İÇİNDEKİLER

PROGRAMLAMA ve C

Yazılım Nedir
Yazılım *(software),* programlama ve programlamayla ilgili konuların geneline verilen isimdir. Yazılım denince akla programlama dilleri bu diller kullanılarak yazılmış kaynak programlar ve oluşturulan çeşitli dosyalar gelir.

Donanım Nedir
Donanım (*hardware*), bilgisayarın elektronik kısmı, yapısına verilen isimdir.

Yazılımın Sınıflandırılması
Yazılım uygulama alanlarına göre çeşitli gruplara ayrılabilir:

1. Bilimsel yazılımlar ve mühendislik yazılımları
Bilimsel konularda ve mühendislik uygulamalarındaki problemlerin çözülmesinde kullanılan yazılımlardır. Bu tür yazılımlarda veri miktarı göreli olarak düşüktür ancak matematiksel ve istatistiksel algoritmalar yoğun olarak kullanılır. Böyle programlar ağırlıklı olarak hesaplamaya yönelik işlemler içerir ve bilgisayarın merkezi işlem birimini (*CPU*) yoğun bir biçimde kullanır. Elektronik devrelerin çözümünü yapan programlar, istatistik analiz paketleri, bu tür programlara örnek olarak verilebilir.

2. Uygulama yazılımları
Veri tabanı ağırlıklı yazılımlardır. Genel olarak verilerin yaratılması, işlenmesi ve dosyalarda saklanması ile ilgilidir. Bu tür programlara örnek olarak stok kontrol programları, müşteri izleme programları, muhasebe programları verilebilir.

3. Yapay zeka yazılımları
İnsanın düşünsel ya da öğrenmeye yönelik davranışlarını taklit eden yazılımlardır. Örnek olarak robot yazılımları, satranç ya da briç oynatan programlar vs. verilebilir.

4. Görüntüsel yazılımlar
Görüntüsel işlemlerin ve algoritmaların çok yoğun olarak kullanıldığı programlardır. Örnek olarak oyun ve canlandırma (*animasyon*) yazılımları verilebilir. Bu yazılımlar ağırlıklı olarak bilgisayarın grafik arabirimini kullanır.

5. Simülasyon yazılımları
Bir sistemi bilgisayar ortamında simüle etmek için kullanılan yazılımlardır.

6. Sistem yazılımları
Bilgisayarın elektronik yapısını yöneten yazılımlardır. Derleyiciler, haberleşme programları, işletim sistemleri birer sistem yazılımıdır. Örneğin bir metin işleme programı da bir sistem yazılımıdır. Uygulama programlarına göre daha düşük düzeyli işlem yaparlar.

Programlama Dillerinin Değerleme Ölçütleri
Kaynaklar şu an halen kullanımda olan yaklaşık 1000 - 1500 programlama dilinin varlığından söz ediyor. Neden bu kadar fazla programlama dili var? Bu kadar fazla programlama dili olmasına karşın neden halen yeni programlama dilleri tasarlanıyor? Bir programlama dilini diğerine ya da diğerlerine göre farklı kılan özellikler neler olabilir? Bir programlama dilini tanımlamak gerekirse hangi niteleyiciler kullanılabilir? Bu sorulara yanıt verebilmek için değerlendirme yapmaya olanak sağlayan ölçütler olmalıdır. Aşağıda bu ölçütler kısaca inceleniyor:

Seviye

Bir programlama dilinin seviyesi (*level)* o programlama dilinin insan algısına ya da makineye yakınlığının ölçüsüdür. Bir programlama dili insan algılasına ne kadar yakınsa o kadar yüksek seviyeli *(high level)* demektir. Yine bir programlama dili bilgisayarın elektronik yapısına ve çalışma biçimine ne kadar yakınsa o kadar düşük seviyeli *(low level)* demektir.
Bilgisayarın işlemcisinin anladığı bir "komut takımı" (*instruction set*) vardır. İşlemci yalnızca kendi komut takımı içinde yer alan komutları çalıştırabilir. Programlama dilinde yazılan metin, bazı süreçler sonunda bilgisayarın işlemcisinin komut takımında yer alan komutlara dönüştürülür.
Yüksek seviyeli dillerle çalışmak programcı açısından kolaydır, iyi bir algoritma bilgisi gerektirmez. Bu dillerde yalnızca nelerin yapılacağı programa bildirilir ama nasıl yapılacağı bildirilmez. Genel olarak programlama dilinin düzeyi yükseldikçe, o dilin öğrenilmesi ve o dilde program yazılması kolaylaşır.
Makine dili bilgisayarın doğal dilidir, bilgisayarın donanımsal tasarımına bağlıdır. Bilgisayarların geliştirilmesiyle birlikte onlara iş yaptırmak için kullanılan ilk diller, makine dilleri olmuştur. Bu yüzden makine dillerine birinci kuşak diller de denir.
Makine dilinin programlarda kullanılmasında karşılaşılan iki temel sorun vardır. Makine dilinde yazılan kodlar doğrudan makinenin işlemcisine, donanım parçalarına verilen komutlardır. Değişik bir *CPU* kullanıldığında ya da bellek düzenlemesi farklı bir şekilde yapıldığında artık program çalışmaz, programın yeniden yazılması gerekir. Çünkü makine dili yalnızca belirli bir *CPU* ya da *CPU* serisine uygulanabilir. Makine dili taşınabilir (*portable*) değildir. Diğer önemli bir sorun ise, makine dilinde kod yazmanın çok zahmetli olmasıdır. Yazmanın çok zaman alıcı ve uğraştırıcı olmasının yanı sıra yazılan programı okumak ya da algılamak da o denli zordur. Özellikle program boyutu büyüdüğünde artık makine dilinde yazılan programları geliştirmek, büyütmek, iyice karmaşık bir hale gelir.
Başlangıçta yalnızca makine dili vardı. Bu yüzden makine dilleri 1. kuşak diller olarak da isimlendirilir. Yazılımın ve donanımın tarihsel gelişimi içinde makine dilinden, insan algılamasına çok yakın yüksek seviyeli dillere (*4. kuşak diller*) kadar uzanan bir süreç söz konusudur.
1950'li yılların hemen başlarında makine dili kullanımının getirdiği sorunları ortadan kaldırmaya yönelik çalışmalar yoğunlaştı. Bu yıllarda makine dillerinde yazılan programlar bilgisayarın çok sınırlı olan belleğine yükleniyor, böyle çalıştırılıyordu. İlk önce makine dilinin algılanma ve anlaşılma zorluğunu kısmen de olsa ortadan kaldıran bir adım atıldı. Simgesel makine dilleri geliştirildi. Simgesel makine dilleri (*Assembly Languages)* makine komutlarından birkaç tanesini paketleyen bazı kısaltma sözcüklerden, komutlardan oluşuyordu. Simgesel makine dillerinin kullanımı kısa sürede yaygınlaştı. Ancak simgesel makine dillerinin makine dillerine göre çok önemli bir dezavantajı söz konusuydu. Bu dillerde yazılan programlar makine dilinde yazılan programlar gibi bilgisayarın belleğine yükleniyor ancak programın çalıştırılma aşamasında yorumlayıcı *(interpreter)* bir program yardımıyla simgesel dilin komutları, bilgisayar tarafından komut komut makine diline çevriliyor ve oluşan makine kodu çalıştırılıyordu. Yani bilgisayar, programı çalışma aşamasında önce yorumlayarak makine diline çeviriyor, daha sonra makine diline çevrilmiş komutları yürütüyordu. Programlar bu şekilde çalıştırıldığında neredeyse 30 kat yavaşlıyordu.
Bu dönemde özellikle iki yorumlayıcı program öne çıkmıştı: *John Mauchly*'nin *UNIVAC 1* için yazdığı yorumlayıcı *(1950)* ve *John Backus* tarafından *1953* yılında *IBM 701* için yazılan *"Speedcoding"* yorumlama sistemi. Bu tür yorumlayıcılar, makine koduna göre çok yavaş çalışsalar da programcıların verimlerini artırıyorlardı. Ama özellikle eski makine dili programcıları, yorumlayıcıların çok yavaş olduklarını, yalnızca makine dilinde yazılan programlara gerçek program denebileceğini söylüyorlardı.

Bu sorunun da üstesinden gelindi. O zamanlar için çok parlak kabul edilebilecek fikir şuydu: Kodun her çalıştırılmasında yazılan kod makine diline çevrileceğine, geliştirilecek bir başka program simgesel dilde yazılan kodu bir kez makine diline çevirsin ve artık program ne zaman çalıştırılmak istense, bilgisayar, yorumlama olmaksızın yalnızca makine kodunu çalıştırsın. Bu fikri *Grace Hopper* geliştirmişti. *Grace Hopper*'ın buluşuna

"compiler" derleyici ismi verildi. (*Grace Hopper* aynı zamanda *Cobol* dilini geliştiren ekipten biridir, *bug(böcek)* sözcüğünü ilk olarak *Grace Hopper* kullanmıştır.) Artık programcılar, simgesel sözcüklerden oluşan *Assembly* programlama dillerini kullanıyor, yazdıkları programlar derleyici tarafından makine koduna dönüştürülüyor ve makine kodu eski hızından bir şey yitirmeksizin tam hızla çalışıyordu. *Assembly* diller 2. kuşak diller olarak tarihte yerini aldı.
Assembly dillerinin kullanılmaya başlamasıyla bilgisayar kullanımı hızla arttı. Ancak en basit işlemlerin bile bilgisayara yaptırılması için birçok komut gerekmesi, programlama sürecini hızlandırma ve kolaylaştırma arayışlarını başlattı, bunun sonucunda da daha yüksek seviyeli programlama dilleri geliştirilmeye başlandı.
Tarihsel süreç içinde *Assembly* dillerinden daha sonra geliştirilmiş ve daha yüksek seviyeli diller 3. kuşak diller sayılır. Bu dillerin hepsi algoritmik dillerdir. Bugüne kadar geliştirilmiş olan yüzlerce yüksek seviyeli programlama dilinden pek azı bugüne kadar varlıklarını sürdürebilmiştir. 3. kuşak dillerin hemen hemen hepsi üç ana dilden türetilmiştir. 3. kuşak dillerin ilkleri olan bu üç dil halen varlıklarını sürdürmektedir:
FORTRAN dili (*FORmula TRANslator*) karmaşık matematiksel hesaplamalar gerektiren mühendislik ve bilimsel uygulamalarda kullanılmak üzere 1954 - 1957 yılları arasında *IBM* firması için *John Backus* tarafından geliştirildi. *FORTRAN* dili, yoğun matematik hesaplamaların gerektiği bilimsel uygulamalarda halen kullanılmaktadır. *FORTRAN* dilinin *FORTRAN IV* ve *FORTRAN 77* olmak üzere iki önemli sürümü vardır. Doksanlı yılların başlarında *FORTRAN - 90* isimli bir sürüm için *ISO* ve *ANSI* standartları kabul edilmiştir. *FORTRAN* dili, 3. kuşak dillerin en eskisi kabul edilir.
COBOL (COmmon Business Oriented Language) 1959 yılında, Amerika'daki bilgisayar üreticileri, özel sektör ve devlet sektöründeki bilgisayar kullanıcılarından oluşan bir grup tarafından geliştirildi. *COBOL*'un geliştirilme amacı, veri yönetiminin gerektiği ticari uygulamalarda kullanılacak taşınabilir bir programlama dili kullanmaktı. *COBOL* dili de halen yaygın olarak kullanılıyor.
ALGOL (The ALGOritmick Language) 1958 yılında Avrupa'da bir konsorsiyum tarafından geliştirildi. *IBM* Firması *FORTRAN* dilini kendi donanımlarında kullanılacak ortak programlama dili olarak benimsediğinden, Avrupa'lılar da seçenek bir dil geliştirmek istemişlerdi. *ALGOL* dilinde geliştirilen birçok tasarım özelliği, modern programlama dillerinin hepsinde kullanılmaktadır.
60'lı yılların başlarında programlama dilleri üzerinde yapılan çalışmalar yapısal programlama kavramını gündeme getirdi. *PASCAL* dili *1971* yılında akademik çevrelere yapısal programlama kavramını tanıtmak için *Profesör Niclaus Wirth* tarafından geliştirildi. Dilin yaratıcısı, dile matematikçi ve filozof *Blaise Pascal*'ın ismini vermiştir. Bu dil, kısa zaman içinde üniversitelerde kullanılan programlama dili durumuna geldi.
Pascal dilinin ticari ve endüstriyel uygulamaları desteklemek için sahip olması gereken bir takım özelliklerden yoksun olması, bu dilin kullanımını kısıtlamıştır. *Modula* ve *Modula-2* dilleri *Pascal* dili temel alınarak geliştirilmiştir.
BASIC dili *1960*'lı yılların ortalarında *John Kemeney* ve *Thomas Kurtz* tarafından geliştirildi. *BASIC* isminin *"Beginner's All Purpose Symbolic Instruction Code"* sözcüklerinin baş harflerinden oluşturulduğu söylenir. Yüksek seviyeli dillerin en eski ve en basit olanlarından biridir. Tüm basitliğine karşın, birçok ticari uygulamada kullanılmıştır. *BASIC* dili de *ANSI* tarafından standartlaştırılmıştır. Ancak *BASIC* dilinin ek özellikler içeren sürümleri söz konusudur. Örneğin *Microsoft* firmasının çıkarttığı *Visual Basic* diline nesne yönelimli programlamaya ilişkin birçok özellik eklendi. Daha sonra bu dil Visual Basic dot Net ismini aldı. Ayrıca *BASIC* dilinin bazı sürümleri uygulama programlarında -örneğin *MS Excel* ve *MS Word* programlarında- kullanıcının özelleştirme ve otomatikleştirme amacıyla yazacağı makroların yazılmasında kullanılan programlama dili olarak da genel kabul gördü.
ADA dili ise *Amerikan Savunma Departmanı (Department of Defence -DoD)* desteği ile 70'li yıllardan başlanarak geliştirildi. *DoD,* dünyadaki en büyük bilgisayar kullanıcılarından biridir. Bu kurum farklı yazılımsal gereksinimleri karşılamak için çok sayıda farklı programlama dili kullanıyordu ve tüm gereksinmelerini karşılayacak bir dil arayışına girdi. Dilin tasarlanması amacıyla uluslararası bir yarışma düzenledi. Yarışmayı kazanan şirket (*CII-Honeywell Bull of France*) *Pascal* dilini temel alarak başlattığı

çalışmalarının sonucunda *Ada* dilini geliştirdi. *Ada* dilinin dokümanları 1983 yılında yayımlandı. *Ada* ismi, düşünür *Lord Byron*'un kızı olan *Lady Ada Lovelace*'ın ismine göndermedir. *Ada Lovelace* delikli kartları hesap makinelerinde ilk olarak kullanılan *Charles Babbage*'in yardımcısıydı. *Charles Babbage* hayatı boyunca *"Fark Makinesi" (Difference Engine)* ve *"Analitik Makine" (Analytical Engine)* isimli makinelerin yapımı üzerinde çalıştı ama bu projelerini gerçekleştiremeden öldü. Yine de geliştirdiği tasarımlar modern bilgisayarların atası kabul edilir. *Ada Lovelace, Charles Babbage*'ın makinesi için delikli kartları ve kullanılacak algoritmaları hazırlıyordu. *Lovelace*'in 1800'lü yılların başında ilk bilgisayar programını yazdığı kabul edilir. *Ada* genel amaçlı bir dildir, ticari uygulamalardan roketlerin yönlendirilmesine kadar birçok farklı alanda kullanılmaktadır. Dilin önemli özelliklerinden biri, gerçek zaman uygulamalarına (*real-time applications / embedded systems*) destek vermesidir. Başka bir özelliği de yüksek modüler yapısı nedeniyle büyük programların yazımını kolaylaştırmasıdır. Ancak büyük, karmaşık derleyicilere gereksinim duyması; *C*, *Modula-2* ve *C++* dillerine karşı rekabetini zorlaştırmıştır.
Çok yüksek seviyeli ve genellikle algoritmik yapı içermeyen programların görsel bir ortamda yazıldığı diller ise 4. kuşak diller olarak isimlendirilirler. Genellikle *4GL* (*fourth generation language*) olarak kısaltılırlar. İnsan algısına en yakın dillerdir. RPG dili 4. kuşak dillerin ilki olarak kabul edilebilir. Özellikle küçük *IBM* makinelerinin kullanıcıları olan şirketlerin, rapor üretimi için kolay bir dil istemeleri üzerine *IBM* firması tarafından geliştirilmiştir.

Programlama dilleri düzeylerine göre bazı gruplara ayrılabilir:

Çok yüksek düzeyli diller ya da görsel diller ya da ortamlar (*visual languages*):
Access, Foxpro, Paradox, Xbase, Visual Basic, Oracle Forms.

Yüksek düzeyli diller.
Fortran, Pascal, Basic, Cobol.

Orta düzeyli programlama dilleri:
Ada, C. (Orta seviyeli diller daha az kayıpla makine diline çevrilebildiğinden daha hızlı çalışır.)

Düşük düzeyli programlama dilleri:
Simgesel makine dili (*Assembly language*).

Makine dili:
En aşağı seviyeli programlama dili. Saf makine dili tamamen 1 ve 0 lardan oluşur.

Okunabilirlik

Okunabilirlik *(readability)* kaynak kodun çabuk ve iyi bir biçimde algılanabilmesi anlamına gelen bir terimdir. Kaynak kodun okunabilirliği söz konusu olduğunda sorumluluk büyük ölçüde programı yazan programcıdadır. Fakat yine verimlilikte olduğu gibi dillerin bir kısmında okunabilirliği güçlendiren yapı ve araçlar bulunduğu için bu özellik bir ölçüde programlama dilinin tasarımına da bağlıdır. En iyi program kodu, sanıldığı gibi "en zekice yazılmış fakat kimsenin anlayamayacağı" kod değildir. Birçok durumda iyi programcılar okunabilirliği hiçbir şeye feda etmek istemezler. Çünkü okunabilir bir program kolay algılanabilme özelliğinden dolayı yıllar sonra bile güncelleştirmeye olanak sağlar. Birçok programcının ortak kodlar üzerinde çalıştığı geniş kapsamlı projelerde okunabilirlik daha da önem kazanır.
C'de okunabilirlik en fazla vurgulanan kavramlardan biridir. İlerideki birçok bölümde okunabilirlik konusuna sık sık değinildiğini göreceksiniz.

Taşınabilirlik

Taşınabilirlik (*portability*) bir sistem için yazılmış olan kaynak kodun başka bir sisteme götürüldüğünde, hatasız bir biçimde derlenerek, doğru bir şekilde çalıştırılabilmesi demektir.
Taşınabilirlik standardizasyon anlamına da gelir. Programlama dilleri *(ISO International Standard Organization)* ve *ANSI (American National Standard Institute)* tarafından standardize edilirler. İlk olarak 1989 yılında standartları oluşturulan *C* Dili, diğer programlama dillerinden daha taşınabilir bir programlama dilidir.

Verimlilik

Verimlilik (*Efficiency*) bir programın hızlı çalışması ve daha az bellek kullanma özelliğidir. Programın çalışma hızı ve kullandığı bellek miktarı pek çok etkene bağlıdır. Şüphesiz kullanılan algoritmanın da hız ve kullanılan bellek üzerinde etkisi vardır. Programın çalıştırıldığı bilgisayarın da doğal olarak hız üzerinde etkisi vardır. Verimlilik bir programlama dilinde yazılmış bir programın çalıştırıldığında kullandığı bellek alanı ve çalışma hızı ile ilgili bir kıstas olarak ele alınabilir. Verimlilik üzerinde rol oynayabilecek diğer etkenler sabit bırakıldığında, kullanılan programlama dilinin tasarımının da verim üzerinde etkili olduğu söylenebilir. Bu açıdan bakıldığında C verimli bir dildir.

Kullanım Alanı

Bazı diller özel bir uygulama alanı için tasarlanırlar. Sistem programlama Yapay zeka uygulamaları, simülasyon uygulamaları, veritabanı sorgulamaları, oyun programlarının yazımı amacıyla tasarlanan ve kullanılan programlama dilleri vardır. Bazı diller ise daha geniş bir kullanım alanına sahiptir. Örneğin veritabanı sorgulamalarında kullanılmak üzere tasarlanan bir dil mühendislik uygulamalarında da kullanılabilir.
C dili de bir sistem programlama dili olarak doğmasına karşın, güçlü yapısından dolayı kısa bir süre içinde genel amaçlı bir dil haline gelmiştir. *Oysa PASCAL*, *BASIC çok daha* genel amaçlı dillerdir.
C ana uygulama alanı "sistem programcılığı" olan bir dildir. Ancak neredeyse tüm uygulama alanları için C dilinde programlar yazılmıştır.

Uygulama Alanlarına Göre Sınıflandırma

Programlama dillerini uygulama alanlarına göre de gruplayabiliriz:

1. Bilimsel ve mühendislik uygulama dilleri:
Pascal, C, FORTRAN.
C Programlama dili üniversitelerdeki akademik çalışmalarda da yoğun olarak kullanılır.

2. Veri tabanı dilleri:
XBASE, (Foxpro, Dbase, CA-Clipper), Oracle Forms, Visual Foxpro.

3. Genel amaçlı programlama dilleri:
Pascal, C, Basic.

4. Yapay zeka dilleri:
Prolog, Lisp
.
5. Simülasyon dilleri
GPSS, Simula 67

6. Makro Dilleri *(Scripting languages)*
awk, Perl, Python, Tcl, JavaScript.

7. Sistem programlama dilleri:
Simgesel makine dilleri, *BCPL, C, C++, occam.*
Günümüzde sistem yazılımların neredeyse tamamının *C* dili ile yazıldığını söyleyebiliriz.

Örnek vermek gerekirse *UNIX* işletim sisteminin % 80'i *C* dili ile geri kalanı ise simgesel makine dili ile yazılmıştır. Bu işletim sistemi ilk olarak *BELL* Laboratuarları'nda oluşturulmuştur. Kaynak kodları gizli tutulmamış, böylece çeşitli kollardan geliştirilmesi mümkün olmuştur. Daha sonra geliştirilen *UNIX* bazlı işletim sistemi uygulamalarına değişik isimler verilmiştir.
C bilimsel ve mühendislik alanlarına kullanılabilen genel amaçlı bir sistem programlama dilidir.

Alt Programlama Yeteneği

Bir bütün olarak çözülmesi zor olan problemlerin parçalara ayrılması ve bu parçaların ayrı ayrı çözülmesinden sonra parçalar arasındaki bağlantının sağlanması programlamada sık başvurulan bir yöntemdir. Bir programlama dili buna olanak sağlayan araçlara sahipse programlama dilinin alt programlama yeteneği vardır denilebilir. Alt programlama yeteneği bir programlama dilinin, programı parçalar halinde yazmayı desteklemesi anlamına gelir.
Alt programlama *Yapısal Programlama Tekniği*'nin de ayrılmaz bir parçasıdır. Alt programlamanın getirdiği bazı önemli faydalar vardır. Alt programlar kaynak kodun küçülmesini sağlar. Çok yinelenen işlemlerin alt programlar kullanılarak yazılması çalışabilir programın kodunu küçültür. Çünkü alt programlar yalnızca bir kere, çalışabilir kod içine yazılır. Program kodu alt programın olduğu yere atlatılarak bu bölgenin defalarca çalıştırılması sağlanabilir.

Alt programlama algılamayı kolaylaştırır, okunabilirliği artırır, aynı zamanda kaynak kodun test edilmesini kolaylaştırır, kaynak kodun daha kolay güncelleştirilmesini sağlar. Alt programlamanın en önemli faydalarından biri de oluşturulan alt programların birden fazla projede kullanılabilmesidir (*reusability*).
C alt programlama yeteneği yüksek bir dildir. C'de alt programlara *işlev* (*function*) denir. İşlevler *C* dilinin yapı taşlarıdır.

Öğrenme ve Öğretme Kolaylığı

Her programlama dilini öğrenmenin ve öğrenilen programlama dilinde uygulama geliştirebilmenin zorluk derecesi aynı değildir. Genel olarak programlama dillerinin düzeyi yükseldikçe, bu programlama dilini öğrenme ve başkalarına öğretme kolaylaşır. Bugün yaygın olarak kullanılan yüksek düzeyli programlı dillerinin bu derece tutulmasının önemli bir nedeni de bu dillerin çok kolay öğrenilebilmesidir. Ancak yüksek düzeyli dilleri öğrenerek yetkin bir beceri düzeyi kazanmakta da çoğu zaman başka zorluklar vardır. Böyle diller çok sayıda hazır aracı barındırırlar. Örneğin yüksek düzeyli bir programlama ortamında, *GUI*'ye ilişkin hazır bir menü çubuğunun özelliklerini değiştirmeye yönelik onlarca seçenek sunulmuş olabilir. Bu durumda programcı, her bir seçeneğin anlamını öğrenmek durumunda kalır. Yani bazı durumlarda programlama dilinin seviyesinin yükselmesi programcı açısından bir algısal kolaylık getirmekle birlikte, programcıya özellikle hazır araçlara yönelik bir öğrenme yükü getirir.

Programlama Tekniklerine Verilen Destekler

Programlamanın tarihsel gelişim süreci içinde, bazı programlama teknikleri *(paradigmaları)* ortaya çıkmıştır. Programlamanın ilk dönemlerinde yazılan programların boyutları çok küçük olduğundan, program yazarken özel bir teknik kullanmaya pek gerek kalmıyordu. Çünkü bir program tek bir programcının her yönüyle üstesinden gelebileceği kadar küçüktü ve basitti. Bir programda değişiklik yapmanın ya da bir programa ekleme yapmanın ciddi bir maliyeti yoktu.
Bilgisayar donanımındaki teknik gelişmeler, insanların bilgisayar programlarından beklentilerinin artması, bilgisayar programların çoğunlukla görsel arayüzler kullanmaları, program boyutlarının giderek büyümesine yol açtı. Programların büyümesiyle birlikte, program yazmaya yönelik farklı teknikler, yöntemler geliştirildi. Bu tekniklere örnek olarak, "*prosedürel programlama", "modüler programlama", "nesne tabanlı programlama", "nesneye yönelmiş programlama", "türden bağımsız programlama*" verilebilir.

"Prosedürel programlama" ile "yapısal programlama" çoğu zaman aynı anlamda kullanılır. Yapısal programlama bir programlama tekniğidir. Bugün artık hemen hemen bütün programlama dilleri yapısal programlamayı az çok destekleyecek şekilde tasarlanmaktadır. Yapısal programlama fikri 1960'lı yıllarda geliştirilmiştir. Yapısal programlama tekniği dört ana ilke üzerine kuruludur:

1. Böl ve üstesinden gel (*divide and conquer*)
Tek bir bütün olarak yazılması zor olan programlar, daha küçük ve üstesinden daha kolay gelinebilecek parçalara bölünürler. Yani program, bölünebilecek küçük parçalarına ayrılır (*functional decomposition*). Bu parçalar alt program, işlev, prosedür, vs. olarak isimlendirilir. Alt programlamanın sağladığı faydalar daha önce açıklanmıştı.

2. Veri gizleme (*Data hiding*)
Yapısal programlama tekniğinde, programın diğer parçalarından ulaşılamayan, yalnızca belli bir bilinirlik alanı olan, yani kodun yalnızca belli bir kısmında kullanılacak değişkenler tanımlanabilir. Bu tür değişkenler genel olarak *"yerel değişkenler"* (*local variables*) olarak isimlendirilirler. Değişkenlerin bilinirlik alanlarının kısıtlanabilmesi hata yapma riskini azalttığı gibi, programların daha kolay değiştirilebilmesini, program parçalarının başka programlarda tekrar kullanabilmesini de sağlar. Alt programların, daha geniş şekliyle modüllerin, bir işi nasıl yaptığı bilgisi, o alt programın ya da modülün kullanıcısından gizlenir. Kullanıcı (*client*) için alt programın ya da modülün işi nasıl yaptığı değil, ne iş yaptığı önemlidir.

3. Tek giriş ve tek çıkış (*single entry single exit*)
Yapısal programlama tekniğini destekleyen dillerde her bir altprogram parçasına girmek için tek bir giriş ve tek bir çıkış mekanizması vardır. Bu araç programın yukarıdan aşağı olarak akışı ile uyum halindedir. Program parçalarına ancak tek bir noktadan girilebilir.

4. Döngüler, diğer kontrol yapıları
Yapısal programlama tekniğinde döngüler ve diğer kontrol deyimleri sıklıkla kullanılır. Artık kullanımda olan hemen hemen bütün programlama dilleri az ya da çok yapısal programlama tekniğini destekler.

Nesneye yönelimli programlama
Nesneye yönelimlilik (*object orientation*) de bir programlama tekniğidir.
Yapısal programlama tekniği 1960lı yıllarda gündeme gelmişken, nesneye yönelimli programlama tekniği 1980'li yıllarda yaygınlaşmıştır.
Bu teknik, kaynak kodların çok büyümesi sonucunda ortaya çıkan gereksinim yüzünden geliştirilmiştir. C dilinin tasarlandığı yıllarda, akla gelebilecek en büyük programların bile kaynak kodları ancak birkaç bin satırdı. Bilgisayar donanımındaki gelişmeler, kullanıcıların bilgisayar programlarından beklentilerinin artması ve grafik arayüzünün etkin olarak kullanılmasıyla, bilgisayar programlarının boyutu çok büyüdü. Kullanımda olan birçok programın büyüklüğü yüzbin satırlarla hatta milyon satırlarla ölçülmektedir.
Nesneye yönelmiş programlama tekniği, herşeyden önce büyük programların daha iyi yazılması için tasarlanmış bir tekniktir. C dilinin yaratıldığı yıllarda böyle bir tekniğin ortaya çıkması söz konusu değildi, çünkü programlar bugünkü ölçülere göre zaten çok küçüktü.
Nesneye yönelmiş programlama tekniğinde, programa ilişkin veri ile bu veriyi işleyen kod, nesne isimli bir birim altında birleştirilir. Yani bu tekniğin yapı taşları nesnelerdir. Bu tekniğin prosedürel programlamada en önemli farkı, programcının programlama dilinin düzleminde değil de doğrudan problemin kendi düzleminde düşünmesi, programı kurgulamasıdır. Bu da gerçek hayata ilişkin bir problemin yazılımda çok daha iyi modellenmesini sağlar.
Nesne yönelimli programlama tekniğinin yaygın olarak kullanılmaya başlanmasıyla birlikte birçok programlama dilinin bünyesine bu tekniğin uygulanmasını kolaylaştırıcı araçlar eklenerek, yeni sürümleri oluşturulmuştur. Örneğin C++ dili, C diline nesne yönelimli programlama tekniğini uygulayabilmek için bazı eklemelerin yapılmasıyla geliştirilmiştir.

Benzer amaçla *Pascal* diline eklemeler yapılarak *Delphi* dili, *Cobol* dilinin yenilenmesiyle *Oocobol*, *ada* dilinin yenilenmesiyle ise *ada 95* dilleri geliştirilmiştir.
Bazı programlama dilleri ise doğrudan nesneye yönelmiş programlama tekniğini destekleyecek şekilde tasarlanarak geliştirilmiştir. Böyle dillere saf nesne yönelimli diller de denir. Örneğin Java, Eiffel, C#, saf nesne yönelimli dillerdir.
Özetle, bir programlama dili hakkında sorulacak sorulardan belki de en önemlilerinden biri, o programlama dilinin belirli bir programlama tekniğini destekleyen araçlara sahip olup olmadığıdır.
C dili, var olan araçlarıyla prosedürel programlama tekniğine tam destek veren bir dildir.

Giriş / Çıkış Kolaylığı

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. Bu dillerin en tipik özelliklerini oluşturur. Fakat *C* giriş çıkış kolaylığı güçlü olmayan bir dildir. *C*'de veri tabanlarının yönetimi için özel kütüphanelerin kullanılması gerekir.

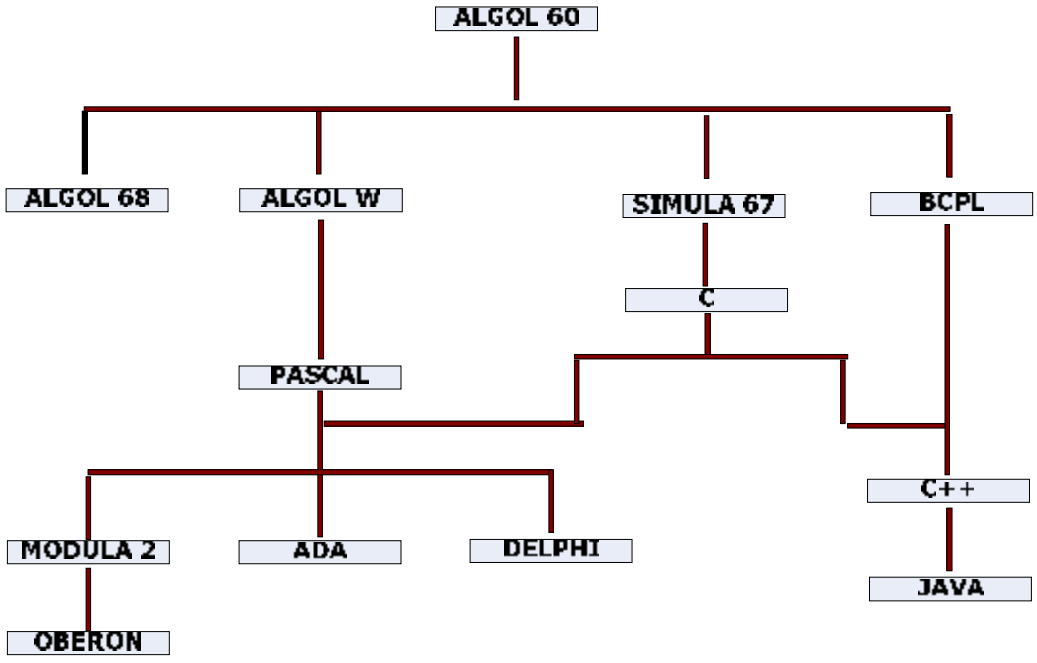
C Nasıl Bir Programlama Dilidir

İncelenen kıstaslardan sonra *C* dili belirli bir yere oturtulabilir:
C orta seviyeli bir programlama dilidir. Diğer yapısal programlama dillerine göre *C* dilinin seviyesi daha düşüktür. *C* dili hem yüksek seviyeli dillerin kontrol deyimleri, veri yapıları gibi avantajlarına sahipken hem de bitsel işleçler gibi makine kodu deyimlerini yansıtan işleçlere sahiptir. Yani *C* dili hem makinenin algısına hem de insanın algılamasına yakın bir dildir. C makineye yeterince yakındır ama programcıya da uzak değildir. Tercih edilmesinin ana nedenlerinden biri budur.
Bir programı *C* dili kullanarak yazmak, aynı programı makine dilinde yazmaya göre çok daha kolay olmasına karşın, *C*'de yazılmış bir programın verimi aynı oranda düşmez. *C* dili verim açısından bakıldığında birçok uygulama için, doğrudan makine diline tercih edilebilir. Makina dili yerine *C* dilinde programlama yapılması oluşturulan programın verimini çok düşürmez.
C dilinin ana uygulama alanı "Sistem programlama" dır. Sistem programlama ne demektir? Donanımın yönetilmesi, yönlendirilmesi ve denetimi için yazılan, doğrudan makinenin donanımla ilişkiye giren programlara sistem programı denir. Örneğin, işletim sistemleri, derleyiciler, yorumlayıcılar, aygıt sürücüleri *(device drivers*), bilgisayarların iletişimine ilişkin programlar, otomasyon programları, sistem programlarıdır.
C'den önce sistem programları *assembly* dillerle yazılıyordu. Günümüzde sistem programlarının yazılmasında *C* dilinin neredeyse tek seçenek olduğu söylenebilir. Bugün cep telefonlarından uçaklara kadar her yerde *C* kodları çalışıyor.
C algoritmik bir dildir. *C* dilinde program yazmak için yalnızca dilin sözdizimini ve anlamsal yapısını bilmek yetmez genel bir algoritma bilgisi de gerekir.
C diğer dillerle kıyaslandığında taşınabilirliği çok yüksek olan bir dildir. Çünkü 1989 yılından bu yana genel kabul görmüş standartlara sahiptir.
C ifade gücü yüksek, okunabilirlik özelliği güçlü bir dildir. C dilinde yazılan bir metnin okunabilirliğinin yüksek olması sözel bir dil olmasından, insanın kullandığı dile yakın bir dil olmasından değildir.
C çok esnek bir dildir. Diğer dillerde olduğu gibi programcıya kısıtlamalar getirmez. Makinanın olanaklarını programcıya daha iyi yansıtır.
C güçlü bir dildir, çok iyi bir biçimde tasarlanmıştır. *C*'ye ilişkin işleçlerin ve yapıların bir çoğu daha sonra başka programlama dilleri tarafından da benimsenmiştir.
C verimli bir dildir. C de yazılan programlar dilin düzeyinin düşük olması nedeniyle hızlı çalışır. Verimlilik konusunda *assembly* diller ile rekabet edebilir.
C doğal bir dildir. *C* bilgisayar sistemi ile uyum içindedir.
C küçük bir dildir. Yeni sistemler için derleyici yazmak zor değildir.
C nin standart bir kütüphanesi vardır. Bu kütüphane ile sık yapılan işlemler için ortak bir arayüz sağlanmıştır.
C'nin eğitimi diğer bilgisayar dillerine göre daha zordur.

C dili *UNIX* işletim sistemi ile bütünleşme içindedir. *UNIX* işletim sisteminde kullanılan bazı araçlar kullanıcının C dilini bildiğini varsayar.
Diğer tüm bilgisayar programlama dillerinde olduğu gibi C dilinin de zayıf tarafları vardır. Esnek ve güçlü bir dil olması programcının hata yapma riskini artırır. C dilinde yazılan kodlarda yapılan yanlışlıkların bulunması diğer dillere göre daha zor olabilir.

C Programlama Dilinin Tarihi

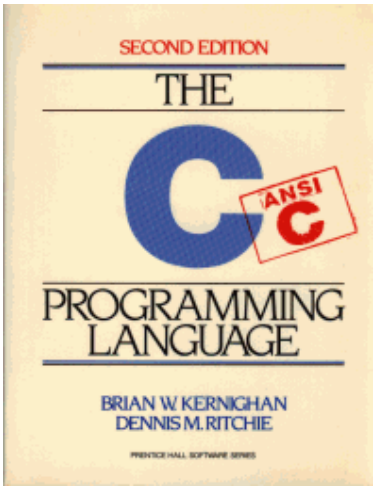
C dilinin *UNIX* işletim sisteminin bir yan ürünü olarak doğduğu söylenebilir. Önce Unix işletim sisteminin tarihine değinelim:
1965 yılında *MIT*'de *MAC* isimli bir proje gerçekleştirildi. *MAC* bir bilgisayar sisteminin zaman paylaşımını sağlayan ilk projelerden biriydi. Bu proje ile aynı bilgisayar *30* a kadar kullanıcı tarafından paylaşılabiliyordu *160* ayrı yazıcıyı kullanmak da mümkündü. Bu projenin başarısından cesaret alan *MIT*, *General Electric* ve *Bell Laboratuarları ile* bir ortak girişim oluşturararak zaman paylaşımlı yeni bir sistem oluşturma çalışmasına başladı. Bu projeye *MULTICS (Multiplexed Information and Computing Service*) ismi verildi. *Bell Laboratuarları* projenin yazılım kısmından sorumluydu. 1969 yılında *Bell Laboratuarları* proje süresinin uzaması ve proje maliyetinin yüksek olması nedeniyle projeden ayrıldı.
1969 yılında *Bell Laboratuarları*'nda *Ken Thompson* öncülüğünde bir grup yeni bir alternatif arayışına girdi. *MULTICS* projesinde çalışan *Ken Thompson* ve ekip arkadaşı *Dennis Ritchie* bu konuda bir hayli deneyim kazanmıştı.
Thompson ve ekibi, insan ve makine arasındaki iletişimi kolaylaştıracak bir işletim sistemi tasarımına girişti. İşletim sisteminin omurgası yazıldığında *MULTICS*'e gönderme yapılarak işletim sistemine *Unix* ismi verildi.
Bu yıllarda programcılar *PL/1, BCPL* gibi yüksek seviyeli dilleri kullanıyorlardı. *Thompson DEC* firmasının ana belleği yalnızca *8K* olan *PDP 7* isimli bilgisayarı üzerinde çalışıyordu. *Thompson* 1960 lı yıllarda *Martin Richards* tarafından geliştirilen *BCPL* dilini kullanmakta deneyimliydi. Kendi dilini tasarlarken *Thompson*, 1960 yıllarının ortalarında *Martin Richards* tarafından geliştirilmiş *BCPL* dilinden yola çıktı. *(BCPL = Business Common Programming Language).* Bu dil de *CPL = Cambridge Programming Language*'den türetilmiştir. *CPL*'in kaynağı da tüm zamanların en eski ve en etkili dillerinden biri olan *ALGOL 60*'dır. *ALGOL 60*, *Pascal, ADA, Modula2* dillerinin de atasıdır. Bu dillere bu yüzden C dilinin kuzenleri de diyebiliriz. Aşağıda *ALGOL 60* dil ailesi görülüyor:



Thompson geliştirdiği bu dilin ismini *B* koydu. *Dennis Ritchie, UNIX* projesine katılınca *B* dilinde programlamaya başladı. *B* dili daha da geliştirilmişti ve artık daha yeni bir teknoloji olan *Dec PDP-11* bilgisayarlarda çalışıyordu. *Thompson, UNIX* işletim sisteminin bir kısmını *B* dilinde yeniden yazdı. *1971* yılına gelindiğinde *B* dilinin *PDP-11* bilgisayarlar ve *UNIX* işletim sisteminin geliştirilmesi için çok uygun olmadığı iyice ortaya çıktı. Bu yüzden *Ritchie, B* programlama dilinin daha ileri bir sürümünü geliştirmeye başladı. Oluşturduğu dili ilk önce *NB (new B)* olarak isimlendirdi. Ama geliştirdiği dil *B* dilinden iyice kopmaya ve ayrı bir karakter göstermeye başlayınca dilin ismini de *C* olarak değiştirdi. *1973* yılında *UNIX* işletim sisteminin büyük bir kısmı *C* dili ile yeniden yazıldı.

Ken Thompson ve Dennis Ritchie Unix İşletim Sistemi üzerinde çalışırken (Yıl: 1972)

C'nin evrimi ve gelişmesi 70'li yıllarda da sürdü. Ancak uzun bir süre C dili dar bir çevrede kullanıldı. Geniş kitleler tarafından tanınması ve kullanılmaya başlaması *1978* yılında *Dennis Ritchie* ve *Brian Kernighan* tarafından yazılan *"The C Programming Language"* kitabı ile olmuştur. Bu kitap aynı zamanda yazılım konusunda yazılan en iyi eserlerden biri olarak değerlendirilmektedir. *C* standartlarının oluşturulmasına kadar olan dönemde bu kitap çoğunluğun benimsediği genel kabul gören gayri resmi (*de facto*) bir standart görevi de görmüştür. Bu kitap kısaca *K&R* (*Kernighan & Ritchie*) olarak isimlendirilmektedir.



1970'li yıllarda *C* programcılarının sayısı azdı ve bunlardan çoğu *UNIX* kullanıcılarıydı. Ama artık 80'li yıllar gelince *C*'nin kullanımı *UNIX* sınırlarını aştı, farklı işletim sistemleri için çalışan derleyiciler piyasaya çıktı. *C* dili de *IBM PC*'lerde yoğun olarak kullanılmaya başladı.
C'nin hızlı bir biçimde yaygınlaşması bazı sorunları da beraberinde getirdi. Derleyici yazan firmalar, referans olarak *Ritchie* ve *Kernighan*'ın kitabını esas alıyorlardı ama söz konusu kitapta bazı noktalar çok da ayrıntılı bir biçimde açıklanmamıştı. Özellikle hangi noktaların *C* dilinin bir özelliği hangi noktaların ise *UNIX* işletim sisteminin bir özelliği olduğu o kadar açık olmadığı için bir takım karışıklıklar ortaya çıkıyordu. Böylece derleyici yazanların ürünlerinde de farklılıklar ortaya çıkıyordu. Ayrıca kitabın yayınlanmasından sonra da dilde bir takım geliştirmeler, iyileştirmeler, değişiklikler yapıldığı için, birbirinden çok farklı derleyiciler piyasada kullanılmaya başlanmıştı.

Hangi C

C tek bir programlama dili olmasına karşın C'nin farklı sürümlerinden söz etmek olasıdır:

Geleneksel C

C dili ilk olarak 1978 yılında yayımlanan *Dennis Ritchie* ve *Brian Kernighan* tarafından yazılmış "*The C Programming Language*" isimli kitapta anlatılmıştı. 1980'li yıllarda C dili bu kitapta anlatılan genel geçer kurallara göre kullanıldı, derleyici yazan firmalar bu kuralları esas aldılar. Bu dönem içinde kullanılan derleyiciler arasında bazı kural ve yorum farklılıkları olsa da ana kurallar üzerinde bir genel bir uzlaşma olduğu söylenebilir. C dilinin standartlaştırılması sürecine kadar kullanılan bu sürüme "*Geleneksel C (Traditional*

C)" ya da "*Klasik C (Classic C)*" denmektedir. Şüphesiz C dilinin bu döneminde derleyici yazan firmalar uzlaşılmış kuralların dışında kendi özel eklentilerini de dile katmaktaydılar.

Standart C (1989)

C dilinin standart hale getirilmesinin dilin ticari kullanımının yaygınlaşmasına yardımcı olacağını düşünen *Amerikan Ulusal Standartlar Enstitüsü (ANSI)* C dili ve kütüphanesi için bir standart oluşturması amacıyla *1982* yılında bir komite oluşturdu. *Jim Brody* başkanlığında çalışan ve *X3J11* numarasıyla anılan bu komitenin oluşturduğu standartlar *1989* yılında kabul edilerek onaylandı. Bu standardın resmi ismi *American National Standard X3.159-1989* olmakla birlikte kısaca *ANSI C* diye anılmaktadır.
ANSI C çalışması tamamlandığında bu kez uluslararası bir standart oluşturmak amacıyla *P.J.Plaguer* başkanlığında oluşturulan bir grup (*ISO/IEC JTC1/SC22/WG14*) *ANSI* standartlarını uluslararası bir standarda dönüştürdü. Bazı küçük biçimsel değişiklikler yapılarak oluşturulan bu standardın resmi ismi *ISO/IEC 9899:1990'* dır. Bu standart daha sonra *ANSI* tarafından da kabul edilmiştir. Ortak kabul gören bu standarda *Standard C (1989)* ya da kısaca *C89* denmektedir.
Geleneksel C' den *C89'* a geçişte önemli değişiklikler olmuştur.

Standart C (1995)

1995 yılında *Wg14* komitesi *C89* standartları üzerinde iki teknik düzeltme ve bir eklenti yayımladı. Bu düzeltme notları ve eklentiler çok önemli değişiklikler olarak görülmemektedir. Bu düzeltmelerle değiştirilmiş standartlara "*C89 with amendment 1*" ya da kısaca *C95* denmektedir

Standart C (1999)

ISO/IEC standartları belirli aralıklarla gözden geçirilmekte, güncellenmektedir. 1995 yılında *Wg14* kurulu *C* standartları üzerinde kapsamlı değişiklikler ve eklentiler yapmak üzere bir çalışma başlattı. Çalışma 1999 yılında tamamlandı. Oluşturulan yeni standardın resmi ismi *ISO/IEC 9899:1999*'dur. Kısaca *C99* olarak anılmaktadır. Bu standart resmi olarak daha önceki tüm sürümlerin yerine geçmiştir. Derleyici yazan firmalar derleyicilerini yavaş yavaş da olsa bu standarda uygun hale getirmektedirler. *C99* standartları *C89/C95*'e birçok eklenti sağlamıştır. Bunlardan bazıları

- Sanal sayıların doğal tür olarak eklenmesi
- Daha büyük bir tamsayı türünün eklenmesi
- Değişken uzunlukta diziler
- Boolean (Mantıksal) veri türü
- İngilizce olmayan karakter setleri için daha kapsamlı destek
- Gerçek sayı türleri için daha iyi destek
- C++ tarzı açıklama satırları
- satır içi (*inline*) işlevler

C99 ile getirilen değişiklikler ve eklentiler *C95*'e göre daha fazla ve önemlidir. Ancak C dilinin doğal yapısı değiştirilmemiştir.

C++ Dili

C++ dili de 1980 li yılların başlarında Bjarne Stroustrup tarafından *AT&T Bell* Laboratuarları'nda geliştirildi. *C++* dili *C*'den ayrı, tamamıyla başka bir programlama dilidir. Dilin tasarımcısı *Bjarne Stroustoup*, *C*'nin orta seviyeli özelliklerine bazı eklemeler yaparak, başta nesne yönelimli programlama tekniği olmak üzere başka programlama tekniklerini de destekleyen ayrı bir dil oluşturdu.
C++ dili de dinamik bir gelişme süreci sonunda 1998 yılında standart haline getirildi. Oluşturulan resmi standardın ismi *ISO/IEC 14882:1998*'dir.
C++ birden fazla programlama tekniğini destekleyen *(multi-paradigm*) bir dildir. Temel sözdizimi büyük ölçüde *C* dilinden alınmıştır. Ancak sözdizim özellikleri birbirine benzese

de dilin araçlarının kullanılma biçimi *C*'den oldukça farklıdır. C++, C ile karşılaştırıldığında çok daha büyük ve karmaşık bir dildir.
C++, C dilini de kapsayan bir üst dil olarak düşünülebilir. Eğer bazı noktalara dikkat edilirse hem C dilinde hem de C++ dilinde geçerli olabilecek programlar yazılabilir. C dilinin böyle kullanımına "*clean* C" ("Temiz C") denmektedir.

Hangi C'yi Kullanmalı

Bu karar verilirken, uygulamanın geliştirileceği alan için nasıl bir C derleyicisinin olduğu şüphesiz önemlidir. Karar verilmesinde bir diğer etken de yazılan kodun ne seviyede taşınabilir olmasının istendiğidir. Seçeneklerin aşağıdakiler olduğu düşünülebilir:

1. *C99*. C dilinin son sürümü. C dilinin tüm özelliklerini içermektedir. Ancak bazı derleyiciler bu sürümü henüz desteklememektedir.

2. *C89*. En fazla kullanılan sürümdür. Bu sürüm kullanıldığında genellikle *C95* eklentileri de kullanılmaktadır.

3. Geleneksel C. Yeni programların yazılmasında artık kullanılmasa da, eskiden yazılan programların bakımında kullanmak gerekebilir.

4. *Temiz C (Clean C)*. Yani C dilinin aynı zamanda C++ diline uyumlu olarak kullanılması.

C99 sürümü, *C89* ve *Klasik C* ile genel olarak yukarıya doğru uyumludur. Yani *Klasik C* programları ve *C89* programları, *C99* dilinin kurallarına göre derlendiğinde ya değişiklik yapmak gerekmez ya da çok az değişiklik yapmak gerekir. Bu değişiklikler için önişlemci programı (*preprocessor*) kullanılabilir.
Geçmişe doğru uyumlu program yazmaksa çoğu zaman mümkün değildir.

Objective C - Concurrent C

C dilinin kullanımı yaygınlaştıktan sonra C dilinden başka diller de türetilmiştir:
Objective C dili, *NeXT* firması tarafından *OpenStep* işletim sistemi için uyarlanmıştır. *NeXT* firması daha sonra *Apple* firması tarafından satın alınmıştır. *Objective C* dilinin *Apple* firmasının yeni kuşak işletim sistemi olan *Rhapasody* için temel geliştirme aracı olması planlanmaktadır.
"*Concurrent C*" dili, *Bell Laboratuarları*'ndan *Dr. Narain Gehani* ve *Dr. William D.Roome* tarafından geliştirilmiştir. Daha çok, paralel programlamada kullanılmaktadır.

SAYI SİSTEMLERİ

Günlük hayatta onluk sayı sistemi kullanılır. Onluk sayı sisteminde bir sayının değeri aslında her bir basamak değerinin *10* sayısının üsleriyle çarpımlarından elde edilen toplam değeridir. Örneğin:

```
1273 = (3 * 1) + (7 * 10 ) + (2 * 100) + (1 * 1000)
```

Ancak bilgisayar sistemlerinde bütün bilgiler ikilik sayı sisteminde *(binary system)* ifade edilir.
Genel olarak sayı sistemi kaçlıksa o sayı sisteminde o kadar simge bulunur.
Örneğin onluk sayı sisteminde *10* adet simge vardır:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Aynı şekilde ikilik sayı sisteminde yalnızca iki adet simge bulunur. Yani yalnızca *0* ve *1*.

Birimler

İkilik sistemde her bir basamağa (*digit*) *1* bit denir. Bit kelimesi *binary digit* sözcüklerinden türetilmiştir.
Örneğin *1011* sayısı *4* bittir yani *4* bit uzunluğundadır.
11011001 sayısı *8* bittir.

8 bitlik bir büyüklük bir *byte* olarak isimlendirilir.
Kilo, büyüklük olarak *1000* kat anlamına gelir. Ancak bilgisayar alanında Kilo, *2*'nin *1000*'e en yakın üssü olan 2^{10} yani *1024* katı olarak kullanılır. Aşağıda daha büyük birimlerin listesi veriliyor:

1 kilobyte	1 KB	1024 byte	2^{10} byte
1 megabyte	1 MB	1024 KB	2^{20} byte
1 gigabyte	1 GB	1024 MB	2^{30} byte
1 terabyte	1 TB	1024 GB	2^{40} byte
1 petabyte	1 PB	1024 TB	2^{50} byte
1 exabyte	1 EB	1024 PB	2^{60} byte
1 zettabyte	1 ZB	1024 EB	2^{70} byte
1 yottabyte	1 YB	1024 ZB	2^{80} byte

Özellikle sistem programcılığında daha küçük birimlere de isim verilir:

4 bit	1 Nybble
8 bit	1 *byte*
16 bit	1 word
32 bit	1 double word
64 bit	1 quadro word

İkilik Sayı Sistemine İlişkin Genel İşlemler

Tamsayı değerlerini göstermek için ikilik sayı sistemi iki farklı biçimde kullanılabilir. Eğer yalnızca sıfır ve pozitif tamsayılar gösteriliyorsa, bu biçimde kullanılan ikilik sayı sistemine *işaretsiz (unsigned)* ikilik sayı sistemi denir. İkilik sayı sisteminin negatif tamsayıları da gösterecek biçimde kullanılmasına, *işaretli (signed)* ikilik sayı sistemi denir.

İkilik Sayı Sisteminden Onluk Sayı Sistemine Dönüşüm

İkilik sayı sisteminde ifade edilen bir sayının onluk sayı sistemindeki karşılığını hesaplamak için en sağdan başlanarak bütün basamaklar tek tek ikinin artan üsleriyle çarpılır. Örneğin:

```
1 0 1 1 = 1 * 2^0  + 1 * 2^1  + 0 * 2^2  + 1 * 2^3  = 11
0010 1001 = (1 * 1) + (1 * 8) + (1 * 32) = 41
```

MSD ve LSD

İkilik sayı sisteminde yazılan bir sayının en solundaki bit, yukarıdaki örnekten de görüldüğü gibi en yüksek sayısal değeri katar. Bu bite *en yüksek anlamlı bit (most significant digit)* denir. Bu bit için çoğunlukla "*MSD*" kısaltması kullanılır.

İkilik sayı sisteminde yazılan bir sayının en sağındaki bit, yine yukarıdaki örnekten de görüldüğü gibi en düşük sayısal değeri katar. Bu bite *en düşük anlamlı bit (least significant digit)* denir. Bu bit için çoğunlukla *LSD* kısaltması kullanılır. Örnek:

```
0101 1101 sayısı için
MSD = 0
LSD = 1
```

İkilik sayı sisteminde yazılan bir sayının belirli bir bitinden söz edildiğinde, hangi bitten söz edildiğinin doğru bir şekilde anlaşılması için basamaklar numaralandırılır.
8 bitlik bir sayı için, sayının en sağındaki bit yani (*LSD*) sayının *0*. bitidir. Sayının en *solundaki bit* (yani *MSD*) sayının *7*. bitidir.

Onluk Sayı Sisteminden İkilik Sayı Sistemine Dönüşüm

Sayı sürekli olarak ikiye bölünür. Her bölümden kalan değer (*yani 1 ya da 0*) oluşturulacak sayının sıfırıncı bitinden başlanarak, basamaklarını oluşturur. Bu işlem *0* değeri elde edilinceye kadar sürdürülür.
Örnek olarak *87* sayısı ikilik sayı sisteminde ifade edilmek istensin:

```
87 / 2 = 43 (kalan 1) Sayının 0. biti 1
43 / 2 = 21 (kalan 1) Sayının 1. biti 1
21 / 2 = 10 (kalan 1) Sayının 2. biti 1
10 / 2 = 5 (kalan 0) Sayının 3. biti 0
5 / 2 = 2 (kalan 1) Sayının 4. biti 1
2 / 2 = 1 (kalan 0) Sayının 5. biti 0
1 / 2 = 0 (kalan 1) Sayının 6. biti 1

87 = 0101 0111
```

İkinci bir yöntem ise, onluk sayı sisteminde ifade edilen sayıdan sürekli olarak ikinin en büyük üssünü çıkarmaktır. Çıkarılan her bir üs için ilgili basamağa *1* değeri yazılır. Bu işlem *0* sayısı elde edilene kadar sürdürülür.
Yine *87* sayısı ikilik sayı sisteminde ifade edilmek istensin:

```
87 - 64 = 23        0100 0000
23 - 16 = 7         0101 0000
7 - 4   = 3         0101 0100
3 - 2   = 1         0101 0110
1 - 1   = 0         0101 0111
```

```
87 = 0101 0111
```

Bire Tümleyen

Bire tümleyen (*one's complement*) sayının tüm bitlerinin tersinin alınmasıyla elde edilen sayıdır. Yani bire tümleyen, sayıdaki *1* bitlerinin *0, 0* bitlerinin *1* yapılmasıyla elde edilir. Bir sayının bire tümleyeninin bire tümleyeni sayının yine kendisidir.

İkiye Tümleyen

Bir sayının bire tümleyeninin *1* fazlası sayının ikiye tümleyenidir (*two's complement*).

Yani önce sayının bire tümleyeni yukarıdaki gibi bulunur. Daha sonra elde edilen sayıya *1* eklenirse sayının ikiye tümleyeni bulunmuş olur.
İkiye tümleyeni bulmak için daha kısa bir yol daha vardır:
Sayının en sağından başlayarak ilk kez *1* biti görene kadar -ilk görülen *1* biti dahil- sayının aynısı yazılır, daha sonraki tüm basamaklar için basamağın tersi (*Yani 1 için 0, 0 için 1*) yazılır. Örneğin:

```
1110 0100 sayısının ikiye tümleyeni  0001 1100 dır.
0101 1000 sayısının ikiye tümleyeni  1010 1000 dır.
```

Bir sayının ikiye tümleyeninin ikiye tümleyeni sayının kendisidir.

8 Bitlik Alana Yazılabilen En Küçük ve En Büyük Değer

8 bitlik bir alana yazılacak en büyük tam sayı tüm bitleri *1* olan sayıdır:

```
1111 1111 = 255
```

8 bitlik bir alana yazılacak en küçük tam sayı tüm bitleri *0* olan sayı, yani 0'dır:

```
0000 0000 = 0
```

Negatif Sayıların Gösterimi

Negatif tamsayıların da ifade edildiği ikilik sayı sistemine "işaretli ikilik sayı sistemi" *(signed binary system)* denir.
İşaretli ikilik sayı sisteminde negatif değerleri göstermek için hemen hemen tüm bilgisayar sistemlerinde aşağıdaki yöntem uygulanır:
Sayının en soldaki biti *işaret biti* (*sign bit*) olarak kabul edilir. İşaret biti *1* ise sayı negatif, *0* ise sayı pozitif olarak değerlendirilir. İkilik sistemde bir negatif sayı, aynı değerdeki pozitif sayının ikiye tümleyenidir. Örnek olarak, ikilik sistemde yazılan *–27* sayısı yine ikilik sistemde yazılan *27* sayısının ikiye tümleyenidir.

Pozitif olan sayıların değeri, tıpkı işaretsiz sayı sisteminde olduğu gibi elde edilir:
Örneğin, *0001 1110* işaretli ikilik sayı sisteminde pozitif bir sayıdır. Onluk sayı sisteminde *30* sayısına eşittir.

Negatif tamsayıların değeri ancak bir dönüşümle elde edilebilir.
Sayının değerini hesaplamak için ilk önce sayının ikiye tümleyeni bulunur. Bu sayının hangi pozitif değer olduğu bulunur. Elde edilmek istenen sayı, bulunan pozitif sayı ile aynı değerdeki negatif sayıdır.

Örneğin, *1001 1101* sayısının onluk sayı sisteminde hangi sayıya karşılık geldiğini bulalım:
İşaret biti 1 olduğuna göre sayı negatiftir.
1001 1101 sayısının ikiye tümleyeni *0110 0011*, yani *99* değeridir.
O zaman *1001 1101 sayısı -99*'dur.

Onluk sayı sisteminde ifade edilen negatif sayıların işaretli ikilik sayı sisteminde yazılması için aşağıdaki yol izlenebilir:
Önce sayının aynı değerli fakat pozitif olanı ikilik sistemde ifade edilir. Daha sonra yazılan sayının ikiye tümleyeni alınarak, yazmak istenilen sayı elde edilir.
Örneğin, ikilik sisteminde *–17* değerini yazalım:

```
17 = 0001 0001
```

Bu sayının ikiye tümleyeni alınırsa

```
1110 1111
```

sayısı elde edilir.
Sayı değeri aynı olan negatif ve pozitif sayılar birbirinin ikiye tümleyenidir.
Bu sayıların ikilik sayı sisteminde toplamları *0*'dır.

İşaretli ikilik sayı sisteminde *1 byte*'lık alana yazılabilecek en büyük ve en küçük sayılar ne olabilir?
En büyük sayı kolayca hesaplanabilir. Sayının pozitif olması için, işaret bitinin *0* olması ve sayı değerini en büyük hale getirmek için diğer bütün bitlerinin *1* olması gerekir, değil mi? Bu sayı *0111 1111*'dır. Bu sayının onluk sayı sistemindeki karşılığı *127*'dir. Peki ya en küçük negatif sayı kaçtır, nasıl ifade edilir?
0111 1111 sayısının ikiye tümleyenini alındığında *–127* sayısı elde edilir.

```
1000 0001 (- 127)
```

Bu sayıdan hala *1* çıkartılabilir:

```
1000 0000 (-128)
```

1000 0000, 1 byte alana yazılabilecek en küçük negatif sayıdır.

1 byte alana yazılabilecek en büyük sayı sınırı aşıldığında negatif bölgeye geçilir.
En büyük pozitif tamsayıya *1* toplandığını düşünelim:

```
0111 1111       (en büyük pozitif tamsayı = 127)
        1
1000 0000       (en küçük tamsayı = -128)
```

İşaretli ikilik sistemde *n byte* alana yazılabilecek en büyük tamsayıya *1* eklendiğinde *n byte* alana yazılabilecek en küçük tamsayı elde edilir.

n *byte* alana yazılabilecek en küçük tamsayıdan *1* çıkarıldığında da *n byte* alana yazılabilecek en büyük tamsayı elde edilir.
Yukarıda açıklamalara göre -1 sayısının işaretli ikilik sayı sisteminde *8* bitlik bir alanda aşağıdaki şekilde ifade edilir:

```
-1 = 1111 1111
```

Yani işaretli ikilik sayı sisteminde tüm bitleri *1* olan sayı -1'dir.

Onaltılık ve Sekizlik Sayı Sistemleri

Onaltılık sayı sisteminde *(hexadecimal system)* sayılar daha yoğun olarak kodlanıp kullanabilir.Onaltılık sayı sisteminde *16* simge bulunur.
İlk *10* simge onluk sistemde kullanılanlarla aynıdır:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Daha sonraki simgeler için alfabenin ilk 6 harfi kullanılır:

```
A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
```

Onaltılık sayı sisteminde yazılmış bir sayıyı onluk sistemde ifade etmek için, en sağdan başlanarak basamak değerleri onaltının artan üsleriyle çarpılır:

```
01AF  = (15 * 1)  + (10 * 16) + (1 * 256) + (0  * 4096) = 431
```

Onluk sayı sisteminde yazılmış bir sayıyı, onaltılık sayı sisteminde ifade etmek için onluk sayı sisteminden ikilik sayı sistemine yapılan dönüşüme benzer yöntem kullanılabilir. Sayı sürekli *16*'ya bölünerek, kalanlar soldan sağa doğru yazılır.

Uygulamalarda onaltılık sayı sisteminin getirdiği önemli bir fayda vardır: Onaltılık sayı sistemi ile ikilik sayı sistemi arasındaki dönüşümler kolay bir biçimde yapılabilir:

Onaltılık sayı sistemindeki her bir basamak ikilik sayı sisteminde *4* bitlik (*1 Nibble*) alanda ifade edilebilir:

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

Örnek olarak, *2ADF* sayısını ikilik sayı sisteminde ifade edilmek istensin:

```
2  =  0010
A  =  1010
D  =  1101
F  =  1111
```

Bu durumda

```
2ADF = 0010 1010 1101 1111
```

İkilik sayı sisteminden onaltılık sayı sistemine de benzer şekilde dönüşüm yapılabilir: Önce bitler sağdan başlayarak dörder dörder ayrılır. En son dört bit eksik kalırsa sıfır ile tamamlanır. Sonra her bir dörtlük grup için doğrudan onaltılık sayı sistemindeki karşılığı yazılır:

```
1010 1110 1011 0001 = AEB1
0010 1101 0011 1110 = 2D3E
```

Onaltılık sayı sisteminde yazılmış işaretli bir sayının pozitif mi negatif mi olduğunu anlamak için sayının en soldaki basamağına bakmak yeterlidir. Bu basamak *[8 - F]* aralığında ise sayı negatiftir.

Aşağıda bazı önemli işaretli tamsayılar, ikilik, onaltılık sayı sistemlerinde ifade ediliyor:

İkilik sayı sistemi	onaltılık sayı sistemi	onluk sayı sistemi
0000 0000 0000 0000	0000	0
0000 0000 0000 0001	0001	1
0111 1111 1111 1111	7FFF	32767
1000 0000 0000 0000	8000	-32768
1111 1111 1111 1111	FFFF	-1

Sekizlik Sayı Sistemi

Daha az kullanılan bir sayı sistemidir. Sekizlik sayı sisteminde *8* adet simge vardır:

```
0 1 2 3 4 5 6 7
```

Sekizlik sayı sisteminin her bir basamağı, ikilik sayı sisteminde *3* bit ile ifade edilir.

```
001     1
010     2
011     3
100     4
101     5
110     6
111     7
```

Sekizlik sayı sisteminin kullanılma nedeni de, ikilik sayı sistemine göre daha yoğun bir ifade biçimine olanak vermesi, ikilik sayı sistemiyle sekizlik sayı sistemi arasında dönüşümlerin çok kolay bir biçimde yapılabilmesidir.

Gerçek Sayıların Bellekte Tutulması

Bir gerçek sayı aşağıdaki gibi ifade edilebilir:

$$x = s * b^e * \sum_{k=1}^{p} f_k * b^{-k}$$

Yukarıdaki genel denklemde

x	ifade edilecek gerçek sayı
b	Kullanılan sayı sisteminin taban değeri (Tipik olarak 2, 8, 10 ya da 16)
e	Üstel değer. Bu değer format tarafından belirlenen e_{min} ve e_{max} arasındaki bir değer
p	Ondalık kısmı belirleyen basamak sayısı
f_k	Sayı sistemindeki basamak değerleri. $0 \le f_k \le b$

Gerçek sayıların ifadesinde, gerçek sayı kabul edilmeyen bazı gösterimler söz konusudur. Bunlar:

sonsuz (*infinity*)
NaN (*not a number*)

değerleridir. Bu değerler işaretli olabilir.

Sistemlerin çoğunda, gerçek sayılar *IEEE* (*Institute of Electrical and Electronics Engineers*) *754* standardına göre tutulur. (*IEEE Standard for Binary Floating-Point Arithmetic - ISO/IEEE std 754-1985*).Bu standarda göre gerçek sayılar için iki ayrı format belirlenmiştir:

1. Tek duyarlıklı gerçek sayı formatı (single precision format)
Bu formatta gerçek sayının gösterimi genel formüle aşağıdaki biçimde uyarlanabilir:

$$x = s * 2^e * \sum_{k=1}^{24} f_k * 2^{-k}$$

Bu formatta gerçek sayı *32* bit (*4 byte*) ile ifade edilir.
32 bit üç ayrı kısma ayrılmıştır.

i. *İşaret biti (sign bit) (1 bit)*
Aşağıda *S* harfi ile gösterilmiştir.
İşaret biti *1* ise sayı negatif, işaret biti *0* ise sayı pozitiftir.

ii. *Üstel kısım (exponent) (8 bit)*
Aşağıda *E* harfleriyle gösterilmiştir.

iii. *Ondalık kısım (fraction) (23 bit)*
Aşağıda *F* harfleriyle gösterilmiştir.

```
SEEEEEEEEFFFFFFFFFFFFFFFFFFFFFFF
```

Aşağıdaki formüle göre sayının değeri hesaplanabilir :
V, gerçek sayının değeri olmak üzere:

```
EEEEEEEE = 255 ve FFF...F ≠ 0 ise V = NaN (Not a number)
```

Örnek :

```
0 11111111 00001000000100000000000 (Sayı değil)
1 11111111 00010101010001001010101 (Sayı değil)
```

```
EEEEEEEE = 255 ve FFF...F = 0 ve S = 1 ise  V = -sonsuz
EEEEEEEE = 255 ve FFF...F = 0 ve S = 0 ise  V = +sonsuz
```

```
0 < EEEEEEEE < 255 ise
V = (-1)S * 2(E - 127) * (1.FFF...F)
```

Önce sayının ondalık kısmının başına *1* eklenir. Daha sonra bu sayı $2^{(E - 127)}$ ile çarpılarak noktanın yeri ayarlanır. Noktadan sonraki kısım ikinin artan negatif üsleriyle çarpılarak elde edilir. Örnekler:

```
0 10000000 00000000000000000000000 = +1 * 2(128 - 127)  * 1.0
                                   = 2 * 1.0
                                   = 10.00
                                   = 2
```

```
0 10000001 10100000000000000000000 = +1 * 2(129 - 127) * 1.101
                                   = 2^2 * 1.101
                                   = 110.100000
                                   = 6.5
```

```
1 10000001 10100000000000000000000 = -1 * 2(129 - 127)  * 1.101
                                   = -2^2 * 1.101
                                   = 110.100000
                                   = -6.5
```

```
0 00000001 00000000000000000000000 = +1 * 2(1 - 127)  * 1.0
                                   = 2-126
```

```
EEEEEEEE = 0 ve FFF...F  ≠ 0 ise
V = (-1)S * 2-126 * (0.FFF...F)
```

Örnekler :

```
0 00000000 10000000000000000000000 = +1 * 2-126 * 0.1
```

```
0 00000000 00000000000000000000001 = +1 * 2-126 * 0.00000000000000000000001
                                   = 2-149 (en küçük pozitif değer)
```

```
EEEEEEEE = 0 ve FFF...F = 0 ve S = 1 ise V = -0
EEEEEEEE = 0 ve FFF...F = 0 ve S = 0 ise V = 0
```

2. Çift duyarlıklı gerçek sayı formatı (double precision format)
Bu formatta gerçek sayının gösterimi genel formüle aşağıdaki biçimde uyarlanabilir:

$$x = s * 2^{e} * \sum_{k=1}^{53} f_k * 2^{-k}$$

Bu formatta gerçek sayı *64* bit yani *8 byte* ile ifade edilir.
64 bit üç ayrı kısma ayrılmıştır.

i. *İşaret biti (sign bit) (1 bit)*
Aşağıda *S* harfi ile gösterilmiştir.
İşaret biti *1* ise sayı negatif, işaret biti *0* ise sayı pozitiftir.

ii. *Üstel kısım (exponent) (11 bit)*
Aşağıda *E* harfleriyle gösterilmiştir.

iii. *Ondalık kısım (fraction) (52 bit)*
Aşağıda *F* harfleriyle gösterilmiştir.

```
SEEEEEEEEEEEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Aşağıdaki formüle göre sayının değeri hesaplanabilir, *V* sayının değeri olmak üzere:

```
EEEEEEEEEEE = 2047 ve FFF...F ≠ 0  ise V = NaN (Not a number)
EEEEEEEEEEE = 2047 ve FFF...F = 0 ve S = 1 ise  V = -sonsuz
EEEEEEEEEEE = 2047 ve FFF...F = 0 ve S = 0 ise  V = +sonsuz
```

$$0 < EEEEEEEEEEE < 2047 \text{ ise } V = (-1)^S * 2^{(EEEEEEEEEEE - 1023)} * (1.FFF...F)$$

Önce sayının ondalık kısmının başına *1* eklenir. Daha sonra bu sayı $2^{(EEEEEEEEEEE-123)}$ ile çarpılarak noktanın yeri ayarlanır. Noktadan sonraki kısım, ikinin artan negatif üsleriyle çarpılarak elde edilir.

$$EEEEEEEEEEE = 0 \text{ ve } FFF...F \neq 0 \text{ ise } V = (-1)^S * 2^{-126} * (0.FFF...F)$$

```
EEEEEEEEEEE = 0 ve FFF...F = 0 ve S = 1 ise V = -0
EEEEEEEEEEE = 0 ve FFF...F = 0 ve S = 0 ise V = 0
```

GENEL KAVRAMLAR ve TERİMLER

Atomlar ve Türleri

Bir programlama dilinde yazılmış kaynak dosyanın (*program*) anlamlı en küçük parçalarına "atom" (*token)* denir. Bir programlama dilinde yazılmış bir metin, atomlardan oluşur.
Bir kaynak dosya, derleyici program tarafından ilk önce atomlarına ayrılır. Bu işleme "*atomlarına ayırma*" *(Tokenizing - Lexical analysis*) denir. Farklı programlama dillerinin atomları birbirlerinden farklı olabilir.
Atomlar aşağıdaki gibi gruplara ayrılabilir:

1. Anahtar Sözcükler

Anahtar sözcükler (*keywords - reserved words)* programlama dili tarafından önceden belirlenmiş anlamlara sahip atomlardır. Bu atomlar kaynak dosya içinde başka bir anlama gelecek biçimde kullanılamazlar. Örneğin bu atomların değişken ismi olarak kullanılmaları geçerli değildir.
Standard ANSI C (*C 89*) dilinin *32* anahtar sözcüğü vardır:

```
auto break case char const continue default do double else enum extern
float for goto if int long  register return short signed sizeof static
struct switch typedef union unsigned void volatile while
```

Derleyici yazan firmalar da kendi yazdıkları derleyiciler için ek anahtar sözcükler tanımlayabilir.
Bazı programlama dillerinde anahtar sözcüklerin küçük ya da büyük harf ile yazılması bir anlam farkı yaratmaz. Ama *C*'de bütün anahtar sözcükler küçük harf olarak tanımlanmıştır. *C* büyük harf küçük harf duyarlığı olan *(case sensitive)* bir dildir. Ama diğer programlama dillerinin çoğunda büyük - küçük harf duyarlığı yoktur *(case insensitive)*.
C dilinde örneğin bir değişkene *"register"* isminin verilmesi geçerli değildir. Çünkü "*register*" bir anahtar sözcüktür*, C* dili tarafından ayrılmıştır. Ama buna karşın bir değişkene *REGISTER, Register, RegisTER* isimleri verilebilir, çünkü bunlar artık anahtar sözcük sayılmazlar.

2. İsimler

Değişkenler, işlevler, makrolar, değişmezler gibi yazılımsal varlıklara programlama dili tarafından belirlenmiş kurallara uyulmak koşuluyla isimler verilebilir. Yazılımsal bir varlığa verilen isme ilişkin atomlar isimlerdir (*identifier).*
Her programlama dilinde olduğu gibi *C* dilinde de kullanılan isimlerin geçerliliğini belirleyen kurallar vardır.

3. İşleçler

*İşleç*ler (*operators)* önceden tanımlanmış bir takım işlemleri yapan atomlardır.
Örneğin +, -, *, / , >=, <= birer *işleç*tir.
Programlama dillerinde kullanılan *işleç* simgeleri birbirinden farklı olabileceği gibi, *işleç* tanımlamaları da birbirinden farklı olabilir. Örneğin birçok programlama dilinde üs alma işleci tanımlanmışken *C* dilinde böyle bir işleç yoktur. C'de üs alma işlemi *işleç* ile değil bir standart *işlev* (*function*) yardımıyla yapılır.
C dilinde bazı *işleç*ler iki karakterden oluşur. Bu iki karakter bitişik yazılmalıdır. Aralarına *boşluk* karakteri koyulursa *işleç* anlamı yitirilir.
C dilinin *45* tane işleci vardır.

4. Değişmezler

Değişmezler (*constants*) doğrudan işleme sokulan, değişken bilgi içermeyen atomlardır.
Örneğin:

```
sayac = son + 10
```

gibi bir ifadede *10* değişmezi doğrudan *son* isimli değişken ile işleme sokulur.

5. Dizgeler

İki tırnak içindeki ifadelere *"Dizge" (string / string litaerals)* denir. Dizgeler programlama dillerinin çoğunda tek bir atom olarak alınır, daha fazla parçaya bölünemezler.

```
"STRİNGLER DE BİRER ATOMDUR"
```

ifadesi bir dizgedir.

6. Ayraçlar, Noktalama İşaretleri

Yukarıda sayılan atom sınıflarının dışında kalan tüm atomlar bu gruba sokulabilir. Genellikle diğer atomları birbirinden ayırma amacıyla kullanıldıkları için ayraç *(separators, punctuators, delimiters)* olarak isimlendirilirler.

Bir C Programının Atomlarına Ayrılması:

Aşağıda *1*'den kullanıcının klavyeden girdiği bir tamsayıya kadar olan tamsayıları toplayan ve sonucu ekrana yazdıran bir *C* programı görülüyor:

```
#include <stdio.h>

int main()
{
    int number, k;
    int total = 0;

    printf("lütfen bir sayı girin\n");
    scanf("%d", &number);

    for(k = 1; k<= number; ++k)
        total += k;

    printf("toplam = %d\n", total);

    return 0;
}
```

Bu kaynak kod aşağıdaki gibi atomlarına ayrılabilir:

```
#     include     <     stdio.h    >    main      (      )      {      int
number      ,      k      ,     total      =      0      ;
printf     (    "lütfen bir sayı girin\n"      )      ;      scanf    (   "%d"
,      &      number      )      ;
for      (      k      =      1      ;      k      <=      ;      ++      k       )
total      +=       k       ;
printf       (        "toplam = %d\n"      ,      toplam      )       ;       }
```

Yukarıdaki atomlar aşağıda gruplandırılıyor:

Anahtar sözcükler:

```
include     int      for      return
```

İsimlendirilenler:

```
main     k      toplam       printf       scanf
```

İşleçler:

```
=     <=     ++     +=
```

Değişmezler:

```
0       1       0
```

Dizgeler:

```
("lütfen bir sayı girin\n"     )     "%d"        "toplam = %d\n"
```

Ayraçlar, noktalama işaretleri

```
<     >     (     )     ,     ;     {     }
```

Nesne

Bellekte yer kaplayan ve içeriklerine erişilebilen alanlara *nesne* (*object)* denir. Bir ifadenin n*esne* olabilmesi için bellekte bir yer belirtmesi gerekir. Programlama dillerinde nesnelere isimleri kullanarak erişilebilir.

```
a = b + k;
```

örneğinde *a, b* ve *k* birer nesnedir. Bu ifadede *a* nesnesine, *b* ve *k* nesnelerine ait değerlerin toplamı atanır.

```
sonuc = 100;
```

sonuc isimli nesneye *100* değişmez değeri atanır.

Nesnelerin bazı özelliklerinden söz edilebilir:

Nesnelerin isimleri (*name)* nesneyi temsil eden yazılımsal varlıklardır. Nesnelere isimleri programcı tarafından verilir. Her dil için nesne isimlendirmede bazı kurallar söz konusudur.

```
vergi = 20000;
```

Burada *vergi* bir *nesne* ismidir.

Nesne ile *değişken* terimleri birbirine tam olarak eşdeğer değildir. Her *değişken* bir *nesne*dir ama her *nesne* bir *değişken* değildir. *Değişken* demekle daha çok, programcının isimlendirdiği *nesne*ler kastedilir. Peki programcının isimlendirmediği nesneler de var mıdır? Evet, göstericiler konusunda da görüleceği gibi, değişken olmayan nesneler de vardır. Nesne kavramı değişken kavramını kapsar.

Nesnelerin değerleri (*value)* içlerinde tuttukları bilgilerdir. Başka bir deyişle nesneler için bellekte ayrılan yerlerdeki *1* ve *0*'ların yorumlanış biçimi, ilgili nesnenin değeridir. Bu değerler programlama dillerinin kurallarına göre, istenildikleri zaman programcı tarafından değiştirilebilirler. *C* dilinde bazı nesnelerin değerleri ise bir kez verildikten sonra bir daha değiştirilemez.

Nesnenin türü (*type)* derleyiciye o nesnenin nasıl yorumlanacağı hakkında bilgi verir. Bir nesnenin türü onun bellekteki uzunluğu hakkında da bilgi verir. Her türün bellekte ne kadar uzunlukta bir yer kapladığı programlama dillerinde önceden belirtilmiştir. Bir nesnenin türü, ayrıca o nesne üzerinde hangi işlemlerin yapılabileceği bilgisini de verir. *Tür,* nesnenin ayrılmaz bir özelliğidir. Türsüz bir nesne kavramı söz konusu değildir. Türler ikiye ayrılabilir:

1. Önceden tanımlanmış veri türleri
Önceden tanımlanmış veri türleri (*default types - built-in types, primitive types)* programlama dilinin tasarımında var olan veri türleridir. Örneğin C dilinde önceden tanımlanmış *11* ayrı veri türü vardır.

2. Programcı tarafından tanımlanan veri türleri
Programcı tarafından tanımlanan veri türleri (*user defined types*) programcının yarattığı türlerdir. Programlama dillerinin çoğunda önceden tanımlanmış türlerin yanında, programcının yeni bir tür oluşturmasını sağlayan araçlar vardır. Örneğin *C* dilinde *yapılar, birlikler, numaralandırma* araçları ile, programcı tarafından yeni bir veri türü yaratılabilir. Programlama dillerindeki tür tanımlamaları birbirlerinden farklı olabilir. Örneğin bazı programlama dillerinde *Boolean* isimli -mantıksal *doğru* ya da *yanlış* değerlerini alan- bir tür vardır. Ama C89 dilinde böyle bir tür doğrudan tanımlanmamıştır.

Bilinirlik alanı (*scope*), bir ismin, dilin derleyicisi ya da yorumlayıcısı tarafından tanınabildiği program alanıdır.

Ömür (*storage duration - lifespan*), programın çalıştırılması sırasında nesnenin varlığını sürdürdüğü zaman parçasıdır.

Bağlantı (*linkage*), nesnelerin programı oluşturan diğer modüllerde tanınabilme özelliğidir.

İfade
Değişken, *işleç* ve *değişmez*lerin bileşimlerine ifade (*expression*) denir.

```
a + b / 2
c * 2,   d = h + 34
var1
```

geçerli ifadelerdir.

Sol Taraf Değeri
Nesne gösteren ifadelere "sol taraf değeri" (*left value - L value)* denir. Bir ifadenin *sol taraf değeri* olabilmesi için mutlaka bir nesne göstermesi gerekir. Bir ifadenin *sol taraf değeri* olarak isimlendirilmesinin nedeni o ifadenin atama işlecinin sol tarafına getirilebilmesidir.
Örneğin *a* ve *b* nesneleri tek başına sol taraf değerleridir. Çünkü bu ifadeler atama işlecinin sol tarafına getirilebilirler. Örneğin :

```
a = 17
```

ya da

```
b = c * 2
```

denilebilir. Ama *a + b* bir sol taraf değeri değildir. Çünkü

```
a + b = 25
```

denilemez.

Değişkenler, her zaman *sol taraf değeri*dirler. *Değişmezler, sol taraf değeri* olamazlar.

Sağ Taraf Değeri:
Daha az kullanılan bir terimdir. Nesne göstermeyen ifadeler *sağ taraf değeri* (*right value*) olarak isimlendirilirler. Tipik olarak, atama işlecinin sol tarafında bulunamayan yalnızca

sağ tarafında bulunabilen ifadelerdir. Değişmezler her zaman sağ taraf değeri oluştururlar.
Bir ifade *sol taraf değeri* değilse *sağ taraf değeridir*. *Sağ taraf değeri* ise *sol taraf değeri* değildir. Her ikisi birden olamaz. Yani atama işlecinin sağ tarafına gelebilen her ifade *sağ taraf değeri* olarak isimlendirilmez. *Sağ taraf değeri*, genellikle bir ifadenin nesne göstermediğini vurgulamak için kullanılır.

Değişmez İfadeler

Yalnızca değişmezlerden oluşan bir ifadeye "değişmez ifade" (*constant expression)* denir. Bir değişmez ifadede değişkenler ya da işlev çağrıları yer alamaz:

```
10
3.5
10 + 20
```

ifadeleri değişmez ifadelerdir.
Değişmez ifadeler, derleme aşamasında derleyici tarafından net sayısal değerlere dönüştürülebilir. C dilinin sözdizim kuralları birçok yerde değişmez ifadelerin kullanılmasını zorunlu kılar.

Deyim

C dilinin cümlelerine deyim (*statement)* denir. C dilinde deyimler ";" ile sonlandırılır.

