, bu dönüştürme işleminde sayının işareti korunur ve herhangi bir bilgi kaybı oluşmaz.

Örnekler:

```
int a;
char ch;
...
ch = '\x65';
a = ch;                          /* a = 0x0065 */


long x;
int y;
...
y = -10;                         /* y = 0xFFF6 */
x = y;                           /* x = 0xFFFFFFF6L */
```

y sayısının negatifliği korunacak biçimde dönüşüm yapılıyor. 4 byte içerisinde -10 sayısını nasıl yazarsınız?

12.2.2 Büyük Tamsayı Türünün Küçük Tamsayı Türüne Atanması

Büyük tür önce küçük türe dönüştürülür; sonra atama yapılır. Bu tür dönüştürme işlemlerinde yüksek anlamlı byte değerlerinin kaybedileceğini anımsayınız.

Örnekler:

```
int a;
long b;
...
b = 0x17688000L;                 /* b pozitif bir sayı */
a = b;                           /* a = 0x80000, negatif bir sayı */
```

Bu örnekte yüksek anlamlı byte değerleri kaybedildikten sonra 0x8000 sayısı elde edilmiştir. b, pozitif bir sayı olduğu halde atama sonucunda negatif bir sayının oluştuğunu görüyorsunuz.

Atama işlemlerinin sonucunda yüksek anlamlı byte değerlerinin kaybedilmesiyle elde edilen sayı, deneyimsiz programcılar tarafından kimi zaman hayretle karşılanmaktadır. Örneklerimizde kaybedilen bilginin açık bir biçimde anlaşılabilmesi için *HEX* sistemi kullanıyoruz. Desimal sistemdeki sayıların yüksek ve alçak anlamlı byte değerlerini ayırmak oldukça zordur.

12.2.3 Tamsayı Türleri ile Gerçek Sayı Türleri Arasındaki Atama İşlemleri

Gerçek sayı türlerinin (float, double, long double) tamsayı türlerine atanması durumunda, sayının yalnızca tam kısmı atanır. Gerçek sayı türlerinden tamsayı türlerine yapılan dönüşümün kurallarını anımsayınız.

```
int a;
...
a = 134.78f;                    /* a = 134 */
...
```

Bu durumda, `float` sabitinin tam kısmı olan **134**, **a** değişkenine atanır. Sayının tam kısmı, tamsayı türünün sınırları dışında kalıyorsa tekrar dönüşüm uygulanır.

```
char a;
...
a = 278.903;                    /* a = 22 */
...
```

Yukarıdaki örnekte **278.903** sayısının tam kısmı olan **278** bir `tamsayı (int)` sabitidir. Bu kez tamsayı türünden karakter türüne bir dönüşüm yapılacaktır.

```
278 = 0x0116
0x16 = 22
a = 22
```

Tamsayı türlerinden gerçek sayı türlerine yapılan atamalarda ise tamsayı, kesir kısmı sıfır olan gerçek sayı biçimine dönüştürülür. Örneğin:

```
float a;
...
a = 1200;               /* a = 1200.0 */
```

80X86 Sembolik Makina Dili Programcısına Not: Gerçek sayıların gerçek sayı formatıyla bellekte tutulduğunu anımsayınız. Bu nedenle gerçek sayılarla tamsayı türleri arasındaki atamalarda format dönüşümü söz konusudur. Örneğin **10** tamsayısı ile **10.0** gerçek sayısı aynı niceliği gösterse de bu iki sayının bellekte tutuluş biçimleri farklıdır. Gerçek sayı formatı için "Ek-A" bölümüne başvurabilirsiniz.

Farklı türlerin birbirlerine atanmasına ilişkin aşağıda verdiğimiz örnek programı çalıştırıp, sonucunu yorumlamalısınız.

```
#include <stdio.h>

void main(void)
{
    char a;
    int b;
    long c;
    double d;
```

```
    c = 0x12345678L;
    b = c;                              /* b = 0x5678 */
    printf("b = %x\n", b);
    a = b;                              /* a = 0x78 */
    printf("a = %x\n", a);
    d = -1.3928;
    b = d;                              /* b = 0xFFFF */
    printf("b = %x\n", b);
    a = '\x82';
    c = a;                              /* c = 0xFFFFFF82 */
    printf("c = %lx\n", c);
}
```

Tür dönüşüm kuralları bütün sistemler için aynıdır. Ancak bu kuralların tür uzunluklarıyla ilişkili olması, verdiğimiz örneklerin büyük kısmını yalnızca *DOS* ve *16 bit Windows* için anlamlı hale getiriyor. Örneğin, tamsayı ve uzun tamsayı türlerinin aynı olduğu sistemlerde bu iki tür arasındaki dönüşümlerde bilgi kaybı söz konusu olmaz. Bu nedenle *Windows 95, Windows NT* ya da *UNIX* sistemleri için de birkaç örnek vermek istiyoruz:

```
int a;                  /* 4 byte */
long b;                 /* 4 byte */
...
b = 0x12345678L;
a = b;                  /* a = 0x12345678L */
```

32 bit sistemlerde tamsayı ile uzun tamsayı türlerinin her ikisi de 4 byte uzunluğundadır.Bu nedenle bu iki tür arasında yapılan dönüşümlerde bilgi kaybı söz konusu olmaz. Fakat `char`, `short int` ile `int` veya `long int` türleri arasındaki dönüşümler açıkladığımız kurallar eşliğinde yapılmaktadır. Örneğin:

```
int a = 0x12345678;     /* 4 byte */
short b;                /* 2 byte */
...
b = a;                  /* b = 0x5678 */
```

Kısa tamsayı **2 byte**, tamsayı **4 byte** uzunluğunda olduğu için, atama sonucunda a değişkenin yüksek anlamlı **2 byte** değeri kaybolur.

12.3 İŞLEM ÖNCESİ OTOMATİK TÜR DÖNÜŞÜMLERİ

Derleyiciler tarafından programcının isteği dışında kendiliğinden yapılan tür dönüşümlerine otomatik tür dönüşümleri diyoruz. Otomatik dönüşümler farklı türlerin birbirlerine atanması durumunda olduğu gibi, hiçbir uyarı (warning) ya da hataya (error) neden olmaksızın derleyici tarafından doğal olarak yapılırlar. Oto-

matik dönüşümlerin en sık rastlanılan biçimi işlem öncesi tür dönüşümleridir.

C derleyicileri operandları aynı türe dönüştürdükten sonra işleme sokarlar. Operandlardaki tür uyuşumunu sağlamaya yönelik işlem öncesi otomatik dönüştürme kuralı aşağıdaki gibi özetlenebilir:

Küçük operand büyük operandın türüne dönüştürüldükten sonra işleme sokulur. İşlem sonucunda üretilen değer operandlarla aynı türdendir.

Bu kurala göre, örneğin tamsayı ile uzun tamsayı türleri birbirleriyle toplanacak olsa, önce tamsayı uzun tamsayı türüne dönüştürülür, daha sonra toplama işlemi yapılır.

```
a = 1000 + 50L;
```

1. Adım: Toplama işleminin birinci operandı `int`, ikinci operandı `long int` türündendir. Bu durumda, küçük tür olan `int`, büyük tür olan `long int` türüne dönüştürülür.

2. Adım: `1000L + 50L` işlemi yapılır. Sonuç `1050L` sayısıdır.

Şimdi aşağıdaki örneği inceleyelim.

```
int a, b;
...
a = 1000;
...
b = a * 50;
/* b = ? */
```

Bu örnekte çarpma işleminin her iki operandı da `int` türündendir. Bu nedenle herhangi bir dönüşüm söz konusu değildir; sonuç da `int` türünden olacaktır. 50000 *DOS* ve *Windows 3.1* sistemlerinde işaretli tamsayı sınırlarını aştığı için `long int` olarak ele alınır. Ancak sonucun int türünden olması gerektiği için bu `long int` değer derleyici tarafından `int` türüne dönüştürülür. Böylece, `b` değişkenine negatif bir sayı atanır.

İ1: a * 50 ► 50000L ►0000C350L, `int` türüne dönüştürülerek `C350` elde edilir; bu negatif bir sayıdır.

İ2: b = İ1 ► C350 ► -15536

Daha karmaşık bir örnek verelim:

```
char a;
int b;
long c;
int x;
...
a = '\x20';
b = 0x1BC0;
```

```
c = 0x1101CF20L;
...
x = b * a + c / 16;
/* x = ? */
```

İşlemler öncelik sırasına göre tek tek açıklanıyor (dönüşümün yapılacağı türler parantez içinde belirtilmiştir).

`i1: b * a ► int * char (int) ► 0x1BC0 * 0x0020 = 0x37800 ► 0x7800 (int)`

a, int türüne dönüştürüldükten sonra çarpma işlemi yapılır. `0x1BC0 * 0x0020 = 0x37800`, sonuç `int` türünden olacağından yüksek anlamlı byte değerleri kaybedildikten sonra `0x7800` sayısı elde edilir.

`i2: c / 16 ► long / int (long) ► 0x1101CF20L / 16L = 0x01102472L (long)`

`16` `long` türüne dönüştürüldükten sonra bölme işlemi yapılır. Sonuç `long` türündendir.

`i3: i1 + i2 ► int (long) + long ► 0x00007800L + 01102472L = 0x01109C72L (long)`

`i1` `long` türüne dönüştürüldükten sonra işleme sokulur. Sonuç `long` türündendir.

`i4: x = i3 ► x = 0x01109C72L`

Yukarıdaki ifadenin ilk adımında, `b * a` işlemi ile bir bilgi kaybı oluşmuştur. Bilgi kaybı oluşmasını istemiyorsanız, işlemlerin öncelik sırasına dikkat etmelisiniz. Örneğin:

```
x = 100 * 5000 / 2L;
```

işleminin sonucu `long` türünden olduğu halde yine de bir bilgi kaybı söz konusudur. Fakat önceliği yüksek olan operatörün operandlarından birini `long` olarak seçersek:

```
x = 100L * 5000 / 2;
```

bu durumdan kurtulmuş oluruz. İfade daha açık bir biçimde de yazılabilirdi:

```
x = 100L * 5000L / 2L
```

Bir operand **double** ise bu durum diğer operandın da **double** türüne dönüştürüleceği anlamına gelir. Örneğin:

```
float a;
double b, c;
...
a = 10.2f;
b = 20.8;
...
c = a * b;    float (double) * double    212.16 (double)
```

C'de tamsayılı bölme yapmak için iki operandın da tamsayı türlerinden (`char, short, int, long`) olması yeterlidir. Tamsayılı bölme için ayrı bir operatörün olmadığına dikkat etmelisiniz!..

```
x = 10 / 4 ► int / int ► 2 (int)
x = 100000L / 3► long / int (long) ► 33333L (long)
```

Pascal Programcısına Not: Pascal'da tamsayı bölme için `div` isimli ayrı bir operatör vardır. Örneğin:

```
a := 10 div 3;
/* a = 1 */
```

Benzer biçimde operandlardan biri `float` diğeri `long` ise her ikisi `float` türüne dönüştürülerek işlem yapılır.

```
x = 1300L * 1.2f; ►  long (float) * float ►  1560.0f (float)
```

küçük türün büyük türe dönüştürülmesi biçiminde özetlediğimiz otomatik tür dönüşümünün birkaç istisnası vardır. Örneğin aynı tamsayı türünün hem **işaretli** (`signed`) hem de **işaretsiz** (`unsigned`) biçimleri birlikte işleme sokulmuşsa, bu durumda otomatik dönüştürme işlemi işaretsiz türe doğru yapılır. Örneğin:

```
unsigned x, y;
...
x = 42500U;
...
y = x + 1000; unsigned + int (unsigned)   42500U + 1000U = 43500U
```

`int` sabiti olan 1000, `unsigned` türüne dönüştürüldükten sonra toplama işlemi yapılmıştır.

Her iki operand da `short` ve/veya `char` türündeyse her ikisi de işlem öncesi otomatik olarak `int` türüne dönüştürülür (tıpkı `float` türünün `double` türüne dönüştürüldüğü gibi). İki `short` türün `int` türüne dönüştürülmesi kuralının *DOS* için bir önemi olmasa da `short` ve `int` türlerinin birbirlerinden farklı olduğu **32 bit** sistemlerde *(UNIX, Windows 95 ya da Windows NT)* önemli sonuçları olabilir. **32 bit** *UNIX* ve *Windows* sistemleri için aşağıdaki gibi bir örnek verebiliriz:

```
short int a, b;        /* 2 byte */
int c;                 /* 4 byte */
...

a = 1000;
b = 100;
...
c = a * b;      short (int) * short (int) ► 100000 (int)
```

Örneklerimizde hep aritmetik operatörleri kullandık. Otomatik dönüştürme işlemi ilişkisel operatörler için de geçerlidir. Aşağıdaki örneği inceleyiniz:

```
char ch;
...
ch = 150;
```

```
if (ch == 150)
    printf("Eşit\n");
else
    printf("Eşit değil\n");
```

Size biraz tuhaf gelebilir; fakat yukarıdaki örnekte `if` deyimi **Yanlış** olarak değerlendirilecektir. Dolayısıyla bu örnekte ekrana **"eşit değil"** yazısı çıkar. Nedenini adım adım açıklayalım.

1. Adım: `ch = 150` işleminin sağ taraf değeri `int` sol taraf değeri `[signed] char` türündendir. `150` `char` türüne dönüştürüldükten sonra atama işlemi yapılır. `150 (0x0096) [signed] char` türüne dönüştürüldüğünde `-106` gibi negatif bir sayı elde edilir.

2. Adım: `if` parantezinin içerisindeki `ch == 150` ifadesi **Yanlış (sıfır)** sonucunu verir. Çünkü `-106` değerine sahip olan `ch`, `int` türüne dönüştürüldükten sonra karşılaştırma işlemine sokulacaktır.

`ch = 150 ► char (int) == int ► 0xFF96 == 0096 ► 0 (int)`

Açıkladığımız bu kurallar size için biraz karmaşık gelebilir. Yukarıdaki örneklerden küçük bir özet çıkarabiliriz.

1. Bir operand `long double` türünde ise diğeri işlem öncesi bu türe dönüştürülür.

2. Operandlardan bir tanesi `double` türündeyse diğeri de (**long double** değilse) `double` türüne dönüştürülür. Sonuç `double` türündendir.

3. Bir operand `float` diğeri tamsayı türlerindense diğer operand `float` türüne dönüştürülür.

4. `short` ve/veya `char` türünden iki operand işlem öncesi `int` türüne dönüştürülür. (Integral promotion.)

5. İşaretli ve işaretsiz türler arasındaki işlemlerde dönüştürme işaretsiz türe doğru yapılır.

6. Yukarıdaki durumların dışında büyük türe dönüştürme kuralları geçerlidir.

Bu kurallar şimdi var olan ve gelecekte tasarlanacak tüm sistemler için geçerlidir. Son olarak, en sık rastlanan durumları bir tablo biçiminde vermek istiyoruz. Her iki operand da sonucun türüne dönüştürüldükten sonra işleme sokulmaktadır.

1. Operand	2. Operand	Sonuç
char	char	int
char	short	int
char	int	int
char	long	long
short	short	int
short	int	int
int	long	long
int	unsigned int	unsigned int
unsigned int	long	long
long	unsigned long	unsigned long
float	float	float
int	float	float
long	float	float
float	double	double
double	long double	long double

12.4 BİLİNÇLİ TÜR DÖNÜŞTÜRMELERİ

Bilinçli tür dönüştürmeleri, programcının bilerek ve isteyerek derleyiciye yaptırdığı dönüştürmelerdir. Bilinçli tür dönüştürmeleri, **tür dönüştürme operatörü** kullanılarak yapılır. Genel biçimi aşağıdaki gibidir:

```
(tür) değişken ya da sabit
```

Burada `(tür)`, dönüştürmenin yapılacağı türü gösterir. Örneğin:

```
int a, b;
...
a = 10000;
b = 5;
...
x = (long) a * b;
```

Bu ifadede olanları adım adım izleyelim:

1. Adım: `a` değişkeni `long` türüne dönüştürülür.

2. Adım: İşlem öncesi otomatik tür dönüşüm kuralına göre `b` değişkeni de derleyici tarafından `long` türüne dönüştürülecektir.

3. Adım: İki `long` türünden sayı çarpılır; böylece sonuç da `long` türünde olur. Çarpım sonucunda bilgi kaybı söz konusu değildir.

Bir başka örnek:

```
int a, b;
double c;
```

```
...
a = 10;
b = 4;
..
c = (double) a / b;
```

Burada a değişkeni bilinçli olarak double türüne dönüştürülmüştür. Derleyici b değişkenini de double türüne dönüştürdükten sonra işleme sokar.

12.5 TÜR DÖNÜŞTÜRME OPERATÖRÜ

Tür dönüştürme operatörü tek operand alan önek bir operatördür. Bu operatörü öncelik sırasını değiştiren parantez operatörü ile karıştırmayınız. Tür dönüştürme operatörü şu ana kadar gördüğümüz artırma (++), eksiltme (—), mantıksal değil (!), bit değil (~), operatörleriyle **sağdan sola** eşit önceliklidir.

```
++   —   !   ~   (tür)   Sağdan Sola
```

Aşağıdaki örnekleri inceleyiniz:

```
x = (long) a * b;
```

Yalnızca a bilinçli olarak long türüne dönüştürülüyor.

```
...
x = (long) (a * b);
```

a * b işleminin sonucu long türüne dönüştürülüyor.

```
...
x = (double) ((long) a * b);
```

a önce long türüne dönüştürülüyor, çarpma işlemi yapılıyor, daha sonra çıkan sonuç double türüne dönüştürülüyor.

SORAMADIKLARINIZ...

S1) 16 bit sistemlerde bir karakter değişkeninin içeriğini printf fonksiyonu ile *HEX* biçiminde aşağıdaki örnekte olduğu gibi ekrana yazdırmak istiyoruz:

```
char ch = 0xF6;
...
printf("x = %x\n", ch);
...
```

Ekrana F6 sayısının basılması gerekmez mi? Oysa, FFF6 sayısı basılıyor, neden?

C1) printf fonksiyonu "%x" formatı ile int türüne dönüştürdükten sonra *HEX* biçimde yazmaya çalışır. Yani bu örnekte ch değişkeni önce int türüne dönüştürülecek , sonra ekrana yazdırılacaktır.

```
ch = 0xF6 = 1111 0110B
```

işaret biti 1 değerinde olduğu için ch negatif bir sayıdır. int türüne negatifliği korunarak dönüştürülür.

```
(int) ch ► FFF6H
```

Çıkan FFF6 sonucu normaldir. Fakat ch değişkeni, [signed] char yerine unsigned char biçiminde tanımlansaydı bu problem ortaya çıkmayacaktı.

```
unsigned char ch = 0xF6;
...
printf("x = %x\n", ch);
...
```

Çünkü artık ch, int türüne dönüştürülürken negatif olarak değerlendirilmeyecektir.

```
(int) ch ► 0x00F6
```

ch değişkenini unsigned char biçiminde tanımlamak yerine, bu türe bilinçli dönüşüm de yapılabilir.

```
char ch = 0xF6;
...
printf("%x\n", (unsigned char) ch);
..
```

Uzun tamsayıları *HEX* biçiminde yazdırmak için "lx" formatının kullanıldığını da anımsayınız...

13 YER VE TÜR BELİRLEYİCİLERİ

Belirleyiciler nesnelerin ikincil özellikleri hakkında bilgi veren anahtar sözcüklerdir. Belirleyicilerin önemli bir kısmı yer ve tür belirleyicileri adı altında bu bölümde ele alınmaktadır. Ancak bazı belirleyicilerin okuyucu tarafından tam olarak anlaşılması mümkün olmayabilir. Bu durumda paniğe kapılmadan ilerlemenizi, daha sonra bu konuya dönerek bir tekrar yapmanızı salık veririz.

Bölüm içerisinde verilen örnekleri, şu ana kadar gelinen düzeyi gözönünde bulundurarak ve ancak anlatılan konuları kapsayacak biçimde seçtik. Konuyu daha iyi açıklayabilecek olduğu halde karmaşık örneklerden mümkün olduğunca kaçındık.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Yer ve tür belirleyici anahtar sözcükler nelerdir?

2) Bildirim işleminde belirleyiciler nasıl kullanılır?

3) `register` belirleyicisi hangi amaçla kullanılır, işlevi nedir?

4) `static` yerel ve global değişkenlerin ömürleri ve faaliyet alanları nasıldır; hangi amaçla kullanılırlar?

5) Modül kavramını açıklayınız?

6) Büyük projeleri modüllere ayırarak tasarlamanın avantajları nelerdir?

7) `extern` belirleyicisinin işlevi nedir?

8) `const` belirleyicisi hangi amaçla kullanılır?

13.1 GENEL OLARAK BELİRLEYİCİLER (Specifiers)

Belirleyicileri "yer belirleyicileri" ve "tür belirleyicileri" olmak üzere iki gruba ayırabiliriz. Yer belirleyicileri genel olarak nesnelerin tutuldukları yerler hakkında bilgi verirler. C'de dört tane yer belirleyici anahtar sözcük vardır:

```
auto
register
```

```
static
extern
```

Yer belirleyici anahtar sözcüklerden `auto`, `static` ve `extern` nesnelerin ömürleri üzerinde de etkili olurlar. Tür belirleyicileri ise nesnelerin içerisindeki değerlerin değiştirilmesine ilişkin bilgi verirler. C'de iki "tür belirleyici anahtar sözcük vardır."

```
const
volatile
```

13.2 YER VE TÜR BELİRLEYİCİLERİYLE BİLDİRİM İŞLEMİ

Bildirim işlemini yer ve tür belirleyicilerini de içerecek biçimde yeniden ele alacağız.

Genel biçim:

```
[yer belir.] [tür belir.]     <tür>  nesne1[, [nesne2], [nesne3],...];
auto          const
register      volatile
static
extern
```

(Yukarıdaki genel biçimi 5. Bölümde verilenle karşılaştırınız.)

Yer belirleyicisi, tür belirleyicisi ya da tür ifade eden anahtar sözcüklerin dizilimi herhangi bir biçimde olabilir. Örneğin:

```
auto const int a = 10;
const auto int a = 10;
int const auto a = 10;
int auto const a = 10;
...
```

bildirimlerinin hepsi geçerlidir. Fakat anahtar sözcükleri yukarıda belirttiğimiz sırada kullanmanızı salık veririz; yani önce yer belirleyicisi daha sonra tür belirleyicisi ve ondan sonra da türe ilişkin anahtar sözcük.

```
auto const int a = 10;
```

gibi...

İki yer belirleyici anahtar sözcük birlikte kullanılamaz. Örneğin:

```
auto register int a;     /* Hata! auto ve register birarada kullanılmaz */
extern static float b;   /* Hata! extern ve static aynı anda kullanılmaz */
...
```

`extern` yer belirleyici anahtar sözcüğü ile tür belirleyici anahtar sözcükler birlikte kullanılmaz.

```
extern const int a =  10;      /* Hata! extern ile tür belirleyicisi
                                  const birlikte kullanılamaz */
extern volatile int a = 10;    /* Hata! extern ile tür belirleyicisi
                                  volatile birlikte kullanılamaz */
```

13.3 auto BELİRLEYİCİSİ

auto yerel değişkenler için kullanılan bir yer belirleyicisidir. Bu anahtar sözcük, nesnenin faaliyet alanı bittikten sonra kaybolacağını gösterir. Hatırlayacağınız gibi, yerel değişkenler ilgili blok icra edilmeye başlandığında yaratılıyorlar, bloğun icrası bittikten sonra da kayboluyorlardı. İşte auto belirleyicisi bu durumu vurgulamak için kullanılmaktadır. Zaten bir yerel değişken başka bir yer belirleyici anahtar sözcük kullanılmadığı sürece (default olarak) auto biçiminde ele alınır.

```
{
    auto int a;
    float b;  /* derleyici auto olarak ele alıyor */
    ...
}
```

80X86 Sembolik Makina Dili Programcısına Not: auto yer belirleyicisine sahip nesneler *STACK* bölgesini kullanırlar.

auto yer belirleyicisi global değişkenlerle ya da parametre değişkenleriyle birlikte kullanılamaz. Örneğin:

```
auto int a;              /* Hata! Global bir değişken auto biçiminde
                            bildirilemez */

fonk (auto int b)        /* Hata! Parametre değişkeni auto biçiminde
                            bildirilemez */
{
   auto int c;.          /* Yerel değişken auto olabilir */
   int d;.               /* Yerel değişken kendiliğinden auto olarak
                            ele alınır */
   ...
}
```

13.4 register BELİRLEYİCİSİ

register belirleyicisi, değişkenin "bellekte değil de *CPU* yazmaçlarının içerisinde" tutulacağını belirten bir anahtar sözcüktür. Değişkenlerin bellek yerine yazmaçlar içerisinde tutulması programın çalışmasını hızlandırır. Öncelikle yazmaç (register) kavramını kısaca açıklamayı uygun buluyoruz.

13.4.1 Yazmaç Nedir?

Yazmaç (register) CPU *(Central Processing Unit)* içerisinde bulunan tampon bellek bölgeleridir. CPU içerisindeki aritmetik ve mantıksal işlemleri yapan birimin yazmaçlar ve belleklerle bağlantısı vardır. Genel olarak CPU tarafından yapılan aritmetik ve mantıksal işlemlerin her iki operandı da belleğe ilişkin olamaz. Örneğin bellekte bulunan **data1** ve **data2** ile gösterdiğimiz 2 sayıyı toplayarak **data3** ile gösterdiğimiz başka bir bellek bölgesine yazmak isteyelim. Bu *C*'deki

```
data3 = data1 + data2;
```

işlemine karşılık gelmektedir. *CPU* bu işlemi ancak 3 adımda gerçekleştirebilir:

1) Önce **data1** bellekten CPU yazmaçlarından birine çekilir ► `MOV reg, data1`

2) Yazmaç ile **data2** toplanır ► `ADD reg, data2`

3) Toplam **data3** ile belirtilen bellek alanına yazılır.►`MOV data3, reg`

Belleğe yazma ve bellekten okuma işlemleri yazmaçlara yazma ve yazmaçlardan okuma işlemlerine göre daha yavaştır. Çünkü belleklere erişim için belli bir makina zamanı gerekir.

CPU yazmaçları hangi sistem söz konusu olursa olsun sınırlı sayıdadır. Bu nedenle birkaç değişkenden fazlası register belirleyicisi ile tanımlanmış olsa bile yazmaçlarda saklanamayabilir. *C* derleyicileri yazmaçlarda saklayamayacakları değişkenler için genel olarak hata ya da uyarı mesajı vermezler. Yani derleyiciler tutabilecekleri yazmaç sayısından fazla **register** belirleyicisine sahip değişkenler ile karşılaştıklarında bunlara ilişkin **register** belirleyicilerini dikkate almazlar.

register belirleyicisi ancak yerel ya da parametre değişkenleri ile kullanılabilir; global değişkenler ile kullanılamaz.

Örneğin:

```
register int a;                 /* Hata! */

int sample(register int x)      /* Hata değil */
{
    register float x;           /* Hata değil */
    ...
}
```

Ne kadar değişkenin yazmaçlarda saklanabileceği bilgisayar donanımlarına ve derleyicilere bağlıdır. Ayrıca, uzunluğu tamsayı (int) türünden büyük olan türler genellikle yazmaçlarda saklanmazlar; bu durumlarda da derleyicilerden hata veya uyarı mesajı beklenmemelidir.

Sonuç olarak, **register** belirleyicisi hızın önemli olduğu çok özel ve kısa kodlarda ancak birkaç değişken için kullanılmalıdır. **register** belirleyicisinin

olur olmaz yerde bilinçsizce kullanımı çalışma hızının yavaşlamasına bile neden olabilir!..

13.5 static BELİRLEYİCİSİ

`static` belirleyicisine sahip değişkenler programın çalışması süresince bellekten kaybolmazlar. Bir bakıma `static` belirleyicisi, `auto` belirleyicisinin zıt anlamlısıdır. `static` belirleyicisi parametre değişkenleriyle birlikte kullanılamaz; ancak yerel ya da global değişkenlerle birlikte kullanılabilir.

`static` anahtar sözcüğünün yerel ya da global değişkenlerle kullanımını ayrı başlıklar halinde inceleyeceğiz.

13.5.1 static Yerel Değişkenler

`static` yer belirleyicisine sahip olan yerel değişkenler programın icrası boyunca bellekte kalırlar. Başka bir deyişle, `static` anahtar sözcüğü yerel değişkenlerin ömrünü uzatmaktadır. `static` yerel değişkenler tıpkı global değişkenler gibi programın çalışmasıyla yaratılır, programın icrası bitene kadar da bellekte tutulurlar.

80X86 Sembolik Makina Dili Programcısına Not: `static` yerel değişkenler derleme aşamasında derleyici tarafından *DATA SEGMENT* bölgesine yerleştirilirler.

`static` yerel değişkenler programcı tarafından ilkdeğer verildikten sonra kullanılırlar. **İlkdeğer verme işlemi programın çalışması sırasında değil, derleme zamanında derleyici tarafından yapılır.** `static` yerel değişkenler ilk değerleriyle birlikte belleğe yüklenirler.

Aşağıdaki örneği yazarak deneyiniz.

```
#include <stdio.h>

int inc(void)
{
    static int x = 0;          /* Bu kısım yalnızca derleme sırasında
                                  ve bir kez işlem görür */

    ++x;                       /* Fonksiyonun her çağırılışında x
                                  yeniden yaratılmaz, değerini korur */
    return x ;
}

void main(void)
{
    int a;

    a = inc();
    printf("a = %d\n", a);     /* a =  1 */
    a = inc();
```

```
    printf("a = %d\n", a);          /* a = 2 */
}
```

Bu örnekte `inc` fonksiyonunun içerisindeki yerel `x` değişkeninin değeri fonksiyonun her çağrılışında bir artırılır. Çünkü `static` yerel bir değişken olduğu için `x`, fonksiyonun her çağırılışında yeniden yaratılmayacak, değerini koruyacaktır.

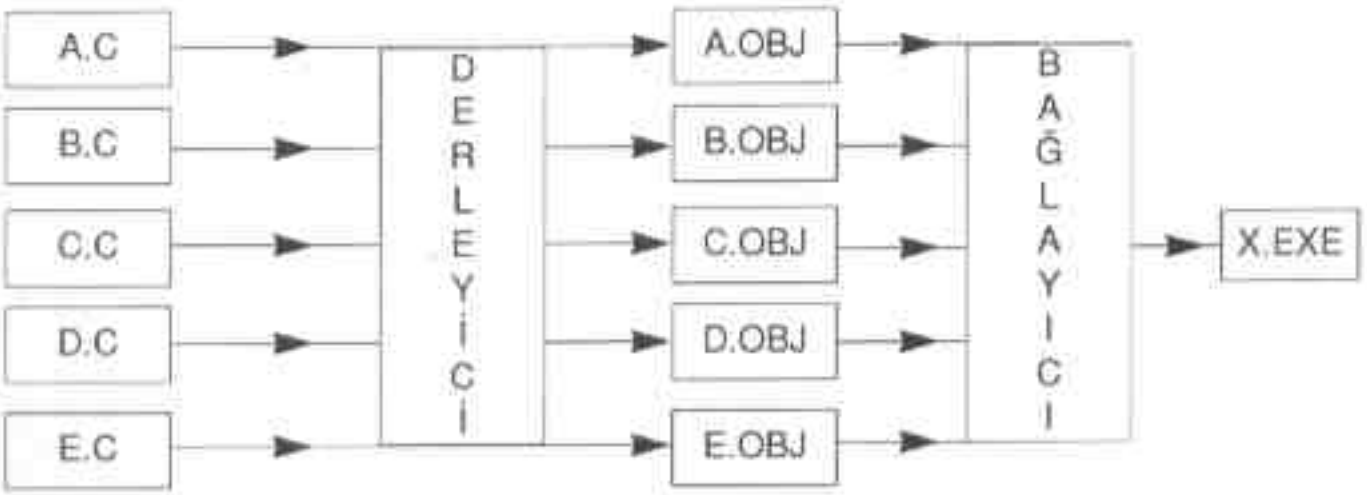
Peki, `static` yerel değişkenlerin global değişkenlerden farkı nedir? Örneğin yukarıdaki `inc` fonksiyonunda yerel `x` değişkeni global olsaydı farkeden şey ne olurdu? Evet, ömür açısından global değişkenlere benzemekle birlikte global değişkenler ile `static` yerel değişkenlerin faaliyet alanları farklıdır. `static` yerel değişkenler, diğer yerel değişkenler gibi blok faaliyet alanı kuralına uyarlar.

13.5.2 static Global Değişkenler

`static` belirleyicisi global bir değişken ile birlikte kullanılırsa, global değişkenin faaliyet alanını yalnızca tanımlandığı modülü kapsayacak biçimde daraltır. `static` global değişkenler yalnızca tanımlandıkları modül içerisinde faaliyet gösterebilirler. Sanırız, bu noktada modül terimini açıklamamız gerekiyor.

Bir proje birbirlerinden bağımsız olarak derlenebilen birden fazla kaynak dosyadan oluşabilir. Projelerin bağımsız olarak derlenebilen herbir kaynak dosyasına modül denir. Örneğin bir **X** projesi, **A.C, B.C, C.C, D.C, E.C** dosyalarından oluşmuş olsun. Aşağıdaki şekli inceleyiniz.

13.5.2.1 Modül Nedir?



Burada modüller birbirlerinden bağımsız olarak derlendikten sonra, hepsi birlikte bağlayıcı (linker) ile birleştirilerek tek bir `X.EXE` dosyası elde edilmiştir.

Büyük projelerin modüllere ayrılmasının ne gibi faydaları vardır? İlk söyleyeceğimiz şey şu olacak: Yukarıdaki örnekte eğer siz, bütün modülleri tek bir kaynak kod içinde birleştirirseniz; en ufak bir değişiklikte tüm projeyi tekrar derlemek zorunda kalırsınız. Oysa modüllere ayrılmış projelerde yalnız değişikliğin yapıldığı modülün derlenmesi yeterlidir. Çünkü diğer modüller zaten derlenmiştir

ve onlar yalnızca bağlama aşamasında işlem görürler. Modüller halinde yazmanın bir diğer avantajı da grup çalışması yaparken ortaya çıkar. Bu durumda projelerin bağımsız parçaları (modülleri) ayrı kişiler tarafından hazırlanabilir.

Global bir değişken normal olarak tüm modüller içerisinde faaliyet gösterebilirken, (bu durumda diğer modüllerde **extern** bildiriminin yapılmış olması gerekir) **static** global değişkenler yalnızca tanımlandıkları modül içerisinde faaliyet gösterebilirler.

Tür	Faaliyet alanı	Ömürü
Global Değişken	Dosya	Programın çalışma zamanı kadar.
static Yerel Değişken	Blok	Programın çalışma zamanı kadar
static Global Değişken	Modül	Programın çalışma zamanı kadar
Yerel Değişken	Blok	Blok icra edilmeye başlandığında yaratılır; blok icrası bitince kaybolur

Global değişkenler için **static** tanımlamasının yalnızca faaliyet alanı üzerinde etkili olduğuna, ömür üzerine etkili olmadığına dikkat ediniz.

80X86 Sembolik Makina Dili Programcısına Not: `static` global değişkenler derleyici tarafından DATA SEGMENT bölgesine yerleştirilir. Bu değişkenler ilgili modül içerisine `PUBLIC` olarak yazılmadıklarından diğer modüllerde kullanılamazlar.

13.6 extern BELİRLEYİCİSİ

Bütün belirleyiciler içinde belki de anlaşılması en güç olanı **extern** belirleyicisidir. **extern** belirleyicisi genel olarak nesnenin başka bir modülde bulunduğunu bildirmek için kullanılır. Bir örnekle açıklayalım:

Bir proje **A.C** ve **B.C** biçiminde iki modülden oluşmuş olsun:

A.C

```
int a;

float x1()
{
    ...
    a = 100;
    ...
}

main()
{
    ...
}
```

B.C

```
int y1()
{
    ...
    a = 300; /* hata */
}
```

A.C modülünde tanımlanmış olan global **a** değişkeni dosya faaliyet alanına sahip olduğu için normal olarak **B.C** modülü içerisinde de faaliyet gösterebilir, değil mi? Fakat iki modülün ayrı ayrı derlendiği yukarıdaki örnekte problemli bir durum söz konusudur. Çünkü **B.C** modülünün derlenmesi sırasında derleyici **a** değişkeninin **A.C** modülü içerisinde global olarak tanımlandığını bilemez. Böylece **B.C** modülünü derlerken **a** değişkeni hakkında hiçbir bilgi bulamayan derleyici, bu durumu hata (error) olarak belirler (bir değişken kullanılmadan önce bildirimin yapılması gerektiğini anımsayınız). **İşte extern belirleyicisi derleyiciye, ilgili global değişkenin kendi modülü içerisinde değil de bir başka modül içerisinde tanımlı olduğunu bildirmek amacıyla kullanılmaktadır.**

B.C modülündeki **a** değişkenini **extern** olarak bildirerek problem ortadan kaldırılabilir.

B.C

```
extern int a;     /* a başka bir modülde tanımlanmış */

int y1()
{
    ...
    a = 300;
    ...
}
...
```

extern bildirimini gören derleyici değişkenin başka bir modülde tanımlandığını varsayarak hata durumunu ortadan kaldırır. Ancak, derleyici makina kodu üretirken **extern** olarak bildirilmiş bir değişkenin bellekteki yerini tespit edemeyeceğinden, bu işlemi bütün modülleri gözden geçirecek olan bağlayıcı programa bırakır. Böylece değişkenin tanımlandığı modülü bulup, **extern** olarak bildirilmiş olanlarla ilişkilendirme işlemi bağlayıcı (linker) tarafından yapılmaktadır. Yani **extern** belirleyicisi ile programcı derleyiciye, derleyici ise bağlayıcıya bildirimde bulunmaktadır.

extern belirleyicisini gören derleyicinin bellekte bir yer ayırmadığını vurgulayalım. Bu nedenle **extern** bildirimi -tıpkı fonksiyon prototiplerinde olduğu gibi- bir tanımlama işlemi değildir.

80X86 Sembolik Makina Dili Programcısına Not: C derleyicileri **extern** belirleyicisiyle karşılaştıklarında amaç kod içerisine ilgili değişkenin başka modülde olduğunu bağlayıcıya anlatmak için EXTRN bildirimini yerleştirirler. Örneğin:

```
extern int a;
```

biçiminde bir bildirim ile derleyici amaç kod içerisine bağlayıcı için,

```
EXTRN _a: word
```

bildirimini yazar. Sembolik makina dilinde EXTRN bildirimi kaynak kodun herhangi bir yerinde yapılabilir. Ancak hangi segment içerisinde kullanılacaksa bildirimin o segment içerisinde yapılması salık verilmektedir.

Yalnız değişkenler için değil, çağrıldıkları modüllerde tanımlanmamış olan fonksiyonlar için de **extern** belirleyicisinin kullanılması söz konusudur. **C derleyicileri, kendi modülleri içerisinde tanımlanmadıkları halde çağrılan (standart C fonksiyonları gibi) fonksiyonları otomatik olarak extern kabul ederler. Bu nedenle fonksiyon prototiplerinde ayrıca extern belirleyicisini yazmaya gerek yoktur; çünkü derleyici tarafından yazılmış varsayılır.** Örneğin yukarıda verdiğimiz birinci örnekte **B.C** modülünde bulunan **y1** fonksiyonu içerisinde, **A.C** modülünde bulunan **x1** fonksiyonu çağrılıyor olsun:

A.C

```
int a;

float x1()
{
    ...
    a = 100;
    ...
}

main()
{
    ...
}
```

B.C

```
extern int a;
extern float x1(void);

int y1()
{
    float f;
    ...
    f = x1();
    a = 300;
    ...
}
```

B.C modülünde **x1** fonksiyonu için yazılmış olan prototip ifadesini inceleyiniz:

```
extern float x1(void);
```

Bu örnekte **x1** fonksiyonu başka bir modülde tanımlı olduğu için prototip ifadesine **extern** belirleyicisi konulmuştur; ancak konulmasaydı derleyici zaten **extern** varsayacaktı. (Tıpkı yerel değişkenler için **auto** belirleyicisini varsaydığı gibi.)

Son olarak, **extern** belirleyicisinin tek bir modül söz konusu olduğundaki "amaç dışı kullanımını" ele almak istiyoruz. Aşağıdaki örnekte **main** fonksiyonu içerisindeki global **x** değişkeni, tanımlanmadan önce kullanıldığından hataya neden olmaktadır.

```
void main()
{
    ...
    x = 100;
    ...
}

int x;

int fonk()
{
    ...
    x = 300;
    ...
}
```

Derleme işlemi bu noktaya geldiğinde x hakkında hiçbir bilgi edinilmemiş!

Derleme işleminin bir yönü vardır.
Bu yön yukarıdan aşağıya doğrudur.

Gerçekten de yukarıdaki kodu derlerseniz, **main** fonksiyonun içerisindeki **x** değişkenin bildiriminin bulunamadığını ifade eden bir hata mesajıyla karşılaşırsınız. Bu durumda, eğer bir global değişkeni tanımlamadan önce kullanıyorsanız, hata oluşmaması için daha önce **extern** bildiriminde bulunmalısınız.

```
extern int x;

void main()
{
    ...
    x = 100;
    ...
}

int x;

int fon()
{
    ...
    x = 200;
    ...
}
```

Tabi, **extern** bildiriminde bulunmak yerine global değişkeni programın tepesinde tanımlamak çok daha doğal bir çözümdür.

80X86 Sembolik Makina Dili Programcısına Not: Yukarıdaki örnekte derleyici a değişkenini hem DATA SEGMENT bölgesine yerleştirip hem de EXTRN olarak belirleyemez. Bu yüzden aynı modül içerisinde gerçek tanımlamayla karşılaştığında EXTRN bildirimini amaç kod içerisine yazmayacaktır. Dolayısıyla bu örnekte **extern** belirleyicisi gerçek amacıyla kullanılmamıştır.

13.6.1 extern Bildirimlerinin Yapılış Yerleri

`extern` bildirimleri kaynak kodun herhangi bir yerinde yapılabilir. Global ya da yerel olması söz konusu değildir; bildirimin yapıldığı yerden dosya sonuna kadar olan bölge içinde geçerlidir. Ancak `extern` bildirimlerinin programın tepesinde ya da programcıya ait bir başlık dosyasının içinde yapılmasını salık veririz.

13.7 const BELİRLEYİCİSİ

`const`, ilkdeğer atandıktan sonra nesnenin içeriğinin değiştirilemeyeceğini anlatan tür belirleyici bir anahtar sözcüktür. Yerel, global ve parametre değişkenleriyle birlikte kullanılabilir.

Örneğin:

```
const double PI = 3.14159265; /* Geçerli */

main(void)
{
    const int i = 10;          /* Yalnızca tanımlanırken ilk değer
                                  verilebilir */
    ...
    i = 100;                   /* Hata! i'nin değeri değiştirilemez */
    ...
}
```

Bir değişken `const` belirleyicisi ile tanımlanacaksa ilkdeğer verilmelidir. Aksi halde `const` belirleyicisi kullanmanın bir anlamı kalmaz. Aşağıdaki örneği inceleyiniz:

```
void sample (void)
{
    const int a;     /* anlamsız! */
    ...
}
```

Bu örnekte a yerel bir değişkendir; dolayısıyla rastgele bir değere sahiptir. İçeriğini bir daha değiştiremeyeceğimize göre tanımlanmasının da bir anlamı olamaz.

`const` belirleyicisinin kullanım amacı ne olabilir diye düşünebilirsiniz? Sıklıkla şu merak edilir:

"Eğer `const` belirleyicisi koruma amaçlı olarak kullanılıyorsa kim kime karşı korunuyor?.." `const` belirleyicisinin iki yararlı işlevi vardır:

1) Okunabilirliği artırır. Çünkü programı inceleyen bir kişi `const` belirleyicisine sahip değişkenin değerinin bir daha değiştirilmeyeceğini düşünerek daha fazla bilgi edinir.

2) Yanlışlıkla nesnenin değerinin değiştirilmesi engellenir.

`const` belirleyicisi değeri hiç değişmeyecek sabitler için kullanılmalıdır. `const` bildirimlerinin nesne yarattığına nesnenin yalnızca okunabildiğine (read only) dikkat ediniz.

Not: C++'da `const` belirleyicisi zorunluluk olmadıkça nesne yaratmaz. (`#define` önişlemci komutu gibidir; ancak derleme aşamasında işlem görür.)

13.8 volatile BELİRLEYİCİSİ

Derleyiciler optimizasyon amacıyla nesneleri geçici olarak yazmaçlarda tutabilirler. Yazmaçlardaki bu çeşit geçici barınmalar `register` belirleyicisi kullanılmasa da derleyiciler tarafından yapılabilir.

Örneğin:

```
int kare (int a)
{
    int x;

    x = a * a;
    return x;
}
```

Yukarıdaki fonksiyonda **x** geçici bir değişkendir; dolayısıyla derleyici **x** değişkenini bellekte bir yerde saklayacağına, geçici olarak yazmaçlarından birinde saklasa da işlevsel bir farklılık ortaya çıkmaz. Bu çeşit uygulamalarda derleyicinin değişkenleri geçici olarak yazmaçlarda saklaması işlemleri hızlandırmaktadır. Aşağıdaki kodu inceleyiniz:

```
int a;
int b;
...

a = b;
if (a == b) {
    ...
}
```

Bu örnekte doğal olarak ilk adımda **b** değişkeni **a** değişkenine aktarılmak üzere yazmaçlardan birine çekilecektir. Ancak derleyici `if` içerisindeki ifadede

```
a == b
```

karşılaştırmasını yapmak için bellekteki **b** yerine yazmaçtaki **b**'yi kullanabilir!..

Verdiğimiz iki örnekte de derleyici birtakım optimizasyonlarla programı işlevi değişmeyecek biçimde daha hızlı çalışır hale getirmek istemiştir. Ancak kimi uygulamalarda derleyicilerin bu biçimde davranması hatalara neden olabilmektedir. İkinci örnekte:

```
a = b;

/* Kesme gelerek b'yi değiştirebilir! */

if (a == b) {
    ...
}
```

Bir donanım kesmesi (örneğin 8h gibi) b'yi değiştiriyorsa, bu durum if deyimi tarafından farkedilmeyebilir. İşte bu tür durumlarda değişkenlerin optimizasyon amacıyla geçici olarak yazmaçlarda tutulması arzu edilmeyen sonuçların oluşmasına yol açabilmektedir. volatile **"Değişkenleri optimizasyon amacıyla yazmaçlarda bekletme, onları bellekteki gerçek yerlerinde kullan!"** anlamına gelen bir tür belirleyicisidir. Bu anlamıyla volatile belirleyicisini register belirleyicisi ile zıt anlamlı olarak düşünebiliriz. Yukarıdaki örnekte b değişkenini volatile olarak bildirerek anlattığımız gibi bir problemin çıkması engellenebilir.

```
int a;
volatile int b;
   ...
```

volatile çok özel uygulamalarda kullanılabilen bir belirleyicidir.

Yerel, parametre ya da global değişkenlerle birlikte kullanılabilen volatile belirleyicisi ancak çok özel uygulamalarda önemli olabilmektedir. Bu belirleyicinin bilinçsizce kullanılmasının performansı kötü yönde etkileyebileceğini unutmayınız!

Not: volatile belirleyicisinin işlevini tam olarak anlamamış olabilirsiniz. Çünkü bu belirleyicinin işlevinin anlaşılması için kavramsal bilginin geliştirilmesi gerekebilir. O zaman, kaygıya kapılmadan ilerleyin! Birgün tekrar okuduğunuzda ne demek istediğimizi daha iyi anlayacaksınız.

SORAMADIKLARINIZ...

S1) static yerel değişkenler ne amaçla kullanılıyorlar? Onların yerine global değişkenler kullanılsa işlevsel bir farklılık ortaya çıkar mı?

C1) static yerel değişkenler ile global değişkenlerin ömürleri aynı fakat faaliyet alanları farklıdır. static yerel değişkenlerin faaliyet alanlarının dar olması (blok faaliyet alanına sahip olması) soyutlamayı (abstraction) ve yeniden kullanılabilirliği (reusability) kuvvetlendirmektedir. Çünkü global değişkenlere bağlı olan fonksiyonlar kolaylıkla projeden projeye taşınamazlar ve kendi içlerinde bir bütünlük oluşturamazlar.

S2) Farklı modüllerde aynı isimli iki static global değişken tanımlanabilr mi?

C2) Tanımlanabilir. *C*'de "aynı faaliyet alanına sahip aynı isimli birden fazla değişken tanımlanamaz" kuralını anımsayınız. İki farklı modülde tanımlanan aynı isimli iki `static` global değişkenin faaliyet alanları farklıdır, değil mi?

S3) Farklı modüllerin birleştirilmesi nasıl olur?

C3) Bu soru aslında *C*'ye ilişkin bir soru değildir. Çünkü modülleri doğru bir biçimde organize ettikten ve derledikten sonra artık *C* ile bir ilişkimiz kalmıyor. Modüllerin nasıl birleştirileceği konusu ise işletim sistemine ve kullandığınız bağlayıcı programa bağlıdır. *DOS* altında çalışan derleyicilerin tümleşik çevreli uyarlamalarında farklı modüllerin birleştirilmesi "bir proje dosyası oluşturulmasıyla" mümkündür.

S4) Bir global değişkenin hiçbir modülde tanımlanmadığını ancak tüm modüllerde `extern` olarak bildirildiğini varsayalım. Bu durumda ne olur?

C4) Bu durumda bütün modüller hatasız olarak derlenir. Hata bağlama aşamasında, bağlayıcının `extern` olarak bildirilen nesneyi hiçbir modülde bulamaması biçiminde ortaya çıkacaktır.

14 DÖNGÜLER

Bu bölümde C'nin döngü yapısı ele alınmaktadır. C, döngüler bakımından oldukça yalın, güçlü ve esnek olarak tasarlanmıştır. C'deki bu esneklik diğer programlama dillerinde çalışmış olanları olumsuz yönde etkileyebilmektedir. Bu nedenle döngülerin çalışma biçimlerinin eksiksiz bir biçimde anlaşılması C öğrencileri için büyük önem taşır. Programın çalışma zamanının büyük bölümünü döngülerde harcadığını düşünürsek bunun nedeni daha iyi anlaşılabilir.

İngilizce **"loop"** sözcüğünün karşılığı olarak yazarlar ya **"döngü"** ya da **"çevrim"** sözcüklerini kullanıyorlar. Biz kitabımızda **"döngü"** sözcüğünü tercih ettik.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1. **while** döngüsü nasıl çalışır?
2. Kontrolün başta yapıldığı **while** döngüsü ile sonda yapıldığı **while** döngüsü arasındaki ayrımlar nelerdir?
3. **for** döngüleri nasıl çalışır?
4. **while** döngüleri ile **for** döngüleri arasındaki farklılıklar ve benzerlikler nelerdir?
5. **break** ve **continue** anahtar sözcüklerinin işlevleri nelerdir?

14.1 GENEL OLARAK DÖNGÜLER

Döngüler, programın belli bir bölümünün yinelemeli olarak icra edilmesini sağlayan programlama dillerinin "olmazsa olmaz" deyimleridir. Programın akışı üzerinde etkili olduğu için döngü deyimlerini birer **kontrol deyimi** olarak ele alabiliriz.

Programlama dillerini incelediğimizde temel olarak 2 tür döngü deyiminin bulunduğunu görüyoruz:

1) `for` döngüleri
2) `while` döngüleri

for döngüleri belli bir sayıda yineleme sağlamak amacıyla kullanılırlar. Aşağıdaki BASIC örneğinde:

```
100 FOR K = 1 TO 100
.
.
.
150 NEXT K
...
```

for döngüsü, **K** değişkeni 1'den **100**'e kadar değer alarak toplam **100** kez yinelemeye neden olur. Programlama dillerinin çoğunda **for** döngüleri sintaks olarak birbirlerine benzer. Örneğin PASCAL'da **for** döngüleri şu biçimdedir:

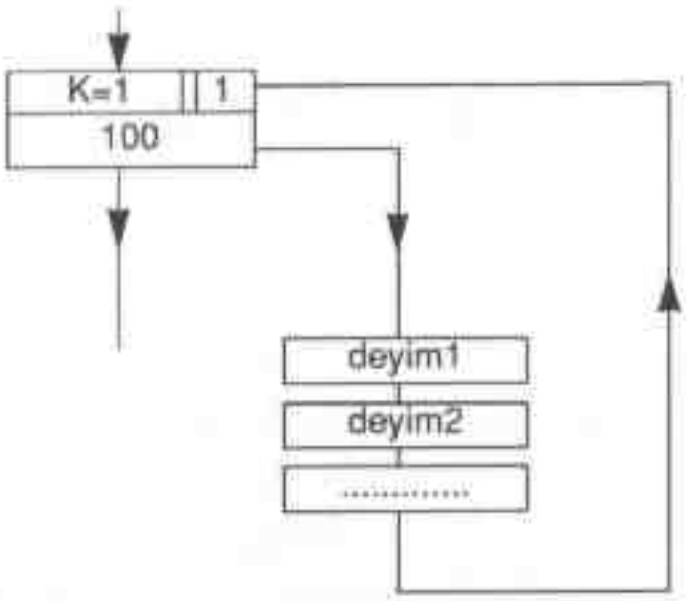
```
FOR K := 1 TO 100 DO
BEGIN
.
.
.
END;
...
```

FORTRAN'daki de:

```
DO 60 K = 1, 100, 1
.
.
.
60 CONTINUE
...
```

biçimiyle diğerlerine benzemektedir.

for döngülerini akış diyagramlarında şöyle gösteriyoruz:



`while` döngülerine gelince bunlar bir koşul doğru olduğu sürece yinelenen döngülerdir. Bu tür döngülerde koşul genellikle bir değişkene bağlı olarak verilir. İşte PASCAL'dan bir örnek:

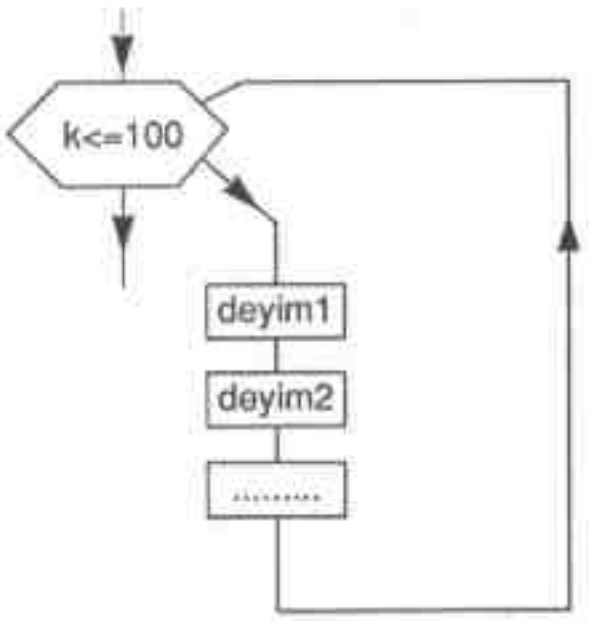
```
WHILE K <= 100 DO
BEGIN
.
.
.
END;
```

DBASE ya da uyumlu dillerde de yapı benzer biçimdedir:

```
DO WHILE K <= 100
.
.
.
ENDDO
...
```

Pascal Programcısına Not: Pacal'daki `repeat ... until` döngüsünü bir çeşit `while` döngüsü olarak ele alabiliriz. Bu döngünün yukarıdaki örneklerde ele alınan `while` döngülerinden iki farkı vardır: 1) Kontrol sondadır; 2) Koşul sağlanmadığı sürece yinelenmeye neden olur.

Akış diyagramlarında while döngülerini şu biçimde gösteriyoruz:



Bir döngü söz konusu olduğunda derleyici hangi deyimlerin döngüye dahil olduğunu belirlemek zorundadır. Programlama dillerinin bir bölümünde döngü sınırlarının belirlenmesi amacıyla çeşitli anahtar sözcüklerin kullanıldığını görüyoruz: BASIC'te NEXT, FORTRAN'da CONTINUE, DBASE ve diğer XBASE dillerinde ise ENDDO gibi.

C'deki döngüleri de `while` ve `for` döngüleri olmak üzere iki kısma ayırarak inceleyeceğiz.

14.2 while DÖNGÜSÜ

Diğer dillerde olduğu gibi C'de de `while` döngüleri koşul sağlandığı sürece yinelemeye neden olmaktadır. `while` döngülerini iki gruba ayırabiliriz:

1) Kontrolün başta yapıldığı `while` döngüleri

2) Kontrolün sonda yapıldığı `while` döngüleri `(do - while)`

Sırasıyla inceleyelim.

14.2.1 Kontrolün Başta Yapıldığı while Döngüleri

Genel biçimi aşağıdaki gibidir:

```
while (ifade)
    deyim
...
```

`while` bir anahtar sözcüktür. C derleyicileri `while` anahtar sözcüğünden sonra parantezler arasında bir ifade bekler. Döngü bu ifadenin Doğru (sıfır dışı bir değer) olduğu sürece yinelenir. Genel biçimde belirtilen **deyim**; yalın, bileşik ya da başka bir kontrol deyimi olabilir. Daha açık bir deyişle, `if` deyiminde olduğu gibi, eğer bir tek deyim söz konusuysa bloklama yapılmasına gerek yoktur, ancak birden fazla deyim blok içine alınmalıdır. Örneğin aşağıdaki `while` döngüsünde yalnızca `deyim1` döngü içerisindedir. Döngü çıkışındaki ilk deyim ise `deyim2`'dir.

```
while (ifade)
    deyim1
deyim2
...
```

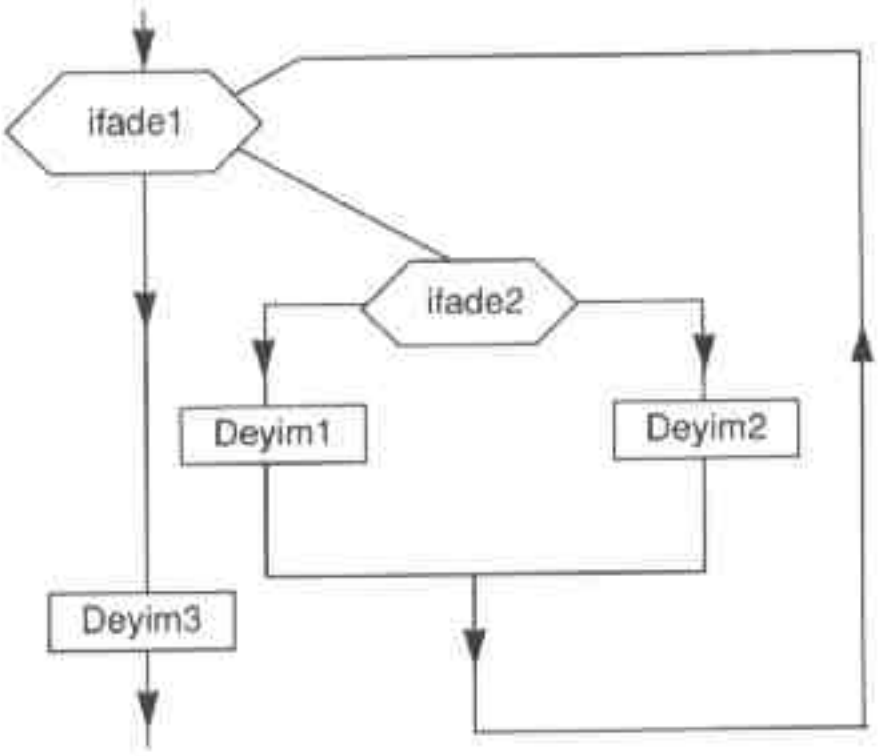
Aşağıdaki örnekte ise `deyim1` ve `deyim2`'nin her ikisi de döngü içerisindedir. `deyim3` döngünün dışındaki ilk deyimdir.

```
while (ifade) {
    deyim1
    deyim2
}
deyim3
...
```

`while` döngüsünün içerisindeki deyim, herhangi bir kontrol deyimi de olabilir.

```
while (ifade1)
    if (ifade2)
        deyim1
    else
        deyim2
deyim3
...
```

Bu örnekte bloklama yapılmadığı için `while` içerisinde yalnızca `if` deyimi vardır. `Deyim3`, döngü çıkışındaki ilk deyimdir. Olası bir karışıklığı engellemek için akış diyagramı ile gösterelim:



`while` döngüsü, "parantez içindeki ifadenin sayısal değeri sıfır dışı bir değer (Doğru) olduğu sürece yinelemeye neden olur" demiştik; örneklerle açıklayalım:

```
k = 1;
while (k <= 20) {

    printf("%d\n", k);
    ++k;
}
...
```

ifade doğru olduğu sürece (sıfır dışı bir değer) döngü devam eder.

Yukarıdaki `while` döngüsü `k` değişkeni 20'den küçük ya da eşit olduğu sürece yinelenecektir.

k	k <= 20	
1	1	Doğru, döngüye devam
2	1	Doğru, döngüye devam
3	1	Doğru, döngüye devam
...	...	
20	1	Doğru, döngüye devam
21	0	Yanlış, döngüden çık!..

k, döngünün içerisinde artırıldığına göre 20 yineleme sonra döngünün sonlanacağını söyleyebiliriz.

Aşağıdaki `while` döngüsünü inceleyelim:

```
while ((ch = getchar()) != 'q')
        printf("%c\n", ch);
...
```

Bu örnekte **while** döngüsünün devam edebilmesi için klavyeden girilen karakterin 'q' olmaması gerekir. Ya da şöyle söyleyebiliriz: Bu döngü 'q' karakterine basıldığında sonlanır!

Aşağıdaki örnekte neler oluyor dersiniz?

```
k = 20;
while (k) {
    printf("%d\n", k);
    --k;
}
...
```

Burada `while` parantezinin içerisindeki ifadede yalnızca k değişkeni vardır. k her yinelemede **1** eksilecek ve sonunda **0** olunca döngü de sonlanacaktır.

Bazı döngülerden çıkış mümkün olmayabilir:

```
while (1) {
    .
    .                   /* sonsuz döngü */
    .
}

while (-9) {
    .
    .                   /* sonsuz döngü */
    .
}
```

Yukarıdaki ilk iki örnekte `while` içerisindeki ifade hiçbir zaman 0 olamayacağı için döngülerden çıkış da mümkün değildir. Bu tür döngülere **sonsuz döngüler (infinite loops)** diyoruz. Aşağıdaki örnekte ise döngü hiç yinelenmez. Böyle döngülere de **boş döngü (null loop)** denilmektedir.

```
while (0) {
    .
    .                   /* boş döngü */
    .
}
```

Boş döngülerin bir işlevi olabilir mi?..

C'de sonsuz döngülerden **break** *anahtar sözcüğü ile çıkılabilir.* **(Bakınız: 14.4)**

Son olarak aşağıdaki örneği inceleyelim.

```
long k;
...
k = 0;
while (k++ != 100000L)
       ;
...
```

Burada bloklama yapılmadığına göre döngünün içerisinde yalnızca boş deyim vardır. Yani bu örnekte döngü, kendi içinde yinelenerek son bulur; böyle bir kod da olsa olsa belli bir gecikme sağlamak amacıyla yazılmış olabilir.

14.2.2 Kontrolün Sonda Yapıldığı while Döngüleri

Bu tür `while` döngülerinde kontrol sonda olduğu için döngü içindeki deyimler en az bir kere işlem görür. Genel biçimini aşağıda veriyoruz:

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

`do`, döngünün başını gösteren bir anahtar sözcüktür. Döngünün iç bölgesi `do` anahtar sözcüğünden `while` anahtar sözcüğüne kadar olan bölgedir. Diğerlerinde olduğu gibi, tek bir deyim için bloklamaya gerek yoktur; fakat birden fazla deyim için bloklama yapılmalıdır. Örneğin:

```
do
    deyim1
while (ifade);
deyim2
...
```

Burada yalnızca `deyim1` döngünün içerisindedir.

```
do {
    deyim1
    deyim2
} while (ifade);
deyim3
...
```

Bu örnekte ise bloklama yapıldığı için `deyim1` ve `deyim2`'nin her ikisi de döngünün içerisindedir. `Deyim3`, döngünün dışındaki ilk deyimdir. Aşağıdaki örneği inceleyiniz.

```
do {
    printf("(E)vet ya da (H)ayır\n");
    ch = getch();
} while ((ch = toupper(ch)) != 'E' && ch != 'H');
...
```

Bu örnekte kullanıcı **E, e, H, h** harflerinden birini girmeye zorlanmaktadır. Bu harflerden bir tanesi girilmediği sürece döngü yineleneceğinden çıkış da mümkün olmaz. Aynı kodu kontrol başta olacak biçimde tasarlasaydık fazladan bir deyim daha yazmak zorunda kalırdık!

```
ch = 0;
while ((ch = toupper(ch)) != 'E' && ch != 'H') {
    printf("(E)vet ya da (H)ayır\n");
    ch = getch();
}
...
```

14.3 for DÖNGÜLERİ

C'deki `for` döngüleri diğer dillerdekilerle kıyaslandığında çok daha geniş işlevlere sahiptir. `for` döngülerinin genel biçimi şöyledir:

```
for (ifade1; ifade2; ifade3)
    deyim
...
```

C derleyicileri `for` anahtar sözcüğünden sonra parantezler içerisinde iki tane sonlandırıcı (noktalı virgül) beklerler. Bu iki sonlandırıcı `for` döngüsünü `ifade1, ifade2` ve `ifade3` ile gösterdiğimiz 3 kısma ayırmaktadır. Şimdi `for` döngüsünü oluşturan bu 3 kısmın işlevlerini tek tek ele alıp açıklayalım:

ifade1: `for` döngüsünün birinci kısmı olan bu ifade, döngüye ilk girişte yalnızca bir kez yapılır. **ifade1** uygulamalarda genellikle döngü değişkenine ilk değer vermek amacıyla kullanılır.

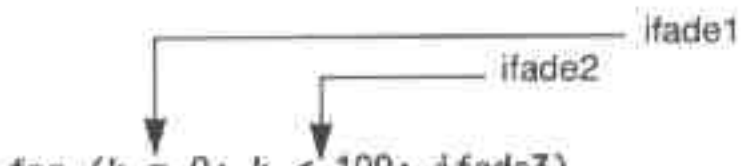
Örneğin:

```
          ifade1
          ↓
for (k = 0; ifade2; ifade3)
    deyim
```

ifade2: `for` döngüsünün ikinci kısmı olan `ifade2` döngüye ilk girişte ve sonraki her yinelemede işlem görür. **`for` döngüleri ifade2'nin sayısal değeri Doğru (sıfır dışı bir değer) olduğu sürece yinelemeye neden olurlar.** Uygulamada `ifade2`

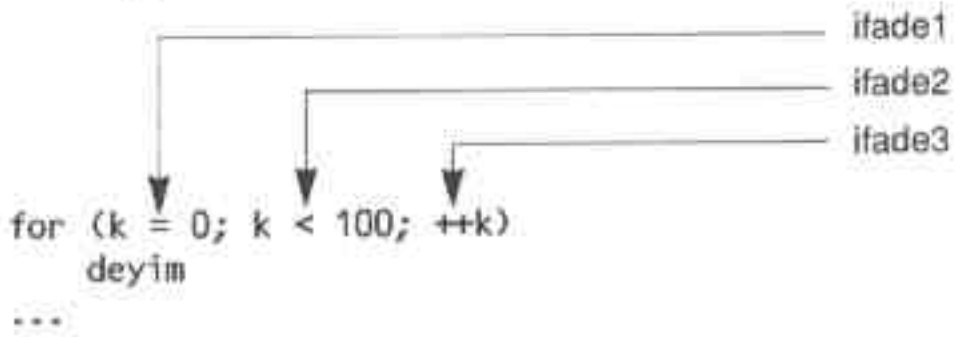
genellikle yinelenecek miktarı belirleyen ilişkisel bir operatörle kullanılmaktadır.

Örneğin:

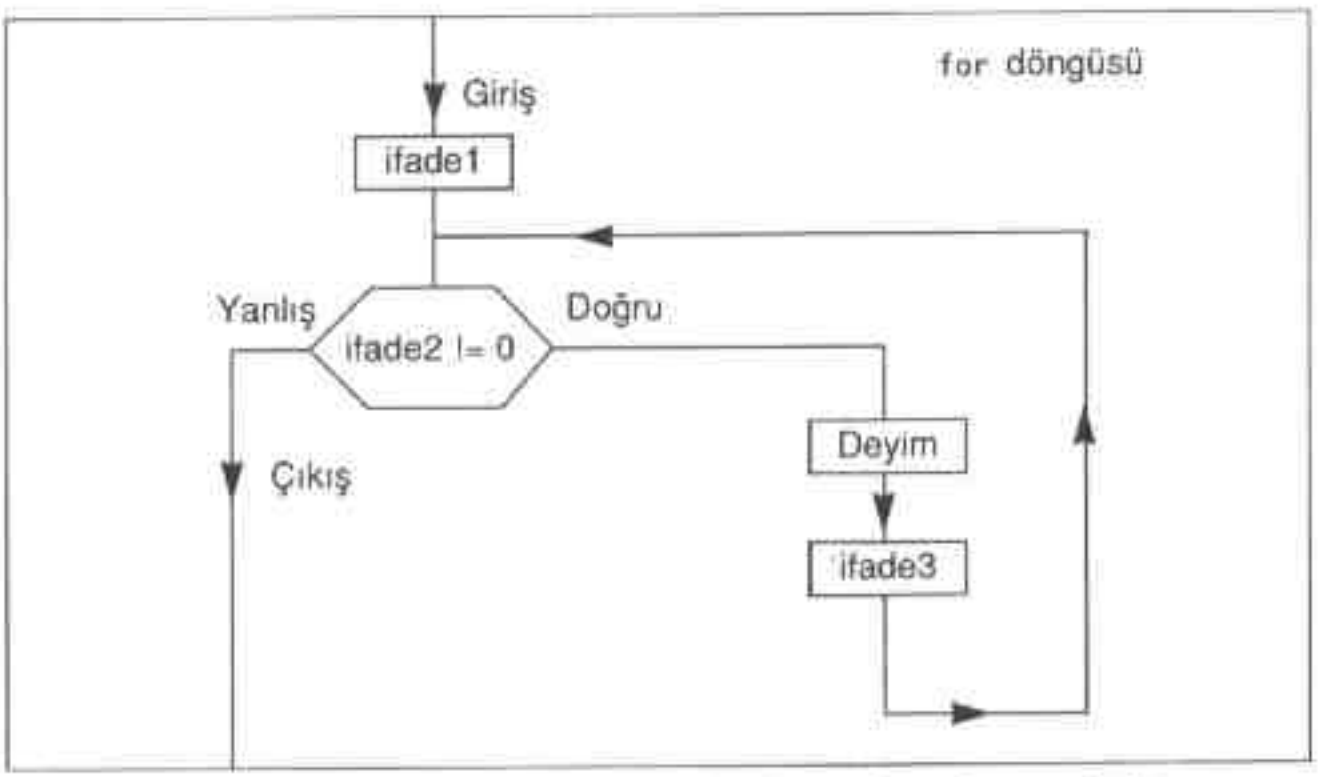


ifade3: Bu kısım her yinelemenin sonunda bir kez işlem görür. Uygulamada genellikle döngü değişkeninin artırılması amacıyla kullanılmaktadır.

Örneğin:



for döngüsünün çalışmasını akış diyagramıyla aşağıdaki gibi gösterebiliriz:



C'deki for döngülerini diğer dillerdeki for döngülerine benzetebiliriz:

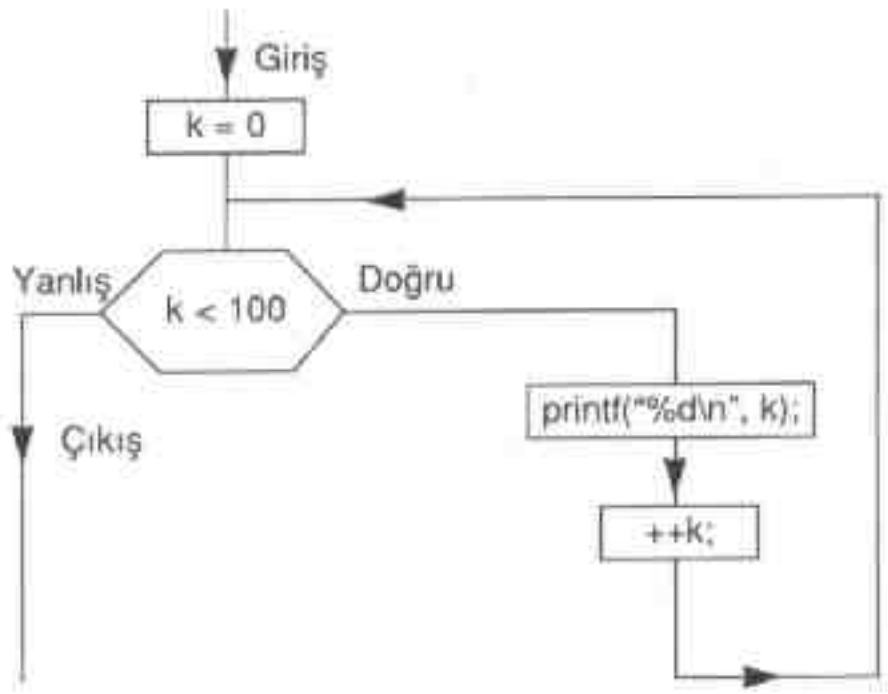
```
for (ilkdeğer; koşul; artım)
    deyim
...
```

`while` döngülerinde olduğu gibi `for` döngülerinde de döngü içerisindeki **deyim** yalın, bileşik ya da başka bir kontrol deyimi olabilir. Yani tek bir deyim için bloklamaya gerek yoktur; ancak birden fazla deyim döngü içerisine alınacaksa bloklama yapılmalıdır.

Aşağıda 0'dan 99'a kadar sayıları ekranda gösteren bir örnek verilmiştir, inceleyiniz:

```
#include <stdio.h>

void main(void)
{
    int k;

    for (k = 0; k < 100; ++k)
       printf("%d\n", k);
}
```

döngünün çalışmasını akış diyagramıyla gösterirsek:



Yukarıdaki örnekte **k değişkeni** döngü çıkışında hangi değeri alıyor dersiniz? Yanıt 100, değil mi? Çünkü `k` artarak en son **100** değerine gelecek ve `k < 100` ifadesi `0` değerini alarak döngü sonlanacak. Bunu aşağıdaki gibi bir kodla da test edebilirsiniz.

```
...
for (k = 0; k < 100; ++k)
    ;                          /* Döngünün içinde boş deyim var */
printf("%d\n", k);             /* k = 100 */
...
```

for döngüsünün herhangi bir ya da birden fazla kısmı olmayabilir; ancak parantezler içinde iki sonlandırıcının mutlaka bulunması gerekir. Aşağıdaki örneği inceleyiniz:

```
k = 0;
for (; k < 100; ++k)
    printf("%d\n", k);
...
```

Burada for döngüsünün birinci kısmı yoktur. Birinci kısım, döngüye girişte yalnızca bir kez işlem gördüğüne göre döngünün başında ayrı bir deyim olarak da konulabilir. Şimdi aşağıdaki örneği inceleyiniz:

```
k = 0;
for (; k < 100; ) {
    printf("%d\n", k);
    ++k;
}
...
```

Burada ise for döngüsünün bir ve üçüncü kısımları yoktur. Üçüncü kısım her yinelemenin sonunda bir kez işlem gördüğüne göre döngü içerisine son deyim olarak yazılabilir. Gerçekten de bu örnek **for** döngüsünün çalışmasını çok iyi bir biçimde yansıtmaktadır.

Nihayet üç kısmı da olmayan bir for döngüsü de mümkündür.

```
for(;;) {
    ...                          /* sonsuz döngü */
}
```

bu biçimdeki for döngüleri sonsuz döngü sağlamak amacıyla kullanılırlar.

Bir ve üçüncü kısımları olmayan for döngülerinin while döngüleri ile eşdeğer olduğuna dikkat ediniz:

```
for (;ifade;) {
    ...
}

while (ifade) {
    ...
}
```

O halde C'deki for döngülerini while döngülerinin gelişmiş bir biçimi olarak da ele alabiliriz...

for döngülerinin 3 kısmı da **virgül operatörü** ile genişletilebilir. Örneğin:

```
for (i = 0, j = 200; i < 100; ++i, j = j - 2) {
    ...
}
```

Bu örnekte döngünün birinci kısmını:

```
i = 0, j = 200
```

ifadesi oluşturmaktadır. İkinci kısımdaki koşul yalnızca i değişkenine bağlıdır. Üçüncü kısımda ise i bir artarken j iki eksilmektedir:

```
++i, j = j - 2
```

Aşağıdaki örneği inceleyiniz:

```
for (i = 1, toplam = 0;  i <= 100; toplam += i, ++i)
    ;
printf("toplam = %d\n", toplam);
...
```

Burada 1'den 100 kadar sayılar for döngüsü içinde toplanmıştır. for döngüsünün üçüncü kısmına dikkat ediniz!

```
toplam += i, ++i
```

ifadesi yerine,

```
++i, toplam += i
```

yazabilir miydik?..

for döngüleri genellikle belli bir sayıda yineleme sağlamak amacıyla kullanılsa da aslında çok daha geniş bir biçimde ele alınabilir. Sizi, diğer programlama dillerinin zararlı koşullamalarından kurtaracağız!

for içindeki ifadeler istenildiği gibi seçilebilir. Örneğin bir sayaç söz konusuysa kullanılan değişken tamsayı olmak zounda değildir:

```
float f;
...
for (f = 0; f < 3.14; f = f + 0.01) {
    ...
}
```

for döngülerinin içinde bulunan üç ifadenin aşağıdaki gibi bir ilişki içinde olması zorunlu değildir.

```
for (ilkdeğer; koşul; artım) {
    ...
}
```

İfade tanımına uyan herşey for döngüsünün parçalarını oluşturabilir:

```
for (ch = getchar(); toupper(ch) != 'E'; ch = getchar()) {
    ...
}
```

Yukarıdaki örnekte klavyeden girilen karakter 'e' ya da 'E' olmadığı sürece döngü devam ediyor. Aynı döngüyü şöyle de yazabilirdik:

```
ch = getchar();
for (; toupper(ch) != 'E'; ) {
    ...
    ch = getchar();
}
```

Şimdi de aşağıdaki döngüyü inceleyelim:

```
for (printf("ifade1\n1"), k = 0; printf("ifade2\n\n"), k < 5;
    printf("ifade3\n"), ++k)
    ;
```

Programın akışı döngüye girer girmez ekrana:

```
ifade1
ifade2
```

yazıları çıkar; ve bunu sonra 5 tane

```
ifade3
ifade2
```

yazısı izler. Virgül operatörünün sağdaki operandının değerini üretiğini anımsatarak yorumunu size bırakıyoruz...

14.4 break ve continue ANAHTAR SÖZCÜKLERİ

Döngülerin işleyişinde etkili olan iki anahtar sözcük vardır: break ve continue.

break anahtar sözcüğü döngüleri sonlandırarak program akışını döngünün dışındaki ilk deyime atlatır.

Örneğin:

```
for (;;) {
    ...
    ch = getchar();
    if (ch == 'q')
        break;
}
...
```

Burada klavyeden girilen karakter 'q' olduğunda break anahtar sözcüğü ile döngü kırılmaktadır. for (;;) sonsuz bir döngü olduğundan çıkış da ancak

break ile mümkün olabilir!

```
...
while (k < 100) {
    ...
    if (sample() < 0)
       break;
    ...
}
...
```

Bu örnekte ise **sample** fonksiyonunun geri dönüş değeri sıfırdan küçük ise döngü kırılmaktadır. Aşağıdaki örnekte de 0'dan 100'e kadar olan tamsayılar ekrana yazdırılıyor.

```
k = 0;
for (;;) {
    if (k >= 100)
       break;
    printf("%d\n");
    ++k;
}
```

Bu program parçasının;

```
for (k = 1; k < 100; ++k)
    printf("%d\n", k);
```

ile eşdeğer olduğunu dikkat ediniz.

İç içe döngülerin söz konusu olduğuna durumlarda **break** yalnızca içteki döngüyü kırabilir. Örneğin:

```
for (k = 0; k < MAX - 1; ++k) {
    ...
    for (l = 0; l < MAX - 1; ++l) {
        ...
        if (flag == 1) {
            flag = 0;
            break;
        }
        ...
    }
    ...
}
```

Not: **break** anahtar sözcüğü **while** ve **for** döngülerinin yanında **switch** deyiminden de çıkışı sağlamaktadır. **switch** ve **break** arasındaki ilişki 16. Bölümde ele alınmaktadır.

`continue` anahtar sözcüğü o anda içinde bulunulan yinelemeyi keserek bir sonraki yinelemeye geçilmesine neden olur. Aşağıdaki örneği inceleyiniz:

```
for (k = 0; k < MAX; ++k) {
    if (k % NUM == 0)
        continue;

    /*k NUM!a tam bölünmüyorsa işlem yap! */
}
...
```

`k, NUM`a tam bölünüyorsa işlem yapma!

Bu örnekte `k` bir tamsayı olduğuna göre:

```
k % NUM == 0
```

koşulu sağlanıyorsa `k, NUM` sayısına tam bölünebiliyor demektir. Bu durumda `continue` ile döngünün devam etmesi engellenerek sonraki yinelemeye geçilmiştir.

`for` döngüsü içerisinde `continue` kullanıldığında `for` döngüsünün üçüncü kısmı işleme sokulduktan sonra yineleme yapılır.

Dolayısıyla yukarıdaki örnekte `continue` anahtar sözcüğünden sonra `k`, bir artırılarak döngüye devam edilir. Aşağıdaki örneği inceleyiniz:

```
do {
    ch = getchar();
    if (isupper(ch))    /* Büyük harf ise işlem yapma! */
        continue;
    ...                 /* Büyük harf değilse işlemler... */
} while (ch != 'q');    /* Girilen karakter 'q' ise döngüden çık */
```

İlerleyen bölümlerde verdiğimiz örneklerde `break` ve `continue` anahtar sözcüklerinin daha çarpıcı örnekleriyle karşılaşacaksınız.

14.5 BİRAZ DA UYGULAMA...

Bu bölümde `while` ve `for` döngülerine ilişkin -şu ana kadar gelmiş olduğumuz durumu da gözönüne alarak- birkaç örnek vereceğiz. Bu örnekleri mutlaka bilgisayarınızda yazarak denemelisiniz. Konular ilerledikçe zaten doğal olarak döngülerin çok çeşitli örnekleriyle karşılaşacaksınız.

1) Aşağıdaki program *ASCII* karakter tablosunu ekrana dökmektedir.

```
#include <stdio.h>

void main(void)
{
    int k;

    for (k = 0; k < 256; ++k)
```

```
        printf("%c ", k);
}
```

2) Parametresi olan tamsayının faktöriyel değeri ile geri dönen `factorial` isimli fonksiyonu inceleyeniz:

```
#include <stdio.h>

long factorial (int n)
{
    int k;
    long fact = 1;

    for (k = 2; k <= n; ++k)
       fact *= k;
    return fact;
}

void main()
{
    printf("%ld\n", factorial(6));    /* 720 */
}
```

3) Aşağıda, parametresi olan uzun tamsayının basamak sayısına geri dönen "basamak" isimli fonksiyon tasarlanmıştır.

```
#include <stdio.h>

int basamak(int n)
{
    int k = 0;

    do {
       ++k;
       n /= 10;
    } while (n);
    return k;
}

void main()
{
    long x = 1536812L;

    printf("%d\n", basamak(x));       /* 7 */
}
```

4) Aşağıdaki program verilen bir sayıyı ikilik sistemde ekrana yazar.

```
#include <stdio.h>

void main()
{
    int x = 100;
    int k, bit;

    for (k = 15; k >= 0; --k) {
        bit = (x >> k) & 1;
        printf("%d", bit);
        if (k % 4 == 0)
            putchar('\n');
    }
}
```

Bu örnekte sayı ötelendikten sonra en düşük anlamlı biti maskelenmektedir.

SORAMADIKLARINIZ...

S1) for döngüsünün üçüncü kısmında yalın biçimde bulunan artırma ve eksiltme operatörlerinin önek ya da sonek durumunda olması arasında bir fark var mıdır? Örneğin:

```
for (k = 0; k < 100; ++k) {
    ...
}
```

ile

```
for (k = 0; k < 100; k++) {
    ...
}
```

birbirinin eşdeğeri midir?

C1) for döngülerindeki yalın artırım ifadeleri arasında bir farklılık yoktur; dolayısıyla yukarıdaki iki döngü birbirinin eşdeğeridir. Artırma ya da eksiltme operatörü yalın olarak kullanıldığında (başka hiçbir operatör olmaksızın) önek ve sonek durumları arasında hiçbir ayrım olmadığını anımsayınız. Yukarıdaki döngünün aşağıdaki eşdeğer biçimi açıklayıcı olacaktır.

```
k = 0;
for (; k < 100; ) {
    ...
    ++k; ya da k++;
}
```

ÖNİŞLEMCİ KAVRAMI VE SEMBOLİK SABİTLER

15

Bu bölümde önişlemci konusuna bir giriş yapılmaktadır. Önişlemci komutlarından `#include` ayrıntılı bir biçimde ele alınırken, `#define` yalnızca sembolik sabitlerle sınırlı tutulmuştur. `#define` komutunun makro biçimindeki karmaşık kullanımları ve diğer önişlemci komutları 30. Bölümde ele alınmaktadır.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Önişlemcinin görevi nedir ve hangi aşamada faaliyet gösterir?
2) `#include` önişlemci komutu nerede ve nasıl kullanılır?
3) `#define` önişlemci komutunun işlevleri nelerdir?
4) Sembolik sabitler nasıl belirtilirler ve ne amaçla kullanılırlar?

15.1 ÖNİŞLEMCİ KAVRAMI (Preprocessor)

Giriş bölümünde kaynak programların hangi aşamalardan geçerek çalışabilir kod biçimine dönüştürüldüklerini açıklamıştık; bir kez daha anımsatmak istiyoruz.

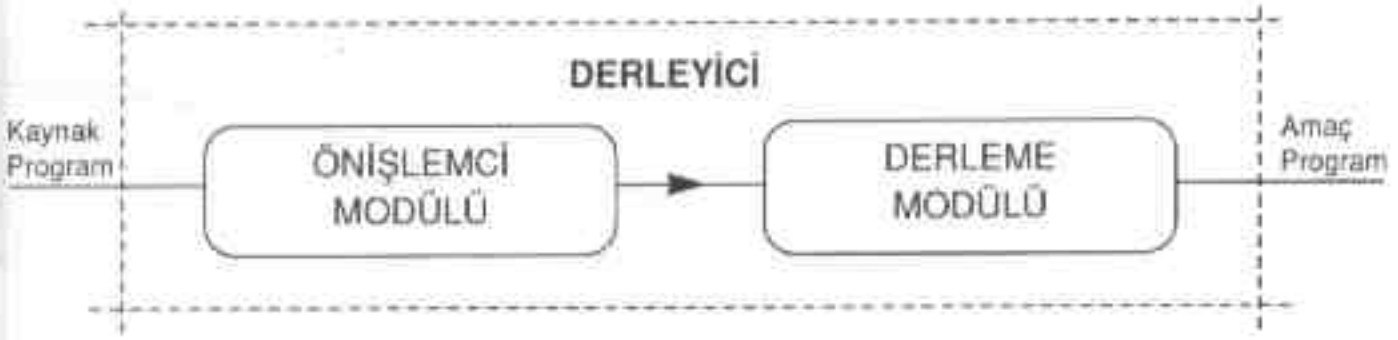
Buna göre:

1) Kaynak program bir editörde yazılır (`.C`)
2) Derlenerek amaç kod haline getirilir (`.OBJ`)
3) Bağlayıcı program ile çalışabilir koda dönüştürülür (`.EXE`)

Şimdiye kadar tek bir program olarak ele aldığımız C derleyicisi aslında iki ayrı modülden oluşmaktadır:

1) Önişlemci modülü
2) Derleme modülü

Aşağıdaki şekli inceleyiniz:



Derleme işleminin bütün fonksiyonları derleme modülü tarafından yerine getirilmektedir. Zaten derleyici deyince esas olarak aklımıza gelen modül de budur. **Önişlemci, kaynak program üzerinde birtakım düzenlemeler ve değişiklikler yapan bir ön modülüdür.** Önişlemcinin çıktısı derleme modülünün girdisini oluşturmaktadır. Yani kaynak program ilk aşamada önişlemci tarafından ele alınır; önişlemci kaynak programda çeşitli düzenlemeler ve değişiklikler yapar; daha sonra düzenlenmiş ve değiştirilmiş olan bu kaynak program derleme modülü ile amaç koda dönüştürülür.

C'de bütün # ile başlayan ifadeler önişlemciye aittir. # karakterinin sağında bulunan anahtar sözcükler (bunlara önişlemci komutları diyoruz) önişlemciye ne yapması gerektiğini anlatır.

Örneğin:

```
#include
#define
#if
#ifdef
#ifndef
...
```

Önişlemcinin amaç koda dönüştürmeye yönelik hiçbir işlem yapmadığını vurgulayalım. Önişlemci yalnızca kaynak programı # içeren satırlardan arındırmaktadır.

Bu bölümde `#include` komutu ile, `#define` komutunun yalın biçimde kullanımı ile oluşturulan sembolik sabitleri göreceğiz. Diğer önişlemci komutları ileride ayrı bir bölüm içinde ele alınmaktadır.

15.2 #include

`#include` , ilgili kaynak dosyanın derleme işlemine dahil edileceğini anlatan bir önişlemci komutudur. Bu komut ile önişlemci belirtilen dosyayı diskten okuyarak komutun yazılı olduğu yere yerleştirir.

`#include` komutundaki dosya ismi iki biçimde belirtilebilir:

1) Açısal parantezler ile (< >)

Örneğin:

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

2) İki tırnak içerisinde.

Örneğin:

```
#include "stdio.h"
```

Açısal parantezle belirtilen dosya, yalnızca önişlemci tarafından belirlenen dizin içerisinde aranır. Çalıştığımız derleyiciye ve sistemin kurulumuna bağlı olarak bu dizin farklı olabilir. Örneğin:

```
\TC\INCLUDE
\BORLANDC\INCLUDE
\C600\INCLUDE
...
```

gibi. Benzer biçimde *UNIX* sistemleri için bu dizin, örneğin:

```
/USR/INCLUDE
...
```

biçiminde olabilir. Genellikle **standart başlık dosyaları** -önişlemci tarafından belirlenen dizinde olduğundan- açısal parantezler ile kaynak koda dahil edilirler.

Dosya ismi iki tırnak ile yazıldığında önişlemci ilgili dosyayı, önce geçerli dizinde (current directory); burada bulamazsa, bu kez de sistem ile belirlenen dizinde arayacaktır. Örneğin, `C:\SAMPLE` dizininde çalışıyor olalım:

```
#include "key.h"
```

ile önişlemci `KEY.H` isimli dosyayı önce `C:\SAMPLE` dizininde; eğer burada bulamazsa bu kez sistem ile belirlenen dizinde arar. Programcıların oluşturdukları başlık dosyaları -genellikle sisteme ait dizinde olmadıkları için- iki tırnak ile kaynak koda dahil edilirler.

`include` ile belirlenen dosya ismi **yol (path)** içerebilir. Örneğin:

```
#include <sys\stat.h>
#include "c:\sample\key.h"
...
```

gibi.

Bu durumda açısal parantez ya da iki tırnak gösterimleri arasında bir fark yoktur. Her iki gösterimde de önişlemci, ilgili dosyayı yalnızca yolun belirttiği dizinde arar.

C'ye yeni başlayanların çoğunda olduğu gibi sizler de, `#include` komutunun yalnızca başlık dosyaları için kullanıldığı sanısına kapılmış olabilirsiniz. Oysa böyle bir zorunluluk yok! Herhangi bir kaynak dosya bu yöntemle derlemeye dahil edilebilir.