```
result = number1 * number2
```

bir ifadedir. Ancak

```
result = number1 * number2;
```

bir deyimdir. Bu deyim derleyicinin, *number1* ve *number2* değişkenlerin değerlerinin çarpılarak, elde edilen değerin *result* değişkenine atanmasını sağlayacak şekilde kod üretmesine neden olur.
Bazı deyimler yalnızca derleyiciye bilgi vermek amacıyla yazılır, derleyicinin bir işlem yapan kod üretmesine yol açmaz. Böyle deyimlere bildirim deyimleri *(declaration statement*) denir. Bazı deyimler derleyicinin bir işlem yapan kod üretmesini sağlar. Böyle deyimlere yürütülebilir deyimler *(executable statement*) denir.

BİR C PROGRAMI OLUŞTURMAK

C dilinde yazılan bir programın çalıştırılabilir hale getirilebilmesi için, çoğunlukla aşağıdaki süreçlerden geçilir:

1. Kaynak dosyanın oluşturulması
Kaynak dosya, metin düzenleyici bir programda (*text editörü*) yazılır. Kaynak dosya bir metin dosyasıdır. C dilinin kurallarına göre yazılan dosyaların uzantısı, geleneksel olarak ".c" seçilir.

2. Kaynak dosyanın derleyici program (*compiler*) tarafından derlenmesi:
Bir programlama dilinde yazılmış programı başka bir programlama diline çeviren programlara "çevirici" (*translator*) denir. Dönüştürülmek istenen programın yazıldığı dile "kaynak dil" (*source language*), dönüşümün yapıldığı dile ise "hedef dil" (*target language*) denir. Hedef dil, makine dili ya da simgesel makine dili ise, böyle çevirici programlara "derleyici" (*compiler*) denir.

Derleyici program kaynak dosyayı alır, çeviri işleminde eğer başarılı olursa bu kaynak dosyadan bir "amaç dosya" (*object file*) üretir.
Derleyici programın derleme işlemini yapma sürecine "derleme zamanı " (*compile time*) denir. Derleme işlemi başarısızlık ile de sonuçlanabilir.

Bir derleyici program, kaynak metni makine diline çevirme çabasında, kaynak metnin C dilinin sözdizim kurallarına uygunluğunu da denetler.
Kaynak metinde dilin kurallarının çiğnendiği durumlarda, derleyici program bu durumu bildiren bir ileti (*diagnostic message*) vermek zorundadır. Derleyici programın verdiği ileti:

i) Bir "hata iletisi" (*error message*) olabilir. Bu durumda, derleyici programlar çoğunlukla amaç dosya üretmeyi reddeder.
ii) Bir uyarı iletisi olabilir (*warning message*). Bu durumda, derleyici programlar çoğunlukla amaç dosyayı üretir.

C standartlarına göre derleyici programlar, dilin kurallarının çiğnenmesi durumlarının dışında da, programcıyı mantıksal hatalara karşı korumak amacıyla, istedikleri kadar uyarı iletisi üretebilir.

Unix/Linux sistemlerinde oluşturulan amaç dosyaların uzantısı ".o" dur. *DOS ve Windows* sistemlerinde amaç dosyalar "*.obj*" uzantısını alır.

Derleyici programlar, genellikle basit bir arayüz ile işletim sisteminin komut satırından çalıştırılacak biçimde yazılır. Arayüzün basit tutulmasının nedeni başka bir program tarafından kolay kullanabilmesini sağlamak içindir. Örneğin, *Microsoft* firmasının C/C++ derleyicisi aslında "*cl.exe*" isimli bir programdır. *UNIX* sistemlerindeki *GNU*'nun *gcc* derleyicisi ise aslında *gcc.exe* isimli bir programdır.
Derleyici programları daha kolay yönetmek için, IDE (*integrated development environment*) denilen geliştirme ortamları kullanılabilir. IDE derleyici demek değil, derleyiciyi çalıştıran ve program yazmayı kolaylaştıran geliştirme ortamlarıdır. Örneğin *MinGW* ve *DevC++* derleyici değil, IDE programlarıdır. Bu programlar gcc derleyicisini kullanmaktadır.

3. Daha önce elde edilmiş amaç dosyalar "bağlayıcı" (*linker*) program tarafından birleştirilerek çalıştırılabilir bir dosya elde edilir. *UNIX* sistemlerinde genellikle çalıştırılabilir dosyanın uzantısı olmaz. *Windows* sistemlerinde çalıştırılabilir dosyaların uzantısı "*.exe*" olarak seçilir.

Önişlemci Program

C ve C++ dillerinde derleyici programdan daha önce kaynak kodu ele alan "önişlemci" (*preprocessor*) isimli bir program kullanılır. Önişlemci program ayrı bir konu başlığı altında ele alınacak.

VERİ TÜRLERİ

Nesnelerin en önemli özelliklerinden biri, nesnenin türüdür. *Tür (type*), nesnenin olmazsa olmaz bir özelliğidir. Türü olmayan bir nesneden söz etmek mümkün değildir. Derleyiciler nesnelerle ve verilerle ilgili kod üretirken, *tür* bilgisinden faydalanır. Derleyiciler nesnenin *tür* bilgisinden, söz konusu veriyi bellekte ne şekilde tutacaklarını, verinin değerini elde etmek için veri alanındaki *1* ve *0* ları nasıl yorumlayacaklarını, veriyi hangi işlemlere sokabileceklerini öğrenir.
Programlama dilleri açısından baktığımız zaman türleri iki ayrı gruba ayırabiliriz.

1. Önceden Tanımlanmış Türler

Programlama dilinin tasarımından kaynaklanan ve dilin kurallarına göre varlığı güvence altına alınmış olan türlerdir. Her programlama dili programcının doğrudan kullanabileceği, çeşitli özelliklere sahip veri türleri tanımlar. *C* dilinde de önceden tanımlanmış *11* tane veri türü vardır.

2. Programcının Tanımlanmış Olduğu Türler

Programlama dillerinin çoğu, önceden tanımlanmış veri türlerine ek olarak, programcının da yeni türler tanımlanmasına izin verir. Programcının tanımlayacağı bir nesne için önceden tanımlanmış veri türleri yetersiz kalırsa, programcı dilin belli *sözdizim (sentaks)* kurallarına uyarak kendi veri türünü yaratabilir. C dilinde de programcı yeni bir veri türünü derleyiciye tanıtabilir, tanıttığı veri türünden nesneler tanımlayabilir.

Farklı programlama dillerindeki önceden tanımlanan veri türleri birbirlerinden farklı olabilir. Daha önce öğrenmiş olduğunuz bir programlama dilindeki türlerin aynısını *C* dilinde bulamayabilirsiniz.

C dilinin önceden tanımlanmış *11* veri türü vardır. Bu veri türlerinden *8* tanesi tamsayı türünden verileri tutmak için, kalan *3* tanesi ise gerçek sayı türünden verileri tutmak için tasarlanmıştır. Biz bu türlere sırasıyla *"tamsayı veri türleri" (integer types)* ve *"gerçek sayı veri türleri" (floating types)* diyeceğiz.

Tamsayı Veri Türleri

C dilinin toplam *4* ayrı tamsayı veri türü (*integer types)* vardır. Ancak her birinin kendi içinde işaretli (*signed)* ve işaretsiz (*unsigned*) biçimi olduğundan toplam tamsayı türü *8* kabul edilir.
İşaretli (signed) tamsayı türlerinde *pozitif* ve *negatif* tam sayı değerleri tutulabilirken, *işaretsiz (unsigned)* veri türlerinde *negatif* tamsayı değerleri tutulamaz.
Aşağıda C dilinin temel tamsayı veri türleri tanıtılıyor:

İşaretli karakter türü:
Bu veri türüne kısaca *signed char* türü denir.
Şüphesiz *char* sözcüğü ingilizce *"character"* sözcüğünden kısaltılmıştır, Türkçe *"karakter"* anlamına gelir. Ancak bu türün ismi *C*'nin anahtar sözcükleri olan *signed* ve *char* sözcükleri ile özdeşleşip, *"signed char"* olarak söylenir. İşaretli *char* türünden bir nesnenin *1 byte*'lık bir alanda tutulması *C* standartlarınca güvence altına alınmıştır.

1 byte'lık bir alan işaretli olarak kullanıldığında bu alanda saklanabilecek değerler *-128 / 127* değerleri arasında olabilir.

İşaretsiz karakter türü:
İşaretsiz *char* türünün işaretli olandan farkı *1 byte'lık* alanın işaretsiz olarak, yani yalnızca *0* ve pozitif sayıların ifadesi için kullanılmasıdır. Bu durumda işaretsiz *char* türünde *0 - 255* arasındaki tamsayı değerleri tutulabilir.

Karakter türü:

İşaretli ya da işaretsiz olarak kullanılacağı derleyicinin seçimine bağlı olan bir türdür.

İşaretli ve işaretsiz kısa tamsayı veri türü
short ve *int* sözcükleri *C* dilinin anahtar sözcüklerinden olduğu için bu türe genellikle *short int* ya da kısaca *short* türü denir.

İşaretli veya işaretsiz *short* türünden bir nesne tanımlandığı zaman, nesnenin bellekte kaç *byte* yer kaplayacağı sistemden sisteme değişebilir. Sistemlerin çoğunda, *short int* veri türünden yaratılan nesne bellekte *2 byte'lık* bir yer kaplar. İşaretli *short int* türünden bir nesne *-32768 - +32767* aralığındaki tamsayı değerlerini tutabilirken, işaretsiz *short* türünde tutulabilecek değerler *0 - 65535* aralığında olabilir.

İşaretli ve işaretsiz tamsayı türü
Bu türe *kısaca int* türü denir.
İşaretli ve işaretsiz *int* türünden bir nesne tanımlandığı zaman, nesnenin bellekte kaç *byte* yer kaplayacağı sistemden sisteme değişebilir. Çoğunlukla *16* bitlik sistemlerde, *int* türünden veri *2 byte*, *32* bitlik sistemlerde ise *int* türünden veri türü *4 byte* yer kaplar. *16* bitlik sistem demekle işlemcinin *yazmaç (register)* uzunluğunun *16* bit olduğu anlatılır.

int veri türünün *2 byte* uzunluğunda olduğu sistemlerde bu veri türünün sayı sınırları, işaretli *int* türü için *-32768 - +32767,* işaretsiz *int* veri türü için *0 - +65535* arasında olur.

int veri türünün *4 byte* uzunluğunda olduğu sistemlerde bu veri türünün sayı sınırları, işaretli *int* türü için *-2147483648 - +2147483647,* işaretsiz *int* veri türü için *0 -* +4.294.967.295 arasında olur.

İşaretli ve işaretsiz uzun tamsayı veri türü
long ve *int* sözcükleri *C* dilinin anahtar sözcüklerinden olduğu için bu türe genellikle *long int* ya da kısaca *long* türü denir.

İşaretli veya işaretsiz *long* türünden bir nesne tanımlandığı zaman, nesnenin bellekte kaç *byte* yer kaplayacağı sistemden sisteme değişebilir. Sistemlerin çoğunda, *long int* veri türünden yaratılan nesne bellekte *4 byte'lık* bir yer kaplar. İşaretli *long int* türünden bir nesne *-2147483648- +2147483648* aralığındaki tamsayı değerlerini tutabilirken, işaretsiz *long* türünde tutulabilecek değerler *0 +4.294.967.295* aralığında olabilir.

Gerçek Sayı Türleri

C dilinde gerçek sayı değerlerini tutabilmek için *3* ayrı veri türü tanımlanmıştır. Bunlar sırasıyla, *float, double* ve *long double* veri türleridir. Gerçek sayı veri türlerinin hepsi işaretlidir. Yani gerçek sayı veri türleri içinde hem pozitif hem de negatif değerler tutulabilir. Gerçek sayıların bellekte tutulması sistemden sisteme değişebilen özellikler içerebilir. Ancak sistemlerin çoğunda *IEEE 754* sayılı standarda uyulur.

Sistemlerin hemen hepsinde *float* veri türünden bir nesne tanımlandığı zaman bellekte *4 byte* yer kaplar. *4 byte lık* yani *32* bitlik alana özel bir kodlama yapılarak gerçek sayı değeri tutulur. *IEEE 754* sayılı standartta *4 byte lık* gerçek sayı formatı *"single precision" (tek duyarlık)* olarak isimlendirilirken, *8 byte lık* gerçek sayı formatı *"double precision" (çift duyarlık)* olarak isimlendirilmiştir.

Sistemlerin hemen hepsinde *double* veri türünden bir nesne tanımlandığı zaman bellekte *8 byte* yer kaplar.

long double veri türünün uzunluğu sistemden sisteme değişiklik gösterir. Bu tür, sistemlerin çoğunda *8, 10, 12 byte* uzunluğundadır.

Sayı sistemleri bölümümüzden, gerçek sayıların *n byte'lık* bir alanda özel bir biçimde kodlandığını hatırlayacaksınız. Örneğin *IEEE 754* sayılı standartta *4* ya da *8 byte lık* alan üç ayrı parçaya bölünmüş, bu parçalara özel anlamlar yüklenmiştir. Örneğin gerçek sayının *4 byte lık* bir alanda tutulması durumunda *1* bit, sayının işaretini tutmak için kullanılırken, *23* bit, sayının *ondalık (fraction)* kısmını tutar. Geriye kalan *8* bit ise, noktanın en sağa alınması için gerekli bir çarpım faktörünü dolaylı olarak tutar. Bu durumda gerçek sayının *8 byte'lık* bir alanda kodlanması durumunda, hem tutulabilecek sayının büyüklüğü artarken hem de noktadan sonraki duyarlık da artar.

Aşağıda *C* dilinin doğal veri türlerine ilişkin bilgiler bir tablo şeklinde veriliyor:

TAMSAYI TÜRLERİ (INTEGER TYPES)

TÜR İSMİ	UZUNLUK (*byte*) (DOS /UNIX)	SINIR DEĞERLERİ
signed char	1	-128 - 127
unsigned char	1	0 - 255
char	1	Derleyiciye bağlı
signed short int	2	-32.768 / 32.767
unsigned short int	2	0 / 65.535
signed int	2	-32.768 / 32.767
	4	-2.147.483.648 - 2.147.483.647
unsigned int	2	0 / 65.535
	4	0 / 4.294.967.295
signed long int	4	-2.147.483.648 - 2.147.483.647
unsigned long int	4	0 / 4.294.967.295
_Bool (C 99)	1	false / true
signed long long int (C 99)	8	-9.223.372.036.854.775.808 / 9.223.372.036.854.775.807
unsigned long long int(C 99)	8	0 / 18.446.744.073.709.551.615

GERÇEK SAYI TÜRLERİ (FLOATING TYPES)

TÜR İSMİ	UZUNLUK (*byte*)	SINIR DEĞERLERİ	
		en küçük pozitif değer	en büyük pozitif değer
float	4	1.17×10^{-38} (6 basamak duyarlık)	3.40×10^{38} (6 basamak duyarlık)
double	8	2.22×10^{-308} (15 basamak duyarlık)	1.17×10^{-308} (15 basamak duyarlık)
long double	8/10/12	taşınabilir değil	
float _Complex (C 99)			
double _Complex (C 99)			
long double _Complex (C 99)			
float _Imaginary (C 99)			
double _Imaginary (C 99)			
long double _Imaginary (C 99)			

Yukarıda verilen tablo, sistemlerin çoğu için geçerli de olsa *ANSI C* standartlarına göre yalnızca aşağıdaki özellikler güvence altına alınmıştır:

char türü *1 byte* uzunluğunda olmak zorundadır.
short veri türünün uzunluğu *int* türünün uzunluğuna eşit ya da *int* türü uzunluğundan küçük olmalıdır.
long veri türünün uzunluğu *int* türüne eşit ya da *int* türünden büyük olmak zorundadır.
Yani

short <= int <= long

Derleyiciler genel olarak derlemeyi yapacakları sistemin özelliklerine göre *int* türünün uzunluğunu işlemcinin bir kelimesi kadar alırlar. *16* bitlik bir işlemci için yazılan tipik bir uygulamada

char türü *1 byte*

int türü *2 byte* (işlemcinin bir kelimesi kadar)
short türü *2 byte (short = int*)
long türü *4 byte* (*long > int*)

alınabilir.
Yine 32 bitlik bir işlemci için yazılan tipik bir uygulamada

char türü 1 *byte*
int türü *4 byte* (işlemcinin bir kelimesi kadar)
short türü *2 byte* (*short < int*)
long türü *4 byte* (*long = int*)

alınabilir.

C dilinin en çok kullanılan veri türleri tamsayılar için *int* türüyken, gerçek sayılar için *double* veri türüdür. Peki hangi durumlarda hangi veri türünü kullanmak gerekir? Bunun için hazır bir reçete vermek pek mümkün değil, zira kullanacağımız bir nesne için tür seçerken birçok etken söz konusu olabilir. Ama genel olarak şu bilgiler verilebilir: Gerçek sayılarla yapılan işlemler tam sayılarla yapılan işlemlere göre çok daha yavaştır. Bunun nedeni şüphesiz gerçek sayıların özel bir şekilde belirli bir *byte* alanına kodlanmasıdır. Tamsayıların kullanılmasının yeterli olduğu durumlarda bir gerçek sayı türünün kullanılması, çalışan programın belirli ölçüde yavaşlaması anlamına gelir. Bir tamsayı türünün yeterli olması durumunda gerçek sayı türünün kullanılması programın okunabilirliğinin de azalmasına neden olurr.

BİLDİRİM ve TANIMLAMA

Bildirim Nedir?
Bir kaynak dosya içinde yazılan geçerli *C* deyimlerinin bir kısmı, bir işlem yapılmasına yönelik değil, derleyiciye derleme zamanında kullanacağı bir bilgi vermeye yöneliktir. Böyle deyimlere *bildirim* (*declaration*) deyimi denir. Derleyici geçerli bir bildirim deyiminin üstünden geçtiğinde bir bilgi alır ve aldığı bu bilgiyi yine derleme zamanında kullanır.

Tanımlama Nedir?
Ancak bazı bildirim deyimleri vardır ki, derleyici bu deyimlerden aldığı bilgi sonucunda, bellekte bir yer ayırır. Tanımlama *(definition*), derleyicinin bellekte yer ayırmasını sağlayan bildirim deyimleridir.

Değişkenlerin Tanımlanması
C dilinde bir değişken derleyiciye tanıtılmadan kullanılamaz. Derleyicinin söz konusu değişken için bellekte bir yer ayırmasını sağlamak için, uygun bir sözdizimi ile, değişkenin ismi ve türü derleyiciye bildirilir. Bildirim işlemi yoluyla, derleyiciler değişkenlerin hangi özelliklere sahip olduklarını anlar. Böylece bu değişkenler için programın çalışma zamanına yönelik olarak bellekte uygun bir yer ayırma işlemi yapabilir.

C dilinde eğer yapılan bir bildirim işlemi, derleyicinin bellekte bir yer ayırmasına neden oluyorsa bu işleme *tanımlama (definition*) denir. Tanımlama nesne yaratan bir bildirimdir. Programlama dillerinin çoğunda nesneler kullanılmadan önce derleyiciye tanıtılırlar.
Her tanımlama işlemi aynı zamanda bir bildirim işlemidir. Ama her bildirim işlemi bir tanımlama olmayabilir. Başka bir deyişle, tanımlama nesne yaratılmasını sağlayan bir bildirim işlemidir.

C dilinde bir değişkeni bildirimini yapmadan önce kullanmak geçersizdir, derleme işleminde hata *(error)* oluşumuna yol açar.

Bir değişkenin derleyiciye tanıtılması, değişkenin türünün ve isminin derleyiciye bildirilmesidir. Derleyici bu bilgiye dayanarak değişken için bellekte ne kadar yer ayıracağını, değişkenin için ayrılan *byte*'lardaki 1 ve 0 ların nasıl yorumlanacağı bilgisini elde eder.

Değişken Tanımlama İşleminin Genel Biçimi
C'de bildirim işlemi aşağıdaki şekilde yapılır :

```
<tür belirten sözcükler> <değişken ismi> <;>
```

Tanımlamanın bir noktalı virgülle sonlandırıldığını görüyorsunuz. Nasıl normal dilde, nokta cümleleri sonlandırıyorsa, *C* dilinde de noktalı virgül atomu, *C* dilinin cümleleri olan deyimleri sonlandırır. "Noktalı virgül" atomu *C* dilinin cümlelerinin noktasıdır. Bundan sonra noktalı virgül atomuna *"sonlandırıcı atom"* (*terminator)* diyeceğiz. Noktalı virgül ayraç türünden bir atomdur. *C*'de deyimler, çoğunlukla bu ayraç ile birbirlerinden ayrılır.

```
a = x + 1; b = x + 2;
```

ifadelerinde bulunan noktalı virgüller bunların ayrı birer deyim olduklarını gösterir. Eğer bir tek noktalı virgül olsaydı derleyici iki ifadeyi tek bir ifade gibi yorumlardı.

```
a = x + 1  b = x + 2;
```

Yukarıdaki ifade tek bir ifade gibi yorumlanır ve derleyici buna geçerli bir anlam veremez.

Tür belirten anahtar sözcükler, *C* dilinin önceden tanımlanmış veri türlerine ilişkin anahtar sözcüklerdir. Bu sözcüklerin kullanılmasıyla, tanımlanacak değişkenlerin, *11* temel veri türünden hangisine ait olduğu bildirilir. *C* dilinin önceden tanımlanmış veri türlerine ilişkin, bildirim işleminde kullanılabilecek anahtar sözcükler şunlardır:

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

Bu sözcüklerin hepsi anahtar sözcük olduğundan küçük harf ile yazılmalıdır, *C* dilinin büyük harf küçük harf duyarlı (*case sensitive*) bir dil olduğunu hatırlayalım. *C* dilinin tüm anahtar sözcükleri küçük harf ile tanımlanmıştır.

Tür belirten anahtar sözcükler aşağıdaki çizelgede listelenen seçeneklerden biri olmalıdır. Köşeli ayraç içindeki ifadeler kullanılması zorunlu olmayan, yani seçime bağlı olan anahtar sözcükleri gösteriyor. Aynı satırdaki tür belirten anahtar sözcükler tamamen aynı anlamda kullanılabilir:

1	işaretli char türü	`[signed] char` `char`
2	işaretsiz char türü	`unsigned char`

3	işaretli kısa tamsayı türü	`[signed] short [int]`
		`[signed] short`
		`short [int]`
		`short`

4	işaretsiz kısa tamsayı türü	`unsigned short [int]` `unsigned short`

5	işaretli tamsayı türü	`[signed] int`
		`int`
		`signed`

6	işaretsiz tamsayı türü	`unsigned int` `unsigned`

7	işaretli uzun tamsayı türü	`[signed] long [int]`
		`[signed] long`
		`long [int]`
		`long`

8	işaretsiz uzun tamsayı türü	`unsigned long [int]` `unsigned long`
9	float türü	`float`
10	double türü	`double`
11	long double türü	`long double`

Yukarıdaki tablodan da görüldüğü gibi, belirli türleri birden fazla şekilde ifade etmek mümkündür.

```
char a;
signed char a;
```

```
int a;
signed int a;
signed a;
```

```
long a;
long int a;
signed long a;
signed long int a;
```

Yukarıda aynı kolon üzerindeki bildirimlerin hepsi, derleyici tarafından birbirine eşdeğer olarak ele alınır.

Bildirim sözdiziminde, değişken ismi olarak, *C* dilinin isimlendirme kurallarına uygun olarak seçilen herhangi bir isim kullanılabilir.
C dilinde *isimlendirilen (identifiers*) varlıklar başlıca 6 grubu içerir. *Değişkenler* (*variables*) bunlardan yalnızca biridir. *İşlevler* (*functions)*, *etiketler* (*labels)*, *makrolar* (*macros)*, *yapı* ve *birlik* isimleri (*structure and union tags*), *enum* değişmezleri *(enum constants)* isimlerini programcılardan alır.

C Dilinin İsimlendirme Kuralları

İsimlendirmede yalnızca *63* karakter kullanılabilir. Bunlar:
İngiliz alfabesinde yer alan *26* küçük harf karakteri:

```
a,b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z
```

İngiliz alfabesinde yer alan 26 büyük harf karakteri:

```
A,B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
```

rakam karakterleri

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

alttire (_) (*underscore*) karakteridir.

İsimlendirmelerde yukarıda belirtilen karakterlerin dışında başka bir karakterin kullanılması geçersizdir. Örneğin, boşluk karakterleri, Türkçeye özgü (*ç, ğ, ı, ö, ş, ü, Ç, Ğ, İ, Ö, Ş, Ü*) karakterler, +, -, /, *, & ya da $ karakterleri bir isimde kullanılamaz.

Değişken isimleri rakam karakteriyle başlayamaz. Harf karakteri ya da alttire karakteri dışında, yukarıda geçerli herhangi bir karakterle başlayabilir. Bu kural derleyicinin değişmezlerle isimleri birbirinden ayırmasını kolaylaştırır.
Değişken isimleri alttire '_' karakteriyle başlayabilir.

C'nin *32* anahtar sözcüğü isimlendirme amacı ile kullanılamaz.

Uzun Değişken İsimleri

İsimler boşluk içeremeyeceği için uygulamalarda genellikle boşluk hissi vermek için *alttire (underscore)* karakteri kullanılır.

```
genel_katsayi_farki, square_total, number_of_cards
```

gibi.

İsimlendirmede başka bir seçenek de her sözcüğün ilk harfini büyük, diğer harfleri küçük yazmaktır:

```
GenelKatsayiFarki, SquareTotal, NumberOfCards
```

C dili standartları isim uzunluğu konusunda bir sınırlama koymamıştır. Ancak ismin ilk *31* karakterinin derleyici tarafından dikkate alınmasını zorunlu kılar. Ancak derleyicilerin çoğu, çok daha uzun değişken isimlerini işleme sokabilirler. *31* karakterden daha uzun isimler kullanıldığında programcı için çok az da olsa şöyle bir risk söz konusudur: Herhangi bir derleyici ilk *31* karakteri aynı olan iki farklı ismi aynı isim olarak ele alabilir.

C, büyük harf küçük harf duyarlılığı olan bir dil olduğu için, isimlendirmelerde de büyük harf ile küçük harfler farklı karakterler olarak ele alınır:

```
var, Var, VAr, VAR, vAR, vaR
```

değişkelerinin hepsi ayrı değişkenler olarak ele alınır.

İsimlendirmede Nelere Dikkat Edilmeli?

İsimlendirme, yazılan programların okunabilirliği açısından da çok önemlidir. İyi yazılmış olan bir programda kullanılan isimlerin dilin kurallarına göre uygun olmalarının dışında, bazı başka özelliklere de sahip olması gerekir:
1. Seçilen isimler anlamlı olmalıdır.

Verilen isimler, değişkenin kullanım amacı hakkında, okuyana bilgi vermelidir. Çoğu zaman, bir değişkenin ismini x, y, z gibi alfabenin harflerinden seçmek kötü bir tekniktir. Tanımlanan değişken, içinde neyin değerini tutacaksa buna yönelik bir isim vermek, kodu okuyanın işini kolaylaştırır, algılamasını hızlandırır.
Ancak geleneksel olarak döngü değişkenlerine *i, j, k* isimleri verilir.

2. İsimler farklı özelliklere sahip değişkenlerin ait oldukları grubun ismi olarak seçilmemelidir.
Örneğin sayaç görevini üstlenen bir değişkenin ismini

```
counter
```

olarak seçmek yerine

```
prime_counter, valid_word_counter vs.
```

olarak seçmek çok daha iyidir. Zira programın içinde birden fazla sayaç bulunabilir. Bunların isimleri, neyi saydıklarına göre değişirse, kodu okuyan kişinin anlamlandırması çok daha kolaylaşır.

3. İsimlendirmede dil bütünlüğü olmalıdır.
Yazılımda kullanılan temel dilin İngilizce olduğunu kabul etmek zorundayız. Yazılım projelerinde isimlendirme genellikle İngilizce tabanlı yapılır. Ancak bazı değişkenlerin isimlerinin İngilizce seçilirken bazı başka değişkenlerin isimlerinin Türkçe seçilmesi programın okunabilirliğine zarar verir.

4. C'de değişken isimleri, geleneksel olarak küçük harf yoğun seçilirler.
Tamamı büyük harflerle yazılmış değişken isimleri daha çok simgesel değişmezler için kullanılırlar.

Bildirim Örnekleri

```
int x;
unsigned long int var;
double MFCS;
unsigned _result;
signed short total;
```

Tür belirten anahtar sözcüklerin yazılmasından sonra aynı türe ilişkin birden fazla nesnenin bildirimi, isimleri arasına virgül atomu koyularak yapılabilir. Bildirim deyimi yine noktalı virgül atomu ile sonlandırılmalıdır.

```
unsigned char ch1, ch2, ch3, ch4;
float FL1, Fl2;
unsigned total, subtotal;
int _vergi_katsayisi, vergi_matrahi;
```

Farklı türlere ilişkin bildirimler virgüllerle birbirinden ayrılamaz.

```
long x, int y;          /* Geçersiz! */
```

signed ve *unsigned* sözcükleri tür belirten anahtar sözcük(ler) olmadan yalnız başlarına kullanılabilirler. Bu durumda int türden bir değişkenin bildiriminin yapıldığı kabul edilir:

```
signed x, y;
```

ile

```
signed int x, y;
```

tamamen aynı anlamdadır. Yine

```
unsigned u;
```

ile

```
unsigned int u;
```

Tamamen aynı anlamdadır.

Bildirim deyiminde, tür belirten anahtar sözcük birden fazla ise bunların yazım sırası önemli değildir. Ama okunabilirlik açısından önce işaret belirten anahtar sözcüğün sonra tür belirten anahtar sözcüğün kullanılması gelenek haline gelmiştir. Örneğin :

```
signed long int x;
signed int long x;
long signed int x;
long int signed x;
int long signed x;
int signed long x;
```

bildirimlerinin hepsi geçerlidir. Seçimlik olan anahtar sözcükler özellikle kullanılmak isteniyorsa birinci yazım biçimi okunabilirlik açısından tercih edilmelidir.

Bildirimlerin Kaynak Kod İçindeki Yerleri

C dilinde genel olarak 3 yerde değişken bildirimi yapılabilir :

1. Blokların içinde
2. Tüm blokların dışında.
3. İşlevlerin parametre değişkeni olarak, işlev parametre ayraçlarının içinde.

İşlev parametre ayraçları içinde yapılan bildirimler, başka bir sözdizim kuralına uyar. Bu bildirimleri "işlevler" konusunda ayrıntılı olarak ele alacağız.

C dilinde eğer bildirim blokların içinde yapılacaksa, bildirim işlemi blokların ilk işlemi olmak zorundadır. Başka bir deyişle bildirimlerden önce, bildirim deyimi olmayan başka bir deyimin bulunması geçersizdir.
Bir bildirimin mutlaka işlevin ana bloğunun başında yapılması gibi bir zorunluluk yoktur. Eğer iç içe bloklar varsa içteki herhangi bir bloğun başında da yani o bloğun ilk işlemi olacak biçimde, bildirim yapılabilir. Örnekler :

```
{
    int  var1, var2;
    char ch1, ch2, ch3;
    var1 = 10;
    float f;   /* Geçersiz */
}
```

Yukarıdaki örnekte *var1, var2, ch1, ch2, ch3* değişkenlerinin tanımlanma yerleri doğrudur. Ancak *f* değişkeninin bildirimi geçersizdir. Çünkü bildirim deyiminden önce bildirim deyimi olmayan başka bir deyim yer alıyor. Bu durum geçersizdir.
Aynı program parçası şu şekilde yazılmış olsaydı bir hata söz konusu olmazdı :

```
{
    int  var1, var2;
    char ch1, ch2, ch3;
    var1 = 10;
    { float f; }
}
```

Bu durumda artık *f* değişkeni de kendi bloğunun başında tanımlanmış olur.

C dilinde tek başına bir sonlandırıcı atom bir deyim oluşturur. Ve böyle bir deyime, *boş deyim* (*null statement*) denir. *C* sözdizimine göre oluşan bu deyim yürütülebilir bir deyimdir. Dolayısıyla aşağıdaki kod parçasında *y* değişkeninin tanımlaması derleme geçersizdir.

```
{
    int x;;
    int y;      /* Geçersiz! */
}
```

Yukarıdaki kod parçasında *x* değişkeninin bildiriminden sonra yer alan sonlandırıcı atom yürütülebilir bir deyim olarak ele alınır. Bu durumda *y* değişkeninin bildirimi geçersizdir. Aynı şekilde içi boş bir blok da *C* dilinde bir yürütülebilir deyim olarak ele alınır. Bu yazım, sonlandırıcı atomun tek başına kullanılmasına tamamen eşdeğerdir. Aşağıdaki kod parçası da geçersizdir:

```
{
    int x;
    { }
    int y;      /* Geçersiz! */
}
```

Bir ya da birden fazla deyimin bir blok içine alınmasıyla elde edilen yapı *C* dilinde *bileşik deyim* (*compound statement)* ismini alır. Bileşik deyimler de yürütülebilir deyimlerdir. Aşağıdaki kod parçası da geçersizdir:

```
{
    {int x;}
    int y;      /* Geçersiz */
}
```

[C++ dilinde blok içinde bildirimi yapılan değişkenlerin blok başlarında bildirilmeleri zorunlu değildir. Yani *C++*'da değişkenler blokların içinde herhangi bir yerde bildirilebilir.]

Tanımlanan Değişkenlere İlkdeğer Verilmesi

Bir değişken tanımlandığında bu değişkene bir ilkdeğer verilebilir (*initialize)*. Bu özel bir sözdizim ile yapılır:

```
int a = 20;
```

İlkdeğer verme deyimi bir atama deyimi değildir, bir bildirim deyimidir. Bu deyim ile, programın çalışma zamanında bir değişkenin önceden belirlenen bir değer ile hayata başlaması sağlanır. Yukarıdaki bildirimi gören derleyici a değişkenini *20* değeri ile başlatacak bir kod üretir. İlkdeğer verme deyiminde atama işlecinin sağ tarafında kullanılan ifadeye "ilkdeğer verici" (*initializer*) denir.

Bir bildirim deyimi ile birden fazla değişkene de ilkdeğer verilebilir:

```
int a = 10, b = 20, c = 30;
```

Yukarıdaki deyimle *a* değişkenine *10*, *b* değişkenine *20*, *c* değişkenine *30* ilkdeğerleri veriliyor.

İlkdeğer verme deyimi bir atama deyimi değildir:

```
void func()
{
    int a;
    a = 20;
    int b; /* Geçersiz! */
    /****/
}
```

Yukarıdaki kod parçasında b değişkeninin tanımı geçersizdir. Çünkü *b* değişkeninin tanımından önce blok içinde yürütülebilir bir deyim yazılmıştır. Ancak aşağıdaki kod geçerlidir:

```
void func()
{
    int a = 20;
    int b; /* Geçerli! */
    /****/
}
```

DEĞİŞMEZLER

Veriler ya nesnelerin içinde ya da doğrudan *değişmez* (*sabit - constant*) biçiminde bulunur. *Değişmezler* nesne olmayan, programcı tarafından doğrudan sayısal büyüklük olarak girilen verilerdir. *Değişmezler*in sayısal değerleri derleme zamanında tam olarak bilinir. Örneğin:

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

ifadesi bize *y* ve *z* içindeki değerlerin toplanacağını ve sonucun *x* değişkenine aktarılacağını anlatır. Oysa

```
d = x + 10;
```

ifadesinde *x* değişkeni içinde saklanan değer ile *10* sayısı toplanmıştır. Burada *10* sayısı herhangi bir değişkenin içindeki değer değildir, doğrudan sayı biçiminde yazılmıştır.

Değişmezlerin Türleri

Nesnelerin türleri olduğu gibi değişmezlerin de türleri vardır. Nesnelerin türleri bildirim yapılırken belirlenir. Değişmezlerin türlerini ise derleyici, belirli kurallara uyarak değişmezlerin yazılış biçimlerinden saptar. Değişmezlerin türleri önemlidir, çünkü *C* dilinde *değişmezler, değişkenler* ve *işleç*ler bir araya getirilerek *ifadeler* (*expressions*) oluşturulur. *C'de* ifadelerin de türü vardır. İfadelerin türleri, içerdikleri değişmez ve değişkenlerin türlerinden elde edilir.

Tamsayı Değişmezleri

Tamsayı değişmezlerinin (*integer constants*) değerleri tamsayıdır.
Bir tamsayı değişmezi

```
signed int
unsigned int
signed long
unsigned long
```

türlerinden olabilir.
Bir tamsayı değişmezinin türü, yazımında kullanılan sayı sistemine ve değişmezin aldığı soneke göre belirlenir.

Tamsayı Değişmezlerinin Onluk Onaltılık ve Sekizlik Sayı Sistemlerinde Yazımı

Varsayılan yazım onluk sayı sistemidir.
Onaltılık sayı sisteminde yazım:

```
0Xbbb..
```

ya da

```
0xbbb..
```

biçimindedir. Burada *b* karakterleri onaltılık sayı sistemindeki basamakları gösteriyor. *9*'dan büyük basamak değerleri için *A, B, C, D, E, F* karakterleri ya da *a, b, c, d, e, f* karakterleri kullanılır.
Sekizlik sayı sisteminde yazım:

```
0bbbb..
```

biçimindedir.
Burada *b* karakterleri sekizlik sayı sistemindeki basamakları (*0 1 2 3 4 5 6 7*) gösteriyor.

Tamsayı Değişmezlerinin Aldığı Sonekler

Bir tamsayı değişmezi, hangi sayı sisteminde yazılırsa yazılsın,
u, U, l ya da *L* soneklerini alabilir.

u ya da *U* sonekleri tamsayı değişmezinin işaretsiz tamsayı türünden olduğunu belirler.
l ya da *L* sonekleri tamsayı değişmezinin *long* türden olduğunu belirler.
l soneki *1* karakteri ile görsel bir karışıklığa neden olabileceğinden, *'L'* soneki kullanılmalıdır. Bir karakter değişmezi hem *u, U* soneklerinden birini hem de *l, L* soneklerinden birini alabilir. Bu durumda soneklerin yazım sırasının bir önemi yoktur.

Tamsayı Değişmezlerinin Türleri

Tamsayı değişmezinin türü aşağıdaki tabloya göre belirlenir:

Yazım biçimi	tür
bb...b	signed int signed long unsigned long
0xbb...b 0bb...b	signed int unsigned int signed long unsigned long
bb...bU 0bb...bU 0Xbb...bU	unsigned int unsigned long
bb...bL 0bb..bL 0xbb..bL	signed long unsigned long
bb...bUL 0bb..bUL 0xbb..bUL	unsigned long

Yukarıdaki tablo şöyle yorumlanmalıdır: Belirli bir yazım için verilen türlerden, yukarıdan aşağı doğru olmak üzere, taşma olmaksızın ilgili değeri tutabilecek ilk tür, değişmezin türüdür. Aşağıdaki örnekleri inceleyin:
int türünün ve *long* türünün *2 byte* olduğu sistemler için (DOS) örnekler

Değişmez	*int* türü *2 byte long* türü *4* byte (DOS)	*int* türü ve *long* türü *4 byte* (Windows - Unix)
456	signed int	signed int
59654	signed long	signed int
125800	signed long	signed int
3168912700	unsigned long	unsigned long
0X1C8	signed int	signed int
0XE906	unsigned int	signed int
0X1EB68	signed long	signed int
0XBCE1C53C	unsigned long	unsigned long
0710	signed int	signed int
0164406	unsigned int	signed int
0365550	signed long	signed int
027470342474	unsigned long	unsigned long
987U	unsigned int	unsigned int
45769U	unsigned int	unsigned int
1245800U	unsigned long	unsigned int
3589456800U	unsigned long	unsigned int
0X3DBU	unsigned int	unsigned int

0XB2C9U	unsigned int	unsigned int
0X130268U	unsigned long	unsigned int
0XD5F2C3A0U	unsigned long	unsigned int
01733U	unsigned int	unsigned int
0131311U	unsigned int	unsigned int
04601150U	unsigned long	unsigned int
032574541640U	unsigned long	unsigned int
25600L	signed long	signed long
3298780970L	unsigned long	unsigned long
0X6400L	signed long	signed long
0Xc49F672AL	unsigned long	unsigned long
062000L	signed long	signed long
030447663452L	unsigned long	unsigned long
890765UL	unsigned long	unsigned long
0XD978DUL	unsigned long	unsigned long
03313615UL	unsigned long	unsigned long

Karakter Değişmezleri

Karakter değişmezleri tipik olarak *char* türden nesnelere atanan değişmezlerdir. Karakter değişmezleri *C'de* farklı biçimlerde bulunabilir.

i. Bir karakterin görüntüsü *tek tırnak (single quote)* içinde yazılırsa derleyici tarafından doğrudan karakter değişmezi olarak ele alınır. Örnek :

```
'a'
'J'
'Ç'
':'
'8'
'<'
```

Yukarıdaki atomların her biri birer karakter değişmezidir.

C'de tek tırnak içinde yazılan *char* türden değişmezler, aslında o karakterin (kullanılan sistemin karakter kodundaki (*örneğin ASCII*)) kod numarasını gösteren bir tamsayıdır.

```
char ch;
ch = 'a';
/***/
```

ASCII karakter kodlamasının kullanıldığını düşünelim. Bu örnekte aslında *ch* isimli *char* türden bir değişkene *'a'* karakterinin *ASCII* tablosundaki kod numarası olan *97* değeri aktarılır. Tek tırnak içindeki karakter değişmezlerini görünce aslında onların küçük birer tamsayı olduğu bilinmelidir. Yukarıdaki örnekte istenirse *ch* değişkenine aşağıdaki gibi bir atama yapılabilir:

```
ch = 'a' + 3;
```

Bu durumda *ch* değişkenine sayısal olarak *100* değeri atanır. Bu tamsayı da *ASCII* tablosundaki *'d'* karakterinin kod numarasıdır.

Ters Bölü Karakter Değişmezleri

Karakter değişmezlerinin diğer yazımlarında *tek tırnak* içinde *ters bölü* `'\'` karakteri ve bunu izleyen başka karakter(ler) kullanılır. İngilizce de bu biçimlere *"escape sequence"* denir.

Tek tırnak içindeki *ters bölü* karakterinden sonra yer alan bazı karakterler çok kullanılan bazı karakterlerin yerlerini tutar. Bunların listesi aşağıda veriliyor:

Önceden Tanımlanmış Ters Bölü Karakter Değişmezleri

	Tanım	ASCII No
'\0'	sonlandırıcı karakter (*null character*)	0
'\a'	çan sesi (*alert*)	7
'\b'	geri boşluk (back space)	8
'\t'	tab karakteri (tab)	9
'\n'	aşağı satır (new line)	10
'\v'	düşey tab (vertical tab)	11
'\f'	sayfa ileri (form feed)	12
'\r'	satır başı (carriage return)	13
'\"'	çift tırnak (double quote)	34
'\''	tek tırnak (single quote)	39
'\?'	soru işareti (question mark)	63
'\\'	ters bölü (back slash)	92

Kullanılışlarına bir örnek :

```
char ch;
ch = '\a';
```

Onaltılık Sayı Sisteminde Yazılan Karakter Değişmezleri

Tek tırnak içinde *ters bölü* ve `'x'` karakterlerinden sonra *onaltılık* (*hexadecimal)* sayı sisteminde bir sayı yazılırsa bu sistemin kullandığı karakter setinde, o sayısal değerin gösterdiği kod numaralı karakter değişmezidir.

```
'\x41'  /* 41H kod numaralı karakterdir. */
'\xff'  /* FFH kod numaralı karakterdir. */
'\x1C'   /* 1C kod numaralı karakterdir.  */
```

Aşağıda *harf* isimli *char* türden değişkene *41H* değeri atanıyor:

```
char harf;
harf = '\x41';
```

Bu da onluk sayı sisteminde *65* değeridir. *ASCII* karakter setinin kullanıldığını varsayalım. *65* kod nolu *ASCII* karakteri *'A'* karakteridir. Dolayısıyla harf isimli değişkene *'A'* atanmış olur.

Sekizlik Sayı Sisteminde Yazılan Karakter Değişmezleri

Tek tırnak içinde *ters bölü* karakterinden sonra sekizlik sayı sisteminde bir değer yazılırsa, bu kullanılan karakter setindeki o sayısal değerin gösterdiği kod numaralı karaktere işaret eden bir karakter değişmezidir. Tek tırnak içindeki ters bölü karakterini izleyen sekizlik sistemde yazılmış sayı üç basamaktan uzun olmamalıdır. Sekizlik sistemde yazılan sayının başında sıfır rakamı olma zorunluluğu yoktur.

```
'\012'  /* 10 numaralı ASCII karakteri, Tam sayı değeri 10 */
'\16'   /* 14 numaralı ASCII karakteri. Tam sayı değeri 14 */
'\123'  /* 83 numaralı ASCII karakteri. Tam sayı değeri 83 */
```

Program içinde kullanımına bir örnek:

```
char a, b;

a = '\xbc' ;   /* onaltılık sayı sisteminde yazılmış karakter değişmezi */
b = '\012';    /* sekizlik sayı sisteminde yazılmış karakter değişmezi */
```

Örneğin, *7* numaralı *ASCII* karakteri olan çan sesi karakteri, değişmez olarak üç ayrı biçimde de yazılabilir:

```
'\x7'   /* hex gösterimli karakter değişmezi */
'\07'   /* oktal gösterimli karakter değişmezi */
'\a'    /* önceden belirlenmiş ters bölü karakter değişmezi */
```

Burada tercih edilecek biçim son biçim olmalıdır. Hem taşınabilir bir biçimdir hem de okunabilirliği daha iyidir. Başka karakter setlerinde çan sesi karakteri *7* sıra numaralı karakter olmayabilir, ama önceden belirlenmiş ters bölü karakter değişmezi şeklinde ifade edersek hangi sistem olursa olsun çan sesi karakterini verir. Ayrıca kodu okuyan çan sesi karakterinin *7* numaralı *ASCII* karakteri olduğunu bilmeyebilir, ama *'\a'* nın çan sesi karakteri olduğunu bilir.
Karakter değişmezleri konusunu kapatmadan önce karakter setleri konusunda da biraz bilgi verelim:
Günümüzde en çok kullanılan karakter seti *ASCII* karakter setidir. *ASCII (American Standard Code for Information Interchange)* sözcüklerinin baş harflerinden oluşan bir kısaltmadır. *ASCII* karakter kodunda karakterler *7* bitlik bir alanda kodlanmıştır. Bazı bilgisayarlar ise *8* bit alana genişletilmiş kodlama kullanırlar ki bu sette *128* yerine *256* karakter temsil edilebilir.
Farklı bilgisayarlar farklı karakter kodlaması kullanabilir. Örnek olarak *IBM mainframe* leri daha eski olan *EBCDIC* kodlamasını kullanır. *Unicode* ismi verilen daha geliştirilmiş bir karakter kodlaması vardır ki karakterler *2-4 byte*'lık alanda temsil edildikleri için bu kodlamada dünyada var olan tüm karakterlerin yer alması hedeflenmiştir. Gelecekte birçok makinenin bu karakter kodlamasını destekleyecek biçimde tasarlanacağı düşünülüyor.

Karakter Değişmezleri Nerede Kullanılır

Karakter değişmezleri tamsayı değişmezleridir. Ancak C'de daha çok bir yazı bilgisi ile ilgili kullanırlar. Yazılar karakter değişmezleri ile değerlerini alabilecekleri gibi, bir yazının değiştirilmesi amacıyla karakter değişmezleri kullanılabilir.

Karakter Değişmezleri int Türdendir

C'de karakter değişmezleri *int* türden olarak ele alınır ve işleme sokulur. Bu konu "tür dönüşümleri" bölümünde ele alınacak.

Gerçek Sayı Değişmezleri

Gerçek sayı değişmezleri (floating constants) değerleri gerçek sayı olan değişmezlerdir. *C* dilinde bir gerçek sayı değişmezi *float, double* ya da *long double* türden olabilir. C89 standartlarına göre bir gerçek sayı değişmezi yalnızca onluk sayı sistemi kullanılarak yazılabilir.

float Türden Değişmezler

Nokta içeren, *'f'* ya da *'F'* soneki almış değişmezler, *float* türdendir. Örneğin:

```
1.31F
10.F
-2.456f
```

float türden değişmezlerdir.

double Türden Değişmezler

Nokta içeren *'f'* ya da *'F'* soneki almamış değişmezler ile *float* türü sınırını ya da duyarlığını aşmış değişmezler *double* türden değişmezler olarak değerlendirilir. Örneğin:

*-24.5 d*ouble türden değişmezdir.

long double Türden Değişmezler

long double türden değişmezler noktalı ya da üstel biçimdeki sayıların sonuna *'l'* ya da *'L'* getirilerek elde edilir:

```
1.34L
10.2L
```

long double türden değişmezlerdir.

Gerçek Sayı Değişmezlerinin Üstel Biçimde Yazılması

Gerçek sayı değişmezleri üstel biçimde *(scientific notation)* yazılabilir. Bunun için değişmezin sonuna *'e'* ya da *'E'* eki getirilir. Bu, sayının *on üzeri* bir çarpanla çarpıldığını gösterir. *'E'* ya da *'e'* karakterlerini *'+', '-'* ya da doğrudan bir rakam karakteri izleyebilir.

```
2.3e+04f
1.74e-6F
8.e+9f
```

burada e, *10* 'un üssü anlamına gelir:

```
1.34E-2f ile 0.0134
-1.2E+2F ile 120.f
```

aynı değişmezlerdir.

İŞLEVLER

İşlev Nedir?

C'de alt programlara *işlev (function*) denir. İngilizcedeki *"function"* sözcüğü bu bağlamda matematiksel anlamıyla değil diğer programlama dillerinde ya da ortamlarında kullanılan, *"alt program", "prosedür",* sözcüklerinin karşılığı olarak kullanılır.
Bir *işlev*, bağımsız olarak çalıştırılabilen bir program parçasıdır.

Programı İşlevlere Bölerek Yazmanın Faydaları

Bir programı alt programlara yani işlevlere bölerek yazmak bazı faydalar sağlar:

1. Programın kaynak kodu küçülür. Böylece oluşturulması hedeflenen çalışabilir dosya da (örneğin .exe uzantılı dosya) küçülür.

2. Kaynak dosyanın okunabilirliği artar. Okunabilirliğin artması, kodu yazanın ve okuyanın işini kolaylaştırır. Böylece proje geliştirme süresinin azalması yönünde kazanım sağlanmış olur.

3. Belirli kod parçalarının programın farklı yerlerinde yinelenmesi, programda yapılacak olası bir değişikliğin maliyetini artırır. Programın farklı yerlerinde, kodun kullanıldığı yere bağlı olarak değişiklikler yapmak gerekir. Kaynak dosyalarda böyle değişiklikler yapmak hem zaman alıcıdır hem de risklidir. Çünkü bir değişikliğin yapılmasının unutulması durumunda ya da değişiklik yapılmaması gereken bir yerde kodun değiştirilmesi durumunda program yanlış çalışabilir. Oysa ortak kod parçaları işlevler şeklinde paketlendiğinde, yalnızca işlevlerde değişiklik yapılmasıyla, istenen değişiklik gerçekleştirilmiş olur.

4. Programda hata arama daha kolay gerçekleştirilir. Projelerdeki hata arama maliyeti azalır.

5. Yazılan işlevler başka projelerde de kullanılabilir. Alt programlar tekrar kullanılabilir (*reusable*) bir birim oluştururlar. Böylelikle de projelerdeki kodlama giderlerini azaltırlar.

İşlevler *C*'nin temel yapı taşlarıdır. Çalıştırılabilen bir *C* programı en az bir *C* işlevinden oluşur. Bir *C* programının oluşturulmasında işlev sayısında bir kısıtlama yoktur.
İşlevlerin onları çağıran işlevlerden aldıkları girdileri ve yine onları çağıran işlevlere gönderdikleri çıktıları vardır. İşlevlerin girdilerine *aktüel parametreler (actual parameters)* ya da *argümanlar (arguments)* denir. İşlevlerin çıktılarına ise *geri dönüş değeri* (*return value*) diyoruz.

Bir işlev başlıca iki farklı amaçla kullanılabilir:
1. İşlev, çalışması süresince belli işlemleri yaparak belirli amaçları gerçekleştirir.
2. İşlev, çalışması sonunda üreteceği bir değeri kendisini çağıran işleve gönderebilir.

İşlevlerin Tanımlanması ve Çağrılması

Bir işlevin ne iş yapacağının ve bu işi nasıl yapacağının *C* dilinin sözdizimi kurallarına uygun olarak anlatılmasına, yani o işlevin *C* kodunun yazılmasına, o *işlevin tanımı (definition)* denir. İşlev tanımlamaları *C* dilinin sözdizimi kurallarına uymak zorundadır.

Bir *işlev çağrısı (call / invocation)* ise o işlevin kodunun çalışmaya davet edilmesi anlamına gelir. İşlev çağrı ifadesi karşılığında derleyici, programın akışını ilgili işlevin kodunun bulunduğu bölgeye aktaracak şekilde bir kod üretir. Programın akışı işlevin kodu içinde akıp bu kodu bitirdiğinde, yani işlevin çalışması bittiğinde, programın akışı yine işlevin çağrıldığı noktaya geri döner. Bir işleve yapılacak çağrı da yine bazı sözdizimi kurallarına uymalıdır.

İşlevlerin Geri Dönüş Değerleri

Bir işlevin yürütülmesi sonunda onu çağıran işleve gönderdiği değere, işlevin geri dönüş değeri *(return value)* denir. Her işlev bir geri dönüş değeri üretmek zorunda değildir. Bir işlev yapacağı bir işle ilgili olarak bir geri dönüş değeri üretir ya da üretmez.
İşlevlerin geri dönüş değerleri farklı amaçlar için kullanılabilir:

1. Bazı işlevler zaten tek bir değeri elde etmek, tek bir değeri hesaplamak amacıyla tanımlanırlar. Elde ettikleri değeri de kendilerini çağıran işlevlere geri dönüş değeri olarak iletirler. Bir küpün hacim değerini bulan bir işlev tanımladığımızı düşünelim. Böyle bir işlev, hacmini bulacağı küpün kenar uzunluğunu çağrıldığı yerden alır, bu değeri kullanarak hacim değerini hesap eder, hesap ettiği değeri dışarıya geri dönüş değeri olarak iletebilir.

2. Her işlevin amacı bir değer hesaplamak değildir. Bazı işlevler ise çağrılmalarıyla kendilerine sorulan bir soruya yanıt verirler. Örneğin bir sayının asal olup olmadığını sınayan bir işlev tanımlandığını düşünelim. İşlev çağrıldığı yerden, asallığını sınayacağı değeri alır. Tanımında bulunan bazı kodlar ile sayının asal olup olmadığını sınar. Sayının asal ya da asal olmamasına göre dışarıya iki farklı değerden birini geri dönüş değeri olarak gönderebilir. Bu durumda işlevin geri dönüş değeri, hesap edilen bir değer değil, sorunun yanıtı olarak yorumlanacak bir değerdir.

3. Bazı işlevler ise ne bir değeri hesaplamak ne de bir soruya yanıt vermek için tanımlanırlar. Tanımlanma nedenleri yalnızca bir iş yapmaktır. Ancak işlevin yapması istenen işin, başarıyla yapılabilmesi konusunda bir garanti yoktur. Örneğin bir dosyayı açmak için bir işlev tanımlandığını düşünelim. İşlev çağrıldığı yerden açılacak dosyanın ismi bilgisini alıyor olabilir. Ancak dosyanın açılabilmesi çeşitli nedenlerden dolayı güvence altında değildir. Çağrılan işlev istenen dosyayı ya açar ya açamaz. İşlev geri dönüş değeriyle yaptığı işin başarısı hakkında bilgi verir. Bu durumda işlevin geri dönüş değeri, hesap edilen bir değer değil, yapılması istenen işin başarısı konusunda verilen bir bilgi olarak yorumlanır.

4. Bazı işlevler hem belli bir amacı gerçekleştirirler hem de buna ek olarak amaçlarını tamamlayan bir geri dönüş değeri üretirler. Bir yazı içinde bulunan belirli bir karakteri silecek bir işlev tasarlandığını düşünelim. İşlevin varlık nedeni yazının içinden istenen karakterleri silmektir. Çağrıldığı yerden, silme yapacağı yazıyı ve silinecek karakterin ne olduğu bilgisini alır ve işini yapar. Ancak işini bitirdikten sonra yazıdan kaç karakter silmiş olduğunu geri dönüş değeri ile çağrıldığı yere bildirilebilir.

5. Bazı işlevlerin ise hiç geri dönüş değerleri olmaz.
i) İşlevin amacı yalnızca bir işi gerçekleştirmektir, yaptığı işin başarısı güvence altındadır. Örneğin yalnızca ekranı silme amacıyla tasarlanmış olan bir işlevin geri dönüş değerine sahip olması gereksizdir. Sistemlerin çoğunda çıktı ekranının silinmesi konusunda bir başarısızlık riski yoktur.
ii) İşlev dışarıya bir değer iletir ancak değeri iletme işini geri dönüş değeri ile değil de başka bir aracı kullanarak gerçekleştirir.
İşlevlerin geri dönüş değerlerinin de türleri söz konusudur. İşlevlerin geri dönüş değerleri herhangi bir türden olabilir. Geri dönüş değerlerinin türleri, işlevlerin tanımlanması sırasında belirtilir.

İşlevlerin Tanımlanması

İşlevlerin kodunun yazılması için *tanımlama (definition)* terimi kullanılır. *C*'de işlev tanımlama işleminin genel biçimi şöyledir:

```
[Geri dönüş değerinin türü] <işlev ismi> ([parametreler])
{
    /***/
}
```

Yukarıdaki gösterimde bulunması zorunlu sözdizim elemanları *açısal ayraç* içinde, bulunması zorunlu olmayan sözdizim elemanları ise *köşeli ayraç* içinde belirtilmiştir. Tanımlanan işlevler en az bir blok içermelidir. Bu bloğa işlevin ana bloğu denir. Ana blok içinde istenildiği kadar iç içe blok yaratılabilir. Aşağıdaki işlev tanımından *func* isimli işlevinin parametre almadığı ve geri dönüş değerinin de *double* türden olduğu anlaşılır.

```
double func()
{
}
```

Yukarıdaki tanımı inceleyin. Önce işlevin geri dönüş değerinin türünü gösteren anahtar sözcük yazılır. Bildirim ve tanımlama konusunda anlatılan C'nin doğal türlerini belirten anahtar sözcük(ler) ile işlevin hangi türden bir geri dönüş değeri ürettiği belirtilir. Yukarıda tanımlanan *func* isimli işlevin geri dönüş değeri *double* türdendir.
Daha sonra işlevin ismi yazılır. İşlevin ismi C dilinin isimlendirme kurallarına uygun olarak seçilmelidir. Geleneksel olarak işlev isimleri de, değişken isimleri gibi küçük harf yoğun olarak seçilirler. İşlev ismini izleyen, açılan ve kapanan ayraçlara işlevin parametre ayraçları denir. Bu ayracın içinde, işlevin parametre değişkenleri denen değişkenlerin bildirimi yapılır. *func* isimli işlevin parametre ayracının içinin boş bırakılması bu işlevin parametre değişkenine sahip olmadığını gösteriyor. Parametre ayracını açılan ve kapanan küme ayraçları, yani bir blok izliyor. İşte bu bloğa da işlevin ana bloğu (*main block)* denir. Bu bloğun içine işlevin kodları yazılır.

void Anahtar Sözcüğü

Tanımlanan bir işlevin bir geri dönüş değeri üretmesi zorunlu değildir. İşlev tanımında bu durum geri dönüş değerinin türünün yazıldığı yere *void* anahtar sözcüğünün yazılmasıyla anlatılır:

```
void func()
{
}
```

Yukarıda tanımlanan *func* işlevi geri dönüş değeri üretmiyor. Geri dönüş değeri üretmeyen işlevlere *void* işlevler denir.

İşlev tanımında geri dönüş değerinin türü bilgisi yazılmayabilir. Bu durum, işlevin geri dönüş değeri üretmediği anlamına gelmez. Eğer geri dönüş değeri tür bilgisi yazılmaz ise, C derleyicileri tanımlanan işlevin *int* türden bir geri dönüş değerine sahip olduğunu varsayar. Örneğin:

```
func()
{

}
```

Yukarıda tanımlanan *func* işlevinin geri dönüş değerinin türü *int* türüdür. Yani işlevin yukarıdaki tanımıyla

```
int func()
{
}
```

tanımı arasında derleyici açısından bir fark yoktur. Geri dönüş değerinin türünün yazılmaması geçmişe doğru uyumluluk için korunan bir kuraldır. *int* türüne geri dönen bir işlevin tanımında *int* sözcüğünün yazılması tavsiye edilir.
[C++ dilinde işlev tanımında geri dönüş değerinin türünün yazılması zorunludur.]

İşlevlerin Tanımlanma Yerleri

C dilinde bir işlev tanımı içinde bir başka işlev tanımlanamaz. Yani içsel işlev tanımlamalarına izin verilmez. Örneğin aşağıdaki gibi bir tanımlama geçersizdir, çünkü *func* işlevi tanımlanmakta olan *foo* işlevinin içinde tanımlanıyor:

```
double foo()
{
    /***/
    int func()  /* Geçersiz */
    {
        /***/
    }
    /***/
}
```

Tanımlamanın aşağıdaki biçimde yapılması gerekirdi:

```
double foo()
{
    /***/
}

int func()
{
    /***/
}
```

İşlevlerin Geri Dönüş Değerlerinin Oluşturulması

C dilinde işlevlerin geri dönüş değerleri *return* deyimi (*return statement)* ile oluşturulur. *return* deyiminin iki ayrı biçimi vardır:

```
return;
```

Ya da *return* anahtar sözcüğünü bir ifade izler:

```
return x * y;
```

return anahtar sözcüğünün yanındaki ifadenin değeri, geri dönüş değeri olarak, işlevi çağıran kod parçasına iletilir.

return ifadesinin değişken içermesi bir zorunluluk değildir. Bir işlev bir değişmez değerle de geri dönebilir.

```
return 1;
```

return deyiminin bir başka işlevi de içinde bulunduğu işlevi sonlandırmasıdır. Bir işlevin kodunun yürütülmesi sırasında *return* deyimi görüldüğünde işlevin çalışması sona erer.

```
int func()
{
    /**/
    return x * y;
}
```

Yukarıdaki örnekteki *func* işlevinde *return* anahtar sözcüğünün yanında yer alan *x * y* ifadesi ile oluşturulan *return* deyimi, *func* işlevini sonlandırıyor, *func* işlevinin bir geri dönüş değeri üretmesini sağlıyor.

Bazı programcılar *return* ifadesini bir ayraç içinde yazarlar. Bu ayraç *return* deyimine ek bir anlam katmaz. Yani

```
return x * y;
```

gibi bir deyim

```
return (x * y);
```

biçiminde de yazılabilir. Okunabilirlik açısından özellikle uzun *return* ifadelerinde ayraç kullanımı salık verilir.

```
return (a * b - c * d);
```

Bir işlevin tanımında işlevin geri dönüş değeri türü yazılmışsa, bu işlevin tanımı içinde *return* deyimiyle bir geri dönüş değeri üretilmelidir. Bu mantıksal bir gerekliliktir. Ancak *return* deyimiyle bir geri dönüş değeri üretilmemesi derleme zamanı hatasına neden olmaz. Bu durumda işlevin çalışması işlevin ana bloğunun sonuna gelindiğinde sona erer , işlev çağrıldığı yere bir çöp değeri iletir. Bu da istenmeyen bir durumdur. C derleyicilerinin çoğu, geri dönüş değeri üreteceği yolunda bilgi verilen bir işlevin *return* deyimiyle bir değer üretmemesini mantıksal bir uyarı iletisiyle işaretler.

"Warning: Function func should return a value"

Geri dönüş değeri üretmeyen işlevlerde, yani *void* işlevlerde, *return* anahtar sözcüğü yanında bir ifade olmaksızın tek başına da kullanılabilir:

```
return;
```

Bu durumda *return* deyimi içinde yer aldığı işlevi, geri dönüş değeri oluşturmadan sonlandırır.
C dilinde işlevler yalnızca tek bir geri dönüş değeri üretebilir. Bu da işlevlerin kendilerini çağıran işlevlere ancak bir tane değeri geri gönderebilmeleri anlamına gelir. Ancak, işlevlerin birden fazla değeri ya da bilgiyi kendilerini çağıran işlevlere iletmeleri gerekiyorsa, C dilinde bunu sağlayacak başka araçlar vardır. Bu araçları daha sonraki bölümlerde ayrıntılı olarak göreceksiniz.

main İşlevi

main de diğer işlevler gibi bir işlevdir, aynı tanımlama kurallarına uyar. C programlarının çalışması, ismi *main* olan işlevden başlar. C programları özel bir işlem yapılmamışsa, *main* işlevinin çalışmasının bitişiyle sonlanır. *main* işlevine sahip olmayan bir kaynak dosyanın derlenmesinde bir sorun çıkmaz. Ancak bağlama *(linking)* aşamasında bağlayıcı *main* işlevinin bulunmadığını görünce bağlama işlemini gerçekleştiremez. Bağlayıcı programlar bu durumda bir hata iletisi verir.

```
int main()
{
    return 0;
}
```

Biçiminde tanımlanmış bir *main* işlevi de *int* türden bir değer döndürmelidir. *main* işlevinin ürettiği geri dönüş değeri, programın çalışması bittikten sonra işletim sistemine iletilir.
Geleneksel olarak, *main* işlevinin *0* değerine geri dönmesi programın sorunsuz bir şekilde sonlandırıldığı anlamına gelir. *main* işlevinin *0* dışında bir değere geri dönmesi ise, kodu okuyan tarafından programın başarısızlıkla sona erdirildiği biçiminde yorumlanır. Yani bazı nedenlerle yapılmak istenenler yapılamamış, bu nedenle *main* işlevi sonlandırılmıştır.

main işlevi geri dönüş değeri üreteceğini bildirmiş olmasına karşın *return* deyimiyle belirli bir değeri geri döndürmezse, *main* işlevinden de bir çöp değeri gönderilir.
Derleyicilerin çoğu, *main* işlevinin geri dönüş değeri üretmemesi durumunda da bir mantıksal uyarı iletisi üretir.
Diğer taraftan *main* işlevi de *void* bir işlev olarak tanımlanabilir:

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

Ancak bir sözdizimi hatası olmamasına karşın *main* işlevinin *void* bir işlev olarak tanımlanması doğru kabul edilmez.

İşlevlerin Çağrılması

C dilinde bir işlev çağrısı (*function call - function invocation),* ismi işlev çağrı işleci olan bir işleç ile yapılır. İşlev çağrı işleci olarak *()* atomları kullanılır. Çağrılacak işlevin ismi bu işleçten önce yazılır.

```
func();
```

Yukarıda deyim ile *func* isimli işlev çağrılır.

Bir işlev çağrıldığı zaman programın akışı, işlevin kodunun yürütülmesi için bellekte işlevin kodunun bulunduğu bölgeye atlar. İşlevin kodunun çalıştırılması işlemi bittikten sonra da akış yine çağıran işlevin kalınan yerinden sürer.
Bir işlevin geri dönüş değeri varsa, işlev çağrı ifadesi, işlevin geri dönüş değerini üretir. Geri dönüş değeri üreten bir işleve yapılan çağrı ifadesi söz konusu işlevin ürettiği geri dönüş değerine eşdeğerdir.
İşlevin geri dönüş değeri bir değişkene atanabileceği gibi doğrudan aritmetik işlemlerde de kullanılabilir. Örneğin:

```
sonuc = hesapla();
```

Burada *hesapla* işlevine yapılan çağrı ifadesiyle üretilen geri dönüş değeri, *sonuc* isimli değişkene atanır. Bir başka deyişle bir işlev çağrı ifadesinin ürettiği değer, ilgili işlevin ürettiği (eğer üretiyorsa) geri dönüş değeridir. Yukarıdaki örnekte önce *hesapla()* işlevi çağrılır, daha sonra işlevin kodunun çalıştırılmasıyla elde edilen geri dönüş değeri *sonuc* değişkenine atanır.
İşlev çağrı ifadeleri nesne göstermez yani sol taraf değeri *(L value)* değildir. Yani *C* dilinde aşağıdaki gibi bir atama deyimi geçersizdir:

```
func() = 5;     /* Geçersiz */
```

İşlevlerin geri dönüş değerleri sağ taraf değeridir.

```
sonuc = func1() + func2() + x + 10;
```

gibi bir ifade geçerlidir. Çağrılmış olan *func1* ve *func2* işlevleri çalıştırılarak üretilen geri dönüş değerleri ile *x* değişkeni içindeki değer ve *10* değişmezi toplanır. İfadeden elde edilen değer, *sonuc* isimli değişkene atanır.

İşlev Çağrılarının Yeri

İşlevler, ancak tanımlanmış işlevlerin içinden çağrılabilirler. Blokların dışından işlev çağrılamaz.
[C++ dilinde blok dışında yazılan ilkdeğer verme deyimlerinde ilkdeğer verici (initializer) ifade bir işlev çağrısı olabilir.]

Çağıran işlev ile çağrılan işlevin her ikisi de aynı amaç kod (*object code*) içinde bulunmak zorunda değildir. Çağıran işlev ile çağrılan işlev farklı amaç kodlar içinde bulunabilir. Çünkü derleme işlemi sırasında bir işlevin çağrıldığını gören derleyici, amaç kod içine çağrılan işlevin adını ve çağrı biçimini yazar. Çağıran işlev ile çağrılan işlev arasında bağlantı kurma işlemi, bağlama aşamasında bağlayıcı program *(linker)* tarafından yapılır. Bu nedenle tanımlanan bir işlev içinde, tanımlı olmayan bir işlev çağrılsa bile derleme aşamasında bir hata oluşmaz. Hata bağlama aşamasında oluşur. Çünkü bağlayıcı çağrılan işlevi bulamaz.

İşlev Parametre Değişkenlerinin Tanımlanması

İşlevler çağrıldıkları yerlerden alacakları bilgileri, işlev çağrı ifadeleri ile alırlar. Bir işlevin formal parametreleri (*formal parameters)* ya da *parametre değişkenleri*, işlevlerin kendilerini çağıran işlevlerden aldıkları girdileri tutan değişkenleridir. Bir işlevin parametre sayısı ve bu parametrelerin türleri gibi bilgiler, işlevlerin tanımlanması sırasında derleyiciye bildirilir. İşlev çağrısı ile gönderilen argüman ifadelerin değerleri, işlevin ilgili parametre değişkenlerine kopyalanır.
Örneğin bir küpün hacmini hesaplayan işlev, çağrıldığı yerden bir küpün kenar uzunluğunu alacağına göre, bu değerin kopyalanması için, bir parametre değişkenine sahip olması gerekir. Benzer şekilde iki sayıdan daha büyük olanını bulan bir işlevin iki tane parametre değişkenine sahip olması gerekir.
C dilinde işlevlerin tanımlanmasında kullanılan iki temel biçim vardır. Bu biçimler birbirlerinden işlev parametrelerinin derleyicilere tanıtılma şekli ile ayrılırlar. Bu biçimlerden birincisi eski biçim *(old style)* ikincisi ise yeni biçim *(new style)* olarak adlandırılır.
Eski biçim hemen hemen hiç kullanılmaz, ama C standartlarına göre halen geçerlidir. Bu biçimin korunmasının nedeni geçmişe doğru uyumluluğun sağlanmasıdır. Kullanılması gereken kesinlikle yeni biçimdir. Ancak eski kodların ya da eski kaynak kitapların incelenmesi durumunda bunların anlaşılabilmesi için eski biçimin de öğrenilmesi gerekir.

Eski Biçim

Eski biçimde (*old style)*, işlevin parametre değişkenlerinin yalnızca isimleri, işlev parametre ayraçları içinde yazılır. Eğer parametre değişkenleri birden fazla ise aralarına virgül atomu koyulur. Daha sonra bu değişkenlerin bildirimi yapılır. Bu bildirimler daha önce öğrendiğimiz, C dilinin bildirim kurallarına uygun olarak yapılır. Örnek:

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

Yukarıda tanımlanan alan işlevinin iki parametre değişkeni vardır ve bu parametre değişkenlerinin isimleri *x* ve *y*'dir. Her iki parametre değişkeni de *double* türdendir. İşlevin geri dönüş değeri *double* türdendir.

```
int func (a, b, c)
int a;
double b;
long c;
{
    /***/
}
```

Bu örnekte ise *func* işlevi üç parametre değişkenine sahiptir. Parametre değişkenlerinin isimleri *a*, *b* ve *c*'dir. İsmi *a* olan parametre değişkeni *int* türden, *b* olanı *double* türden ve ismi *c* olanı ise *long* türdendir.

Eski biçim, yeni biçime göre uzundur. Çünkü işlev ayraçlarının içinde ismi yer alan parametre değişkenleri alt satırlarda yeniden bildirilir.

Yeni Biçim

Yeni biçimde (*new style*), eski biçime göre hem daha kısadır hem de okunabilmesi eski biçime göre çok daha kolaydır.

Yeni biçimde işlev parametre değişkenlerinin bildirimi işlev ayraçlarının içinde yalnızca bir kez yapılır. Bu biçimde, işlevin ayraçlarının içine parametre değişkenin türü ve yanına da ismi yazılır. Eğer birden fazla işlev parametre değişkeni varsa bunlar virgüllerle ayrılır, ancak her bir değişkenin tür bilgisi yeniden yazılır. Örnek :

```
int func(int x, int y)
{
    /***/
}

double foo(double a, int b)
{
    /***/
}
```

İşlev parametre değişkenleri aynı türden olsa bile her defasında tür bilgisinin yeniden yazılması zorunludur. Örneğin:

```
int foo (double x, y)  /* Geçersiz */
{
    /***/
}
```

bildirimi hatalıdır. Doğru tanımlama aşağıdaki biçimde olmalıdır:

```
int foo (double x, double y)
{
    /***/
}
```

[C++ dilinde eski biçim işlev tanımlamaları geçerli değildir.]

Parametre Değişkenine Sahip Olmayan İşlevler

Her işlev parametre değişkenine sahip olmak zorunda değildir. Bazı işlevler istenen bir işi yapabilmek için dışarıdan bilgi almaz. Parametre değişkenine sahip olmayan bir işlevin tanımında, işlev parametre ayracının içi boş bırakılır. İşlev parametre ayracının içine *void* anahtar sözcüğünün yazılması durumunda da işlevin parametre değişkenine sahip olmadığı sonucu çıkar.

```
int foo()
{
    /***/
}
```

ile

```
int foo(void)
{
    /***/
}
```

tamamen aynı anlamdadır

Argümanların Parametre Değişkenlerine Kopyalanması

Bir işlevin parametre değişkenleri, o işlevin çağrılma ifadesiyle kendisine gönderilen argümanları tutacak olan yerel değişkenlerdir. Örnek:

```
void func(int a)
{
    /***/
}

int main()
{
    int x = 10;
    /***/
    func (x);

    return 0;
}
```

Yukarıdaki örnekte *main* işlevi içinde, *func* isimli işlev çağrılıyor ve çağrılan işleve *x* isimli değişkenin değeri argüman olarak geçiliyor. İşlev çağrısı, programın çalışma zamanında, programın akışının *func* işlevinin kodunun bulunduğu yere sıçramasına neden olur. *func* işlevindeki *a* isimli parametre değişkeni için bellekte bir yer ayrılır ve *a* parametre değişkenine argüman olan ifadenin değeri, yani *x* değişkeninin değeri atanır. Yani

```
int a = x;
```

işleminin derleyicinin ürettiği kod sonucu otomatik olarak yapıldığı söylenebilir.

```
int main()
{
    int x = 100, y = 200, z;

    z = add(x, y);
    /***/

    return 0;
}

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

Yukarıda tanımlanan *add* işlevi çağrıldığında programın akışı bu işleve geçmeden önce, *x* ve *y* değişkenlerinin içinde bulunan değerler, *add* işlevinin parametre değişkenleri olan *a* ve *b*'ye kopyalanır.

İşlev Çağrı İfadelerinin Kullanımları

1. Geri dönüş değeri üretmeyen bir işleve yapılan çağrı, genellikle kendi başına bir deyim oluşturur. Böyle bir işleve yapılan çağrı, bir ifadenin parçası olarak kullanılmaz. İşlev çağrısını genellikle sonlandırıcı atom izler;

```
void func()
{
    /***/
}

int main()
{
    func();
    return 0;
}
```

2. Geri dönüş değeri üreten işlevlerin ürettiği değerlerin kullanılması zorunlu değildir. Ancak bazı işlevlerin geri dönüş değerlerinin kullanılmaması mantıksal bir yanlışlık olabilir. İki sayının toplamı değerine geri dönen bir işlev tanımlandığını düşünelim. Böyle bir işlev çağrıldığı yerden iki değer alarak bunların toplamı değerine geri dönüyor olsun. Böyle bir işlevin varlık nedeni bir değer hesaplamaktır. İşlevin hesapladığı değer kullanılmıyorsa işlev boş yere çağrılmış, işlevin kodu boş yere çalışmış olur.
Ancak bazı işlevler bir iş yaptıkları gibi yaptıkları işle ilgili tamamlayıcı bir bilgiyi de geri döndürür. Böyle bir işlev yalnızca yaptığı iş için çağrılabilir. Yani işlevi çağıran kod parçası işlevin geri döndürdüğü değer ile ilgilenmeyebilir.

Örneğin *foo()* işlevi *int* türden bir değeri geri dönüş değeri üreten işlev olsun.

```
a = foo();
```

Yukarıdaki ifadede *foo* işlevinin geri dönüş değeri *a* isimli değişkene atanır. Bu işlev bir kez çağrılmasına karşın artık geri dönüş değeri *a* değişkeninde tutulduğu için, bu geri dönüş değerine işlev yeniden çağrılmadan istenildiği zaman ulaşılabilir. Ancak:

```
foo();
```

şeklinde bir işlev çağrı ifadesinde geri dönüş değeri bir değişkende saklanmaz. Bu duruma geri dönüş değerinin kullanılmaması *(discarded return value)* denir.

3. Sık karşılaşılan durumlardan biri de, bir işlev çağrısıyla elde edilen geri dönüş değerinin bir başka işlev çağrısında argüman olarak kullanılmasıdır. Aşağıdaki örneği inceleyin:

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

int square(int a)
{
    return a * a;
}

int main()
{
    int x = 10;
    int y = 25;
    int z = square(add(x, y));

    return 0;
}
```

Yukarıda tanımlanan *add* işlevi, iki tamsayının toplamı değeri ile geri dönerken, *square* işlevi ise dışarıdan aldığı değerin karesi ile geri dönüyor. *main* işlevi içinde yapılan

```
square(add(x, y));
```

çağrısı ile *add* işlevinin geri dönüş değeri *square* işlevine argüman olarak geçiliyor.

4. Bir işlevin ürettiği geri dönüş değeri bir başka işlevin *return* deyiminde *return* ifadesi olarak kullanılabilir. Bir başka deyişle, bir işlev geri dönüş değerini bir başka işlevi çağırarak oluşturabilir. İki sayının karelerinin toplamına geri dönen *sum_square* isimli bir işlev tanımlanmak istensin:

```
int sum_square(int a, int b)
{
    return add(square(a), square(b));
}
```

Tanımlanan *sum_square* işlevi daha önce tanımlanmış *add* işlevine yapılan çağrının ürettiği geri dönüş değeri ile geri dönüyor. *add* işlevine gönderilen argümanların da, *square* işlevine yapılan çağrılardan elde edildiğini görüyorsunuz.

İşlevlerin Kendi Kendini Çağırması

Bir işlev kendisini de çağırabilir. Kendisini çağıran bir işleve özyinelemeli işlev (*recursive function*) denir. Bir işlev kendini neden çağırır? Böyle işlevlerle hedeflenen nedir? Bu konu ileride ayrı bir başlık altında ele alınacak.

Standart C İşlevleri

Standard C işlevleri, C dilinin standartlaştırılmasından sonra, her derleyicide bulunması zorunlu hale getirilmiş işlevlerdir. Yani derleyicileri yazanlar standart C işlevlerini kendi derleyicilerinde mutlaka tanımlamak zorundadırlar. Bu durum C dilinin taşınabilirliğini (*portability*) artıran ana etmenlerden biridir.

Bir işlevin derleyiciyi yazanlar tarafından tanımlanmış ve derleyici paketine eklenmiş olması, o işlevin standart C işlevi olduğu anlamına gelmez. Derleyiciyi yazanlar programcının işini kolaylaştırmak için çok çeşitli işlevleri yazarak derleyici paketlerine eklerler. Ama bu tür işlevlerin kullanılması durumunda, oluşturulan kaynak kodun başka bir derleyicide derlenebilmesi yönünde bir güvence yoktur, yani artık kaynak kodun taşınabilirliği azalır. Örneğin *printf* işlevi standart bir *C* işlevidir. Yani *printf* işlevi her derleyici paketinde aynı isimle bulunmak zorundadır.

Standart C işlevlerinin derlenmiş kodları özel kütüphanelerin içindedir. Başlık dosyaları içinde, yani uzantısı *.h* biçiminde olan dosyaların içinde standart C işlevlerinin bildirimleri bulunur. İşlev bildirimi konusu ileride ayrıntılı bir biçimde incelenecek.
Kütüphaneler (*libraries*), derlenmiş dosyalardan oluşur. *DOS* işletim sisteminde kütüphane dosyalarının uzantısı *.lib*, *UNIX* işletim sisteminde ise *.a (archive)* biçimindedir. *WINDOWS* altında uzantısı *.dll* biçiminde olan dinamik kütüphaneler de bulunur.

Derleyicileri yazanlar tarafından kaynak kodu yazılmış standart *C* işlevleri önce derlenerek *.obj* haline getirilirler ve daha sonra aynı gruptaki diğer işlevlerin *.obj* halleriyle birlikte kütüphane dosyalarının içine yerleştirilirler. Standart C işlevleri bağlama aşamasında, bağlayıcı *(linker)* tarafından çalışabilir (*.exe*) kod içine yazılırlar. Tümleşik çalışan derleyicilerde bağlayıcılar, amaç kod içinde bulamadıkları işlevleri, yerleri önceden belirlenmiş kütüphaneler içinde arar. Oysa komut satırlı uyarlamalarda (*command line version*) bağlayıcıların hangi kütüphanelere bakacağı komut satırında belirtilir.

Neden Standart İşlevler

Bazı işlevlerin bulunmasının dilin standartları tarafından güvence altına alınması ile aşağıdaki faydalar sağlanmış olur.

i) Bazı işlemler için ortak bir arayüz sağlanmış olur.
Mutlak değer hesaplayan bir işlevi yazmak çok kolaydır. Ancak standart bir C işlevi olan *abs* işlevinin kullanılmasıyla ortak bir arayüz sağlanır. Her kaynak kod kendi mutlak değer hesaplayan işlevini tanımlamış olsaydı, tanımlanan işlevlerin isimleri, parametrik yapıları farklı olabilirdi. Bu durum da kod okuma ve yazma süresini uzatırdı.

ii) Bazı işleri gerçekleştirecek işlevlerin kodları sistemden sisteme farklılık gösterebilir. Bu da kaynak dosyanın taşınabilirliğini azaltır. Bu işlemleri yapan standart işlevlerin tanımlanmış olması kaynak kodun başka sistemlere taşınabilirliği artırır.

iii) Bazı işlevlerin yazılması belirli bir alanda bilgi sahibi olmayı gerektirebilir. Örneğin bir gerçek sayının bir başka gerçek sayı üssünü hesaplayan bir işlevi verimli bir şekilde yazabilmek için yeterli matematik bilgisine sahip olmak gerekir.

iv) Sık yapılan işlemlerin standart olarak tanımlanmış olması, programcının yazacağı kod miktarını azaltır. Böylece proje geliştirme süresi de kısalır.

v) Derleyicilerin sağladığı standart işlevler çok sayıda programcı tarafından kullanılmış olduğu için çok iyi derecede test edilmişlerdir. Bu işlevlerin tanımlarında bir hata olma olasılığı, programcının kendi yazacağı işlevlerle kıyaslandığında çok düşüktür.

İyi bir C programcısının C dilinin standart işlevlerini çok iyi tanıması ve bu işlevleri yetkin bir şekilde kullanabilmesi gerekir.

printf İşlevi

printf standart bir C işlevidir. *printf* işlevi ile ekrana bir yazı yazdırılabileceği gibi, bir ifadenin değeri de yazdırılabilir.
Değişkenlerin içindeki değerler aslında bellekte ikilik sistemde tutulur. Bir değişkenin değerinin ekrana, hangi sayı sisteminde ve nasıl yazdırılacağı programcının isteğine bağlıdır. Değişkenlerin değerlerinin ekrana yazdırılmasında standart *printf* işlevi kullanılır. *printf* aslında çok ayrıntılı özelliklere sahip bir işlevdir. Şimdilik işinize yarayacak kadar ayrıntıyı öğreneceksiniz. *printf* işlevlerle ilgili yukarıda açıklanan genel kurallara uymaz. *printf* işlevi değişken sayıda parametreye sahip bir işlevdir. Bir işlevin kaç tane parametre değişkeni varsa o işlev çağrıldığında, işleve o kadar argüman geçilmelidir, değil mi? Oysa *printf* işlevine istenen sayıda argüman geçilebilir. Bu işleve kaç tane argüman geçilirse işlevin o kadar sayıda parametre değişkenine sahip olacağı düşünülebilir. Bu nasıl oluyor? Değişken sayıda parametreye sahip işlevler ileri bir konu olduğundan, bu konu ancak sonraki bölümlerde ele alınacak.
printf işlevine ilk gönderilen argüman genellikle çift tırnak içinde yer alan bir yazıdır. Çift tırnak içinde yer alan böyle yazılara *dizge (string)* denir. Dizgeler konusu ileride ayrı bir bölümde ele alınacak.
printf işlevine argüman olarak geçilen dizge içinde yer alan tüm karakterler ekrana yazılır. Ancak *printf* işlevi dizge içindeki % karakterini ve bunu izleyen belirli sayıda karakteri ekrana yazmaz. İşlev, dizge içindeki % karakterlerini yanındaki belirli sayıda karakter ile birlikte *formatlama* karakterleri (*conversion specifiers*) olarak yorumlar. Formatlama karakterleri, çift tırnaktan sonra yazılan argümanlarla bire bir eşleştirilir. Formatlama karakterleri önceden belirlenmiştir, kendileriyle eşlenen bir ifadenin değerinin ekrana ne şekilde yazdırılacağı bilgisini işleve aktarırlar. Bu format bilgisi

* Argüman olan ifadenin hangi türden olarak yorumlanacağı
* İfadenin değerinin ekrana hangi sayı sistemi kullanılarak yazılacağı
* İfadenin kaç karakterlik bir alana yazdırılacağı
* Pozitif tamsayıların yazımında '+' karakterinin yazdırılıp yazdırılmayacağı
* Gerçek sayıların yazımında üstel notasyonun kullanılıp kullanılmayacağı
* Gerçek sayıların yazımında noktadan sonra kaç basamağın yazılacağı

gibi açıklamalardır. Aşağıdaki programı inceleyin:

```
#include <stdio.h>

int main()
{
    int x = 25;
    double pi = 3.1415;

    printf("x = %d\npi = %lf\n", x, pi);

    return 0;
}
```

main işlevi içinde yapılan

```
printf("x = %d\npi = %lf\n", x, pi);
```

çağrısında işleve gönderilen birinci argüman olan çift tırnak içindeki yazıda iki ayrı format dizgesi kullanılıyor: *%d* ve *%lf.*
%d format karakterleri ikinci argüman olan *x* ile, *%lf* format karakterleri ise 3. argüman olan *pi* ile eşleniyor. Format karakterleri ile eşlenen ifadelerin değerleri, istenen formatlama özellikleri ile ekrana yazılır. Örneğin yukarıdaki çağrıyla ekrana

```
x = 25
pi = 3.14150
```

yazısı yazdırılır.
Aşağıda formatlama karakterlerinden bazılarının anlamı veriliyor. *printf* işlevi ileride ayrıntılı olarak ele alınacak.

Format karakteri	Anlamı
%d	Bir ifadeyi *int* türden yorumlayarak, elde ettiği değeri onluk sayı sisteminde yazar.
%ld	Bir ifadeyi *long* türden yorumlayarak, elde ettiği değeri onluk sayı sisteminde yazar.
%x	Bir ifadeyi *unsigned int* türden yorumlayarak, elde ettiği değeri onaltılık sayı sisteminde yazar. Basamak sembolleri olarak *a, b, c, d, e, f* (küçük) harflerini kullanır.
%X	Bir ifadeyi *unsigned int* türden yorumlayarak, elde ettiği değeri onaltılık sayı sisteminde yazar. Basamak simgeleri olarak *A, B, C, D, E, F* (büyük) harflerini kullanır.
%lx	Bir ifadeyi unsigned long türünden yorumlayarak, onaltılık sayı sisteminde yazar.
%u	Bir ifadeyi *unsigned int* türünden yorumlayarak, onluk sayı sisteminde yazar.
%o	Bir ifadeyi unsigned int türünden yorumlayarak, sekizlik sayı sisteminde yazar
%f	*float* ve *double* türlerinden ifadelerin değerlerini onluk sayı sisteminde yazar.
%lf	*double* ve *long double* türlerinden ifadelerin değerlerini onluk sayı sisteminde yazar.
%e	Gerçek sayıları üstel biçimde yazar.
%c	*char* veya *int* türünden bir ifadeyi bir karakterin sıra numarası olarak yorumlayarak, ilgili karakterin görüntüsü ekrana yazdırır.
%s	Verilen adresteki yazıyı ekrana yazdırır.

Yukarıdaki tabloda görüldüğü gibi *double* türü hem *%f* format karakteri hem de *%lf* format karakteri ile yazdırılabilir. Ama *%lf* okunabilirliği artırdığı için daha çok tercih edilir.

Yukarıdaki tabloya göre *unsigned int* türünden *u* isimli değişkenin değeri aşağıdaki şekillerde yazdırabilir:

```
#include <stdio.h>

int main()
{
    unsigned int u = 57054;
    printf("u = %u\n", u); /* u değerini onluk sistemde yazar */
    printf("u = %o\n", u); /* u değerini sekizlik sistemde yazar */
    printf("u = %X\n", u); /* u değerini onaltılık sistemde yazar */

    return 0;
}
```

long türden bir ifadenin değerini yazdırırken *d, o, u* ya da *x* karakterlerinden önce *l* karakteri kullanılır:

```
#include <stdio.h>

int main()
{
    long int lo = 23467;
    unsigned long int unlo = 65242;

    printf("unlo = %ld\n", lo);       /* onluk sistemde yazar */
    printf("unlo = %lu\n", unlo);     /* onluk sistemde yazar */
    printf("unlo = %lo\n", unlo);     /* sekizlik sistemde yazar */
    printf("unlo = %lX\n", unlo);     /* onaltılık sistemde yazar */

    return 0;
}
```

Yukarıdaki örneklerde *unsigned int* türden bir ifadenin değerinin *printf* işleviyle sekizlik ya da onaltılık sayı sisteminde yazdırılabileceğini gördünüz. Peki işaretli türden bir tamsayının değeri sekizlik ya da onaltılık sistemde yazdırılamaz mı? Yazdırılırsa ne olur? Söz konusu işaretli tamsayı pozitif olduğu sürece bir sorun olmaz. Sayının işaret biti *0* olduğu için sayının nicel büyüklüğünü etkilemez. Yani doğru değer ekrana yazılır, ama sayı negatifse işaret biti *1* demektir. Bu durumda ekrana yazılacak sayının işaret biti de nicel büyüklüğün bir parçası olarak değerlendirilerek yazılır. Yani yazılan değer doğru olmaz.

% karakterinin yanında önceden belirlenmiş bir format karakteri yoksa , % karakterinin yanındaki karakter ekrana yazılır.
Yüzde karakterinin kendisini ekrana yazdırmak için format karakteri olarak %% kullanılır:

```
printf("%%25\n");
```

scanf İşlevi

scanf işlevi, klavyeden her türlü bilginin girişine olanak tanıyan standart bir C işlevidir. *scanf* işlevi de *printf* işlevi gibi aslında çok ayrıntılı, geniş kullanım özellikleri olan bir işlevdir. Ancak bu noktada *scanf* işlevi yüzeysel olarak ele alınacak.

scanf işlevinin de birinci parametresi bir dizgedir. Ancak bu dizge yalnızca klavyeden alınacak bilgilere ilişkin format karakterlerini içerir. *printf* işlevinde olduğu gibi *scanf* işlevinde de bu format karakterleri önceden belirlenmiştir. % karakterinin yanında yer alırlar. *scanf* işlevinin kullandığı format karakterlerinin *printf* işlevinde kullanılanlar ile hemen hemen aynı olduğu söylenebilir. Yalnızca gerçek sayılara ilişkin format karakterlerinde önemli bir farklılık vardır. *printf* işlevi *%f* formatı ile hem *float* hem de *double* türden verileri ekrana yazabilirken *scanf* işlevi *%f* format karakterini yalnızca float türden veriler için kullanır. *double* türü için *scanf* işlevinin kullandığı format karakterleri *%lf* şeklindedir. *scanf* işlevinin format kısmında format karakterlerinden başka bir şey olmamalıdır. *printf* işlevi çift tırnak içindeki format karakterleri dışındaki karakterleri ekrana yazıyordu, ancak *scanf* işlevi format karakterleri dışında dizge içine yazılan karakterleri ekrana basmaz, bu karakterler tamamen başka anlama gelir. Bu nedenle işlevin nasıl çalıştığını öğrenmeden bu bölgeye format karakterlerinden başka bir şey koymayın. Buraya konulacak bir boşluk bile farklı anlama gelir.

```
#include <stdio.h>

int main()
{
    int x, y;

    printf("iki sayı girin : ");
    scanf("%d%d", &x, &y);
    printf("%d + %d = %d\n", x, y, x + y);

    return 0;
}
```

Yukarıdaki örnekte, programı kullanan kişiye değer girmesinin beklendiğini söyleyen bir yazı, *printf* işleviyle ekrana yazdırılıyor. Bu iş *scanf* işlevi ile yapılmazdı. *scanf* işlevi ile ekrana bir yazı yazdırmak mümkün değildir. *scanf* yalnızca giriş amacıyla tanımlanmış bir işlevdir, çıkış işlemi yapmaz.

```
scanf("%d%d", &x, &y);
```

çağrısı ile programın çalışma zamanında klavyeden girilecek değerler *x* ve y değişkenlerine aktarılır. *x* ve *y* değişkenleri için onluk sayı sisteminde klavyeden giriş yapılır. Giriş arasına istenildiği kadar boşluk karakteri konulabilir. Yani ilk sayıyı girdikten sonra *SPACE*, *TAB* ya da *ENTER* tuşuna bastıktan sonra ikinci değer girilebilir. Örneğin:

```
5 60
```

biçiminde bir giriş, geçerli olacağı gibi;

```
5
60
```

biçiminde bir giriş de geçerlidir. *scanf* işlevine gönderilecek diğer argümanlar & adres işleci ile kullanılır. & bir gösterici işlecidir. Bu işleci göstericiler konusunda öğreneceksiniz.

Klavyeden Karakter Alan C İşlevleri

Sistemlerin hemen hemen hepsinde klavyeden karakter alan üç ayrı C işlevi bulunur. Bu işlevlerin biri tam olarak standarttır ama diğer ikisi sistemlerin hemen hemen hepsinde bulunmasına karşın standart C işlevi değildir. Şimdi bu işlevleri inceleyelim:

getchar İşlevi

Standart bu işlevin parametrik yapısı aşağıdaki gibidir:

```
int getchar(void);
```

İşlevin geri dönüş değeri klavyeden alınan karakterin, kullanılan karakter seti tablosundaki sıra numarasını gösteren *int* türden bir değerdir. *getchar* işlevi klavyeden karakter almak için *enter* tuşuna gereksinim duyar.
Aşağıda yazılan programda önce *getchar* işleviyle klavyeden bir karakter alınıyor daha sonra alınan karakter ve karakterin sayısal değeri ekrana yazdırılıyor.

```
#include <stdio.h>

int main()
{
    char ch;

    ch = getchar();
    printf("\nKarakter olarak ch = %c\nASCII numarası ch = %d\n", ch, ch);

    return 0;
}
```

getchar derleyicilerin çoğunda *stdio.h* başlık dosyası içinde bir makro olarak tanımlanır. Makrolar ile ileride tanışacaksınız.

getch İşlevi

Standart olmayan bu işlevin parametrik yapısı çoğunlukla aşağıdaki gibidir:

```
int getch(void);
```

getch standart bir C işlevi olmamasına karşın neredeyse bütün derleyici paketleri tarafından sunulur. Standart *getchar* işlevi gibi *getch* işlevi de klavyeden alınan karakterin kullanılan karakter setindeki sıra numarasıyla geri döner. Sistemlerin çoğunda bu işlevin *getchar* işlevinden iki farkı vardır:

1. Basılan tuş ekranda görünmez.
2. Sistemlerin çoğunda *ENTER* tuşuna gereksinim duymaz.

Yukarıda verilen programda *getchar* yerine *getch* yazarak programı çalıştırırsanız farkı daha iyi görebilirsiniz.

getch işlevi özellikle tuş bekleme ya da onaylama amacıyla kullanılır:

```
printf("devam için herhangi bir tuşa basın...\n");
getch();
```

Burada klavyeden alınan karakterin ne olduğunun bir önemi olmadığı için işlevin geri dönüş değeri kullanılmıyor. Derleyici paketlerinin hemen hepsinde bu işlevin bildirimi standart olmayan *conio.h* isimli başlık dosyasında olduğundan, işlevin çağrıldığı dosyaya *conio.h* başlık dosyası eklenmelidir:

```
#include <conio.h>
```

Bu işlem önişlemci komutları bölümünde ayrıntılı şekilde ele alınacak.

getche İşlevi

Standart olmayan bu işlevin parametrik yapısı çoğunlukla aşağıdaki gibidir:

```
int getche(void);
```

getche İngilizce *get char echo* sözcüklerinden kısaltılmıştır. *getche* işlevi de basılan tuşun karakter setindeki sıra numarasıyla geri döner ve sistemlerin çoğunda *enter* tuşuna gereksinim duymaz. Ama klavyeden alınan karakter ekranda görünür.
Sistemlerin çoğunda

`getchar`	enter tuşuna gereksinim duyar	alınan karakter ekranda görünür.
`getch`	enter tuşuna gereksinim duymaz	alınan karakter ekranda görünmez
`getche`	enter tuşuna gereksinim duymaz	alınan karakter ekranda görünür.

Ekrana Bir Karakterin Görüntüsünü Yazan C İşlevleri

C dilinde ekrana bir karakterin görüntüsünü basmak için bazı standart C işlevleri kullanılabilir:

putchar İşlevi

Bu standart işlevin parametrik yapısı aşağıdaki gibidir:

```
int putchar(int ch);
```

putchar standart bir *C* işlevidir. Bütün sistemlerde bulunması zorunludur. Parametresi olan karakteri ekranda imlecin bulunduğu yere yazar. Örneğin:

```
#include <stdio.h>

int main()
{
    char ch;

    ch = getchar();
    putchar (ch);

    return 0;
}
```

Yukarıdaki kodda *putchar* işlevinin yaptığı iş *printf* işlevine de yaptırılabilirdi;

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

ile

```
putchar(ch)
```

tamamen aynı işi görür.

putchar işlevi ile *'\n'* karakterini yazdırıldığında *printf* işlevinde olduğu gibi imleç sonraki satırın başına geçer. *putchar* işlevi ekrana yazılan karakterin *ASCII* karşılığı ile geri döner.

putchar işlevi derleyicilerin çoğunda *stdio.h* başlık dosyası içinde bir makro olarak tanımlanmıştır.

putch İşlevi

Standart olmayan bu işlevin parametrik yapısı çoğunlukla aşağıdaki gibidir:

```
int putch(int ch);
```

putch standart bir C işlevi değildir. Dolayısıyla sistemlerin hepsinde bulunmayabilir. Bu işlevin *putchar* işlevinden tek farkı *'\n'* karakterinin yazdırılması sırasında ortaya çıkar. *putch*, '\n" karakterine karşılık yalnızca *LF(line feed) (ASCII 10)* karakterini yazar. Bu durum imlecin bulunduğu kolonu değiştirmeksizin aşağı satıra geçmesine yol açar.

Yorum Satırları

Kaynak dosya içinde yer alan önişlemci ya da derleyici programa verilmeyen açıklama amaçlı yazılara yorum satırları (*comment lines*) denir.

Yorum satırları /* atomuyla başlar */ atomuyla sonlanır. Bu iki atom ile, bu iki atom arasında kalan tüm karakterler, önişlemci programın kaynak kodu ele almasından önce tek bir boşluk karakteriyle yer değiştirir. Yorum satırları herhangi sayıda karakter içerebilir. Örnek:

```
/* Bu bir açıklama satırıdır */
```

Yorum satırları birden fazla satıra ilişkin olabilir:

```
/*
    bu satirlar
    kaynak koda dahil
    değildir.
*/
```

Bir dizge ya da bir karakter değişmezi içinde yorum satırı bulunamaz:

```
#include <stdio.h>

int main()
{
    printf("/* bu bir yorum satiri degildir */");

    return 0;
}
```

Yukarıdaki programın derlenip çalıştırılmasıyla, ekrana

```
/* bu bir yorum satırı değil */
```

yazısı yazdırılır.

Bir yorum satırının kapatılmasının unutulması tipik bir hatadır.

```
#include <stdio.h>

int main()
{
    int x = 1;
    int y = 2;

    x = 10;   /* x'e 10 değeri atanıyor
    y = 2;   /* y'ye 20 değeri atanmıyor */

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

    return 0;
}
```

[C++ dilinde yorum satırı oluşturmanın bir başka biçimi daha vardır.
Yorum satırı // karakterleriyle başlar, bulunulan satırın sonuna kadar sürer. Yorum satırını sonlandırılması bulunulan satırın sonu ile olur, yorum satırını sonlandıracak ayrı bir karakter bulunmaz. Örnek:
//Geçerli bir Açıklama satırı
Bu biçim C89 standartlarına göre geçerli değildir ancak C99 standartlarıyla C'ye de eklenmiştir.]

İç İçe Yorum Satırları

İç içe yorum satırları (*nested comment lines*) oluşturmak geçerli değildir:

```
/*
    /*
     */
*/
```

Yukarıdaki örnekte birinci */ atomundan sonraki kod parçası kaynak dosyaya dahildir. Ancak derleyicilerin çoğu uygun ayarların seçilmesiyle iç içe yorum satırlarına izin verir. İç içe yorum satırlarına gereksinim, özellikle bir yorum satırının kopyalanarak başka bir yorum satırı içine yapıştırılması durumunda oluşur. Bazen de, yorum satırı içine alınmak istenen kod parçasının içinde de bir başka yorum satırı olduğundan, içsel yorum satırları oluşur.

Yorum Satırları Neden Kullanılır

Yorum satırları çoğu zaman kaynak kodun okunabilirliğini artırmak için kullanılır. Kaynak koddan doğrudan çıkarılamayan bilgiler açıklama satırlarıyla okuyucuya iletilebilir.
Bazen de yorum satırları bir kaynak dosyanın bölüm başlıklarını oluşturmak amacıyla kullanılır.
Kaynak kodun açıkça verdiği bir bilgiyi, yorum satırıyla açıklamak programın okunabilirliğini bozar.

İŞLEÇLER

İşleç Nedir

İşleçler, nesneler veya değişmezler üzerinde önceden tanımlanmış birtakım işlemleri yapan atomlardır. İşleçler, mikroişlemcinin bir işlem yapmasını ve bu işlem sonunda da bir değer üretilmesini sağlar. Programlama dillerinde tanımlanmış olan her bir işleç en az bir makine komutuna karşılık gelir.

Benzer işlemleri yapmalarına karşılık programlama dillerinde işleç atomları birbirlerinden farklılık gösterebilir.

C programlama dilinde ifadeler çoğunlukla işleçleri de içerirler.

```
c = a * b / 2 + 3      /* 4 işleç vardır ifadedeki sırasıyla =, *, /, +  */
++x * y--              /* 3 işleç vardır, ifadedeki sırasıyla ++, *, --  */
a >= b                 /* 1 işleç vardır. >= */
```

Terim Nedir

İşleçlerin işleme soktukları nesne veya değişmezlere terim (*operand)* denir. C'de işleçler aldıkları terim sayısına göre üç gruba ayrılabilir.

i) Tek terimli işleçler (*unary operators*)
Örneğin ++ ve -- işleçleri tek terimli işleçlerdir.

ii) İki terimli işleçler *(binary operators)*
Aritmetiksel işleçler olan toplama '+' ve bölme '/' işleçleri örnek olarak verilebilir.

iii) Üç terimli işleç (*ternary operator*)
C'de üç terimli tek bir işleç vardır. Bu işlecin ismi "koşul işleci" dir*(conditional operator).*
İşleçler konumlarına göre yani teriminin ya da terimlerinin neresinde bulunduklarına göre de gruplanabilir:

1. Sonek Konumundaki İşleçler (*postfix operators*)
Bu tip işleçler terimlerinin arkasına getirilirler.
Örneğin sonek ++ işleci (*x*++)

2. Önek Konumundaki İşleçler *(prefix operators)*
Bu tip işleçler terimlerinin önüne getirilirler.
Örneğin mantıksal değil işleci (!x)

3. Araek Konumundaki İşleçler (*infix operators*)
Bu tip işleçler terimlerinin aralarına getirilirler.
Örneğin aritmetik toplama işleci (*x* + *y*)

İşleçlerin Değer Üretmesi

İşleçler, yaptıkları işlemin sonucunda bir değer üretir. İşleçlerin ürettiği değer, aynı ifade içinde var olan bir başka işlece terim olabilir. İfade içinde en son değerlendirilen işlecin ürettiği değer ise ifadenin değeri olur. Bir ifadenin değeri, ifade içinde yer alan işleçlerin ürettiği değerlere göre saptanır.
İşleçlerin en önemli özelliği, yaptıkları işlemin sonucu olarak bir değer üretmeleridir. Programcı, bir ifade içinde işleçlerin ürettiği değeri kullanır ya da kullanmaz. İşleçlerin ürettiği değer aşağıdaki biçimlerde kullanılabilir:

i. İşlecin ürettiği değer bir başka değişkene aktarılabilir:

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

Yukarıdaki örnekte *y + z* ifadesinin değeri, yani + işlecinin ürettiği değer, *x* değişkenine aktarılır.

ii. Üretilen değeri bir işleve argüman olarak gönderilebilir:

```
func(y + z);
```

Yukarıdaki örnekte *func* işlevine argüman olarak *y + z* ifadesinin değeri, yani toplama işlecinin ürettiği değer gönderiliyor.

iii. Üretilen değer *return* deyimi ile işlevlerin geri dönüş değerlerinin oluşturulmasında kullanılabilir:

```
int func()
{
    return (y + z)
}
```

Yukarıda *func* isimli işlevinin geri dönüş değeri *y + z* ifadesinin değeri *y*ani + işlecinin ürettiği değerdir.

İşleçlerin ürettiği değerin hiç kullanılmaması C sözdizimi açısından bir hataya neden olmaz. Ancak böyle durumlarda derleyiciler çoğunlukla bir uyarı iletisi vererek programcıyı uyarır. Örneğin:

```
int main()
{
    int x = 20;
    int y = 10;
    x + y;

    return 0;
}
```

Yukarıdaki kod parçasında yer alan

```
x + y
```

ifadesinde '+' işleci bir değer üretir. '+' işlecinin ürettiği değer terimlerinin toplamı değeri, yani *30*'dur. Ancak bu değer kullanılmıyor. Böyle bir işlemin bilinçli olarak yapılma olasılığı düşüktür. *Borland* derleyicilerinde verilen uyarı iletisi şu şekildedir:

```
warning : "code has no effect!"
(uyarı : "kodun etkisi yok")
```

İşleçlerin Önceliği

C dilinde ifadelerin türleri ve değerleri söz konusudur. Bir ifadenin değerini derleyici şu şekilde saptar: İfade içindeki işleçler öncelik sıralarına göre değer üretir, üretilen değerler, ifade içindeki önceliği daha az olan işleçlere terim olarak aktarılır. Bu işlemin sonunda tek bir değer elde edilir ki bu da ifadenin değeridir.

```
int x = 10;
int y = 3;
int z = 15;

printf("%d\n", z % y / 2 + 7 -x++ * y);
```

Yukarıdaki kod parçasında *printf* işlevi çağrısıyla

```
x % y / 2 + 7 -x++ * y
```

ifadesinin değeri ekrana yazdırılır. Yazdırılan değer nedir? İfade içindeki işleçler öncelik sıralarına göre değer üretir, üretilen değerler, diğer işleçlerin terimi olur. En son kalan değer ise ifadenin değeri, yani ekrana yazdırılan değer olur.
Her programlama dilinde işleçlerin birbirlerine göre önceliği söz konusudur. Eğer öncelik kavramı söz konusu olmasaydı, işleçlerin neden olacağı işlemlerin sonuçları makineden makineye, derleyiciden derleyiciye farklı olurdu.

C'de toplam *45* işleç vardır. Bu işleçler *15* ayrı öncelik seviyesinde yer alır.
C dilinin işleç öncelik tablosu bölüm sonunda verilmiştir.
Bir öncelik seviyesinde eğer birden fazla işleç varsa, bu işleçlerin aynı ifade içinde yer alması durumunda, işleçlerin soldan sağa mı sağdan sola mı öncelikle ele alınacağı da tanımlanmalıdır. Buna, öncelik yönü *(associativity)* denir. Ekteki tablonun 4. sütunu ilgili öncelik seviyesine ilişkin öncelik yönünü belirtiyor. Tablodan da görüldüğü gibi her öncelik seviyesi soldan sağa öncelikli değildir. 2, 13 ve 14. öncelik seviyelerinin sağdan sola öncelik yönüne sahip olduğunu *(right associative)* görüyorsunuz. Diğer bütün öncelik seviyeleri soldan sağa öncelik seviyesine *(left associative)* sahiptir.
Bir simge, birden fazla işleç olarak kullanılabilir. Örneğin, ekteki tabloyu incelediğinizde `'*'` simgesinin hem çarpma işleci hem de bir gösterici işleci olan içerik alma işleci olarak kullanıldığını göreceksiniz. Yine *'&' (ampersand)* simgesi hem *bitsel ve* işleci hem de göstericilere ilişkin *adres işleci* olarak kullanılır.

İşleçlerin Yan Etkileri

C dilinde işleçlerin ana işlevleri, bir değer üretmeleridir. Ancak bazı işleçler, terimi olan nesnelerin değerlerini değiştirir. Yani bu nesnelerin bellekteki yerlerine yeni bir değer yazılmasına neden olurlar. Bir işlecin, terimi olan nesnenin değerini değiştirmesine işlecin yan etkisi (*side effect)* denir. Yan etki, bellekte yapılan değer değişikliği olarak tanımlanır. Örneğin atama işlecinin, ++ ve -- işleçlerinin yan etkisi vardır. Bu işleçler, terimleri olan nesnelerin bellekteki değerlerini değiştirebilir.

İşleçler Üzerindeki Kısıtlamalar

Programlama dilinin kurallarına göre, bazı işleçlerin kullanımlarıyla ilgili birtakım kısıtlamalar söz konusu olabilir. Örneğin ++ işlecinin kullanımında, işlecin teriminin nesne gösteren bir ifade olması gibi bir kısıtlama söz konusudur. Eğer terim olan ifade bir nesne göstermiyorsa, yani sol taraf değeri değilse, derleme zamanında hata oluşur.

Kısıtlama, işlecin terim ya da terimlerinin türleriyle de ilgili olabilir. Örneğin kalan (%) işlecinin terimlerinin bir tamsayı türünden olması gerekir. Kalan işlecinin terimleri gerçek sayı türlerinden olamaz. Terimin gerçek sayı türlerinden birinden olması geçersizdir.

İşleçlerin Yaptıkları İşlere Göre Sınıflandırılması

Aşağıda işleçler yaptıkları işlere göre sınıflanıyor:

Aritmetik işleçler *(arithmetic operators)*
Bu işleçler aritmetik bazı işlemlerin yapılmasına neden olur. Toplama, çıkarma, çarpma, artırma, eksiltme işleçleri ile işaret işleçleri, aritmetik işleçlerdir.

Karşılaştırma işleçleri *(relational operators)*
Bu işleçlere ilişkisel işleçler de denir.
Bu işleçler bir karşılaştırma işlemi yapılmasını sağlar. Büyüktür, büyük ya da eşittir, küçüktür, küçük ya da eşittir, eşittir, eşit değildir işleçleri karşılaştırma işleçleridir.

Mantıksal işleçler (*logical operators)*

Bu işleçler, mantıksal işlemler yapar. Mantıksal ve, mantıksal veya, mantıksal değil işleçleri bu gruba girer.

Gösterici işleçleri (*pointer operators*)
Bu işleçler, adresler ile ilgili bazı işlemlerin yapılmasını sağlar. Adres işleci, içerik işleci ile köşeli ayraç işleci bu gruba girer.

Bitsel işlem yapan işleçler (*bitwise operators*)
Bu işleçler, bitsel düzeyde bazı işlemlerin yapılmasını sağlar. Bitsel değil işleci, bitsel kaydırma işleçleri, bitsel ve, veya, özel veya işleçleri bu gruba giren işleçlerdir.

Atama işleçleri *(assignment operators*)
Bir nesneye atama yapılmasını sağlayan işleçlerdir. Atama işleci ve işlemli atama işleçleri bu gruba girer.

Özel amaçlı işleçler (*special purpose operators*)
Bunlar farklı işlerin yapılmasını sağlayan ve farklı amaçlara hizmet eden işleçlerdir. Koşul işleci, *sizeof* işleci, tür dönüştürme işleci bu gruba giren işleçlerdir.

İlk üç grup, programlama dillerinin hepsinde vardır. Bitsel işlem yapan işleçler ve gösterici işleçleri yüksek seviyeli programla dillerinde genellikle bulunmaz. Programlama dillerinin çoğu, kendi uygulama alanlarında kolaylık sağlayacak birtakım özel amaçlı işleçlere de sahip olabilir.

Aritmetik İşleçler
Aritmetik işleçler, basit artimetiksel işlemler yapan işleçlerdir.

Toplama (+) ve Çıkarma(-) İşleçleri
İki terimli, araek konumundaki *(binary infix)* işleçlerdir. Diğer bütün programlama dillerinde oldukları gibi, terimlerinin toplamını ya da farkını almak için kullanırlar. Yani ürettikleri değer, terimlerinin toplamı ya da farkı değerleridir.
Bu işlecin terimleri herhangi bir türden nesne gösteren ya da göstermeyen ifadeler olabilir. Terimlerinin aynı türden olması gibi bir zorunluluk da yoktur. İşleç öncelik tablosunun 4. seviyesinde bulunurlar. Öncelik yönleri soldan sağadır. Her iki işlecin de yan etkisi yoktur. Yani bu işleçler terimlerinin bellekte sahip oldukları değerleri değiştirmez.

Toplama ve çıkarma işleçleri olan + ve – işleçlerini tek terimli + ve – işleçleriyle karıştırmamak gerekir.

İşaret İşleci Olan – ve +
Bu işleçler, tek terimli, önek konumundaki (*unary prefix)* işleçlerdir. İşaret işleci eksi (-), teriminin değerinin ters işaretlisini üretir. Yani derleyici, işaret eksi işlecinin kullanılması durumunda terim olan değeri *-1* ile çarpacak şekilde kod üretir. Bu işlecin terimi herhangi bir türden nesne gösteren ya da göstermeyen ifade olabilir. İşleç öncelik tablosunun ikinci seviyesinde bulunurlar. Öncelik yönü sağdan soladır. İşlecin bir yan etkisi yoktur, yani terimi olan nesnenin bellekteki değerini değiştirmez.

"İşaret eksi" işlecinin ürettiği, bir nesne değildir, bir sağ taraf değeridir. Aşağıdaki ifade matematiksel olarak doğru olmasına karşın C dilinde doğru değildir, derleme zamanında hata oluşumuna neden olur:

```
int x;
-x = 5;
```

x bir nesne olmasına karşı *–x* ifadesi bir nesne değil, *x* nesnesinin değerinin ters işaretlisi olan değerdir.
-x ve *0 - (x)* eşdeğer ifadelerdir.
-x ifadesi bir sol taraf değeri değildir, bu ifadeye bir atama yapılamaz.

İşaret işleci artı *(+),* yalnızca matematiksel benzerliği sağlamak açısından C diline eklenmiş bir işleçtir. Derleyici tarafından, tek terimli, önek konumunda bir işleç olarak ele alınır. Terimi olan ifade üzerinde herhangi bir etkisi olmaz. Teriminin değeriyle aynı değeri üretir. *+x* ifadesi ile *0 + (x)* ifadeleri eşdeğerdir.

```
#include <stdio.h>

int main()
{
    int x = -5;

    x = -x - x;
    printf("x = %d\n", x);

    return 0;
}
```

```
x = -x - x;
```

Yukarıdaki ifadede 3 işleç vardır. Soldan sağa bu işleçleri sayalım: Atama işleci `'='`, işaret işleci eksi `'-'`, çıkarma işleci `'-'`. İfadenin değerinin hesaplanmasında işleç önceliklerine göre hareket edilir. Önce ikinci seviyede bulunan eksi işaret işleci değer *'5'* değerini üretir. Üretilen *5* değeri çıkarma işlecinin terimi olur. Yapılan çıkartma işleminden üretilen değer *10'*dur. Bu değer de atama işlecinin terimi olur, böylece *x* değişkenine *10* değeri atanır.

Çarpma (*) ve Bölme (/) İşleçleri

İki terimli, araek konumundaki işleçlerdir. Çarpma işlecinin ürettiği değer, terimlerinin çarpımıdır. Bölme işlecinin ürettiği değer ise sol teriminin sağ terimine bölümünden elde edilen değerdir. Bu işleçlerin terimleri herhangi bir türden olabilir. Terimlerinin aynı türden olması gibi bir zorunluluk yoktur. İşleç öncelik tablosunun 3. seviyesinde bulunurlar. Öncelik yönleri soldan sağadır. Her iki işlecin de yan etkisi yoktur.

Bölme işlecinin kullanımında dikkatli olmak gerekir. İşlecin her iki terimi de tamsayı türlerinden ise işlecin ürettiği değer de bir tamsayı olur. Yani bir tamsayıyı başka bir tamsayıya bölmekle bir gerçek sayı elde edilmez.
C programlama dilinde * simgesi aynı zamanda bir gösterici işleci olarak da kullanılır. Ama aynı simge kullanılmasına karşın bu iki işleç hiçbir zaman birbirine karışmaz çünkü aritmetik çarpma işleci iki terimli iken gösterici işleci tek terimlidir.

Kalan (%)İşleci

İki terimli, araek konumunda bir işleçtir. Terimlerinin her ikisi de tamsayı türlerinden (*char, short, int, long*) olmak zorundadır. Herhangi bir teriminin gerçek sayı türünden olması geçersizdir. İşlecin ürettiği değer, sol teriminin sağ terimine bölümünden kalandır. İşlecin yan etkisi yoktur. Örneğin:

```
k = 15 % 4;         /* burada k ya 3 değeri atanır*/
x = 2 % 7;          /* burada x e 2 değeri atanır*/
int c = 13 - 3 * 4 + 8 / 3 - 5 % 2;
```

Burada *c* değişkenine *2* değeri atanır. Çünkü işlem şu şekilde yapılır:

```
c = 13 - (3 * 4) + (8 / 3) - (5 % 2)
c = 13 - 12  + 2 - 1;
c = 2;
```

Aşağıdaki programda *3* basamaklı bir sayının birler, onlar ve yüzler basamakları ekrana yazdırılıyor:

```
#include <stdio.h>

int main()
{
    int x;

    printf("3 basamakli bir sayi girin: ");
    scanf("%d", &x);
    printf("birler basamagi = %d\n", x % 10);
    printf("onlar basamagi = %d\n", x % 100 / 10);
    printf("yuzler basamagi = %d\n", x / 100);

    return 0;
}
```

Artırma (++) ve Eksiltme (--) İşleçleri

Artırma *(++)* ve eksiltme (--) işleçleri C dilinin en çok kullanılan işleçlerindendir. Tek terimli işleçlerdir. Önek ya da sonek durumunda bulunabilirler. ++ işleci terimi olan değişkenin değerini 1 artırmak, -- işleci de terimi olan değişkenin değerini 1 eksiltmek için kullanılır. Dolayısıyla yan etkileri söz konusudur. Terimleri olan nesnenin bellekteki değerini değiştirirler. Bu iki işleç de 2. öncelik seviyesinde bulunduğundan diğer aritmetik işleçlerden daha yüksek önceliğe sahiptir. 2. öncelik seviyesine ilişkin öncelik yönü sağdan soladır.

Yalın olarak kullanıldıklarında, yani bulundukları ifade içinde kendilerinden başka hiçbir işleç olmaksızın kullanıldıklarında önek ya da sonek durumları arasında hiçbir fark yoktur. ++ işleci terimi olan nesnenin değerini 1 artırır, -- işleci terimi olan nesnenin değerini 1 eksiltir. Bu durumda

++c; ve c++ ;

deyimleri tamamen birbirine denk olup

```
c = c + 1;
```

anlamına gelirler.

--c; ve c--;
deyimleri tamamen birbirine denk olup

```
c = c - 1;
```

anlamına gelir.
Bir ifade içinde diğer işleçlerle birlikte kullanıldıklarında, önek ve sonek biçimleri arasında farklılık vardır: Önek durumunda kullanıldığında, işlecin ürettiği değer, artırma ya da eksiltme yapıldıktan sonraki değerdir. Yani terimin artırılmış ya da azaltılmış değeridir. Sonek durumunda ise işlecin ürettiği değer, artırma ya da eksiltme yapılmadan önceki değerdir. Yani terimi olan nesnenin artırılmamış ya da azaltılmamış değeridir. Nesnenin değeri ifadenin tümü değerlendirildikten sonra artırılır ya da eksiltilir.

```
x = 10;
y = ++x;
```

Bu durumda:

```
++x  ⇒ 11
```
ve *y = 11* değeri atanır.

```
x = 10;
y = x++;
```

Bu durumda

```
x++ => 10
```
ve *y* değişkenine *10* değeri atanır.

Aşağıdaki programı inceleyin:

```
#include <stdio.h>

int main()
{
    int a = 10;
    int b = ++a;
    printf("a  = %d b = %d\n", a, b);
    a = 10;
    b = a++;
    printf("a  = %d b = %d\n", a, b);

   return 0;
}
```

Yukarıdaki birinci *printf* çağrısı ifadesi ekrana

```
11 11
```

değerlerini yazdırırken ikinci *printf* çağrısı ekrana

```
11 10
```

değerlerini yazdırır. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

int main()
{
    int x = 10;
    int y = 5;
    int z = x++ % 4 * --y;

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

    return 0;
}
```

Yukarıda kodu verilen *main* işlevinde işlem sırası şu şekilde olur:

```
z = x++ % 4 * 4;
z = 10 % 4 * 4;
z = 2 * 4;
z = 8;
y => 4
x => 11
```

Aşağıdaki örneği derleyerek çalıştırın:

```
#include <stdio.h>

int func(int x)
{
    return ++x;
}

int  main()
{
    int a = 10;
    int b = func(a++);
    printf("a = %d\n", a);
    printf("b = %d\n", b);

    return 0;
}
```

C Standartlarında Kullanılan Bazı Önemli Terimlere İlişkin Açıklama

C ve C++ standartlarında sıklıkla kullanılan ve derleyicinin kodu yorumlama biçimi hakkında bilgi veren önemli terimler vardır:

Davranış

Derleyicinin belirli bir kod parçasını yorumlama ve anlamlandırma biçimine *"derleyicinin davranışı" (behavior)* denir.

Tanımsız Davranış

C'de ve C++'da bazı ifadeler, derleyiciden derleyiciye değişebilen fakat standartlarda açık olarak belirtilmemiş olan yorumlama farlılıklarına yol açabilir. Böyle ifadelerden kaçınmak gerekir. Bu tür ifadelerde derleyicinin davranışına "*tanımsız davranış*" (*undefined behavior*) denir. Programcının böyle ifadeler yazması programlama hatası olarak kabul edilir. Çünkü eğer bir ifade tanımsız davranış olarak belirleniyorsa bir sistemde programın çalıştırılması sonucunda nasıl bir durumla karşılaşılacağının hiçbir güvencesi yoktur. Tanımsız davranışa yol açan kodlar sözdizimi açısından geçerlidir. Örneğin bir ifadede bir değişken ++ ya da –- işlecinin terimi olarak kullanılmışsa aynı ifadede o değişken artık bir kez daha yer almamalıdır. Yer alırsa artık tanımsız davranıştır.

Belirlenmemiş Davranış

Kaynak kodun derleyici tarafından farklı yorumlanabildiği fakat bu konuda seçeneklerin sınırlı olduğu durumlara belirlenmemiş davranış (*unspecified behavior*) denir. Derleyiciler belirsiz davranışlarda hangi seçeneğin seçilmiş olduğunu belgelemek zorunda değildir. Şüphesiz programcının belirsiz davranışa yol açacak ifadelerden kaçınması gerekir.

Derleyiciye Özgü Davranış

C dilinin bazı özellikleri, esneklik sağlamak amacı ile standartlarda derleyici yazanların seçimlerine bırakılmıştır. Örneğin *int* türünün uzunluğunun ne olduğu, varsayılan *char* türünün *signed* mı *unsigned* mı olduğu, iç içe yorumlamaların kabul edilip edilmediği tamamen derleyici yazanlara bağlıdır. Derleyiciler, bu özelliklerin nasıl seçildiklerini belgelemek zorundadır. Bu tür davranışa derleyiciye özgü davranış (i*mplementation dependent behaviour*) denir. Bu davranış özellikleri pekçok derleyicide menülerden değiştirilebilmektedir.

Bulgu İletileri

C standartları temel olarak derleyiciyi yazanlar için bir klavuz biçimindedir. Derleyici sorunlu bir kodla karşılaştığında uygun dönüştürme işlemlerini yapamıyorsa sorunun nedenine ilişkin bir bildirimde bulunmak zorundadır. Standartlarda derleyicilerin sorunu programcıya bildirme durumuna "bulgu iletisi" (*diagnostic message*) denmektedir. Standartlar içinde belirtilmiş olan sözdizimsel ve anlamsal kuralların çiğnendiği durumlarda bir uyarı iletisi verilmelidir. Bu iletinin uyarı (*warning*) ya da hata (*error*) biçiminde olması, derleyicinin isteğine bırakılmıştır.
Ancak derleyicilerin hemen hepsinde uyarılar, derleyiciler tarafından giderilebilecek küçük yanlışlar için, hata ise daha büyük yanlışlar için verilir.
Örneğin bir göstericiye farklı türden bir adresin doğrudan atanması C'nin kurallarına aykırıdır. Bu durumda derleyici standartlara göre bir ileti vermelidir. Aslında standartlara göre, uyarı ya da hata iletisi verilebilir, ama C derleyicilerinin hemen hepsi uyarı iletisi verir.
Standartlarda bazı kuralların çiğnenmesi durumunda derleyicinin açıkça bir ileti vermeyebileceği belirtilmiştir (*nodiagnostic required*). Aslında C standartlarında belirtildiği gibi kural çiğnenmeleri durumunda derleyicinin işlemi başarı ile bitirip bitirmeyeceği açıkça belirtilmemiştir. Yani standartlara göre derleyici, doğru bir programı derlemeyebilir, yanlış bir programı derleyebilir.

Ancak C++ standartlarında durum böyle değildir. Dilin kuralına uymayan kodlarda derleyici bir ileti vermeli, derleme işlemini başarısızlık ile sonuçlanmalıdır.
Derleyiciler, standartlarda belirtilen özelliklerin dışında da bazı özelliklere sahip olabilir. Bu tür özelliklere derleyicilerin eklentileri denir. Derleyicilerin eklentilerini kullanmak taşınabilirliği azaltır.

++ ve -- İşleçleriyle İlgili Tanımsız Davranışlar

++ ve -- işleçlerinin bazı kullanımları, tanımsız davranış özelliği gösterir. Böyle kodlardan sakınmak gerekir.

1. Bir ifadede bir nesne ++ ya da -- işleçlerinin terimi olmuşsa, o nesne o ifadede bir kez daha yer almamalıdır. Örneğin aşağıdaki ifadelerin hepsi tanımsız davranış özelliği gösterirler:

```
int x = 20, y;
int a = 5;

y = ++x + ++x;          /* tanımsız davranış */
y = ++x + x             /* tanımsız davranış */
a = ++a;                /* tanımsız davranış */
```

"Koşul işleci", "mantıksal ve işleci", "mantıksal veya işleci" ve "virgül" işleciyle oluşturulan ifadelerde bir sorun yoktur. Bu işleçlerle ilgili önemli bir kurala ileride değineceğiz.

2. Bir işlev çağrılırken işleve gönderilen argümanların birinde bir nesne ++ ya da -- işlecinin terimi olmuşsa, bu nesne, işleve gönderilen diğer argüman olan ifadelerde kullanılmamalıdır.
Argüman olan ifadelerin, işlevlerin ilgili parametre değişkenlerine kopyalanmasına ilişkin sıra, standart bir biçimde belirlenmemiştir. Bu kopyalama işlemi, bazı sistemlerde soldan sağa bazı sistemlerde ise sağdan soladır. Aşağıdaki örneği inceleyin:

```
int a = 10;

void func(int x, int y)
{
    /***/
}

int main()
{
    func (a, a++);      /* Tanımsız davranış */

    /***/
}
```

Karşılaştırma İşleçleri (ilişkisel işleçler)

C programlama dilinde toplam 6 tane karşılaştırma işleci vardır:

<	küçüktür işleci *(less than)*
>	büyüktür işleci *(greater than)*
<=	küçüktür ya da eşittir işleci *(less than or equal)*
>=	büyüktür ya da eşittir işleci *(greater than or equal)*
==	eşittir işleci *(equal)*
!=	eşit değildir işleci *(not equal)*

Bu işleçlerin hepsi, iki terimli, araek konumundaki *(binary infix)* işleçlerdir.

İlk dört işleç, işleç öncelik tablosunun 6. seviyesinde bulunurken diğer iki karşılaştırma işleci öncelik tablosunun 7. seviyesinde bulunur. Yani karşılaştırma işleçleri, kendi aralarında iki öncelik grubu oluşturur. Karşılaştırma işleçleri, aritmetik işleçlerden daha düşük öncelikli seviyededir.

Diğer programlama dillerinin çoğunda *bool* ya da *boolean* (Matematikçi *George Bool*'un isminden) ismi verilen bir mantıksal veri türü de doğal bir veri türü olarak programcının kullanımına sunulmuştur. Böyle dillerde *bool* veri türü, yalnızca mantıksal doğru ya da mantıksal yanlış değerlerini alabilen bir türdür. Bu dillerde karşılaştırma işleçlerinin ürettiği değerler ise bu türdendir. Örneğin *C++* ya da *Java* dillerinde durum böyledir.

C dilinde karşılaştırma işleçleri, oluşturdukları önermenin doğruluğu ve yanlışlığına göre *int* türden *1* ya da *0* değerini üretir. Önerme doğru ise *1* değeri üretilirken, önerme yanlış ise *0* değeri üretilir. Bu işleçlerin ürettiği değerler de tıpkı aritmetik işleçlerin ürettiği değerler gibi kullanılabilir.
Aşağıdaki *signum* isimli işlevin tanımını inceleyin:

```
int signum(int val)
{
    return (val > 0) - (val < 0);
}
```

signum işlevine gönderilen argüman *0*'dan büyük bir değerse işlev *+1* değerine, argüman *0*'dan küçük bir değerse işlev *-1* değerine, argüman *0* değeriyse işlev, *0* değerine geri dönüyor. *signum* işlevinin geri dönüş değeri, karşılaştırma işleçlerinin değer üretmesinden faydalanılarak elde ediliyor.

Bazı programlama dillerinde

```
(val > 0) - (val < 0);
```

gibi bir işlem hata ile sonuçlanır. Çünkü örneğin *Pascal* dilinde

```
val > 0
```

ifadesinden elde edilen değer doğru (*True)* ya da yanlış *(False)* dir. Yani üretilen değer *bool* ya da *boolean* türündendir. Ama C doğal bir dil olduğu için karşılaştırma işleçlerinin ürettikleri değer *bool* türü ile kısıtlanmamıştır. C'de mantıksal veri türü yerine *int* türü kullanılır. Mantıksal bir veri türünün tamsayı türüyle aynı olması C'ye esneklik ve doğallık kazandırmıştır. C dilinde yazılan birçok kalıp kod, karşılaştırma işleçlerinin *int* türden *1* ya da *0* değeri üretmesine dayanır. Örneğin

```
x = y == z;
```

Yukarıdaki deyim, C dili için son derece doğaldır ve okunabilirliği yüksektir. Bu deyimin yürütülmesiyle *x* değişkenine ya *1* ya da *0* değeri atanır. Karşılaştırma işleci, atama işlecinden daha yüksek öncelik seviyesine sahip olduğuna göre önce karşılaştırma işleci olan '==' değer üretir, işlecin ürettiği değer bu kez atama işlecinin terimi olur. Bu durumda *y* değişkeninin değerinin *z* değişkenine eşit olup olmamasına göre *x* değişkenine *1* ya da *0* değeri atanır.

Karşılaştırma işlecinin kullanılmasında bazı durumlara dikkat edilmelidir:

```
int x = 12;

5 < x <  9
```

Yukarıdaki ifade matematiksel açıdan doğru değildir. Çünkü *12* değeri *5* ve *9* değerlerinin arasında değildir. Ancak ifade C kodu olarak ele alındığında doğru olarak değerlendirilir. Çünkü 6. seviyede olan küçüktür *(<)* işlecine ilişkin öncelik yönü soldan sağadır. Önce soldaki `'<'` işleci değer üretecek ve ürettiği değer olan *1* sağdaki `'<'` işlecinin terimi olur. Bu durumda

```
1 < 9
```

ifadesi mantıksal olarak doğru olduğu için *1* değeri elde edilir.

Mantıksal İşleçler

Bu işleçler, terimleri üzerinde mantıksal işlem yapar. Terimlerini doğru *(true)* ya da yanlış *(false)* olarak yorumladıktan sonra işleme sokar. C'de öncelikleri farklı seviyede olan üç mantıksal işleç vardır:

(!) mantıksal değil işleci *(logical not)*
(&&) mantıksal ve işleci *(logical and)*
(||) mantıksal veya işleci *(logical or)*

Ancak "mantıksal ve", "mantıksal veya" işleçleri, bilinen anlamda işleç öncelik kurallarına uymaz. Bu konuya biraz ileride değineceğiz.

C'de mantıksal veri türü olmadığını biliyorsunuz. Mantıksal veri türü olmadığı için bu türün yerine *int* türü kullanılır ve mantıksal doğru olarak *1,* mantıksal yanlış olarak da *0* değeri kullanılır.

C dilinde herhangi bir ifade, mantıksal işleçlerin terimi olabilir. Bu durumda söz konusu ifade, mantıksal olarak yorumlanır. Bunun için ifadenin sayısal değeri hesaplanır. Hesaplanan sayısal değer, 0 dışı bir değer ise *doğru (1)*, *0* ise *yanlış (0)* olarak yorumlanır. Örneğin:

```
25 Doğru (Çünkü 0 dışı bir değer)
-12 Doğru (Çünkü 0 dışı bir değer)
0 Yanlış (Çünkü 0)
```

ifadesi mantıksal bir işlecin terimi olduğu zaman yanlış olarak yorumlanır. Çünkü sayısal değeri sıfıra eşittir.

Mantıksal Değil İşleci

Mantıksal değil işleci, önek konumunda bulunan tek terimli bir işleçtir. Bu işleç, teriminin mantıksal değerinin tersini üretir. Yani terimi mantıksal olarak *"doğru"* biçiminde yorumlanan bir değer ise işleç yanlış anlamında *int* türden *0* değerini üretir. Terimi, mantıksal olarak *"yanlış"* biçiminde yorumlanan bir değer ise işleç doğru anlamında *int* türden *1* değerini üretir.

x	!x
Doğru (0 dışı değer)	Yanlış (0)
Yanlış (0)	Doğru (1)

Örnekler :

```
a = !25;  /* a değişkenine 0 değeri atanır  */
b = 10 * 3 < 7 + !2
```

İşlem sırası:

```
!2 = 0
10 * 3 = 30
7 + 0 = 7
30 < 7 = 0
b = 0 (atama işleci en düşük öncelikli işleçtir)

y = 5;
x = !++y < 5 != 8;
```

İşlem sırası:

```
++y ⇒ 6
!6 ⇒ 0     /* ++ ve ! işleçleri aynı öncelik seviyesindedir ve öncelik
yönü sağdan soladır. */
0 < 5 ⇒ 1
1 != 8 ⇒ 1
x = 1
```

Mantıksal ve (&&) işleci

Bu işleç ilişkisel işleçlerin hepsinden düşük, || (veya / or) işlecinden yüksek önceliklidir. Terimlerinin ikisi de *doğru* ise d*oğru* (1), terimlerinden biri *yanlış* ise *yanlış (0)* değerini üretir.

x	y	x && y
doğru	doğru	doğru
doğru	yanlış	yanlış
yanlış	doğru	yanlış
yanlış	yanlış	yanlış

```
x = 3 < 5 && 7;
3 < 5 ⇒ 1
7 ⇒ 1
1 && 1  ⇒ 1
x = 1
```

&& işlecinin, önce sol tarafındaki işlemler öncelik sırasına göre tam olarak yapılır. Eğer bu işlemlerde elde edilen sayısal değer *0* ise, && işlecinin sağ tarafındaki işlemler hiç yapılmadan, *yanlış* (*0*) sayısal değeri üretilir. Örneğin:

```
x = 20;
b = !x == 4 && sqrt(24);
!20 ⇒ 0
0 == 4 ⇒ 0
```

Sol taraf *0* değeri alacağından işlecin sağ tarafı hiç yürütülmez dolayısıyla da *sqrt* işlevi çağrılmaz. Sonuç olarak *b* değişkenine *0* değeri atanır.
Uygulamalarda mantıksal işleçler çoğunlukla karşılaştırma işleçleriyle birlikte kullanılır:

```
scanf("%d", &x);
y = x >= 5 && x <= 25;
```

Bu durumda *y* değişkenine, ya *1* ya da *0* değeri atanır. Eğer *x* değişkeninin değeri *5*'den büyük ya da eşit ve *25*'den küçük ya da eşit ise *y* değişkenine *1* değeri, bunun dışındaki durumlarda *y* değişkenine *0* değeri atanır.

```
ch = 'c'
z = ch >= 'a' && ch <= 'z'
```

Yukarıdaki örnekte, *ch* değişkeninin küçük harf olup olmaması durumuna göre *z* değişkenine *1* ya da *0* atanır.

Mantıksal veya (||) İşleci

Önceliği en düşük olan mantıksal işleçtir. İki teriminden biri *doğru* ise *doğru* değerini üretir. İki terimi de *yanlış* ise *yanlış* değerini üretir.

x	y	x || y
doğru	doğru	doğru
doğru	yanlış	doğru
yanlış	doğru	doğru
yanlış	yanlış	yanlış

```
a = 3 || 5          /* a = 1 */
x = 0 || -12        /* x = 1 */
sayi = 0 || !5      /* sayi = 0 */
```

&& ve || İşleçlerinin Kısa Devre Davranışı

"Mantıksal ve", "mantıksal veya" işleçlerinde önce soldaki terimlerinin değerlendirilmesi güvence altına alınmıştır. "Mantıksal ve" işlecinin soldaki terimi yanlış olarak yorumlanırsa işlecin sağ terimi hiç ele alınmaz. Aynı durum "mantıksal veya" işleci için de geçerlidir. "Mantıksal veya" işlecinin önce soldaki terimine bakılır. Sol terimi doğru olarak yorumlanırsa işlecin sağ terimi hiç dikkate alınmaz. C dili tarafından güvence altına alınan bu özelliğe *"kısa devre davranışı" (short circuit behavior)* denir. Kısa devre davranışına neden gerek duyulmuştur? Çünkü bu özellik bazı kodların çok daha verimli yazılmasını sağlar. C'nin ileride göreceğimiz birçok kalıp kodu kısa devre davranışının kullanımına bağlıdır.

Aşağıdaki ifadeyi inceleyin:

```
result =  ch == 'A' || ch == 'E'
```

Yukarıdaki ifade ile, *result* isimli değişkene, *ch* değişkeninin değerinin 'A' ya da *'B'* ye eşit olması durumunda *1* değeri, aksi halde *0* değeri atanır. *ch* eğer *'A'* ya eşit ise ikinci karşılaştırma yapılmaz.
Mantıksal işleçler bir değer üretebilmek için terimlerini önce *1* ya da *0,* yani *doğru* ya da *yanlış* olarak yorumlar, ama yan etkileri yoktur. Yani terimlerinin nesne olması durumunda bu nesnelerin bellekteki değerlerini *1* ya da *0* olarak değiştirmezler.

Atama İşleçleri

Atama işleçleri, C dilinde öncelik tablosunun en alttan ikinci seviyesinde, yani *14.* seviyesinde bulunur ve yalnızca virgül işlecinden daha yüksek önceliklidir. Atama işleçlerinin bulunduğu 14. seviye, sağdan sola öncelik yönüne sahiptir.

Yalın Atama İşleci

Diğer işleçler gibi atama işleci de, yaptığı atama işleminin yanısıra, bir değer üretir.

Atama işlecinin ürettiği değer, nesneye atanan değerin kendisidir. Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    int x;

    printf("ifade degeri = %d\n", x = 5);
    printf("x = %d\n", x);

    return 0;
}
```

main işlevi içinde yapılan birinci *printf* çağrısı ile, *x = 5* ifadesinin değeri yazdırılıyor. *x = 5* ifadesinin değeri atama işlecinin ürettiği değer olan *5* değeridir. Yani ilk *printf* çağrısı ile ekrana *5* değeri yazdırılır. Atama işleci yan etkisi sonucu *x* nesnesinin değerini *5* yapar. Bu durumda ikinci *printf* çağrısı ile *x* değişkeninin değeri ekrana yazdırıldığından ekrana yazılan, *5* değeri olur.
Atama işlecinin ürettiği değer nesne değildir. Aşağıdaki deyim geçersizdir:

```
(b = c) = a;        /* Geçersiz! */
```

b = c atamasından elde edilen değer, *c* nesnesinin kendisi değil, *c* nesnesinin sayısal değeridir.
C'nin birçok kalıp kodunda, atama işlecinin ürettiği değerden faydalanılır. Aşağıdaki *main* işlevini inceleyin:

```
#include <stdio.h>

int main()
{
    int a, b, c, d;

    a = b = c = d = 5;
    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("c = %d\n", c);
    printf("d = %d\n", d);
    printf("e = %d\n", e);

    return 0;
}
```

İşleç öncelik tablosundan da görüleceği gibi, atama işleçleri sağdan sola öncelik yönüne sahiptir. Bu yüzden:

```
a = b = c = d = 5;
```

deyimi C'de geçerlidir.

Bu deyimde önce *d* değişkenine *5* değeri atanır. Atama işlecinin ürettiği *5* değeri, bu kez *c* değişkenine atanır. Sağdan sola doğru ele alınan her atama işleci, nesneye atanan değeri ürettiğine göre, tüm değişkenlere *5* değeri aktarılmış olur. Atama işlecinin ürettiği değerden faydalanmak, özellikle kontrol deyimlerinde karşınıza çok çıkacak.

İşlemli Atama İşleçleri

Bir işlemin terimi ile, işlem sonucunda üretilen değerin atanacağı nesne aynı ise, işlemli atama işleçleri kullanılabilir.

```
<nesne1> = <nesne1> işlem <terim2>
```

ile

```
<nesne1>  işlem= <terim2>
```

aynı anlamdadır.

İşlemli atama işleçleri, atama işleciyle aynı öncelik seviyesindedir.

İşlemli atama işleçleri, hem okunabilirlik hem de daha kısa yazım için tercih edilir. Aşağıdaki ifadeler eşdeğerdir:

```
deger1 += 5;           deger1 = deger1 + 5;
sonuc *= yuzde;        sonuc  = sonuc * yuzde;
x %= 5                 x = x % 5;

katsayi = katsayi * (a * b + c * d);
```

ifadesi de yine

```
katsayi *= a * b + c * d;
```

şeklinde yazılabilir. Şimdi de aşağıdaki *main* işlevini inceleyin:

```
#include <stdio.h>

int main()
{
    int x = 3;
    int y = 5;

    x += y *= 3;
    printf("x = %d\n", x);
    printf("y = %d\n", y);

    return 0;
}
```

```
x += y *= 3;
```

deyimiyle önce *y* değişkenine *15* değeri atanır. Bu durumda *= işleci *15* değerini üretir ve üretilen *15* değeri bu kez += işlecinin terimi olur. Böylece *x* değişkenine *18* değeri atanır.
Özellikle += ve -= işleçlerinin yanlış yazılması, bulunması zor hatalara neden olabilir.

```
x += 5;
```

deyimi *x* değişkeninin değerini 5 artırırken, işlecin yanlışlıkla aşağıdaki gibi yazılması durumunda

```
x =+ 5;
```

x değişkenine *5* değeri atanır. Çünkü burada iki ayrı işleç söz konusudur: Atama işleci olan = ve işaret işleci olan +.

Yukarıdaki örneklerden de görüldüğü gibi, atama grubu işleçlerin yan etkileri vardır. Yan etkileri, işlecin sol teriminin bellekteki değerinin değiştirilmesi, yani işlecin sağ tarafındaki terimi olan ifadenin değerinin sol tarafındaki nesneye aktarılması şeklinde kendini gösterir.

Virgül İşleci

İki ayrı ifadeyi tek bir ifade olarak birleştiren virgül işleci, C'nin en düşük öncelikli işlecidir.

```
ifade1;
ifade2;
```

ile

```
ifade1, ifade2;
```

aynı işleve sahiptir.

Virgül işlecinin, önce sol terimi olan ifadenin sonra sağ terimi olan ifadenin ele alınması güvence altındadır. Bu işlecin ürettiği değer, sağ tarafındaki ifadenin ürettiği değerdir. Virgül işlecinin sol teriminin, üretilen değere bir etkisi olmaz.

```
x = (y++, z = 100);
```

gibi bir deyimle *x* ve *z* değişkenlerine *100* değeri atanır.

Aşağıdaki örnekte *if* ayracı içindeki ifadenin ürettiği değer *0*'dır.

```
if (x > 5,0) {
    /***/
}
```

Virgül işleçleri ile bir bileşik deyim basit deyim durumuna getirilebilir:

```
if (x == 20) {
    a1 = 20;
    a2 = 30;
    a3 = 40;
}
```

yerine

```
if (x == 20)
    a1 = 20, a2 = 30, a3 = 40;
```

yazılabilir.
Virgül işlecinin sağ terimi nesne gösteren bir ifade olsa bile işlecin oluşturduğu ifade bir nesne değildir:

```
int x, y;
/***/
(x, y) = 10;
```

Yukarıdaki atama işlemi geçersizdir.

[C++ dilinde virgül işlecinin oluşturduğu bir ifade sol taraf değeri olabilir. Yukarıdaki atama C++ dilinde geçerlidir.]

Öncelik İşleci

Öncelik işleci ( ), bir ifadenin önceliğini yükseltmek amacıyla kullanılır.

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

Öncelik işleci, C'nin en yüksek öncelikli işleçler grubundadır. Öncelik işleci 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
```

Öncelik işlecinin terimi nesne gösteren bir ifade ise, işlecin ürettiği ifade de nesne gösterir:

```
int x;
(x) = 20;   /* Geçerli  */
```

İşleç Önceliği ve Bir İşlemin İşlemci Tarafından Önce Yapılması

İşleç önceliği, bir işlemin işlemci tarafından daha önce yapılması anlamına gelmez. Aşağıdaki ifadeyi düşünelim:

```
x = func1() * func2() + func3();
```

Çarpma işlecinin toplama işlecinden daha yüksek öncelikli olduğunu biliyorsunuz. Ancak bu öncelik, örneğin yukarıdaki deyimde *func1* işlevinin *func3* işlevinden daha önce çağrılacağı güvencesi anlamına gelmez. Öncelik işlecinin de kullanımı böyle bir güvence sağlamaz.

```
x = func1() * (func2() + func3());
```

Bu kez de örneğin *func2* işlevinin *func1* işlevinden daha önce çağrılmasının güvencesi yoktur.Bu kez de aşağıdaki *main* işlevini inceleyin:

```
#include <stdio.h>

int main()
{
    int x = 10;
    int y = x + (x = 30);
    printf("y = %d\n", y);

    return 0;
}
```

main işlevi içinde yazılan

```
y = x + (x = 30);
```

deyimi şüpheli kod oluşturur. *y* değişkenine atanan değerin ne olacağı konusunda bir güvence yoktur. *x = 30* ifadesinin öncelik ayracı içine alınması, toplama teriminin sol terimi olan *x* ifadesinin değerinin *30* olarak ele alınacağını, güvence altına almaz. Sonuçta, *y* değişkenine *60* değeri aktarılabileceği gibi, *40* değeri de atanabilir.

Ancak C dilinin 4 işleci terimlerine ilişkin, daha önce işlem yapma güvencesini verir. Bu işleçler:

mantıksal ve, *mantıksal veya*, *koşul* ve *virgül* işleçleridir.

Mantıksal ve/veya işleçlerinin kısa devre davranışlarını öğrenmiştiniz. Kısa devre davranışının gerçekleştirilebilmesi için bu işleçlerin sol terimlerinin daha önce yapılması güvence altına alınmıştır.
İleride göreceğiniz koşul işlecinin de bir değer üretebilmesi için, önce ilk teriminin değerlendirilmesi gerekir. Virgül işlecinin ise zaten varlık nedeni önce sol, daha sonra sağ teriminin yapılmasını sağlamaktır.

C Dilinin İşleç Öncelik Tablosu

Seviye	İşleç	Tanım	Öncelik Yönü (associativity)
1	( )	Öncelik kazandırma ve işlev çağrı (*precedence and function call*)	soldan sağa
	[ ]	köşeli ayraç işleci (*subscript*)	
	.	yapı elemanına yapı nesnesi ile ulaşım (*structure access with object*)	
	->	yapı elemanına yapı göstericisi ile ulaşım (*structure access with pointer*)	
2	+	işaret işleci (*unary sign*)	sağdan sola
	-	işaret işleci (*unary sign*)	
	++	artırma işleci (*increment*)	
	--	eksiltme işleci (*decrement*)	
	~	bitsel değil işleci (*bitwise not*)	
	!	mantıksal değil işleci(*logical not*)	
	*	içerik işleci (*indirection*)	
	&	adres işleci (*address of*)	
	sizeof	*sizeof* işleci (sizeof)	
	(tür)	tür dönüştürme işleci (*type cast operator*)	
3	*	çarpma işleci (*multiplication*)	soldan sağa
	/	bölme işleci(*division*)	
	%	kalan işleci *(modulus)*	
4	+	toplama işleci (*addition*)	soldan sağa
	-	çıkarma işleci (*subtraction*)	
5	<<	bitsel sola kaydırma işleci*(bitwise shift left)*	soldan sağa
	>>	bitsel sağa kaydırma işleci *(bitwise shift right)*	
6	<	küçüktür işleci (*less than*)	soldan sağa
	>	büyüktür işleci (*greater than*)	
	<=	küçük eşittir işleci (*less than or equal*)	
	>=	büyük eşittir işleci *(greater than or equal)*	
7	==	eşittir işleci *(equal)*	soldan sağa
	!=	eşit değildir işleci *(not equal to)*	
8	&	bitsel ve işleci *(bitwise and)*	soldan sağa
9	^	bitsel özel veya işleci (*bitwise exor*)	soldan sağa
10	|	bitsel veya işleci (*bitwise or*)	soldan sağa
11	&&	mantıksal ve işleci *(logical and)*	soldan sağa
12	||	mantıksal veya işleci (*logical or*)	soldan sağa
13	?:	koşul işleci *(conditional)*	sağdan sola
14	=	atama işleci (*assignement*)	sağdan sola
	+=	toplamalı atama işleci *(assignment with addition)*	
	-=	çıkarmalı atama işleci (*assignment with subtraction*)	
	*=	çarpmalı atama işleci *(assignment with multiplication)*	
	/=	bölmeli atama işleci *(assignment with division)*	
	%=	kalanlı atama işleci *(assignment with modulus)*	
	<<=	bitsel sola kaydırmalı atama işleci *(assignment with bitwise left shift)*	
	>>=	bitsel sağa kaydırmalı atama işleci *(assignment with bitwise right shift)*	
	&=	bitsel ve işlemli atama işleci *(assignment with bitwise and)*	
	|=	bitsel veya işlemli atama işleci *(assignment with bitwise or)*	
	^=	bitsel özel veya işlemli atama işleci *(assignment with bitwise exor)*	
15	,	virgül işleci (*comma*)	soldan sağa

BİLİNİRLİK ALANI VE ÖMÜR

Daha önceki konularda nesnelerin isimlerinden, değerlerinden ve türlerinden söz edilmişti. Nesnelerin C dili açısından çok önem taşıyan üç özelliği daha söz konusudur. Bunlar bilinirlik alanı (*scope*), ömür (*storage duration*) ve bağlantı (*linkage*) özelliğidir.

Bilinirlik Alanı

Bilinirlik alanı (*scope*), bir ismin tanınabildiği program aralığıdır. Derleyiciye bildirilen isimler, derleyici tarafından her yerde bilinmez. Her isim derleyici tarafından ancak "o ismin bilinirlik alanı" içinde tanınabilir. Bilinirlik alanı doğrudan kaynak kod ile ilgili bir kavramdır, dolayısıyla derleme zamanına ilişkindir. C dilinde derleyici, bildirimleri yapılan değişkenlere kaynak kodun ancak belirli bölümlerinde ulaşılabilir. Yani bir değişkenin tanımlanması, o değişkene kaynak dosyanın her yerinden ulaşılabilmesi anlamına gelmez. Bilinirlik alanları C standartları tarafından 4 ayrı grupta toplanmıştır:

i. Dosya Bilinirlik Alanı (*File scope*) : Bir ismin bildirildikten sonra tüm kaynak dosya içinde, yani tanımlanan tüm işlevlerin hepsinin içinde bilinmesidir.

ii. Blok Bilinirlik Alanı (*Block scope*): Bir ismin bildirildikten sonra yalnızca bir blok içinde, bilinmesidir.

iii. İşlev Bilinirlik Alanı (*Function Scope*): Bir ismin, bildirildikten sonra yalnızca bir blok içinde bilinmesidir. Yalnızca *goto* etiketlerini kapsayan özel bir tanımdır. Bu bilinirlik alanına "*goto* deyimi" konusunda değinilecek.

iv. İşlev Bildirimi Bilinirlik Alanı (*Function Prototype Scope*): İşlev bildirimlerindeki, işlev parametre ayracı içinde kullanılan isimlerin tanınabilirliğini kapsayan bir tanımdır. Bu bilinirlik alanına "İşlev Bildirimleri" konusunda değinilecek.

Bir kaynak dosya içinde tanımlanan değişkenler, bilinirlik alanlarına göre "yerel" ve "global" olmak üzere ikiye ayrılabilir:

Yerel Değişkenler

Blokların içinde ya da işlevlerin parametre ayraçları içinde tanımlanan değişkenlere, yerel değişkenler (*local variables*) denir. C dilinde blokların içinde tanımlanan değişkenlerin tanımlama işlemlerinin, bloğun en başında yapılması gerektiğini biliyorsunuz. Yerel değişkenler, blok içinde tanımlanan değişkenlerdir, bir işlevin ana bloğu içinde ya da içsel bir blok içinde bildirilmiş olabilirler.

Yerel değişkenlerin bilinirlik alanı, blok bilinirlik alanıdır. Yani yerel değişkenlere yalnızca tanımlandıkları blok içinde ulaşılabilir. Tanımlandıkları bloğun daha dışındaki bir blok içinde bu değişkenlere erişilemez.

Aşağıdaki programda tanımlanan değişkenlerin hepsi yereldir. Çünkü *x, y, z* isimli değişkenler blokların içinde tanımlanıyor. Bu değişkenler yalnızca tanımlanmış oldukları blok içinde kullanılabilir. Tanımlandıkları blok dışında bunların kullanılması geçersizdir.

Yorum satırları içine alınan deyimler geçersizdir. *z* ve *y* değişkenleri bilinirlik alanlarının dışında kullanılmıştır. Yukarıdaki örnekte değişkenlerin hepsi yerel olduğu için blok bilinirlik alanı kuralına uyar, ancak bu durum, 3 değişkenin de bilinirlik alanının tamamen aynı olduğu anlamına gelmez. Örnek programda *x* değişkeni en geniş bilinirlik alanına sahipken *y* değişkeni daha küçük ve *z* değişkeni de en küçük bilinirlik alanına sahiptir:

```
#include <stdio.h>

int main()
{
    int x = 10;

    printf("x = %d\n", x);
    {
        int y = 20;

        printf("y = %d\n", y);
        x = 30;
        {
            int z = 50;
            y = 60;
            printf("z = %d\n", z);
            printf("x = %d\n", x);
            printf("y = %d\n", y);

        }
        z = 100; /* Geçersiz! */
        y = x;
        printf("x = %d\n", x);
        printf("y = %d\n", y);
    }
    y = 500; /* Geçersiz! */
    printf("x = %d\n", x);

    return 0;
}
```

İşlevlerin parametre değişkenleri de *(formal parameters),* blok bilinirlik alanı kuralına uyar. Bu değişkenler işlevin ana bloğu içinde bilinir. İşlev parametre değişkeninin bilinirlik alanı, işlevin ana bloğunun kapanmasıyla sonlanır. Yani işlev parametre değişkeninin bilinirlik alanı, işlevin ana bloğudur.

```
void func (int a, double b)
{
    /* a ve b bu işlevin her yerinde bilinir. */
}
```

Yukarıdaki örnekte *func* işlevinin parametre değişkenleri olan *a* ve *b* isimli değişkenler, *func* işlevinin her yerinde kullanılabilir.

Global Değişkenler

C dilinde blokların dışında da değişkenlerin tanımlanabileceğini biliyorsunuz. Blokların dışında tanımlanan değişkenler *"global değişkenler" (global variables)* olarak isimlendirilir.

Derleme işleminin bir yönü vardır. Bu yön kaynak kod içinde yukarıdan aşağıya doğrudur. Bir değişken yerel de olsa global de olsa, tanımlaması yapılmadan önce kullanılması geçersizdir. Global değişkenler tanımlandıkları noktadan sonra kaynak dosyanın sonuna kadar her yerde bilinir:

```
#include <stdio.h>

int g;

void func()
{
    g = 10;
}

int main()
{
    g = 20;

    printf("g = %d\n", g);     /* g = 20 */
    func();
    printf("g = %d\n", g);     /* g = 10 */

    return 0;
}
```

Yukarıdaki örnekte *g* değişkeni blok dışında tanımlandığı için -ya da hiçbir işlevin içinde tanımlanmadığı için- global değişkendir. *g* değişkeninin bilinirlik alanı, dosya bilinirlik alanıdır. Yani *g* değişkeni, tanımlandıktan sonra tüm işlevlerin içinde kullanılabilir. Yukarıdaki programda önce *g* global değişkenine *20* değeri atanıyor.Daha sonra bu değer *printf* işleviyle ekrana yazdırılıyor. Daha sonra *func* işlevi çağrılıyor. *func* işlevi çağrılınca kodun akışı *func* işlevine geçer. *func* işlevi içinde de *g* global değişkeni bilinir. *func* işlevinde global *y* değişkenine *10* değerinin atanmasından sonra bu değer yine *printf* işleviyle ekrana yazdırılıyor.

Aynı İsimli Değişkenler

C dilinde aynı isimli birden fazla değişken tanımlanabilir. Genel kural şudur: İki değişkenin bilinirlik alanları aynı ise, bu değişkenler aynı ismi taşıyamaz. Aynı ismi taşımaları derleme zamanında hata oluşturur. İki değişkenin bilinirlik alanlarının aynı olması ne anlama gelir? İki değişkenin bilinirlik alanları, aynı kapanan küme ayracı ile sonlanıyorsa, bu değişkenlerin bilinirlik alanları aynı demektir.

```
{
    float a;
    int b;
    double a;          /* Geçersiz */
    {
        int c;
        /*...*/
    }
}
```

Yukarıdaki kod geçersizdir. Çünkü her iki *a* değişkeninin de bilinirlik alanı aynıdır. Farklı bilinirlik alanlarına sahip birden fazla aynı isimli değişken tanımlanabilir. Çünkü derleyiciler için, artık bu değişkenlerin aynı isimli olması önemli değildir. Bunlar bellekte farklı yerlerde tutulur. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

int main()
{
    int x = 100;

    printf("%d\n", x);
    {
        int x = 200;
        printf("%d\n", x);
        {
            int x = 300;
            printf("%d\n", x);
        }
    }
    return 0;
}
```

Yukarıdaki program parçasında bir hata bulunmuyor. Çünkü her üç *x* değişkeninin de bilinirlik alanları birbirlerinden farklıdır. Peki yukarıdaki örnekte iç bloklarda *x* ismi kullanıldığında derleyici bunu hangi *x* değişkeni ile ilişkilendirir?
Bir kaynak kod noktası, aynı isimli birden fazla değişkenin bilinirlik alanı içinde ise, bu noktada değişkenlerden hangisine erişilir?
Derleyici, bir ismin kullanımı ile karşılaştığında bu ismin hangi yazılımsal varlığa ait olduğunu bulmaya çalışır. Bu işleme "isim arama" (*name lookup*) denir. İsim arama, dar bilinirlik alanından geniş bilinirlik alanına doğru yapılır. Yani derleyici söz konusu ismi önce kendi bloğunda arar. Eğer isim, bu blok içinde tanımlanmamış ise bu kez isim kapsayan bloklarda aranır. İsim, kapsayan bloklarda da bulunamaz ise bu kez global isim alanında aranır.
Dar bilinirlik alanına sahip isim, daha geniş bilinirlik alanında yer alan aynı ismi maskeler, onun görünmesini engeller. Aşağıdaki programı inceleyin:

```
void func1()
{
    int  k;
    /***/
}

void func2()
{
    int  k;
    /***/
}
void func3()
{
    int  k;
    /***/
}
```

Yukarıdaki kod parçasında bir hata söz konusu değildir. Her üç işlevde de *k* isimli bir değişken tanımlanmış olsa da bunların bilinirlik alanları tamamen birbirinden farklıdır.
Bir global değişkenle aynı isimli yerel bir değişken olabilir mi? İki değişkenin bilinirlik alanları aynı olmadığı için bu durum bir hataya neden olmaz.
Aynı isimli hem bir global hem de bir yerel değişkene erişilebilen bir noktada, erişilen yerel değişken olur. Çünkü aynı bilinirlik alanında, birden fazla aynı isimli değişken olması durumunda, o alan içinde en dar bilinirlik alanına sahip olanına erişilebilir. Aşağıdaki kodu inceleyin:

```
#include <stdio.h>

int  g = 20;           /* g global değişken */

void func()
{
    /* global g değişkenine atama yapılıyor. */
    g = 100;
    /* global g değikeninin değeri yazdırılıyor. */
    printf("global g = %d\n", g);
}

int main()
{
    int g; /* g yerel değişken */

    /* yerel g değişkenine atama yapılıyor */
    g = 200;
    /* yerel g yazdırılıyor. */
    printf("yerel g = %d\n", g);
    func();
    /* yerel g yazdırılıyor. */
    printf("yerel g = %d\n", g);

    return 0;
}
```

İşlevlerin kendileri de bütün blokların dışında tanımlandıklarına göre global varlıklardır. Gerçekten de işlevler kaynak kodun her yerinden çağrılabilir. Aynı bilinirlik alanına ilişkin, aynı isimli birden fazla değişken olmayacağına göre, aynı isme sahip birden fazla işlev de olamaz.
[Ancak C++ dilinde isimleri aynı parametrik yapıları farklı işlevler tanımlamak mümkündür.]

Bildirilen bir isme bilinirlik alanı içinde her yerde ulaşılamayabilir. Çünkü bir isim, daha dar bir bilinirlik alanında aynı isim tarafından maskelenmiş olabilir. Bu yüzden "bilinirlik alanı" dışında bir de *"görülebilirlik" (visibility)* teriminden söz edilebilir.
[C++ dilinde global bir ismin yerel bir isim tarafından maskelenmesi durumda, global isme çözünürlük işleci (scope resolution operator) ismi verilen bir işleçle erişim mümkündür.]

Nesnelerin Ömürleri

Ömür *(storage duration / lifespan)*, nesnelerin, programın çalışma zamanı içinde bellekte yer kapladığı süreyi anlatmak için kullanılan bir terimdir. Bir kaynak kod içinde tanımlanmış değişkenlerin hepsi, program çalışmaya başladığında aynı zamanda yaratılmaz. Programlarda kullanılan varlıklar, ömürleri bakımından üç gruba ayrılabilir:

1. Statik ömürlü varlıklar
2. Otomatik ömürlü varlıklar
3. Dinamik Ömürlü varlıklar

i. Statik Ömürlü Varlıklar

Statik ömürlü varlıklar *(static duration – static storage class)*, programın çalışmaya başlamasıyla bellekte yerlerini alır, programın çalışması bitene kadar varlıklarını sürdürür, yani bellekte yer kaplar. Statik ömürlü varlıklar, genellikle amaç kod (*.obj*) içine yazılır. C dilinde statik ömürlü üç ayrı varlık grubu vardır:

global değişkenler
dizgeler (çift tırnak içindeki yazılar)
statik yerel değişkenler

Dizgeler ile statik yerel değişkenleri daha sonra göreceksiniz.
Global değişkenler statik ömürlü varlıklardır. Yani global değişkenler programın çalışması süresince yaşayan, yani programın çalışması süresince bellekte yer kaplayan değişkenlerdir.

ii. Otomatik Ömürlü Varlıklar

Otomatik ömürlü nesneler programın çalışmasının belli bir zamanında yaratılan, belli süre etkinlik gösterdikten sonra yok olan, yani ömürlerini tamamlayan nesnelerdir. Bu tür nesnelerin ömürleri, programın toplam çalışma süresinden kısadır.
Yerel değişkenler, otomatik ömürlüdür. Programın çalışma zamanında tanımlandıkları bloğun çalışması başladığında yaratılırlar, bloğun çalışması bitince yok olurlar, yani ömürleri sona erer.

```
void func(int a, int b)
{
    int result;
    /***/
}
```

Yukarıdaki *func* işlevinin ana bloğu içinde *result* isimli bir yerel değişken tanımlanıyor. Programın çalışması sırasında *func* işlevinin koduna girildiğinde *result* değişkeni yaratılır. Programın akışı *func* işlevinden çıktığında, *result* değişkeninin ömrü sona erer.
Statik ömürlü değişkenlerle otomatik ömürlü değişkenler arasında ilkdeğer verme *(initialization)* açısından da fark vardır. Statik ömürlü olan global değişkenlere de yerel değişkenlerde olduğu gibi ilkdeğer verilebilir.
İlkdeğer verilmemiş ya da bir atama yapılmamış bir yerel değişkenin içinde bir çöp değer bulunur. Bu değer o an bellekte o değişken için ayrılmış yerde bulunan *1* ve *0* bitlerinin oluşturduğu değerdir.
İlkdeğer verilmemiş statik ömürlü değişkenlerin *0* değeri ile başlatılması güvence altındadır. İlk değer verilmemiş ya da bir atama yapılmamış global değişkenler içinde her zaman *0* değeri vardır. Yani bu değişkenler derleyici tarafından üretilen kod yardımıyla *0* değeriyle başlatılır.
Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

int g;

int main()
{
    int y;

    printf("g = %d\n", g);
    printf("y = %d\n", y);  /* Yanlış /

    return 0;
}
```

Bir yerel değişkenin ilkdeğer verilmeden ya da kendisine bir atama yapılmadan kullanılması bir programlama hatasıdır. Derleyicilerin hemen hemen hepsi böyle durumlarda mantıksal bir uyarı iletisi verir.
[C++ dilinde böyle bir zorunluluk yoktur.]

Global değişkenlere ancak değişmez ifadeleriyle ilkdeğer verilebilir. Global değişkenlere ilkdeğer verme işleminde kullanılan ifadede *(initializer*), değişkenler ya da işlev çağrı ifadeleri kullanılamaz. İfade yalnızca değişmezlerden oluşmak zorundadır.

[Global değişkenlere değişmez ifadesi olmayan ifadelerle ilkdeğer verilmesi C++ dilinde geçerlidir. Yeni C derleyicilerin çoğu, global değişkenlere değişmez ifadesi olmayan ifadelerle ilkdeğer verilmesi durumunda da kodu geçerli sayma eğilimindedir. Taşınabilirlik açısından bu durumdan kaçınılmasını salık veriyoruz.]

Ancak yerel değişkenlere ilkdeğer verilme işleminde böyle bir kısıtlama yoktur.

```
#include <stdio.h>

int func(void);

int x = 5;
int y = x + 5;      /* Geçersiz */
int z = func();     /* Geçersiz */

int main()
{
    int a = b;
    int k = b - 2;
    int m = func();
    /***/
}
```

Yukarıdaki programda *main* işlevi içinde *a, k, m* değişkenlerinin tanımlanmaları geçerlidir.

iii. Dinamik Ömürlü Varlıklar

Dinamik bellek işlevleri ile yerleri ayrılmış nesneler, dinamik ömürlüdür. Dinamik bellek işlevleri ile yaratılmış nesneleri daha sonra göreceksiniz.

Global ve Yerel Değişkenlerin Karşılaştırılması

Bir programda bir değişken gereksinimi durumunda, global ya da yerel değişken kullanılması bazı avantajlar ya da dezavantajlar getirebilir. Ancak genel olarak global değişkenlerin bazı sakıncalarından söz edilebilir. Özel bir durum söz konusu değil ise, yerel değişkenler global değişkenlere tercih edilmeli, global değişkenler ancak zorunlu durumlarda kullanılmalıdır. Global değişkenler aşağıdaki sakıncalara neden olabilir:

1.Global değişkenler statik ömürlü olduklarından programın sonuna kadar bellekte yerlerini korur. Bu nedenle belleğin daha verimsiz olarak kullanılmalarına neden olurlar.

2. Global değişkenler tüm işlevler tarafından ortaklaşa paylaşıldığından, global değişkenlerin çokça kullanıldığı kaynak dosyaları okumak daha zordur.

3. Global değişkenlerin sıkça kullanıldığı bir kaynak dosyada, hata arama maliyeti daha yüksektir. Global değişkene ilişkin bir hata söz konusu ise, bu hatayı bulmak için tüm işlevler araştırılmalıdır. Tüm işlevlerin global değişkenlere ulaşabilmesi, bir işlevin global bir değişkeni yanlışlıkla değiştirebilmesi riskini de doğurur.

4. Global değişkenlerin kullanıldığı bir kaynak dosyada, değişiklik yapmak da daha fazla çaba gerektirir. Kaynak kodun çeşitli bölümleri, birbirine global değişken kullanımlarıyla sıkı bir şekilde bağlanmış olur. Bu durumda kaynak kod içinde bir yerde değişiklik yapılması durumunda başka yerlerde de değişiklik yapmak gerekir.

5. Programcıların çoğu, global değişkenleri mümkün olduğu kadar az kullanmak ister. Çünkü global değişkenleri kullanan işlevler, başka projelerde kolaylıkla kullanılamaz. Kullanıldıkları projelerde de aynı global değişkenlerin tanımlanmış olması gerekir. Dolayısıyla global değişkenlere dayanılarak yazılan işlevlerin yeniden kullanılabilirliği azalır.

6. Global değişkenler global isim alanını kirletir. Bu noktaya ileride "bağlantı" kavramı ele alındığı zaman yeniden değinilecek.

İşlevlerin Geri Dönüş Değerlerini Tutan Nesneler

İşlevler geri dönüş değerlerini, geçici bir nesne yardımıyla kendilerini çağıran işlevlere iletir. Aşağıdaki programı inceleyin:

```
#include <stdio.h>

int add(int x, int y)
{
    return x + y;
}

int main()
{
    int a, b, sum;

    printf("iki sayı girin: ");
    scanf("%d%d",  &a, &b);
    sum = add(a, b);
    printf("toplam = %d\n", sum);

    return 0;
}
```

Bir işlevin geri dönüş değerinin türü aslında, işlevin geri dönüş değerini içinde taşıyacak geçici nesnenin türü demektir. Yukarıda tanımı verilen *add* isimli işlevin *main* işlevi içinden çağrıldığını görüyorsunuz. Programın akışı, *add* işlevi içinde *return* deyimine geldiğinde, geçici bir nesne yaratılır. Bu geçici nesne, *return* ifadesiyle ilkdeğerini alır. Yani *return* ifadesi aslında oluşturulan geçici nesneye ilkdeğerini veren ifadedir. Geri dönüş değeri üreten bir işleve yapılan çağrı, bu işlevin geri dönüş değerini içinde tutan geçici nesneyi temsil eder. Peki bu geçici nesnenin ömrü ne kadardır? Bu nesne, *return* deyimiyle yaratılır ve işlev çağrısını içeren ifadenin değerlendirilmesi sona erince yok edilir. Yani örnekteki *main* işlevi içinde yer alan

```
sum = add(a, b);
```

deyiminin yürütülmesinden sonra, geçici nesnenin ömrü de sona erer.

KONTROL DEYİMLERİ

C dilinde yazılmış bir programın cümlelerine deyim(*statement)* dendiğini biliyorsunuz. Bazı deyimler, yalnızca derleyici programa bilgi verir. Bu deyimler derleyicinin işlem yapan bir kod üretmesine neden olmaz. Böyle deyimlere "bildirim deyimi" (*declaration statement*) denir.
Bazı deyimler derleyicinin işlem yapan bir kod üretmesine neden olur. Böyle deyimlere "yürütülebilir deyim" (*executable statement*) denir.
Yürütülebilir deyimler de farklı gruplara ayrılabilir:

Yalın Deyim:

Bir ifadenin, sonlandırıcı atom ile sonlandırılmasıyla oluşan deyimlere yalın deyim *(simple statement)* denir;

```
x = 10;
y++;
func();
```

Yukarıda 3 ayrı yalın deyim yazılmıştır.

Boş Deyim:

C dilinde tek başına bulunan bir sonlandırıcı atom ';', kendi başına bir deyim oluşturur. Bu deyime boş deyim *(null statement)* denir. Boş bir blok da boş deyim oluşturur:

```
;
{}
```

Yukarıdaki her iki deyim de boş deyimdir.

Bileşik Deyim:

Bir blok içine alınmış bir ya da birden fazla deyimin oluşturduğu yapıya, bileşik deyim *(compound statement)* denir. Aşağıda bir bileşik deyim görülüyor.

```
{
    x = 10;
    y++;
    func();
}
```

Kontrol deyimi:

Kontrol deyimleri, programın akış yönünü değiştirebilen deyimlerdir. Kontrol deyimleri ile programın akışı farklı noktalara yönlendirilebilir. Bunlar, C dilinin önceden belirlenmiş bazı sözdizimi kurallarına uyar, kendi sözdizimleri içinde en az bir anahtar sözcük içerir. C dilinde aşağıdaki kontrol deyimleri vardır:

if deyimi
while döngü deyimi
do while döngü deyimi
for döngü deyimi
break deyimi
continue deyimi
switch deyimi
goto deyimi
return deyimi

if DEYİMİ

C dilinde program akışını denetlemeye yönelik en önemli deyim *if* deyimidir.
En yalın biçimiyle *if* deyiminin genel sözdizimi aşağıdaki gibidir:

```
if (ifade)
    deyim;
```

if ayracı içindeki ifadeye koşul ifadesi *(conditional expression*) denir.
if ayracını izleyen deyime, *if* deyiminin doğru kısmı *(true path*) denir.

if deyiminin doğru kısmını oluşturan deyim, bir yalın deyim (*simple statement*) olabileceği gibi, bir boş deyim (*null statement),* bir bileşik deyim *(compound statement*) ya da başka bir kontrol deyimi de *(control statement)* olabilir.

Yalın *if* deyiminin yürütülmesi aşağıdaki gibi olur:
Önce koşul ifadesinin sayısal değerini hesaplar. Hesaplanan sayısal değer, mantıksal *DOĞRU* ya da *YANLIŞ* olarak yorumlanır. Koşul ifadesinin hesaplanan değeri *0* ise yanlış, *0*'dan farklı bir değer ise doğru olarak yorumlanır. Örneğin koşul ifadesinin hesaplanan değerinin *–5* olduğunu düşünelim. Bu durumda kontrol ifadesi doğru olarak değerlendirilir. Eğer ifade *DOĞRU* olarak yorumlanırsa, *if* deyiminin doğru kısmı yapılır, ifade *YANLIŞ* olarak yorumlanırsa doğru kısmı yapılmaz. Yalın *if* deyimi, bir ifadenin doğruluğuna ya da yanlışlığına göre, bir deyimin yapılması ya da yapılmamasına dayanır. Aşağıdaki programı derleyerek çalıştırın:

```
int main()
{
    int x;

    printf("bir sayi girin : ");
    scanf("%d", &x);

    if (x > 10)
        printf("if deyiminin doğru kısmı!\n");

    return 0;
}
```

main işlevi içinde yazılan *if* deyimiyle, klavyeden girilen tamsayının *10*'dan büyük olması durumunda *printf* çağrısı yürütülür, aksi halde yürütülmez.

Yanlış Kısmı Olan if Deyimi

if kontrol deyimi, *else* anahtar sözcüğünü de içerebilir. Böyle *if* deyimine, yanlış kısmı olan *if* deyimi denir. Yanlış kısmı olan *if* deyiminin genel biçimi aşağıdaki gibidir:

```
if (ifade)
deyim1;
else
    deyim2;
```

Bu kez *if* deyiminin doğru kısmını izleyen deyimden sonra *else* anahtar sözcüğünün, daha sonra ise bir başka deyimin yer aldığını görüyorsunuz. Genel biçimdeki *deyim2*'ye *if* deyiminin yanlış kısmı *(false path)* denir.
if deyiminin koşul ifadesi, mantıksal olarak *DOĞRU* ya da *YANLIŞ* olarak yorumlanır. Bu kez koşul ifadesinin *DOĞRU* olması durumunda *deyim1, YANLIŞ* olarak yorumlanması durumunda *deyim2* yapılır. Yanlış kısmı olan *if* deyimi, bir koşul ifadesinin doğru ya da

yanlış olmasına göre iki ayrı deyimden birinin yapılmasına yöneliktir. Yani ifade doğru ise bir iş, yanlış ise başka bir iş yapılır.
Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

int main()
{
    char ch;

    printf("bir karakter girin : ");
    ch = getchar();

    if (ch >= 'a' && ch <= 'z')
        printf("%c kucuk harf!\n", ch);
    else
        printf("%c kucuk harf degil!\n", ch);

    return 0;
}
```

Yukarıdaki *main* işlevinde standart *getchar* işlevi kullanılarak klavyeden bir karakter alınıyor. Alınan karakterin sıra numarası, *ch* isimli değişkene atanıyor. Koşul ifadesinin doğru ya da yanlış olması durumuna göre, klavyeden alınan karakterin küçük harf olup olmadığı bilgisi ekrana yazdırılıyor. Koşul ifadesine bakalım:

```
ch >= 'a' && ch <= 'z'
```

Bu ifadenin doğru olması için "mantıksal ve *(&&)*" işlecinin her iki teriminin de doğru olması gerekir. Bu da ancak, *ch* karakterinin küçük harf karakteri olması ile mümkündür.

if deyiminin doğru ve/veya yanlış kısmı bir bileşik deyim olabilir. Bu durumda, koşul ifadesinin doğru ya da yanlış olmasına göre, birden fazla yalın deyimin yürütülmesi sağlanabilir. Aşağıdaki örneği inceleyin:

```
/***/
if (x > 0) {
    y = x * 2 + 3;
    z = func(y);
    result = z + x;
}
else {
    y = x * 5 - 2;
    z = func(y - 2);
    result = z + x - y;
}
/***/
```

Yukarıdaki *if* deyiminde, *x > 0* ifadesinin doğru olup olmasına göre, *result* değişkeninin değeri farklı işlemlerle hesaplanıyor. *if* deyiminin hem doğru hem de yanlış kısımlarını bileşik deyimler oluşturuyor.
Bir *if* deyiminin yanlış kısmı olmak zorunda değildir. Ancak bir if deyimi yalnızca *else* kısmına sahip olamaz. Bu durumda *if* deyiminin doğru kısmına boş deyim ya da boş bileşik deyim yerleştirilmelidir:

```
if (ifade)
    ;
else
   deyim1;
```

ya da

```
if (ifade)
    { }
else
    deyim1;
```

Yalnızca yanlış kısmı olan, doğru kısmı bir boş deyim olan bir *if* deyimi, okunabilirlik açısından iyi bir seçenek değildir. Böyle durumlarda daha iyi bir teknik, koşul ifadesinin mantıksal tersini alıp, *if* deyiminin yanlış kısmını ortadan kaldırmaktır:

```
if (!ifade)
    deyim1;
```

Aşağıdaki kod parçasını inceleyin:

```
/***/
if (x > 5)
    ;
else {
    func1(x);
    func2(x);
}
/***/
```

Yukarıdaki *if* deyiminde, *x* değişkeninin değeri *5*'ten büyükse bir şey yapılmıyor, aksi halde *func1* ve *func2* işlevleri *x* değişkeninin değeri ile çağrılıyor. Koşul ifadesi ters çevrilerek *if* deyimi yeniden yazılırsa:

```
/***/
if (x <= 5) {
    func1(x);
    func2(x);
}
/***/
```

if ayracının içinde, ifade tanımına uygun herhangi bir ifade bulunabilir:

```
if (10)
    deyim1;

if (-1)
   deyim2;
```

Yukarıdaki koşul ifadelerinin değeri, her zaman doğru olarak yorumlanır. Çünkü ifadeler, sıfırdan farklı değere sahiptir.

Aşağıdaki koşul ifadesi ise her zaman yanlış olarak yorumlanacağından *if* deyiminin doğru kısmı hiçbir zaman yürütülmez:

```
if (0)
    deyim1;
```

Aşağıdaki *if* deyiminde ise, *x* değişkeninin değerinin *0* olup olmamasına göre, *deyim1* ve *deyim2* yürütülür:

```
if (x) {
    deyim1;
    deyim2;
    /***/
}
```

Yukarıdaki yapıyla aşağıdaki yapı eşdeğerdir:

```
if (x != 0) {
    deyim1;
    deyim2;
}
```

Aşağıdaki örneği inceleyin :

```
if (!x) {
    deyim1;
    deyim2;
}
```

Bu *if* deyiminde ise ancak *x* değişkeninin değerinin *0* olması durumunda *deyim1* ve *deyim2* yürütülür.
Yine yukarıdaki yapıyla aşağıdaki yapı eşdeğerdir:

```
if (x == 0) {
    deyim1;
    deyim2;
}
```

Koşul İfadesinde Atama İşlecinin Kullanılması

if deyiminin koşul ifadesinde atama işleci sıklıkla kullanılır. Böylece, atama işlecinin ürettiği değerden faydalanılır: Aşağıdaki kod parçasını inceleyin:

```
if ((x = getval()) > 5)
    func1(x);
else
    func2(x);
```

if deyiminin koşul ifadesinde ise, çağrılan *getval* işlevinin geri dönüş değeri, *x* değişkenine aktarılıyor. Atama işlecinin ürettiği değerin nesneye atanan değer olduğunu anımsayın. Atama işleci ile oluşturulan ifadenin, öncelik ayracı içine alındığını görüyorsunuz. Bu durumda hem *getval* işlevinin geri dönüş değeri *x* değişkenine aktarılıyor hem de işlevin geri dönüş değerinin *5't*en büyük olup olmadığı sorgulanıyor. Öncelik ayracı kullanılmasaydı *getval* işlevinin geri dönüş değerinin *5*'ten büyük olup olmamasına göre *x* değişkenine *0* ya da *1* değeri atanırdı. Bu durumda da ya *func1* işlevi *1* değeriyle ya da *func2* işlevi *0* değeriyle çağrılırdı. Deyim aşağıdaki gibi de yazılabilirdi, değil mi?

```
x = getval();
if (x > 5)
    func1(x);
else
    func2(x);
```

Ancak kalıp kod, daha karmaşık deyimlerin yazılmasında kolaylık sağlar. Aşağıdaki *if* deyiminin nasıl yürütüleceğini düşünün. "Mantıksal ve" işlecinin birinci kısmının, daha önce ele alınmasının, yani "kısa devre" davranışının güvence altında olduğunu anımsayın.

```
if ((y = getval()) > 5 && isprime(x))
    func1(y);
func2(y);
```

İç İçe if Deyimleri

if deyiminin doğru ya da yanlış kısmını, başka bir *if* deyimi de oluşturabilir:

```
if (ifade1)
    if (ifade2) {
        deyim1;
        deyim2;
        deyim3;
    }
deyim4;
```

Bu örnekte ikinci *if* deyimi birinci *if* deyiminin doğru kısmını oluşturur. Birinci ve ikinci *if* deyimlerinin yanlış kısımları yoktur.
İç içe *if* deyimlerinde, son *if* anahtar sözcüğünden sonra gelen *else* anahtar sözcüğü, en içteki *if* deyimine ait olur:

```
if (ifade1)
    if (ifade2)
        deyim1;
else
    deyim2;
```

Yukarıdaki örnekte, yazım biçimi nedeniyle *else* kısmının birinci *if* deyimine ait olması gerektiği gibi bir görüntü verilmiş olsa da, *else* kısmı ikinci *if* deyimine aittir. *else* anahtar sözcüğü, bu gibi durumlarda, kendisine yakın olan *if* deyimine ait olur *(dangling else)*.
else anahtar sözcüğünün birinci *if* deyimine ait olması isteniyorsa, birinci *if* deyiminin doğru kısmı bloklanmalıdır:

```
if (ifade1) {
    if (ifade2)
        deyim1;
}
else
    deyim2;
```

Yukarıdaki örnekte *else* kısmı birinci *if* deyimine aittir.