ANA.C

```
/* Derlenecek Dosya */

#include <stdio.h>
#include "topla.c"

main()
{
   printf("%d\n", topla(10, 20));

}
```

TOPLA.C

```
int topla (int a, int b)
{
   return a+b;
}
```

Yukarıdaki örnekte önişlemci `ANA.C` dosyası içerisine `TOPLA.C` dosyasını dahil etmiştir. (Bu örnekte bir tek modülün söz konusu olduğunu belirtelim. Çünkü `TOPLA.C`, modül dosyalarında olduğu gibi bağımsız olarak derlenip bağlama aşamasında birleştirilmemektedir.)

`#include` komutu kaynak programın herhangi bir yerinde olabilir. Fakat, standart başlık dosyaları gibi, içerisinde çeşitli bildirimlerin bulunuduğu dosyalar için en iyi yer kuşkusuz programın en tepesidir.

`#include` komutu **iç içe geçmiş (nested)** bir biçimde de bulunabilir. Örneğin çok sayıda dosyayı kaynak koda dahil etmek için şöyle bir yöntem izleyebilirsiniz:

ANA.C

```
#include "project.h"

main()
{
   ...
}
```

PROJECT.H

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stype.h>
#include "ctalib.h"
#include "tour.h"
```

`ANA.C` dosyası içerisine yalnızca `PROJECT.H` dahil edilmiştir; önişlemci bu dosyayı kaynak koda dahil ettikten sonra yoluna bu dosyadan devam edecektir.

15.3 #define KOMUTU VE SEMBOLİK SABİTLER

#define en sık kullanılan önişlemci komutudur. Yalın sembolik sabitlerden karmaşık makrolara kadar geniş bir kullanım alanı vardır. Ancak biz bu bölümde **#define** komutunun yalnızca yalın kullanımı üzerinde duracağız.

Önişlemci **#define** komutu ile karşılaşınca şunları yapar:

1) **define** anahtar sözcüğünden sonra gelen boşluk karakterlerini atarak ilk boşluksuz karakter kümesini ayırır. Buna **STR1** diyelim.

2) İlk boşluksuz karakter kümesi olan **STR1**'den sonra satır sonuna kadar olan bütün karakterleri alır. Buna da **STR2** diyelim.

3) Kaynak program içerisinde **STR1** karakterlerini gördüğü yere **STR2** karakterlerini yerleştirir.

Bu durumda **#define** metin editörlerinde olduğu gibi bir yazının başka bir yazı ile değiştirilmesi işlemini yapmaktadır.

```
#include<stdio.h>
                    STR1
                    STR2
#define MAX 100

                              100
void main()
{
    int k;

    for (k = 0; k < MAX; ++k)
        printf("%d\n", k);
}
```

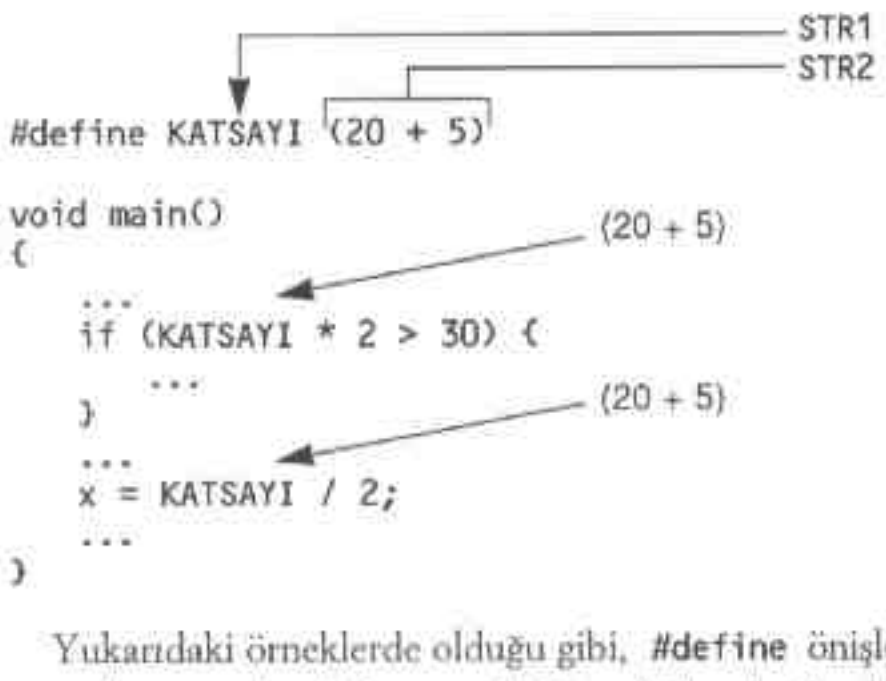
Yukarıdaki örnekte önişlemci kaynak programda **MAX** gördüğü yere 100 yerleştirir. Böylece, derleme modülü programı ele aldığında kaynak program içerisinde **MAX** değil 100 görecektir. Aşağıdaki örneği inceleyiniz:

```
                    STR1
                    STR2
#define KATSAYI 20 + 5

void main()                    20 + 5
{
    ...
    if (KATSAYI * 2 > 30) {
        ...                    20 + 5
    }
    ...
    x = KATSAYI / 2;
    ...
}
```

#define komutu ile önişlemci hesaplama yapmaz; yalnızca yer değiştirme işlemi yapar! Bu nedenle yukarıdaki örnekte önişlemci, KATSAYI gördüğü yere 25 değil, 20 + 5 yazacak; dolayısıyla,

```
if (20 + 5 * 2 > 30) {
    ...
}
```

deyimi de Yanlış olarak ele alınacaktır. Çarpmanın toplamaya olan önceliğinin yarattığı bu olumsuzluğu ancak öncelik operatörü kullanarak giderebilirsiniz:



Yukarıdaki örneklerde olduğu gibi, #define önişlemci komutuyla tanımlanmış olan sayısal sabitlere sembolik sabitler denilmektedir. Sembolik sabitlerin nesne olmadığına dikkat ediniz!

```
#define MAX 100

main()
{
    ...
    MAX = 200;          /* Hata, MAX nesne değil! */
    ...
}
```

Yukarıdaki örnekte önişlemci MAX yerine 100 yazacağına göre, derleme modülü kaynak programı ele aldığında:

```
100 = 200;
```

ifadesiyle karşılaşacaktır. Derleyiciler böyle durumları hata mesajı ile programcıya bildirirler. Hata mesajlarının içeriği ise derleyiciden derleyiciye değişebilir. Örneğin, *BORLAND* derleyicileri bu durumlarda atama operatörüne ait bir sol taraf değerinin olmadığına işaret ederler:

```
Lvalue required in function ...
```

#define komutunun kullanımı yalnızca sayısal sembolik sabitlerle sınırlı değildir. Birkaç basit örnek vermek istiyoruz:

```
#define MESAJ  "Merhaba"

main()
{
    ...
    printf(MESAJ);
    ...
}
```

Bu örnekte önişlemci MESAJ gördüğü yerlere "Merhaba" yazacaktır. Dolayısıyla ekrana **Merhaba** yazısının çıkması beklenir.

Önişlemci iki tırnak içerisindeki ifadeleri değiştirmeye çalışmaz. Bu nedenle aşağıdaki örnekte ekrana **Merhaba** değil, MESAJ yazısı çıkar.

```
#define MESAJ  Merhaba

main()
{
    ...
    printf("MESAJ");
    ...
}
```

#define komutu, önişlemciye ait satırlar üzerinde de değişiklikler yapabilmektedir. Örneğin, #include komutunun gereksinim duyduğu dosya ismi #define ile bildirilmiş olabilir.

```
#define FILE_NAME      "C:\SOURCE\SAMPLE"
...
#include FILE_NAME
```

Bir sembolik sabit kendisinden daha önce tanımlanmış başka bir sembolik sabite de bağlı olabilir. Aşağıdaki örneği inceleyiniz:

```
#define MAX 100                    100
#define MIN (MAX - 75)

void main()
{                                  (100 - 75)
    ...
    x = MIN
    ...
}
```

Aşağıdaki örneğin ilginizi çekeceğini düşünüyoruz:

```
#define ana             main
#define tam             int
#define cevrim          for
#define yazf            printf

#define MAX 100

void ana()
{
    tam k;

    cevrim (k = 0; k < MAX; ++k)
        yazf("%d\n", k);
}
```

Bu örnekte önişlemci anahtar sözcüklerin *Türkçe* karşılıkları yerine asıllarını yerleştirir.

Sembolik sabitler programın herhangi bir yerinde bildirilebilirler. Globallığı ya da yerelliği söz konusu değildir. Ancak bildirildikleri yerden dosyanın sonuna kadar olan bölge içinde etki gösterirler.

15.3.1 Sembolik Sabitler Niçin Kullanılırlar?

Sembolik sabitler temel olarak iki amaçla kullanılmaktadır.

1) Okunabilirliği artırırlar. Yani kaynak programı inceleyen bir kişi daha kolay ve daha çabuk bilgi edinir. Örneğin, personel programınızda 479 kişi üzerinde çalışıyorsanız ve bu sayı sıkça kullanılıyorsa sembolik sabit biçiminde belirtebilirsiniz. Böylece programınıza bakan bir kişi 479 sayısının ne amaçla kullanıldığını hemen anlayacaktır.

```
#define PERSONEL_SAYISI       479
...
if (adet > PERSONEL_SAYISI) {
    ....
}
```

Benzer biçimde örneğin, geri dönüş değeri yalnızca Doğru - Yanlış bilgisi olan fonskiyonlar sembolik sabitlerle daha okunabilir hale getirilebilirler.

```
#define FALSE  0
#define TRUE   1
...

int islower(int ch)
{
```

```
    if (ch >= 'a' && ch <= 'z')
        return TRUE;
    return FALSE;
}
```

Standart başlık dosyalarında da çeşitli sembolik sabitlerin tanımlanmış olduğunu görmekteyiz. Örneğin `stdio.h` dosyası içerisinde:

```
#define NULL 0
```

tanımlaması yapılmıştır. `NULL` sembolik sabiti pekçok durumda `0` sayısından daha okunabilirdir.

2) Sembolik sabitler parametrik programların yazılabilmesine de olanak sağlarlar. Pekçok yerde kullanılan bir sabitin başka bir değerle değiştirilmesi durumunda, değişikliğin yalnızca sembolik sabitlerde yapılması yeterli olmaktadır.

```
#define MAX    100
...

if (MAX > 200) {
    ...
}

for (k = 0; k < MAX; ++k) {
    ...
}
```

Yukarıdaki örnekte yalnızca `MAX` sembolik sabitini değiştirerek bütün `MAX` ifadelerini de değiştirmiş oluyoruz. Gerçekte olan şudur: Değişikliği biz yapacağımıza bunu önişlemciye yaptırıyoruz!

SORAMADIKLARINIZ...

S1) `#include` komutunda belirtilen dosyanın arandığı "varsayılan dizin" nasıl belirlenir?

C1) *DOS* altında çalışan derleyicilerin tümleşik çevreli uyarlamalarında menüler aracılığıyla bu dizini belirleyebilirsiniz. Yine komut satırlı uyarlamalarında çeşitli seçenekler ile `include` dizini belirtilmektedir.

S2) `#define` ile yapılan yer değiştirme işleminde büyük harf - küçük harf duyarlılığı (case sensitivity) var mıdır? Örneğin:

```
#define MAX 100
```

ifadesi

```
a = max * 2;
```

ifadesindeki `max` sözcüğü üzerinde etkili olur mu?

C2) `#define` işleminin büyük harf küçük harf duyarlılığı vardır. Büyük harflerle yazılmış `MAX` sembolik sabiti ancak aynı biçimde büyük harf yazılmışlar üzerinde etkili olur. Şunu anımsatmakta da yarar görüyoruz:

```
#define MAX    100
```

ile bildirilmiş bir sembolik sabit:

```
a = MAXIMUM + 2;
```

ifadesindeki `MAXIMUM` sözcüğünün `MAX` hecesini değiştirmez. `#define` işleminin değiştireceği sembolik sabitler atomlarla ayrılmış olmalıdır.

S3) Yukarıdaki örneklerde sembolik sabitler hep büyük harflerle yazıldılar; yoksa sembolik sabitlerin büyük harflerle yazılmaları biçiminde bir zorunluluk mu var?

C3) Hayır sembolik sabitlerin büyük harflerle yazılmaları biçiminde bir zorunluluk yok! Ancak sembolik sabitlerin kolaylıkla göze çarpması için genellikle büyük harflerle yazıldıklarını söyleyebiliriz. Sembolik sabitlerin büyük harflerle yazılması biçiminde bir gelenek yerleşmiştir.

S4) `NULL` sözcüğü bütün sistemlerde `stdio.h` dosyası içerisinde sembolik sabit biçiminde bildirilmiş midir, yoksa sistem bağımlı mıdır?

C4) `NULL` sembolik sabiti derleyicilerin çoğunda sayısal `0` biçiminde bildirilmiştir. Ancak bazı derleyicilerde gösterici `0` biçiminde de bildirilmiş olabilir. Dolayısıyla bu bildirim tam anlamıyla taşınabilir değildir.

switch DEYİMİ ve KOŞUL OPERATÖRÜ

16

Bu bölümde `if` deyimi ile çağrışım ilişkisi olan `switch` deyimi ve koşul operatörü ele alınmaktadır. `switch` deyimi birden çok seçenek arasında işlem yapmak için kullanılan bir kontrol deyimidir. Bölüm içerisinde her iki konu da ayrıntılarıyla incelenmektedir. Ayrıca bölüm sonunda **goto** deyimi hakkında da bilgiler bulacaksınız.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Sabit ifadeleri ile diğer ifadeler arasındaki farklılık nedir?
2) `switch` deyiminin çalışma biçimi nasıldır ve hangi durumlarda kullanılır?
3) Koşul operatörü nasıl çalışır ve hangi durumlarda kullanılır?
4) Koşul operatörünün if deyimi ile benzerlikleri ve farklılıkları nelerdir?
5) **goto** deyimi nasıl kullanılır; **goto** etiketlerinin faaliyet alanları nasıldır?

16.1 SABİT İFADELERİ (Constant Expression)

Sabitlerin operatörlerin ve değişkenlerin kombinezonlarına ifade denildiğini anımsayınız. Yalnızca operatör ve sabitlerden oluşan ifadelere sabit ifadeleri denir. Sabit ifadeleri değişken içermezler. Örneğin:

```
(10  + a) * 2
```

bir sabit ifadesi değildir. Ancak

```
10 * 2 - 7
```

bir sabit ifadesidir. Sabit ifadelerinin sayısal değeri derleme zamanında derleyici tarafından hesaplanabilir. Bu nedenle derleyici için bir sabit ile sabit ifadesi arasında fark yoktur. Oysa değişken içeren ifadelerin sayısal değerleri ancak programın çalışma zamanı sırasında kesin olarak belirlenebilir.

16.1.1 Sabit İfadelerinin Gerekli Olduğu Yerler

C'de sabit ifadesinin zorunlu olduğu durumlar vardır. Bu durum derleyicinin derleme aşamasında somut bir sayısal değer hesaplama zorunluluğundan ortaya çıkmaktadır. Sabit ifadelerinin gerekli olduğu yerler şunlardır:

1) Statik ömürlü nesnelere (global değişkenler ve `static` yerel değişkenler) ve dizi elemanlarına ancak sabit ifadeleriyle ilkdeğer verilebilir. Örneğin `x` ve `y` global olmak üzere:

```
int x = 100;
int y = x;
```

ikinci tanımlama işlemindeki verilen ilkdeğer sabit ifadesi olmadığı için hatalıdır. Benzer biçimde:

```
void fonk(void)
{
    int x = 100;
    static int y = x * 2;
    ...
}
```

yukarıdaki fonksiyonda tanımlanan `static y` değişkenine ilkdeğer sabit ifadesiyle verilmemiştir.

Yerel değişkenlere herhangi bir ifadeyle ilkdeğer verilebilir. Örneğin:

```
{
    int x = 100;
    int y = fonk(50) + x;
    ...
}
```

yukarıdaki ilkdeğer verme işlemleri geçerlidir. (Dizi elemanlarına ilkdeğer verme işlemi dizilerin anlatıldığı 17. Bölümde ele alınmaktadır.)

2) Tanımlama sırasında dizi uzunlukları sabit ifadeleriyle belirtilmelidir. (Bu durum dizilerin anlatıldığı 17. Bölümde ele alınmaktadır.)

3) `case` ifadeleri ve koşullu önişlemci komutları da sabit ifadeleri almak zorundadır. (case ifadeleri bu bölümde koşullu önişlemci komutları ise 30. Bölümde ele alınmaktadır.)

16.2 switch DEYİMİ

`if` deyimi Doğru ya da Yanlış olmak üzere iki seçeneğe sahiptir. Oysa `switch` deyimi belli bir ifadenin çeşitli sayısal değerlerine karşı farklı işlemlerin yapılması için kullanılmaktadır. Genel biçimi aşağıdaki gibidir:

```
switch (ifade) {
    case <sabit ifadesi1>: ..............[break];
    case <sabit ifadesi2>: ..............[break];
    case <sabit ifadesi3>: ..............[break];
    ....
    [default: ............;]
}
```

`switch`, `case` ve `default` birer anahtar sözcüktür. Derleyici `switch` parantezinin içerisindeki ifadenin sayısal değerini hesaplar. Eğer bu sayısal değere eşit olan bir `case` ifadesi bulursa programın akışını oraya yönlendirir. Eğer böyle bir ifade yoksa programın akışı `default` ile belirtilen kısma geçer. Örneğin:

```
switch(n) {
    case 1: printf("Bir\n");
    case 2: printf("İki\n");
    case 3: printf("üç\n");
    case 4: printf("Dört\n");
    case 5: printf("Beş\n");
    default: printf("Hiçbiri\n");
}
```

burada `n = 1` ise programın akışını `case 1:` kısmına yöneltilir. Akış buradan sırasıyla aşağı inerek devam edecektir. Dolayısıyla ekranda:

```
Bir
İki
Üç
Dört
Beş
Hiçbiri
```

yazılarını görürsünüz. Döngülerden çıkmak için kullanılan `break` anahtar sözcüğü `switch` içerisinden çıkmak için de kullanılabilir. Bu durumda bir `case` ifadesi `break` ile sonlandırılırsa programın akışı diğer `case` ifadelerine geçmez.

```
switch(n) {
    case 1: printf("Bir\n"); break;
    case 2: printf("İki\n"); break;
    case 3: printf("Üç\n"); break;
    case 4: printf("Dört\n"); break;
    case 5: printf("Beş\n"); break;
    default: printf("Hiçbiri\n"); break;
}
```

`case` anahtar sözcüğünün yanındaki ifadenin sabit ifadesi olması zorunludur. Çünkü derleyici derleme zamanında `case` ifadelerinin sayısal değerini hesaplamak zorundadır. Aşağıdaki örneği inceleyiniz:

```
switch (x - 1) {
    case 8   :..... break;
    case 12  :..... break;
    case y-3 :..... break;
    default  :.............;
}
```

Buradaki son **case** sabit ifadesi olmadığı için **switch** geçersizdir. Birden fazla sayısal değer için aynı işlemlerin yapılması isteniyorsa aşağıdaki gibi bir yol izleyebilirsiniz. Bunun daha kısa bir yapılış yöntemi yoktur.

```
switch (vmode) {
    case 2:
    case 7:
        vseg = 0xb000; break;
    case 3:
        vseg = 0xb800; break;
    default:
        printf("Lütfen video modu değiştiriniz!..\n");
}
```

Yukarıdaki örnekte **vmode** değişkeninin 2 ve 7 değerleri için aynı işlemler yapılmaktadır.

default anahtar sözcüğünün sonda olması ve **case** sabitlerinin artan sırada olması bir zorunluluk değildir. Yani yukarıdaki örnek şöyle de düzenlenebilirdi.

```
switch (vmode) {

    default:
        printf("Lütfen video modu değiştiriniz!..\n"); break;
    case 3:
        vseg = 0xb800; break;
    case 7:
    case 2:
        vseg = 0xb000;
}
```

switch deyiminin son **case** ifadesi için **break** gerekmediğine dikkat ediniz. Diğer kontrol deyimlerinde olduğu gibi **switch** deyimi de yalnızca bir deyim içeriyorsa bloklama yapmaya gerek yoktur. Örneğin:

```
switch(n)
    case 11:
    case 13: x = n - 1;
z = n / 2;
```

burada **switch** içerisinde yalnızca `x = n - 1;` deyimi vardır. `z = n / 2;` dışındaki ilk deyimdir.

Aşağıda örneği inceleyiniz:

```
int yilingunu(int gun, int ay)
{
    int ygun = gun;

    switch (ay - 1) {
```

```
        case 11: ygun += 30;
        case 10: ygun += 31;
        case  9: ygun += 30;
        case  8: ygun += 31;
        case  7: ygun += 31;
        case  6: ygun += 30;
        case  5: ygun += 31;
        case  4: ygun += 30;
        case  3: ygun += 31;
        case  2: ygun += 28;
        case  1: ygun += 31;
        }

    return ygun;
}

void main(void)
{
    printf("%d\n", yilingunu(31, 12));

}
```

yilingunu fonksiyonu ay ve gün ile belirlenen yıl içerisindeki bir tarihin yılın kaçıncı günü olduğunu buluyor.(Şubat ayının 28 çektiği varsayılmıştır.) `switch` içerisinde kümülatif toplam yapabilmek için hiç `break` kullanılmadığına dikkat ediniz.

16.3 KOŞUL OPERATÖRÜ

Koşul operatörü C'nin 3 operand alan ve karşılaştırma yapan özgün bir operatörüdür. Operandları sıradan birer ifade olabilir. Genel kullanım biçimi aşağıdaki gibidir:

```
ifade1 ? ifade2 : ifade3
```

Koşul operatörü `?` ve `:` olmak üzere ayrık iki sembolden oluşur. Derleyici önce `?` karakterinin solundaki birinci ifadenin sayısal değerini hesaplar. Genel biçimde bunu `ifade1` olarak isimlendirdik. Eğer `ifade1` sıfır dışı bir sayısal değere sahipse bu durum koşul operatörü tarafından Doğru olarak ele alınır ve yalnızca `ifade2` yapılır. Eğer `ifade1`'in sayısal değeri sıfır ise bu kez yalnızca `ifade3` yapılacaktır. Diğer operatörlerde olduğu gibi koşul operatöründen de bir değer üretilir. Bu değer koşulun sağlanması durumuna göre `ifade2` ya da `ifade3` olabilir. Örneğin:

```
y = x > 3 ? a + 1 : a - 1;
```

Burada önce `x > 3` ifadesinin sayısal değeri hesaplanır. Bu ifade sıfır dışı bir değerse (yani doğruysa) `y` değişkenine `a + 1` sıfır ise (yani yanlışsa) `a - 1` atanacaktır. Aynı işlem `if` ile de yapılabilirdi:

```
if (x > 3)
    y = a + 1;
else
    y = a - 1;
```

Koşul operatörü C'nin düşük öncelikli operatörlerindendir. Öncelik tablosunun alt kısmında atama operatörünün hemen yukarısında bulunur.

```
......
? :
= , +=, /=, *=, /=, %=, &=, ...
,
```

Yukarıdaki örnekte işlemlerin yapılış sırası şöyledir:

```
İ1: x > 3 ? a + 1 : a - 1    (x > 3 doğruysa a + 1, yanlışsa a - 1)
İ2: y = İ1
```

Koşul operatörü üç kısmı ile birlikte tek bir operatör olarak ele alınır. Yukarıdaki örnekte `? :` ve `=` olmak üzere toplam iki operatör vardır. Koşul operatörü ancak kendisinden daha düşük öncelikli operatörlerle ayrıştırılabilir. Örneğin aşağıdaki ifade hatalıdır.

```
x = a > b ? a : y = b;
```

Bu ifadede:

```
İ1: a > b ? a : y
İ2: İ1 = b     /* Hatalı, çünkü sol taraf değeri yok */
İ3: x = İ2
```

Aşağıdaki ifadede * koşul operatörüne ait değildir.

```
x = ((a > b) ? 10 : 20) * 2;
```

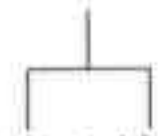
```
İ1: (a > b) ? 10 : 20
İ2: İ1 * 2
İ3: x = İ2
```

Koşul operatörü birçok durumda okunabilirliği kuvvetlendirmek için tercih edilir. Örneğin:

```
z = (a > b) ? a : b;
```

z değişkenine `max(a, b)` atanmaktadır. Programcılar algılamayı kolaylaştırdığı için koşul operatörürünün ilk operandını parantez içerisine alırlar. Ancak böyle bir zorunluluk yoktur.

Parantez gerekli değil



```
z = (a > b) ? a : b;
```

Koşul operatörünün operandları olan ifadeler ele alınırken kendi içlerindeki öncelik sıraları geçerlidir. Örneğin:

```
z = (a * a  > b) ? a / 2 : b;
```

Burada:

```
İ1: (a * a > b) ? a / 2 : b
İ1₁: a * a
İ1₂: İ1₁ > b
İ1₃: İ1₂ doğruysa a / 2 yanlışsa b
İ2: z = İ1
```

Koşul operatörünü `if` deyiminin başka bir biçimi gibi ele almamak gerekir. Koşul operatörü her şeyden önce bir operatördür ve bir değer üretir. Zaten bu operatör üretilen değerin bir yere aktarılması amacıyla kullanılmaktadır. Koşul operatörü programlarda genellikle üç biçimde karşımıza çıkar.

1) `return` ifadesi ile geri dönüş değerini oluştururken

```
return (x % 3) ? 0 : 1;
```

gibi. Fonksiyon x'in 3'e tam bölünmesine göre 1 ya 0 değeriyle geri dönmektedir.

2) Atama operatörüyle birlikte:

```
z = (a < b) ? a : b;          /* min (a, b) */
```

a ve b'nin küçüğü z değişkenine atanıyor.

3) Fonksiyon parametresi olarak.

Aşağıdaki örneği inceleyiniz:

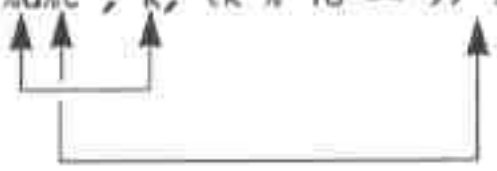
```
fonk(a > b ? a : b);
```

Burada **fonk** isimli fonksiyon tek parametreyle çağrılmıştır. Fonksiyona kopyalanacak parametre duruma göre **a** ya da **b** olabilir. Şimdi de aşağıdaki örneği inceleyiniz:

```
#include <stdio.h>

void main(void)
{
    int k;

    for (k = 0; k < 50; ++k)
        printf("%d%c", k, (k % 10 == 9) ? '\n' :'');
}
```

Ekrana her satıra onar onar sayılar yazılıyor. **printf** fonksiyonunun parametrelerini yakından inceleyelim:

```
printf("%d%c", k, (k % 10 == 9) ? '\n' :'');
```



%d formatına karşılık gelen parametre **k**'dır. (**%c** formatına karşılık gelen karakter ise **(k % 10 == 9)** ifadesinin sonucuna göre '\n' ya da '' olmaktadır. Her 10 sayıda bir ifade doğru olacağına göre imleç aşağıdaki satıra geçecektir.

Koşul operatörünün ikinci ve üçüncü operandı aynı türe dönüştürülür. Örneğin:

```
double x;
int y;

z = (a == b) ? x : y;
```

burada **x** **double** **y** **int** türündendir. Ancak iki ifade arasında büyük türe dönüştürülme yapılacağından ifade yanlış bile olsa üretilen değerin türü **double**'dır.

16.4 goto DEYİMİ

Yapısal programlama dillerinde programın akışının başka bölgelere gönderilmesi algılamayı ve tasarımı zorlaştırdığı için pek salık edilmez. **goto** anahtar sözcüğü program akışını istenilen bir bölgeye yönlendirmek için kullanılır. **goto** yanına bir etkiket ismi verilerek kullanılmalıdır.

goto <etiket ismi>;

Etiket, değişken isimlendirme kuralına uygun herhangi bir isim olabilir. Programın akışı etiket ile belirtilen bölgeye gider.

goto REPEAT;

...

REPEAT:

goto etiketinin faaliyet alanı fonksiyon faaliyet alanıdır. Yani **goto** ile başka bir fonksiyonun bir bölgesine atlama yapılamaz. Aşağıdaki örnekte klavyeden alınan karakter **q** olduğunda içiçe iki döngüden çıkılmıştır.

```
#include <stdio.h>

void main(void)
{
    int i, j;
    char ch;

    for (i = 0; i < 10; ++i)
        for (j = 0; j < 10; ++j) {
            ch = getch();
            if (ch == 'q')
                goto EXIT;
            printf("i = %d j = %d\n", i, j);
        }
    EXIT:
    printf("program sonlandırılıyor!\n");
}
```

SORAMADIKLARINIZ...

S1) Statik ömürlü değişkenlere sabit ifadeleriyle ilkdeğer verilmesi neden zorunludur?

C1) Statik ömürlü değişkenlerin aldıkları ilkdeğerler **object modül** içerisine yazılırlar. Bu nedenle bu ilkdeğerlerin derleme aşamasında belirlenmesi gerekir. Oysa değişken içeren ilkdeğerler derleme aşamasında belirlenemezler.

17 DİZİLER

Bu bölümde programlama dillerinin en önemli veri yapıları olan diziler ele alınmaktadır. Dizilerin bildirimleri, dizi elemanlarına erişim, dizilere ilkdeğer verilmesi gibi konular bölüm içerisinde ayrıntılı bir biçimde incelenmiştir. Bu bölüm aynı zamanda göstericiler konusuna da bir hazırlık niteliği taşımaktadır. Bölümün sonunda dizilerle ilgili birkaç yalın uygulama da bulacaksınız.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Adres nedir? Yazılımsal olarak hangi bileşenlerden oluşur?

3) **sizeof** operatörünün işlevi nedir ve nasıl kullanılır?

3) Dizi bildirimlerinin genel biçimi nasıldır?

4) Dizilerin bellekteki yerleşimleri nasıldır?

5) Dizi elemanlarına erişim nasıl sağlanmaktadır?

6) Dizi isimlerinin anlamı nedir?

7) Dizi elemanlarına nasıl ilkdeğer verilir?

8) Klavyeden karakter dizisi alan ve ekrana karakter dizisi yazan standart *C* fonksiyonları hangileridir ve nasıl kullanılırlar?

9) Karakter dizilerini bitiren sonlandırıcı karakter (NULL karakter) hangi amaçla kullanılır?

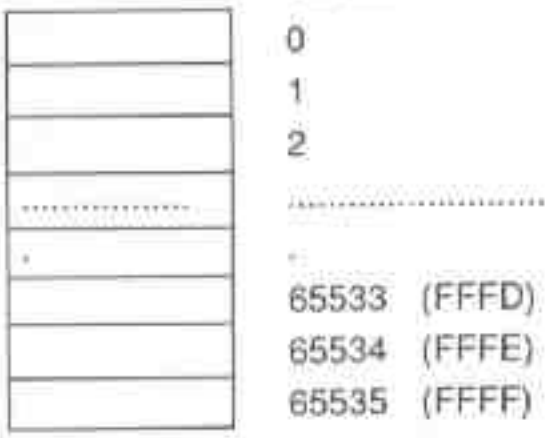
10) Sonlandırıcı karakter kim tarafından yerleştirilir ve kim tarafından kullanılır?

17.1 ADRES KAVRAMI

Adres hem donanıma hem de yazılıma ilişkin bir kavramdır. Donanımsal olarak bellekte yer gösteren bir sayıdan ibarettir. Mikroişlemci bellekte bir bölgeye ancak o bölgenin adres bilgisiyle erişebilir. Oysa yazılımsal olarak adres, yalnızca bellek bölgesinin yerini gösteren bir sayıdan ibaret değildir; aynı zamanda o bellek bölgesinde bulunan bilginin nasıl ele alınacağını belirten bir tür bilgisini de içermektedir.

Bellekteki her byte diğerlerinden farklı bir adresle temsil edilir. Sıfırdan başlayarak her byte'a artan sırada bir sayı karşılık getirerek elde edilen adresleme sistemine **doğrusal adresleme sistemi**, bu sistem kullanılarak elde edilen adreslere de **doğrusal adresler (linear address)** denilmektedir.

Örneğin, 64K'lık bir belleğin doğrusal olarak adreslenmesi aşağıdaki biçimde yapılır:



Bilgisayara ilişkin pekçok nicelikte olduğu gibi doğrusal adreslemede de 10'luk sistem yerine 16'lık sistemdeki gösterimler tercih edilir. Bundan böyle yalnızca **"adres"** dediğimizde **"doğrusal adres"** anlaşılmalıdır.

17.2 NESNELERİN ADRESLERİ

Her nesne bellekte yer kapladığına göre bir adrese sahiptir. Nesnelerin adresleri, -sistemlerin çoğunda- derleyici ve programı belleğe yükleyen işletim sistemi tarafından ortaklaşa olarak belirlenir. Nesnelerin adresleri program yüklenmeden önce kesin olarak bilinemez ve programcı tarafından da önceden tespit edilemez. Programcı, nesnelerin adreslerini ancak programın çalışması sırasında (run time) öğrenebilir.

Örneğin:

```
{
    char a;
    ...
}
```

biçiminde bir tanımlamayla karşılaşan derleyici bellekte a değişkeni için 1 byte yer ayıracaktır. Derleyicinin a değişkeni için nereyi ayıracağını önceden bilemeyiz. Bunu ancak programın çalışması sırasında öğrenebiliriz. Yukarıdaki örnekte

a'nın yerel bir değişken olduğuna da dikkat etmelisiniz. a değişkeni ilgili blok icra edilmeye başlandığında yaratılıp, bloğun icrası bittiğinde de yok olmaktadır. a değişkeninin 1FC4 adresinde olduğu varsayılarak aşağıdaki şekil çizilmiştir.

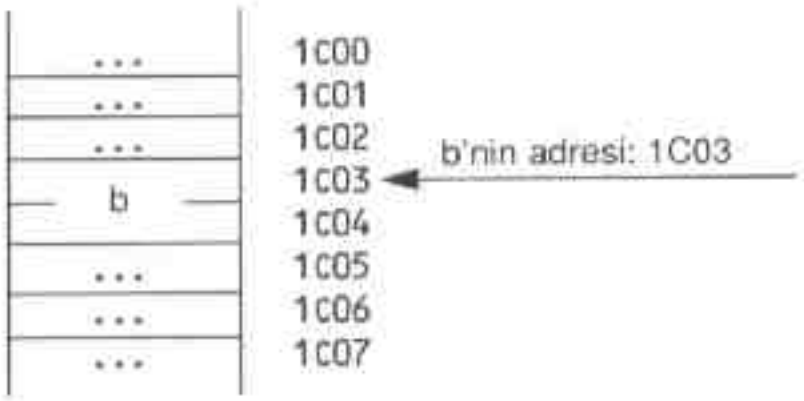
80X86 Sembolik Makina Dili Programcısına Not: Yerel değişkenler *STACK* bölgesinde saklanırlar. Derleyiciler yerel değişkenler için ilgili blok icra edilmeye başladığında:

```
SS:[BP-n]
```

adreslerini ayırırlar. Ancak SS değeri program yüklenince BP'nin değeri de bloğun icrası sırasında kesin olarak belirlenebilir. Global değişkenlerde de durum benzer biçimdedir. Derleyici ve bağlayıcı yalnızca global değişkenlerin offset adreslerini tam olarak belirleyebilirler. Oysa global değişkenlerin Segment adresleri belleğin o andaki durumuna bağlı olarak ve ancak yükleyici program tarafından kesin olarak belirlenebilir. Ayrıca *80386*, *80486* ve *Pentium* mikroişlemcilerinde doğrusal adres fiziksel adres ile aynı olmayabilir. Örneğin, eğer mikroişlemcinin **sayfalama (paging)** yapmasına izin verilmişse **sanal bellek (virtual memory)** söz konusu olduğundan doğrusal adres fiziksel adrese eşit olmaz.

Tanımlanan nesne 1 byte'tan daha uzunsa, o zaman nesnenin adresi nasıl belirleniyor dersiniz?

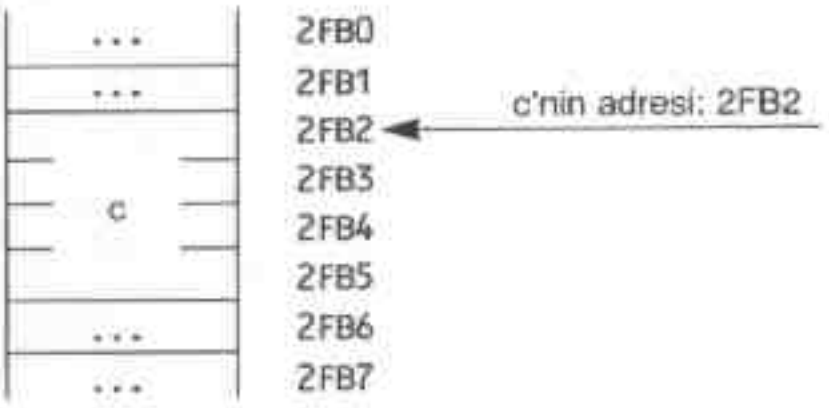
```
{
    int b;
    ...
}
```



1 byte'tan uzun olan nesnelerin adresleri, onların ilk byte'larının (düşük anlamlı byte'larının) adresleriyle belirtilir. Yukarıdaki örnekte b değişkeninin adresi 1C03'tür. Zaten b değişkeninin tamsayı türünden olduğu bilindiğine göre diğer parçasının 1C04 adresinde olacağı da açıktır.

```
{
    long c;
    ...
}
```

Benzer biçimde long türünden olan c değişkeninin bellekteki yerleşiminin aşağıdaki gibi olduğu varsayılırsa, adresinin 2FB2 olduğunu söyleyebiliriz.



Yerel ya da global olsun, ardışıl bir biçimde tanımlanmış nesnelerin bellekte de ardışıl bir biçimde tutulacağının bir garantisi yoktur. Örneğin:

```
{
    char a;
    char b;
    ...
}
```

buradaki b değişkeninin a değişkeninden 2 byte sonra bulunacağının bir garantisi yoktur.

80X86 Sembolik Makina Dili Programcısına Not: Yerel değişkenler için *STACK* şu biçimde ayarlanmaktadır (stack frame).

```
int sample()
{
    int a, b;
    ...
}
```

Sembolik makina dili karşılığı:

```
_sample proc near
push   bp
mov    bp, sp
sub    sp, 4    ;Yerel değişkenlerin toplam uzunluğu kadar SP geriye alınır.
...
mov    sp, bp
pop    bp
ret
_sample endp
```

Bu durumda yerel değişkenlerden birisi [bp-2] de diğeri ise [bp-4] de bulunmaktadır. Fakat ilk yazılanın mı (bu örnekte a), yoksa ikinci yazılanın mı (bu örnekte b) [bp-2] de olacağı C derleyicileri tarafından standart bir biçimde belirlenmemiştir. Örneğin, *BORLAND'ın, Turbo C 2.0* uyarlamasından sonra sistem değiştirdiği görülmektedir.

17.3 sizeof OPERATÖRÜ

`sizeof` bir nesnenin ya da veri türünün bellekte byte cinsinden kapladığı alanın belirlenmesini sağlayan özel amaçlı bir operatördür. İki biçimde kullanılır:

1) Nesne ismi ile:

```
sizeof(<nesne ismi>)
```

2) Tür belirten bir anahtar sözcükler ile:

```
sizeof(<tür>)
```

Nesne ya da tür isminin `sizeof` operatörünün parantezleri içerisine yazıldığına dikkat ediniz.

`sizeof` operatörü ilgili türün (ya da nesnenin) bellekte kaç byte yer kapladığına ilişkin bir tamsayı değeri üretir. Örneğin :

```
int a;
...
printf("%d\n", sizeof(a));
```

yukarıdaki ifadede `a` değişkeninin bellekte kaç byte yer kapladığı `printf` fonksiyonu ile ekrana yazdırılmaktadır. Dolayısıyla *DOS* altında çalışıyorsanız 2, *UNIX* altında çalışıyorsanın 4 sayısını ekranda görmeniz gerekir.

`sizeof` tür belirten anahtar sözcüklerle birlikte aşağıdaki gibi kullanılabilir:

```
sizeof(int)
sizeof(float)
sizeof(usigned short int)
sizeof(long)
...
```

Bir noktayı hatırlatmak istiyoruz: **sizeof fonksiyon değil bir operatördür;** parantezlere sahip olması sizi yanıltmasın! Eğer `sizeof` bir fonksiyon olsaydı, programın çalışma zamanı sırasında değerlendirilirdi. Oysa `sizeof` bir operatör olduğu için derleme sırasında ele alınmaktadır.

`sizeof` operatörünün ne amaçla kullanıldığını merak ediyorsunuzdur. Fakat `sizeof` operatörünün kullanım nedenini ancak göstericiler konusundan sonra net olarak anlayabilirsiniz. Şimdilik şu kadarını söyleyelim: `sizeof`, *C*'de tür uzunluklarının donanıma bağlı olmasının ortaya çıkarttığı sakıncaları ve taşınabilirlik problemlerini gidermek amacıyla kullanılmaktadır.

17.3.1 sizeof Operatörünün Önceliği

`sizeof` operatörü ++, —, !, (tür) ve ~ ile **sağdan sola** eşit öncelikli olan grupta bulunur. Şu ana kadar gördüğümüz operatörleri dikkate alarak öncelik tablosunun ilgili kısmını aşağıda veriyoruz:

()								soldan sağa
++	—	!	~	(tür)	sizeof			**sağdan sola**
*	/	%						soldan sağa

...

17.4 DİZİ KAVRAMI VE BİLDİRİMİ

Bellekte sürekli bir biçimde bulunan aynı türden nesnelerin oluşturduğu kümeye dizi diyoruz. Tanımdan da anlaşılacağı gibi diziyi dizi yapan iki temel özellik vardır.

1) Elemanlarının bellekte sürekli bir biçimde bulunması.
2) Elemanlarının aynı türden nesneler olması.

Dizi bildirim işleminin genel biçimi aşağıdaki gibidir:

```
<Tür>  <Dizi_İsmi> [<Eleman_sayısı>];
```

`<Tür>`	Dizi elemanlarının türlerini belirtmek için kullanılan bir anahtar sözcüktür.
`<Dizi_İsmi>`	Değişken isimlendirme kurallarına uyan herhangi bir isimdir.
`[<Eleman_Sayısı>]`	Dizinin kaç elemana sahip olacağını gösteren bir sabit ifadesidir. (Buradaki köşeli parantezler eleman sayısının isteğe bağlı olduğunu göstermiyor! Eleman sayısı köşeli parantezlerin içine yazılmalıdır.)

Örnek dizi bildirimleri:

```
char s[30];                 /* s, 30 elemanlı, her elemanı
                               karakter olan bir dizi */
float real[10];             /* real, 10 elemanlı, her elemanı
                               float olan bir dizi */
unsigned long  xlm[15];     /* xlm, 15 elemanlı, her elemanı
                               unsigned long olan bir dizi */
double max[100];            /* max, 100 elemanlı, her elemanı
                               double olan bir dizi */
  ...
```

Bildirim sırasında dizilerin eleman sayısı tamsayı türünden bir sabit ifadesiyle belirtilmek zorundadır. Örneğin:

```
int n = 100;
...
int x[n];
int y[10 * 2];
```

yukarıdaki ilk tanımlama dizi uzunluğunun değişken ile belirtilmesi nedeniyle geçersizdir. İkinci tanımlama sabit ifadesiyle yapıldığı için geçerlidir.

Dizilerin eleman sayısını sembolik sabitlerle de belirtebilirsiniz. Örneğin:

```
#define MAX    100
...
char s[MAX];
```

Diziler de diğer nesneler gibi yerel ve global faaliyet alanlarına sahip olabilirler. Dizilerin fonksiyonlara parametre olarak geçirilmeleri göstericileri ilgilendiren bir konu olduğundan, göstericilerin anlatıldığı bölümde ele alınacaktır.

17.5 DİZİ ELEMANLARINA ERİŞİM VE İNDEKS OPERATÖRÜ

Dizilerin her elemanı ayrı birer nesne gibi ele alınabilir. Dizi elemanlarına erişmek amacıyla **indeks operatörü** kullanılmaktadır. İndeks operatörü aslında bir gösterici (pointer) operatörüdür ve göstericiler konusunda ayrıntılı bir biçimde ele alınacaktır. Bu nedenle biz burada ayrıntıya girmeden, yalnızca bu operatörün nasıl kullanıldığını açıklayan temel bilgiler vereceğiz.

İndeks operatörü köşeli parantezlerle gösterilen **tek operandlı sonek** bir operatördür. Dizi ismi ve bir tamsayı ile aşağıdaki biçimde kullanılır:

```
<dizi ismi> [n]        Dizinin n. indisli terimi
```

İndeks operatörünün operandı dizi ismidir (aslında bir adrestir). Köşeli parantezler içerisinde, dizinin kaçıncı elemanına erişeceğimizi anlatan bir tamsayı sabiti ya da değişkeni olmalıdır. **Dizinin ilk elemanı sıfırıncı indisli elemanıdır.**

Örneğin:

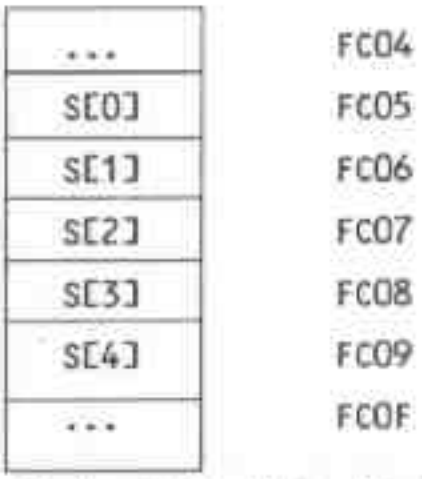
```
a[5]          ──►  a dizisinin 5. indisli elemanı
                   (6. sıradaki elemanı)
personel[0]   ──►  personel dizisinin 0. indisli elemanı
                   (İlk elemanı)
s[n]          ──►  s dizisinin n. indisli elemanı.
name[15]      ──►  name dizisinin 15. indisli elemanı
x[k]          ──►  x dizisinin k. indisli elemanı
...
```

Dizinin ilk elemanının sıfırıncı indisli eleman olduğunu bir kez daha vurgulamak istiyoruz. Dizi bildirimlerinde köşeli parantez içerisine yazdığımız sayılar dizinin kaç elemana sahip olduğunu göstermektedir; sizi yanıltmasın! Örneğin:

```
char s[5];
```

bildirimi, s dizisinin 5 elemanlı bir karakter dizisi olduğunu belirtir. Bu durumda

dizinin ilk elemanı `s[0]` olduğuna göre son elemanı da `s[4]` olmalıdır. Dizi elemanlarının bellekte sürekli bir biçimde bulunduğunu belirtmiştik:

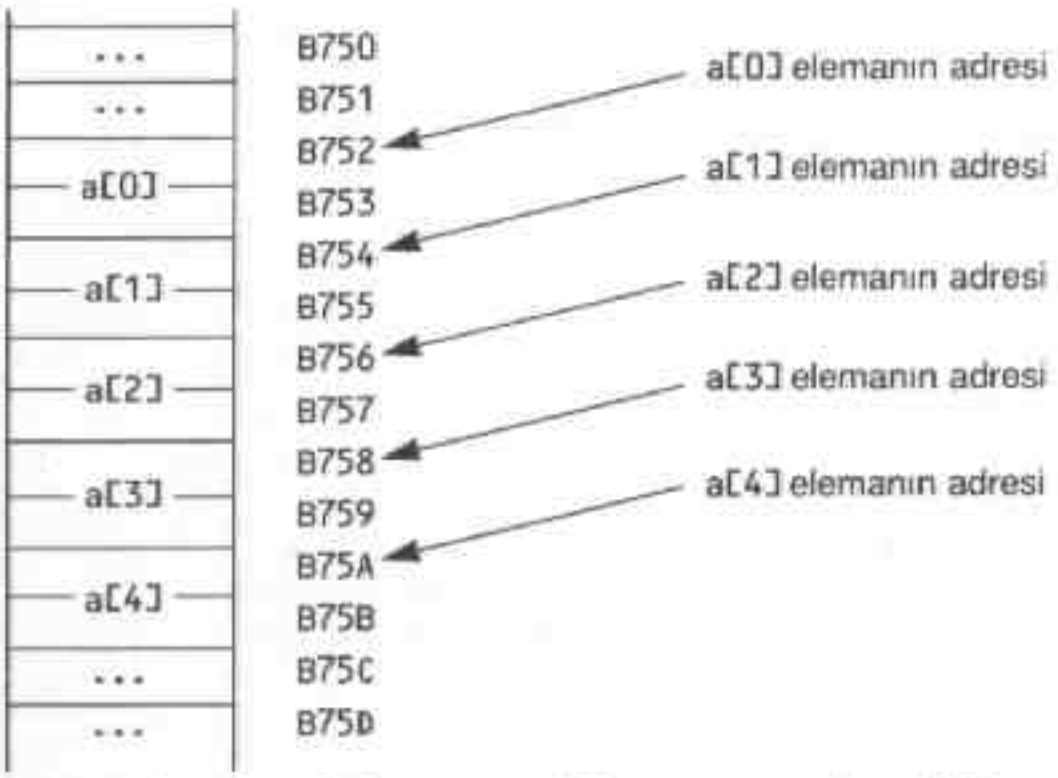


Yukarıdaki şekilde dizinin `FC05` adresinden başlayarak belleğe yerleştiği varsayılmıştır. Dizinin her elemanı karakter türünden olduğuna göre bellekte 1 byte yer kaplar. Benzer biçimde örneğin, bir tamsayı dizisinin her elemanının bellekte 2 (DOS ve Windows 3.1 için) byte yer kaplayacağını söyleyebiliriz.

Örneğin:

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

tanımlaması ile derleyici `a` dizisi için bellekte toplam 10 byte yer ayırır. `a` dizisinin bellekteki yerleşimi aşağıdaki gibi olacaktır:



Dizi elemanları da birer nesne olduğuna göre onların da birer adrese sahip olmaları gerekmez mi? Yukarıdaki örnekte, `a[0]` elemanının adresi: `B752`, `a[1]` elemanın adresi: `B754...` biçimindedir.

5 elemanlı `a` dizisinin `a[0], a[1], a[2], a[3], a[4]` elemanları kendi başlarına birer nesnedir, peki dizinin ismi olan `a` nedir?

17.6 DİZİ İSİMLERİ

Dizi isimleri nesne değildir; dizilerin bellekteki başlangıç adreslerini gösteren sembolik sabitlerdir. Örneğin:

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

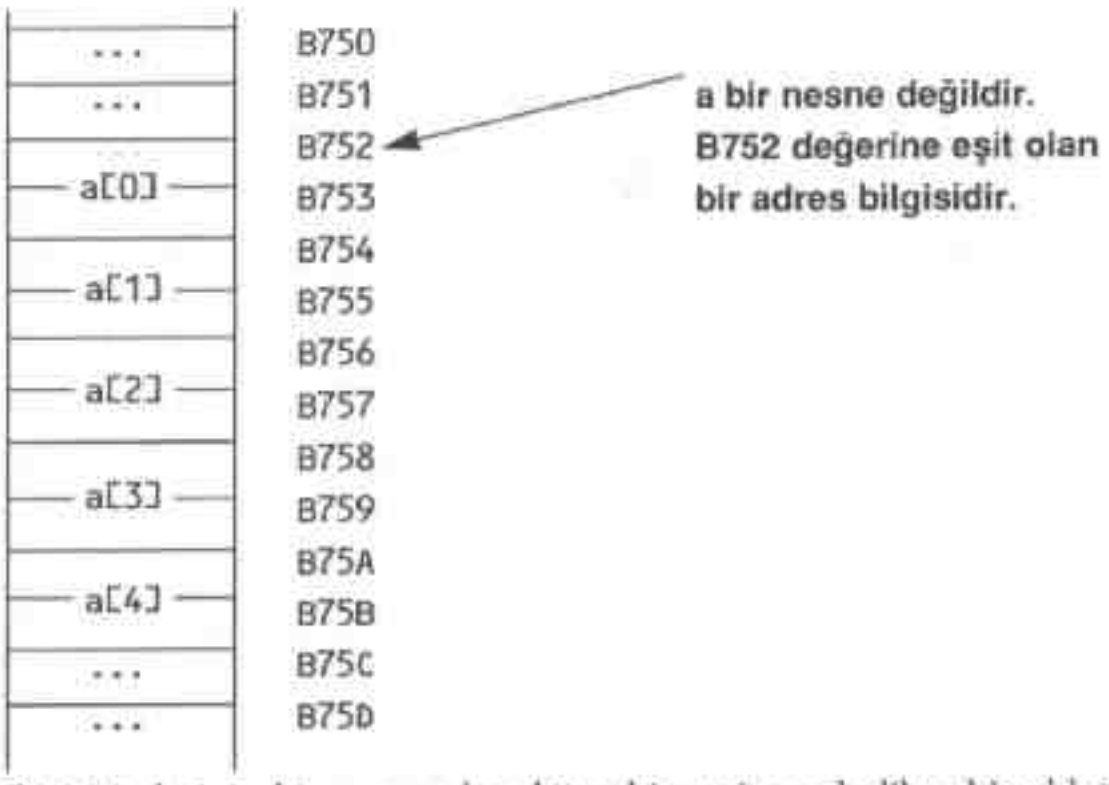
bildiriminde

```
a[0] = 100;
a[1] = 150;
...
```

yazabilirsiniz; ancak:

```
a = 500;                    /* Hata! dizi ismi nesne değil */
```

yazamazsınız. Çünkü **a** için bellekte bir yer ayrılmamıştır ve dolayısıyla bir nesne değildir.



Dizi isimlerinin bir nesne olmadığını bir çeşit sembolik sabit olduğunu bilmek çok önemli! Bunu daha nasıl vurgulayabiliriz?

Dizi ismi nesne değildir!

Dizi ismi nesne değildir!

Dizi ismi nesne değildir!

Dizi ismi nesne değildir!

Dizi ismi nesne değildir!

Adres değerlerinin yazdırılması için -*DOS* altında çalışıyorsanız- **printf** fonksiyonunu **"%p"** formatıyla kullanabilirsiniz. Örneğin bir dizi ismiyle belirtilen adresi yazdırmak istersek:

```
#include <stdio.h>
main()
{
    char s[20];
    ...
    printf("Dizinin başlangıç adresi: %p", s);
}
```

17.7 DİZİ ELEMANLARINA İLKDEĞER VERİLMESİ

Dizi elemanlarına ilkdeğer aşağıdaki biçimde verilmektedir:

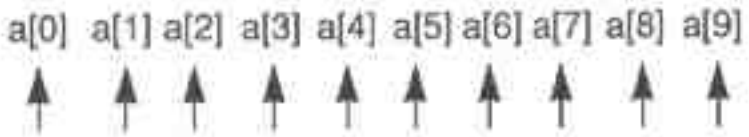
```
<tür> <dizi ismi> [[uzunluk]] = {si1, si2, si3, ...};
```

Örneğin:

```
int a[10] = {100, 450, 30, -45, 189, 74, 160, 600, 432, 529};
double kilo[5] = {79.2, 48.7, 68.3, 80.3, 53.6};
char s[10] = {'k', 'a', 'a', 'n', '\0'};
unsigned long para[5] = {110000, 450000, 78000, 960000, 11700000};
...
```

Küme parantezlerinin sonlandırıcı (noktalı virgül) ile bitirildiğine dikkat ediniz. Bu bölüme kadar ilk kez küme parantezinin sonlandırıcı ile bitirildiğine tanık oluyorsunuz.

Verdiğimiz bu ilkdeğerler derleyici tarafından dizi elemanlarına sırasıyla yerleştirilir. Bu durumda birinci örnekte verilen ilkdeğerlerin diziye yerleşimi aşağıdaki biçimde olacaktır:



int a[10]= {100, 450, 30, -45, 198, 74, 160, 600, 432, 529}

Dizinin tüm elemanlarına ilkdeğer verme gibi bir zorunluluk yoktur; daha az elemana ilkdeğer verilebilir. Örneğin:

```
int a[10] = {100, 50, 25, 80, 70};
```

burada a dizisi 10 elemana sahip olduğu halde ilk 5 elemanına ilkdeğer verilmiştir. Bu durumda ilk değer verilmemiş elemanlar, dizinin yerel ya da global olmasına bakılmaksızın 0 değerini alacaktır. Fakat dizinin uzunluğundan fazla elemana ilkdeğer verilmesi derleme zamanında hata (error) olarak değerlendirilir. Örneğin aşağıdaki dizi tanımlaması fazla sayıda elemana ilkdeğer verildiği için hatalıdır.

```
int a[5] = {100, 200, 400, 600, 150, 800}; /* Hata! */
```

Dizi elemanlarına ancak sabit ifadeleriyle ilkdeğer verilebilir. Örneğin aşağıdaki ilkdeğer verme işlemi geçersizdir:

```
int a[5] = {10, 20, x - 2,  30, 40 };    /* x - 2 sabit ifadesi
                                            değil!.. */
```

Bazı derleyicilerde yerel dizilere ilkdeğer verme işlemi de geçerli değildir. Bu durum özellikle çokişlemli sistemlerde çalışan (multiprocessing) eski derleyiciler için söz konusudur. Bu tür sistemlerde yalnızca global dizilere ilkdeğer verilebilir; aklınızda bulunsun!

17.7.1 Dizi Uzunluğu Belirtilmeden İlkdeğer Verme İşlemi

Dizi bildiriminde dizi uzunluğunun sabit ifadesiyle belirtilmek zorunda olduğunu söylemiştik. Ancak dizilere ilkdeğer verilirken dizi uzunluğu hiç yazılmayabilir. Örneğin:

```
int a[] = {100, 200, 350, -150, 800};
char s[] = {'i', 'l', 'k', 'd', 'e', 'ğ', 'e', 'r', '\0'};
float val[] = {10.3, 67.2, 78.9, 60.8, 45.3, 56.98, 80.0};
  ...
```

Derleyici bu biçimde bir ilkdeğerleme işlemiyle karşılaştığında dizinin kaç elemanına ilkdeğer verildiğini belirleyerek, dizinin o uzunlukta açıldığını varsayar. Yani aslında bu biçimdeki ilkdeğer vermelerde dizinin uzunluğu dolaylı olarak belirtilmektedir. Yukarıdaki örneklerde derleyici **a** dizisinin **5**, **s** dizisinin **9**, **val** dizisinin ise **7** eleman uzunluğunda açıldığını varsayacaktır.

Karakter dizilerine doğrudan iki tırnak içerisinde ilkdeğer verebiliriz. Örnegin:

```
char s[8] = "Merhaba";
char isim[] = "Kaan Aslan";
char kurum[] = "C ve Sistem Programcıları Derneği";
...
```

Bu durumda iki tırnak işaretlerinin içerisindeki karakterler derleyici tarafından sırasıyla ilgili diziye yerleştirilirler.

```
char isim[] = "Kaan Aslan"
```

örneğini ele alalım:

```
'K'          isim[0]
'a'          isim[1]
'a'          isim[2]
...
```

İki tırnak işaretleri arasında ilkdeğer verme işleminde derleyici dizinin sonuna sonlandırıcı karakter diye isimlendirilen (NULL karakter) bir karakter yerleştirir.

Karakter dizileri ve sonlandırıcı karakter hakkında ileride ayrıntılı bilgi verilmektedir.

Not: İki tırnak (stringler) ifadeleri 20. Bölümde, karakter dizileri ise bölüm içerisinde ileride ayrıntılı bir biçimde ele alınmaktadır.

17.8 DİZİ BİLDİRİMLERİNDE BELİRLEYİCİLERİN KULLANILMASI

Dizi bildirimlerinde yer ve tür belirleyicileri kullanılabilir. `static, volatile, register, auto` belirleyicileri dizi tanımlamalarıyla birlikte kullanıldığında dizinin her elemanı üzerinde etkili olurlar. Örneğin:

```
{
    static char s[10];
    ...
}
```

tanımlamasında `s` dizisinin 10 elemanı da `static` yerel bir değişkendir.

`extern` ve `const` belirleyicileri dizi bildirimleriyle beraber kullanıldıklarında -genel olarak adrese dayalı işlem yaptıklarından- dizi sınırının ötesinde bir etki alanına sahip olurlar. (Bu açıklamayı ancak göstericiler konusundan sonra anlamlandırabilirsiniz.)

17.9 KLAVYEDEN KARAKTER DİZİSİ ALAN VE EKRANA KARAKTER DİZİSİ YAZAN STANDART C FONKSİYONLARI

Daha önce klavyeden tek bir karakter alan ve ekrana tek bir karakter yazan fonksiyonları incelemiştik. Şimdi ise aynı işlemi birden çok karakter için yapan fonksiyonları inceleyeceğiz. Bu fonksiyonların nasıl çalıştıklarını ancak göstericiler konusundan sonra anlayabilirsiniz. Diğer giriş-çıkış fonksiyonlarında olduğu gibi aşağıdaki tüm fonksiyonların da prototipi `stdio.h` dosyası içindedir.

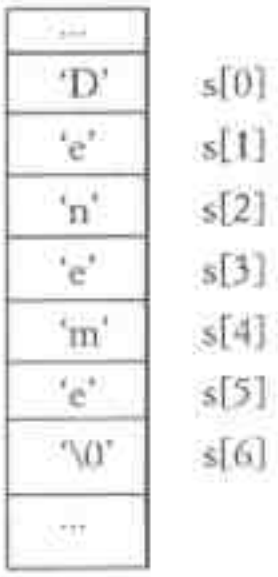
`gets` fonksiyonu klavyeden karakter dizisi almakta kullanılan standart bir C fonksiyonudur. Klavyeden girilen karakterlerin yerleşeceği dizinin ismini parametre olarak alır. Dizi isimleri aslında dizilerin bellekteki başlangıç adreslerini gösterdiğine göre şöyle de diyebiliriz: "`gets` fonksiyonu parametre olarak bir adres almaktadır". Örneğin:

```
char s[20];
...
gets(s);
```

ile klavyeden *ENTER* tuşuna basılana kadar girilmiş olan tüm karakterler `s` dizisi içerisine sırasıyla yerleştirilirler. Klavyeden,

```
Deneme
```

yazısı girilmiş olsun. Bu durumda **gets** fonksiyonu **Deneme** yazısını oluşturan karakterleri aşağıdaki gibi sırasıyla **s** dizisine yerleştirecektir.



17.9.1 gets

gets fonksiyonu klavyeden girilen karakterleri diziye yerleştirdikten sonra dizinin sonuna **sonlandırıcı karakter** diye isimlendirilen sıfır numaralı *ASCII* karakterini (`'\0'`) koyar.

Fonksiyonun çalışmasında vurgulanması gereken bir nokta var. **gets** klavyeden girilen karakterlerin hepsini diziye yerleştirmeye çalışır; dizi için hiçbir sınır kontrolü yapmaz. Eğer dizinin uzunluğundan fazla karakter girerseniz, dizi taşacağı için beklenmeyen sonuçlarla karşılaşabilirsiniz. Bu tür durumlarda ortaya çıkabilecek olumsuzluklar göstericiler konusunda **"gösterici hataları"** başlığı altında incelenmektedir. **gets** fonksiyonun `'\0'` karakterini dizinin sonuna eklediğini söylemiştik. Bu durumda eğer **n** elemanlı bir dizi açmışsanız, dizinin içerisine **gets** fonksiyonuyla en fazla **n-1** karakter girmelisiniz. Sonlandırıcı karakterin de bir yer kapladığını unutmayın! Aşağıda **gets** fonksiyonuyla ilgili bir taşma örneği veriyoruz:

```
char il[10];
...
gets(il);
```

Klavyeden girilen karakterler :

```
Kirklareli
```

gets fonksiyonu girilen karakteri diziye sırasıyla yerleştirecek.

`gets` fonksiyonunun eklediği sonlandırıcı karakter (`'\0'`) taşmaya neden olmuştur. Taşma durumlarında ortaya çıkabilecek problemlerin derleme zamanına değil de çalışma zamanına ilişkin olduğuna dikkat çekmek istiyoruz.

17.9.2 puts

`puts` bir karakter dizisinin içeriğini ekrana yazdırmak için kullanılır. İçeriği yazdırılacak olan karakter dizisinin ismini parametre olarak alır. `puts` karakter dizisini ekrana yazdıktan sonra imleci (cursor) sonraki satırın başına geçirmektedir.

```
char s[30];
...
gets(s);
puts(s);
```

Yukarıdaki örnekte `gets` ile klavyeden alınan karakterler tekrar `puts` ile geri yazdırılıyor.

Karakter dizilerini ekrana yazdırmak için `printf` fonksiyonunu `"%s"` formatıyla da kullanabiliriz.

`puts(s)` ile `printf("%s\n", s)` işlevsel olarak birbirleriyle eşdeğerdir.

17.10 BİRAZ DA UYGULAMA...

Bu bölümde dizilere ilişkin birkaç uygulama üzerinde durulmaktadır. Örneklerdeki ortak olan nokta bir indis yardımıyla dizi elemanlarının taranmasıdır. Bu örnekleri bilgisayarınızda yazarak çalıştırmalısınız.

1) Bir tamsayı dizisinde bulunan elemanların sayısal toplamını bulan program:

```
#include <stdio.h>

int dizi[10] = {100, -200, 400, -100, 550, 900, 500, 300, 450, 600};

void main()
{
    int toplam, k;

    for (k = 0, toplam = 0; k < 10; ++k)
        toplam += a[k];
    printf("toplam = %d\n", toplam);
}
```

2) Aşağıdaki program bir dizi içerisindeki en büyük sayıyı buluyor:

```
#include <stdio.h>

int a[10] = {100, -200, 400, -100, 550, 900, 500, 300, 450, 600};

void main()
{
    int max,k;

    max = a[0];
    for (k = 1; k < 10; ++k)
        if (a[k] > max)
            max = a[k];
    printf("En büyük eleman:%d\n", max);
}
```

En büyük sayıyı bulan algoritmanın oldukça kolay olduğuna söyleyebiliriz. Yukarıdaki programda:

```
max = a[0];
```

ile dizinin ilk elemanı en büyük varsayılarak `max` değişkenine atanmıştır. Daha büyüğüne rastlandıkça

```
max = a[k] ;
```

ile `max` değiştirilmektedir. Yukarıdaki programda biz yalnızca en büyük dizi elemanını buluyoruz. Peki, en büyük elemanın dizinin kaçıncı indisli elemanı olduğunu nasıl bulabiliriz? İnceleyiniz:

```
...
for (k = 1; k < 10; ++k)
    if (max > a[k]) {
        max = a[k];
        i = k;
    }
...
```

max değiştirildikçe o andaki dizi indisi de i değişkenine saklanıyor.

3) Bir dizi içerisindeki sayıları büyükten küçüğe doğru sıraya dizen program.

```
#include <stdio.h>

#define MAX 10

int a[MAX] = {49, 16, 22, 3, 45, 58, 6, 79, 8,- 9};

void main()
{
    int k, l max, indis;

    for (k = 0; k < MAX; ++k) {
        max = a[k];
        indis = k;
        for (l = k + 1; l < MAX; ++l)
            if (a[l] > max) {
                max = a[l];
                indis = l;
            }
         a[indis] = a[k];
         a[k] = max;
    }
    for (k = 0; k < MAX; ++k)
        printf("%d\n", a[k]);
}
```

Yukarıdaki örnekte olduğu gibi "en büyük elemanı bul başa koy" biçimindeki sıralamaya **seçerek sıralama (selection sort)** denilmektedir. Temel kavramların açıklandığı bölümde algoritmanın karmaşıklığı konusuna da değinilmişti (2.8). En kötü olasılıkla gereken karşılaştırma sayısı olarak tanımladığımız algoritmanın karmaşıklığı hız üzerinde etkili olan temel bir ölçüttür. Seçerek sıralama algoritmasının karmaşıklığı: `n * (n-1) / 2` dir. Üzerinde düşünmeyi size bırakıyoruz...

4) Yanyana elemanları karşılaştırarak yer değiştirmeye **kabarcık sıralaması (bubble sort)** denir. Aşağıdaki programı inceleyiniz:

```
#include <stdio.h>

#define MAX   10

int a[MAX] = {200, -60, 456, 700, 1200, 68, 90, 100, 4, 60};

main()
{
  int k, l, tampon;

  for (k = 0; k < MAX - 1; ++k)
      for (l = 0; l < MAX - 1; ++l)
          if (a[l] < a[l+1]) {
```

```
                tampon = a[l];
                a[l] = a[l+1];
                a[l+1] = tampon;
            }
    for (k = 0; k < MAX; ++k)
        printf("%d\n", a[k]);
}
```

İki döngünün de `MAX - 1` kadar yinelendiğine dikkat ediniz. Gerçekten de bu yöntemle sıraya dizme genel olarak, seçerek sıraya dizmeden daha uzun zaman almaktadır. Kabarcık sıralaması algoritmasının karmaşıklığı:

$(n - 1)^2$

biçimindedir; yorumlayınız. Kabarcık sıralaması küçük değişikliklerle algoritma karmaşıklığı düşürülerek daha verimli bir hale getirilebilir.

17.11 KARAKTER DİZİLERİ

Karakter dizileri en sık kullanılan dizilerdir. C'de yazı ve mesaj gibi alfabetik bilgiler karakter dizileri içerisinde saklanırlar. Karakterler üzerine işlemlerin hızlı ve etkin bir biçimde yapılması için **"sonlandırıcı karakter"** denilen bir tanımlamadan faydalanılmaktadır. Sonlandırıcı karakter kavramıyla daha önce `gets` fonksiyonunun tanıtılması sırasında karşılaşmıştınız.

17.10.1 Sonlandırıcı Karakter

Karakter dizilerini bitirdiği kabul edilen özel karaktere sonlandırıcı karakter diyoruz. Örneğin *ASCII* tablosunun sıfır numaralı karakteri (`'\x0'` ya da `'\0'`) *DOS* ve *UNIX*'te sonlandırıcı karakter olarak kullanılır. Sonlandırıcı karakter kavramını birkaç soru yardımıyla anlamaya çalışalım:

1) Sonlandırıcı karakter kaç numaralı karakterdir?

ASCII tablosunun sıfır numaralı (`'\0'`) karakteridir. Sıfır sayısı `stdio.h` içerisinde `NULL` sembolik sabiti olarak bildirildiği için sonlandırıcı karaktere `NULL` karakter de denir. Bunu sıfır karakterinin *ASCII* numarasıyla (`'0'`) karıştırmamalısınız.

2) Sonlandırıcı karakteri dizilerin sonuna kim ekliyor?

Karakter dizilerine tek tek ilk değer verilmesi durumunda sonlandırıcı karakter dizinin sonuna programcı tarafından eklenmelidir.

Aşağıdaki örneği inceleyiniz:

```
char s[7] = {'D', 'e', 'n', 'e', 'm', 'e', '\0'};
```

7 elemanlı s dizisine tek tek ilkdeğer verilmiştir. Bu örnekte sonlandırıcı karakteri yazının sonuna **programcının kendisi** koymuştur. Oysa karakter dizilerine iki tırnak ile ilkdeğer verildiğinde sonlandırıcı karakteri **derleyici** koymaktadır.

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

Sonlandırıcı karakterler yazıların sonuna **fonksiyonlar tarafından** da konulabilirler. Örneğin **gets** fonksiyonu klavyeden girilen karakterleri diziye yerleştirdikten sonra, sonlandırıcı karakteri kendisi eklemektedir.

```
char s[30];
...
gets(s);
```

gets fonksiyonunun sonlandırıcı karakteri eklediğini test etmek için aşağıdaki kodu deneyebilirsiniz.

```
#include <stdio.h>

main()
{
    char s[20];
    int k;
    gets(s);   /* Klavyeden karakter dizisi alınıyor */
    for (k = 0; k < 20; ++k)
        printf("%c : %d\n", s[k], s[k]);
}
```

Klavyeden girdiğiniz son karakterden bir sonra **gets** fonksiyonunun koyduğu sonlandırıcı karakteri göreceksiniz. Sonraki karakterler -s yerel bir dizi olduğuna göre- rastgele karakterlerdir.

3) Sonlandırıcı Karakter Ne Amaçla Kullanılıyor?

Sonlandırıcı karakter işlemleri hızlandırmak için kullanılır. Çünkü sonlandırıcı karakter kullanıldığında ayrıca dizinin uzunluğunun bilinmesine gerek yoktur. Karakter dizileri üzerine işlem yapan algoritmalar dizilerin sonlandırıcı karakter ile bittiğini varsayarak daha hızlı işlem yaparlar. Karakter dizilerinin fonksiyonlara adres yoluyla geçirilmesi durumunda da ayrıca dizilerin uzunluklarının geçirilmesine gerek kalmaz. Ancak sonlandırıcı karakterin dizi içinde yer kaplaması bir dezavantajdır. Bu nedenle n elemanlı bir diziye en fazla n-1 tane karakter girilmelidir.

puts ve %s formatıyla **printf** sonlandırıcı karakteri görene kadar bütün karakterleri ekrana yazar. Bu durumda herhangi bir biçimde sonlandırıcı karakter

ezilirse, bu fonksiyonlar tesadüfen ilk sonlandırıcı karakteri görene kadar yazma işlemine devam edeceklerdir. Örneğin:

```
char s[30] = "Deneme";
```

?	s[0]
'D'	s[0]
'e'	s[1]
'n'	s[2]
'e'	s[3]
'm'	s[4]
'e'	s[5]
'\0'	s[6]
?	s[7]
?	s[8]
?	s[9]
?	s[10]

```
s[6] = 'x';
puts(s);
```

ile sonlandırıcı karakter ortadan kaldırılmıştır; bu durumda puts:

```
Denemex....
```

yazdıktan sonra tesadüfen ilk sonlandırıcı karakteri görene kadar yazmaya devam eder. **puts** ve **printf** fonksiyonları karakter dizilerini yazarken yalnızca sonlandırıcı karakteri dikkate alırlar; dizinin tanımlanmış olan uzunluğu ile ilgilenmezler.

Yukarıda anlattıklarımızı özetlersek, sonlandırıcı karakter kullanımı derleyicinin de onayladığı karşılıklı bir varsayımdır. Sonlandırıcı karakteri bazen programcı, bazen derleyici bazen de fonksiyonlar diziye yerleştirirler. Karakter dizileri üzerinde işlem yapan algoritmalar da dizilerin bittiğini bu karakterden anlarlar. Sonlandırıcı karakter yalnız karakter dizileri için söz konusudur (Tamsayı dizilerinde sıfır sayısı zaten kullanılan bir sayıdır). Diğer tür dizilerde sonlandırıcı karakter diye bir kavram yoktur.

Aşağıda sonlandırıcı karakterin dizilerde kullanımına ilişkin iki örnek veriyoruz:

1) Karakter dizisi içiresindeki yazının uzunluğunu bulan program:

```
#include <stdio.h>
void main()
```

```
{
    int  k = 0;
    char s[50];

    printf("String: ");
    gets(s);                    /* Klavyeden string alınıyor */
    while (s[k] != '\0')        /* Sonlandırıcı karaktere kadar */
        ++k;
    printf("uzunluk = %d\n", k);
}
```

Bu örnekte önce **gets** fonksiyonu klavyeden girilen karakterleri diziye yerleştirip, sonuna '\0' numaralı karakteri koyuyor. Daha sonra sonlandırıcı karaktere kadar **while** döngüsü içerisinde sayaç ilerletilmiştir. Aynı algoritma **for** döngüsüyle daha kısa yapılabilirdi:

```
for (k = 0; s[k] != '\0'; ++k)
    ;
```

2) Karakter dizisini tersyüz eden program:

```
#include <stdio.h>

void main(void)
{
    char s[30];
    int n, j, temp;

    printf("String: ");
    gets(s);
    for (n = 0; s[n] != '\0'; ++n)      /* Yazının uzunluğu bulunuyor */
        ;
    for (j = 0; j  <  n / 2; ++j) {
        temp = s[n-j-1];                /* Yer değiştirme */
        s[n-j-1] = s[j];
        s[j] = temp;
    }
    puts(s);
}
```

17.12 ÇOK BOYUTLU DİZİLER

C'de çok boyutlu diziler içsel olarak tek boyutlu dizi biçiminde saklanır. Çok boyutlu dizi bildirimlerinin genel biçimi aşağıdaki gibidir:

```
<tür> [n1], [n2], [n3],...
```

Boyut uzunluklarını belirten **n1**, **n2**, **n3**, ... sabit ifadeleri olmak zorundadır. Örneğin:

```
int x[10] [20];
```

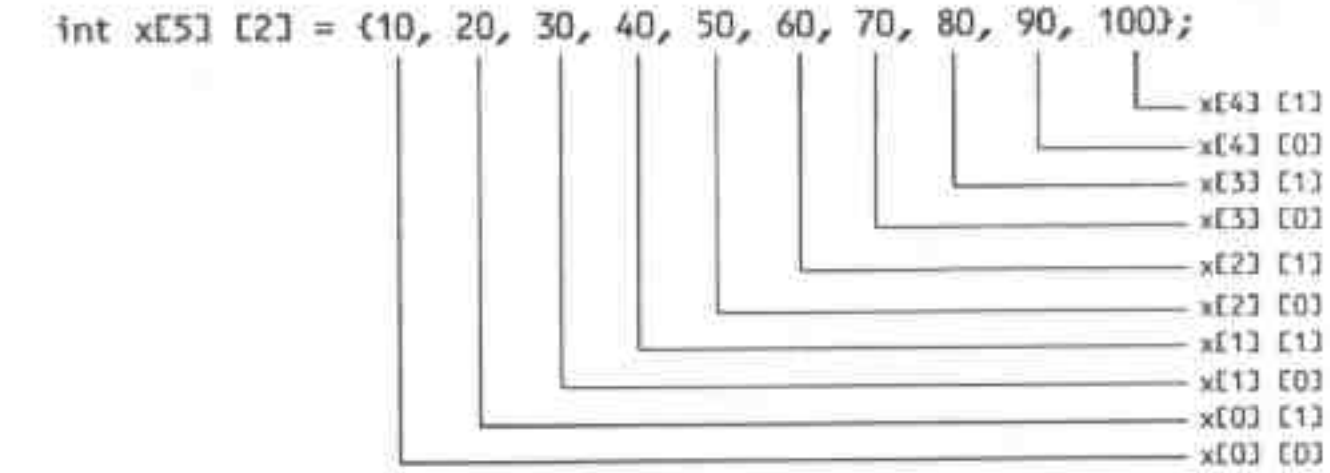
Burada **x** iki boyutlu **int** türünden bir dizidir. Boyut uzunlukları sırasıyla 10 ve 20'dir.

```
double n[5] [10] [15];
```

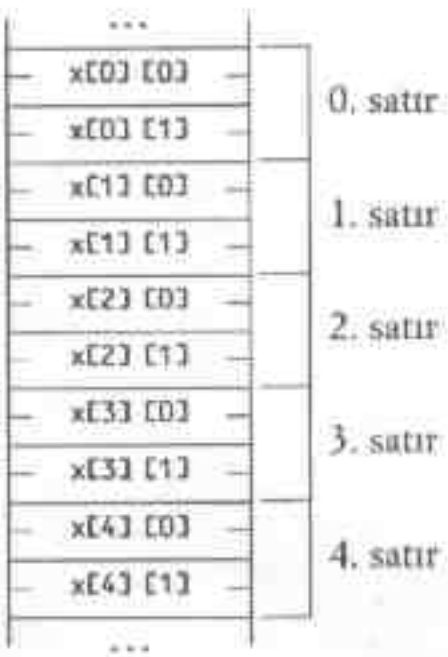
n üç boyutlu **double** türünden bir dizidir. Boyut uzunlukları sırasıyla 5, 10 ve 15'tir.

Uygulamada genellikle iki boyutlu dizilere rastlanır. İki boyutlu diziler matris olarak da isimlendirilirler. Matrislerin ilk boyutuna satır ikinci boyutuna sütun denir. İki boyutlu dizilere boyut sırasına göre küme parantezleri içerisinde ilkdeğer verilebilir.

Çok boyutlu diziler tek boyuta indirgenerek bellekte tutulurlar. Örneğin 5 satır, 2 sütunlu **int** türünden bir x dizisi için verilen elemanların bellekteki dizilimleri aşağıdaki gibi olacaktır.



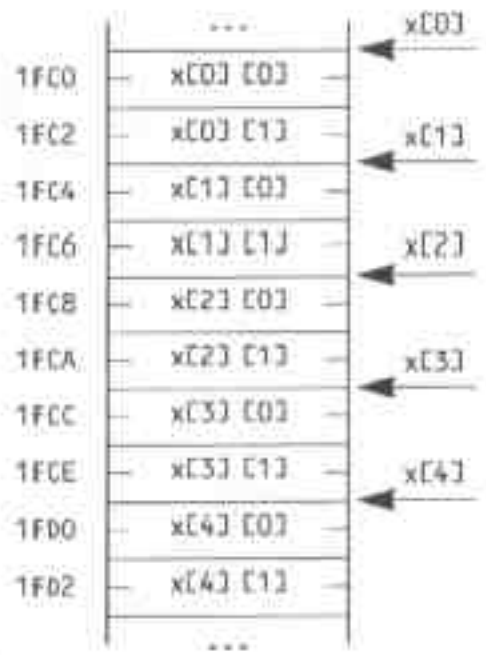
Çok boyutlu diziler belleğe düşük boyutu düşük anlamlı adreste bulunacak biçimde sürekli olarak yerleşirler.



Çok boyutlu dizilerin elemanlarına boyut sayısı kadar indeks operatörü kullanarak erişebiliriz. Örneğin, yukarıdaki x dizisinin i ve j indisli elemanına aşağıdaki gibi n sayısı atanabilir.

```
X[i][j] = n;
```

İki boyutlu dizilerin satır elemanları sol taraf değeri değildir; satırların bellekteki başlangıç adreslerini gösterirler. Örneğin yukarıdaki x matrisi belleğe 1FC0 adresinden başlayarak yerleşmiş olsun.



x[0], x[1], x[2], x[3], x[4] ifadeleri birer adres göstermektedir. Aynı durum ikiden fazla boyuta sahip olan diziler için de söz konusudur.

Aşağıdaki örnekte iki matrisin karşılıklı elemanları toplanarak başka bir matrise yazılmaktadır.; inceleyiniz.

```
#include <stdio.h>

#define ROW 5
#define COL 2

void main(void)
{
    int a[ROW][COL] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };
    int b[ROW][COL] = {5, 10, 15, 20, 25, 30, 35, 40, 45, 50 };
    int c[ROW][COL];
    int i, j;

    for (i = 0; i < ROW; ++i)
        for (j = 0; j < COL; ++j) {
            c[i][j] = a[i][j] + b[i][j];
            printf("[%d][%d] = %d\n", i, j, c[i][j]);
        }
}
```

SORAMADIKLARIMIZ...

S1) Tanımlanmış bir nesnenin adresini değiştirebilir miyiz?

C1) Hayır. Nesnelerin adresleri programcı tarafından belirlenemez ve değiştirilemez.

S2) Dizi tanımlamalarında açılacak dizinin uzunluğu neden değişkenler ile belirtilemiyor? Sabit ifadesi biçiminde belirtme zorunluluğu nereden kaynaklanıyor? Örneğin neden:

```
int n = 10;
...
char s[n];
...
```

mümkün değildir?

C2) Tanımlama işlemleri derleme zamanında değerlendirilirler. Derleyici, derleme zamanı içerisinde tanımlanmış nesneler için ne uzunlukta yer ayıracağını bilmek zorundadır. Oysa bir değişkenin değeri çalışma zamanı sırasında değişebilmektedir. Yani:

```
int n = 10;
...
n = k * 2;
...
{
    char s[n];
    ...
}
```

örneğinde n değişkeninin hangi değerde olduğunun bilinmesi derleme zamanında mümkün olmaz. n programın çalışmasıyla değiştiğine göre derleyici n değerini derleme aşamasında kestiremez. Kuşkusuz eğer değişkenin değerinin hiç değişmeyeceği garanti edilseydi böyle bir tanımlama da mümkün olabilirdi. Nitekim *C++* 'da `const` belirleyicisiyle tanımlanmış bir değişken -içerisindeki değer bir daha değiştirilemeyeceği için- dizi uzunluğunu belirlemekte kullanılmaktadır.