```
if (ifade1) {
    if (ifade2)
        deyim1;
    else {
        deyim2;
        deyim3;
    }
    deyim4;
}
else
    deyim5;
```

Yukarıdaki örnekte birinci *if* deyiminin doğru kısmı, birden fazla deyimden oluştuğu için - bu deyimlerden birisi de yine başka bir *if* deyimidir- bloklama yapılıyor. *deyim5,* birinci *if* deyiminin yanlış kısmını oluşturur.

else if Merdiveni

Aşağıdaki *if* deyimlerini inceleyin:
Eğer bir karşılaştırmanın doğru olarak sonuçlanması durumunda yapılan diğer karşılaştırmaların doğru olması söz konusu değilse, bu tür karşılaştırmalara ayrık karşılaştırma denir. Ayrık karşılaştırmalarda, if deyimlerinin ayrı ayrı kullanılması kötü tekniktir:

```
if (m == 1)
    printf("Ocak\n");
if (m == 2)
    printf(Şubat\n");
if (m == 3)
    printf("Mart\n");
/***/
if (m == 12)
    printf("Aralık\n");
```

Yukarıdaki örnekte *m* değişkeninin değerinin *1* olduğunu düşünün. Bu durumda ekrana *Ocak* yazısı yazdırılır. Fakat daha sonra yer alan *if* deyimleriyle *m* değişkeninin sırasıyla *2, 3, ... 12*'ye eşit olup olmadığı ayrı ayrı sınanır. Ama *x* değişkeni *1* değerine sahip olduğundan, bütün diğer *if* deyimleri içindeki kontrol ifadelerinin *yanlış* olarak değerlendirileceği bellidir. Bu durumda birinci *if* deyiminden sonraki bütün *if* deyimleri gereksiz yere yürütülmüş olur. Aynı zamanda kodun okunabilirliği de bozulur.

Ayrık karşılaştırmalarda *else if* merdivenleri kullanılmalıdır:

```
if (ifade1)
    deyim1;
else
    if (ifade2)
        deyim2;
else
    if (ifade3)
            deyim3;
        else
            if (ifade4)
                deyim4;
            else
                deyim5;
```

Bu yapıda, herhangi bir *if* deyiminin koşul ifadesi doğru olarak değerlendirilirse programın akışı hiçbir zaman başka bir if deyimine gelmez. Bu yapıya, *else if* merdiveni (*cascaded if / else if ladder*) denir. *else if* merdivenlerinin yukarıdaki biçimde yazılışı, özellikle uzun *else if* merdivenlerinde okunabilirliği bozduğu için aşağıdaki yazım biçimi, okunabilirlik açısından tercih edilmelidir:

```
if (ifade1)
    deyim1;
else if (ifade2)
    deyim2;
else if (ifade3)
    deyim3;
else if (ifade4)
    deyim4;
else
    deyim5;
```

Merdiveninin en sonundaki *if* deyiminin yanlış kısmının özel bir önemi vardır. Yukarıdaki örnekte *deyim5,* merdivenin son *if* deyiminin *else* kısmında yer alıyor. Merdiven içindeki hiçbir *if* deyiminin koşul ifadesi doğru değilse, son *if* deyimininin yanlış kısmı yapılır, değil mi? Yukarıdaki merdivenin yürütülmesi sonucu, *deyim1, deyim2, deyim3, deyim4, deyim5*'den biri mutlaka yapılır.
Son basamaktaki *if* deyiminin yanlış kısmı olmayan bir *else if* merdiveninden, hiçbir iş yapılmadan da çıkılabilir.
Hem okunabilirlik açısından hem de verim açısından, *else if* merdiveninde olasılığı ya da sıklığı daha yüksek olan koşullar, daha yukarıya kaydırılmalıdır.

Sık Yapılan Hatalar

Özellikle C'ye yeni başlayanların sık yaptığı bir hata, yalın *if* deyimiyle yanlış kısmı olan *if* deyimini birbirine karıştırmaktır. Yani *if* deyiminin yanlış kısmı unutulur:

```
#include <stdio.h>

int main()
{
    int x;

    printf("bir sayi girin: ");
    scanf("%d", &x);

    if (x % 2 == 0)
        printf("%d cift sayi!\n", x);
    printf("%d teksayi!\n", x);

    return 0;
}
```

if ayracı içindeki ifadenin yanlış olması durumunda bir yanlışlık söz konusu değildir. Ama ifade doğru ise ekrana ne yazılır? Klavyeden 28 değerinin girildiğini düşünelim:

```
28 çift sayi!
28 tek sayi!
```

Belki de en sık yapılan hata, *if* ayracının sonuna yanlışlıkla sonlandırıcı atomun *(;)* yerleştirilmesidir. Aşağıdaki *main* işlevini inceleyin:

```
#include <stdio.h>

int main()
{
    int x;

    printf("bir sayi girin: ");
    scanf("%d", &x);

    if (x % 2 == 0);
        printf("%d cift sayi!\n", x);

    return 0;
}
```

Yukarıdaki *main* işlevinde, *x % 2 == 0* ifadesi doğru da olsa yanlış da olsa *printf* işlevi çağrılır. *printf* çağrısı *if* deyiminin dışındadır. *if* deyiminin doğru kısmını bir boş deyimin *(null statement)* oluşturması sözdizim kurallarına kesinlikle uyan bir durumdur. Yazılan *if* deyimi, gerçekte *"x çift ise bir şey yapma"* anlamına gelir. Bilinçli bir biçimde yazılma

olasılığı yüksek olmayan bu durum için, derleyicilerin çoğu mantıksal bir uyarı iletisi vermez.
Aynı hata, yanlış kısmı olan bir *if* deyiminin doğru kısmında yapılsaydı bir sözdizim hatası oluşurdu, değil mi?

```
if (x > 5);
    printf("doğru!\n");
else
    printf("yanlış\n");
```

else anahtar sözcüğünün, bir *if* deyimine bağlı olarak kullanılması gerektiğini biliyorsunuz. Yukarıdaki kod parçasındaki *if* deyimi, doğru kısmı *"hiçbir şey yapma"* anlamına gelen bir *if* deyimidir. Dolayısıyla, *else* anahtar sözcüğü hiçbir *if* deyimine bağlanmamış olur. Bu da bir sözdizim hatasıdır. Çünkü bir *if deyimine bağlanmayan,* bir *else* olamaz.

Tabi ki bir *if* deyiminin doğru ya da yanlış kısmını bir boş deyim (*null statement*) oluşturabilir. Bu durumda okunabilirlik açısından, bu boş deyimin bir *tab* içeriden yazılması, boş deyimin bilinçli olarak yerleştirildiği konusunda güçlü bir izlenim verir.

```
if ((val = getval()) != 0)
    ;
```

Sık yapılan başka bir yanlışlık, *if* ayracı içinde karşılaştırma işleci *(==)* yerine atama işlecinin *(=)* kullanılmasıdır.

```
/***/
if (x == 5)
    printf("eşit\n");
/***/
```

Yukarıdaki *if* deyiminde, x değişkeninin değeri *5*'e eşitse *printf* işlevi çağrılıyor. Karşılaştırma işlecinin yan etkisi yoktur. Yani yukarıdaki *if* ayracı içinde *x* değişkeninin değeri, *5* değişmezi ile yalnızca karşılaştırılıyor, değiştirilmiyor. Oysa karşılaştırma işlecinin yerine yanlışlıkla atama işleci kullanılırsa:

```
/***/
if (x = 5)
    printf("eşit\n");
/***/
```

Atama işleci, atama işlecinin sağ tarafındaki ifadenin değerini üretir, *if* ayracı içindeki ifadenin değeri *5* olarak hesaplanır. *5,* sıfır dışı bir değer olduğundan, *x* değişkeninin değeri ne olursa olsun, *printf* işlevi çağrılır. Atama işlecinin yan etkisi olduğundan, *x* değişkenine de *if* deyiminin yürütülmesi ile *5* değeri atanır.
C derleyicilerinin çoğu, *if* ayracı içindeki ifade yalın bir atama ifadesi ise, durumu şüpheyle karşılayarak, mantıksal bir uyarı iletisi verir. Örneğin *Borland* derleyicilerinde tipik bir uyarı iletisi aşağıdaki gibidir:

```
warning : possibly incorrect assignment! (muhtemelen yanlış atama!)
```

Oysa *if* ayracı içinde atama işleci bilinçli olarak da kullanılabilir:

```
if (x = func())
    m = 20;
```

Bilinçli kullanımda, derleyicinin mantıksal uyarı iletisinin kaldırılması için, ifade aşağıdaki gibi düzenlenebilir:

```
if ((x = func()) != 0)
    m = 20;
```

Yukarıdaki örnekte olduğu gibi, atama işlecinin ürettiği değer, açık olarak bir karşılaştırma işlecine terim yapılırsa, derleyiciler bu durumda bir "mantıksal uyarı" iletisi vermez.
Çok yapılan başka bir hata da, *if* deyiminin doğru ya da yanlış kısmını bloklamayı unutmaktır. Yani bir bileşik deyim yerine yanlışlıkla yalın deyim yazılır.

```
if (x == 10)
    m = 12;
    k = 15;
```

Yukarıdaki *if* deyiminde yalnızca

```
m = 12;
```

deyimi *if* deyiminin doğru kısmını oluşturur.

```
k = 15;
```

deyimi, *if* deyimi dışındadır. Bu durum genellikle programcının, *if* deyiminin doğru ya da yanlış kısmını önce yalın bir deyimle oluşturmasından sonra, doğru ya da yanlış kısma ikinci bir yalın deyimi eklerken, bloklamayı unutması yüzünden oluşur.
Kodun yazılış biçiminden de, *if* deyiminin doğru kısmının, yanlışlıkla bloklanmadığı anlaşılıyor. Doğrusu aşağıdaki gibi olmalıydı:

```
if (x == 10) {
    m = 12;
    k = 15;
}
```

Aşağıdaki *if* deyimi ise, yine *if* anahtar sözcüğü ile eşlenmeyen bir *else* anahtar sözcüğü kullanıldığı için geçersizdir:

```
if ( x == 10)
    m = 12;
    k = 15;
else    /* Geçersiz */
    y = 20;
```

Bu tür yanlışlıklardan sakınmak için bazı programcılar, *if* deyiminin doğru ya da yanlış kısmı yalın deyimden oluşsa da, bu basit deyimi bileşik deyim olarak yazarlar:

```
if (x > 10) {
    y = 12;
}
else {
    k = 5;
}
```

Yukarıdaki örnekte *if* deyiminin doğru ya da yanlış kısmına başka bir yalın deyimin eklenmesi durumunda sözdizim hatası ya da bir yanlışlık oluşmaz. Ancak gereksiz bloklamadan kaçınmak okunabilirlik açısından daha doğrudur.

if ayracı içinde, bir değerin belirli bir aralıkta olup olmadığının sınanmak istendiğini düşünün:

```
if (10 < x < 20)
    func();
```

Yukarıdaki *if* deyiminde, *x* değişkeni *10 – 20* değerleri arasında ise *func* işlevinin çağrılması istenmiş. Ancak *if* ayracı içinde yer alan ifade her zaman doğrudur. Yani *func* işlevi çağrısı her zaman yürütülür. Küçüktür işleci soldan sağa öncelik yönüne sahip olduğu için, önce daha soldaki küçüktür işleci değer üretir. İşlecin ürettiği değerin *1* ya da *0* olduğunu biliyorsunuz. Üretilen *1* ya da *0* değeri, daha sağdaki küçüktür işlecinin terimi olur. *20* değeri, *1* ya da *0*'dan daha büyük olduğuna göre, ifade her zaman doğrudur. Tehlikeli bir başka yanlışlık da, *if* ayracı içindeki ifadenin bir işlev çağrı ifadesi olması durumunda, işlev çağrı işlecinin unutulmasıdır:

```
if (func())
    m = 12;
```

yerine

```
if (func)
    m = 12;
```

yazıldığını düşünün. Bu durum bir sözdizim hatası oluşturmaz. Bu durumda her zaman, *if* deyiminin doğru kısmı yürütülür. C dilinde bir işlev ismi, o işlevin kodunun bellekteki yerine eşdeğer bir adres bilgisi olarak ele alınır. Bu adres bilgisi her zaman sıfırdan farklı bir değer olduğundan, koşul ifadesi her zaman doğru olarak değerlendirilir.

Sınama İşlevleri

bool veri türü, C'nin doğal veri türlerinden olmadığı için, *C* dilinde yazılan sınama işlevleri, yani bir soruya yanıt veren işlevler çoğunlukla *int* türüne geri döner.

[*C99* standartlarıyla *bool* türü de doğal veri tür olarak eklenmiştir. C99 standartları ile C diline göre _bool anahtar sözcüğü eklenmiştir.]

Örneğin bir tamsayının asal sayı olup olmadığını sınayan bir işlev yazıldığını düşünelim. İşlevin parametrik yapısı aşağıdaki gibi olur:

```
int isprime(int val);
```

Sınama işlevlerinin geri dönüş değerlerinde yaygın olarak kullanılan anlaşma şöyledir: Eğer işlev, sorulan soruya doğru ya da olumlu yanıt veriyorsa, *0* dışı herhangi bir değerle geri döner. Sorulan sorunun ya da yapılan sınamanın sonucu, olumsuz ya da yanlış ise, işlev *0* değeriyle geri döner. Bu durum sınama işlevini çağıran kod parçasının işini kolaylaştırır, aşağıdaki gibi kalıp kodların yazılmasına olanak verir:
Sınamanın olumlu sonuçlanması durumunda bir iş yapılacaksa aşağıdaki gibi bir deyim yazılabilir:

```
if (isprime(val))
    deyim;
```

Sınamanın olumsuz sonuçlanması durumunda bir iş yapılacaksa

```
if (!isprime(val))
    deyim;
```

yazılabilir.
Aşağıdaki programı inceleyin:

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

int main()
{
    int ch;

    printf("bir karakter girin : ");
    ch = getchar();
    if (isupper(ch))
        printf("%c buyuk harf!\n", ch);
    else
        printf("%c buyuk harf degil!\n", ch);

    return 0;
}
```

isupper, kendisine kod numarası gönderilen karakterin, büyük harf karakteri olup olmadığını sınayan standart bir işlevdir. Eğer kod numarasını aldığı karakter büyük harf ise işlev, sıfırdan farklı bir değere geri döner. Büyük harf değil ise, işlevin geri dönüş değeri *0*'dır. Bu durumda *main* işlevinde yer alan *if* deyiminin koşul ifadesi "ch büyük harf ise" anlamına gelir, değil mi? Koşul ifadesi

```
if (!isupper(ch))
```

biçiminde yazılsaydı, bu *"ch büyük harf değil ise"* anlamına gelirdi.

Aşağıda, bir yılın artık yıl olup olmadığını sınayan *isleap* isimli bir işlev tanımlanıyor. *4*'e tam bölünen yıllardan, *100*'e tam bölünmeyenler ya da *400*'e tam bölünenler artık yıldır:

```
#include <stdio.h>

int isleap(int  y)
{
    return y % 4 == 0 && (y % 100 != 0 || y % 400 == 0);
}

int main()
{
    int year;

    printf("bir yil girin: ");
    scanf("%d", &year);

    if (isleap(year))
        printf("%d yili artik yildir!\n", year);
    else
        printf("%d yili artik yil degildir!\n", year);

    return 0;
}
```

Standart Karakter Sınama İşlevleri

Karakter sınama işlevleri, karakterler hakkında bilgi edinilmesini sağlayan işlevlerdir. Derleyicilerin çoğunda bu işlevler, *ctype.h* başlık dosyası içinde aynı zamanda makro olarak tanımlanır. Bu nedenle, karakter sınama işlevleri çağrılmadan önce kaynak koda *ctype.h* dosyası mutlaka eklenmelidir. Karakter sınama işlevleri, *ASCII* karakter repertuarının ilk yarısı için geçerlidir. Yani Türkçe karakterler için kullanılması durumunda

geri dönüş değerleri güvenilir değildir. Karakter sınama işlevlerinin Türkçemize özel *ç, ğ, ı, ö, ş, ü, Ç, Ğ, I, Ö, Ş, Ü*, karakterleri için doğru olarak çalıştırılması, yerelleştirme *(localization*) ile ilgili bir konudur. Bu konuya daha sonraki bölümlerde değinilecek. Aşağıda, standart karakter sınama işlevlerini içeren bir tablo veriliyor:

İşlev	**Geri Dönüş Değeri**
isalpha	Alfabetik karakterse doğru, değilse yanlış.
isupper	Büyük harf ise doğru, değilse yanlış.
islower	Küçük harf ise doğru, değilse yanlış.
isdigit	Sayısal bir karakterse doğru, değilse yanlış.
isxdigit	Onaltılık sayı sistemi basamak simgelerinden birini gösteren bir karakterse, yani *0123456789ABCDEFabcdef* karakterlerinden biri ise doğru, değilse yanlış.
isalnum	Alfabetik ya da sayısal bir karakterse doğru, değilse yanlış.
isspace	Boşluk karakterlerinden biri ise (*space, carriage return, new line, vertical tab, form feed*) doğru, değilse yanlış.
ispunct	Noktalama karakterlerinden biriyse, yani kontrol karakterleri, alfanümerik karakterler ve boşluk karakterlerinin dışındaki karakterlerden ise doğru, değilse yanlış.
isprint	Ekranda görülebilen yani print edilebilen bir karakterse (space karakteri dahil) doğru, değilse yanlış.
isgraph	Ekranda görülebilen bir karakterse (space karakteri dahil değil) doğru, değilse yanlış.
iscntrl	Kontrol karakteri ya da silme karakteri ise (ASCII setinin ilk 32 karakter ya da 127 numaralı karakter) doğru, değilse yanlış.

Bu işlevlerden bazılarını kendimiz yazmaya çalışalım:

islower İşlevi

islower, kendisine kod numarası gönderilen karakterin, küçük harf karakteri olup olmadığını sınayan, standart bir işlevdir. Kod numarasını aldığı karakter küçük harf ise işlev, *sıfır* dışı bir değere, yani mantıksal "doğru" değerine geri döner. Küçük harf değil ise işlevin geri dönüş değeri *sıfır* değeridir. Bu işlev aşağıdaki biçimde yazılabilir:

```
#include <stdio.h>

int islower (int ch)
{
     return ch >= 'a' && ch <= 'z';
}

int main()
{
    char ch;

    printf("bir karakter girin: ");
    ch = getchar();

    if (islower(ch))
        printf("küçük harf\n");
    else
        printf("küçük  harf değil\n");

    return 0;
}
```

Yukarıda yazılan *islower* işlevinde önce parametre değişkeninin küçük harf olup olmadığı sınanıyor:

```
ch >= 'a' && ch <= 'z';
```

Küçük harflerin ardışık olarak yerleştirildiği bir karakter kodunda, yukarıdaki ifadenin değeri, ancak *ch* değişkeninin değerinin, bir küçük harfin sıra numarası olması durumunda doğrudur.

isalpha İşlevi

isalpha da standart bir işlevdir. Parametresine kod numarası aktarılan karakter alfabetik karakterse, yani büyük ya da küçük harf ise işlev sıfır dışı bir değere, alfabetik bir karakter değilse sıfır değerine geri döner.

```
int isalpha (int ch)
{
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}
```

isdigit İşlevi

isdigit, standart bir işlevdir. Parametresine kod numarası aktarılan karakter bir rakam karakteri ise, işlev sıfırdan farklı değer ile, rakam karakteri değilse sıfır değeri ile geri döner:

```
int isdigit (int ch)
{
    return (ch >= '0' && ch <= '9');
}
```

isalnum İşlevi

isalnum da standart bir işlevdir. Parametresine kod numarası aktarılan karakter alfabetik karakter ya da bir rakam karakteri ise sıfır dışı değere, aksi halde sıfır değerine döner. Aşağıda bu işlev iki ayrı biçimde yazılıyor:

```
#include <ctype.h>

int isalnum1(int ch)
{
    return ch >= 'a' && ch <= 'z' || ch >= 'A' && ch <= 'Z'
    || ch >= '0' && ch <= '9';
}

int isalnum2(int ch)
{
    return isalpha(ch) || isdigit(ch);
}
```

isxdigit İşlevi

isxdigit, bir karakterin, onaltılık sayı sistemine ait bir basamak simge olup olmadığını sınayan standart bir işlevdir. Eğer kod numarasını aldığı karakter *0123456789ABCDEFabcdef* karakterlerinden biri ise işlev sıfırdan farklı değere, bu karakterlerden biri değil ise sıfır değerine geri döner.

```
int isxdigit (int ch)
{
    return ch >= '0' && ch <= '9' || ch >= 'A' && ch <= 'F'
    || ch >= 'a' && ch <= 'f';
}
```

Standart Karakter Dönüşüm İşlevleri

Küçük harften büyük harfe ya da büyük harften küçük harfe dönüşüm yapmak, sık gereken işlemlerdendir. Standart *toupper* ve *tolower* işlevleri bu amaçla kullanılır.

tolower İşlevi

tolower standart bir C işlevidir. Parametresine kod numarası aktarılan karakter büyük harf ise, onun küçük harf karşılığının kod numarasıyla geri döner. *tolower* işlevine büyük harf olmayan bir karakterin kod numarası aktarılırsa, işlev aynı değeri geri döndürür. Aşağıda bu işlev tanımlanıyor:

```
#include <stdio.h>

int tolower (int ch)
{
    if (ch >= 'A' && ch <= 'Z')
        return ch - 'A' + 'a';

    return ch;
}

int main()
{
    char ch;

    printf("bir karakter girin :");
    ch = getchar();
    printf("%c\n", tolower(ch));

    return 0;
}
```

toupper İşlevi

toupper, standart bir C işlevidir. Parametresine sıra numarası aktarılan karakter, eğer küçük harf ise, onun büyük harf karşılığının sıra numarasıyla geri döner. *toupper* işlevine küçük harf olmayan bir karakterin sıra numarası aktarılırsa, işlev aynı değeri geri döndürür. İşlev aşağıda tanımlanıyor:

```
int toupper(int ch)
{
    if (ch >= 'a' && ch <= 'z')
        return ch - 'a' + 'A';
    return ch;
}
```

if Deyimini kullanan Örnek Programlar

Aşağıda iki sayıdan daha büyük olanına geri dönen *get_max2* ve üç sayıdan en büyüğüne geri dönen *getmax3* işlevi yazılıyor:

```
int get_max2(int a, int b)
{
    if (a > b)
        return a;
    return b;
}
```

```
int get_max3(int a, int b, int c)
{
    int max = a;

    if (b > max)
        max = b;
    if (c > max)
        max = c;

    return max;
}
```

```
#include <stdio.h>

int main()
{
    int x, y, z;

    printf("iki sayi girin : ");
    scanf("%d%d", &x, &y);
    printf("%d ve %d sayilarindan buyugu = %d\n", x, y, get_max2(x, y));
    printf("uc sayi girin : ");
    scanf("%d%d%d", &x, &y, &z);
    printf("%d %d ve  %d sayilarindan en buyugu = %d\n",x, y, z,
    get_max3(x, y, z));

    return 0;
}
```

Aşağıdaki programda *get_hex_char* isimli bir işlev tanımlanıyor. İşlev, kendisine gönderilen *0* ile *15* arasındaki bir tamsayının onaltılık sayı sistemindeki simgesi olan karakterin sıra numarası ile geri dönüyor.

```
#include <stdio.h>

int get_hex_char(int  number)
{
    if (number >= 0 && number <= 9)
        return ('0' + number);

    if (number >= 10 && number <= 15)
        return ('A' + number - 10);

    return -1;
}

int main()
{
    int number;

    printf("0 ile 15 arasinda bir sayi girin : ");
    scanf("%d", &number);
    printf("%d = %c\n", number, get_hex_char(number));

    return 0;
}
```

Aşağıdaki programda *get_hex_val* isimli bir işlev tanımlanıyor. İşlev, kendisine kod numarası gönderilen, onaltılık sayı sisteminde bir basamak gösteren simgenin, onluk sayı

sistemindeki değerine geri dönüyor. İşleve gönderilen karakter, onaltılık sayı sistemine ilişkin bir karakter değilse, *-1* değeri döndürülüyor.

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

int get_hex_val(int ch)
{
    ch = toupper(ch);
    if (isdigit(ch))
        return ch - '0';
    if (ch >= 'A'  && ch <= 'F')
        return ch - 'A' + 10;
    return -1;
}

int main()
{
    char hex;

    printf("hex digit gösteren bir karakter girin: ");
    hex = getchar();
    printf("%c =  %d\n", hex, get_hex_val(hex));

    return 0;
}
```

Aşağıdaki programda *change_case* isimli bir işlev tanımlanıyor. *change_case* işlevi, kendisine gönderilen karakter küçük harf ise, bu karakteri büyük harfe dönüştürüyor, büyük harf ise karakteri küçük harfe dönüştürüyor. Eğer harf karakteri değilse işlev, karakterin kendi değeriyle geri dönüyor.

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

int change_case(int ch)
{
    if (isupper(ch))
        return tolower(ch);

    return toupper(ch);
}

int main()
{
    int c;

    printf("bir karakter girin : ");
    c = getchar();
    c = change_case(c);
    putchar(c);

    return 0;
}
```

Aşağıdaki C programında katsayıları klavyeden alınan ikinci dereceden bir denklem çözülüyor:

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

int main()
{
    double a, b, c;
    double delta;

    printf("denklemin katsayilarini girin\n");
    printf("a = ");
    scanf("%lf", &a);
    printf("b = ");
    scanf("%lf", &b);
    printf("c = ");
    scanf("%lf", &c);

    delta = b * b - 4 * a * c;
    if (delta < 0)
        printf("denkleminizin gercek koku yok\n");
    else if (delta == 0) {
        printf("denkleminizin tek gercek koku var\n");
        printf("kok = %lf\n", -b / (2 * a));
    }
    else {
        double kokdelta = sqrt(delta);
        printf("denkleminizin 2 gercek koku var\n");
        printf("kok 1 = %lf\n", (-b + kokdelta) / (2 * a));
        printf("kok 2 = %lf\n", (-b - kokdelta) / (2 * a));
    }

    return 0;
}
```

İŞLEV BİLDİRİMLERİ

Derleme işlemi, derleyici tarafından kaynak kod içinde yukarıdan aşağıya doğru yapılır. Derleme aşamasında derleyici, bir işlev çağrısı ile karşılaştığında, çağrılan işlevin geri dönüş değerinin türünü bilmek zorundadır. Bir işlevin geri dönüş değerinin türü, geri dönüş değerinin hangi *CPU* yazmacından (*registers*) alınacağını belirler. Programın doğru çalışması için derleme zamanında bu bilginin elde edilmesi zorunludur.
Eğer çağrılan işlevin tanımı, çağıran işlevden daha önce yer alıyorsa, derleyici derleme işlemi sırasında işlev çağrı ifadesine gelmeden önce, çağrılan işlevin geri dönüş değeri türü hakkında zaten bilgi sahibi olur. Çünkü derleme işlemi yukarıdan aşağı doğru yapılır. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

double get_val(double x, double y)
{
    return x * y / (x + y);
}

int main()
{
    double d;

    d = get_val(4.5, 7.3);
    printf("d = %lf\n", d);

    return 0;
}
```

Yukarıdaki örnekte *get_val* işlevi, kendisini çağıran *main* işlevinden daha önce tanımlanıyor. Çağrı ifadesine gelmeden önce derleyici, *get_val* işlevinin geri dönüş değeri türünü zaten bilir.
Çağrılan işlevin tanımı çağıran işlevden daha sonra yapılırsa, derleyici işlev çağrı ifadesine geldiğinde, çağrılan işlevin geri dönüş değerinin türünü bilemez. Bu sorunlu bir durumdur:

```
#include <stdio.h>

int main()
{
    double d;

    //işlevin geri dönüş değerininin türü bilinmiyor.
    d = get_val(4.5, 7.3);
    printf("d = %lf\n", d);

    return 0;
}

double get_val(double x, double y)
{
    return x * y / (x + y);
}
```

Yukarıda *get_val* işlevi, *main* işlevi içinde çağrılıyor. Fakat *get_val* işlevinin tanımı, kaynak kod içinde *main* işlevinden daha sonra yer alıyor. Derleyici, derleme zamanında *get_val* işlevinin çağrısı ile karşılaştığında, bu işlevin geri dönüş değerinin türünü bilmez.

C derleyicisi bir işlev çağrı ifadesi ile karşılaştığında, işlevin geri dönüş değeri türü hakkında henüz bilgi edinememişse, söz konusu işlevin geri dönüş değerinin *int* türden olduğunu varsayar.
Yukarıdaki örnekte derleyici, *get_val* işlevinin geri dönüş değerinin *int* türden olduğunu varsayar, buna göre kod üretir. Daha sonra derleme akışı işlevin tanımına geldiğinde, bu kez işlevin geri dönüş değerinin *double* türünden olduğu görülür. Hedef kod oluşumunu engelleyen bu çelişkili durumu derleyiciler bir hata iletisi ile bildirir.

[C++ dilinde eğer çağrılan işlev çağıran işlevden daha önce tanımlanmamışsa, işlevin geri dönüş değeri int türden kabul edilmez. Bu durumda işlev bildiriminin yapılması zorunludur. Bu bildirimin yapılmaması durumunda derleme zamanında hata oluşur.]

Çağrılan işlevi, çağıran işlevin üstünde tanımlamak her zaman mümkün değildir. Büyük bir kaynak dosyada onlarca işlev tanımlanabilir. Tanımlanan her işlevin birbirini çağırması söz konusu olabilir. Bu durumda çağrılacak işlevin çağıran işlevden önce tanımlanması çok zor olur. Kaldı ki, C dilinde iki işlev birbirini de çağırabilir. Bu, özyinelemeli (*recursive*) bir çağrı düzeneğine karşılık gelir. Bu tür bir işlev tasarımında, artık çağrılan işlevin daha önce tanımlanması mümkün olamaz.

```
double func1()
{
    /***/
    func2();
    /***/
}

double func2()
{
    /***/
    func1();
    /***/
}
```

Yukarıdaki işlev tanımlamalarından hangisi daha yukarı yerleştirilirse yerleştirilsin, yine de çelişkili bir durum söz konusu olur.
Diğer taraftan, çağrılan işlevlerin tanımları çoğu zaman aynı kaynak dosya içinde yer almaz. Bu durumda derleyicinin çağrılan işlev hakkında bilgi alması nasıl gerçekleşir?

İşlev Bildirimi Nedir

İşlev bildirimi, derleyiciye bir işlev hakkında bilgi veren bir deyimdir. Derleyici, işlev çağrısına ilişkin kodu buradan aldığı bilgiye göre üretir. Ayrıca derleyici, aldığı bu bilgiyle, bazı kontroller de yapabilir. Yaptığı kontroller sonucunda hata ya da uyarı iletileri üreterek olası yanlışlıkları engeller.

İşlev Bildirimlerinin Genel Biçimi

Bir işlev bildiriminin genel biçimi aşağıdaki gibidir:

```
[geri dönüş değeri türü] <işlev ismi> ([tür1], [tür2].....);
```

Örneğin *get_val* işlevi için bildirim aşağıdaki biçimde yapılabilir:

```
double get_val(double, double);
```

Derleyici böyle bir bildirimden aşağıdaki bilgileri elde eder:

1. *get_val* işlevin geri dönüş değeri *double* türdendir. Bu bilgiden sonra artık derleyici bu işlevin çağrılması durumunda geri dönüş değerini *int* türden varsaymaz, *double* türden bir geri dönüş değeri elde edilmesine göre bir kod üretir.

2. *get_val* işlevinin iki parametre değişkeni vardır. Bu bilgilendirmeden sonra artık derleyici bu işlevin doğru sayıda argüman ile çağrılıp çağrılmadığını sorgulama şansına sahip olur. Eğer işlevin yanlış sayıda argüman ile çağrıldığını görürse, durumu bir hata iletisi ile bildirir.

3. *get_val* işlevinin parametre değişkenleri *double* türdendir. Bu bilgiden sonra derleyici işleve başka türden argümanlar gönderilmesi durumunda, argümanlar üzerinde otomatik tür dönüşümü uygular. Bu konu *"otomatik tür dönüşümü"* isimli bölümde ele alınacak.

Aşağıda örnek bildirimler veriliyor:

```
int multiply (int, int);
double power (double, double);
void clrscr(void);
```

Tıpkı işlev tanımlamalarında olduğu gibi, işlev bildirimlerinde de işlevin geri dönüş değeri belirtilmemişse, derleyici bildirimin *int* türden bir geri dönüş değeri için yapıldığını kabul eder:

```
func(double);
```

bildirimi ile

```
int func(double);
```

bildirimi tamamen eşdeğerdir. Ancak okunabilirlik açısından *int* anahtar sözcüğünün açıkça yazılması daha iyidir.

[C++ dilinde geri dönüş değerinin türünün yazılması zorunludur.]

Eğer bildirilen işlev, geri dönüş değeri üretmiyorsa, *void* anahtar sözcüğü kullanılmalıdır:

```
void func(double);
```

İşlev bildirimleri ile, yalnızca derleyiciye bilgi verilir. Bu bir tanımlama (*definition*) işlemi değildir. Dolayısıyla yapılan bildirim sonucunda derleyici programın çalışma zamanına yönelik olarak bellekte bir yer ayırmaz.

Bildirimde Parametre Ayracının İçinin Boş Bırakılması

C'nin standartlaştırma süreci öncesinde işlev bildirimlerinde,parametre ayracının içi boş bırakılıyordu. Bildirimde, işlevin parametre değişkenlerinin türleri ve sayısı hakkında bir bilgi verilmiyordu. Bildirimin tek amacı, bir işlevin geri dönüş değerinin türü hakkında bilgi vermekti. Dolayısıyla aşağıdaki gibi bir bildirim

```
double func();
```

func işlevin parametre değişkenine sahip olmadığı anlamına gelmiyordu. Standartlaştırma süreci içinde işlev bildirimlerine yapılan eklemeyle, işlevlerin parametre değişkenlerinin sayısı ve türleri hakkında da bilgi verilmesi olanağı verildi. Ancak bu durumda da ortaya şöyle bir sorun çıktı. Eski kurallara göre yazılan bir kod yeni kurallara göre derlendiğinde

```
double func();
```

gibi bir bildirim, işlevin parametre değişkenine sahip olmadığı biçiminde yorumlanırsa, bildirilen işlevin örneğin

```
func(5)
```

biçiminde çağrılması durumunda bir sözdizim hatası ortaya çıkardı. Geçmişe doğru uyumluluğu sağlamak için şöyle bir karar alındı. Eğer işlevin parametre değişkeni yoksa bildirimde parametre ayracının içine *void* anahtar sözcüğü yazılmalıdır. Aşağıdaki bildirimleri inceleyin:

```
double foo();
double func(void);
```

Standartlara göre *foo* işlevinin bildiriminden derleyici, *foo* işlevinin parametre değişkenleri hakkında bir bilginin verilmediği sonucunu çıkarır ve işlev çağrısıyla karşılaştığında işleve gönderilen argümanların sayısına ilişkin bir kontrol yapmaz. Yani böyle bildirilen bir işlev, kurallara uygun bir şekilde istenen sayıda bir argümanla çağrılabilir.
func işlevinin bildiriminden derleyici, *func* işlevin parametre değişkenine sahip olmadığı sonucunu çıkarır ve işlev çağrısıyla karşılaştığında, işleve bir ya da daha fazla sayıda argüman gönderildiğini görürse, bu durumu derleme zamanı hatası olarak belirler.

[C++ dilinde ise her iki bildirim de eşdeğerdir. Yani işlev bildiriminde parametre ayracının içinin boş bırakılmasıyla buraya *void* anahtar sözcüğünün yazılması arasında bir fark yoktur]

Bildirimlerde Parametre Değişkenleri İçin İsim yazılması

İşlev bildirimlerinde, parametre değişkenlerinin türlerinden sonra isimleri de yazılabilir. Bildirimlerde yer alan parametre değişkenleri isimlerinin bilinirlik alanları, yalnızca bildirim parametre ayracı ile sınırlıdır. Standartlar bu durumu, ayrı bir bilinirlik alanı kuralı ile belirlemiştir. İşlev bildirim ayracı içinde kullanılan isimler, yalnızca bu ayraç içinde bilinir. Bu ayracın dışına çıkıldığında bu isimler bilinmez. Bu bilinirlik alanı kuralına "İşlev Bildirimi Bilinirlik Alanı Kuralı" (*function prototype scope*) denir.
Buraya yazılan parametre isimleri, yalnızca okunabilirlik açısından faydalıdır. Buradaki parametre isimlerinin, işlevin tanımında kullanılacak parametre isimleriyle aynı olması gibi bir zorunluluk yoktur.
Yukarıdaki bildirimleri parametre değişkenlerine isim vererek yeniden yazalım:

```
float calculate(float a, float b);
int multiply(int number1, int number2);
double pow(double base, double exp);
```

İşlevlerin tanımlarını görmeden yalnızca işlevlerin bildirimlerini okuyanlar, bildirimlerde kullanılan parametre değişkeni isimlerinden, bu değişkenlerin işlev çağrılarından hangi bilgileri bekledikleri konusunda fikir sahibi olurlar.

İşlev Bildirimlerinin Yerleri

Bir işlevin bildirimi, programın herhangi bir yerinde yapılabilir. Bildirimler global düzeyde yapılmışsa, yani tüm blokların dışında yapılmışsa, bildirildikleri yerden dosya sonuna kadar olan alan içinde geçerliliklerini sürdürür. Söz konusu işlev çağrılmadan, işlevin bildirimi yapılmış olmalıdır.
Ancak uygulamalarda çok az rastlanmasına karşılık, işlev bildirimleri yerel düzeyde de yapılabilir. Bu durumda bildirim ile, yalnızca bildirimin yapılmış olduğu bloğa bilgi verilmiş olur. Başka bir deyişle işlev bildirimi de, değişken tanımlamaları gibi, bilinirlik alanı kuralına uyar.
Genel olarak işlev bildirimleri programın en yukarısında ya da programcının tanımladığı başlık dosyalarının birinin içinde yapılır. Başlık dosyaları *(header files),* ileride ayrıntılı olarak ele alınacak.

Standart C İşlevlerinin Bildirimleri

Standart C işlevlerinin bildirimleri, standart başlık dosyaları içine yerleştirilmiştir. Programcı, uygulamalarda standart bir C işlevinin bildirimini kendi yazmaz, bu bildiriminin bulunduğu başlık dosyasını *#include* önişlemci komutuyla kendi kaynak koduna ekler.

İşlev Bildiriminin Yapılmaması

Çağrılan bir işlevin kaynak kodunun, aynı kaynak dosya içinde yer alması gibi bir zorunluluk yoktur. Derleyici, bir işlev çağrısı ile karşılaştığında, çağrılan işlevin kaynak kodunu aramaz. Ürettiği hedef dosya *(object file)* içine, bağlayıcı için, ilgili işlevin çağrıldığını belirten bir bilgi yazar. Çağrılan işlevin derlenmiş kodunu bulmak ve hedef dosyaları uygun bir biçimde birleştirmek, bağlayıcı programın görevidir.
Çağrılan bir işlevin tanımının aynı kaynak dosyada olmadığını düşünelim. Bu durumda çağrılan işlevin bildirimi yapılmamışsa ne olur? C dilinde bu durum bir sözdizim hatası değildir.

C++ dilinde bu durum doğrudan sözdizim hatasıdır. Derleyici, bir işlevin çağrılmasından önce, söz konusu işlevin tanımını ya da bildirimini mutlaka görmek zorundadır.

C dilinde derleyici, bildirimini görmediği bir işlevin geri dönüş değerini *int* türden kabul ederek hedef dosyayı üretir. Eğer kaynak kod içinde işlevin geri dönüş değeri kullanılmamışsa, ya da çağrılan işlevin geri dönüş değeri gerçekten *int* türden ise bir sorun çıkmaz. Ancak işlevin geri dönüş değeri kullanılmışsa ve geri dönüş değeri *int* türden değilse, bir çalışma zamanı hatası söz konusudur. Aşağıdaki örneği derleyicinizde derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    double d;

    d = sqrt(9.);
    printf("d = %lf\n", d);

    return 0;
}
```

Yukarıdaki programın derlenmesi sırasında bir hata oluşmaz. Ancak derleyici, çağrılan *sqrt* işlevinin geri dönüş değerinin *int* türden olduğunu varsayarak kod üretir. Oysa, derlenmiş *sqrt* işlevinin geri dönüş değeri *double* türdendir. Çalışma zamanı sırasında, işlevin geri döndürdüğü *double* türden değer yerine, derleyicinin ürettiği kod sonucunda, *int* türden bir değer çekilmeye çalışılır. Bu, programın çalışma zamanına yönelik bir hatadır.
Şimdi de *main* işlevinden önce aşağıdaki bildirimi ekleyerek programı yeniden derleyerek çalıştırın:

```
double sqrt(double val);
```

Standart C işlevlerinin bildirimleri, sonu *.h* uzantılı olan başlık (*header*) dosyaları içindedir. *#include* önişlemci komutuyla ilgili başlık dosyasının kaynak koda eklenmesiyle, aslında standart C işlevlerinin de bildirimi yapılmış olur. Zira önişlemci programın çıktısı olan kaynak program, artık derleyiciye verildiğinde, eklenmiş bu dosyada işlevin bildirimi de bulunur. Şüphesiz, ilgili başlık dosyasını kaynak koda eklemek yerine, standart C işlevlerinin bildirimleri programcı tarafından da yapılabilir. Bu durumda da bir yanlışlık söz konusu olmaz. Yukarıdaki örnekte çağrılan *sqrt* işlevinin bildirimi iki şekilde kaynak koda eklenebilir:
i. İşlev bildiriminin bulunduğu başlık dosyasının, bir önişlemci komutuyla kaynak koda eklenmesiyle:

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

ii) İşlevin bildirimi doğrudan yazılabilir:

```
double sqrt(double val);
```

Ancak tercih edilecek yöntem başlık dosyasını kaynak koda eklemek olmalıdır. Çünkü programcı tarafından işlevin bildirimi yanlış yazılabilir.
Başlık dosyalarının kaynak koda eklenmesinin nedeni yalnızca işlev bildirimi değildir. Başlık dosyalarında daha başka bildirimler de vardır: Makrolar, simgesel değişmezler, tür ismi bildirimleri, yapı bildirimleri vs.

İşlev Bildirimi İle Argüman-Parametre Uyumu Sorgulaması

İşlev bildirimlerin ana amacı, yukarıda da belirtildiği gibi, derleyiciye işlevin geri dönüş değeri türü hakkında bilgi vermektir. Ancak işlev bildirimlerinde işlev parametrelerinin türleri belirtilmişse, derleyici prototip bildirimindeki parametre değişkeni sayısını işlev çağrı ifadesindeki işleve gönderilen argüman sayısı ile karşılaştırır. Örneğin:

```
float process(float, float);
```

biçiminde bir bildirim yazıldığında eğer *process* işlevi eksik ya da fazla argüman ile çağrılırsa derleme zamanında hata oluşur.

```
x = process(5.8);              /* Geçersiz! Eksik argüman ile çağrı*/
y = process(4.6, 7.9, 8.0)     /* Geçersiz! Fazla argüman ile çağrı*/
```

Bildirim Sözdizimine İlişkin Ayrıntılar

Aynı türden geri dönüş değerine sahip işlevlerin bildirimi, virgüllerle ayrılarak yazılabilir, ama böyle bir bildirim biçimi, programcılar tarafından genel olarak pek tercih edilen bir durum değildir.

```
double func1(int), func2(int, int), func3(float);
```

Yukarıdaki bildirim geçerlidir. Böyle bir bildirimle, derleyiciye *func1*, *func2*, *func3* işlevlerinin hepsinin geri dönüş değerinin *double* türden olduğu bilgisi verilir.
İşlev bildirimleri, değişken tanımlamalarıyla da birleştirilebilir. Bu da okunabilirlik açısından tercih edilen bir durum değildir.

```
long func1(int), long func2(void), x, y;
```

Yukarıdaki bildirim deyimi ile *func1* ve *func2* işlevlerinin bildirimi yapılırken, *x* ve *y* değişkenleri tanımlanıyor.
Bir işlevin bildiriminin yapılmış olması, o işlevin tanımlamasını ya da çağrılmasını zorunlu kılmaz. Bildirimi yapılan bir işlevi tanımlamamak hata oluşturmaz.
Bir işlevin bildirimi birden fazla kez yapılabilir. Bu durum, bir derleme zamanı hatası oluşturmaz. Ama yapılan bildirimler birbirleriyle çelişmemelidir.
Kaynak dosya içinde aynı işleve ilişkin bildirimlerin farklı yerlerde, aşağıdaki biçimlerde yapıldığını düşünelim:

```
int func (int, int);
func (int, int);
int func(int x, int y);
func(int number1, int number2);
```

Yukarıdaki bildirimlerinin hiçbirinde bir çelişki söz konusu değildir. İşlev parametre değişkenlerinin isimleri için daha sonraki bildirimlerde farklı isimler kullanılması bir çelişki yaratmaz. Çünkü bu isimlerin bilinirlik alanı (*scope*), yalnızca bildirimin yapıldığı ayracın içidir. Ancak aşağıdaki farklı bildirimler geçersizdir.

```
double func(int x, double y);
double func(int x, float y);  /* Geçersiz! */
long sample(double x);
sample (double x);            /* Geçersiz! */
```

İşlev Bildirimlerinin Yerleri

C ve C++ dillerinde bir proje, çoğu zaman birden fazla kaynak dosyadan, yani modülden oluşur. Kaynak dosyaların çoğunda, başka kaynak dosyalar içinde tanımlanan işlevler çağrılır. Yani çoğu zaman kaynak dosyalar arasında bir hizmet alma verme ilişkisi vardır. Başka modüllere hizmet verecek bir modül, iki ayrı dosya şeklinde yazılır. Dosyalardan biri kodlama (*implementation*) dosyasıdır. Bu dosyanın uzantısı *.c* dir. Bu dosya tarafından diğer modüllere sunulan hizmetlere ilişkin bildirimler, *.h* uzantılı başka bir dosyaya yerleştirilir. Bu dosyaya başlık dosyası (*header file*) denir. Diğer modüllere hizmet verecek işlevlerin bildirimleri, başlık dosyası içine yerleştirilmelidir.
Hizmet alan kodlama dosyası, hizmet veren modülün başlık dosyasının içeriğini, kendi dosyasına *#include* önişlemci komutuyla ekler. Böylece bildirim yapılmış olur. Bu durum "*önişlemci komutları*" konusunda yeniden ele alınacak.

TÜR DÖNÜŞÜMLERİ

Bilgisayarların aritmetik işlemleri gerçekleştirmesinde bir takım kısıtlamalar söz konusudur. Bilgisayarların aritmetik bir işlemi gerçekleştirmesi için genellikle işleme sokulan terimlerin uzunluklarının aynı olması, yani bit sayılarının aynı olması ve bellekte aynı formatta ifade edilmeleri gerekir. Örneğin işlemci *16* bit uzunluğunda iki tam sayıyı doğrudan toplayabilir ama *16* bit uzunluğunda bir tam sayı ile *32* bit uzunluğundaki bir gerçek sayıyı doğrudan toplayamaz.
C programlama dili, değişik türlerin aynı ifade içinde bulunmalarına izin verir. Yani tek bir ifadede bir tamsayı türünden değişken, *float* türden bir değişmez ya da *char* türden bir değişken birlikte yer alabilir. Bu durumda C derleyicisi, bunları herhangi bir işleme sokmadan önce, bilgisayar donanımının ifadeyi değerlendirebilmesi için uygun tür dönüşümlerini yapar.
Örneğin *16* bitlik *int* türden bir değerle *64* bitlik *double* türden bir değer toplandığında, önce *16* bitlik *int* türden değer, *64 bit* uzunluğunda *double* türden bir değer olarak ifade edilir, daha sonra toplama işlemi gerçekleştirilir. Yine *16* bitlik bir *int* türden bir değerle *64* bitlik bir *double* türden bir değer çarpıldığında derleyici, önce *int* türden değeri *64* bitlik *double* türden bir değere dönüştürür. Bu tür dönüşümü daha karmaşıktır, çünkü *int* ve *double* türden değerler bellekte farklı biçimlerde tutulur.

Bu tür dönüşümler, programcının bir kod yazmasına gerek duyulmaksızın otomatik olarak gerçekleştirilir. Böyle dönüşümlere, otomatik tür dönüşümleri (*implicit type conversions*) diyeceğiz. Diğer taraftan C dili, programcıya herhangi bir ifadeyi, bir işleç kullanarak başka bir türden ele alma olanağı da verir. Programcı tarafından yapılan böyle tür dönüşümlerine bilinçli tür dönüşümleri (*explicit type conversions/type casts*) diyeceğiz.
Önce otomatik tür dönüşümlerini inceleyelim. Otomatik tür dönüşümleri ne yazık ki karmaşık yapıdadır ve iyi öğrenilmemesi durumunda programlarda hatalar kaçınılmazdır. C'de 11 ayrı doğal veri türü olduğunu biliyorsunuz. Herhangi bir hataya neden olmamak için bunların her türlü ikili bileşimi için nasıl bir otomatik tür dönüşümü yapılacağını çok iyi bilmek gerekir.

Hangi Durumlarda Tür Dönüşümü Yapılır

Aşağıda belirtilen dört durumda mutlaka otomatik bir tür dönüşümü yapılır:
1. Aritmetik ya da mantıksal bir ifadenin terimleri aynı türden değilse:
Böyle yapılan tür dönüşümlerine *işlem öncesi tür dönüşümleri* diyeceğiz.

2. Atama işleci kullanıldığında atama işlecinin sağ tarafındaki ifadenin türü ile sol tarafındaki ifadenin türü aynı değilse:

```
double toplam;
long sayi1, sayi2;
/***/
toplam = sayi1 + sayi2;
```

Bu durumda yapılan tür dönüşümlerine *atama tür dönüşümleri* diyeceğiz.

3. Bir işlev çağrısında işleve gönderilen bir argümanın türü ile işlevin ilgili parametre değişkeninin türü aynı değilse:

```
double sqrt (double val);

void func()
{
    int number;
    double result = sqrt(number);
    /***/
}
```

Yukarıdaki örnekte çağrılan *sqrt* işlevine gönderilen argüman olan *number* değişkeninin türü, çağrılan işlevin parametre değişkeninin türünden farklıdır. Bu durumda bir tür dönüştürme işlemi yapılır.

4. Bir *return* ifadesinin türü ile ilgili işlevin geri dönüş değerinin türü aynı değilse:

```
double func(int val)
{
    /****/
    return val;
}
```

Yukarıda tanımlanan *func* isimli işlevin içinde yer alan *return* ifadesinin türü *int* iken, işlevin bildirilen geri dönüş değeri türü *double* türüdür. Bu durumda da, derleyici tarafından bir otomatik tür dönüştürme işlemi yapılır.

3. ve 4. maddeler de bir atama işlemi olarak düşünülebilir. İşlev çağrı ifadesindeki argümanlar, parametre değişkenlerine kopyalanarak, geçirilir. Yani örtülü bir atama işlemi söz konusudur. Yine *return* ifadeleri de aslında işlevlerin geri dönüş değerlerini tutacak geçici nesnelere kopyalanırlar.

İşlem Öncesi Aritmetik Tür Dönüşümleri

İşlem öncesi otomatik tür dönüşümleri, iki terimli işleçlerin bulunduğu ifadelerde terimlerin türlerinin farklı olması durumunda uygulanır. Otomatik tür dönüşümü sonucunda, farklı iki tür olması durumu ortadan kaldırılarak terimlerin her ikisinin de türlerinin aynı olması sağlanır. Örneğin

```
int i;
double  d, result;
result = i + d;
```

Bu ifadenin sağ tarafında yer alan *i* ve *d* değişkenlerinin türleri farklıdır. Terimlerden biri *int,* diğeri *double* türdendir. Bu durumda derleyici, terimlerden birini geçici bir bölgede diğerinin türünden ifade edecek bir kod üretir. Dolayısıyla işlem, ortak olan türde yapılır. Peki *int* türünden olan terim mi *double* türünde ifade edilir, yoksa *double* türünden olan terim mi *int* türünde ifade edilir? Derleyici böyle bir dönüşümü bilgi kaybı olmayacak biçimde yapmaya çalışır.

Bu durumda bilgi kaybını engellemek için genel olarak daha küçük türden olan terim, daha büyük türde olan terimin türünde ifade edilir.
Kuralları ayrıntılı olarak öğrenmek için oluşabilecek durumları iki ana grup altında inceleyelim:
1. Terimlerden biri gerçek sayı türlerinden birine ait ise:

Terimlerden birinin *long double* türünden, diğerinin farklı bir türden olması durumunda diğer terim *long double* türünde ifade edilir ve işlem *long double* türünde yapılır.

Terimlerden birinin *double* türünden, diğerinin farklı bir türden olması durumunda diğer terim *double* türünde ifade edilir ve işlem *double* türünde yapılır.

Terimlerden birinin *float* türünden, diğerinin farklı bir türden olması durumunda diğer terim *float* türünde ifade edilir ve işlem *float* türünde yapılır.

2. Terimlerden hiçbiri gerçek sayı türlerinden değilse:

Eğer ifade içindeki terimlerden herhangi biri *signed char*, *unsigned char*, *signed short int* ya da *unsigned short int* türden ise aşağıdaki algoritma uygulanmadan önce bu türler *int*

türüne dönüştürülür. Yapılan bu dönüşüme *"tam sayıya yükseltme" (integral promotion)* denir.
Daha sonra aşağıdaki kurallar uygulanır:

Terimlerden birinin *unsigned long* türünden, diğerinin farklı bir türden olması durumunda diğer terim *unsigned long* türünde ifade edilir ve işlem *unsigned long* türünde yapılır.

Terimlerden birinin *signed long* türünden, diğerinin farklı bir türden olması durumunda diğer terim *signed long* türünde ifade edilir ve işlem *signed long* türünde yapılır.

Terimlerden birinin *unsigned int* türünden, diğerinin farklı bir türden olması durumunda diğer terim *unsigned int* türünde ifade edilir ve işlem *unsigned int* türünde yapılır.
İstisnalar:

Eğer terimlerden biri *signed long int* diğeri *unsigned int* türünden ise ve kullanılan sistemde bu türlerin uzunlukları aynı ise (*UNIX* ve Win 32 sistemlerinde olduğu gibi) her iki terim de *unsigned long int* türüne dönüştürülür.
Eğer terimlerden biri *signed int* diğeri *unsigned short int* türünden ise ve kullanılan sistemde bu türlerin uzunlukları aynı ise (DOS işletim sisteminde olduğu gibi) her iki terim de *unsigned int* türüne dönüştürülür.

İşaretli bir tamsayı türünden işaretsiz tamsayı türüne dönüşüm yapılması durumunda dikkatli olunmalıdır:

```
#include <stdio.h>

int main()
{
    int x = -2;
    unsigned int y = 1;

    if (y > x)
        printf("dogru!\n");
    else
        printf("yanlis!\n");

    return 0;
}
```

Yukarıdaki programın çalışmasıyla ekrana "yanlış" yazısı yazdırılır.

```
y > x
```

ifadesinde `'>'` işlecinin sol terimi *unsigned int* türünden iken sağ terimi *int* türdendir. Bu durumda yapılacak otomatik tür dönüştürme işlemi sonucunda *int* türden olan terim, işlem öncesinde *unsigned int* türünde ifade edilir. *-2* değeri *unsigned int* türünde ifade edildiğinde artık negatif bir değer olmayıp büyük bir pozitif sayı olur. Örneğin *2 byte*'lık *int* türü söz konusu olduğunda bu değer *65534*'tür. Dolayısıyla *y > x* ifadesi yanlış olarak yorumlanır. Çünkü otomatik tür dönüşümünden sonra *y > x* ifadesi artık yanlıştır.

İşlev çağrı ifadeleri de, işleçlerle birlikte başka ifadeleri oluşturuyorsa, otomatik tür dönüşümlerine neden olabilir. Zira geri dönüş değerine sahip olan işlevler için işleve yapılan çağrı ifadesi, işlevin geri dönüş değerine karşılık gelir. Örneğin:

```
int i = 5;
...
pow(2, 3) + i
```

ifadesinde *pow* işlevinin geri dönüş değeri *double* türden olduğu için, *int* türden olan *i* değişkeni de, işlemin yapılabilmesi için geçici bir bölgede *double* türünde ifade edilerek işleme sokulur.

Atama Tür Dönüşümleri

Bu tür dönüşümlerin çok basit bir kuralı vardır: Atama öncesinde, atama işlecinin sağ tarafındaki ifade, atama işlecinin sol tarafındaki nesnenin türünde ifade edilir:

Küçük türlerin büyük türlere dönüştürülmesinde bilgi kaybı söz konusu değildir. Örneğin:

```
double leftx;
int righty = 5;

leftx = righty;
```

Yukarıdaki örnekte *righty* değişkeninin türü *int*'tir. Önce *double* türe otomatik dönüşüm yapılır, daha sonra *double* türünde ifade edilen *righty* değişkeninin değeri *leftx* değişkenine atanır.

Aşağıda *16* bitlik sistemler için bazı örnekler veriliyor:

TÜR	desimal	hex	dönüştürülecek tür	hex	desimal
int	138	0x008A	long int	0x0000008A	138L
char	'd' (100)	0x64	int	0x0064	100
int	-56	0xFFC8	long int	0xFFFFFFC8	-56L
char	'\x95' (-07)	0x95	int	0xFF95	-107
unsigned int	45678	0xB26E	long int	0X0000B26EL	45678
char	'0' (48)	0x30	long int	0x00000030L	30L

Negatif olan bir tamsayı, küçük türden büyük türe dönüştürüldüğünde sayının yüksek anlamlı bitleri, negatifliğin korunması amacıyla *1* biti ile beslenir.
Derleyici tarafından yapılan atama tür dönüşümlerinde, atama öncesi, büyük türün küçük türe dönüştürülmesi durumunda bilgi kaybı söz konusu olabilir.

Aşağıdaki basit kurallar verilebilir:
Eğer atama işlecinin her iki tarafı da tam sayı türlerinden ise (*char, short, int, long*), atama işlecinin sağ tarafının daha büyük bir türden olması durumunda bilgi kaybı olabilir. Bilgi kaybı ancak, atama işlecinin sağ tarafındaki değerin, sol taraftaki türün sınırları içinde olmaması durumunda söz konusu olur. Bilgi kaybı, yüksek anlamlı *byte*'ların kaybolması şeklinde ortaya çıkar. Örnek:

```
long m = 0x12345678;
int y = m;
printf ("m = %x\n", m);
```

Yukarıdaki örnekte *int* türden olan *y* değişkenine *long* türden bir değişkenin değeri atanmıştır. Kodun 16 bitlik bir sistemde, örneğin DOS altında çalıştığını düşünüyoruz. DOS altında *int* türü için sayı sınırları *–32768 +32767* değerleridir. Bu sayılar da, iki *byte*'lık bir alan için işaretli olarak yazılabilecek en büyük ve en küçük değerlerdir. Onaltılık sayı sisteminde her bir basamak *4* bite ve her iki basamak 1 *byte* alana karşılık gelir. Dolayısıyla *0x12345678* sayısı 8 hex basamak, yani *4 byte* uzunluğunda bir sayıdır. Oysa atamanın yapılacağı nesne *int* türdendir ve bu tür max. 4 hex basamak *(2 byte)* uzunlukta olabilir. Bu durumda *m* değişkenine ilişkin değerin yüksek anlamlı *2 byte*'ı yani (4 hex basmağı) yitirilir. Atama işleminden sonra, *printf* işleviyle, *y* değişkeninin değeri *5678* olarak yazdırılır.

Atama işlecinin sağ terimi, bir gerçek sayı türünden ise(*float*, *double*, *long double*) ve sol terimi tam sayı türünden ise önce gerçek sayı değerinin ondalık kısmı kaybedilir. Eğer gerçek sayıdan elde edilen tamsayı kısmı, atamanın yapıldığı tamsayı türünden ifade edilemiyorsa, bu durum tanımsız davranıştır *(undefined behaviour*). Bu durumun oluştuğu kodlardan kesinlikle kaçınmak gerekir. Ama derleyicilerin hemen hepsi, bu durumda aşağıdaki şekilde tür dönüşümü yapar:

Atama işlecinin sağ terimi olan gerçek sayı bir ondalık kısım içeriyorsa, önce ondalık kısmı kaybedilir. Ondalık kısmı kaybedildikten sonra kalan tamsayı değer, eğer sol terimin türünün sınırları içinde kalıyorsa daha fazla bir bilgi kaybı olmaz, fakat sol taraf türünün sınırları aşılıyorsa fazladan bir bilgi kaybı daha olur ve bu kez yüksek anlamlı *byte*'lar kaybedilir. Örnek:

```
#include <stdio.h>

int main()
{
    double y = 234.12;
    int x;

    x = y;
    printf("x = %d\n", x); /* x değişkenine 234 değeri atanır*/

    y = 7689523345.347;
    x = y;      /* Yanlış */
    printf("x = %d\n", x);

    return 0;
}
```

Şimdi de aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    char ch;

    ch = 135;

    if (ch == 135)
        printf("dogru!\n");
    else
        printf("yanlış!\n");

    return 0;
}
```

Program çalıştırıldığında ekrana neden "yanlış" yazısı yazdırılıyor?
ch değişkenine *135* değeri atanıyor:

```
ch = 135;
```

Bu durumda yüksek anlamlı *byte* kaybedileceğinden *ch* değişkenine atanan değer

```
1000 0111
```

değeri olur.

```
ch == 135
```

karşılaştırma işleminde *char* türden olan *ch* değişkeni, karşılaştırma işlemi öncesi *signed int* türüne yükseltilir. İşlem öncesi tamsayıya yükseltme sonucu, yüksek anlamı *byte*'ler *1* bitleriyle beslenir. Çünkü *ch* negatif bir değere sahiptir. Karşılaştırma işlemi öncesinde *ch*'nin değeri

```
1111 1111 1000 0111
```

olur. Oysa karşılaştırma işlecinin sağ terimi olan *135* değeri, *int* türden bir değişmezdir. Yani aslında karşılaştırılan değerler aşağıdaki gibi olur:

```
1111 1111 1000 0111
0000 0000 1000 0111
```

Karşılaştırma yanlış olarak sonuçlanır.

Tamsayıya Yükseltme

Daha önce de açıklandığı gibi tamsayıya yükseltme (*integral promotion*), bir ifade içinde bulunan *char, unsigned char, short, unsigned short* türlerinin, ifadenin derleyici tarafından değerlendirilmesinden önce, otomatik olarak *int* türüne dönüştürülmeleri anlamına gelir.

Peki dönüşüm, *signed int* türüne mi, *unsigned int* türüne mi yapılır?
Genel kural şudur: Tür dönüşümüne uğrayacak terimin değeri *int* türünde ifade edilebiliyorsa *int*, edilemiyorsa *unsigned int* türüne dönüşüm yapılır.

Örneğin *unsigned short* ve *int* türlerinin aynı uzunlukta olduğu *DOS* işletim sisteminde *unsigned short* türü, tamsayıya yükseltilirken *unsigned int* türüne dönüştürülür.

Eğer tam sayıya yükseltilecek değer, *signed char, unsigned char* ya da *signed short* türlerinden ise, dönüşüm *signed int* türüne yapılır.
Bilgi kaybı ile ilgili şu hususu da göz ardı etmemeliyiz. Bazı durumlarda bilgi kaybı tür dönüşümü yapıldığı için değil yapılmadığı için oluşur. Sınır değer taşmaları, bu duruma iyi bir örnek olabilir.
Örnek: (DOS altında çalıştığımızı düşünelim)

```
long x = 1000 * 2000;
```

Yukarıdaki kod ilk bakışta normal gibi görünüyor. Zira çarpma işleminin sonucu olan *2000000* değeri *DOS* altında *signed long* türü sayı sınırları içinde kalır. Oysa bilgi kaybı atama işleminden önce gerçekleşir. *1000* ve *2000 int* türden değişmezlerdir, işleme sokulduklarında çarpma işlecinin de ürettiği değer int türden olur. Bu durumda *2 byte* uzunlukta olan *int* türü, *2000000* değerini tutamayacağı için yüksek anlamlı *byte* kaybedilir. *2000000* onaltılık sayı sisteminde *0x1E8480* olarak gösterilebilir. Yüksek anlamlı *byte* kaybedilince işlem sonucu, *0x8480* olur. *0x8480* negatif bir sayıdır. Çünkü işaret biti *1*'dir. İkiye tümleyenini alırsak

```
0x8480             1000 0100 1000 0000
ikiye tümleyeni    0111 1011 1000 0000    (0x7B80 = 31616)
```

Görüldüğü gibi işlem sonucu üretilecek değer *–31616* dir. Bu durumda *x* değişkeninin türü *long* türü de olsa, atanacak değer *–31616* olur.

İşlev Çağrılarında Tür Dönüşümü

Daha önce söylendiği gibi, bir işleve gönderilen argümanlarla, bu argümanları tutacak işlevin parametre değişkenleri arasında tür farkı varsa otomatik tür dönüşümü gerçekleşir

ve argümanların türü, parametre değişkenlerinin türlerine dönüştürülür. Ancak bu tür dönüşümünün gerçekleşmesi için, derleyicinin işlev çağrı ifadesine gelmeden önce işlevin parametre değişkenlerinin türleri hakkında bilgi sahibi olması gerekir. Derleyici bu bilgiyi iki ayrı şekilde elde edebilir:
Çağrılan işlev çağıran işlevden daha önce tanımlanmışsa derleyici, işlevin tanımlamasından parametre değişkenlerinin türünü belirler.
İşlevin bildirimi yapılmışsa derleyici, parametre değişkenlerinin türü hakkında önceden bilgi sahibi olur. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

double func(double x, double y)
{
    /***/
}

int main()
{
    int a, b;
    /***/
    func(a, b);

    return 0;
}
```

Yukarıdaki örnekte *main* işlevi içinde çağrılan *func* işlevine argüman olarak, *int* türden olan *a* ve *b* değişkenlerinin değerleri gönderiliyor. İşlev tanımı çağrı ifadesinden önce yer aldığı için *int* türden olan *a* ve *b* değişkenlerinin değerleri, *double* türüne dönüştürülerek *func* işlevinin parametre değişkenleri olan *x* ve *y* değişkenlerine aktarılır. *func* işlevinin *main* işlevinden sonra tanımlanması durumunda, otomatik tür dönüşümünün yapılabilmesi için, işlev bildirimi ile derleyiciye parametre değişkenlerinin türleri hakkında bilgi verilmesi gerekir.

```
#include <stdio.h>

double func(double x, double y);

int main()
{
    int a, b;
    /***/
    func(a, b);

    return 0;
}

double func(double x, double y)
{
    /***/
}
```

Peki çağrılan işlev çağıran işlevden daha sonra tanımlanmışsa ve işlev bildirimi yapılmamışsa -tabi bu durumda derleme zamanında hata oluşmaması için işlevin *int* türden bir geri dönüş değerine sahip olması gerekir- tür dönüşümü gerçekleşebilecek mi? Bu durumda derleyici, işlevin parametre değişkenlerinin türleri hakkında bilgi sahibi olamayacağı için, işleve gönderilen argümanlara, varsayılan argüman dönüşümü denilen işlemi uygular. Varsayılan argüman dönüşümü şu şekilde olur:

char ya da *short* türünden olan argümanlar tamsayıya yükseltilir *(integral promotion). float* türünden olan argümanlar *double* türüne dönüştürülür. Bunun dışındaki türlerden olan argümanlar için tür dönüşümü yapılmaz.

Tür Dönüştürme İşleci

Tür dönüştürme işleci (*typecast operator)* ile bir ifade bir işleme sokulmadan önce başka bir türden ifade edilebilir. Tür dönüştürme işleci, önek konumunda bulunan tek terimli bir işleçtir.

İşleç, bir ayraç ve ayraç içine yazılan bir tür bilgisinden oluşur:

```
(double)x
```

İşlecin ürettiği değer, terimi olan ifadenin ayraç içindeki türden ifade edilmiş değeridir. Tür dönüştürme işleci de, diğer tüm tek terimli işleçler gibi, işleç öncelik tablosunun ikinci öncelik seviyesinde bulunur.
Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    int x = 10;
    int y = 4;
    double z;

    z = (double)x / y;
    printf("z = %lf\n", z);

    return 0;
}
```

Yukarıdaki programda

```
z = (double)x / y
```

ifadesinde önce tür dönüştürme işleci değer üretir. Tür dönüştürme işlecinin ürettiği değer, *x* nesnesinin *double* türde ifade edilmiş değeridir. Bu durumda bölme işlecine sıra geldiğinde bölme işlecinin terimi *double* türden *10* değeri olur. Bu kez de otomatik tür dönüşümü ile bölme işlecinin sağ terimi *double* türüne dönüştürülerek bölme işlemi *double* türünde yapılır. Bu durumda bölme işleci *2.5* değerini üretir.
Şüphesiz ifade aşağıdaki biçimde yazılsaydı yine bilgi kaybı oluşmazdı:

```
z = x /(double)y
```

Ancak ifade aşağıdaki gibi yazılsaydı:

```
z = (double) (x / y)
```

bu durumda tür dönüştürme işlecinin terimi *(x / y)* ifadesi olurdu. Bu da bilgi kaybını engellemezdi.
Bir verinin istenerek kaybedilmesi durumunda okunabilirlik açısından, otomatik tür dönüşümü yerine, tür dönüştürme işleci ile bilinçli bir dönüşüm yapılmalıdır.

```
int i;
double d;
/***/
i = d;
```

double türden olan *d* değişkeninin değerinin *int* türden *i* değişkenine atanması güvenilir bir davranış göstermez. Atama sonunda, en iyi olasılıkla *i* değişkenine *d* nin değerinin yalnızca tam sayı kısmı atanır. Böyle bir kodu okuyanlar bu atamanın yanlışlıkla yapıldığı izlenimini edinirler. Derleyicilerin çoğu da olası bilgi kaybını uyarı iletisiyle bildirir. Bu atamanın bilinçli bir şekilde yapılması durumunda tür dönüştürme işleci kullanılmalıdır:

```
int i;
double d;
/***/
i = (int)d;
```

Aşağıdaki programda, klavyeden girilen bir gerçek sayı tam sayıya yuvarlanıyor. Girilen değerin ondalık kısmı *.5'te*n daha büyükse sayı yukarıya, *.5*'ten daha küçükse sayı aşağıya yuvarlanıyor:

```
#include <stdio.h>

int main()
{
    double d;
    int x;

    printf("bir gercek sayi girin : ");
    scanf("%lf", &d);

    if (d > 0)
        x = d + .5;
    else
        x = d - .5;

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

    return 0;
}
```

DÖNGÜ DEYİMLERİ

Bir program parçasının yinelemeli olarak çalıştırılmasını sağlayan kontrol deyimlerine "döngü deyimi" (*loop statement)* denir. C dilinde 3 ayrı döngü deyimi vardır:

while döngü deyimi
do while döngü deyimi
for döngü deyimi

Bunlardan en fazla kullanılanı, *for* döngü deyimidir. *for* döngü deyimi, yalnızca C dilinin değil, tüm programlama dillerinin en güçlü döngü yapısıdır. *while* ya da *do while* döngü deyimleri olmasa da, bu döngüler kullanılarak yazılan kodlar, *for* döngüsüyle yazılabilir. Ancak okunabilirlik açısından *while* ve *do while* döngülerinin tercih edildiği durumlar vardır.

while Döngü Deyimi

while döngü deyiminin genel sözdizimi aşağıdaki gibidir:

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

while anahtar sözcüğünü izleyen ayraç içindeki ifadeye *kontrol ifadesi (control expression)* denir. *while* ayracını izleyen ilk deyime *döngü gövdesi (loop body)* denir. Döngü gövdesini bir basit deyim, boş deyim, bileşik deyim ya da bir kontrol deyimi oluşturabilir.
while döngü deyiminin yürütülmesi söyle olur: Önce kontrol ifadesinin sayısal değeri hesaplanır. Bu ifade mantıksal olarak değerlendirilir. İfade *0* değerine sahipse, yanlış olarak yorumlanır. Bu durumda döngü gövdesindeki deyim yürütülmez, programın akışı döngü deyimini izleyen ilk deyimle sürer. Kontrol ifadesi sıfırdan farklı bir değere sahipse, doğru olarak yorumlanır bu durumda döngü gövdesindeki deyim yürütülür.
Döngü gövdesindeki deyimin yürütülmesinden sonra kontrol ifadesinin değeri yeniden hesaplanır. Kontrol ifadesi sıfırdan farklı bir değere sahip olduğu sürece döngü gövdesindeki deyim yürütülür. Döngüden kontrol ifadesinin sıfır değerine sahip olmasıyla, yani ifadenin yanlış olarak yorumlanmasıyla çıkılır.
C dilinin *while* döngüsü, bir koşul doğru olduğu sürece bir ya da birden fazla işin yaptırılmasını sağlayan bir döngü deyimidir.
Aşağıdaki örneği inceleyelin:

```
#include <stdio.h>

int main()
{
    int i = 0;

    while (i < 100) {
        printf("%d ", i);
        ++i;
    }
    return 0;
}
```

main işlevinde yer alan *while* döngüsünü inceleyelim. Döngü gövdesini bir bileşik deyim oluşturuyor. *i < 100* ifadesi doğru olduğu sürece bu bileşik deyim yürütülür. Yani *printf* işlevi çağrılır, daha sonra *i* değişkeninin değeri *1* artırılır. *i* değişkeninin değeri *100* olduğunda, kontrol ifadesi yanlış olacağından döngüden çıkılır. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

int main ()
{
    char ch = 'A';

    while (ch <= 'Z')
        putchar(ch++);

    return 0;
}
```

Yukarıdaki *main* işleviyle İngiliz alfabesinin tüm büyük harf karakterleri sırayla ekrana yazdırılıyor. *ch* isimli değişkene önce *'A'* değeri atandığını görüyorsunuz. Döngü *ch <= 'Z'* ifadesi doğru olduğu sürece döner. Döngü gövdesini bu kez bir basit deyim oluşturuyor. Sonek konumundaki ++ işlecinin nesnenin kendi değerini ürettiğini biliyorsunuz. Ancak işlecin yan etkisi nedeniyle *ch* değişkeninin değeri *1* artırılıyor. *ch* değişkeninin değeri *'Z'* olduğunda kontrol ifadesi halen doğrudur. Ancak döngünün bir sonraki turunda kontrol ifadesi yanlış olduğundan döngüden çıkılır.

while döngü deyiminde döngü gövdesindeki deyimin en az bir kez yapılması güvence altında değildir. Önce kontrol ifadesi ele alındığından, döngüye ilk girişte kontrol ifadesinin yanlış olması durumunda, döngü gövdesindeki deyim hiç yürütülmez.

Kontrol İfadeleri

Herhangi bir ifade *while* döngüsünün kontrol ifadesi olabilir.
Kontrol ifadesi bir işlev çağrısı içerebilir:

```
while (isupper(ch)) {
    /***/
}
```

Yukarıdaki *while* döngüsü, *isupper* işlevi *sıfır* dışı bir değere geri döndüğü sürece, yani *ch* büyük harf karakteri olduğu sürece döner.
Aşağıdaki *while* döngüsü ise, *isupper* işlevi *0* değerine geri döndüğü sürece, yani *ch* büyük harf karakteri olmadığı sürece döner.

```
while (!isupper(ch)) {
    /***/
}
```

while döngü deyiminin kontrol ifadesinde virgül işleci de kullanılabilir. Aşağıdaki örneği inceleyin:

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

int main()
{
    char ch;

    while (ch = getch(), toupper(ch) != 'Q')
        putchar(ch);

    return 0;
}
```

Virgül işlecinin sol terimi olan ifadenin daha önce yapılmasının güvence altında olduğunu biliyorsunuz. Yukarıdaki *main* işlevinde yer alan *while* döngüsünün kontrol ifadesine bakalım:

```
while (ch = getch(), toupper(ch) != 'Q')
```

Önce virgül işlecinin sol terimi olan ifade yapılacağına göre, *getch* işlevi çağrılır. Klavyeden alınan karakterin kod numarası *ch* değişkenine atanır. Daha sonra *toupper* işlevinin çağrılmasıyla, *ch* değişkeni değerinin *'Q'* karakteri olup olmadığı sınanır. Virgül işlecinin ürettiği değerin, sağ teriminin değeri olduğunu anımsayın. Bu durumda döngünün sürdürülmesi hakkında söz sahibi olan ifade

```
toupper(ch) != 'Q'
```

ifadesidir. Yani döngü *ch 'Q'* veya *'q'* dışında büyük harf karakteri olduğu sürece döner.

Kontrol ifadesini bir değişken de oluşturabilir:

```
while (x) {
    /***/
}
```

Yukarıdaki döngü, *x* değişkeni sıfırdan farklı bir değere sahip olduğu sürece döner.

Kontrol ifadesi bir değişmez de olabilir:

```
while (1) {
    /***/
}
```

Yukarıdaki *while* deyiminde kontrol ifadesi olarak *1* değişmezi kullanılıyor. *1* sıfır dışı bir değer olduğundan, yani kontrol ifadesi bir değişkene bağlı olarak değişmediğinden, böyle bir döngüden koşul ifadesinin yanlış olmasıyla çıkılamaz. Bu tür döngülere *sonsuz döngü* (*infinite loops*) denir. Sonsuz döngüler programcının bir hatası sonucu oluşabildiği gibi, bilinçli olarak, yani belirli bir amacı gerçekleştirmek için de oluşturulabilir. *while* ayracı içine *1* değişmez değerinin olduğu *while* döngüsü, bilinçli olarak oluşturulmuş bir sonsuz döngü deyimidir.
Atama işlecinin kontrol ifadesi içinde kullanılması da, sık rastlanan bir durumdur:

```
while ((val = get_value()) > 0) {
    foo(val);
    /***/
}
```

Yukarıdaki *while* döngüsünde *get_value* işlevinin geri dönüş değeri, *val* isimli değişkene atanıyor. Atama işleci ile oluşturulan ifade öncelik ayracı içine alındığını görüyorsunuz. Atama işlecinin ürettiği değer, nesneye atanan değer olduğundan, büyüktür işlecinin sol terimi yine *get_value* işlevinin geri dönüş değeridir. Bu durumda döngü *get_value* işlevinin geri dönüş değeri, *0*'dan büyük olduğu sürece döner. Döngü gövdesi içinde çağrılan *foo* işlevine *val* değerinin argüman olarak gönderildiğini görüyorsunuz. *foo* işlevi, *get_value* işlevinin geri dönüş değeri ile çağrılmış olur.

break Deyimi

break anahtar sözcüğünü doğrudan sonlandırıcı atom izler:

```
break;
```

Bu biçimde oluşturulan deyime "*break deyimi" (break statement)* denir. *break* deyimi bir döngü deyiminin ya da *switch* deyiminin gövdesinde kullanılabilir. Bir döngü deyiminin yürütülmesi sırasında *break* deyimi ile karşılaşıldığında, döngüden çıkılır, programın akışı döngü gövdesi dışındaki ilk deyim ile sürer. Yani koşulsuz olarak döngüden çıkılır. Aşağıdaki programı inceleyin:

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

int main ()
{
    int val;

    while (1) {
        printf("bir sayi girin : ");
        scanf("%d", &val);
        if (val < 0)
            break;
        printf("karekok %d = %lf\n", val, sqrt(val));

    }
    printf("donguden çıkıldı program sonlanıyor!\n");

    return 0;
}
```

Programda bilinçli olarak bir sonsuz döngü oluşturuluyor. Döngünün her turunda *val* isimli değişkene klavyeden bir değer alınıyor. Eğer klavyeden *0*'dan küçük bir değer girilirse, *break* deyimi ile döngüden çıkılıyor.
break deyimi yalnızca bir döngü deyiminin ya da *switch* deyiminin gövdesinde kullanılabilir. Aşağıdaki kod parçası geçersizdir:

```
if (x > 100) {
    if (y < 200)
        break;
    /***/
}
```

continue Deyimi

continue anahtar sözcüğünü de doğrudan sonlandırıcı atom izler:

```
continue;
```

Bu şekilde oluşturulan deyime "continue deyimi" (*continue statement*) denir.
Programın akışı bir döngü deyimi içinde *continue* deyimine geldiğinde, sanki döngünün turu bitmiş gibi döngünün bir sonraki turuna geçilir.

```
int getval(void);
int isprime(void);

while (1) {
    val = getval();
    if (val < 0)
        break;
    /* deyimler */
    if (isprime(val))
        continue;
    /* deyimler */
}
```

Yukarıdaki *main* işlevinde bir sonsuz döngü oluşturuluyor. Döngünün her turunda *getval* isimli işlevin geri dönüş değeri *val* değişkeninde saklanıyor. Eğer *val* değişkenine atanan değer *0* ise *break* deyimiyle döngüden çıkılıyor. Daha sonra yer alan *if* deyimi ile *val* değerinin asal olup olmadığı sınanıyor. *val*'e atanan değer asal ise, döngünün kalan kısmı yürütülmüyor, *continue* deyimiyle döngünün bir sonraki turuna geçiliyor.

continue deyimi, özellikle döngü içinde uzun *if* deyimleri olduğunda, okunabilirliği artırmak amacıyla kullanılır.

```
while (k++ < 100) {
    ch = getch();
    if (!isspace(ch)) {
        /* deyimler */
    }
}
```

Yukarıdaki yazılan *while* döngüsü içinde, klavyeden *getch* işlevi ile *ch* değişkenine bir karakterin kod numarası alınıyor. Klavyeden alınan karakter bir boşluk karakteri değilse deyimlerin yürütülmesi isteniyor. Yukarıdaki kod parçasının okunabilirliği, *continue* deyiminin kullanılmasıyla artırılabilir:

```
while (k++ < 100) {
    ch = getch();
    if (isspace(ch))
        continue;
    /* deyimler */
}
```

Bazı programcılar da *continue* deyimini döngü gövdesinde yer alacak bir boş deyime seçenek olarak kullanırlar:

```
while (i++ < 100)
    continue;
```

continue deyimi yalnızca bir döngü deyiminin gövdesinde kullanılabilir. *continue* deyiminin, döngü dışında bir yerde kullanılması geçerli değildir.

Sık Yapılan Hatalar

while döngü deyiminin gövdesinin yanlışlıkla boş deyim yapılması sık yapılan bir hatadır:

```
#include <stdio.h>

int main()
{
    int i = 10;

    while (--i > 0);          /* burada bir boş deyim var */
        printf("%d\n", i);
    return 0;
}
```

Yukarıdaki döngü *while* ayracı içindeki ifadenin değeri *0* olana kadar döner. *printf* çağrısı döngü deyiminin gövdesinde değildir. *while* ayracını izleyen sonlandırıcı atom, döngünün gövdesini oluşturan deyim olarak ele alınır. Döngüden çıkıldığında ekrana *0* değeri yazılır. Eğer bir yanlışlık sonucu değil de, bilinçli olarak *while* döngüsünün gövdesinde boş deyim (*null statement*) bulunması isteniyorsa, okunabilirlik açısından bu boş deyim, *while* ayracından hemen sonra değil, alt satırda bir tab içeriden yazılmalıdır.

while döngü deyimiyle ilgili yapılan bir başka tipik hata da, döngü gövdesini bloklamayı unutmaktır. Yani döngü gövdesindeki deyimin bir bileşik deyim olması gerekirken, yanlışlıkla bir yalın deyim kullanılır:

```
#include <stdio.h>

int main()
{
    int i = 1;

    while (i <= 100)
        printf("%d ", i);
        i++;
    return 0;
}
```

1'den *100*'e kadar olan sayıların, aralarında birer boşlukla ekrana yazdırılmak istendiğini düşünelim. Yukarıdaki *while* deyiminde *i*++ deyimi, yani döngü değişkeninin artırılması döngünün gövdesine ait değildir. Bu durumda *i* <= *100* ifadesi hep doğru olacağından sonsuz döngü oluşur ve ekrana sürekli olarak *1* değeri yazılır.

if deyiminde olduğu gibi *while* ayracı içinde de karşılaştırma işleci olan "==" yerine yanlışlıkla atama işleci "=" kullanılması, yine sık yapılan hatadır:

```
while (x == 5) {
    /***/
}
```

gibi bir döngü, *x* değişkeninin değeri *5* olduğu sürece dönerken aşağıdaki deyim, bir sonsuz döngü oluşturur:

```
while (x = 5) {
    /***/
}
```

Döngünün her turunda *x* değişkenine *5* değeri atanır. Atama işlecinin ürettiği değer olan *5, "doğru"* olarak yorumlanacağından, döngü sürekli döner.

Kontrol İfadesinde Sonek Konumundaki ++ ya da -- İşlecinin Kullanılması

Kontrol ifadesi içinde sonek konumundaki ++ ya da -- işleci sık kullanılır. Böyle bir durumda, önce ifadenin değerine bakılarak döngünün sürdürülüp sürdürülmeyeceği kararı verilir, sonra artırma ya da eksiltme işlecinin yan etkisi kendisini gösterir. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

int main()
{
    int i = 0;

    while (i++ < 100)
        ;
    printf("\n%d\n", i);   /* ekrana 101 değerini basar. */

    return 0;
}
```

n Kez Dönen while Döngüsü

n bir pozitif tamsayı olmak üzere, *n* defa dönen bir *while* döngüsü oluşturmak için

```
while (n-- > 0)
```

ya da

```
while (n--)
```

kod kalıpları kullanılabilir. Aşağıda, bir tamsayının belirli bir üssünü hesaplayan *power* isimli bir işlev yazılıyor. İşlevi inceleyin:

```
int power(int base, int exp)
{
    int result = 1;

    while (exp--)
        result *= base;

    return result;
}
```

İşlev içinde yazılan *while* döngüsü, *exp* değişkeninin değeri kadar döner, değil mi? Bu durumda *base* değişkeni, *exp* kez kendisiyle çarpılmış olur.

Döngü Gövdesinin Boş Deyim Olması

Bazen döngü gövdesi bilinçli bir şekilde boş deyim yapılır. Okunabilirlik açısından bu durumda boş deyimin normal bir deyim gibi tablama kuralına uygun olarak yazılması tavsiye edilir. Aşağıdaki programı inceleyin:

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

int main()
{
    int ch;

    printf("Evet mi Hayir mi? [e] [h] : ");

    while ((ch = toupper(getch())) != 'E' && ch != 'H')
        ;
    if (ch == 'E')
        printf("evet dediniz!\n");
    else
        printf("hayır dediniz!\n");

    return 0;
}
```

Yukarıdaki *main* işlevi içinde yazılan *while* döngüsü ile, kullanıcı klavyeden *'e', 'E', 'h', 'H'* harflerinden birini girmeye zorlanıyor. Döngüyü dikkatli bir şekilde inceleyin. Döngünün kontrol ifadesi içinde "mantıksal ve" işleci *"&&"* kullanılıyor. "Mantıksal ve" işlecinin sol teriminin daha önce yapılmasının güvence altında olduğunu anımsayın. Standart olmayan *getch* işlevi ile, klavyeden bir karakter alınıyor. Alınan karakterin sıra numarası, yani *getch* işlevinin geri dönüş değeri, standart *toupper* işlevine argüman olarak gönderiliyor. Böylece eğer klavyeden küçük harf karakteri girilmişse büyük harfe dönüştürülmüş olur. *toupper* işlevinin geri dönüş değeri *ch* değişkenine atanıyor. Ayraç

içinde yer alan ifadenin değeri, *ch* değişkenine atanan değerdir. *"Mantıksal ve"* işlecinin sağ tarafındaki ifadenin bütünü ile atama işlecinin ürettiği değerin de *'E'* karakterine eşitsizliği sorgulanıyor. *ch* değişkenine atanan değer *'E'* ise "mantıksal ve" işlecinin ikinci kısmına hiç bakılmaz, kontrol ifadesinin değeri yanlış olarak yorumlanır. Böylece döngüden çıkılır. *ch* değişkenine atanan değerin *'H'* olması durumunda, *&&* işlecinin sağ terimi değerlendirilir yani *ch* değişkeninin değerinin *'H'* karakterine eşitsizliği sorgulanır. Eğer *ch 'H'* değerine eşit ise kontrol ifadesi yine yanlış olarak yorumlanır, döngüden çıkılır. Bunun dışındaki tüm durumlarda, kontrol ifadesi doğru olarak yorumlanacağından döngünün dönmesi sürer. Bir başka deyişle, döngüden çıkılması ancak klavyeden *'e', 'E', 'h', 'H'* karakterlerinden birinin girilmesi ile mümkün olur.

while Döngü Deyiminin Kullanıldığı Örnekler

Aşağıda, bir tamsayının kaç basamaklı olduğu bilgisiyle geri dönen *num_digit* isimli bir işlevin tanımı yer alıyor. Programı derleyerek çalıştırın:

```
#include <stdio.h>

int num_digit(int val)
{
    int digit_counter = 0;

    if (val == 0)
        return 1;

    while (val != 0) {
        digit_counter++;
        val /= 10;
    }
    return digit_counter;
}

int main()
{
    int x;

    printf("bir tamsayi girin :");
    scanf("%d", &x);
    printf("%d sayisi %d basamaklı!\n", x, num_digit(x));

    return 0;
}
```

Basamak sayısını hesaplamak için çok basit bir algoritma kullanılıyor. Sayı, *sıfır* elde edilinceye kadar sürekli *10*'a bölünüyor. *num_digit* işlevinde, önce parametre değişkeni olan *val* in değerinin *0* olup olmadığı sınanıyor. Eğer *val 0* değerine eşit ise, *1* değeri ile geri dönülüyor. *0* sayısı da *1* basamaklıdır değil mi? Daha sonra oluşturulan *while* döngüsü *val != 0* koşuluyla döner. Yani *val* değişkeninin değeri *0* olunca bu döngüden çıkılır. Döngünün her turunda gövde içinde *digit_counter* değişkeninin değeri *1* artırılıyor. Daha sonra *val /= 10;* deyimiyle *val* değişkeni onda birine eşitleniyor.
Aşağıda bu kez kendisine gönderilen bir tamsayının basamak değerlerinin toplamı ile geri dönen *sum_digit* isimli bir işlev tanımlanıyor:

```
#include <stdio.h>

int sum_digit(int val)
{
    int digit_sum = 0;
```

```
    while (val) {
        digit_sum += val % 10;
        val /= 10;
    }

    return digit_sum;
}

int main()
{
    int val;

    printf("bir tamsayi girin :");
    scanf("%d", &val);

    printf("%d sayisinin basamaklari toplami = %d\n", val, sum_digit(val));

    return 0;
}
```

sum_digit işlevinde, yine parametre değişkeni olan *val,* bir döngü içinde sürekli *10*'a bölünüyor, *val 0* oluncaya kadar döngü gövdesindeki deyimler yürütülüyor. Döngü gövdesi içinde

```
digit_sum += val % 10;
```

deyimi ile *val* değişkeninin birler basamağı, değeri *digit_sum* değişkenine katılıyor. Böylece döngüden çıkıldıktan sonra *digit_sum* değişkeni, dışarıdan gönderilen sayının basamakları değerlerinin toplamını tutuyor.

Aşağıda, kendisine gönderilen bir tamsayının tersine geri dönen *get_rev_num* isimli bir işlev yazılıyor:

```
#include <stdio.h>

int get_rev_num(int val)
{
    int rev_number = 0;

    while (val) {
        rev_number = rev_number * 10 + val % 10;
        val /= 10;
    }
    return rev_number;
}

int main()
{
    int val;

    printf("bir tamsayi girin :");
    scanf("%d", &val);
    printf("%d sayisinin tersi = %d\n", val, get_rev_num(val));

    return 0;
}
```

get_rev_num işlevi içinde tanımlanan *rev_number* değişkeni *0* değeriyle başlatılıyor. İşlev içinde yer alan *while* döngüsünün, parametre değişkeni olan *val*'in değeri *0* olana kadar dönmesi sağlanıyor. Döngünün her turunda, *rev_number* değişkenine

```
rev_number * 10 + val % 10
```

ifadesinin değeri atanıyor.
İşleve gönderilen değerin *1357* olduğunu düşünelim:

rev_number	val
0	1357
7	135
75	13
753	1
7531	0

Döngü çıkışında *rev_number* değişkeninin değeri *7531* olur.
Aşağıdaki programda, bir tamsayıyı çarpanlarına ayıran ve çarpanları küçükten büyüğe ekrana yazdıran, *display_factors* isimli bir işlev tanımlanıyor:

```
#include <stdio.h>

void display_factors(int number)
{
    int k = 2;

    printf("(%d) -> ", number);
    while (number != 1) {
        while (number % k == 0) {
            printf("%d ", k);
            number /= k;
        }
        ++k;
    }
    printf("\n");
}
```

Aşağıdaki programda *3* basamaklı sayılardan *abc* == $a^3 + b^3 + c^3$ eşitliğini sağlayanlar ekrana yazdırılıyor:

```
#include <stdio.h>

int main()
{
    int k = 100;

    while (k < 1000) {
        int y = k / 100;
        int o = k % 100 / 10;
        int b = k % 10;
        if (y * y * y + o * o * o + b * b * b == k)
            printf("%d\n", k);
        ++k;
    }
    return 0;
}
```

Aşağıdaki işlevin hangi değeri hesapladığını bulmaya çalışın:

```
int func(int val)
{
    int sum = 0;

    while (val) {
        sum += val % 10;
        if (sum > 10)
            sum = 1 + sum % 10;
        val /= 10;
    }
    return sum;
}
```

do while Döngü Deyimi

do while döngü deyiminin genel sözdizimi aşağıdaki gibidir:

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

do while döngüsünde kontrol ifadesi sondadır. *while* ayracından sonra sonlandırıcı atom bulunmalıdır. Yani buradaki sonlandırıcı atom, döngü deyiminin sözdiziminin bir parçasıdır. *do while* döngüsünün yürütülmesi aşağıdaki gibi olur:
do anahtar sözcüğünü izleyen deyim döngüye girişte bir kez yapılır, daha sonra *while* ayracı içindeki kontrol ifadesine bakılır. Kontrol ifadesi doğru olduğu sürece döngü gövdesini oluşturan deyim yapılır. *do while* döngüsünün *while* döngüsünden farkı nedir? *while* döngüsünde döngü gövdesindeki deyimin en az bir kez yapılması güvence altında değildir. Ancak *do while* döngüsünde kontrol sonda yapıldığı için gövdedeki deyim en az bir kez yapılır.
Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    int val;

    do {
        printf("0 - 100 arasi bir deger girin : ");
        scanf("%d", &val);
    } while (val < 0 || val > 100);

    printf("val = %d\n", val);

    return 0;
}
```

main işlevinde *do while* döngüsü ile kullanıcı, *0 – 100* aralığında bir değer girmeye zorlanıyor. Eğer girilen değer *0*'dan küçük ya da *100*'den büyükse, kullanıcıdan yeni bir değer isteniyor.
Daha önce *while* döngüsü kullanarak yazılan *num_digit* isimli işlev, bu kez *do while* döngüsü ile yazılıyor:

```
int num_digit(int val)
{
    int digit_counter = 0;

    do {
        digit_counter++;
        val /= 10;
    } while(val != 0);

    return digit_counter;
}
```

Aşağıda tanımlanan *print_ulam* işleviyle bir tamsayıya ilişkin *ulam* serisi ekrana yazdırılıyor:

```
#include <stdio.h>

void print_ulam(int val)
{
    printf("%d icin ulam serisi\n", val);
    do {
        printf("%d ", val);
        if (val % 2 == 0)
            val /= 2;
        else
            val = val * 3 + 1;
    } while(val > 1);
    printf("%d\n", val);
}

int main()
{
    int x;

    printf("bir sayi girin: ");
    scanf("%d", &x);
    print_ulam(x);

    return 0;
}
```

for Döngü Deyimi

for döngü deyiminin genel sözdizimi aşağıdaki gibidir:

```
for (ifade1; ifade2; ifade3)
    deyim;
```

Derleyici *for* anahtar sözcüğünden sonra bir ayraç açılmasını ve ayraç içinde iki *noktalı virgül* atomu bulunmasını bekler. Bu iki noktalı virgül, *for* ayracını üç kısma ayırır. Bu üç kısımda da ifade tanımına uygun ifadeler yer alabilir.

for ayracı içinde iki noktalı virgül mutlaka bulunmalıdır. *for* ayracının içinin boş bırakılması, ya da *for* ayracı içinde *bir, üç* ya da daha fazla sayıda noktalı virgülün bulunması geçersizdir.
for ayracının kapanmasından sonra gelen ilk deyim, döngü gövdesini *(loop body)* oluşturur. Döngü gövdesi, yalın bir deyimden oluşabileceği gibi, bileşik deyimden de yani blok içine alınmış birden fazla deyimden de, oluşabilir. Döngü gövdesini bir boş deyim ya da bir kontrol deyimi de oluşturabilir.

for ayracı içindeki her üç ifadenin de ayrı ayrı işlevi vardır.

for ayracının ikinci kısmını oluşturan ifadeye kontrol ifadesi *(control expression*) denir. Tıpkı *while* ayracı içindeki ifade gibi, döngünün sürdürülmesi konusunda bu ifade söz sahibidir. Bu ifadenin değeri sıfırdan farklı ise, yani "doğru" olarak yorumlanırsa, döngü sürer. Döngü gövdesindeki deyim yürütülür. Kontrol ifadesinin değeri *0* ise, yani ifade yanlış olarak yorumlanırsa programın akışı *for* döngü deyimini izleyen ilk deyimin yürütülmesiyle sürer.

Programın akışı *for* deyimine gelince, *for* ayracının birinci kısmındaki ifade değerlendirilir. Birinci kısımdaki ifade genellikle döngü değişkenine ilkdeğer verme amacıyla kullanılır. Ancak şüphesiz böyle bir zorunluluk yoktur.

for ayracının üçüncü kısmındaki ifade, döngü gövdesindeki deyim ya da deyimler yürütüldükten sonra, kontrol ifadesi yeniden sınanmadan önce ele alınır. Bu kısım çoğunlukla, bir döngü değişkeninin artırılması ya da azaltılması amacıyla kullanılır. Aşağıdaki programı inceleyin:

```
#include <stdio.h>

int main()
{
    int i;

    for (i =  0; i < 2; ++i)
        printf("%d ", i);
    printf("\nson deger = %d\n", i);

    return 0;
}
```

Programın akışı *for* döngü deyimine gelince, önce *for* ayracı içindeki birinci ifade ele alınır. Yani *i* değişkenine *0* değeri atanır.
Şimdi programın akışı *for* ayracının ikinci kısmına, yani kontrol ifadesine gelir ve *i < 2* koşulu sorgulanır. Kontrol ifadesinin değeri sıfırdan farklı olduğu için, ifade mantıksal olarak doğru kabul edilir. Böylece programın akışı döngü gövdesine geçer. Döngü gövdesinin bir basit deyim tarafından oluşturulduğunu görüyorsunuz. Bu deyim yürütülür. Yani ekrana *i* değişkeninin değeri yazılarak imleç alt satıra geçirilir.
Programın akışı, bu kez *for* ayracının üçüncü kısmına gelir ve buradaki ifade ele alınır, yani *i* değişkeninin değeri *1* artırılır, *i* değişkeninin değeri *1* olur.
İkinci ifade yeniden değerlendirilir ve *i < 2* ifadesi doğru olduğu için bir kez daha döngü gövdesindeki deyim yürütülür.
Programın akışı yine *for* ayracının üçüncü kısmına gelir ve buradaki ifade ele alınır, yani *i* değişkeninin değeri 1 artırılır. *i* değişkeninin değeri *2* olur.
Programın akışı yine *for* ayracının ikinci kısmına gelir. Buradaki kontrol ifadesi yine sorgulanır. *i < 2* ifadesi, bu kez yanlış olduğu için programın akışı, döngü gövdesine girmez, döngü gövdesini izleyen ilk deyimle sürer. Yani ekrana:

```
sondeger = 2
```

yazılır.

Döngü Değişkenleri

for döngüsünde bir döngü değişkeni kullanılması gibi bir zorunluluk yoktur. Örneğin aşağıdaki döngü, kurallara tamamen uygundur:

```
for (func1(); func2(); func3())
    func4();
```

Yukarıdaki *for* döngü deyimiyle, döngüye girişte *func1* işlevi çağrılır. *func2* işlevi sıfır dışı bir değere geri döndükçe döngü gövdesindeki deyim yürütülür yani *func4* işlevi çağrılır. Kontrol ifadesine yeniden gelmeden, yani *func4* işlevinin çağrılmasından sonra bu kez *func3* işlevi çağrılır.
Aşağıdaki *for* döngü deyimiyle klavyeden `'x'` karakteri girilmediği sürece, alınan karakter ekrana yazdırılıyor:

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

int main()
{
    char ch;

    for (ch = getch(); ch != 'x' ; ch = getch())
        putchar(ch);

    return 0;
}
```

Döngü değişkeninin tamsayı türlerinden birinden olması gibi bir zorunluluk yoktur. Döngü değişkeni, gerçek sayı türlerinden de olabilir:

```
#include <stdio.h>

int main()
{
    double i;

    for (i = 0.1; i < 6.28; i += 0.01)
        printf("%lf ", i);

    return 0;
}
```

Yukarıdaki döngüde, *double* türden bir döngü değişkeni seçiliyor. *for* ayracının birinci kısmında döngü değişkenine *0.1* değeri atanıyor. Ayracın üçüncü kısmında ise döngü değişkeni *0.01* artırılıyor. Döngü, *i* değişkeninin değerinin *6.28*'den küçük olması koşuluyla dönüyor.

for Ayracı İçinde Virgül İşlecinin Kullanılması

Virgül işleci ile birleştirilmiş ifadelerin, soldan sağa doğru sırayla ele alındığını anımsayın. *for* döngülerinin birinci ve üçüncü kısmında virgül işlecinin kullanılmasına sık rastlanır. Aşağıdaki döngü deyimini inceleyin:

```
#include <stdio.h>

int main()
{
    int i, k;

    for (i = 1, k = 3; i * k < 12500; i += 2, k += 3)
        printf("(%d %d)", i, k);

    return 0;
}
```

Yukarıdaki *for* deyiminde, *for* ayracının birinci kısmında virgül işleci kullanılarak yazılan ifade ile, *i* değişkenine *1, k* değişkenine *3* değeri atanıyor. Döngü, *i * k* ifadesinin değeri *12500*'den küçük olduğu sürece döner. *for* ayracının üçüncü kısmında *i* değişkeninin değeri *2, k* değişkeninin değeri *3* artırılıyor.

for Ayracı İçindeki ifadelerin Olmaması

for döngü deyimi ayracının birinci kısmında bir ifade bulunmayabilir. Bu tamamen kurallara uygun bir durumdur. *1*'den *100*'e olan kadar sayıların ekrana yazdırılmak istendiğini düşünelim. Döngü değişkenine ilkdeğer verme işlemi, *for* ayracının birinci kısmından, *for* döngüsü dışına alınabilir:

```
#include <stdio.h>

int main()
{
    int i = 0;

    for (; i < 100; ++i)
        printf("%d ", i);
    return 0;
}
```

for döngü ayracının üçüncü kısmında da bir ifade bulunmayabilir. Döngü değişkeninin artırılması ya da eksiltilmesi, *for* ayracı içi yerine, döngü gövdesinde gerçekleştirilebilir:

```
#include <stdio.h>

int main()
{
    int i = 0;

    for (; i < 100;) {
        printf("%d ", i);
        i++;
    }
    return 0;
}
```

Birinci ve üçüncü ifadesi olmayan *for* döngüleri, *while* döngüleriyle tamamen eşdeğerdir. *while* döngüleriyle yazılabilen her kod, bir *for* döngüsüyle de yazılabilir. *while* döngüsü, birinci ve üçüncü kısmı olmayan *for* döngülerine okunabilirlik açısından daha iyi bir seçenek olur.
for ayracının ikinci ifadesi de hiç olmayabilir. Bu durumda kontrol ifadesi olmayacağı için döngü, bir koşula bağlı olmaksızın sürekli döner. Yani sonsuz döngü oluşturulur. Ancak iki adet *noktalı virgül,* yine ayraç içinde mutlaka bulunmak zorundadır. Aynı iş, bu kez bir sonsuz döngünün bilinçli kullanılmasıyla yapılıyor:

C programcılarının çoğu bilinçli bir şekilde sonsuz döngü oluşturmak istediklerinde *for (;;)* kalıbını yeğler. Bu kalıp *while (1)* kalıbına eşdeğerdir. İkisi de sonsuz döngü belirtir. Sonsuz döngü oluşturmak için *for (;;)* biçimi *while (1)* biçimine göre daha çok tercih edilir.

```
#include <stdio.h>

int main()
{
    int i = 0;
```

```
    for (;;) {
        if (i == 100)
            break;
        printf("%d ", i);
        i++;
    }
    return 0;
}
```

Şimdi de, aşağıdaki döngü deyiminin yürütülmesiyle ekrana ne yazdırılacağını kestirmeye çalışın:

```
#include <stdio.h>

int main()
{
    double d;

    for (d = 1.5; d < 3,0; d += 0.1)
        printf("%lf ", d);

    return 0;
}
```

Ekrana hiçbir şey yazılmaz! Döngünün kontrol ifadesinin *d < 3,0* olduğunu görüyorsunuz. Gerçek sayı değişmezi yazarken `'.'` yerine yanlışlıkla *virgül* karakteri kullanılmış. Bu durumda *virgül* işlecinin ürettiği değer, ikinci terim olan *0* değeridir. Kontrol ifadesi yanlış olarak yorumlanır böylece döngü gövdesindeki deyim hiç yürütülmez.

n Kez Dönen for Döngüleri

n 0'dan büyük bir tamsayı olmak üzere, aşağıdaki döngülerden hepsi *n* kez döner.

```
for (i = 0; i < n; ++i)

for (i = 1; i <= n; ++i)

for (i = n - 1; i >= 0; --i)

for (i = n; i > 0; --i)
```

for Döngülerinde continue Deyiminin Kullanımı

Bir döngünün gövdesi içinde *continue* deyiminin kullanılması ile, gövde içinde geriye kalan deyimlerin atlanarak döngünün bir sonraki turuna geçilir. *for* döngüsü gövdesi içinde *continue* deyimi ile karşılaşıldığında, programın akışı *for* ayracının üçüncü ifadesine gelir ve bu ifade ele alınır.

Döngü Değişkeninin Bayrak Amaçlı Kullanılması

Bazı uygulamalarda, *for* döngüsünün döngü değişkeni, bir bayrak görevi de görür. Bir *for* döngüsü içinden, belirli bir koşul oluştuğunda çıkılması gereksin:

```
for (i = 0; i < 100; ++i)
    if (is_valid(i))
        break;
```

Yukarıdaki döngü deyiminin çalıştırılması sonucunda iki farklı durum söz konusudur. Eğer döngünün gövdesinde *break* deyimi yürütülürse, yani herhangi bir *i* değeri için *is_valid* işlevi *sıfır* dışı bir değere geri dönerse, döngü çıkışında *i* değişkeninin değeri, *100*'den

küçük olur. *break* deyimi hiç yürütülmeden döngü tüm turlarını tamamlarsa, döngü çıkışında *i* değişkeninin değeri *100* olur.
Döngü çıkışında *i* değerinin *100* olup olmadığının sınanması ile, döngüden nasıl çıkıldığı anlaşılabilir.

for Döngü Deyiminin Kullanımına Örnekler

Aşağıdaki programda, iki sayının ortak bölenlerinin en büyüğü ve ortak katlarının en küçüğünü hesaplayan *okek* ve *obeb* isimli işlevler tanımlanıyor:

```
#include <stdio.h>

int obeb(int number1, int number2)
{
    int i;
    int min = (number1 < number2) ? number1 : number2;

    for (i = min; i >= 1; --i)
        if (number1 % i == 0 && number2 % i == 0)
            return i;
    return 1;
}

int okek(int number1, int number2)
{
    int i;
    int max = (number1 > number2) ? number1 : number2;

    for (i = max; i <= number1 * number2; i += max)
        if (i % number1 == 0 && i % number2 == 0)
            return i;
    return number1 * number2;
}

int main()
{
    int x, y;
    int n = 5;

    while (n--) {
        printf("iki tamsayi girin : ");
        scanf("%d%d", &x, &y);
        printf("obeb = %d\n", obeb(x, y));
        printf("okek = %d\n", okek(x, y));
    }
    return 0;
}
```

Aşağıda tanımlanan işlev, ortak bölenlerin en büyüğünü *Euclid* algoritmasıyla buluyor:

```
int obeb(int a, int b)
{
    int temp;

    while (b) {
        temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

Aşağıdaki programda, bir tamsayı için faktöriyel değerini hesaplayan, *fact* isimli bir işlev yazılıyor. Program, *int* türünün *4 byte* olduğu bir sistemde derlenmeli:

```
#include <stdio.h>

int fact(int number)
{
    int i;
    int result = 1;

    if (number == 0 || number == 1)
        return 1;

    for (i = 2; i <= number; ++i)
        result *= i;

    return result;
}

int main()
{
    int k;

    for (k = 0; k < 14; ++k)
        printf("%2d! = %-10d\n", k, fact(k));

    return 0;
}
```

Aşağıda e sayısı, *fact* işlevi kullanılarak bir seri toplamıyla bulunuyor:

```
#include <stdio.h>

int fact(int);

int main()
{
    int k;
    double e = 0.;

    for (k = 0; k < 14; ++k)
        e += 1. / fact(k);

    printf("e = %lf\n", e);

    return 0;
}
```

Aşağıdaki programda, kendisine gönderilen bir tamsayının asal olup olmadığını sınayan *isprime* isimli bir işlev yazılıyor. İşlev, eğer kendisine gönderilen sayı asal ise sıfırdan farklı bir değere, asal değil ise *0* değerine geri dönüyor. Sınama amacıyla yazılan *main* işlevinde *isprime* işlevi çağrılarak, *1000*'den küçük asal sayılar ekrana yazdırılıyor:

```
#include <stdio.h>

int isprime(int number)
{
    int k;

    if (number == 0 || number == 1)
        return 0;

    if (number % 2 == 0)
        return number == 2;

    if (number % 3 == 0)
        return number == 3;

    if (number % 5 == 0)
        return number == 5;

    for (k = 7; k * k <= number; k += 2)
        if (number % k == 0)
            return 0;

    return 1;
}

int main()
{
    int k;
    int prime_counter = 0;

    for (k = 0; k < 1000; ++k)
        if (isprime(k)) {
            if (prime_counter % 10 == 0 && prime_counter)
                putchar('\n');
            prime_counter++;
            printf("%3d ", k);
        }
    return 0;
}
```

Bölenlerinin toplamına eşit olan tamsayılara, mükemmel tamsayı *(perfect integer*) denir. Örneğin *6* ve *28* tamsayıları mükemmel tamsayılardır.

```
1 + 2 + 3 = 6
1 + 2 + 4 + 7 + 14 = 28
```

Aşağıdaki program ile *10000*'den küçük mükemmel sayılar aranıyor. Bulunan sayılar ekrana yazdırılıyor:

```
#include <stdio.h>

int is_perfect(int number);

int main()
{
    int k;

    for (k = 2; k < 10000; ++k)
        if (is_perfect(k))
            printf("%d perfect\n", k);
```

```
    return 0;
}

int is_perfect(int number)
{
    int i;
    int total = 1;

    for (i = 2; i <= number / 2; ++i)
        if (number % i == 0)
            total += i;
    return number == total;
}
```

Aşağıdaki programda klavyeden sürekli karakter alınması sağlanıyor, alınan karakterler ekranda gösteriliyor. Arka arkaya *"xyz"* karakterleri girildiğinde program sonlandırılıyor:

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

int main()
{
    char ch;
    int total = 0;

    while (total < 3) {
        ch = getch();
        putchar(ch);
        if (ch == 'x' && total == 0)
            total++;
        else if (ch == 'y' && total == 1)
            total++;
        else if (ch == 'z' && total == 2)
            total++;
        else total = 0;
    }
    return 0;
}
```

İç İçe Döngüler

Bir döngünün gövdesini başka bir döngü deyimi oluşturabilir. Böyle yaratılan döngülere iç içe döngüler (*nested loops*) denir. Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    int i, k;

    for (i = 0; i < 5; ++i)
        for (k = 0; k < 10; ++k)
            printf("(%d %d) ", i, k);
    printf("\n\n(%d %d) ", i, k);
    return 0;
}
```

Dıştaki *for* döngüsünün gövdesindeki deyim, bir başka for döngüsüdür. *i < 5* ifadesi doğru olduğu sürece içteki *for* döngü deyimi yürütülür. Son yapılan *printf* çağrısı ekrana hangi değerleri yazdırır? Dıştaki döngü *i < 5* koşuluyla döndüğüne göre dıştaki döngüden çıktıktan sonra i değişkeninin değeri *5* olur. İçteki for döngü deyiminin son kez

yapılmasından sonra da *k* değişkeninin değeri *10* olur. Bu durumda en son *printf* çağrısı ekrana *(5 10)* yazdırır.
Şimdi de aşağıda kodu verilen *put_star* işlevinin ne iş yaptığını bulmaya çalışın:

```
#include <stdio.h>

void put_stars(int n)
{
    int i, k;
    for (i = 1; i <= n; ++i) {
        for (k = 1; k <= i; ++k)
            putchar('*');
        putchar('\n');
    }
}

int main()
{
    int val;

    printf("bir deger girin : ");
    scanf("%d", &val);
    put_stars(val);

    return 0;
}
```

Aşağıdaki programda $abc = a^3 + b^3 + c^3$ eşitliğini sağlayan üç basamaklı sayıları ekrana yazdırıyor.

```
#include <stdio.h>

int main()
{
    int i, j, k;
    int number = 100;

    for (i = 1; i <= 9; ++i)
        for (j = 0; j <= 9; ++j)
            for (k = 0; k <= 9; ++k) {
                if (i * i * i + j * j * j + k * k * k == number)
                    printf("%d\n", number);
                number++;
            }

    return 0;
}
```

İç İçe Döngülerde break Deyiminin Kullanılması

İç içe döngülerde *break* deyimi kullanımına dikkat etmek gerekir. İçteki bir döngünün gövdesinde *break* deyiminin kullanılması ile, yalnızca içteki döngüden çıkılır: Aşağıdaki örneği inceleyin:

```
while (1) {
    while (1) {
        if (ifade)
            break;
        /***/
    }
    /*iç döngüden break ile çıkıldığında akış bu noktaya gelir */
}
```

Eğer iç içe döngülerden yalnızca içtekinden değil de döngülerin hepsinden birden çıkmak istenirse bu durumda *goto* kontrol deyimi kullanılmalıdır. Bu konuyu *goto* kontrol deyimi bölümünde göreceksiniz.
Burada ikinci *while* döngüsü tek bir kontrol deyimi olarak ele alınacağı için bloklamaya gerek yoktur.

Döngülerden Çıkış

Bir döngüden nasıl çıkılabilir? Bir döngüden çıkmak için aşağıdaki yollardan biri kullanılabilir.
1. Kontrol ifadesinin yanlış olmasıyla:
Döngü deyimlerinin, kontrol ifadelerinin doğru olduğu sürece döndüğünü biliyorsunuz.

2. *return* deyimi ile:
Bir işlev içinde yer alan *return* deyimi işlevi sonlandırdığına göre, bir döngü deyimi içinde *return* deyimi ile karşılaşıldığında döngüden çıkılır.

3. *break* deyimi ile:
break deyiminin kullanılması ile, programın akışı döngü deyimini izleyen ilk deyimle sürer.

4. *goto* deyimi ile:
goto deyimi ile bir programın akışı aynı işlev içinde döngünün dışında bir başka noktaya yönlendirilebilir. Böylece döngüden çıkılabilir.

5. Programı sonlandıran bir işlev çağrısı ile:
Standart *exit* ya da *abort* işlevleri ile programın kendisi sonlandırılabilir.

Bir döngüden çıkmak amacıyla, kontrol ifadesinin yanlış olmasını sağlamak için döngü değişkenine doğal olmayacak bir biçimde değer atanması, programın okunabilirliğini bozar. Böyle kodlardan kaçınmak gerekir.

KOŞUL İŞLECİ

Koşul işleci (*conditional operator / ternary operator*), C dilinin üç terimli tek işlecidir. Herhangi bir ifade koşul işlecinin terimlerinden biri olabilir. Koşul işlecinin genel sözdizimi aşağıdaki gibidir:

```
ifade1 ? ifade2 : ifade3
```

Koşul işleci, yukarıdaki biçimden de görüldüğü gibi, birbirinden ayrılmış iki atomdan oluşur. *?* ve : atomları, işlecin üç terimini birbirinden ayırır.
Derleyici, bir koşul işleci ile karşılaştığını, *?* atomundan anlar. *?* atomunun solundaki ifadenin (*ifade1*) sayısal değeri hesaplanır. Bu ifade mantıksal olarak yorumlanır. Eğer *ifade1*'in *0*'dan farklı ise, bu durumda yalnızca *ifade2*' nin sayısal değeri hesaplanır. *ifade1*'in değeri *0* ise, bu kez yalnızca *ifade3*'ün sayısal değeri hesaplanır.
Diğer işleçlerde olduğu gibi koşul işleci de bir değer üretir. Koşul işlecinin ürettiği değer *ifade1* doğru ise (*0* dışı bir değer ise) *ifade2*'nin değeri, *ifade1* yanlış ise *ifade3*'ün değeridir. Örnek:

```
m = x > 3 ? y + 5 : y - 5;
```

Burada önce *x > 3* ifadesinin sayısal değeri hesaplanır. Bu ifadenin değeri *0*'dan farklı ise yani doğru ise, koşul işleci *y + 5* değerini üretir. *x > 3* ifadesinin değeri *0* ise yani ifade yanlış ise, koşul işleci *y – 5* değerini üretir. Bu durumda *m* değişkenine *x > 3* ifadesinin doğru ya da yanlış olmasına göre *y + 5* ya da *y – 5* değeri atanır.

Aynı işlem *if* deyimi ile de yapılabilir :

```
if (x > 3)
    m = y + 5;
else
    m = y - 5;
```

Koşul işleci, işleç öncelik tablosunun 13. öncelik seviyesindedir. Bu seviye atama işlecinin hemen üstüdür. Aşağıdaki ifadeyi ele alalım:

```
x > 3 ? y + 5 : y - 5 = m
```

Koşul işlecinin önceliği atama işlecinden daha yüksek olduğu için, önce koşul işleci ele alınır. *x > 3* ifadesinin *doğru* olduğunu ve işlecin *y + 5* değerini ürettiğini düşünelim.

```
y + 5 = m
```

Koşul işlecinin ürettiği değer sol taraf değeri olmadığından, yukarıdaki ifade geçersizdir. Normal olarak koşul işlecinin ilk terimini ayraç içine almak gerekmez. Ancak bu terimin, okunabilirlik açısından genellikle ayraç içine alınması tercih edilir.

```
(x >= y + 3) ? a * a : b
```

Koşul işlecinin üçüncü terimi konusunda dikkatli olmak gerekir. Örneğin:

```
m = a > b ? 20 : 50 + 5
```

a > b ifadesinin doğru olup olmamasına göre koşul işleci, *20* ya da *55* değerini üretir ve son olarak da *m* değişkenine koşul işlecinin ürettiği değer atanır. Ancak *m* değişkenine

```
a > b ? 20 : 50
```

ifadesinin değerinin *5* fazlası atanmak isteniyorsa bu durumda ifade aşağıdaki gibi düzenlenmelidir:

```
m = (a > b ? 20 : 50) + 5;
```

Koşul işlecinin üç terimi de bir işlev çağrı ifadesi olabilir, ama çağrılan işlevlerin, geri dönüş değeri üreten işlevler olması gerekir. Üç teriminden birinin geri dönüş değeri *void* olan bir işleve ilişkin işlev çağrı ifadesi olması, geçersiz bir durum oluşturabilir.
Aşağıdaki kod parçasını inceleyin:

```
#include <stdio.h>

int func1(void);
int func2(void);
int func3(void);

int main()
{
    int m;

    m = func1() ? func2() : func3();

    return 0;
}
```

Yukarıda koşul işlecinin kullanıldığı ifadede *m* değişkenine, *func1* işlevinin geri dönüş değerinin sıfır dışı bir değer olması durumunda *func2* işlevinin geri dönüş değeri, aksi halde *func3* işlevinin geri dönüş değeri atanır.
Koşul işlecinin ürettiği, bir nesne değil bir değerdir. Koşul işlecinin ürettiği değer nesne göstermediği için bu değere bir atama yapılamaz. Aşağıdaki *if* deyimini inceleyin:

```
if (x > y)
    a = 5;
else
    b = 5;
```

Yukarıdaki *if* deyiminde *x > y* ifadesinin doğru olması durumunda *a* değişkenine, yanlış olması durumunda ise *b* değişkenine *5* değeri atanıyor. Aynı iş koşul işlecinin kullanılmasıyla yaptırılırsa:

```
(x > y) ? a : b = 5;   /* Geçersiz! */
```

Bu durum derleme zamanı hatasına yol açar. Çünkü koşul işlecinin ürettiği *a* ya da *b* değişkenlerinin değeridir, nesnenin kendisi değildir. Böyle bir atama sol tarafın nesne gösteren bir ifade olmaması nedeniyle derleme zamanında hata oluşturur.
Aynı nedenden dolayı aşağıdaki ifade de geçersizdir:

```
(x > 5 ? y : z)++;     /* Geçersiz! */
```

Ayraç içindeki ifade değerlendirildiğinde elde edilen, *y* ya da *z* nesneleri değil, bunların değerleridir. Yani sonek konumundaki ++ işlecinin terimi nesne değildir.

[C++ dilinde koşul işlecinin 2. ya da 3. teriminin nesne olması durumunda işlecin ürettiği değer sol taraf değeridir. Yani yukarıdaki deyimler C de geçersiz iken C++'da geçerlidir.]

Koşul İşlecinin Kullanıldığı Durumlar

if deyiminin yerine koşul işleci kullanmak her zaman doğru değildir. Koşul işlecinin kullanılmasının salık verildiği tipik durumlar vardır. Bu durumlarda genel fikir, koşul işlecinin ürettiği değerden aynı ifade içinde faydalanmak, bu değeri bir yere aktarmaktır:

1. Koşul işlecinin ürettiği değer bir nesneye atanabilir.

```
p = (x == 5) ? 10 : 20;
m = (a >= b + 5) ? a + b : a - b;
```

Yukarıdaki deyimlerin işini görecek *if* deyimleri de yazılabilirdi:

```
if (x == 5)
    p = 10;
else
    p = 20;
```

```
if (a >= b + 5)
    m = a + b;
else
    m = a - b;
```

2. Bir işlev, koşul işlecinin ürettiği değer ile geri dönebilir:

```
return x > y ? 10 : 20;
```

Bu örnekte *x > y* ifadesinin doğru olup olmamasına göre işlev, *10* ya da *20* değerine geri döner. Yukarıdaki ifade yerine aşağıdaki *if* deyimi de kullanılabilirdi :

```
if (x > y)
    return 10;
return 20;
```

3. Koşul işlecinin ürettiği değer ile bir işlev çağrılabilir:

```
func(a == b ? x : y);
```

Yukarıdaki deyimde, *a, b*'ye eşit ise *func* işlevi *x* değeri ile, *a, b*'ye eşit değil ise *y* değeri ile çağrılır. Aynı işi gören bir *if* deyimi de yazılabilirdi:

```
if (a == b)
    func(x);
else
    func(y);
```

4. Koşul işlecinin ürettiği değer, bir kontrol deyiminin kontrol ifadesinin bir parçası olarak da kullanılabilir:

```
if (y == (x > 5 ? 10 : 20))
    func();
```

Yukarıdaki deyimde *x > 5* ifadesinin doğru olup olmamasına göre, *if* ayracı içinde, y değişkeninin *10* ya da *20* değerine eşitliği sorgulanır.
Yukarıdaki durumlarda, koşul işlecinin *if* deyimine tercih edilmesi iyi tekniktir. Bu durumlarda koşul işleci daha okunabilir bir yapı oluşturur.
Koşul işlecinin bilinçsizce kullanılmaması gerekir. Eğer koşul işlecinin ürettiği değerden doğrudan faydalanılmayacaksa koşul işleci yerine *if* kontrol deyimi tercih edilmelidir. Örneğin:

```
x > y ? a++ : b++;
```

Deyiminde koşul işlecinin ürettiği değerden faydalanılmıyor. Burada aşağıdaki *if* deyimi tercih edilmelidir:

```
if (x > y)
    a++;
else
    b++;
```

Başka bir örnek:

```
x == y ? printf("eşit\n") : printf("eşit değil\n");
```

Bu örnekte, *printf* işlevinin bir geri dönüş değeri üretmesinden faydalanılarak koşul işleci kullanılmış. Koşul işleci, *x == y* ifadesinin doğru olup olmamasına göre, ikinci veya üçüncü ifade olan *printf* işlevi çağrılarından birinin geri dönüş değerini üretir. Bu da aslında ekrana yazılan karakter sayısıdır. Ama ifade içinde, koşul işlecinin ürettiği değerin kullanılması söz konusu değildir. Burada da *if* deyimi tercih edilmelidir:

```
if (x == y)
    printf("eşit\n");
else
    printf("eşit değil\n");
```

Koşul işlecinin ikinci ve üçüncü terimlerinin türleri farklı ise, diğer işleçlerde olduğu gibi tür dönüştürme kuralları devreye girer:

```
int i;
double d;

m = (x == y) ? i : d;
```

Bu örnekte *i* değişkeni *int* türden, *d* değişkeni ise *double* türdendir. *x == y* karşılaştırma ifadesi doğru ise, koşul işlecinin ürettiği değerin türü *double* türüdür.

Bazı durumlarda, *if* deyiminin de, koşul işlecinin de, kullanılması gerekmez:

```
if (x > 5)
    m = 1;
else
    m = 0;
```

Yukarıdaki *if* deyimi yerine aşağıdaki deyim yazılabilirdi:

```
m = (x > 5) ? 1 : 0;
```

Koşul işlecinin üreteceği değerlerin yalnızca *1* veya *0* olabileceği durumlarda, doğrudan karşılaştırma işleci kullanmak daha iyi teknik olarak değerlendirilmelidir:

```
m = x > 5;
```

Başka bir örnek :

```
return x == y ? 1 : 0;
```

yerine

```
return x == y;
```

yazılabilirdi.
Koşul işlecinin öncelik yönü sağdan soladır. Bir ifade içinde birden fazla koşul işleci varsa, önce en sağdaki değerlendirilir. Aşağıdaki kod parçasını inceleyin:

```
#include <stdio.h>

int main()
{
    int x = 1, y = 1,  m;

    m = x  <  5 ? y == 0 ? 4 : 6 : 8;
    printf("m = %d\n", m);

    return 0;
}
```

Yukarıdaki *main* işlevinde *printf* işlevi çağrısı ile *m* değişkeninin değeri olarak ekrana *6* yazılır. İfade aşağıdaki gibi ele alınır:

```
m = x  <  5 ? (y == 0 ? 4 : 6) : 8;
```

Koşul İşlecinin Kullanımına Örnekler

Aşağıda iki sayıdan büyük olanına geri dönen, *max2* isimli işlev tanımlanıyor.

```
int max2(int a, int b)
{
    return a > b ? a : b;
}
```

$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}+\ldots$ serisi $\frac{pi}{4}$ e yakınsar. Aşağıda bir döngü ile, *pi* sayısı hesaplanıyor.
Döngü gövdesinde koşul işlecinin kullanımını inceleyin:

```
#include <stdio.h>

int main()
{
    double sum = 0.;
    int k;

    for (k = 0; k < 10000; ++k)
        sum += (k % 2 ? -1. : 1.) / (2 * k + 1);

    printf("pi = %lf\n", 4. * sum);

    return 0;
}
```

ÖNİŞLEMCİ KOMUTLARI (1)

C derleyicileri iki ayrı modülden oluşur:

1. Önişlemci Modülü
2. Derleme Modülü

Önişlemcinin, bilgisayarın işlemcisi ya da başka bir donanımsal elemanıyla hiçbir ilgisi yoktur. Önişlemci, belirli bir iş gören bir yazılım programıdır.

Önişlemci, kaynak dosya üzerinde birtakım düzenlemeler ve değişiklikler yapan bir ön programdır. Önişlemci programının bir girdisi bir de çıktısı vardır. Önişlemcinin girdisi kaynak dosyanın kendisidir. Önişlemci programın çıktısı ise derleme modülünün girdisini oluşturur. Yani kaynak program ilk aşamada önişlemci tarafından ele alınır. Önişlemci modülü, kaynak dosyada çeşitli metinsel düzenlemeler, değişiklikler yapar. Daha sonra değiştirilmiş ya da düzenlenmiş olan bu kaynak dosya, derleme modülü tarafından amaç koda dönüştürülür.

C programlama dilinde # ile başlayan bütün satırlar, önişlemci programa verilen komutlardır (*directives*).
Önişlemci program, önceden belirlenmiş bir komut kümesindeki işlemleri yapabilir. Her bir komut, # atomunu izleyen bir sözcükle belirlenir. Aşağıda tüm önişlemci komutlarının listesi veriliyor:

```
#include
#define
#if
#else
#elif
#ifdef
#ifndef
#endif
#undef
#line
#error
#pragma
```

Önişlemci komutlarını belirleyen yukarıdaki sözcükler, C dilinin anahtar sözcükleri değildir. Sıra derleyiciye geldiğinde bunlar, önişlemci tarafından kaynak dosyadan silinmiş olur. Örneğin, istenirse *include* isimli bir değişken tanımlanabilir, ama bunun okunabilirlik açısından iyi bir fikir olmadığı söylenebilir. Önişlemci komutlarını belirten sözcükler, ancak # karakterini izledikleri zaman özel anlam kazanır.

Önişlemci program, amaç kod oluşturmaya yönelik hiçbir iş yapmaz, kaynak kod içinde bazı metinsel düzenlemeler yapar. Kendisine verilen komutları yerine getirdikten sonra, # ile başlayan satırları kaynak dosyadan siler. Derleme modülüne girecek programda # ile başlayan satırlar artık yer almaz.

Şimdilik önişlemci komutlarından yalnızca *#include* ve *#define* komutlarını göreceksiniz. Geriye kalan önişlemci komutları ileride ayrıntılı olarak ele alınacak.

#include Önişlemci Komutu

Bu önişlemci komutunun genel sözdizimi aşağıdaki gibidir:

```
#include <dosya ismi>
```

ya da

```
#include "dosya ismi"
```

#include komutu ile, ismi verilen dosyanın içeriği, bu komutun yazıldığı yere yapıştırılır. Bu komut ile önişlemci, belirtilen dosyayı diskten okuyarak komutun yazılı olduğu yere yerleştirir. Bu komutla yapılan iş, metin düzenleyici programlardaki "kopyala - yapıştır" (*copy – paste*) işlemine benzetilebilir.

#include önişlemci komutuyla, kaynak dosyaya eklenmek istenen dosyanın ismi iki ayrı biçimde belirtilebilir:

1. Açısal ayraç içinde:

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

2. Çift tırnak içinde

```
#include "general.h"
#include "genetic.h"
```

Dosya ismi eğer açısal ayraç içinde verilmişse, sözkonusu dosya önişlemci tarafından, yalnızca önceden belirlenmiş bir dizin içinde aranır. Çalışılan derleyiciye ve sistemin kurulumuna bağlı olarak, önceden belirlenmiş bu dizin farklı olabilir. Örneğin:

```
\tc\include
\borland\include
\c600\include
```

gibi. Benzer biçimde *UNIX* sistemleri için bu dizin, örneğin:

```
/usr/include
```

biçiminde olabilir. Standart başlık dosyaları, açısal ayraç içinde kaynak koda eklenir.

Sistemlerin çoğunda dosya ismi iki tırnak içine yazıldığında, önişlemci ilgili dosyayı önce çalışılan dizinde *(current directory*) arar. Burada bulamazsa sistem ile belirlenen dizinde arar. Örneğin:

```
C:\sample
```

dizininde çalışıyor olalım.

```
#include "strfunc.h"
```

komutu ile, önişlemci *strfunc.h* isimli dosyayı önce *C:\sample* dizininde arar. Eğer burada bulamazsa sistem tarafından belirlenen dizinde arar. Programcıların kendilerinin oluşturdukları başlık dosyaları, genellikle sisteme ait dizinde olmadıkları için, çift tırnak içinde kaynak koda eklenir.
#İnclude önişlemci komutu ile kaynak koda eklenmek istenen dosya ismi, dosya yolu (*path*) da içerebilir:

```
#include <sys\stat.h>
#include "c:\headers\myheader.h"
```

#include önişlemci komutu kaynak programın herhangi bir yerinde bulunabilir. Fakat standart başlık dosyaları gibi, içinde çeşitli bildirimlerin bulunduğu 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 eklemek etmek için şöyle bir yöntem izlenebilir.

```
ana.c

#include "project.h"

int main()
{
    /****/
}
```

```
project.h

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

ana.c dosyası içine yalnızca *project.h* dosyası ekleniyor. Önişlemci bu dosyayı kaynak koda ekledikten sonra yoluna bu dosyadan devam eder.

Başlık Dosyaları Neden Kullanılır?

Özellikle büyük programlar, modül ismi verilen ayrı ayrı parçalar halinde yazılır. Bu modüllerden bazılarının amacı, diğer modüllere hizmet vermektir. C ve C++ dillerinde, genel hizmet verecek kodlar (*server codes*), genel olarak iki ayrı dosya halinde yazılır. İşlev tanımlamaları, global değişken tanımlamaları uzantısı *.c* olan dosyada yer alır. Bu dosyaya, kodlama dosyası (*implementation file*) denir. Hizmet alacak kodları (*client codes*) ilgilendiren bildirimler ise bir başka dosyada tutulur. Bu dosyaya, başlık dosyası *(header file)* denir. Bir başlık dosyası, bir modülün arayüzüdür *(interface)*. Modül dışarıyla olan ilişkisini arayüzü ile kurar.
Verilen hizmetlerden faydalanacak kullanıcı kodlar, hizmet veren kodların kendisini değil, yalnızca arayüzünü görür. Hizmet alan kodlar, hizmet veren kodların arayüzlerine bağlı olarak yazılır. Böylece hizmet veren kodların kendisi ile arayüzleri, birbirinden net olarak ayrılmış olur.
Hizmet veren kodların arayüzleriyle tanımlarını birbirinden ayırmanın ne gibi faydaları olabilir?
Kullanıcı kodlar, yani hizmet alan kodlar, hizmet veren işlevlerin tanımlarına göre değil de, arayüzlerine bağlı olarak yazılır. Bundan aşağıdaki faydalar sağlanabilir:

1. Hizmet veren kodları yazanlar, aynı arayüze bağlı kalmak kaydıyla, tanım kodlarında değişiklik yapabilir. Bu durumda hizmet alan kodlarda bir değişiklik yapılması gerekmez.

2. Kullanıcı kodları yazacak programcı, hizmet veren kodlara ilişkin uygulama ayrıntılarını bilmek zorunda kalmadığından, daha kolay soyutlama yapar.

3. Birden fazla programcının aynı projede çalışması durumunda, proje geliştirme süresi kısaltılmış olur.

#define Önişlemci Komutu

#define önişlemci komutunun işlevi, metin düzenleyici programlardaki "bul - değiştir" *(find - replace)* özelliğine benzetilebilir. Bu komut kaynak kod içindeki bir yazıyı başka bir yazı ile değiştirmek için kullanılır.

Önişlemci, *define* sözcüğünden sonraki boşlukları atarak, boşluksuz ilk yazı kümesini elde eder. Bu yazıya *STR1* diyelim. Daha sonra satır sonuna kadar olan tüm yazı kümesi elde edilir. Buna da *STR2* diyelim. Önişlemci, kaynak kod içinde *STR1* yazısı yerine *STR2* yazısını yerleştirir:

```
#define  SIZE   100
```

önişlemci komutuyla, önişlemci kaynak kod içinde gördüğü her bir *SIZE* atomu yerine *100* atomunu yerleştirir. Derleme modülüne girecek kaynak programda, *SIZE* atomu artık yer almaz.

#define önişlemci komutu kullanılarak çoğunlukla bir isim, sayısal bir değerle yer değiştirilir. Sayısal bir değerle değiştirilen isme, "simgesel değişmez" *(symbolic constant)* denir. Simgesel değişmezler nesne değildir. Derleme modülüne giren kaynak kodda, simgesel değişmezlerin yerini sayısal ifadeler almış olur.
#define önişlemci komutuyla tanımlanan isimlere, "basit makro" (*simple macro*) da denir. Simgesel değişmezler, geleneksel olarak büyük harf ile isimlendirilir. Böylece kodu okuyanın değişkenlerle, simgesel değişmezleri ayırt edebilmesi sağlanır. Bilindiği gibi C dilinde, değişken isimlendirmelerinde ağırlıklı olarak küçük harfler kullanılır.

Bir simgesel değişmez, başka bir simgesel değişmezin tanımlamasında kullanılabilir. Örneğin:

```
#define MAX      100
#define MIN      (MAX - 50)
```

Yer değiştirme işlemi, *STR1*'in kaynak kod içinde bir atom halinde bulunması durumunda yapılır:

```
#define SIZE       100
```

Bu tanımlamadan sonra kaynak kodda

```
size = MAX_SIZE;
printf("SIZE = %d\n", size);
```

gibi deyimlerin bulunduğunu düşünelim. Önişlemci bu deyimlerin hiçbirinde bir değişiklik yapmaz.

```
size = MAX_SIZE
```

ifadesinde *SIZE* ayrı bir atom değildir. Atom olan *MAX_SIZE*'dır. Yer değiştirme işlemi büyük küçük harf duyarlığı ile yapılacağından, kaynak kod içinde yer alan *size* ismi de değiştirilecek atom değildir.

```
printf("SIZE = %d\n", max_size)
```

ifadesinde atom olan dizge ifadesidir. Yani dizge içindeki *SIZE,* tek başına ayrı bir atom değildir.

#define önişlemci komutu ile değişmezlere ve işleçlere ilişkin yer değiştirme işlemi yapılamaz.

Aşağıdaki *#define* önişlemci komutları geçerli değildir:

```
#define +       -
#define 100    200
```

Simgesel değişmezler, C dilinin değişken isimlendirme kurallarına uygun olarak isimlendirilmelidir:

```
#define    BÜYÜK  10
```

tanımlaması geçersizdir.

Önişlemci program, *#include* komutu ile kaynak koda eklenen dosyanın içindeki önişlemci komutlarını da çalıştırır. Bu durumda içinde simgesel değişmez tanımlamaları yapılmış bir dosya, *#include* komutu ile kaynak koda eklendiğinde, bu simgesel değişmezler de kaynak kod içinde tanımlanmış gibi geçerli olur.

#define önişlemci komutunda dizgeler de kullanılabilir:

```
#define HATA_MESAJI    "DOSYA AÇILAMIYOR \n"
/***/
printf(HATA_MESAJI);
/***/
```

Simgesel değişmez tanımında kullanılacak dizge uzunsa, kodun okunmasını kolaylaştırmak için, birden fazla satıra yerleştirilebilir. Bu durumda, son satır dışındaki satırların sonuna "\" atomu yerleştirilmelidir.

Okunabilirlik açısından, tüm simgesel değişmez tanımlamaları alt alta gelecek biçimde yazılmalıdır. Seçilen simgesel değişmez isimleri, kodu okuyan kişiye bunların ne amaçla kullanıldığı hakkında fikir vermelidir.

Bir simgesel değişmezin tanımlanmış olması, kaynak kod içinde değiştirilebilecek bir bilginin olmasını zorunlu hale getirmez. Tanımlanmış bir simgesel değişmezin kaynak kod içinde kullanılmaması, herhangi bir hataya yol açmaz.

Simgesel Değişmezler Kodu Daha Okunabilir Kılar

Simgesel değişmezler, yazılan kodun okunabilirliğini ve algılanabilirliğini artırır. Bazı değişmezlere isimlerin verilmesi, bu değişmezlerin ne amaçla kullanıldığı hakkında daha fazla bilgi verilebilir. Aşağıdaki örneğe bakalım:

```
#define PERSONEL_SAYISI   750
```

```
void foo()
{
    /***/
    if (x == PERSONEL_SAYISI)
        /***/
}
```

Kaynak kod içinde *PERSONEL_SAYISI* simgesel değişmezi yerine doğrudan *750* değeri kullanılmış olsaydı, kodu okuyanın, bu değişmezin ne anlama geldiğini çıkarması çok daha zor olurdu, değil mi?

Simgesel Değişmezlerle Türlere İsim Verilmesi

#define önişlemci komutuyla C'nin doğal veri türlerine de isimler verilebilir:

```
#define BYTE    char
#define BOOL    int

BYTE foo(BYTE b);
BOOL isprime(int val);
```

char türünün aslında *1 byte*'lık bir tamsayı türü olduğunu biliyorsunuz. *char* isminin kullanılması çoğunlukla yazılarla ya da karakterle ilgili bir iş yapıldığı izlenimini verir. Oysa bellek blokları üzerinde genel işlemler yapan işlevler de çoğunlukla *char* türünü kullanır. Bu durumda yapılan işle ilgili daha fazla bir fikir vermek için, örneğin *BYTE* ismi kullanılabilir. C'de *BOOL* türünün olmadığını hatırlıyorsunuz. C'de *bool* veri türü yerine mantıksal bir veri türü olarak *int* türü kullanılır. Ancak programın okunabilirliğini artırmak için *#define* önişlemci komutuyla *BOOL* ismi kullanılabilir.
Bu kullanıma seçenek olan *typedef* anahtar sözcüğünü ve yeni tür ismi tanımlamalarını ileride ele alacağız.

İşlevlerin Simgesel Değişmezlerle Geri Dönmesi

Okunabilirliği artırmaya yönelik bir başka kullanım da işlevlerin geri dönüş değerlerine yöneliktir. Bazı işlevlerin geri dönüş değerlerinin bir soruya yanıt verdiğini, bazı işlevlerin geri dönüş değerlerinin de bir işlemin başarısı hakkında fikir verdiğini biliyorsunuz. Böyle işlevler, geri dönüş değeri ifadeleri yerine simgesel değişmezler kullanırlarsa okunabilirlik açısından daha iyi olabilir:

```
return VALID;
return INVALID;
return TRUE;
return FALSE;
return FAILED;
```

gibi.

İşlevlerin Simgesel Değişmezlerle Çağrılması

Bazı işlevlere de, çağıran kod parçası tarafından simgesel değişmezler gönderilir. C'nin standart başlık dosyalarında da bu amaçla bazı simgesel değişmezler tanımlanmıştır. Örneğin *stdlib.h* başlık dosyası içinde

```
#define EXIT_FAILURE   1
#define EXIT_SUCCESS   0
```

biçiminde tanımlamalar vardır. Yani *stdlib.h* başlık dosyası kaynak koda eklenirse *EXIT_FAILURE* simgesel değişmezi *1, EXIT_FAILURE* simgesel değişmezi, *0* yerine kullanılabilir. Bu simgesel değişmezler, standart *exit* işlevine yapılan çağrılarda kullanılır:

```
exit(EXIT_FAILURE);
```

stdio.h başlık dosyası içinde standart fseek işlevine argüman olarak gönderilmesi amacıyla üç simgesel değişmez tanımlanmıştır:

```
#define SEEK_SET       0
#define SEEK_CUR       1
#define SEEK_END       2
```

Bayrak Değişkenlerin Simgesel Değişmezlerle Değerini Alması

C programlarında, bayrak değişkenleri de çoğunlukla simgesel değişmezlerle değerlerini alır:

```
pos_flag = ON;
validiy_flag = INVALID;
```

switch kontrol deyimindeki *case* ifadeleri de çoğunlukla simgesel değişmezlerle oluşturulur. Bu konuyu *switch* kontrol deyiminde inceleyeceğiz.

Simgesel Değişmezler Yoluyla Programın Değiştirlmesi

Bir değişmezin program içinde pek çok yerde kullanıldığı durumlarda, bu değişmeze yönelik bir değiştirme işlemi tek yerden yapılabilir. Böylece söz konusu program, bir simgesel değişmeze bağlı olarak yazılıp daha sonra simgesel değişmezin değiştirilmesiyle farklı parametrik değerler için yeniden derlenerek çalıştırılabilir. Örneğin kullanıcının belirli sayıda tahmin yaparak bir sayıyı bulmasına dayanan bir oyun programını yazdığımızı düşünelim. Programda, oyuncu *10* tahmin hakkına sahip olsun. Bu durumda kaynak kodun birçok yerinde *10* değeri kullanılmış olur, değil mi? Daha sonra oyun programında oyuncunun tahmin sayısının *20* 'ye çıkarılmak istendiğini varsayalım. Kaynak kod içinde oyuncunun tahmin sayısını gösteren *10* değişmezlerinin değiştirilerek *20* yapılması gerekir. Bu değiştirme işleminin programcı tarafından tek tek yapılması hem zor hem de hataya açıktır. Kaynak kodda kullanılmış olan her *10* değişmezi, oyuncunun tahmin hakkını göstermeyebilir. Oysa oyuncunun hakkını gösteren değer yerine bir simgesel değişmez tanımlanıp

```
#define NO_OF_GUESS    10
```

program bu simgesel değişmez kullanılarak yazılırsa, bu simgesel değişmez tanımında yapılacak değişiklikle tüm program içinde *10* değerleri kolayca *20* değerine dönüştürülebilir.

Gerçek Sayı Değişmezleri Yerine Kullanılan Simgesel Değişmezler

Simgesel değişmezlerin kullanımı, özellikle gerçek sayı değişmezlerin kullanılmasında olası tutarsızlıkları, yazım yanlışlarını engeller.
Örneğin matematiksel hesaplamalar yapan bir programda, *pi* sayısının sık sık kullanıldığını düşünelim. *pi* sayısı yerine

```
#define PI  3.14159
```

simgesel değişmezi kullanılabilir. Her defasında *pi* sayısı, bir değişmez olarak yazılırsa, her defasında aynı değer yazılamayabilir. Örneğin kaynak kodun bir yerinde *3.14159* değişmezi yazılmışken kaynak kodun bir başka noktasında yanlışlıkla *3.15159* gibi bir değer de yazılabilir. Derleyici programın böyle tutarsızlıklar için mantıksal bir uyarı iletmesi olanağı yoktur. Simgesel değişmez kullanımı bu tür hataları ortadan kaldırır. Yine derleyicilerin çoğu, *math.h* başlık dosyası içinde de pek çok matematiksel değişmez tanımlar.

Taşınabilirlik Amacıyla Tanımlanan Simgesel Değişmezler

Bazı simgesel değişmezler hem taşınabilirlik sağlamak hem de ortak arayüz oluşturmak amacıyla tanımlanır. Standart başlık dosyalarından *limits.h* içinde kullanılan tamsayı türlerinin sistemdeki sınır değerlerini taşıyan standart simgesel değişmezler tanımlanmıştır:

Simgesel Değişmez	Olabilecek En Küçük Değer	Anlamı
CHAR_BIT	8	char türündeki bit sayısı
SCHAR_MİN	-127	signed char türünün en küçük değeri
SCHAR_MAX	127	signed char türünün en büyük değeri
UCHAR_MAX	255	unsigned char türünün en büyük değeri

SHRT_MIN	-32.767	signed short int türünün en küçük değeri
SHRT_MAX	32.767	signed short int türünün en büyük değeri
USHRT_MAX	65535	unsigned short türünün en büyük değeri
INT_MIN	-32.767	signed int türünün en küçük değeri
INT_MAX	32.767	signed int türünün en büyük değeri
UINT_MAX	65.535	unsigned int türünün en büyük değeri
LONG_MIN	-2.147.483.648	signed long int türünün en küçük değeri
LONG_MAX	-2.147.483.647	signed long int türünün en büyük değeri
ULONG_MAX	4.294.967.295	unsigned long int türünün en büyük değeri
LLONG_MIN	9.233.372.036.854.775.808	signed long long int türünün en küçük değeri (C99)
LLONG_MAX	9.233.372.036.854.775.807	signed long long int türünün en büyük değeri (C99)
ULLONG_MAX	18.446.744.073.709.551.615	unsigned long long int türünün en büyük değeri (C99)
CHAR_MIN	SCHAR_MIN ya da 0	char türünün en küçük değeri. Sistemdeki char türü işaretliyse bu simgesel değişmezin değeri SCHAR_MIN değerine eşittir. char türü işaretsiz ise UCHAR_MAX değerine eşittir.
CHAR_MAX	SCHAR_MAX ya da UCHAR_MAX	char türünün en büyük değeri. Sistemdeki char türü işaretliyse bu simgesel değişmezin değeri SCHAR_MAX değerine eşittir. char türü işaretsiz ise UCHAR_MAX değerine eşittir.
MB_LEN_MAX	1	Çoklu byte karakterinin sahip olabileceği en fazla byte sayısı. (Bu türün desteklendiği lokallerde)

Simgesel Değişmezlerin Tanımlanma Yerleri

#define komutu kaynak kodun herhangi bir yerinde kullanılabilir. Ancak tanımlandığı yerden kaynak kodun sonuna kadar olan bölge içinde etki gösterir. Önişlemci program doğrudan bilinirlik alanı kavramına sahip değildir. Bir bloğun başında tanımlanan bir simgesel değişmez yalnızca o bloğun içinde değil tanımlandığı yerden kaynak kodun sonuna kadar her yerde etkili olur.
Simgesel değişmezler bazen başlık dosyasının içinde bazen de kaynak dosyanın içinde tanımlanır.

Simgesel Değişmezlerin Kullanılmasında Sık Yapılan Hatalar

Tipk bir hata, simgesel değişmez tanımlamasında gereksiz yere '=' karakterini kullanmaktır:

```
#define N = 100
```

Bu durumda önişlemci *N* gördüğü yere

```
= 100
```

yazısını yapıştırır. Örneğin

```
int a[N];
```

gibi bir tanımlama yapıldığını düşünelim. Önişlemci bu tanımlamayı

```
a[= 100];
```

biçimine getirir ki bu da geçersizdir.

#define önişlemci komutu satırını yanlışlıkla ';' atomu ile sonlandırmak bir başka tipik hatadır.

```
#define N   100;
```

Bu durumda önişlemci *N* gördüğü yere

```
100;
```

yerleştirir.

```
int a[N];
```

tanımlaması

```
int a[100;];
```

haline gelir. Bu tanımlama geçersizdir. Bu tür hatalarda genellikle derleyici, simgesel değişmez kaç yerde kullanılmışsa o kadar hata iletisi verir.

Simgesel değişmezlerin tanımlanmasında dikkatli olunmalıdır. Önişlemci modülünün herhangi bir şekilde aritmetik işlem yapmadığı, yalnızca metinsel bir yer değiştirme yaptığı unutulmamalıdır:

```
#define  MAX   10 + 20

int main()
{
    int result;

    result = MAX * 2;
    printf("%d\n", result);

    return 0;
}
```

Yukarıdaki örnekte *result* değişkenine *60* değil *50* değeri atanır. Ancak önişlemci komutu

```
#define  MAX       (10 + 20)
```

biçiminde yazılsaydı , *result* değişkenine *60* değeri atanmış olurdu.

Standart C İşleçlerine İlişkin Basit Makrolar

Kaynak metnin yazıldığı *ISO 646* gibi bazı karakter setlerinde '&', '|', '^' karakterleri olmadığından, bazı C işleçlerinin yazımında sorun oluşmaktadır.
C89 standartlarına daha sonra yapılan eklemeyle dile katılan *iso646* başlık dosyasında, standart bazı C işleçlerine dönüştürülen basit makrolar tanımlanmıştır. Aşağıda bu makroların listesi veriliyor:

```
#define and      &&
#define and_eq   &=
#define bitand   &
#define bitor    |
#define compl    ~
#define not      !
#define not_eq   !=
#define or       ||
#define or_eq    |=
#define xor      ^
#define xor_eq   ^=
```

switch DEYİMİ

switch deyimi bir tamsayı ifadesinin farklı değerleri için, farklı işlerin yapılması amacıyla kullanılır. *switch* deyimi, özellikle *else if* merdivenlerine okunabilirlik yönünden bir seçenek oluşturur.

Deyimin genel biçimi aşağıdaki gibidir:

```
switch (ifade) {
    case ifade1 :
    case ifade2 :
    case ifade3 :
    .......
    case ifade_n:
    default:
}
```

switch, *case,* ve *default* C dilinin anahtar sözcükleridir.

switch Deyiminin Yürütülmesi

switch ayracı içindeki ifadenin sayısal değeri hesaplanır. Bu sayısal değere eşit değerde bir *case* ifadesi olup olmadığı yukarıdan aşağı doğru sınanır. Eğer böyle bir *case* ifadesi bulunursa programın akışı o *case* ifadesine geçirilir. Artık program buradan akarak ilerler. *switch* ayracı içindeki ifadenin sayısal değeri hiçbir *case* ifadesine eşit değilse, eğer varsa, *default* anahtar sözcüğünün bulunduğu kısma geçirilir.

```
#include <stdio.h>

int main()
{
    int a;

    printf("bir sayi girin : ");
    scanf("%d", &a);
    switch (a) {
        case 1: printf("bir\n");
        case 2: printf("iki\n");
        case 3: printf("üç\n");
        case 4: printf("dört\n");
        case 5: printf("beş\n");
    }
    return 0;
}
```

Yukarıdaki örnekte *scanf* işlevi ile, klavyeden *a* değişkenine *1* değeri alınmış olsun. Bu durumda programın ekran çıktısı şu şekilde olur:

```
bir
iki
üç
dört
beş
```

Eğer uygun *case* ifadesi bulunduğunda yalnızca bu ifadeye ilişkin deyim(ler)in yürütülmesi istenirse *break* deyiminden faydalanılır. *break* deyiminin kullanılmasıyla, döngülerden olduğu gibi *switch* deyiminden de çıkılır. Daha önce verilen örneğe *break* deyimleri ekleniyor:

```
#include <stdio.h>

int main()
{
    int a;

    printf("bir sayi girin: ");
    scanf("%d", &a);
    switch (a) {
        case 1 : printf("bir\n"); break;
        case 2 : printf("iki\n"); break;
        case 3 : printf("üç\n"); break;
        case 4 : printf("dört\n"); break;
        case 5 : printf("beş\n");
    }
    return 0;
}
```

Uygulamalarda, *switch* deyiminde çoğunlukla her *case* ifadesi için bir *break* deyiminin kullanılır. Tabi böyle bir zorunluluk yoktur.
case ifadelerini izleyen ":" atomundan sonra istenilen sayıda deyim olabilir. Bir *case* ifadesini birden fazla deyimin izlemesi durumunda bu deyimlerin bloklanmasına gerek yoktur. Yani bir *case* ifadesini izleyen tüm deyimler, bir blok içindeymiş gibi ele alınır. *case* ifadelerinin belirli bir sırayı izlemesi gibi bir zorunluluk yoktur.

default case

default bir anahtar sözcüktür. *switch* deyimi gövdesine yerleştirilen *default* anahtar sözcüğünü *':'* atomu izler. Oluşturulan bu *case*'e *default case* denir.
Eşdeğer bir *case* ifadesi bulunamazsa programın akışı *default case* içine girer.
Daha önce yazılan *switch* deyimine *default case* ekleniyor.

```
#include <stdio.h>

int main()
{
    int a;

    printf("bir sayi girin: ");
    scanf("%d", &a);
    switch (a) {
        case 1 : printf("bir\n"); break;
        case 2 : printf("iki\n"); break;
        case 3 : printf("üç\n"); break;
        case 4 : printf("dört\n"); break;
        case 5 : printf("dört\n"); break;
        default: printf("hiçbiri\n");
    }
    return 0;
}
```

Yukarıda da anlatıldığı gibi *switch* ayracı içindeki ifadenin sayısal değerine eşit bir *case* ifadesi bulunana kadar derleme yönünde, yani yukarıdan aşağıya doğru, tüm *case* ifadeleri sırasıyla sınanır. *case* ifadelerinin oluşma sıklığı ya da olasılığı hakkında elde bir bilgi varsa, olasılığı ya da sıklığı yüksek olan *case* ifadelerinin daha önce yazılması gereksiz karşılaştırma sayısını azaltabilir.

case ifadelerinin, tamsayı türünden *(integral types)* değişmez ifadesi olması gerekir. Bilindiği gibi değişmez ifadeleri, derleme aşamasında derleyici tarafından net sayısal değerlere dönüştürülebilir:

```
case 1 + 3:         /* Geçerli */
```

mümkün çünkü *1 + 3* değişmez ifadesi ama ,

```
case x + 5:         /* Geçersiz */
```

çünkü değişmez ifadesi değil. Derleyici, derleme aşamasında sayısal bir değer hesaplayamaz.

```
case 'a' :
```

Yukarıdaki *case* ifadesi geçerlidir. 'a' bir karakter değişmezidir. *case* ifadesi tamsayı türünden bir değişmez ifadesidir.

```
case 3.5 :
```

Yukarıdaki *case* ifadesi geçersizdir. *3.5* bir gerçek sayı değişmezidir.

switch kontrol deyimi yerine bir *else if* merdiveni yazılabilir. Yani *switch* deyimi olmasaydı, yapılmak istenen iş, bir *else if* merdiveni ile de yapılabilirdi. Ancak bazı durumlarda *else if* merdiveni yerine *switch* deyimi kullanmak okunabilirliği artırır. Örneğin:

```
if (a == 1)
    deyim1;
else if (a == 2)
    deyim2;
else if (a == 3)
    deyim3;
else if (a == 4)
    deyim4;
else
    deyim5;
```

Yukarıdaki *else if* merdiveni ile aşağıdaki *switch* deyimi işlevsel olarak eşdeğerdir:

```
switch (a) {
    case 1 : deyim1; break;
    case 2 : deyim1; break;
    case 3 : deyim1; break;
    case 4 : deyim1; break;
    default: deyim5;
}
```

Her *switch* deyiminin yerine aynı işi görecek şekilde bir *else if* merdiveni yazılabilir ama her *else if* merdiveni bir *switch* deyimiyle karşılanamaz. *switch* ayracı içindeki ifadenin bir tamsayı türünden olması zorunludur. *case* ifadeleri de tamsayı türlerinden değişmez ifadesi olmak zorundadır. *switch* deyimi, tamsayı türünden bir ifadenin değerinin değişik tamsayı değerlerine eşitliğinin sınanması ve eşitlik durumunda farklı işlerin yapılması için kullanılır. Oysa *else if* merdiveninde her türlü karşılaştırma söz konusu olabilir. Örnek:

```
if (x > 20)
    m = 5;
else if (x > 30 && x < 55)
    m = 3;
else if (x > 70 && x < 90)
    m = 7;
else
    m = 2;
```

Yukarıdaki *else if* merdiveninin yerine bir *switch* deyimi yazılamaz.

switch deyimi bazı durumlarda *else if* merdivenine göre çok daha okunabilir bir yapı oluşturur, yani *switch* deyiminin kullanılması, herşeyden önce, kodun daha kolay okunabilmesini, anlamlandırılmasını sağlar.

Birden fazla *case* ifadesi için aynı işlemlerin yapılması şöyle sağlanabilir.

```
case 1:
case 2:
case 3:
    deyim1;
    deyim2;
    break;
case 4:
```

Bunu yapmanın daha kısa bir yolu yoktur. Bazı programcılar kaynak kodun yerleşimini aşağıdaki gibi düzenlerler:

```
case 1: case 2: case 3: case 4: case 5:
    deyim1; deyim2;
```

Aşağıdaki programı önce inceleyin, sonra derleyerek çalıştırın:

```
void print_season(int month)
{
    switch (month) {
        case 12:
        case 1 :
        case 2 : printf("winter"); break;
        case 3 :
        case 4 :
        case 5 : printf("spring"); break;
        case 6 :
        case 7 :
        case 8 : printf("summer"); break;
        case 9 :
        case 10:
        case 11: printf("autumn");
    }
}
```

print_season işlevi, bir ayın sıra numarasını, yani yılın kaçıncı ayı olduğu bilgisini alıyor, bu ay yılın hangi mevsimi içinde ise, o mevsimin ismini ekrana yazdırıyor. Aynı iş bir *else if* merdiveniyle nasıl yapılabilirdi? Her *if* deyiminin koşul ifadesi içinde mantıksal veya işleci kullanılabilirdi:

```
void print_season(int month)
{
    if (month == 12 || month == 1 || month == 2)
        printf("winter");
    else if (month == 3 || month == 4 || month == 5)
        printf("spring");
    else if (month == 6 || month == 7 || month == 8)
        printf("summer");
    else if (month == 9 || month == 10 || month == 11)
        printf("autumn");
}
```

Simgesel değişmezler, derleme işleminden önce önişlemci tarafından değiştirileceği için, *case* ifadelerinde yer alabilir:

```
#define TRUE              1
#define FALSE             0
#define UNDEFINED         2

case TRUE      :
case FALSE     :
case UNDEFINED :
```

Yukarıdaki *case* ifadeleri geçerlidir.

case ifadeleri olarak karakter değişmezleri de kullanılabilir:

```
#include <stdio.h>

int main()
{
    switch (getchar()) {
    case '0': printf("sıfır\n"); break;
    case '1': printf("bir\n"); break;
    case '2': printf("iki\n"); break;
    case '3': printf("üç\n"); break;
    case '4': printf("dört\n"); break;
    case '5': printf("beş\n"); break;
    default : printf("gecersiz!\n");
    }
    return 0;
}
```

case ifadelerini izleyen deyimlerin *15 - 20* satırdan uzun olması okunabilirliği zayıflatır. Bu durumda yapılacak işlemlerin işlev çağrılarına dönüştürülmesi iyi bir tekniktir.

```
switch (x) {
    case ADDREC:
        addrec();
        break;
    case DELREC:
        delrec();
        break;
    case FINDREC:
        findrec();
        break;
}
```

Yukarıdaki örnekte *case* ifadesi olarak kullanılan *ADDREC*, *DELREC, FINDREC* daha önce tanımlanmış simgesel değişmezlerdir. Her bir *case* için yapılan işlemler, birer işlev içinde sarmalanıyor.

```
char ch = getch();

switch (ch) {
    case 'E' : deyim1; break;
    case 'H' : deyim2; break;
    default  : deyim3;
}
```

Bir *switch* deyiminde aynı sayısal değere sahip birden fazla *case* ifadesi olamaz. Bu durum derleme zamanında hata oluşturur.

switch deyimi, başka bir *switch* deyiminin ya da bir döngü deyiminin gövdesini oluşturabilir:

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

#define    ESC    0X1B

int main()
{
    int ch;

    while ((ch = getch()) != ESC)
        switch (rand() % 7 + 1) {
            case 1: printf("Pazartesi\n"); break;
            case 2: printf("Sali\n"); break;
            case 3: printf("Carsamba\n"); break;
            case 4: printf("Persembe\n"); break;
            case 5: printf("Cuma\n"); break;
            case 6: printf("Cumartesi\n"); break;
            case 7: printf("Pazar\n");
        }
    return 0;
}
```

Yukarıdaki *main* işlevinde *switch* deyimi, dıştaki *while* döngüsünün gövdesini oluşturuyor. *switch* deyimi, döngü gövdesindeki tek bir deyim olduğundan, dıştaki *while* döngüsünün bloklanmasına gerek yoktur. Tabi *while* döngüsünün bloklanması bir hataya neden olmaz.

Ancak *case* ifadeleri içinde yer alan *break* deyimiyle yalnızca *switch* deyiminden çıkılır. *while* döngüsünün de dışına çıkmak için *case* ifadesi içinde *goto* deyimi kullanılabilir.

Şimdi de aşağıdaki programı inceleyin. Programda *display_date* isimli bir işlev tanımlanıyor. İşlev gün, ay ve yıl değeri olarak aldığı bir tarih bilgisini İngilizce olarak aşağıdaki formatta ekrana yazdırıyor:

```
5th Jan 1998
```

```
include <stdio.h>

void display_date(int day, int month, int year)
{
    printf("%d", day);
```

```
    switch (day) {
        case 1  :
        case 21 :
        case 31 : printf("st "); break;
        case 2  :
        case 22 : printf("nd "); break;
        case 3  :
        case 23 : printf("rd "); break;
        default : printf("th ");
    }

    switch (month) {
        case 1 : printf("Jan "); break;
        case 2 : printf("Feb "); break;
        case 3 : printf("Mar "); break;
        case 4 : printf("Apr "); break;
        case 5 : printf("May "); break;
        case 6 : printf("Jun "); break;
        case 7 : printf("Jul "); break;
        case 8 : printf("Aug "); break;
        case 9 : printf("Sep "); break;
        case 10: printf("Oct "); break;
        case 11: printf("Nov "); break;
        case 12: printf("Dec ");
    }
    printf("%d", year);
}
```

```
int main()
{
    int day, month, year;
    int n = 20;

    while (n-- > 0) {
        printf("gun ay  yil olarak bir tarih girin : ");
        scanf("%d%d%d", &day, &month, &year);
        display_date(day, month, year);
        putchar('\n');
    }
    return 0;
}
```

İşlevin tanımında iki ayrı *switch* deyimi kullanılıyor. İlk *switch* deyimiyle, gün değerini izleyen *(th, st, nd, rd)* sonekleri yazdırılırken, ikinci *switch* deyimiyle, aylara ilişkin kısaltmalar (J*an, Feb. Mar.)* yazdırılıyor.
*case i*fadelerini izleyen deyimlerden biri *break* deyimi olmak zorunda değildir. Bazı durumlarda *break* deyimi özellikle kullanılmaz, uygun bir *case* ifadesi bulunduğunda daha aşağıdaki *case* lerin içindeki deyimlerin de yapılması özellikle istenir. Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

int isleap(int y)
{
    return y % 4 == 0 && (y % 100 != 0 || y % 400 == 0);
}

int day_of_year(int day, int month, int year)
{
    int sum = day;
```

```
    switch (month - 1) {
        case 11: sum += 30;
        case 10: sum += 31;
        case 9  : sum += 30;
        case 8  : sum += 31;
        case 7  : sum += 31;
        case 6  : sum += 30;
        case 5  : sum += 31;
        case 4  : sum += 30;
        case 3  : sum += 31;
        case 2  : sum += 28 + isleap(year);
        case 1  : sum += 31;
    }
    return sum;
}

int main()
{
    int day, month, year;
    int n = 5;

    while (n-- > 0) {
        printf("gun ay  yil olarak bir tarih girin : ");
        scanf("%d%d%d", &day, &month, &year);
        printf("%d yilinin %d. gunudur!\n", year, day_of_year(day, month,
year));
    }

    return 0;
}
```

day_of_year işlevi dışarıdan gün, ay ve yıl değeri olarak gelen tarih bilgisinin ilgili yılın kaçıncı günü olduğunu hesaplayarak bu değerle geri dönüyor. İşlev içinde kullanılan *switch* deyimini dikkatli bir şekilde inceleyin. *switch* deyiminin ayracı içinde, dışarıdan gelen ay değerinin *1* eksiği kullanılıyor. Hiçbir *case* içinde bir *break* deyimi kullanılmıyor. Uygun bir *case* ifadesi bulunduğunda, daha aşağıda yer alan tüm *case* içindeki deyimler de yapılır. Böylece, dışarıdan gelen ay değerinden daha düşük olan her bir ayın kaç çektiği bilgisi, gün toplamını tutan *sum* değişkenine katılıyor.

goto DEYİMİ

Diğer programlama dillerinde olduğu gibi C dilinde de programın akışı, bir koşula bağlı olmaksızın kaynak kod içinde başka bir noktaya yönlendirilebilir. Bu, C dilinde *goto* deyimi ile yapılır:

goto deyiminin genel sözdizimi aşağıdaki gibidir:

```
<goto etiket;>
....
<etiket:>
<deyim;>
```

goto, C dilinin *32* anahtar sözcüğünden biridir. Etiket (*label*), programcının verdiği bir isimdir. Şüphesiz isimlendirme kurallarına uygun olarak seçilmelidir. Programın akışı, bu etiketin yerleştirilmiş olduğu yere yönlendirilir. Etiket, *goto* anahtar sözcüğünün kullanıldığı işlev içinde herhangi bir yere yerleştirilebilir. Etiket isminden sonra *':'* atomu yer almak zorundadır. Etiketi izleyen deyim de *goto* kontrol deyiminin sözdiziminin bir parçasıdır. Etiketten sonra bir deyimin yer almaması bir sözdizim hatasıdır.
Etiketin *goto* anahtar sözcüğünden daha sonraki bir kaynak kod noktasına yerleştirilmesi zorunluluğu yoktur. Etiket *goto* anahtar sözcüğünden önce de tanımlanmış olabilir:

```
#include <stdio.h>

int main()
{
    /***/
    goto GIT;
/***/
GIT:
    printf("goto deyimi ile buraya gelindi\n");

    return 0;
}
```

Yukarıdaki programda, etiket goto anahtar sözcüğünden daha sonra yer alıyor.

```
int main()
{
GIT:
    printf("goto deyimi ile gelinecek nokta\n");
/***/
    goto GIT;
    /***/

    return 0;
}
```

Yukarıdaki programda, etiket goto anahtar sözcüğünden daha önce yer alıyor.

goto etiketleri, geleneksel olarak büyük harf ile, birinci sütuna dayalı olarak yazılır. Böylece kaynak kod içinde daha fazla dikkat çekerler.
goto etiketleri bir işlev içinde, bir deyimden önce herhangi bir yere yerleştirilebilir. Yani etiket, aynı işlev içinde bulunmak koşuluyla, *goto* anahtar sözcüğünün yukarısına ya da aşağısına yerleştirilebilir. Bu özelliğiyle *goto* etiketleri, yeni bir bilinirlik alanı kuralı oluşturur. Bir isim, işlev içinde nerede tanımlanırsa tanımlansın o işlev içinde her yerde bilinir. Bu bilinirlik alanı kuralına "işlev bilinirlik alanı" *(function scope)* denir.

goto etiketleri bulunduğu bloğun isim alanına eklenmez. goto etiket isimleri ayrı bir isim alanında değerlendirilir. Bir blok içindeki goto etiketi ile aynı isimli bir yerel değişken olabilir:

```
void func()
{
    int x;
    goto x;
x:
    x = 20;
}
```

Yapısal programlama tekniğinde *goto* deyiminin kullanılması önerilmez. Çünkü *goto* deyiminin kullanılması bir takım sakıncalar doğurur:
1. goto deyimi programların okunabilirliğini bozar. Kodu okuyan kişi *goto* deyimiyle karşılaştığında işlevin içinde etiketi arayıp bulmak zorunda kalır ve programı bu noktadan okumayı sürdürür.

2. *goto* deyimlerinin kullanıldığı bir programda bir değişiklik yapılması ya da programın, yapılacak eklemelerle, geliştirilmeye çalışılması daha zor olur. Programın herhangi bir yerinde bir değişiklik yapılması durumunda, eğer program içinde başka yerlerden değişikliğin yapıldığı yere *goto* deyimleri ile sıçrama yapılmış ise, bu noktalarda da bir değişiklik yapılması gerekebilir. Yani *goto* deyimi program parçalarının birbirine olan bağımlılığını artırır, bu da genel olarak istenen bir şey değildir.

Bu olumsuzluklara karşın, bazı durumlarda *goto* deyiminin kullanılması programın okunabilirliğini bozmak bir yana, diğer seçeneklere göre daha okunabilir bir yapının oluşmasına yardımcı olur:
İçiçe birden fazla döngü varsa, ve içteki döngülerden birindeyken, yalnızca bu döngüden değil, bütün döngülerden birden çıkılmak isteniyorsa *goto* deyimi kullanılmalıdır. Aşağıdaki kod parçasında iç içe üç döngü bulunuyor. En içteki döngünün içinde *func* işlevi çağrılarak işlevin geri dönüş değeri sınanıyor. İşlev eğer *0* değerine geri dönerse programın akışı *goto* deyimiyle tüm döngülerin dışına yönlendiriliyor:

```
#include <stdio.h>

int test_func(int val);

int main()
{
    int i, j, k;

    for (i = 0; i < 100; ++i) {
        for (j = 0; j < 100; ++j) {
            for (k = 0; k < 20; ++k) {
                /*...*/
                if (!test_func(k))
                    goto BREAK;
            /*...*/
            }
        }
    }
BREAK:
    printf("döngü dışındaki ilk deyim\n");
    return 0;
}
```

Yukarıdaki işlev içinde iç içe üç ayrı döngü deyimi yer alıyor. En içteki döngünün içinde çağrılan bir işlev ile bir sınama işlemi yapılmış, sınamanın olumsuz sonuçlanması durumunda, programın akışı en dıştaki döngü deyiminin sonrasına yönlendiriliyor.

Oysa *goto* deyimi kullanmasaydı, ancak bir bayrak *(flag)* değişkenin kullanılmasıyla aynı amaç gerçekleştirilebilirdi. Her döngünün çıkışında bayrak olarak kullanılan değişkenin değerinin değiştirilip değiştirilmediği sınanmak zorunda kalınırdı.

```
#include <stdio.h>

#define     BREAK       0
#define     NO_BREAK    1

int test_func(int val);

int main()
{
    int i, j, k;
    int flag = NO_BREAK;

    for (i = 0; i < 100; ++i) {
        for (j = 0; j < 100; ++j) {
            for (k = 0; k < 20; ++k) {
                /*...*/
                if (!test_func(k)) {
                    flag = BREAK;
                    break;
                }
            /*...*/
            }
            if (flag == BREAK)
                break;
        }
        if (flag == BREAK)
            break;
    }

    printf("döngü dışındaki ilk deyim\n");

    return 0;
}
```

goto deyiminin kullanılması okunabilirlik yönünden daha iyidir.
Aşağıdaki örnekte ise *goto* deyimiyle hem *switch* deyiminden hem de *switch* deyiminin içinde bulunduğu *for* döngüsünden çıkılıyor:

```
#define     ADDREC      1
#define     LISTREC     2
#define     DELREC      3
#define     SORTREC     4
#define     EXITPROG    5

int get_option(void);
void add_rec(void);
void list_rec(void);
void del_rec(void);
void sort_rec(void);

int main()
{
    int option;
```

```
    for (;;) {
        option = get_option();
        switch (option) {
            case ADDREC    :add_rec();break;
            case LISTREC   :list_rec();break;
            case DELREC    :del_rec(); break;
            case SORTREC   :sort_rec(); break;
            case EXITPROG  :goto EXIT;
        }
    }
EXIT:
    return 0;
}
```

Yukarıdaki *main* işlevinde *option* değişkeninin değeri *EXITPROG* olduğunda programın akışı, *goto* deyimiyle sonsuz döngünün dışına gönderiliyor. *goto* deyimi yerine *break* deyimi kullanılsaydı, yalnızca *switch* deyiminden çıkılmış olurdu.
goto deyimiyle, bir işlevin içindeki bir noktadan, yine kendi içindeki bir başka noktaya sıçrama yapılabilir. Böyle sıçramalara yerel sıçramalar *(local jumps)* denir. Bir işlevin içinden başka bir işlevin içine sıçramak başka araçlarla mümkündür. Böyle sıçramalara yerel olmayan sıçramalar *(non-local jumps)* denir. C dilinde, yerel olmayan sıçramalar ismi *setjmp* ve *longjmp* olan standart işlevlerle yapılır. Bu sıçramalar çoğunlukla "Olağan dışı hataların işlenmesi" *(exception handling)* amacıyla kullanılır.

RASTGELE SAYI ÜRETİMİ ve KONTROL DEYİMLERİNE İLİŞKİN GENEL UYGULAMALAR

Rastgele sayı üretimi matematiğin önemli konularından biridir. Rastgele sayılar ya da daha doğru ifadeyle, rastgele izlenimi veren sayılar (*sözde rastgele sayılar - pseudo random numbers*) istatistik, ekonomi, matematik, yazılım gibi pek çok alanda kullanılır.

Rastgele sayılar bir rastgele sayı üreticisi *(random number generator*) tarafından üretilir. Rastgele sayı üreticisi aslında matematiksel bir işlevdir. Söz konusu işlev, bir başlangıç değerini alarak bir değer üretir. Daha sonra üretmiş olduğu her değeri yeni girdi olarak alır, yeniden bir sayı üretir. Üreticinin ürettiği sayılar rastgeledir.

rand İşlevi

Standart *rand* işlevi rastgele sayı üretir. Bu işlevin bildirimi aşağıdaki gibidir:

```
int rand(void);
```

C standartları *rand* işlevinin rastgele sayı üretimi konusunda kullanacağı algoritma ya da teknik üzerinde bir koşul koymamıştır. Bu konu derleyiciyi yazanların seçimine bağlı (*implementation dependent*) bırakılmıştır. *rand* işlevinin bildirimi, standart bir başlık dosyası olan *stdlib.h* içindedir. Bu yüzden *rand* işlevinin çağrılması durumunda bu başlık dosyası *"include"* önişlemci komutuyla kaynak koda eklenmelidir.

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

rand işlevi her çağrıldığında *[0, RAND_MAX]* aralığında rastgele bir tamsayı değerini geri döndürür. *RAND_MAX stdlib.h* başlık dosyası içinde tanımlanan bir simgesel değişmezdir. C standartları bu simgesel değişmezin en az *32767* değerinde olmasını şart koşmaktadır. Derleyicilerin hemen hepsi *RAND_MAX* simgesel değişmezini *32767* olarak, yani *2 byte* işaretli *int* türünün en büyük değeri olarak tanımlar:

```
#define     RAND_MAX    32767
```

Aşağıdaki program parçasında, *0* ile *RAND_MAX* arasında *10* adet rastgele sayı üretilerek ekrana yazdırılıyor. Programı derleyerek çalıştırın:

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

int main()
{
    int k;

    for (k = 0; k < 10; ++k)
        printf("%d ", rand());

    return 0;
}
```

Yukarıdaki kaynak kodla oluşturulan programın her çalıştırılmasında ekrana aynı sayılar yazılır. Örneğin yukarıdaki program, *DOS* altında *Borland Turbo C 2.0* derleyicisi ile derleyip çalıştırıldığında ekran çıktısı aşağıdaki gibi oldu:

```
346 130 10982 1090  11656  7117  17595  6415  22948  31126
```

srand İşlevi

Oluşturulan program her çalıştırıldığında neden hep aynı sayı zinciri elde ediliyor? *rand* işlevi rastgele sayı üretmek için bir algoritma kullanıyor. Bu algoritma derleyiciden derleyiciye değişse de, rastgele sayı üretiminde kullanılan ana tema aynıdır. Bir başlangıç değeri ile işe başlanır. Buna tohum değeri (*seed value*) denir. Bu değer üzerinde bazı işlemler yapılarak rastgele bir sayı elde edilir. Tohum değer üzerinde yapılan işlem bu kez elde edilen rastgele sayı üzerinde yinelenir.
rand işlevi çağrılarını içeren bir program her çalıştırıldığında aynı tohum değerinden başlanacağı için aynı sayı zinciri elde edilir.

Bir başka standart işlev olan *srand* işlevi, rastgele sayı üreticisinin tohum değerini değiştirmeye yarar. *srand* işlevinin *stdlib.h* başlık dosyasında yer alan bildirimi aşağıdaki gibidir:

```
void srand (unsigned seed);
```

srand işlevine gönderilen değer, işlev tarafından rastgele sayı üreticisinin tohum değeri yapılır. *srand* işlevine argüman olarak başka bir tohum değeri gönderildiğinde işlevin ürettiği rastgele sayı zinciri değişir.
Aşağıda *rand* ve *srand* işlevleri tanımlanıyor:

```
#define     RAND_MAX    32767

unsigned long int next = 1;

int rand()
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next / 65536) % 32768;
}

void srand(unsigned int seed)
{
    next = seed;
}
```

srand işlevi çağrılmaz ise başlangıç tohum değeri *1*'dir.
Yukarıdaki programa *srand* işlevi çağrısını ekleyerek yeniden derleyin, çalıştırın:

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

int main()
{
    int k;

    srand(100);

    for (k = 0; k < 10; ++k)
        printf("%d ", rand());

    return 0;
}
```

Program bu şekliyle *DOS* altında *Borland Turbo C 2.0* derleyicisi ile derleyip çalıştırıldığında ekran çıktısı aşağıdaki gibi oldu:

```
1862  11548  3973  4846  9095  16503  6335  13684  21357  21505
```

Ancak bu kez oluşturulan program da her çalıştırıldığında yine yukarıdaki sayı zinciri elde edilir, değil mi? *rand* işlevinin kullanmakta olduğu önceden seçilmiş *(default)* tohum değeri kullanılmasa da, bu kez her defasında *srand* işlevine gönderilmiş olan tohum değeri kullanılır. Programı birkaç kere çalıştırıp gerçekten hep aynı sayı zincirinin üretilip üretilmediğini görün.

Bazı durumlarda, programın her çalıştırılmasında aynı rastgele sayı zincirinin üretilmesi istenmez. Örneğin bir oyun programında programın çalıştırılmasıyla hep aynı sayılar üretilirse, oyun hep aynı biçimde oynanır. Programın her çalışmasında farklı bir sayı zincirinin elde edilmesi için, *srand* işlevinin rastgele sayı üreticisinin tohum değerini programın her çalışmasında başka bir değer yapması gerekir. Bu amaçla çoğu zaman standart *time* işlevi işlevinden faydalanılır.

time standart bir C işlevidir, bildirimi standart bir başlık dosyası olan *time.h* dosyası içindedir. Parametre değişkeni gösterici olan *time* işlevini, ancak ileride ayrıntılı olarak ele alacağız. Şimdilik *time* işlevini işimizi görecek kadar inceleyeceğiz. *time* işlevi kendisine *0* değeri gönderildiğinde, önceden belirlenmiş bir tarihten (sistemlerin çoğunda *01.01.1970* tarihinden) işlevin çağrıldığı ana kadar geçen saniye sayısını geri döndürür. İşlevin geri dönüş değeri, derleyicilerin çoğunda *long* türden bir değerdir. İçinde rastgele sayı üretilecek programda, *srand* işlevine argüman olarak *time* işlevinin geri dönüş değeri gönderilirse, program her çalıştığında, belirli bir zaman geçmesi nedeniyle, rastgele sayı üreticisi başka bir tohum değeriyle ilkdeğerini alır. Böylece programın her çalıştırılmasında farklı sayı zinciri üretilir:

```
srand(time(0));
```

srand işlevine yapılan bu çağrı, derleyicilerin çoğunda standart olmayan *randomize* isimli bir makro olarak tanımlanmıştır:

```
randomize();
```

Yukarıdaki işlev çağrısı yerine bu makro da kullanılabilir. Makrolar konusu ileride ayrıntılı olarak ele alınacak.

Yukarıdaki daha önce yazılan örnek programı her çalıştığında farklı sayı zinciri üretecek duruma getirelim:

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

int main()
{
    int k;

    srand(time(0));
    for (k = 0; k < 10; ++k)
        printf("%d ", rand());

    return 0;
}
```

Programlarda bazen belirli bir aralıkta rastgele sayı üretilmesi istenir. Bu amaçla kalan işleci kullanılabilir. Aşağıdaki ifadeleri inceleyin:

```
rand() % 2
```

Yalnızca *0* ya da *1* değerini üretir.

```
rand() % 6
```

0 - 5 aralığında rastgele bir değer üretir

```
rand() % 6 + 1
```

1 - 6 aralığında rastgele bir değer üretir. (örneğin bir zar değeri)

```
rand() % 6 + 3
```

3 - 8 aralığında rastgele bir değer üretir.