```
const int n = 10;
char s[n];
...
```

tanımlamaları *C++*'da geçerlidir.

S3) C'de dizi taşmaları neden derleyici tarafından kontrol edilmiyor?

C3) Dizi taşmalarının farkına varılması ancak derleyicinin amaç program içerisine kontrol kodları yerleştirmesiyle mümkün olabilir. Çünkü dizi taşması çalışma zamanı sırasında meydana gelmektedir. Fakat programcının dikkatsiz davranarak dizi taşmasına neden olacağı düşüncesi C'nin felsefesine uygun değildir. C'yi tasarlayanlar üretilecek kodun etkinliğinin ve doğallığının bozulmaması için herşeyi programcının sorumluluğuna bırakmışlardır. Biz bunu 1. Bölümde esneklik kavramıyla açıklamıştık.

Dizi taşmalarının neden olduğu hatalar **"gösterici hataları"** başlığıyla sonraki bölümde ayrıntılı bir biçimde ele alınmaktadır.

18

GÖSTERİCİLER

C bir gösterici dilidir, göstericisiz bir *C* programı düşünülemez. Bu nedenle göstericiler konusu da *C* öğreniminin bel kemiğini oluşturur. Ancak *C* öğrencileri diğer programlama dillerinin çoğunda bulunmayan göstericilere zor isimlar ve anlamakta güçlük çekerler. Göstericiler konusunun *C* eğitiminde kötü bir ünü vardır. Biz de bunları gözönünde bulundurarak bu bölümde konuyu mümkün olduğunca açık ve anlaşılabilir bir biçimde ele almaya gayret ettik. Tam bir anlaşılabilirlik sağlamak için şekillerden faydalandık. Ancak kullandığımız teknik ne kadar iyi olursa olsun gayret yine öğrencinin kendisine kalmaktadır. Size bir önerimiz olacak: Bu konuyu yalnız bir kez okumayınız. En az iki kez okuyunuz.

Bölüm içerisinde fazlaca uygulama koymak yerine uygulamaları ayrı bir bölümde topladık. Bunun yanı sıra ilerideki bölümlerde başka konular içerisinde göstericilerin doğal olarak kullanıldığını da göreceksiniz.

Unutmayın, bu bölümde anlamadığınız en ufak bir nokta kalmaması gerekir.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Gösterici nedir ve içerisinde hangi tür bilgiler tutulur?

2) Gösterici bildirimleri nasıl yapılır?

3) Göstericilerin türleri ne anlam ifade etmektedir?

4) Göstericilerin *UNIX* ve *DOS* sistemlerinde uzunlukları ne kadardır?

5) Gösterici operatörleri nelerdir? Hangi amaçla kullanılırlar?

6) Gösterici operatörlerinin diğer operatörlere göre öncelikleri nasıldır?

7) Göstericilerin artırılması ve eksiltilmesi sırasında içlerindeki adreslerin değişimi nasıl olmaktadır?

8) Gösterici operatörleriyle ++ ve — operatörlerinin yanyana kullanılmaları derleyiciler tarafından nasıl yorumlanır?

9) Gösterici hataları ne anlama gelir? Ne zaman ve ne biçimde ortaya çıkarlar ve hangi sonuçlara neden olabilirler?

10) Fonksiyon parametrelerinde gösterici kullanılmasının anlamı nedir?

11) Dizilerin fonksiyonlara parametre yoluyla geçirilmesi nasıl yapılmaktadır?

12) Göstericilerin faydaları nelerdir?

13) Göstericilerin neden olduğu uyarılar ve hatalar hangi durumlarda ortaya çıkarlar?

14) `void` göstericiler nasıl ve neden kullanılırlar?

18.1 GİRİŞ

Şimdiye kadar görmüş olduğumuz veri/nesne türlerinin hepsi temel olarak aritmetik, ilişkisel ve mantıksal operatörlerle birlikte sayısal işlemleri yürütmek amacıyla kullanılmaktadır. Örneğin, `float` türünden bir değişkenin içeriğini bir sayıyla çarpabilir, sonucu da başka bir sayıya bölebilirsiniz; ya da bir tamsayı değişkenini mantıksal operatörlerle işleme sokabilirsiniz.

Şu ana kadar ele aldığımız bütün veri/nesne türlerini iki gruba ayırmıştık.

1) **Tamsayı türleri:**

```
char / unsigned char
short / unsigned short
int / unsigned int
long / unsigned long
```

2) **Gerçek Sayı Türleri:**

```
float
double
long double
```

C'de adresler ve göstericiler ayrı ve üçüncü bir tür olarak ele alınmaktadır.

17. Bölümde adres kavramına bir giriş yapmıştık. Bu bölümde işlevsel olarak di-

18.2 AYRI BİR TÜR OLARAK ADRES

ğer türlere benzemeyen adresler hakkında biraz daha ayrıntılı bilgi edineceğiz. **Yazılımsal olarak adres bilgisi yalnızca bir sayıdan ibaret değildir; aynı zamanda o adresin gösterdiği yerdeki bilginin ne biçimde yorumlanacağını belirten bir tür bilgisini de içermektedir.**

Adres bilgisi ayrı bir veri/nesne türüdür. Bu durumda C'deki veri/nesne türlerini 3 bölüme ayırabiliriz:

1) Tamsayı türleri

2) Gerçek sayı türleri

3) Adresler ve göstericiler

Yazılımsal olarak bir adresin iki bileşeni vardır.

1) Bellekte yer gösteren sayısal bir değer
2) Adresin türü

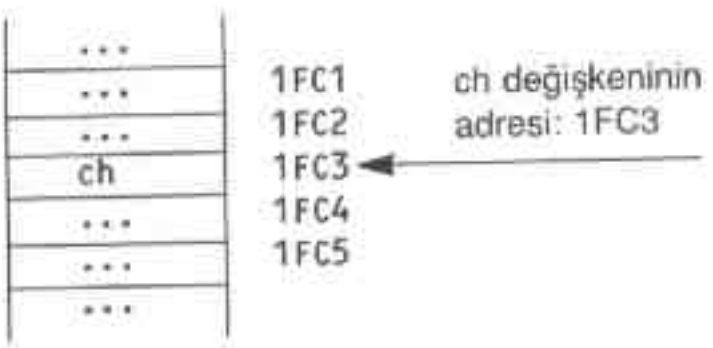
Adreslerin sayısal bileşenlerinin bellekte bir yer gösterdiğini biliyoruz; şimdi de **"adreslerin türleri"** kavramını biraz açmak istiyoruz.

18.2.1 Adreslerin Türleri

Bir adresin türü demekle o adrese ilişkin bellek bölgesinde bulunan bilginin derleyici tarafından yorumlanış biçimi anlaşılmalıdır. Örneğin:

```
char  ch;
```

gibi bir bildirim sonrasında ch değişkeninin belleğe aşağıdaki biçimde yerleşmiş olduğunu kabul edelim:



Yukarıdaki şekle baktığımızda 1FC3 sayısının bir adres belirttiğini anlıyoruz. 1FC3 adresinde bulunan bilgi **char** türündendir. Çünkü derleyici 1FC3 adresinde bulunan bilgiyi karakter olarak yorumlamaktadır. **Adres yalnız başına bir sayı olarak anlam taşımaz; gösterdiği tür ile birlikte belirtilmesi gerekir.**

Dizi isimlerinin gerçekte dizilerin bellekteki başlangıç adresleri olduğunu anımsayınız. **Dizi isimlerine ilişkin adreslerin türleri dizilerin türleriyle aynıdır.** Aşağıdaki örnek bildirimleri inceleyiniz:

```
char s[30];
int t[20];
unsigned long dizi[50];
...
```

Dizi isimlerinin belirttiği adresler dizi ile aynı türden olduğuna göre yukarıda-

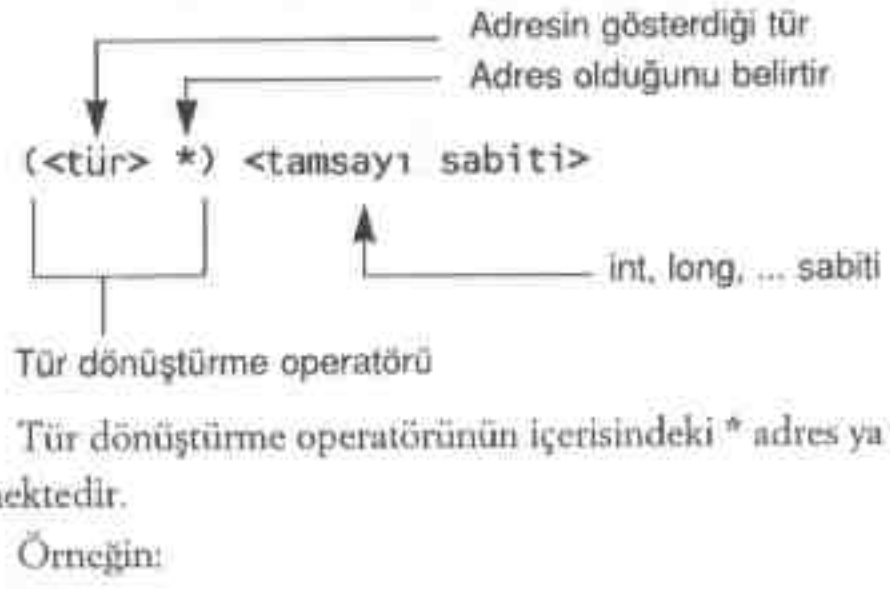
ki örnekte `s` adresinin türü `char`, `t` adresinin türü `int`, **dizi** adresinin türü ise `unsigned long`'dur.

Her türe ilişkin sabitler olduğuna göre, adres türüne ilişkin sabitlerin de olması gerekmez mi?

18.3 ADRES SABİTLERİ

Sabitleri ele aldığımız 6. Bölümde, tamsayı ve gerçek sayı türlerine ilişkin sabitlerin nasıl belirtildiklerini açıklamıştık. Peki acaba adres sabitleri nasıl belirtiliyorlar?

Adresler tamsayı görünümünde olsalar da tamsayı sabitleri gibi belirtilmezler. **Adres sabitleri, tamsayı türlerindeki sabitler üzerinde bilinçli tür dönüştürmesi yapılarak elde edilirler.** Bir tamsayı sabitini adres türüne çevirmek için tür dönüştürme operatörü kullanılır.



Tür dönüştürme operatörünün içerisindeki * adres ya da göstericiyi temsil etmektedir.

Örneğin:

```
0X1FC0
```

HEX biçiminde yazılmış bir **tamsayı** (`int`) sabitidir. Ancak:

```
(float *) 0X1FC0
```

`float` türünü gösteren bir adres sabitidir. Yani derleyici `1FC0` adresinden başlayan bilgiyi `float` olarak yorumlayacaktır. Diğer bir örnek:

```
0X13C0L
```

bir uzun tamsayı sabitidir. Ancak:

```
(int *) 0X13C0L
```

ifadesi **tamsayı** (`int`) türünü gösteren bir adres sabiti olarak değerlendirilir. Derleyici, `13C0` adresiyle belirtilen yerdeki bilgiyi **tamsayı** (`int`) olarak yorumlayacaktır.

Adres türüne bilinçli dönüşüm yalnızca sabitlerle yapılmayabilir; örneğin:

```
(char *) num
```

num isimli değişken hangi türden olursa olsun, yukarıdaki ifade ile karakter türünü gösteren adres türüne dönüştürülmüştür.

Tamsayı (int) sabitleri tamsayı türünden değişkenlere, gerçek sayı sabitleri gerçek sayı türünden değişkenlere doğal olarak atanıyorlar. Yani her sabit türünün atanabileceği o türden bir değişken tanımlayabiliyoruz. Peki adres sabitleri hangi tür değişkenlere atanabilirler; adres sabitlerine ilişkin nesneler var mıdır? Vardır, işte onlara gösterici (pointer) diyoruz!..

'a'	char türünden bir sabittir.
char a;	char türünden bir değişkendir.
123	int türünden bir sabittir.
int b;	int türünden bir değişkendir.
1500L	long türünden bir sabit
long c;	long türünden bir değişken
(<tür> *) 0x1FC0	adres sabiti
gösterici	**adres barındıran bir değişken**

18.4 GÖSTERCİLERİN BİLDİRİMLERİ

Göstericiler adres bilgilerini saklamak ve adreslerle ilgili işlemler yapmak için kullanılan nesnelerdir. Göstericilerin içerisinde adresler bulunur. Bu nedenle gösterici ile adres hemen hemen eş anlamlı olarak düşünülebilir. Ancak gösterici deyince bir nesne, adres deyince de daha çok nesne olmayan bir bilgi akla gelmektedir.

Gösterici bildirimlerinin genel biçimi şöyledir:

```
<tür> * <gösterici ismi>;
```

<tür> göstericinin (içerisindeki adresin) türüdür. char, int, float, .. gibi herhangi bir tür olabilir.

Burada * göstericiyi ya da adresi temsil etmektedir.

Örnek gösterici bildirimleri:

```
float *f;
char *s;
int *dizi;
```

```
unsigned long *PDWORD;
...
```

Gösterici bildirimlerinin diğer türlere ilişkin bildirimlerden * atomu ile ayrıldığına dikkat ediniz:

```
char s;
```

bildiriminde s karakter türünden bir değişkendir. Oysa;

```
char *s;
```

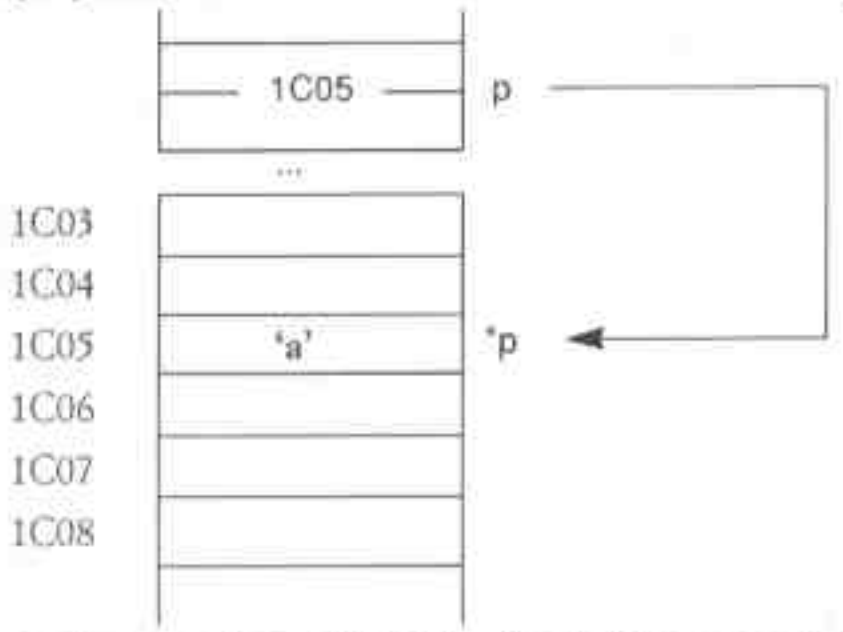
bildiriminde s bir göstericidir. İçerisine karakteri gösteren bir adres konulmalıdır. Bu bildirimden derleyici s nesnesinin bir gösterici olduğunu, s nesnesinin içinde bulunan adresin gösterdiği yerin char olarak yorumlanacağını anlar.

*** atomu, ileride ayrıntılı bir biçimde ele alınacak olan tek operandlı, önek bir gösterici operatörüdür. Adresin gösterdiği bellek bölgesindeki nesneyi temsil eder.**

Aşağıdaki örneği inceleyiniz:

```
char *p;
...
p = (char *) 0x1C05;
*p = 'a';
...
```

p bir gösterici, *p ise göstericinin gösterdiği adreste bulunan nesnedir. Dolayısıyla yukarıdaki örnekte 'a' karakteri 1C05 adresine yazılır.



Şüphesiz, p göstericisinin kendisi de bir nesne olduğu için onun da bir adresi vardır. Ancak biz bu örnekte p'nin adresiyle ilgilenmiyoruz. (Anlaşılması kolay olsun diye bellekte a karakterinin görünümü sayısal olarak değil de 'a' biçiminde verilmiştir. Bellekte herşeyin ikilik sistemde bir sayı olduğunu unutmayınız!)

Şimdi aşağıdaki örneği inceleyelim:

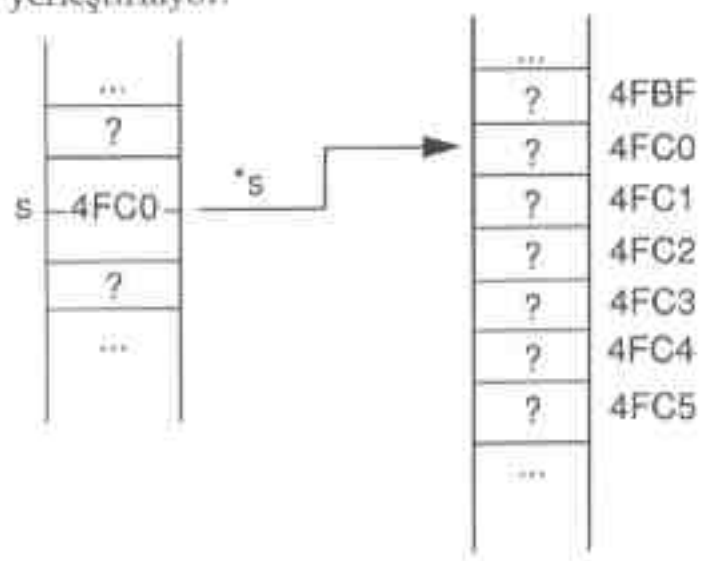
```
char *s;
s = (char *) 0x4FC0;
...
*s = 'a';
++s;
*s = 'b';
...
```

Burada adım adım neler olduğunu izleyelim:

1.Adım:

```
s = (char *) 0x4FC0;
```

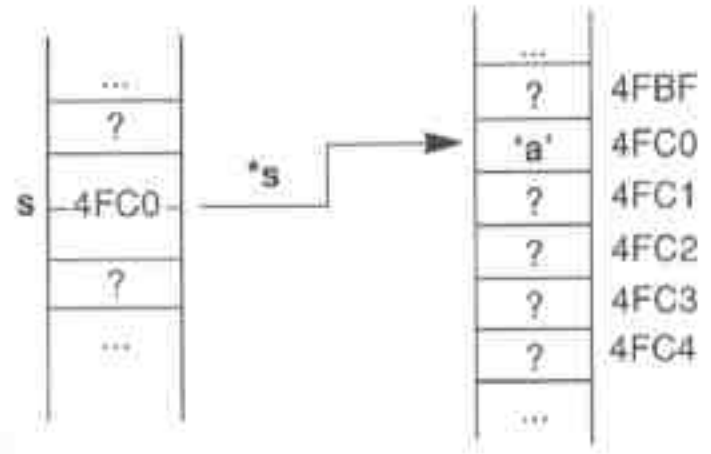
ile karakter türünden s isimli göstericinin içerisine karakteri gösteren 4FC0 adresi yerleştiriliyor.



2. Adım:

```
*s = 'a';
```

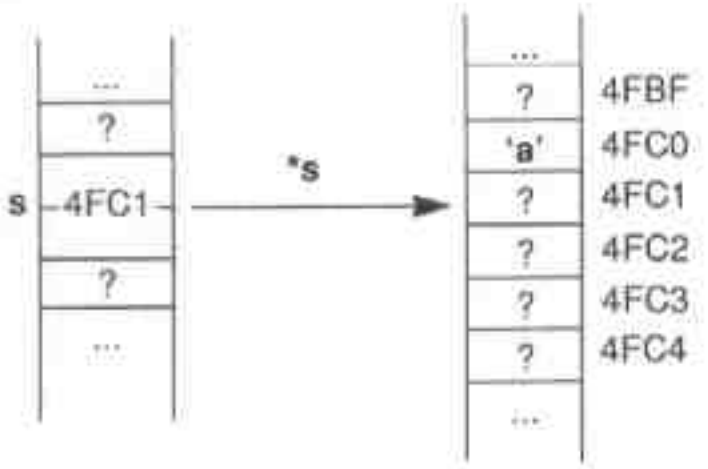
ile 4FC0 adresine 'a' yazılmıştır.



3. Adım:

```
++s;
```

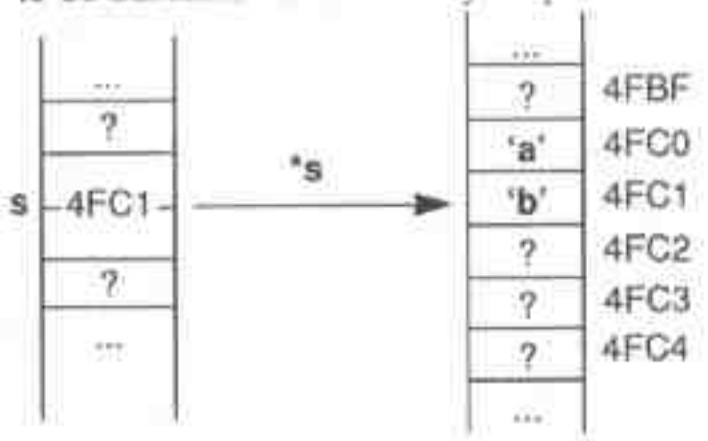
ile s göstericisinin içerisindeki adres 1 artırılmıştır. Şimdi `*s`, `4FC1` adresindeki nesneyi temsil ediyor.



4. Adım:

```
*s = 'b';
```

4FC1 adresine 'b' karakteri yazılıyor.



Adres bilgisinin yalnızca sayısal değerinin gösterici içerisinde tutulduğunu farketmişsinizdir. Evet, adresin türü bellekte tutulmaz; çünkü adresin türü yalnızca derleyici için gerekli olan bir bilgidir. Peki, örneklerimizde aynı türden adres sabitlerinin aynı türden göstericilere atandığını farkettiniz mi? Örneğin:

```
int *p;
...
p = (int*) 0x1B64
```

int türünden bir adres sabiti

int türünden bir gösterici

Bir göstericiye aynı türden adres sabiti atamaya özen göstermelisiniz. Farklı türden adreslerin farklı türden göstericilere atanması durumunun doğuracağı sakınca **18.14**'te ele alınmaktadır.

18.5 GÖSTERİCİLERİN UZUNLUKLARI

Bir gösterici tanımlamasıyla karşılaşan derleyici -diğer tanımlamalarda yaptığı gibi- bellekte o gösterici için bir yer tahsis eder. Derleyicilerin göstericiler için tahsis ettikleri yerlerin uzunluğu donanım bağımlı olup sistemden sisteme değişebilmektedir. 32 bit sistemlerde (örneğin *UNIX* ve *Windows 95* sistemlerinde) göstericiler **4 byte** uzunluğundadır. 8086 mimarisinde ve *DOS* altında çalışan derleyicilerde ise göstericiler **2 byte** ya da **4 byte** olabilirler. *DOS*'ta **2 byte** uzunluğundaki göstericilere yakın göstericiler (near pointer), **4 byte** uzunluğundaki göstericilere ise uzak göstericiler (far pointer) denilmektedir. Yakın ve uzak gösterici kavramları ileride ayrıntılı bir biçimde ele alınacaktır. Ancak biz uygulamalarımızda şimdilik göstericilerin **2 byte** olduğunu varsayacağız.

Göstericilerin uzunluğunun türlerinden bağımsız olduklarını vurgulamak istiyoruz. Örneğin:

```
char *s;
int *p;
float *f;
...
```

DOS altında `s`, `p` ve `f` isimli göstericilerin hepsi de bellekte 2 byte ya da 4 byte yer kaplarlar. Çünkü göstericilerin türü yalnızca gösterdikleri yerle ilişkilidir.

18.6 GÖSTERİCİ OPERATÖRLERİ

Gösterici operatörleri adres işlemleri yapan operatörlerdir. C'de dört tane gösterici operatörü vardır. Biz bu bölümde bunlardan üçü olan: **İçerik (`*`)**, **adres (`&`)** ve **indeks (`[n]`)** operatörlerini ayrıntılı bir biçimde ele aldık. Son gösterici operatörü olan ok operatörünü (->) yapılarla ilişkisi nedeniyle yapılar bölümünde incelemeyi uygun gördük.

18.6.1 İçerik Operatörü (*) (Indirection operator)

Daha önce çarpma operatörü olarak ele aldığımız * aynı zamanda gösterici operatörü olarak da kullanılır. **18.4** 'te verdiğimiz örnekler size * operatörünün göstericilerle nasıl kullanıldığı hakkında küçük bir fikir vermiştir. * operatörü **tek operandlı, önek** bir gösterici operatörüdür. Operandının mutlaka bir adres olması gerekir. Operand olan adres bir gösterici biçiminde olabileceği gibi, dizi ismi ya da adres sabiti biçiminde de bulunabilir. * operatörü, operandı olan adresin gösterdiği yerdeki nesneye erişmek amacıyla kullanılır. `p` bir adres belirtiyorsa `*p` bu adreste bulunan nesneyi temsil eder.

Örneğin:

```
char *p;
...
p =(char *) 0x1FC0;
```
gösterici

```
* p = 'a';
```
içerik operatörü

Bu örnekte 'a' karakteri p göstericisinin içerisindeki adrese , yani 3FC0 bölgesine yazılıyor.

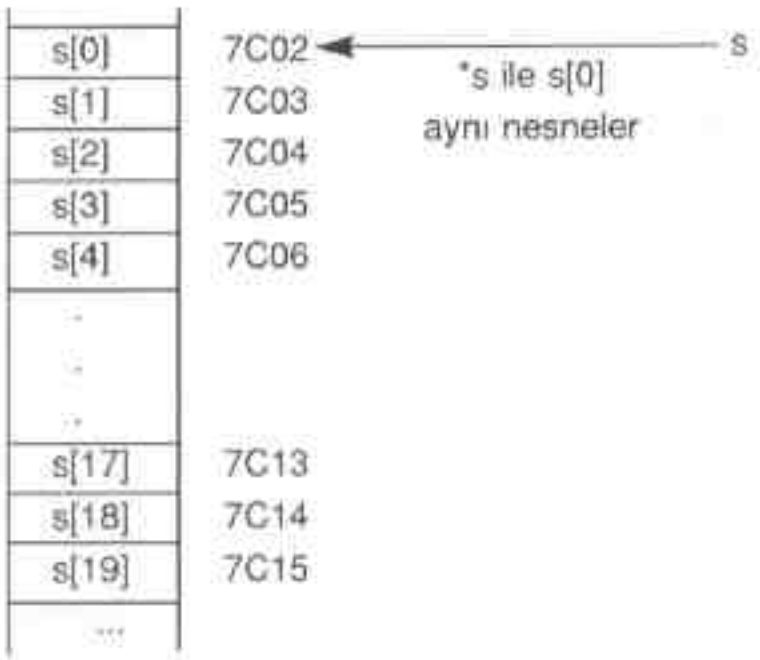
*operatöründün operandının gösterici olması zorunlu değildir; ancak adres olması zorunludur. Dizi isimleri de birer adres olduğuna göre, * operatörü dizi isimleriyle birlikte de kullanılabilir. Örneğin:

```
char s[20];
...
* s = 'a';
```
içerik operatörü

Dizi ismi

Burada 'a' karakteri dizinin ilk elemanına atanıyor değil mi?

s[0] = 'a' ile *s = 'a' ifadelerinin aynı anlama geldiğine dikkat ediniz.



* operatörünü adres sabitleriyle birlikte de kullanabilirsiniz. Örneğin:

```
* (char *) 0XFC00 = 'a';
```
içerik operatörü

adres sabiti

Burada 'a' karakteri doğrudan **FC00** adresine yazılır.

18.6.1.1 İçerik Operatörünün Önceliği

İçerik operatörü daha önce gördüğümüz ! ~ ++ — - operatörleri ile **sağdan sola** eşit önceliğe sahiptir.

```
() ...                                  Soldan sağa
!  ~  ++  —  (tür) sizeof  * ...        Sağdan sola
*  /  %                                 Soldan sağa
...                                     ...
```

* operatörünün önceliğinden doğabilecek hatalara dikkatinizi çekmek istiyoruz. Örneğin aşağıdaki ifadeyi inceleyiniz.

```
*s + 1 = 'x';
```

Burada * operatörünün önceliği + operatöründen yüksek olduğu için:

```
İ1: *s
İ2: İ1 + 1
İ3: İ2 = 'x' Hata! İ2 nesne değil!
  ...
```

ifadesi derleme aşamasında hata olarak değerlendirilir. Fakat, aşağıdaki ifade geçerlidir:

```
* (s+1) = 'x';
```

Not: * operatörünün ++ ve — operaörleriyle birlikte kullanımları ileride ayrı bir bölüm halinde ele alınmaktadır.

Aşağıdaki örnek de (* operatörünün operandı adres olmadığı için) hatalıdır.

```
int  p;
...
*p = 'a';                    /* Hata! p bir adres değil, tamsayı! */
```

* hem içerik hem de çarpma operatörü olarak kullanılsa da işlevleri birbiriyle karışmaz. Çünkü çarpma operatörü olarak * iki operandlı olup arack (infix) durumundadır. Derleyici operatörün ifade içerisindeki konumuna bakarak hangi amaçla kullanıldığını anlayabilir. Örneğin:

```
                 → İçerik operatörü
c = a * * b;
    └──────→ Çarpma operatörü
```

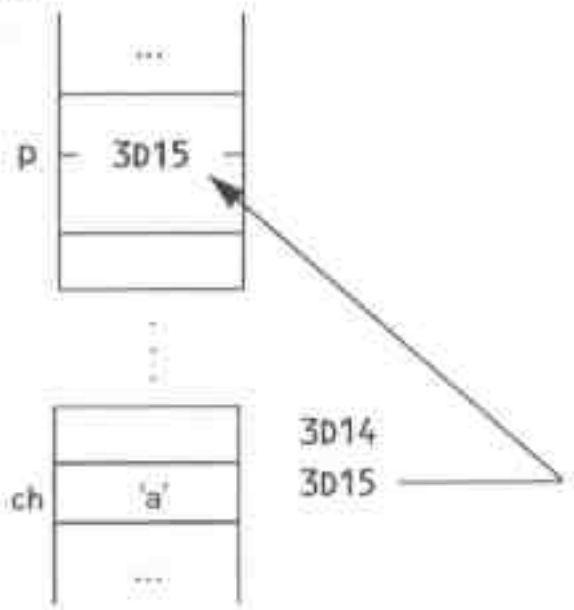
ifadesinde ilk * çarpma operatörü olarak ikinci * gösterici operatörü olarak kullanılmıştır.

18.6.2 Adres Operatörü (&) (Adress operator)

& operatörü tek operandlı önek bir gösterici operatörüdür. Operandı bir nesne olmak zorundadır. Bu operatörün ürettiği değer, nesnenin bellekteki adresidir. Örneğin:

```
char ch;
char *p;
...
ch = 'a';
p = &ch;
```

burada ch değişkenin adresi p göstericisine atanmaktadır. Aşağıdaki şekli inceleyiniz.



& operatörü yalnızca nesnelerle kullanılabilir. Çünkü ancak nesnelerin adresleri olabilir!.. Örneğin:

```
#define MAX    100
...
char *p;
...
p = &MAX;              /* hata! */
```

MAX bir sembolik sabittir; nesne değildir. Dolayısıyla & operatörünün operandı olamaz.

& operatörü ile üretilen adresin türü, operandı olan nesnenin türü ile aynıdır.

Örneğin:

```
int a;
int *p;
...
p = &a;
```

Burada **&a** sonucu elde edilen adres **int** türündendir.

Bu durumda *C*'de adres bilgisi 4 biçimde bulunabilir:

1) Adres sabitleri biçiminde

`(char *) 0x1FC0` gibi...

2) Dizi isimleri biçiminde

```
int x[20];
...
```

x gibi. Adresin türü dizinin türüyle aynıdır.

3) Adres (&) operatörü ile

```
int a;
...
```

`&a` gibi. Adresin türü nesnenin türüyle aynıdır.

4) Gösterici biçiminde

```
char *p;
...
```

p gibi. Adresin türü bildirim sırasında belirtilir.

18.6.2.1 Adres Operatörünün Önceliği

& operatörü, * gösterici operatörüyle **sağdan sola** eşit önceliklidir. Öncelik tablosunun ilgili bölümünü (şimdiye kadar görmüş olduğumuz operatörleri kapsayacak biçimde) tekrar veriyoruz:

`() ...`	Soldan sağa
`! ~ ++ — (tür) sizeof * & ...`	**Sağdan sola**
`* / %`	Soldan sağa
`...`	...

Örneğin:

```
char ch;
...
*&ch = 'a';
```

ifadesinde:

`I1: &ch`	**ch** değişkenin adresi (türü **char**)
`I2: *I1`	**ch** değişkenin kendisidir.
`I3: I2 = 'a'`	**ch** değişkenin içine 'a' konuluyor.

Önceki bölümde -*DOS* altında çalışıyorsak- adres bilgilerini **printf** fonksiyonunu "**%p**" formatıyla kullanarak yazdırabileceğinizi açıklamıştık. *UNIX* altında

çalışan derleyicilerde adresleri yazdırmak için bilinçli tür dönüştürmesi yapmanız gerekebilir. Örneğin:

```
char *p;
...
printf("%lx\n" (unsigned long ) p);
...
```

18.6.3 İndeks Operatörü ([n]) (Index operator)

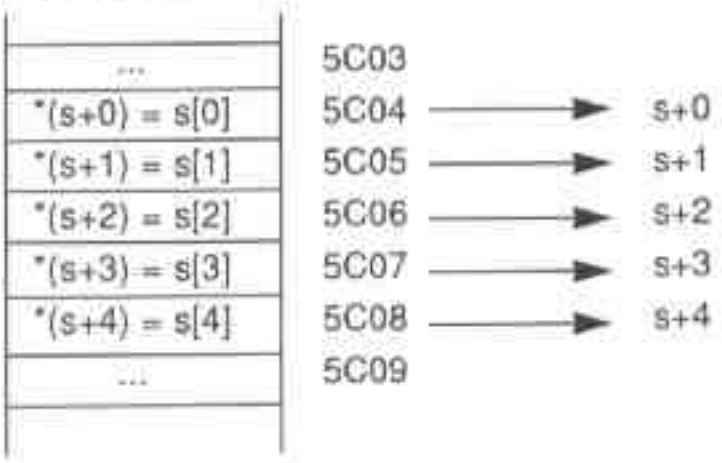
Dizi elemanlarına erişmek için kullandığımız `[n]` aslında tek operandlı sonek bir gösterici operatörüdür. Operandının mutlaka adres olması gerekir. `[n]` operatörü, operandı olan adresten `n` uzaklıktaki nesneyi temsil eder. Bu durumda:

`* (s+n)` ile `s[n]` aynı anlamdadır.

Örneğin::

```
char s[5];
```

ile tanımlanan `s` dizisi belleğe `5C04` adresinden başlayarak yerleşmiş olsun Aşağıdaki şekli inceleyiniz:



Dizi elemanlarına erişmek için kullandığımız indeks operatörü aslında genel amaçlı bir gösterici operatörüdür demiştik. * operatörünün göstericilerle `[n]` operatörünün ise dizilerle kullanılması yönünde bir zorunluluk yoktur; böyle bir koşullamayı üzerinizden atmalısınız!

Aşağıdaki örneği inceleyiniz:

```
char s[5];
char *p;
...
p = s;
for (k = 0; k < 5; ++k)
    p[k] = 0;
 ...
```

Burada `s` dizisinin başlangıç adresi, `p` göstericisine atanmış; `p` göstericisi de daha sonra indeks operatörüyle kullanılmıştır.

18.6.3.1 Index Operatörünün Önceliği

İndex operatörü C'nin en öncelikli operatörleri arasındadır. Öncelik tablosunun -şimdiye kadar görülen operatörleri dikkate alarak- ilgili bölümünü aşağıda veriyoruz.

```
()  []  ...                          Soldan Sağa
!  ~  ++  --  (tür)  sizeof  *  &     Sağdan sola
...
```

Görüldüğü gibi indeks operatörü diğer gösterici operatörlerinden daha önceliklidir. Örneğin:

```
char s[20];
char *p;
...
p = &s[0];
```

ifadesinde önce `s[0]` işlemi yapılır; daha sonra adresi alınır.

`i1: s[0]`	Dizinin ilk elemanı
`i2: &i1`	Dizinin ilk elemanının adresi
`i3: p = i2`	Dizinin başlangıç adresi `p` göstericisine atanıyor. Bu işlem `p = s` ile aynı işleve sahiptir.

Madem bir dizinin ismi, dizinin başlangıç adresini gösteren nesne olmayan bir adrestir; o halde `s` bir dizi ismi olmak üzere:

`s` ile `&s[0]` eşdeğer ifadelerdir.

18.7 GÖSTERİCİLERİN ARTIRILMASI VE EKSİLTİLMESİ

Göstericiler *, /, %, ... gibi aritmetik operatörlerle ve bit operatörleriyle kullanılamazlar. Göstericilerin ilişkisel ve mantıksal operatörlerle kullanılması uzak göstericilerin anlatıldığı bölümde incelenmektedir. Fakat, bir gösterici tamsayılarla artırılabilir ya da eksiltilebilir. Örneğin, `p` bir gösterici olmak üzere aşağıdaki ifadelerin hepsi geçerlidir:

```
++p;
p = p - 2;
p += 4;
p -= 10;
--p;
...
```

Göstericiler yalnızca tamsayı sabitleri ile değil tamsayı değişkenleriyle de artırılabilir ve azaltılabilirler:

```
char *p;
int a;
...
p = p + a;
p = p - 2;
p += a;
p = p + a + 10;
```

Gösterici bir artırıldığında içerisindeki adres göstericinin türünün uzunluğu kadar artmaktadır.

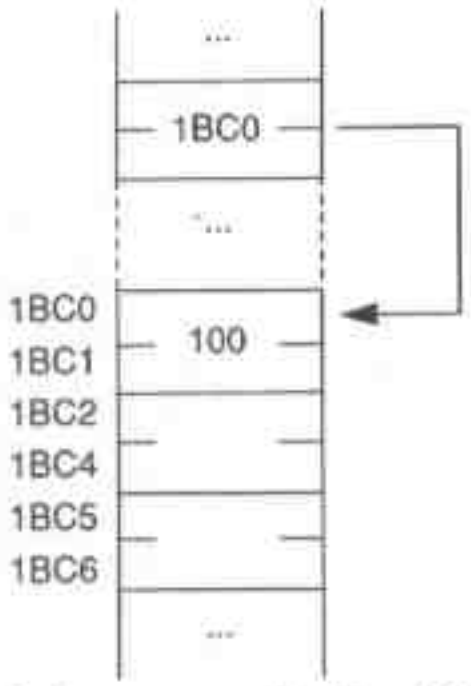
Örneğin:

```
int x[20];
int *p;
...
p = x;
*p = 100;
++p;
*p = 200;
...
```

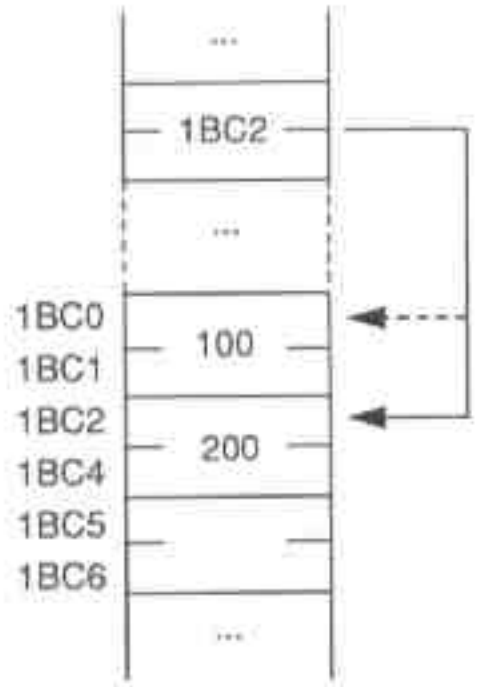
Burada ilk önce `x` dizisinin başlangıç adresi `p` göstericisine atanmıştır. `x` dizisinin `1BC0` adresinden başlayarak belleğe yerleştiğini varsayalım. Bu durumda:

```
*p = 100;
```

ile `100` sayısı `1BC0` adresine (`x[0]` içerisine) yerleştiriliyor. (`1BC0 - 1BC1`)



Daha sonra `++p` ile gösterici `1` artırılıyor. Bu durumda `p` göstericisinin içerisindeki adres -türü `int` olduğu için- `2` artacaktır. Böylece `200` sayısı `1BC2` adresine (`x[1]` içerisine) yazılmaktadır.



Aynı biçimde `int` türünden bir gösterici `1` eksiltilirse içerisindeki adres de `2` eksilir. Bu durum sizi şaşırtmamalı. Çünkü bir göstericiyi `1` eksiltince, göstericinin bir önceki elemanı gösterebilmesi için içerisindeki adresin tür uzunluğu kadar eksiltilmesi gerekmektedir. Aynı biçimde, aşağıdaki örnekte `double` türünden `x` dizisinin başlangıç adresi `d` isimli göstericiye atanmıştır. `x` dizisi için bellekte derleyici tarafından `1FC0` bölgesinin tahsis edildiğini varsayalım:

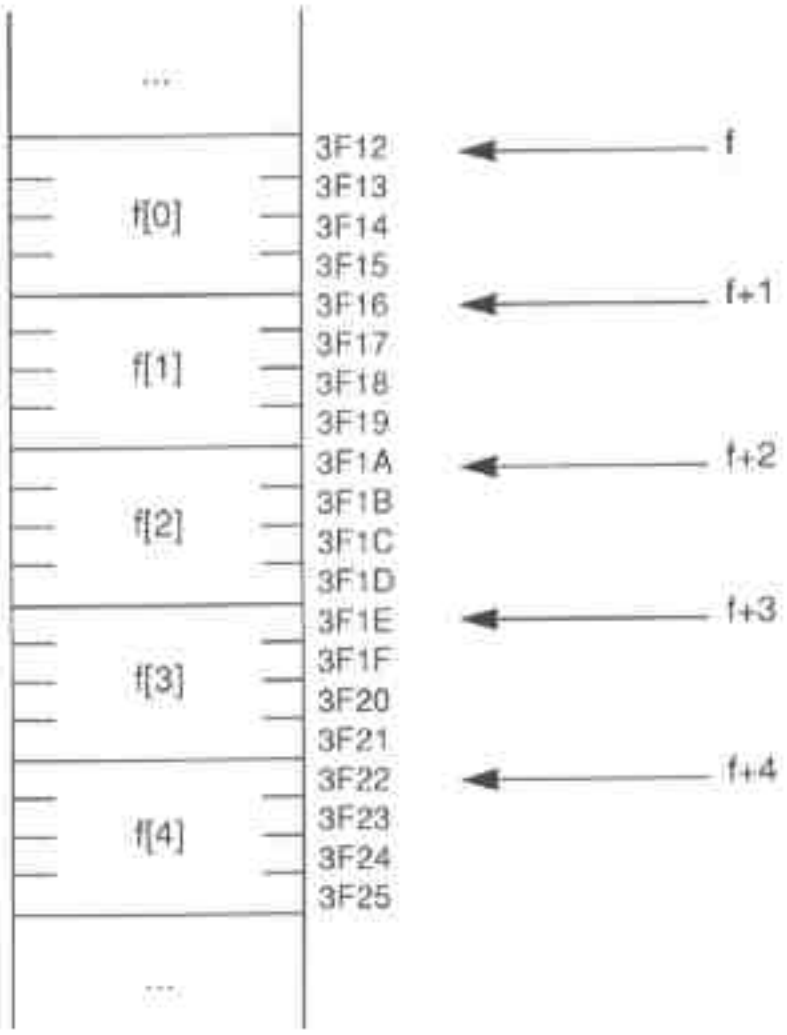
```
double x[20];
double *d;
...
d = x;
*d = 0.1;          /* 1FC2 adresinden başlayarak yazılıyor */
d = d + 2;         /* d içerisindeki adres 16 artarak 1FD2 oluyor */
*d = 0.2;          /* 1FD2 adresinden başlayarak yazılıyor */
...
```

bu örnekte `d = d + 2` ile gösterici `2` artırıldığında içerisindeki adres `16` artmaktadır.

Aynı durum indeks operatörü için de geçerlidir. `x[n]` ifadesi `x` adresinden `n` byte sonraki nesne anlamına gelmez. `x` adresinden `sizeof(*x) * n` byte sonraki nesne anlamına gelir. Örneğin

```
float f[5];
```

biçimindeki `f` dizisinin bellekte `3F12` adresinden başlayarak yerleştiğini varsayalım:



Örneğin: `f[3]` ifadesi `f` adresinden 12 yani `3 * sizeof(float)` kadar uzaklıktaki nesneyi gösterir. `x[n]` ve `*(x+n)` ifadelerinin aynı anlama geldiğini anımsayınız. O halde, `f[3]` ile `*(f+3)` de aynı anlama gelir, değil mi?

18.8 GÖSTERİCİ OPERATÖRLERİNİN ARTIRMA VE EKSİLTME OPERATÖRLERİYLE BİRLİKTE KULLANILMASI

Gösterici operatörleri ile ++ ve -- operatörlerinin ifade içerisinde yanyana kullanımı C öğrencilerini çatışmaya sevk etmektedir. Bu nedenle ayrı bir konu içerisinde ele almayı uygun gördük.

18.8.1 İndeks Operatörü ile Kullanılması

++ ve -- operatörlerinin indeks operatörü ile birlikte kullanılmasında çatışmaya yol açacak bir durum yoktur. Yalnızca indeks operatörünün daha yüksek öncelikli olduğunu anımsamak yeterli.

Örneğin:

```
s[n] = 10;
...
a = ++s[n];
```

ifadesinde s[n] bir artırılır.

```
i1: s[n]      ► 10
i2: ++i1      ► 11
i3: a = i2    ► 11
```

benzer biçimde:

```
a = s[n]—;
```

ifadesinde önce s[n], a değişkenine atanır, sonra bir eksiltilir.

18.8.2 Adres operatörü ile

++ ve -- operatörleri adres operatörü ile yanyana kullanılamaz. Örneğin:

```
a = &++b;
```

işlemi geçerli değildir. Çünkü ifadenin eşdeğeri:

```
a = & (b = b +1)
```

biçimindedir. Atama operatöründen elde edilen değerin nesne olmadığını anımsayınız. Oysa & operatörü yalnızca nesnelerle kullanılabilir. Benzer biçimde:

```
a = —&b;
```

işlemi de &b nesne olmadığı için geçerli değildir.

18.8.3 İçerik Operatörü İle

++ ve -- operatörlerinin * operatörüyle kullanılmasına ilişkin üç durum söz konusu olabilir.

1) *operand++ ya da *operand-- Durumu

Bu tür ifadelerde ++ ve -- sonek durumunda ve içerik operatörü olan * ile sağdan sola eşit önceliğe sahip olduğuna göre:

Önce ***operand** işleme sokulur; ifadenin sonunda **operand** bir artırılır ya da eksiltilir.

Aşağıdaki örnekte s karakter türünden bir göstericidir, inceleyiniz:

```
while (*s++ != '\0')
   ;
```

Burada döngüden çıkınca s, NULL karakterden bir sonrayı gösterir. Çünkü döngü boş deyim ile kapatıldığına göre yalnızca kendi içerisinde dönecektir. Döngü içerisindeki

```
*s++ != '\0'
```

ifadesini ele alalım:

```
İ1: *s
İ2: İ1 != '\0'
İ3: s = s + 1
```

Yani *s, NULL karaktere eşit olduğu zaman da s sonek durumunda olduğundan bir artırılır. Oysa aşağıdaki örnekte s döngüden çıkışta **NULL** karakteri göstermektedir, inceleyiniz:

```
while (*s != '\0')
    ++s;
```

2) *++operand ya da *-- operand Durumu

Her iki operatör de sağdan sola öncelikli olduğuna göre önce **operand** artırılır ya da azaltılır daha sonra içeriği elde edilir. Örneğin:

```
char s[20];
char *p;
...
p = s;
*++p = 'A';                    /* s[1] = 'A'; */
```

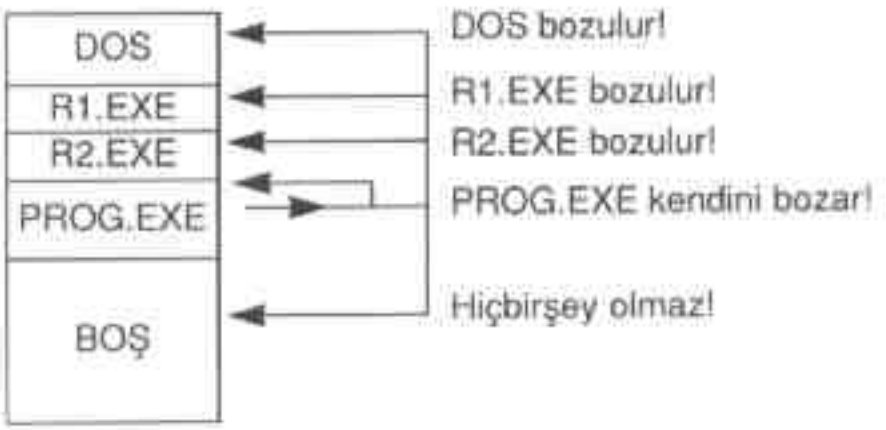
3) ++*operand ya da -- *operand Durumu

Burada sağdan sola öncelik kuralına göre ***operand** bir artırılır ya da bir eksiltilir. Örneğin:

```
char ch = 'a';
char *s;
...
s = &ch;                       /* s ch karakterini gösteriyor */
++*s;                          /* ch  bir artırılarak 'b' oluyor  */
```

18.9 GÖSTERİCİ HATALARI

Hiç şüphesiz C programcılarının en sık karşılaştıkları hatalar göstericilerin yanlış kullanımları sonucunda ortaya çıkan "gösterici hataları" dır. Gösterici hatalarını yapmak kolay, ancak bulmak ve düzeltmek bazı durumlarda oldukça zor olabilmektedir. Nasıl yapılırsa yapılsın, gösterici hatalarının tek bir nedeni vardır: **Sistemden izin almadan tahsis edilmemiş bir bellek bölgesine veri aktarmak!** Çünkü belleğin istediğimiz bir bölgesine istediğimiz zaman birşey yazamayız. Veri aktardığımız bölgede önemli işlevlere sahip başka programlar olabilir. Bu programların bozulması da beklenmeyen sonuçların doğmasına yol açabilir. Aşağıdaki örneği inceleyiniz:



DOS altında çalıştığımızı düşünelim. Bu durumda *DOS*'un kendisi de bellektedir değil mi? Şekilde gösterilen *R1.EXE, R2.EXE* programları bellekte kalan (memory resident) programlar olsunlar. *PROG.EXE* ise bizim gösterici hatası yaptığımız kendi programımız olsun. *PROG.EXE* içerisinde yanlış bir biçimde kullanılan bir gösterici kendisini, *DOS*'u ya da *R1.EXE, R2.EXE* programlarından birini bozabilir. Başka programlarda yapılan bir byte değişiklik bile onların yanlış çalışmasına ya da kilitlenmesine neden olabilmektedir. Tabi, izinsiz veri aktarılan bölge zaten boş olan bir bölge de olabilir. Bu durumda hiçbir şey olmaz.

Gösterici hataları programın çalışma zamanına (run time) ilişkindir. Bu nedenle bazıları çok gizli bir biçimde ortaya çıkabilir. Bu bölümü en sık karşılaşılan gösterici hatalarına ayırdık.

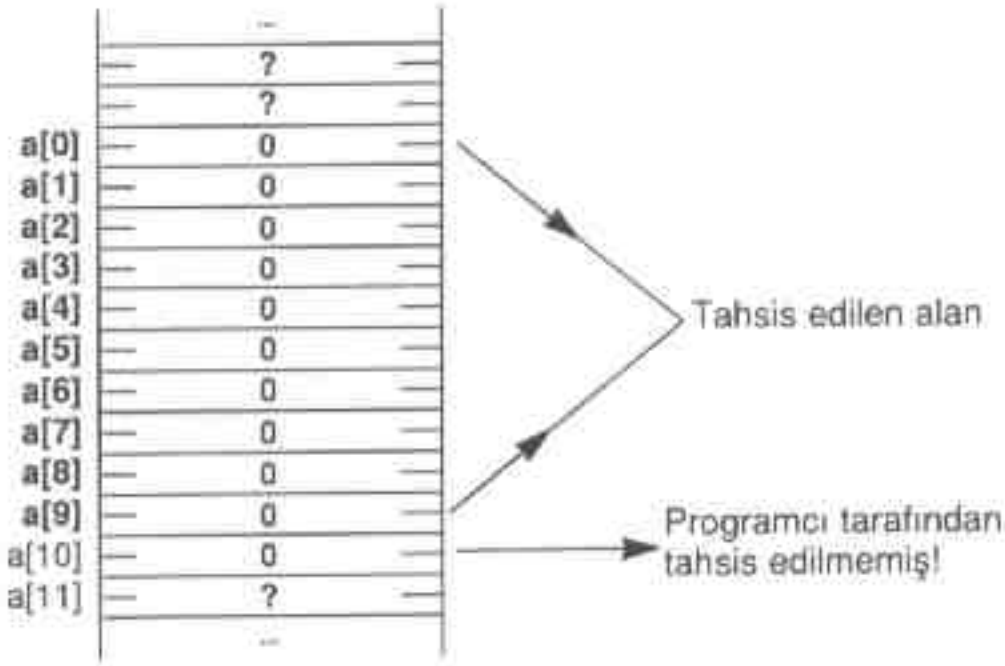
1) Dizi Taşmalarından Doğan Gösterici Hataları

Bir dizi için tahsis edilmiş alanın dışına veri atamasıyla yapılan gösterici hatalarıdır. C'nin seviyesi gereği dizi taşmalarının derleyici tarafından kontrol edilmediğini önceki bölümde belirtmiştik.

Aşağıdaki örneği inceleyiniz:

```
int a[10];
...
for (k = 0; k <= 10; ++k)
    a[k] = 0;
...
```

Döngü işlemi `k = 10` için de sürecektir. Oysa `a[10]` için tahsisat yapılmamıştır ve o bölgenin kim tarafından, ne amaçla kullanıldığı da bilinmemektedir. Derleyici `a` dizisi için `a[0]`'dan `a[9]`'a kadar toplam **10** elemanlık (**20** byte) yer ayırır. Oysa tahsis edilmemiş olan `a[10]` bölgesinin kim tarafından ve ne amaçla kullanıldığı hakkında herhangi bir bilgimiz yok. Aşağıdaki şekli inceleyiniz:



İndeks operatörü de bir gösterici operatörü olduğuna göre, bu operatörü kullanarak dizi için tahsis edilen alanın dışına atama yapılabileceğine dikkatinizi çekmek istiyoruz. Ayrıca, a dizisi için `a[-1]`, `a[-2]`, ... gibi ifadelerin de sintaks açısından geçerli olduğunu fakat buraya yapılacak atamaların gösterici hatalarına yol açacağını belirtelim.

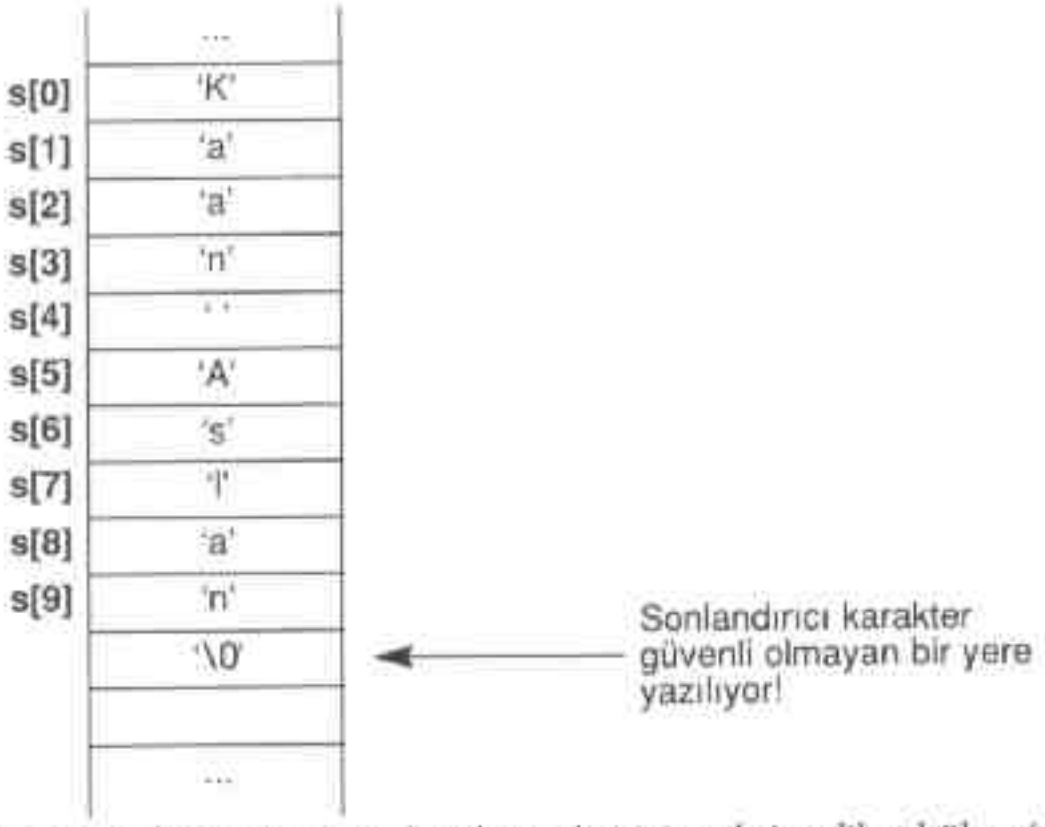
Bazen dizi taşmalarına gizli bir biçimde fonksiyonlar da neden olabilirler. Örneğin:

```
char s[10];
...
printf("Adı soyadı:");
gets(s);
```

ifadesinde s için toplam **10** karakterlik alan tahsis edilmiştir. `gets` fonksiyonunu anımsayınız; klavyeden girilen karakterleri diziye yerleştirdikten sonra sonuna `NULL` karakteri yerleştiriyordu. O halde yukarıdaki örnekte programın çalışması sırasında **9** karakterden fazla giriş yapılması gösterici hatasına neden olacaktır. Sonlandırıcı karakter de bir yer kaplar ve tahsis edilmiş bölge içerisinde olması gerekir. Örneğin, klavyeden girilen ad ve soyad:

```
Kaan Aslan
```

olsun. **gets** fonksiyonu karakterleri diziye aşağıdaki gibi yerleştirir:



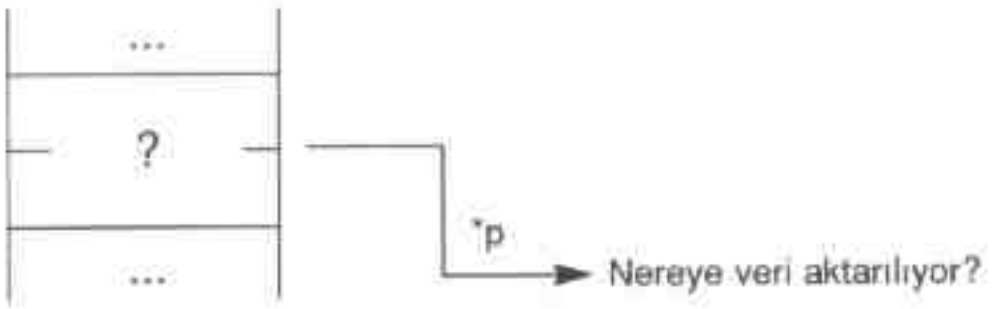
Gördüğünüz gibi sonlandırıcı karakter, dizi için tahsis edilen bölgenin dışına yerleştirilmiştir. Bu örnekte girdiğimiz isim daha uzun olsaydı tahsis edilmemiş olan bölgeye daha fazla karakter yazılacaktı. Bu tür hatalarla karşılaşmamak için dizinin yeteri kadar uzun olmasına dikkat etmelisiniz.

2) İlkdeğer Verilmemiş Göstericilerin Neden Olduğu Hatalar

İlkdeğer verilmemiş yerel değişkenlerin içerisinde rastgele değerlerin olduğunu anımsayınız. O halde, ilkdeğer verilmemiş olan yerel göstericilerin gösterdikleri adreslere veri aktarmak, belleğin rastgele bir bölgesine veri aktarmak anlamına gelir. Örneğin:

```
void main()
{
    char *p;          /* p yerel bir değişken olduğundan içerisinde
                         rastgele bir değer var */

    *p = 'a';         /* rastgele bir yere 'a' yazılıyor */
    ...
}
```

Burada rastgele bir yere veri aktarılmaktadır. Verinin aktarıldığı yerde işletim sisteminin, derleyicinin ya da bellekte kalan başka bir programın (memory resident) kodu bulunabilir. Hatta bu biçimde bir hata ile kendi programınızı da bozabilirsiniz.



İlkdeğer verilmemiş global değişkenlerin içerisinde sıfır olduğunu anımsayınız. Sıfır sayısı göstericilerde test amacıyla kullanıldığı için derleyicilerin çoğunda isteğe bağlı olarak bu hata çalışma zamanı sırasında kontrol edilmektedir. Örneğin aşağıdaki kodu çalıştırdığınızda:

```
char *p;

void main()
{
    *p = 'a';          /* NULL pointer assigment */
}
```

"NULL pointer assignment" biçiminde bir çalışma zamanı hatasıyla karşılaşabilirsiniz. Bu kontrol derleyicinin çalışabilen program içerisine yerleştirdiği "kontrol kodu" sayesinde yapılmaktadır.

80X86 Sembolik Makina Dili Programcısına Not: *C* derleyicileri NULL göstericiye atama yapılıp yapılmadığını çeşitli biçimlerde kontrol ederler. Ancak bu kontrol derleyicilerin çoğunda isteğe bağlı olarak (optional) yapılmaktadır. Göstericileri bu biçimde kontrol eden fonksiyonlar aslında derleyicilerin giriş modülleri (start up module) içerisinde bulunurlar.

İlkdeğer verilmemiş göstericilerin oluşturduğu hatalar daha dramatik biçimlerde de ortaya çıkabilir. Örneğin:

```
void main()
{
    char *p;

    gets(p);  /* gets ile girilen karakterler nereye yazılıyor? */
    ...
}
```

burada hata `gets` fonksiyonuna yaptırılmıştır. `gets` fonksiyonu klavyeden girilen karakterleri parametresi ile belirtilen adresten başlayarak yerleştirdiğine göre, bir önceki hataya benzer (hatta daha da kötüsü) bir hatayla karşı karşıyayız demektir!

3) Güvenli Olmayan İlkdeğerlerin Neden Olduğu Hatalar

Bir göstericiye herhangi bir ilkdeğer verilmesi onun güvenli bir bölgeyi gösterdiği anlamına gelmez. Örneğin:

```
char *p;
...
p =  (char *) 0XFC13;
*p = 'a';                          /* FC13 adresi güvenli mi? */
...
```

Burada göstericiye bir değer verilmiştir; ancak verilen bu değerin güvenli bir bölge olup olmadığı (bu bölgenin kullanılıp kullanılmadığı) belli değildir. Her ne kadar bellek alanı içerisinde belli amaçlarla kullanılan güvenli bölgeler varsa da `FC13` böyle bir bölgeyi göstermiyor.

80X86 Sembolik Makina Dili Programcısına Not: Sistemden izin alınmadan bellekte herhangi bir bölgeye veri aktarılması kimi mikroişlemcilerin koruma mekanizması (protection mechanism) sayesinde içsel olarak engellenmektedir Örneğin *80286, 80386 ve 80486* ve *Pentium* işlemcileri koruma mekanizmasına sahiptir. *80286, 80386 , 80486* ve *Pentium* işlemcilerinin koruma mekanizmasının kullanıldığı moduna korumalı mod (protected mode) denir. Korumalı modda çalışan mikroişlemciler veri aktarım komutlarını icra ederlerken gerekli kontrolleri kendi içlerinde yaparlar. Örneğin:

```
MOV  [ESI], EAX
```

komutunun icrası sırasında mikroişlemci, `[ESI]` bölgesinin güvenli olup olmadığını tespit ederek, ya komutu icra eder ya da etmez. Güvenli bölgeye yapılmadığı tespit edilen veri aktarımlarında *80286, 80386* ve *80486* ve *Pentium* işlemcileri bir çeşit kesme (exception) çağırarak işlemi bitirirler. Bu tür kesmelerin kodları (exception handlers) işletim sistemini yazanlar tarafından çoğu kez ilgili prosesi sonlandıracak biçimde tasarlanmaktadır.

Koruma (protection) mekanizması özellikle UNIX, WINDOWS 3.1, WINDOWS 95, OS/2 gibi çokişlemli (multiprocessing) sistemler için gereklidir. Fakat DOS temelde tekişlemli (singleprocessing) bir işletim sistemi olarak tasarlanmıştır. Yani DOS'ta çalışan bir programcı gösterici hatası yapsa da bundan yalnızca kendisi zarar görür. Koruma mekanizması olmasaydı çokişlemli sistemlerde bilinçsiz bir programcı başkasının kodunu bozabilirdi.

18.10 FONKSİYON PARAMETRELERİNDE GÖSTERİCİLERİN KULLANILMASI

Fonksiyon parametrelerinin kopyalanarak aktarıldığını daha önceki bölümde açıklamıştık. Tekrar anımsatmak istiyoruz. Parametre aktarım kuralına göre, bir fonksiyon çağrılmadan önce parametreler çağrılan fonksiyonun parametre değişkenlerine kopyalanarak geçirilirler.

```
...                          fonk (int n)
x = fonk (a);                {
...                              ...
                             }
```

Peki eğer parametre bir adres ise o zaman ne olur? Adres parametrelerinin kopyalandığı parametre değişkenlerinin de gösterici türünden olması gerekmez mi? Aşağıdaki şekli inceleyiniz:

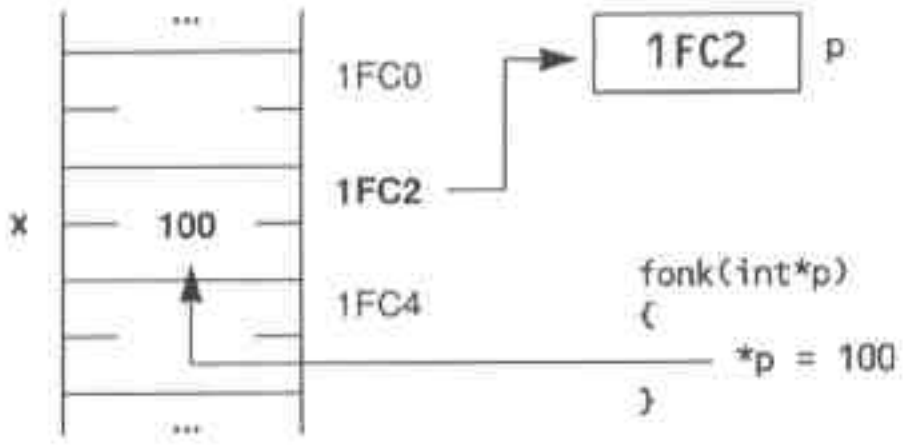
```
...                         fonk (int *p)
x = fonk (&a);              {
...                             ...
                            }
```

Parametre aktarım işlemi aslında bir çeşit atama işlemidir. Adresler ise gösterici-lere doğal olarak atanabilirler. Bir fonksiyonun parametresi gösterici ise bu fonksiyonun bir adres ile çağrılması gerektiğini düşünmelisiniz. Böyle çağırma biçimlerine **adres ile çağırma (call by reference)** denir.

Şimdi de bir fonksiyona parametre olarak adres gönderilmesinin anlamını araştıralım. Aşağıdaki programı inceleyiniz.

```
#include <stdio.h>
void sample (int *);          /*sample fonksiyonunun prototipi */
void main()
{
    int x;

    sample (&x);
    printf("%d\n", x);
}
void sample (int *p)
{
    *p = 100;
}
```

Bu örnekte x değişkeninin içerisindeki değer yerine, onun bellekte bulunduğu adres parametre olarak geçirilmiştir. Örneğimizde parametre olarak gönderilen adres int türünden olduğu için,bu adresi alan parametre değişkeni de int türünden bir gösterici biçimindedir. x değişkenin bellekte 1FC2 bölgesine yerleştiğini varsayalım.

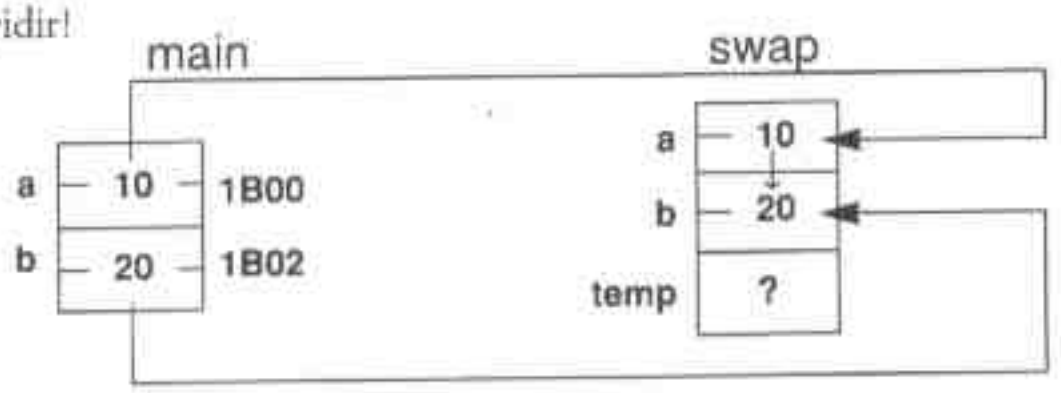


Örneğimizde p göstericisine x değişkeninin adresi olan 1FC2 kopyalandığına göre fonksiyon içerisinde *p , x değişkeninin kendisini göstermektedir. Buradan

şöyle bir sonuç çıkartabiliriz: **Bir yerel değişkenin içeriğini bir fonksiyonun değiştirebilmesi için, fonksiyona o yerel değişkenin adresi geçirilmelidir.** Benzer bir örnek daha vermek istiyoruz. İki yerel değişkenin içeriğini birbirleriyle yer değiştiren bir fonksiyon yazmak istediğinizi düşünelim. Önce aşağıdaki yanlış tasarımı inceleyiniz.

```
/* Yanlış tasarım */

#include <stdio.h>

void swap(int a, int b);

void main()
{
    int a = 10, b = 20;

    swap(a, b);
    printf("a = %d b = %d\n", a, b);
}

void swap(int a, int b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
}
```

Burada değiştirme işlemi yapılamaz. Çünkü içerikleri değiştirilen değişkenler `main` fonksiyonunun yerel değişkenleri değil, `swap` fonksiyonun parametre değişkenleridir!

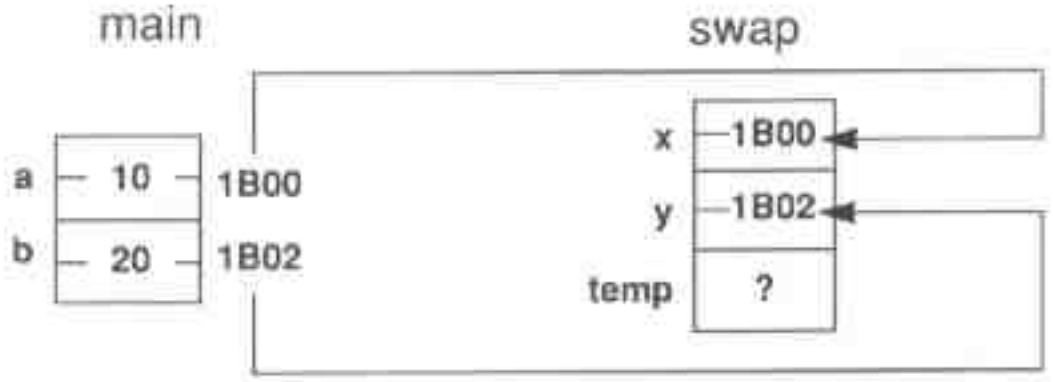


Oysa `swap` fonksiyonunun değiştirme işlemini doğru bir biçimde yapabilmesi için, `a` ve `b` değişkenlerinin içerisindeki değerlerin değil, adreslerinin parametre olarak geçirilmesi gerekirdi.

```
#include <stdio.h>

void swap(int *, int *);

void main()
{
```

```
    int a = 10, b = 20;

    swap(&a, &b);
    printf("a = %d b = %d\n", a, b);
}

void swap(int *x, int *y)
{
    int temp;

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

`main` fonksiyonunun içerisindeki yerel a ve b değişkenlerine ait adreslerin sırasıyla 1B00 ve 1B02 olduğu varsayalım Aşağıdaki şekli inceleyiniz:



a ve b değişkenlerinin adresleri `swap` fonksiyonuna kopyalandığında artık:

*x ► a
*y ► b

olmaktadır.

18.11 DİZİLERİN FONKSİYONLARA PARAMETRE YOLUYLA GEÇİRİLMESİ

Diziler konusunda, dizileri dizi yapan iki önemli özelliğin varlığından bahsetmiştik. Bunları yinelemek istiyoruz:

1) Dizi elemanlarının hepsi aynı türdendir.

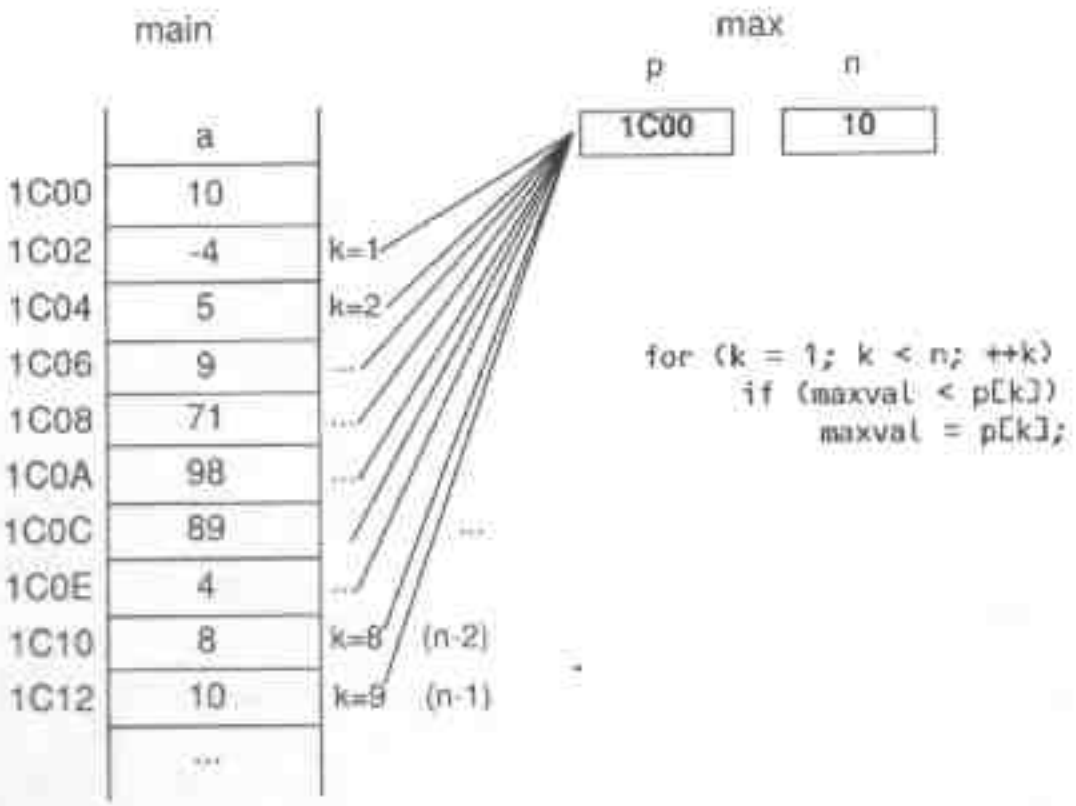
2) Dizi elemanları bellekte sürekli bir biçimde bulunurlar.

Bu durumda bir diziyi fonksiyona parametre yoluyla geçirmek için yalnızca o dizinin başlangıç adresini ve uzunluğunu geçirmek yeterlidir. Çünkü başlangıç adresini ve uzunluğunu alan bir fonksiyon, gösterici operatörleriyle dizinin tüm elemanlarına erişebilir.

Aşağıdaki örneği inceleyiniz.

```
#include <stdio.h>

int max(int *, int);          /* max fonksiyonunun prototipi */

void main()
{
    int a[10] = {10, -4, 5, 9, 71, 98, 89, 4, 8, 10};
    int m;

    m = max(a, 10);
    printf("En büyük sayı = %d\n", m);
}

int max(int *p, int n)
{
    int maxval;
    int k;

    maxval = p[0];
    for (k = 1; k < n; ++k)
        if (maxval < p[k])
            maxval = p[k];
    return maxval;
}
```

Bu örnekte **max** fonksiyonunu tek başına incelediğimizde, dizinin bellekteki başlangıcını gösteren **int** türünden bir adres ve dizinin uzunluğunu gösteren yine **int** türünden bir sayı aldığını görüyoruz. Örneğin **a** dizisinin **1C00**'dan başlayarak belleğe yerleştiğini varsayalım. Bu durumda bu adres **max** fonksiyonunun gösterici parametresine kopyalanacaktır. Aşağıdaki şekli inceleyiniz:



Örneğimizde a dizisinin adresi p isimli gösterici parametresine kopyalandıktan sonra, for döngüsü içerisinde indeks operatörü ile tüm elemanları taranmıştır.

```
...
for (k = 1; k < n; ++k)
    if (maxval < p[k])
        maxval = p[k];
...
```

Karakter dizilerinin fonksiyonlara geçirilmesi durumunda dizi uzunluğunu geçirmeye gerek yoktur. Çünkü dizinin NULL karakter ile bittiği herkes tarafından bilinmektedir.

Aşağıda bir karakter dizisini ekrana yazdıran putstr isimli fonksiyonu örnek olarak veriyoruz.

```
#include <stdio.h>

void putstr(char *str)
{
    while (*str != '\0') {
        putchar(*str);
        ++str;
    }
}
```

Döngü NULL karaktere kadar yineleniyor

Bu örnekte putstr fonksiyonuna main fonksiyonunda yerel olarak tanımlanmış s dizisinin başlangıç adresi geçirilmiştir. Dizi elemanları NULL karaktere kadar yazdırılmaktadır. Bu örneği gözönüne alarak standart puts fonksiyonunun da benzer biçimde tasarlandığını düşününüz. Fonksiyonu aşağıdaki kod ile test edebilirsiniz:

```
void main()
{
    char s[] = "İstanbul";

    putstr(s);
}
```

18.12 GERİ DÖNÜŞ DEĞERİ ADRES OLAN FONKSİYONLAR

Bir fonksiyon nasıl parametre olarak adres alabiliyorsa geri dönüş değeri olarak da adres verebilir. Geri dönüş değerinin adres olduğu, fonksiyon tanımlanırken * operatörü ile aşağıdaki gibi belirtilmektedir:

```
<adresin türü > * <fonksiyon ismi> ([parametreler])
{
    ...
}
```

Örneğin karakter türünden bir adrese geri dönen ve parametre almayan **fonk** isimli fonksiyon aşağıdaki gibi tanımlanır:

```
char * fonk (void)
{
    ...
}
```

Burada * adresi anlatıyor. Eğer geri dönüş ifadesinden * karakterini kaldırırsanız o zaman fonksiyonun geri dönüş değerinin **char** türünden bir adres değil yalnızca **char** olacağına dikkat ediniz.

Bir fonksiyon adresle geri döndüğüne göre bu fonksiyonun geri dönüş değeri aynı türden bir göstericiye atanmalıdır. Örneğin:

```
char *p;
...
p = fonk();
...
```

gibi.

Benzer biçimde **long** türünden bir adrese geri dönen **sample** fonksiyonunu da

```
long * sample(void)
{
    ...
}
```

biçiminde tanımlayabiliriz. Bu fonksiyon çağrıldıktan sonra da geri dönüş değerinin atanacağı değişken **long** türünden bir gösterici olmalıdır:

```
long *t;
...
t = sample();
...
```

Adrese geri dönen fonksiyonlara uygulamada sıkça karşılaşırız. Standart *C* fonksiyonlarının birçoğu adrese geri dönmektedir. Aşağıda klavyeden alınan karakterlerin başlangıç adresine geri dönen **getname** isimli fonksiyonu örnek olarak veriyoruz. Sonraki bölümde adrese geri dönen fonksiyonlar hakkında pekçok uygulama bulacaksınız.

```
#include <stdio.h>
char *getname(void)
{
    static char s[50];
    printf("Adı soyadı:");
```

```
    gets(s);
    return s;
}

void main()
{
    char *p;

    p = getname();
    printf("%s\n", p);
}
```

Örneğimizde **getname** fonksiyonu, içerisinde yerel olarak tanımlanmış olan `s` dizisinin başlangıç adresine geri dönmektedir. `s` dizisinin türü `char` ve fonksiyonun geri dönüş değeri de `char` türünden bir adres olduğuna göre bir tür uyuşumunun sağlandığına dikkat ediniz.

```
p = getname();
```

ile `s` dizisinin başlangıç adresi `p` göstericisine atanıyor. Ayrıca `s` dizisi `static` olarak alındığından fonksiyon sonlandıktan sonra güvenli bir biçimde bellekte kalacağını anımsatalım.

18.13 GÖSTERİCİLER NEDEN KULLANILIRLAR?

Hakkında bu kadar çok şey öğrendiğimiz göstericilerin ne işe yaradıklarını merak ediyorsunuzdur. Evet, nedir göstericilerin faydası?.

Göstericilerin iki önemli faydası vardır:

1) Belli bir adresteki bilgiye erişmek (veri ya da program kodu olabilir).

2) Bellekte sürekli bulunan yapıların yalnızca adreslerini geçirerek fonksiyonlara aktarmak.

Göstericiler sayesinde belleğin istediğimiz bir bölgesine erişebiliriz. İstediğimiz bir bellek bölgesine erişme işlemi sistem programlarının çoğunda yoğun bir biçimde kullanılmaktadır. Örneğin bellek içeriğini görüntüleyen **debug** programları, video belleğini doğrudan kullanan giriş çıkış fonksiyonları gibi. Bellekte sürekli bir biçimde bulunan yapıların yalnızca başlangıç adreslerinin geçirilmesi ile fonksiyonlara aktarılması ise göstericilerin en önemli kullanım nedenlerini oluştururlar. Örneğin 100 elemanlı bir dizinin, tüm elemanlarını tek tek bir fonksiyona kopyalamak yerine, dizinin başlangıç adresini ve uzunluğunu kopyalamak yeterlidir. Bellekte sürekli olarak bulunan tek veri yapısı dizi değildir, sonraki bölümlerde inceleyeceğimiz **yapılar (structures)** ve **birlikler (unions)** de bellekte sürekli bulunduklarından yalnızca başlangıç adresleri ile fonksiyonlara geçirilebilirler.

18.14 GÖSTERİCİLERE İLİŞKİN UYARILAR (Warning) VE HATALAR (Error)

Bir göstericiye ancak aynı türden bir adres bilgisi atanmalıdır. Bunun dışında örneğin atama işleminin sağ taraf ya da sol taraf değerlerinden birisi adres, diğeri başka türden olursa standart C derleyicileri buna bir uyarı mesajıyla karşılık verirler. Aşağıdaki örneği inceleyiniz:

```
int *p;
int x;
...
p = x;              /* uyarı */
...
x = p ;             /* uyarı */
```

Burada **int** türünden bir gösterici ile **int** türünden bir değişken arasındaki atama durumunu görüyorsunuz. Derleyiciler her iki atama durumunu da şüpheli bularak bir uyarı mesajı ile bildirirler. Sol taraf ve sağ taraf değerlerinin aynı türden adresler olduğu aşağıdaki örnekler normal bir biçimde değerlendirilirler.

```
char *p;
char  *t;
char s[20];
char ch;
...
p = s;                  /* Normal */
...
p = &ch;                /* Normal */
...
p = (char *) 0x1FC0;    /* Normal */
...
p = t;                  /* Normal */
```

Gösterici ile diğer türler arasındaki atamalarda uyarı mesajları bilinçli tür dönüştürmeleri ile engellenebilir:

```
char *p;
int x;
...
p = (char *) x ;
...
x = (int) p;
```

Derleyiciler uyarı mesajlarını programcının yapmış olabileceği olası yanlışlıklar için verirler. Ancak yukarıdaki örnekte bilinçli tür dönüştürmesi yapan programcı, bu işlemi yanlışlıkla değil, bilerek ve isteyerek yaptığını vurgulamaktadır.

Fonksiyon parametrelerinin kopyalanması da bir çeşit atama işlemi olduğuna göre, fonksiyonun parametre değişkeni ile kopyalanan parametre arasında da adres uyumu bulunmalıdır.

```
...
fonk(int *p)
{
    ...
}
void main(void)
{
    int x;
    ...
    fonk(x);
}
```

p = x
int * = int
?

Yukarıdaki örnekte **fonk** isimli fonksiyon karakter türünden gösterici yerine int bir değerle çağrılmıştır; aynı uyarı mesajı burada da söz konusudur.

Atama operatörünün iki tarafında da farklı türden adresler varsa bu durum derleyiciler tarafından başka bir uyarı mesajı ile programcıya bildirilir. Örneğin:

```
char *p;
int x[20];
...
p = x;              /* uyarı! p ile x adreslerinin türü aynı değil */
```

Yine bu durumda da bilinçli tür dönüştürmesi ile uyarı ortadan kaldırılabilir:

```
char *p;
int x[20];
...
p = (char *) x;  /* Normal! Bilinçli olarak yapılmış */
```

Not: C++'da yukarıda açıkladığımız iki uyumsuz durum da uyarı değil hata (error)'dır. **C++**'da yalnızca bir göstericiye aynı türden bir adresin atanmasına izin verilir.

18.15 void GÖSTERİCİLER

void göstericiler her hangi bir türe ait olmayan göstericilerdir. Bildirimleri **void** anahtar sözcüğü kullanılarak yapılır. Örneğin:

```
void *p;
void *adr;
...
```

gibi.

void göstericiler yalnızca adres saklamak amacıyla kullanılırlar. Bu yüzden **void göstericilerle diğer tür göstericiler arasında yapılan atamalarda uyarı söz konusu olmaz. void** göstericileri kullanırken bilinçli tür dönüştürmesi yapma zorunluluğu yoktur. Örneğin:

```
char *p;
void *v;

v = p;          /* Normal! Uyarı yok */
...
p = v;          /* Normal! Uyarı yok */
```

Benzer biçimde fonksiyon parametrelerinin kopyalanması sırasında da parametre değişkeni ya da kopyalanan parametre **void** ise uyarı söz konusu olmaz.

```
fonk (void *p)
{
    ...
}
void main()
{
    int *x;
    ...
    fonk(x);
}
void * = int *
```

p = x
uyarı yok!

```
fonk (int *p)
{
    ...
}
void main()
{
    void *x;
    ...
    fonk(x);
}
int * = void *
```

p = x
uyarı yok!

Belirli bir türe sahip olmadıklarından **void** göstericiler * ve [n] gösterici operatörleriyle kullanılamazlar. Çünkü bu operatörler nesneye erişmek için tür bilgisine de ihtiyaç duyarlar.

```
int x[100];
void *p;
...
p = x;          /* Normal */
...
*p = 100;       /* Hata! void gösterici * operatörüyle kullanılamaz */
...
p[10] = 100;    /* Hata! void gösterici [n] operatörü ile kullanılamaz */
```

Bir **void** gösterici artırılamaz ya da eksiltilemez. Çünkü bir göstericiyi 1 artırdığımızda içerisindeki adres göstericinin türünün uzunluğu kadar artmaktadır. Oysa **void** göstericinin türü yoktur.

```
void *p;
char s[20];
...
p = s;
...
++p;            /* Hata! void gösterici artırılamaz */
...
p = p - 2;      /* Hata! void gösterici eksiltilemez. */
```

18.15.1 void Göstericiler Neden Kullanılırlar?

`void` göstericiler adresleri geçici olarak saklamak amacıyla kullanılırlar. Diğer tür göstericiler arasındaki atama işlemlerinde uyarı ya da hata oluşturmadıklarından dolayı, türden bağımsız adres işlemlerinin yapıldığı fonksiyonlarda parametre değişkeni biçiminde de bulunabilirler. Örneğin:

```
void free (void *p)
{
     ...
}
```

`free` isimli fonksiyonun parametresi `void` gösterici olduğu için herhangi bir türden gösterici ile çağrılabilir; bu durumda uyarı ya da hata durumu söz konusu olmaz. `void` göstericiler aynı nedenden dolayı fonksiyonların geri dönüş değerini de oluşturabilirler. Örneğin:

```
void *malloc(unsigned size)
{
     ...
}
```

Burada `malloc` fonksiyonun geri dönüş değeri bir `void` göstericidir. Yani geri dönüş değeri herhangi türden bir göstericiye atanırsa uyarı ya da hata durumu söz konusu olmaz.

18.16 GÖSTERİCİLERİN BİLDİRİMİNDE YER VE TÜR BELİRLEYİCİLERİNİN KULLANILMASI

Gösterici bildirimlerinde yer belirleyici anahtar sözcüklerin (`auto, register, static, extern`) hepsi kullanılabilir. Yer belirleyicileri diğer türler için hangi işlevlere sahipse göstericiler için de aynı işlevlere sahiptir. Yine bildirimlerde tür belirleyici anahtar sözcükler (`const, volatile`) de kullanılabilir. Ancak göstericilerin `const` tür belirleyicisi ile bildirimi ayrı bir başlık halinde incelemeye değecek derecede karmaşıklık içermektedir.

18.16.1 Sabit Göstericiler (const pointers)

`const` tür belirleyicisi ile bildirilmiş olan göstericilere sabit göstericiler denir. `const` anahtar sözcüğünün bildirimdeki yerine bağlı olarak sabit göstericilerin anlamı değişmektedir. Sabit göstericilere ilişkin üç durum söz konusudur.

a) Gösterdikleri yer sabit olan sabit göstericiler

Bu tür sabit göstericilerde, adresin gösterdiği yer değiştirilemez; ancak göstericinin içeriği değiştirilebilir.

Bildirimleri aşağıdaki gibidir:

```
const <tür>  * <gösterici_ismi>;
```

Örnek bildirimler:

```
const char *s;
const int *p;
const float *f;
...
```

Daha açık bir ifadeyle p bir gösterici olmak üzere, *p ya da p[n] değiştirilemez, fakat p değiştirilebilir.

Örneğin:

```
char s[40];
const char *p;
....
p = s;          /*  Normal! Göstericinin içeriği değiştirilebilir */
...
p[n] = 'a';     /* Hata! göstericinin gösterdiği yer değiştirilemez */
...
```

Bu tür sabit göstericiler özellikle fonksiyon parametresi olarak karşımıza çıkarlar. Örneğin:

```
int strlen(const char *str)
{
     ....
}
```

Burada strlen fonksiyonunun parametresi olan str göstericisinin gösterdiği yerdeki bilgi, fonksiyon içerisinden değiştirilemez.

b) Kendisi sabit olan sabit göstericiler

Bu tür sabit göstericilerin gösterdikleri yer değiştirilebilir, ancak içerikleri değiştirilmez. Bildirimleri aşağıdaki gibi yapılır:

```
<tür> *const <gösterici_ismi>;
```

Örnek bildirimler:

```
char *const s;
int *const p;
float *const val;
...
```

Bu durumda p bir gösterici olmak üzere *p ya da p[n] değiştirilebilir, ancak p'ye ilk değer verildikten sonra bir daha değiştirilemez!

```
int x[10];
int y[10];
char const *p = x;
...

p[0] = 100;     /* Normal! Göstericinin gösterdiği
                yer değiştirilebilir */
p = y;          /* Hata! Göstericinin içeriği değiştirilemez! */
...
```

Kendisi sabit olan göstericilerin ilk değer verilerek kullanıldığına dikkat ediniz!

c) Hem kendisi hem de gösterdiği yer sabit olan sabit göstericiler

Bu tür sabit göstericilerde ise ne göstericinin kendisini ne de onun gösterdiği yeri değiştirebilirsiniz. Bildirimi iki `const` anahtar sözcüğü kullanılarak aşağıdaki biçimde yapılmaktadır.

```
const <tür> *const <gösterici_ismi>;
```

Örnek bildirimler:

```
const char *const p;
const int *const  x;
const double *const d;
...
```

Bu tür göstericiler tam bir koruma sağlamak amacıyla ilkdeğer verilerek kullanılırlar. Yani, `p` bir gösterici olmak üzere hem `*p` ya da `p[n]` hem de `p` korunmuştur, değiştirilemez.

```
int x[20];
int y[10]
const int *const p = x;
...

p = y;      /* Hata! Göstericinin içeriği değiştirilemez */
...
p[0] = 'a'; /* Hata! Göstericinin gösterdiği yer değiştirilemez */
```

Hem kendisi hem de gösterdikleri yer sabit olan göstericiler de uygulamada genellikle parametre değişkeni olarak karşımıza çıkarlar. Örneğin:

```
int sample(const char *const p)
{
    ...
}
```

gibi bir fonksiyonda parametre değişkeni olan `p` göstericisinin ne kendisi ne de

gösterdiği yer değiştirilebilir. p göstericisinin ilkdeğerini fonksiyon çağrılınca alacağına dikkat ediniz.

18.17 GÖSTERİCİLERE İLKDEĞER VERİLMESİ

Diğer türdeki değişkenlerde olduğu gibi göstericilere de ilkdeğer verilebilir. Göstericilere ilkdeğer verme işlemi aynı türden bir adresle yapılmalıdır. İlkdeğer verme işleminin genel biçimi aşağıdaki gibidir:

```
<tür> * <gösterici_ismi> = <ilkdeğer>;
```

Örneğin:

```
char s[20];
float x;
int *p = (int *) 0x1FC0;
char *str = (char *) 0x1C00;
char *t = s;
float *f = &x;
  ...
```

Yanlış anlaşılabilecek bir noktayı açmak istiyoruz: Bir göstericiye yukarıdaki biçimde ilkdeğer verildiğinde, verilen ilkdeğer göstericinin içerisine yazılır göstericinin gösterdiği yere değil!

Zaten verilen ilkdeğerin göstericinin gösterdiği yere yazılması açık bir gösterici hatası oluştururdu, değil mi?..

SORAMADIKLARINIZ...

S1) Derleyici gösterici hatalarını neden bulamıyor. Örneğin neden güvenli olmayan bir bölgeye yapılan atamalara müdahale etmiyor?

C1) Bir kere gösterici hataları programın çalışma zamanı sırasında ortaya çıktığından derleyicinin böyle bir kontrolü derleme aşamasında yapması mümkün değildir. Bu durumda derleyici olsa olsa gösterici hatasının yapılıp yapılmadığını çalışabilen kod içerisine yerleştirdiği birtakım kontrol kodları ile saptayabilir. Ancak böyle bir kontrol de C'nin felsefesine aykırıdır. Kontrol etmek yerine programcının gösterici hatası yapmayacak derecede bilinçli olması en etkin çözümdür.

Çokişlemli (multiprocessing) sistemlerde böyle bir korumaya gerçekten ihtiyaç olabilir. Ancak bu koruma durumu derleyici ya da işletim sistemi tarafından değil, mikroişlemci tarafından sağlanmaktadır.

S2) Gösterici hatası yapıldığında en kötü olasılıkla ne olabilir?

C2) Kişisel bilgisayarlarda *DOS* altında çalışıyorsanız gösterici hatası ile başınıza gelebilecek en kötü şey sabit diskinizin formatlanması olabilir. Ancak telaşlanmanıza gerek yok. Çünkü böyle bir olasılığın milyarda bir kadar bile olmadığını söyleyebiliriz. Gösterici hataları çoğunlukla bilgisayarın kilitlenmesi ya da programların beklenmeyen bir biçimde çalışmasına neden olurlar. Mikroişlemcilerin koruma mekanizmasına sahip olduğu çokişlemli sistemlerde gösterici hataları ile programcı ancak kendisine zarar verebilir. Sisteme ya da bellekte çalışmakta olan başka programa zarar veremez.

80X86 Sembolik Makina Dili Programcısına Not: Gösterici hatası ile sabit disk nasıl formatlanabilir, diye merak edebilirsiniz? Gösterici hatası bellekte tesadüfen program kodunun bulunduğu bölgeyi, diski formatlayan bir kesme oluşturacak biçimde bozmuş olabilir. Yukarıda da belirttiğimiz gibi bu olasılık dikkate alınmayacak derecede düşük bir olasılıktır. Bilgisayarın kilitlenmesinin nasıl olduğu da merak edilen bir konudur. Program kodunun yürüdüğü `CS:IP` programcının gösterici hatası ile kontolden çıkmış ve bir döngüye girmiş olabilir. Bu durumda kontrol işletim sistemine geri bırakılamaz. Ya da yolunu kaybeden `CS:IP` birtakım portlara değer göndererek kilitleme durumu da yaratabilir.

S3) Belleğin o andaki konumunu yani hangi programın belleğin neresinde bulunduğunu öğrenmeye yarayan bir sistem fonksiyonu var mıdır?

C3) Bellek işletim sisteminin kontrolü altındadır. Bellek kontrolü ve tahsisatının hangi yöntem ile yapıldığı işletim sisteminden işletim sistemine değişebilir. Fakat genellikle işletim sisteminin bu tür fonksiyonları dökümante edilmemiştir yani normal referans kitaplarında bulunmaz.

1 GÖSTERİCİ UYGULAMALARI

Bu bölümü göstericilere ilişkin uygulamalara ayırdık. Uygulama konularımızın çoğu standart *C* fonksiyonlarının yazımına ilişkindir. Amacımız uygulama yaparken aynı zamanda standart *C* fonksiyonlarının bir bölümünü de bu sayede öğrenmek. Yani bir taşla iki kuş vurmuş olacağız. Örnekleri tek tek incelemenizi ve bilgisayarınızda yazarak çalıştırmanızı salık veririz. Alıştırma amacıyla yazdığımız standart *C fonksiyonlarının başına* bir alt tire (underscore) karakteri koymayı uygun bulduk. Böylece yazdığınız örneklerde bu alt tire karakterini kaldırarak kütüphanede bulunan gerçeğiyle daha kolay karşılaştırma olanağı bulacaksınız.

19.1 KARAKTER DİZİLERİNİN UZUNLUĞUNUN BULUNMASI

`strlen` (string - length sözcüklerinden kısaltılmıştır) fonksiyonu karakter dizilerinin uzunluğunu bulan standart bir *C* fonksiyonudur. Prototipi `STRING.H` dosyası içerisindedir:

```
int strlen (char *str);
```

Yukarıdaki prototip ifadesinden de gördüğünüz gibi `strlen` fonksiyonu, karakter dizisinin başlangıç adresini parametre olarak alır. `strlen,` parametresi olan bu adresten başlayarak sonlandırıcı karakteri görene kadar tüm karakterlerin sayısını bulur. Aşağıdaki örneği inceleyiniz:

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

void main()
{
    char s[80];

    printf("Karakter Dizisi:");
    gets(s);            /* Klavyeden karakter dizisi okunuyor */
    printf("%d\n", strlen(s));      /* Uzunluğu yazdırılıyor */
}
```

Önce gets ile klavyeden karakter dizisi alınmıştır. Örneğin **İstanbul** karakterlerini girdiğimizi varsayalım:

Karakter Dizisi: İstanbul

strlen fonksiyonunun geri dönüş değerinin 8 olması gerekir. Şimdi strlen fonksiyonunu kendimiz yazalım, ne dersiniz? Bizim yazdığımızı vurgulamak amacıyla başına bir alt tire (underscore) karakteri koyuyoruz.

```
int _strlen (char *str)
{
    int n = 0;

    while (*str != '\0') {
        ++n;
        ++str;
    }
    return n;
}
```

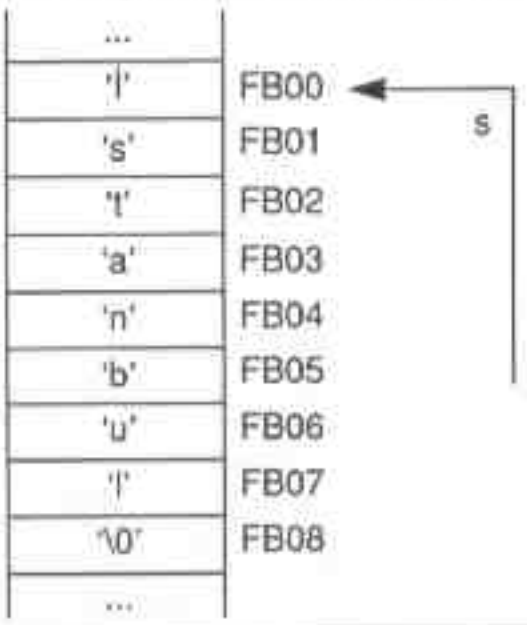
yazdığınız bu fonksiyonu kaynak koda ekledikten sonra main fonksiyonundan çağırarak denemelisiniz.