Ancak derleyici programların sağladığı rastgele sayı üreticilerinin ürettikleri rastgele sayıların, düşük anlamlı bitleri çoğunlukla rastgele kabul edilemez. Bu durumda yukarıdaki ifadeler, üretilmesi gereken tüm sayılar için eşit bir dağılım sağlamaz. Dağılımın daha düzgün olabilmesi için bazı yöntemler kullanılabilir:

```
rand() % N
```

ifadesi yerine

```
rand()/(RAND_MAX / N + 1)
```

ya da

```
(int)((double)rand() / ((double)RAND_MAX + 1) * N)
```

ifadeleri yazılabilir.
Ya da aşağıdaki gibi bir işlev tanımlanabilir:

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

#define  N     10

int mrand()
{
    unsigned int x = (RAND_MAX + 1u) / N;
    unsigned int y = x * N;
    unsigned int r;

    while ((r = rand()) >= y)
        ;
    return r / x;
}
```

srand(time(0)) çağrısının bir döngü içinde yer alması sık yapılan bir hatadır.

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

int zar_at()
{
    srand(time(0));
    return  rand() % 6 + 1 +  rand() % 6 + 1;
}
```

```
int main()
{
    int k;

    for (k = 0; k < 10; ++k)
        printf("%d\n", zar_at());

    return 0;
}
```

Yukarıda yazılan programda yer alan *zar_at* isimli işlev, bir çift zar atıldığında elde iki zarın toplamı değeriyle geri dönüyor. *srand(time(0))* çağrısı *zar_at* işlevi içinde yapılıyor. *main* işlevi içinde oluşturulan *for* döngüsüyle *10* kez *zar_at* işlevi çağrılıyor. İşlevin her çağrısında *time* işlevi hep aynı geri dönüş değerini üretir. Bu durumda *srand* işlevine hep aynı argüman geçildiğinden *rand* işlevi çağrıları da hep aynı iki sayıyı üretir. Yani ekrana *10* kez aynı değer yazdırılır. *srand(time(0))* çağrısının *main* işlevi içindeki *for* döngüsünden önce yapılması gerekirdi, değil mi?

Aşağıdaki *main* işlevinde uzunlukları *3 - 8* harf arasında değişen İngiliz alfabesindeki harfler ile oluşturulmuş rastgele *10* sözcük ekrana yazdırılıyor:

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

#define    TIMES                10
#define    MIN_WORD_LEN         3
#define    MAX_WORD_LEN         8

void write_word(void)
{
    int len = rand() % (MAX_WORD_LEN - MIN_WORD_LEN + 1) + MIN_WORD_LEN;

    while (len--)
        putchar('A' + rand() % 26);
}

int main()
{
    int k;

    srand(time(0));
    for (k = 0; k < TIMES; ++k) {
        write_word();
        putchar('\n');
    }
    return 0;
}
```

Aşağıda bu kez yazdırılan sözcüklerin içinde sesli harf olmaması sağlanıyor:

```
int isvowel(int c)
{
    return c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U';
}

void write_word(void)
{
    int len = rand() % (MAX_WORD_LEN - MIN_WORD_LEN + 1) + MIN_WORD_LEN;
    int ch;
```

```
    while (len--) {
        while (isvowel(ch = rand() % 26 + 'A'))
            ;
        putchar(ch);
    }
}
```

Aşağıda rastgele bir tarihi ekrana yazdıran *print_random_date* isimli bir işlev tanımlanıyor. İşlev her çağrıldığında *1.1.MIN_YEAR*, *31.12.MAX_YEAR* tarihleri arasında rastgele ancak geçerli bir tarih bilgisini ekrana yazıyor:

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

#define    MAX_YEAR   2010
#define    MIN_YEAR   1900

void print_random_date()
{
    int d, m, y;

    y = rand() % (MAX_YEAR - MIN_YEAR + 1) + MIN_YEAR;
    m = rand() % 12 + 1;
    switch (m) {
        case 4 : case 6 : case 9 : case 11:
             d = rand() % 30 + 1; break;
        case 2 : d = rand() % (isleap(y) ? 29 : 28) + 1; break;
        default: d = rand() % 31 + 1;
    }
    printf("%d/%d/%d\n", d, m, y);
}

int isleap(int y)
{
    return y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
}

int main()
{
    int k;

    srand(time(0));
    for (k = 0; k < 20; ++k)
        print_random_date();

    return 0;
}
```

Olasılık problemleri, olasılığa konu olayın bir bilgisayar programı ile gerçekleştirilmesi yoluyla çözülebilir. İyi bir rastgele sayı üreticisi kullanıldığı takdirde, olasılığa konu olay, bir bilgisayar programı ile oynatılır, olay bilgisayarın işlem yapma hızından faydalanılarak yüksek sayılarda yinelemeye sokulur. Şüphesiz hesaplanmak istenen olaya ilişkin olasılık değeri, yapılan yineleme sayısına ve rastgele sayı üreticisinin niteliğine bağlı olur. Aşağıdaki kod yazı tura atılması olayında tura gelme olasılığını hesaplıyor:

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

#define     TIMES       30000
#define     HEADS           1

int main()
{
    int heads_counter = 0;
    int k;

    srand(time(0));
    for (k = 0; k < TIMES; ++k)
        if (rand() % 2 == HEADS)
            heads_counter++;
    printf("tura gelme olasılığı = %lf", (double) heads_counter / TIMES);

    return 0;
}
```

Yukarıdaki program *TIMES* simgesel değişmezinin farklı değerleri için çalıştırıldığında ekran çıktısı aşağıdaki şekilde oldu:

```
#define     TIMES   100
tura gelme olasılığı = 0.480000
#define     TIMES   500
tura gelme olasılığı = 0.496000
#define     TIMES   2500
tura gelme olasılığı = 0.506800
#define     TIMES   10000
tura gelme olasılığı = 0.503500
#define     TIMES   30000
tura gelme olasılığı = 0.502933
#define     TIMES   100000
tura gelme olasılığı = 0.501450
#define     TIMES   1000000
tura gelme olasılığı = 0.500198
```

Aşağıda bir başka olasılık çalışması yapılıyor:
Craps hemen hemen dünyanın her yerinde bilinen, iki zarla oynanan bir kumardır.
Oyunun kuralları şöyledir :
Zarları atacak oyuncu oyunu kasaya karşı oynar. Atılan iki zarın toplam değeri 7 ya da 11 ise oyuncu kazanır. Atılan iki zarın toplam değeri 2, 3, 12 ise oyuncu kaybeder. (Buna craps denir!)
İki zarın toplam değeri yukarıdakilerin dışında bir değer ise (yani 4, 5, 6, 8, 9, 10) oyun şu şekilde sürer :
Oyuncu aynı sonucu buluncaya kadar zarları tekrar atar. Eğer aynı sonucu bulamadan önce oyuncu 7 atarsa (yani atılan iki zarın toplam değeri 7 olursa) oyuncu kaybeder.
Eğer 7 gelmeden önce oyuncu aynı sonucu tekrar atmayı başarırsa , kazanır.
Birkaç örnek :

Oyuncunun attığı zarlar	Oyun sonucu
11	Oyuncu kazanır
3	Oyuncu kaybeder
9 8 6 3 12 5 8 4 2 4 9	Oyuncu kazanır
6 5 8 9 2 3 7	Oyuncu kaybeder
7	Oyuncu kazanır
10 4 8 11 8 3 6 5 4 9 10	Oyuncu kazanır

Aşağıdaki program, bu oyunu oynayan oyuncunun kazanma olasılığını hesaplıyor:

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

#define    NKEZ       1000000

int zar_at()
{
    int zar1 = rand() % 6 + 1;
    int zar2 = rand() % 6 + 1;
    return zar1 + zar2;
}

/* oyuncu kazanırsa 1 değerine, oyuncu kaybederse 0 değerine geri döner */
int oyun()
{
    int zar_toplam;

    zar_toplam = zar_at();
    switch (zar_toplam) {
        case 7 :
        case 11: return 1;
        case 2 :
        case 3 :
        case 12: return 0;
    }
    return oyun_devami(zar_toplam);
}

/* oyuncu 4, 5, 6, 8, 9, 10 atmissa oyunun devamı.
oyuncu kazanırsa 1 değerine, oyuncu kaybederse 0 değerine geri döner */
int oyun_devami(int zar_toplam)
{
    int yeni_zar;

    for (;;) {
        yeni_zar = zar_at();
        if (yeni_zar == zar_toplam)
            return 1;
        if (yeni_zar == 7)
            return 0;
    }
}

int main()
{
    int k;
    int kazanma_sayisi = 0;

    srand(time(0));

    for (k = 0; k < NKEZ; ++k)
        kazanma_sayisi += oyun();
    printf("kazanma olasiligi = %lf\n", (double)kazanma_sayisi / NKEZ);

    return 0;
}
```

Rastgele Gerçek Sayı Üretimi

Rastgele gerçek sayı üreten bir standart C işlevi yoktur. Ancak *RAND_MAX* simgesel değişmezinden faydalanarak

```
(double)rand() / RAND_MAX
```

ifadesi ile *0 – 1* aralığında rastgele bir gerçek sayı üretilebilir. Aşağıda rastgele gerçek sayı üreten *drand* isimli bir işlev tanımlanıyor. İşlevi inceleyerek, rastgele bir gerçek sayıyı nasıl ürettiğini anlamaya çalışın:

```
#define    PRECISION      2.82e14

double drand()
{
    double sum = 0;
    double denom = RAND_MAX + 1;
    double need;

    for (need = PRECISION; need > 1; need /= (RAND_MAX + 1.)) {
        sum += rand() / denom;
        denom *= RAND_MAX + 1.;
    }
    return sum;
}

int main()
{
    int k;

    for (k = 0; k < 10; ++k)
        printf("%lf\n", drand());
    return 0;
}
```

Aşağıda *pi* sayısı *Monte Carlo* yöntemi diye bilinen yöntemle bulmaya çalışılıyor. Bu yöntemde birim kare içinde yarıçapı karenin kenar uzunluğuna eşit bir daire parçası olduğu düşünülür. Birim kare içinde rastgele alınan bir nokta, ya daire parçasının içinde ya da dışında olur. Rastgele alınan bir noktanın daire parçasının içinde olma olasılığı, yarıçapı *1* birim olan bir dairenin alanının dörtte birinin, kenarı *1* birim olan karenin alanına oranıdır. Bu da $\frac{p}{4}$ değerine eşittir. O zaman *n* tane rastgele nokta alıp bu noktaların kaç tanesinin dairenin içinde olduğunu bulursak, bu değerin n sayısına oranının *4* katı p sayısını verir:

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

#define    NTIMES     10000000

int main()
{
    double x, y;
    int k;
    int inside_counter = 0;

    srand(time(0));
    for (k = 0; k < NTIMES; ++k) {
        x = (double)rand() / RAND_MAX;
```

```
        y = (double)rand() / RAND_MAX;
        if (x * x + y * y <= 1)
            inside_counter++;
    }
    printf("hesaplanan pi degeri = %lf\n", 4. * inside_counter / NTIMES);

    return 0;
}
```

DİZİLER

Veri Yapısı Nedir
Bir konuyla ilgili, mantıksal ilişki içindeki verilerin bellekte saklanmasına yönelik düzenlemelere veri yapısı denir. Veri yapıları bellekte belirli bir düzen içinde tutulmuş verilere ulaşılabilmesine, bu veriler üzerinde bazı işlemlerin etkin bir biçimde yapılmasına olanak sağlar.

Dizi Nedir
Bellekte bitişik bir biçimde bulunan, aynı türden nesnelerin oluşturduğu veri yapısına dizi (*array)* denir. Dizi veri yapısının en önemli özelliği, mantıksal bir ilişki içindeki aynı türden verilerin bellekte bitişik (*contigous*) olarak tutulmasıdır. Bunun da uygulamalarda sağladığı fayda şudur: Dizinin bir elemanına, elemanın konum bilgisiyle değişmez bir zamanda ulaşılabilir. Yani dizinin eleman sayısı ne olursa olsun, konumu bilinen bir elemana ulaşım zamanı aynıdır. Bu da bazı uygulamaların etkin bir şekilde gerçekleştirilmesini kolaylaştırır.

C Dilinde Diziler
C dilinde dizi *(array)*, aynı türden bir ya da daha fazla nesnenin bellekte dizi veri yapısı biçiminde tutulmasını sağlayan araçtır.
C'de bir dizinin tanımlanmasıyla birden fazla sayıda nesne tek bir deyimle tanımlanabilir. *10* elemana sahip bir dizi tanımlamak yerine, şüphesiz isimleri farklı *10* ayrı nesne de tanımlanabilir. Ama *10* ayrı nesne tanımlandığında bu nesnelerin bellekte bitişik olarak yerleşmeleri güvence altına alınmış bir özellik değildir. Oysa dizi tanımlamasında, dizinin elemanı olan bütün nesnelerin bellekte bitişik olarak yer almaları güvence altına alınmış bir özelliktir. Dizi de bir veri türü olduğuna göre, dizilerin de kullanılmalarından önce tanımlanmaları gerekir.

Dizilerin Tanımlanması
Dizi tanımlamalarının genel biçimi:

```
<tür> <dizi ismi> [<eleman sayısı>];
```

Yukarıdaki genel biçimde köşeli ayraç, eleman sayısının seçimlik olduğunu değil, eleman sayısı bilgisinin köşeli ayraç içine yazılması gerektiğini gösteriyor.

tür	: Dizi elemanlarının türünü gösteren anahtar sözcüktür.
dizi ismi	: İsimlendirme kurallarına uygun olarak verilecek herhangi bir isimdir.
eleman sayısı	: Dizinin kaç elemana sahip olduğunu gösterir.

Örnek dizi bildirimleri:

```
double a[20];
int ave[10];
char path[80];
```

Yukarıdaki tanımlamalarda
a, 20 elemanlı, her bir elemanı *double* türden olan bir dizidir.
ave, 10 elemanlı, her bir elemanı *int* türden olan bir dizidir.
path, 10 elemanlı, her bir elemanı *char* türden olan bir dizidir.

Tanımlamada yer alan, eleman sayısı belirten ifadenin bir tamsayı türünden değişmez ifadesi olması zorunludur. Bir başka deyişle derleyici bu ifadenin değerini derleme zamanında elde edebilmelidir:

```
int x = 100;
int a[x];       /* Geçersiz */
int b[5.];      /* Geçersiz */
int c[10 * 20];
int d[sizeof(int) * 100];
```

Yukarıdaki deyimlerden *a* ve *b* dizilerinin tanımlamaları geçersizdir. *a* dizisinin tanımında *b*oyut belirten ifade olarak değişmez ifadesi olmayan bir ifade kullanılıyor. *b* dizisinin tanımında ise boyut belirten ifade bir gerçek sayı türündendir. *c* dizisinin tanımında ise bir hata söz konusu değildir. *10 * 20* bir değişmez ifadesidir. *d* dizisinin tanımı da bir hata oluşturmaz çünkü *sizeof* işlecinin ürettiği değer derleme zamanında elde edilir.

Dizi bildirimlerinde eleman sayısını belirten ifade yerine sıklıkla simgesel değişmezler kullanılır:

```
#define ARRAY_SIZE  100

int a[ARRAY_SIZE];
```

Program içinde dizi boyutu yerine hep *ARRAY_SIZE* simgesel değişmezi kullanılabilir. Böylece programda daha sonra dizi boyutuna ilişkin bir değişiklik yapılmak istendiğinde, yalnızca simgesel değişmezin değerinin değiştirilmesi yeterli olur.
Diğer değişken bildirimlerinde olduğu gibi, virgül ayracıyla ayrılarak, birden fazla dizi, tür belirten sözcüklerin bir kez kullanılmasıyla tanımlanabilir:

```
int x[100], y[50], z[10];
```

x, *y* ve *z,* elemanları *int* türden olan dizilerdir.

Diziler ve diğer nesneler türleri aynı olmak kaydıyla tek bir tanımlama deyimiyle tanımlanabilir:

```
int a[10], b, c;
```

a int türden *10* elemanlı bir dizi, *b* ve *c int* türden nesnelerdir.

Dizi elemanlarının her biri ayrı birer nesnedir. Dizi elemanlarına *köşeli ayraç* işleciyle *[]* ulaşılabilir. Köşeli ayraç işleci bir gösterici işlecidir. Bu işleç "Göstericiler" konusunda ayrıntılı bir şekilde ele alınacak.
Köşeli ayraç işlecinin terimi dizi ismidir. Aslında bu bir adres bilgisidir, çünkü bir dizi ismi işleme sokulduğunda, işlem öncesi derleyici tarafından otomatik olarak dizinin ilk elemanının adresine dönüştürülür. Köşeli ayraç içinde dizinin kaçıncı indisli elemanına ulaşılacağını gösteren bir tamsayı ifadesi olmalıdır.
C dilinde bir dizinin ilk elemanı, dizinin sıfır indisli elemandır.

T bir tür bilgisi olmak üzere

```
T a[SIZE];
```

gibi bir dizinin ilk elemanı *a[0]* son elemanı ise *a[SIZE - 1]*'dir. Örnekler:

```
dizi[20]    /* a dizisinin 20 indisli yani 21. elemanı olan nesne */
ave[0]      /* ave dizisinin 0 indisli yani birinci elemanı olan nesne */
total[j]    /* total dizisinin j indisli elemanı olan nesne*/
```

Görüldüğü gibi "bir dizinin *n.* elemanı" ve "bir dizinin *n* indisli elemanı" terimleri dizinin farklı elemanlarını belirtir. Bir dizinin *n* indisli elemanı o dizinin *n + 1* . elemanıdır.

Bir dizi tanımlaması ile karşılaşan derleyici, tanımlanan dizi için bellekte yer ayırır. Ayrılacak yer şüphesiz

```
dizinin eleman sayısı * bir elemanın bellekte kapladığı yer
```

kadar *byte* olur. Örneğin:

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

gibi bir dizi tanımlaması yapıldığını düşünelim. *Windows* işletim sisteminde çalışılıyorsa derleyici a dizisi için bellekte *4 * 5 = 20 byte* yer ayırır.

Dizi indis ifadelerinde ++ ya da -- işleçleri sık kullanılır:

```
int a[20];
int k = 10;
int i = 5;

a[k++] = 100;
```

deyimiyle dizinin *10* indisli elemanına yani dizinin *11.* elemanına *100* değeri atanıyor. Daha sonra *k* değişkeninin değeri *1* artırılarak *11* yapılıyor.

```
a[--i] = 200;
```

deyimiyle dizinin *4* indisli elemanına yani dizinin *5.* elemanına *200* değeri atanıyor. Daha sonra *i* değişkeninin değeri *1* azaltılarak *4* yapılıyor.

Köşeli ayraç işlecinin kullanılmasıyla artık dizinin herhangi bir elemanı diğer değişkenler gibi kullanılabilir. Aşağıdaki örnekleri inceleyin:

```
a[0] = 1;
```

a dizisinin ilk elemanına *1* değeri atanıyor.

```
printf("%d\n", b[5]);
```

b dizisinin *6*. elemanının değeri ekrana yazdırılıyor:

```
++c[3];
```

c dizisinin *4*. elemanının değeri *1* artırılıyor:

```
d[2] = e[4];
```

d dizisinin *3.* elemanına *e* dizisinin *5.* elemanı atanıyor:

Diziler üzerinde işlem yapmak için sıklıkla döngü deyimleri kullanılır. Bir döngü deyimi yardımıyla bir dizinin tüm elemanlarına ulaşmak, çok karşılaşılan bir durumdur.

Aşağıda *SIZE* elemanlı *a* isimli bir dizi için *for* ve *while* döngü deyimlerinin kullanıldığı bazı kalıplar gösteriliyor:

a dizisinin bütün elemanlarına *0* değeri atanıyor:

```
for (i = 0; i < SIZE; ++i)
     a[i] = 0;
```

Aynı iş bir şüphesiz bir *while* döngü deyimiyle de yapılabilirdi:

```
i = 0;
while (i < SIZE)
     a[i++] = 0;
```

Aşağıda *a* dizisinin elemanlarına standart *scanf* işleviyle standart giriş biriminden değer alınıyor:

```
for (i = 0; i < SIZE; i++)
     scanf("%d", &a[i]);
```

ya da

```
i = 0;
while (i < SIZE)
     scanf("%d", &a[i++]);
```

Aşağıda *a* dizisinin elemanlarının toplamı hesaplanıyor:

```
for (total = 0, i = 0; i < SIZE; i++)
     total += a[i];
```

ya da

```
total = 0;
i = 0;

while (i < SIZE)
    total += a[i++];
```

Dizilerin Taşırılması

Bir dizi tanımlamasını gören derleyici dizi için bellekte dizinin tüm elemanlarının sığacağı büyüklükte bir alan ayırır:

```
double a[10];
```

Gibi bir tanımlama yapıldığında, çalışılan sistemde *double* türünün bellekte 8 *byte* yer kapladığı var sayılırsa, dizi için bellekte bitişik (*contiguous*) toplam *80 byte*'lık bir yer ayrılır.
Dizinin son elemanı *a[9]* olur. Çok sık yapılan bir hata, dizinin son elemanına ulaşmak amacıyla yanlışlıkla bellekte derleyici tarafından ayrılmamış bir yere değer atamak, yani diziyi taşırmaktır:

```
a[10] = 5.;
```

deyimiyle bellekte ne amaçla kullanıldığı bilinmeyen *8 byte*'lık bir alana, yani güvenli olmayan bir bellek bölgesine değer aktarma girişiminde bulunulur. Dizi taşmaları derleme zamanında kontrol edilmez. Böyle hatalar programın çalışma zamanı ile ilgilidir.

Dizilere İlkdeğer Verilmesi

Değişken tanımlamalarında tanımlanan bir değişkenin "ilkdeğer verme sözdizimi" diye isimlendirilen bir kural ile belirli bir değerle başlatılması sağlanabiliyordu.
Tanımlanan dizilere de ilkdeğer verilebilir:

```
double sample[5] = {1.3, 2.5, 3.5, 5.8, 6.0};
char str[4] = {'d', 'i', 'z', 'i'};
unsigned a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Dizilere yukarıdaki gibi ilkdeğer verildiğinde, verilen değerler dizinin ilk elemanından başlayarak dizi elemanlarına sırayla atanmış olur.
Dizilerin tüm elemanlarına ilkdeğer verme zorunluluğu yoktur. Dizinin eleman sayısından daha az sayıda elemana ilkdeğer verilmesi durumunda kalan elemanlara *0* değeri atanmış olur. Bu kural hem yerel hem de global diziler için geçerlidir.
Bu durumda bir dizinin bütün elemanlarına *0* değeri verilmek isteniyorsa bunun en kısa yolu aşağıdaki gibidir:

```
int a[20] = {0};
```

Yalnızca dizinin ilk elemanına *0* ilkdeğeri veriliyor. Bu durumda derleyici dizinin kalan elemanlarına otomatik olarak *0* değeri yerleştirecek kodu üretir.

Dizi elemanlarına ilkdeğer verilmesinde kullanılan ifadeler, değişmez ifadeleri (*constant expression)* olmalıdır.

```
int a[10] = {b, b + 1, b + 2};     /* Geçersiz */
```

gibi bir ilkdeğer verme işlemi geçersizdir.

[Yukarıdaki tanımlama C++ dilinin kurallarına uygundur. C++ dilinde dizi elemanlarına değişmez ifadeleriyle ilkdeğer vermek zorunlu değildir]

Bir diziye ilkdeğer verme işleminde, dizi eleman sayısından daha fazla sayıda ilkdeğer vermek geçersizdir:

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

Yukarıdaki örnekte *b* dizisi *5* elemanlı olmasına karşın, ilkdeğer verme deyiminde *6* değer kullanılıyor. Bu durum derleme zamanında hata oluşturur.

İlkdeğer verme işleminde dizi boyutu belirtilmeyebilir. Bu durumda derleyici dizi uzunluğunu, verilen ilkdeğerleri sayarak kendi hesaplar. Dizinin o boyutta açıldığını kabul eder. Örneğin:

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

Derleyici yukarıdaki deyimi gördüğünde *a* dizisinin *5* elemanlı olduğunu kabul eder. Bu durumda yukarıdaki gibi bir bildirimle aşağıdaki gibi bir bildirim eşdeğerdir:

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

Başka örnekler :

```
char name[] = {'B' 'E ', 'R', 'N', 'A', '\0'};
unsigned short count[ ] = {1, 4, 5, 7, 8, 9, 12, 15, 13, 21};
```

Derleyici *name* dizisinin boyutunu *6*, *count* dizisinin boyutunu ise *10* olarak varsayar.
Diziye ilkdeğer verme listesi bir virgül atomuyla sonlandırılabilir:

```
int a[] = { 1, 4, 5, 7, 8, 9, 12, 15, 13,
            2, 8, 9, 8, 9, 4, 15, 18, 25,
          };
```

Yerel ve Global Diziler

Bir dizi de diğer nesneler gibi yerel ya da global olabilir. Yerel diziler blokların içinde tanımlanan dizilerdir. Global diziler ise global isim alanında, yani tüm blokların dışında tanımlanır. Global bir dizinin tüm elemanları, global nesnelerin özelliklerine sahip olur.

Yani dizi global ise, dizi elemanı olan nesneler dosya bilinirlik alanına (*file scope*) ve statik ömre (*static storage duration*) sahip olurlar. Global bir dizi söz konusu olduğunda eğer dizi elemanlarına değer verilmemişse, dizi elemanları *0* değeriyle başlatılır. Ama yerel diziler söz konusu olduğunda, dizi elemanı olan nesneler blok bilinirlik alanına (*block scope*) ömür açısından ise otomatik ömür karakterine (*automatic storage class*) sahip olur. Değer atanmamış dizi elemanları içinde çöp değerler (*garbage values*) bulunur. Aşağıdaki programı yazarak derleyin:

```
#include <stdio.h>

#define            SIZE       10
int g[SIZE];

int main()
{
    int y[SIZE];
    int i;

    for (i = 0; i < SIZE; ++i)
        printf("g[%d] = %d\n", i, g [i]);

    for (i = 0; i < SIZE; ++i)
        printf("y[%d] = %d\n", i, y [i]);

    return 0;
}
```

Dizilerin Birbirine Atanması

Dizilerin elemanları nesnedir. Ancak bir dizinin tamamı bir nesne olarak işlenemez:

```
int a[SIZE], b[SIZE];
```

gibi bir tanımlamadan sonra, *a* dizisi elemanlarına *b* dizisinin elemanları kopyalanmak amacıyla, aşağıdaki gibi bir deyimin yazılması sözdizim hatasıdır.

```
a = b;      /* Geçersiz */
```

Yukarıdaki gibi bir atama derleme zamanı hatasına neden olur. Çünkü dizilerin isimleri olan *a* ve *b* nesne göstermez. Dizinin bellekte kapladığı toplam alan doğrudan tek bir nesne olarak işlenemez. Yani dizinin elemanları birer nesnedir ama dizinin tamamı bir nesne değildir. C'de dizi isimleri dizilerin bellekte yerleştirildikleri bloğun başlangıcını gösteren, dizinin türü ile aynı türden adres değerleridir. Dolayısıyla değiştirilebilir sol taraf değeri (*modifiable L value)* değillerdir.
İki dizi birbirine ancak bir döngü deyimi ile kopyalanabilir:

```
for (i = 0; i < SIZE; ++i)
    a[i] = b[i];
```

Yukarıdaki döngü deyimiyle *b* dizisinin her bir elemanının değeri *a* dizisinin eş indisli elemanına atanıyor. Dizilerin kopyalanması için başka bir yöntem de bir standart C işlevi olan *memcpy* işlevini kullanmaktır. Bu işlev "göstericiler" konusunda ele alınacak.

Dizilerin Kullanımına İlişkin Örnekler

Aynı türden nesneler bir dizi altında tanımlanırlarsa, bir döngü deyimi yardımıyla dizi elemanlarının tamamını işleme sokan kodlar kolay bir biçimde yazılabilir. Aşağıdaki örneği dikkatle inceleyin:

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

#define SIZE       10

int main()
{
    int a[SIZE];
    int toplam = 0;
    int k;

    srand(time(0));
    for (k = 0; k < SIZE; ++k) {
        a[k] = rand() % 100;
        printf("%d ", a[k]);
    }
    for (k = 0; k < SIZE; ++k)
        toplam += a[k];
    printf("\na elemanlari toplami = %d\n", toplam);

    return 0;
}
```

main işlevinde yer alan ilk *for* döngü deyimiyle *a* dizisinin elemanlarına standart *rand* işlevi çağrılarıyla *0 – 99* aralığında rastgele değerler atanıyor. Yine aynı döngü içinde dizinin her bir elemanının değeri ekrana yazdırılıyor. Bunu izleyen ikinci *for* deyimiyle *a* dizisinin her bir elemanının değeri sırasıyla *toplam* isimli değişkene katılıyor.
Aşağıdaki programda ise *int* türden bir dizinin en küçük değere sahip olan elamanının değeri bulunuyor:

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

#define SIZE       10

int main()
{
    int a[SIZE];
    int toplam = 0;
    int k, min;

    srand(time(0));
    for (k = 0; k < SIZE; ++k) {
        a[k] = rand() % 100;
        printf("%d ", a[k]);
    }
    min = a[0];
    for (k = 1; k < SIZE; ++k)
        if (min > a[k])
            min = a[k];
    printf("\nen kucuk eleman = %d\n", min);

    return 0;
}
```

Algoritmayı biliyorsunuz. *min* isimli değişken, dizinin en küçük elemanının değerini tutması için tanımlanıyor. Önce dizinin ilk elemanının dizinin en küçük elemanı olduğu var sayılıyor. Daha sonra bir *for* döngü deyimiyle dizinin *1* indisli elemanından başlanarak

dizinin diğer elemanlarının değerlerinin *min* değişkeninin değerinden daha küçük olup olmadığı sınanıyor. Eğer dizinin herhangi bir elemanın değeri *min* değişkeninin değerinden daha küçük ise *min* değişkeninin değeri değiştiriliyor ve yeni bulunan elemanın değeri *min* değişkenine atanıyor. Döngü çıkışında artık *min* değişkeni, dizinin en küçük elemanının değerini tutar, değil mi?

Aşağıdaki programda ise *int* türden bir dizinin tek ve çift sayı olan elemanlarının aritmetik ortalamaları ayrı ayrı hesaplanıyor:

```
#include <stdio.h>

#define SIZE       10

int main()
{
    int a[SIZE] = {2, 3, 1, 7, 9, 12, 4, 8, 19, 10};
    int sum_of_odds = 0;
    int sum_of_even = 0;
    int no_of_odds = 0;
    int k;

    for (k = 0; k < SIZE; ++k)
        if (a[k] % 2) {
            sum_of_odds += a[k];
            no_of_odds++;
        }
        else
            sum_of_even += a[k];
    if (no_of_odds)
        printf("Teklerin ortalamasi = %lf\n",(double)sum_of_odds
    /no_of_odds);
    else
        printf("Dizide tek sayi yok!\n");

    if (SIZE - no_of_odds)
        printf("Ciftlerin ortalamasi = %lf\n",(double)sum_of_even /(SIZE -
no_of_odds));
    else
        printf("Dizide cift sayi yok!\n");

    return 0;
}
```

Aşağıdaki programda bir dizi içinde arama yapılıyor:

```
#include <stdio.h>

#define SIZE       10

int main()
{
    int a[SIZE] = {2, 3, 1, 7, 9, 12, 4, 8, 19, 10};
    int k;
    int searched_val;

    printf("aranacak degeri girin : ");
    scanf("%d", &searched_val);
    for (k = 0; k < SIZE; ++k)
        if (a[k] == searched_val)
            break;
```

```
    if (k < SIZE)
        printf("a[%d] = %d\n", k, a[k]);
    else
        printf("aranan deger dizide yok!\n");

    return 0;
}
```

searched_val isimli değişken, dizide aranacak değeri tutmak için tanımlanıyor. Bu değişkenin değeri standart *scanf* işleviyle klavyeden alınıyor. Daha sonra oluşturulan bir *for* döngüsüyle, dizinin her bir elemanının aranan değere eşitliği döngü gövdesinde yer alan bir *if* deyimiyle sınanıyor. Eğer dizinin herhangi bir elemanı aranan değere eşit ise *break* deyimi ile döngüden çıkılıyor. Döngü çıkışında eğer döngü değişkeni olan *k, SIZE* değerinden küçük ise aranan değer bulunmuş yani döngüden *break* deyimi ile çıkılmıştır. Döngüden *break* deyimi ile çıkılmamışsa *k* değişkenin değeri *SIZE* değerine eşit olur, değil mi?
Dizinin elemanları dizinin içinde sırasız yer alıyorsa, bir değerin dizide bulunmadığı sonucunu çıkarmak için dizinin tüm elemanlarına bakılmalıdır.
Ancak dizi sıralı ise *binary search* ismi verilen bir algoritmanın kullanılması arama işleminin çok daha verimli yapılmasını sağlar:

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

#define     SIZE    100

int main()
{
    int a[SIZE];
    int k, mid, searched_val;
    int val = 1;
    int low = 0;
    int high = 1;

    srand(time(0));

    for (k = 0; k < SIZE; ++k) {
        a[k] = val;
        val += rand() % 10;
        printf("%d ", a[k]);
    }
    printf("\naranacak degeri girin : ");
    scanf("%d", &searched_val);
    while (low <= high) {
        mid = (low + high) / 2;
        if (a[mid] == searched_val)
            break;
        if (a[mid] > searched_val)
            high = mid - 1;
        else
            low = mid + 1;
    }
    if (low > high)
        printf("%d degeri dizide bulunamadi!\n", searched_val);
    else
        printf("a[%d] = %d\n", mid, searched_val);

    return 0;
}
```

Yukarıdaki *main* işlevinde sıralı bir dizinin içinde arama yapmak amacıyla *binary search* isimli algoritma kullanılıyor. Dizi sıralanmış olduğuna göre, dizinin ortadaki elemanına bakılmasıyla, dizideki elemanların yarısı artık sorgulama dışı bırakılır, değil mi?
low değişkeni arama yapılacak dizi parçasının en düşük indisini, *high* değişkeni ise en büyük indisini tutuyor. Daha sonra *low* değeri, *high* değerinden küçük ya da eşit olduğu sürece dönen bir *while* döngüsü oluşturulduğunu görüyorsunuz. *mid* değişkeni arama yapılacak dizi parçasının ortadaki elemanının indisini tutuyor. Dizinin *mid* indisli elemanının aranan değer olup olmadığına bakılıyor. Aranan değer bulunamamışsa iki olasılık vardır: *mid* indisli dizi elemanı aranan değerden büyük ise *high* değişkeninin değeri *mid - 1* yapılıyor. Böylece arama yapılacak dizi boyutu yarıya düşürülüyor. *mid* indisli dizi elemanı aranan değerden küçük ise *low* değişkeninin değeri *mid + 1* yapılıyor. Böylece yine arama yapılacak dizi boyutu yarıya düşürülüyor.
while döngüsü çıkışında eğer *low* değeri *high* değerinden büyükse aranan değer bulunamamış demektir. Aksi halde dizinin *mid* indisli elemanı aranan değerdir.

Dizilerin Sıralanması

Dizinin elemanlarını küçükten büyüğe ya da büyükten küçüğe sıralamak için farklı algoritmalar kullanılabilir. Aşağıda, algısal karmaşıklığı çok yüksek olmayan "kabarcık sıralaması" (*bubble sort)* isimli algoritma ile bir dizi sıralanıyor:

```
#include <stdio.h>

#define SIZE       10

int main()
{
    int a[SIZE] = {2, 3, 1, 7, 9, 12, 4, 8, 19, 10};
    int i, k, temp;

    for (i = 0; i < SIZE - 1; ++i)
        for (k = 0; k < SIZE -1 - i; ++k)
            if (a[k] > a[k + 1]) {
                temp = a[k];
                a[k] = a[k + 1];
                a[k + 1] = temp;
            }
    for (k = 0; k < SIZE; ++k)
        printf("%d ", a[k]);

    return 0;
}
```

Aynı algoritma bir *do while* döngüsü kullanılarak da kodlanabilirdi:

```
#include <stdio.h>

#define SIZE              10
#define UNSORTED          0
#define SORTED            1

int main()
{
    int a[SIZE] = {12, 25, -34, 45, -23, 29, 12, 90, 1, 20};
    int i, k, temp, flag;

    do {
        flag = SORTED;
        for (k = 0; k < SIZE - 1; ++k)
            if (a[k] > a[k + 1]) {
```

```
                temp = a[k];
                a[k] = a[k + 1];
                a[k + 1] = temp;
                flag = UNSORTED;
            }
    } while (flag == UNSORTED);

    for (i = 0; i < SIZE; ++i)
        printf("a[%d] = %d\n", i, a[i]);

    return 0;
}
```

Aşağıdaki programda bir dizinin elemanları küçükten büyüğe "araya sokma" (*insertion sort)* algoritmasıyla sıraya diziliyor:

```
#include <stdio.h>

#define SIZE       10

int main()
{
    int a[SIZE] = {2, 3, 1, 7, 9, 12, 4, 8, 19, 10};
    int i, k, temp;

    for (i = 1; i < SIZE; ++i) {
        temp = a[i];
        for (k = i; k > 0 && a[k - 1] > temp; --k)
            a[k] = a[k - 1];
        a[k] = temp;
    }

    for (i = 0; i < SIZE; ++i)
        printf("%d ", a[i]);

    return 0;
}
```

Sıralama yönünü küçükten büyüğe yapmak yerine büyükten küçüğe yapmak için içteki döngüyü aşağıdaki gibi değiştirmek yeterli olur.

```
for (k = i; k > 0 && dizi[k - 1] < temp; --k)
```

Aşağıdaki programda ise diziyi küçükten büyüğe sıralamak için "seçme sıralaması" *(selection sort)* algoritması kullanılıyor:

```
#include <stdio.h>

#define SIZE       10

int main()
{
    int a[SIZE] = {2, 3, 1, 7, 9, 12, 4, 8, 19, 10};
    int i, k, min, index;

    for (k = 0; k < SIZE; ++k) {
        min = a[k];
        index = k;
        for (i = k + 1; i < SIZE; ++i)
            if (a[i] < min) {
```

```
                min = a[i];
                index = i;
            }
        a[index] = a[k];
        a[k] = min;
    }

    for (k = 0; k < SIZE; ++k)
        printf("%d ", a[k]);

    return 0;
}
```

Aşağıdaki programda dizinin değeri tek olan elemanları küçükten büyüğe olacak şekilde dizinin başına, dizinin çift olan elemanları ise küçükten büyüğe dizinin sonuna yerleştiriliyor:

```
#include <stdio.h>

#define SIZE       10

int main()
{
    int a[SIZE] = {2, 3, 1, 7, 9, 12, 4, 8, 19, 10};
    int i, k, temp;

    for (i = 0; i < SIZE - 1; ++i)
        for (k = 0; k < SIZE - 1 - i; ++k)
            if (a[k] % 2 == a[k + 1] % 2 && a[k] > a[k + 1] ||
                a[k] % 2 == 0 && a[k + 1] % 2 != 0) {
                temp = a[k];
                a[k] = a[k + 1];
                a[k + 1] = temp;
            }
    for (k = 0; k < SIZE; ++k)
        printf("%d ", a[k]);

    return 0;
}
```

Amacı gerçekleştirmek için yine "kabarcık sıralaması" algoritmasının kullanıldığını, ancak takas yapmak için sınanan koşul ifadesinin değiştirildiğini fark ettiniz mi?

Aşağıdaki programda bir dizi ters çevriliyor:

```
#include <stdio.h>

#define SIZE       10

int main()
{
    int a[SIZE] = {2, 3, 1, 7, 9, 12, 4, 8, 19, 10};
    int k;

    for (k = 0; k < SIZE / 2; ++k) {
        int temp = a[k];
        a[k] = a[SIZE - 1 - k];
        a[SIZE - 1 - k] = temp;
    }

    for (k = 0; k < SIZE; ++k)
```

```
        printf("%d ", a[k]);

    return 0;
}
```

Dizinin ters çevrilmesi için dizi boyutunun yarısı kadar dönen bir döngü içinde, dizinin baştan *n*. elemanı ile sondan *n*. elemanı takas ediliyor.

Aşağıda tanımlanan *urand* isimli işlev her çağrıldığında *0* - *MAX* değerleri arasında farklı bir rastgele sayı üretiyor. İşlevin *MAX* adet tamsayıyı ürettikten sonra çağrıldığında hata durumunu bildirmek için *-1* değerine geri dönüyor:

```
#include <stdio.h>

#define     MAX     100

int flags[MAX] = {0};

int urand(void)
{
    int k, val;

    for (k = 0; k < MAX; ++k)
        if (flags[k] == 0)
            break;
    if (k == MAX)
        return -1;

    while (flags[val = rand() % MAX])
        ;
    ++flags[val];

    return val;
}

int main()
{
    int k;

    srand(time(0));

    for (k = 0; k < MAX; ++k)
        printf("%d ", urand());

    printf("\n\n%d\n", urand());

    return 0;
}
```

İşlev, bir tamsayının daha önce üretilip üretilmediğini anlayabilmek için *flags* isimli bir global bayrak dizisini kullanıyor. *flags* dizisinin bir elemanının değeri *0* ise o indise karşılık gelen değerin işlev tarafından henüz üretilmediği anlaşılıyor. Dizi elemanının değeri eğer *1* ise, o değerin daha önce üretildiği anlaşılıyor.

```
while (flags[val = rand() % MAX])
    ;
```

döngüsünden *flags* dizisinin *val* değişkenine atanan rastgele indisli bir elemanının değeri sıfır olduğunda çıkılır, değil mi?

Aşağıdaki program, rastgele üretilen bir sayısal loto kuponunu ekrana yazıyor:

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

#define    KOLON_SAYISI       8

void kolon_yaz()
{
    int numaralar[50] = {0};
    int k, no;

    for (k = 0; k < 6; ++k) {
        while (numaralar[no = rand() % 49 + 1])
            ;
        numaralar[no]++;
    }
    for (k = 1; k < 50; ++k)
        if (numaralar[k])
            printf("%2d ", k);
}

int main()
{
    int k;

    srand(time(0));

    for (k = 0; k < KOLON_SAYISI; ++k) {
        printf("kolon %2d : ", k + 1);
        kolon_yaz();
        printf("\n");
    }
    return 0;
}
```

kolon_yaz isimli işlev, tek bir kolonu ekrana yazdırıyor. İşlevde numaralar isimli yerel dizinin, yine bir bayrak dizisi olarak kullanıldığını görüyorsunuz. Dizinin herhangi bir indisli elemanının değerinin *0* olması, o indis değerinin daha önce üretilmeyen bir sayı olduğunu gösteriyor. *for* döngüsü içinde yer alan *while* döngüsü, daha önce üretilmeyen bir sayı bulununcaya kadar dönüyor. Böylece *6* kez dönen *for* döngüsüyle *6* farklı sayı üretilmiş oluyor.
Aşağıdaki programda bir dizinin en büyük ikinci elemanının değeri bulunuyor:

```
#include <stdio.h>
#define    SIZE       10

int main()
{
    int a[SIZE] = {12, 34, 3, 56, 2, 23, 7, 18, 91, 4};
    int k;
    int max1 = a[0];
    int max2 = a[1];

    if (a[1] > a[0]) {
        max1 = a[1];
        max2 = a[0];
    }
```

```
    for (k = 2; k < SIZE; ++k)
        if (a[k] > max1) {
            max2 = max1;
            max1 = a[k];
        }
        else if (a[k] > max2)
            max2 = a[k];

    printf("en buyuk ikinci deger = %d\n", max2);

    return 0;
}
```

Aşağıdaki programda yalnızca bir dizinin içinde tek (*unique*) olan elemanların değerleri ekrana yazdırılıyor.

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

#define     SIZE        100

int main()
{
    int a[SIZE];
    int i, k;
    int counter;

    srand(time(0));

    for (k = 0; k < SIZE; ++k) {
        a[k] = rand() % 30;
        printf("%d ", a[k]);
    }

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

    for (i = 0; i < SIZE; ++i) {
        counter = 0;
        for (k = 0; k < SIZE; ++k)
            if (a[k] == a[i])
                if (++counter == 2)
                    break;
        if (counter == 1)
            printf("%d ", a[i]);
    }
    printf("\n");

    return 0;
}
```

Aşağıdaki program *SIZE* elemanlı bir dizinin tüm elemanlarına *0 - MAX* aralığında birbirinden farklı rastgele değerler yerleştiriyor:

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

#define SIZE    50
#define MAX     100
```

```
int main()
{
    int a[SIZE];
    int k;

    srand(time(0));

    for (k = 0; k < SIZE; ++k) {
        int val;
        while (1) {
            int i;
            val = rand() % MAX;
            for (i = 0; i < k; ++i)
                if (val == a[i])
                    break;
            if (i == k)
                break;
        }
        a[k] = val;
    }
/* dizi yazdırılıyor  */
    for (k = 0; k < SIZE; ++k)
        printf("%d ", a[k]);
    printf("\n");

    return 0;
}
```

Aşağıdaki programda sıralı iki dizi, bir sıralı dizi biçiminde birleştiriliyor:

```
#include <stdio.h>

#define SIZE    10

int main()
{
    int a[SIZE] = {2, 3, 6, 7, 8, 9, 13, 45, 78, 79};
    int b[SIZE] = {1, 2, 4, 5, 7, 9, 10, 18, 33, 47};
    int c[SIZE + SIZE];
    int k;
    int index1 = 0, index2 = 0;

    for (k = 0; k < SIZE + SIZE; ++k)
        if (index1 == SIZE)
            c[k] = b[index2++];
        else if (index2 == SIZE)
            c[k] = a[index1++];
        else {
            if (a[index1] < b[index2])
                c[k] = a[index1++];
            else
                c[k] = b[index2++];
        }

    for (k = 0; k < SIZE + SIZE; ++k)
        printf("%d ", c[k]);

    return 0;
}
```

char Türden Diziler ve Yazılar

Karakter dizileri, *char* türden dizilerdir. Karakter dizilerinin, bazı ek özellikleri dışında, diğer dizi türlerinden bir farkı yoktur. *char* türden diziler, daha çok, içlerinde yazı tutmak için tanımlanır.

```
char str[100];
```

Yukarıdaki tanımlamada *str* dizisi, bütün elemanları *char* türden olan *100* elemanlı bir dizidir. *char* türden bir dizi içinde bir yazı tutmak, dizinin her bir elemanına sırayla yazının bir karakterinin sıra numarasını atamak anlamına gelir. Bu arada *char* türden bir dizinin içinde bir yazı tutmanın zorunlu olmadığını, böyle bir dizi pekala küçük tamsayıları tutmak amacıyla da kullanılabilir.

Yukarıda tanımlanan dizi içinde *"Ali"* yazısı tutulmak istensin:

```
str[0] = 'A';
str[1] = 'l';
str[2] = 'i';
```

Dizi *100* karakterlik olmasına karşın dizi içinde *100* karakterden daha kısa olan yazılar da tutulabilir. Peki dizi içinde saklanan yazıya nasıl erişilebilir? Yazının uzunluk bilgisi bilinmiyor. Örneğin yazı ekrana yazdırılmak istendiğinde, *int* türden diziler için daha önce yazılan aşağıdaki gibi bir döngü deyiminin kullanıldığını düşünelim:

```
for (k = 0; k < 100; ++k)
    putchar(s[k]);
```

Böyle bir döngü ile yalnızca *Ali* yazısı ekrana yazdırılmaz, dizinin diğer *97* elemanının da görüntüleri, yani çöp değerler ekrana yazdırılır, değil mi?

C dilinde karakterler üzerinde işlemlerin hızlı ve etkin bir biçimde yapılabilmesi için *"sonlandırıcı karakter"* (*null character*) kavramından faydalanılır.
Sonlandırıcı karakter, *ASCII* tablosunun ya da sistemde kullanılan karakter setinin sıfır numaralı *('\x0' ya da '\0')* karakteridir. Dolayısıyla sayısal değer olarak *0* sayısına eşittir. Görüntüsü yoktur. Sonlandırıcı karakter *'0'* karakteri ile karıştırılmamalıdır.

'0' karakterinin *ASCII* karakter setindeki kod numarası *48*'dir. Dolayısıyla tamsayı olarak değeri *48*'dir. Oysa *'\0'* karakterinin *ASCII* sıra numarası *0*'dır. Dolayısıyla tamsayı olarak değeri *0*'dır. Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    printf("%d\n", '0');
    printf("%d\n", '\0');

    return 0;
}
```

Yukarıdaki ilk *printf* işlevinin çağrılmasıyla ekrana *48* değeri yazdırılırken, ikinci *printf* işlevinin çağrılmasıyla ekrana *0* değeri yazdırılır.

char Türden Dizilere İlkdeğer Verilmesi

char türden dizilere ilkdeğer verme işlemi *(initializing)* aşağıdaki biçimlerde yapılabilir:
Diğer türden dizilerde olduğu gibi virgüllerle ayrılan ilkdeğerler, küme ayracı içinde yer alır:

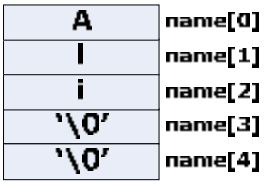
```
char name[7] = {'N', 'e', 'c', 'a', 't', 'i', '\0'};
```

Diğer dizilerde olduğu gibi dizi eleman sayısından daha fazla sayıda elemana ilkdeğer vermek geçersizdir.

```
char name[5] = {'N', 'e', 'c', 'a', 't', 'i'};   /* Geçersiz */
```

Dizi eleman sayısı kadar elemana ya da dizi eleman sayısından daha az sayıda elemana ilkdeğer verilebilir. Daha az sayıda elemana ilkdeğer verilmesi durumunda ilkdeğer verilmemiş elemanlar, diğer dizilerde olduğu gibi, *0* değeriyle başlatılır. *0* değerinin sonlandırıcı karakter olduğunu biliyorsunuz:

```
char name[5] = {'A', 'l', 'i'};
```



Dizi elemanlarına ilkdeğer verilirken dizi boyutu belirtilmeyebilir. Bu durumda derleyici dizi boyutunu verilen ilkdeğerleri sayarak saptar. Derleyici diziyi bu boyutta açılmış varsayar.

```
char name[ ] = {'A', 'l', 'i'};
```

Yukarıdaki tanımlama deyimiyle derleyici, *name* dizisinin *3* elemanlı olarak açıldığını varsayar.
char türden dizilerin tanımlanmasında dizinin boyut değeri yazılmadan, dizinin elemanlarına virgüllerle ayrılmış değerlerle ilkdeğer verilmesi durumunda, derleyici sonlandırıcı karakteri dizinin sonuna otomatik olarak yerleştirmez. Bu durumda yazının sonunda bulunması gereken sonlandırıcı karakter ilkdeğer olarak listede bulunmak zorundadır:

```
char name[] = {'A', 'l', 'i', '\0'};
```

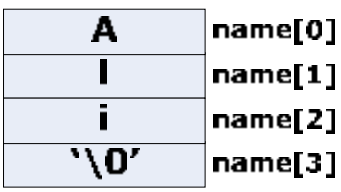
Aynı durum dizinin boyutu ile verilen ilkdeğerlerin sayısını aynı olduğunda da geçerlidir:

```
char isim[7] = {'N', 'e', 'c', 'a', 't', 'i', '\0'};
```

Bu şekilde ilkdeğer vermek zahmetli olduğundan, ilkdeğer vermede ikinci bir biçim oluşturulmuştur. Karakter dizilerine ilkdeğerler çift tırnak içinde de verilebilir:

```
char name[] = "Ali";
```

Bu biçimin diğerinden farkı, derleyicinin sonuna otomatik olarak sonlandırıcı karakteri yerleştirmesidir.



Yukarıdaki örnekte derleyici, *name* dizisini *4* elemanlı olarak açılmış varsayar. Dizinin eleman sayısından daha fazla sayıda elemana ilkdeğer vermek geçersizdir.

```
char city[5] = "İstanbul";        /* Geçersiz */
```

Bu durumun bir istisnası vardır. Eğer dizinin eleman sayısı kadar elemana çift tırnak içinde ilkdeğer verilirse bu durum geçerlidir. Derleyici bu durumda sonlandırıcı karakteri dizinin sonuna yerleştirmez.
[Bu durum C dili standartlarına yöneltilen eleştirilerden biridir. C++ dilinde bu durum sözdizim hatasıdır.]

```
char name[3] = "Ali";  /* C'de geçerli, C++'da geçersiz */
```

A	name[0]
l	name[1]
i	name[2]
???	Buraya '\0' yerleştirilmiyor.

'\0' karakteri, karakter dizileri üzerinde yapılan işlemleri hızlandırmak için kullanılır. Örneğin *int* türden bir dizi kullanıldığında, dizi elemanları üzerinde döngüleri kullanarak işlem yaparken dizi uzunluğunun mutlaka bilinmesi gerekir. Ama *char* türden diziler söz konusu olduğunda artık dizi uzunluğunu bilmek gerekmez, çünkü yazının sonunda *'\0'* karakter bulunacağından, kontrol ifadeleriyle bu durum sınanarak yazının sonuna gelinip gelinmediği anlaşılabilir. Ancak *'\0'* karakterin, karakter dizilerinde yazıların son elemanı olarak kullanılmasının bir zararı da diziye fazladan bir karakter, yani *'\0'* karakteri eklemek zorunluluğudur. Bu nedenle *SIZE* elemanlı bir dizide en fazla *SIZE – 1* uzunluğunda bir yazı saklanabilir.
C dilinde bir yazıyı klavyeden alan ya da bir yazıyı ekrana yazan standart işlevler bulunur.

gets işlevi

Daha önce ele alınan *getchar*, *getch* ve *getche* işlevleri klavyeden tek bir karakter alıyorlardı. *gets,* klavyeden karakter dizisi almakta kullanılan standart bir C işlevidir. Kullanıcı, klavyeden karakterleri girdikten sonra *enter* tuşuna basmalıdır. İşlev, klavyeden girilecek karakterlerin yerleştirileceği dizinin ismini parametre olarak alır. Daha önce de belirtildiği gibi dizi isimleri aslında bir adres bilgisi belirtir. *gets* işlevinin de parametresi aslında *char* türden bir adrestir. Ancak göstericilerle ilgili temel kavramlar henüz anlatılmadığı için şimdilik *gets* işlevini yalnızca işinize yarayacak kadar öğreneceksiniz. Örneğin :

```
char s[20];
gets(s);
```

ile klavyeden *enter* tuşuna basılana kadar girilmiş olan tüm karakterler, *name* dizisi içine sırayla yerleştirilir. Klavyeden *"Necati"* yazısının girildiğini varsayalım:
gets işlevi, klavyeden girilen karakterleri diziye yerleştirdikten sonra dizinin sonuna sonlandırıcı karakteri yerleştirir.

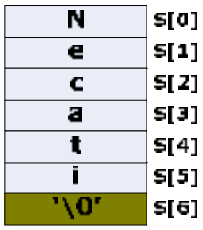
N	s[0]
e	s[1]
c	s[2]
a	s[3]
t	s[4]
i	s[5]
'\0'	s[6]

gets işlevi, dizi için hiçbir şekilde dizinin taşmasına yönelik bir kontrol yapmaz. *gets* işlevi ile dizi eleman sayısından daha fazla karakter girilirse, dizi taşacağı için beklenmeyen sonuçlarla karşılaşılabilir. Bu tür durumları göstericiler konusunda *"gösterici hataları"* başlığı altında ayrıntılı olarak inceleyeceğiz.
gets işlevi *'\0'* karakterini dizinin sonuna eklediği için, *SIZE* boyutunda bir dizi için *gets* işleviyle alınacak karakter sayısı en fazla *SIZE – 1* olmalıdır. Çünkü sonlandırıcı karakter de diğer karakterler gibi bellekte bir yer kaplar. Örnek :

```
char isim[6];
gets(isim);
```

ile klavyeden *Necati* isminin girildiğini düşünelim:

isim dizisinin tanımlanmasıyla derleyici bu dizi için bellekte *6 byte* yer ayırır (*isim[0] ...isim[5]).*



gets işlevi bu durumda *'\0'* karakterini, derleyicinin dizi için ayırmadığı bir bellek hücresine yazar. Bu tür durumlara "dizinin bir taşırılması hatası" *(off bye one)* denir. Taşma durumuyla ilgili olarak ortaya çıkacak hatalar, derleme zamanına değil çalışma zamanına *(run time)* ilişkindir.
Aşağıdaki programı inceleyin:

```
#include  <stdio.h>

#define     SIZE     100

int main()
{
    char str[SIZE];
    int ch;
    int index = 0;

    printf("bir yazı girin: ");
    printf("\n\n");

    while ((ch = getchar()) != '\n')
        str[index++] = ch;
    str[index] = '\0';

    return 0;
}
```

Yukarıdaki *main* işlevinde klavyeden alınan bir yazı *str* dizisi içinde saklanıyor. Klavyeden *'\n'* karakteri alınana kadar, girilen tüm karakterler *str* dizisinin elemanlarına sırayla atanıyor. Klavyeden *'\n'* karakteri alındığında, diziye yazılan son karakterden sonra, dizideki yazının sonunu işaretlemesi amacıyla *'\0'* karakter yazılıyor.

Klavyeden alınan bir yazı, *char* türden bir dizinin içine standart *scanf* işleviyle de yerleştirilebilir. Bu amaçla *%s* format karakterleri kullanılır. Ancak bu durumda klavyeden

girilen karakterlerin hepsi diziye yerleştirilmez. Klavyeden alınan ilk boşluk karakteri ile diziye yerleştirme işlemi sona erer. Aşağıdaki programı inceleyin:

```
#include <stdio.h>

int main()
{
    char name[20];
    char fname[30];
    int no;

    printf("isim soyisim ve numara girin : ");
    scanf("%s%s%d", name, fname, &no);
    /***/
    return 0;
}
```

Programın çalışma zamanında *scanf* işlevi çağrıldığında aşağıdaki girişin yapıldığını düşünelim. ('_' karakteri boşluk karakterini gösteriyor):

```
__Necati___Ergin___564
```

Bu durumda *Necati* yazısı *name* dizisine, *Ergin* yazısı *fname* dizisine, *564* tamsayı değeri ise *no* isimli değişkene yerleştirilir.

puts işlevi

puts, standart bir C işlevidir. Bu işlev, bir karakter dizisinde tutulan yazıyı ekrana yazdırmak için kullanılır. Yazıyı saklayan karakter dizisinin ismini (dizi ismi derleyici tarafından otomatik olarak dizinin başlangıç adresine dönüştürülmektedir) parametre olarak alır. *puts* işlevi, karakter dizisini ekrana yazdıktan sonra imleci sonraki satırın başına geçirir:

```
#include <stdio.h>

int main()
{
    char name[20];

    printf("bir isim girin : ");
    gets(name);
    puts(name);

    return 0;
}
```

Yukarıdaki örnekte *gets* işlevi ile klavyeden alınan yazı, *puts* işlevi ile ekrana yazdırılıyor.

Karakter dizileri içinde tutulan yazıları ekrana yazdırmak için, standart *printf* işlevi de kullanılabilir. Bu durumda formatlama karakterleri olarak *%s* kullanılarak dizinin ismi (dizinin başlangıç adresi) ile eşlenir.

```
printf("%s\n", name);
```

ile

```
puts(name);
```

aynı işi yapar. Ancak *printf* işlevi, dizi içinde tutulan yazıyı ekrana yazdırdıktan sonra imleci alt satıra taşımaz.

```
puts(name);
```

deyimi yerine aşağıdaki kod parçası da yazılabilirdi:

```
for (i = 0; name[i] != '\0'; ++i)
     putchar(name[i]);
putchar('\n');
```

puts işlevi ve *%s* format karakteriyle kullanıldığında *printf* işlevi, sonlandırıcı karakter görene kadar bütün karakterleri ekrana yazar. Bu durumda, yazının sonundaki sonlandırıcı karakter herhangi bir şekilde ezilirse her iki işlev de ilk sonlandırıcı karakteri görene kadar yazma işlemini sürdürür. Aşağıdaki programı inceleyin:

```
#include <stdio.h>

int main()
{
    char city[] = "Ankara";

    city[6] = '!';
    puts(city);

    return 0;
}
```

```
city[6] = '!';
```

atamasıyla *Ankara* yazısının sonundaki sonlandırıcı karakter ezilerek üzerine *!* karakteri yazılıyor. Daha sonra çağrılan *puts* işlevi ekrana

```
Ankara!
```

yazısını yazdıktan sonra ilk sonlandırıcı karakteri görene kadar ekrana yazmayı sürdürür. Göstericiler konusunda bu durumun bir gösterici hatası oluşturduğunu göreceksiniz. *puts* ve *printf* işlevleri, karakter dizilerini yazarken yalnızca sonlandırıcı karakteri dikkate alır. Bu işlevler karakter dizilerinin uzunluklarıyla ilgilenmez.

Karakter Dizileriyle İlgili Bazı Küçük Uygulamalar

Aşağıdaki programda bir karakter dizisi içinde tutulan yazının uzunluğu bulunuyor:

```
#include <stdio.h>

#define    SIZE       100

int main()
{
    char str[SIZE];
    int k;
    int len = 0;

    printf ("bir yazi girin : ");
    gets(str);
    printf("yazi = (%s)\n", str);
    for (k = 0; str[k] != '\0'; ++k)
        len++;
    printf("(%s) yazisinin uzunlugu = %d\n", str, len);
```

```
    return 0;
}
```

Programda yer alan

```
for (k = 0; str[k] != '\0'; ++k)
    len++;
```

döngü deyiminin yürütülmesinden sonra, döngü değişkeni olan *k*'nın değeri de yazının uzunluğu olur, değil mi?

Aşağıdaki programda *char* türden bir dizi içine alınan yazı, ekrana tersten yazdırılıyor:

```
#include <stdio.h>

#define    SIZE    100

int main()
{
    char s[SIZE];
    int k;

    printf("bir yazı girin :");
    gets(s);
    for (k = 0; s[k] != '\0'; ++k)
        ;

    for (--k; k >= 0; --k)
        putchar(s[k]);

    return 0;
}
```

Aşağıdaki programda önce bir karakter dizisine bir yazı alınıyor. Daha sonra yazının küçük harf karakterleri büyük harfe, büyük harf karakterleri küçük harfe dönüştürülüyor:

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

#define    SIZE       100

int main()
{
    char str[SIZE];
    int k;

    printf ("bir yazi girin : ");
    gets(str);
    printf("yazi = (%s)\n", str);

    for (k = 0; str[k] != '\0'; ++k)
        str[k] = isupper(str[k]) ? tolower(str[k]) : toupper(str[k]);

    printf("donustulmus yazi = (%s)\n", str);

    return 0;
}
```

Aşağıdaki programda bir karakter dizisine klavyeden alınan yazı ters çevriliyor:

```
#include <stdio.h>

#define    SIZE       100

int main()
{
    char str[SIZE];
    int k, temp, len;

    printf ("bir yazi girin : ");
    gets(str);
    for (len = 0; str[len] != '\0'; ++len)
        ;

    for (k = 0; k < len / 2; ++k) {
        temp = str[k];
        str[k] = str[len - 1 - k];
        str[len - 1 - k] = temp;
    }
    printf("ters cevrilmis yazi = (%s)\n", str);
    return 0;
}
```

Yukarıdaki kodda kullanılan algoritmayı inceleyin. Birinci *for* döngüsü ile yazının uzunluğu bulunuyor. Daha sonra yazının uzunluğunun yarısı kadar dönen bir *for* döngü deyimi oluşturuluyor. Döngünün her turunda yazının baştan n. karakteri ile sondan n. karakteri yer değiştiriliyor. Yazı uzunluğu tek sayı ise, yazının ortasındaki karakter yerinde kalır. Yazının sonundaki sonlandırıcı karakter

```
str[len]
```

olduğuna göre, yazının son karakteri

```
str[len - 1]
```

karakteridir, değil mi?
Aşağıdaki programda ise klavyeden girilen bir yazının içinde bulunan tüm İngilizce harfler sayılıyor ve kaç tane oldukları ekrana yazdırılıyor:

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

#define    SIZE       500

int main()
{
    char str[SIZE];
    int letter_counter[26] = {0};
    int k;

    printf("bir yazi girin : ");
    gets(str);
    for (k = 0; str[k] != '\0'; ++k)
        if (isalpha(str[k]))
            letter_counter[toupper(str[k]) - 'A']++;
    for (k = 0; k < 26; ++k)
        if (letter_counter[k])
            printf("%3d tane %c\n", letter_counter[k], 'A' + k);
    return 0;
```

```
}
```

main işlevi içinde kullanılan *letter_counter* isimli dizi, bir sayaç dizisi olarak kullanılıyor. Dizinin *1.* elemanı *'A', 'a'* karakterlerinin, dizinin 2. elemanı *'B', 'b'* karakterlerinin, dizinin sonuncu elemanı *'Z', 'z'* karakterlerinin sayacı olarak görev yapıyor. Bu yerel dizi, ilkdeğer verme deyimiyle sıfırlanıyor. İlk *for* döngü deyimiyle, yazının tüm karakterleri dolaşılıyor, yazının herhangi bir karakteri eğer bir harf karakteri ise, büyük harfe dönüştürülerek bu karakterden 'A' değeri çıkarılıyor. Elde edilen değerin *letter_counter* dizisine indis yapıldığını ve *letter_counter* dizisinin bu indisli elemanının değerinin 1 artırıldığını görüyorsunuz.
İkinci *for* döngü deyimiyle ise bu kez sayaç dizisinin, değeri *0* olmayan elemanlarının değerleri ekrana yazdırılıyor.
Aşağıdaki programda ise bir diziye alınan bir yazı içinden rakam karakterleri siliniyor. Kodu inceleyin:

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

#define    SIZE       500

int main()
{
    char str[SIZE];
    int k;
    int index = 0;

    printf("bir yazi girin : ");
    gets(str);

    printf("yazi = (%s)\n", str);

    for (k = 0; str[k] != '\0'; ++k)
        if (!isdigit[k])
            str[index++] = str[k];
    str[index] = '\0';
    printf("yazi = (%s)\n", str);

    return 0;
}
```

Yazıdan rakam karakterlerini silmek için yazı, bulunduğu yere yeniden kopyalanıyor. Ancak kopyalama yapılırken, rakam karakterleri kopyalanmıyor. *index* isimli değişken, dizinin neresine yazılacağını gösteriyor. Eğer bir karakter rakam karakteri değilse, bu karakter dizinin *index* indisli elemanına atanıyor, sonra *index* değişkeninin değeri *1* artırılıyor. Ancak yazının tamamını dolaşan *for* döngüsünden çıktıktan sonra, silme işleminden sonra oluşan yazının sonuna, sonlandırıcı karakter ekleniyor.

Aşağıdaki programda bir yazının toplam sözcük sayısı bulunuyor:

```
#include <stdio.h>

#define    SIZE          200
#define    OUTWORD       0
#define    INWORD        1

int is_sep(int ch);

int main()
```

```
{
    char str[SIZE];
    int word_counter = 0;
    int k;
    int word_flag = OUTWORD;

    printf("bir yazi girin : ");
    gets(str);

    for (k = 0; str[k] != '\0'; ++k)
        if (is_sep(str[k]))
            word_flag = OUTWORD;
        else if (word_flag == OUTWORD) {
            word_flag = INWORD;
            word_counter++;
        }

    printf("toplam %d sozcuk var!\n", word_counter);

    return 0;
}

int is_sep(int ch)
{
    char seps[] = " \t.,;:?!";
    int k;

    for (k = 0; seps[k] != '\0'; ++k)
        if (ch == seps[k])
            return 1;
    return 0;
}
```

is_sep işlevi, sıra numarasını aldığı bir karakterin, sözcükleri birbirinden ayıran ayraç karakterlerinden biri olup olmadığını sınıyor.
main işlevi içinde tanımlanan *word_flag* isimli bayrak değişkeni, bir sözcüğün içinde mi dışında mı olunduğunu gösteriyor. Bu değişkene ilkdeğer olarak, kelimenin dışında (*OUT*) değerinin verildiğini görüyorsunuz.
Bir *for* döngü deyimiyle yazının her bir karakterinin ayraç karakteri olup olmadığı sınanıyor. Eğer ayraç karakteri ise *word_flag* değişkenine *OUT* değeri atanıyor. Eğer karakter ayraç karakteri değilse ve aynı zamanda bayrağın değeri *OUT* ise, bayrağa *IN* değeri atanıyor ve sözcük sayısını tutan sayacın değeri *1* artırılıyor.

Aşağıdaki programda, bir yazının içinde ardışık olarak yer alan eş karakterlerin sayısı bire indiriliyor:

```
#include <stdio.h>

#define    SIZE    100

int main()
{
    char str[SIZE];
    int index = 0;
    int k;

    printf("bir yazi girin : ");
    gets(str);

    for (k = 0; str[k] != '\0'; ++k)
```

```
        if (str[k] != str[k + 1])
            str[++index] = str[k + 1];
    printf("yazi = (%s)\n", str);

    return 0;
}
```

sizeof İşleci

sizeof, bir ifadenin türünün bellekte kaç *byte* yer kapladığı değerini üreten bir işleçtir. *sizeof* işleci tek terimli önek konumunda bir işleçtir.
sizeof işlecinin terimi aşağıdakilerden biri olabilir:

1. Terim olarak tür belirten sözcükler kullanılabilir. Bu durumda terimin ayraç içine alınması zorunludur. Örnekler:

```
sizeof(int)
sizeof(double)
sizeof(long)
```

İşleç bu durumda terimi olan tür bilgisinin kullanılan sistemde kaç *byte* yer kaplayacağı değerini üretir. Örneğin *Windows* ya da *UNIX* sistemlerinde

```
sizeof(int)
```

gibi bir ifadenin değeri *4*'tür.

2. Terim olarak bir ifade kullanılabilir. Bu durumda terimin ayraç içine alınması zorunlu değildir. Ancak programcıların çoğu okunabilirlik açısından terimi ayraç içine almayı yeğler:

```
double x;

sizeof (x)
sizeof(17.8)
sizeof(func())
```

İşleç bu durumda, terimi olan ifadenin ait olduğu türün, kullanılan sistemde kaç *byte* yer kaplayacağı değerini üretir. Örneğin *Windows* ya da *UNIX* sistemlerinde

```
sizeof(x)
```

gibi bir ifadenin değeri *8*'dir. Böyle bir ifade doğal olarak, ilgili sistemde *x* nesnesinin bellekte kaç *byte* yer kapladığını belirlemekte de kullanılabilir. *sizeof* işleci en çok bu biçimiyle kullanılır. Yani işlecin terimi nesne gösteren bir ifade seçilerek, terimi olan nesnenin bellekte kaç *byte* yer kapladığı öğrenilir.

3. sizeof işleci terim olarak bir dizi ismi aldığında, *byte* olarak o dizinin toplam uzunluğunu değer olarak üretir:

```
double a[10];
sizeof(a)
```

ifadesi *80* değerini üretir.

Diğer taraftan *sizeof* işlecinin terim olarak dizinin bir elemanını alarak ürettiği değer, dizi hangi türden ise o türün kullanılan sistemdeki *byte* olarak uzunluğu olur. Yani yukarıdaki örnekte

```
sizeof(a[0])
```

ifadesi *8* değerini üretir.

Bu durumda

```
sizeof(a) / sizeof(a[0])
```

ifadesi dizi boyutunu verir. Örnek:

```
for (i = 0; i < sizeof(a) / sizeof(a[0]); ++i)
    a[i] = 0;
```

a bir dizi ismi olmak üzere, yukarıdaki döngü, *a* dizisinin eleman sayısı kadar döner. Yukarıdaki döngüde dizi boyutunun açık biçimde yazılması yerine

```
sizeof(a) / sizeof(a[0]
```

biçiminde yazılması size şaşırtıcı gelebilir. Böyle bir yazım biçiminin bir faydası olabilir mi? Dizi tanımlamalarında, ilkdeğer verilen dizilerin boyutlarının belirtilmesine gerek olmadığını, derleyicinin dizi boyutunu verilen ilkdeğerlerin sayısından çıkardığını biliyorsunuz. Aşağıdaki kodu inceleyin:

```
#include <stdio.h>

int main()
{
    int a[] = {2, 5, 7, 8, 9, 23, 67};
    int k;

    for (k = 0; k < sizeof(a) / sizeof(a[0]); ++k)
        printf("%d ", a[k]);
    printf("\n");

    return 0;
}
```

Yukarıdaki *main* işlevi içinde a isimli *int* türden bir dizi tanımlanıyor. Tanımlanan diziye ilkdeğer veriliyor. Derleyici verilen ilkdeğerlerin sayısını sayarak dizinin boyutunu *8* olarak saptar ve kodu buna göre üretir. *main* işlevi içinde yer alan *for* döngü deyimi, dizinin eleman sayısı kadar, yani 8 kez döner.
Şimdi kaynak kodda değişiklik yapıldığını, *a* dizisine birkaç eleman daha eklendiğini düşünelim:

```
int a[] = {2, 5, 7, 8, 9, 23, 67, 34, 58, 45, 92};
```

Bu durumda *for* döngü deyiminde bir değişiklik yapılmasına gerek kalmaz. Çünkü derleyici bu kez dizinin boyutunu *11* olarak hesaplar. *for* döngü deyimi içinde kullanılan

```
sizeof(a) / sizeof(a[0]
```

ifadesi de bu kez *11* değerini üretir.

sizeof İşlecinin Önceliği

Tek terimli tüm işleçlerin, işleç öncelik tablosunun ikinci seviyesinde yer aldığını biliyorsunuz. *sizeof* da ikinci seviyede bulunan bir işleçtir.

sizeof Bir İşlev Değildir

sizeof işlecinin terimi çoğu kez bir ayraç içine yazıldığından, işlecin kullanımı bir işlev çağrısı görüntüsüne benzer:

```
sizeof(y)
```

Ancak *sizeof* bir işlev değil bir işleçtir. *sizeof* C dilinin *32* anahtar sözcüğünden biridir.

sizeof İşlecinin Ürettiği Değerin Türü

sizeof işlecinin ürettiği değer *unsigned int* türdendir. İşlecin ürettiği değerin türünü *signed int* kabul etmek hatalıdır. İşlecin ürettiği değer, *signed int* türden negatif bir sayıyla işleme sokulduğunda tür dönüşümü işaretsiz yöne yapılır:

```
-2 * sizeof(int)
```

-2, işaretli *int* türden bir değişmezdir. *sizeof(int)* ifadesinin ürettiği değer ise *unsigned int* türden *4* değeridir. İşlem öncesi yapılacak otomatik tür dönüşümü ile *-2* değeri *unsigned int* türüne dönüştürülür. İşlem *unsigned int* türde yapılır. Yani işlemin sonucu *8* olmaz.
[Aslında sizeof operatörünün ürettiği değer standart bir *typedef* türü olan *size_t* türündendir. Standart typedef türleri "Tür İsimleri Bildirimleri ve typedef Belirleyicisi " isimli bölümde ele alınıyor.]

sizeof İşlecinin Terimi Olan İfadenin Yan Etkisi

sizeof işlecinin terimi olan ifade yan etki göstermez. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

int func()
{
    printf("func()\n");

    return 1;
}

int main()
{
    unsigned int x = sizeof(func());
    printf("x = %u\n", x);

    return 0;
}
```

main işlevi içinde, *func* işlevi çağrılmaz. *sizeof* işleci, terimi olan ifadeye yalnızca bir tür bilgisi olarak bakar. Örnek kodda yer alan

```
func()
```

ifadesinin türü *int* türüdür.

sizeof İşleci Ne Amaçla Kullanılır

Belirli bir türden nesnenin bellekte kaç *byte* yer kaplayacağı, sistemden sisteme farklılık gösterebilir. Tür uzunluğu güvence altında bulunan tek doğal tür, *char* türüdür. *char* türden bir nesne tüm sistemlerde 1 *byte* yer kaplar. Tür uzunluklarının sistemden sisteme farklı olabilmesi, bazı uygulamalarda taşınabilirlik sorunlarına yol açabilir. *sizeof* işlecinin, genel olarak bu tür taşınabilirlik sorunlarını ortadan kaldırmaya yönelik olarak kullanıldığı söylenebilir.

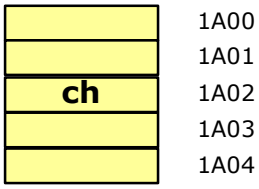
GÖSTERİCİLER

Birinci bölümde yazılan kodlarda nesnelerin değerleri kullanıldı. Örneğin değişkenlerin değerleri işlevlere argüman olarak gönderildi. İşlevler nesnelerin değerlerini geri döndürdü. "Göstericiler" (*pointers)* ile artık nesnelerin değerlerinin yanı sıra nesnelerin adresleri üzerinde de durulacak.
Nasıl, oturduğunuz evin adresinden söz ediliyorsa, programda kullandığınız nesnelerin de adreslerinden söz edilebilir. *Bir nesnenin adresi o nesnenin bellekteki konumunu gösteren bir bilgidir.*
İşte C dilinde yazılan birçok kod, nesnelerin adresi olan bilgileri kullanılır. Yazılımsal bazı amaçların gerçekleştirilmesi için, nesnelerin adresleri değişkenlerde saklanır, işlevlere gönderilir, işlev çağrılarıyla işlevlerden geri dönüş değeri olarak elde edilir.

Her nesne bellekte yer kapladığına göre belirli bir adrese sahiptir. Nesnelerin adresleri, sistemlerin çoğunda, derleyici ve programı yükleyen işletim sistemi tarafından ortaklaşa belirlenir. Nesnelerin adresleri program yüklenmeden önce kesin olarak bilinemez ve programcı tarafından da önceden saptanamaz. Programcı nesnelerin adreslerini ancak programın çalışması sırasında (*run time)* öğrenebilir. Örneğin:

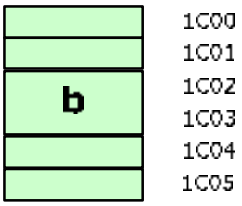
```
char ch;
```

Gibi bir tanımlamayla karşılaşan derleyici bellekte *ch* değişkeni için *1 byte* yer ayırır. Derleyicinin *ch* değişkeni için bellekte hangi *byte'ı* ayıracağı önceden bilinemez. Bu ancak programın çalışması sırasında öğrenilebilir. Yukarıdaki örnekte *ch* değişkeninin yerel yerel olduğunu düşünelim. *ch* değişkeni, tanımlanmış olduğu bloğun kodu yürütülmeye başlandığında yaratılır, bloğun kodunun yürütülmesi bittiğinde de ömrü sona erer. Aşağıdaki şekil *ch* değişkeninin *1A02* adresinde olduğu varsayılarak çizilmiştir:

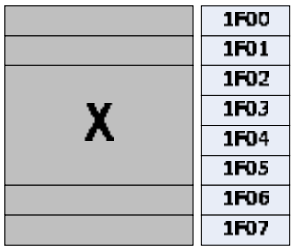


Tanımlanan nesne *1* byte'dan daha uzunsa, o zaman nesnenin adresi nasıl belirlenir?

```
int b;
```



1 *byte*'tan uzun olan nesnelerin adresleri, onların ilk *byte*'larının adresleriyle belirtilir. Yukarıdaki örnekte *b* değişkeninin adresi *1C02*'dir. Zaten *b* değişkeninin *int* türden olduğu bilindiğine göre diğer parçasının *1C03* adresinde olacağı da açıktır (*int* türden bir nesnenin ilgili sistemde *2 byte* yer kapladığı varsayılıyor).
Benzer biçimde *long* türden olan *y* değişkeninin bellekteki yerleşiminin aşağıdaki gibi olduğu varsayılırsa, adresinin *1F02* olduğu söylenebilir:



int türden bir değişken tanımlanmış olsun:

```
int x;
```

Böyle bir nesnenin adresinin yazılımsal olarak kullanılabilmesi için, bu bilginin bir biçimde ifade edilmesi, bellekte tutulması, yorumlanması gerekir.
x değişkeninin adresi olan bilginin de, bir türü olmalıdır. Böyle bir tür bilgisi, nesnenin kendi türünden türetilir (*derived type*) ve C'de aşağıdaki biçimde gösterilir:

```
(int *)
```

double türden bir nesnenin adresi olabilecek bir bilginin türü de

```
(double *)
```

olarak gösterilir.

x, T türünden bir nesne olmak üzere *x* nesnesinin adresi olan bilginin türünün

(T *) türü olduğu kabul edilir.

Gösterici Nesneler

Yazılımsal bazı amaçları gerçekleştirmek için nesnelerin adreslerinin de değişkenlerde tutulması gerekir. Nasıl bir tamsayı değeri tamsayı türlerinden bir değişkende tutuluyorsa, bir adres bilgisi de, türü bir adres bilgisi olan değişkende saklanır.

```
int x;
```

Yukarıda tanımlanan *x* nesnesinin türü *int* türüdür. Böyle bir nesnenin adresi olan bilgi *(int *)* türünden olduğuna göre bu bilgi doğal olarak *int* * türünden olan bir değişkende saklanmalıdır.

Gösterici Değişkenlerin Bildirimleri

```
int *ptr;
```

Yukarıda *ptr* isimli bir değişken tanımlanıyor. *ptr* değişkeni *int* türden bir nesnenin adresi olan bilgiyi tutabilecek bir değişkendir. *ptr* değişkeninin değeri, *int* türden bir nesnenin adresi olan bilgidir. *ptr* değişkenine *int* türden bir nesnenin adresi olabilecek bilgi atanmalıdır.
Benzer şekilde *double* türden bir nesnenin adresi olan bilgiyi saklamak için de

```
double *dp;
```

gibi bir değişken tanımlanabilir.
Gösterici değişkenler, adres bilgilerini saklamak ve adreslerle ilgili işlemler yapmak için kullanılan nesnelerdir. Gösterici değişkenlerin değerleri adrestir.
Gösterici değişkenlerin bildirimlerinin genel biçimi şöyledir:

```
<tür> *<gösterici ismi>;
```

<tür>, göstericinin (içindeki adresin) türüdür. *char, int, float*... gibi herhangi bir tür olabilir.

Burada * atomu tür bilgisinin bir parçasıdır.
Aşağıda örnek gösterici bildirimleri yer alıyor:

```
float *f;
char *ps;
int *dizi;
unsigned long *Pdword;
```

Gösterici bildirimleri, diğer türlere ilişkin bildirimlerden * atomu ile ayrılır.

```
char s;
```

bildiriminde *s, char* türden bir değişken iken

```
char *ps;
```

bildiriminde *ps, char* türden bir göstericidir, yani türü *char* * olan bir nesnedir. Bu değişkene *char* türden bir nesnenin adresi atanmalıdır. Böyle bir bildirimden şu bilgiler çıkarılabilir:

ps bir nesnedir, yani bellekte bir yer kaplar. *ps* nesnesi için bellekte ayrılan yerdeki *1*'ler ve *0*'lar *char* türden bir nesnenin adresinin, sayısal değeri olarak yorumlanır.
Tanımlamada yer alan '*' bir işleç değildir. Sözdizim kuralı olarak nesnenin bir gösterici olduğunu anlatır.
Gösterici bildirimleri ile normal bildirimler bir arada yapılabilir. Örneğin:

```
int *p, a;
```

Burada *p int* türden bir gösterici değişkendir, ama *a int* türden bir normal bir değişkendir. Aynı türden birden fazla göstericinin bildirimi yapılacaksa, araya virgül atomu konularak, her gösterici değişkenin bildirimi * atomu ile yapılmalıdır.

```
char *p1, *p2
```

Yukarıdaki bildirimde *p1* ve *p2 char* türden gösterici değişkenlerdir.

```
double *p1, *p2, d, a[20];
```

Yukarıdaki bildirimde *p*1 ve *p2 double* türden gösterici değişkenler, *d double* türden bir değişken ve *a* ise elemanları *double* türden *20* elemanlı bir dizidir.

Gösterici Değişkenlerin Uzunlukları

Bir gösterici nesnesinin tanımı ile karşılaşan derleyici –diğer tanımlamalarda yaptığı gibi- bellekte o gösterici değişkeni için yer ayırır. Derleyicilerin göstericiler için ayırdıkları yerlerin uzunluğu donanıma bağlı olup sistemden sisteme değişebilir. *32* bit sistemlerde (örneğin *UNIX* ve *Windows 3.1* sonrası sistemlerde) gösterici değişkenler *4 byte* uzunluğundadır. *8086* mimarisinde ve *DOS* altında çalışan derleyicilerde ise gösterici değişkenler 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*) denir.
Göstericilerin uzunlukları türlerinden bağımsızdır.

Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    char *cp, ch;
    int *ip, i;
    double *dp, d;

    printf("sizeof(ch) = %u\n", sizeof(ch));
    printf("sizeof(i) = %u\n", sizeof(i));
    printf("sizeof(d) = %u\n", sizeof(d));
    printf("sizeof(cp) = %u\n", sizeof(cp));
    printf("sizeof(ip) = %u\n", sizeof(ip));
    printf("sizeof(dp) = %u\n", sizeof(dp));

    printf("sizeof(char *) = %u\n", sizeof(char *));
    printf("sizeof(int *) = %u\n", sizeof(int *));
    printf("sizeof(double *) = %u\n", sizeof(double *));

    return 0;
}
```

Yukarıdaki programda hem *char*, *int*, *double* türlerinden hem de *char* *, *int* *, *double* * türlerinden nesnelerin tanımlandığını görüyorsunuz. Daha sonra *printf* işleviyle bu nesnelerin *sizeof* değerleri ekrana yazdırılıyor. *T* türünden bir nesnenin *sizeof* değeri ne olursa olsun *T** türünden bir nesnenin *sizeof* değeri hep aynıdır, değil mi? Yukarıdaki program *UNIX* işletim sistemi için derlenip çalıştırıldığında ekran çıktısı aşağıdaki gibi olur:

```
sizeof(ch) = 1
sizeof(i) = 4
sizeof(d) = 8
sizeof(cp) = 4
sizeof(ip) = 4
sizeof(dp) = 4
sizeof(char *) = 4
sizeof(int *) = 4
sizeof(double *) = 4
```

Adres Bilgisi Olan İfadeler

Bazı ifadeleri adres türündendir. Yani bu ifadelerin değeri adrestir. Bir gösterici değişkene, türü adres olan ifade yani bir adres değeri atanmalıdır.

```
int *ptr;
```

gibi tanımlanan bir değişken, içinde *int* türden bir değişkenin adresi olan bilgiyi saklayacak değişkendir. Böyle bir adres bilgisi *ptr* değişkenine nasıl atanabilir?

Adres Değişmezleri

`1200`,*int* türden bir tamsayı değişmezidir. Böyle bir değişmez, örneğin *int* türden bir nesneye atanabilir:

```
int x = 1200;
```

Tür dönüştürme işlemiyle bir tamsayı değişmezi bir adres bilgisine dönüştürülebilir:

```
(int *)1200
```

Yukarıdaki ifadenin türü *(int *)* türüdür. Böyle bir ifade *int* türden bir nesnenin adresi olabilecek bir bilgidir. *int* türden *1200* değişmezi tür dönüştürme işleciyle *int* türden bir nesnenin adresi olabilecek bir türe dönüştürülmüştür. Adres değişmezlerinin yazımında geleneksel olarak onaltılık sayı sistemi kullanılır:

```
(double *)0x1AC4
```

Yukarıdaki ifade, *double* türden bir nesnenin adresi olabilecek bir bilgidir. Adres bilgisinin tamsayı kısmının yazılmasında onaltılık sayı sisteminin kullanıldığını görüyorsunuz.

Göstericilerle İlgili Tür Uyumu

Bir gösterici değişkene aynı türden bir adres bilgisi yerleştirilmelidir. Örneğin :

```
int *p;
p = 100;
```

Burada p gösterici değişkenine adres olmayan bir değer atanıyor. Böyle bir atamanın yanlış olduğu kabul edilir. Derleyicilerin hemen hepsi bu durumu mantıksal bir uyarı iletisi ile bildirir. Bu durum ileride ayrıntılı olarak ele alınacak.
[Böyle bir atama C++ dilinde geçerli değildir.

```
int *p;
p = (char *) 0x1FC0;
```

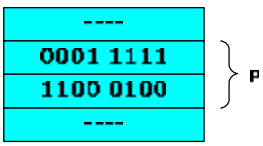
Burada *int* türden *p* göstericisine *char* türden bir adres bilgisi atanıyor. Yanlış olan bu durum da derleyicilerin çoğu tarafından mantıksal bir uyarı iletisi ile işaretlenir.
[C++ derleyicileri böyle bir atama durumunda da sözdizim hatası iletisi vererek amaç dosya üretimini reddeder]

```
int *p;
p = (int *) 0x1FC4;   /* geçerli ve uygun bir atamadır */
```

Bir adres bilgisi göstericiye atandığında adresin sayısal bileşeni gösterici içine yerleştirilir.

```
int *p;
p = (int *) 0x1FC4;
```

Burada bellekte *p* gösterici değişkeninin tutulduğu yere *0x1FC4* sayısal değeri yerleştirilir.



Gösterici değişkenler içlerinde adres bilgileri taşıdığına göre bir göstericiye aynı türden başka bir gösterici değişkenin değerinin atanması da tamamen uygundur.

```
int *p, *q;

p = (int *) 0x1AA0;
q  = p;
```

Yukarıdaki atama ile *q* göstericisine *p* göstericisinin değeri atanıyor. Yani bu atama deyiminden sonra q göstericisinin de içinde *(int *) 0x1AA0* adresi bulunur.

```
int k;
```

gibi bir tanımlama yapıldığında *k* değişkeni *int* türdendir. İçindeki değer *int* türden bir değer olarak yorumlanır.

```
20
```

gibi, tek bir değişmezden oluşan bir ifade de *int* türdendir, çünkü *20 int* türden bir değişmezdir. Başka bir deyişle *k* ifadesiyle *20* ifadesinin türleri aynıdır. Her iki ifadenin türü de *int* türüdür. Ancak *k* ifadesi nesne gösteren bir ifade iken *20* ifadesi nesne göstermeyen bir ifadedir, yani bir sağ taraf değeridir.