```
void main(void)
{
        char s[80];

        printf("Karakter Dizisi:");
        gets(s);
        printf("uzunluk = %d\n", _strlen(s));
}
```

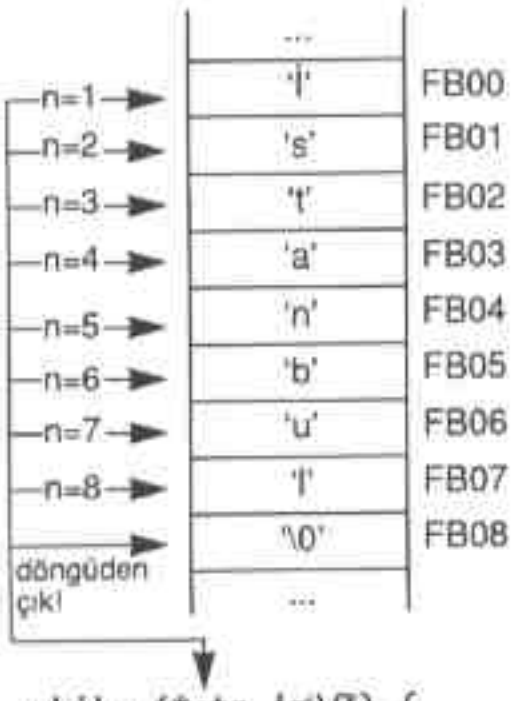
Algoritma nasıl çalışıyor acaba? Klavyeden **İstanbul** karakterlerini girmiş olalım:

Karakter Dizisi: İstanbul

gets fonksiyonu bu karakterleri **s** adresinden başlayarak tahsis edilmiş alan içerisine yerleştirecektir. **s** dizisi için tahsis edilen bölgenin **s** adresinden başladığını varsayalım.



s adresi fonksiyonunun gösterici parametresine kopyalanarak geçirilir.



`strlen` fonksiyonu daha değişik bir biçimde de düzenlenebilirdi. Örneğin **Ritchie** ve **Kernighan** "The C programming Language" isimli kitaplarında aşağıdaki gibi bir algoritma kullanmıştır.

```
int _strlen(char *str)
{
    int n;

    for (n = 0; str[n] != '\0'; ++n)
        ;
    return n;
}
```

Burada `n` değişkeni hem sayaç hem de indeksleme amacıyla kullanılmaktadır. `for` döngüsünün ikinci kısmını inceleyiniz. Bu ifade aşağıdaki gibi de kısaltılabilirdi:

```
for (n = 0; str[n]; ++n)
            ;
```

`NULL` karakter, sayısal olarak aynı zamanda sıfıra eşit olduğuna göre bu döngü de `NULL` karakteri gördüğünde sonlanır. Ancak siz daha okunabilir olması nedeni ile ilk biçimi tercih etmelisiniz.

19.2 KARAKTER DİZİSİ İÇERİSİNDE ARAMA

`strchr` (string - character sözcüklerinden kısaltılmıştır) fonksiyonu, bir karakter dizisi içerisinde herhangi bir karakteri aramak amacıyla kullanılan standart bir C fonksiyonudur. Prototipi `STRING.H` dosyası içerisindedir.

```
char *strchr(char *str, int ch);
```

Bu fonksiyon, parametresi olan `ch` karakterini `str` adresinden başlayarak `NULL` karakter görene kadar arar. Eğer bulursa bulduğu yerin adresiyle, bulamazsa **0** (NULL gösterici) değeriyle geri döner.

```
char *_strchr(char *str, int ch)
{
    while (*str != '\0') {
        if (*str == ch)
            return str;
        ++str;
    }
    if (ch == '\0')
        return str;
    return NULL;
}
```

Dizi, NULL karaktere kadar `while` döngüsü içerisinde taranıyor. Eğer,

```
if (*str == ch)
    return str;
...
```

ifadesiyle `ch` karakteri dizi içerisinde bulunursa, bulunduğu yerin adresi olan `str` değeriyle geri dönmektedir. `ch` karakterinin bulunamaması durumunda döngü `*str` değerinin `NULL` karaktere eşit olmasıyla sonlanır. Bu durumda aranan karakterin `NULL` karakter olup olmadığına da bakılmıştır. Çünkü `strchr` fonksiyonu ile `NULL` karakter de aranabilir!..

```
if (ch == '\0')                  → Aranılan karakter NULL karakteri mi?
    return str;                  → str NULL karakteri gösteriyor
return NULL;                     → Karakter bulunamadı
```

Yazdığınız fonksiyonu aşağıdaki kod ile deneyebilirsiniz:

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

void main()
{
    char s[20];
    char *p;
```

```
    gets(s);
    p = _strchr(s, 'a');
    if (p)
        printf("Bulundu:%s\n", p);
    else
        printf("Bulunamadı\n");
}
```

Örnekte klavyeden bir string alınmıştır. Eğer string içerisinde 'a' karakteri varsa, 'a' karakterinin bulunduğu yerden dizinin sonuna kadar olan tüm karakterler ekrana yazılır. `_strchr` yerine `strchr` yazarak kütüphanede bulunan orijinalini çalıştırınız. İşlevleri aynı değil mi?..

19.3 BİR KARAKTER DİZİSİNİN BAŞKA BİR KARAKTER DİZİSİNE KOPYALANMASI

`strcpy` (string - copy sözcüklerinden kısaltılmıştır) bir adresten başlayarak NULL karakter görene kadar başka bir adrese kopyalama yapan standart C fonksiyonudur. Tüm string fonksiyonlarında olduğu gibi prototipi `STRING.H` içerisindedir.

```
char *strcpy(char *dest, char *source);
```

`strcpy` fonksiyonu kopyalamanın yapıldığı adresin (dest) kendisine geri döner. `strcpy` NULL karakteri de kopyalamaktadır.

```
char *_strcpy(char *dest, char *source)
{
    int k;
    for (k = 0; (dest[k] = source[k]) != '\0'; ++k)
        ;
    return dest;
}
```

`for` döngüsünü inceleyiniz. Önce atama yapılmış, daha sonra NULL karakter ile karşılaştırılmıştır. Böylece NULL karaktere gelindiğinde önce NULL karakter kopyalanacak ve ondan sonra döngü sonlanacaktır.

Yukarıdaki örnekte indeks operatörü kullanıldığından göstericilerin değeri değişmez. Eğer aynı örnek göstericileri artırarak yapılmak istenseydi, o zaman geri dönüş adresi olan **dest** 'in saklanması gerekirdi. İnceleyiniz:

```
char *_strcpy(char *dest, char *source)
{
    char *temp = dest;
    while ((*source++ = *dest++) != '\0')
        ;
    return temp;
}
```

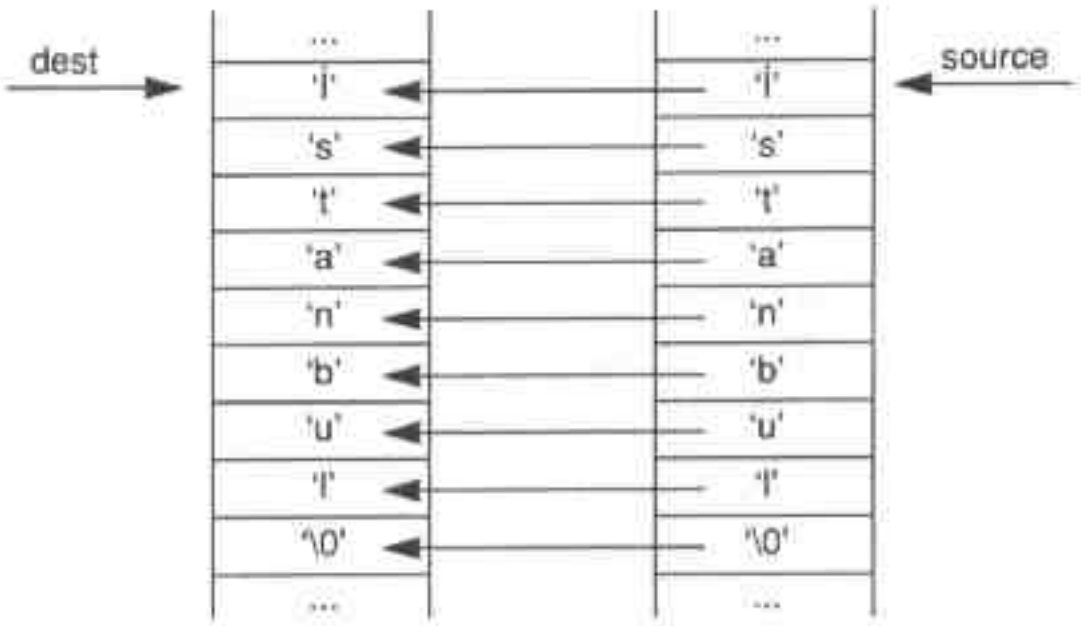
Bu örnekte `while` döngüsü içerisindeki ifadeye dikkat etmelisiniz. Göstericiler sonek durumunda artırılıyorlar. Bu durumda artırma atama işleminden daha sonra yapılır değil mi?

İ1: *dest
İ2: *source
İ3: İ2 = İ1 ► *source = *dest
İ4: İ3 != '\0' ► (*source = *dest) != '\0'
İ5: dest = dest + 1
İ6: source = source + 1

Denemeyi aşağıdaki kod ile yapabilirsiniz.

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

void main()
{
    char source[] = "İstanbul";
    char dest[10];

    _strcpy(dest, source);
    printf("%s\n", dest);
}
```

`_strcpy` yerine `strcpy` yazdıktan sonra orijinalini de deneyerek eşdeğer olduğuna dikkat etmelisiniz. Algoritmanın çalışmasını aşağıdaki şekil özetliyor:



Bir noktaya dikkat ediniz: Eğer `dest` adresi ile tahsis edilmiş olan alan yeteri kadar uzun değilse bir gösterici hatasına maruz kalabilirsiniz. Hiçbir C fonksiyonu bir bölgenin tahsis edilip edilmediğini kendi içerisinde kontrol etmez.

19.4 BİR KARAKTER DİZİSİ İÇERİSİNDE TÜM KARAKTERLERİN KÜÇÜK HARFE YA DA BÜYÜK HARFE DÖNÜŞTÜRÜLMESİ

`toupper` ve `tolower` fonksiyonlarını anımsayınız. Bu fonksiyonlar yalnızca tek bir karakteri büyük harf ya da küçük harfe dönüştürüyorlardı. Oysa `strupr` (string - **upper** case sözcüklerinden) ve `strlwr` (string - **lower** case sözcüklerinden) fonksiyonları, `NULL` karakter görene kadar bir dizinin tüm karakterlerini dönüştürmektedir. Prototiplerini inceleyiniz:

```
char *strupr(char *s);
char *strlwr(char *s);
```

Her iki fonksiyonun da prototipleri `STRING.H` dosyası içinde bildirilmiştir. Geri dönüş değerlerinin karakteri gösteren bir adres olduğuna dikkat ediniz. `strupr` ve `strlwr` fonksiyonları parametreleri olan karakter dizisinin başlangıç adreslerine geri dönerler. Yani geri dönüş değerleri parametreleriyle aynıdır.

```
char *_strupr (char *str)
{
    int k;

    for (k = 0; str[k] != '\0'; ++k)
        if (str[k] > = 'a' && str[k] <= 'z')
            str[k] = str[k] - 'a' + 'A';
    return str;
}
```

`for` döngüsü içerisinde tüm karakterler eğer küçük harf ise büyük harfe çeviriliyorlar. Bunun için daha önce öğrenmiş olduğumuz `islower` ve `toupper` fonksiyonlarını da kullanabilirsiniz. (Bu iki fonksiyon aslında birer makrodur; bu durumda `CTYPE.H` dosyasını mutlaka kaynak koda dahil etmelisiniz!).

```
...
for (k = 0; str[k] != '\0'; ++k)
    if (islower(str[k])
        str[k] = toupper(str[k]);
....
```

`strlwr` fonksiyonunu da benzer biçimde siz yazabilirsiniz. Aşağıda bir deneme kodu veriyoruz.

```
#include <stdio.h>
#include <string.h>
void main()
{
    char s[20];

    gets(s);
    _strupr(s);
    printf("%s\n", s);
}
```

19.5 KARAKTER DİZİLERİNİN KARŞILAŞTIRILMASI

İki karakter dizisinin karşılaştırılması demek *ASCII* tablosu gözönünde bulundurularak karakter dizilerinin öncelik ilişkisini belirlemek demektir.

Örneğin:

"Ali" dizisi **"Alm"** dizisinden daha küçüktür . Çünkü eşitliği bozan **'m'** karakteri **ASCII** tablosunda 'i' karakterinden daha sonra gelir. Benzer biçimde:

- **"Ali"** dizisi **"ALI"** dizisinden daha büyüktür. Çünkü küçük harfler **ASCII** tablosunda büyük harflerden daha sonra gelir.

- **"Example"** dizisi **"Exam"** dizisinden daha büyüktür.

- **"A123"** dizisi ile **"A123"** dizisi birbirine eşittir.

İki karakter dizisini karşılaştıran `strcmp` (**str**ing - **comp**are sözcüklerinden kısaltılmıştır) fonksiyonunun prototipi aşağıdaki gibidir; inceleyiniz:

```
int strcmp (char *s1, char *s2);
```

Bu fonksiyon eğer : Birinci karakter dizisi (s1) ikinci karakter dizisinden (s2) büyükse pozitif bir değere, küçükse negatif bir değere ve eğer iki karakter dizisi birbirine eşitse sıfır değerine geri döner. Bu durumu sembolik olarak aşağıdaki biçimde gösterebiliriz.

```
s1 > s2 ise +
s1 < s2 ise -
s1 == s2 ise 0
```

Bu sembolik gösterimde göstericilerin sayısal değerlerinin değil onların gösterdikleri yerdeki karakterlerin karşılaştırıldığını vurgulamak istiyoruz. Fonksiyonun tasarımını inceleyiniz:

```
int  _strcmp (char *s1, char *s2)
{
    while (*s1 == *s2) {
        if (*s1 == '\0')
            return 0;
        ++s1;
        ++s2;
    }
    return *s1 - *s2;
}
```

Çözümlemesi oldukça kolay. `while` döngüsü ile iki dizinin karşılıklı karakterleri eşit olduğu sürece tarama sürdürülüyor. `*s1 == *s2` eşit olduğunda, `*s1 == '\0'` ifadesinin doğru olması `*s2`'nin de `NULL` olduğu ve işlemin bittiği anlamına gelir. Bu durumda iki dizi birbirine eşittir ve sıfır ile geri dönülmüştür. Peki, döngüden çıkış nasıl oluyor? Eşit olmayan ilk karakter döngüyü sonlandırır,

değil mi? Bu durumda *s1 ve *s2 ilk eşit olmayan karakterler olduğuna göre: *s1 - *s2 ifadesi s1 dizisi daha büyükken pozitif ve daha küçükken ise negatif bir değer oluşturur. Aynı algoritma for çevirimi içerisinde indeks operatörü ile de yapılabilirdi:

```
....
for (k = 0; s1[k] == s2[k]; ++k)
    if (s1[k] == '\0')
        return 0;
return s1[k] - s2[k];
...
```

String karşılaştırma işlemi C programlarında oldukça sık karşımıza çıkar. Aşağıda, strcmp fonksiyonu için için bir deneme örneği veriyoruz.

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

void main()
{
    char s[20];
    char password[] = "strcmp";

    printf("Şifre Giriniz:");
    gets(s);
    if (!_strcmp(password, s))
        printf("Doğru şifre\n");
    else
        printf("Yanlış şifre\n");
}
```

_strcmp yerine strcmp yazdıktan sonra kütüphanedeki orijinalini de çalıştırınız.

19.6 BİR KARAKTER DİZİSİNİN SONUNA BAŞKA BİR KARAKTER DİZİSİNİN EKLENMESİ

strcat (string - concatanate sözcüklerinden kısaltılmıştır) bir karakter dizisinin sonuna başka bir karakter dizisini ekleyen standart C fonksiyonudur. Tüm string fonksiyonlarında olduğu gibi prototipi STRING.H dosyası içerisindedir.

```
char *strcat (char *s1, char *s2);
```

stcat fonksiyonu s1 dizisinin sonuna NULL karakteri ezerek s2 dizisini kopyalar. Yani işlem sonucunda s1 dizisi s2 dizisi kadar daha büyümektedir. Ayrıca ekleme işleminin sonunda s1 dizisine NULL karakter de yine eklenir. Örneğin:

```
char s1[20] = "Ankara";
char s2[] = "İstanbul"
...
strcat(s1, s2);
```

işlemi ile s1

"Ankaraİstanbul"

biçimine dönüşür. Aşağıdaki tasarımı inceleyiniz:

```
char *_strcat(char*s1, char*s2)
{
    char *temp = s1;
    while (*s1 != '\0')
        ++s1;
    while ((*s1++ = *s2++) != '\0')
        ;
    return temp;
}
```

s1 dizisinin sonu bulunuyor

strcpy (s1,s2)

Bu algoritmada önce **while** döngüsü ile önce **s1** dizisinin sonu bulunmuştur.

```
while (*s1 != '\0')
    ++s1;
```

Döngüden çıkıldığında **s1** adresi **NULL** karakteri gösterir. Artık gerisi **s2** adresinden **s1** adresine yapılan bir kopyalama işlemidir. Bunu **strcpy** fonksiyonu ile de gerçekleştirebiliriz. **s1** adresi değiştirildiği için en başta **temp** isimli başka bir göstericide saklanmıştır. İşlem sonucunda **s1** dizisinin sonuna NULL karakterin konulduğuna dikkat ediniz. Uygulamada **s1** dizisinin **s2** dizisini içine alabilecek derecede uzun olarak açılması gerekir. Aksi taktirde ortaya çıkacak gösterici hatası beklenmeyen sonuçlar doğurabilir. Yazdığınız fonksiyonu aşağıdaki kod ile deneyebilirsiniz:

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

void main()
{
    char s1[20];
    char s2[40];

    printf("s1:");
```

```
    gets(s1);
    printf("s2:");
    gets(s2);
    _strcat(s1, s2);
    printf("s1:%s\n", s1);
}
```

`_strcat` yerine **`strcat`** yazarak kütüphanede bulunan orijinalini deneyerek karşılaştırınız.

19.7 KARAKTER DİZİSİNİN TERS ÇEVRİLMESİ

`strrev` (**str**ing - **rev**erse sözcüklerinden kısaltılmıştır) bir karakter dizisini ters-yüz eden standart *C* fonksiyonudur. Örneğin:

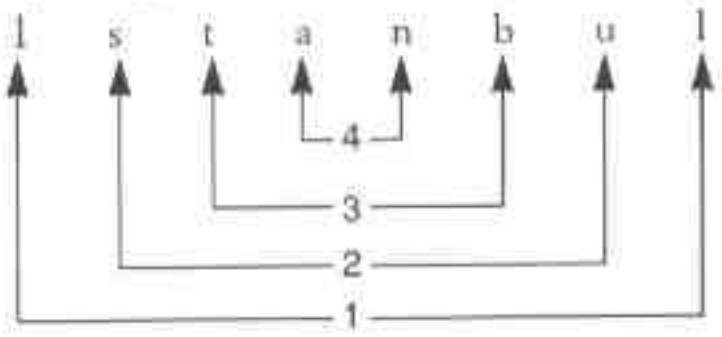
```
char s[] = "İstanbul";
...
strrev(s);
```

ile `s` dizisi:

```
lubnatsİ
```

biçimine dönüştürülür.

`strrev` fonksiyonunu nasıl tasarlayacağız? En açık yol, baştaki elemanlarla sondakileri karşılıklı yer değiştirmektir.



Baştaki ve sondaki elemanların karşılıklı yer değiştirmeleri dizinin uzunluğunun yarısı kadar yapılmalıdır. Yani `strlen(str) / 2` kadar. Dizinin eleman sayısı tek ise ortadaki elemanın yer değiştirmeyeceği açıktır.

```
char * _strrev(char *str)
{
    int n, k, temp;

    n = strlen(str);
    for (k = 0; k < n / 2; ++k) {
        temp = str[k];
        str[k] = str[n-k-1];
        str[n-k-1] = temp;
    }
    return str;
}
```

Yazdığınız fonksiyonu aşağıdaki koda ekleyerek deneyebilirsiniz.

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

void main()
{
    char s[80];

    gets(s);
    printf("%s\n", _strrev(s));
}
```

`_strrev` yerine `strrev` yazarak kütüphanedeki orijinalini de çalıştırınız.

19.8 KARAKTER DİZİLERİNİN HERHANGİ BİR KARAKTERLE DOLDURULMASI

`strset` (string- **set** sözcüklerinden kısaltılmıştır) bir karakter dizisini herhangi bir karakterle dolduran standart C fonksiyonudur. Prototipi `STRING.H` dosyasındadır.

```
char *strset (char *str, int ch)
```

`strset` fonksiyonunun geri dönüş değeri doldurulan karakter dizisinin (burada `str`) başlangıç adresidir. `strset` fonksiyonunun tasarımının diğerlerinden daha kolay olduğunu söyleyebiliriz. Tek yapılacak şey `str` dizisine sırasıyla NULL karakter görene kadar `ch` karakterini atamak.

```
char *_strset(char *str, int ch)
{
    int k;

    for (k = 0; str[k] != '\0'; ++k)
        str[k] = ch;
    return str;
}
```

Fonksiyonu kaynak koda ekleyerek aşağıdaki örnekle test edebilirsiniz.

```
#include <stdio.h>

void main()
{
    char s[] = "İstanbul";

    printf("%s\n", _strset(s, 'x'));
}
```

Ekranda **xxxxxxx** görmeniz gerekir. `_strset` yerine `strset` yazarak kütüphanedeki orijinali ile karşılaştırınız.

19.9 BİR KARAKTER DİZİSİNİN İLK N KARAKTERİNİN BAŞKA BİR KARAKTER DİZİSİNE KOPYALANMASI

`strncpy` (**str**ing - **n**umber - **cpy** sözcüklerinden kısaltılmıştır) bir karakter dizisinin ilk `n` karakterini başka bir karakter dizisine kopyalayan standart C fonksiyonudur. Prototipini inceleyiniz:

```
char *strncpy (char *dest, char *source, int n);
```

`strncpy` fonksiyonu `source` adresiyle belirtilen dizinin ilk `n` karakterini `dest` adresiyle belirtilen diziye kopyalar. `strncpy` fonksiyonu eğer:

```
n <= strlen(source)
```

ise NULL karakteri dizinin sonuna eklemez. Fakat eğer:

```
n > strlen(source)
```

ise NULL karakteri kopyanın yapıldığı dizinin sonuna ekler. Örneğin:

```
char s1[] = "İstanbul";
char s2[15] = "Ankara";
...
strncpy(s2, s1, 3);
printf("%s\n", s2);
...
```

ile ekranda:

```
İstara
```

görmeniz gerekir. `3 < strlen(s1)` olduğuna dikkat ediniz. `strncpy` bu durumda NULL karakteri dizinin sonuna eklememiştir. `strncpy` fonksiyonu özellikle bir dizinin belli bir bölümünün başka bir diziyle değiştirilmesi gerektiği durumlarda kullanılır. Geri dönüş değeri yine kopyalamanın yapıldığı diziyi gösteren `dest` adresidir.

```
char * _strncpy(char *dest, char *source, int n)
{
    int k;

    for (k = 0; k < n && source[k] != '\0'; ++k)
        dest[k] = source[k];
    if (n > k)
        dest[k] = '\0';
    return dest;
}
```

`for` döngüsünün devam etmesi için hem `k < n` hem de `source[k] != '\0'` koşulunun sağlanması gerekir.

`n > strlen(source)` için NULL karakterin kopyalandığına dikkat ediniz. Yazdığınız fonksiyonu aşağıdaki koda dahil ederek deneyebilirsiniz:

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

void main()
{
    char s1[] = "İstanbul";
    char s2[50] = "Ankara";

    _strncpy(s2, s1, 8 );
    printf("%s\n", s2);
}
```

`_strncpy` yerine `strncpy` yazarak kütüphanedeki orjinali ile karşılaştırınız.

19.10 İKİ KARAKTER DİZİSİNİN İLK N KARAKTERİNİN KARŞILAŞTIRILMASI

`strncmp` (string-number-compare sözcüklerinden kısaltılmıştır) iki karakter dizisinin ilk n karakterini karşılaştıran standart C fonksiyonudur. Prototipini inceleyiniz:

```
int strncmp (char *s1, char *s2, int n);
```

`strncmp` fonksiyonu, birinci karakter dizisinin ilk n karakteri, ikinci dizinin ilk n karakterinden büyükse pozitif bir değere, küçükse negatif bir değere ve iki dizinin ilk n karakteri birbirine eşitse 0 değerine geri döner. Bu durumu sembolik olarak:

s1 > s2 ise +
s1 < s2 ise -
s1 == s2 ise 0

biçiminde gösterebiliriz. Bu durumda örneğin:

```
char s1[] = "Bugün hava çok güzel";
char s2[] = "Bugün kırlara gitmek istiyorum";
...
```

`strncmp(s1, s2, 5)` fonksiyonu sıfır değerine geri dönecektir. Dolayısıyla,

```
if (!strncmp(s1, s2, 5))
    ...
```

ifadesi Doğru olarak değerlendirilir. Oysa `s1` ve `s2` dizilerinin bütünsel olarak birbirinden farklı olduğuna dikkat ediniz. Yani,

```
if (!strcmp(s1, s2))
    ...
```

Yanlış olarak değerlendirilir. `strncmp` fonksiyonunda da karşılaştırma işlemi NULL karaktere kadar yapılmaktadır. Eğer `n > strlen(s1)` ya da `n > strlen(s2)` ise karşılaştırma işlemi ilk NULL karakter görülünce biter. Bu durumda aşağıdaki örnekte `strncmp` fonksiyonu negatif bir değere geri dönecektir:

```
char s1[] = "İstan"
char s2[] = "İstanbul";
...
strncmp(s1, s2, 50)
...
```

Şimdi de fonksiyonun tasarımı üzerinde duralım.

```
int _strncmp (char *s1, char *s2, int n)
{
    int k;

    for (k = 0; k < n - 1 && s1[k] == s2[k]; ++k)
        if (s1[k] == '\0')
            return 0;
    return s1[k] - s2[k];
}
```

`for` döngüsü `k < n -1 ve a1[k] == s2[k]` koşullarının sağlandığı sürece yinelenir. İki dizinin aynı anda NULL karakter ile sonlanıp sonlanmadığı `if` deyimi ile kontrol edilmiştir.

19.11 BİR KARAKTER DİZİSİNİN SONUNA BAŞKA BİR KARAKTER DİZİSİNİN İLK N KARAKTERİNİN EKLENMESİ

`strncat` (string-number-concatanate sözcüklerinden kısaltılmıştır) bir karakter dizisinin sonuna başka bir karakter dizisinin ilk `n` karakterini ekleyen standart C fonksiyonudur. Prototipi diğerlerinde olduğu gibi `STRING.H` dosyası içerisindedir.

```
char *strncat(char *s1, char *s2, int n);
```

`strncat` fonksiyonunun geri dönüş değeri eklemenin yapıldığı dizinin başlangıç adresi (`s1`) dir.

```
char* _strncat(char*s1, char*s2, int n)
{
    int k;
    char *temp = s1;

    while (*s1! = '\0')   <-- s1 dizisinin sonu bulunuyor
        ++s1;
```

```
    for (k = 0; k < n && s2[k] != '\0'; ++k)
        *s1++ = *s2++;
    *s1 = '\0';
    return temp;
```

s2 dizisinin ilk n karakteri s1 dizisinin sonuna kopyalanıyor

Uygulamamızda s1 dizisinin sonu bulunduktan sonra for çevirimi içerisinde s2 dizisinin ilk n karakteri kopyalanmaktadır. Fonksiyonu aşağıdaki örnek ile deneyebilirsiniz:

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

main()
{
    char s1[20] = "An";
    char s2[] = "kara kutu";

    strncat(s1, s2, 4);
    printf("%s\n", s1);
}
```

19.12 SEÇEREK SIRALAMA YÖNTEMİ (Selection Sort)

Diziler bölümünde kabarcık sıralaması (bouble sort) yöntemini açıklamış ve bir örnek program vermiştik. Şimdi de seçerek sıralama (selection sort) yöntemi üzerinde durmak istiyoruz. Seçerek sıralama yöntemi kısaca "en büyüğünü bul başa koy" cümlesiyle özetlenebilir.

Aşağıda, n elemanlı bir diziyi seçerek sıralama yöntemiyle büyükten küçüğe doğru sıralayan **ssort** isimli fonksiyon örnek olarak verilmiştir, inceleyiniz:

```
#include <stdio.h>

void ssort(int *p, int n)
{
    int k, l;
    int max, i;

    for (k = 0; k < n; ++k) {
        max = p[k];
        i = k;
        for (l = k + 1; l < n; ++l)
            if (p[l] > max) {
                max = p[l];
                i = l;
            }
        p[i] = p[k];
        p[k] = max;
    }
}
```

Algoritmada içiçe iki döngü kullanılıyor. İçteki döngünün dıştaki döngüden bir fazla değerle başladığına dikkat ediniz:

```
...
max = p[k]
i = k;
for (l = k + 1; l < n; ++l)
    if (p[l] > max) {
        max = p[l];
        i = l;
    }
...
```

(k,n) alt dizisinin en büyük elemanı ve indisi bulunuyor

`(k, n)` alt dizisinin en büyük elemanı ve onun indisi bulunarak sırasıyla `max` ve i değişkenlerinde tutulmaktadır. Bu alt dizinin ilk elemanı olan `p[k]` iç döngüye girilmeden önce,

```
max = p[k] ;
i = k;
```

ile en büyük eleman varsayılmıştır. Dışarıdaki döngü her defasında alt diziyi daraltmaktadır. Nihayet, `(k, n)` alt dizisinin en büyük elemanı bulunduktan sonra ilk eleman ile yer değiştirilmiştir.

```
p[i] = p[k];
p[k] = max;
```

SORAMADIKLARINIZ...

S1) Kütüphane içerisinde bulunan bir fonksiyon aynı zamanda programcı tarafından da tanımlanmış ise hangisi geçerlidir? Örneğin kendi tasarladığımız bir fonksiyona kütüphanede bulunan bir fonksiyonun ismini versek, bu fonksiyon çağrıldığında bizim yazdığımız mı, yoksa kütüphanede bulunan mı çağrılır?

C2) Bir fonksiyon hem kaynak kod içerisinde hem de kütüphanede bulunuyorsa kaynak kod içerisinde bulunan geçerlidir. Örneğin `strcpy` fonksiyonunu kaynak program içerisinde tekrar aynı isimle yazsak, bu durumda bizim yazdığımız fonksiyon çağrılır, kütüphanede bulunan değil!..

80X86 Sembolik Makina Dili Programcısına Not: Bir fonksiyon kaynak program içerisinde tanımlanmışsa C derleyicileri onu object modül içerisine EXTRN olarak yazmazlar. Dolayısıyla bağlayıcı programın (linker) bu fonksiyonu kütüphane içerisinde araması da söz konusu değildir. Yine, birden fazla modülle çalışıldığı durumlarda kütüphane içerisinde bulunan bir fonksiyon aynı zamanda modüllerden birinde de tanımlanmış ise, bu fonksiyonun başka bir modülden çağrılması durumu sizi çatışmaya sevk edebilir. Ancak böyle durumlarda da bağlayıcı program arama işlemini önce modüllerde yapacağından yine modül içerisinde tanımlanan fonksiyon çalışabilen koda yazılacaktır.

20 STRINGLER

Stringler *C* öğrencilerinin en çok zorluk çektiği konulardan bir tanesidir. Bu nedenle ayrı bir bölüm içerisinde ve ayrıntılı bir biçimde incelemeyi uygun gördük.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Stringler derleyici tarafından nasıl ele alınır?

2) Stringler ile göstericiler arasındaki ilişki nasıldır?

3) Stringlerin ömürleri nasıldır?

20.1 STRING NEDİR?

C programlarında "iki tırnak içerisindeki ifadelere" string ifadeleri ya da kısaca stringler denir. Örneğin:

```
"İstanbul"
"sayı = %d\n"
"Lütfen bir tuşa basınız..."
...
```

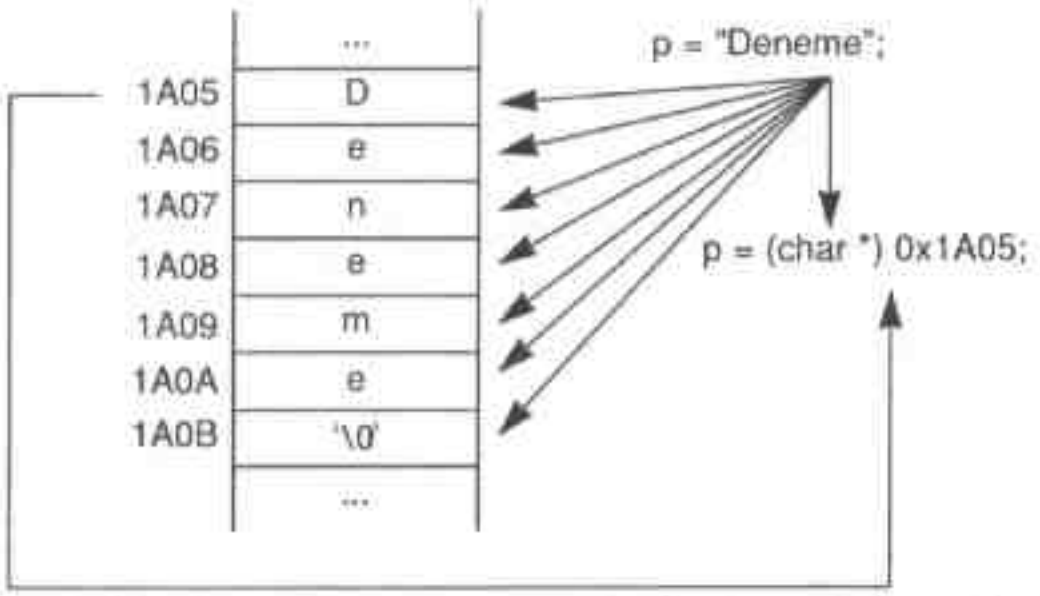
ifadeleri birer stringtir.

Stringlerin tek bir atom olarak ele alındığını giriş bölümünden anımsıyorsunuzdur. C'de stringler aslında karakteri gösteren birer adrestir. **C derleyicileri, derleme aşamasında bir stringle karşılaştığında, önce onu belleğin güvenli bir bölgesine yerleştirir, sonuna NULL karakteri ekler ve daha sonra string yerine yerleştirdiği yerin başlangıç adresini koyar.** Bu durumda string ifadeleri aslında stringlerin bellekteki başlangıç yerini gösteren karakter türünden birer adrestir.

Örneğin:

```
char *p;
...
p = "Deneme";
```

gibi bir kodun derlenmesi sırasında, derleyici önce `"Deneme"` stringini belleğin güvenli bir bölgesine yerleştirir; daha sonra yerleştirdiği yerin başlangıç adresini string ifadesi ile değiştirir.



80X86 Sembolik Makina Dili Programcısına Not: Stringler object modül içerisinde *DATA* segment bölgesine yerleştirilir. Bilindiği gibi, *Borland* ve *Microsoft* derleyicilerinde bu segment *_DATA* ve *_BSS* ismiyle bulunmaktadır. Stringler object modül içerisine **PUBLIC** olarak yazılmazlar.

String ifadeleri karakter türünden göstericilere atanmalıdır. String ifadeleri dizi isimlerine atanamaz. Aşağıdaki örneği inceleyiniz:

```
char s[20];
...
s = "İstanbul";
```

Bu ifade `s` bir nesne olmadığı için geçersizdir. Ancak bu durumu dizilere iki tırnak içerisinde ilkdeğer verme işlemiyle karıştırmayınız!..

```
char s[10] = "İstanbul";
```

Çünkü dizilere ilkdeğer verme işleminde derleyici önce diziyi belirtilen uzunlukta açar, daha sonra iki tırnak içerisindeki ifadeleri -`NULL` karakter dahil olmak üzere- dizi elemanlarına sırasıyla yerleştirir. Yani dizilere ilkdeğer verme işleminde derleyici, string ifadelerinde olduğu gibi bir adres yerleştirmez.

20.2 STRINGLERİN FONKSİYON PARAMETRESİ OLARAK KULLANILMASI

Stringlerin fonksiyonlara parametre olarak geçirilmesi durumuna sıklıkla rastlanır. Aslında bu gibi durumlarda fonksiyona parametre olarak karakteri gösteren bir adres geçirilmektedir.

Örneğin:

```
puts("Merhaba");
```

burada derleyici "`Merhaba`" stringini belleğe yerleştirip sonuna NULL karakteri koyduktan sonra, `puts` fonksiyonunun parametresi de artık karakteri gösteren bir adres halini alır. `puts` fonksiyonunun parametresi olan adresten başlayarak

NULL karakteri görene kadar tüm karakterleri ekrana yazdığını anımsayınız. Bu durumda ekranda,

```
Merhaba
```

yazısını görmeniz gerekir, değil mi?

Aşağıdaki örneği inceleyiniz:

```
char str[20];
...
strcpy(str, "İstanbul");
```

bu örnekte de `"İstanbul"` stringi `str` adresinden başlayarak kopyalanmaktadır. String ifadelerinin bulunduğu yerde karakteri gösteren bir adres düşünmelisiniz. Şimdi aşağıdaki ifadeyi yorumlayalım:

```
strcpy("Ankara", "İstanbul");
```

Derleyici derleme aşamasında iki stringi de bellekte güvenli bir bölgeye yerleştirir. Çalışma zamanı sırasında `strcpy` fonksiyonu `"İstanbul"` stringini gösteren adresten başlayarak `"Ankara"` stringini gösteren adrese NULL karakter görene kadar kopyalama yapacağına göre, bu ifadenin hiçbir anlamı yoktur! Üstelik bu ifade, ikinci string birinciden uzun olduğu için bir gösterici hatasına da neden olur. Şimdi aşağıdaki ifadeyi inceleyiniz:

```
p = "Ankara" + "İstanbul";
```

bu örnekte ne yapılmak istenmiştir dersiniz? `"Ankara"` stringinin başlangıç adresi ile `"İstanbul"` stringinin başlangıç adresi toplanıyor değil mi? Peki bu toplamın yararlı bir sonucu olabilir mi? Derleyiciler -anlamsızlığından dolayı- iki gösterici toplamına izin vermezler. Fakat bir göstericiden başka bir göstericiyi çıkarmanın yararlı sonuçları olabilir. Bu yüzden göstericilerin birbirinden çıkartılması derleyicilerin hepsinde geçerli bir işlemdir.

C derleyicileri kaynak kodun çeşitli yerlerinde tamamen özdeş stringlere rastlasa bile bunlar için farklı yerler ayırabilirler.

Örneğin bir programın içerisinde:

```
printf("Bellek yetersiz...\n");
...
printf("Bellek yetersiz...\n");
...
printf("Bellek yetersiz...\n");
...
```

farklı yerlerde `"Bellek yetersiz...\n"` gibi özdeş stringler bulunsa bile derleyici bunlar için tekrar ve farklı yerler ayırabilir. Bu da büyük uygulamalar için belleğin verimsiz kullanılmasına yol açabilir.

Yer ayırma işleminin derleme aşamasında yapıldığına dikkat ediniz. Aşağıdaki örnekte ekrana hep aynı adres mi yoksa farklı farklı adresler mi basılır, dersiniz?

```
...
char *p;
int k;
...
for (k = 0; k < 10; ++k) {
    p = "Deneme";
    printf("%p\n", p );
}
...
```

Evet, bu örnekte ekrana hep aynı adres basılır. Eğer yer ayırma işlemi çalışma zamanı sırasında yapılsaydı, o zaman farklı adresler basılabilirdi...

20.3 STRINGLERİN ÖMÜRLERİ

Stringler statik ömürlü nesnelerdir. Tıpkı global değişkenler gibi programın yüklenmesiyle yaratılırlar, programın icrası bittiğinde de bellekten silinirler. Dolayısıyla stringler çalışabilen kodu büyütürler. Birçok sistemde statik verilerin toplam uzunluğunda belli bir sınırlama söz konusudur. Örneğin, *DOS* altında bu sınırlama bellek modeline bağlı olarak değişmektedir.

80X86 Sembolik Makina Dili Programcısına Not: *DOS* altında toplam statik veri miktarı (global değişkenler, statik yerel değişkenler ve string ifadeleri) bellek modeline göre değişir. *Tiny, Small* ve *Medium, Compact* ve *Large* modellerde toplam statik veri 64K'yı geçemez. Çünkü statik değişkenler aynı grup içerisinde önceden belirlenmiş isimlerdeki segmentlere yazılırlar. Örneğin *Borland* derleyicilerinde bu bellek modellerinde statik veriler, *DGROUP* grubuna atanmış olan *_DATA* ve *_BSS* segmentlerine yazılmaktadır. Oysa *Huge* modelde statik verilerin yazılacağı segmentler, program ismine bağlı olarak değişirler. Böylece, *Huge* modelde static veriler aynı kaynak kod içerisinde 64K'yı geçemez, ancak toplam olarak 64K'yı geçebilir.

20.4 STRINGLERİN BİRLEŞTİRİLMESİ

Stringlerin tek bir atom olarak ele alındığını giriş bölümlerinden anımsıyorsunuzdur. Bu durumda bir stringi aşağıdaki gibi parçalayamayız:

```
char *p;
...
p = "Bu gün hava
        çok güzel";
```

Ancak string ifadeleri büyüdükçe bunu tek bir satırda yazmak hem sıkıntı yaratmakta hem de okunabilirliği bozmaktadır. Uzun stringlerin parçalanmasına olanak vermek amacıyla derleyiciler yanyana yazılan string ifadelerini birleştirirler.

Örneğin:

```
p = "Bugün hava "
    "çok güzel";
```

geçerli bir ifadedir. Bu durumda iki string ifadesi birleştirilecek ve aşağıdaki biçime getirilecektir.

```
p = "Bugün hava çok güzel";
```

İki string arasında hiçbir operatörün bulunmadığına dikkat ediniz:

```
p = "Ankara," " İstanbul";
```

ifadesi ile

```
p = "Ankara, İstanbul";
```

ifadesi eşdeğerdir.

Birleştirmenin yanı sıra tek bir ters bölü (back-slash) ile satır sonlandırılarak sonraki satıra geçiş sağlanabilir. Örneğin:

```
p = "Bugün \"
hava çok güzel";
```

ifadesi ile:

```
p = "Bugün hava çok güzel;
```

ifadesi eşdeğerdir. Ters bölü işaretinden sonra stringin aşağıdaki satırın başından itibaren devam ettiğine dikkat ediniz.

Örneğin:

```
p = "Bugün hava\
      çok güzel";
```

ifadesi aşağıdakiyle eşdeğerdir:

```
p = "Bugün hava      çok güzel";
```

Ancak ters bölü karakteri ile sonraki satırın başından devam etme standart olarak her C derleyicisinde geçerli olmayabilir. Çatışmalı bir durumla karşılaştığınızda çalıştığınız derleyicilerin referans kitaplarına başvurmalısınız.

20.5 STRINGLERDE TERS BÖLÜ KARAKTERLERİNİN KULLANILMASI

Stringler içerisinde ters bölü karakter sabitleri de kullanılabilir. Derleyiciler stringler içerisinde bir ters bölü karakteri gördüklerinde, onu yanındaki karakter ile birlikte tek bir karakter olarak ele alırlar.

Örneğin:

```
p="Adı\nSoyadı"
```

benzer biçimde:

```
p = "Adı\tSoyadı";
```

ifadesinde \t tek bir karakterdir (tab karakter).

20.6 STRINGLERLE GÖSTERİCİLERE İLKDEĞER VERİLMESİ

Stringler kullanılarak doğrudan göstericilere ilkdeğer verilebilir. Örneğin:

```
char *p = "İstanbul";
char *err = "Bellek yetersiz";
char *s = "Devam etmek için bir tuşa basınız";
...
```

String ifadeleri aslında karakteri gösteren birer adres olduğuna göre ilkdeğer verilen göstericilerin de karakter türünden göstericiler olması gerekir. İki tırnak içerisinde dizilere ilkdeğer vermeyle göstericilere ilkdeğer verme arasındaki ayrıma tekrar dikkatinizi çekmek istiyoruz.

```
char *p = "Deneme";
char s[10] = "Deneme";
```

Göstericilere ilkdeğer verildiğinde derleyici bunu bir string ifadesi olarak ele almaktadır. Yani string belleğe yerleştirildikten sonra başlangıç adresi göstericiye atanır. Oysa dizilerde önce dizi açılır, daha sonra karakterler tek tek dizi elemanlarına yerleştirilir. Dizilere ilkdeğer verirken kullandığımız iki tırnak ifadeleri adres belirtmezler.

SORAMADIKLARINIZ...

S1) Bir program içerisindeki toplam statik veri, sistem tarafından belirlenen sınırı aşarsa hata hangi aşamada ve nasıl ortaya çıkar? Örneğin, *DOS* altında Large modelde 64K olan statik veri sınırının aşılması durumunda hata nasıl ve kim tarafından tespit edilir?

C1) Statik veri sınırı eğer bir modülde aşılmış ise bu durumda hata derleyici tarafından farkedilir. Oysa birden fazla modülün toplamı biçiminde bir sınır aşımı söz konusuysa bu durumda hata bu modülleri birleştiren bağlayıcı (linker) program tarafından tespit edilecektir.

S2) String ifadeleri içerisinde çift tırnak (") ya da ters bölü karakterinin kendisini (\) kullanabilir miyiz?

C2) String ifadelerinde doğrudan çift tırnak ya da ters bölü karakterini kullanamazsınız; çünkü bildiğiniz gibi bu karakterlerin başka işlevleri vardır. Ancak bu karakterler başına bir ters bölü konularak ters bölü karakter sabitleri olarak ifade edilebilirler. Bu durumda string ifadeleri içerisinde:

" yerine \"

\ yerine \\

yazabilirsiniz. Örneğin:

```
p = "\"İstanbul\"";
```

gibi bir string ifadesi geçerlidir. Bu durumda;

```
puts(p);
```

ile ekrana **"İstanbul"** yazısı basılacaktır.

21 YAKIN, UZAK VE DEV GÖSTERİCİLER

80X86 mikroişlemcilerini kullanan 16 bit işletim sistemlerinde göstericiler **yakın, uzak** ve **dev** olmak üzere 3 kısma ayrılırlar. Bu nedenle **yakın, uzak** ve **dev** göstericiler yalnızca **80X86** mimarisinde çalışan derleyiciler ve özellikle de *DOS* işletim sistemi için anlamlı ve geçerlidir. Dolayısıyla bu bölüm standart ve taşınabilir bir bilgi içermemektedir. Başka sistemlerde çalışan okuyucularımız isterlerse bu bölümü atlayabilirler. Ancak biz ne olursa olsun bu bölümü dikkatli bir biçimde okumanızı -bakış açınızı genişleteceği düşüncesiyle- salık veririz.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) **80X86** mikroişlemcisinin gerçek moddaki adresleme biçimi nasıldır?

2) **Segment** ve **offset** kavramları ne anlam ifade etmektedir?

3) **Yakın, uzak** ve **dev** göstericilerin bildirimleri nasıl yapılır?

4) **Yakın, uzak** ve **dev** göstericilerin uzunlukları ne kadardır?

5) Varsayılan gösterici türü nasıl belirlenir?

6) Uzak göstericiler neden kullanılırlar, faydaları nelerdir?

7) Uzak göstericilere nasıl değer aktarılır?

8) Uzak göstericiler arasında karşılaştırma işlemleri nasıl yapılır?

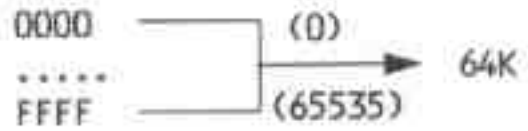
9) Uzak ve dev göstericilerin artırılması ve eksiltilmesi işlemlerini derleyiciler nasıl ele alırlar?

10) Bellek modeli nedir, kaç tane bellek modeli vardır?

11) Bellek modelinin göstericiler için önemi nedir?

21.1 GİRİŞ

18. Bölümde göstericilerin *8086* mimarisinde 2 byte ya da 4 byte yer kapladığını belirtmiştik. Fakat verdiğimiz örneklerde göstericileri hep 2 byte olarak varsaydık. Oysa 2 byte göstericilerle bellekte ancak 64K uzunluğunda bir yer adreslenebilir. Çünkü 2 byte (16 bit) ile ifade edilebilecek en büyük sayı 64K'dır.



Oysa *8086* mikroişlemcisinin (*80286-80386-80486, Pentium* real mode) 1MB adres alanına sahip olduğunu anımsayınız. O halde şu sonucu çıkarabiliriz:

2 byte göstericilerle 1MB içerisinde yalnızca 64K uzunluğunda bir bölge adreslenebilir.

Peki, **1MB** içerisinde herhangi bir yere erişebilmek için göstericilerin ne kadar uzunlukta olması gerekir dersiniz? **1MB** yani **1024K** bellek en az **20** bit (5 HEX digit) yani **2.5** byte ile adreslenebilir.

2^{20} = 1,048,576 = 1024K

Oysa, **1MB** içerisinde adresleme yapan göstericiler 2,5 byte değil 4 byte'lik göstericilerdir. Konunun daha iyi anlaşılması için *80X86* ailesinin gerçek moddaki adresleme biçiminin incelenmesi gerekir.

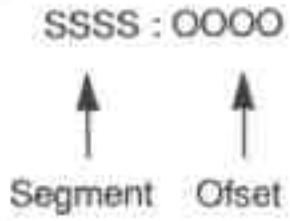
21.2 80X86 AİLESİNİN GERÇEK MODDAKİ ADRESLEME BİÇİMİ

80X86 ailesinin gerçek moddaki adreslemesi **2 byte segment** ve **2 byte offset** bilgisiyle yapılmaktadır. Mikroişlemci **2 byte segment** ve **2 byte offset** değerlerini kendi içerisinde 20 bitlik (5 *HEX* digit) doğrusal adrese dönüştürür. Yani, *80X86* işlemcileri doğrusal adresi 20 bit halinde değil de iki ayrı 2 byte halinde alırlar. Segment ve offset çiftinden 20 bitlik doğrusal adres ise şu biçimde elde edilir:

1. Adım: Segment değeri 16 ile çarpılır. (16 ile çarpma *HEX* sistemde sayının sağına bir tane sıfır ekleme anlamına gelir.)

2. Adım: Bu değer Offset ile toplanır.

Segment ve offset değerleri genellikle aralarına ':' karakteri konularak gösterilirler.



Örneğin:

`1C05 : 1F13` değerlerinden oluşan **segment** ve **offset** çiftinden mikroişlemcinin elde edeceği doğrusal adresi araştıralım.

1. Adım: Segment **16** ile çarpılır. Bu *HEX* sistemde sayının sağına **0** koymak demektir.

`1C050`

2. Adım: Elde edilen değer offset ile toplanır.

```
  1C050
+  1F13
-------
  1DF63
```

Mikroişlemci, `1C05:1F13` segment offset çiftinden `1DF63` doğrusal adresini elde etmektedir. Çıkan sonucun 5 *HEX* digit yani 20 bit olduğuna dikkat ediniz. Başka bir örnek:

`1D7A : 232C` çiftinden elde edilecek doğrusal adres nedir?

Segment değerinin sağına `0` konur offset ile toplanırsa:

```
  1D7A0
+  232C
-------
  1FACC
```

`1FACC` doğrusal adresi elde edilir.

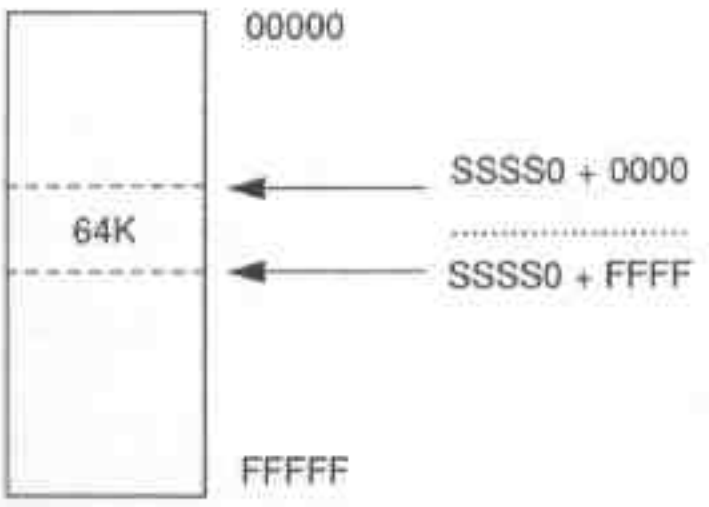
Şimdi de tersten giderek, örneğin `1FC45` doğrusal adresine ilişkin `segment : offset` çiftlerini araştıralım.

```
1FC4 : 0005
1FC3 : 0015
1FC2 : 0025
...     ...
```

Yukarıda da gördüğünüz gibi birden fazla `segment : offset` çifti aynı doğrusal adrese ilişkin olabilmektedir. Peki, ilginç bir deneye ne dersiniz? Segment kısmını sabit tutmak üzere offset değerini mümkün olduğunca değiştirelim:

```
SSSS : 0000
...     ...
SSSS : FFFF
```

Bu durumda segment ile gösterilen bellek bölgesinden ancak 64K uzaklaşabiliriz. Şekli inceleyiniz:



21.3 near, far ve huge ANAHTAR SÖZCÜKLERİ

80X86 mimarisinde gösterici tanımlamakta kullandığımız üç anahtar sözcük vardır: `near`, `far` ve `huge`. Bu anahtar sözcükler gösterici bildirimlerinde aşağıdaki gibi kullanılırlar:

```
<tür bildiren anahtar sözcük>   [ near | far | huge ]   * <değişken ismi>
```

yukarıdaki sintaks biçiminden de gördüğünüz gibi `near`, `far` ve `huge` anahtar sözcükleri bildirim yapılırken kullanılmayabilir. Gerçekten de, önceki bölümlerde göstericilerin bildirimlerini bu anahtar sözcükleri kullanmadan yapmıştık.

Örneğin:

```
char far *p;   /* far anahtar sözcüğü kullanılmış */
int near *t;   /* near anahtar sözcüğü kullanılmış */
char huge *h;  /* huge anahtar sözcüğü kullanılmış */
float *f;      /* herhangi birisi kullanılmamış */
...
```

`near`, `far` ve `huge` anahtar sözcükleri Standart *C*'de yoktur. Çünkü bu anahtar sözcükler *80X86* sistemlerindeki göstericiler için anlam taşır. Dolayısıyla `near` ve `far` anahtar sözcükleri donanım bağımlıdır ve taşınabilir değildir.

21.4 YAKIN GÖSTERİCİLER (near Pointers)

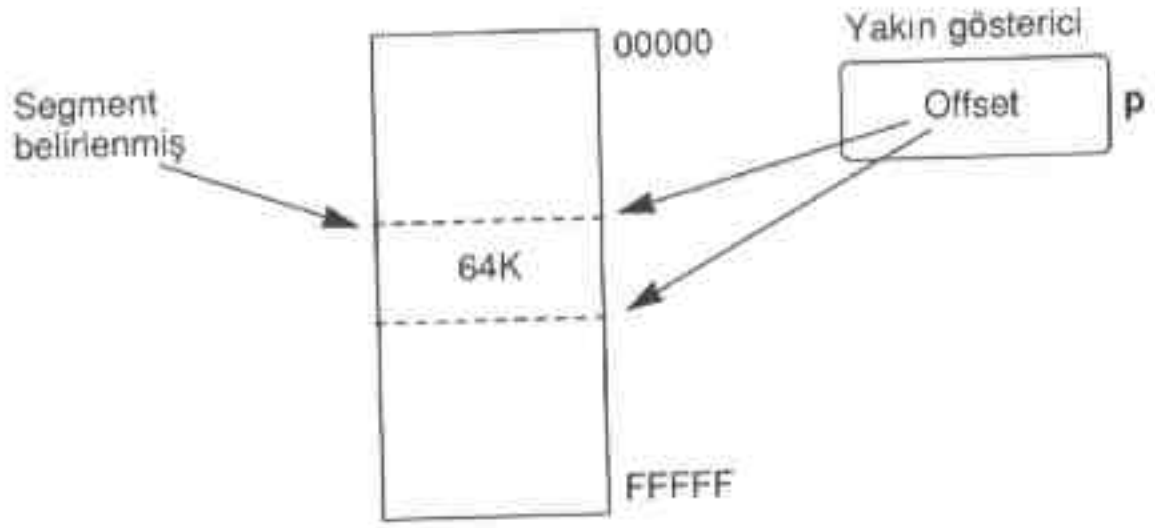
80X86 mimarisinde 2 byte uzunluğunda olan göstericilere **yakın göstericiler (near pointer)** denir. Yakın göstericiler yalnızca offset içeren göstericilerdir. Bunların segment değerleri sistem tarafından programın yüklenmesi sırasında belirlenmiştir. Yakın göstericiler `near` anahtar sözcüğü kullanılarak bildirilirler. Örneğin:

```
char near *s;
int near *p;
float near *f;
...
```

Bu durumda: `sizeof(s)`, `sizeof(p)`, `sizeof(f)`, `sizeof(char near *)`, `sizeof(int near *)`,... ifadelerinin hepsi 2 değerini üretir.

Yakın göstericilerle 1MB bellek alanı içerisinde istediğimiz bir yere ulaşamayız. Yalnızca sistem tarafından belirlenmiş olan segment değerinden 64K ilerleyebiliriz.

Örneğin:



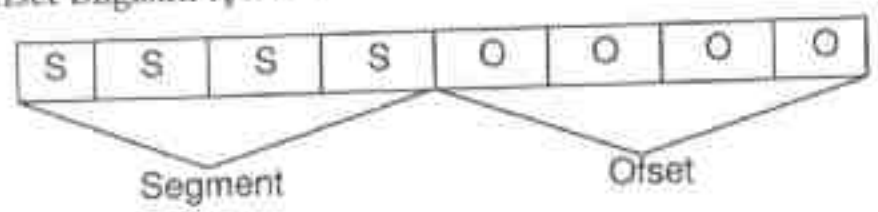
21.5 UZAK GÖSTERİCİLER (far Pointers)

Uzak göstericiler hem **segment** hem de **offset** içeren 4 byte uzunluğundaki göstericilerdir. Bildirimleri **far** anahtar sözcüğü kullanılarak yapılır. Örneğin:

```
char far *scrp;
int far *p;
float far *t;
...
```

Bu durumda: **sizeof(scrp), sizeof(p), sizeof(t), sizeof(char far *), sizeof(int far *),...** ifadeleri 4 değerini üretecektir.

Bir uzak göstericinin yüksek anlamlı 2 byte'ı segment, düşük anlamlı 2 byte'ı offset bilgisini içerir .

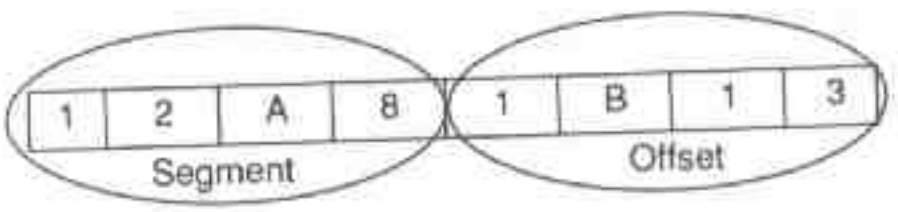


Uzak göstericilere **long** ya da **int** sabitleri üzerinde bilinçli tür dönüştürmesi yaparak değer verebiliriz.

Örneğin:

```
char far *p;
...
p =  (char far *) 0x12A81B13;
```

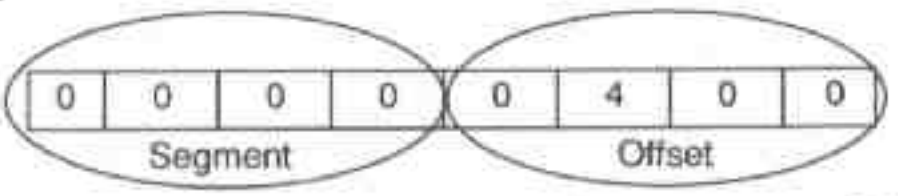
p göstericisinin segment değeri **12A8** ve offset değeri değeri **1B13**'tür.



Benzer biçimde örneğin:

```
p = (char far *) 0x400;
```

işlemiyle uzak göstericinin segment kısmına 0000 ve offset kısmına 0400 değerleri yerleşir.



Bilinçli tür dönüştürmesi yaparken **far** anahtar sözcüğünün de kullanıldığına dikkat ediniz:

```
(char far *) ...
```

Uzak göstericilerle artık 1MB bellek üzerinde istediğimiz bir yere erişebiliriz. Bunun için tek yapacağımız şey erişmek istediğimiz yerin **segment** ve **offset** değerlerini belirlemek ve onları bir uzak göstericiye yerleştirmektir. Örneğin:

```
char far *p = (char far *) 0xB8000000;
...
*p = 'a';
...
```

işlemi ile 'a' karakteri **B800 : 0000** ile belirtilen (doğrusal adresi **B8000**) yere yazılır.

```
*p = 'a';
```

işlemi ile **p** göstericisinin içerisindeki segment ve offset değerlerinden doğrusal adresin elde edilmesi programcı tarafından değil derleyici ve mikroişlemci tarafından yapılmaktadır. Programcı açısından bir yakın göstericinin kullanımı ile uzak göstericinin kullanımı arasında fark yoktur.

80X86 Sembolik Makina Dili Programcısına Not: Uzak göstericinin gösterdiği yere değer yazma işlemi sembolik makina dilinde 2 adımda yapılır.

1. Adım: Derleyici uzak göstericinin segment ve offset değerlerini bir segment ve indeks yazmacına yükler. Derleyiciler bu durumda genellikle **ES** segment yazmacını kullanırlar.

2. Adım: Veri aktarım komutlarıyla bilgi yazılır.

Uzak göstericilere değer aktarılmasında mikroişlemcinin uzak gösterici yükleyen komutlarından faydalanılır. Bu komutlar:

```
LDS (Reg), (Mem)
LES (Reg), (Mem)
```

biçimindedir. **LES** ve **LDS** memory operandının düşük anlamlı 2 byte değerini belirtilen indeks yazmacına, yüksek anlamlı 2 byte değerini ise yine belirtilen segment yazmacına yazar. Örneğin:

```
char far *p;
...
```

```
*p = 'a'
```
işleminde p göstericisi [BP-4]'te olsun.
```
*p = 'a' ;
```
işlemi şöyle gerçekleştirilir:
```
LES   BX, [BP-4]
MOV byte ptr ES:[BX], 'a'
```

Nesne adreslerinin göstericiye aktarılması durumunda C derleyicileri göstericinin yakın ya da uzak olmasına göre yalnızca offset ya da hem segment hem offset aktarımı yaparlar. Örneğin:

```
char ch;
char near *p;
char far *f;
....
p = &ch;
```

ifadesiyle ch değişkeninin bulunduğu adresinin yalnızca offset bilgisi yakın göstericiye aktarılır. Oysa:

```
f = &ch;
```

ifadesi ile ch nesnesinin hem segment hem de offset adresleri f göstericisine aktarılmaktadır.

21.5.1 Uzak Göstericilerin Artırılması ve Eksiltilmesi

Uzak göstericilerin artırılması ya da eksiltilmesi işleminden yalnızca onların offset kısımları etkilenir. Bu durum bir uzak göstericinin offset kısmı sürekli artırıldığında FFFF değerine ulaştıktan sonra tekrar 0000 değerine döneceği anlamına gelmektedir. Uzak göstericilerin artırılmasında segment kısmı değişmez.

Aşağıdaki örneği inceleyiniz:

```
main()
{
    char far *p = (char far *) 0x1000FFFE;
    printf("p = %Fp\n", p);          /* p = 1000:FFFE */
    ++p;
    printf("p = %Fp\n", p);          /* p = 1000:FFFF */
    ++p;
    printf("p = %Fp\n", p);          /* p = 1000:0000 */
}
```

Benzer biçimde uzak göstericinin eksiltilmesi durumunda da segment kısmı etkilenmeden yalnızca offset kısmı eksiltilir. Buradan şöyle bir sonuç çıkarabiliriz: **Bir uzak göstericiyi artırarak bellekte yalnızca 64K bir alan üzerinde ilerleyebiliriz. 64 K'lık alanın aşılması için segment değerinin de değiştirilmesi gerekir.** Ay-

rıca, yukarıdaki örneklerden de gördüğünüz gibi `printf` fonksiyonu ile bir uzak göstericinin değeri `"%Fp"` formatıyla yazdırılmaktadır.

21.5.2 Uzak Göstericilerin Karşılaştırılması

Uzak göstericiler de yakın göstericiler gibi karşılaştırma amacıyla ilişkisel operatörlerle birlikte kullanılabilirler. Ancak uzak göstericilerin `>, <, >=, <=` ve `==, !=` operatörleri ile kullanımları arasında işlevsel farklar vardır. `>, <, >=, <=` operatörleri ile uzak göstericilerin yalnızca offset kısımları karşılaştırılır. Örneğin:

```
char far *p1 = (char far *) 0x10006666;
char far *p2 = (char far *) 0x50005555;
...
if (p1 > p2)
    ...
```

ifadesi Doğru olarak değerlendirilir. Çünkü yalnızca offset kısımları ele alındığında:

```
0x6666 > 0x5555
```

Doğru olarak değerlendirilecektir. Oysa adresler bütünsel olarak ele alınsaydı, `p2` göstericisi `p1` göstericisinden daha büyük bir doğrusal adresi gösterdiğinden ifade Yanlış olarak değerlendirilecekti.

C derleyicileri, uzak göstericilerin `==` ve `!=` operatörleriyle karşılaştırılması durumunda hem segment hem de offset bilgisini karşılaştırma işlemine sokarlar. Aşağıdaki örneği inceleyiniz:

```
char far *p1 = (char far *) 0x10005555;
char far *p2 = (char far *) 0x20005555;
...
```

Bu durumda :

```
if (p1 == p2)
    ...
```

ifadesi **Yanlış** olarak değerlendirilir. Çünkü bu iki göstericinin offset değerleri aynı olmasına karşın segment değerleri farklıdır. İki uzak göstericinin birbirine eşit olması için hem segment hem de offset değerlerinin birbirine eşit olması gerekir. Elde edilen doğrusal adreslerin eşdeğerliliği uzak göstericileri eşit yapmaz. Örneğin:

```
char far *p1 = (char far *) 0x12001C10;
char far *p2 = (char far *) 0x12011C00;
...
```

Burada `p1` ve `p2` göstericileri aynı doğrusal adresi gösterdikleri halde, `p1 == p2` ifadesi Yanlış olarak değerlendirilir. Fakat isterseniz uzak göstericile-

ri bilinçli tür dönüştürmesi ile **unsigned long** türlerine dönüştürerek istediğiniz gibi karşılaştırabilirsiniz. Örneğin:

```
if ((unsigned long) p1 > (unsigned long) p2)
    ....
```

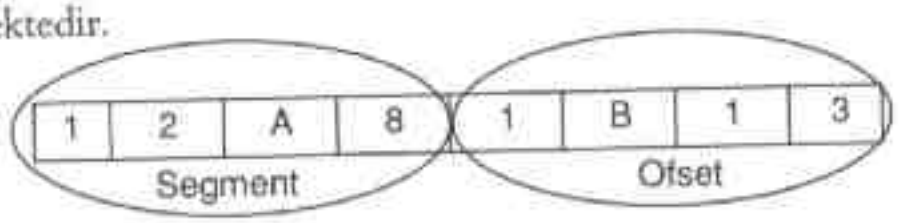
Göstericiler **unsigned long** türüne dönüştürüldüklerinden artık segment ve offset değerleri bir bütün olarak ele alınırlar. Ancak şunu da belirtelim: Uzak göstericilerin >, <, >=, <= operatörleriyle bu biçimde karşılaştırılmaları anlamlı değildir. Çünkü bir uzak gösterici daha büyük bir doğrusal adrese sahip olabildiği halde daha küçük çıkabilir.

21.6 DEV GÖSTERİCİLER (huge Pointers)

Dev göstericiler de uzak göstericiler gibi 4 byte uzunluğundadır. Bildirimleri **huge** anahtar sözcüğü kullanılarak yapılır. Örneğin:

```
char huge *p;
int huge *t;
float huge *h;
...
```

ile **p**, **t** ve **h** birer dev gösterici olarak tanımlanmışlardır. Bu durumda: **sizeof (p), sizeof(t), sizeof(h)** ya da **sizeof(char huge *), ...** ifadeleri 4 değerini üretir. Dev göstericilerin de -tıpkı uzak göstericilerde olduğu gibi- yüksek anlamlı 2 byte değeri segment, düşük anlamlı 2 byte değeri ise offset bilgisi içermektedir.



Uzak göstericilerle dev göstericiler birbirlerine çok benzerler. Aralarındaki fark artırıldıklarında ya da eksiltildiklerinde ortaya çıkar.

21.6.1 Dev Göstericilerin Artırılması ve Eksiltilmesi

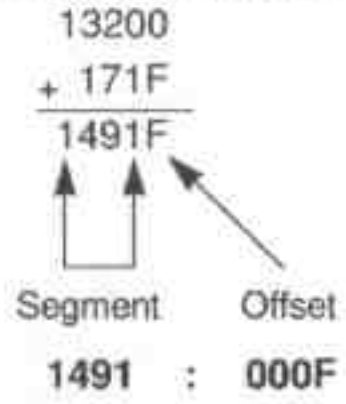
Her bakımdan uzak göstericilere benzeyen dev göstericiler, uzak göstericilerden farklı olarak artırıldıklarında ya da eksiltildiklerinde **normalleştirme (normalization)** işlemine tabi tutulurlar. Normalleştirme, bir göstericinin offset kısmının segment kısmına aktarılarak sadeleştirilmesi işlemidir. Normalleştirilmiş bir gösterici aşağıdaki biçimde bulunur:

```
SSSS:000X
```

Normalleştirme işlemi ile offset bilgisini oluşturan 16 bitin yüksek anlamlı 12 biti sıfırlanarak segment kısmına aktarılır. Örneğin:

1320:171F

göstericisi normalleştirilirse:



1491:000F biçimine dönüşür. Şimdi bu adres bilgisinin bir dev gösterici içerisinde olduğunu düşünelim:

```
char huge *p = (char huge *) 0x1491000F;
...
```

p dev göstericisi 1 artırıldığında ne olur dersiniz? Dev göstericilerin artırma işlemleri sonunda normalleştirildiğini söylemiştik. O halde artırma sonucunda:

1491:0010

değeri yerine bu değerin normalleştirilmiş biçimi olan:

1492:0000

değeri elde edilecektir. Normalleştirme işlemi sayesinde dev göstericiler artırılarak 64K sınırını aşabilirler. Bir dev göstericiyi arttırarak 1MB belleği başından sonuna kadar tarayabiliriz. Aşağıdaki örnek programda herhangi bir tuşa bastıkça 16'şar kümeler halinde belleğin içeriği görüntülenmektedir.

```
#include <stdio.h>

void main(void)
{
    unsigned char huge *p = (char huge *) 0;
    int k = 0;

    for (;;) {
        if (k == 0) {
            putchar('\n');
                if(getch() == 'q')
                    break;
            printf("%Fp", p);
            k = 16;
        }
        printf("%02x", *p);
        --k;
        ++p;
    }
}
```

21.6.2 Dev Göstericilerin Karşılıştırılması

İki dev gösterici herhangi bir ilişkisel operatör ile karşılaştırıldığında, karşılaştırma işlemi gösterdikleri doğrusal adresler dikkate alınarak yapılır. Örneğin:

```
char huge *p1 = (char huge *) 0x10006666;
char huge *p 2 = (char huge *) 0x50005555;
...

if (p1 > p2)
    ...
```

işlemi Yanlış olarak değerlendirilir. Çünkü p1 ve p2 göstericilerinin yalnızca offset bilgileri değil normalleştirilmiş segment ve offset bilgileri birlikte karşılaştırılmaktadır. Oysa p1 ve p2 uzak gösterici olsalardı, o zaman karşılaştırma işlemi Doğru olarak değerlendirilirdi. Örneğin:

```
char huge *p1 = (char huge *) 0x20010101;
char huge *p2 = (char huge *) 0x20000111;
...

if (p1 == p2)
    ...
```

ifadesi Doğru olarak değerlendirilir. p1 ve p2 göstericilerin doğrusal adreslerinin aynı olduğuna dikkat ediniz. Bunu normalleştirme işlemi ile daha iyi görebilirsiniz.

```
2001 : 0101          2011 : 0001
2000 : 0111          2011 : 0001
```

21.7 UZAK ve DEV GÖSTERİCİLER ARASINDAKİ İLİŞKİ

Dev göstericilerin normalleştirme işlemine tabi tutulması, uzak göstericilerle arasındaki tek ayrımı oluşturur. Dev göstericilerin her artırımdan sonra normalleştirilmesi sürekli artırımlarla *64K* alanın dışarısına çıkılmasını mümkün kılmaktadır. Oysa, uzak göstericilerde artırma işleminden yalnızca offset kısmı etkilendiği için bu mümkün değildir. Ancak dev göstericiler de yapay göstericilerdir. Çünkü normalleştirme işlemi mikroişlemci tarafından değil, derleyici tarafından yapay olarak yapılmaktadır. Kaldı ki, pekçok durumda da normalleştirme işleminin yapılması istenmez. Oysa uzak göstericiler *80X86* ailesinin tasarımıyla yakın bir uyum içerisindedir. Bu nedenlerle uygulamalarda dev göstericiler uzak göstericilere kıyasla çok seyrek olarak kullanılırlar.

21.8 VARSAYILAN GÖSTERİCİLER

Gösterici tanımlamalarında **near, far** ya da **huge** anahtar sözcüklerinden hiçbiri kullanılmamışsa varsayılan gösterici **"bellek modeli (memory model)"** denilen bir kavrama göre **yakın** ya da **uzak** gösterici olabilmektedir.

Örneğin:

```
char far *p;
int near *t;
float *f;
...
```

burada **p** uzak, **t** de bir yakın göstericidir. Ancak **f** göstericisi bellek modeline bağlı olarak yakın ya da uzak gösterici olabilir. Peki, bellek modeli nedir?..

21.8.1 Bellek Modeli (Memory model)

Bellek modeli özellikle DOS altında anlamlı olan ve bir programın uzunluğunun hangi sınırlar içerisinde olması gerektiğini anlatan bir kavramdır. Bellek modeli kavramının yeterli bir biçimde anlaşılabilmesi ancak sembolik makina dilinin iyi düzeyde bilinmesiyle mümkün olabilir. Bu nedenle burada yüzeysel bir anlatım yöntemine başvuracağız. Fakat, sembolik makina dili programcıları için de söyleyecek birşeylerimiz olacak tabi...

Yüksek seviyeli programlama dillerinde kullanılan 6 tane bellek modeli vardır. Bunlar: **tiny, small, medium, compact, large** ve **huge** modellerdir. Her bellek modeli için varsayılan göstericilerin yakın mı, yoksa uzak mı olduğu aşağıdaki tabloda belirtilmiştir.

Model	Varsayılan Gösterici Türü	
Tiny	Yakın	
Small	Yakın	
Medium	Yakın	► Fonksiyon göstericileri uzak
Compact	Uzak	► Fonksiyon göstericisi yakın
Large	Uzak	
Huge	Uzak	

Bu durumda örneğin, **Small** modelde çalışıyorsak varsayılan göstericilerimiz yakın göstericiler, **Large** modelde çalışıyorsak, varsayılan göstericilerimiz uzak göstericilerdir. Tabloda **Medium** ve **Compact** modeller için ayrıca bir not görüyorsunuz. Bu notları ancak fonksiyon göstericilerinin açıklandığı bölümde anlamlandırabilirsiniz.

Bellek modelinin değiştirilmesi derleyicilerin tümleşik çevreli uyarlamalarında menüler yoluyla, komut satırlı uyarlamalarında ise komut satırı seçenekleriyle ayarlanmaktadır. *C* derleyicileri için varsayılan bellek modeli genellikle **Small** model olarak belirlenmiştir.

80X86 Sembolik Makina Dili Programcısına Not: Bellek modelinin sembolik makina programcıları için anlamı büyüktür. Bu nedenle bellek modellerini tek tek ele alıp incelemeyi uygun görüyoruz.

Tiny model: Burada kod, data ve stack toplamı 64K değerini geçemez. Dolayısıyla `CS`, `DS` ve `SS` segment yazmaçları aynı değeri gösterir. `Tiny` model programlar `.COM` dosyalarına çevirilebilirler. Bu nedenle bütün data ve fonksiyon göstericileri yakın göstericilerdir.

Small model: Kod ile data ve stack birbirlerinden ayrılmıştır. Kod için bir 64K, data ve stack için ise ayrı bir 64K bellek kullanılabilir. Data ve stack aynı grubun üyesidirler. Bu durumda program toplam olarak 128K değerini geçemez. `Small` model programlarda `CS` farklı, ancak `DS` ve `SS` aynı değeri gösterir. Varsayılan data ve fonksiyon göstericileri yakın göstericilerdir.

Medium model: Burada kod, modül ismine bağlı olarak ayrı segmentler içerisine yazılır. Yani kod modül başına 64K değerini geçemez, ancak toplam projede 64K değerini geçebilir. Bu nedenle varsayılan fonksiyon göstericileri uzak göstericilerdir. Data ve stack aynı grup içerisinde olduğundan toplam olarak 64K değerini geçmez. `CS` ayrı, fakat `DS` ve `SS` aynı değerdedir. Varsayılan data göstericileri yakın göstericilerdir.

Compact model: Medium modelin tersine burada toplam kod için tek bir segment ayrılmıştır. Dolayısıyla toplam kod 64K değerini geçemez. Ancak data ve stack aynı grup içerisinde değildir. Yani toplam statik veri ve toplam yerel veri ayrı ayrı 64K kadar olabilir. `CS`, `DS` ve `SS` segment yazmaçlarının hepsi birbirlerinden ayrı segmentleri göstermektedir. Varsayılan data göstericileri uzak, fonksiyon göstericileri ise yakın göstericilerdir.

Large model: Kod modül ismine bağlı olarak ayrı isimli segmentlerde saklanır. Dolayısıyla kod 64K değerini aşabilir. Fakat data ve stack için tek ve ayrı bir segment ayrılmıştır. Bu nedenle data ya da stack 64K değerini aşamaz. `CS`, `DS` ve `SS` segment yazmaçları tamamen ayrı değerleri göstermektedir. Varsayılan data ve fonskiyon göstericilerinin her ikisi de uzak göstericilerdir.

Huge model: Large modelden farklı olarak statik veriler için de modül ismine bağlı olarak ayrı segmentler kullanılmaktadır. Dolayısıyla toplam kod ve data 64K değerini aşabilir. Stack için de 64K bir alan ayrılmıştır. `CS`, `DS` ve `SS` tamamen farklı değerlerdedir. Varsayılan data ve fonskiyon göstericilerinin her ikisi de uzak göstericilerdir.

C derleyicileri için varsayılan bellek modelinin genellikle `Small` model olduğunu ifade ettik. Yani bellek modelini değiştirmediyseniz, şu ana kadar kullandığınız varsayılan göstericiler yakın göstericilerdir. Biz de kitabımızda -geçmiş bölümlerde de yaptığımız gibi- gösterici bildiriminde **near**, **far** ya da **huge** anahtar sözcüklerinden hiçbirini kullanmamışsak, onların yakın göstericiler olduğunu varsayacağız.

SORAMADIKLARINIZ...

S1) *80X86* sisteminde doğrusal adres neden 4 byte **segment : offset** çifti ile dolaylı bir biçimde elde ediliyor? Neden 2,5 byte (20 bit) doğrudan adres bilgisi kullanılmıyor?

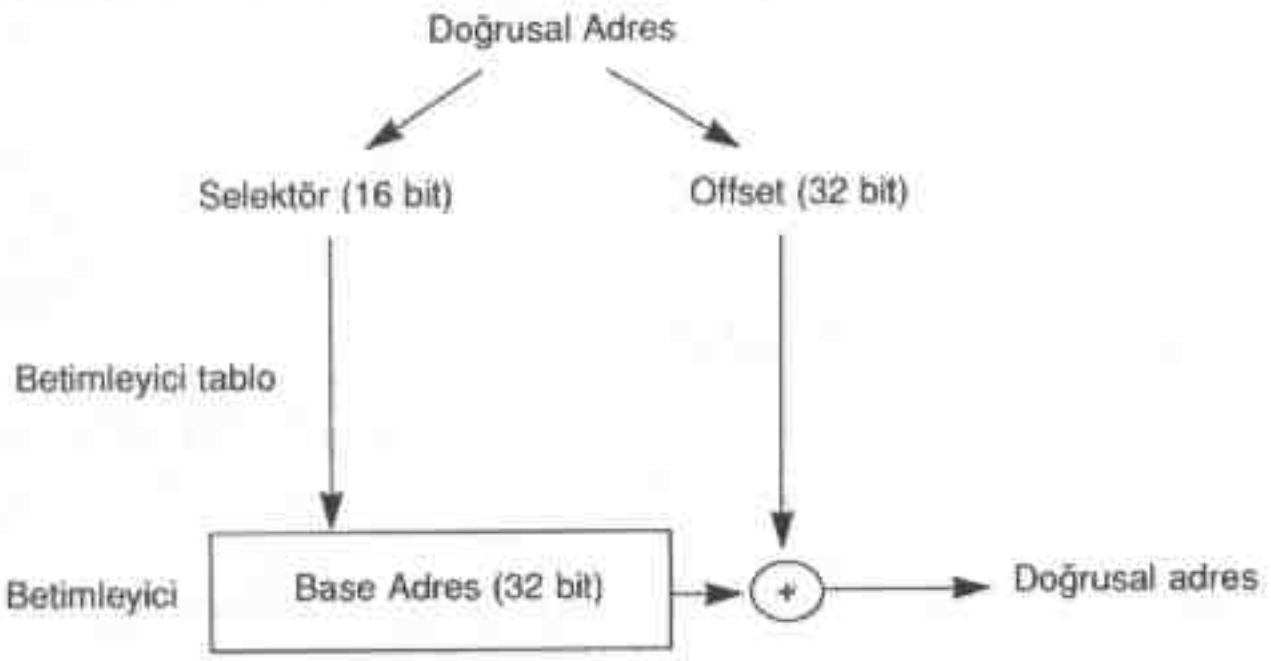
C1) Doğrusal adreslerin 2 ayrı 2 byte ile ifade edilmesi çok daha etkin bir yöntemdir. 2,5 byte gibi bir bilginin organizasyonu ve ve aktarılması kullanılan mimari ile uyum içinde değildir. Üstelik doğrusal adresin **segment** ve **offset** ikilisiyle belirlenmesi makina komutlarını da kısaltır.

80X86 Sembolik Makina Dili Programcısına Not: Doğrusal adres 20 bit ile doğrudan ifade edilseydi, mikroişlemcinin yazmaçları da 20 bit olmak zorunda kalırdı. Bellekten bir defada okunabilen en küçük bilginin 1 byte olduğunu gözönüne alırsak böyle bir sistemin hiç de etkin olamayacağı sonucunu çıkarabiliriz. Üstelik, böyle bir sistemde belleğe doğrudan erişen her makina komutu eskisinden 4 bit daha uzun olacaktır. Oysa yalnızca, 64K alanın dışarısına çıkılacağı zaman segment değerini değiştirmek çok daha etkin bir yöntemdir.

S2) Yakın, uzak ve dev göstericiler *80X86* ailesi içerisindeki diğer işletim sistemleri için de anlamlı mıdır? Örneğin **Windows 3.1, Windows NT, OS/2, XENIX** ve **UNIX** sistemlerinde çalışan *C* derleyicilerinde de bu anahtar sözcükler var mıdır?

C2) Yakın, uzak ve dev götericilerin bölüm içerisinde anlattığımız özellikleri *8086* ile *80286, 80386, 80486* ve *Pentium* işlemcilerinin gerçek modu için geçerlidir. Oysa, diğer işletim sistemlerinin hemen hepsi korumalı modda çalışmaktadır ve korumalı moddaki adresleme biçimi gerçek moddakinden oldukça farklıdır. Ancak bu durum korumalı modda çalışan *C* derleyicilerinde `near`, `far` ya da `huge` anahtar sözcüklerin kullanılmayacağı anlamına gelmez. Farklı sistemlerde aynı anahtar sözcükler farklı bir anlam ifade edebilirler.

80X86 Sembolik Makina Dili Programcısına Not: Korumalı moddaki adresleme biçimi oldukça karmaşıktır. Mikroişlemci korumalı moda geçtiği zaman segment yazmaçlarının işlevleri değişir. Korumalı modda segment değerlerine **selektör (selector)** denir. Selektör, **betimleyici tablo (descriptor table)** denilen bir tabloda indeks belirtmektedir. Korumalı modda doğrusal adres iki bileşenden oluşur: **Selektör ve offset**. Doğrusal adres **selektör** tarafından gösterilen yerden çekilen **base adres** ile bu **offset** değerinin toplanmasıyla bulunur. Korumalı moddaki doğrusal adresin nasıl tespit edildiği aşağıdaki şekilde özetlenmiştir.



Korumalı modda doğrusal adres sayfalama mekanizması aktif değilse fiziksel adrese eşittir. Eğer sayfalama mekanizması aktif ise doğrusal adres bir dizi işlemden sonra fiziksel adrese dönüştürülür.

22 UZAK GÖSTERİCİLERE İLİŞKİN UYGULAMALAR VE EKRAN FONKSİYONLARININ TASARIMI

Bu bölüm uzak göstericilere ilişkin uygulamalara ayrılmıştır. Uygulama konusu olarak ekran belleğini doğrudan kullanan çeşitli ekran fonksiyonlarının tasarımlarını seçtik. Amacımız uygulama yaparken aynı zamanda **80X86** mimarisi hakkında da temel bilgiler edinmek. Ancak bu bölüm de **80X86** mimarisine ve işletim sistemine bağlı taşınabilir olmayan uygulamalar içermektedir. Buradaki uygulamalar tipik olarak DOS işletim sistemi için düşünülmüştür. Dolayısıyla Windows, Unix gibi korumalı modda çalışan işletim sistemlerinde ekran belleğine doğrudan erişilemeyeceği için kodlar taşınabilir değildir. 80X86 mimarisini kullanıyorsanız örnekleri mutlaka yazarak bilgisayarınızda denemelisiniz. Windows 3.1 ya da Windows 95 ile çalışan okuyucularımız DOS imlecine düşerek de uygulamaları çalıştırabilirler.

22.1 EKRANDAKİ GÖRÜNTÜ NASIL ELDE EDİLİYOR?

Bilgisayarlarımızda kullandığımız ekranlar **CRT (Cathode Ray Tube)** sistemiyle çalışmaktadır. Bu sistemde ekranın önünde etrafı camla örtülmüş fosforla kaplı bir tüp vardır. Tüpten gönderilen elektron ışınları ekranda bir noktaya çarptığı zaman fosfor kısa bir süre parlar. Parlamanın sürekli olması için bu işlemin de saniyede belli bir sayıda (örneğin 30 kere) yinelenmesi gerekir. Bu yineleme işlemine **tazeleme (refreshing)** denilmektedir. Eğer tazeleme yeteri kadar fazla yapılmazsa görüntü titreşir. Belli bir tazeleme hızı için titreşme durumu ise fosforun özelliklerine bağlı olarak değişmektedir.

CRT sisteminde tek bir noktanın ekranı taramasıyla görüntü elde edilir . Bu tarama işlemi satır satır yapılabileceği gibi (horizontal retrace) sütün sütun da

(vertical retrace) yapılabilir. Ekranda kaç noktanın taranacağı ise çözünürlükle ilişkilidir. Çözünürlük (resolution) yazılım yoluyla ayarlanabilen video moda bağlı olarak değişir. Örneğin 320 x 200, 640x 450, 640x480, ..., gibi

Renkli ekranlara gelince bunlarda genellikle **kırmızı (red), yeşil (green)** ve **mavi (blue)** olmak üzere üç ayrı renk fosfor kullanılmaktadır. Bu üç renk fosfor belli bir noktaya elektron tabancaları ile gönderilir. Ara renkler üç rengin şiddetleri değiştirilerek elde edilmektedir. Kırmızı, mavi ve yeşilden diğer renklerin elde edildiği ekranlara **RGB (red, green, blue)** ekranlar denir. **LCD** (Liquid - crystal display) teknolojisinin de **CRT**'den farklı olmakla birlikte benzer özellikler taşıdığını söyleyebiliriz.

22.2 EKRAN KONTROL KARTLARI

Yukarıda açıkladığımız görüntü elde etme işlemi kişisel bilgisayarlarda adına **"ekran kontrol kartı"** denilen ayrı bir kontrol birimi tarafından yürütülmektedir. Ekran kontrol kartları genellikle ana kartın genişleme yuvalarına (expansion socket) takılırlar. Aslında ekran kartlarının elektron tabancasını yönlendirmek ve istenilen bölgeye ışın yollatmak gibi oldukça yalın bir işlevi vardır. Ekran kartları kendi içlerinde bu işlemlerin hızlı bir biçimde yürütülmesini sağlayacak ayrı bir işlemciye ve belleğe sahiptir. Ekran kartlarında bulunan bu tür belleklere **ekran belleği (screen memory)** denir.

Kişisel bilgisayarlarımızda kullandığımız ekran kartları hakkında kronolojik sırayla bilgi vermek istiyoruz.

Monochorome Display Adapter (MDA) : İlk **IBM PC**'lerde kullanılan bu kart isminden de anlaşılacağı gibi renksizdi. (**Monochrome** ya da renksiz terimi aslında iki rengi anlatmaktadır. Bu renklerden bir tanesinin siyah olması gerektiğini düşünebilirsiniz. Çünkü siyah renk için bir ışımaya gerek yoktur. Diğeri genellikle beyaz olarak seçilir; ancak daha farklı bir renk de olabilir). Ekrandaki her karakter **9x14** boyutunda bir nokta matrisle elde edilir. Ekranda en küçük birim olan noktaya grafik terminolojisinde **pixel (picture element)** denilmektedir. **MDA** kartları grafik modu olmayan kartlardı. Yani, bu kartlarda görüntü yalnızca karakterler biçiminde örgütlenmişlerdi.

Color Graphics Adapter (CGA) : MDA kartlarının grafik özelliğinin bulunmaması ve renksiz oluşu oldukça kısıtlayıcı bir durum oluşturmaktaydı. Bu nedenle *IBM*, grafik özelliği olan renkli ve yeni bir kart tasarlamıştır. O günlerde RAM fiyatlarının oldukça pahalı olması nedeniyle **CGA** kartlarının tasarımı da az sayıda rengi destekleyecek biçimde yapılmıştır. **CGA** kartları **320x200** çözünürlükte aynı anda ancak 4 renk gösterebiliyordu. Fakat renksiz olmak koşulu ile **640x200** (high resolution) çözünürlüğü de desteklemekteydi.

Hercules Graphics Adapter (HGA) : **HGA**, *IBM MDA* ekranlarıyla kullanılabilen daha yüksek çözünürlüğe sahip (720x348) bir karttır. *MDA*'dan farklı olarak bu kartta grafik modu da bulunmaktadır. Ancak bu kart da renksizdir.

Enhanced Graphics Adapter: *CGA* kartları aynı anda 4 renk gösterebiliyordu. RAM fiyatlarının ucuzlaması ve kullanıcı isteklerinin artmasıyla IBM 1984 yılında daha fazla rengi destekleyen **EGA** kartlarını tasarlamıştır. **EGA** kartları aynı anda 16 rengi destekleyebiliyordu ve monitörü de özeldi. **EGA** ile birlikte grafik çözünürlüğü de artmıştır. Örneğin **640x350** çözünürlükte 16 renk gösterebilen modlar ilk kez **EGA** ile tanımlanmıştır.