Yine bir adres değişmezinden oluşan

```
(int *) 0x1A00
```

ifadesinin türü de *int* türden bir adrestir, yani *(int *)* türünden bir ifadedir. Ancak bu ifade de sol taraf değeri değildir.

Görüldüğü gibi gösterici değişkenleri, belirli bir adres türünden nesnelerdir. Yani değerleri adres olan değişkenlerdir.

Gösterici İşleçleri

C dilinin bazı işleçleri adres bilgileri ile ilgili olarak kullanılır. Göstericiler ile ilgili kodlar bu işleçleri kullanır. Gösterici işleçleri şunlardır:

*	içerik işleci	indirection operator (dereferencing operator)
&	adres işleci	address of operator
[ ]	köşeli ayraç işleci	index operator (subscript operator)
->	ok işleci	arrow operator

ok işleci yapı türünden adreslerle kullanıldığı için bu işleç "yapılar" konusunda ayrıntılı olarak incelenecek.

Adres İşleci

Adres işleci *(adress of operator),* önek konumunda tek terimli (*unary prefix*) bir işleçtir. İşleç öncelik tablosunun ikinci seviyesinde yer alır. Bu işlecin ürettiği değer, terimi olan nesnenin adresidir. Adres işlecinin terimi mutlaka bir nesne olmalıdır. Çünkü yalnızca nesnelerin -sol taraf değerlerinin- adres bilgilerine ulaşılabilir. *Adres* işlecinin teriminin nesne olmayan bir ifade olması geçersizdir.

```
int k;
```

gibi bir tanımalamadan sonra yazılan

```
&k
```

ifadesini ele alalım. Bu ifadenin ürettiği değer *int* türden bir adres bilgisidir.
Bu ifadenin türü *(int *)* türüdür.

& işleci diğer tek terimli işleçler gibi, işleç öncelik tablosunun 2. seviyesinde bulunur. Bu öncelik seviyesinin öncelik yönünün "sağdan sola" olduğunu biliyorsunuz.
Bir gösterici değişkeni, içinde bir adres bilgisi tutan bir nesne olduğuna göre, bir gösterici değişkene adres işlecinin ürettiği bir adres bilgisi atanabilir.

```
int x = 20;
int *ptr;

ptr = &x;
```

Böyle bir atamadan sonra şunlar söylenebilir:
ptr nesnesinin değeri *x* değişkeninin adresidir. *ptr* nesnesi *x* değişkeninin adresini tutar.
Adres işleci ile elde edilen adres, aynı türden bir gösterici değişkene atanmalıdır. Örneğin aşağıdaki programda bir gösterici değişkene farklı türden bir adres atanıyor:

```
char ch = 'x';
int *p;

p = &ch;    /* Yanlış */
```

Tabi bu işlecin ürettiği adres bilgisi de sol taraf değeri değildir. Örneğin:

```
int x;

++&x    /* Geçersiz */
```

gibi bir işlem hata ile sonuçlanır. Artırma işlecinin terimi nesne olmalıdır. Yukarıdaki ifadede ++ işlecinin terimi olan *&x* ifadesi bir nesne değildir. Yalnızca bir adres değeridir.

Adres Değerlerinin Ekrana Yazdırılması

Standart *printf* işlevi ile doğal veri türlerinden ifadelerin değerlerinin ekrana yazdırılabileceğini biliyorsunuz. Bir ifadenin değerini ekrana yazdırmak için, *printf* işleviyle birinci argüman olarak geçilen dizge içinde önceden belirlenmiş format karakterlerinin (*conversion specifiers*) kullanıldığını hatırlayın. Acaba bir adres bilgisi de uygun format karakteri kullanılarak ekrana yazdırılabilir mi? Evet! Standart *printf* işlevinde bu amaç için *%p* format karakterleri kullanılır. *%p* format karakterleri ile eşlenen argüman bir adres bilgisi ise, *printf* işlevi ilgili adres bilgisinin yalnızca sayısal bileşenini onaltılık sayı sisteminde ekrana yazdırır.
Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    int *ptr;
    int x = 20;

    ptr = &x;
    printf("x nesnesinin adresi = %p\n", &x);
    printf("ptr değişkeninin değeri = %p\n", ptr);
    printf("ptr nesnesinin adresi = %p\n", &ptr);

    return 0;
}
```

ptr bir nesne olduğu için *ptr* nesnesi de adres işlecinin terimi olabilir, değil mi? *ptr* nesnesinin değeri olan adres, *x* nesnesinin adresidir. Ama *ptr* nesnesinin kendi adresinden de söz edilebilir. Bir gösterici değişkenin değeri olan adres ile gösterici değişkenin kendi adresi farklı şeylerdir.

```
printf("ptr nesnesinin adresi = %p\n", &ptr);
```

çağrısıyla *ptr* değişkeninin kendi adresi ekrana yazdırılıyor.

Dizi İsimlerinin Adres Bilgisine Dönüştürülmesi

C dilinde dizi isimleri bir işleme sokulduğunda derleyici tarafından otomatik olarak bir adres bilgisine dönüştürülür.

```
char s[5];
```

gibi bir dizi tanımlamasında sonra, dizinin ismi olan *s* bir işleme sokulduğunda bu dizinin ilk elemanının adresine dönüştürülür.
Dizi isimleri derleyici tarafından, diziler için bellekte ayrılan blokların başlangıç yerini gösteren bir adres bilgisine dönüştürülür. Yukarıdaki örnekte dizinin bellekte aşağıdaki şekilde yerleştirildiğini düşünün:

s[0]	**1C00**
s[1]	**1C01**
s[2]	**1C02**
s[3]	**1C03**
s[4]	**1C04**

Bu durumda dizi ismi olan s, *char* türden *1C00* adresine eşdeğerdir. Yani bu adresi bir adres değişmezi şeklinde yazılmış olsaydı:

```
(char *)0x1COO
```

biçiminde yazılırdı.

Bu durumda *s* ifadesi ile *&s[0]* ifadesi aynı adres bilgisidir, değil mi?

Gösterici değişkenlere kendi türlerinden bir adres bilgisi atamak gerektiğine göre aşağıdaki atamaların hepsi geçerli ve doğrudur:

```
int a[100];
long l[20];
char s[100];
double d[10];
int *p;
long *lp;
char *cp;
double *dp;
p = a;
lp = l;
cp = s;
dp = d;
```

Bir göstericiye yalnızca aynı türden bir dizinin ismi atanabilir. Örneğin:

```
int *p;
char s[] = "Necati";
p = s;      /YANLIŞ */
```

Dizi İsimleri Nesne Göstermez

```
int a[100];
int *ptr;
```

gibi bir tanımlamadan sonra

```
a
```

gibi bir ifade kullanılırsa, bu iafade derleyici tarafından otomatik olarak *int* * türüne dönüştürülür. Yani bu ifadenin türü de *(int *)* türüdür.

```
ptr
```

ifadesi nesne gösteren bir ifadeyken, yani bir sol taraf değeriyken,

```
a
```

ifadesi nesne göstermeyen bir ifade değeridir. Değiştirilebilir sol taraf değeri (*modifiable L value*) olarak kullanılamaz. Örneğin

```
a++
```

ifadesi geçersizdir.
C dilinde hiçbir değişkenin ya da dizinin programın çalışma zamanında bulunacağı yer programcı tarafından belirlenemez. Programcı değişkeni tanımlar, derleyici onu herhangi bir yere yerleştirebilir.

Dizi isimleri göstericiler gibi sol taraf değeri olarak kullanılamaz. Örneğin, *s* bir dizi ismi olmak üzere

```
++s;
```
deyimi geçersizdir.

* İçerik İşleci

İçerik işleci (*indirection operator / dereferencing operator*) de, önek konumunda bulunan tek terimli (*unary prefix*) bir işleçtir. İçerik işlecinin terimi bir adres bilgisi olmalıdır. Bir adres ifadesi, * işlecinin terimi olduğunda, elde edilen ifade bellekte o adreste bulunan, nesneyi temsil eder. Dolayısıyla, * işleci ile oluşturulan bir ifade bir nesneye karşılık gelir, sol taraf değeri olarak kullanılabilir.

```
int a;
```

gibi bir bildirimde a nesnesinin türü *int* türüdür. Çünkü *a* nesnesi içinde *int* türden bir veri tutulur.

```
int *p;
```

bildiriminde *p*'nin türü *int* türden bir adrestir. Yani *p* nesnesinin türü (*int* *) türüdür. *p* nesnesinin içinde (*int* *) türünden bir veri tutulur.

```
char *ptr;
```

gibi bir bildirimden iki şey anlaşılır:

ptr char türden bir göstericidir. İçine *char* türden bir adres bilgisi yerleştirilmek için tanımlanmıştır. *ptr* göstericisi * işleci ile birlikte kullanıldığında elde edilen nesne *char* türdendir. Yani **ptr char* türden bir nesnedir.
Örneğin:

```
int *p;

p = (int *) 0x1FC2;
*p = 100;
```

Burada **p*'nin türü *int* türüdür. Dolayısıyla **p = 100* gibi bir işlemden (DOS altında) yalnızca 0x1FC2 byte'ı değil, *0x1FC2* ve *0x1FC3 byte*'larının her ikisi birden etkilenir. Göstericinin içindeki adresin sayısal bileşeni nesnenin düşük anlamlı *byte*'ının adresini içerir. Bu durumda bir gösterici değişkene, bellekteki herhangi bir bölgenin adresi atanabilir. Daha sonra * işleci ile o bellek bölgesine erişilebilir.

* işlecinin terimi bir adres bilgisi olmak zorundadır. Yani terim adres değişmezi olabilir, dizi ismi olabilir, bir gösterici değişken olabilir veya adres işleci ile elde edilmiş bir adres ifadesi olabilir.

İçerik işleci yalnız gösterici nesneleriyle değil, adres bilgisinin her biçimi ile (adres değişmezleri ve dizi isimleri vs.) kullanılabilir. Bu işleç, terimi olan adresteki nesneye erişmekte kullanılır. İşleç ile elde edilen değer, terimi olan adreste bulunan nesnenin değerdir.
İçerik işleci ile üretilen bir ifade nesne belirtir. Nesnenin türü terim olan nesnenin adresi ile aynı türdendir.

Aşağıdaki programı derleyerek çalıştırın, ekran çıktısını yorumlayın:

```
#include <stdio.h>

int main()
{
    char s[] = "Balıkesir";
```

```
    int a[] = {1, 2, 3, 4, 5};
    int x = 10;
    int *ptr;

    putchar(*s);
    printf("%d\n", *a);
    *&x = 20;
    printf("x = %d\n", x);
    ptr = &x;
    *ptr = 30;
    printf("x = %d\n", x);

    return 0;
}
```

Yukarıdaki programda,

i) s *char* türden bir dizinin ismi olduğuna göre *char* türden bir adrese dönüştürülür. Bu adres *s* dizisinin başlangıç adresidir. **s* ifadesi bu adresteki nesne olduğuna göre, **s* ifadesi dizimizin ilk elemanı olan nesnedir, yani **s* ifadesinin değeri *'B'*dir, değil mi?

ii) *a* int türden bir dizinin ismi olduğuna göre *int* türden bir adrese dönüştürülür. Bu adres *a* dizisinin başlangıç adresidir. **a* ifadesi bu adresteki nesne olduğuna göre, **a* ifadesi *int* türden dizimizin ilk elemanı olan nesnedir, yani **a* ifadesi *a[0]* nesnesidir. Bu nesnenin değeri *1*'dir.

iii) **&x* ifadesinde ise iki ayrı işleç kullanılıyor. Adres ve içerik işleçleri. Bu işleçlerin her ikisi de işleç öncelik tablosunda ikinci seviyede yer alıyor. İşleç öncelik tablosunun ikinci seviyesine ilişkin öncelik yönü sağdan sola olduğuna göre, ifadenin değerlendirilmesinde önce adres işleci değer üretir. Adres işlecinin ürettiği değer *x* nesnesinin adresidir, içerik işlecinin terimi bu adres olur. İçerik işleci o adresteki nesneye ulaştığına göre **&x* ifadesi *x* nesnesinin adresindeki nesne, yani *x* nesnesinin kendisidir.

iv) *ptr* göstericisine *x* nesnesinin adresi atanıyor. İçerik işlecinin terimi *ptr* nesnesi olduğundan, *ptr* nesnesinin değeri olan adresteki nesneye ulaşılır. Bu durumda da **ptr* nesnesi yine *x* nesnesinin kendisidir, değil mi?

İçerik işlecinin öncelik tablosunun ikinci düzeyinde olduğunu biliyorsunuz. *s* bir dizi ismi olmak üzere

```
*s + 1;
```

ifadesinde önce içerik işleci değer üretir. İçerik işlecinin ürettiği değer toplama işlecinin terimi olur. Oysa ifade

```
*(s + 1)
```

biçiminde olsaydı önce + işleci ele alınırdı.

Derleyiciler * atomu içeren bir ifadede * atomunun çarpma işleci mi yoksa adres işleci mi olduğunu ifade içindeki kullanımına bakarak anlar. Çarpma işleci iki terimli iken içerik işleci tek terimli önek konumunda bir işleçtir.

```
*s * 2
```

ifadesinde birinci '*' içerik işleci iken ikincisi * aritmetik çarpma işlecidir.

Bir Göstericinin Bir Nesneyi Göstermesi

```
int x = 20;
int *ptr ;

ptr = &x;
*ptr = 30;
```

Yukarıdaki kod parçasında *ptr* göstericisine *int* türden *x* nesnesinin adresi atanıyor. Bu atamadan sonra *ptr* göstericisinin değeri *x* nesnesinin adresidir. Bu durumda "*ptr x* değişkenini gösteriyor" denir.
ptr x'i gösteriyor ise **ptr, x* nesnesinin kendisidir. Daha genel bir söyleyişle, *ptr* bir gösterici değişken ise **ptr* o göstericinin gösterdiği nesnedir!

Tanımlanan bir değişkene değişkenin ismiyle doğrudan ulaşabildiğiniz gibi, onu gösteren bir göstericiyi içerik işlecine terim yaparak dolaylı bir biçimde ulaşabilirsiniz, değil mi? İşlecin İngilizce ismi olan *"indirection operator"* de bu durumu vurgular.

Parametre Değişkeni Gösterici Olan İşlevler

Göstericiler daha çok bir işlevin parametre değişkeni olarak kullanılır. Bir gösterici bir nesne olduğuna göre bir işlevin parametre değişkeni herhangi bir türden gösterici olabilir:

```
void func(int *p)
{
    /***/
}
```

İşlevlerin parametre değişkenleri, işlev çağrılarıyla kendilerine geçilen argüman ifadeleriyle ilkdeğerlerini aldığına göre, bir işlevin parametre değişkeni bir gösterici ise işlev de aynı türden bir adres bilgisi ile çağrılmalıdır.
Böyle bir işlev, parametre değişkenine adresi kopyalanan yerel bir değişkenin değerini değiştirebilir:

```
#include <stdio.h>

void func(int *ptr)
{
    *ptr = 20;
}

int main()
{
    int a = 10;

    func(&a);
    printf("%d\n", a);

    return 0;
}
```

Yukarıdaki örnekte *main* işlevi içinde tanımlanan yerel *a* isimli değişkenin adresi *func* işlevine gönderiliyor. *func* işlevi çağrıldığında, yaratılan parametre değişkeni *ptr* ilkdeğerini *&a* ifadesinden alır. İşlevin koduna geçildiğinde artık parametre değişkeni olan *ptr* gösterici değişkeni, adresi gönderilen *a* nesnesini gösterir. Bu durumda

```
*ptr
```

ifadesi a nesnesinin kendisidir.

```
*ptr = 20;
```

deyimiyle *a* nesnesine *20* değeri atanır, değil mi?

Bir işlev bir değer elde edip çağıran işleve bu değeri iletmek isterse iki yöntem kullanılabilir:

i. Elde edilen değer çağrılan işlev tarafından geri dönüş değeri olarak üretilir.
ii. Elde edilen değer çağıran işlevin göndermiş olduğu adrese yerleştirilir. Tabi bunun için çağrılan işlevin parametre değişkeninin bir gösterici olması gerekir. Aşağıda kendisine gönderilen bir sayının faktoriyelini hesaplayarak bu değeri parametre olarak gönderilen adrese kopyalayan bir işlev yazılıyor:

```
#include <stdio.h>

void factorial(int n, long *p);

int main()
{
    long a;
    int k;

    for (k = 0; k < 14; ++k) {
        factorial(k, &a);
        printf ("%2d! = %ld\n", k, a);
    }

    return 0;
}

void factorial(int n, long *p)
{
    *p = 1;

    if (n == 0 || n == 1)
        return;

    while (n  > 1)
        *p *= n--;
}
```

a bir yerel değişken olsun. C dilinde bir işlev

```
func(a);
```

biçiminde çağırılmışsa, çağrılan bu işlevin, a değişkenini değiştirme şansı yoktur. Bu tür işlev çağrısına "değer ile çağırma" (*call by value*) denir. "İşlevler" konusunda anımsayacağınız gibi bu durumda *a* değişkeninin değeri *func* işlevinin parametre değişkenine kopyalanarak aktarılır. Yani *func* işlevinin parametre değişkeni *x* nesnesinin kendisi değildir. *func* işlevinin kodu görülmese de işlev çağrısından sonra yerel *a* değişkeninin değerinin değişmediği söylenebilir.

İşlevin *a* değişkeninin değiştirebilmesi için

```
func(&a);
```

biçiminde çağrılması gerekir. Örneğin standart *scanf* işlevine & işleci ile bir nesnenin adresinin argüman olarak yollanmasının nedeni budur. Bu biçimde yapılan işlev çağrısına

C'de *"adres ile çağırma"* (*call by reference*) denir. Böyle bir çağrıda işleve bir nesnenin adresi gönderilir. İşlevin parametre değişkeni de bu adresi tutacak bir gösterici olur.

int türden iki yerel nesnenin değerleri takas edilmek istensin. Bu iş, bulunulan işlev içinde aşağıdaki gibi yapılabilir:

```
int main()
{
    int a = 10, b = 20, temp;

    temp = a;
    a = b;
    b = temp;
    /*....*/
}
```

Takas işleminin bir işlev tarafından yapılması istenirse, aşağıdaki gibi bir işlev iş görür müydü?

```
void swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}

int main()
{
    int a = 10, b = 20;
    swap(a, b);
    printf("a = %d\nb = %d\n", a, b);

    return 0;
}
```

Yukarıdaki program çalıştırıdığında ekrana

```
a = 10
b = 20
```

yazar! Yazılan *swap* işlevi *a* ve *b* değişkenlerinin değerlerini değiştirmez. Yerel nesneler olan *a* ve *b* değişkenlerinin değerleri ancak bu değişkenlerin adresleri bir işleve gönderilerek değiştirilebilirdi. Oysa yukarıdaki *swap* işlevi *a* ve *b* değişkenlerinin değerlerini parametre değişkenleri olan x ve y değişkenlerine kopyalıyor. Yani değerleri değiştirilen parametre değişkenleri olan *x* ve *y*.
Takas işlemini yapacak işlev kendisini çağıracak kod parçasından adres değerleri alacağı için gösterici parametre değişkenlerine sahip olmalı:

```
void swap(int *p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

Adreslerle İşlemler - Adreslerin Artırılması ve Eksiltilmesi (Gösterici Aritmetiği)

C dilinde bir adres bilgisi bir tamsayı ile toplanabilir, bir adres bilgisinden bir tamsayı çıkartılabilir. Böyle bir ifade toplanan ya da çıkartılan adres türündendir. Örneğin *int* türden bir nesnenin adresi ile 1 tamsayısı toplanırsa yine *int* türden bir nesnenin adresi bilgisi elde edilir.
Bir adres bilgisine *1* toplandığında, adresin sayısal bileşeni adrese sahip nesnenin türünün uzunluğu kadar artar. Bu durumda örneğin DOS işletim sisteminde *char* türden bir göstericinin değeri, *1* artırıldığında adresin sayısal bileşeni *1*, *int* türden bir gösterici *1* artırıldığında ise adresin sayısal bileşeni *2* artar, *double* türden bir gösterici *1* artırıldığında ise adresin sayısal bileşeni *8* artar.
Bir gösterici değişkenin bellekte bir nesneyi gösterdiğini düşünelim. Bu gösterici değişkenin değeri 1 artırılırsa bu kez gösterici değişkeni, gösterdiği nesneden bir sonraki nesneyi gösterir duruma gelir.

```
#include <stdio.h>

int main()
{
    int k;
    int a[10];

    for (k = 0; k < 10; ++k) {
        *(a + k) = k;
        printf("%d ", a[k]);
    }

    return 0;
}
```

Yukarıdaki örnekte *main* işlevi içinde tanımlanan *a* dizisinin elemanlarına gösterici aritmetiği kullanılarak ulaşılıyor.

```
*(a + k)
```

a adresinden *k* uzaklıktaki nesne anlamına gelir. Bu da dizinin *k* indisli elemanıdır, değil mi? Dizi *int* türden değil de *double* türden olsaydı dizinin elemanlarına yine böyle ulaşılabilirdi, değil mi?
Gösterici aritmetiği türden bağımsız bir soyutlama sağlar.

İki adres bilgisinin toplanması geçersizdir. Ancak aynı dizi üzerindeki iki adres bilgisi birbirinden çıkartılabilir. İki adres birbirinden çıkartılırsa sonuç bir tamsayı türündendir. İki adres birbirinden çıkartıldığında önce adreslerin sayısal bileşenleri çıkartılır, sonra elde edilen değer adresin ait olduğu türün uzunluğuna bölünür. Örneğin *a int* bir dizi olmak üzere türden bir adres olmak üzere:

&a[2] - &a[0] ifadesinden elde edilen değer *2* dir.

Aşağıdaki programda gösterici aritmetiği sorgulanıyor. Programı derleyerek çalıştırın ve ekran çıktısını inceleyerek yorumlamaya çalışın:

```
#include <stdio.h>

int main()
{
    char s[10];
    int a[10];
    double d[10];
```

```
    printf("%p\n", (char *)0x1AC0 + 1);
    printf("%p\n", (int *)0x1AC0 + 1);
    printf("%p\n", (double *)0x1AC0 + 1);
    printf("%d\n", &s[9] - &s[0]);
    printf("%d\n", &a[9] - &a[0]);
    printf("%d\n", &d[9] - &d[0]);

    return 0;
}
```

Adres değerlerinin karşılaştırılması

Aynı blok üzerindeki iki adres, karşılaştırma işleçleriyle karşılaştırılabilir:

```
#include <stdio.h>

int main()
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int *ptr = a;

    while (ptr < a + 10) {
        printf("%d ", *ptr);
        ++ptr;
    }
    return 0;
}
```

[] Köşeli Ayraç İşleci :

Daha önce dizi elemanlarına erişmekte kullandığımız köşeli ayraç aslında iki terimli bir gösterici işlecidir. Köşeli ayraç işleci *(index / subscript operator)* işleç öncelik tablosunun en yüksek öncelik seviyesindedir. İşlecin birinci terimi köşeli ayraçtan önce yer alır. Bu terim bir adres bilgisi olur. İkinci terim ise köşeli ayraç içine yazılacak tam sayı türünden bir ifade olur.

```
p[n]
```

ifadesi ile

```
*(p + n)
```

tamamen eşdeğer ifadelerdir.
Yani köşeli ayraç işleci, bir adresten *n* ilerideki nesneye erişmek için kullanılır. [] işleci ile elde edilen nesnenin türü terimi olan adresin türü ile aynı türdendir. Aşağıdaki programın ekrana ne yazdıracağını önce tahmine etmeye çalışın. Daha sonra programı derleyip çalıştırın:

```
#include <stdio.h>

int main()
{
    int a[5] = {1, 2, 3, 4, 5};
    int *ptr = a + 2;

    printf("%d\n", ptr[-1]);
    printf("%d\n", ptr[0]);
    printf("%d\n", ptr[1]);

    return 0;
}
```

[] işlecinin birinci terimi dizi ismi olmak zorunda değildir. Daha önce de belirtildiği gibi bir dizinin ismi bir ifade içinde kullanıldığında derleyici tarafından o dizinin ilk elemanının adresine yani dizinin başlangıç adresine dönüştürülür.

[] işleci işleç öncelik tablosunun en yüksek düzeyinde bulunur. Örneğin:

```
&p[n]
```

ifadesinde önce köşeli ayraç işleci değer üretir. İşlecin ürettiği değer, bir nesneye ilişkindir. Adres işlecinin öncelik seviyesi köşeli ayraç işlecinden daha düşük olduğu için, işlecin ulaştığı nesne bu kez adres işlecinin terimi olur.

Şüphesiz [] içindeki ifadenin sayısal değeri negatif olabilir. Örneğin

```
p[-2]
```

geçerli bir ifadedir. Benzer şekilde bu ifade

```
*(p - 2)
```

ifadesi ile aynı anlamdadır.

Aşağıdaki örnekte köşeli ayraç işleci ile adres işleci aynı ifade içinde kullanılıyor.

```
#include <stdio.h>

int main()
{
    char ch = 'A';

    (&ch)[0] = 'B'
    putchar(ch);

    return 0;
}
```

++ ve -- İşleçlerinin Gösterici İşleçleriyle Birlikte Kullanılması

C dilinin bir çok kod kalıbında gösterici işleçleriyle ile artırma ya da eksiltme işleci birlikte kullanılır.

1. İçerik işleci ile ++ işlecinin aynı ifade içinde yer alması
*a) ++*p* durumu

```
#include <stdio.h>

int main()
{
    int a[5] = {1, 2, 3, 4, 5};
    int k;
    int *ptr = a;

    ++*ptr;

    for (k = 0; k < 5; ++k)
        printf("%d ", a[k]);   /* 2 2 3 4 5 */

    return 0;
}
```

++**ptr* ifadesinde iki işleç kullanılıyor: İçerik işleci ile artırma işleci. Her iki işleç de işleç öncelik tablosunun ikinci seviyesinde bulunur. İkinci seviyenin öncelik yönü sağdan sola olduğuna göre önce daha sağda bulunan içerik işleci değer üretir. İçerik işleci *ptr* göstericisinin gösterdiği nesneye ulaşır böylece bu nesne, artırma işlecine terim olur. Bu durumda *ptr* göstericisinin gösterdiği nesnenin değeri *1* artırılır. Kısaca

```
++*ptr;
```

deyimi , "*ptr*'nin gösterdiği nesnenin değerini *1* artır" anlamına gelir.

**++p* durumu
p göstericisinin 1 fazlası olan adresteki nesneye ulaşılır. Yani ifadenin değeri p göstericisinin gösterdiği nesneyi izleyen nesnenin değeridir. Tabi ifadenin değerlendirilmesinden sonra ++ işlecinin yan etkisinden dolayı *p* göstericisinin değeri *1* artırılır. Yani *ptr* bir sonraki nesneyi gösterir. Aşağıdaki örneği dikkatle inceleyin:

```
#include <stdio.h>

int main()
{
    int a[5] = {1, 2, 3, 4, 5};
    int k;
    int *ptr = a;

    *++ptr = 10;
    *ptr = 20;

    for (k = 0; k < 5; ++k)
        printf("%d ", a[k]);   /* 1 20 3 4 5 */

    return 0;
}
```

```
x = *++p;
```

deyimi ile *x* değişkenine artırılmış adresteki bilgi atanır.

**p++* durumu
++ işleci ve * işlecinin ikisi de ikinci öncelik seviyesindedir. Bu öncelik seviyesine ilişkin öncelik yönü sağdan soladır. Önce ++ işleci ele alınır ve bu işleç ifadenin geri kalan kısmına *p* göstericisinin artmamış değerini üretir. Bu adresteki nesneye ulaşılır daha sonra *p* göstericisinin değeri *1* artırılır. **p++* ifadesinin değeri *p* göstericisinin gösterdiği

nesnenin değeridir. ++ işlecinin yan etkisinden dolayı ifadenin değerlendirilmesinden sonra *p* göstericisinin değeri *1* artırılır. Yani *p* bir sonraki nesneyi gösterir. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

int main()
{
    int a[5] = {1, 2, 3, 4, 5};
    int k;
    int *ptr = a;

    *ptr++ = 10;
    *ptr = 20;

    for (k = 0; k < 5; ++k)    /* 10 20 3 4 5 */
        printf("%d ", a[k]);

    return 0;
}
```

Adres işleci ile ++ işlecinin aynı ifade içinde yer alması
&x++ /* Geçersiz*/
x++ ifadesinin ürettiği değer adres işlecinin terimi olur. *x*++ ifadesinin ürettiği değer sol taraf değeri değildir. Adres işlecinin teriminin sol taraf değeri olması gerekir. Bu durumda derleme zamanında hata oluşur.

&++x /* Geçersiz */
++*x* ifadesinin ürettiği değer adres işlecinin terimi olur. ++*x* ifadesinin ürettiği değer sol taraf değeri değildir ve adres işlecinin teriminin sol taraf değeri olması gerekir. İfade geçersizdir.
[C++ dilinde önek ++ işlecini bir sol taraf değeri ürettiği için bu ifade C++ da geçerlidir.]

++&x /* Geçersiz */
Adres işlecinin ürettiği değer önek konumundaki artırma işlecinin terimi olur. ++ işlecinin teriminin nesne gösteren bir ifade olması gerekir. Oysa *&x* ifadesi nesne gösteren bir ifade değildir. İfade geçersizdir.

Adres işleci *(&)* ile artırma *(++)* ya da eksiltme *(--)* işleçlerinin her türlü bileşimi derleme zamanında hata oluşmasına neden olur.

Köşeli ayraç işleci ile ++ işlecinin aynı ifade içinde yer alması
++*p[i]* durumu
Köşeli ayraç işleci birinci öncelik seviyesinde, ++ işleci ise ikinci öncelik seviyesindedir. Bu durumda derleyici tarafından önce köşeli ayraç işleci ele alınır. *p[i]* ifadesi bir nesne gösterir. Dolayısıyla ++ işlecinin terimi olmasında bir sakınca yoktur. Söz konusu ifade

```
p[i] = p[i] + 1;
```

anlamına gelir. Yani *p[i]* nesnesinin değeri *1* artırılır.

p[i]++ durumu
```
x = p[i]++;
```

Önce p[i] nesnesinin artmamış değeri üretilir, ifadenin geri kalanında *p[i]* nesnesinin artmamış değeri kullanılır. Yani yukarıdaki örnekte *x* değişkenine p[i] nesnesinin artırılmamış değeri atanır, daha sonra p[i] nesnesi 1 artırılır.

p[++i] durumu
```
x = p[++i];
```

Önce ++i ifadesinin değeri elde edilir. Bu ifadenin değeri *i*'nin değerinin *1* fazlasıdır. Daha sonra *p* adresinden (*i* + *1*) uzaklıktaki nesneye ulaşılır. ++ işlecinin yan etkisi olarak *i* değişkeninin değeri *1* artırılır.
p[i++] durumu
```
x = p[i++];
```

Önce *i*++ ifadesinin değeri elde edilir. Bu ifadenin değeri *i*'nin kendi değeridir. Daha sonra *p* adresinden *i* uzaklıktaki nesneye ulaşılır. ++ işlecinin yan etkisi olarak *i* değişkeninin değeri *1* artırılır.

Gösterici Değişkenlere İlkdeğer Verilmesi

Diğer türden değişkenlerde olduğu gibi gösterici değişkenlere de tanımlanmaları sırasında ilkdeğer verilebilir. Göstericilere ilkdeğer verme işlemi göstericinin türünden bir adres bilgisi ile yapılmalıdır.
Örnekler:

```
char s[100];
double x;
int *ptr = (int *) 0x1A00;
char * str = (char *) 0x1FC0;
char *p = s;
double *dbptr = &x;
int i, *ptr = &i;
```

Son deyimde tanımlanan *i* isimli değişken *int* türden, *ptr* isimli değişken ise *int* * türdendir. *ptr* değişkenine aynı deyimle tanımlanan *i* değişkeninin adresi atanıyor.

Dizilerin İşlevlere Göstericiler Yoluyla Geçirilmesi

Bir dizinin başlangıç adresi ve boyutu bir işleve gönderilirse işlev dizi üzerinde işlem yapabilir. İşlevin dizinin başlangıç adresini alacak parametresi aynı türden bir gösterici değişken olmalıdır. İşlevin diğer parametresi dizinin boyutunu tutacak *int* türden bir değişken olabilir.
Bir dizinin başlangıç adresini parametre olarak alan işlev, dizi elemanlarına köşeli ayraç işleci ya da içerik işleci ile erişebilir. Ancak dizi elemanlarının kaç tane olduğu bilgisi işlev tarafından bilinemez. Bu nedenle dizi uzunluğu ikinci bir argüman olarak işleve gönderilir.
Örnek:

```
#include <stdio.h>

void display_array (const int *p, int size)
{
    int i;

    for (i = 0; i < size; ++i)
        printf("%d ", p[i]);
}

int main()
{
    int a[5] = {3, 8, 7, 6, 10};

    display_array(a, 5);

    return 0;
}
```

Yukarıda tanımlanan *display_array* işlevi *int* türden bir dizinin başlangıç adresini ve boyutunu alıyor, dizinin tüm elemanlarının değerlerini ekrana yazdırıyor. Parametre değişkeni olan *p* göstericisinin bildiriminde yer alan *const* anahtar sözcüğüne daha sonra değinilecek.
Aşağıda aynı işlev işini yaparken bu kez içerik işlevini kullanıyor:

```
void display_array (const int *p, int size)
{
    while (size--)
        printf("%d ", *p++);
}
```

Gösterici Parametre Değişkenlerinin Tanımlanması

Bir işlevin parametre değişkeninin gösterici olması durumunda, bu gösterici iki farklı biçimde tanımlanabilir:

```
void func(int *ptr);
void func(int ptr[]);
```

Derleyici açısından iki biçim arasında hiçbir farklılık yoktur. Ancak bazı programcılar, işlev dışarıdan bir dizinin başlangıç adresini istiyorsa ikinci biçimi tercih ederler:

```
void sort_array(int ptr[], int size);
```

Bu biçim yalnızca işlev paraemtresi olan göstericilere ilişkindir. Global ya da yerel göstericiler bu biçimde tanımlanamaz:

```
void foo(void)
{
    int ptr[]; /* Geçersiz */
}
```

Aşağıda *int* türden dizilerle ilgili bazı faydalı işlemler yapan işlevler tasarlanıyor. İşlevlerin tanımlarını dikkatli bir şekilde inceleyin. İşlevlerin bazılarının parametreleri olan göstericilerin bildiriminde *const* anahtar sözcüğünün kullanıldığını göreceksiniz. *const* anahtar sözcüğünü şimdilik gözönüne almayın. Bu anahtar sözcük ileride ayrıntılı bir biçimde ele alınacak.

Diziler Üzerinde İşlem Yapan İşlevlere Örnekler

Aşağıda bir dizinin elemanlarına rastgele değerler yerleştirmek amacıyla bir işlev tanımlanıyor:

```
void set_random_array(int *ptr, int size, int max_val)
{
    int k;

    for (k = 0; k < size; ++k)
        ptr[k] = rand() % (max_val + 1);
}
```

Bu işlev başlangıç adresini ve boyutunu aldığı dizinin elemanlarını *0 – max_val* aralığında rastgele değerlerle dolduruyor.

```
int sum_array(const int *ptr, int size)
{
    int sum = 0;
    int k;

    for (k = 0; k < size; ++k)
        sum += ptr[k];
    return sum;
}
```

sum_array işlevi dizinin elemanlarının toplamı değeriyle geri dönüyor. Dizinin tüm elemanlarının değeri *sum* isimli yerel nesneye katılıyor. İşlev *sum* nesnesinin değeri ile geri dönüyor.

```
int max_array(const int *ptr, int size)
{
    int max = *ptr;
    int k;

    for (k = 1; k < size; ++k)
        if (ptr[k] > max)
            max = ptr[k];

    return max;
}
```

```
int min_array(const int *ptr, int size)
{
    int min = *ptr;
    int k;

    for (k = 1; k < size; ++k)
        if (ptr[k] < min)
            min = ptr[k];

    return min;
}
```

max_array işlevi adresini ve boyutunu aldığı dizinin en büyük elemanının değeri ile geri dönüyor. *min_array* işlevi ise, benzer şekilde en küçük elemanın değeriyle geri dönüyor.

```
void sort_array(int *ptr, int size)
{
    int i, k, temp;

    for (i = 0; i < size - 1; ++i)
        for (k = 0; k < size - 1 - i; ++k)
            if (ptr[k] > ptr[k + 1]) {
                temp = ptr[k];
                ptr[k] = ptr[k + 1];
                ptr[k + 1] = temp;
            }
}
```

sort_array işlevi adresini ve boyutunu aldığı diziyi" kabarcık sıralaması" algoritmasıyla küçükten büyüğe doğru sıralıyor.
Aşağıda yazılan işlevleri sınayan bir *main* işlevi görüyorsunuz. Tüm işlevlerin tanımını *main* işlevinin tanımı ile birlikte derleyerek programı çalıştırın:

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

#define    SIZE       100

int main()
{
    int a[ARRAY_SIZE];

    srand(time(0));
    set_random_array(a, SIZE, 1000);
    printf("dizi 0 - 1000 araliginda rastgele sayilarla dolduruldu!\n");
    printf("dizi yazdiriliyor!\n");
    display_array(a, SIZE);
    printf("dizinin toplami = %d\n", sum_array(a, ARRAY_SIZE));
    printf("dizinin en buyuk elemani = %d\n", max_array(a, SIZE));
    printf("dizinin en kucuk elemani = %d\n", min_array(a, SIZE));
    sort_array(a, SIZE);
    printf("dizi siralama isleminden sonra yazdiriliyor!\n");
    display_array(a, SIZE);

    return 0;
}
```

İşlevlerin Kendilerine Geçilen Adres Bilgilerini Başka İşlevlere Geçmeleri

İşlevler başka işlevleri çağırabilir, çağırdıkları işlevlere kendi parametre değişkenlerine geçilen bilgileri argüman olarak gönderebilir. Şüphesiz bu durum parametreleri gösterici olan işlevler için de geçerlidir.
int türden bir dizinin aritmetik ortalamasını hesaplamak amacıyla *mean_array* isimli bir işlev tanımlayalım. Dizinin aritmetik ortalamasını bulmak için önce toplamını bulmak gerekir, değil mi?

```
double mean_array(const int *ptr, int size)
{
    return (double)sum_array(ptr, size) / size;
}
```

mean_array işlevi, kendisine geçilen dizi adresi ile dizi boyutunu, dizinin toplamını hesaplamak amacıyla *sum_array* işlevine argüman olarak geçiyor.

sort_array işlevinde ise, dizinin ardışık iki elemanı doğru sırada değil ise bu elemanlar takas ediliyor. Önce *int* türden iki nesnenin değerini takas edecek bir işlev yazalım. Yazdığımız bu işlevi *sort_array* işlevi içinde çağıralım:

```
void swap(int *p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

void sort_array(int *ptr, int size)
{
    int i, k;

    for (i = 0; i < size - 1; ++i)
        for (k = 0; k < size - 1 - i; ++k)
            if (ptr[k] > ptr[k + 1])
                swap(ptr + k, ptr + k + 1);
}
```

sort_array işlevinde

```
if (ptr[k] > ptr[k + 1])
    swap(ptr + k, ptr + k + 1);
```

deyimine dikkat edin. Bu deyimle, dışarıdan adresi alınan dizinin *k* indisli elemanı, *k + 1* indisli elemanından daha büyükse, dizinin *k* ve *k + 1* indisli elemanı olan nesnelerin değerleri *swap* işlevi çağrılarak takas ediliyor. *swap* işlevine argüman olarak iki nesnenin de adresi geçiliyor. *swap* işlevi aşağıdaki gibi de çağrılabilirdi, değil mi?

```
swap(&ptr[k], &ptr[k + 1]);
```

Aşağıda bir diziyi ters çeviren *reverse_array* isimli işlev tanımlanıyor:

```
void reverse_array(int *ptr, int size)
{
    int *pend = ptr + size - 1;
    int n = size / 2;

    while (n--)
        swap(ptr++, pend--);
}
```

pend gösterici değişkenine dizinin son elemanının başlangıç adresi atanıyor.
while döngüsü dizinin eleman sayısının yarısı kadar dönüyor. Döngünün her turunda, dizinin baştan *n*. elemanı ile sondan *n.* elemanı takas ediliyor. *ptr* ve *pend* gösterici değişkenleri, sonek konumunda olan ++ ve -- işleçlerinin terimi oluyor. Döngünün her turunda işleçlerin yan etkisi nedeniyle, *ptr* göstericisi bir sonraki nesneyi gösterirken, *pend* göstericisi bir önceki nesneyi gösteriyor.

Geri Dönüş Değeri Adres Türünden Olan İşlevler

Bir işlevin parametre değişkeni bir adres türünden olabildiği gibi, bir işlevin geri dönüş değeri de bir adres türünden olabilir. Böyle bir işlevin tanımında, işlevin geri dönüş değerinin türünün yazılacağı yere bir adres türü yazılır.
Örneğin *int* türden bir nesnenin adresini döndüren *func* isimli işlev aşağıdaki gibi tanımlanabilir:

```
int *func(void)
{
    /***/
}
```

func işlevinin geri dönüş değeri türü yerine *int* * yazıldığını görüyorsunuz. Yukarıdaki işlev tanımından * atomu kaldırılırsa, işlevin *int* türden bir değer döndürdüğü anlaşılır.
Adrese geri dönen bir işlev ne anlama gelir? İşlev çağrıldığı yere, *int* türden bir nesnenin adresini iletir. İşlev çağrı ifadesi, işlevin geri dönüş değerine yani bir adres bilgisine eşdeğerdir. İşlevin geri dönüş değeri bir nesnede saklanmak istenirse aynı türden bir gösterici değişkene atanmalıdır:

```
int *ptr;
ptr = func();
```

Benzer biçimde:

İşlevlerin geri dönüş değerlerini geçici bir nesne yardımıyla oluşturduklarını biliyorsunuz. Bir işlevin geri dönüş değerinin türü, geri dönüş değerini içinde taşıyacak geçici nesnenin türüdür. Bu durumda, bir adres türüne geri dönen bir işlevin, geri dönüş değerini içinde tutacak geçici nesne de bir göstericidir.
Bir adres bilgisiyle geri dönen işlevlere C programlarında çok rastlanır. Standart C işlevlerinin bazıları da, adres türünden bir değer döndürür.
Aşağıdaki kod parçasını inceleyin:

```
#include <stdio.h>

int g = 10;

int *foo()
{
    return &g;
}

int main()
{
    int *ptr;

    printf("g = %d\n", g);
    ptr = foo();
    *ptr = 20;
    printf("g = %d\n", g);

    return 0;
}
```

foo işlevi çağrıldığında global *g* isimli değişkenin adresini döndürüyor. İşlev geri dönüş değerini

```
return &g;
```

deyimiyle üretiyor. *&g* ifadesi *(int *)* türündendir. Bu ifadenin değeri yine *(int *)* türünden olan geçici nesneye atanıyor. *main* işlevi içinde çağrılan *foo* işlevinin geri döndürdüğü adres, *ptr* gösterici değişkenine atanıyor.

Bir dizinin en büyük elemanını bulup bu elemanın değerine geri dönen *getmax* isimli işlev daha önce yazılmıştı. Aşağıda aynı işlev bu kez en büyük dizi elemanının adresine geri dönecek şekilde yazılıyor:

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

#define    SIZE       20

int *getmax(const int *p, int size)
{
    int *pmax, i;

    pmax = (int *)p;
    for (i = 1; i < size; ++i)
        if (p[i] > *pmax)
            pmax = (int *)(p + i);
    return pmax;
}


int main()
{
    int a[SIZE];
    int k;

    srand(time(0));
    for (k = 0; k < SIZE; ++k) {
        a[k] = rand() % 1000;
        printf("%d ", a[k]);
    }
    printf("\n");
    printf("max = %d\n", *getmax(a, SIZE));

    return 0;
}
```

İşlevin kodunu inceleyin. Yerel *pmax* gösterici değişkeni, dizinin en büyük elemanının adresini tutmak için tanımlanıyor. Başlangıçta dizinin ilk elemanı en büyük kabul edildiğinden, *pmax* göstericisine önce dizinin ilk elemanının adresi atanıyor:

```
pmax = (int *)p
```

atamasıyla dışarıdan alınan dizinin başlangıç adresinin *pmax* isimli göstericiye atandığını görüyorsunuz. Daha sonra oluşturulan *for* döngüsüyle sırasıyla dizinin diğer elemanlarının, *pmax* göstericisinin gösterdiği nesneden daha büyük olup olmadıkları sınanıyor. *pmax*'ın gösterdiği nesneden daha büyük bir dizi elemanı bulunduğunda, bu elemanın adresi *pmax* göstericisine atanıyor.

```
pmax = (int *)(p + i)
```

atamasında *p + i* ifadesinin *&p[i]* ifadesine eşdeğer olduğunu biliyorsunuz.
Döngü çıkışında *pmax* gösteri değişkeni dizinin en büyük elemanının adresini tutar, değil mi?
main işlevinde *SIZE* uzunluğunda bir dizi önce rastgele değerle dolduruluyor. Daha sonra dizi elemanlarının değerleri ekrana yazdırılıyor.

Aşağıdaki işlev çağrısıyla ekrana *getmax* işlevinin geri döndürdüğü adresteki nesnenin değeri yazdırılıyor.

```
printf("max = %d\n", *getmax(a, SIZE));
```

Bu da dizinin en büyük elemanının değeridir, değil mi?

Bir işlevin bir adres bilgisine geri dönmesinin başka faydaları da olabilir. Hesaplanmak istenen bir değeri dışarıya iletmek yerine, hesaplanan bu değeri içinde tutacak bir nesnenin adresi dışarıya iletilebilir. Nasıl olsa bir nesnenin adresi elimizdeyken o nesnenin değerine de ulaşabiliriz, değil mi?

Type sözcüğü bir tür belirtiyor olsun. *Type* türünden bir değer döndüren bir işlevin bildirimi aşağıdaki gibidir:

```
Type foo(void);
```

Böyle bir işlevin geri dönüş değerini saklamak için bu kez *Type* türünden bir nesneye işlev çağrı ifadesini atamak gerekir.

```
Type val = foo();
```

Type türden bir nesne bellekte *100 byte* yer kaplıyor olsun. Bu durumda, çağrılan *foo* işlevinin çalışması sırasında *return* deyimine gelindiğinde *100 byte*'lık bir geçici nesne oluşturulur. *return* ifadesinden bu geçici nesneye yapılan atama *100 byte*'lık bir bloğun kopyalanmasına neden olur. Geçici nesnenin *val* isimli değişkene aktarılması da, yine *100 byte*'lık bir bellek bloğunun kopyalanmasına neden olur.

Şimdi de aşağıdaki bildirime bakalım:

```
Type *foo(void);
```

Bu kez işlev hesapladığı nesnenin değerini içinde taşıyan *Type* türünden bir nesnenin adresini döndürüyor. Bu durumda, işlevin geri dönüş değerini içinde taşıyacak geçici nesne yalnızca, *2 byte* ya da *4 byte* olur, değil mi? Yani *100 byte*'lık bir blok kopyalaması yerine yalnızca *2* ya da *4 byte'lık* bir kopyalama söz konusudur.
Böyle bir işlevin geri dönüş değerini saklamak için, işlev çağrı ifadesinin bu kez *Type* * türünden bir nesneye atanması gerekir.

```
Type *ptr;
ptr = foo();
```

foo işlevinin hesapladığı değeri içinde tutan *Type* türünden nesnenin adresi, *Type* türünden bir gösterici değişkene atanıyor. *Type* türünün *sizeof* değeri ne olursa olsun bu işlem, yalnızca bir göstericinin *sizeof* değeri kadar büyüklükte bir bloğun kopyalanmasına neden olur, değil mi?
Bir işlevin, bir değerin kendisini dışarıya iletmesi yerine o değeri içinde tutan nesnenin adresini dışarıya iletmesi, bellek ve zaman kullanımı açısından maliyeti düşürür.
Bir de başka bir faydadan söz edelim. Bir işlevden bir nesnenin değerini alınırsa, bu değeri taşıyan nesne değiştirilemez. Ancak bir işlevden bir nesnenin adresi alınırsa, adresi alınan nesne değiştirilebilir.

Yukarıda yazılan *getmax* işlevi bize dizinin en büyük elemanının adresini döndürüyordu, değil mi? Aşağıdaki *main* işlevini inceleyin:

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

#define    SIZE   100

void set_random_array(int *ptr, int size, int max_val);
void display_array(const int *ptr, int size);
int *getmax(const int *ptr, int size);

int main()
{
    int a[SIZE];

    srand(time(0));

    set_random_array(a, SIZE, 1000);
    display_array(a, SIZE);
    *getmax(a, SIZE) = -1;
    display_array(a, SIZE);
    *getmax(a, SIZE) = -1;
    display_array(a, SIZE);

    return 0;
}
```

Aşağıdaki deyime bakalım:

```
*getmax(a, SIZE) = -1;
```

İşlevin geri döndürdüğü adresin, içerik işlecine terim yapıldığını görüyorsunuz. Bu ifadeyle işlevin geri döndürdüğü adresteki nesneye ulaşılarak bu nesneye *–1* değeri atanıyor. Yani dizinin en büyük elemanının değeri *-1* yapılıyor. Dizinin en büyük elemanın değerini geri döndüren bir işlevle bu işin yapılması mümkün olamazdı.

Şimdi aşağıdaki programı dikkatle inceleyin ve yazılan *selec_sort* isimli işlevde ne yapıldığını anlamaya çalışın:

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

#define    SIZE   100

void swap(int *p1, int *p2);
void display_array (const int *p, int size);
void set_random_array(int *ptr, int size, int max_val);
int *getmax(const int *ptr, int size);

void select_sort(int *ptr, int size)
{
    int k;

    for (k = 0; k < size - 1; ++k)
        swap (getmax(ptr + k, size - k), ptr + k);
}

int main()
{
    int a[SIZE];

    srand(time(0));
    set_random_array(a, SIZE, 1000);
    printf("siralanmadan once\n");
    display_array(a, SIZE);
    select_sort(a, SIZE);
    printf("siralanmadan sonra\n");
    display_array(a, SIZE);

    return 0;
}
```

Yerel Nesnelerin Adresleriyle Geri Dönmek

Adrese geri dönen bir işlev asla otomatik ömürlü yerel bir nesnenin adresiyle geri dönmemelidir. Otomatik ömürlü yerel nesnelerin adresleriyle geri dönmek tipik bir programlama hatasıdır.
Klavyeden girilen bir ismin başlangıç adresine geri dönen bir işlev yazılmak istensin:

```
char *getname()
{
    char name_entry[40];

    printf("bir isim girin: ");
    gets(name_entry);
    return name_entry;     /* Yanlış!  */
}
```

```
#include <stdio.h>

int main()
{
    char *ptr = get_name();
    printf("alınan isim = %s\n", ptr);

    return 0;
}
```

get_name işlevi içinde yerel bir dizi tanımlanıyor. Kullanıcının girdiği isim yerel diziye yerleştiriliyor, daha sonra yerel dizinin adresiyle geri dönülüyor.
Yerel değişkenlerin otomatik ömürlü olduğunu, yani ait oldukları bloğun yürütülmesi sonunda bellekten boşaltıldıklarını biliyorsunuz. *get_name* işlevinin geri dönüş değeri, yani yerel *name_entry* dizisinin başlangıç adresi *main* işlevi içinde *ptr* göstericisine atanıyor. Oysa artık yerel dizi bellekten boşaltıldığı için, *ptr* gösterici değişkenine atanan adresin hiçbir güvenilirliği yoktur. *ptr* göstericisinin gösterdiği yerden okuma yapmak ya da buraya yazmak gösterici hatasıdır.
Adrese geri dönen bir işlev yerel bir değişkenin adresiyle ya da yerel bir dizinin başlangıç adresiyle geri dönmemelidir. Yukarıda yazılan *getname* işlevinin çağrılması çalışma zamanı hatasına neden olur. C derleyicilerinin çoğu, bu durumu mantıksal bir uyarı iletisi ile belirler.

NULL Adres Değişmezi (Null Gösterici)

Bir gösterici değişken, içinde adres bilgisi tutan bir nesnedir, değil mi?

```
int x = 10;
int *ptr = &x;
```

Yukarıdaki deyimleri aşağıdaki cümlelerle ifade edebiliriz:
ptr göstericisi *x* nesnesinin adresini tutuyor.
ptr göstericisi *x* nesnesini gösteriyor.
**ptr* nesnesi , *ptr*'nin gösterdiği nesnedir.
**ptr* , *x* nesnesinin kendisidir.

Öyle bir gösterici olsun ki hiçbir nesneyi göstermesin. Hiçbir yeri göstermeyen bir göstericinin değeri öyle bir adres olmalıdır ki, bu adresin başka hiçbir amaçla kullanılmadığı güvence altına alınmış olsun. İşte hiçbir yeri göstermeyen bir adres olarak kullanılması amacıyla bazı başlık dosyalarında standart bir simgesel değişmez tanımlanmıştır. Bu simgesel değişmez *NULL* simgesel değişmezi olarak bilinir.

NULL bir simgesel değişmezdir. Bu simgesel değişmez standart başlık dosyalarından *stdio.h*, *string.h* ve *stddef.h* içinde tanımlanmıştır.
NULL adresi herhangi türden bir göstericiye atanabilir. Böyle bir atama tamamen sözdizimsel kurallara uygundur, uyarı gerektiren bir durum da söz konusu değildir.

```
int *iptr = NULL;
char *cptr = NULL;
double *dptr = NULL;
```

NULL adresi hiçbir yeri göstermeyen bir göstericinin değeridir. Bir gösterici ya bir nesneyi gösterir (yani bu durumda göstericinin değeri gösterdiği nesnenin adresidir) ya da hiçbir nesneyi göstermez (bu durumda göstericinin değeri *NULL* adresidir).

Bir adres bilgisinin doğru ya da yanlış olarak yorumlanması söz konusu olduğu zaman, adres bilgisi *NULL* adresi ise *"yanlış"* olarak yorumlanır. *NULL* adresi dışındaki tüm adres bilgileri "doğru" olarak yorumlanır. *ptr* isimli bir göstericinin değeri *NULL* adresi değil ise, yani *ptr* göstericisi bir nesneyi gösteriyorsa bir işlev çağrılmak istensin. Böyle bir *if* deyiminin koşul ifadesi iki ayrı biçimde yazılabilir:

```
if (ptr != NULL)
    foo();

if (ptr)
    foo();
```

Bu kez *ptr* isimli gösterici değişkenin değeri *NULL* adresi ise, yani *ptr* göstericisi bir nesneyi göstermiyorsa, *foo* işlevi çağrılmak istensin. *if* deyiminin koşul ifadesi yine iki ayrı biçimde yazılabilir:

```
if (ptr == NULL)
    foo();
```

```
if (!ptr)
    foo();
```

Bir gösterici değişkene herhangi bir türden *0* değeri atandığında, atama öncesi 0 değeri otomatik olarak *NULL* adresine dönüştürülür:

```
int *ptr = 0;
```

Yukarıdaki deyimle ptr gösterici değişkenine *NULL* adresi atanıyor.
Peki *NULL* adresi ne için kullanılır?
Adrese geri dönen bir işlevin eğer başarısızlığı söz konusu ise, işlev başarısızlık durumunu *NULL* adresine geri dönerek bildirebilir.
int türden bir dizi içinde bulunan ilk asal sayının adresi ile geri dönen bir işlev yazmak isteyelim:

```
int *get_first_prime(const int *ptr, int size);
```

İşlevin birinci parametresi dizinin başlangıç adresi, ikinci parametresi ise dizinin boyutu olsun. İşlevi aşağıdaki biçimde yazdığımızı düşünün:

```
int is_prime(int val);

int *get_first_prime(const int *ptr, int size)
{
    int k;

    for (k = 0; k < size; ++k)
        if  (isprime (ptr[k]))
            return ptr + k;
    /* ???????  */
}
```

Yukarıdaki işlevde, dışarıdan başlangıç adresi alınan dizimizin her elemanın asal olup olmadığı sınanıyor, ilk asal sayı görüldüğünde bu elemanın adresiyle geri dönülüyor. Peki ya dizinin içinde hiç asal sayı yoksa, *for* döngü deyiminden çıkıldığında işlev bir çöp değeri geri döndürür, değil mi? Peki bu durumda işlev hangi geri dönüş değerini üretebilir?

Madem ki *NULL* adresi hiçbir yeri göstermeyen bir adres, o zaman adrese geri dönen bir işlev başarısızlık durumunu *NULL* adresine geri dönerek bildirebilir, değil mi?

```
#include <stdio.h>

int is_prime(int val);

int *get_first_prime(const int *ptr, int size)
{
    int k;

    for (k = 0; k < size; ++k)
        if  (isprime (ptr[k]))
            return ptr + k;
    return NULL;
}
```

C dilinde, adrese geri dönen bir işlevin başarısızlık durumunda *NULL* adresine geri dönmesi, çok sık kullanılan bir *konvensiyon*dur.

Parametre değişkeni gösterici olan bir işlev, kendisine geçilen *NULL* adresini bir bayrak değeri olarak kullanabilir. Aşağıdaki gibi bir işlev tasarladığımızı düşünelim:

```
void func(char *ptr)
{
    if (ptr == NULL) {
        /***/
    }
    else {
        /***/
    }
}
```

İşlev kendisine geçilen adresin *NULL* adresi olup olmamasına göre farklı işler yapıyor. Tabi bu durumun işlevi çağıran kod parçası tarafından bilinmesi gerekir. Standart C işlevlerinden *time* işlevi böyledir. *time* işlevinin gösterici parametresine *NULL* adresi geçildiğinde işlev herhangi bir nesneye değer atamaz. Hesapladığı değeri yalnızca geri dönüş değeri olarak dışarıya iletir. Ancak işleve *NULL* adresi dışında bir adres gönderildiğinde işlev verilen adresteki nesneye hesapladığı değeri yazar.

Birçok programcı bir gösterici değişkene güvenilir bir adres atamadan önce, göstericiye *NULL* adresi değerini verir. Böylece kod içinde ilgili göstericinin henüz bir nesneyi göstermediği bilgisi güçlü bir biçimde verilerek, kodun okunabilirliği artırılır.

Bir göstericinin ömrü henüz sona ermeden, gösterdiği nesnenin ömrü sona erebilir. Bu durumda göstericinin değeri olan adres güvenilir bir adres değildir. Kod içinde bu durumu vurgulamak için göstericiye *NULL* adresi atanabilir.

Göstericilere İlişkin Uyarılar ve Olası Gösterici Hataları

Bir Göstericiye Farklı Türden Bir Adres Atanması:

Bir göstericiye farklı türden bir adres atandığında, C derleyicilerin çoğu durumu şüpheyle karşılayarak mantıksal bir uyarı iletisi verir. Ancak derleyici yine de farklı türden adresin sayısal bileşenini hedef göstericiye atar. Borland derleyicileri bu durumda aşağıdaki uyarı iletisini verir:

```
warning : suspicious pointer conversion in function ......
```

Aşağıdaki kodu inceleyin:

```
#include <stdio.h>

int main()
{
    double d = 1874567812.987;
    int *ptr;

    ptr = &d;
    *ptr = 0;
    printf("d = %lf\n", d);

    return 0;
}
```

Yukarıdaki kodda *double* türden *d* değişkeninin adresi *int* türden bir nesneyi gösterecek *ptr* değişkenine atanıyor. Böyle bir atamadan sonra *ptr double* türden *d* değişkenini değil, *int* türden bir nesneyi gösterir.

```
*ptr = 0;
```

atamasıyla d değişkeninin ilk *4 byte*'ına tamsayı formatında *0* değeri atanmış olur. Böyle bir atamadan sonra d değişkeninin değeri istenilmeyen bir biçimde değiştirilmiş olur, değil mi?

Eğer göstericiye farklı türden adres bilinçli bir biçimde atanıyorsa tür dönüştürme işleci kullanılmalıdır:

```
#include <stdio.h>

int main()
{
    double d;
    unsigned int k;
    unsigned char *ptr;

    printf("bir gercek sayi girin :");
    scanf("%lf", &d);
    ptr = (unsigned char *)&d;

    for (k = 0; k < sizeof(double); ++k)
        printf("%u\n", ptr[k]);

    return 0;
}
```

Yukarıdaki *main* işlevinde *double* türden *d* değişkeninin adresi *unsigned char* türünden bir göstericiye atanıyor. *ptr* gösterici değişkeni *byte byte* ilerletilerek *d* değişkeninin her bir *byte*'ının değeri tamsayı olarak yorumlanarak ekrana yazdırılıyor.

```
ptr = (unsigned char *)&d;
```

deyiminde atamanın bilinçli olarak yapıldığını göstermek için *d* değişkeninin adresi önce *unsigned char* türden bir adres bilgisine dönüştürülüyor, daha sonra atama yapılıyor.

Bir Gösterici Değişkene Adres Olmayan Bir Değerin Atanması

Bu da bilinçli olarak yapılma olasılığı çok az olan bir işlemdir. C derleyicileri şüpheli olan bu durumu mantıksal bir uyarı iletisi ile programcıya bildirirler. Örneğin bu uyarı iletisi *Borland* derleyicilerinde aşağıdaki gibidir:

```
"non-portable pointer conversion"
```

Peki bir göstericiye adres bilgisi olmayan bir değer atanırsa ne olur? C'de bu doğrudan bir sözdizim hatası değildir. Yine otomatik tür dönüşümü söz konusudur. Atama işlecinin sağ tarafındaki ifadenin türü, atama işlecinin sol tarafında bulunan nesne gösteren ifadenin türüne çevrilerek atama yapılır. Dolayısıyla, atanan değer gösterici değişkenin türünden bir adrese çevrilerek göstericiye atanır. Örneğin:

```
void func()
{
    int x = 1356;
    int *ptr;
    ptr = x;
    /***/
}
```

Yukarıdaki örnekte

```
ptr = x;
```

atama deyimi ile *ptr* değişkenine *x* nesnesinin adresi değil, değeri atanıyor. *x* değişkeninin değeri olan *1356,* atama öncesi tür dönüşümüyle bir adres bilgisine dönüştürülerek *ptr* değişkenine atanır. Artık *ptr,* x değişkenini göstermez, *1356* adresindeki nesneyi gösterir:
**ptr* nesnesine erişmek artık bir gösterici hatasıdır.

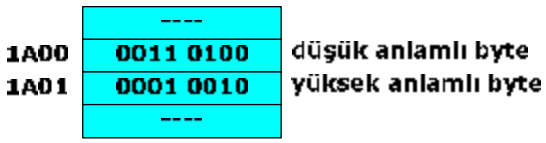
Nesnelerin Bellekteki Yerleşimleri

Bir *byte'tan* daha büyük olan değişkenlerin belleğe yerleşim biçimi kullanılan mikroişlemciye göre değişebilir. Bu nedenle değişkenlerin bellekteki görünümleri taşınabilir bir bilgi değildir. Mikroişlemciler iki tür yerleşim biçimi kullanabilir:

i)Düşük anlamlı *byte* değerleri belleğin düşük sayılı adresinde bulunacak biçimde. Böyle yerleşim biçimine *little endian* denir*. Intel* işlemcileri bu yerleşim biçimini kullanır. Bu işlemcilerin kullanıldığı sistemlerde örneğin

```
int x = 0x1234;
```

biçimindeki bir *x* değişkeni bellekte *1A00* adresinden başlayarak yerleştirilmiş olsun:

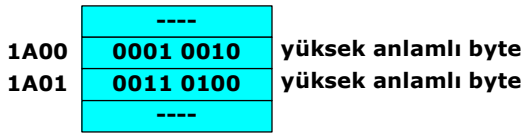


Şekilden de görüldüğü gibi *x* değişkeninin düşük anlamlı *byte* değeri *(34H)* düşük sayısal adreste (*1A00H*) olacak biçimde yerleştirilmiştir.

ii) İkinci bir yerleşim biçimi, düşük anlamlı *byte*'ın yüksek sayısal adrese yerleştirilmesidir.
Böyle yerleşim biçimine *big endian* denir*. Motorola* işlemcileri bu yerleşim biçimini kullanır. Bu işlemcilerin kullanıldığı sistemlerde örneğin

```
int x = 0x1234;
```

biçimindeki bir *x* değişkeni bellekte *1A00* adresinden başlayarak yerleştirilmiş olsun:



Şekilden de görüldüğü gibi *x* değişkeninin düşük anlamlı *byte* değeri *(34H)* yüksek sayısal adreste (*1A00H*) olacak biçimde yerleştirilmiştir.
Aşağıdaki kod kullanılan sistemin *little endian* ya da *big endian* oldugunu sınıyor:

```
#include <stdio.h>

int main()
{
    int x = 1;

   if (*(char *)&x)
       printf("little endian\n");
    else
       printf("big endian\n");

    return 0;
}
```

Yazılan kodda önce adres işleciyle x değişkeninin adresi elde ediliyor. Adres işlecinin ürettiği değer *int* * türündendir. Daha sonra tür dönüştürme işleciyle, elde edilen adres bilgisi *char* * türüne dönüştürülüyor. *char* * türünden adresin de içerik işlecinin terimi olduğunu görüyorsunuz. Bu durumda içerik işleci *x* nesnesinin en düşük sayısal adresindeki *char* türden nesneye erişir, değil mi? Eğer bu nesnenin değeri *1* ise sistem *"little-endian"* dır.

Yazıların İşlevlere Gönderilmesi

Yazılar karakter dizilerinin içinde bulunurlar. Bir işlevin bir yazı üzerinde işlem yapabilmesi için bir yazının başlangıç adresini alması yeterlidir. Yani işlev yazının (karakter dizisinin) başlangıç adresi ile çağrılır. Yazıyı içinde tutan *char* türden dizinin boyutu bilgisini işleve geçirmeye gerek yoktur. Çünkü yazıların sonunda sonlandırıcı karakter vardır. Karakter dizileri üzerinde işlem yapan kodlar dizinin sonunu sonlandırıcı karakter yardımıyla belirler.

Yazılarla ilgili işlem yapan bir işlev *char* türden bir gösterici değişken ile üzerinde işlem yapacağı yazının başlangıç adresini alır. İşlev, yazının sonundaki sonlandırıcı karakteri görene kadar bir döngü ile yazının tüm karakterlerine erişebilir.
str, char türünden bir gösterici olmak üzere yazı üzerinde sonlandırıcı karakteri görene kadar işlem yapabilecek döngüler şöyle oluşturulabilir:

```
while (*str != '\0') {
    /***/
    ++str;
 }

for (i = 0; str[i] != '\0'; ++i) {
    /***/
}
```

puts ve gets İşlevleri

stdio.h içinde bildirilen standart *puts* işlevinin parametre değişkeni *char* türünden bir göstericidir:

```
int puts(const char *str);
```

İşlev, *str* adresindeki yazıyı standart çıkış birimine yazar. İşlev yazma işlemini tamamladıktan sonra ekrana bir de *'\n'* karakteri yazar. Eğer yazma işlevi başarılı olursa işlevin geri dönüş değeri negatif olmayan bir değerdir. Başarısızlık durumunda işlev, negatif bir değere geri döner. Aşağıda benzer işi gören *myputs* isimli bir işlev tanımlanıyor:

```
#include <stdio.h>

void myputs(const char *str)
{
    while (*str != '\0')
        putchar(*str++);
    putchar('\n');
}

int main()
{
    char s[] = "NecatiErgin";
    int k;

    for (k = 0; k < 11; ++k)
        myputs(s + k);

    return 0;
}
```

stdio.h içinde bildirilen standart *gets* işlevi de aslında gösterici parametreli bir işlevdir:

```
char *gets(char *ptr);
```

İşlev standart giriş biriminden aldığı yazıyı parametresine aktarılan adrese yerleştirir. Eğer giriş işlemi başarılı olursa işlev *ptr* adresine geri döner. Başarısızlık durumunda işlevin geri dönüş değeri *NULL* adresidir. Aşağıda benzer işi gören *mygets* isimli bir işlev tanımlanıyor:

```
#include <stdio.h>

char *mygets(char *ptr)
{
    int ch;
    int index = 0;

    while ((ch = getchar()) != '\n')
        ptr[index++] = ch;

    ptr[index] = '\0';

    return ptr;
}
```

Yazılarla İlgili İşlem Yapan Standart İşlevler

C'nin standart bazı işlevleri bir yazının başlangıç adresini parametre olarak alarak yazı ile ilgili birtakım faydalı işlemler yapar. Bu işlevlere dizge işlevleri denir. Dizge işlevlerinin bildirimleri *string.h* dosyası içindedir.

strlen İşlevi

En sık çağrılan standart C işlevlerinden biridir. İşlevin ismi olan *strlen,* "*string length*" sözcüklerinden gelir. Bu işlev bir yazının karakter uzunluğunu yani yazının kaç karakterden oluştuğu bilgisini elde etmek için kullanılır.
İşlevin bildirimi:

```
size_t strlen(const char *str);
```

biçimindedir. İşlevin parametre değişkeni, uzunluğu hesaplanacak yazının başlangıç adresidir. İşlev sonlandırıcı karakter görene kadar karakterlerin sayısını hesaplar. Geri dönüş değeri türü yerine yazılan *size_t* nin şimdilik *unsigned int* türünün bir başka ismi olduğunu düşünebilirsiniz.

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

int main()
{
    char s[100];

    printf("bir yazı giriniz : ");
    gets(s);
    printf("(%s) yazısının uzunluğu = %u\n", s, strlen(s));

    return 0;
}
```

Standart C işlevi olan *strlen* aşağıdaki biçimlerde tanımlanabilir:

```
unsigned int mystrlen1(const char *str)
{
    unsigned int length = 0;

    while (*str != '\0') {
        ++length;
        ++str;
    }
    return length;
}
```

```
unsigned int mystrlen2(const char *str)
{
    unsigned int len;

    for (len = 0; str[len] != '\0'; ++len)
        ;
    return len;
}
```

```
unsigned  int mystrlen3(const char *str)
{
    const char *ptr = str;

    while (*str != '\0')
        str++;
    return str - ptr;
}
```

strchr İşlevi

İşlevin ismi olan *strchr* ,"*string character*" sözcüklerinden gelir. *strchr* işlevi bir karakter dizisi içinde belirli bir karakteri aramak için kullanılan standart bir C işlevidir.
İşlevin *string.h* dosyası içindeki bildirimi aşağıdaki gibidir:

```
char *strchr(const char *str, int ch);
```

Bu işlev ikinci parametresi olan *ch* karakterini, birinci parametresi olan *str* adresinden başlayarak sonlandırıcı karakter görene kadar arar. Aranan karakter sonlandırıcı karakterin kendisi de olabilir. İşlevin geri dönüş değeri, *ch* karakterinin yazı içinde bulunabilmesi durumunda bulunduğu yerin adresidir. Eğer *ch* karakteri yazı içinde bulunamazsa, işlev *NULL* adresine geri döner.
strchr işlevi aşağıdaki gibi tanımlanabilir:

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

int main()
{
    char s[100];
    char *p, ch;

    printf("bir yazı girin : ");
    gets(s);
    printf("yazı içinde arayacağınız karakteri girin : ");
    scanf("%c", &ch);
    p = strchr(s, ch);
    if (p == NULL)
        printf("aranan karakter bulunamadı\n");
    else
        printf("bulundu: (%s)\n", p);

    return 0;
}

char *strchr(const char *str, int ch)
{
    char c = ch;

    while (*str != '\0') {
        if (*str  == c)
            return (char *)str;
        ++str;
    }
    if (ch == '\0')
        return (char *)str;

    return NULL;
}
```

strrchr İşlevi

Standart olan bu işlev *strchr* işlevi gibi bir yazının içinde bir karakteri arar. Arama yazının sonundan başlanarak yapılır. Yani *"ankara"* yazısı içinde *'n'* karakteri arandığında işlev yazının son karakterinin adresini döndürür. İşlevin *string.h* başlık dosyası içindeki bildirimi aşağıdaki gibidir:

```
char *strrchr(const char *str, int ch);
```

İşlev, *str* adresindeki yazı içinde bulunabilecek son *ch* karakterinin adresini döndürür. Eğer yazının içinde *ch* karakteri yoksa işlevin geri dönüş değeri *NULL* adresidir. İşlev ile yazının sonundaki sonlandırıcı karakter de aranabilir.

```
#include <stdio.h>

char *mstrrchr(const char *str, int ch)
{
    const char *p = str;
    const char *pf = NULL;

    while (*p) {
        if (*p == ch)
            pf = p;
        p++;
    }

    if (ch == '\0')
        return (char *)p;

    return (char *)pf;
}

int main()
{
    char s[100];
    char *ptr;
    int ch;

    printf("bir yazi girin : ");
    gets(s);
    printf("aranacak karakteri girin: ");
    ch = getchar();

    ptr = mstrrchr(s, ch);

    if (ptr == NULL)
        printf("(%s) yazısında (%c) bulunamadı!\n", s, ch);
    else
        printf("bulundu: (%s)\n", ptr);

    return 0;
}
```

strstr İşlevi

İşlevin ismi *string string* sözcüklerinden gelir. Bu işlevle bir yazı içinde başka bir yazı aranır. İşlevin bildirimi aşağıdaki gibidir:

```
char *strstr(const char *p1, const char *p2);
```

Eğer *p1* adresindeki yazı içinde *p2* adresindeki yazı varsa, işlev yazının bulunduğu yerin adresini döndürür. Eğer yoksa işlevin geri dönüş değeri *NULL* adresidir. Aşağıdaki programı derleyerek çalıştırın:

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

#define SIZE       100


int main()
{
    char s1[SIZE];
    char s2[SIZE];
    char *ptr;

    printf("içinde arama yapılacak yazıyı girin: ");
    gets(s1);
    printf("aranacak yazıyı girin: ");
    gets(s2);

    ptr = strstr(s1, s2);

    if (!ptr)
        printf("(%s) yazisi icinde (%s) yazisi yok!\n", s1, s2);
    else
        printf("bulundu! (%s)\n", ptr);

    return 0;
}
```

strcspn İşlevi

Bu işlev ile bir yazının içinde başka bir yazıda olan ilk karakterin indisi bulunur. İşlevin *string.h* içindeki bildirimi şöyledir:

```
size_t strcspn(const char *p1, const char *p2);
```

Geri dönüş değeri şu şekilde de tanımlanabilir : İşlev *p1* yazısı içinde yazının başından başlayarak, *p2* yazısının karakterlerinden herhangi birini içermeyen yazının uzunluğu değerine geri döner:

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

#define SIZE       100

int main()
{
    char s1[SIZE];
    char s2[SIZE];
    size_t index;

    printf("icinde arama yapilacak yaziyi girin: ");
    gets(s1);
    printf("karakterler: ");
    gets(s2);
    index = strcspn(s1, s2);
    printf("(%s)\n", s1 + index);
    s1[index] = '\0';
    printf("(%s)\n", s1);

    return 0;
}
```

strpbrk İşlevi

Standart *strpbrk* işlevi ile bir yazıda başka bir yazının karakterlerinden herhangi biri aranır:

```
char *mstrpbrk(const char *s1, const char *s2)
```

Eğer *s1* adresindeki yazının içinde *s2* adresindeki yazının karakterlerinden herhangi biri varsa işlev bu karakterin adresini döndürür. Eğer *s1* yazısı içinde *s2* yazısının karakterlerinin hiçbiri yoksa işlev *NULL* adresine geri döner.
Aşağıda *strpbrk* işlevinin örnek bir tanımı ile bir sınama kodu yer alıyor:

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

#define    SIZE   100

char *mstrpbrk(const char *s1, const char *s2)
{
    const char *p1, *p2;

    for (p1 = s1; *p1 != '\0'; ++p1)
        for (p2 = s2; *p2 != '\0'; ++p2)
            if (*p1 == *p2)
                return (char *)p1;
    return NULL;
}

int main()
{
    char str1[SIZE];
    char str2[SIZE];
    char *ptr;

    printf("birinci yazıyı girin : ");
    gets(str1);
    printf("ikinci yazıyı girin : ");
    gets(str2);
    ptr = strpbrk(str1, str2);
    if (ptr == NULL)
        printf("\"%s yazisinda\" (%s) karakterlerinden hic biri yok!\n",
            str1, str2);
    else
        printf("bulundu : (%s)\n", ptr);

    return 0;
}
```

strcpy İşlevi

Standart bir C işlevidir. İşlevin ismi olan *strcpy*, "*string*" ve "*copy*" sözcüklerinden gelir. İşlev ikinci parametresinde tutulan adresten başlayarak, *sonlandırıcı karakter* görene kadar, sonlandırıcı karakter de dahil olmak üzere, tüm karakterleri birinci parametresinde tutulan adresten başlayarak sırayla yazar. İşlevin *string.h* başlık dosyası içindeki bildirimi aşağıdaki gibidir:

```
char *strcpy(char *dest, const char *source);
```

İşlevin geri dönüş değeri kopyalamanın yapılmaya başlandığı adres yani *dest* adresidir. *strcpy* işlevi aşağıdaki gibi tanımlanabilir:

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

int main()
{
    char dest[100] = "C öğreniyoruz!";
    char source[100];

    printf("kopyalanacak yazıyı girin : ");
    gets(source);
    printf("kopyalama yapılmadan önce yazı :(%s)\n", dest);
    strcpy(dest, source);
    printf("kopyalama yapildiktan sonra yazı :(%s)\n", dest);

    return  0;
}
```

```
char *mystrcpy(char *dest, const char *source)
{
    int i;

    for (i = 0; (dest[i] = source[i]) != '\0'; ++i)
        ;
    return dest;
}
```

İşlev içinde kullanılan *for* döngüsünün ikinci kısmında önce atama yapılıyor, daha sonra atama ifadesinin değeri yani atama işlecinin sağ tarafında bulunan değer *sonlandırıcı karakter* ile karşılaştırılıyor. Böylece ilgili adrese sonlandırıcı karakter de kopyalandıktan sonra döngüden çıkılıyor.
İşlev aşağıdaki gibi de yazılabilirdi:

```
/***/
for (i = 0; source[i] != '\0'; ++i)
    dest[i] = source[i];
dest[i] = '\0';
/***/
```

for döngü deyiminde köşeli ayraç işleci kullanıldığı için, birinci parametre değişkenine kopyalanan *dest* gösterici değişkeninin değeri değiştirilmiyor. İşlevin sonunda *dest* adresi ile geri dönülüyor. İşlevin yazımında *while* döngüsü kullanılarak, *dest* değişkeni içindeki adres sürekli artırılabilirdi. Bu durumda işlevin *dest* göstericisinin ilkdeğeriyle geri dönebilmesini sağlayabilmek için, *dest* göstericisindeki değeri değiştirmeden önce, bu değeri başka bir gösterici değişken içinde saklamak gerekirdi:

```
char *mystrcpy(char *dest, const char *source)
{
    char *temp = dest;

    while ((*source++ = *dest++) != '\0')
        ;
    return temp;
}
```

strcat İşlevi

Standart bir C işlevidir. İşlevin ismi *string* ve *concatanate* sözcüklerinden gelir *strcat* işlevi bir yazının sonuna başka bir yazının kopyalanması amacıyla kullanılır. İşlevin *string.h* dosyası içindeki bildirimi aşağıdaki gibidir:

```
char *strcat(char *s1, const char *s2);
```

strcat işlevi eklemenin yapılacağı ve başlangıç adresi *s1* birinci parametre değişkeninde tutulan yazının sonundaki sonlandırıcı karakteri ezerek, başlangıç adresi ikinci parametre değişkeninde tutulan yazıyı birinci yazının sonuna (sonlandırıcı karakter de dahil olmak üzere) ekler. Yani işlem sonunda *s1* adresindeki yazının uzunluğu *s2* adresindeki yazının uzunluğu kadar artar.
İşlevin geri dönüş değeri, sonuna eklemenin yapıldığı yazının başlangıç adresi, yani *s1* adresidir.

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

int main()
{
    char s1[100], s2[100];

    printf("sonuna ekleme yapılacak yazıyı girin : ");
    gets(s1);
    printf("girdiğiniz yazının uzunluğu = %d\n", strlen(s1));
    printf("eklemek istediğiniz yazıyı girin : ");
    gets(s2);
    printf("eklenecek yazının uzunluğu = %d\n", strlen(s2));
    strcat(s1, s2);
    printf("ekleme yapıldıktan sonra 1. yazı : ");
    puts(s1);
    printf("ekleme yapıldıktan sonra yazının uzunluğu : %d\n", strlen(s1))

    return 0;
}
```

strcat işlevi aşağıdaki gibi tanımlanabilir:

```
char *mystrcat(char *s1, const char *s2)
{
    char *temp = s1;

    while (*s1 != '\0')
        ++s1;
    while ((*s1++  == *s2++) != '\0')     /* strcpy(s1, s2); */
        ;

    return temp;
}
```

Bir yazının sonuna başka bir yazıyı eklemek, sona eklenecek yazıyı diğer yazının sonundaki sonlandırıcı karakterin bulunduğu yere kopyalamak anlamına gelir, değil mi? Dolayısıyla *strcat* işlevi aşağıdaki biçimlerde de tanımlanabilir.

```
char *mystrcat(char *s1, const char *s2)
{
    strcpy(s1 + strlen(s1), S2);
    return s1;
}

char *mystrcat(char *s1, const char *s2)
{
    strcpy(strchr(s1, '\0'), s2);
    return s1;
}
```

strcmp işlevi

Standart bir C işlevidir. İşlevin ismi *string compare* sözcüklerinden gelir. İşlev iki karakter dizisini karşılaştırmakta kullanılır. Karşılaştırma, iki karakter dizisi içindeki yazının, kullanılan karakter seti tablosu gözönünde bulundurularak, öncelik ya da eşitlik durumunun sorgulanmasıdır. Örneğin:

Adana yazısı *Ankara* yazısından daha küçüktür. Çünkü eşitliği bozan *'n'* karakteri *ASCII* karakter tablosunda *'d'* karakterinden sonra gelir.

ankara yazısı *ANKARA* yazısından daha büyüktür. Çünkü küçük harfler *ASCII* tablosunda büyük harflerden sonra gelir.
Küçük "*masa*" büyük "*MASA*" dan daha büyüktür.

kalem yazısı *kale* yazısından daha büyüktür.

strcmp işlevinin *string.h* başlık dosyası içindeki bildirimi aşağıdaki gibidir:

```
int strcmp(const char *s1, const char *s2);
```

İşlev birinci parametre değişkeninde başlangıç adresi tutulan yazı ile, ikinci parametre değişkeninde başlangıç adresi tutulan yazıları karşılaştırır.

İşlevin geri dönüş değeri,
birinci yazı ikinci yazıdan daha büyükse pozitif bir değerdir,
birinci yazı ikinci yazıdan daha küçükse negatif bir değerdir,
birinci yazı ile ikinci yazı birbirine eşit ise *0* değeridir.

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

int main()
{
    char s[20];
    char password[ ] = "Mavi ay";

    printf("parolayı girin : ");
    gets(s);

    if (!strcmp(s, password))
        printf("Parola doğru!..\n");
    else
        printf("Parola yanlış!..\n");
    return 0;
}
```

strcmp işlevi aşağıdaki gibi tanımlanabilir:

```
int mystrcmp(const char *s1, const char *s2)
{
    while (*s1 == *s2) {
        if (*s1 == '\0')
            return 0;
        ++s1;
        ++s2;
    }
    return *(unsigned char *)s1 >  *(unsigned char *)s2 ? 1 : -1;
}
```

strncpy işlevi

Standart bir C işlevidir. İşlevin ismi *string number copy* sözcüklerinden gelir. İşlev bir yazının (karakter dizisinin) ilk *n* karakterini başka bir yere kopyalamakta kullanılır. İşlevin *string.h* içindeki bildirimi aşağıdaki gibidir :

```
char *strncpy(char *dest, const char *source, size_t n);
```

İşlev birinci parametre değişkeninde başlangıç adresi tutulan yazıya, ikinci parametre değişkeninde adresi tutulan yazıdan, üçüncü parametresinde tutulan sayıda karakteri kopyalar.
İşlevin geri dönüş değeri kopyalamanın yapılacağı adres yani *dest* adresidir.

Üçüncü parametre olan n sayısı eğer kopyalanacak yazının uzunluğundan daha küçük ya da eşit ise işlev kopyalama sonunda sonlandırıcı karakteri birinci dizinin sonuna eklemez. Yani *n <= strlen(source)* ise sonlandırıcı karakter eklenmez.
Üçüncü parametre olan n sayısı eğer kopyalanacak yazının uzunluğundan daha büyük ise işlev kopyalama sonunda sonlandırıcı karakteri birinci dizinin sonuna ekler.
Yani *n > strlen(source)* ise sonlandırıcı karakter de kopyalanır.
Aşağıda *strncpy* işlevi tanımlanıyor:

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

#define    SIZE   100

char *mstrncpy(char *s1, const char *s2,  unsigned int  n)
{
    char *s = s1;

    while (n > 0 && *s2)  {
        *s++ = *s2++;
        n--;
    }

    while (n--)
        *s++ = '\0';

    return s1;
}

int main()
{
    char str1[SIZE];
    char str2[SIZE];
    int n;

    printf("birinci yaziyi girin : ");
    gets(str1);
    printf("ikinci yaziyi girin : ");
    gets(str2);
    printf("ikinci  yazidan kac karakter kopyalanacak? ");
    scanf("%d", &n);
    strncpy(str1, str2, n);

    printf("(%s)\n", str1);

    return 0;
}
```

Üçüncü parametreye aktarılan değer kopyalanan yazının uzunluğundan küçük ya da eşitse kopyalanan yere sizce neden sonlandırıcı karakter eklenmez?
Bu işlevin yazılar içinde yer değiştirme işlemi yapabilmesi istenmiştir:

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

#define    SIZE    100


int main()
{
    char str1[SIZE] = "Necati Ergin";
    char str2[SIZE] = "Mehmet Aktunc";

    strncpy(str1, str2, 6);

    printf("(%s)\n", str1);

    return 0;
}
```

strncat İşlevi

Standart bir C işlevidir. İşlevin ismi ingilizce "*string number concatanate*" sözcüklerinden gelir. Bir yazının sonuna başka bir yazıdan belirli bir sayıda karakteri kopyalamak amacıyla kullanılır. *string.h* başlık dosyası içinde bulunan bildirimi aşağıdaki gibidir:

```
char *strncat(char *s1, const char *s2, size_t n);
```

İşlev birinci parametre değişkeni içinde başlangıç adresi verilen yazının sonuna, ikinci parametresinde başlangıç adresi tutulan karakter dizisinden, üçüncü parametresinde tutulan tamsayı adedi kadar karakteri kopyalar.

İşlevin geri dönüş değeri sonuna ekleme yapılacak yazının başlangıç adresidir.
İşlevin tanımı ve işlevi sınayan bir *main* işlevi örnek olarak aşağıda veriliyor:

```
char *mstrncat(char *s1, const char *s2, unsigned int n)
{
    char *ptr;
    for (ptr = s1; *ptr != '\0'; ++ptr)
        ;

    while (n-- && *s2 != '\0')
        *ptr++ = *s2++;

    *ptr = '\0';

    return s1;
}
```

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

#define    SIZE    100

int main()
{
    char dest[SIZE];
    char source[SIZE];
```

```
    int n;

    printf("birinci yaziyi girin : ");
    gets(dest);
    printf("ikinci yaziyi girin : ");
    gets(source);
    printf("1. yazinin sonuna kac karakter kopyalanacak : ");
    scanf("%d", &n);
    mstrncat(dest, source, n);
    printf("eklemeden sonra 1. yazi = (%s)\n", dest);

    return 0;
}
```

strncmp İşlevi

Standart bir C işlevidir. İşlevin ismi *string number compare* sözcüklerinden gelir.. *strcmp* işlevine benzer, ancak bu işlev iki yazının tümünü değil de, belirli bir sayıda karakterlerini karşılaştırma amacıyla kullanılır.
İşlev birinci parametre değişkeninde başlangıç adresi tutulan yazı ile, ikinci parametre değişkeninde başlangıç adresi tutulan yazıların, üçüncü parametresinde tutulan sayıdaki karakterlerini karşılaştırır.
İşlev, birinci yazının ilk *n* karakteri ikinci yazının ilk *n* karakterinden daha büyükse pozitif bir değere
Birinci yazının ilk *n* karakteri ikinci yazının ilk *n* karakterinden daha küçükse negatif bir değere
Birinci yazının ve ikinci yazının ilk *n* karakteri birbirine eşit ise *0* değerine geri döner.
Aşağıda işlevin tanımı ve işlevi sınayan bir *main* işlevi veriliyor:

```
int strncmp(const char *s1, const char *s2, unsigned int n)
{
    while (n--) {
        if (*s1 != *s2)
            return *(unsigned char *)s1 < *(unsigned char *)s2 ? -1 : 1;
        if (*s1 == '\0')
            return 0;
        s1++;
        s2++;
    }
    return 0;
}
```

```
#include <stdio.h>

#define    SIZE    100

int main()
{
    char str1[SIZE];
    char str2[SIZE];
    int n, result;

    printf("birinci yaziyi girin : ");
    gets(str1);
    printf("ikinci yaziyi girin : ");
    gets(str2);
    printf("iki yazinin kac karakteri karsilastirilacak? ");
    scanf("%d", &n);
    result = strncmp(str1, str2, n);
    if (result == 0)
```

```
        printf("(%s) == (%s)\n", str1, str2);
    else if (result > 0)
        printf("(%s) > (%s)\n", str1, str2);
    else
        printf("(%s) < (%s)\n", str1, str2);

    return 0;
}
```

strset İşlevi

Standart olmayan bu işlev derleyicilerin çoğunda bulunur. İşlevin ismi *string* ve *set* sözcüklerinden gelir. Bir karakter dizisinin belirli bir karakterle doldurulması amacıyla kullanılır. İşlevin *string.h* başlık dosyası içindeki bildirimi aşağıdaki gibidir:

```
char *strset(char *str, int ch);
```

İşlev birinci parametre değişkeninde başlangıç adresi olan yazıyı sonlandırıcı karakter görene kadar ikinci parametre değişkeninde tutulan karakterle doldurur. Yazının sonundaki sonlandırıcı karaktere dokunmaz.
İşlevin geri dönüş değeri yine doldurulan yazının başlangıç adresidir.