Video Graphics Array (VGA): **VGA** ortaya çıkana kadar kullanılan kartların hepsi lojik kartlardı. Yani bilgiler kontrol kartından monitöre seri haberleşme kuralları çerçevesinde bit bit aktarılmaktaydı. Oysa **VGA** sisteminde aynı anda gösterilecek renk sayısının artmasıyla analog haberleşme sistemine geçilmiştir. **VGA** kartlarında digital-analog bir dönüştürücü kullanılarak renk bilgileri monitöre bit bit değil gerilimin bir fonksiyonu olarak analog biçimde gönderilmektedir. **VGA** kartları ile pekçok video mod da ortaya çıkmıştır. Örneğin, **640x480, 16 renk** modu grafik programlarında en yaygın kullanılan video modudur. Bu kart ile aynı anda görülebilecek renk sayısı da 16 dan 256'ya yükseltilmiştir. **VGA** kartlarının bugün için bir standart oluşturduğu söylenebilir.

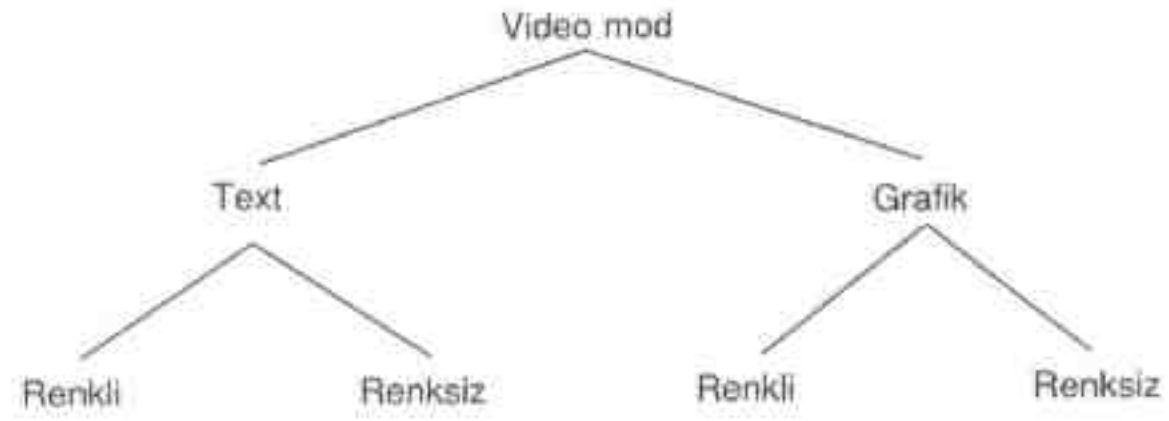
22.3 VIDEO MODLARI

Kontrol kartları programlama yoluyla değişik çözünürlük ve renk sayılarını içeren özellikler gösterebilecek biçimde tasarlanmıştır. **Video modu, kontrol kartının hangi çözünürlükte ve renk sisteminde çalıştığını gösteren sayısal bir bilgidir.** *PC* sisteminde video modu *BIOS* kesmeleri (interrupt) ile değiştirilir.

80X86 Sembolik Makina Dili Programcısına Not: Video modun değiştirilmesi *10H BIOS* kesmesinin *0* numaralı fonksiyonu ile yapılır. AL yazmacına video mod yerleştirildikten sonra kesme çağrılmalıdır.

```
...
MOV   AH, 0
MOV   AL, video_mod
INT   10H
...
```

Video modların **Text** ve **Grafik** olmak üzere iki bölüme ayrılır. Her iki mod da kendi aralarında renkli ve renksiz (monochrome) olmak üzere tekrar iki bölüme ayrılmaktadır.



Text modlarında en küçük birim bir karakterdir ve karakterler kalıp olarak ekrana basılırlar. Görüntü daktilo sayfasına benzer; karakterler hep aynı boylarda ve biçimlerdedir. Bu nedenle text modlarında görüntünün elde edilmesi ve ekrana basılma işlemi oldukça hızlıdır. EGA ve VGA kartlarının bir kısmı çeşitli çözünürlükte text modlarını desteklemektedir. Örneğin: **40x25, 80x25, 132x 25, 132x 43** .... gibi . En yaygın kullanılan text modu **80x25** çözünürlüğe sahip olan **"standart text modu"** dur. Tüm ekran kontrol kartları bu modu desteklemek zorundadır.

80X86 Sembolik Makina Dili Programcısına Not: VGA kartlarında text modundaki karakterler *BIOS* kesmesi kullanılarak değiştirilebilir. Bunun için **10H** kesmesinin **11H** fonksiyonu kullanılır. Bu yolla programcı isterse tek bir karakteri ya da isterse karakterlerin hepsini değiştirebilir. Örneğin, Türkçe klavye programlarındaki Türkçe karakterler bu yolla elde edilirler.

Grafik modunda ise en küçük birim bir noktadır. Ekrandaki noktaya grafik terminolojisinde **pixel** dendiğini anımsayınız. Grafik modlarda programcı pixelleri bağımsız olarak ekrana basabilir. Grafikler pixellerin biraraya gelmesinden oluşurlar. Grafik modlarda ekrana yazı yazmak da mümkündür. Ancak bu durumda pixellerin ekrana bir yazı oluşturacak biçimde tek tek basılması gerekir. Pixelleri biraraya getiren programcılar çok değişik biçimlerde ekrana yazılar yazabilirler. Grafik programlarında **boyutu, rengi** ve **biçimi** ile belirtilen yazılara **font** denilmektedir. Ancak görüntü zenginliğine karşın grafik modlarda çalışmak text modlarına göre oldukça yavaştır; çünkü nokta kontrolünün tek tek yapılması gerekir.

22.3.1 Video Modları ve Monitörler

Video modlarını renkli ve renksiz olmak üzere 2 bölüme ayırmıştık. Ancak video modu renkli olsa da eğer monitörümüz renksiz ise kuşkusuz renk göremeyiz. Aşağıda VGA monitörleri ile video modlar arasındaki ilişkiyi bir tablo halinde veriyoruz:

Video Mod	Monitör	Görüntü
Renksiz	Renksiz	Renksiz
Renksiz	Renkli	Renksiz
Renkli	Renksiz	Gri tonları
Renkli	Renkli	Renkli

Renksiz (monochrome) demekle " siyah ile başka bir renk (çoğu kez beyaz)" anlatılmak istenmiştir. Renksiz yerine iki renkli ya da siyah-beyaz terimlerini de kullanabilirdik. Yukarıdaki tabloda birkaç şeye dikkat etmenizi isteyeceğiz. Eğer video mod renksiz ise monitörünüz ne olursa olsun görüntü renksizdir. Ancak, renksiz bir VGA monitörüne sahip olsanız da renkli modlarda grinin tonlarını (gray scale) görebilirsiniz.

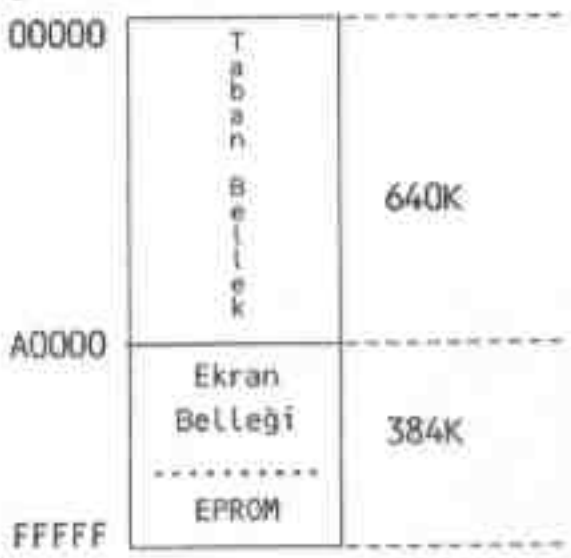
22.4 80X25 STANDART TEXT MODUNDA EKRAN BELLEĞİNİN ORGANİZASYONU

Video kartlarının üzerinde bir RAM bloğu bulunduğunu söylemiştik. Kontol kartı bu RAM bloğu içerisindeki bilgileri belli periyotlarla ekrana gönderir. Görüntü, ekran kartının üzerindeki RAM bloğunun içeriğinin ekrana basılması yoluyla elde edilir. Ekran görüntüsünün bu biçimde elde edilmesi kuşkusuz donanımsal bir özelliktir. O halde şu sorulara yanıt vermeliyiz:

- Ekran belleği nerededir?
- Organizasyonu nasıldır?

22.4.1 Ekran Belleğinin Yeri

Ekran belleğinin başlangıç adresi ve uzunluğu kullanılan ekran kartına bağlı olarak değişir. Örneğin VGA kartlarında ekran belleği DOS belleğinin bitim noktasından başlamaktadır. Bu noktayı giriş bölümünde **üst bellek bölgesi (upper memory area)** olarak isimlendirmiştik. Ekran kartlarının tümü üst bellek bölgesini (upper memory area) kullanmaktadır.



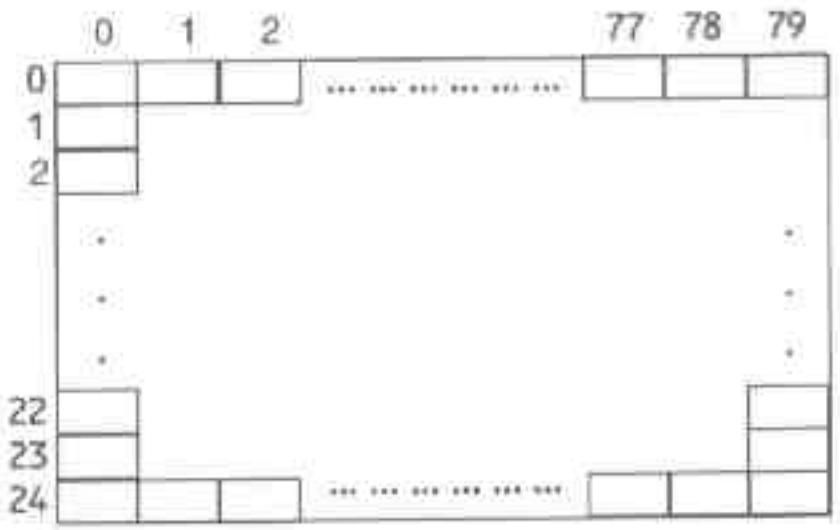
Eski kartlar daha düşük miktarda bellek içermekteydiler. Kartlar geliştikçe ve aynı anda gösterebildikleri renk sayısı arttıkça sahip oldukları bellek miktarı da artmıştır.

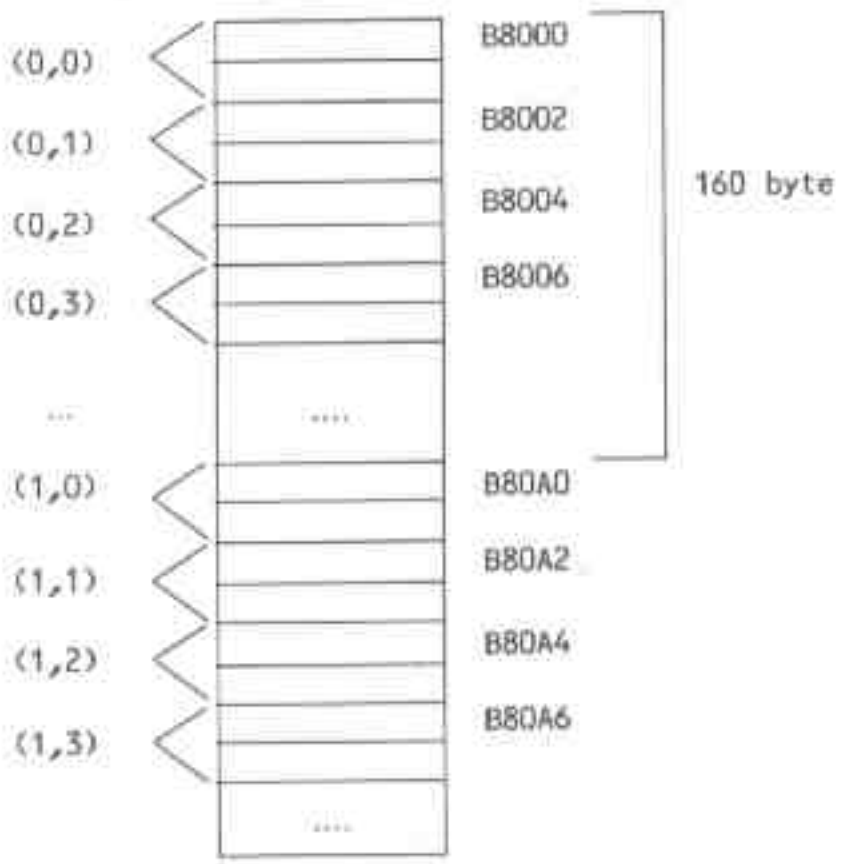
Ekran belleğinin başlangıcı **A0000H** olsa da aktif bölgesi video moda göre değişir. Tüm renkli text modları için aktif bölgenin başlangıç adresi **B8000H**, renksiz text modları için ise **B0000H**'dır. **EGA** ve **VGA** kartlarında 16 renk grafik modlar için aktif bölge **A0000H**'dan başlar.

Video Mod	Video kartı	Aktif bölgenin Başlangıç Adresi
Renkli -Text	Hepsi	B8000
Renksiz -Text	Hepsi	B0000
Grafik 16 renk	EGA/VGA	A0000
Grafik 256 renk	VGA	A0000

22.4.3 Ekran Belleğinin Text Modlardaki Organizasyonu

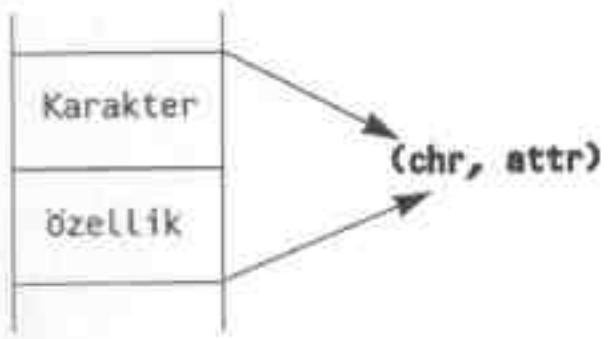
Text modlarında ekrandaki her bölge ekran belleğinin aktif bölgesinde 2 byte ile temsil edilir. Ekranın sol-üst köşesinin koordinatını **(0, 0)** biçiminde ele alırsak, **25X80** standart text modunda ekranın sağ-alt köşesinin koordinatı **(24, 79)** olur. Text modlarında, ekranın sol-üst köşesi **(0, 0)** ekran belleğinin aktif bölgesinin başlangıç adresi olmak üzere, sırasıyla ikişer byte ile temsil edilmektedir. Aşağıdaki şekilleri inceleyiniz:





Ekran belleğinin organizasyonunun soldan sağa ve yukarıdan aşağıya artan sırada olduğuna dikkat ediniz. Bu durumda, ekran belleğinin ilk 160 byte'ı 0. satır için, ikinci 160 byte'ı 1. satır için, üçüncü 160 byte, 2.satır için,... ayrılmıştır.

Ekrandaki her bir bölgenin ekran belleğinde 2 byte ile temsil edildiğini ifade ettik. Bu iki byte'ın ilki ekrana çıkacak **ASCII karakterinin numarasını** ikincisi ise **özelliğini (attribute)** gösterir.



Ekran kontrol kartı ekran belleğinin içeriğini belirli periyotlarla ekrana gönderir. Görüntülerin ekrana çıkması için kart üzerindeki son durak ekran belleğidir. Bu nedenle yöntem ne olursa olsun ekrana çıkacak bilgi mutlaka önce ekran belleğinin aktif bölgesine yazılmalıdır. Bu gerekliliğin nedeni kuşkusuz donanımın tasarımında aranmalıdır. Bu durumda, ekran belleği mikroişlemcinin adres alanı içerisinde olduğuna göre, text modlarda ekranın belli bir bölgesine bir karakter basmak isteniyorsa, tek yapılacak şey o bölgenin ekran belleğinin neresine düştüğünü hesaplamak ve bir uzak gösterici yardımıyla o bölgeye ilgili karakterin ASCII numarasını yazmaktır.

Önce ekran belleğinin aktif bölgesinin başlangıç adresini gösteren bir uzak gösterici tanımlamaya çalışalım. Video mod renkli ise:

```
char far *vp = (char far *) 0xB8000000;
```

B800:0000 ► B8000

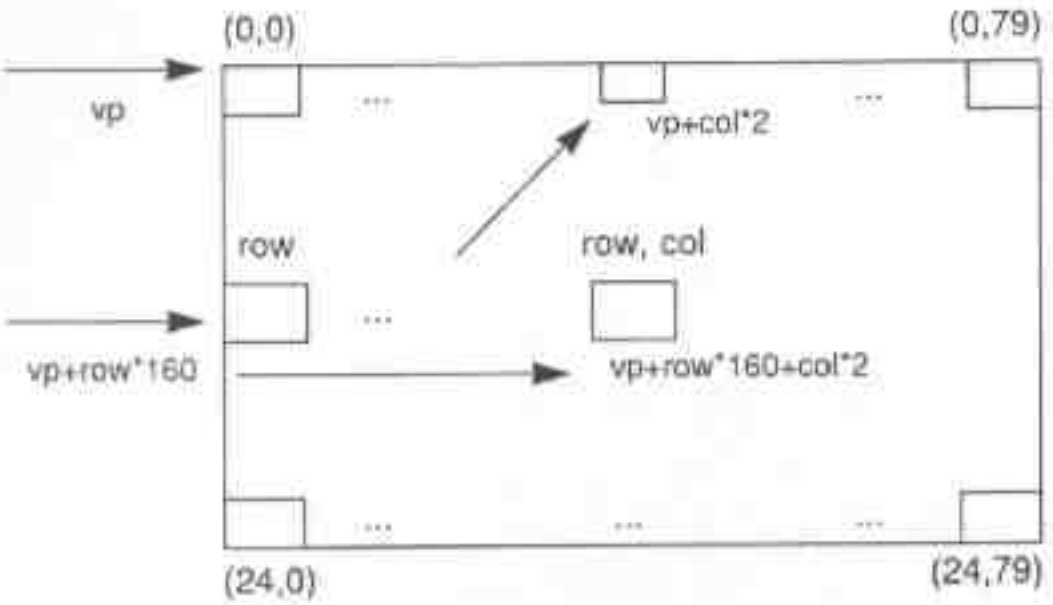
video mod renksiz ise:

```
char far *vp = (char far *) 0xB0000000;
```

B000:0000 ► B0000

Ekranın `row` ve `col` ile belirtilen bölgesine erişmek için `vp` göstericisini ne kadar ilerletmek gerekir dersiniz? Her satır **80** kolon olduğuna göre, ve ekran belleğinde her bir karakter **2** byte ile temsil edildiğine göre:

Koordinat	Adres
(0, 0)	vp
(0,col)	vp + col * 2
(1, 0)	vp + 160
(2, 0)	vp + 2 * 160
(3, 0)	vp + 3 * 160
(4, 0)	vp + 4 * 160
...	...
row	vp + row * 160
row, col	**vp + row * 160 + col * 2**



vp ekran belleğinin aktif bölgesinin başlangıç adresini göstermek üzere:

```
vp = vp + row * 160 + col * 2;
```

ile (row,col) ile belirtilen bölgeyi gösterecek duruma getirilmiştir. Bu durumda:

*vp ► **Ekrana çıkacak ASCII karakterini,**

*(vp+1) ► **Karakterin özelliğini**

gösterir. Özellik (attribute) konusunu ileride ele alınacaktır.

22.5 EKRAN FONKSİYONLARI

Bu bölümde ekran belleğini kullanarak doğrudan ekran işlemleri yapan fonksiyonlar yazacağız. Bu fonksiyonların hiçbiri standart C fonksiyonu değildir. Okuyucu buradan edineceği ipucu ve bilgi ile kendi özel fonksiyonlarını tasarlayabilir.

22.5.1 Ön Hazırlıklar

Bu bölümde ele alacağımız tüm fonksiyonları aynı dosya içerisinde biriktirmelisiniz. Biz bu dosyanın isminin SCR.C olduğunu varsayacağız. Öncelikle işe bu dosyanın içerisinde ekran belleğinin aktif bölgesinin başlangıç adresini gösteren global bir uzak gösterici tanımlamakla başlayacağız. Bu gösterici renkli modlar için:

```
char far *vp = (char far *) 0xB8000000;
```

renksiz modlar için:

```
char far *vp = (char far *) 0xB0000000;
```

biçiminde tanımlanabilir. Bu durumda SCR.C dosyasının organizasyonu aşağıdaki biçimde olmalıdır.

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

char far *vp = (char far*) 0xBX000000;
```

B8000000 renkli modlar için B0000000 renksiz modlar için

```
FONKSİYONLAR BURAYA YAZILACAK

void main()
{
        Deneme Kodunu Buraya Yazabilirsiniz

}
```

22.5.2 _writec

İlk olarak belli bir satır ve sütuna tek bir karakter yazan `_writec` fonksiyonunu tasarlayacağız. Prototipini inceleyiniz:

```
void _writec(int row, int col, int ch);
```

Öncelikle, `vp` global uzak göstericisinin değerini değiştirmemek için fonksiyon içerisinde yerel bir `scrp` göstericisi tanımlamak gerekir.

```
void _writec(int row, int col, int ch)
{
    char far *scrp = vp;          ← scrp ekran belleğininin başlangıç adresini gösterir
    scrp += row * 160 + col * 2;  ← Ekran belleğinde (row, col) bölgesine ilerleniyor.
    *scrp = ch;                   ← Karakter yazılıyor
}
```

Fonksiyonu aşağıdaki kod ile deneyebilirsiniz:

```
void main(void)
{
    _writec(10, 10, 'D');
    _writec(10, 11, 'e');
    _writec(10, 12, 'n');
    _writec(10, 13, 'e');
    _writec(10, 14, 'm');
    _writec(10, 15, 'e');
}
```

22.5.3 _writes

Şimdi de belli bir satır ve sütundan başlayarak bir stringi ekrana yazan fonksiyonu tasarlayalım. Prototipini inceleyiniz:

```
char * _writes(int row, int col, char *str);
```

Tasarımın oldukça basit olduğu söylenebilir. Önce uzak göstericiyi (`row, col`) bölgesine ilerleteceğiz. Daha sonra da `NULL` karakteri görene kadar dizinin elemanlarını teker teker yazacağız. `_writes` fonksiyonunun geri dönüş değeri, ekrana yazılan karakter dizisinin başlangıç adresidir.

```
char * _writes (int row, int col, char *str)
{
    char far *scrp = vp;              /* vp global değişken */
    int k;

    scrp += row * 160 + col * 2;
    for (k = 0; str[k] != '\0'; ++str) {
```

```
        *scrp = str[k];     /* Karakter ekrana yazılıyor */
        scrp += 2;          /* özellik byte'ı atlanıyor. */
    }
    return str;
}
```

Döngü yapısı ile ilgili bazı açıklamalarımız olacak.

```
for (k = 0; str[k] != '\0'; ++k) {
    *scrp = str[k];
    scrp += 2;
}
```

Döngünün `str[k]` değerlerinin `NULL` karaktere eşit olana kadar devam ettiğine dikkat ediniz.

```
*scrp = str[k];
```

ile karakter ilgili bölgeye yazdırılmıştır. Ekranda bir yana geçmek için ekran belleğinde 2 byte ilerlemek gerekir. (Özellik byte'ını atlıyoruz.)

```
scrp += 2;
```

Denemeyi aşağıdaki kod ile yapabilirsiniz.

```
void main(void)
{
    _writes(10, 10, "Bu bir denemedir!");
}
```

22.5.4 _fillc

Bu fonksiyon, belli bir karakteri birden fazla sayıda yatay olarak ekrana yazdırmak amacıyla kullanılabilir. Prototipini inceleyiniz

```
void _fillc(int row, int col, int ch, int n);
```

parametrelerinin `_writec` fonksiyonundan ne farkı var?

```
void _fillc(int row, int col, int ch, int n)
{
    char far *scrp = vp;

    scrp += row * 160 + col * 2;
    for (k = 0; k < n; ++k) {
        *scrp = ch;
        scrp += 2;
    }
}
```

Aşağıdaki kod ile deneyebilirsiniz

```
void main()
{
    _fillc(10, 10, 'x', 10);
}
```

22.6 ÖZELLİK KAVRAMI

Özellik, ekrana basılan karakterin ışıma biçimine ilişkin bilgi veren bir byte uzunluğunda bir bilgidir. Özellikleri ikiye ayırabiliriz:

Renkli özellikler, ekrana basılan karakterin şekil (foreground) ve zemin (background) renklerini gösterir. Renkli özellikler renkli video modlar için söz konusudur. Eğer video mod renkli fakat monitörünüz renksiz ise renk yerine grinin tonlarını görebilirsiniz. Renksiz özellikler rengin dışındaki ışıma özellikleridir. Renksiz özellikler de renksiz video modlar için söz konusudur. Renkli ve renksiz özelliklerin neler olduğu önceden belirlenmiştir.

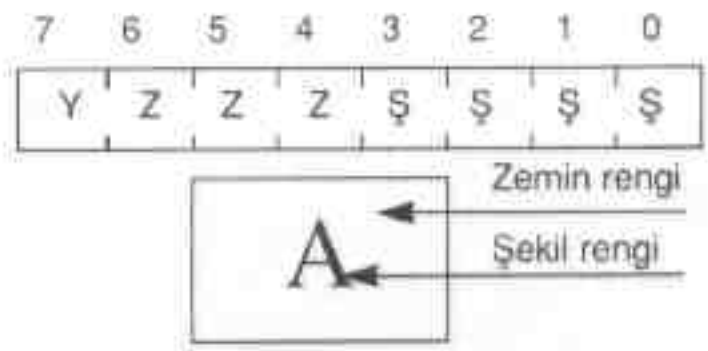
22.6.1 Renksiz Özellikler

Renksiz özellikler, renksiz video modlar için kullanılan özelliklerdir. Bunların işlevleri ve sayısal değerleri önceden belirlenmiştir.

Normal	►	**0x07**	Normal görüntü
Reverse	►	**0x70**	Ters görüntü (taramalı)
Bold	►	**0x0F**	Fazal ışıklı görüntü
Blink	►	**0x87**	Yanıp sönme (göz kırpma)
Under	►	**0x01**	Altı çizgili

22.6.2 Renkli Özellikler

Renkli özellikler ekranda gözükecek karakterin şekil ve zemin rengini belirtir. Özellik byte'ı içerisinde şekil rengi için 4 bit zemin rengi için de 3 bit ayrılmıştır. Geriye kalan 1 bit yanıp sönme (blink) bilgisini içerir.



Şekil ve zemin renklerini elde etmek için ilgili renk bilgilerini özellik byte'ındaki uygun yerlere yerleştirmek gerekir. Toplam 16 şekil renginin ve 8 zemin renginin hangileri olduğu önceden belirlenmiştir. Ancak VGA kartlarında bu varsayılan renkler çeşitli biçimlerde değiştirilebilmektedir. Aşağıda şekil renklerinin sembolik sabitler halinde tanımlandığını görüyorsunuz:

Şekil Renkleri

```
#define BLACK      0        /* Siyah */
#define BLUE       0x01     /* Mavi */
#define GREEN      0x02     /* Yeşil *
/#define CYAN      0x03
#define RED        0x04     /* Kırmızı */
#define MAGENTA    0x05
#define BROWN      0x06     /* Kahverengi */
#define WHITE      0x07     /* Beyaz */
#define GRAY       0x08     /* Gri */
#define LBLUE      0x09     /* Açık mavi */
#define LGREEN     0x0A     /* Açık yeşil */
#define LCYAN      0x0B
#define LRED       0x0C     /* Açık kırmızı */
#define LMAGENTA   0x0D
#define YELLOW     0x0E      /* Sarı */
#define LWHITE     0x0F      /* Açık beyaz */
```

Zemin Renkleri

```
#define BLACK_     0x00     /* Siyah */
#define BLUE_      0x10     /* Mavi */
#define GREEN_     0x20     /* Yeşil */
#define CYAN_      0x30
#define RED_       0x40     /* Kırmızı */
#define MAGENTA_   0x50
#define BROWN_     0x60     /* Kahverengi */
#define WHITE_     0x70     /* Beyaz */

#define BLINK_     0x80     /* Yanıp sönme */
```

Zemin renklerinin yukarıdaki gibi tanımlanması sizi şaşırtmasın. Şekil ve zemin renklerinin bu biçimde tanımlanması Bit Veya işlemine sokularak özellik byte'ı içerisinde yerlerini almasını sağlar. Örneğin, şekil rengi beyaz ve zemin rengi mavi bir özelliği belirtmek istiyorsanız:

```
WHITE | BLUE_
```

biçiminde bir ifade kullanabilirsiniz. Önişlemci kodu,

`0x07 | 0x10` biçiminde açar. Bu ifade de:

`0000 0111 | 0001 0000 = 0001 0111` işlemiyle eşdeğerdir. Bit Veya işleminde sıfırın etkisiz eleman olduğunu anımsayınız. Bu durumda şekil ve zemin renkleri kendi bit pozisyonlarına girmektedir.

Renksiz ve renkli özellikleri uygulamalarımızda SCR.H isimli bir dosyanın içerisine sembolik sabit biçiminde yazacağız.

SCR.H

```
/*Renksiz Özellikler */
# define NORM          0x07
# define REV           0x70
# define BOLD          0x0F
# define UNDER         0x08

/* Şekil Renkleri */
# define BLACK         0
# define BLUE          0x01
# define GREEN         0x02
# define CYAN          0x03
# define RED           0x04
# define MAGENTA       0x05
# define BROWN         0x06
# define WHITE         0x07
# define GRAY          0x08
# define LBLUE         0x09
# define LGREEN        0x0A
# define LCYAN         0x0B
# define LRED          0x0C
# define LMAGENTA      0x0D
# define YELLOW        0x0E
# define LWHITE        0x0F

/* Zemin Renkleri */
# define BLACK_        0
# define BLUE_         0x10
# define GREEN_        0x20
# define CYAN_         0x30
# define RED_          0x40
# define MAgENTA_      0x50
# define BROWN         0x60
# define WHITE         0x70

/* Yanıp Sönme */
# define BLINK         0x80
```

Uygulamalarımızda `SCR.H` dosyasını `SCR.C` dosyasına dahil ederek kullanacağız:

SCR.C

```
#include <stdio.h>
#include <conio.h>
#include "scr.h"

        Fonksiyonlar

void main()
{
     ...   Deneme kodu
}
```

22.6.2 writec

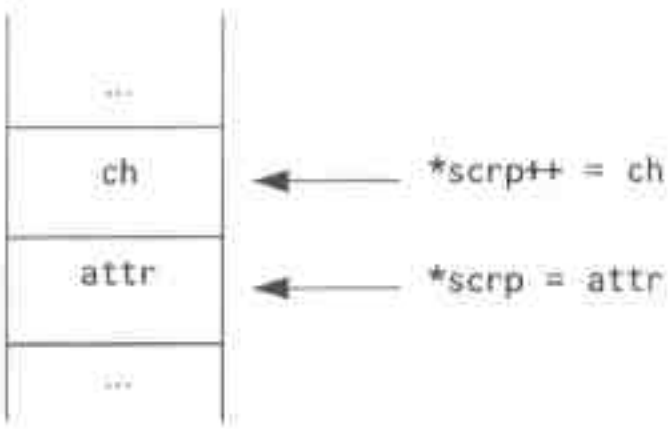
_writec fonksiyonu bir karakteri özellik bilgisini dikkate almaksızın ekrana yazıyordu. Şimdi de karakteri özelliği ile birlikte ekrana yazdıran fonksiyonu tasarlayalım.

```
void writec (int row, int col, int ch, int attr)
{
        char far *scrp = vp;

        scrp += row * 160 + col * 2;
        *scrp++ = ch;
        *scrp = attr;
}
```

Özellik bilgisinin nasıl yazıldığına dikkat ediniz:

```
*scrp++ = ch;
```

ifadesinde ++ sonek durumunda bulunduğu için scrp atama işleminden sonra artırılır. Böylece scrp artık ilgili karakterin özellik byte'ını gösterecek biçime getirilmiştir.



Fonksiyonu aşağıdaki kod ile deneyebilirsiniz:

```
void main(void)
{
    _writec(10, 10, 'D', WHITE | BLUE_);
    _writec(10, 11, 'e', RED | WHITE_);
    _writec(10, 12, 'n', RED | WHITE_);
    _writec(10, 13, 'e', RED | WHITE_);
    _writec(10, 14, 'm', RED | WHITE_);
    _writec(10, 15, 'e', RED | WHITE_);
    _writec(10, 16, '!', RED | WHITE_);
}
```

22.6.3 writes

Şimdi de sıra `_writes` fonksiyonunun özellikli uyarlamasına geldi. Prototipini aşağıda görüyorsunuz:

```
char *writes(int row, int col, char *str, int attr);
```

Bu fonksiyonun `_writes` fonksiyonundan tek farkı `attr` parametresinin olmasıdır. Aşağıdaki tasarım ile `_writes` fonksiyonunu karşılaştırınız

```
char *writes (int row, int col, char *str, int attr)
{
    char far *scrp = vp;      /* vp global değişken */
    int k;

    scrp += row * 160 + col * 2;
    for ( k = 0; str[k] != '\0'; ++str) {
        *scrp++ = str[k];     /* karakter ekrana yazılıyor */
        *scrp++ = attr;       /* özellik yazılıyor */
    }
    return str;
}
```

Fonksiyonu aşağıdaki kod ile test edebilirsiniz:

```
void main(void)
{
    writes(10, 10, "Bu bir denemedir!", RED | WHITE_);
}
```

22.6.4 fillc

`_fillc` fonksiyonu özellik bilgisi olmadan bir karakteri n defa ekrana yazıyordu. Oysa `fillc` fonksiyonu özellik bilgisini de dikkate almaktadır.

```
void fillc(int row, int col, int ch, int n)
{
    char far *scrp = vp;

    scrp += row * 160 + col * 2;
    for (k = 0; k < n; ++k) {
        *scrp++ = ch;
        *scrp++ = attr;
    }
}
```

Fonksiyonu aşağıdaki kod ile deneyebilirsiniz.

```
void main(void)
{
    fillc(0, 0, 'x', BOLD,  2000);
    getch();
}
```

22.6.5 vfillc

`vfillc` fonksiyonunu `fillc` fonksiyonunun düşey yazan biçimi olarak düşünebilirsiniz. Parametrelerinin de `fillc` fonksiyonu ile tamamen aynı olduğuna dikkat ediniz:

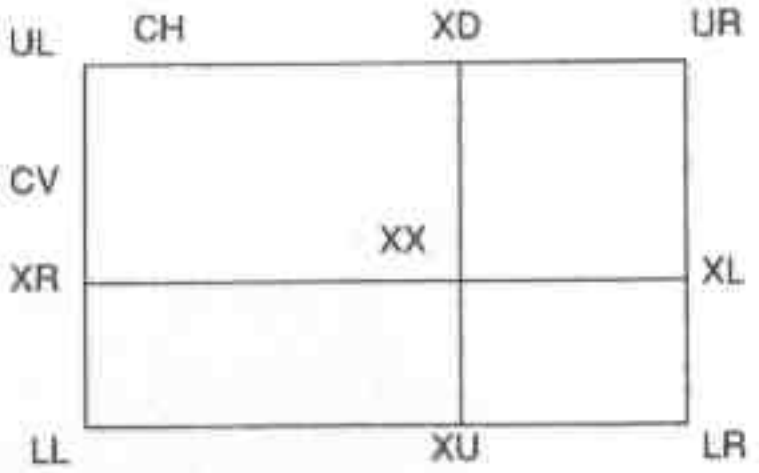
```
void vfillc(int row, int col, int ch, int attr, int n)
{
    char far *scrp = vp;

    scrp += row * 160 + col * 2;
    for (k = 0; k < n; ++k) {
        *scrp = ch;           /* karakter yazılıyor */
        *(scrp+1) = attr;     /* özellik yazılıyor */
        scrp += 160;          /* sonraki satıra geçiliyor */
    }
}
```

Fonksiyonu aşağıdaki kod ile deneyebilirsiniz

```
void main(void)
{
    vfillc(10, 10, 'a' BOLD, 5);
}
```

22.7 GRAFİK KARAKTERLER

Yatay ya da düşey çizgi çizmek, dikme çıkmak gibi amaçlarla kullanılan ve birbirlerini tamamlayan karakterlere grafik karakterler denir. Grafik karakterler ASCII tablosunun ikinci yarısında tanımlanmışlardır. Okunabilirliği artırmak amacıyla grafik karakterleri de sembolik sabitler biçiminde `SCR.H` dosyası içerisinde tanımlayacağız. Aşağıdaki şekilde grafik karakterlerinin hangi sembolik sabitlerle kullanılacağı açıklanmaktadır.



Ayrıca bu karakterlerin bir de çift çizgili biçimleri vardır. Çift çizgili biçimlerinin sonuna **D** (**D**ouble sözcüğünden) getirerek belirteceğiz. Grafik kararkterlerinin sayısal değerlerini `SCR.H` dosyasına sembolik sabitler biçiminde yazabilirsiniz.

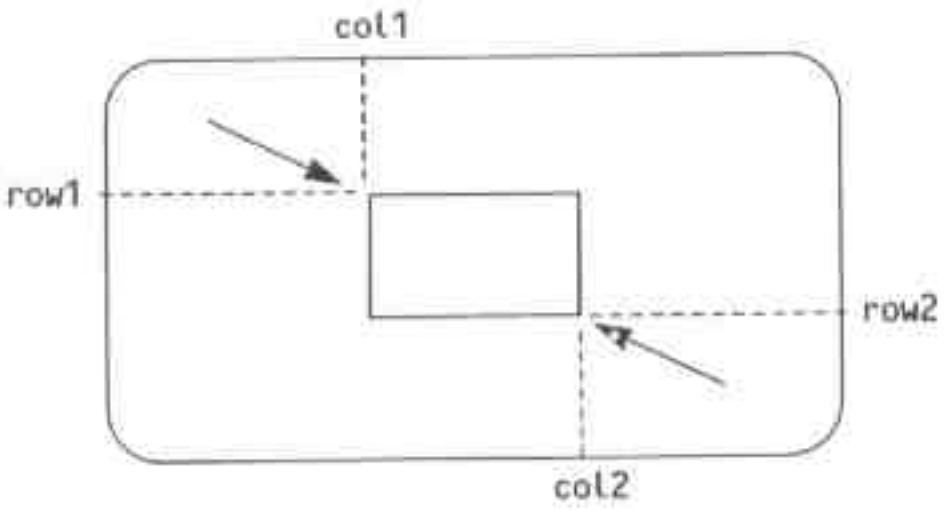
SCR.H

```
.....
/* Grafik karakterler */
#define UL      218     /* ┌ */
#define UR      191     /* ┐ */
#define LL      192     /* └ */
#define LR      217     /* ┘ */
#define XD      194     /* ┬ */
#define XR      195     /* ├ */
#define XL      180     /* ┤ */
#define XX      180     /* ┼ */
#define C_H     196     /* ─ */
#define C_V     179     /* │ */
#define ULD     201
#define URD     187
#define LLD     200
#define LRD     188
#define XUD     202
#define XRD     204
#define XXD     209
#define C_VD    186
```

Bu karakterlerin bir de çift çizgili biçimleri vardır. Bunların ASCII karakter numaralarını ASCII tablosundan bulabilirsiniz.

22.6.6 Çerçeve çizen fonksiyon

Şimdi de sıra çerçeveye çizmeye geldi.. Çerçeveyi daha önce yazdığımız `writec`, `fillc` ve `vfillc` fonksiyonlarını kullanarak çizeceğiz. Ekranda bir çerçeve sol üst ve sağ alt köşegenlerinin koordinatları ile belirlenebilir.



Fonksiyonun tasarımını inceleyiniz:

```
void frame(int row1, int col1, int row2, int col2, int attr)
{
    int k;

    writec(row1, col1, C_UL, attr);              /* Köşe karakterleri */
    writec(row1, col2, C_UR, attr);
    writec(row2, col1, C_LL, attr);
    writec(row2,col2, C_LR, attr);
    vfillc(row1+1, col1, C_V, attr, row2-row1-1); /* Yatay çizgiler */
    vfillc(row1+1, col2, C_V, attr, row2-row1-1);
    fillc(row1, col1+1, C_H, attr, col2-col1-1);  /* Düşey çizgiler */
    fillc(row2, col1+1, C_H, attr, col2-col1-1);
}
```

Aşağıdaki gibi bir kod ile deneyebilirsiniz:

```
void main(void)
{
    frame(10, 10, 20, 20, NORM);
}
```

Çerçeve fonksiyonunun çift çizgili uyarlamasını tasarlamayı size bırakıyoruz...

23

DİNAMİK BELLEK YÖNETİMİ

Belleğin verimli bir biçimde kullanılmasını hedefleyen *"dinamik bellek yönetimi"*, C'nin en önemli konularından biridir. C'de dinamik bellek yönetimi, dinamik bellek fonksiyonlarıyla yapılır. Bu bölüm içerisinde dinamik bellek fonksiyonları, "standart dinamik bellek fonksiyonları" ve "sistem bağımlı dinamik bellek fonksiyonları" olmak üzere iki bölüme ayrılarak ayrıntılı bir biçimde incelenmiştir.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Dinamik bellek fonksiyonlarına neden ihtiyaç duyulmaktadır?

2) `malloc`, `realloc`, `calloc` ve `free` fonksiyonları nasıl kullanılır?

3) Sistem bağımlı dinamik bellek fonksiyonlarına neden ihtiyaç duyulmaktadır?

23.1 GİRİŞ

Dizilerin derleyiciler tarafından, derleme aşamasında ele alındığını biliyorsunuz. Bu yüzden programın çalışma zamanı sırasında bir dizinin uzunluğunu değiştirmek mümkün değildir. Oysa, pekçok uygulamada dizilerin ne kadar uzunlukta açılacağı programın çalışma zamanı sırasında ve ancak birtakım işlemlerden sonra belirlenebilmektedir. Örneğin, bir mağazaya gelen müşterilerin isim ve soyadlarını bir karakter dizisine eklediğimizi düşünelim. Mağazaya kaç kişinin geleceği belli değildir. Bu durumda karakter dizisini hangi uzunlukta açmak gerekir? Ya da bir dizindeki (directory) dosyaları isme göre sıraya dizmek amacıyla dosya bilgilerini geçici olarak bir dizide saklayacağımızı düşünelim. Dizi ne kadar uzunlukta açılmalıdır, dersiniz? Belli değil; çünkü dizin içerisinde istenildiği kadar çok sayıda dosya olabilir. Somut örnekleri çoğaltabiliriz. Bu tür örneklere özellikle veri tabanı uygulamalarında sıklıkla rastlarız. Aslında bu örneklerde anlatılmak istenen, bazı uygulamalarda dizilerin gerçek uzunluğunun programın çalışması sıra-

sında ve ancak birtakım olaylar sonucunda kesin olarak belirlenebildiğidir. Bu durumda dizilerle çalışan programcı herhangi bir gösterici hatasıyla karşılaşmamak için dizileri *"en kötü olasılığı gözönünde bulundurarak"* açmak zorunda kalır. Bu yöntem ise belleğin verimsiz bir biçimde kullanılması anlamına gelir. Üstelik, açılan diziler yerel ise ilgili blok sonlanana kadar, global ise sürekli olarak bellekte tutulacaktır. Oysa, dizi ile ilgili işlem biter bitmez, dizi için ayrılan bellek bölgesinin boşaltılması belleğin verimli kullanımı için gereklidir.

Programın çalışma zamanı sırasında belli sayıda sürekli bellek bölgesinin tahsis edilmesine ve istenildiğinde geri bırakılmasına olanak sağlayan yöntemlerin kullanılmasına dinamik bellek yönetimi denir. **C'de dinamik bellek yönetimi dinamik bellek fonksiyonlarıyla yapılmaktadır.**

23.2 DİNAMİK BELLEK FONKSİYONLARI

Dinamik bellek fonksiyonları, programın çalışma zamanı sırasında, belli uzunlukta sürekli bellek bölgesini sisteme danışarak tahsis eden ve istenildiğinde geri bırakan kütüphane fonksiyonlarıdır. Dinamik bellek fonksiyonlarını, **"standart dinamik bellek fonksiyonları"** ve **"sistem bağımlı dinamik bellek fonksiyonları"** olmak üzere iki kısıma ayırabiliriz.

Standart dinamik bellek fonksiyonları, tüm sistemlerde aynı isimle ve aynı işlevleri yerine getirecek biçimde bulunan fonksiyonlardır. Sistem bağımlı dinamik bellek fonksiyonları ise -ismi üzerinde- sistemden sisteme değişebilen ve taşınabilir olmayan fonksiyonlardır.

23.3 STANDART DİNAMİK BELLEK FONKSİYONLARI

Bu bölümde standart dinamik bellek fonksiyonlarını tek tek ele alarak örneklerle inceleyeceğiz.

23.3.1 malloc

`malloc` en çok kullanılan dinamik bellek fonksiyonudur. Prototipi `STDLIB.H` dosyasının yanı sıra, Borland derleyicilerinde ALLOC.H, Microsoft derleyicilerinde ise MALLOC.H dosyalarında da bildirilmiştir; inceleyiniz:

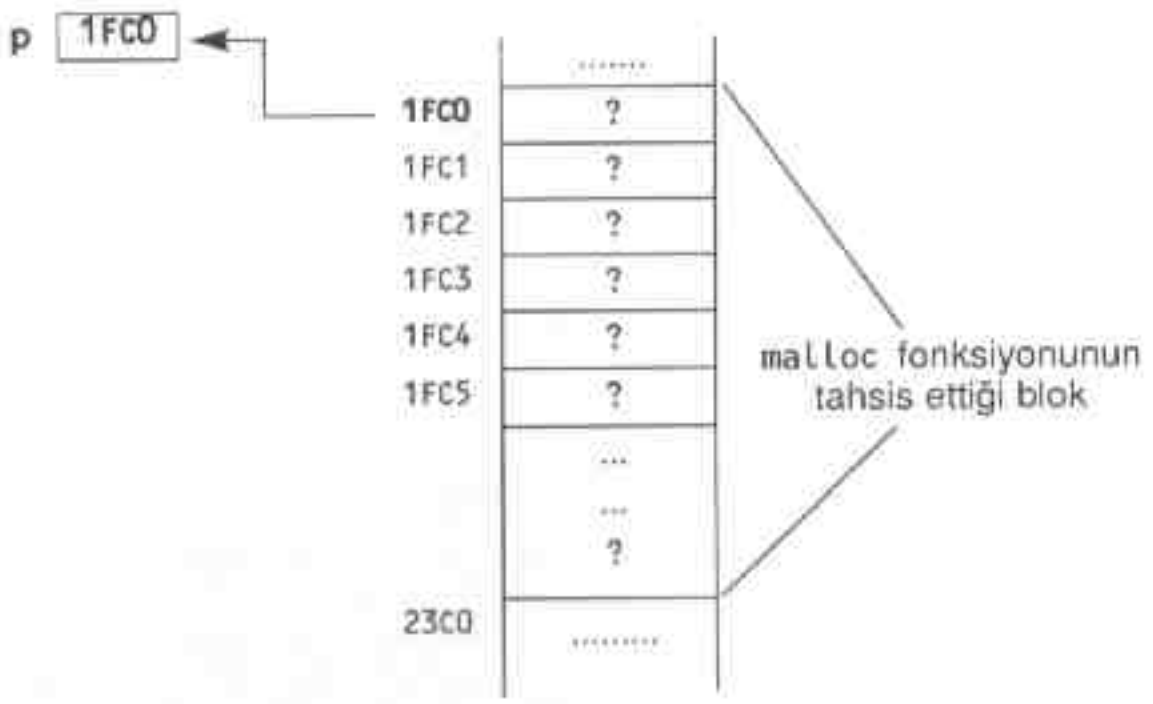
```
void *malloc(unsigned size);
```

`malloc` fonksiyonu, programın çalışma zamanı sırasında sisteme danışarak belleğin güvenli bir bölgesinde parametresi ile belirtilen byte kadar uzunlukta sürekli bellek bölgesini tahsis eder. Geri dönüş değeri olarak da tahsis ettiği sürekli bloğun başlangıç adresini vermektedir. `malloc` fonksiyonu herhangi bir nedenden ötürü tahsisat işlemini yapamazsa `0` değerine (`NULL` pointer) geri döner. Tahsisatın başarılı bir biçimde yapılıp yapılmadığı mutlaka fonksiyon çağrıldıktan sonra programcı tarafından kontrol edilmelidir.

Aşağıdaki örneği izleyiniz:

```
char *p;
...

p = malloc (1024);
if (p == NULL) {
    printf("Yetersiz bellek!..\n");
    exit(1);
}
...
```

Burada, `malloc` fonksiyonundan sisteme danışarak 10 byte sürekli bellek tahsis etmesi istenmiştir. Fonksiyonun geri dönüş değerinin `p` göstericisine atandığına dikkat ediniz.



`malloc` fonksiyonun tahsisatı yapabildiğini varsayalım. Tahsis ettiği bloğun başlangıç adresi de `1FC0` olsun. Bu adres `p` göstericisine atandığına göre, `p` göstericisinin `1024` elemanlı bir diziden işlevsel bir farkı kalmamıştır!..

```
p = malloc(1024);
...
```

Tahsis edilen bu blok artık * ve [] gösterici operatörleriyle tıpkı bir dizi gibi güvenle kullanılabilir. Ancak, `malloc` fonksiyonun bu tahsisatı yapabilmesi garanti değildir. Çünkü sistemde tahsis edilebilecek süreksi ve boş bellek bölgesi kalmamış olabilir. Programcı tahsisatın yapıldığını varsayarak programına devam ederse gösterici hataları ile karşı karşıya kalabilir. Bu nedenle tahsisatın başarısının mutlaka kontrol edilmesi gerekir:

```
p = malloc(10);
if (p == NULL) {
    printf("Yetersiz bellek!..\n");
    exit(1);
}
```

Tahsisatın yapılamadığı durumda **malloc** fonksiyonundan **p** göstericisine atanan değer, NULL gösterici (sıfır) olacağından, örneğimizde ekrana bu durumu anlatan bir mesaj yazdırılarak program sonlandırılmıştır. Kontrolü aşağıdaki gibi, bir hamlede de yapabilirsiniz:

```
if ((p = malloc(10)) == NULL) {
        printf("Yetersiz bellek!..\n");
        exit(1);
}
```

`malloc` fonksiyonunun geri dönüş değerinin `void *` olduğuna dikkat ediniz. `void` göstericiler hangi göstericilere atanırlarsa atansınlar bir uyarı ya da hata durumu oluşturmazlar. Aşağıdaki örneği inceleyiniz:

```
int *p;
...
p = malloc(100);
if (p == NULL) {
    printf("Yetersiz bellek!...\n");
    exit(1);
}
```

Burada **p** göstericisi `int` türündendir. `int` türünün uzunluğu da DOS ve Windows 3.1 altında çalışan derleyicilerde 2 byte, UNIX ve Windows 95 altında çalışan derleyicilerde ise 4 byte olduğu için:

```
p = malloc(100);
```

ile DOS ve Windows 3.1 altında çalışıyorsak 50 elemanlı, UNIX ve Windows 95 altında çalışıyorsak 25 elemanlı bir `int` blok tahsis edebiliriz. Böyle bir durum programın taşınabilirliğini bozabilir. Ne yapmalı dersiniz?

`sizeof` operatörü imdadımıza yetişiyor!..

```
p = malloc(sizeof(int) * 50);
```

Şimdi yukarıdaki ifade ile hangi sistemde olursa olsun 50 elemanlı bir `int` blok tahsis edilecektir. Çünkü `sizeof` bir operatördür ve DOS altındaki derleyiciler `sizeof(int)` için 2 değerini UNIX altında derleyiciler ise 4 değerini üretecek biçimde yazılmışlardır.

Prototipinden de görüldüğü gibi `malloc` fonksiyonunun parametresi `unsigned int` türündendir. Bu da DOS altında en fazla **64K** (**65535** byte) sürekli bellek tahsis edilebileceği anlamına gelir. Oysa UNIX sistemlerinde `unsigned int` türü **4** byte uzunluğundadır. Yani, bu sistemlerde teorik olarak `malloc` ile **4GB** (**4294967295** byte) tahsisat işlemi yapılabilir. Tabi bu durum, tahsisatın kesinlikle yapılabileceği anlamına gelmez.

`malloc` fonksiyonunun program çalışırken güvenli bölgeyi nasıl tespit ettiğini ve orayı bizim için nasıl tahsis ettiğini merak edebilirsiniz. Bellek yönetimi işletim sisteminin kontrolünde olduğu için, *"böyle bir tahsisat işlemi ancak işletim sistemine danışılarak yapılabilir"* biçiminde düşünebilirsiniz. Bu doğru bir yaklaşımdır; ancak tahsisat işleminin ayrıntısı da sistemden sisteme değişebilmektedir.

80X86 Sembolik Makina Dili Programcısına Not: `malloc` ve diğer dinamik bellek fonksiyonlarıyla *DOS* altında tahsisatın nasıl yapıldığı oldukça karmaşıktır. Bunun için öncelikle programın yüklenme biçiminden bahsetmek faydalı olacaktır. *DOS*, *.COM* ve *.EXE* dosyalarını yüklemek için eldeki bütün boş belleği tahsis eder. (*.EXE* dosyasının başlık kısmında program için ne kadar bellek gerektiği belirtiliyorsa da genellikle derleyicilerin giriş kodları (start up module) tarafından mevcut belleğin tümü tahsis edilir). Program yüklenirken ya da programın giriş kodu tarafından yapılan bu tahsisatlarda *DOS*'un INT 21h, Function:48h ve INT 21h, Fonksiyon:49h kesmeleri kullanılır. `malloc` ve diğer dinamik bellek fonksiyonları tahsis edilmiş bellek bölgelerinin adreslerini kendi içlerinde bir tablo yardımıyla tutarlar. Tahsisat bilgileri için, -belleğin tamamı zaten *DOS* düzeyinde tahsis edildiğinden- içsel bir tablo tutmak yeterlidir.

`malloc` ve diğer standart dinamik bellek fonksiyonlarının geri dönüş değerlerinin türü eski sistemlerde `void *` yerine `char *` biçiminde tanımlanmıştır olabilir. Özellikle 1983 yılına kadar yazılmış olan derleyicilerde `malloc` fonksiyonunun geri dönüş değeri `char *` biçimindedir.

```
char *malloc(unsigned size);
```

Bu durumda, eğer yazdığınız programın bu tür sistemlerde de problemsiz çalışmasını istiyorsanız bilinçli tür dönüşümü uygulamalısınız.

Örneğin:

```
int *p;
...
p = (int *)  malloc (sizeof(int) * 100);
```

Eğer bilinçli tür dönüştürmesi yapmazsanız, farklı türdeki göstericilerin birbirlerine atanmasında ortaya çıkan bir uyarı durumuyla karşılaşabilirsiniz. Aslında

bilinçli tür dönüşümü tam bir taşınabilirlik sağlamanın yanı sıra okunabilirliği de artırmaktadır. Çünkü:

```
p = (int *)  malloc(sizeof(int) * 100);
```

ifadesine bakan bir kişi p göstericisinin türü hakkında hemen bir bilgi edinecektir.

Aşağıda malloc fonksiyonunun kullanımına ilişkin basit bir örnek program görüyorsunuz. Bilgisayarınızda yazarak deneyiniz.

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

void main(void)
{
    char *s;

    s = malloc(80);
    if (!s) {
        printf("Yetersiz bellek!..\n");
        exit(1);
    }
    gets(s);
    printf("%s\n", s);
}
```

Aşağıdaki örnek programda da malloc ile tahsis edilebilecek toplam alan (**heap** olarak isimlendirilir) bulunmaktadır:

```
void main(void)
{
    long t = 0;
    char *p;

    for (;;) {
        p = malloc(1024);
        if (p == NULL)
            break;
        t += 1024;
    }
    printf("Toplam alan: %ld\n", t);
}
```

malloc fonksiyonu ile tahsis edilen bloklar arasında aritmetik ya da fiziksel bir ilişki söz konusu değildir. Örneğin:

```
char *p;
char *s;
```

```
...
p = malloc(50);
if (!p) {
    printf("Yetersiz bellek!...\n");
    exit(1);
}
s = malloc(50);
if (!s) {
    printf("Yetersiz bellek!...\n");
    exit(1);
}
...
```

Tahsisat başarılı ise `s` ile `p` arasında herhangi bir ilişki olmak zorunda değildir. Örneğimizde `s` bloğunun daha sonra tahsis edildiği için hemen `p` bloğunun altında olması yönünde bir garanti yoktur. Bu nedenle `malloc` daha önce tahsis edilmiş bir bellek bloğunu büyütmek amacıyla kullanılamaz.

Tahsisat işleminden sonra kontrol yapmanın öneminden bahsetmiştik. Ancak programcılar çoğu kez küçük miktarda ve çok sayıda bloğun tahsis edilmesi durumunda kontrolü gereksiz bulma eğilimindedirler. Oysa, kontrolden vazgeçmek yerine daha kolaylaştırıcı yöntemler denenmelidir. Örneğin `p1, p2, p3, p4, p5` gibi 5 ayrı gösterici için 10'ar byte alan tahsis edilmek istensin. Bu durumda kontrolü mantıksal operatörler ile bir hamlede yapabilirsiniz.

```
char *p1, *p2, *p3, *p4, *p5;
...
p1 = malloc(10);
p2 = malloc(10);
p3 = malloc(10);
p4 = malloc(10);
p5 = malloc(10);

if (!(p1 && p2 && p3 && p4 && p5)) {
    printf("Yetersiz bellek!...\n");
    exit(1);
}
...
```

`malloc` fonksiyonu blokların herhangi birinde başarısız olursa bu durum `if` deyimi içerisinde tespit edilmektedir.

Son olarak `malloc` ile tahsis edilen bloğun içerisinde rastgele değerlerin bulunduğunu belirtelim. Tıpkı yerel değişkenlerde olduğu gibi tahsis edilen blok üzerinde herhangi bir ilkdeğer verme işlemi uygulanmaz. Tahsis edilen blok içerisindeki değerler, bellekte o anda bulunan rastgele değerlerdir.

Adrese geri dönen fonksiyonların kendi içlerinde **malloc** ile dinamik tahsisat yaptıklarına da sıklıkla rastlanmaktadır. Klavyeden isim alan **getname** isimli bir fonksiyonu örnek olarak verebiliriz:

```
#include <stdio.h>
#include <stdlib.h>
char *getname(void)
{
    char *s;
    s = malloc(30);
    if (s == NULL)
        return NULL;
    printf("Adı Soyadı:");
    gets(s);
    return s;
}
void main()
{
    char *p;
    p = getname();
    puts(p);
}
```

Gördüğünüz gibi **getname** fonksiyonu içerisinde dinamik tahsisat yapılmıştır. **getname** her çağrıldığında bellekte farklı bir bölge tahsis edilir. Oysa aynı fonksiyon **static yerel bir dizi** kullanılarak tasarlansaydı, fonksiyon her çağrıldığında öncekini silerdi. Karşılaştırmasını size bırakıyoruz.

```
char *getname(void)
{
    static char s[30];
    gets(s);
    return s;
}
```

Kendi içerisinde dinamik tahsisat yapan standart C fonksiyonları da vardır. Zamanı geldiğinde bu fonksiyonlar hakkında bilgiler bulacaksınız.

23.3.2 calloc

calloc fonksiyonu işlevsel olarak **malloc** fonksiyonuna oldukça benzer. Prototipini inceleyiniz:

```
void *calloc (unsigned count, unsigned size);
```

calloc, birinci parametresiyle belirtilen sayı ile ikinci parametresiyle belirtilen sayının çarpımı kadar (**count * size**) sürekli belleği tahsis etmek amacıyla kullanılır. Örneğin, 50 elemanlı bir **int** blok tahsis etmek isteyelim:

```
int *p;
...
p = calloc(50, sizeof(int));
if (p == NULL) {
    printf("Yetersiz bellek!...\n");
    exit(1);
}
...
```

`calloc` fonksiyonu tahsisatı yapabilmek amacıyla içerisinde `malloc` fonksiyonunu çağırır. Fakat `calloc`, **`malloc` fonksiyonundan farklı olarak tahsis ettiği bellek bloğunu sıfırlar.** İşlevsel olarak `malloc` fonksiyonundan başkaca bir farkı yoktur. Örneğin 50 elemanlı bir `int` diziyi tahsis ettikten sonra sıfırlamak istediğimizi düşünelim. Bu işlemi `malloc` fonksiyonu ile yaparsak, diziyi ayrıca döngü içerisinde sıfırlamamız gerekir.

```
int *p;
int k;
...
p = (int *) malloc(sizeof(int) * 100);
if (p == NULL) {
    printf("Yetersiz bellek!...\n");
    exit(1);
}
for (k = 0; k < 100; ++k)
    p[k] = 0;
...
```

oysa `calloc` zaten sıfırlama işlemini kendi içerisinde yapmaktadır.

```
int *p;
...
p = (int *) calloc(100, sizeof(int));
if (p == NULL) {
    printf("Yetersiz bellek!...\n");
    exit(1);
}
...
```

Sıfırlama işlemi dışında `malloc` fonksiyonu için söylenen herşey `calloc` fonksiyonu için de geçerlidir. Örneğin `calloc` fonksiyonu da DOS altında en fazla **64K** sürekli bellek tahsis edebilir. Yani iki parametresinin çarpımı **65535**'ten büyük olmamalıdır. `calloc` fonksiyonunun da geri dönüş değerinin `void *` olduğuna dikkat ediniz. Eski sistemler için yaptığımız uyarı `calloc` için de geçerlidir.

23.3.3 realloc

`realloc` daha önce `malloc` ya da `calloc` fonksiyonlarıyla tahsis edilmiş bellek bloğunu büyütmek ya da küçültmek amacıyla kullanılır. Prototipi diğer dinamik bellek fonksiyonlarında olduğu gibi `STDLIB.H` ve `ALLOC.H` içerisindedir.

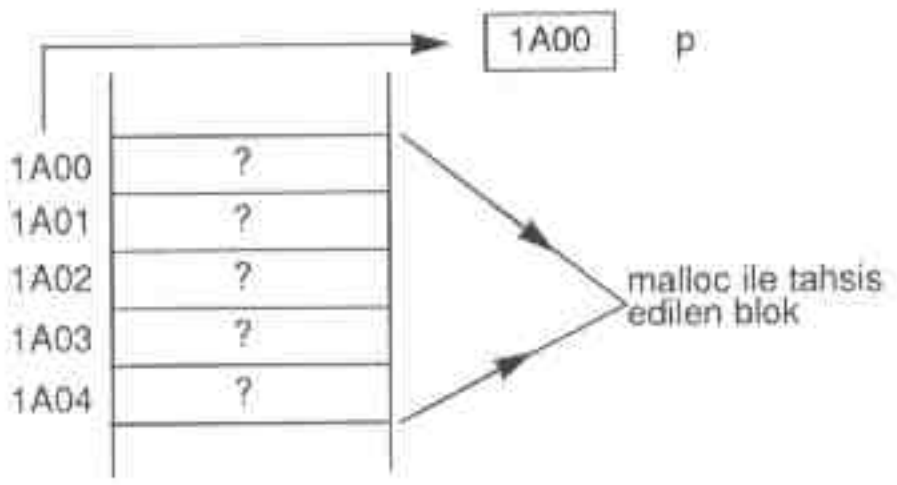
```
void *realloc(void *block, unsigned newsize);
```

`realloc` fonsiyonunun birinci parametresi, daha önce tahsis edilen bellek bloğunun başlangıç adresi, ikinci parametresi ise bloğun toplam yeni uzunluğudur. `realloc`, daha önce tahsis edilen bloğun hemen altında sürekliliği bozmayacak biçimde tahsisat yapar. Eğer daha önce tahsis edilmiş bloğun aşağısında istenilen uzunlukta sürekli yer bulamazsa `realloc` bu sefer, bloğun tamamı için bellekte başka yerler araştırır. Eğer istenilen toplam uzunlukta sürekli bir blok bulursa burayı tahsis eder ve eski değerleri taşır. Bulamazsa yapılacak bir şey yoktur; `NULL` değeriyle geri döner. Aşağıdaki örneği inceleyiniz:

```
char *p;
...
p = malloc(5);
if (!p) {
    printf("Yetersiz bellek!...\n");
    exit(1),
}
for (k = 0; k < 5; ++k)
    p[k] = 'a' + k;
...
p = realloc(p, 10);
if (!p) {
    printf("Yetersiz bellek!...\n");
    exit(1),
}
```

Örneğimizde önce `malloc` fonksiyonu ile 5 byte tahsis edilmiştir. Şunu unutmayınız: `realloc` fonksiyonu kullanılmadan önce mutlaka `malloc` ya da `calloc` ile tahsisat yapılmış olması gerekir.

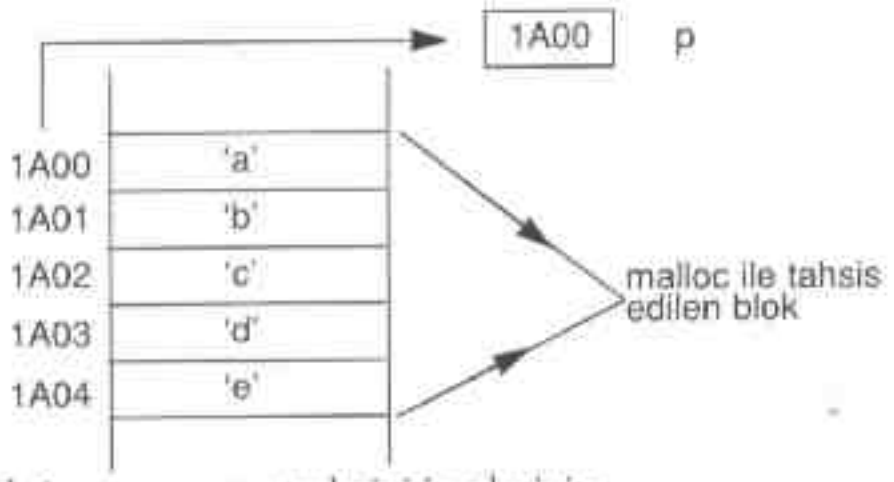
`malloc` ile tahsis edilen bloğun `1A00` adresinden başladığını varsayalım:



`for` döngüsü ile tahsis edilen diziye `'a'`, `'b'`, `'c'`, `'d'`, `'e'` değerleri veriliyor.

```
for (k = 0; k < 5; ++k)
    p[k] = 'a' + k;
```

Örneğimizde daha sonra `realloc` ile bu blok 10 byte olarak büyütülmüştür.

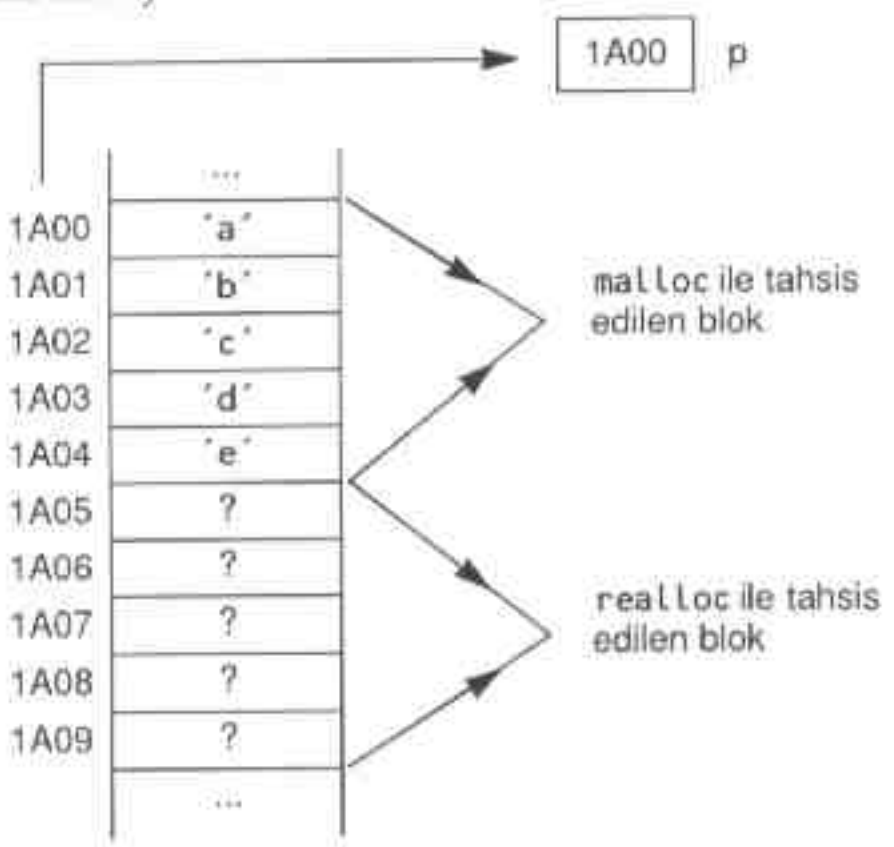


`realloc` fonksiyonun parametrelerini inceleyiniz:

```
p = realloc(p, 10)
```

Bloğun başlangıç adresi — Yeni Uzunluğu

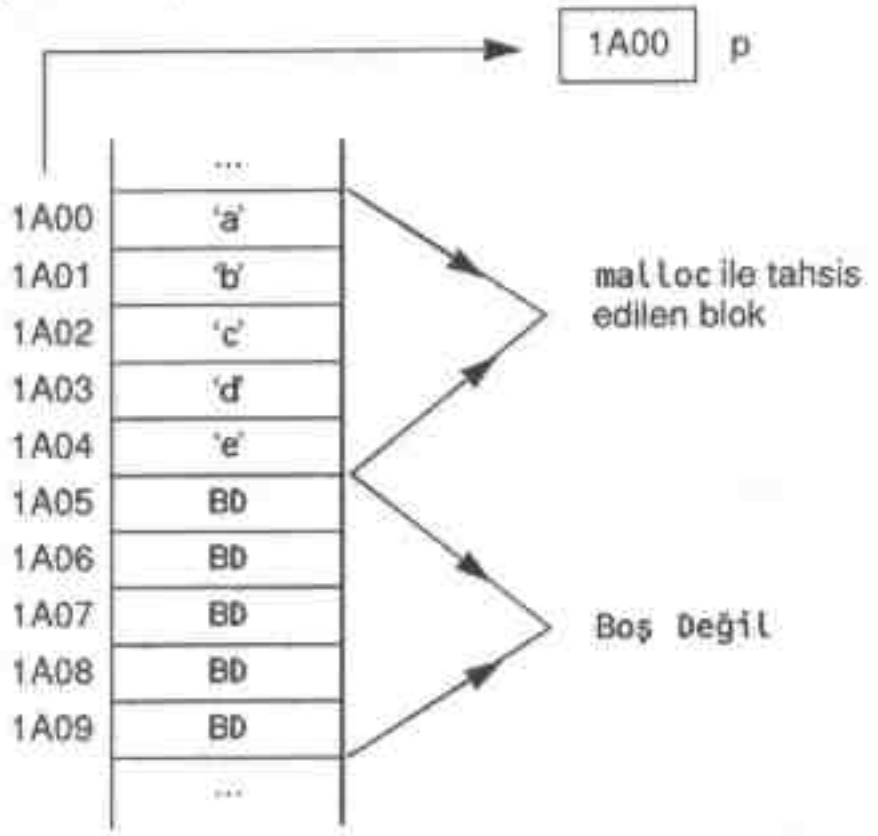
Bu durumda `realloc` daha önce tahsis edilmiş bloğun hemen altında sürekliliği bozmayacak biçimde bir 5 byte daha tahsis etmeye çalışır. Bunu başardığını düşünelim. Şekli inceleyiniz:



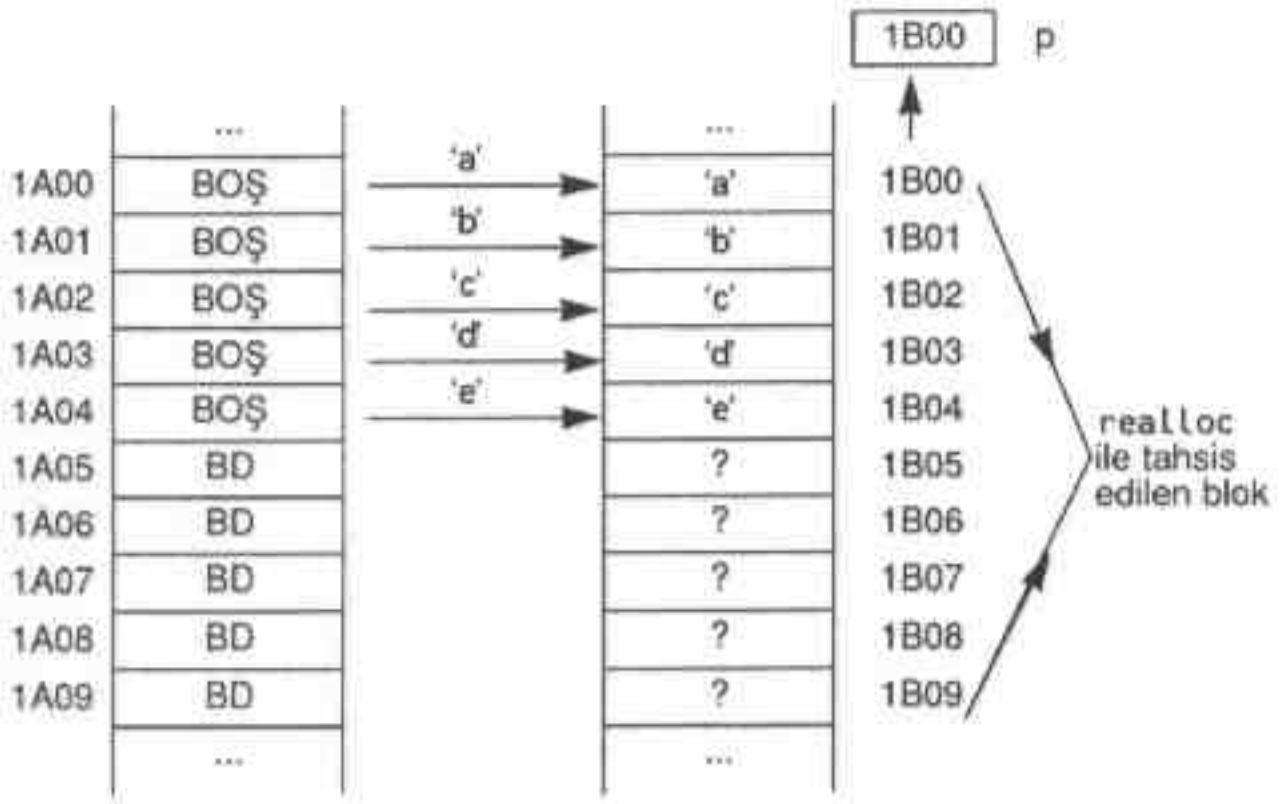
`realloc` eski bloğun altında istenilen uzunlukta boş yer bulamazsa belleğin başka yerinde toplam uzunluk kadar boş yer araştırır. Eğer böyle bir yer bulursa

burayı tahsis eder ve eski değerleri taşır. Şimdi `realloc` fonksiyonunun `malloc` ile tahsis edilen bölgenin hemen altında boş yer bulamadığını varsayalım.

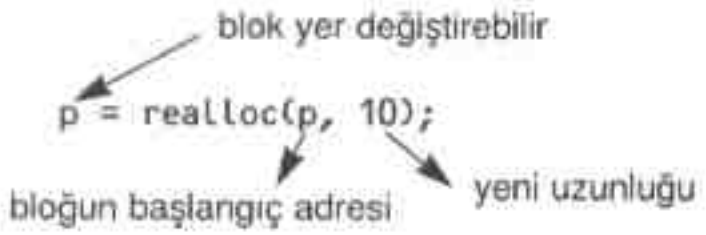
Aşağıdaki şekli inceleyiniz:



Bu durumda `realloc`, belleğin başka bir yerinde ikinci parametresiyle belirtilen uzunlukta boş ve sürekli bir bölge araştırır. Böyle bir bölge bulursa burayı tahsis eder, değerleri taşır ve eski yeri boşaltır.



Bu nedenden dolayı `realloc` fonksiyonun geri dönüş değeri göstericiye tekrar atanmak zorundadır.



`realloc` fonksiyonun tahsis ettiği blok içerisinde rastgele değerler vardır; `realloc` tahsis ettiği bloğa ilkdeğer vermez. `realloc` fonksiyonu yalnızca bellek bloğunu büyütmek amacıyla değil aynı zamanda küçültmek amacıyla da kullanılabilir. Örneğin:

```
char *p;
...
p = malloc(100);
if (p == NULL) {
    printf("Yetersiz bellek!...\n");
    exit(1);
}
...
p = realloc(p, 50);
```

burada önce `malloc` fonksiyonuyla **100** byte bellek tahsis edilmiş, daha sonra `realloc` fonksiyonu ile **50** byte'a indirilmiştir. `realloc` fonksiyonunun çağrılma biçimine bakınız:

```
p = realloc(p, 50);
```

Sizlere ilk bakışta mantıklı gelmeyebilir, fakat `realloc` daha önce tahsis edilmiş olan bloğu başka yere taşıyarak da küçültme işlemini yapabilir. Bu yüzden geri dönüş değerini yine göstericiye atamak gerekir. Çünkü blokları küçültürken de taşıma işlemini yapması, belleğin yüksek verimle kullanımı için gerekli olabilir. Tabi, küçülteceğiniz bloklar için işlemin başarısını kontrol etmenize hiç gerek yok!

Son olarak **realloc** ile dizilerin uzunluklarının değiştirilemeyeceğini vurgulayalım. Diziler zaten dinamik bellek fonksiyonlarıyla kullanılamazlar.

23.3.4 free

Bellek yönetimine dinamiklik sağlayan en önemli özellik daha önce tahsis edilmiş olan blokların istenildiğinde sisteme iade edilebilmesidir. Böylece, kullanılmayan blokların verimi düşürmesi engellenir. `free` fonksiyonunun prototipini inceleyiniz.

```
void free(void *block);
```

bu fonksiyon daha önce tahsis edilmiş olan bellek bloğunu geri boşaltarak siste-

me iade eder. Geri dönüş değerinin `void` olduğuna dikkat ediniz. (Diğerleri gibi `void *` değil yalnızca `void`). Örneğin:

```
char *p;

p = malloc(100);
if (p == NULL) {
    printf("Yetersiz bellek!...\n");
    exit(1);
}
...
free(p);
...
```

Ancak tahsis edilmiş bloklar `free` fonksiyonu ile boşaltılmamışsa programın sonlanmasıyla otomatik olarak boşaltılırlar.

23.4 SİSTEM BAĞIMLI DİNAMİK BELLEK FONKSİYONLARI

Sistem bağımlı dinamik bellek fonksiyonları sistemden sisteme ve aynı sistemde de derleyiciden derleyiciye değişebilen fonksiyonlardır. Bu bölümde DOS altında Borland ve Microsoft derleyicilerinde bulunan özel fonksiyonların birkaçı üzerinde duracağız.

`malloc, calloc` ve `realloc` fonksiyonları ile DOS altında en fazla 64K sürekli bölge tahsis edilebildiğini gördük. DOS altında daha büyük alanların tahsisi ancak derleyicilerin özel fonksiyonları ile yapılabilir. Burada örnek olarak Borland ve MicroSoft derleyicilerinde bulunan aynı işlevlere sahip `farmalloc` ve `halloc` fonksiyonlarını inceleyeceğiz.

23.7.1 farmalloc ve halloc

Borland derleyicilerinde `farmalloc`, Microsoft derleyicilerinde ise `halloc` ismiyle bulunan fonksiyonlar 64K sınırının yukarısında sürekli bellek tahsis etmek için kullanılırlar. Bu fonksiyonları `malloc` fonksiyonunun 64K sınırını aşan uyarlamaları olarak düşünebilirsiniz.

Borland:
```
void far *farmalloc(unsigned long size);
```
Microsoft
```
void huge *halloc (long num, unsigned size)
```

Yukarıdaki prototip ifadesinden de gördüğünüz gibi, `farmalloc` fonksiyonunun parametresi `unsigned long` türündendir. Her ne kadar, `unsigned long` türü ile yazılabilecek en büyük sayı 4GB olsa da, DOS'ta 640K bellek sınırı olduğu için, tahsis edilecek değer de uygulamada 640K'dan büyük olamaz. Tüm dina-

mik bellek fonksiyonlarında olduğu gibi, `farmalloc` fonksiyonu da tahsis ettiği bloğun başlangıç adresi ile geri dönmektedir. Prototipi Borland derleyicilerinde `ALLOC.H` dosyası içerisinde bulunur.

`farmalloc` fonksiyonun geri dönüş adresinin uzak gösterici olduğuna dikkat ediniz. Ancak, 64K sınırının aşılmasından dolayı tahsis edilen bölgeye dev göstericilerle erişmek daha uygun olur. Örneği izleyiniz:

```
char huge *p;
...
p = (char huge *) farmalloc(100000);
if (!p) {
    printf("Yetersiz bellek");
    exit(1);
}
```

`farmalloc` ile ne kadar alan tahsis edebildiğinizi aşağıdaki örnek programla deneyebilirsiniz:

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

void main()
{
    char far *p;
    long n;

    for (n = 655350L; n > 0; —n) {
        p =  farmalloc(n);
        if (p)
            break;
    }
    printf("En büyük alan:%ld \n", n);
}
```

Burada `farmalloc` tahsisatı yapabildiği zaman döngüden çıkılmış ve tahsisat miktarı yazdırılmıştır.

Microsoft derleyicilerinde bulunan `halloc`, işlevsel olarak `farmalloc` ile aynı olmasına karşın, parametre ve geri dönüş değeri olarak farklıdır. `halloc` fonksiyonun prototipi Microsoft derleyicilerinde `MALLOC.H` başlık dosyası içerisinde bildirilmiştir.

```
void huge *halloc (long num, unsigned size);
```

`halloc`, `num * size` kadar sürekli bellek tahsis eder. İki parametresinin çarpımı `long` türünden bir değer olduğu için 64K sınırı bu fonksiyonla aşılabilir. Geri dönüş değerinin dev gösterici olduğuna dikkat etmelisiniz.

```
int huge *p;
...
p = halloc(50000L, sizeof(int));
if (!p) {
    printf("Yetersiz bellek...\n");
    exit(1);
}
...
```

`farmalloc` ve `halloc` fonksiyonlarının dışında sisteminizde 64K sınırının ötesinde tahsisat işlemi yapan başka fonksiyonlar da olabilir. Biz yalnızca Borland ve Microsoft derleyicilerindeki isimleriyle onları belirtip geçeceğiz.

Borland:
`farcalloc, farrealloc, farfree, allocmem, freemem, coreleft`
MicroSoft:
`hfree, _ffree, _fmalloc, _heapwalk, _heapmin`

SORAMADIKLARINIZ...

S1) Bellek problemi olmayan Windows 95 ve Unix sistemlerinde en fazla ne kadar dinamik tahsisat yapılabilir? Maksimum tahsisat miktarı o anda boş olan RAM miktarı kadar mıdır?

C1) Windows 95 ve Unix işletim sistemleri 80x86 mimarisinde korumalı modda çalışırlar. Korumalı mod sanal bellek kullanımını mümkün kılmaktadır. Sanal bellek, fiziksel RAM'in yeterli olmadığı durumlarda programların çeşitli parçalarının diskte tutularak diskin RAM gibi kullanılması ile karakterize olan bir bellek yönetim tekniğidir. Bu nedenle sanal bellek kullanılan sistemlerde o anda boş olan fiziksel RAM miktarının çok ötesinde dinamik tahsisatlar yapılabilir.

S2) Bir programda `malloc` ile yüksek miktarda dinamik bellek tahsis edilmiş ancak `free` fonksiyonu ile boşaltılması unutulmuş ise bu durumun programın çalışması bitince diğer programların kullandığı bellek üzerinde azaltıcı etkisi olur mu?

C2) Dinamik bellek fonksiyonlarıyla tahsisat işlemi işletim sistemi seviyesinde değil, program için tahsis edilen bellek alanı içerisinde içsel olarak yapılmaktadır. Bu nedenle `free` ile boşaltma işlemi unutulmuş olsa bile tahsis edilmiş olan alan programın sonlanmasıyla zaten bellekten boşaltılmış olur. Yani sonradan çalışacak programlar için herhangi bir etkisi olmaz.

GÖSTERİCİ DİZİLERİ GÖSTERİCİLERİ GÖSTEREN GÖSTERİCİLER VE FONKSİYON GÖSTERİCİLERİ

2

Bu bölümde karmaşık gösterici konuları ele alınmaktadır. Göstericilerin oluşturduğu diziler, göstericileri gösteren göstericiler ve fonksiyon göstericiler seyrek kullanılsalar da önemli işlevlere sahiptirler. Kavramsal karmaşıklığı nedeniyle konuları anlamakta zorluk çekebilirsiniz; bu normaldir. Ancak anlayamadığınız noktaların peşini bırakmamalısınız.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Gösterici dizileri nasıl bildirilir?

2) Normal dizilerle gösterici dizileri arasındaki benzerlikler nelerdir?

3) Gösterici dizilerine nasıl ilkdeğer verilir?

4) Göstericileri gösteren göstericiler nasıl bildirilirler ve nerede kullanılırlar?

5) Gösterici dizileriyle göstericileri gösteren göstericiler arasındaki benzerlikler nelerdir?

6) Fonksiyon göstericileri nasıl bildirilirler?

7) Fonksiyon göstericileri hangi amaçla kullanılırlar?

8) Fonksiyon göstericilerine bilinçli tür dönüştürmeleri nasıl yapılmaktadır?

24.1 GÖSTERİCİ DİZİLERİNİN BİLDİRİMİ

Elemanları göstericilerden oluşan dizilere "gösterici dizileri" denir. Bildirimleri aşağıdaki biçimde yapılır:

```
<tür>  * <dizi_ismi> [dizi_uzunluğu];
```

Örneğin:

```
char *s[10];
int *sample[50];
float *kilo[10];
....
```

Gösterici dizilerinin bildirimleri normal dizi bildirimlerinden farklı olarak * operatörü ile yapılmaktadır.

```
char s[10];
```

her elemanı karakter olan bir diziyi belirtir. Oysa:

```
char *s[10];
```

her elemanı karakter göstericisi olan bir diziyi belirtmektedir. Derleyici bir gösterici dizisinin bildirimi ile karşılaşınca diğer dizilerde yaptığı gibi, bellekte belirtilen sayıda göstericiyi tutabilecek kadar sürekli yer tahsis eder. Örneğin:

```
char *s[10];
```

bildirimi ile derleyici, s dizisinin 10 elemanlı ve her elemanın **char** türünden bir gösterici olduğunu anlar. **near** ya da **far** anahtar sözcükleri kullanılmadığına göre DOS ve Windows 3.1 altında yapılacak tahsisat memory modele göre değişecektir. Böylece derleyici dizi için eğer **near** modellerden birinde çalışılıyorsa 20 byte, **far** modellerden birinde çalışılıyorsa 40 byte sürekli bir bölge tahsis eder. Bu durumda;

```
s[0], s[1], s[2],..s[9]
```

dizi elemanlarının herbiri karakteri gösteren birer göstericidir. Elemanların hepsi diğerlerinden bağımsız olarak kullanılabilir. Aşağıdaki örneği inceleyiniz:

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

#define MAX 10

void main()
{
    char *s[MAX];
    int k;
```

```
    for (k = 0; k < MAX; ++k) {
        s[k] = malloc(30);
        if (s[k] == NULL) {
            printf("Yetersiz bellek!...\n");
            exit(1);
        }
        printf("Adı soyadı:");
        gets(s[k]);
    }
    for (k = 0; k < MAX; ++k) {
        printf("%s\n", s[k]);
        free(s[k]);
    }
}
```

Bu örnekte her elemanı karakteri gösteren 10 elemanlı bir gösterici dizisi açılmış ve dizinin her elemanı için döngü içerisinde `malloc` ile 30'ar byte tahsis edilmiştir. Tahsis edilen blokların başlangıç adresleri dizi elemanlarına yerleştirildikten sonra `gets` fonksiyonu ile klavyeden girilen karakterlerin tahsis edilen bloklara aktarıldığını görüyorsunuz. `s` yerel bir dizi olduğuna göre içerisinde rastgele değerler olduğunu vurgulayalım. Dolayısıyla `malloc` ile yer ayırmadan yapılan veri aktarımları gösterici hatasına yol açacaktır.

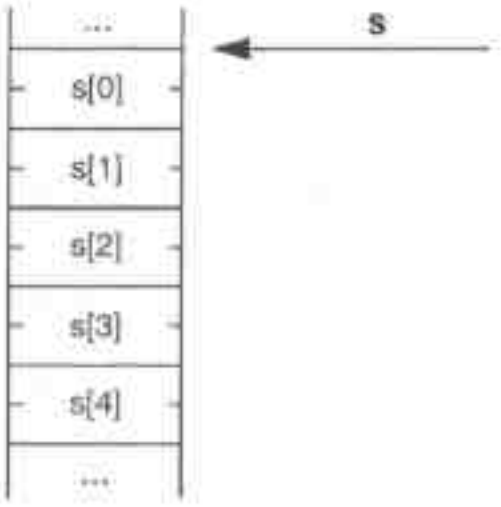
80X86 DOS ve Windows 3.1 altında gösterici dizilerinin elemanları **yakın** ya da **uzak** göstericiler olabilir. Örneğin:

```
char far *s[10];
char near *p[10];
...
```

Yukarıdaki ilk bildirimde derleyici 10 elemanlı gösterici dizisinin her elemanını karakteri gösteren bir uzak gösterici, ikinci bildirimde ise karakteri gösteren bir yakın gösterici olarak değerlendirir. Böylece `s` dizisi için `10 * 4 = 40` byte, `p` dizisi için ise `10 * 2 = 20` byte yer ayıracaktır.

Dizi isimlerinin dizilerin başlangıç adreslerini gösterdiğini anımsayalım. Bu durumda gösterici dizisine ilişkin isimler de gösterici dizilerinin başlangıç adresleri olacaktır, değil mi? Örneğin:

```
char *s[5];
```



Peki gösterici dizilerinin isimleri hangi türden adreslerdir? Örneğin:

```
char s[20];
```

bildiriminde s karakteri gösteren bir adres (çünkü *s char türünden),

```
int p[10];
```

bildiriminde p, int türünü gösteren bir adres (çünkü *p int türünden). Peki

```
char *t[5];
```

bildiriminde t hangi türü gösteren bir adrestir dersiniz? Bu sorunun yanıtı *t nesnesinin hangi türden olduğuyla verilebilir. *t karakteri gösteren bir gösterici olduğuna göre, t adresinin türü de **"karakteri gösteren göstericiyi gösteren"** bir adres türüdür. Tür dönüştürme operatörüyle şöyle belirtilir:

```
(char **)
```

24.2 GÖSTERİCİ DİZİLERİNE İLKDEĞER VERİLMESİ

Normal dizilere ilkdeğer nasıl veriliyorsa gösterici dizilerine de ilkdeğer aynı biçimde verilir. Örneğin:

```
int *p[] = {
    (int *) 0x1FC0,
    (int *) 0x1FC2,
    (int *) 0x1FC4,
    (int *) 0x1FC6 ,
    (int *) 0x1FC8
    );
```

Kuşkusuz ki, gösterici dizisine atanan adreslerin bir amaç doğrultusunda kullanılabilmesi için güvenli bölgeleri göstermesi gerekir. Bu nedenle uygulamada karakter gösterici dizisi dışındaki gösterici dizilerine ilkdeğer verilmesi durumu-

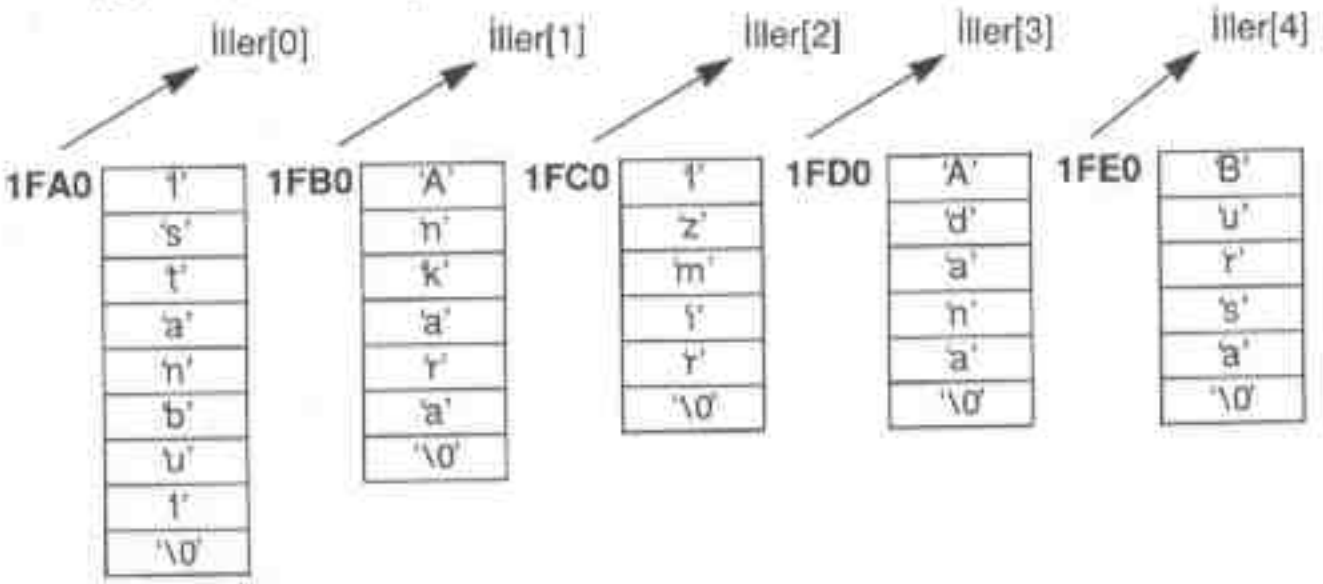
na pek rastlanmaz. **Gösterici dizilerine ilkdeğer verme işlemi genellikle karakter gösterici dizilerine string ifadeleriyle ilkdeğer verilmesi biçiminde karşımıza çıkmaktadır.**

24.2.1 Karakter Gösterici Dizilerine String İfadeleriyle İlkdeğer Verilmesi

Stringler derleyici tarafından belleğin güvenli bir bölgesine yerleştirildikten sonra gösterici dizilerine ilkdeğer olarak atanabilirler. Örneğin:

```
char *iller[] = {"İstanbul", "Ankara", "İzmir", "Adana", "Bursa"
};
```

Aşağıdaki şekli inceleyiniz:



Bu örnekte, derleyicinin stringleri sırasıyla: **1FA0, 1FB0, 1FC 0, 1FD0, 1FE0** adreslerinden başlayarak yerleştirdiği varsayılmıştır. Stringlerin yerleşimi konusunda hiçbir aritmetik ya da fiziksel ilişki söz konusu olmak zorunda değildir. Bu işlemlerden sonra dizinin tüm elemanlarını bir döngü içerisinde yazdırabiliriz.

```
...
for (k = 0; k < 5; ++k)
    printf("%s\n", iller[k]);
...
```

Karakter gösterici dizileri hata mesajlarının saklanması ve bunların bildirilmesi amacıyla oldukça sık kullanılmaktadır. Hataların yönetimi için önce hata mesajları belirlenir ve global bir karakter gösterici dizisinde saklanır.

```
char *err[] = {"Bellek Yetersiz!", "Hatalı şifre",
"Dosya bulunamadı", "Belirtilen dosya zaten var",
"Sürücü hazır değil", "Belirlenemeyen hata!"
};
```

Daha sonra ilgili hataya yanıt verecek olan bir hata fonksiyonu tasarlanır. Aşağıda basit bir örnek görüyorsunuz:

```
void disperr(int errno)
{
    printf("Hata: %s (%d)\n", err[errno], errno);
    if(errno == 0 || errno == 1 || errno == 5) {
        exit(errno);
    return;
}
```

disperr fonksiyonu ilgili hatayı ekrana yazar; eğer hatalar 0, 1 ya da 5 numaralı hatalardan bir tanesi ise programa da son verilmektedir. Şimdi program içerisinde bunun nasıl kullanıldığını inceleyelim:

```
...
gets(password);
if (strcmp(password, "snxal13"))
    disperr(1);
...
p = malloc(100);
if (!p)
    disperr(0);
...
```

Örneğimizde önce klavyeden **gets** fonksiyonu ile bir şifre alınmış; şifre uyuşmazlığı durumunda program sonlandırılmıştır. Daha sonra **malloc** fonksiyonunun kullanıldığını görüyorsunuz; tahsisat başarısızlıkla sonuçlanırsa **disperr** fonksiyonu ile ekrana:

```
Hata:Yetersiz bellek (0)
```

yazılarak program sonlandırılır.

Aşağıdaki örnekte de aylar bir karakter gösterici dizisi içerisinde tutularak **dispdate** fonksiyonu tarafından yazdırılmaktadır.

```
#include <stdio.h>

char *months[12] = {
"Ocak", "Şubat", "Mart", "Nisan",
"Mayıs", "Haziran", "Temmuz", "Ağustos",
"Eylül", "Ekim", "Kasım", "Aralık"
};

void dispdate(int g, int a, int y)
{
    printf("%02d-%s-%04d\n", g, months[a-1], y);
}

void main(void)
{
    dispdate(10, 11, 1995);
}
```

24.3 GÖSTERİCİLERİ GÖSTEREN GÖSTERİCİLER

Bir göstericinin gösterdiği yerde başka bir gösterici de olabilir. Böyle göstericilere göstericileri gösteren göstericiler denilmektedir. Bildirimleri şöyle yapılır:

```
<göstericinin türü> ** <gösterici ismi>;
```

Örneğin:

```
char **p;
```

Bu bildirimden şunlar anlaşılır:

1) p bir göstericiyi gösteren göstericidir.

p bir göstericiyi gösteren göstericidir

```
char * * p;
```

2) p'nin gösterdiği yerde bulunan nesne yani *p, karakter türünden bir göstericidir.

*p karakter türünden göstericidir

```
char * [* p] ;
```

3) **p karakter türünden bir nesnedir.

**p karakter türünden bir nesnedir

```
char [* * p] ;
```

Yukarıdaki tanımlama işleminde eğer p yerel ise içerisinde rastgele bir değer vardır. Yani p'ye güvenli bir adres atanması gerekir. Bu işlem dinamik bellek fonksiyonlarıyla çalışma zamanı sırasında yapılabilir. Tahsisatın nasıl yapıldığını inceleyiniz:

```
p = (char **) malloc(sizeof(char *));
```

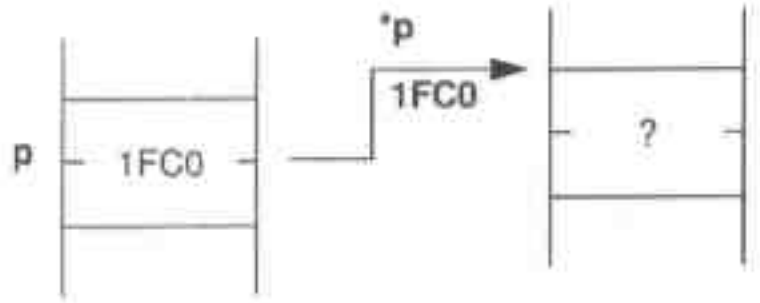
Tür dönüştürme operatörleri içerisindeki ifadeyi inceleyiniz. Bu ifadenin karakter göstericisi için (char *), karakter türünden göstericiyi gösteren gösterici için (char **) ise biçiminde olduğuna dikkat ediniz. p göstericisinin gösterdiği yerde bir karakter göstericisi olduğuna göre *p için tahsis edilecek yerin uzunluğu da sizeof(char *) ile belirtilir değil mi?

Karakter türünden göstericiyi gösteren gösterici türüne dönüştürülüyor

Karakter göstericisinin uzunluğu

```
p = (char * *) malloc(sizeof (char *));
```

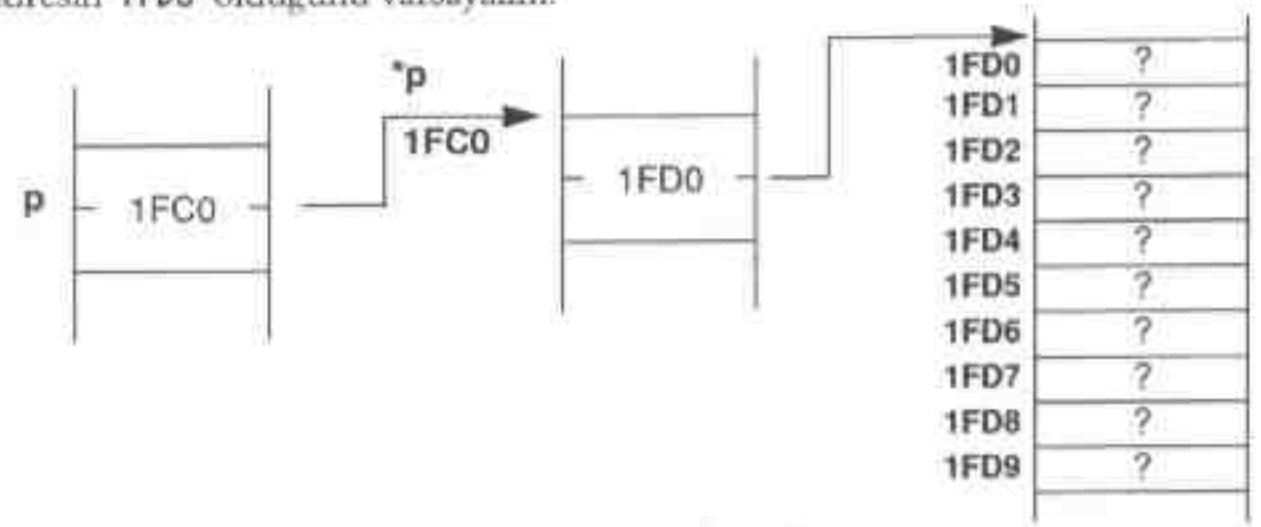
Örneğin malloc ile tahsis edilen bölge 1FC0 olsun. Artık p güvenli bir bölgeyi göstermektedir.



*p içerisinde de rastgele bir değer olduğuna dikkat ediniz. Bu durumda **p ile gösterilen nesne de rastgeledir değil mi? O halde *p için de dinamik tahsisat yapmamız gerekiyor.

```
*p = (char *) malloc(10);
```

ile *p içerisine tahsis edilen 10 byte'lık alanın başlangıç adresi yerleştirilir. Bu adresin 1FD0 olduğunu varsayalım.



Artık *p'de güvenli bir biçimde kullanılabilir. Örneğin:

```
gets(*p);
```

ile yeni tahsis edilen 10 byte'lık alana bilgi aktarabilirsiniz.

Aşağıdaki örnekte göstericiyi gösteren gösterici yardımıyla stringleri tutan bir dizi elde edilmiştir. Aynı örneği gösterici dizilerinde de vermiştik, karşılaştırınız.

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

#define MAX 10

void main()
{
    int k;
    char **p;

    p = (char **) malloc(sizeof(char *) * MAX);
    if (p == NULL) {
        printf("Yetersiz bellek!..\n");
        exit(1);
```

```
    }
    for (k = 0; k < MAX; ++k) {
        p[k] = (char *) malloc(30);
        if (p[k] == NULL) {
            printf("Yetersiz bellek!..\n");
            exit(1);
        }
        printf("Adı soyadı:");
        gets(p[k]);
    }
    for (k = 0; k < MAX; ++k) {
        printf("%s\n", p[k]);
        free(p[k]);
    }
    free(p);
}
```

Örneğimizde önce p göstericisi için MAX kadar yer ayrılmıştır.

```
p = (char **) malloc (sizeof(char *) * MAX);
```

artık p[k] ifadelerinin hepsi karakter türünden bir göstericidir. p[k]'lar için de dinamik olarak yer ayrıldığına dikkat ediniz.

Yerel bir göstericinin bir fonksiyon tarafından değiştirilmesi için fonksiyona parametre olarak o göstericinin adresinin gönderilmesi gerekir. Bu durumda gösterici adresinin kopyalanacağı parametre değişkeninin de göstericiyi gösteren gösterici olması gerekir. Aşağıdaki örneği inceleyiniz:

```
void fonk(char **p)
{
    ...
}

void main(void)
{
    char *s;
    ...
    fonk(&s);
}
```

Burada s karakter türünden bir gösterici olduğu için &s ifadesi de karakter türünden göstericiyi gösteren bir adrestir.

Gösterici dizileriyle göstericileri gösteren göstericiler arasında benzerlik göstericilerle normal diziler arasındaki benzerlik gibidir. Aşağıdaki bildirimleri inceleyiniz:

```
char **p;
char *s[10];
```

Bir kere s ile p adreslerinin türü aynıdır. Bu tür (char **) ile temsil edilir. Yani:

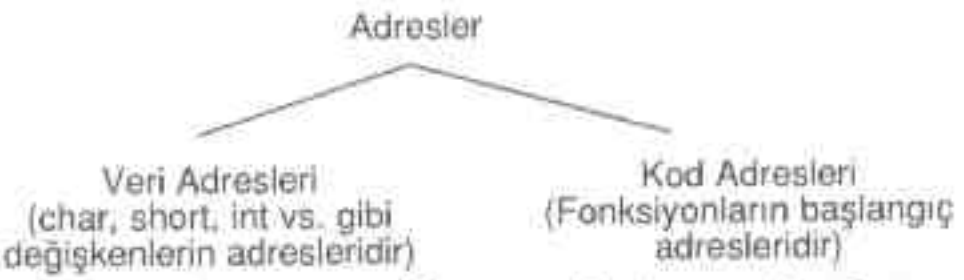
```
p = s;
```

atamasında tür uyuşumu tam olarak sağlanır. Dikkat ediniz, p bir nesne olduğu halde dizi ismi olan s bir nesne değildir. Ayrıca s için 10 göstericilik yer ayrıldığı halde p'nin gösterdiği yer için bir tahsisat yapılmamıştır.

24.4 FONKSİYON GÖSTERİCİLERİ

Program kodları da değişkenler gibi bellekte yer kapladığına göre onların da adresinden söz edilebilir. Bir fonksiyonun adresi diye o fonksiyona ilişkin kodun bellekteki başlangıç adresi anlatılmaktadır.

Genel olarak adresleri veri adresleri ve kod adresleri olmak üzere iki bölüme ayırabiliriz.



24.5 FONKSİYON GÖSTERİCİLERİNİN BİLDİRİMLERİ

Bir fonksiyon göstericisi, gösterdiği fonksiyonun parametre ve geri dönüş değerlerinin türü belirtilerek bildirilir. Genel biçimi aşağıdaki gibidir:

```
<geri dönüş değerinin türü> (* gösterici ismi) ([parametre türleri]);
```

Örneğin:

```
int (*p) (void);
```

biçiminde bildirilmiş p göstericisinin içerisine geri dönüş değeri int ve parametresi void olan bir fonksiyonun adresi konulabilir. Benzer biçimde:

```
int (*s) (int, int);
```

biçiminde tanımlanmış olan s göstericisine, geri dönüş değeri int parametreleri int, int olan bir fonksiyonun başlangıç adresi konulabilir. Fonksiyon göstericilerinin bildiriminde * operatörünün parantez içerisinde yazıldığına dikkat ediniz. Eğer parantezleri unutursanız ifade derleyici tarafından fonksiyon göstericisi olarak değil fonksiyon prototipi olarak ele alınır. Örneğin:

```
int *s (int, int);
```

ifadesi geri dönüş değeri **int** türünden bir gösterici, parametreleri int, int

olan s isimli fonksiyonun prototipidir.

Fonksiyon göstericileri de **80X86** mimarisinde **near** ya da **far** olabilir. Fonksiyon göstericilerinde **near** ya da **far** anahtar sözcükleri parantezler içerisine yazılmalıdır. Örneğin:

```
int (far *p) (void);
```

gibi bir bildirimle **80X86** mimarisini kullanan **DOS** ve **Windows 3.1** sistemlerinde p için 4 byte yer ayrılır. Genel biçimden gördüğünüz gibi fonksiyon göstericisini bildirirken parametre türleri hiç yazılmayabilir. Bu durum göstericinin gösterdiği fonksiyonun parametre almadığı anlamına gelmez; derleyici tarafından parametre kontrolünün yapılmayacağı anlamına gelir. Örneğin:

```
int (*p) ();
```

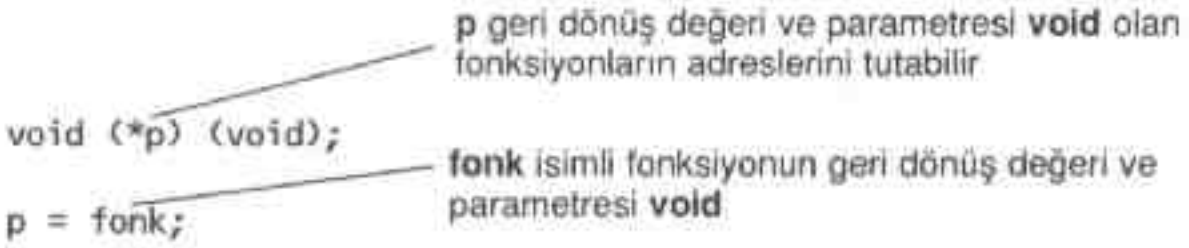
gibi bir bildirim ile derleyici p göstericisi ile belirtilen fonksiyonun geri dönüş değerinin int olduğunu ancak parametrelerinin dikkate alınmayacağını gösterir.

24.6 FONKSİYON İSİMLERİ ve FONKSİYON ADRESLERİ

Nasıl bir dizinin ismi dizinin bellekteki başlangıç adresini gösteriyorsa bir fonksiyonun ismi de o fonksiyonun bellekteki başlangıç adresini göstermektedir. Fonksiyon isimleri aynı türden bir fonksiyon göstericisine atanabilirler. Aşağıdaki örneği inceleyiniz:

```
void fonk(void)
{
    printf("Merhaba\n");
}

void main(void)
{
    void (*p) (void);

    p = fonk;
    ...
}
```

Yukarıdaki örnekte fonk ismi yalnız başına bu fonksiyonun bellekteki başlangıç adresini gösteren bir kod adresidir. **fonk** fonksiyonun geri dönüş değeri ve parametresi **void** olduğundan bu adres geri dönüş değeri ve parametresi void olan fonksiyonların adreslerini tutabilen bir göstericiye atanmıştır.



Aşağıdaki örneği inceleyiniz:

```
int topla(int a, int b)
{
    return a + b;
}

void main(void)
{
    int (*p) (int, int);

    p = topla;
    ...
}
```

Burada p geri dönüş değeri int parametreleri int, int olan fonksiyonların adreslerini tutacak biçimde tanımlanmıştır. topla isimli fonksiyon da int geri dönüş değerine ve int, int parametreye sahip olduğu için adresi bu göstericinin içerisine atanabilir. Atama işlemini inceleyiniz:

```
p = topla;
```

yani şöyle değil!

```
p = topla(a, b);
```

Bu ifade topla fonksiyonunun geri dönüş değerinin p göstericisine atanacağı anlamına gelir değil mi?

Bir fonksiyon göstericisine geri dönüş değeri ve parametresi farklı olan bir fonksiyonun adresi atanmamalıdır. Bu duruma C derleyicileri tür uyuşumunun sağlanmadığını gösteren bir uyarı mesajıyla karşılık verirler. Örneğin yukarıdaki topla fonksiyonunun adresini void (*p) (void) biçiminde tanımlanmış bir göstericiye atayamazsınız.

```
void (*p) (void);
...
p = topla; /* Hata! p geri dönüş değeri ve parametresi void olan
              fonksiyonların başlangıç adreslerini tutabilir */
```

Tür dönüştürme operatörü fonksiyon göstericileri için de kullanılabilir. Aşağıdaki örneği inceleyiniz:

```
int sample(void)
{
    ...
}
void main()
{
    void (*p) (void);
    ...
    p = (void (*) (void)) sample;
    ...
}
```

sample geri dönüş değeri int ve parametresi void olan bir fonksiyonun adresidir, değil mi? Oysa p göstericisinin içerisine geri dönüş değeri void ve parametresi void olan bir fonksiyonun adresi konulabilir. Ancak bu işlem tür dönüştürme operatörü kullanılarak bilinçli bir biçimde yapılmıştır; inceleyiniz:

Tür dönüştürme operatörünün içerisindekileri dikkatle inceleyiniz.

Şimdi 0xFC10 sayısını geri dönüş değeri ve parametresi void olan bir fonksiyon adresine dönüştürerek fonsiyon göstericisine atayalım:

```
void (*p) (void);
...
p = (void (*) (void)) 0xFC10;
```

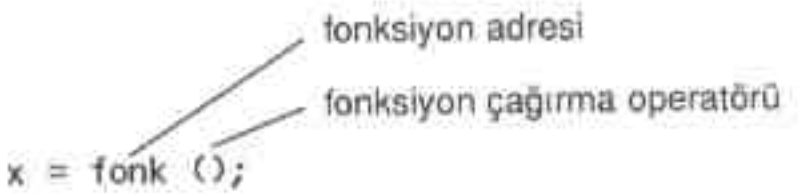
Fonksiyon göstericilerini tanımlarken parametre türleri yazılmazsa derleyici tür uyuşumunu sağlamak için parametre kontrolünü yapmaz. Yani:

```
void (*p) ();
```

biçiminde tanımlanmış bir p göstericisine atanan fonksiyon adreslerinin parametre türleri kontrol edilmez.

24.7 FONKSİYON ÇAĞIRMA OPERATÖRÜ ve FONKSİYON GÖSTERİCİLERİ İLE FONKSİYON ÇAĞIRMA

Fonskiyon çağırmak için kullandığımız (...) aslında tek operand alan sonek bir operatördür. Operand olarak bir kod adresi alır ve programın akışını o adrese yönlendirir. Örneğin:



işleminde (...) operatörünün operandı olan **fonk** bir kod adresidir.

Bir fonksiyonun adresi fonksiyon göstericisine atandıktan sonra fonksiyon çağırma operatörü ile göstericinin gösterdiği fonksiyon çağrılabilir. Aşağıdaki örneği inceleyiniz:

```
void fonk(void)
{
    printf("Deneme!..\n");
}

void main(void)
{
    void (*p) (void);

    p = fonk;
    p();
}
```

Burada

```
p ();
```

ifadesinde **p** bir fonksiyon göstericisidir. **p** göstericisini kullanılarak göstericinin gösterdiği fonksiyon

```
(*p) ();
```

biçiminde de çağrılabilir. C derleyicileri bu çağırma biçimine de izin verirler. Eğer siz özellikle **p**'nin bir fonksiyon göstericisi olduğunu vurgulamak istiyorsanız bu biçimi tercih etmelisiniz. Yoksa,

```
p ();
```

biçimindeki çağırma daha doğaldır.

80X86 tabanlı mikroişlemciler reset edildiğinde **FFFF : 0000** adresinden çalışmaya başlarlar. Bilgisayarınız açıldığında yapılan self test işlemleri ve işletim sisteminin yüklenmeye başlatılması EPROM içerisindeki giriş programı sayesinde yapılmaktadır. Özetle programımızın akışını **FFFF:0000** segment ve offset değerine gönderdiğimizde bütün işlemler bilgisayar açıldığındaki gibi yeniden yapılacağından reset işlemine girilmiş olur. Sistemi reset eden C programını bir fonksiyon göstericisi yardımıyla tasarlayabiliriz.

```
#include <stdio.h>

void main(void)
{
    void (far *reset) (void) = (void (far *) (void)) 0xFFFF0000;

    reset();
}
```

Örneğimizde `reset` isimli fonksiyon göstericisine reset adresi aktarılmıştır. `reset`'in uzak bir gösterici olduğuna dikkat ediniz. `far` anahtar sözcüğü parantez içerisine yazıldığına dikkat ediniz.

80X86 Sembolik Makina Dili Programcısına Not: 80X86 sistemi reset edildiğinde işlemcinin türüne göre `CS:IP` çiftlerinin değerleri farklı olabilir. Örneğin klasik 8086'da `CS:IP` değerleri `FFFF:0000` iken 80386, 80486 ve Pentium işlemcilerinde `F000:FFF0` biçimindedir. Ancak her iki işlemci grubunda da başlangıç adresinin `FFFF0` olduğuna dikkat ediniz. Sistem reset edildiğinde EPROM içerisindeki başlangıç kodu self test işlemini yaparak BIOS kesmeleri için kesme vektörünü oluşturur. İşletim sisteminin yüklenmek üzere aranması da ismine "bootstrap" program denilen ve EPROM'da bulunan bir kod tarafından yapılmaktadır. Bu kod disk ya da sürücüde bulunan disketin boot sektörünü belleğe okuyarak yüklemeyi oradaki "bootstrap" programa bırakır.

24.8 PARAMETRELERİ VE GERİ DÖNÜŞ DEĞELERİ FONKSİYON GÖSTERİCİSİ OLAN FONKSİYONLAR

Fonksiyon göstericileri fonksiyonların parametresi ve geri dönüş değeri olarak kullanılabilirler. Örneğin:

```
double fonk (int x, int y, int (*p) (int, int))
{
        ...
}
```

biçiminde tanımlanmış olan `fonk` isimli fonksiyonun 3. parametresi geri dönüş değeri `int` parametreleri de `int, int` biçiminde olan bir fonksiyonun adresidir. Benzer biçimde bir fonksiyonun geri dönüş değeri de bir fonksiyon göstericisi olabilir. Bu durumda fonksiyonun nasıl tanımlandığına bakınız.

```
int (* fonk (void)) (void)
{
    ...
}
```

`fonk` isimli fonksiyonun parametresi `void` ve geri dönüş değeri "parametresi `void` geri dönüş değeri `int` olan" bir fonksiyonun adresidir.

Böyle bir fonksiyonun geri dönüş değeri aynı türden bir fonksiyon göstericisine atanabilir.

```
int (*p) (void);
...
p = fonk();
```

24.9 KARMAŞIK BİLDİRİMLER

Fonksiyon göstericileriyle ilgili bildirimler bazen çok karmaşık olabilirler. Örneğin aşağıdaki fonksiyonun geri dönüş değeri ve parametre türleri nasıldır?

```
int  (*(*fonk(void)) (void)) (void)
{
      ...
}
```

Tanımlama ifadesini içten dışa adım adım inceleyelim:

`(*fonk (void))`	`fonk`'un parametresi `void` ve geri dönüş değeri bir fonksiyon adresidir.
`(*fonk(void)) (void)`	geri dönüş değerine ilişkin fonksiyonun parametresi `void`
`(*(*fonk(void)) (void))`	geri dönüş değerine ilişkin fonksiyonun geri dönüş değeri de bir fonksiyon adresi
`(*(*fonk(void)) (void)) (void)`	geri dönüş değerine ilişkin olan fonksiyonun geri dönüş değeri parametresi `void` olan bir fonksiyona ilişkindir.

ve nihayet:

`int (*(*fonk(void)) (void)) (void)`	geri dönüş değerine ilişkin fonksiyonun geri dönüş değeri, geri dönüş değeri `int` ve parametresi `void` olan bir fonksiyona ilişkindir.

Böyle bir fonksiyonun adresini aynı türden göstericiye atayabiliriz. O gösterici de şöyle tanımlanabilir:

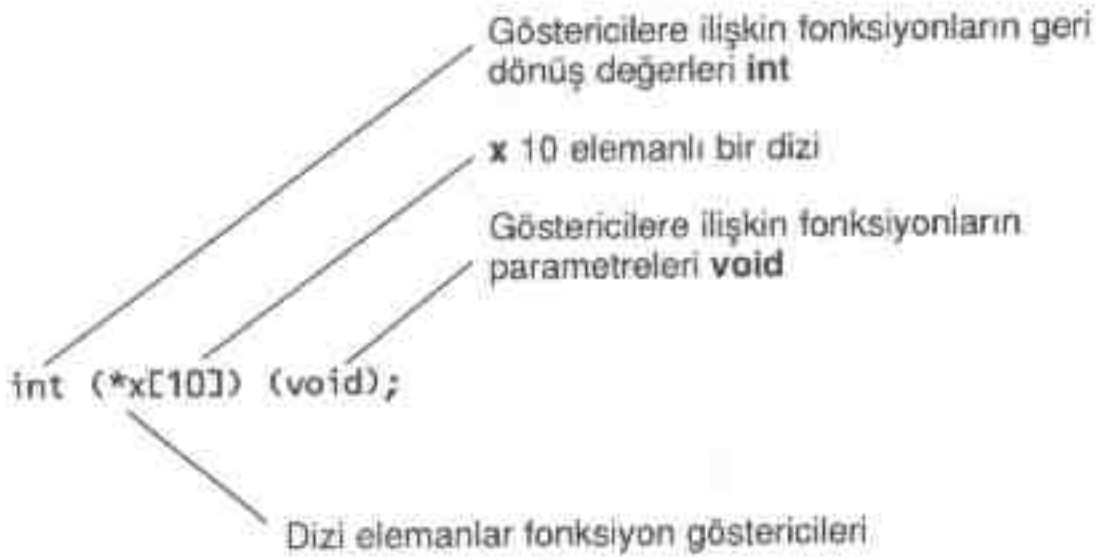
```
int  (* (* (*p) (void)) (void)) (void);
```

Nasıl? Gerçekten karmaşık değil mi?..

Şimdi de elemanları fonksiyon göstericilerinden oluşan gösterici dizilerinin nasıl tanımlandıklarına bakalım. Örneğin 10 elemanlı ve elemanları "geri dönüş değeri `int` ve parametresi `void` olan" fonksiyon göstericilerinden oluşan bir gösterici dizisi nasıl tanımlanır? İnceleyiniz:

```
int (*x[10]) (void);
```

ifade sizi şaşırtabilir. * operatörünün parantez içinde olması dizinin türünün fonksiyon göstericisi olduğu anlamına geliyor.



SORAMADIKLARINIZ...

S1) Bir göstericinin gösterdiği yerde bir göstericiyi gösteren gösterici bulunabilir mi? Başka bir ifadeyle göstericiyi gösteren göstericiyi gösteren gösterici tanımlanabilir mi?

C1) Evet böyle bir gösterici aşağıdaki gibi tanımlanabilir.

```
char ***p;
```

Ancak bu biçimde tanımlanmış bir göstericiyi uygulamada kullanabileceğimiz bir durum bulmak da oldukça güç. Aslında tanımlamada daha fazla * da biraraya getirilebilir. Örneğin:

```
char *****p;
```

gibi bir tanımlama da geçerlidir. Ancak dediğimiz gibi böylesi bir göstericinin uygulamada kullanılması söz konusu değildir.

25 YAPILAR

C programlarının en önemli elemanlarından biri olan yapılar bölüm içerisinde ayrıntılı bir biçimde ele alınmaktadır.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Yapıların bildirimi nasıl yapılmaktadır?
2) Yapı değişkenleri nasıl tanımlanır?
3) Nokta operatörünün işlevi nedir?
4) Yapı elemanlarının bellekteki yerleşim biçimi nasıldır?
5) Yapı değişkenlerine nasıl ilkdeğer verilir?
6) Yapılar fonksiyonlara nasıl parametre olarak geçirilirler?
7) Yapılar ve diziler arasındaki benzerlikler ve ayrılıklar nelerdir?
8) Ok operatörünün işlevi nedir?

25.1 GİRİŞ

Aralarında mantıksal bir ilişki bulunan farklı türden bilgiler *"yapılar (structures)"* içerisinde mantıksal bir bütün olarak ifade edilebilirler. Yapılar diziler gibi belleğe sürekli bir biçimde yerleşen nesnelerdir. Dizilerde olduğu gibi başlangıç adresleri geçirilerek fonksiyonlara kolaylıkla aktarılabilirler.

25.2 YAPILARIN BİLDİRİMİ

Yapıların bildirimi ile yapı değişkenlerinin tanımlanması iki ayrı işlem olarak yapılır. Programcı ancak yapı bildirimi ile yapıyı derleyiciye tanıttıktan sonra o yapı türünden bir değişken tanımlayabilir. Yapı bildirimlerinin genel biçimi aşağıdaki gibidir:

```
struct [yapı_ismi] {
    <tür> <yapı_elemanı>;
    <tür> <yapı_elemanı>;
    <tür> <yapı_elemanı>;
    ...
};
```

Yukarıdaki genel biçimde:

`struct`	Bildirim için gerekli bir anahtar sözcüktür.
`yapı_ismi`	Yapıyı anlatan isimlendirme kurallarına uygun herhangi bir isim olabilir.
`yapı_elemanı`	Yapıyı oluşturan değişken isimleridir; isimlendirme kurallarına uygun herhangi bir isim olabilir.

Bildirimin küme parantezlerinden sonra noktalı virgül ile sonlandırıldığına dikkat ediniz.

Örneğin, düzlemde bir nokta `x` ve `y` bileşenlerinden oluştuğuna göre, bunu ayrı ayrı elemanlar yerine bir yapı biçiminde de bildirebiliriz:

```
struct NOKTA {
    int x;
    int y;
};
...
```

Benzer biçimde, tarih bilgileri de herbiri `int` türünden üç ayrı değişken yerine, yapı kullanılarak mantıksal bir bütünlük içinde ifade edilebilir:

```
struct DATE {
    int day;   /* gün */
    int month; /* ay */
    int year;  /* yıl */
};
```

Yukarıdaki bildirimde `DATE` yapının ismi; `day`, `month` ve `year` ise `int` türünden yapı elemanlarıdır.

Yapı bildirimleriyle derleyici yalnızca yapılar hakkında bilgi edinir; bellekte onlar için herhangi bir yer ayırmaz. Yapı bildirimlerini bir çeşit şablon tanımlaması gibi düşünebilirsiniz. Tıpkı fonksiyon prototiplerinde olduğu gibi yapı bildirimleri de yalnızca derleyiciyi bilgilendirmek amacıyla kullanılmaktadır.

25.3 YAPI DEĞİŞKENLERİNİN TANIMLANMASI

Bellekte yer ayırma işlemi yapı değişkenlerinin tanımlanmasıyla olur. Yapı değişkenlerinin tanımlanması aşağıdaki biçimde yapılır:

Genel biçim:
struct <yapı_ismi> <yapı_değişkeninin_ismi>;

Yapı değişkenlerinin yapı bildiriminden sonra tanımlanması zorunludur. Örneğin:

```
struct NOKTA {
    int x;
    int y;
};
...
struct NOKTA a;
```

benzer biçimde **DATE** yapısı türünden bir **d** değişkeni:

```
struct DATE {
    int day;
    int month;
    int year;
};
...
struct DATE d;
```

Derleme işleminin yukarıdan aşağıya doğru bir yönü olduğunu anımsayınız. Derleyici yapı değişkenine ilişkin yapının biçimini bilmeden bellekte onun için yer ayırabilir mi?

Yapı değişkenlerinin türünü yapı isimleriyle birlikte ifade etmelisiniz. Örneğin

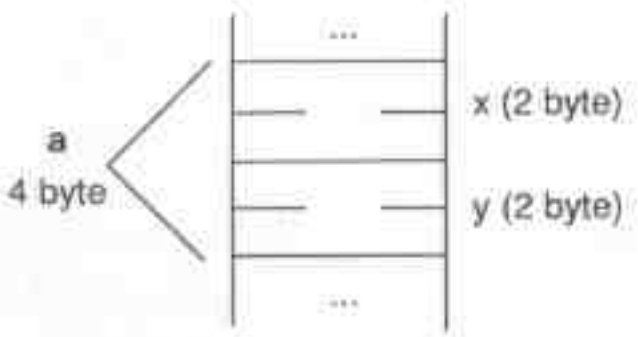
```
struct NOKTA a;
```

ile tanımlanmış olan a değişkenini, *"a, NOKTA türünden bir yapı değişkenidir"* ya da *"a, struct NOKTA türündendir"* biçiminde okuyabilirsiniz.

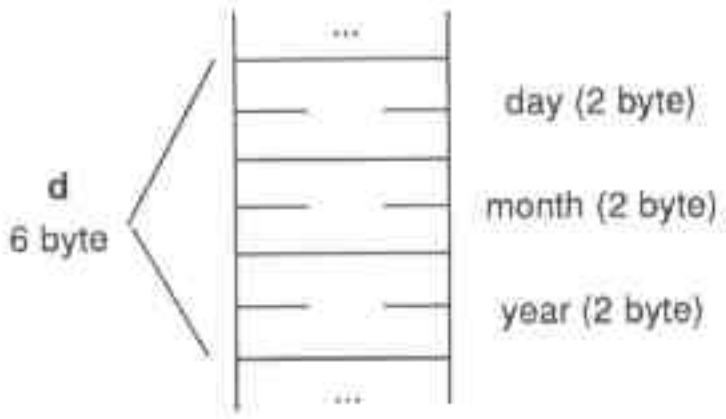
Bir yapı değişkeninin tanımlandığını gören derleyici bellekte ilgili yapının tüm elemanlarını içine alacak uzunlukta sürekli yer tahsis eder. Örneğin:

```
struct NOKTA a;
```

tanımlaması ile derleyici bellekte **a** yapı değişkeni için 16 bit sistemlerde 4 byte 32 bit sistemlerde 8 byte yer tahsis edecektir. Çünkü, yapının iki tane **int** türünden elemanı vardır.



Bu durumda **sizeof(a)** ya da **sizeof(struct NOKTA)** ifadeleri de 16 bit sistemlerde 4 değerini üretecektir. Aynı biçimde **DATE** türünden bir yapı değişkeni için derleyici 16 bit sistemlerde bellekte 6 byte sürekli yer ayırır.



`sizeof(d)` ya da `sizeof(struct DATE)` ifadelerinin 6 değerini üreteceğine dikkat ediniz.

25.3.1 Yapı Bildirimleri ile Tanımlama İşleminin Birlikte Yapılması

Yapı bildirimiyle değişken tanımlama işlemleri birlikte de yapılabilir. Bunun için bildirim işleminden sonra değişken listesi yazılır.

```
struct <yapı_ismi> {
    ...
} [değişken_listesi];
```

Bu biçimde bildirilen değişkenler, yapının bildirildiği yere bağlı olarak yerel ya da global olabilirler. Örneğin:

```
struct DATE {
    int day, month, year;
} x, y, z;
```

Bu durumda derleyici hem yapı bildirimini hem de bu yapı türünden değişken tanımlamasını geçerli sayar. `DATE` isimli yapının bildirimi global bir biçimde yapılmışsa değişkenler global, yerel bir biçimde yapılmışsa değişkenler de yerel faaliyet alanlarına sahip olurlar.

25.4 YAPI ELEMANLARINA ERİŞME VE NOKTA (.) OPERATÖRÜ

Bir yapı değişkeni gerçekte kendi içerisinde yapı elemanlarını barındıran bileşik bir değişkendir. Yapı elemanlarına nokta operatörü ile erişilir.

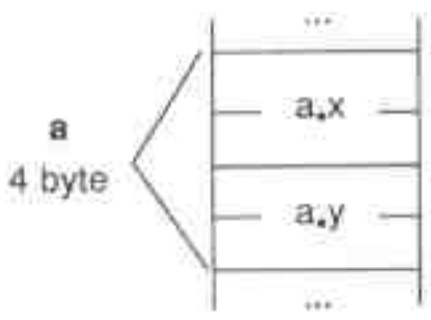
25.4.1 Nokta Operatörü

Nokta operatörü iki operandlı araek bir operatördür. Sol tarafındaki operand bir yapı değişkeni, sağ tarafındaki operand ise ilgili yapının bir elemanı olmak zorundadır.

Örneğin:

```
struct NOKTA a;
...
```

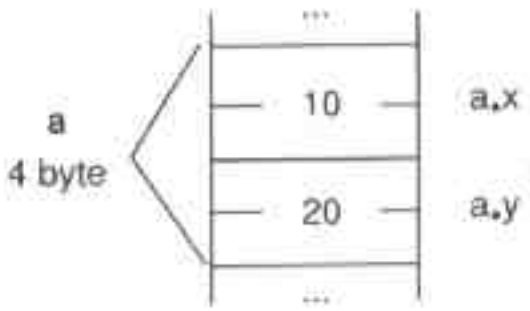
a nesnesinin kendisi `struct NOKTA` türünden `a.x` ve `a.y` nesneleri ise `int` türündendir.



a.x ve a.y birer nesne olduğuna göre sol taraf değeri olarak kullanılabilir. Örneğin:

```
a.x = 10;
a.y = 20;
```

ile yapının a ve b elemanlarına değer atayabiliriz.



Nokta operatörü C'nin en öncelikli operatör grubu içerisinde bulunur. Öncelik tablosunun tepe kısmını -şu ana kadar gördüğümüz operatörleri dikkate alarak- tekrar veriyoruz.

```
() [] .                          Soldan sağa
! ++ — * & sizeof  (tür) -       Sağdan sola
...                ...
```

25.5 YAPI ELEMANLARININ BELLEKTEKİ YERLEŞİMİ

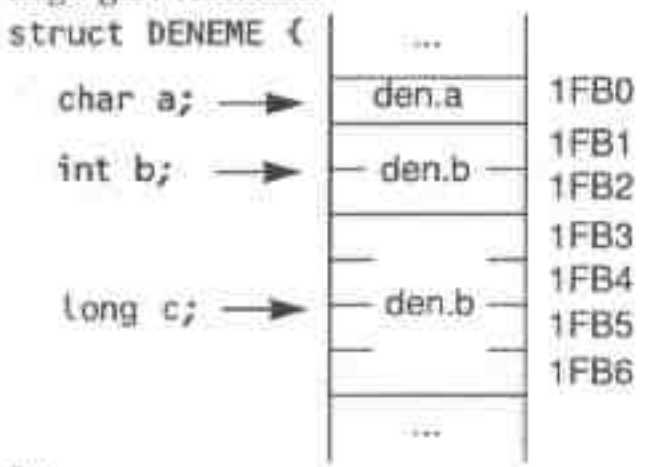
Bir yapı değişkeninin elemanları tıpkı dizilerde olduğu gibi belleğe sürekli bir biçimde yerleşir. Yapı elemanlarının yerleşimi oldukça basit bir kural ile belirlenmiştir:

Yapı bildiriminde ilk belirtilen eleman belleğin düşük anlamlı adresinde bulunacak biçimde yerleştirilir.

Örneğin:

```
struct DENEME {
    char a;
    int b;
    long c;
};
...
struct DENEME den;
```

bildirimi ile **den** isimli yapı değişkeni belleğin **1FB0** adresinden başlayarak yerleştirilmiş olsun. Bu durumda elemanların yerleşim biçimi aşağıdaki şekilde görüldüğü gibi olacaktır.



Söz konusu nesnelerin türleri de şöyledir:

Nesne	Tür	Uzunluk
den	struct DENEME	7
den.a	char	1
den.b	int	2
den.c	long	4

25.6 YAPI ELEMANLARI OLARAK DİZİLER VE GÖSTERİCİLER

Diziler ve göstericiler de yapı elemanı olabilirler. Örneğin:

```
struct INSAN {
    char adi_soyadi[30];
    char dogum_yeri[30];
    int no;
    int yas;
};
...
struct INSAN x;
```

bildirimi ile derleyici `x` için bellekte 64 byte yer ayırır.

Peki, `x.adi_soyadi` ya da `x.dogum_yeri` nesne midir? Yanıt hayır. Bu iki ifade de yapı içerisindeki dizilerin bellekteki başlangıç adreslerini gösteren dizi isimleridir. Ya aşağıdaki ifade ne anlama geliyor dersiniz?

```
x.adi_soyadi[n]
```

Bu ifadede `.` ve `[n]` olmak üzere iki operatör vardır. Bu iki operatörün de soldan sağa eşit öncelikli olduğunu anımsayınız. Bu durumda `x` yapısına ait `adi_soyadi` dizisinin `n.` elemanı anlatılmak istenmiştir. Yani bu ifade karakter türünden bir nesneyi göstermektedir. Benzer biçimde göstericiler de yapıların elemanları olabilirler. Aşağıdaki yapıyı inceleyiniz:

```
struct SAMPLE {
     int a;
     long b;
     char *c;
};
...
struct SAMPLE smp;
```

`smp.c` bir göstericidir. Ancak ilkdeğer verilmediğine göre güvenli bir bölgeyi göstermez. Diğer değişkenlerde olduğu gibi, ilkdeğer verilmemiş yerel yapı değişkenlerinin içerisinde de rastgele değerler bulunur. `smp.c` göstericisi güvenli bir ilk değer verildikten sonra kullanılmalıdır. Bunu `malloc` fonksiyonu ile yapabiliriz:

```
smp.c = (char *) malloc(100);
if (!smp.c) {
     printf("Yetersiz bellek\n");
     exit(1);
}
```

25.7 YAPI BİLDİRİMLERİNİN YAPILIŞ YERLERİ

Bir yapı bildirimi global olarak ya da yerel olarak yapılabilir. Yapı bildirimi yerel olarak yapılmışsa tanımlama da yalnızca bildirimin bulunduğu blokta yapılabilir. Örneğin:

```
void main()
{
    struct DATE {                    ──► yapı yerel olarak bildirilmiş
       int day, month, year;         ──► Değişken tanımlaması yapılabilir
    };
    struct DATE d;
    ...
}
fonk()                               ──► Hata! yapı yerel olarak bildirildiği için burada değişken tanımlaması yapılamaz
{
    struct DATE x;
    ...
}
```

Burada `DATE` yapısının bildirimi yerel olarak `main` fonksiyonunun içerisinde yapıldığı için bu blok dışında bir yerde yapı değişkeni bildirilemez. Oysa global olarak bildirilen yapılara ilişkin değişkenler kaynak kodun herhangi bir yerinde tanımlanabilir. Örneğin:

```
struct DATE {                        ──► Yapı global olarak bildirilmiş
    int day, month, year;
};

void main()                          ──► Değişken bildirimi yapılabilir
{
    struct DATE d;
    ...
}                                    ──► DATE yapısı global olarak bildirildiği için değişken bildirimi yapılabilir

fonk()
{
    struct DATE x;
    ...
}
```

Yukarıdaki `DATE` yapısı global olarak bildirildiği için, `fonk` isimli fonksiyonun içerisinde de değişken tanımlanabilir. Uygulamada ise yapı bildirimleri ya kaynak kodun tepesinde ya da herhangi bir başlık dosyasının içerisinde yapılır. Örneğin standart *C* fonksiyonlarının kullandığı yapıların bildirimi standart başlık dosyalarının içerisine konulmuştur.

25.8 YAPI DEĞİŞKENLERİNE İLKDEĞER VERİLMESİ

Yapı değişkenlerine dizilerde olduğu gibi küme parantezleri içerisinde ilkdeğer verilir. Örneğin:

```
struct SAMPLE {
    char *p;
    int n;
    float f;
};
...
struct SAMPLE x = {"örnek", 100, 5.78};
```

Küme parantezinden sonra ifadenin noktalı virgülle kapatıldığına dikkat ediniz.

Diğer bir örnek:

```
struct DATE {
    int day, month, year;
};
...
struct DATE date = {1, 4, 1995};
```

Bu durumda derleyici ilkdeğerleri yapı elemanlarına sırasıyla yerleştirir. Yani yukarıdaki örnekte yapı elemanlarının alacağı değerler şöyledir:

```
date.day    ► 1
date.month  ► 4
date.year   ► 1995
```

Eğer yapı içerisinde bir dizi tanımlanmışsa, dizinin istenildiği kadar elemanına küme parantezleri arasında ilk değer verebilir. Örneğin:

```
struct NO {
    int x[5];
    char *p;
};
...
struct NO n = {{10, 20, 30}, "Stokta var"};
```

burada n isimli yapı değişkeni içerisindeki x dizisinin yalnızca 3 elemanına ilk değer verilmiştir. Küme parantezleri kullanılmazsa dizininin tüm elemanlarına ilk değer vermek gerekir. Örneğin aşağıdaki ifade hatalıdır:

```
struct NO n = {10, 20, 30, "Stokta var"};
```

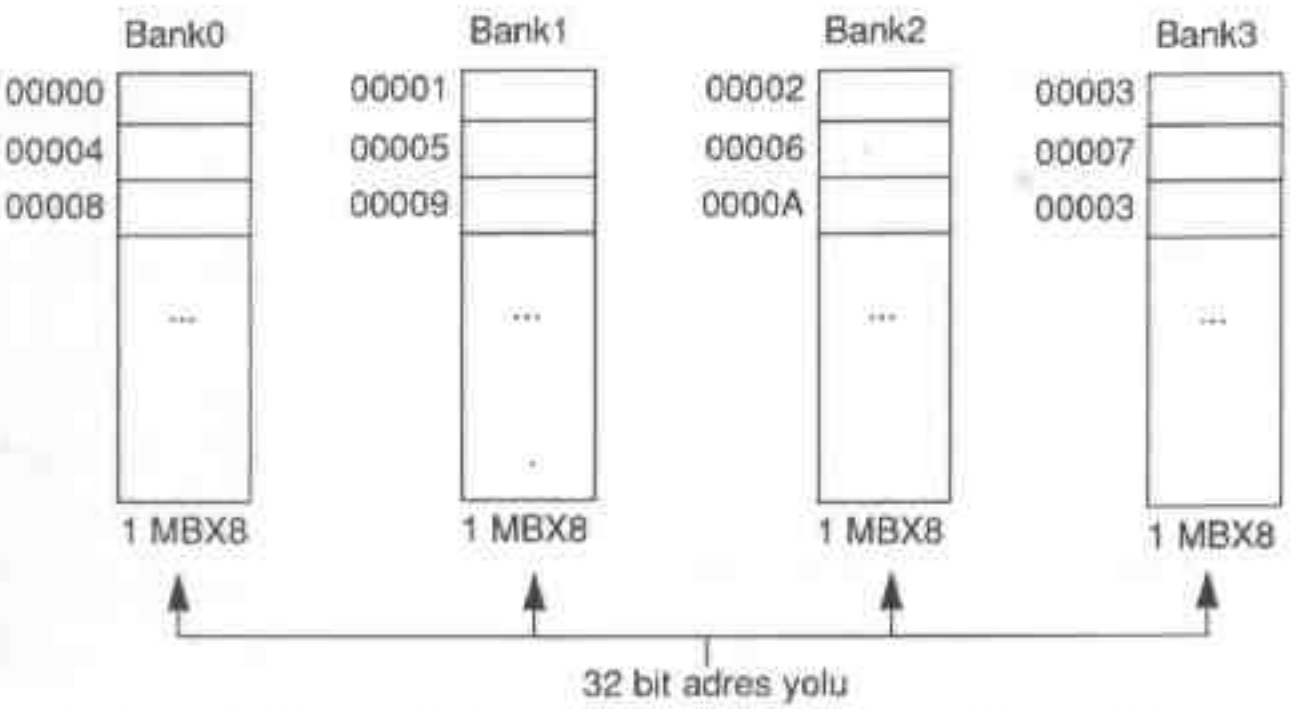
Derleyici "Stokta var" stringiyle belirtilen adresi x dizi elemanının 3. indisli elemanına atayacaktır.

```
n.x[0] = 10
n.x[1] = 20
n.x[2] = 30
n.x[3] = "Stokta var"  => Hata!..
...
```

25.9 HİZALAMA (Alignment) KAVRAMI VE YAPILAR

Bir yapı değişkeni için derleyicilerin bellekte tahsis edeceği alan 1 byte , 2 byte (word) ya da 4 byte'ın (double word) katları biçiminde olabilir. Mikroişlemcilerin çoğu bellekte kendi yazmaç uzunluklarının katlarından başlayan bilgilere daha hızlı erişirler. Örneğin 8086 mikroişlemcisi 16 bit olduğu için 2'nin katlarındaki bilgilere (yani çift adreslerde bulunan bilgilere) daha kolay erişir. Bu yüzden 16 bit sistemlerde çalışan derleyiciler seçimli olarak 2 ya da daha uzun byte'tan oluşan bilgileri çift numaralı adreslerde tutarlar. Örneğin 2 byte uzunluğundaki bir bilgi `3FC3` adresi yerine `3FC4` adresinde tutulsa mikroişlemci bu bilgiye daha kolay erişecektir. Benzer biçimde 32 bit sistemlerde çalışan derleyiciler de (Intel 80X86 32 bit korumalı modda çalışan UNIX, Windows 95 gibi) 4 byte'ın katlarında bulunan bilgilere daha hızlı erişirler (4 byte'ın katları HEX sistemde sonu `0, 4, 8, C` ile sayılardır).

80X86 Sembolik Makina Dili Programcısına Not: Mikroişlemcinin 2 byte ya da 4 byte sınırlarından başlayan bilgilere daha hızlı erişmesi tasarımsal bir özelliktir. Örneğin 386, 486 işlemcilerinin bellek organizasyonu aşağıdaki gibi tasarlanmıştır:



Şekilden de anlaşıldığı gibi 8 bitlik ayrı ayrı RAM bloklarından oluşan 32 bit paralel olarak mikroişlemciye bağlanmıştır. 386, 486 mikroişlemcileri bellekte bir bölgeye erişirlerken erişecekleri uzunluğu da BE_0, BE_1, BE_2, BE_3 uçlarıyla belirtirler. Örneğin mikroişlemci `00005` adresinden başlayan 2 byte değerine

```
MOV AX, [00004]
```

gibi bir komutla erişecek olsa adres yoluna 00004 bilgisini koyarak BE_1 ve BE_2 çıkışlarını 0 seviyesine getirir. Örneğin 00007 adresinden başlayan iki byte'lık bilgi istense, mikroişlemci önce adres yoluna 00004 bilgisini koyarak BE_3 = 0 ile adresin ilk byte değerine sonra da adres yoluna 00008 koyarak BE_0 = 0 ile adresin ikinci byte değerine erişecektir. Hizalamanın hız üzerindeki etkisi böylelikle anlaşılabilir.

Derleyiciler hizalama özelliği ile yapı değişkenlerini duruma göre 2 byte'ın (word alignment) ya da 4 byte'ın (double word alignment) katlarında tutabilirler. Örneğin 16 bit sistemlerde derleyicinin hizalama seçeneği **word alignment** biçimindeyse yapıdaki yerleşim şöyle olur:

- Yapı değişkenleri çift adreslerden başlayacak biçimde yerleştirilir.
- `char` türü dışındaki yapı elemanları çift adreslerden başlayacak biçimde yerleştirilir.
- Eğer toplam yapının uzunluğu tek bir değer ise sonuna bir byte boşluk eklenerek çift olması sağlanır.

Benzer biçimde 32 bitlik sistemlerde hizalama seçeneği **double word alignment** biçimindeyse yapıdaki yerleşim de şöyle olacaktır:

- Yapı değişkenleri 4 byte'ın katları olan (double word) adreslerden başlayacak biçimde yerleştirilir.
- `char` türü dışındaki yapı elemanları 4 byte'ın katları olan adreslerden başlayacak biçimde yerleştirilir.
- Eğer toplam yapının uzunluğu 4 byte'ın katları kadar değilse yeteri kadar boşluk bırakılarak 4 byte'ın katları kadar olması sağlanır.

Aşağıdaki yapı bildirimini inceleyiniz:

```
struct Deneme {
        int a;
        char str[11];
        double b;
};
```

16 bit sistemlerde çalışıyorsanız ve derleyicinizin hizalama seçeneği **word alignment** biçimindeyse

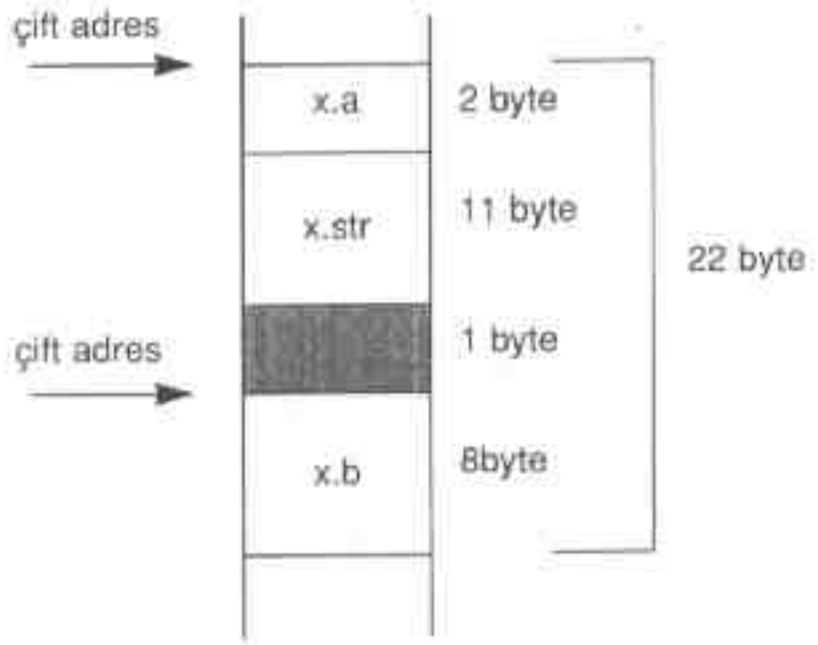
```
Deneme x;
```

gibi bir bildirim ile:

1) `x` yapı değişkeni çift bir adresten başlayarak yerleştirilir.

2) `x` yapı değişkeninin `b` elemanı da çift bir adresten başlar. Bu yüzden `str` dizisiyle `b` arasında 1 byte boş bırakılacaktır.

3) `x` yapı değişkeninin uzunluğu `sizeof(struct Deneme)` `b` elemanı için verilen boşluktan dolayı 22'dir.



25.10 YAPI GÖSTERİCİLERİ

Yapı değişkenleri de birer nesne olduğuna göre onların da adresleri söz konusu olabilir. Örneğin:

```
struct DENEME {
        char ch;
        int n;
        long l;
};
...
struct DENEME x;
```

bildirimleri ile:

`x.ch` nesnesi `char`, `x.n` nesnesi `int` ve `x.l` nesnesi `long` türündendir. Bu durumda:

`&x.ch` ile elde edilen adresin türü `char`,

`&x.n` ile elde edilen adresin türü `int`,

`&x.l` ile elde edilen adresin türü ise `long`'tur.

`x`'in kendisi de nesne olduğundan, onun da adresi alınabilir. `&x` ile elde edilen adres `struct DENEME` türündendir.

Her türden adresi içine alabilecek bir gösterici tanımlanabildiğine göre yapıları gösteren adresleri de içine alabilecek göstericiler tanımlanabilmelidir. Yapı göstericilerinin genel biçimleri aşağıdaki gibidir:

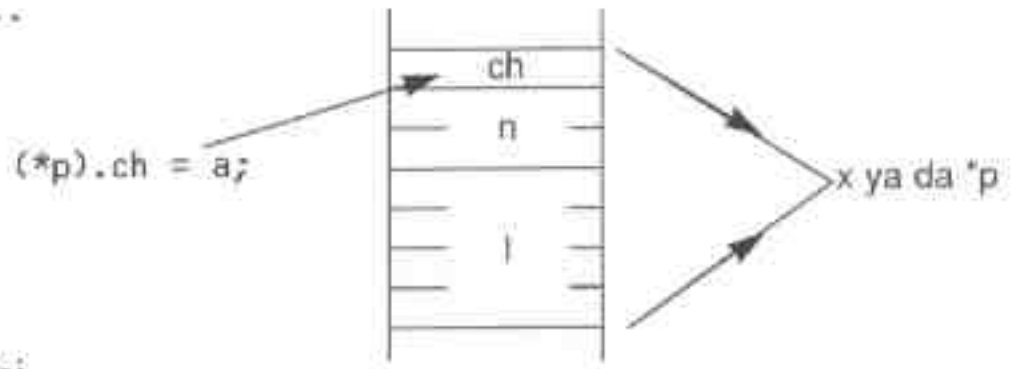
```
struct <yapı_ismi> *gösterici_ismi;
```

Örneğin:

```
struct DATE *d;
struct DENEME *p;
```

Aşağıdaki örneği inceleyiniz. Burada önce yapı değişkeninin adresi aynı türden yapı göstericisine atanmış daha sonra da bu gösterici sayesinde yapı elemanlarına erişilmiştir.

```
struct DENEME x;
struct DENEME *p;

p = &x;
(*p).ch = 'a';
...
```



gibi...

Yapı göstericisi kullanarak yapı elemanlarına erişilmesi işleminde bir noktayı vurgulamak istiyoruz:

```
(*p) . ch = "a";
```

*p ifadesinin parantez operatörüyle önceliklendirilmesi zorunludur. Bu yapılmazsa . operatörünün * operatörüne göre önceliğinden doğan bir derleme hatasıyla karşılaşılır.

```
*p . ch = 'a';          /* Hata!.. */
```

Çünkü p.ch ifadesini öncelikli ele alan derleyici . operatörünün sol tarafındaki operandının bir yapı değişkeni olmadığını görür. Aşağıdaki örneği inceleyiniz:

```
#include <stdio.h>

struct DENEME {
    char ch;
    int n;
    long l;
};

void main()
{
    struct DENEME x;
    struct DENEME *p;

    p = &x;
    (*p).ch = 'a';
    (*p).n = 100;
    (*p).l = 200L;

    printf("x.ch = %c\n", x.ch);
```

```
    printf("x.n = %d\n", x.n);
    printf("x.l = %ld\n", x.l);
}
```

Burada önce x yapı değişkeninin adresi p göstericisine geçirilmiş daha sonra p kullanılarak x içerisine değerler yazılmıştır.

25.11 İÇ İÇE YAPILAR

Bir yapının içinde başka bir yapı nesnesi tanımlanabilir. İç içe yapıların tanımlanması C'de iki biçimde yapılmaktadır:

1) İçerideki yapının bildirimini daha yukarıda yaparak. Bu durumda derleyici doğal akış yönünde ilerlerken içeride bildirilen yapıyı tanıyabilir.

```
struct DATE {
    int day, month, year;
};
...
struct PERSON {
    char name[30];
    struct DATE bday;
};
```

DerlemeYönü

Yukarıda Bildirilmiş

Yukarıdaki örnekte DATE isimli yapı PERSON isimli yapının daha yukarısında bildirilmiştir. Böylece derleyici PERSON yapısı içerisindeki DATE yapısını derleme işleminin doğal akışı içerisinde tanıyabilir.

2) İçerideki yapının dışarıdaki yapının içerisinde bildirilmesiyle. Bu durumda değişken tanımlaması da yapılmalıdır:

```
struct PERSON
    char name[30];
    struct DATE {
        int day, month, year;
    } bday;
};
```

yapı bildirimi ile tanımlama işlemi aynı anda yapılmış

Bu biçimde yapılan bildirimlerde içerideki yapılar da tek başlarına tanımlama amacıyla kullanılabilir. Örneğin:

```
struct PERSON per;
```

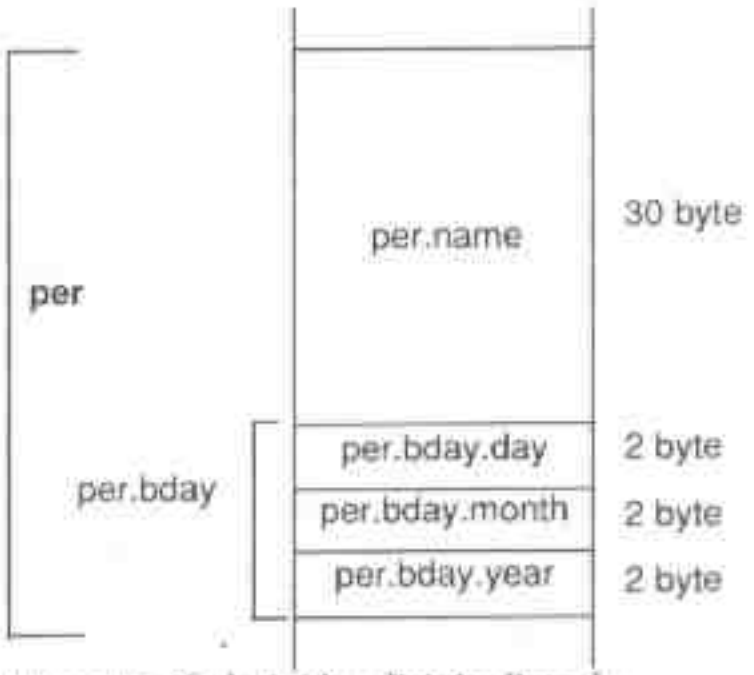
tanımlamasının yanı sıra,

```
struct DATE d;
```

gibi bir tanımlama da geçerlidir. Yukarıdaki PERSON isimli yapı türünden bir değişken tanımlayalım:

```
struct PERSON per;
...
```

derleyici **per** için DOS ve 16 bit Windows sistemlerinde 36 byte, UNIX ve 32 bit Windows sistemlerinde ise 42 byte yer ayırır. İzleyiniz:



`per.bday.day` ifadesini inceliyiniz. Burada:

`per` ► struct PERSON türünden
`per.bday` ► struct DATE türünden
`per.bday.day` ► int türündendir.

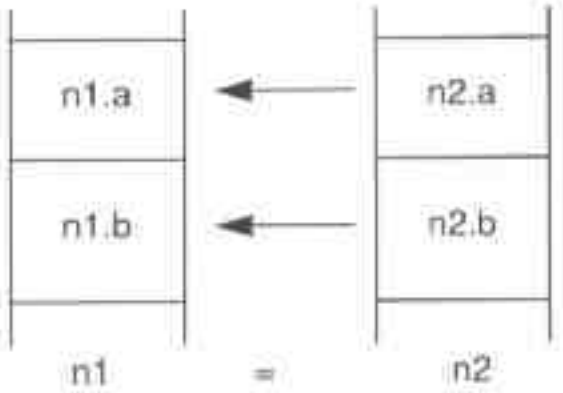
- operatörünün soldan sağa öncelikli olduğunu anımsayınız.

25.12 YAPI DEĞİŞKENLERİ ARASINDAKİ İŞLEMLER

C'de yalnızca aynı türden iki yapı değişkeni birbirlerine atanabilir. Bunun dışında yapı değişkenleri toplama, çarpma gibi aritmetik ya da mantıksal işlemlere sokulamazlar. Örneğin:

```
struct X {
    int a;
    long b;
};

struct X n1, n2;
...

n1 = n2;
```

işlemi geçerlidir. Bu durumda **n2** yapı değişkeninin elemanları karşılıklı olarak **n1** yapı değişkeninin elemanlarına atanır.



İsimleri farklı olan iki yapının elemanları bire bir aynı olsa bile atama işlemi geçersizdir. Örneğin:

```
struct X1 {
    int a;
    long b;
};

struct X2 {
    int a;
    long b;
};

struct X1 n1;
struct X2 n2;
...
n1 = n2;
```

işlemi geçersizdir.

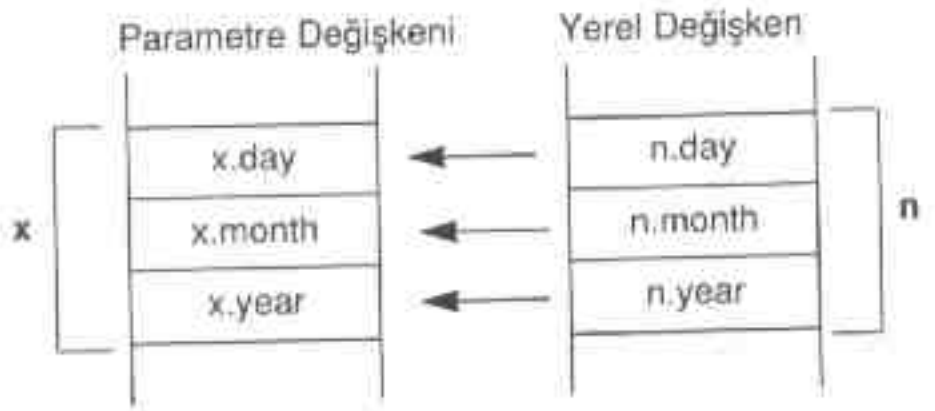
25.13 YAPILARIN FONKSİYONLARA PARAMETRE OLARAK GEÇİRİLMESİ

C'de parametrelerin kopyalanarak fonksiyonlara geçirildiğini anımsayınız. Yapı değişkenleri de fonksiyonlara kopyalanarak parametre olarak geçirilebilirler. Aşağıdaki örneği inceleyiniz:

```
struct DATE {
    int day, month, year;
};

void dispDATE(struct DATE x)
{
    printf("Yıl:%d\n", x.year);
    printf("Ay:%d\n", x.month);
    printf("Gün:%d\n", x.day);
}

void main(void)
{
```

```
    struct DATE n;

    n.day = 4;
    n.month = 5;
    n.year = 1995;
    dispDATE(n);
}
```

Fonksiyon parametrelerinin kopyalanmasının da bir çeşit atama işlemi olduğunu anımsayınız. Bu durumda parametre olarak geçirilen yapı değişkeni ile kopyalamanın yapılacağı parametre değişkeninin aynı türden yapılara ait olması gerekmez mi? Bu biçimde yapının parametre olarak fonksiyonlara geçirilmesi durumunda bütün yapı elemanları karşılıklı olarak parametre elemanlarına kopyalanırlar. Yani yukarıdaki örnekte `n.day`, `n.month` ve `n.year` yapı elemanları sırasıyla `x.day`, `x.month` ve `x.year` elemanlarına kopyalanacaktır.

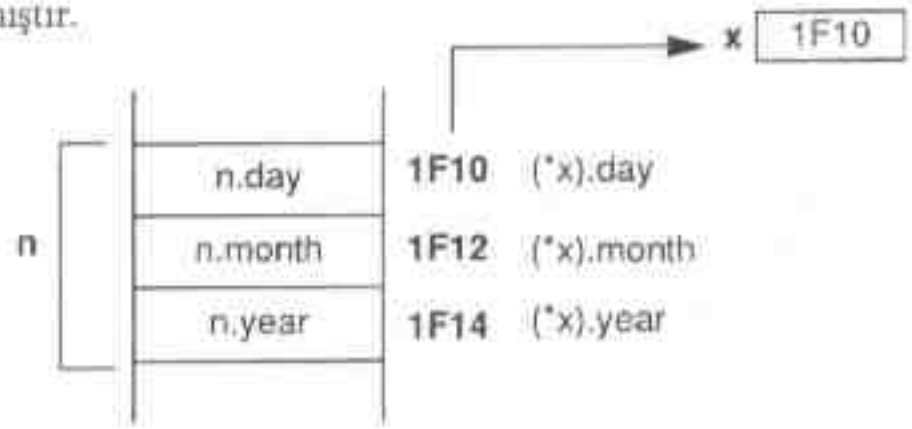


Bu aktarım biçimi büyük yapılar için oldukça verimsiz sayılabilir. Çünkü, örneğin bir döngü içerisinde büyük bir yapının bu biçimde aktarılması önemli bir zaman kaybı oluşturabilir. İşte madem ki yapılar bellekte sürekli bir biçimde bulunmaktadır; o halde onlar da tıpkı dizilerde olduğu gibi adresleri geçirilerek fonksiyonlara aktarılabilirler. Yukarıdaki örneği aşağıdaki biçimde yeniden düzenleyelim:

```
struct DATE {
    int day, month, year;
};
void dispDATE(struct DATE *x)
{
    printf("Yıl:%d\n", (*x).year);
    printf("Ay:%d\n", (*x).month);
    printf("Gün:%d\n", (*x).day);
}
void main(void)
{
    struct DATE n;

    n.day = 4;
```

```
    n.month = 5;
    n.year = 1995;
    dispDATE(&n);
}
```

dispDATE fonksiyonuna n yapı değişkeninin yalnızca adresi geçirilmiştir. **Intel** işlemcilerinde ve *DOS* altında bu adres bellek modeline göre 2 byte ya da 4 byte olabilir. Örneğimizdeki n isimli yapı değişkeninin 1F10 adresine sahip olduğu varsayılmıştır.



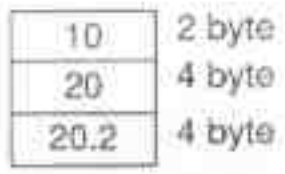
Bu durumda yalnızca n yapı değişkeninin adresi olan 1F10, dispDATE fonksiyonunun x parametresine kopyalanır. Bu adresi alan dispDATE fonksiyonu * ve . operatörü ile yapının herhangi bir elemanına erişebilir.

25.13.1 Yapılar ve Diziler

Yapılar ve diziler belleğe sürekli yerleşmeleri nedeniyle birbirlerine benzerler. Bir dizi tüm elemanları aynı türden olan bir yapı biçiminde düşünülebilir. Ya da yapılar elemanları farklı türlerden olabilen diziler biçiminde tanımlanabilirler. Bir dizi isminin aslında o dizinin bellekteki başlangıç adresi olduğunu anımsayınız. Bir yapı nesnesinin adresi de o yapının bellekteki başlangıç adresidir.

Bir dizi fonksiyona yalnızca başlangıç adresi ve uzunluğu geçirilerek aktarılabilirdi. Karakter dizileri NULL karakter ile sonlandığına göre, karakter dizileri için dizinin uzunluğunu geçirmeye gerek olmadığını anımsayınız. Benzer biçimde bir yapının fonksiyona parametre olarak geçirilmesinde de yapının uzunluğunu geçirmeye gerek yoktur. Çünkü yapı bildirimini gören derleyici yapının uzunluğunu ve yapı elemanlarının yapının başlangıç adreslerinden ne kadar uzakta olduğunu anlayabilir ve onların yerlerini bulabilir. Örneğin:

```
int fonk(struct DENEME *den)
{
    ...
    (*den).a = 10
    (*den).b = 20;
    (*den).c = 20.2;
    ...
}
```



25.14 OK (->) OPERATÖRÜ

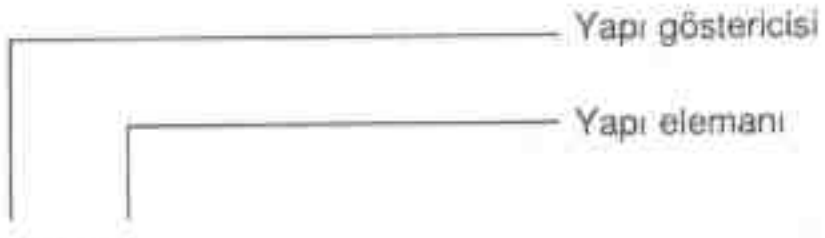
. operatörünün * operatörüne göre önceliği okunabilirlik konusunda problemlere yol açmaktadır. Yapı göstericileri yoluyla yapı nesnelerine erişmek için ismine ok operatörü (arrow operator) denilen bir operatör kullanılır.

```
(*p) . a
```

ile

```
p -> a
```

aynı anlama gelir. -> iki operandlı araek bir operatördür. Bu operatörün - ve > karakterlerinin yan yana getirilerek yapıldığına dikkat ediniz. Eğer araya bir boşluk karakteri koyarsanız derleyici bu karakterleri iki ayrı operatör gibi değerlendireceğinden hata oluşacaktır. -> operatörünün sol tarafındaki operand bir yapı göstericisi sağ tarafındaki operand ise bir yapı elemanı olmak zorundadır.



Yukarıda örnek olarak verdiğimiz **dispDATE** fonksiyonunu -> operatörünü kullanarak aşağıdaki gibi de yazabiliriz:

```
void dispDATE(struct DATE *x)
{
    printf("Yıl:%d\n", x -> year);
    printf("Ay:%d\n", x -> month);
    printf("Gün:%d\n", x -> day);
}
```

25.14.1 -> Operatörünün Önceliği

-> operatörü C'nin en öncelikli operatör grubu içerisinde bulunmaktadır. Önceliği () . ve [n] ile aynıdır.

```
() . [n] ->                    Soldan Sağa
! ++ -- * & sizeof (tür) ~     Sağdan sola
...
```

Şimdi yapılar konusunun uygulaması olarak C'nin standart tarih ve zaman fonksiyonlarını ele alacağız.

25.15 TARİH VE ZAMAN FONKSİYONLARI

Geçerli sistem zamanının bulunması ve bunlar üzerinde işlemlerin yapılmasına ait uygulamalara çok sık rastlanır. Tarih ve zaman işlemleri için `TIME.H` başlık dosyası kullanılır. Bütün tarih ve zaman fonksiyonlarının prototipleri, sembolik sabitlerin ve standart yapıların tanımlamaları bu dosya içerisinde bulunmaktadır.

time Fonksiyonu

01/01/1970 tarihinden itibaren geçen saniye sayısını bulur. Bu değer bazı tarih ve zaman fonksiyonlarında girdi olarak kullanılmaktadır. `time` fonksiyonunun prototipi aşağıdaki gibidir:

```
long time (long *t);
```

01/01/1970'den geçen saniye sayısı geri dönüş değeri ya da parametre yoluyla elde edilebilir. Eğer geri dönüş değeri yoluyla elde edilecekse parametre olarak `NULL` gösterici girilmelidir. Örneğin:

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

...
long t;
t = time(NULL);
```

bunun yerine parametre yoluyla elde edilmek istenirse geri dönüş değeri kullanılmayabilir:

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

...
long t;
time(&t);
```

gibi...

localtime Fonksiyonu

`localtime` çağrıldığı andaki sistem tarihinin ve zamanının elde edilmesi amacıyla kullanılır. **01/01/1970**'ten geçen saniye sayısını (yani `time` fonksiyonunun çıktısını) parametre olarak alır, bunu o andaki tarih ve zaman değerlerine dönüştürerek statik olarak tahsis edilmiş bir yapı içerisinde saklar; geri dönüş değeri bu yapının başlangıç adresidir. Prototipi aşağıdaki gibidir:

```
struct tm *localtime(long *timer);
```

`time` fonksiyonundan elde edilen değer `localtime` fonksiyonunda parametre olarak kullanılmaktadır. Geri dönüş değeri ise `TIME.H` dosyasında tanımlanan

`struct tm` isimli yapı göstericisidir.

`struct tm` yapısını inceleyiniz:

```
struct tm
{
    int tm_sec;     /* saniye */
    int tm_min;     /* dakika */
    int tm_hour;    /* saat */
    int tm_mday;    /* gün */
    int tm_mon;     /* ay (0-11 arası, ocak = 0) */
    int tm_year;    /* yıl */
    int tm_wday;    /* haftanın günü (0-6 arası, Pazar = 0) */
    int tm_yday;    /* Yılın günü (0-364 arası, 1 Ocak = 0) */
    int tm_isdst;   /* İleri saat uygulamasının yapılıp yapılmadığı */
};
```

Ekrana saati ve tarihi yazdıran yalın bir örnek verelim isterseniz.

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

void main(void)
{
    long timer;
    struct tm *tinfo;

    timer = time(NULL);
    tinfo = localtime(&timer);
    printf("Tarih: %02d/%02d/%02d\n", tinfo->tm_mday,
    tinfo-> tm_mon+1,tinfo->tm_year);
    printf("Zaman: %02d:%02d:%02d\n", tinfo->tm_hour,
    tinfo->tm_min,tinfo->tm_sec);
}
```

`tinfo` değişkeninin `struct tm` türünden bir yapı göstericisi olduğuna dikkat ediniz. `localtime` fonksiyonunun verdiği adresin türü de aynıdır. `localtime` yapı alanını kendi içerisinde dinamik bellek fonksiyonlarıyla tahsis etmediğinden fonksiyon her çağrıldığında aynı alanın üzerine ve eskisini bozacak biçimde değer yazar. Şimdi de bir tuşa basana kadar saatin canlı olarak basıldığı bir örnek verelim.

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

void main(void)
{
    long timer;
    struct tm *tinfo;

    while (!kbhit()) {
```

```
        timer = time(NULL);
        tinfo = localtime(&timer);
        printf("%02d:%02d:%02d\r", tinfo->tm_hour,
        tinfo->tm_min,tinfo->tm_sec);
    }
}
```

Zamanın değişiminin algılanabilmesi için **time** fonksiyonunun da döngü içerisine konulduğuna dikkat ediniz. **localtime** fonksiyonu zamanı alma işlemini yapmaz; yalnızca **time** fonksiyonu tarafından alınmış olan zamanı **struct tm** biçimine dönüştürür. **printf** içerisinde kullandığımız '\r' (**carriage return**) karakteri, imleci bulunulan satırın başına geçirir. İmleç hareketlerinden dolayı ekranda oluşan perdelenmeyi ancak ekran belleğine doğrudan yazarak giderebilirsiniz.

ctime Fonksiyonu

Bu fonksiyon **time** fonksiyonunun çıktısını parametre olarak alır, 26 karakterlik **NULL** ile biten bir diziye yerleştirerek dizinin başlangıç adresiyle geri döner. Tarih ve zaman bilgisinin dizi içerisindeki durumu şöyledir:

```
GGG AAA gg SS:dd:ss YYYY
```

GGG	=	İngilizce günlerin ilk üç harfinden elde edilen kısaltma (Mon, Tue, Wed, ...)
AAA	=	İngilizce ayların ilk üç harfinden elde edilen kısaltma (Jan, Feb, Mar, ...)
gg	=	Sayısal olarak ayın hangi günü olduğu (1, 2, ... , 31)
SS	=	Saat (1, 2, ... , 24)
dd	=	Dakika (0, 1, ... , 59)
ss	=	Saniye (0, 1, ... , 59)
YYYY	=	Yıl

ctime ilgili diziyi kendi içerisinde statik olarak tahsis ettiği için hep aynı adresi geri vermektedir. Prototipini inceleyiniz:

```
char *ctime(const long *timer);
```

ctime ile tarihin ve zamanın ekrana yazılmasına bir örnek:

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

void main(void)
{
    long timer;

    timer = time(NULL);
```

```
    printf("%s\n", ctime(&timer));
}
```

asctime Fonksiyonu

`ctime` fonksiyonuna benzer bir işlevi yerine getirir. Parametre olarak `ctime` fonksiyonundan farklı olarak `struct tm` türünden bir gösterici alır. Tıpkı `ctime` fonksiyonunda olduğu gibi tarihi ve zamanı 36 karakterlik bir diziye yerleştirir. Prototipi aşağıdaki gibidir:

```
char *asctime(const struct tm *tblock);
```

Aşağıdaki örnekte `asctime` fonksiyonundan alınan bilgiler ekrana yazdırılmaktadır.

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

void main(void)
{
    long timer;
    struct tm *tinfo;

    timer = time(NULL);
    tinfo = localtime(&timer);
    printf("%s\n", asctime(tinfo));
}
```

SORAMADIKLARINIZ...

S1) Bölüm içerisinde verilen örneklerde yapı isimleri genellikle büyük harflerle yazıldı. Sembolik sabitlerde olduğu gibi yapı isimlerinde de büyük harfleri kullanmak gibi bir gelenek yerleşmiş midir?

C1) Programcıların bir bölümü (benim gibi) yapı isimlerini de büyük harflerle yazmak eğilimindedir. Ancak bu sembolik sabitlerde olduğu gibi yaygın bir eğilim değildir. Örneğin bazı programcılar yalnızca yapı isimlerinin ilk harflerini büyük yazarlar.

```
struct Deneme {
    ...
};
```

gibi.

S2) Yapıların uzunluğu için bir sınır var mıdır?

C2) Yapılar belleğe sürekli yerleşen nesnelerdir. Bu nedenle 80X86 16 bit işletim sistemlerinde yapılar da diziler gibi 64K değerini aşamazlar. Ancak korumalı modda çalışan 32 bit işletim sistemlerinde böyle bir sınır söz konusu değildir.

26 BİRLİKLER

Birlikler de yapılar gibi belleğe sürekli bir biçimde yerleşen nesnelerdir. Tanımlanmaları ve bildirimleri birbirinin aynısı olsa da içerdiği nesnelerin yerleşim biçimleri arasında fark vardır. Birlikler yapılara göre seyrek kullanılırlar.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Birliklerin bildirimi nasıl yapılmaktadır?
2) Birlik değişkenleri nasıl tanımlanır?
3) Sayıların bellekteki yerleşimleri Intel ve Motorola işlemcilerinde nasıldır?
4) Birlik elemanlarının bellekteki organizasyonu nasıldır?
5) Kesme nedir? 80X86 mikroişlemcilerinin gerçek modunda kesme çağırma işlemi nasıl yapılmaktadır?

26.1 BİRLİKLERİN BİLDİRİMİ

Birliklerin bildirimi de yapılarda olduğu gibidir. Yalnızca `struct` anahtar sözcüğü yerine `union` anahtar sözcüğü kullanılır.

```
union [birlik ismi] {
     tür <birlik elemanı> ;
     tür <birlik elemanı> ;
     tür <birlik elemanı> ;
...
};
```

Yukarıdaki genel biçimde:

`union`	Bildirim için gerekli bir anahtar sözcüktür.
`birlik_ismi`	Birliği anlatan isimlendirme kurallarına uygun herhangi bir isim olabilir.
`birlik_elemanı`	Birlik elemanıdır; isimlendirme kurallarına uygun herhangi bir isim olabilir.

Örneğin:

```
union DWORD {
     unsigned char byte;
```

```
    unsigned  int word;
    unsigned long dword;
};
```

Diğer bir örnek:

```
union  DBL_FORM {
    double n;
    unsigned char s[8];
};
```

26.2 BİRLİK DEĞİŞKENLERİNİN TANIMLANMASI

Tıpkı yapılarda olduğu gibi birliklerde de bellekte yer ayırma işlemi bildirim ile değil tanımlama işlemi ile yapılmaktadır. Birlik bildirimlerinin yapı bildirimlerinden tek farkı `struct` anahtar sözcüğü yerine `union` anahtar sözcüğü kullanılmasıdır:

Genel biçim:

```
union <birlik_ismi>  <birlik_değişkeninin_ismi>;
```

Örneğin:

```
union DWORD a, b;
```

ile `a` ve `b` `union DWORD` türünden iki değişken olarak tanımlanmışlardır. Benzer biçimde:

```
union DBL_FORM x, y;
```

`x` ve `y` `union DBL_FORM` türünden iki değişken biçimindedir. Yine tıpkı yapılarda olduğu gibi birliklerde de bildirim ile tanımlama işlemi birlikte yapılabilir. Örneğin:

```
union  DBL_FORM {
    double n;
    unsigned char s[8];
} x, y;
```

Bu durumda `x` ve `y` değişkenlerinin faaliyet alanları birliğin bildirildiği yere bağlı olarak global ya da yerel olabilir.

Birlik elemanlarına da • operatörüyle erişilir. Örneğin yukarıdaki tanımlama dikkate alınırsa:

```
union DWORD a;
```

`a. byte` birliğin `unsigned char` olan ilk elemanını belirtmektedir. Benzer biçimde birlik türünden de göstericiler tanımlanabilir. Ok operatörü yine yapılar-

da olduğu gibi gösterici yoluyla birlik elemanlarına erişmekte kullanılır. Örneğin:

```
union DWORD *p;
...
```

`p -> word` ifadesi ile `p` adresi ile belirtilen bölgedeki birliğin **word** elemanına erişilmektedir.

Genel olarak şunları söyleyebiliriz: **Yapılarla birliklerin bildirilmesi, tanımlanması ve yapı değişkenlerinin kullanılması tamamen aynı biçimde yapılmaktadır. İkisi arasındaki tek fark yapı ve birlik elemanlarının organizasyonunda ortaya çıkar.**

Birlik elemanlarının organizasyonu konusuna değinmeden önce bir byte'tan daha uzun bilgilerin bellekteki görünümleri üzerinde durmak istiyoruz.

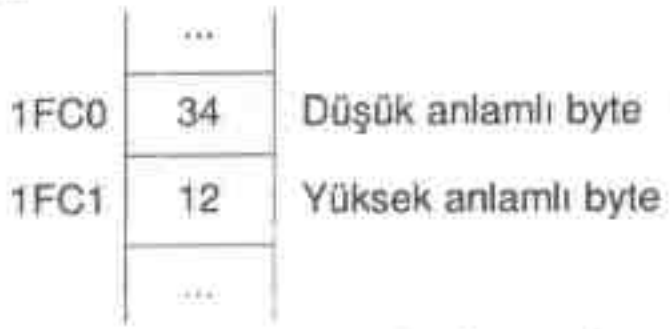
26.3 SAYILARIN BELLEKTEKİ YERLEŞİMLERİ

Bir byte'tan daha büyük olan sayıların belleğe yerleşim biçimi kullanılan mikroişlemciye göre değişebilir. Bu nedenle sayıların bellekteki görünümleri taşınabilir bir bilgi değildir. Mikroişlemciler iki tür yerleşim biçimi kullanabilirler.

1) Düşük anlamlı byte değerleri belleğin düşük anlamlı adresinde bulunacak biçimde. 80X86 ailesi Intel işlemcileri bu yerleşim biçimini kullanır. Bu işlemcilerin kullanıldığı sistemlerde örneğin:

```
int x = 0x1234;
```

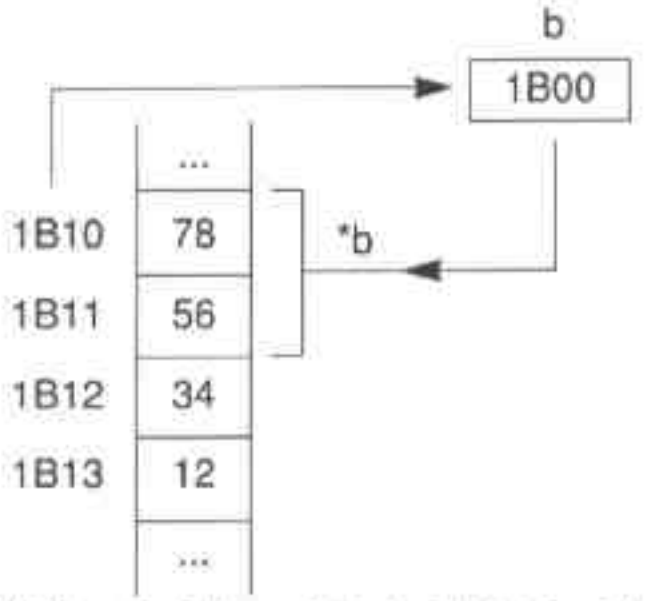
biçimindeki bir x değişkeninin bellekte `1FC0` adresinden başlayarak yerleştiğini varsayalım:



Şekilden de görüldüğü gibi x değişkenini oluşturan sayılar düşük anlamlı byte değeri (34H) düşük anlamlı bellek adresinde (1FC0H) olacak biçimde yerleştirilmiştir. Şimdi aşağıdaki kodu inceleyiniz.

```
...
unsigned long a = 0x12345678;
unsigned int *b;
b = (unsigned int *) &a;
printf("%x\n", *b);
```

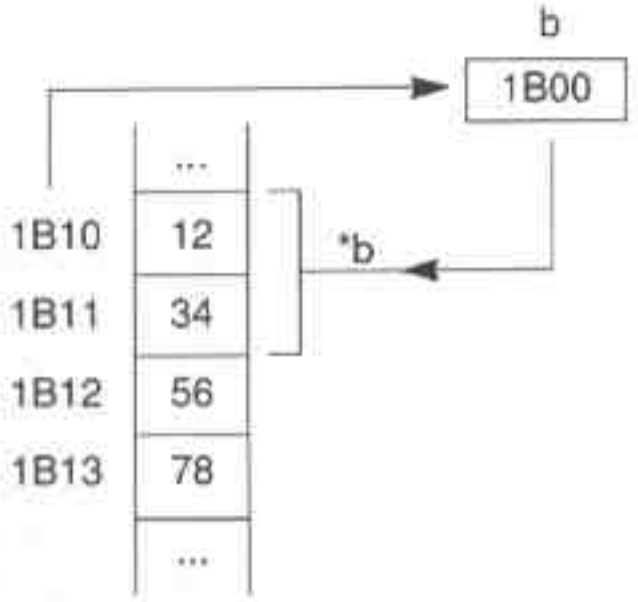
ile ekrana HEX sistemde **5678** sayıları basılır. Aşağıdaki şekli inceleyiniz. `b` değişkeninin `1810` adresinden başlayarak yerleştiği varsayılmıştır.



2) Düşük anlamlı byte değerleri belleğin yüksek anlamlı adresinde bulunacak biçimde. Motorola işlemcileri bu yerleşim biçimini kullanır. Örneğin yukarıdaki kod Motorola işlemcilerinin kullanıldığı bir sistemde yazılmış olsaydı:

```
...
unsigned long a = 0x12345678;
unsigned int *b;

b = (unsigned int *) &a;
printf("%x\n", *b);
```

ekrana HEX sistemde **1234** basılırdı.



26.4 BİRLİK ELEMANLARININ ORGANİZASYONU

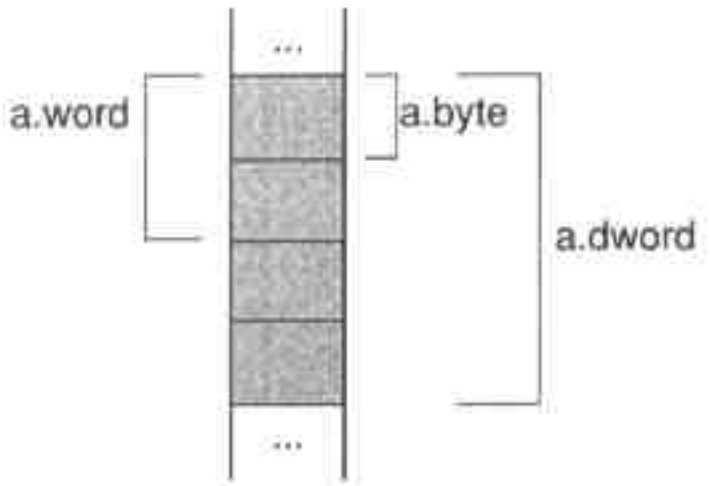
Birlik değişkenleri için birliğin en uzun elemanı kadar yer ayrılır. Birlik elemanlarının hepsi aynı orjinden başlayacak biçimde belleğe yerleşirler. Örneğin:

```
union DWORD {
    unsigned char byte;
```

```
    unsigned  int word;
    unsigned long dword;
};
...
union DWORD a;
```

bildirimi ile **a** değişkeni için 4 byte yer ayrılacaktır. Çünkü **a** değişkeninin **dword** elemanı 4 byte ile birliğin en uzun elemanıdır.

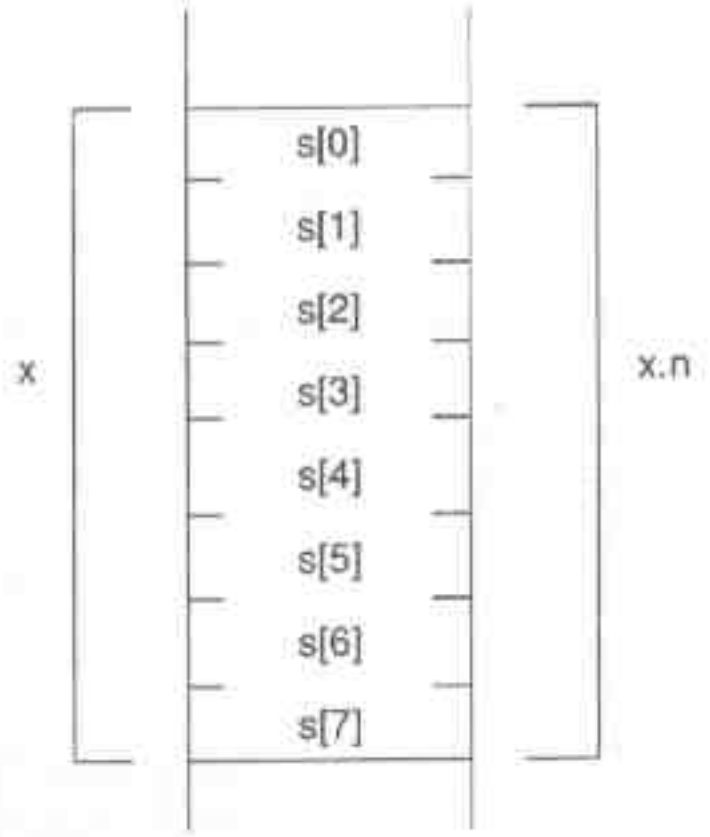
Anımsatma: DOS ve Windows 3.1 gibi 16 bit işletim sistemlerinde sizeof (long) = 4, sizeof(int) = 2'dir. Unix, Windows 95, Windows NT gibi 32 bit işletim sistemlerinde sizeof(long) = 4 ve sizeof(int) = 4'tür.



Birlik bir dizi içeriyorsa dizi tek bir eleman olarak ele alınır. Örneğin:

```
union  DBL_FORM {
    double n;
    unsigned char s[8];
};
...
union DBL_FORM x;
```

tanımlaması ile **x** değişkeni için ne kadar yer ayrılacaktır? **DBL_FORM** birliğinin iki elemanı vardır. Birincisi 8 byte uzunluğunda bir karakter dizisi, ikincisi de 8 byte uzunluğunda **double** bir sayıdır. İki uzunlukta aynı olduğuna göre **x** için 8 byte yer ayrılacağını söyleyebiliriz.



Buna göre :

```
union DBL_FORM x;
...
x.n = 10.2;
```

ile

`x.s[0], x.s[1], x.s[2], ..., x.s[7]` sırasıyla **double** elemanlarının byte değerlerini göstermektedir.

Birlik elemanlarının aynı orjinden başlayarak yerleşmesi bir elemanını değiştirince diğer elemanlarının da içeriğinin değişeceği anlamına gelir. Zaten birliklerin kullanılmasının asıl amacı da budur.

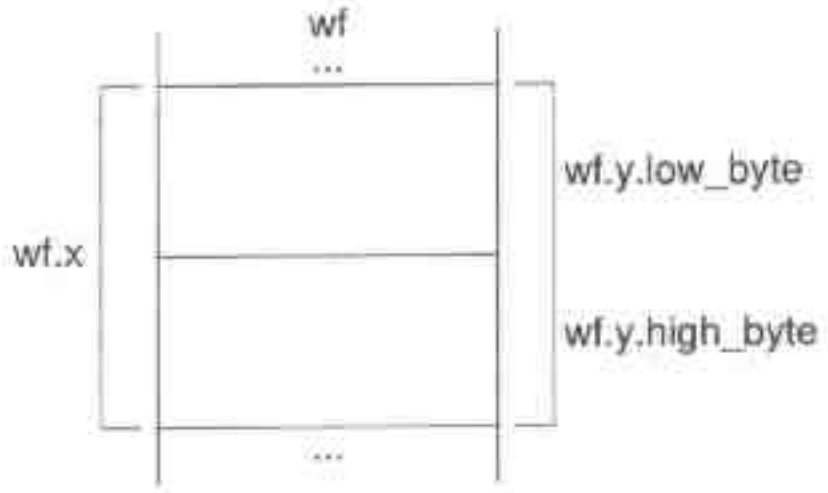
Birlik elemanı olarak yapıların kullanılması uygulamada en sık karşılaşılan durumdur. Örneğin:

```
struct WORD {
    unsigned char low_byte;
    unsigned char high_byte;
};

union WORD_FORM {
    unsigned int x;
    struct WORD y;
};
```

bildirimlerinden sonra `union WORD_FORM` türünden bir birlik tanımlanırsa:

```
union WORD_FORM       wf;
```

bu birliğin alçak (low_byte) ve yüksek (high_byte) anlamlı byte değerlerine ayrı ayrı erişebiliriz; hem de onu bir bütün olarak kullanabiliriz.



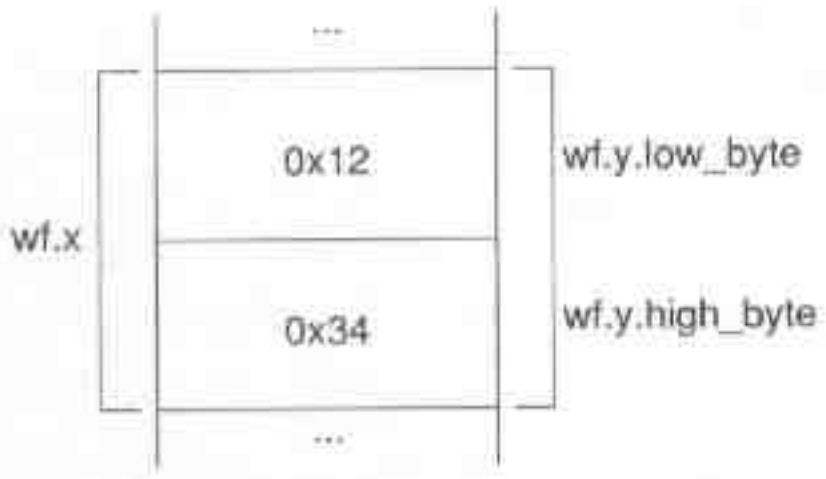
Yani, Intel işlemcilerinin bulunduğu 16 bit sistemlerde:

```
wf.y.low_byte = 0x12;
wf.y.high_byte = 0x34;
```

işlemlerinden sonra:

```
printf("%x\n", wf.y);
```

ile **3412** sayısını ekrana yazdıracaktır. (Motorola işlemcilerinde belleğin düşük anlamlı bölgesinde düşük anlamlı byte değeri olacağına göre sayının ters olarak gözükmesi gerekir.)



80X86 mimarisindeki sistem programlama uygulamalarında sıklıkla rastladığımız başka bir durum **CPU** içerisindeki yazmaçların bir birlik ile temsil edilmesidir. Önce kısaca Intel işlemcilerinin yazmaç yapısından bahsetmek yerinde olacaktır:

26.5 INTEL 80X86 İŞLEMCİLERİNİN YAZMAÇ YAPISI

16 bit Intel 8086 işlemcisinde 14 yazmaç vardır. Bu yazmaçlar işlevlerine göre birkaç gruba ayrılırlar.

4 tanesi genel amaçlı yazmaçlardır: AX, BX, CX, DX

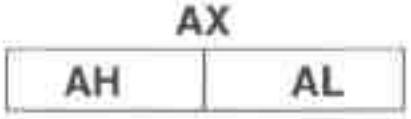
4 tanesi segment yazmaçlarıdır: CS, SS, DS, ES

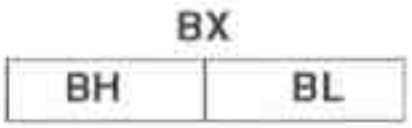
2 tanesi indeks yazmacıdır: SI, DI

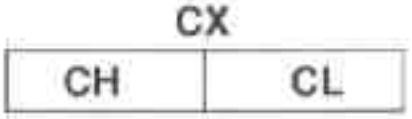
3 tanesi gösterici yazmacıdır: IP, SP, BP

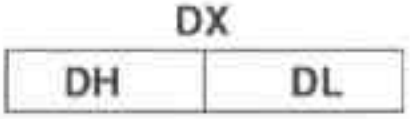
1 tane bayrak yazmacı vardır: F

8086 yazmaçlarının hepsi 16 bittir. Genel amaçlı yazmaçların dışındakiler tek parça olarak kullanılırlar. Ancak genel amaçlı yazmaçlar 8 bitlik iki ayrı yazmaçmış gibi de kullanılabilirler:







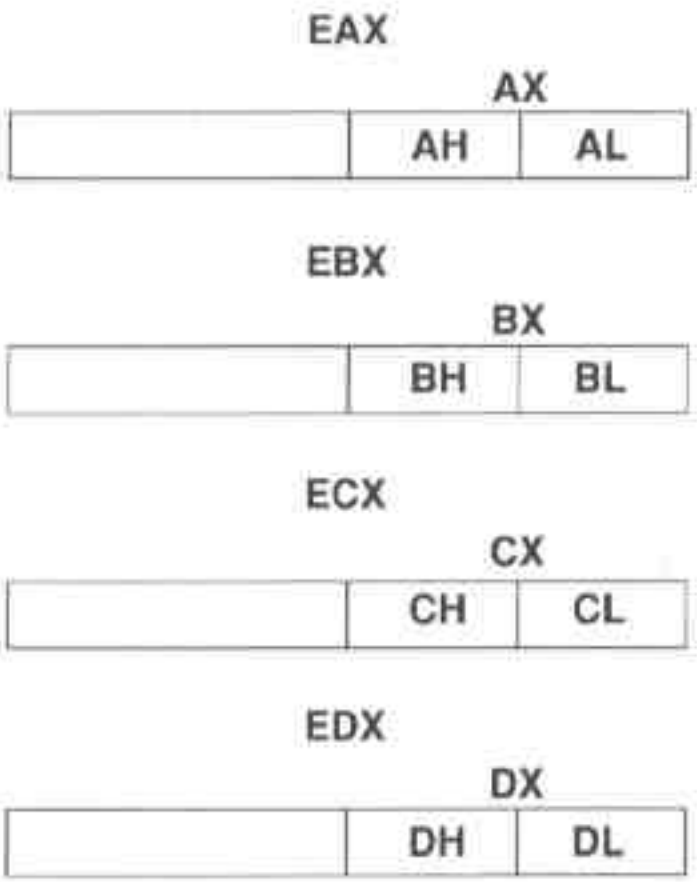


AH ve **AL** yazmaçlarına değer yüklendiğinde bu **AX** yazmacını da bütünsel olarak etkilemektedir. **80386** ve **80486** ve **Pentium** mikroişlemcilerinde -bir çoğu koruma mekanizmasını sağlamaya yönelik- **8086**'dan daha fazla yazmaç vardır. Bu işlemciler 32 bit olduğundan yazmaçları da 32 bittir.

80386 ve **80486** ve **Pentium** işlemcilerindeki genel amaçlı yazmaçlar **8086** ile uyumun korunabilmesi için genişletilmişlerdir. Segment yazmaçları dışındaki yazmaçların başına **E** harfi (İngilizce extended sözcüğünden geliyor) getirilerek genişletilmiş 32 bit biçimleri kullanılabilir:

```
EAX, EBX, ECX, EDX, EBP, EIP, ESP, ...
```

Bu durumda 32 bit genel amaçlı yazmaçlar bağımsız 4 kısımla kullanılabilirler.



Genişletilmiş yazmaçların yüksek anlamlı 16 bitinin ayrık bir biçimde kullanılamadığına dikkat ediniz. 32 bit ailede bunlardan başka:

2 ilave segment yazmacı (16 bit) : `FS, GS`

4 tane korumalı modla ilgili yazmaç: `GDTR, IDTR, LDTR, TR`

3 tane kontrol yazmacı: `CR0, CR1, CR2`

8 debug yazmacı: `DR0, DR1, ...DR7`

5 test yazmacı: `TR3, ..., TR7`

vardır.

26.5.1 8086 Yazmaç Yapısının Yapı ve Birliklerle Temsil Edilmesi

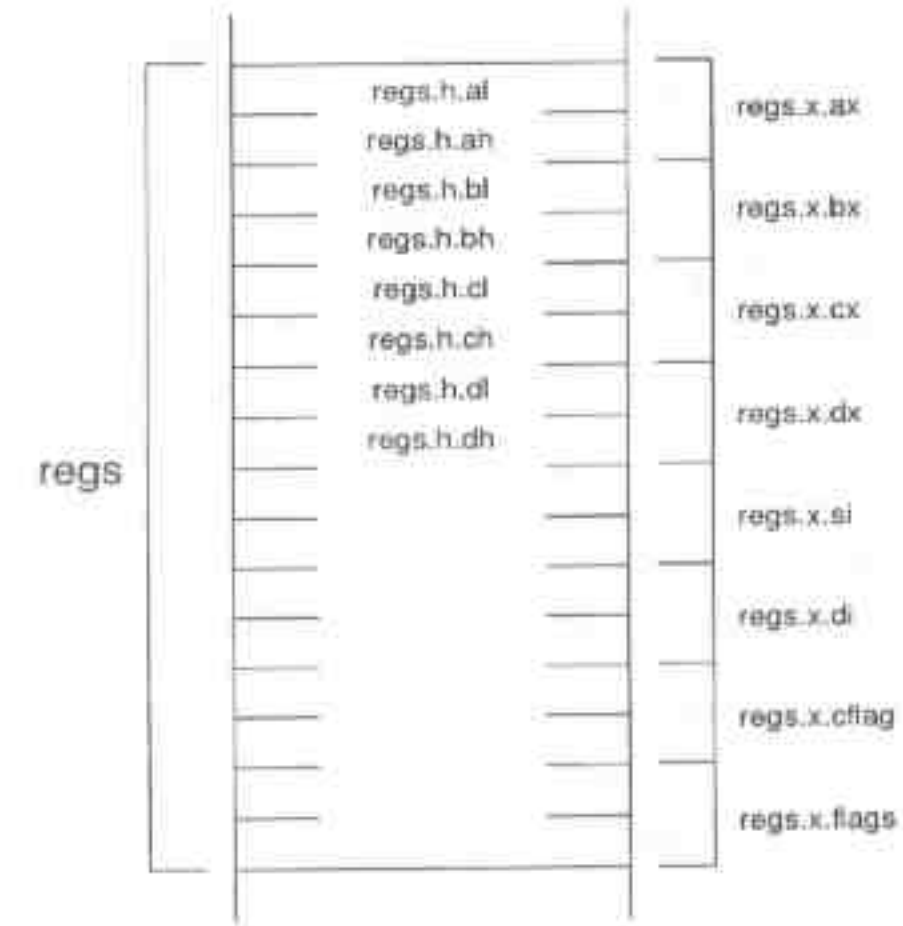
Yazmaçları temsil için iki yapı ve bir birlik kullanılabilir.

```
struct BYTEREGS {
    unsigned char al, ah, bl, bh;
    unsigned char cl, ch, dl, dh;
};

struct WORDREGS {
    unsigned int ax, bx, cx, dx;
    unsigned int si, di, cflag, flags;
};

union REGS {
    struct WORDREGS x;
    struct BYTEREGS h;
};
```

`union REGS` türünden bir değişken tanımlayalım:

```
union REGS regs;
```

`regs` birliğinin elemanlarının bellekteki yerleşimi şöyledir:



`union REGS` türünün uzunluğunu **sizeof** (`union REGS`) ifadesi ile derleyiciye hesaplattırabilirsiniz.

`regs.h.ah`	`AX` yazmacının yüksek anlamlı byte değeri
`regs.h.al`	`AX` yazmacının düşük anlamlı byte değeri
`regs.x.ax`	`AX` yazmacının tamamı

...

Tabi `union REGS` biçiminde tanımladığımız bir değişken bellekte bir yerdedir. Oysa mikroişlemcinin yazmaçları kendi içerisindedir. Biz yalnızca bu birlik ile yazmaç takımlarının bellekte bir benzetimini yapmış olduk.

26.6 BİRLİK KULLANMANIN YARARLARI

Birlikler iki amaçla kullanılabilir. Birincisi yer kazancı sağlamak içindir. Birlik kullanarak farklı zamanlarda kullanılacak birden fazla değişken için ayrı ayrı yer ayırma zorunluluğu ortadan kaldırılır. Örneğin bir işletmede çalışan kişilerin maaş bilgilerini tutan bir yapı düşünün. Çalışanların bir kısmının banka hesap numaraları, banka hesap numaraları olmayanların sıra numaraları kullanıyor olsun. Bu iki bilgide aynı anda kullanılmayacaksa aşağıdaki gibi bir tanımlama yer kaybına yol açacaktır:

```
struct PERSON {
    char adi_soyadi[30];
    char departman[20];
    long maas;
    char hesap_no[15];
    int sira_no;
    ...
};
```

Oysa `hesap_no` ve `sira_no` bir birlik içinde ifade edilirse daha etkin bir yerleşim sağlanır.

```
struct PERSON {
    char adi_soyadi[30];
    char departman[20];
    long maas;
    union {
        char hesap_no[15];
        int sira_no;
    } hesap_sira_no;
    ...
};
```

Birlik kullanmanın ikinci nedeni "bir bütünün parçaları üzerinde işlem yapmak ya da parçalar üzerinde işlem yaparak bütünü oluşturmaktır." Bunun için `int` türünden bir sayının byte değerlerini ayrıştırma örneğini yineleyebiliriz:

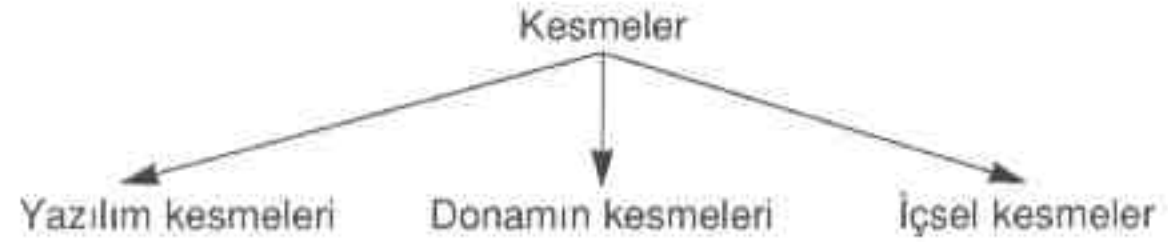
```
struct WORD {
    unsigned char low_byte;
    unsigned char high_byte;
};

union WORD_FORM {
    unsigned int x;
    struct WORD y;
};

union WORD_FORM wf;
```

Tanımlamalarından sonra `wf.x` elemanına değer atadığımızda `wf.y.low_byte` ve `wf.y.high_byte` ile yüksek ve alçak anlamlı byte değerleri üzerinde kolaylıkla işlem yapabiliriz.

Birlikler konusunda uygulama amacıyla **80X86** sistemindeki kesmeler konusu ele alınacaktır.

26.7 KESMELER (Interrupts)

Kesmeler mikroişlemcinin o anda çalıştırdığı koda ara vererek başka bir kodun icra edilmesi durumudur. Kesmeler tanım olarak normal fonksiyon çağırmalarına benzese de birtakım farklılıkları vardır. Kesmeleri çağrılma biçimlerine göre üç gruba ayırabiliriz.



Yazılım Kesmeleri (software interrupts): Bunlar programcının `INT` makina komutuyla programa dahil ettiği kesmelerdir. İşlevsel olarak fonksiyon çağırmadan büyük bir farkı yoktur.

80X86 Sembolik Makina Dili Programcısına Not: `INT` makina komutunun dolaylı segmentler arası fonksiyon çağırmadan (indirect intersegment call) iki farkı vardır. Birincisi dolaylı segmentler arası fonksiyon çağırmada çağrılacak fonksiyonun adresi herhangi bir yerde olabilir:

```
target_fonk          dd          1FC01800H
call          dword ptr          target_fonk
```

Oysa `INT N` gibi bir kesme çağırmasında kesme kodu (interrupt handler) gerçek modda belleğin tepesinde bulunan kesme vektöründen komut icrası sırasında otomatik olarak alınmaktadır. İkinci farklılık, kesme çağırmasında `CS` ve `IP` yazmaçlarının yanı sıra bayrak yazmacının da yığına atılmasıdır. Segmentler arası dolaylı fonksiyon çağırmada yalnızca `CS` ve `IP` yazmaçları yığına atılır.

Donanım Kesmeleri (hardware interrupts): Gerçek kesmeler donanım kesmeleridir. Donanım kesmeleri dışsal bir birim tarafından mikroişlemcinin `INT` ucunun aktive edilmesi yoluyla çağrılırlar. Intel tabanlı **PC** mimarisinde dışsal kesme birimleriyle mikroişlemci arasındaki iletişimi "kesme denetleyicisi (interrupt controller)" diye isimlendirilen bir işlemci kurar. **PC** mimarisinde toplam 15 ayrı dışsal kesme tanımlanabilir. (Bu dışsal kesmeler **IRQ** diye de isimlendirilmektedir.) Dışsal kesmelerin çoğu tasarımsal olarak belli birimlere atanmışlardır. Örneğin 8 numaralı dışsal kesme **8254** zamanlayıcısı tarafından saniyede yaklaşık **18.2** kere çağrılarak birtakım işlemleri yerine getirir. Yine 9 numaralı dışsal kesme klavyenin tuşuna her basıldığında çağrılır. Klavyenin tuşuna her basıldığında 9H kesme kodu **8042** klavye denetleyicisinden basılan tuşun **scan kodu**'nu alarak belleğin düşük anlamlı bölgesinde bulunan klavye tamponuna yazmaktadır. Böylece basılan tuşun kodları bellekte kullanıma hazır bir biçime getirilmiş olur. Bunların dışında örneğin ekrandaki farenin konum değiştirmesini sağlayan da bir donanım kesmesidir. Bu kesme seri porttan farenin içerisindeki bir kontrol birimi yardımıyla periyodik olarak verilerek farenin hareket etmesini sağlar.

İçsel Kesmeler (internal interrupts): Mikroişlemcinin bir makina kodunu icra ederken problemle karşılaştığında kendisinin çağırdığı kesmelerdir. Örneğin mikroişlemci `DIV` komutunu icra ederken bölen değerin sıfır olduğunu görürse "0 nolu sıfıra bölme (divide by zero)" kesmesini çağırır. Intel'in **80X86** mikroişlemcileri korumalı modun sürekliliğini sağlamak için çok sayıda içsel kesme kullanırlar. İşletim sistemi tasarımcıları bu kesmeleri bellek ve proses kontrollerini sağlamak amacıyla yönlendirmektedir.

26.8 INTEL İŞLEMCİLERİNDE KESMELER

Intel işlemcilerinin toplam 256 kesme kapasitesi vardır. Her kesme bir numarayla ve genellikle HEX sistem kullanılarak belirtilir. Örneğin: `10H` kesmesi, `21H` kesmesi gibi. Kesmenin türü ne olursa olsun (yazılım kesmesi, donanım kesmesi ya da içsel kesme) mikroişlemci kesme ile karşılaşığında aynı biçimde davranır. Tıpkı fonksiyon çağırmasında olduğu gibi, programa ara vererek kesme kodunu icra eder sonra programda kalınan yerden devam eder.

80X86 Sembolik Makina Dili Programcısına Not: 80X86 mikroişlemcileri gerçek modda dışsal bir kesme geldiğinde ya da `INT` komutunun icrası sırasında;

```
PUSHF
PUSH CS
PUSH IP
```

işlemleri ile kesmeden sonraki komutun adresini ve bayrak yazmacını yığına attıktan sonra kesme kodunun adresini belleğin tepesinde bulunan kesme vektöründen alırlar. `INT` komutu içerisinde bu işlemler otomatik olarak yapılmaktadır. Kesmeden dönüş için `IRET` komutu kullanılır. Bu komut sırasıyla:

```
POP IP
POP CS
POPF
```

yaparak kalınan yerden işlemin devamını sağlar.

Intel **80386, 80486** ve **Pentium** işlemcilerinin korumalı moddaki kesme mekanizması karmaşıktır. Kesme kodlarının yerlerini gösteren kesme vektörü yerine her biri 8 byte uzunluğunda bir kesme betimleyici tablosu kullanılır.

Intel işlemcilerinde 256 kesme olsa da dallanma yoluyla bu sayı istenildiği kadar çoğaltılabilir. Bir kesmenin farklı yollarına fonksiyon denir. Örneğin `10H` kesmesinin birçok fonksiyonu vardır. Fonksiyon numarası genellikle `AH` yazmacının içerisine konur. (`AH` yazmacı 8 bit olduğuna göre bir kesmenin en fazla 256 fonksiyonu olabilir). Fonksiyonlar da yine alt fonksiyonlar biçiminde dallara ayrılabilirler. Alt fonksiyon numarası ise genellikle `AL` yazmacının içerisinde saklanmaktadır. Tabi bir kesmenin fonksiyonu ya da alt fonksiyonu olması zorunlu değildir.

Kesmeler de tıpkı fonksiyonlar gibi parametre alabilir ve geri dönüş değeri verebilirler. Kesme parametreleri kesme çağrılmadan önce yazmaçlara yerleştirilir.

Kesmelerin geri dönüş değerleri kesme çağrıldıktan sonra yazmaçlardan alınır. Kesme parametrelerinin hangi yazmaçlara yerleştirileceği ve geri dönüş değerlerinin hangi yazmaçlardan alınacağı önceden belirlenmiştir.

26.9 KESMELERİN ÇAĞRILMASI

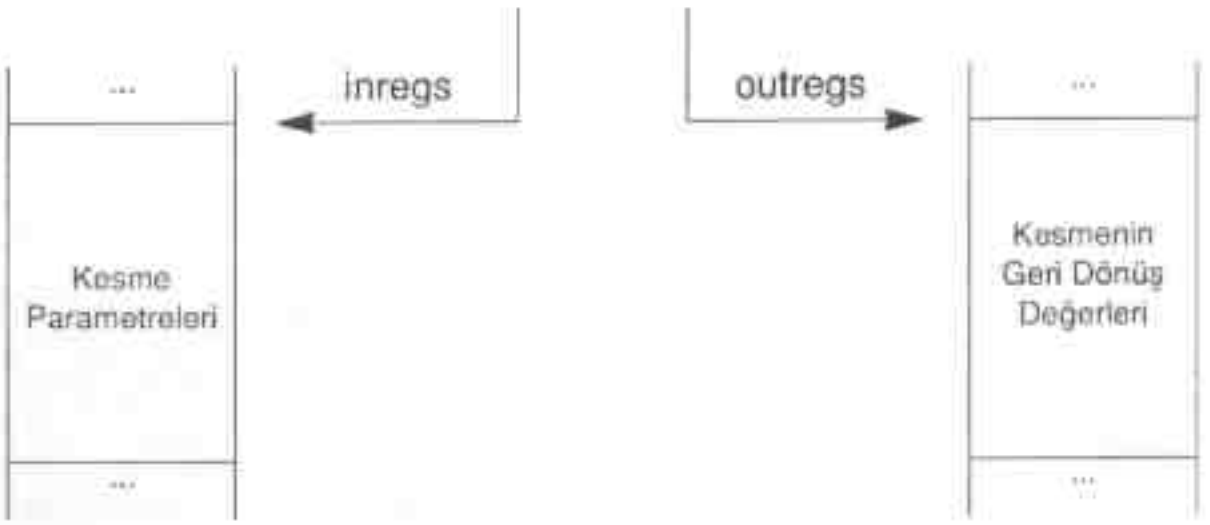
80X86 tabanlı mikroişlemcilerde yazmaçlar bellekte `union REGS` ile temsil edildikten sonra özel fonksiyonlarla kesme çağrılmaktadır. `REGS` birliğini yukarıda ele almıştık. Bu birlik **DOS.H** başlık dosyası içerisinde bildirilmiştir. Kesme çağıran fonksiyonlar bellekteki yazmaç benzetimini alarak bunları mikroişlemcinin gerçek yazmaçlarına kopyalar ve kesmeyi çağırırlar. Bu fonksiyonlar kesme sonrasındaki yazmaç değeryerini yine programcıya geri verirler. Aşağıda kesme çağırma amacıyla en sık kullanılan `int86` fonksiyonu ele alınmaktadır.

int86 Fonksiyonu

Kesme çağırmak için en sık kullanılan fonksiyon budur. Prototipi **DOS.H** içerisinde bildirilmiştir; inceleyiniz:

```
int int86 (int interrup_no, union REGS *inregs, union REGS *outregs);
```

Bu fonksiyon ikinci parametresiyle belirtilen adresteki yazmaç bilgilerini alarak gerçek **CPU** yazmaçlarına kopyalar. Daha sonra birinci parametresiyle belirtilen kesmeyi çağırır. `int86` kesme sonunda CPU yazmaçlarının durumunu üçüncü parametresiyle belirtilen adresten itibaren tekrar belleğe yazmaktadır.



Kesme çağrılmadan önce kesme parametrelerinin `union REGS` biçiminde ifade edilerek adresinin fonksiyonun ikinci parametresine geçirilmesi gerekir. Fonksiyon çağrıldıktan sonra `int86` bu bilgileri CPU yazmaçlarına kopyalayarak kesmeyi çağıracaktır. Kesme sonunda kesmenin geri dönüş değerleri yine bu fonksiyon tarafından CPU yazmaçlarından alarak üçüncü parametresiyle belirtilen adrese yazılır.

26.10 KESMELERLE İLGİLİ ÖRNEKLER

10H kesmesi temel video işlemleri için kullanılır. Kendi içerisinde pekçok fonksiyona ve alt fonksiyona ayrılmaktadır. Aşağıda 10H ile ilgili birkaç örnek göreceksiniz.

Video modun değiştirilmesi

```
INT 10h, Fonksiyon 00H
```

Parametreler:

AH: Fonksiyon numarası

AL: Video mod

Geri Dönüş Değeri: Yok

Video modun bir byte uzunluğunda bir sayı olduğunu anımsayınız. union REGS bildirimi için **DOS.H** dosyasını kaynak koda dahil etmelisiniz.

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

void set_video_mode(int mode)
{
    union REGS regs;

    regs.h.ah = 0;
    regs.h.al = mode;
    int86(0x10, &regs, &regs);
}
```

int86 fonksiyonun ikinci parametresiyle üçüncü parametresinin aynı olduğuna dikkat ediniz. Bu durumda kesme çağrıldıktan sonra geri dönüş değerleri aynı adrese yazılarak parametre bilgilerini bozacaktır. Parametre bilgilerini kesme çağrıldıktan sonra kullanmayacağımıza göre bozulmasının da bizim için bir önemi olamaz.

Video modun alınması

```
INT 10h, Fonksiyon 0FH
```

Parametreler:

AH: Fonksiyon numarası

Geri Donüş Değeri:

AL: video mod

```
int get_video_mode(void)
{
```

```
    union REGS regs;

    regs.h.ah = 0x0F;
    int86(0x10, &regs, &regs);
    return regs.h.al
}
```

get_video_mode o andaki video modun (current video mode) hangisi olduğunu bulmakta kullanılır. Bu fonksiyonlar çalıştığınız derleyicinin kütüphanelerinde başka isimlerle zaten olabilirler. Dilerseniz uygulamalarda doğrudan onları da kullanabilirsiniz. Ancak o fonksiyonları yazanlar da yukarıdakilerden farklı birşey yapmamışlardır. Bu iki kesme fonksiyonunu aşağıdaki kod ile deneyebilirsiniz.

```
void main(void)
{
    int mode;

    mode = get_video_mode();
    set_video_mode(1);
    printf("1 numaralı video mod\n");
    getch();
    set_video_mode(mode);
}
```

Deneme kodumuzda o andaki video mod bir değişkende saklandıktan sonra 1 numaralı video moda geçilmiştir. Herhangi bir tuşa basarsanız eski video moda tekrar geri dönersiniz. Ekranda çıkacak yazının iriliği sizi şaşırtmasın. Çünkü 1 numaralı video mod VGA kartlarında 40X25 çözünürlüğe sahiptir.

İmlecin Yerleştirilmesi

`INT 10h, Fonksiyon 02H`

Parametreler:

`AH` : Fonksiyon numarası

`DH` : İmlecin yerleştirileceği satırın numarası (ilk satır 0)

`DL` : İmlecin yerleştirileceği sütunun numarası (ilk sütun 0)

`BH` : Sayfa numarası (0 olmalı)

Geri Dönüş Değeri: Yok

İmlecin (cursor) konumu ancak yukarıdaki kesme ile değiştirilebilir. C'nin bütün standart çıkış fonksiyonlarının imlecin bulunduğu yerden itibaren ekrana yazdığını unutmayınız. `printf`, `puts` gibi standart fonksiyonlarla ekranın istediğiniz bir yerine birşey yazmak istiyorsanız önce imleci oraya yerleştirmeniz gerekir. Bu fonksiyonun ismine de **pos** diyelim.

```
void pos(int row, int col)
{
    union REGS regs;

    regs.h.ah = 2;
    regs.h.dh = row;
    regs.h.dl = col;
    regs.h.bh = 0;
    int86(0x10, &regs, &regs);
}
```

Microsoft ve Borland derleyicilerinde aynı işlemi yapan **gotoxy** fonksiyonu vardır. O fonksiyon da bu kesme çağrılarak yazılmıştır. Ancak bizim yazdığımız **pos** fonksiyonunda ilk parametrenin satır ikinci parametrenin sütun olduğunu belirtelim. Ayrıca **pos** fonksiyonunda ekranın orijin noktası **(0,0)** koordinatıdır. (**gotoxy** fonksiyonunda **(1,1)**'dir).

Aşağıdaki kod ile fonksiyonu test edebilirsiniz:

```
void main(void)
{
    pos(0, 0);
    getch();
    pos(24, 79);
    getch();
    pos(0, 0);
    getch();
    pos(25, 80);          /* Menzil dışı; imleç görünmez olur */
    getch();
    pos(10, 10);
    getch();
    pos(0, 0);
    getch();
}
```

İmleci görünmez yapmanın en iyi yolu menzil dışına çıkarmaktır. Örneğimizde **pos(25, 80)** ile imleç görünmez yapılmıştır.

Yukarıdaki kesmeler konusunda uygulama yapmak amacıyla yalnızca birkaç örnek verdik. Kesmeler konusunda ayrıntılı bilgi için yeterli bir kaynağa başvurabilirsiniz.

SORAMADIKLARINIZ...

S1) Kesme kodları belleğin neresinde bulunur? Örneğin 10H kesmesinin program kodu belleğin neresindedir?

C1) Kesme kodlarının bir bölümü EPROM içerisindedir. Bu tür kesmelere BIOS kesmeleri denir. BIOS kesmeleri işletim sisteminden bağımsız aşağı seviyeli işlemleri yerine getirmektedir. Örneğin video işlemlerini yapan 10H, klavye bilgilerinin alınmasını sağlayan 16H ve disk işlemlerini gerçekleştiren 13H birer BIOS kesmesidir. Çünkü işletim sistemi ne olursa olsun bu işlemlere gereksinim vardır. Zaten bu yüzden bu kesme kodları silinmemek üzere EPROM içerisine konulmuştur. Bazı kesme kodları işletim sistemi tarafından belleğe yüklenirler. Örneğin 21H kesmesinin fonksiyonları DOS işlemlerini yapar ve DOS ile belleğe yüklenmektedir. Bazı kesme kodları da belli programların çalıştırılması ile belleğe yüklenirler. Örneğin fare ile haberleşmeyi sağlayan 33H fare sürücüsünün (DOS'ta mouse.com ya da mouse.sys) yüklenmesiyle kalıcı olarak belleğe yerleşir.

S2) `union REGS` içerisinde **CPU** yazmaçlarının bir kısmı var? Peki diğer yazmaçlar parametre ya da geri dönüş değerleri için kullanılmıyor mu?

C2) Diğer yazmaçlar kesme işlemlerinde nadir olarak kullanılırlar. Bu nedenle `union REGS` içerisinde belirtilmemişlerdir. Ancak başka kesme fonksiyonları ile bu yazmaçlara erişebilirsiniz.

27 BİT ALANLARI

Önceki bölümlerde C'nin en küçük türünün **char** olduğunu gördük. Gerçekten de mikroişlemcilerin üzerinde işlem yaptığı en küçük tür bir byte uzunluğundadır. Yani mikroişlemciler bellekten en az bir byte bilgiyi yazmaçlarına çekerek işleyebilirler ve en az bir byte bilgiyi belleğe yazabilirler. Bu durumda eğer bir byte bilginin belli bitleri değiştirilecekse önce o bilgi bellekten byte olarak CPU yazmaçlarına çekilir; üzerinde bit işlemleri yapıldıktan sonra yine byte olarak belleğe geri yazılır. C'de bit operatörleriyle sayıyı oluşturan bitler üzerinde işlemlerin nasıl yapıldığını operatörleri anlattığımız 9. bölümde görmüştük. Bit alanları yapıların daha özel bir biçimidir ve nesnelere bit düzeyinde erişmek için kullanılırlar.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Bit alanlarının bildirimi nasıl yapılır?

2) Bit alanlarının bellekteki organizasyonu nasıldır?

3) Bit alanlarına neden gereksinim duyulmuktadır?

27.1 BİT ALANLARININ BİLDİRİMİ

4) DOS'un tarih ve zaman formatı nasıldır ve bit alanlarıyla nasıl ifade edilir?

Bit alanlarının bildirimi yapılara ve birliklere benzer. Genel bildirim biçimi aşağıdaki gibidir:

```
<struct [union]>  [bit alanının ismi] {
    <tür> bit_elemanı : n1;
    <tür> bit_elemanı : n2;
    <tür> bit_elemanı : n3;
    ...
};
```

`tür`	`unsigned [int]` ya da `[signed] int` olabilir.
`n1, n2, n3,...`	elemanların bit uzunluklarını belirten tam sayılardır.

Klasik yapı ve birlik bildirimlerinden farklı olarak bit alanlarının bildirimlerinde, ilgili elemanın kaç bit uzunluğunda olduğunu belirtmeye yarayan "iki nokta üst üsteden sonra bir tam sayı" bulunmaktadır. Şimdi birkaç örnek verelim:

```
struct DATE {
    unsigned day: 5;
    unsigned month: 4;
    unsigned year: 7;
};
```

Örneğimizdeki **DATE** isimli bit alanı üç elemandan oluşmaktadır. Sırasıyla **day** isimli eleman için 5 bit, **month** isimli eleman için 4 bit ve **year** isimli eleman için ise 7 bit yer ayrılması istenmiştir. Örneğin **day** elemanı unsigned biçiminde tanımlandığından içerisine **0** ile **31** arasında ([**0, +2^5-1**]) tam sayılar konulabilir. Yine benzer biçimde **month** elemanına **0** ile **15**, **year** elemanına da **0** ile **127** arasında sayılar yerleştirilebilir.

Aşağıdaki örneği inceleyiniz:

```
struct DENEME {
    int a: 3;
    int b: 6;
    unsigned c: 7;
};
```

DENEME isimli bit alanının **a** elemanı **int** türünden olduğu için içerisine ancak **-4** ile **+3** ([-2^2, $+2^2-1$]) arasında sayılar konulabilir. (**n** bit ile yazılabilecek işaretli sayı sınırlarının ikiye tümleyen aritmetiğine göre [-2^{n-1}, $+2^{n-1}-1$] olduğunu anımsayınız.). Benzer biçimde **b** elemanı içerisine de **-32** ile **+31** arasında ([-2^5, $+2^5-1$]) sayılar konulabilir.

27.2 BİT ALANI DEĞİŞKENLERİNİN TANIMLANMASI

Bit alanı değişkenlerinin tanımlanması yapılar ve birliklerde olduğu gibidir. Örneğin:

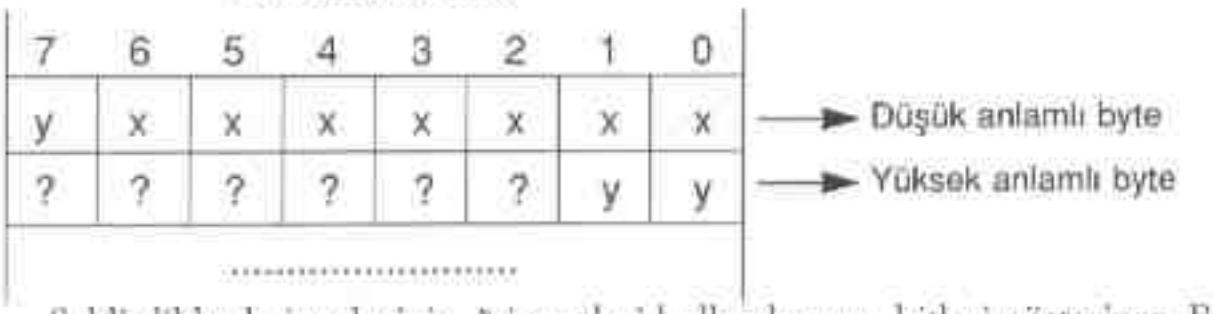
```
struct DATE x;
```

ile **x**, **struct DATE** türünden bir bit alanı değişkeni olarak tanımlanmıştır. Ancak derleyicilerin bit alanı değişkenleri için ayıracakları yer hizalama seçeneğine göre değişebilir. **Derleyiciler bit alanlarıyla karşılaştıklarında yapılarda olduğu gibi hizalama (alignment) seçeneğine bağlı olarak byte (byte alignment) word (word alignment) ya da double word'ün katlarıyla (double word alignment) yer ayırırlar.** Örneğin derleyicinizin hizalama seçeneği **word** hizalamada ise, bit alanının

elemanları toplamı 5 bit bile olsa derleyicinizin bit alanı değişkeni için ayıracağı yer 2 byte olacaktır. Çünkü hizalama seçeneğine göre derleyiciniz bit alanı değişkenini içine alabilecek byte, word ya da double word katlarıyla yer ayırmaktadır.

```
struct TEST {
    unsigned x: 7     /* 0...127 */
    unsigned y: 3;    /* 0...7   */
};
...
struct TEST a;
```

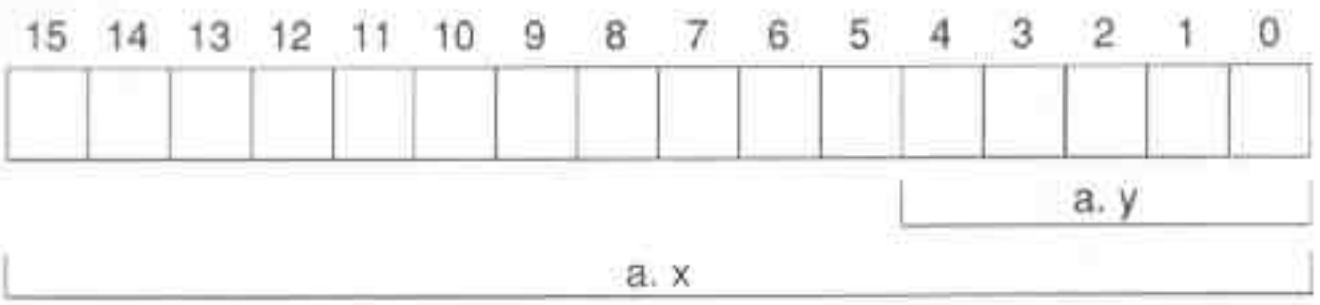
Yukarıdaki örnekte derleyiciniz a değişkeni için hem byte hem de word hizalama seçeneği için 2 byte yer ayırır. Ayrıca bit elemanlarının bellekteki yerleşimi de taşınabilir değildir. **Intel işlemcilerinin kullanıldığı sistemlerde ilk yazılan eleman düşük anlamlı adresteki düşük anlamlı bitlerde bulunacak biçimde bir yerleşim öngörülmüştür.** Buna göre yukarıdaki TEST türünden tanımlanmış olan a değişkeni bellekte aşağıdaki gibi yerleşir:



Şekli dikkatle inceleyiniz. ? işaretleri kullanılmayan bitleri gösteriyor. Byte ve word hizalama seçeneğinin her ikisinde de derleyici a için 2 byte yer ayıracaksa 6 bitin boş bırakılmasının bir anlamı olabilir mi?

Bit alanları birliklerden de oluşturulabilir. Bu durumda birlik kuralları geçerlidir. Yani elemanları oluşturan bitler aynı orijinden başlayarak yerleştirilirler. Örneğin:

```
union SAMPLE {
    unsigned x;
    unsigned y: 5;
};
...
union SAMPLE a;
```



Şekilden de gördüğümüz gibi `a.x`'in düşük anlamlı 5 biti `a.y`'dir. Birliklerden oluşan bit alanlarına uygulamalarda birkaç durum dışında ihtiyaç duyulmaz.

27.3 BİT ALANLARINA İLİŞKİN UYGULAMALAR

Bit alanları özellikle herbiri az sayıda seçenekten oluşan bilgileri tutmak için kullanılır. Örneğin 8 ayrı makinanın açık mı kapalı mı olduğu bilgisini tutmak için 8 ayrı `char` değişkenine gerek yoktur. Açık ya da kapalı olma durumu bir bitlik bit alanı elemanlarıyla ifade edilebilir.

```
struct DURUM {
    unsigned m1: 1;
    unsigned m2: 1;
    unsigned m3: 1;
    unsigned m4: 1;
    unsigned m5: 1;
    unsigned m6: 1;
    unsigned m7: 1;
    unsigned m8: 1;
};
```

Benzer biçimde örneğin bir dersanede öğrencilerin test sorularına verdiği yanıtlar bir dizide tutulacak olsa şöyle bir bit alanı tasarlanabilir.

```
struct TEST {
    unsigned s1: 3;    /* 0...7 (0 = a, 1 = b, 2 = c, 3 = d, 4 = e) */
    unsigned c1: 1;    /* 0...1 (0 = Yanlış, 1 = Doğru) */
    unsigned s2: 3;
    unsigned c2: 1;
};
```

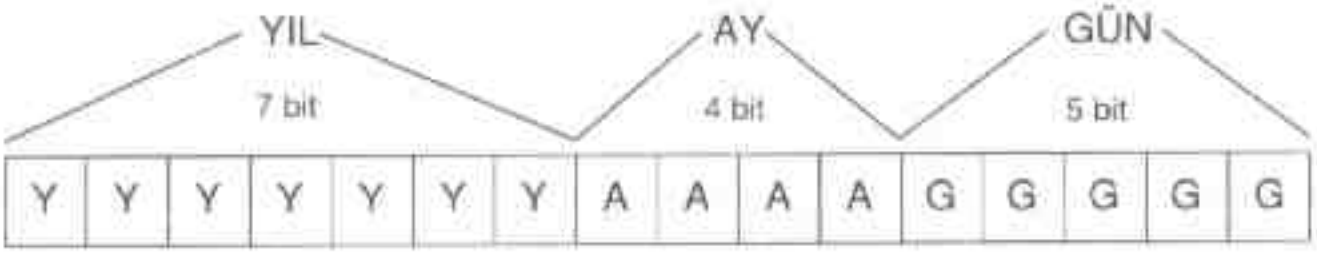
İki sorunun yanıtı ve sonucu bit alanı kullanılarak bir byte'a sığdırılmıştır. Böyle bir bit alanı cinsinden açılacak 50 elemanlı bir dizide ise toplam 100 sorunun bilgisi saklanabilir.

```
struct TEST test[50];
....
```

Programcı n. sorunun yanıtına n'in çift ya da tek olmasına göre `test[n/2].s1` ya da `test[n/2].s2` ifadeleriyle erişebilir.

27.3.1 DOS'un Tarih ve Zaman Formatları

Programlama dillerinin ve paket programların bir bölümü tarih ve zaman bilgisini karakter alanları içerisinde tutarlar. Tarih bilgisi karakter alanı içerisinde **gg/aa/yyyy** biçiminde ancak 8 byte ile tutulabilir. (Tabi ayıraç olarak kullanılan / işaretlerini saklamaya gerek yoktur; bunları program basabilir.) Oysa DOS işletim sistemi dosyalara ilişkin tarih ve zaman bilgilerini **2** byte ile ifade etmektedir. Tarih ve zaman bilgilerinin bit düzeyinde iki byte içerisinde tutulması oldukça ekonomik bir saklama biçimidir.



Şekilden de gördüğümüz gibi gün için 5 bit (0-31), ay için 4 bit (0-15) ve yıl için 7 bit (0-127) ayrılmıştır. Bu yapı bir bit alanı içinde saklanabilir.

```
struct DATE {
    unsigned day: 5;            /* 1 ... 31 */
    unsigned month: 4;          /* 1 ... 12 */
    unsigned year: 7;           /* 1980 ... 2107 */
};
```

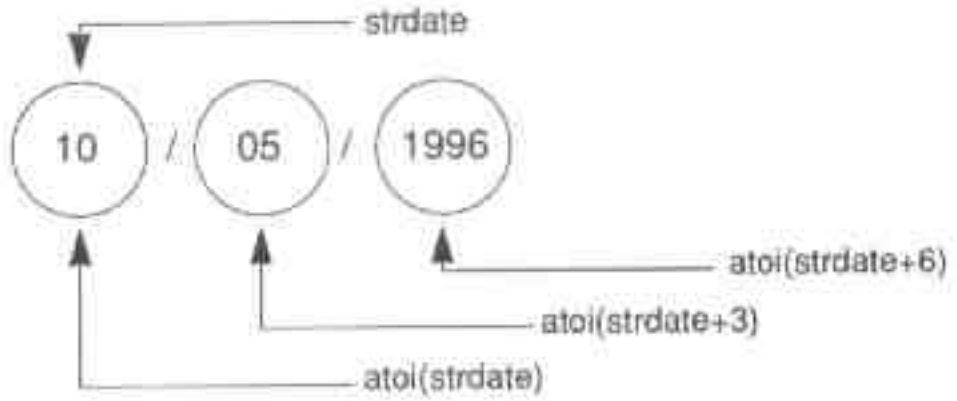
Yıl bilgisi için 7 bit ayrıldığına şaşırmayın. DOS her zaman bu alanda yazılana 1980 katar. Örneğin 1996 için bu alanda 16 yazmalıdır. Siz de tarih bilgilerini programlarınızda yukarıdaki formatta saklayabilirsiniz. Bilgilerin bu biçimde tutulması tarih yoğun işlemlerde yer kazancı sağlayarak veri tabanlarınızın performansını yükseltecektir.

DATE türünden bir bit alanı tanımlamakla işimiz bitmiyor. Program içerisinde tarih işlemlerini kolaylıkla yapabilmemiz için bir dizi arabirim fonksiyon da tasarlamamız gerekir. Aşağıdaki örnekte karakterden **DATE** bit alanına dönüşüm yapan **getDATE** ve **setDATE** fonksiyonları ile iki tarihi karşılaştıran **DATEcmp** fonksiyonları tasarlanmıştır. Ayrıca bu fonksiyonları denemeniz için küçük bir de bir de **main** yazdık.

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

struct DATE {
    unsigned day: 4;
    unsigned month: 5;
    unsigned year: 7;
};

void setDATE(struct DATE *date, char *strdate)
{
```

```
    date -> day = atoi(strdate);
    date -> month = atoi(strdate + 3);
    date -> year = atoi(strdate + 6) - 1980;
}

char *getDATE(DATE *date, char *strdate)
{
    sprintf(strdate,"%02d/%02d/%02d", date -> day, date -> month,
    date -> year + 1980);
    return strdate;
}

int DATEcmp(DATE *date1, DATE *date2)
{
    if (date1 -> year != date2 -> year)
        return date1 -> year - date2 -> year;
    if (date1 -> month != date2 -> month)
        return date1 -> month - date2 -> month;
    if (date1 -> day != date2 -> day)
        return date1 -> day - date2 -> day;
    return 0;
}

void main(void)
{
    DATE d1, d2;
    int result;
    char s1[11];
    char s2[11];

    setDATE(&d1, "10/10/1996");
    setDATE(&d2, "10/05/1996");
    result = DATEcmp(&d1, &d2);
    if (result > 0)
        printf("d1 > d2\n");
    else
        if (result < 0)
            printf("d1 < d2\n");
        else
            printf("d1 == d2\n");
    getDATE(&d1, s1);
    getDATE(&d2, s2);
    printf("%s\n%s\n", s1, s2);
}
```

Fonksiyonlar üzerinde kısaca durmak istiyoruz. **setDATE** karakter dizisi biçiminde verilen tarih bilgisini **DATE** bit alanı biçimine dönüştürüyor. Fonksiyonun iki gösterici parametresine sahip olduğuna dikkat ediniz. **atoi** fonksiyonun ilk sayısal olmayan karakter gördüğünde işlemi sonlandırdığını anımsayınız. Bu özelliği sayesinde tarih dizisi bileşenlerine kolaylıkla ayrılabilmiştir.



`getDATE` fonksiyonu **DATE** bit alanını karakter formatına dönüştürmektedir. Bu fonksiyonda `sprintf` isimli standart *C* fonksiyonu çağrılmıştır. `sprintf` fonksiyonunun normal `printf` fonksiyonundan tek farkı birinci parametresidir. `sprintf` bilgiyi ekran yerine birinci parametresiyle belirtilmiş olan karakter dizisine yazar.

`DATEcmp` ise iki tarihi karşılaştırıyor. Kontrolün yıl, ay, gün biçiminde yapıldığına dikkat ediniz. `DATEcmp`, ilk tarih ikinciden büyükse pozitif bir değere, küçükse negatif bir değere ve eğer iki tarih birbirine eşitse sıfır değerine geri dönmektedir. (Tıpkı `strcmp`'de olduğu gibi.)

DOS'ta zaman formatı da 16 bit içerisinde ifade edilir.

Saat için 5 bit (0..23) dakika için 6 bit (0..59) ve saniye için 5 bit (0.. 59) ayrıldığına dikkat ediniz. 5 bit ile yazılabilecek işaretsiz sayı sınırı 0..31'dir. Ancak DOS yalnızca çift saniyeleri tutar. Yani saniye değeri olarak saniye alanında yazılan sayının iki katı hesaplanmalıdır. Örneğin bu alanın içerisinde 12 varsa gerçek saniye 12 * 2 = 24'tür. Aşağıda zaman bilgisinin bit alanı biçiminde ifadesini görüyorsunuz:

```
struct TIME {
    unsigned second: 5;          /* 0...59 */
    unsigned minute: 6;          /* 0...59 */
    unsigned hour: 5;            /* 0...23 */
};
```

Zaman bit alanı için aşağıdaki arabirim fonksiyonlarını da siz yazınız:

```
void setTIME (struct TIME *time, char *strtime);
char *getTIME(struct TIME *time, char *strtime);
int TIMEcmp(struct TIME *time1, struct TIME *time2);
```

SORAMADIKLARINIZ...

S1) Bit alanları tanımlayarak yapılan işlemler bit operatörleriyle de yapılamaz mı? Bit alanlarına neden gereksinim duyulmuştur?

C1) Bit alanları doğal yapılar değildir. Yani programcı bit alanlarıyla bitler düzeyinde işlem yaptığında derleyici bitleri ayrıştırıp işlemleri gerçekleştirebilmek için yine bit operatörlerini kullanacaktır. Çünkü mikroişlemciler bit düzeyinde aritmetik işlem yapamazlar. Bu açıdan bakıldığında bit alanları yerine programcı da bit operatörleriyle derleyicinin yaptığı işlemleri gerçekleştirebilir. Ancak bit alanları soyutlama sağladığından okunabilirliği ve algılamayı kuvvetlendirmektedir. Bit alanlarıyla çalışmak programcının işini kolaylaştırır.

S2) Bir bit alanı elemanı en fazla kaç bit uzunluğunda olabilir?

C2) Bit alanlarına ilişkin elemanlar en fazla `int` türünün bit uzunluğu kadar olabilirler. Örneğin 80X86 tabanlı 16 bit işletim sistemlerinde `int` türünün uzunluğu 16 bit olduğu için bir bit elemanı da en fazla 16 bit uzunlukta olabilir.

TÜR TANIMLAMALARI VE SAYIMLAMA SABİTLERİ

28

C'nin standart veri türlerine, karmaşık yapılara ve birliklere eskisinin yerini tutabilen başka isimler de verilebilir. Standart veri türlerine ve karmaşık yapılara başka isimler vermek okunabilirliği artırdığı gibi ifadeleri de kısaltmaktadır. Bu bölüm tür tanımlamaları ve sayımlama sabitleri ayrıntılarıyla ele alınmıştır.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Tür tanımlama işleminin genel biçimi nasıldır?

2) Dizi, gösterici, yapı ve birliklere ilişkin tür tanımlamaları nasıl yapılmaktadır?

3) Tür tanımlamalarına neden ihtiyaç duyulmaktadır?

4) Sayımlama sabitleri nasıl yaratılır?

5) `#define` önişlemci komutuyla yaratılan sembolik sabitlerle `enum` ile yaratılan sayımlama sabitleri arasındaki benzerlikler ve farklılıklar nelerdir?

28.1 TÜR TANIMLAMA İŞLEMİ

Tür tanımlama işleminin genel biçimi şöyledir:

```
typedef <tür ismi>  <tanımlanan yeni tür ismi>;
```

`typedef` tür ismi türetmek için kullanılan bir anahtar sözcüktür. `<tür ismi>` C'nin standart türlerine ilişkin bir anahtar sözcük olabileceği gibi yapı, birlik ve sayımlama sabitlerine ilişkin türlerin isimleri de olabilir.

`<tanımlanan yeni tür ismi>` isimlendirme kuralına uygun herhangi bir isimdir.

Örneğin:

```
typedef unsigned int WORD;
```

işlemiyle `unsigned int` türünün `WORD` isminde bir eşdeğeri tanımlanmıştır. Artık `unsigned int` ile `WORD` aynı anlamlarda birbirleri yerine kullanılabilir. Örneğin bildirim işlemi:

```
int a, b;
```

biçiminde yapılabileceği gibi,

```
WORD a, b;
```

biçiminde de yapılabilir. Benzer biçimde:

```
typedef unsigned char BYTE;
typedef unsigned long int DWORD;
```

tür tanımlamalarından sonra:

```
BYTE a;
DWORD b;
```

biçiminde bildirimler yapılabilir. Bir türün **typedef** ile tanımlanması ile eski tür isminin de kaybolmadığına dikkat ediniz. Ayrıca tanımlanmış bir türden hareketle yeni tür tanımlamaları da yapılabilir. Örneğin:

```
typedef unsigned int WORD;
typedef WORD UINT;
```

gibi. Bu durumda hem **unsigned int** hem **WORD** hem de **UINT** derleyici için aynı anlamı ifade eder. Tanımlanmış türler yalnızca bildirim işlemlerinde değil orijinali gibi geçerli her ifadede kullanılabilirler. Örneğin tanımlanmış tür **sizeof** ve **tür dönüştürme** operatörleriyle kullanılabilir.

```
p = (DWORD *) malloc(sizeof(DWORD)*10);
...
```

typedef ile yaratılan yeni tür isminin faaliyet alanı **typedef** tanımlamasının yapıldığı yere göre blok ya da dosya faaliyet alanı kuralına uyar. **typedef** tanımlaması blok başında yapılmışsa yeni yaratılan tür ismi yalnızca o blok içerisinde geçerlidir. Tanımlama global olarak yapılmışsa yaratılan isim tüm dosya içerisinde her yerde kullanılabilir. Uygulamada **typedef** tanımlamaları geenellikle programın tepesinde ya da bir başlık dosyasının içerisinde global olarak yapılmaktadır.

28.2 DİZİ VE GÖSTERİCİLERE İLİŞKİN TÜR TANIMLAMALARI

Dizi ve göstericilerin bulunduğu karmaşık veri yapıları **typedef** ile tanımlanarak yazım ve algılayış kolaylığı sağlanır. Karmaşık ifadelerin **typedef** ile tanımlanması okunabilirliği artırmaktadır.

```
typedef char * CHPTR;
```

ile CHPTR karakter türünden bir gösterici olarak tanımlanmıştır.

Artık:


```
CHPTR x;
```

ile

```
char *x;
```

bildirimleri aynı anlamdadır.

```
typedef char DIZI[20];
```

ile **DIZI**, 20 elemanlı bir karakter dizisi olarak tanımlanmıştır.

```
DIZI a;
```

ile

```
char a[20];
```

aynı anlamdadır. C'ye yeni başlayanlar sezgisel olarak dizi ifadelerine ilişkin tür tanımlamalarının aşağıdaki gibi olması gerektiğini düşünürler:

```
typedef char[20] DIZI;          /* Hatalı!..*/
```

Böyle bir yazım biçimi geçerli değildir. Gösterici dizilerine ve fonksiyon göstericilerine ilişkin tür tanımlamalarına da sık rastlanmaktadır:

```
typedef char * PTRARY[20] ;
```

Burada **PTRARY**, 20 elemanlı karakter türünden bir gösterici dizisidir.

```
PTRARY gdizi;
```

ile

```
char *gdizi[20];
```

aynı anlamdadır. Benzer biçimde örneğin:

```
typedef  int (*FPTR) (void);
```

ile **FPTR** geri dönüş değeri int parametresi void olan fonksiyonların adreslerini tutabilecek bir fonksiyon göstericisi türü olarak tanımlanmıştır.

```
FPTR p;
```

ile

```
int (*p) (void);
```

aynı anlamdadır.

28.3 YAPI, BİRLİK VE BİT ALANLARINA İLİŞKİN TÜR TANIMLAMALARI

Yapı, birlik ve bit alanlarına ilişkin tür tanımlamalarına çok sık rastlanır. Örneğin:

```
struct DENEME {
    int a;
    long b;
    char *ptr;
};
...
typedef struct DENEME DEN;
```

ile DEN, `struct DENEME` türü olarak tanımlanmıştır.

Dolayısıyla:

```
DEN *p;
```

Tür ismi

typedef struct DENEME DEN;

Yeni tanımlanan tür ismi

```
p = (DEN *) malloc(sizeof(DEN));
...
```

ile

```
struct DENEME *p;
p = (struct DENEME *) malloc(sizeof(struct DENEME));
...
```

aynı anlamdadır.

Yapı, birlik ve bit alanlarına ilişkin tür tanımlama işlemleri bildirim işlemleriyle birlikte yapılabilir.

Örneğin:

```
typedef struct DENEME {
    int a;
    long b;
    char *ptr;
} DEN;
```

ile hem `struct DENEME` bildirilmiş olur hem de `DEN` isimli bir tür tanımlanmış olur. Genellikle bu yazım biçimi tercih edilmektedir. Bu durumda aşağıdaki bildirimlerin her ikisi de geçerlidir:

```
struct DENEME x;
```

ya da

```
DEN x;
```

Programcıların çoğu yapı isimleriyle tür isimleri için farklı isimler bulmak yerine birkaç karakter kullanarak aralarında ilişki kurarlar. Çok kullanılan yöntemlerden biri, yapı isminin başına bir alt tire konularak tür isminden ayrılmasıdır:

```
typedef struct _DENEME {
    int a;
    long b;
    char *ptr;
} DENEME;
```

Aynı şeyin `windows.h` içerisinde `tag` sözcüğü eklenerek yapıldığını görüyoruz:

```
typedef struct tagPOINT {
    int x;
    int y;
} POINT;
```

`typedef` anahtar sözcüğünden sonra yapı, birlik ya da bit alanı ismi hiç yazılmayabilir. Ancak bu durumda değişken bildiriminin yeni tanımlanan tür ismiyle yapılması zorunluluk haline gelir.

```
typedef struct {
    int a;
    long b;
    char *ptr;
} DEN;

DEN x;
...
```

Başındaki `typedef` anahtar sözcüğü olmasaydı `DEN` ne anlama gelirdi, yorumlayınız?

28.4 TÜR TANIMLAMALARINA NEDEN İHTİYAÇ DUYULUR?

Tür belirten karmaşık ifadelere yalın isimler vererek ve program içerisinde bu isimleri kullanmak soyutlamayı artırarak okunabilirliği kuvvetlendirmektedir. Programı inceleyen kişi karmaşık operatörler yerine onu temsil eden yalın bir isimle karşılaşır. Tür tanımlamalarına taşınabilirliliği kuvvetlendirmek için de ihtiyaç duyulmaktadır. Tür tanımlamaları sayesinde kullandığınız fonksiyonlara

ilişkin veri yapıları değişse bile kaynak programın değiştirilmesine gerek kalmaz. Örneğin, kullandığınız kütüphanede birtakım fonksiyonların geri dönüş değerleri **unsigned int** olsun. Daha sonraki uyarlamalarında da bunun **unsigned long** yapıldığını düşünelim. Eğer programcı bu fonksiyonlara ilişkin kodlarda tür tanımlaması kullanmışsa daha önce yazdığı kaynak kodları değiştirmesine gerek kalmaz; yalnızca tür tanımlamasını değiştirmek yeterlidir. Örneğin:

```
typedef unsigned int HANDLE;
...
HANDLE hnd;
hnd = GetHandle();
```

Burada **GetHandle** fonksiyonunun geri dönüş değerinin türü sonraki uyarlamalarda değişerek **unsigned long** yapılmış olsun. Yalnızca tür tanımlamasının değiştirilmesi yeterlidir.

```
typedef unsigned long HANDLE;
```

28.5 SAYIMLAMA SABİTLERİ (Enumeration Constants)

15. Bölümde **#define** önişlemci komutuyla nasıl sembolik sabit tanımlandığını görmüştük. Sayımlama sabitleri de sembolik sabit yaratmak amacıyla kullanılmaktadır. Sembolik sabitlerin yanı sıra sayımlama sabitleri tıpkı yapılar ve birlikler gibi bir tür de belirtirler.

Sayımlama türünün ve sayımlama sabitlerinin yaratılması aşağıdaki gibi yapılır:

```
enum [sayımlama türünün ismi] {S1, S2, S3, S4, ...};
```

enum bir anahtar sözcüktür. Derleyici küme parantezleri arasındaki sembolik sabitlere sıfırdan başlayarak sırasıyla artan sırada bir tamsayı karşılık getirir. Örneğin:

```
enum BOOL {FALSE, TRUE };
```

Bildirimiyle **FALSE** sembolik sabiti **0**, **TRUE** sembolik sabiti **1** değerini alır.

```
enum DAYS {
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
};
```

Bildirimiyle **Sunday = 0, Monday = 1, Tuesday = 2, Wednesday = 3, ...** biçiminde sembolik sabitlere sıfırdan başlayarak ardışıl değerler verilir. Artık program içerisinde bu sembolik sabitleri kullanabiliriz. Sayımlama sabitlerine bildirim sırasında = ile değer verilirse sonrakiler ardışıl olarak o sayıyı izlerler. Örneğin:

```
enum DAYS {
    Sunday = 1,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday
};
```

Bu durumda `Sunday = 1, Monday = 2, Tuesday = 3, ...` sembolik sabitler ilki bir olmak üzere ardışıl değerleri alırlar. enum ile yaratılan sembolik sabitlere = operatörü ile int sınırları içerisinde pozitif ya da negatif istenilen sayılar atanabilir. Örneğin:

```
enum Sample {
    M1 = -4,
    M2,
    M3 = 10
    M4,
    M5 = 0
    M6
};
```

Burada `M1 = -4, M2 = -3 M3 = 10, M4 = 11, M5 = 0, M6 = 1` değerlerini alır. Bir sembolik sabite = ile değer atandığında onu izleyen sayımlama sabitlerinin bir sonraki atamaya kadar ardışıl bir biçimde gittiğine dikkat ediniz.

`enum` bildirimi aynı zamanda bir tür de belirtmektedir. Örneğin tıpkı yapılarda ve birliklerde olduğu gibi bu türden bir nesne de tanımlayabilirsiniz:

```
enum DAYS day;
enum BOOL result;
enum Sample smp;
...
```

gibi...

`enum` türünden bir nesne için derleyici ilgili sistemde int türünün uzunluğu kadar yer ayırır. Derleyici için enum türü bir nesne ile int türü bir nesne arasında fark yoktur.

Aşağıdaki program parçasını inceleyiniz:

```
enum BOOL {FALSE, TRUE };

BOOL isbig(void);

...
    {
        BOOL result;

        result = isbig();
        if (result == FALSE);
            end_prog();
        ...
    }
...
```

result değişkeni enum BOOL türündendir. Prototipini gördüğünüz isbig fonksiyonun da geri dönüş değeri BOOL türündendir. Normal olarak enum değişkenlerine enum sabitleri atanırsa da genel olarak böyle bir zorunluluk yoktur.

Sembolik sabit yaratmak amacıyla kullanıdğımız #define ile enum işlemleri arasında işlevsel bir fark yoktur. enum sabitleri de nesne değildir. Örneğin:

```
enum MEYVE {ELMA, ARMUT, KAYISI, KARPUZ, KAVUN};
...
ARMUT = 10;              /* Geçersizdir */
```

Yukarıdaki atama geçersizdir. Derleyici ARMUT karakterlerini gördüğü yere 1 yerleştirecektir. #define önişlemciye ilişkin olduğu halde enum sabitlerini ele alarak işleme sokmak derleme modülüne ilişkindir. Yani #define ve enum farklı aşamalarda ele alınırlar. enum sabitleri çok sayıda ve ardışıl tanımlamalar için tercih edilmektedir. Ayrıca bir tür ismi olarak kullanılması okunabilirliği de artırmaktadır. Örneğin:

```
BOOL isbig(void);
```

gibi bir prototipe bakan programcı isbig fonksiyonunun yalnızca Doğru ya da Yanlış değer ürettiğini, böylelikle bir test amacıyla yazıldığını anlar. Yine derleyicilerin standart başlık dosyalarında pekçok enum türünün ve sembolik sabitlerin tanımlandığını söyleyelim. Örnek olarak elinizdeki derleyiciye ilişkin GRAPHICS.H başlık dosyasını inceleyebilirsiniz. Örneğin aşağıdaki enum tanımlaması 80X86 sistemlerinde çalışan Borland derleyicilerinin GRAHICS.H dosyasından alınmıştır.

```
enum COLORS {
     BLACK,              /* dark colors */
     BLUE,
     GREEN,
     CYAN,
     RED,
     MAGENTA,
     BROWN,
     DARKGRAY,           /* light colors */
     LIGHTGRAY,
     LIGHTBLUE,
     LIGHTGREEN,
     LIGHTCYAN,
     LIGHTRED,
     LIGHTMAGENTA,
     YELLOW,
     WHITE
};
```

SORAMADIKLARINIZ...

S1) Anahtar sözcükler **typedef** ile yeniden isimlendirilebilir mi? Örneğin:

```
typedef  far  FAR;
```

gibi bir tanımlama geçerli midir?

C1) `typedef` ile yalnızca tür isimlerine yeniden isim verilebilir. Yukarıdaki örnekte `far` tek başına bir tür ismi olmadığından tanımlama da geçerli değildir. Bu durumlarda `#define` önişlemci komutu kullanılmalıdır.

```
#define FAR far
```

gibi.

S2) `enum` türlerine de `typedef` ile yeni isimler verilebilir mi?

C2) `enum` türü tıpkı yapılar ve birlikler gibidir. `typedef` ile `enum` türleri de yeniden isimlendirilebilir. Örneğin;

```
typedef enum  _BOOL {FALSE, TRUE } BOOL;
```

ile `enum _BOOL` yerine `BOOL` ismi verilmiştir. Artık ,

```
enum _BOOL x;
```

ile

```
BOOL x;
```

aynı anlamdadır.

S3) `enum` türü bir nesne ile `int` türü bir nesne arasında derleyiciler için bir fark yoksa `enum` türünden nesne kullanmaya niçin gerek duyulmaktadır?

C3) `enum` türünden nesneleri kullanmak okunabilirliği artırır. Normal olarak `enum` nesnelerine `enum` sabitleri atanacağından program koduna bakan birisi nesnenin ne amaçla kullanılacağını da hemen anlayacaktır.

YAPI, BİRLİK VE BİT ALANLARIYLA İLGİLİ KARMAŞIK TANIMLAMALAR

29

Yapı, birlik ve bit alanlarının kullanıldığı ifadeler bazen çok karmaşık bir hale gelebilirler. Fakat böyle karmaşık durumların da çözümlemesini yapabilmek gerekir. Bu bölüm böylesi karmaşık tanımlamalara ayrılmıştır.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir.

1) Yapı, birlik ve bit alanlarına ilişkin diziler nasıl tanılmanır?

2) Yapı, birlik ve bit alanlarına ilişkin dizilerin bellekteki organizasyonları nasıldır?

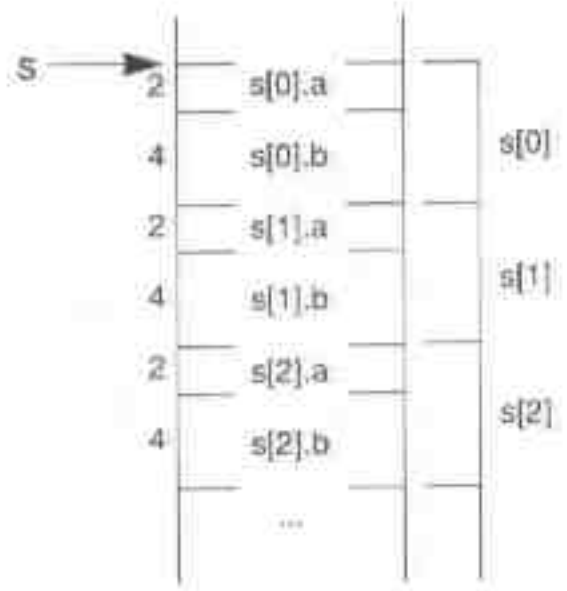
3) Yapı, birlik ve bit alanı elemanı olarak göstericilerin yarattığı karmaşık durumlar nasıl oluşmaktadır?

29.1 YAPI DİZİLERİ

Bir dizi daha önce bildirilmiş bir yapı türünden olabilir. Bu tür dizilerin bildirimlerinin diğer dizi bildirimlerinden biçimce bir farkı yoktur. Örneğin:

```
struct SAMPLE {
    int a;
    long b;
};
...
struct SAMPLE s[10];
```

Yukarıdaki tanımlamayı gören derleyici dizi için bellekte `sizeof(s)` yani `10 * (sizeof(int) + sizeof(long))` kadar sürekli yer ayıracaktır. Dizinin elemanlarına gösterici operatörleriyle eriştikten sonra, yapı elemanlarına da nokta operatörüyle erişebiliriz.



[n] ile . operatörünün soldan sağa eşit öncelikli olduğunu anımsayınız. Dizi isimleri bellekte dizilerin başlangıç adreslerini gösterdiğine göre s de bu yapı dizisinin başlangıç adresini göstermektedir. s adresi `struct SAMPLE` türündendir. Bu türden bir göstericiye doğal olarak atanabilir.

```
struct SAMPLE s[10];
struct SAMPLE *p;
...
p = s;  /* s ile p göstericisinin türü aynı */
```

Bir yapı dizisine ilişkin yapının elemanı başka bir dizi de olabilir. Örneğin:

```
struct DENEME {
    char name[10];
    int no;
};
...
struct DENEME s[10];
```

tanımlamasından sonra:

```
s [n] . name[k]
```

ifadesi yapı dizisinin `n.` elemanına ilişkin olan dizinin `k.` elemanını belirtir. İfade ile belirtilen nesne `char` türündendir. Benzer biçimde:

```
& s [n] . name[k]
```

ifadesi de ilgili karakter nesnesinin adresini gösterir.

29.2 ELEMANI KENDİ TÜRÜNDEN BİR YAPIYI GÖSTEREN YAPILAR

Bir yapının elemanı aynı türden bir yapı olamaz. Örneğin:

```
struct SAMPLE {
    int x;
    long y;
    struct SAMPLE z;
};
```

Çünkü derleyici yukarıdaki yapı için ne kadar yer ayıracağını bilemez. Yani **sizeof (struct SAMPLE)** belli değildir. Fakat bir yapının elemanı kendi türünden bir yapı göstericisi olabilir.

```
struct SAMPLE {
    int z;
    long y;
    struct SAMPLE *z;
};
```

Çünkü bu durumda yapının uzunluğu bellidir. (Örneğimiz için DOS altında ve yakın modellerde 2+4+2 = 8, Unix ve Windws 95'te 4+4+4 = 12). Kendi türünden yapıyı gösteren elemanların bulunduğu yapılara özellikle bağlı liste uygulamalarında çok rastlanır. Bağlı listeler bir önceki elemanın bir sonrakinin yerini gösterdiği listelerdir. Fiziksel sıranın dışında mantıksal sıralamanın gerektiği durumlarda kullanılırlar. Bir bağlı listeye ilişkin örnek bir yapı aşağıdaki gibi olabilir:

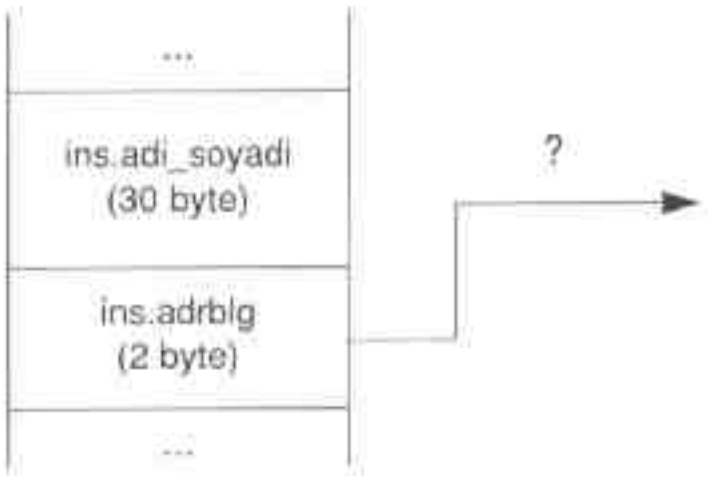
```
struct LIST {
    struct LIST *next;
    ...
};
```

29.3 YAPI ELEMANI OLARAK YAPI GÖSTERİCİLERİ

Aşağıdaki yapı bildirimini inceleyiniz:

```
typedef struct _ADRBILGI {
    char *adres;
    char telno[11];
} ADRBILGI;

typedef struct _INSAN {
    char adi_soyadi [30];
    ADRBILGI *adrblg;
} INSAN;
...
INSAN ins;
```

`INSAN` yapısı içerisinde `ADRBILGI` isminde bir yapı göstericisi bildirilmiştir. Programcı ins değişkenini kullanarak `ADRBILGI` yapısına nasıl erişir?

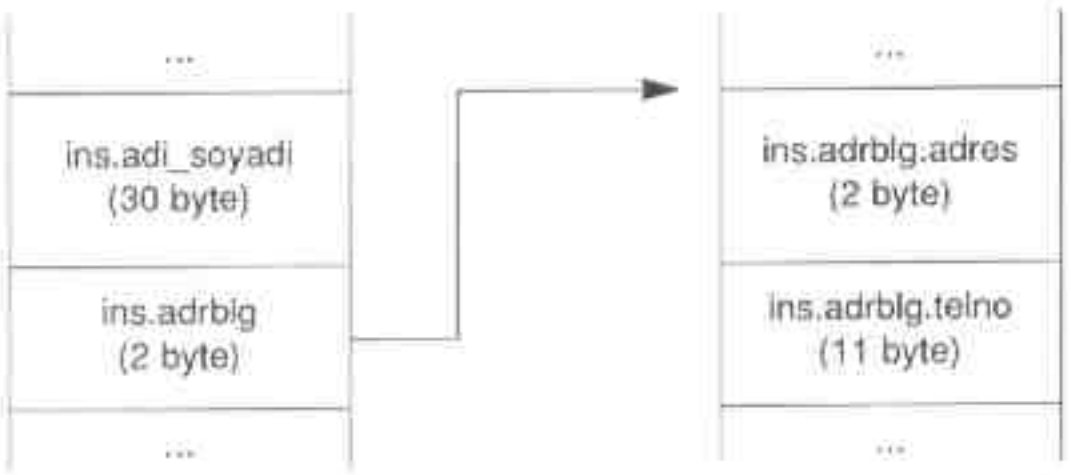


ins yerel bir değişkense **ins.adrblg** içerisinde rastgle bir değer vardır. Öncelikle bu göstericinin tahsis edilmiş bir **ADRBILGI** yapısını göstermesi gerekir. Bunun için yerel bir yapı değişkeni tanımlayabilirsiniz:

```
ADRBILGI adres;
...
ins.adrblg = &adres;
```

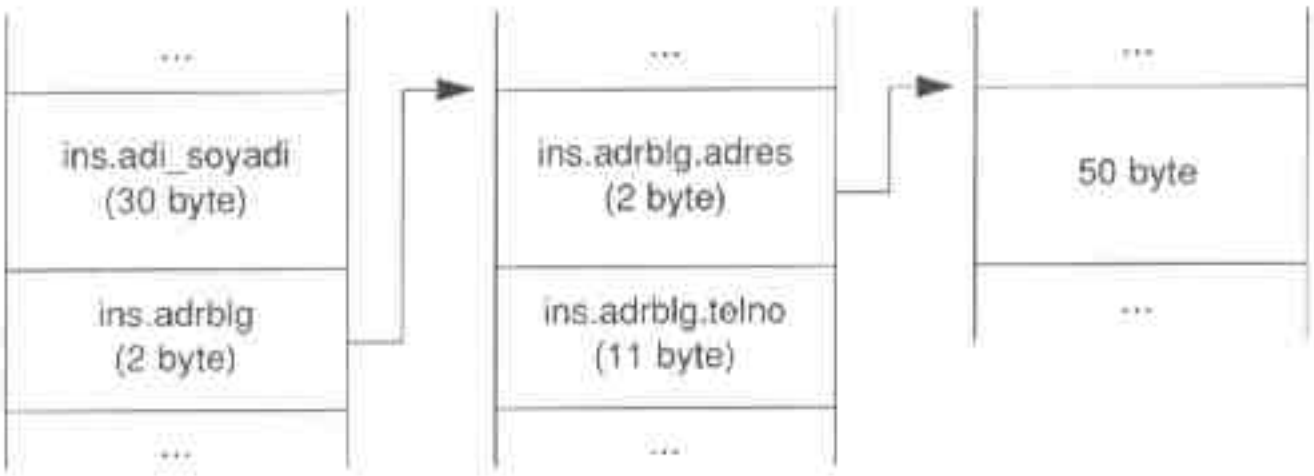
Ya da dinamik bellek fonksiyonlarıyla tahsisat yapabilirsiniz:

```
ins.adrblg = (ADRBILGI *) malloc(sizeof(ADRBILGI));
if (ins.adrblg == NULL) {
    printf("Yetersiz bellek!...\n");
    exit(1);
}
...
```



Şimdi **ins.adrblg** güvenli bir bölgeyi gösteriyor. Ancak bu yapının adres elemanı da bir göstericidir. Bu gösterici için de yer tahsis etmemiz gerekmez mi?

```
ins.adrblg.adres = (char *) malloc(50);
...
```



Bu tahsisatlardan sonra ancak `ins` ile diğer yapılara erişmemizin bir anlamı olabilir. Aşağıdaki ifadeyi yorumlayınız:

```
ins . adrblg -> adres
```

. ile -> operatörünün soldan sağa eşit öncelikli olduğunu anımsayın. `ins.adrblg` bir yapı göstericisidir. -> operatörü ile bu yapının adres elemanına erişilmiştir. Peki aşağıdaki ifadeye ne dersiniz?

```
ins . adrblg -> adres[10]
```

3 operatörde soldan sağa eşit önceliklidir. Bu ifade **ins** yapı değişkeninin elemanı olan adrblg yapı göstericisinin gösterdiği **adres**'in 10. indisli elemanını anlatır. Şimdi başa dönelim. **ins** bir gösterici olsaydı, ne olurdu?

```
INSAN *ins;
...
```

Bu durumda tahsisata `ins` ile başlanması gerekirdi:

```
ins = (INSAN *) malloc(sizeof(INSAN));
...
```

Bundan sonra diğer tahsisatların da yapılması gerekir:

```
ins -> adrblg  = (ADRBILGI *) malloc(sizeof(ADRBILGI));
...

ins -> adrblg -> adres = (char *) malloc(50);
...
```

ins yapı göstericisinin elemanı olan **adrblg** yapı göstericisinin gösterdiği **adres**'in 10. indisli elemanı şöyle anlatılabilir:

```
ins -> adrblg -> adres[10]
```

Bir yapının elemanı bir fonksiyon göstericisi de olabilir. Örneğin:

```
typedef struct _FINFO {
    void (*f) (void);
    ...
} FINFO;
...
FINFO info;
```

tanımlamalarıyla

`info.f` içerisine geri dönüş değeri ve parametresi `void` olan bir fonksiyonun adresi konulabilir. Bundan sonra fonksiyon:

```
info.f()
```

ifadesiyle çağrılabilir.

ÖNİŞLEMCİ KOMUTLARI

30

15. Bölümde önişlemci kavramına bir giriş yaparak `#include` komutunu açıklamış ve `#define` önişlemci komutunun yalın kullanımını ele almıştık. Bu bölümde ise `#define` ile oluşturulan karmaşık makrolar ve diğer önişlemci komutlarını ele alacağız.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Makro yazımında karşılaşılan hatalar nelerdir; ve ne biçimde önlem alınmaktadır?

2) Koşullu derleme komutları ne amaçla kullanılır?

3) Önceden tanımlanmış sembolik sabitler hangileridir ve ne amaçlarla kullanılırlar?

4) `#if`, `#ifdef`, `#ifndef`, `#undef`, `#error` ve `#pragma` komutlarının işlevleri nelerdir?

30.1 MAKROLAR

`#define` komutu parametre alabilir. `#define` komutunun parametrelerle kullanımına makro diyoruz. Önişlemci `define` anahtar sözcüğünden sonraki ilk boşluksuz karakter dizisi içerisinde bir parantezle karşılaşırsa parantez içserisindeki ifadeleri parametre olarak kabul eder. Örneğin:

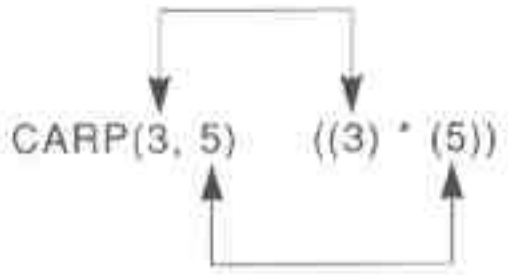
Burada `a` ve `b` `CARP` makrosunun parametreleridir. Önişlemci makroyu parametreleri uygun yerlere yerleştirerek açar. Örneğin:

```
#define CARP(a, b)     ((a) * (b))
```

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

```
#define CARP(a, b)  ((a) * (b))
void main(void)
{
    int x;

    x = CARP(3, 5);
    printf("x = %d\n", x);
}
```

yukarıdaki gibi bir çağırma durumunda önişlemci `CARP(3, 5)` yerine `((3) * (5))` yerleştirecektir.



Böylece derleme modülüne gelindiğinde ifade `x = ((3) * (5))` biçimine dönüştürülmüş olur. Şimdi de iki sayının büyüğünü bulan aşağıdaki makroyu inceleyiniz:

```
#define MAX(a, b)  ((a) > (b) ? (a) : (b))
```

Kaynak kod içerisinden aşağıdaki gibi çağrılmış olsun:

```
x = MAX(10, 20);
```

Önişlemci kodu aşağıdaki gibi açacaktır:

```
x = ((10) > (20) ? (10) : (20));
```

Böylece x'e 20 atanır.

Makro yazımında parantez eksikliğinden çıkan hatalara çok sık rastlanır. Örneğin `CARP` isimli makro aşağıdaki gibi yazılmış olsaydı:

```
#define CARP(a, b)    a * b
```

o zaman `z = CARP(x - a, y - b)` gibi bir çağırma sonrasında önişlemci kodu `x - a * y - b` biçiminde açacağından çarpmanın önceliğinden doğan bir hata oluşurdu. Bu tür hataları önlemek için:

1) Makro parametreleri paranteze alınmalıdır.

2) Makro en dıştan paranteze alınmalıdır.

Makro parametrelerinin paranteze alınması çarpma ve bölme gibi operatörlerin oluşturacağı öncelik sorununu kaldırır. Makronun en dıştan paranteze alınması ise çarpma ve bölmeden daha yüksek öncelikli operatörlerin etkisinden kurtul-

mak için gereklidir. **Çünkü bir makro fonksiyon gibi işlem görmelidir.** Örneğin CARP makrosu en dıştan parantezlere alınmasaydı;

```
#define CARP(a, b)    (a) * (b)
```

!CARP(1, 0) gibi bir ifadeden **1** yerine **0** elde edilirdi. Çünkü:

```
!(1) * (0) = 0
```

! operatörünün önceliği * operatöründen daha yüksektir.

Bazı makroların ++ ve -- operatörleriyle çağrılması da problemlere yol açabilir. Örneğin:

```
#define KARE(x) ((x) * (x))
```

makrosu,

```
int a = 10;
...
b = KARE(++a);
```

diye çağrıldığında önişlemci kodu:

b = ((++a) * (++a)) diye açacaktır. Bu da şüpheli bir koddur. Açılan ifade her sistemde 11'in karesi olmayabilir.

Makroları noktalı virgül ile sonlandırmamak gerekir. Örneğin:

```
#define KARE (x)      ((x) * (x)) ;
```

yukarıdaki makro aşağıdaki gibi çağrılmış olsun:

```
y = KARE(10);
```

Önişlemci kodu aşağıdaki gibi açar:

```
y = ((10) * (10));;
```

Makroda zaten noktalı virgül olduğundan çözülmüş kodda iki tane noktalı virgül bulunur. Gerçi solunda hiçbir ifade olmaksızın bulunan noktalı virgüller C derleyicileri tarafından boş deyim (null statement) olarak ele alındığından hataya yol açmazlar. Ancak yine de makrodaki noktalı virgül gereksizdir.

String ifadelerinin **#define** komutu tarafından değiştirilmediğini söylemiştik. Ancak makro parametresinin önüne # karakteri konursa önişlemci bu parametreyi string ifadesi biçiminde açar. Derleyici de yan yana yazılan stringleri birleştirecektir. Örneğin:

```
#define YAZ(x)  printf(#x "=%d\n", x)
```

gibi bir makro;

```
int n = 20;
...
YAZ(n);
```

biçiminde çağrılırsa önişlemci makroyu;

```
printf("n" "=%d\n", x);
```

biçiminde açar; derleyici de yanyana stringleri birleştireceğinden ifade:

```
printf("n=%d\n", x);
```

biçiminde yazılmış gibi olur.

Makrolar bir satır sürmek zorunda değildir. **Ters bölü** (\) karakteri ile sonraki satırdan devam edilebilir. Örneğin:

```
#define mesaf(no) if ((no) > 10) printf("Anlaşılamayan hata!..\n"); \
else printf ("Hata kodu:%d\n", (no));
```

gibi.

30.1.1 Makrolar Nerede Tanımlanmalıdır?

Sembolik sabitlerde olduğu gibi makrolar da kaynak kodun herhangi bir yerinde tanımlanabilir. Tanımlandıkları yerden dosya sonuna kadar olan bölge içerisinde etkili olurlar. Ancak uygulamada makrolar ya programın tepesine ya da başlık dosyalarının içerisine yerleştirilirler. C'nin standart başlık dosyalarında da birtakım makroların tanımlandığını belirtmiştik. Örneğin `isupper`, `islower`, `isalpha` gibi karakter test fonksiyonları `CTYPE.H` içerisinde bildirilmiş olan birer makrodur. Yine standart C derleyicilerinin çoğunda `getchar` ve `putchar` STDIO.H içerisinde bildirilmiş makrolardır.

30.1.2 Karşılaştırma: Makrolar ve Fonksiyonlar

Bir makro fonksiyon gibi kullanılsa da makro çağırmakla fonksiyon çağırmak arasında fark vardır. Makro çağrıldığında gerçekte programa önişlemci tarafından bir kod eklenir. Dolayısıyla örneğin bir programda aynı makro 10 kez çağrılıyorsa aynı kod 10 kez programa eklenerek programı büyütecektir. Oysa fonksiyon 10 kez çağrılmış bile olsa fonksiyon kodundan yalnızca bir tane çalışabilen program içerisine yerleştirilir. Fonksiyonların çağrılması durumunda yalnızca çağrılma bilgisi kod içerisine yazılmaktadır. Ancak fonksiyonların çağrılması işleminde göreli bir gecikme de söz konusudur. (Tabi bu gecikme süresi normal uygulamalar için önemsizdir). Sonuç olarak programcının kod büyüklüğü ile hız arasında bir seçim yapması gerekir. Çok küçük kodların makro biçiminde, diğerlerinin ise fonksiyon biçiminde yazılması belki de önerilecek en iyi yoldur. İki sayının toplamını, çarpımını bulan iki sayıyı karşılaştıran kodlar makro kullanımına iyi örneklerdir.

80X86 **Sembolik Makina Dili Programcısına Not**: Fonksiyon çağırma işlemindeki göreli yavaşlık (function call overhead) iki nedenden kaynaklanır. Birincisi fonksiyon çağırırken kullanılan **CALL** ve geri dönüşte kullanılan **RET** makina komutlarıdır. İkincisi (asıl önemlisi budur) parametre aktarımı ve stack düzenlemesi işlemleridir. Parametrelerin **stack** bölgesine kopyalanması ve "stack frame" düzenlenmesi işlemi belli sayıda makina komutları ile yapılmaktadır.

30.2 KOŞULLU DERLEME KOMUTLARI

Koşullu derleme komutları `#if`, `#ifdef`, `#ifndef` olmak üzere üç tanedir. Koşullu derleme komutlarıyla karşılaşan önişlemci koşulun sağlanması durumunda program bloğunu derleme işlemine dahil eder; koşul sağlanmıyorsa ilgili program bloğunu çıkartarak derleme işlemine sokmaz.

30.2.1 #if

Genel biçimi şöyledir:

```
#if  <sabit ifadesi>
...
#else
...
#endif
```

Sabit ifadesi sayısal olarak sıfır dışı bir değer ise `#if` doğru olarak değerlendirilerek `#else` anahtar sözcüğüne kadar olan kısım, eğer sabit ifadesinin sayısal değeri **0** ise `#if` yanlış olarak değerlendirilerek `#else` ile `#endif` arasındaki bölüm derleme işlemine dahil edilir. Örneğin:

```
#include <stdio.h>

void main(void)
{
    #if  1
    printf("Bu bölüm derleme işlemine sokulacak!..\n);
    #else
    printf("Bu bölüm derleme işlemine sokulmayacak!..\n");
    #endif
}
```

Önişlemci programı # içeren satırlardan arındırarak aşağıdaki biçimde derleme modülüne verir:

`stdio.h` dosyasına ilişkin önişlemci tarafından çözülmüş kod

```
void main(void)
{
    printf("Bu bölüm derleme işlemine sokulacak!..\n");
}
```

`#if` komutunun sağındaki sabit ifadesi aritmetik ve ilişkisel operatör içerebilir; ancak değişken içermemelidir. Örneğin:

```
#define MAX 100
...
#if  MAX > 50
....
#else
...
#endif
```

gibi. `#else`, `#if` merdivenlerine de sıklıkla rastlanır.

```
#define MAX 100
...
#if  MAX > 50
....
#else
    #if
    ...
    #endif
#endif
```

gibi.

```
#else
#if
```

yerine `#elif` de kullanılabilir. Ancak bu durumda `#elif` komutuna ilişkin `#endif` ile dışarıdaki `#if` komutuna ilişkin `#elif` ortak olmak zorundadır.

```
#define MAX 100
...
#if MAX > 50
....
#elif MAX > 30
    ...
#endif
#endif
```

Yukarıdaki yazım hatalıdır. Bir tane `#endif` olması gerekirdi. İfadenin doğru biçimi aşağıdaki gibidir:

```
#define MAX 100
...
#if  MAX > 50
....
#elif
...
#endif
```

Koşullu derleme işlemi kaynak kodun belli bölümlerini belli durumlarda derleme işlemine sokmak için kullanılır. Şüphesiz bu durum programcı tarafından da kodun eklenip silinmesiyle sağlanabilirdi. Ancak `#if` komutu buna daha kolay olanak sağlamaktadır.

30.2.2 #ifdef ve #ifndef

Belli bir sembolik sabitin tanımlandığı durumda `#else` anahtar sözcüğüne kadar olan kısım, tanımlanmadığı durumda `#else` ile `#endif` arasındaki kısım derleme işlemine dahil edilir. Genel biçimi aşağıdaki gibidir:

```
#ifdef  <sembolik sabit>
...
#else
...
#endif
```

Örneğin:

```
#include <stdio.h>

#define SAMPLE

void main(void)
{

#ifdef  SAMPLE
    printf("SAMPLE tanımlanmış!..\n");
#else
    printf("SAMPLE tanımlanmamış!..\n");
#endif
}
```

Burada SAMPLE `#define` ile tanımlandığı için `#ifdef` Doğru olarak ele alınır ve derleme işlemine ilk `printf` dahil edilir. `#ifdef` ile sorgulama yapılırken sembolik sabitin ne olarak tanımlandığının bir önemi yoktur.

`#ifdef` aynı başlık dosyasını kullanan ve birkaç modülden oluşan programlarda yaygın olarak kullanılır. Örneğin **A1.C**, **A2.C** ve **A3.C** modüllerinden oluşan bir projede **PROJE.H** isimli bir başlık dosyası ortak olarak kullanılıyor olsun. Bu durumda global değişkenlerin yalnızca bir modülde global diğerlerinde `extern` olarak kullanılması gerekmez mi? Bu aşağıdaki gibi sağlanabilir.

```
* PROJECT.H */

#ifdef GLBVAR
#define EXTRN
#else
#define EXTRN extern
```

```
#endif
/* Global Değişkenler */
EXTRN int x;
EXTRN int y;
...
```

Şimdi modüllerin birinde bu include dosyasının yukarısında GLBVAR tanımlanırsa önişlemci EXTRN sözcüğünü sileceğinden (onun yerine boşluk karakterleri koyacağından) o modül için global tanımlama yapılmış olur. Değilse EXTRN yerine extern anahtar sözcüğü önişlemci tarafından yerleştirilecektir.

#ifdef önceden tanımlanmış sembolik sabitlerle taşınabilirliği sağlamak amacıyla da sıklıkla kullanılmaktadır. Örneklerini ileride göreceksiniz.

#ifndef komutu #ifdef komutunun tam tersidir. Yani sembolik sabit tanımlanmamışsa #else kısmına kadarki bölüm, tanımlanmışsa #else ile #ifndef arasındaki kısım derleme işlemine dahil edilir. Örneğin:

```
#ifndef BLINK
#define BLINK   0x80
#endif
```

Burada BLINK sembolik sabiti ancak daha önce tanımlanmamışsa tanımlaması yapılmıştır. Daha önce tanımlanmışsa tanımlama işlemi yapılmamıştır.

30.2.3 defined (...) Önişlemci Operatörü

defined bir sembolik sabitin tanımlanıp tanımlanmadığını anlamak için kullanılan bir önişlemci operatörüdür. Parantez içerisindeki sembolik sabit tanımlanmışsa **1** değerini tanımlanmamışsa **0** değerini üretir. Örneğin

```
#if defined(MAX)
....
#endif
```

burada MAX tanımlanmışsa **defined** 1 değerini üretecektir. Bu değer de #if tarafından doğru olarak kabul edilecektir. Burada yapılmak istenenin #ifdef ile aşağıdaki gibi de yapılabilirdi:

```
#ifdef  MAX
...
#endif
```

30.3 ÖNCEDEN TANIMLANMIŞ SEMBOLİK SABİTLER

Bazı sembolik sabitlerin özel durumlarda derleyici tarafından içsel olarak tanımlandığı varsayılır. Bu özel durumlar test edilerek çeşitli seçenekler değerlendirilmektedir. Bu kısımda önceden tanımlanmış sembolik sabitlerin standart olanlarından bir kısmını inceleyeceğiz.

30.3.1 Kaynak Koda İlişkin Bilgi Veren Sembolik Sabitler

__LINE__: Önişlemci bu sabit yerine kaynak koddaki o anda bulunulan satır numarasını yerleştirir. #include dosyalarının içi bu işleme dahil edilmez. Aşağıdaki kodu inceleyiniz:

```
#include <stdio.h>                                  /* 1. satır */
                                                    /* 2. satır */
void main(void)                                     /* 3. satır */
{                                                   /* 4. satır */
    printf("Satır no : %d\n", __LINE__);            /* 5. satır */
}                                                   /* 6. satır */
```

Önişlemci __LINE__ yerine onun satır numarası olan 5 karakterini yazar. Böylece kod derleme aşamasına gelindiğinde aşağıdaki gibi olacaktır:

STDIO.H dosyasının açılmış biçimi

```
void main(void)
{
    printf("Satır no : %d\n", 5);
}
```

Ekranda 5 sayısını görürsünüz değil mi?

__FILE__ : Önişlemci bu sabit yerine "iki tırnak içerisinde kaynak dosyanın ismini" yazar. Aşağıdaki örnekte kaynak dosyanın ismi ekrana yazdırılıyor; string ifadelerinin karakteri gösteren birer adres olduğunu anımsayınız.

```
#include <stdio.h>

void main(void)
{
    printf("Dosya ismi: %s\n", __FILE__);
}
```

__DATE__ ve __TIME__ : Önişlemci bu sabitlerin yerine derlemenin yapıldığı tarih ve zamanı yazar. __DATE__ "Aaa gg yyyy", TIME ise "ss:dd:ss" biçiminde yazılmaktadır.

30.3.2 Taşınabilirliğe İlişkin Sembolik Sabitler

`__STDC__`: *C*'de kullanıdığımız kimi anahtar sözcükler tam anlamıyla standart değildir. Bu anahtar sözcükler sistemler arası farklılıklara karşılık verebilmek için kullanılmaktadır. Örneğin 80X86 sistemlerinde kullandığımız `far` ve `near` standart olarak her sistemde bulunmayan iki anahtar sözcüktür. Derleyiciniz eğer yalnızca standart *C*'nin anahtar sözcüklerini destekliyorsa `__STDC__` sembolik sabiti tanımlanmış varsayılır. İçerisinde standart olamayan anahtar sözcükler geçen programınızı bu sembolik sabiti kullanarak daha taşınabilir bir hale getirebilirsiniz.

```
#ifdef    __STDC__
#define   far
#endif
```

Yukarıdaki örnekte eğer yalnızca standart *C*'nin anahtar sözcüklerini kullanan bir derleyici ile çalışıyorsanız `__STDC__` tanımlanmış olacağından `far` anahtar sözcüğü silinecektir.

`__TINY__`, `__SMALL__`, `__MEDIUM__`, `__COMPACT__`, `__LARGE__`, `__HUGE__`: Borland derleyicilerinde aktif olarak kullanılan bellek modeline göre bu sembolik sabitlerden bir tanesi tanımlanmış varsayılır. Örneğin, o anda **LARGE** modelde çalışıyorsanız yalnızca `__LARGE__` sembolik sabiti tanımlanmış varsayılır.

`__MSDOS__`: **DOS** ile **UNIX** arasındaki taşınabilirlik problemlerini çözmek için kullanılır. Eğer **MSDOS** altındaki bir derleyicide çalışıyorsanız `__MSDOS__` tanımlanmış kabul edilmektedir.

```
#ifndef  __MSDOS__
#define far
#endif
```

Yukarıdaki örnekte eğer **MSDOS** ile çalışılmıyorsa `far` anahtar sözcüğü silinmektedir.

`__TURBOC__` : **Borland** firmasının **TurboC** uyarlamalarından bir tanesinde çalışıyorsanız bu sembolik sabit derleyiciniz tarafından tanımlanmış varsayılır. Sembolik sabitin tanımlanan değeri (test için önemi olamaz) uyarlamadan uyarlamaya değişmektedir.

`__BORLANDC__` : **Borland** firmasının **BorlandC** uyarlamalarından biri ile çalışıyorsanız bu sembolik sabit tanımlanmış varsayılır.

`__cplusplus` : *C++* derleyicileri ismiyle piyasaya çıkan derleyiciler standart *C*'yi de desteklerler. Bu durumda derleyicilerin hangi *C* uyarlamasında çalıştıkları bu

sembolik sabit yardımıyla öğrenilebilir. Eğer C++ ile çalışılıyorsa __pluplus sembolik sabiti tanımlanmış varsayılır. Örneğin:

```
#ifdef __cplusplus
    extern "C" {
        #include "maydata.h"
    }
#endif
```

yukarıda C++ uyarlamasıyla çalışıldığında "maydata.h" dosyasına **extern "C"** işlemi uygulanmıştır. (**extern "C"** işlemi standart C'de yoktur ve bunun hakkında bir bilgi sahibi olmanız gerekmiyor!..)

30.4 GENEL ÖNİŞLEMCİ KOMUTLARI

30.4.1 #undef

Bir sembolik sabitin ilki ile özdeş olmayan bir biçimde ikinci kez tanımlanması C önişlemcisi tarafından "uyarı" olarak değerlendirilir. Bu durumda ilk tanımlananın mı yoksa ikinci tanımlananın mı geçerli olacağı taşınabilir bir özellik değildir. (Derleyiciler genellikle ikinci tanımlanma noktasına kadar ilkinin ikinci tanımlanma noktasından sonra ise ikincinin geçerli kabul ederler.). Örneğin aşağıdaki gibi tanımalama işlemleri uyarı gerektirir.

```
#define MAX 100
...
#define MAX 200
```

Bir sembolik sabitin ilki ile özdeş olarak tanımlanmasında herhangi bir problemin ortaya çıkmayacağına dikkat ediniz. Bir sembolik sabit ikinci kez tanımlanmak isteniyorsa önce eski tanımlamayı kaldırmak gerekir. Bu işlem **#undef** ile yapılmaktadır. Örneğin:

```
#undef  MAX
#define MAX 200
```

Önce önce MAX tanımlaması kaldırılmış sonra **200** olarak yeniden tanımlanmıştır. **#undef** ile tanımlaması kaldırılmak istenen makro daha önce tanımlanmış olmasa bile bir probleme yol açmaz. Örneğin, yukarıdaki örnekte MAX tanımlanmamış olsaydı bile bir uyarı ya da hataya yol açmazdı.

30.4.2 #error

Önişlemci **#error** komutunu görünce yanındaki mesajı basarak derleme işlemine son verir. Örneğin:

```
#ifdef __TINY__
#error Bu program tiny modelde derlenmez!..
#endif
```

Eğer DOS altında ve **tiny** modelde çalışıyorsanız önişlemci işlemine **"Bu program tiny modelde derlenmez!.."** mesajıyla son verir. Mesajın iki tırnak içerisinde yazılmadığına dikkat ediniz.

30.4.3 #pragma <komut belirticisi>

Bu komut aldığı argümana bağlı olarak derleyiciden derleyiciye değişebilen işlevlere sahiptir. Her **#pragma** komutu her derleyicide olmayabilir. Ancak olmayan bir **#pragma** komutunu kullanmak hata ya da uyarıya yol açmaz. Burada yalnızca örnek olarak 80X86 sistemlerinde çalışan derleyicilere ilişkin **#pragma inline** komutunu açıklayacağız. Çalıştığınız derleyicilerin **#pragma** komutları için özel referans kitaplarına başvurabilirsiniz.

#pragma inline: Programın içerisinde sembolik makina dili komutlarının kullanılacağını belirtir. Sembolik makina dili komutları **80X86** sistemlerinde **asm** anahtar sözcüğüyle yazılmaktadır. Örneğin:

```
#include <stdio.h>

#pragma inline

void pos(int row, int col)
{
    asm mov     ah, 2
    asm mov     bh, 0
    asm mov     dh, row
    asm mov     dl, col
    asm int     10h
}

void main(void)
{
    pos(10, 10);
    getch();
    pos(0, 0);
    getch();
}
```

SORAMADIKLARINIZ...

S1) Kaynak koda bakarak (...) operatörleriyle çağrılmış bir kodun makro mu fonksiyon mu olduğunu anlayabilir miyiz?

C1) Yalnızca çağrılma noktasına bakarak anlayamayız. Fakat bölüm içerisinde de anlatıldığı gibi standart *C* fonksiyonlarının bazıları gerçekte birer makrodur. Hiç olmazsa bunların hangileri olduğunu *C* programcılarının bilmesi gerekir.

S2) Makrolar için de prototip yazmaya gerek var mıdır?

C2) Prototip çağrılma noktasına kadar bir fonksiyonun geri dönüş değerinin tespit edilmesi için kullanılır. Makrolarda böyle bir durum söz konusu değildir. Bir makro kodu mutlaka çağrılma noktasından önce tanımlanmalıdır.

3 DOSYA İŞLEMLERİ

C'de dosya işlemleri fonksiyonlarla yapılır. Dosyaların açılması, kapatılması, içerisine bilgilerin yazılması gibi işlemler için standart *C* fonksiyonları kullanılmaktadır. Yüksek seviyeli programlama dillerinde çalışan okuyucularımız dosyalama işlemleri bakımından *C*'yi zayıf bulabilirler. *C*'nin seviyesi gereği yüksek seviyeli dosya fonksiyonlarının standart olarak kütüphaneye dahil edilmesi uygun görülmemiştir. Bu nedenle veritabanlarının yönetimi için gerekli olan indeksleme, arama gibi işlemler için standart *C* fonksiyonlarını kullanarak yüksek seviyeli kütüphaneler oluşturmanız gerekebilir. Ayrıca yazılım firmaları tarafından bu amaçla oluşturulmuş kütüphaneler yoğun olarak kullanılmaktadır. Dosya konusuna girmeden önce komut satırı argümanlarını açıklamayı uygun gördük.

Bu bölümü okuduktan sonra aşağıdaki soruları yanıtlayabilmeniz gerekir:

1) Komut satırı argümanları kaç tanedir ve nasıl kullanılır?

2) Dosyanın açılması, kapatılması ne anlama gelmektedir?

3) Dosya göstericisi kavramını açıklayınız?

4) `fopen` fonksiyonu niçin kullanılmaktadır ve geri dönüş değeri nedir?

5) Dosya açış modları nelerdir?

6) Dosyadan karakter okuyan ve dosyaya karakter yazan *C* fonksiyonları nasıldır?

7) Bilgiler formatlı bir biçimde dosyaya nasıl yazılırlar ve dosyadan nasıl okunurlar?

8) Metin dosyalarıyla ikili dosyalar arasındaki ayrım nedir?

9) `fread` ve `fwrite` fonksiyonları nasıl çalışır?

31.1 KOMUT SATIRI ARGÜMANLARI

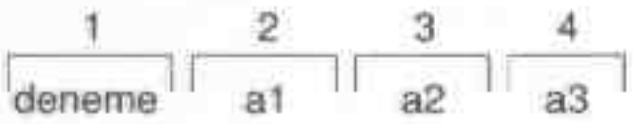
Şimdiye kadar `main` fonksiyonuna hiç parametre yazmadık. Oysa **main** fonksiyonu da isteğe bağlı olarak iki parametre alabilir.

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

`main` fonksiyonunun ilk parametresi komut satırı argümanlarının sayısıdır. Bu sayıya programın ismi de dahildir. Örneğin DOS ya da UNIX'te çalışıyorsanız ve programınızın ismi `deneme` ise komut satırında programınızı (örneğin DOS imlecinde):

```
deneme  a1 a2 a3
```

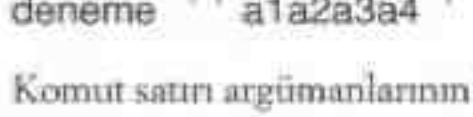
ile yazarak çalıştırdığınızda `argc = 4` olur.



Komut satırı argümanları boşluk karakterleriyle birbirlerinden ayrılmış olmalıdır. Yukarıdaki örnekte program:

```
deneme  a1a2a3
```

biçiminde çalıştırılsaydı, `argc = 2` olurdu.



Komut satırı argümanlarının ikincisi karakter türünden bir göstericiyi gösteren göstericidir. Yani `argv[0]`, `argv[1]`, `argv[2]`, ... birer karakter göstericisidir. Komut satırı argümanları NULL ile sonlandırılmış bir biçimde bu adreslerde bulunurlar. `argv[0]` sürücü ve dizin dahil olmak üzere (full path name) program ismini vermektedir. Diğer argümanlar sırasıyla `argv[1]`, `argv[2]`, `argv[3]`, ... adreslerindedir.

Komut satırı argümanları işletim sistemi tarafından komut satırından alınır ve derleyicinin ürettiği giriş kodu yardımıyla `main` fonksiyonuna parametre olarak kopyalanır.

80X86 Sembolik Makina Dili Programcısına Not: DOS işetim sisteminde komut satırı argümanları PSP içerisine **0x80** ofsetinden başlayarak kopyalanırlar. Derleyicilerin giriş kodları bu argümanları alarak kendi segment bölgesi içerisine yerleştirmektedir. Ayrıca programın ismiyle ilk argüman arasında boşluk yerine özel kesim karakterleri kullanılabilir.

Aşağıda komut satırı argümanlarını basanörnek bir program görüyorsunuz.

```
#include <stdio.h>

void main(int argc, char *argv[])
{
    int k;

    for (k = 0; k < argc; ++k)
        printf("argv[%d] : %s\n", k, argv[k]);
}
```

Yukarıdaki programı komut satırından çeşitli argümanlar vererek çalıştırınız.

Komut satırı argümanlarının isimleri **argc** ve **argv** olmak zorunda değildir. Bunlar yerine herhangi iki isim de kullanabilirsiniz. Ancak **argc** ve **argv** isimleri (**argc**, argument counter, **argv**, argument variable list sözcüklerinden kısaltılmıştır) herkes tarafından geleneksel olarak kullanıldığından programı daha okunabilir hale getirir.

Komut satırı argümanlarını alan programlar önce girilen argümanı yorumlar ve test ederler. Aşağıdaki örneği izleyiniz:

```
void main(int argc, char *argv[])
{
    if (argc == 1) {
      printf(" Lütfen bir argümanla çalıştırınız!..\n");
      exit(1);
    }
    if (argc > 2) {
      printf("Fazla sayıda argüman!..\n");
      exit(1);
    }
    arg_test(argv[1]);
    ...
}
```

Burada eğer program bir komut satırı argümanı verilerek çalıştırılmadıysa bir mesajla sonlandırılıyor. Benzer biçimde fazla sayıda argüman için de böyle bir kontrol yapılmıştır.

DOS'ta olduğu gibi bazı sistemlerde **main** fonksiyonu üçüncü bir parametre alabilir. Buradaki üçüncü parametre sistemin çevre değişkenlerine ilişkin bir karakter türünden bir göstericiyi gösteren göstericidir.

```
void main(int argc, char *argv[], char *env[])
{
    ...
}
```

Burada çevre değişkenleri üzerinde durarak konuyu genişletmeyeceğiz. Ancak son olarak şunu da ekleyelim. `main` fonksiyonuna tüm parametreleri geçmek zorunda değilsiniz. Örneğin:

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

gibi yalnızca birinci parametrenin kullanıldığı bir tanımlama geçerlidir. Ancak yalnızca ikinci parametre kullanılamaz. Örneğin:

```
void main(char *argv[])
{
    ...
}
```

gibi bir tanımlama geçerli değildir.

31.2 DOSYALARA İLİŞKİN TEMEL KAVRAMLAR

Dosyalar disket, disk ya da CD gibi ikincil belleklerde tanımlanmış alanlardır. Dosyalara onlara vermiş olduğumuz isimleri kullanarak erişebiliriz. İsimlerinin yanısıra dosyaların sistemden sisteme değişebilen çeşitli özellikleri (attributes) de vardır.

Dosyanın Açılması : Dosyalar disket, disk ya da CD'lerde olduğuna göre, bunlar üzeinde işlemlerin yapılabilmesi ancak RAM'e çekilmesiyle mümkün olabilir. Büyük dosyaların tamamen RAM'e çekilmesi uygun olmadığından ancak kısım kısım çekilerek işlemler yürütülebilir. **Dosyanın açılması, dosyayla ilgili birtakım başlangıç işlemlerinin yapılması demektir**. Örneğin işletim sistemi dosyanın açılması sırasında ilgili dosyanın ismini ve özelliklerini birtakım içsel tablolarda saklayabilir; dosya işlemleri için disk ile RAM arasında aktarımı sağlayacak tampon bölgeleri tahsis edebilir.

Dosyanın Kapatılması: Dosyanın açılması sırasında yapılan işlemlerin geri alınması anlamına gelir. İşletim sistemi bir dosyanın kapatılması sırasında disk ile RAM arasında aktarımı sağlayacak tampon bölgelerin tazelenmesi gibi kritik işlemler de yapabilirler. Ancak açılan dosyaların işletim sisteminin sağladığı fonksiyonlarla kapatılması gerekir.

Dosya Göstericisi (file pointer): Dosya göstericisi, o anda dosyanın kaçıncı offseti üzerinde işlem yapıldığını gösteren bir sayıdır. İşletim sisteminin dosya işlemlerini yapan sistem fonksiyonları okuma ve yazma işlemlerini dosya gösterici-

sinin gösterdiği yerden yaparlar. Dosya göstericisinin konumlandırılması da sistem fonksiyonları tarafından yapılmaktadır. İşletim sistemlerinin çoğunda örneğin dosyanın 100. byte ofsetinden 10 byte okunacaksa:

1) Dosya gösterici 100. byte değerine konumlandırılır.
2) Dosya göstericisinin gösterdiği yerden 10 byte okunur.

Tampon Bölge (buffer): İşletim sistemi diske erişim işlemini dosyaların belli bölümlerini RAM'de tutarak azaltabilir. Dosyaların belli bölümlerinin tutulduğu bellek bölgelerine tampon bölge denilmektedir. Dosyaya yazma ya da dosyadan okuma yapıldığında işletim sistemi önce bu tampon bölgeye bakar. Eğer ilgili bölümü burada bulursa diske erişim yapmaz; diske erişim yalnızca dosyanın tampon bölgede olmayan kısımları için yapılmaktadır.

31.3 İŞLETİM SİSTEMLERİNİN DOSYA İŞLEMLERİ

İşletim sistemleri dosya işlemlerini çeşitli tablolar yardımıyla yürütürler. Bu tablolardan biri açılan dosyalar hakkında önemli bilgilerin tutulduğu dosya tablosudur. Dosya tablosunun elemanları açık olan dosyaların isimlerini, özelliklerini, diskteki yerlerini vs. tutan kayıtlar biçimindedir. İçsel yapıları işletim sisteminden işletim sistemine değişebilir. Çoğu kez dosya tablosunun uzunluğu sistem konfigüre edilirken kullanıcı tarafından belirlenmektedir. Örneğin **DOS** işletim sisteminde **FILES = nnn** satırı bu görevi yerine getirir. Dosya işlemlerinin işletim sisteminin aşağı seviyeli sistem fonksiyonlarıyla yapıldığını belirtmiştik. Tipik bir işletim sisteminde dosyayı açan, kapayan, okuma ve yazma yapan fonksiyonlar vardır. **Dosya açan fonksiyonlar genellikle dosyanın bilgilerini ulaşmayı sağlayan dosya tablosuna ilişkin bir indeks numarasıyla geri dönerler. Dosya tablosunda açılan dosya bilgilerini bulmakta kullanılan bu sayıya "file handle" denir. Diğer dosya fonksiyonlarının hemen hepsi dosyayı karakterize eden "file handle" değerini girdi alarak dosyaya ulaşırlar.** Aşağıda temsili bir dosya tablosunun yapısını görüyorsunuz.

DOSYA TABLOSU

File Handle	Dosya İsmi	Açış modu	Dosya Göstericisinin Konumu	Kullanılan tampon bölgeler
----	----	----	----	----
10	deneme.dat	read/write	1250	7-10
11	readme.txt	read	5318	12-15
12	sampel.bmp	read	8118	20-25
----	----	----	----	----

Yukarıda da bahsedildiği gibi dosya işlemleri sırasında işletim sistemi diske erişmeyi en aza indirmek amacıyla tamponlama (buffering) tekniği kullanabilir. Tamponlama tekniği ile işletim sistemi, dosyadan bir byte bile okunacak olsa dosyanın belli kısmını RAM'de tampon bir bölgeye çeker. Okuma ve yazma fonksiyonları da önce bu tampon bölgeye bakarlar. Eğer dosyanın ilgili kısmı buradaysa diske erişmeye gerek kalmaz. Tabi tampon bölgeler ile disk arasındaki tüm ilişkileri işletim sistemi içsel olarak düzenler. Tampon bölgelerin hangi uzunlukta tutulacağı işletim sistemlerinde konfigürasyon sırasında belirlenebilir. Örneğin DOS'ta **BUFFERS = nnn** bu işlemi yerine getirmektedir (1 buffer = 1 sektör = 512 byte).

C Programcısı olarak dosya işlemlerini 2 biçimde yürütebiliriz

1) Standart C Fonksiyonlarını Kullanarak. Bunlar sistemlerin hemen hepsinde bulunan taşınabilir fonksiyonlardır. Kendi içlerinde işletim sisteminin sağladığı aşağı seviyeli fonksiyonları kullanırlar.

2) İşletim sisteminin sağladığı aşağı seviyeli fonksiyonları kullanarak. Bu fonksiyonlara nasıl erişileceği işletim sistemleri tarafından belirlenmiştir. Ancak dosyalara bu tür fonksiyonlarla eriştiğinizde yazdığınız kod taşınabilir olmayacaktır. Kitabımızda böylesi özel fonksiyonlara değinmeyeceğiz.

31.4 DOSYA İŞLEMLERİNDE KULLANILAN STANDART C FONKSİYONLARI

31.4.1 Dosyanın Açılması

Bir dosya ile işlem yapılabilmesi için önce dosyanın açılması gerekir. Bunun için `fopen` fonksiyonu kullanılır.

```
FILE *fopen(char *filename, char *mode);
```

`fopen` birinci parametresiyle verilmiş olan dosyayı, ikinci parametresiyle verilmiş olan modda açar. Dosya belirtilirken yol ifadesi kullanılabilir. `fopen` fonksiyonunun **FILE** türünden bir adresle geri döndüğünü görüyorsunuz. **FILE**, `stdio.h` içerisinde bildirilmiş bir yapıdır. `fopen` dosyayı açabilmek için işletim sisteminin aşağı seviyeli fonksiyonlarını kullanmaktadır. Dosyaya erişmek için gerek aşağı seviyeli gerekse yüksek seviyeli birtakım bilgiler bu yapıda saklanır. Dosyanın açılması durumunda geri dönüş değeri ile gösterilen yapı `fopen` tarafından dinamik olarak tahsis edilmektedir. Dosyanın açılamaması durumunda `fopen` NULL göstericiye geri döner. Bu yüzden dosyanın açılıp açılamadığı mutlaka test edilmelidir. Çünkü dosyanın açılamaması için pekçok neden olabilir. Bunlar:

- Dosyanın diskte olmaması.
- Diskteki oragnizasyon bozukluğu
- Yanlış açış modunun kullanılmış olması.

- File handle yetersizliği (Dosya tablosunda boş yer olmaması)
- ...

Açış modu dosya üzerinde yapılacak işlemleri belirlemektedir. Bu parametrenin bir karakter göstericisi olduğunu görüyorsunuz. Açış modu aşağıdaki biçimlerde olabilir:

Örneğin:

Açış modu	İşlem türü
"r"	Yalnız okuma. Bu modda açılmış olan bir dosyaya yazma yapılamaz. Dosyanın açılması için daha önce yaratılmış olması gerekir.
"w"	Dosya diskte olsa da olmasa da yeniden yaratılır. Bu modda açılmış olan bir dosyadan okuma yapılamaz.
"a"	Var olan bir dosyanın sonuna ekleme yapmak amacıyla açılır. Bu modda açılmış olan bir dosyadan okuma yapılamaz.
"r+"	Okuma ve yazma. Bu modda açılmış olan bir dosyaya hem okuma hem de yazma işlemi yapılabilir. Ancak dosyanın daha önce yaratılmış olması gerekir.
"w+"	Dosya diskte olsa da olmasa da yeniden yaratılır. Dosyaya hem okuma hem de yazma işlemi yapılabilir.
"a+"	Var olan bir dosyanın sonuna ekleme yapmak amacıyla açılır. Bu modda açılmış olan bir dosyaya hem okuma hem de yazma işlemi yapılabilir.

```
FILE *fp;
...
fp = fopen("data", "w");
if (fp == NULL) {
     printf("Dosya açılamadı!..\n");
     exit(1);
}
...
```

Örneğimizde "**data**" isimli dosya "**w**" modu kullanılarak yaratılmak istenmiştir. Bu biçimde yaratılan bir dosyaya yalnızca yazma yapılabilir. Daha sonra dosyanın açılıp açılamadığının kontrol edildiğini görüyorsunuz. Yazımı kısaltmak amacıyla **fopen** fonksiyonunu **if** içerisinde de çağırabilirsiniz:

```
FILE *fp;
...
if ((fp = fopen("data", "w")) == NULL) {
    printf("Dosya açılamadı!..\n");
    exit(1);
}
...
```

Bir dosya açıldığında dosya göstericisi otomatik olarak dosyanın başına alınmaktadır (0. offset). `fopen` fonksiyonunun geri dönüş değeri olan göstericinin açılan dosyanın bilgilerinden oluşan bir yapıyı gösterdiğini söylemiştik. **Biz bu göstericiye "dosya bilgi göstericisi" diyeceğiz. Ancak bu göstericiyle dosya tablosunda tutulan ve dosyanın hangi byte değeri üzerinde işlem yapılacağını gösteren "dosya göstericisi"yle karıştırmamalısınız.**

31.4.2 Dosyanın Kapatılması

Açılan her dosya programın sonunda kapatılmalıdır. Dosyanın kapatılmaması özellikle yazım işlemi yapılmışsa problemlere yol açabilir.

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

`fclose` kapatılacak dosyanın hangisi olduğunu parametresi olan dosya bilgi göstericisiyle anlamaktadır. Kapatma işlemi başarılıysa 0 değerine başarısızsa `EOF` değerine geri döner. `EOF` `stdio.h` içerisinde tanımlanmış bir sembolik sabittir.

```
FILE *fp;
    ...
    if ((fp = fopen("data", "w")) == NULL) {
        printf("Dosya açılamadı!..\n");
        exit(1);
    }
    ...
    fclose(fp);
```

Normal olarak açılmış bir dosyanın kapatılamaması söz konusu olmadığından `fclose` fonksiyonunun başarısının test edilmesine gerek duyulmaz.

31.4.3 Dosyadan Bir Karakter Okuyan ve Dosyaya Bir Karakter Yazan Fonksiyonlar

Dosyadan karakter okuyan iki fonksiyon vardır. Bunlar `fgetc` ve onun makrosu biçiminde bulunan `getc` fonksiyonlarıdır. Önce `fgetc` fonksiyonunu inceleyeceğiz.

```
int fgetc(FILE *fp);
```

Dosya göstericisinin gösterdiği yerden bir byte bilgiyi okur. **Okuma ve yazma yapan tüm dosya fonksiyonları dosya göstericisinin değerini okunan ya da yazı-**

lan bilgi kadar artırmaktadır. `fgetc` fonksiyonu da dosyadan 1 byte okuduğuna göre dosya göstericisinin değerini bir artıracaktır. `fgetc` fonksiyonunun parametresi `fopen` fonksiyonundan elde edilen dosya bilgi göstericisidir. Geri dönüş değeri eğer okuma başarılıysa okunan karakter, başarısızsa `EOF` değeridir. `EOF` değeri `stdio.h` başlık dosyası içinde sembolik sabit biçiminde (genellikle -1 değerine) tanımlanmıştır.

```
#define EOF (-1)
```

Okuma işlemi dosyanın okuma modunda açılmamasından, dosyanın sonuna gelindiğinden ya da diskteki organizasyon bozukluğundan başarısızlıkla sonuçlanabilir. Aşağıda bir dosyayı karakter karakter okuyarak ekrana yazan bir mini program örneği görüyorsunuz.

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

void main (int argc, char *argv[])
{
    FILE *fp;
    int ch;

    if (argc != 2) {
        printf("Fazla ya da eksik parametre!..\n");
        exit(1);
    }
    if ((fp = fopen(argv[1], "r")) == NULL) {
        printf("Dosya açılamadı!..\n");
        exit(1);
    }
    ch = fgetc(fp);
    while(ch != EOF) {
        putchar(ch);
        ch = fgetc(fp);
    }
    fclose(fp);
}
```

Örneğimizde komut satırı argümanı olarak alınan dosyadan `fgetc` ile karakterler teker teker okunmaktadır. `fgetc` her okuma yapıldığında dosya göstericisini bir artırır. Dosya sonuna gelindiğinde `EOF` ile döngüden çıkılmıştır.

`getc` fonksiyonu `fgetc` kullanılarak yapılmış bir makrodur. `fgetc` ile aynı işlevlere sahiptir.

```
int getc(FILE *fp);
```

Dosyaya bir karakter yazan `fputc` ve makro biçiminde bulunan `putc` iki standart C fonksiyonudur.

```
int fputc (int ch, FILE * fp);
```

Yazılacak karakter Yazılacak dosya

fputc yazma başarılıysa yazılan karaktere (birinci parametresine), başarısızsa EOF değerine geri döner.

Aşağıda fputc kullanılarak dosya kopyalama örneği verilmiştir; inceleyiniz:

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

void main(int argc, char *argv[])
{
    FILE *kaynak, *hedef;
    int ch;

    if (argc != 3) {
        printf("Fazla ya da eksik parametre!..\n");
        exit(1);
    }
    if ((kaynak = fopen(argv[1], "r")) == NULL) {
        printf("Dosya açılamadı!..\n");
        exit(1);
    }
    if ((hedef = fopen(argv[2], "w")) == NULL) {
        printf("Dosya açılamadı!..\n");
        exit(1);
    }
    ch = fgetc(kaynak);
    while (ch != EOF) {
        fputc(ch, hedef);
        ch = fgetc(kaynak);
    }
    printf("Dosya kopyalandı...\n");
    fclose(kaynak);
    fclose(hedef);
}
```

Örnek programımız iki komut satırı argümanı kullanıyor. Bu argümanlardan birincisi kopyalanacak dosyanın ismi (kaynak dosya) ikincisi de kopyanın yazılacağı dosyanın ismidir. Komut satırından aşağıdaki gibi çalıştırabilirsiniz:

```
<.EXE dosyanın ismi> <Kopyalanacak dısya ismi> <Kopyasının ismi>
```

fputc fonksiyonunun putc isimli bir de makrosu vardır. Aralarında işlevsel bir fark olmadığı için herhangi birini kullanabilirsiniz:

```
int putc(int ch, FILE *fp);
```

31.4.4 Dosya Sonunun Tespit Edilmesi

İşlemlerin çoğunda dosya baştan sona taranır. "Dosya sonuna kadar" işlem yapılması en sık rastlanılan durumdur. Dosya sonuna gelinip gelinmediği `feof` fonksiyonuyla saptanabilir:

```
int feof(FILE *fp);
```

`feof` dosya göstericisi dosya sonunu gösterdiğinde 1 değerini göstermediğinde de 0 değeri ile geri döner. Böylece "dosya sonuna kadar" işlem yapan algoritmalar aşağıdaki gibi bir `while` döngüsüyle tasarlanabilirler:

```
...
while (!feof(fp)) {
    /* İşlemler */
}
...
```

31.4.5 fgets ve fputs Fonksiyonları

Dosyaya bir dizi karakter yazmakta kullanıdığımız iki önemli fonksiyon vardır:

```
int *fgets(char *str, int n, FILE *fp);
int fputs(char *s, FILE *fp);
```

`fgets` klavyeden `string` alan `gets` gibidir. Yalnızca klavye yerine `fp` ile belirtilen dosyadan (ve tabi dosya göstericisinin gösterdiği yerden) okuma yapar. `fgets` dosyadan okuduğu **n - 1** karakteri birinci parametresiyle belirtilen adresten başlayarak yerleştirir ve sonuna NULL karakteri ekler. Eğer `'\n'` karakteri ile karşılaşırsa ya da dosya sonuna gelinmişse okuma işlemini bitirir. Bu durumda `'\n'` karakteri de stringe kopyalanır. `fgets` başarı durumunda okumanın yapıldığı adrese, başarısızlık durumunda ise **NULL** göstericiye geri döner; bu değer test edilmelidir. Aşağıda `fgets` ile bir metin dosyasının içeriğini ekrana yazan örnek görüyorsunuz:

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

#define MAX_LINE      500

void main(int argc, char *argv[])
{
    FILE *fp;
    char str[MAX_LINE];

    if (argc != 2) {
        printf("Fazla ya da eksik parametre!..\n");
        exit(1);
    }
    if ((fp = fopen(argv[1], "r")) == NULL) {
        printf("Dosya açılamadı !..\n");
        exit(1);
```

```
    }
    while (!feof(fp))
        if (fgets(str, MAX_LINE, fp))
            printf("%s", str);
    fclose(fp);
}
```

fputs fonksiyonu da puts fonksiyonu gibidir; ancak yazma işlemini ekran yerine dosyaya yapar. fputs başarı durumunda 0 değerine başarısızlık durumunda 0 dışı bir değere geri döner.

Dosyaya formatlı bilgi yazmak ve dosyadan formatlı bilgi okumak için fprintf ve

31.4.6 Formatlı Dosya İşlemleri

fscanf fonksiyonları kullanılır. fprintf, printf fonksiyonu ile işlevsel olarak tamamen özdeştir. printf ekrana nasıl yazıyorsa fprintf de dosyaya aynı biçimde yazar. Prototiplerini inceleyiniz:

```
int fprintf (FILE *fp, char *format, arg_list);
int fscanf (FILE *fp, char *format, arg_list);
```

Bu fonksiyonlar printf ve scanf fonksiyonlarından farklı olarak dosya göstericisi parametresi aldığını görüyorsunuz. Yazdığınız bir programda printf fonksiyonları yerine fprintf kullanırsanız ekran yerine dosyaya yazma yaparsınız. Aşağıdaki örnek programı inceleyiniz:

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

#define SIZE    10

void main(void)
{
    int p[SIZE] = {10, 2, 56, 80, -2, 23, 80, 21, -4, 41};
    FILE *fp;
    int max, k

    if ((fp = fopen("data", "w")) == NULL) {
        printf("File cannot open!..\n");
        exit(1);
    }
    fprintf(fp, "Sayılar:\n");
    for (k = 0; k < SIZE; ++k)
        fprintf(fp, "%d\n",p[k]);
    max = p[0];
    for (k = 1; k < SIZE; ++k)
        if (p[k] > max)
            max = p[k];
    fprintf(fp, "En Büyük Sayı:\n%d", max);
    fclose(fp);
}
```

data isimli dosya aşağıdaki gibi oluşturulacaktır:

```
Sayılar:
10
2
56
80
-2
23
80
21
-4
41
En Büyük Sayı:
80
```

Bir noktayı vurgulamak istiyoruz. **fprintf** dosyaya ASCII formatında yazar. Örneğin:

```
int x = 1240;
...
fprintf(fp, "%d", x);
```

ile dosya içerisine 2 byte değil 4 byte yazılır. Dosya içerisine '1', '2', '4', '0' ASCII karakterleri yazılmaktadır; ikili sistemde **1240** değil!..

fscanf fonksiyonu da **scanf** fonksiyonu gibidir. Ancak klavye yerine bilgiyi dosyadan okur.

```
int x;
fscanf(fp, "%d", &x);
```

ile dosya göstericisinin gösterdiği yerdeki boşluksuz ASCII karakterleri **x** içerisine yerleştirilir.

31.4.7 Dosya Göstericisinin Konumunun Değiştirilmesi

Okuma ve yazma yapan fonksiyonların hepsi okunan ya da yazılan sayıda byte değeri kadar dosya göstericisini ilerletir. Tüm okuma yazma fonksiyonları dosya göstericisinin konumunu ilerletmektedir. Ancak okuma ve yazma dışında da dosya göstericisinin değeri değiştirilebilir; bunun için **fseek** fonksiyonu kullanılmaktadır:

```
int fseek (FILE *fp, long offset, int origin);
```

İkinci parametresi olan **offset** dosya göstericisinin konumlandırılacağı pozisyonu belirtir. Üçüncü parametre konumlandırma işleminin nereden başlayarak yapılacağını anlatır. Üçüncü parametre **0** ise konumlandırma dosyanın başından itibaren; **1** ise bulunulan yerden itibaren, **2** ise dosyanın sonundan itibaren yapılır. Bu değerler **stdio.h** içerisinde sembolik sabit biçiminde tanımlanmışlardır:

```
0 ► SEEK_SET
1 ► SEEK_CUR
2 ► SEEK_END
```

Örneğin:

```
fseek(fp, 100L, SEEK_CUR);
```

ile dosya göstericisi o andaki pozisyonundan 100 ileriye konumlandırılır. Benzer biçimde fonksiyon aşağıdaki gibi çağrılmış olsaydı:

```
fseek(fp, -100L, SEEK_CUR);
```

Dosya göstericisi o andaki pozisyonundan 100 ge iye konumlandırılırdı (İkinci parametrenin negatif bir değer aldığına dikkat ediniz). **fseek** dosya göstericisini istenilen pozisyona konumlandıramazsa sıfır dışı l-ir değere geri döner.

Dosya göstericisinin tekrar dosyanın başına alınması durumuna çok sık rastlanır. Bu işlem **rewind** fonksiyonu ile yapılabilir. Prototip.ni inceleyiniz:

```
void rewind (FILE *fp);
```

Aynı şey **fseek** ile de yapılabilirdi:

```
fseek(fp, 0L, SEEK_SET);
```

Dosya göstericisinin o anda gösterdiği pozisyonda `ftell` fonksiyonuyla elde edilebilir. Prototipini inceleyiniz:

```
long ftell(FILE *fp);
```

`ftell` başarısızlık durumunda -1L değerine geri döner.

31.4.8 Metin ve İkili Dosyalar

Dosyalar metin (text) ve ikili (binary) dosyalar olmak üzere iki kısma ayrılırlar. Dosyanın hangi türden olduğu **fopen** ile açılırken belirtilmektedir. Açış moduna metin dosyaları için "t" ve ikili dosyalar için "b" eklenir. Örneğin "r+b" biçiminde bir açış modu var olan bir dosyanın okuma ve yazma için ikili olarak açılacağı anlamına gelir. Benzer biçimde "wt" gibi bir açış modu da dosyanın metin modunda yaratılacağını gösterir. Varsayılan açış modu metindir. Yani "t" ya da "b" ile özellikle bir belirleme yapılmamışsa dosya metin modunda açılır.

DOS altında metin ve ikili dosyalar arasında basit iki fark vardır.

1) Metin dosyalarına '\n' karakteri LF (0AH) gönderilirse CR (0DH) / LF (0AH) çifti yazılır. Okuma yapılırken CR nin arkasınan LF geliyorsa yalnızca LF karakteri okunur.

2) CTRL_Z karakterinin (26 numaralı ASCII karakteri) metin dosyalarını sonlandırdığı varsayılır. Bu nedenle dosya eğer metin modunda açılmışsa okuma ve

yazma fonksiyonları bu karakteri gördüğünde dosyanın bittiğini varsayacaklardır.

İçerisinde formatlanmamış yazı bilgisi bulunan dosyalar doğal olarak metin modunda yaratılmışlardır. Bu nedenle bu tür dosyaları metin modunda açabilirsiniz. Ancak çalışabilen dosyalarda olduğu gibi özel dosyaları ikili modda açarak işlem yapmalısınız. Örneğin 31.4.3'te verdiğimiz kopyalama programında ikili açış modu kullanılırsa her türlü dosya kopyalanabilir.

31.4.9 fread ve fwrite Fonksiyonları

Bellekte bulunan bir bilgiyi blok olarak dosyaya yazan dosyadan bilgiyi blok olarak belleğe okuyan `fwrite` ve `fread` en sık kullanılan dosya fonksiyonlarıdır. Prototiplerini inceleyiniz:

```
unsigned fread (void *ptr, unsigned size, unsigned n, FILE *fp);
unsigned fwrite (const void *ptr, unsigned size, unsigned n, FILE *fp);
```

`fread` dosya göstericisinin gösterdiği yerden ikinci parametresi ve üçüncü parametresinin çarpımı kadar bilgiyi `(size * n) ptr` adresinden başlayarak belleğe okur. İkinci parametresi okunacak bilginin uzunluğunu (size), üçüncü parametresi de kaç tane okunacağını (n) göstermektedir. `fread` okuyabildiği kadar bilgiyi okur ve okuyabildiği `n` sayısına geri döner. `fwrite` fonksiyonunun parametreleri de `fread` ile aynı işlevlere sahiptir. Yalnızca `fwrite` okuma yerine yazma yapar.

Bellekte sürekli bir biçimde bulunan diziler ve yapılar bu fonksiyonlar ile doğrudan dosyalara yazılabilirler. Örneğin:

```
fwrite(&n, sizeof(int), 1, fp);
```

ile `n` değişkeninin içerisindeki sayı `fp` ile belirtilen dosyaya yazılmaktadır. `fwrite` parametre olarak `n` değişkeninin adresini ve uzunluğunu almıştır. `n` `int` türünden olduğuna göre uzunluğu `sizeof(int)` biçiminde sistem bağımsız olarak belirlenebilir. Bilginin ASCII formatıyla yazılmadığına doğrudan bellekteki biçimiyle yazıldığına dikkat ediniz. Ayrıca bu örnekte söz konusu olan dosyanın ikili modda açılmış olması gerekir. Benzer biçimde bir diziyi de aşağıdakigibi blok olarak dosyaya yazabiliriz:

```
int a[LEN];
...
fwrite(a, sizeof(int), LEN, fp);
```

Dosyaya toplam `sizeof(int) * LEN` byte bilgi yazılmıştır. Yapılar da bellekte sürekli olarak bulunduklarına göre blok olarak dosyadan okunabilirler. Örneğin:

```
struct INSAN {
    char adi[30];
    char adres[50];
    int no;
```

```
);
....
struct INSAN insan;
FILE *fp;
...
if ((fp = fopen("data", "r+")) == NULL) {
    printf("Dosya açılamadı!..\n");
    exit(1);
}
...
fread(&insan, sizeof(INSAN), 1, fp);
```

Burada dosya göstericisinin gösterdiği yerden **sizeof(INSAN)** kadar bilgi yapıya yerleştirilmektedir.

fread ve **fwrite** fonksiyonlarının geri dönüş değerleri üçüncü parametresiyle belirtilen okunan ya da yazılan parça sayısıdır. Bu sayı test amaçlı kullanılabilir.

```
double n[10];
int test;
...
test = fread(n, sizeof(double), 10, fp);
if (test != 10) {
    printf("Okuma hatası!..\n");
    exit(1);
}
```

SORAMADIKLARINIZ...

S1) Dosya özellikleri nasıl belirlenmektedir. Örneğin DOS işletim sisteminde **fopen** ile yalnızca okunabilen (read only) ya da gizli dosya (hidden) açılabilir mi?

C2) Dosya özellikleri sistemden sisteme değişebildiği için bunların değiştirilmesi de **fopen** ile değil özel sistem fonksiyonlarıyla yapılmaktadır. Örneğin dosya açan aşağı seviyeli DOS fonskiyonları parametre olarak dosya özelliğini de alırlar.

S2) Metin ve ikili dosyaların özellikleri tüm sistemlerde aynı mıdır?

C2) Metin ve ikili dosyaların özellikleri sistemden sisteme değişebilir. Yukarıda örnekler DOS işletim sistemi içinverilmiştir. UNIX işletim sistemi ile DOS arasında farklılık vardır.

S3) Aynı dosyayı birkaç kere **fopen** fonksiyonuyla açabilir miyiz?

C3) Aynı dosyaya ilişkin birkaç dosya bilgi göstericisiyle çalışmak bazı sistemlerde problemlere yolaçabilir. Bunu yapabilmek için sisteminizi iyi tanımalısınız.

EK - A

A.1. OPERATÖRLERİN ÖNCELİK TABLOSU

OPERATÖRLER	ÖNCELİK SIRASI
() [] . ->	Soldan sağa
! ~ ++ — + - * & (tür) sizeof	Sağdan sola
* / %	Soldan sağa
+ -	Soldan sağa
<< >>	Soldan sağa
< >= > >=	Soldan sağa
== !=	Soldan sağa
&	Soldan sağa
^	Soldan sağa
|	Soldan sağa
&&	Soldan sağa
||	Soldan sağa
?:	Sağdan Sola
= += -= *= /= %= &= ^= |= <<= >>=	Sağdan sola
,	Soldan sağa

A.2. ANAHTAR SÖZCÜKLER

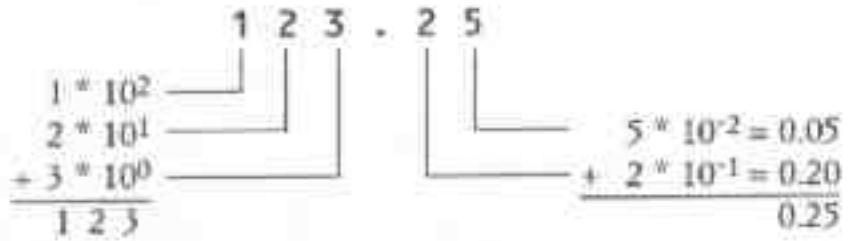
Anahtar sözcükler derleyici için özel anlam ifade eden, değişken olarak kullanılması yasaklanmış olan sözcüklerdir. Standart C'nin anahtar sözcükleri aşağıdadır:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

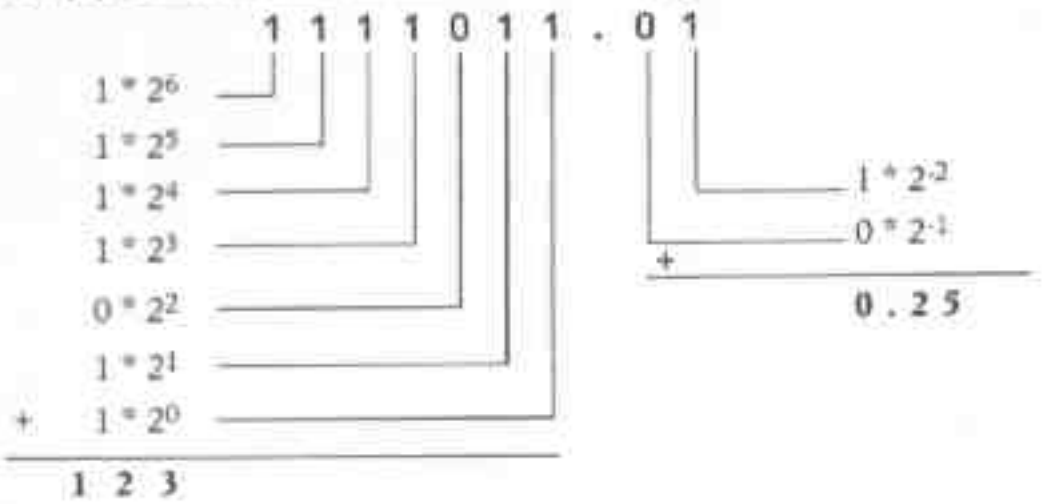
A.3. ANSI / IEEE 754 GERÇEK SAYI FORMATLARI

A.3.1 Gerçek Sayıların Bellekteki Görünümleri

Gerçek sayıların noktadan sonraki kesirli kısımları üzerinde çalıştığımız sayı sisteminin negatif üsleriyle ifade edilir. Örneğin 123.25 sayısı aşağıdaki gibi çözümlenir:



Aynı durum 2'lik sistemde de 2'nin katlarıyla yapılmaktadır. Örneğin yukarıdaki sayıyı 2'lik sistemde sembolik olarak aşağıdaki ifade edebiliriz:



Sayının kesirli kısmını belirtmekte kullandığımız **nokta** da ikilik sistemde ifade edilmelidir. En iyi çözüm sayıyı bir bütün olarak ele almak ve noktanın yerini ilk bite göre belirlemektir. Örneğin yukarıdaki sayı;

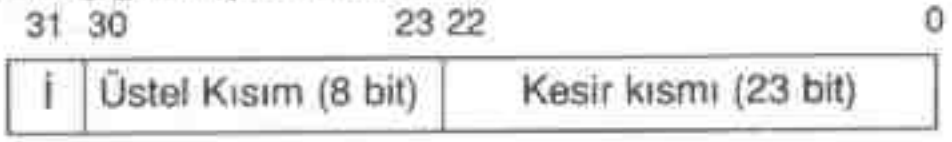
```
1111011.01 = 1.11101101 * 2^6
```

biçiminde ifade edilir. Kayan noktalı gerçek sayı formatlarında noktanın yeri ilk bite göre belirtilir. Ayrıca gerçek sayıların **unsigned** özelliği de yoktur. Bu durumda 1 bit de sürekli işaret için kullanılacaktır. Özetle gerçek sayılar üç kısımdan oluşur:

1) İşaret biti (1 ise negatif, 0 ise pozitif)
2) Kesir kısmı (bu kısma mantis de denilmektedir)
3) Üstel kısmı

A.3.2 Kısa Gerçek Sayı Formatı (Short real format)

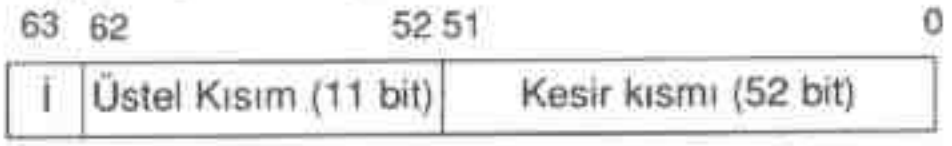
Kısa gerçek sayı formatı 4 byte uzunluğundadır. *C*'deki `float` türü bu formatla ifade edilir. Aşağıdaki şekli inceleyiniz:



Kesir kısmı için 23 bit ayrılmıştır. Üstel kısım doğrudan değil pozitif ise +127 ile ile toplanarak negatif ise +127'den çıkarılarak yazılır. Bu tür üstel ifadeye **yanlı üstel ifade (biased exponent)** denilmektedir. Örneğin sayının üstel kısmı 5 ise buraya 132, −5 ise buraya 122 yazılır. En soldaki bit işaret bitidir. İşaret biti 0 ise sayı pozitif 1 ise negatiftir.

A.3.3 Uzun Gerçek Sayı Formatı (Long real format)

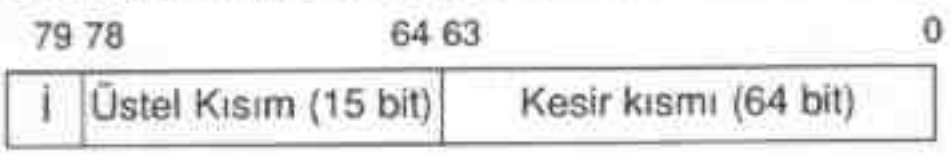
Uzun gerçek sayı formatı 8 byte uzunluğundadır. *C*'deki **double** bu formatla ifade edilir. Aşağıdaki şekli inceleyiniz:



Kesir kısmı için 52 bit, üstel kısım için 11 ayrılmıştır. Üstel kısmın yanlı değeri +1023'tür. Yani üstel kısmın bilgisi +1023'e toplanarak ya da bu değerden çıkartılarak elde edilmektedir. En yüksek anlamlı bit işaret bitidir.

A.3.4 Genişletilmiş Gerçek Sayı Formatı (Extended real format)

10 byte uzunluğundadır. *C*'de `long double` türü bu formatla ifade edilir.



Kesir kısmı için 64 bit ayrılmıştır. Üstel kısmın yanlı değeri +16383'tür. En yüksek anlamlı bit yine işaret bitidir.

EKLER - B
STANDART C FONKSİYONLARI

Bu bölümde çok kullanılan standart C fonksiyonlarının bazıları kısa bir biçimde tanıtılmıştır. Bazı tür fonksiyonlar hakkında ayrıntılı bilgiyi onlara ilişkin bölümlerde bulabilirsiniz. Örneğin dosya fonksiyonları dosyaların anlatıldığı bölümde, dinamik bellek fonksiyonları da dinamik bellek fonksiyonlarının anlatıldığı bölümde ayrıntılı bir biçimde ele alınmaktadır. Bunun dışında kitabımızda açıklanmayan diğer fonksiyonlar için ayrıntılı kaynaklara başvurabilirsiniz. Derleyicilerin çoğunun yardım menülerinde de standart C fonksiyonları ayrıntılı bir biçimde tanıtılmaktadır.

GİRİŞ ÇIKIŞ FONKSİYONLARI (STDIO.H)

int getchar(void);

Klavyeden bir karakter alarak bunun ASCII karşılığını geri dönüş değeri olarak verir. Bu fonksiyon ENTER tuşuna gereksinim duyar.

int putchar(int ch);

Parametresiyle belirtilen karakteri ekrana yazar. Geri dönüş değeri yazılan karakterin kendisidir.

int printf(char *format, ...);

Ekrana formatlı bilgi yazdırmak amacıyla kullanılan en genel çıkış fonksiyonudur. Birinci parametresi format bilgisinin bulunduğu adrese ilişkin bir göstericidir. Programcı bu karakter göstericisini genellikle string ifadesi biçiminde belirtir. `printf` değişken sayıda parametre alan bir fonksiyondur. Hangi sayıda parametre ile kullanılacağı format stringindeki **%** karakterlerinin sayısına bağlı olarak belirlenebilmektedir. `printf` ekrana yazılan karakter sayısına geri döner.

`printf` format stringindeki **%** karakterlerinin dışındaki tüm karakterleri ekrana yazar. % karakterini gördüğünde yanındaki karakteri format karakteri olarak ele alır. Format karakterinin genel biçimi aşağıdaki gibidir:

```
% [önek] [genişlik] [duyarlılık] <format karakteri>
```

Format karakterleri şunlardır:

d	Onluk sistemde **int** türü
0	Sekizlik (octal) sistemde **int** türü.
u	Onluk sistemde **unsigned int** türü
x	On altılık (hexadecimal) sistemde **int** türü (alfabetik karakterler küçük harfle yazılır.)
X	On altılık sistemde (hexadecimal) sistemde **int** türü (alfabetik karakterler büyük harfle yazılır.)
s	Verilen adresten başlayarak NULL karakter görene kadar tüm karakterler
f	Onluk sistemde **double** ve **float** türleri. (Tür **float** ise derleyici tarafından **double** türüne dönüştürülerek **printf** fonksiyonuna aktarılmaktadır.). Duyarlılık özellikle belirtilmemişse 6 duyarlıklı olarak yazdırılır.
e, E	**double** ve **float** türlerini üstel formatta yazar. **[-] m.dddddd e± xx** default duyarlılık 6'dır. **e** üstel gösterimdeki **e**'nin küçük harf olacağını, **E** de büyük harf olacağını anlatmaktadır.
g, G	**double** ve **float** türleri. Eğer sayının üstel kısmı -4'ten küçükse **%e** ya da **%E** gibi yazar. Eğer büyükse **%f** gibi yazar. Ayrıca baştaki ve sondaki sıfırlar yazdırılmaz.
p	16'lık sistemde gösterici
%	% karakterinin kendisi

Önek kısmı şu karakterlerden oluşabilir:

+	Sayıyı her zaman işaretli olarak yazar. Örneğin sayı pozitif 10 ise +10 biçiminde yazacaktır.
-	Sayının basamak sayısından fazla alan üzerine belirleme yapılmışsa sayıyı sola dayalı olarak yazar.
0	Sayı belirlenen alandan daha az yer kaplıyorsa başını sıfırlarla doldur.

Genişlik sayının ne kadar uzunlukta bir alana yazılacağını belirlemekte kullanılır. **double** ve **float** sayılarda **a.b** biçiminde ifade edilir. Burada **a** sayının toplam uzunluğu **b** ise noktadan sonraki uzunluğudur. Ayrıca yalnızca **.b** biçiminde belirtilirse yalnızca sayının noktadan sonraki duyarlılığını belirlemekte kullanılır.

Aşağıdaki örnekleri inceleyiniz:

```
void main(void)
{
    double x = 3.173;
    int y = 100;
    char *p = "Deneme";

    printf("%f\n%5.2f\n %5.1f\n", x, x, x);
    printf("%05d\n", y);
    printf("%10s\n", p);
}
```

Programın çıktısı aşağıdaki gibi olacaktır:

3	.	1	7	3	0	0	0

	3	.	1	7

		3	.	2

0	0	1	0	0

				D	e	n	e	m	e

Duyarlılığı azaltılan **double** ve **float** sayılar için yuvarlama yapıldığına dikkat ediniz:

int scanf(char *format, ...);

Klavyeden her türden bilgi girişi yapabilmek için kullanılan genel amaçlı bir fonksiyondur. **scanf** fonksiyonunun da birinci parameresi **printf** de olduğu gibi format karakterlerine ilişkin göstericidir. **scanf** okuma biçiminin nasıl olması gerektiğini bu karakter stringine bakarak anlayabilir. Okumanın yapılacağı parametreler adres olmalıdır. **scanf** klavyeden girilen karakter sayısına geri döner.

Format stringi **printf** ile aynı anlamdadır. Yalnız **printf** yazmaya ilişkin olduğu halde **scanf** okumaya ilişkindir. Yalnızca bazı farklılıkları vurgulamak istiyoruz.

- Format stringindeki bir boşluk karakateri, giriş alanında ona karşı düşen ilk boşluk olmayan karaktere kadar tüm boşluk karakterlerini atlaması anlamına gelir.

- Bunun dışındaki karakterler giriş alanı içersinde bulunmalıdır; ancak **scanf** bunları dikkate almaz. Örneğin:

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

gibi bir okutmayla:

```
100,200
```

gibi bir giriş yapılmalıdır. Format stringinde virgül kullanıldığı için giriş alanında da virgül konulmuştur.

- **double** sayı okutmak için **"%lf"**, **float** sayı okutmak için **"%f"** kullanılır. Bu-

nun dışında diğer format karakterleri printf ile aynıdır.

FILE *fopen(char *fname, char *mode);

Birinci parametresiyle belirtilmiş olan dosya ismini ikinci parametresiyle belirtilmiş olan modda açar. Başarı durumunda STDIO.H içerisinde bildirilmiş olan FILE türünden bir yapı adresine, başarısızlık durumunda NULL göstericiye geri döner. Açış modları dosya konusunun ele alındığı 31. bölümde ayrıntılı bir biçimde açıklanmıştır.

Aşağıdaki program parçasında "deneme" isimli bir dosya yalnızca okunmak için açılmıştır.

```
FILE *fp;
...
if (fp = fopen("deneme", "r")) == NULL) {
    printf("Dosya açılamadı!..\n");
    exit(1);
}
...
```

int fclose(FILE *fp);

Daha önce **fopen** ile açılmış olan dosya bu fonksiyonla kapatılır. Başarı durumunda sıfır değerine başarısızlık durumunda sıfır dışı bir değer geri döner.

int fputc(int ch, FILE *fp);

ch karakterini **fp** ile belirtilmiş olan dosyada dosya göstericisinin gösterdiği yere yazar. Başarı durumunda yazılan karakterin kendisine başarısızlık durumunda STDIO.H içerisinde sembolik sabit tanımlanmış olan **EOF** değerine döner.

int fgetc(FILE *fp)

Parametresi ile belirtilen dosyadan bir karakter okur ve bunu geri dönüş değeri olarak verir. Başarısızlık durumunda **EOF** değerine geri dönmektedir.

int feof(FILE *fp);

Parametresi ile belirtilen dosyanın sonuna gelinip gelinmediğini test eder. Eğer dosya sonuna gelinmişse sıfır değerine, dosya sonuna gelinmemişse sıfır dışı bir değere geri döner.

int fwrite(void *buf, int size, int count, FILE *fp);

Herbiri **size** kadar **count** tane byte değeri buf adresinden başlayarak **fp** ile belirtilen dosyaya yazar. Başarı durumunda **count** ile temsil edilen yazılan parça sayısına geri döner.

int fread(void *buf, int size, int count, FILE *fp);

Herbiri `size` kadar `count` tane byte değeri buf adresinden başlayarak `fp` ile belirtilen dosyadan okur. Başarı durumunda `count` ile temsil edilen okunan parça sayısına geri döner.

int fseek(FILE *fp, long offset, int origin);

Birinci parametresiyle belirtilen dosyaya ilişkin dosya göstericisini ikinci parametresiyle belirtilen `offset` konumuna yerleştirir. İkinci parametresindeki `offset`, üçüncü parametresindeki `origin`'e göre değişmektedir. `Origin` değerleri STDIO.H dosyası içerisinde aşağıdaki değerlere sembolik sabit olarak tanımlanmıştır.

0 (SEEK_SET)	Dosya başından itibaren
1 (SEEK_CUR)	Dosya göstericisinin o anda bulunduğu konumdan itibaren
2 (SEEK_END)	Dosya sonundan itibaren

STRING FONKSİYONLARI (STRING.H)

int strlen(char *str);

Bir karakter dizisinin uzunluğunu bulmakta kullanılmaktadır. Parametresi olan adresten başlayarak NULL karakteri görene kadar tüm karakterlerin sayısını bulur.

Aşağıdaki örnekte ekrana 9 sayısı basılacaktır:

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

void main(void)
{
    char *p = "Deneme!..";

    printf("%d\n", strlen(str));
}
```

char *strcpy(char *dest, char *source);

İkinci parametresiyle belirtilen adresten başlayarak NULL karakter görene kadar tüm karakterleri (NULL karakter de dahil) birinci parametresiyle belirtilen adresten başlayarak kopyalar. Geri dönüş değeri birinci parametresiyle belirtilen `dest` (yani kopyalamanın yapıldığı) adresidir.

Aşağıdaki örnek programda ekrana **İstanbul** yazısı basılacaktır.

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

void main(void)
{
```

```
    char *p = "İstanbul";
    char str[15];

    strcpy(str, p);
    printf("%s\n", str);
}
```

int strcmp(char *s1, char *s2);

Adresleriyle belirtilmiş olan İki karakter dizisini karşılaştırır. Eğer birinci karakter dizisi ikincisinden büyükse pozitif bir değere, ikinci karakter dizisi birincisinden büyükse negatif bir değere, iki karakter dizisi birbirine eşitse sıfır değerine geri döner.

Aşağıdaki örnekte klavyeden alınan karakter **password** isimli bir dizi ile karşılaştırılmıştır.

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

void main(void)
{
    char *password = "dolunay";
    char str(15(;
    printf("Password:");
    gets(str);
    if (!strcmp(str, password))
        printf("Tamam!..\n");
    else
        printf("Tamam değil!..\n");
}
```

char *strcat(char *s1, char *s2);

İkinci parametresiyle belirtilen adresten başlayarak NULL karakteri görene kadar tüm karakterleri birinci parametresiyle belirtilen dizinin sonuna kopyalar.

Aşağıdaki örnekte ekrana **"İstanbulAnkara"** yazısı çıkacaktır.

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

void main(void)
{
    char *p = "Ankara"
    char str[30] = "İstanbul";

    strcat(str, p);
    printf("%s\n", str);
}
```

char *strncpy (char *dest, char*source, int n);

İkinci parametresiyle belirtilen adresten başlayarak üçüncü parametresiyle belirtilen sayıda karakteri birinci parametresiyle belirtilen adresten başalarak kopyalar. `strncpy` yalnızca `strlen(source) > n` olduğu durumda NULL karakteri dizinin sonuna kopyalar. Bu fonksiyon özellikle bir karakter dizisinin yalnızca bir bölümünün değiştirlecei durumlarda kullanılır.

Aşağıdaki örnekte ekrana `abc` karakterleri çıkacaktır.

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

void main(void)
{
    char *p = "abc"
    char str[10];

    strncpy(str, p, 3);
    str[3] = '\0';
    printf("%s\n", str);
}
```

int strncmp (char *str1, char *str2, int n);

İki karakter dizisinin yalnızca ilk **n** karakterini karşılaştırır. **n** sayısı büyük bile olsa karşılaştırma dizilerden bir tanesi NULL karaktere geldiğinde sonlandırılır.

Aşağıdaki örnekte ekrana `Tamam!..` yazısı basılacaktır:

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

void main(void)
{
    char *p = "Yenişehir";
    char *str = "Yenibosna";

    if (!strncmp(p, str, 4))
        printf("Tamam!..\n");
    else
        printf("Tamam değil!..\n");
}
```

char *strncat(char *s1, char *s2, int n);

İkinci parametresiyle belirtilen karakter dizisinin ilk n karakterini birinci parametresiyle belirtilmiş olan karakter dizisinin sonuna ekler.

Aşağıdaki örnekte ekrana `EskiYeni` yazısı çıkacaktır:

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

```
void main(void)
{
    char *p = "Yenişehir";
    char str[10] = "Eski";

    strncat(str, p, 3);
    printf("%s\n", str);
}
```

char *strchr(char *str, int ch);

Birinci parametresiyle belirtilen karakter dizisi içerisinde ikinci parametresiyle belirtilen karakteri arar. Eğer bulursa bulduğu yerin adresiyle bulamazsa NULL göstericiye geri döner. Aranılan karakter dizi içerisinde birden fazla yerde varsa, `strchr` ilk bulduğu yerin adresiyle geri dönecektir.

Aşağıdaki örnek programda ekrana **kara** yazısı çıkacaktır.

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

void main(void)
{
    char *str = "Ankara";
    char *p;

    p = strchr(str, 'k');
    if (p == NULL) {
        printf("Bulamadı!..\n");
        exit(1);
    }
    printf("%s\n", p);
}
```

char *strrchr(char *str, int ch);

Birinci parametresiyle belirtilen karakter dizisi içerisinde ikinci parametresiyle belirtilen karakteri arar. Eğer bulursa bulduğu yerin adresiyle bulamazsa NULL göstericiyle geri döner. Aranılan karakter dizi içerisinde birden fazla yerde varsa, `strchr` son bulduğu yerin adresiyle geri dönecektir. `strchr` fonksiyonuyla karşılaştırınız.

Aşağıdaki örnek programda ekrana **engi** karakterleri yazılacaktır.

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

void main(void)
{
    char *str = "Kahverengi";
    char *p;
```

```
    p = strrchr(str, 'e');
    if (p == NULL) {
        printf("Bulamadı!..\n");
        exit(1);
    }
    printf("%s\n", p);
}
```

char *strstr(char *s1, char *s2);

İkinci parametresiyle belirtilen karakter dizisini birinci parametresiyle belirtilen karakter dizisi içerisinde arar. Bulursa bulduğu yerin adresiyle bulamazsa NULL göstericiyle geri döner.

Aşağıdaki örnek programda ekrana `bir denemedir` yazısı çıkacaktır.

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

void main(void)
{
    char *str = "Bu bir denemedir";
    char *p;

    p = strstr(str, "bir");
    if (p == NULL) {
        printf("Bulamadı!..\n");
        exit(1);
    }
    printf("%s\n", p);
}
```

char *strupr(char *str);

Parametresiyle belirtilen karakter dizisindeki küçük harfleri büyük harflere çevirir. Küçük harf olmayan karakterlere dokunmaz. Geri dönüş değeri parametresi olan adresin aynısıdır. Bu fonksiyonun yalnızca İngiliz alfabesindeki harflerde dönüşüm yapacağına dikkat ediniz.

Aşağıdaki programda ekrana büyük harflerle `06 - ANKARA` yazısı basılacaktır.

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

void main(void)
{
    char *str = "06 - Ankara";

    printf("%s\n", strupr(str));
}
```

char *strlwr(char *str);

Parametresiyle belirtilen karakter dizisindeki büyük harfleri küçük harflere çevirir. Büyük harf olmayan karakterlere dokunmaz. Geri dönüş değeri parametresi olan adresin aynısıdır. Bu fonksiyonun yalnızca İngiliz alfabesindeki harfler üzerinde dönüşüm yaptığını dikkat ediniz.

Aşağıdaki programda ekrana küçük harflerle **06 - ankara** yazısı basılacaktır.

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

void main(void)
{
    char *str = "06 - ANKARA";

    printf("%s\n", strlwr(str));
}
```

char *strset(char *str, int ch);

Birinci parametresiyle belirtilen karakter dizisini ikinci parametresiyle belirtilen karakterlerle doldurur. Parametresiyle belirtilen adrese geri döner.

Aşağıdaki örnek programda ekrana **xxxxxx** karakterleri yazılacaktır.

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

void main(void)
{
    char *str = "Deneme";

    printf("%s\n", strset(str, 'x'));
}
```

void *memcpy(void *s1, void *s2, unsigned n);

İkinci parametresiyle belirtilen adresten başlayarak üçüncü parametresiyle belirtilen byte kadar bilgiyi birinci parametresiyle belirtilen adresten başlayarak kopyalar. **memcpy** NULL karakteri görünce işlemini kesmez; onu da sıradan bir karakter gibi ele alır. Geri dönüş değeri birinci parametresiyle belirtilen (yani kopyalamanın yapıldığı) adresin kendisidir. Kopyalanacak alan ile kopyanın yapılacağı alanda çakışma durumu varsa **memcpy** fonksiyonunun davranışı taşınabilir değildir. Çakışma durumlarında **memcpy** yerine aynı parametrelere sahip **memmove** fonksiyonu kullanılmalıdır. **strcpy** ve **strncpy** ile karşılaştırınız.

Aşağıdaki örnek programda ekrana **İstanbul** yazısı basılacaktır.

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

```
void main(void)
{
    char *s1 = "İstanbul";
    char s2[10];

    memcpy(s2, s1, strlen(s1) + 1);
    printf("%s\n", s2);
}
```

int memcmp(char *s1, char *s2, unsigned n);

s1 ve s2 parametreleriyle belirtilmiş olan adreslerden başlayarak ilk n byte bilgiyi karşılaştırır. Eğer s1 adresinden başlayan bilgi s2 adresinden başlayan bilgiden büyükse pozitif bir değere, küçükse negatif bir değere, eşitse sıfır değerine geri döner. memcmp NULL karakteri dikkate almaz; NULL karakteri de sıradan bir karakter olarak ele alır.

Aşağıdaki örnek programda ekrana Tamam yazısı basılacaktır.

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

void main(void)
{
    char *s1 = "Yenişehir";
    char *s2 = "Yeni mahalle";

    if (!memcmp(s1, s2, 4))
        printf("Tamam\n");
    else
        printf("Tamam değil\n");
}
```

MATEMATİKSEL FONKSİYONLAR (MATH.H)

double sqrt(double val);

Parametresiyle belirtilmiş olan **double** sayının karekökünü hesaplayarak geri dönüş değeri olarak verir.

double pow (double base, double exp);

Birinci parametresiyle belirtilmiş olan sayının ikinci parametresiyle belirtilmiş olan kuvvetini hesaplayarak geri dönüş değeri olarak verir.

Aşağıdaki program 2^{10} değerini ekrana yazdırır.

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

void main(void)
{
```

```
    printf("%f\n", pow(2., 10.));
}
```

double exp (double val);

Parametresiyle belirtilmiş olan sayının **e** sayısına göre kuvvetini hesaplayarak (e^{val}) geri dönüş değeri olarak verir.

double log (double val);

Parametresi olan sayının **e** tabanına göre logaritmasını hesaplayarak ($\log_e val$) geri dönüş değeri olarak verir.

int abs(int val);
double labs(long val);
long fabs(double val);

Bu fonksiyonların hepsi MATH.H içerisinde makro olarak tanımlanmışlardır. `abs` `int` türünden bir sayının, `labs` `long` türünden bir sayının ve `fabs` ise `double` türünden bir sayının mutlak değerini hesaplar.

double sin(double a);

Parametresi ile belirtilen radyan cinsinden açı değerinin sinüsünü hesaplayarak geri dönüş değeri olarak verir.

Aşağıdaki programda 30^0 derecenin sinüsü hesaplanmaktadır. (2*PI radyan=360°)

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

void main(void)
{
    printf("%f\n", sin(3.1415 / 6));
}
```

double cos(double a);

Parametresi ile belirtilen **radyan** cinsinden açı değerinin kosinüsünü hesaplayarak geri dönüş değeri olarak verir.

double tan(double a);

Parametresi ile belirtilen **radyan** cinsinden açı değerinin tanjantını hesaplayarak geri dönüş değeri olarak verir.

MATH.H içerisinde Tanımlanmış Olan Sembolik Sabitler

Aşağıdaki sembolik sabitler sabitler bütün derleyicilere ilişkin MATH.H dosyası içerisinde aynı isimlerle tanımlanmıştır. Bu sembolik sabitlerin kullanılması okunabilirliği kolaylaştırır.

```
#define M_E             2.71828182845904523536
#define M_LOG2E         1.44269504088896340736
#define M_LOG10E        0.43429448190325182765l
#define M_LN2           0.69314718055994530941
#define M_LN10          2.30258509299404568402
#define M_PI            3.14159265358979323846
#define M_PI_2          1.57079632679489661923
#define M_PI_4          0.78539816339744830962
#define M_1_PI          0.31830988618379067154
#define M_2_PI          0.63661977236758134308
#define M_1_SQRTPI      0.56418958354775628695
#define M_2_SQRTPI      1.12837916709551257390
#define M_SQRT2         1.41421356237309504880
#define M_SQRT_2        0.70710678118654752440
```

ÇOK KULLANILAN GENEL FONKSİYONLAR (STDLIB.H)

int atoi(const char *str);

Parametresi olan adresten başlayarak alfabetik biçimde bulunan sayısal bilgiyi tamsayı biçimine dönüştürerek geri dönüş değeri olarak verir. Geri dönüş değeri `int` olduğu için 16 bit sistemlerde [-32768, +32767] arası sayılar bu fonksiyonla dönüştürülebilirler. `atoi` NULL karakter ya da ilk sayısal olmayan karakter gördüğünde işlemi sonlandırır.

Aşağıdaki programda ekrana 200 sayısı yazılır.

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

void main(void)
{
    char *str = "100abc";
    int n;

    n = atoi(str) + 100;
    printf("%d\n", n);
}
```

int atol(char *str);

Parametresi olan adresten başlayarak alfabetik biçimde bulunan sayısal bilgiyi tamsayı biçimine dönüştürerek geri dönüş değeri olarak verir. Geri dönüş değeri

long olduğu için [-2147483648, +2147483647] arası sayılar bu fonksiyonla dönüştürülebilirler. atol NULL karakter ya da ilk sayısal olmayan karakter gördüğünde işlemi sonlandırır.

Aşağıdaki programda ekrana **10000000** sayısı yazılmaktadır.

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

void main(void)
{
    char *str = "5000000";
    long n;

    n = atol(str) + 5000000;
    printf("%ld\n", n);
}
```

double atof(char *str);

Parametresi olan adresten başlayarak alfabetik biçimde bulunan gerçek sayıyı double biçimine dönüştürerek geri dönüş değeri olarak verir. atof NULL karakter gördüğünde ya da nokta dışında ilk sayısal olmayan karakter gördüğünde işlemi sonlandırır.

Aşağıdaki programda ekrana 123.567 sayısı yazılacakır.

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

void main(void)
{
    char *str = "123.456";

    printf("%f\n", atof(str));
}
```

char *itoa(int val, char *str, int radix);

Birinci parametresiyle verilmiş olan int sayıyı alfabetik biçime dönüştürerek ikinci parametresiyle belirtilen adresten başlayarak yerleştirir. Üçüncü parametresi dönüşümün hangi sayı sistemine göre yapılacağını göstermektedir. itoa fonksiyonu ikinci parametresiyle verilen adrese geri döner. Bu fonksiyon işlevsel olarak atoi (tersten okunuşuna dikkat ediniz) fonksiyonunun tersini yapmaktadır.

Aşağıdaki örnek programda ekrana **100** sayısının ikilik sistemdeki karşılığı basılacaktır.

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

```
void main(void)
{
    char str[20];

    itoa(100, str, 2);
    printf("%s\n", str);
}
```

char *ltoa(int val, char *str, int radix);

Birinci parametresiyle verilmiş olan `long` sayıyı alfabetik biçime dönüştürerek ikinci parametresiyle belirtilen adresten başlayarak yerleştirir. Üçüncü parametresi dönüşümün hangi sayı sistemine göre yapılacağını göstermektedir. `ltoa` fonksiyonu ikinci parametresiyle verilen adrese geri döner.

void exit(int status);

Programı sonlandırarak kontrolü işletim sistemine geri verir. Parametresi olan sayı programın sonlanmasıyla işletim sistemine iletilmektedir. Bu sayının işetim sisteminde alınması ve kullanılmasına ancak özel uygulamalarda rastlanmaktadır.

Aşağıdaki program parçasında dinamik bellek tahsisatı başarısız bir biçimde yapılmışsa program sonlandırılmaktadır.

```
...
p = malloc(1000);
if (p == NULL) {
    printf("Yetersiz bellek!..\n");
    exit(1);
}
...
```

int rand(void);

0 ile `RAND_MAX` arasında rastgele sayı üretir. `RAND_MAX` `stdlib.h` içerisinde tanımlanmış bir sembolik sabittir. `RAND_MAX` genellikle **unsigned** int türünün uzunluğunun yarısına tanımlanmıştır. Yani 16 bit sistemlerde `RAND_MAX`'ın değeri 32767'dir. Rassal sayı aralığını daraltmak için % operatörü ile bölümden elde edilen kalan bulunabilir.

Aşağıdaki örnekte 0 ile 9 arasında 20 tane rassal sayı üretilmektedir.

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

void main(void)
{
    int k;
    int r;
```

```
    for (k = 0; k < 20; ++k) {
        r = rand() % 10;
        printf("%d\n", r);
    }
}
```

void srand(unsigned seed);

rand fonksiyonu programın her çalıştırılmasında aynı rassal sayı dizilimini üretir. Çünkü rassal sayılar aynı başlangıç değerinden hareketle aritmetik bir dizi işlem ile üretilirler. Programın her çalışmasında farklı rassal sayı dizilimleri elde etmek için başlangıç değerlerinin de rassal olarak değişmesi gerekir. İşte srand fonksiyonu rassal sayı üretilmesinde kullanılan başlangıç değerinin belirlenmesinde kullanılmaktadır. srand ile rassal sayı üretecine rastgele bir ilkdeğer verebilmek için genellikle time fonksiyonundan faydalanılmaktadır. Bu fonksiyonun 01/01/1970 yılından çağrıldığı zamana kadar geçen saniye sayısını veren long bir değer geri döndüğünü anımsayınız.

Aşağıdaki örnekte programın her çalışmasında farklı 20 rassal sayı üretilmektedir.

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

void main(void)
{
    int k;
    int r;

    srand((unsigned) time(NULL));
    for (k = 0; k < 20; ++k) {
        r = rand() % 10;
        printf("%d\n", r);
    }
}
```

void * malloc(unsigned size);

Programın çalışma zamanı sırasında belleğin güvenli bir bölgesinde parametresiyle belirtilen sayıda byte kadar sürekli bellek tahsis etmeye çalışır. Eğer tahsis edebilirse tahsis ettiği bloğun başlangıç adresine edemezse NULL göstericiye geri döner.

Aşağıdaki program parçasında 10 tane int eleman için tahsisat yapılmakatadır:

```
...
int p;
```

```
p = malloc(sizeof(int) * 10);
if (p == NULL) {
    printf("Yetersiz bellek!..\n");
    exit(1);
}
...
```

void * calloc(unsigned n, unsigned size);

İki parametresinin çarpımı kadar byte sürekli bölgeyi tahsis etmeye çalışır. Başarı durumunda tahsis edilen bloğun başlangıç adresine başarısızlık durumunda NULL göstericiye geri döner. Genellikle birinci parametresi eleman sayısı ikinci parametresi ise elemanın uzunluğu biçiminde kullanılmaktadır. `calloc` fonksiyonu `malloc` fonksiyonundan farklı olarak tahsis edilen bloğu sıfırlamaktadır.

Aşağıdaki program parçasında 10 tane `int` türünden eleman için tahsisat yazılmaktadır.

```
...
int p;
p = calloc(10, sizeof(int));
if (p == NULL) {
    printf("Yetersiz bellek!..\n");
    exit(1);
}
...
```

void * realloc(void *p, unsigned newsize);

Daha önce `malloc` ya da `calloc` fonksiyonuyla tahsis edilmiş olan bloğu büyütmek ya da küçültmek amacıyla kullanılır. İlk parametresi daha önce tahsis edilmiş olan bloğun başlangıç adresi, ikinci parametresi ise bloğun yeni genişliğini göstermektedir. `realloc` bloğu uzatmak için onun hemen altında sürekli bölge bulamazsa belleğin başka bir yerinde toplam uzunluk kadar sürekli yer araştırır. Eğer böyle bir yer bulursa eski değerleri oraya taşır. Tahsisatın başarısızlığı durumunda `realloc` başarı durumunda uzatılmış olan bloğun başlangıç adresine başarısızlık durumunda ise NULL göstericiye geri dönmektedir. Bloğun yer değiştirebilmesi olasılığına karşı geri dönüş değerinin güncelleştirilmesi gerekir.

Aşağıdaki program parçasında 512 byte olarak tahsis edilmiş olan 1024 byte uzunluğa büyütülmüştür:

```
p = malloc(512);
...
p = realloc(p, 1024);
if (p == NULL) {
    printf("Yetersiz bellek!..\n");
    exit(1);
}
```

void free(void *ptr);

Dinamik bellek fonksiyonlarıyla daha önce tahsis edilmiş olan boşaltarak sisteme iade eder. free fonksiyonu kullanılmamışsa tahsis edilen bloklar programın sonlanmasıyla otomatik olarak boşaltılmaktadır.

```
a = a + (b = b + 2);
```

anlamına gelmiyor mu? İfade geçerlidir.

5.14.3 Virgül (,) Operatörü

İki ayrı ifadeyi tek bir ifade olarak birleştiren virgül operatörü C'nin en düşük öncelikli operatörüdür.

```
ifade1;
ifade2;
```

ile

```
ifade1, ifade2;
```

aynı işleve sahiptir.

Virgül operatörünün önce sol tarafındaki ifade sonra sağ tarafındaki ifade tam olarak yapılır. Bu operatörün ürettiği değer sol tarafındaki operandın değer üretilmesinde hiçbir etkisi yoktur. Örneğin:

```
a = (b = 10, c = 20);
/* a = 20 */
```

İşlem sırası:

```
İ1: b = 10 ⟶ 10
İ2: c = 20 ⟶ 20
İ3: İ1, İ2 ⟶ 20
İ4: a = İ2 ⟶ 20
```

Aşağıdaki örnekte `if` deyimi Yanlış olarak değerlendirilecektir:

```
if (a = 10, b = 0) {
...
}
```

virgül operatörünü birbirine benzeyen farklı ifadeleri tek bir ifade altında birleştirmek için kullanabilirsiniz. Örneğin:

```
if (ifade) {
     x1 = 1;
     x2 = 2;
     x3 = 3;
}
```

ile

```
if (ifade)
     x1 = 1, x2 = 2, x3 = 3;
```

eşdeğer işlevlere sahiptir.