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


int main()
{
    char s[100];
    int ch;

    printf("bir yazı girin :");
    gets(s);
    printf("yazıyı hangi karakterle doldurmak istiyorsunuz : ");
    ch = getchar();
    printf("\nyazı %c karakteriyle dolduruldu (%s)\n", ch, strset(s, ch));

    return 0;
}
```

strset işlevi aşağıdaki gibi tanımlanabilir:

```
#include <stdio.h>

char *mystrset(char *str, int ch)
{
    int i;

    for (i = 0; str[i] != '\0'; ++i)
        str[i] = ch;

    return str;
}
```

strrev İşlevi

Standart olmayan bu işlev de derleyicilerin çoğunda bulunur. İşlevin ismi *string reverse* sözcüklerinden gelir. İşlev bir yazıyı ters çevirmek amacıyla kullanılır. İşlevin *string.h* başlık dosyası içinde yer alan bildirimi aşağıdaki gibidir:

```
char *strrev(char *str);
```

İşlev parametre değişkeninde başlangıç adresi tutulan yazıyı ters çevirir. İşlevin geri dönüş değeri ters çevrilen yazının başlangıç adresidir.
strrev işlevi aşağıdaki gibi tanımlanabilir:

```
char *mystrrev(char *str)
{
    int  i, temp;
    int length = strlen(str);

    for (i = 0; i < length / 2, ++i) {
        temp = str[i];
        str[i] = str[length - i - 1];
        str[length - i - 1] = temp;
    }
    return str;
}
```

strupr ve strlwr İşlevleri

Bu işlev standart olmamalarına karşın hemen hemen her derleyicide bulunur.
İsimleri *string upper* ve *string lower* sözcüklerinden gelir. Bu işlevler bir yazının tüm karakterleri için büyük harf küçük harf dönüştürmesi yapar. İşlevlerin geri dönüş değerleri parametresine aktarılan adrestir. Geri dönüş değerlerine genellikle gereksinim duyulmaz. Her iki işlev de temel *Latin* alfabesinde olan harfler için dönüşüm yapar. Türkçe karakterler için de dönüşüm gerekiyorsa bunun için bir işlev tanımlanmalıdır.

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

int main()
{
    char s[] = "C programcısı olmak için çok çalışmak gerekir!";

    strupr(s);
    puts(s);
    strlwr(s);
    puts(s);
    return  0;
}
```

strupr işlevi aşağıdaki gibi tanımlanabilir:

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

char *mystrupr(char *str)
{
    char *temp = str;

    while (*str != '\0') {
        if (islower(*str))
            *str = toupper(*str);
        ++str;
    }
    return temp;
}
```

Bir Yazının Başlangıç Adresini Tutan Göstericiyi Yazının Sonuna Öteleme

Birçok programda başlangıç adresi tutulan bir yazının sonunu bulmak gerekir.
Bir yazının başlangıç adresini tutan bir gösteri değişkeni, yazının sonundaki sonlandırıcı karakteri gösterir hale getirmek için, üç ayrı kalıp kullanılabilir:

p bir yazının başlangıç adresini tutan gösterici olsun. Aşağıdaki deyimlerin hepsi *p* gösterici değişkeni yazının sonundaki, sonlandırıcı karakterin bulunduğu yere öteler.

```
p += strlen(p);

p = strchr(p, '\0');

while (*p != '\0')
    ++ptr;
```

Aşağıdaki döngüden çıktıktan sonra *p* göstericisi sonlandırıcı karakterden bir sonraki adresi gösterir. Neden?

```
while (*p++ != '\0')
    ;
```

Boş Yazı

Boş yazı (*null string*) uzunluğu *0* olan yazıdır. *str*, *char* türden bir dizi olmak üzere, eğer *str[0]* sonlandırıcı karakter ise, *str* dizisinde boş yazı tutulmaktadır. Boş yazı geçerli bir yazıdır.
Yazılarla ilgili işlem yapan işlevler, adresini aldığı yazıların boş yazı (*null string*) olması durumunda da doğru çalışmalıdır. Aşağıdaki döngüyü inceleyin:

```
while (*++p != '\0')
    ;
```

p bir yazının başlangıç adresini tutan gösterici değişken olmak üzere, yukarıdaki döngüden çıkıldığında *p,* yazının sonundaki sonlandırıcı karakteri gösterir. Ancak *p*'nin gösterdiği yazı eğer boş ise, yukarıdaki döngü, yazıya ait olmayan bellek alanı üzerinde işlem yapmaya başlar. Bu da şüphesiz bir programlama hatasıdır.

Şimdi de aşağıdaki if deyimini inceleyin:

```
int strfunc(const char *ptr)
{
    if (!ptr || !*ptr)
        return 0;
    /***/
}
```

strfunc işlevi içinde yer alan *if* deyiminde yer alan

```
!ptr || !*ptr
```

ifadesi, *ptr* gösterici değişkeninin değerinin *NULL* adresi olması durumunda ya da *ptr*'nin gösterdiği yazının boş olması durumunda doğrudur. Bu ifade, "*ptr* bir yazıyı göstermiyorsa ya da *ptr*'nin gösterdiği yazı boşsa" anlamına gelir. Bu ifadenin mantıksal tersi olan

```
ptr && *ptr
```

ifadesi ise, *ptr* bir yazıyı gösteriyor ve *ptr*'nin gösterdiği yazı boş değil ise anlamına gelir, değil mi?

Her iki ifade de mantıksal işleçlerin kısa devre davranışı özelliğinden faydalanılıyor. Mantıksal *&&, ||* işleçlerinin kısa devre davranışı olmasaydı, *ptr*'nin değeri *NULL* adresiyken, *ptr*'nin gösterdiği nesneye erişilmeye çalışılırdı.

Örnekler

Aşağıda yazılarla ilgili işlem yapan bazı işlevler tanımlanıyor. İşlevlerde gösterici değişkenlerin kullanımını inceleyin.

Aşağıda, bir yazının C'nin kurallarına göre geçerli bir isim olup olmadığı sınayan *is_valid_id* isimli işlev tanımlanıyor. Yazı geçerli bir isim ise işlev sıfır dışı değere, değilse sıfır değerine geri dönüyor:

```
#include <ctype.h>

int is_id(const char *ptr)
{
    int ch;

    //Bos yazi ise gecerli isim degil
    if ((ch = *ptr++) == '\0')
        return 0;

    //İlk karakter harf ya da "alt tire" olmalı
    if (!(isalpha(ch) || ch == '_'))
        return 0;

    //kalan karakterler harf rakam ya da "alt tire" olmalı
    while ((ch = *ptr++) != '\0')
        if (!(isalnum(ch) || ch == '_'))
            return 0;

    return 1;
}
```

Gösterici Hataları

Göstericileri kullanarak RAM üzerinde bir bölgeye erişilebilir. Bir programın çalışması sırasında bellekte çalışıyor durumda olan başka programlar da olabilir. Göstericileri kullanarak o anda çalışmakta olan programın bellek alanına veri aktarılırsa oradaki programın kodu bozulacağı için programın çalışmasında çeşitli bozukluklar çıkabilir. Bu bozukluk tüm sistemi olumsuz yönde etkileyebilir.
Kim tarafından kullanıldığını bilmediğimiz bellek bölgelerine güvenli olmayan bölgeler denir. Güvenli olmayan bölgelere erişilmesine ise "*gösterici hataları*" denir.
Gösterici hataları yapıldığında sistem kilitlenebilir, programlar yanlış çalışabilir. Gösterici hataları sonucundaki olumsuzluklar hemen ortaya çıkmayabilir.
Gösterici hataları güvenli olmayan bölgelere erişildiğinde değil oralara veri aktarıldığında oluşur.
Gösterici hataları derleme sırasında derleyici tarafından saptanamaz. Bu tür hatalar programın çalışma zamanı sırasında olumsuzluklara yol açar. Tanımlama yöntemiyle elde edilmiş olan bellek bölgelerine güvenli bölgeler denir. Bir nesne tanımlandığında, o nesne için derleyici tarafından bellekte ayrılan yer, programcı için ayrılmış bir alandır ve güvenlidir.

Gösterici Hatası Oluşturan Tipik Durumlar

i)İlkdeğer Verilmemiş Göstericilerin Yol Açtığı Hatalar:

Daha önce belirtildiği gibi göstericiler de birer nesnedir. Diğer nesnelerden farkları içlerinde adres bilgileri tutmalarıdır. Göstericiler de nesne olduklarına göre diğer nesneler gibi yerel ya da global olabilirler. Global olarak tanımlanmış göstericiler *0* değeriyle başlatılırken, yerel göstericiler çöp değerleriyle başlatılır:

Yerel bir gösterici tanımlandıktan sonra, herhangi bir şekilde bu göstericiye bir değer ataması yapılmaz ise göstericinin içinde rastgele bir değer bulunacağından, bu gösterici * işleci ile ya da [ ] işleci ile kullanıldığında, bellekte rastgele bir yerde bulunan bir nesneye ulaşılır. Dolayısıyla, elde edilen nesneye bir atama yapıldığı zaman, bellekte, bilinmeyen rastgele bir yere yazılmış olunur. Bu durum tipik bir gösterici hatasıdır.
Örnekler :

```
int main()
{
    int *p;

    *p = 25;    /* YANLIŞ */

    return 0;
}
```

Yukarıdaki örnekte tanımlanan *p* göstericisinin içinde bir çöp değer var.

```
*p = 25;
```

deyimiyle rastgele bir yere yani güvenli olmayan bir yere veri aktarılıyor. Verinin aktarıldığı yerde işletim sisteminin, derleyicinin ya da bellekte kalan başka bir programın (*memory resident*) kodu bulunabilir. Bazı sistemlerde verinin aktarıldığı yerde programın kendi kodu da bulunabilir.

İlkdeğer verilmemiş global göstericilerin içinde (ya da statik yerel göstericilerin içinde) sıfır değeri bulunur. Sıfır sayısı (*NULL adresi*) göstericilerde sınama amacıyla kullanılır. Bu adrese bir veri aktarılması durumunda, derleyicilerin çoğunda isteğe bağlı olarak bu hata çalışma zamanı sırasında sınanır. Örnek:

```
char *p;

int main()
{
    *p = 'm';   /* NULL pointer assignment */

    return 0;
}
```

Yukarıdaki kodun çalıştırılmasında *"NULL pointer assignment"* şeklinde bir çalışma zamanı hatasıyla karşılaşılabilir. Bu sınama derleyicinin çalışabilen program içine yerleştirdiği "sınama kodu" sayesinde yapılır.

İlkdeğer verilmemiş göstericilerin neden olduğu hatalar işlev çağrılarıyla da ortaya çıkabilir:

```
int main()
{
    char *ptr;

    gets(ptr); /* ????? */
    return 0;
}
```

Yukarıdaki kod parçasında standart *gets* işlevi ile klavyeden alınan karakterler, bellekte rastgele bir yere yazılır. Standart *gets* işlevi klavyeden alınan karakterleri kendisine argüman olarak gönderilen adresten başlayarak yerleştirdiğine göre, daha önceki örnekte verilen hata klavyeden girilen bütün karakterler için söz konusudur.

ii)Güvenli Olmayan İlkdeğerlerin Neden Olduğu Gösterici Hataları

Bir göstericiye ilkdeğer verilmesi , o göstericinin güvenli bir bölgeyi gösterdiği anlamına gelmez. Örneğin :

```
char *ptr;
/***/
ptr = (char *) 0x1FC5;
*ptr = 'M';
```

Yukarıdaki örnekte *ptr* göstericisine atanan *(char *) 0x1FC5* adresinin güvenli olup olmadığı konusunda hiçbir bilgi yoktur. Adrese ilişkin bölgenin kullanıp kullanılmadığı bilinemez. her ne kadar bellek alanı içinde belli amaçlar için kullanılan güvenli bölgeler varsa da *1FC5* böyle bir bölgeyi göstermez.

iii)Dizi Taşmalarından Doğan Gösterici Hataları

Bilindiği gibi bir dizi tanımlaması gördüğünde derleyici, derleme sırasında dizi için bellekte toplam dizi uzunluğu kadar yer ayırır. C derleyicileri derleme zamanında bir dizinin taşırılıp taşırılmadığını kontrol etmez.

```
int main()
{
    int a[10], k;

    for (k = 0; k <= 10; ++k)
        a[k] = 0;
    /***/
    return 0;
}
```

Yukarıdaki örnek bir çok sistemde derlenip çalıştırıldığında döngü *k* değişkeninin *10* değeri için de sürer. Oysa *a[10]* elemanı için bir yer ayrılmamıştır. Dizinin son elemanı *a[9]* elemanıdır. Derleyici *a* dizisi için *a[0]*'dan başlayarak *a[9]* elemanları için bellekte toplam *10* nesnelik yani 40 *byte*'lık bir yer ayırır. Oysa yeri ayrılmamış olan *a[10]* bölgesinin kim tarafından ve ne amaçla kullanıldığı hakkında herhangi bir bilgi yoktur. Bu konuda bir garanti bulunmamakla birlikte derleyicilerin çoğu, ardışık olarak tanımlanan elemanları bellekte bitişik olarak *(contiguous*) olarak yerleştirirler. Ama bu güvence altına alınmış bir özellik değildir. Yani bunun güvence altına alındığı düşünülerek kod yazılması problemlere yol açar. Yukarıdaki örnekte *a[10]* bölgesi derleyicilerin çoğunda *a* dizisinden hemen sonra tanımlanan ve döngü değişkeni olarak kullanılan *k* değişkenine ayrılır. Dolayısıyla *a[10]* elemanına *0* değerini vermekle aslında *k* değişkenine *0* değeri aktarılmış olabilir ve bu da döngünün sonsuz bir döngüye dönüşmesine yol açabilir.
Köşeli ayraç işleci de bir gösterici işlecidir, bu işleci kullanarak dizi için ayrılan alanın dışına atama yapılabilir. C dili derleyicileri kaynak kodu bu amaçla denetlemezler. Ayrıca a dizisi için a[-1], a[-2].. gibi ifadeler de sözdizim açısından geçerlidir ve buraya yapılacak atamalar da gösterici hatalarına yol açar.
Bazen dizi taşmalarına işlevler de gizli bir biçimde neden olabilir. Örneğin:

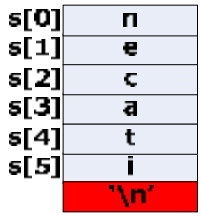
```
void func()
{
    char str[6];

    printf("isim girin : ");
    gets(str);
    /***/
}
```

func işlevi içinde tanımlanan *str* dizisi için toplam *6* karakterlik yer ayrılıyor. Standart *gets* işlevi klavyeden alınan karakterleri kendisine gönderilen adresten başlayarak belleğe yerleştirdikten sonra, sonlandırıcı karakteri de diziye yazar. O halde yukarıdaki örnekte programın çalışması sırasında 6 ya da daha fazla karakterin girilmesi gösterici hatasına neden olur. Sonlandırıcı karakter de ('\0') bellekte bir yer kaplayacağı için program için ayrılmış bir bellek alanı içinde bulunması gerekir. Örneğin klavyeden girilen isim

```
necati
```

olsun. *gets* işlevi bu karakterleri aşağıdaki gibi yerleştirir :



Sonlandırıcı karakter, dizi için ayrılan bölgenin dışına yerleştiriliyor. Bu örnekte girilen isim daha uzun olsaydı, program için ayrılmamış bir bölgeye daha fazla karakter yazılacaktı. Bu tür hatalarla karşılaşmamak için dizi yeteri kadar uzun olmalı ya da standart bir *C* işlevi olan *gets* işlevi yerine, dizi uzunluğundan daha fazla sayıda eleman yerleştirilmesine izin vermeyecek bir işlev kullanılmalıdır. Bu amaçla *fgets* isimli işlev çağrılabilir. Standart *fgets* işlevini dosyalar konusunda göreceksiniz.
Dizgelerle ilgili işlemler yapan standart C işlevlerinden *strcpy, strcat, strncpy, strncat* işlevlerinin yanlış kullanılması da benzer hataları oluşturabilir.

void Türden Göstericiler

Bazı işlevler bellek blokları üzerinde genelleştirilmiş işlemler yapar. Bu işlevler işlem yaptıkları bellek bloklarında ne olduğuyla ilgilenmez. Bir bellek bloğunun içeriğinin bellekte başka bir yere kopyalandığını düşünün. İşlev kaynak adresten hedef adrese *byte byte* kopyalama yaparak bu amacı gerçekleştirebilir. Böyle bir işlevin parametre değişkenleri hangi türden olmalıdır?

void göstericiler herhangi bir türden olmayan göstericilerdir. Bu türden değişkenlerin tanımlarında *void* anahtar sözcüğü kullanılır:

```
void *ptr;
```

void göstericilerin tür bilgisi yoktur. *void* göstericilerde adreslerin yalnızca sayısal bileşenleri saklanır. Bu yüzden *void* göstericilerle diğer türden göstericiler (adresler) arasında yapılan atamalar geçerlidir. *void* türden bir göstericiye herhangi bir türden bir adres sorunsuzca atanabilir. Belirli türden bir gösterici değişkene *void* türden bir adres de aynı şekilde sorunsuzca atanabilir.

```
char *ptr;
void *vp;
/***/

ptr = vp;          /* Geçerli*/
vp = ptr;          /* Geçerli */
```

void göstericiler belirli bir türe ait olmadıkları için, tür bilgisine sahip olan göstericiler üzerinde uygulanan bazı işlemler *void* türden göstericilere uygulanamaz:

i) void türden göstericilerin * ya da [ ] işleçlerinin terimi olması geçersizdir. Bu işleçler bir nesneye erişmek için tür bilgisine gereksinim duyar.

```
void func()
{
    double a[50];
    void *vptr;

    vptr = a;   /* Geçerli */
    *vptr = 3.7;       /* Geçersiz! */
    vptr[2] = 5.6;     /* Geçersiz! */
    /****/
}
```

Yukarıdaki kod parçasında **vptr* ve *vptr[2]* ifadeleri geçersizdir.

ii) void türden bir adres ile bir tamsayının toplanması ya da *void* türden bir adresten bir tamsayının çıkartılması geçersizdir. Çünkü gösterici aritmetiğine göre bir göstericinin değeri *n* kadar artırıldığında, gösterici içindeki adresin sayısal bileşeni *n* ile göstericinin gösterdiği nesnenin tür uzunluğunun çarpımı kadar artar. *void* göstericilerin türleri olmadığı için bu durumda sayısal bileşenin ne kadar artacağı da bilinemez.
void türden göstericiler ++ ve -- işleçlerinin terimi olamaz.

```
++ptr;
```

ifadesi

```
ptr = ptr + 1;
```

ifadesine eşdeğerdir.

```
void func()
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    void *vptr = a;
    void *p;
    p = vptr + 2;  /* Geçersiz */
    ++vptr;        /* Geçersiz */
    --vptr;        /* Geçersiz */
    vptr += 2;     /* Geçersiz */
    vptr -= 3;     /* Geçersiz */
}
```

iii) Benzer şekilde, *void* türden iki adres birbirinden çıkartılamaz. Diğer türden adresler için, aynı blok içindeki iki nesneye ilişkin iki adresin birbirinden çıkartılması tamamen geçerlidir. Böyle bir ifadenin değeri iki adres arasındaki nesne sayısına denk bir tamsayı oldugunu hatırlayın.

```
void func()
{
    void *p1, *p2;
    int k;
    /***/
    k = p1 - p2;   /* Geçersiz */
}
```

void gösterici değişkenleri adreslerin yalnızca sayısal bileşenlerini saklamak amacıyla kullanılırlar. Diğer tür göstericiler arasındaki atama işlemlerinde uyarı ya da hata oluşturmadıklarından dolayı, türden bağımsız adres işlemlerinin yapıldığı işlevlerde parametre değişkeni biçiminde de bulunabilirler. Örneğin:

```
void func(void *p);
```

func isimli işlevin parametre değişkeni *void* türden bir gösterici olduğundan bu işleve argüman olarak herhangi bir türden bir adres bilgisi gönderilebilir. Yani *func* işlevi herhangi bir nesnenin adresi ile çağrılabilir. Bu durumda derleme zamanında bir hata oluşmadığı gibi, derleyiciden bir uyarı iletisi de alınmaz.
func işlevi, aldığı adresteki nesnenin türüne bağlı bir işlem yapmaz.

C dilinde işlevler *void* türden adreslere de geri dönebilir. *void* türden adreslerin herhangi bir türden gösterici değişkene atanması geçerlidir

```
void *func(void);

int main()
{
    int *p;
    char *str;

    p = func();    /* Geçerli */
    /***/
    str = func();  /* Geçerli */

    return 0;
}
```

Parametre Değişkeni void Gösterici Olan Standart C İşlevleri

Standart kütüphanede başı *mem* harfleri ile başlayan biçiminde bir grup işlev vardır. Bu işlevler türden bağımsız olarak bellek bloklarıyla ilgili genel işlemler yapar.

void göstericilerin kullanımını en iyi açıklayan örnek, standart *string.h* başlık dosyası içinde bildirilen *memcpy işlevidir.*

```
void *memcpy(void *pdest, const void *psource, size_t nbytes);
```

memcpy işlevi ikinci parametresiyle belirtilen adresten başlayarak (*psource*), *nbytes* sayıda *byte*'ı birinci parametresiyle belirtilen adresten(*pdest*) başlayarak kopyalar. İşlevin sonlandırıcı karakterle ya da yazılarla bir ilişkisi yoktur, koşulsuz bir kopyalama yapar. Yani bir blok kopyalaması söz konusudur. İşlev kopyaladığı bellek bloğunda ne olduğu ya da hangi türden bir veri olduğuyla ilgilenmeksizin kaynak adresten hedef adrese belirli sayıda *byte*'ı kopyalar. Özellikle sistem programlarında çok kullanılan bir standart işlevdir.

memcpy işleviyle örneğin aynı türden herhangi iki dizi birbirine kopyalanabilir:

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

#define    SIZE   10

int main()
{
    int a[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int b[SIZE];
    int i;

    memcpy(b, a, sizeof(int));
    for (i = 0; i  < SIZE; ++i)
        printf("%d\n", b[i]);

    return 0;
}
```

Aşağıdaki işlev çağrıları da eşdeğerdir:

```
char s[50] = "Ali";
char d[50];
strcpy(d, s);
memcpy(d, s, strlen(s) + 1);
```

memcpy işlevinin gösterici olan parametreleri *void* türdendir. Çünkü hangi türden adres geçirilirse geçirilsin, derleyicilerden herhangi bir uyarı iletisi alınmaz. Bu işlev aşağıdaki gibi tanımlanabilir:

```
void *mymemcpy(void *vp1, const void *vp2, unsigned int n)
{
    char *p1 = vp1;
    const char *p2 = vp2;
    unsigned int k;

    for (k = 0; k < n; ++k)
        p1[k] = p2[k];

    return vp1;
}
```

İşlev şu biçimde de tanımlanabilirdi:

```
#include <stdio.h>

void *mymemcpy(void *vp1, const void *vp2, size_t n)
{
    char *p1 = vp1;
    const char *p2 = vp2;

    while (n--)
        *p1++ = *p2++;

    return vp1;
}

int main()
{
    int a[5] = {1, 2, 3, 4, 5};
    int b[5], i;

    mymemcpy(b, a, 10);
    for (i = 0; i < 5; ++i)
        printf("%d\n", b[i]);

    return 0;
}
```

Ancak *memcpy* işlevinin çakışık blokların kopyalanmasında davranışı güvence altında değildir.

memmove işlevi

memove da bir bloğu bir yerden bir yere kopyalar. İşlevin bildirimi *memcpy* işlevininki ile tamamen aynıdır.

```
void *memmove(void *dest, const void *source, unsignt int nbytes);
```

İki işlev arasındaki tek fark *memmove* işlevinin çakışık blokların kopyalanmasında davranışının güvence altında olmasıdır.
Çakışık olmayan blokların kopyalanmasında *memmove* işlevi mi tercih edilmelidir? Hayır, bu iyi bir tavsiye olamaz. Çünkü:

1. *memmove* işlevinin böyle bir güvence vermesinin ek bir yükü vardır.
2. *memmove* işlevin kullanılması durumunda kodu okuyan kişi kopyalamanın yapıldığı blokla kopyalamanın yapılacağı blokların bir şekilde çakıştığı konusunda güçlü bir izlenim edinir.

Bu durumda, kaynak blokla hedef bloğun çakışması durumunda ya da çakışma riskinin bulunduğu durumlarda *memmove* işlevi, aksi halde yani söz konusu blokların çakışmadığı kesinlikle biliniyorsa *memcpy* işlevi tercih edilmelidir.
Aşağıda *memmove* işlevi için yazılan örnek bir kod yer alıyor:

```
void *mymemmove(void *vp1, const void *vp2, unsigned int n)
{
    char *p1 = vp1;
    const char *p2 = vp2;

    if (p1 > p2 && p2 + n > p1) {
        p1 += n;
        p2 += n;
        while (n--)
            *--p1 = *--p2;
    }
```

```
    else
        while (n--)
            *p1++ = *p2++;

    return vp1;
}
```

memset işlevi

Standart bir C işlevidir. Adresi verilen bir bellek bloğunu belirli bir karakter ile yani bir *byte* değeri ile doldurmak için kullanılır. İşlevin bildirimi:

```
void *memset(void *block, int c, unsigned int nbytes);
```

Bu işlev *block* adresinden başlayarak *nbytes* büyüklüğündeki bloğu, ikinci parametresiyle belirtilen byte değeriyle doldurulur.

İşlev örneğin, herhangi bir türden bir diziyi sıfırlamak amacıyla kullanılabilir:

```
double d[100];
memset(d, 0, sizeof(d));
```

İşlev aşağıdaki gibi tanımlanabilir:

```
#include <stdio.h>

void *mymemset(void *block, int c, unsigned int n);

int main()
{
    int a[10];
    int i;

    mymemset(a, 0, sizeof(a));

    for (i = 0; i < 10; ++i)
        printf("%d\n",a[i]);

    return 0;
}

void *mymemset(void *block, int c, size_t n)
{
    char *p=(char *)block;

    while (n--)
        *p++ = c;

    return block;
}
```

memchr işlevi

Standart bir C işlevidir. Adresi verilen bir bellek bloğunda belirli bir *byte* değerini aramak için kullanılır. İşlevin *string.h* başlık dosyası içindeki bildirimi aşağıdaki gibidir:

```
void *memchr(const void *block, int c, unsigned int nbytes);
```

İşlev *block* adresinden başlayan *nbytes* içinde *c* değerine sahip bir *byte*'ı arar. İşlev ilk bulduğu *c* değerine sahip *byte*'ın adresi ile geri döner. İşlevin geri dönüş değeri *void*

türden bir adrestir. Eğer aranan *byte* bulunamaz ise işlev *NULL* adresine geri döner. İşlevin başarısı mutlaka sınanmalıdır. İşlev aşağıdaki gibi tanımlanabilir:

```
void *mymemchr(const void *p_block, int c, unsigned int nbytes)
{
    const char *p = p_block;

    while (nbytes--) {
        if (*p == c)
            return (void *)p;
        p++;
    }
    return NULL;
}
```

memcmp işlevi

Standart bir C işlevidir. Adresleri verilen iki bellek bloğunu karşılaştırmak amacıyla kullanılır. İşlevin *string.h* başlık dosyası içindeki bildirimi aşağıdaki gibidir:

```
int memcmp(const void *pblock1, const void *pblock1, unsigned int nbytes);
```

İşlev *block1* ve *block2* adreslerinden başlayan *nbytes* büyüklüğündeki iki bloğu karşılaştırır. İşlevin geri dönüş değeri *int* türdendir. Geri dönüş değerinin yorumlanması standart *strcmp* işlevine benzer:

İki bellek bloğu tamamen aynıysa, yani iki bloktaki tüm *byte'lar* birbirine eşitse, işlev *0* değerine geri döner.
Birinci bellek bloğu ikinci bellek bloğundan daha büyükse işlev *0*'dan büyük bir değere geri döner. Birinci bellek bloğu ikinci bellek bloğundan daha küçükse işlev *0*'dan küçük bir değere geri döner. Karşılaştırma şöyle yapılır: Bloklar düşük sayısal adreslerden başlayarak *byte byte* işaretsiz tamsayı olarak karşılaştırılır. Farklı olan ilk *byte* ile karşılaşıldığında, daha büyük olan tamsayıya sahip blok daha büyüktür.
İşlev aşağıdaki gibi tanımlanabilir:

```
int mymemcmp(const void *vp1, const void *vp2, size_t nbytes)
{
    const unsigned char *p1 = vp1;
    const unsigned char *p2 = vp2;
    unsigned int k;

    for (k = 0; k < nbytes; ++k)
        if (p1[k] != p2[k])
            return p1[k] < p2[k] ? -1 : 1;

    return 0;
}
```

Aşağıda yazılan işlevi sınayan bir *main* işlevi yazılıyor:

```
int main()
{
    unsigned char s1[100] = {0};
    unsigned char s2[100] = {0};

    if (!mymemcmp(s1, s2, 100))
        printf("bloklar esit!\n");

    s1[90] = 1;
```

```
    if (mymemcmp(s1, s2, 100) > 0)
        printf("birinci blok buyuk!\n");

    return 0;
}
```

DİZGELER

C dilinde iki tırnak içindeki karakterlere dizge ifadesi (*string literal*) ya da kısaca dizge (*string*) denir. Örneğin:

```
"Necati Ergin"
"x = %d\n"
"lütfen bir sayı giriniz : "
```

ifadelerinin hepsi birer dizgedir.
Dizgelerin tek bir atom olarak ele alındığını önceki konularımızdan anımsayacaksınız. C'de dizgeler, derleyiciler tarafından aslında *char* türden bir dizinin adresi olarak ele alınır. C derleyicileri, derleme aşamasında bir dizgeyle karşılaştığında, önce bu dizgeyi oluşturan karakterleri belleğin güvenli bir bölgesine yerleştirir, sonuna *sonlandırıcı karakteri* ekler. Daha sonra dizge yerine, yerleştirildiği yerin başlangıç adresini koyar. Bu durumda dizge ifadeleri aslında, dizgelerin derleyici tarafından yerleştirildiği dizinin başlangıç adresidir. Örneğin:

```
char *p;
p = "Necati Ergin";
```

gibi bir kod parçasının derlenmesi sırasında, derleyici önce *"Necati Ergin*" dizgesini belleğin güvenli bir bölgesine yerleştirir. Daha sonra yerleştirdiği yerin başlangıç adresini dizge ifadesi ile değiştirir.
Dizgeler *char* türden bir dizinin başlangıç adresi olarak ele alındığına göre, dizgelerin *char* türden gösterici değişkenlere atanmaları geçerlidir. Aşağıdaki *main* işlevini derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    printf("adres = %p\n", "Necati");
    printf("adres = %s\n", "Necati");

    return 0;
}
```

main işlevi içinde yapılan ilk *printf* çağrısında *"Necati"* dizgesi ile *%p* format karakteri eşleniyor. *printf* işlevi ile, adres bilgilerinin *%p* format karakteriyle ekrana yazdırılabileceğini anımsayın. Bu durumda çalışan kod, *"Necati"* yazısının yerleştirildiği yerin başlangıç adresini ekrana yazar. İkinci *printf* çağrısında ise *"Necati"* dizgesi %s format karakteri ile eşleniyor. Bu durumda ekrana ilgili adresteki yazı, yani *Necati* yazısı yazdırılır.
Şimdi de aşağıdaki çağrıya bakın:

```
putchar(*"Necati");
```

"Necati" dizgesinin bu kez içerik işlecinin terimi olduğunu görüyorsunuz. İçerik işleci, terimi olan adresteki nesneye erişimi sağladığına göre, bu nesnenin değeri *'N'* karakterinin kod numarasıdır. Çağrılan *putchar* işlevinin çalışmasıyla ekrana *N* karakteri basılır.
Aşağıda tanımlanan *get_hex_char* işlevi, onaltılık sayı sisteminde bir basamak değeri hangi karakter ile gösteriliyorsa, o karakterin kullanılan karakter setindeki kod numarasını döndürüyor:

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

```
int get_hex_char(int val)
{
    return "0123456789ABCDEF"[val];
}

int main()
{
    int k;

    for (k = 0; k < 16; ++k)
        putchar(get_hex_char(k));

    return 0;
}
```

İşlevin geri dönüş değeri

```
"0123456789ABCDEF"[val]
```

ifadesinin değeridir. Bu da dizgenin yerleştirildiği adresten *val* uzaklıktaki nesnenin değeridir. *char* türden bu nesnenin değeri de dizgede yer alan karakterlerden herhangi birinin sıra numarasıdır. *main* işlevi içinde *0 – 15* aralığındaki değerlere karşılık gelen karakterler bir döngü içinde *get_hex_char* işlevi ile elde edilerek *putchar* işlevine argüman olarak gönderiliyor. Programın ekran çıktısı

```
0123456789ABCDEF
```

olur.
Dizgelerin bellekte kaplayacakları yer derleme zamanında belirlenir. Aşağıdaki program her çalıştırıldığında, ekrana hep aynı adres değeri yazılır:

```
#include <stdio.h>

int main()
{
    int k;

    for (k = 0; k < 10; ++k)
        printf("%p\n", "Necati");

    return 0;
}
```

Dizgelerin İşlevlere Argüman Olarak Gönderilmesi

Parametre değişkeni *char* türden bir gösterici olan işlevi, *char* türden bir adres ile çağırmak gerektiğini biliyorsunuz. Çünkü *char* türden bir gösterici değişkene, doğal olarak *char* türden bir adres atanmalıdır.
Derleyiciler açısından dizgeler de *char* türden bir adres belirttiklerine göre, parametre değişkeni *char* türden gösterici olan bir işlevi, bir dizge ile çağırmak son derece doğal bir durumdur:

```
puts("Necati Ergin");
```

Burada derleyici *"Necati Ergin"* dizgesini belleğe yerleştirip sonuna *sonlandırıcı karakter*i koyduktan sonra artık bu dizgeyi, karakterlerini yerleştirdiği bellek bloğunun başlangıç adresi olarak görür. *puts* işlevinin parametre değişkenine de artık *char* türden bir adres kopyalanır. *puts* işlevi parametre değişkeninde tutulan adresten başlayarak sonlandırıcı karakteri görene kadar tüm karakterleri ekrana yazar. Bu durumda ekranda

```
Necati Ergin
```

yazısı çıkar.
Aşağıdaki örnekte de *"Necati Ergin*" dizgesi *str* adresine kopyalanır. Dizge ifadelerinin bulunduğu yerde, *char* türden bir dizinin adresi bulunduğu düşünülmelidir.

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

int main()
{
    char str[20];

    strcpy(str, "Necati Ergin");
    puts(str);

    return 0;
}
```

Dizgeler Salt Okunur Yazılardır

Dizgeler salt okunur bellek alanlarında tutulabilir. Bu yüzden bir dizgenin içeriğinin kaynak kod içinde değiştirilmesi yanlıştır. Dizgeleri değiştiren kodlar tanımsız davranış *(undefined behavior)* özelliği gösterir. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

int main()
{
    char *ptr = "Durak";
    *ptr = 'B'; /* Yanlış */

    puts(ptr);

    return 0;
}
```

main işlevi içinde tanımlanan *ptr* isimli gösterici değişken *"Durak*" dizgesini gösteriyor. *ptr* göstericisinin gösterdiği nesneye atama yapılması yanlıştır. Yukarıdaki program derleme zamanına yönelik bir hata içermiyor.

Özdeş Dizgeler

C derleyicileri kaynak kodun çeşitli yerlerinde tamamen özdeş dizgelere rastlasa bile bunlar için farklı yerler ayırabilir. Ya da derleyici, dizgelerin salt okunur yazılar olmasına dayanarak, özdeş dizgelerin yalnızca bir kopyasını bellekte saklayabilir. Özdeş dizgelerin nasıl saklanacağı derleyicinin seçimine bırakılmıştır. Birçok derleyici, özdeş dizgelerin bellekte nasıl tutulacakları konusunda programcının seçim yapmasına olanak verir.

Dizgelerin Karşılaştırılması

Dizgelerin doğrudan karşılaştırılması yanlış bir işlemdir.

```
/***/
if ("Ankara" == "Ankara")
    printf("dogru!\n");
else
    printf("yanlis!\n");
/***/
```

Yukarıdaki kodun çalıştırılması durumunda, ekrana *"yanlış"* ya da "*doğru*" yazdırılması güvence altında değildir. Eğer derleyici iki *"Ankara"* dizgesini bellekte ayrı ayrı yerlere yerleştirmiş ise, == karşılaştırma işleci *0* değeri üretir. Ancak bir derleyici, "Ankara" yazısını tek bir yere yerleştirip her iki dizgeyi aynı adres olarak da ele alabilir. Böyle bir durumda == karşılaştırma işleci *1* değeri üretir.
Benzer bir yanlışlık aşağıdaki kod parçasında da yapılıyor:

```
#include <stdio.h>

int main()
{
    char *pstr = "Mavi ay";
    char s[20];

    printf("parolayı giriniz : ");
    gets(s);
    if (pstr == s)
        printf("dogru parola\n");
    else
        printf("yanlış parola\n");

    return 0;
}
```

Yukarıdaki programda *s* bir dizinin ismidir. *s* ismi işleme sokulduğunda derleyici tarafından otomatik olarak dizinin başlangıç adresine dönüştürülür. *pstr* ise *char* türden bir gösterici değişkendir. *pstr* göstericisine *"Mavi ay"* dizgesi atandığında, derleyici önce *"Mavi ay"* dizgesini bellekte güvenli bir yere yerleştirir. Daha sonra dizgenin yerleştirildiği yerin başlangıç adresini p*str* göstericisine atar. Kullanıcının, parola olarak *"Mavi ay"* girişi yaptığını varsayın. Bu durumda *if* deyimi içinde yalnızca *s* adresiyle *pstr* göstericisinin değeri olan adresin eşit olup olmadığı sınanır. Bu adresler eşit olmadıkları için ekrana *"yanlış parola"* yazılır. İki yazının birbirine eşit olup olmadığı standart *strcmp* işlevi ile sınanmalıydı:

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

int main()
{
    char *pstr = "Mavi ay";
    char s[20];

    printf("parolayı giriniz : ");
    gets(s);
    if (!strcmp(pstr, s))
        printf("dogru parola\n");
    else
        printf("yanlış parola\n");

    return 0;
}
```

Tabi eşitlik ya da eşitsizlik karşılaştırması gibi, büyüklük küçüklük karşılaştırması da doğru değildir.

```
if ("CAN" > "ATA")
    printf("doğru!\n");
else
    printf("yanliş!\n");
```

Yukarıdaki *if* deyiminde *CAN* ve *ATA* isimleri *strcmp* işlevinin yaptığı biçimde, yani bir yazı olarak karşılaştırılmıyor. Aslında karşılaştırılan yalnızca iki adresin sayısal bileşenidir.

Dizgelerin Ömrü

Dizgeler statik ömürlü varlıklardır. Dizgeler, tıpkı global değişkenler gibi programın yüklenmesiyle bellekte yer kaplamaya başlar, programın sonuna kadar bellekte kalır. Dolayısıyla dizgeler çalışabilen kodu büyütür. Birçok sistemde statik verilerin toplam uzunluğunda belli bir sınırlama söz konusudur.
Dizgeler derleyici tarafından *.obj* modüle bağlayıcı program tarafından da *.exe* dosyasına yazılır. Programın yüklenmesiyle hayat kazanırlar.

Çalıştırılabilen bir program çoğunlukla üç ana bölümden oluşur:

kod
data
yığın

Kod bölümünde, işlevlerin makine kodları vardır. Data bölümünde, statik ömürlü varlıklar bulunur. Global değişkenler ile dizgeler bu bölümde bulunur. Yığın bölümü yerel değişkenlerin saklandığı bellek alanıdır. Yığın bölümü her sistemde sınırlı bir alandır. Örneğin *DOS* işletim sisteminde *64K* büyüklüğünde bir yığın alanı söz konusudur. Yani hiçbir zaman yerel değişkenlerin bellekte kapladığı alan *64K* değerini geçemez. *WINDOWS* işletim sisteminde varsayılan yığın sınır değeri *1 MB* dır. Ancak bu sınır değeri istenildiği kadar büyütülebilir.

Bir İşlevin Bir Dizgeyle Geri Dönmesi

Adrese geri dönen bir işlevin, yerel bir değişkenin ya da yerel bir dizinin adresi ile geri dönmesi gösterici hatasıdır. İşlev sonlandığında yerel değişkenler bellekten boşaltılacağı için, işlevin geri döndürdüğü adres güvenli bir adres olmaz. Aşağıdaki programda böyle bir hatayı yapılıyor:

```
#include <stdio.h>

char *getname()
{
    char s[100];

    printf("adı ve soyadı giriniz : ");
    gets(s);

    return s;      /* Yanlış! */
}

int main()
{
    char *ptr;

    ptr = getname();
    puts(ptr);

    return 0;
}
```

Ancak *char* türden bir adrese geri dönen bir işlev bir dizge ile geri dönebilir. Bu durumda bir çalışma zamanı hatası söz konusu olmaz. Dizgeler statik ömürlü varlıklar olduklarından programın çalışma süresi boyunca bellekteki yerlerini korur. Örneğin aşağıdaki işlev geçerli ve doğrudur:

```
char *getday(int day)
{
    switch (day) {
    case 0: return "Sunday";
    case 1: return "Monday";
    case 2: return "Tuesday";
    case 3: return "Wednesday";
    case 4: return "Thursday";
    case 5: return "Friday";
    case 6: return "Saturday";
}
```

Dizgelerin Birleştirilmesi

Dizgeler tek bir atom olarak ele alınır. Bir dizge aşağıdaki gibi parçalanamaz:

```
void func()
{
    char *ptr =  "Necati Ergin'in C Ders
    Notlarını okuyorsunuz";           /* Geçersiz */
    /***/
}
```

Ancak dizgelerin uzunluğu arttıkça dizgeleri tek bir satırda yazmak zorlaşabilir. Ekrandaki bir satırlık görüntüye sığmayan satırlar kaynak kodun okunabilirliğini bozar. Uzun dizgelerin parçalanmasına olanak vermek amacıyla, C derleyicileri, aralarında boşluk karakteri dışında başka bir karakter olmayan dizgeleri birleştirerek tek bir dizge olarak ele alır. Örneğin:

```
ptr = "Necati Ergin'in C Ders "
"Notlarını okuyorsunuz";
```

geçerli bir ifadedir. Bu durumda derleyici iki dizgeyi birleştirirek kodu aşağıdaki biçimde ele alır:

```
ptr = "Necati Ergin'in C Ders Notlarını okuyorsunuz";
```

Derleyicinin iki dizgeyi birleştirmesi için, dizgelerin arasında boşluk karakterlerinin (*white space*) dışında hiçbir karakterin olmaması gerekir:

```
p = "Necati" "Ergin";
```

ifadesi ile

```
p = "NecatiErgin";
```

ifadesi eşdeğerdir.
Birleştirmenin yanısıra, bir ters bölü karakteri ile sonlandırılarak sonraki satıra geçiş sağlanabilir. Örneğin:

```
ptr = "Necati Ergin'in C Ders \
Notlarını okuyorsunuz";
```

deyimi ile

```
ptr = "Necati Ergin'in C Ders Notlarını okuyorsunuz";
```

deyimi eşdeğerdir. Tersbölü karakterinden sonra, dizge aşağıdaki satırın başından devam etmelidir. Söz konusu dizge aşağıdaki gibi yazılırsa:

```
ptr = "Necati Ergin'in C Dersi \
         Notlarını okuyorsunuz";
```

satır başındaki boşluk karakterleri de dizgeye katılır. Sonuç aşağıdaki ifadeye eşdeğer olur:

```
ptr = "Necati Ergin'in C Dersi         Notlarını okuyorsunuz";
```

Dizgelerde Ters Bölü Karakter Değişmezlerinin Kullanılması

Dizgeler içinde ters bölü karakter değişmezleri de (*escape sequence*) kullanılabilir. Derleyiciler dizgeler içinde bir ters bölü karakteri gördüğünde, onu yanındaki karakter ile birlikte tek bir karakter olarak ele alır. Örneğin:

```
char *p;
p = "Necati\tErgin";
```

ifadesinde *\t* tek bir karakterdir (*9* numaralı *ASCII* karakteri olan *tab* karakteri). Yani

```
printf("dizgedeki karakter sayısı = %d\n", strlen(p));
```

ifadesi ile ekrana

```
dizgedeki karakter sayısı = 12
```

yazdırılır.
Dizge ifadelerinde doğrudan çift tırnak ya da ters bölü karakterleri kullanılamaz. Bu karakterlerin özel işlevleri vardır. "Çift tırnak" karakter değişmezinin kendisini ifade etmek için, çift tırnak karakterinden önce bir ters bölü karakteri kullanılır. Örneğin:

```
#include <stdio.h>

int main()
{
    puts("\"Necati Ergin\"");
    return 0;
}
```

main işlevi içinde yapılan

```
puts(ptr);
```

çağrısı ile ekrana

```
"Necati Ergin"
```

yazısı yazdırılır.
Dizge içinde yer alan ters bölü karakter değişmezi, onaltılık sayı sisteminde de ifade edilebilir. Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    puts("\x41\x43\x41\x42\x41");

    return 0;
}
```

Boş Dizge

Bir dizge boş olabilir (*null string*) yani içinde hiçbir karakter bulunmayabilir. Bu durum iki çift tırnak karakterinin arasına hiçbir karakter yazılmaması ile oluşturulur.

```
char *ptr = "";
```

Yukarıdaki deyim ile *ptr* göstericisine boş dizge atanıyor. Boş dizgeler de bellekte bir adres belirtir. Derleyici boş bir dizge gördüğünde, bellekte güvenilir bir yere yalnızca sonlandırıcı karakteri yerleştirir.
Boş bir dizge uzunluğu *0* olan bir dizgedir. Aşağıdaki programı derleyerek çalıştırın:

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

int main()
{
    char *ptr = "";

    printf("uzunluk = %d\n", strlen(ptr));

    return 0;
}
```

Dizgelerle Gösterici Değişkenlere İlkdeğer Verilmesi

Dizgeler kullanılarak *char* türden gösterici değişkenlere ilkdeğer verilebilir. Örneğin:

```
char *p = "İstanbul";
char *err = "Bellek yetersiz";
char *s = "Devam etmek için bir tuşa basınız";
```

Dizgeler derleyiciler tarafından *char* türden bir dizi adresi olarak ele alındığına göre, *char* türden gösterici değişkenlere dizgelerle ilkdeğer verilmesi doğal bir durumdur.
Dizilere çift tırnak içinde ilkdeğer vermek ile, gösterici değişkenlere dizgelerle ilkdeğer vermek arasındaki farka dikkat etmek gerekir:

```
char *p = "Deneme";
char s[10] = "Deneme";
```

deyimleri tamamen farklıdır.
Gösterici değişkenlere ilkdeğer verildiğinde, derleyici bunu bir dizge ifadesi olarak ele alır. Yani dizge belleğe yerleştirildikten sonra başlangıç adresi göstericiye atanır. Oysa dizilerde önce dizi için yer ayrılır, daha sonra karakterler tek tek dizi elemanlarına yerleştirilir. Dizilere ilkdeğer verirken kullanılan çift tırnak ifadeleri adres bilgisine dönüştürülmez. Dizi elemanlarına tek tek *char* türden değişmezlerle ilkdeğer verme işlemi zahmetli olduğu için, programcının işini kolaylaştırmak amacı ile böyle bir ilkdeğer verme kuralı getirilmiştir.

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

deyimi aslında

```
char s[10] = {'D', 'e', 'n', 'e', 'm', 'e', '\0'};
```

ile aynı anlamdadır.

Yazı Tutan char Türden Dizilerle Bir Dizgeyi Gösteren char Türden Göstericilerin Karşılaştırılması

C dilinde bir yazı bilgisi en az iki ayrı biçimde saklanabilir:

1. Bir yazı *char* türden bir dizi içinde saklanabilir:

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

void foo()
{
    char s1[100] = "Necati Ergin";
    char s2[100];
    char s3[100];

    printf("bir yazı giriniz: ");
    gets(s2);
    strcpy(s3, s2);
    /***/
}
```

Yukarıdaki *foo* işlevinde *s1* dizisine *Necati Ergin* yazısı ilkdeğer olarak yerleştiriliyor.
s2 dizisine ise standart *gets* işlevi çağrısıyla klavyeden bir yazı alınıyor.
s2 dizisindeki yazı, standart *strcpy* işlevine yapılan çağrı ile *s3* dizisine kopyalanıyor.

2. Yazı bir dizge olarak saklanarak *char* türden bir gösterici değişkenin bu dizgeyi göstermesi sağlanabilir:

```
char *ptr = "Necati Ergin";
```

İki yöntem, birbirinin tamamen eşdeğeri değildir. Aşağıdaki noktalara dikkat edilmesi gerekir:
Dizgeler statik ömürlü varlıklar oldukları için programın sonlanmasına kadar bellekte yer kaplar. Bir gösterici değişkenin bir dizgeyi gösterirken, daha sonra başka bir dizgeyi gösterir duruma getirilmesi, daha önceki dizgenin bellekten boşaltılacağı anlamına gelmez:

```
char *p = "Bu dizge programın sonuna kadar bellekte kalacak.";
p = "artık yukarıdaki dizge ile bir bağlantı kalmayacak...";
```

Yazının *char* türden bir dizi içinde tutulması durumunda bu yazıyı değiştirmek mümkündür. Dizi elemanlarına yeniden atamalar yapılarak yazı istenildiği gibi değiştirilebilir. Ama dizgelerin değiştirilmesi tanımlanmamış davranış özelliği gösterir, yanlıştır:

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

void foo()
{
    char *ps = "Metin";
    char *pstr = "Ankara";

    ps[1] = 'Ç';           /* Yanlış! */
    strcpy(pstr, "Bolu");  /* Yanlış! */
    /***/
}
```

GÖSTERİCİ DİZİLERİ

Göstericiler de birer nesne olduğuna göre gösterici dizileri de tanımlanabilir:
Elemanları gösterici değişkenlerden oluşan dizilere *"gösterici dizileri"* (*pointer arrays*) denir. Örnekler:

```
char *pstr[10];
int *ap[50];
float *kilo[10];
```

Gösterici dizilerinin bildirimleri, normal dizi bildirimlerinden farklı olarak '*' atomu ile yapılır. Örneğin:

```
char str[100];
```

bildiriminde, *str 100* elemanlı *char* türden bir dizi iken

```
char *pstr[100];
```

bildiriminde ise, *pstr 100* elemanlı *char* türden bir gösterici dizisidir. Yani dizinin her bir elemanı *char* * türünden nesnedir. Dizinin her bir elemanı bir gösterici değişkendir. Derleyici, bir gösterici dizisi tanımlaması ile karşılaşınca, diğer dizilerde yaptığı gibi, bellekte belirtilen sayıda gösterici nesnesini tutabilecek kadar, bitişik bir bellek bloğu ayırır. Örneğin:

```
char *p[10];
```

bildirimi ile derleyici, *p* dizisinin *10* elemanlı ve her elemanın *char* türden bir gösterici olduğunu anlar. Kullanılan sistemde göstericilerin uzunluğunun *4 byte* olduğu varsayılırsa, derleyici bu dizi için *40 byte'lık* bir bellek bloğu ayırır. Bu durumda;

```
p[0], p[1], p[2], ...p[9]
```

dizi elemanlarının her biri *char* * türündendir. Bu dizinin elemanlarından her bir nesne, diğerinden bağımsız olarak kullanılabilir.

Gösterici Dizilerine İlkdeğer Verilmesi

Gösterici dizilerine de ilkdeğer verilebilir: Örneğin:

```
int *p[] = {
     (int *) 0x1FC0,
     (int *) 0x1FC2,
     (int *) 0x1FC4,
     (int *) 0x1FC6,
     (int *) 0x1FC8
 }
```

Kuşkusuz, gösterici dizisinin elemanlarına atanan adreslerin bir amaç doğrultusunda kullanılabilmesi için, güvenli bölgeleri göstermesi gerekir. Bu nedenle uygulamada *char* türden gösterici dizisi dışındaki gösterici dizilerine ilkdeğer verilmesi durumuna sık rastlanmaz. Genellikle *char* türden gösterici dizilerine, dizge ifadeleriyle ilkdeğer verilir.

char Türden Gösterici Dizileri

Uygulamalarda en sık görülen, ch*ar* türden olan gösterici dizileridir.
Dizgelerin, C derleyicisi tarafından otomatik olarak *char* türden dizi adreslerine dönüştürüldüğünü biliyorsunuz. Bu durumda *char* türden bir gösterici dizisinin elemanlarına dizgeler ile ilkdeğer verilebilir:

```
char *aylar[] = {"Ocak", "Şubat", "Mart", "Nisan", "Mayıs", "Haziran",
"Temmuz", "Ağustos", "Eylül", "Ekim", "Kasım", "Aralık"};
```

aylar dizisinin her bir elemanı *char* türden bir göstericidir. Bu gösterici dizisine, dizgeler ile ilkdeğer veriliyor. Başka bir deyişle, *char* türden bir gösterici dizisi olan *aylar* dizisinin her bir elemanı, yılın aylarının ismi olan yazıları gösteriyor.

char Türden Gösterici Dizilerinin Kullanılma Temaları

char türden gösterici dizileri yazılarla ilgili işlemler yapan programlarda sıkça kullanılır. Program içinde sık kullanılacak yazıların, kaynak kod içinde her defasında bir dizge olarak yazılması yerine, bir gösterici dizisinin elemanlarında saklanması çok rastlanan bir temadır. Aşağıdaki örnekte *error_messages* dizisinin her bir elemanında, hata iletilerine ilişkin yazılar saklanıyor.

```
char *error_messages[] = {"Bellek Yetersiz!", "Hatalı şifre",  "Dosya
bulunamadı", "Belirtilen dosya zaten var", "sürücü hazır değil", "Okunacak
dosya açılamıyor", "yazılacak dosya açılamıyor!..", "Belirlenemeyen
hata!"};
```

Artık programın herhangi bir yerinde yazılardan biri yazdırılmak istendiğinde, gösterici dizisinin herhangi bir elemanına erişilerek ilgili yazının başlangıç adresi elde edilir.

```
/*...*/
if (fp == NULL) {
    printf("%s\n", err[5]);
    return 5;
}
```

Şimdi de aşağıdaki programı inceleyin:

```
#include <stdio.h>

int day_of_week(int day, int month, int year);

char *months[] = {"January", "February", "March", "April", "May", "June",
"July", "August", "September", "October", "November", "December"};

char *days[] = {"Sunday",   "Monday",  "Tuesday", "Wednesday", "Thursday",
"Friday",  "Saturday" };

void display_date(int day, int mon, int year)
{
    printf("%02d ", day);
    puts(aylar[mon - 1]);
    printf(" %d ", year);
    printf("%s", days[day_of_week]);
}
```

Yukarıdaki programda ismi *months* ve *days* olan *char* türden gösterici dizileri tanımlanıyor. *months* dizisinin her bir elemanı, bir ay ismini içeren yazının başlangıç adresini, *days* dizisinin her bir elemanı ise bir gün ismini içeren yazının başlangıç adresini tutuyor. Gösterici dizisinin elemanlarına dizgelerle ilkdeğer veriliyor. *display_date* isimli işlevin, bir tarihe ilişkin gün, ay ve yıl değerlerini aldığını, ilgili tarihi ekrana yazdırırken, gösterici dizilerinden faydalandığını görüyorsunuz.
Belirli sayıda bir yazı grubu içinde, bir yazının var olup olmadığının sınanması amacıyla da *char* türden gösterici dizileri sıklıkla kullanılır.
Bir yazının belirli bir yazıyla aynı olup olmadığı standart *strcmp* işleviyle sınanabilir. Peki, bir yazının, bir grup yazı içinde var olup olmadığı nasıl saptanabilir?

Aşağıda tanımlanan *get_month* işlevi, kendisine başlangıç adresi gönderilen bir yazının, İngilizce ay isimlerinden biri olup olmadığını sınıyor. Sınadığı yazı geçerli bir ay ismi ise, işlev bu ayın kaçıncı ay olduğu bilgisiyle (*1 - 12*) geri dönüyor. İşlev, kendisine gönderilen yazı bir ay ismine karşılık gelmiyor ise *0* değeriyle geri dönüyor.

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

#define    SIZE       20

int get_month(const char *str)
{
    char *months[] = {"Ocak", "Subat", "Mart", "Nisan", "Mayis", "Haziran",
                  "Temmuz", "Agustos", "Eylul", "Ekim", "Kasim", "Aralik"};
    int k;

    for (k = 0; k < 12; ++k)
        if (!stricmp(months[k], str))
            return k + 1;
    return 0;
}

int main()
{
    char s[SIZE];
    int result;

    printf("bir ay ismi giriniz .");
    gets(s);
    result = get_month(s);
    if (result)
        printf("%s yilin %d. ayidir\n", s, result);
    else
        printf("%s gecerli bir ay ismi degildir\n", s);

    return 0;
}
```

Standart olmayan *stricmp* işlevinin, iki yazının karşılaştırmasını büyük harf küçük harf duyarlılığı olmadan yapması dışında, *strcmp* işlevinden başka bir farkı bulunmadığını anımsayın.
Gösterici dizileri tıpkı diğer diziler gibi, yerel ya da global olabilir. Dizinin global olması durumunda, dizi hayata başladığında dizinin elemanlarının hepsinin içinde *0* değerleri bulunurken, yerel bir gösterici dizisinin içinde rastgele değerler olur. Dizinin her bir elemanı içinde bir adres bilgisi olduğuna göre, atama yapılmamış global gösterici dizilerinin her bir elemanı içinde *0* adresi (*NULL adresi*) bulunur. İlkdeğer verilmemiş yerel gösterici dizilerinin elemanları içinde ise rastgele adres değerleri bulunur.
Bir gösterici hatasına neden olmamak için önce gösterici dizisi elemanlarına güvenli adresler yerleştirmek gerekir. Dizgeler statik ömürlü varlıklar oldukları için, bir gösterici dizisinin elemanlarına dizgelerle değer vermek bir gösterici hatasına neden olmaz. Zira dizgeler, daha önce de belirtildiği gibi, önce derleyici tarafından bellekte güvenli bir bölgeye yerleştirilir, daha sonra ise yerleştirildikleri bloğun başlangıç adresi olarak ele alınır.
Bazı durumlarda, bir gösterici dizisinin son elemanına özellikle *NULL* adresi atanır. Böyle bir gösterici dizisi, dizi boyutu belirtilmeden işlenebilir. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

int main()
{
    int k;
    char *names[] = {"Ali", "Hasan", "Mehmet", "Sebahat", "Fatma",
                     "Tuncay", "Deniz", "Kerim", "Necati",  NULL};

    for (k = 0; names[k] != NULL; ++k)
        puts(names[k]);
    return 0;
}
```

Aşağıdaki örnekte, *char* türden bir gösterici dizisinin elemanları gösterdikleri yazıların büyüklüklerine göre sıralanıyor:

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

#define    SIZE   10

int main()
{
    int i, k;
    char *names[SIZE] = {"Ali", "Hasan", "Mehmet", "Sebahat", "Fatma",
    "Tuncay", "Kaan", "Taylan", "Aleyna", "Deniz"};

    printf("dizi yazdiriliyor!\n");
    for (k = 0; k < SIZE; ++k)
        puts(names[k]);

    for (k = 0; k < SIZE -1; ++k)
        for (i = 0; i < SIZE - 1 -k; ++i)
            if (strcmp(names[i], names[i + 1]) > 0) {
                char *temp = names[i];
                names[i] = names[i + 1];
                names[i + 1] = temp;
            }
    printf("siralanmis dizi yazdiriliyor!\n");
    for (k = 0; k < SIZE; ++k)
        puts(names[k]);

    return 0;
}
```

Sıralama amacıyla yine *"kabarcık sıralaması"* algoritması kullanılıyor. Gösterici dizisinin ardışık iki elemanının değerleri

```
strcmp(names[i], names[i + 1]) > 0
```

koşulunun doğru olması durumunda takas ediliyor. Bu, "dizinin *i* indisli elemanı olan göstericinin gösterdiği yazı, *i + 1* indisli elemanının gösterdiği yazıdan daha büyükse" anlamına gelir, değil mi? Eğer takas işlemi, daha önce diğer türden dizileri sıralarken kullanılan

```
if (names[i] >  names[i + 1] > 0)
```

koşuluyla yapılsaydı ne olurdu?

Şimdi aşağıdaki kod parçasını inceleyin:

```
#include <stdio.h>

int main()
{
    char *names[6] = {"Ali", "Hasan", "Mehmet", "Sebahat", "Fatma" "Tunc"};
    int k;

    for (k = 0; k < 6; ++k)
        puts(names[k]);

    return 0;
}
```

Böceği görebildiniz mi?
Aralarında boşluk karakteri dışında başka bir karakter olmayan dizgelerin, derleyici tarafından otomatik olarak birleştirilerek tek bir dizge olarak değerlendirildiğini biliyorsunuz. Yukarıdaki kod parçasında *names* isimli dizinin elemanlarına ilkdeğer verirken kullanılan *"Fatma"* dizgesiyle *"Tuncay"* dizgesi arasında virgül atomu konulmamış. Bu durumda derleyici bu iki dizgeyi birleştirerek tek bir dizge olarak yani *"FatmaTuncay"* biçiminde ele alır. Bu dizge gösterici dizisinin sondan bir önceki elemanına atanır. Bu durumda dizinin son elemanına *NULL* adresi atanmış olur değil mi? *main* işlevi içindeki for döngüsü de *NULL* adresine ulaşır. Şüphesiz bu bir gösterici hatasıdır.

Aşağıda bir tamsayıyı bir yazıya dönüştüren *syaz* isimli bir işlev tanımlanıyor. İşlev tanımlarında yer alan gösterici dizilerinin kullanımını inceleyiniz:

```
#include <stdio.h>

void yuzyaz(unsigned int val)
{
    static const char *birler[] = {"", "bir", "iki", "uc", "dort", "bes",
"alti", "yedi", "sekiz", "dokuz"};
    static const char *onlar[] = {"", "on", "yirmi", "otuz", "kirk",
"elli", "altmis", "yetmis", "seksen", "doksan"};
    int y = val / 100;
    int o = val % 100 / 10;
    int b = val % 10;

    if (y > 1)
        printf("%s", birler[y]);
    if (y > 0)
        printf("yuz");
    if (o > 0)
        printf("%s", onlar[o]);
    if (b > 0)
        printf("%s", birler[b]);
}

void syaz(unsigned int val)
{
    int milyar, milyon, bin, yuz;

    if (val >= 1000000000) {
        milyar = val / 1000000000;
        yuzyaz(milyar);
        printf("milyar");
    }

    if (val > 1000000) {
        milyon = val % 1000000000 / 1000000;
```

```
        yuzyaz(milyon);
         if (milyon)
            printf("milyon");
    }

    if (val > 1000) {
        bin = val % 1000000 / 1000;
        if (bin >= 1)
            yuzyaz(bin);
        if (bin > 1)
            printf("bin");
    }
    yuz = val % 1000;
    yuzyaz(yuz);
}

int main()
{
    unsigned int val;

    printf("bir tamsayi giriniz:");
    scanf("%u", &val);
    syaz(val);
    printf("\n");

    return 0;
}
```

GÖSTERİCİYİ GÖSTEREN GÖSTERİCİLER

Gösterici değişkenler içlerinde adres bilgisi tutan nesnelerdir.

```
int *p;
```

gibi bir tanımlamayla ismi *p* olan, bellekte *2* ya da *4* byte yer kaplayacak olan bir nesne yaratılmış olur. Bu nesne *int* * türündendir.
p bir nesne olduğuna göre, *p* değişkeninin de adresinden söz edilebilir.
p değişkeni *adres* işlecinin terimi olabilir:

```
&p
```

ifadesinin türü nedir? Bu ifade *int* * türünden bir nesnenin adresi olabilecek bir türdendir. Bu tür C dilinde

```
(int **)
```

türü olarak gösterilir. O zaman *p* gibi bir değişkenin adresi bir başka nesnede tutulmak istenirse, *p*'nin adresini tutacak nesne *int* ** türünden olmalıdır. Aşağıdaki kod parçasını inceleyin:

```
void func()
{
    int x = 10;
    int *p = &x;
    int **ptr = &p;
    /******/
}
```

Yukarıdaki programda *int* ** türünden olan *ptr* isimli değişkene, *int* * türünden olan *p* değişkeninin adresi atanıyor. Bunun anlamı şudur: *ptr* değişkeninin değeri, *p* değişkeninin adresidir. Başka bir deyişle, *ptr* göstericisi *p* göstericisini gösterir. Bu durumda

```
*ptr
```

ifadesi *ptr* nesnesinin gösterdiği nesnedir. Yani **ptr* ifadesi *p* nesnesinin kendisidir. **ptr* ifadesine yapılan atama aslında *p* nesnesini değiştirir. Aşağıdaki programı inceleyin:

```
#include <stdio.h>

int main()
{
    int x = 10;
    int y = 20;
    int *p;
    int **pp;

    printf("x = %d\n", x);
    printf("y = %d\n", y);
    p = &x;
    *p = 100;

    pp = &p;
    *pp = &y;
    *p = 200;
    printf("x = %d\n", x);
    printf("y = %d\n", y);
```

```
    **pp = 2000;

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

    return 0;
}
```

Yukarıdaki *main* işlevinde, *int* türden *x* değişkeninin adresi, *p* isimli gösterici değişkene atandıktan sonra
deyimiyle, *p* göstericisinin gösterdiği nesneye yani *x* değişkenine *100* değeri atanıyor.

```
*p = 100;
```

```
pp = &x;
```

deyimiyle ise *p* nesnesinin adresi *pp* isimli göstericiye atanıyor. Bu atamadan sonra *pp* göstericisi *p* göstericisini gösterir.

```
*pp = &y;
```

deyimi ile *pp* göstericisinin gösterdiği nesneye yani *p* değişkenine bu kez *y* değişkeninin adresi atanıyor. Bu atamadan sonra yürütülecek

```
*p = 200;
```

ataması ile artık *p* göstericisinin gösterdiği nesneye yani *y* değişkenine *200* değeri atanır. Şimdi de aşağıdaki deyimi inceleyin:

```
**pp = 2000;
```

İçerik işlecinin, işleç öncelik tablosunun ikinci düzeyinde yer aldığını ve sağdan sola öncelik yönüne sahip olduğunu biliyorsunuz.
Bu durumda önce **pp* ifadesi ile *pp* göstericisinin gösterdiği nesneye yani *p* nesnesine erişilir. Daha sonra **(*pp)* ifadesiyle de *pp* göstericisinin gösterdiği nesnenin gösterdiği nesneye, yani *p* nesnesinin gösterdiği nesneye, yani *y* değişkenine ulaşılır. Deyimin yürütülmesiyle *y* değişkenine *2000* değeri atanmış olur.

**pp ifadesi, *pp*'nin gösterdiği nesnenin gösterdiği nesneye, yani *y* nesnesine karşılık gelir.

Yerel Bir Gösterici Değişkenin Değerini Değiştiren İşlevler

Yerel bir nesnenin değerini değiştirecek bir işlev, yerel nesnenin adresi ile çağrılmalıdır. Yerel bir gösterici değişkenin değerini değiştirecek bir işlev de, yerel göstericinin değerini değil adresini almalıdır. Aşağıdaki örneği inceleyin:

```
void swap_ptr(int **p1, int **p2)
{
    int *temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

Yukarıdaki programda *int* * türünden iki nesnenin değerini takas etmek amacıyla *swap_ptr* isimli bir işlev tanımlanıyor. İşlevin *int* ** türünden iki parametresi olduğunu görüyorsunuz. Bu işlev şüphesiz değerlerini takas edeceği nesnelerin adresleri ile çağrılmalıdır:

```
#include <stdio.h>

int main()
{
    int x = 10;
    int y = 20;
    int *p = &x;
    int *q = &y;

    printf("*p = %d\n", *p);
    printf("*q = %d\n", *q);
    swap_ptr(&p, &q);
    printf("*p = %d\n", *p);
    printf("*q = %d\n", *q);

    return 0;
}
```

main işlevi içinde *int* türden *x* ve *y* isimli değişkenler tanımlanıyor. Daha sonra tanımlanan *p* gösterici değişkeni *x* nesnesini, *q* gösterici değişkeni ise *y* değişkenini gösteriyor.
Daha sonra çağrılan *swap_ptr* işlevine *p* ve *q* göstericilerinin adresleri gönderiliyor. İşleve yapılan çağrıdan sonra, *p* göstericisi *y* değişkenini, *q* göstericisi ise *x* değişkenini gösterir.
Şimdi de aşağıdaki işlevi inceleyin:

```
void ppswap(int **p1, int **p2)
{
    int temp = **p1;
    **p1 = **p2;
    **p2 = temp;
}
```

ppswap işlevi hangi nesnelerin değerlerini takas ediyor?

Bir Gösterici Dizisi Üzerinde İşlem Yapan İşlevler

Bir dizi üzerinde işlem yapan işlevin dizinin başlangıç adresi ile dizinin boyutunu alması gerektiğini biliyorsunuz.

int türden bir dizi ile ilgili işlem yapan bir işlevin bildirimi aşağıdaki gibi olabilir:

```
void process_array(int *, int size);
```

Böyle bir işlev dizinin ilk elemanının adresi ve dizinin boyutu ile çağrılır, değil mi?

```
int a[10];
```

gibi bir dizi söz konusu olduğunda, dizi ismi olan *a*, işleme sokulduğunda otomatik olarak bu dizinin adresine dönüştürülür.
Yani derleyici açısından bakıldığında *a* ifadesi *&a[0]* ifadesine eşdeğerdir.
Bu işlev

```
process_array(a, 10);
```

biçiminde çağrılabilir.
Bu kez elemanları *char* * türden olan bir dizi tanımlanmış olsun:

```
char *a[100];
```

Yine *a* ifadesi bir işleme sokulduğunda, bu dizinin ilk elemanı olan nesnenin adresine, yazni dizinin başlangıç adresine dönüştürülür.

Bu dizinin ilk elemanı olan nesne *char* * türünden olduğuna göre, bu nesnenin adresi *char* ** türündendir. Böyle bir dizi üzerinde işlem yapacak işlev, bu dizinin başlangıç adresi ile dizinin boyutunu alacağına göre, aşağıdaki gibi bildirilmelidir:

```
void process_array(char **parray, int size);
```

Bu işlev

```
process_array(a, 10);
```

biçiminde çağrılabilir.
Şimdi de aşağıdaki programı inceleyin:

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

void swap_ptr(char **p1, char **p2)
{
    char *temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

void display_str_array(char **p, int size)
{
    int k;

    for (k = 0; k < size; ++k)
        printf("%s ", p[k]);
    printf("\n");
}

void sort_str_array(char **p, int size)
{
    int i, k;

    for (k = 0; k < size - 1; ++k)
        for (i = 0; i < size - 1 - k; ++i)
            if (strcmp(p[i], p[i + 1]) > 0)
                swap_ptr(p + i, p + i + 1);
}

int main()
{
    char *names[10] = {"Eda", "Abdurrahman", "Berk", "Zarife", "Yusuf",
    "Levent", "Sezgi", "Sukufe", "Ufuk", "Cansu"};

    display_str_array(names, 10);
    sort_str_array(names, 10);
    getchar();
    display_str_array(names, 10);

    return 0;
}
```

ÇOK BOYUTLU DİZİLER

C dilinde iki ya da daha çok boyuta sahip diziler tanımlanabilir:

```
int a[5][10][20];
```

Yukarıdaki bildirimle *a* dizisi *3* boyutlu bir dizidir.

```
int m[5][10];
```

m dizisi *2* boyutlu bir dizidir.

Uygulamalarda daha çok kullanılan *2* boyutlu dizilerdir. 2 boyutlu diziler matris olarak da isimlendirilir.

Matrisler

C'de iki boyutlu bir dizi aslında belirli bir boyuttaki tek boyutlu dizilerin dizisi olarak ele alınır:

```
int a[5][10];
```

a dizisi her elemanı *10* elemanlı *int* türden bir dizi olan *5* elemanlı bir dizidir. Yani *a* dizisinin gerçekte boyutu *5'* tir. İkinci köşeli ayraç içinde yer alan *10* ifadesi *a* dizisinin elemanları olan dizilerin boyutudur.

Bu dizi tüm diğer diziler gibi bellekte bitişik bir blok halinde bulunur. Yani derleyici böyle bir dizi için bellekte *50 x sizeof(int) byte* büyüklüğünde bir blok ayarlar.

Çok boyutlu bir dizinin elemanlarıyla, bu dizinin problem düzlemindeki karşılığı olan matrisin elemanlarını birbirlerine karıştırmak, sık yapılan hatalardandır.
5 x 10 boyutunda bir matrisin *50* elemanı vardır. Ancak yukarıdaki *a* dizisinin yalnızca *5* elemanı vardır.

Yukarıdaki dizide yer alan toplam *50* tane *int* türden nesnenin her birine nasıl ulaşılabilir? Bunun için köşeli ayraç işleci dizi ismiyle birlikte *2* kez kullanılabilir:

Aşağıdaki programı inceleyin. Bu programda bir matriste yer alan tüm elemanlara *0-100* aralığında rastgele değerler veriliyor. Daha sonra matriste yer alan tüm elemanların değeri ekrana yazdırılıyor:

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

#define    ROW      5
#define    COL     10

int main()
{
    int a[ROW][COL];
    int i, k;

    srand(time(0));

    for (i = 0; i < ROW; ++i)
        for (k = 0; k < COL; ++k)
            a[i][k] = rand() % 100;
    /********/
```

```
    for (i = 0; i < ROW; ++i) {
        for (k = 0; k < COL; ++k)
            printf("%2d ", a[i][k]);
        printf("\n");
    }

    return 0;
}
```

Yukarıdaki kodu inceleyin. *a* dizisi *10* elemanlı int türden dizilerin oluşturduğu bir dizidir. Yani *a[0]* bu dizilerden ilkine ve *a[4]* bu dizilerden sonuncusuna karşılık gelir. *a[0]* bu dizilerden ilkine karşılık geldiğine göre a dizisinin ilk elemanı olan *10* elemanlı dizinin örneğin *5* indisli elemanına

```
a[0][5]
```

biçiminde ulaşılabilir, değil mi? Köşeli ayraç işlecinin birinci öncelik seviyesinde ve soldan sağa doğru öncelik yönüne sahip bir işleç olduğunu anımsayın.

Böyle bir iki boyutlu diziye, problem düzlemindeki karşılığı olarak, yani bir matris olarak bakıldığında, bu matriste yer alan ilk eleman

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

nesnesidir. Bu nesne matrisin *1.* satır ve *1.* sütununda yer alan nesnedir. Matrisin son elemanı ise

```
a[4][9]
```

nesnesidir. Bu nesne matrisin *5.* satır ve *10.* sütununda yer alan nesnedir. Matristeki ilk nesnenin adresi *int* türden bir göstericiye atanıp bu gösterici sürekli artırılırsa, matristeki tüm elemanların adresleri elde edilerek son elemanın adresine ulaşılabilir. Şimdi yukarıdaki örneğe bir ekleme yapalım:

```
int main()
{
    int a[ROW][COL];
    int i, k;
    int *ptr;

    srand(time(0));

    for (i = 0; i < ROW; ++i)
        for (k = 0; k < COL; ++k)
            a[i][k] = rand() % 100;

    /********/

    for (i = 0; i < ROW; ++i) {
        for (k = 0; k < COL; ++k)
            printf("%2d ", a[i][k]);
        printf("\n");
    }
    printf("*****************************************\n");

    ptr = &a[0][0];
```

```
    i = ROW * COL;

    while (i--)
        printf("%2d ", *ptr++);
    printf("\n");

    return 0;
}
```

Daha önce yazılan *main* işlevine bir eklemenin yapıldığını görüyorsunuz.

```
ptr = &a[0][0];
```

deyimi ile

int * türünden *ptr* nesnesine matris içinde yer alan ilk *int* türden nesnenin adresi atanır. *i* değişkenine *ROW * COL* ifadesinin değeri, yani matriste yer alan toplam eleman sayısı atanmıştır.

Bu durumda *while (i--)* döngüsü matrisin eleman sayısı kadar döner, değil mi? Döngünün gövdesinde yer alan *printf* çağrısı ile döngünün her turunda **ptr* nesnesinin değeri yazdırılmış ve daha sonra sonek konumunda olan ++ işleciyle göstericinin değeri *1* artırılmış yani göstericinin bellekte bir sonraki *int* türden nesneyi göstermesi sağlanmıştır. Böylece matris içinde yer alan bütün değerler ekrana yazdırılmıştır.
Buradan şu anlaşılmalıdır: İki boyutlu dizinin elemanları aslında belleğe ardışıl olarak yerleştirilen tek boyutlu dizilerdir.

İki Boyutlu Dizilerin İşlevlere Geçirilmesi

Aslında iki boyutlu bir dizinin işleve geçirilmesi tek boyutlu dizilerin işleve geçirilmesinden farklı değildir. Bir diziyi işleve geçirmek için dizinin ilk elemanının adresi ve dizinin boyutu işleve gönderilir, değil mi?

Peki yukarıdaki örnekte yer alan *a* gibi bir dizinin ilk elemanı nedir? Bu dizinin ilk elemanı *a[0]*'dır. Ve bu *10* elemanlı *int* türden bir dizidir. Şimdi bu elemanın adresinin alındığını düşünelim:

```
&a[0]
```

Bu ifadenin türü nedir? C diline yeni başlayanlar böyle bir ifadenin türünün (*int* **) türü olması gerektiğini düşünürler. Oysa bu ifadenin türü daha önce karşılaşmadığımız bir türdür:
10 elemanlı *int* türden bir dizinin adresi olabilecek bir tür!
Bu tür biligisi C dilinde şöyle ifade edilir:

```
int (*)[10];
```

Yani a dizisinin ilk elemanı bu türden bir göstericiye atanabilir:

```
int (*ptr)[10] = &a[0];
```

Bir dizinin ismi bir ifade içinde işleme sokulduğunda otomatik olarak dizinin ilk elemanın adresine dönüştürülür, değil mi? O zaman *&a[0]* ifadesi yerine doğrudan *a* ifadesi de yazılabilir:

```
int (*ptr)[10] = a;
```

Evet, *a* ifadesinin türü *int* ** değil, *int (*)[10]* türüdür.

Bu durum şöyle de ifade edilebilir: *ptr int* türden *10* elemanlı bir diziyi gösteren göstericidir. *ptr* herhangi bir boyuttaki *int* türden bir diziyi değil, yalnızca *10* elemanlı *int* türden bir diziyi gösterebilir. *ptr* göstericisi örneğin ++ işleci ile *1* artırılırsa bellekte yer alan bir sonraki *10* elemanlı *int* türden diziyi gösterir. Yani

```
ptr + 1
```

adresinin sayısal bileşeni *ptr* adresinin sayısal bileşeninden *sizeof(int) * 10* kadar daha büyüktür.

ptr + 1 ifadesi iki boyutlu *a* dizisinin ikinci elemanı olan *10* elemanlı *int* türden dizinin adresine karşılık gelir. Bu durumda

```
*(ptr + 1)
```

ifadesi aslında *a[1]* dizisidir.

Şüphesiz **(ptr + 1)* ifadesi yerine *ptr[1]* ifadesi de kullanılabilir. Bu durumda bu gösterici yardımıyla iki boyutlu dizinin tüm elemanlarına bir döngü ile ulaşılabilir:

```
for (k = 0; k < ROW; ++k)
    ptr[k]
```

gibi bir döngü deyimi ile döngünün her turunda matrisin bir satırına, yani iki boyutlu dizinin bir elemanı olan *COL* uzunluğundaki dizilere ulaşılır.

Bu durumda *2* boyutlu bir dizi yani bir matris üzerinde işlem yapacak bir işlevin parametre değişkeni böyle bir gösterici olabilir. İşlevin diğer parametresi de dizinin boyutunu alabilir. Aşağıdaki programı inceleyin:

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

#define    ROW        5
#define    COL        10

void fill_matrix(int (*ptr)[COL], int size)
{
    int i, k;

    for (i = 0; i < size; ++i)
        for (k = 0; k <COL; ++k)
            ptr[i][k] = rand() % 100;
}

void display_matrix(int (*ptr)[COL], int size)
{
    int i, k;

    for (i = 0; i < size; ++i) {
        for (k = 0; k <COL; ++k)
            printf("%2d ", ptr[i][k]);
        printf("\n");
    }
}
```

```
int main()
{
    int a[ROW][COL];

    srand(time(0));

    fill_matrix(a, ROW);
    display_matrix(a, ROW);

    return 0;
}
```

*int (*ptr)[10]* gibi bir gösterici bir işlevin parametre değişkeni olarak kullanıldığında *int ptr[][10]* biçiminde de yazılabilir. Yani aşağıdaki iki bildirim birbirine eşdeğerdir:

```
void fill_matrix(int (*ptr)[10], int size);
void fill_matrix(int ptr[][10], int size);
```

Hatta istenirse ilk köşeli ayraç içine de bir tamsayı yazılabilir. Ancak derleyici açısından bu tamsayının bir önemi yoktur. Bazı programcılar yalnızca okunabilirlik açısından dizinin boyutunu ilk köşeli ayraçların içine yazarlar. Yukarıdaki bildirim şöyle de yapılabilirdi:

```
void fill_matrix(int ptr[5][10], int size);
```

Matrisleri işleve geçirmenin bir başka yolu da matristeki ilk elemanın adresini, matrisin satır ve sütun sayısını işleve geçirmek olabilir. Yukarıda yazdığımız işlevleri şimdi değiştiriyoruz:

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

#define     ROW     5
#define     COL     10

void fill_matrix(int *ptr, int rowsize, int colsize)
{
    int i, k;

        for (i = 0; i < rowsize; ++i)
        for (k = 0; k < colsize; ++k)
                ptr[i + k] = rand() % 100;
}

void display_matrix(int *ptr, int rowsize, int colsize)
{
    int i, k;

    for (i = 0; i < rowsize; ++i) {
            for (k = 0; k < colsize; ++k)
                    printf("%d ", ptr[i + k]);
            printf("\n");
        }
}

int main()
{
    int a[ROW][COL];

    srand(time(0));
```

```
    fill_matrix(&a[0][0], ROW, COL);
    display_matrix(&a[0][0], ROW, COL);

    return 0;
}
```

Bu kez işlevler matristeki ilk elemanın adresini, matrisin satır ve sütun sayısını alarak matristeki tüm elemanlara, bunların bellekte bitişik olarak yer aldığı bilgisinden hareketle gösterici aritmetiğiyle ulaşıyor.

Tabi bu işlevlere dizi ismi de argüman olarak geçilebilir. Bu durumda tür dönüştürme işleci kullanılmalıdır:

```
void fill_matrix((int *)a, ROW, COL);
void display_matrix((int *)a, ROW, COL);
```

Matris Elemanı Olan Dizilerin İlk Elemanlarının Adresi

Dizi isimlerinin derleyici tarafından otomatik olarak dizilerin ilk elemanının adresine dönüştürüldüğünü biliyorsunuz. Benzer biçimde iki boyutlu bir dizinin elemanı olan tek boyutlu bir diziye köşeli ayraç işleci ile ulaşıldığında, bu ifade otomatik olarak bu tek boyutlu dizinin ilk elemanının adresine dönüştürülür. Aşağıdaki örneği inceleyin:

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

#define     ROW           5
#define     COL           10

void fill_array(int *ptr, int size)
{
    int k;

    for (k = 0; k < size; ++k)
        ptr[k] = rand() % 100;
}

void display_array(const int *ptr, int size)
{
    int k;

    for (k = 0; k < size; ++k)
        printf("%2d ", ptr[k]);

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

Bu kez örnekte yer alan *display_array* isimli işlev, başlangıç adresini ve boyutunu aldığı *int* türden bir dizinin elemanlarının değerlerini ekrana yazdırırken, *fill_array* işlevi ise başlangıç adresini ve boyutunun aldığı dizinin elemanlarına rastgele değerler atıyor.

```
int main()
{
    int a[ROW][COL];
    int i;
    srand(time(0));

    for (i = 0; i < ROW; ++i)
        fill_array(a[i], COL);
```

```
    /******/

    for (i = 0; i < ROW; ++i)
        display_array(a[i], COL);

    return 0;
}
```

main işlevi içinde yer alan *for* döngü deyimini inceleyelim:

```
for (i = 0; i < ROW; ++i)
    fill_array(a[i], COL);
```

a[i] ifadesi döngünün her turunda *a* dizisinin elemanı olan *COL* boyutundaki tek boyutlu dizilere karşılık gelir değil mi? Bu ifade işleme sokulduğunda derleyici tarafından otomatik olarak bu dizinin ilk elemanının adresine dönüştürülür.

a[0] ifadesi otomatik olarak *&a[0][0]* ifadesine dönüştürülür. *fill_array* isimli işleve ikinci argüman olarak da *COL* ifadesi gönderilir. Çünkü *COL* ifadesi adresi *a[i]* olan dizinin boyutunu gösterir.
Bir dizinin tanımı sırasında dizi boyutunu gösteren ifadelerin değişmez ifadesi *(constant expresion)* olması gerektiğini biliyorsunuz. Aynı koşul çok boyutlu diziler için de geçerlidir. Eğer bir matrisin boyutları derleme zamanında değil de programın çalışma zamanında belli oluyorsa dinamik bellek yönetimi kullanılmalıdır. Bu konudaki örnekler *"Dinamik Belek Yönetimi"* bölümünde ele alınmaktadır.

İki Boyutlu Dizilere İlkdeğer Verilmesi

İki boyutlu dizilere de ilkdeğer verilebilir. Verilen ilkdeğerler sırasıyla iki boyutlu dizinin elemanı olan tek boyutlu dizilerin elemanlarına atanır.

```
#include <stdio.h>

int main()
{
    int a[3][5] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
    int i, k;

    for (i = 0; i < 3; ++i) {
        for (k = 0; k < 5; ++k)
            printf("%2d ", a[i][k]);
        printf("\n");
    }
    return 0;
}
```

İstenirse ilkdeğerleri içeren bloğun içinde içsel bloklar kullanılarak, eleman olan dizilerin belirli sayıda elemanına ilkdeğer verilebilir. İlk değer verilmeyen elemanlara otomatik olarak *0* değeri atanır. Aşağıdaki örneği de derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    int a[3][5] = { {1, 2}, {3, 4, 5}, {6, 7, 8, 9}};
    int i, k;

    for (i = 0; i < 3; ++i) {
        for (k = 0; k < 5; ++k)
```

```
            printf("%2d ", a[i][k]);
        printf("\n");
    }
    return 0;
}
```

char Türden İki Boyutlu Diziler

Nasıl bir yazı *char* türden bir dizinin içinde tutulabiliyor ise, mantıksal bir ilişki içindeki *n* tane yazı iki boyutlu bir dizi içinde tutulabilir:

```
char words[10][50];
```

Yukarıda tanımlanan *words* isimli dizinin *10* elemanı vardır. *words* isimli dizinin her bir elemanı *char* türden *50* elemanlı dizidir. *words* dizisinin içinde uzunluğu *49* karakteri geçmeyen, *10* tane yazı tutulabilir.

```
words[3]
```

Bu yazılardan dördüncüsünün adresi

```
words[6][2]
```

Bu yazılardan yedincisinin üçüncü karakteridir.

Aşağıdaki programı inceleyin:

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

#define     ARRAY_SIZE              10

char *name_array[ARRAY_SIZE] = {"Ali", "Veli", "Hasan", "Necati", "Deniz",
"Kaan", "Selami","Salah", "Nejla", "Figen"};

int main()
{
    char names[ARRAY_SIZE][20];
    int k;

    for (k = 0; k < ARRAY_SIZE; ++k)
        strcpy(names[k], name_array[k]);

    for (k = 0; k < ARRAY_SIZE; ++k)
        printf("(%s) ", names[k]);
    printf("\n");

    for (k = 0; k < ARRAY_SIZE; ++k)
        strrev(names[k]);

    for (k = 0; k < ARRAY_SIZE; ++k)
        printf("(%s) ", names[k]);
    printf("\n");

    return 0;
}
```

main işlevi içinde iki boyutlu *names* isimli bir dizi tanımlanıyor.

```
char names[ARRAY_SIZE][20];
```

Bu dizi içinde uzunluğu en fazla *19* karakter olabilecek *ARRAY_SIZE* sayıda isim tutulabilir, değil mi?

```
for (k = 0; k < ARRAY_SIZE; ++k)
    strcpy(names[k], name_array[k]);
```

döngü deyimiyle *name_array* isimli bir gösterici dizisinin elemanlarının gösterdiği isimler, iki boyutlu *names* dizisinin elemanı olan *char* türden dizilere standart *strcpy* işleviyle kopyalanıyor. Aşağıdaki döngü deyimiyle her bir isim ekrana yazdırılıyor:

```
for (k = 0; k < ARRAY_SIZE; ++k)
    printf("(%s) ", names[k]);
```

Aşağıdaki döngü deyimi ile standart olmayan *strrev* isimli işleve yapılan çağrılarla, iki boyutlu dizi içinde tutulan isimlerin hepsi ters çevriliyor:

```
for (k = 0; k < ARRAY_SIZE; ++k)
    strrev(names[k]);
```

char Türden İki Boyutlu Dizilere İlkdeğer Verilmesi

char türden bir diziye çift tırnak içinde yer alan karakterlerle ilkdeğer verilebileceğine göre iki boyutlu bir dizinin elemanları olan *char* türden tek boyutlu dizinin elemanlarına da benzer şekilde ilkdeğer verilebilir:

```
char names[5][10] = {"Ali", "Veli", "Hasan", "Tuncay", "Deniz"};
```

Yukarıda, *names* isimli iki boyutlu dizinin elemanı olan *10* elemanlı *char* türden dizilere, çift tırnak içinde yer alan yazılarla ilkdeğer veriliyor.

Şimdi de iki boyutlu *char* türden bir dizi üzerinde işlem yapacak bazı işlevler tanımlayalım:

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

#define    ARRAY_SIZE     20

void swap_str(char *p1, char *p2)
{
    char temp[20];
    strcpy(temp, p1);
    strcpy(p1, p2);
    strcpy(p2, temp);
}

void sort_names(char ptr[][20], int size)
{
    int i, k;

    for (i = 0; i < size - 1; ++i)
        for (k = 0; k < size - 1 - i; ++k)
            if (strcmp(ptr[k], ptr[k + 1]) > 0)
                swap_str(ptr[k], ptr[k + 1]);
}

void display_names(const char ptr[][20], int size)
{
    int i;

    for (i = 0; i < size; ++i)
```

```
        printf("(%s) ", ptr[i]);
    printf("\n");
}

int main()
{
    char names[ARRAY_SIZE][20] = {"Ali", "Veli", "Hasan", "Necati",
"Deniz", "Kaan", "Selami","Salah", "Nejla", "Huseyin", "Derya", "Funda",
"Kemal", "Burak", "Burcu", "Ozlem",  "Nuri", "Metin", "Guray", "Anil"};

    display_names(names, ARRAY_SIZE);
    sort_names(names, ARRAY_SIZE);
    printf("**********************************\n");
    display_names(names, ARRAY_SIZE);

    return 0;
}
```

Yukarıda tanımlanan işlevlerden *swap_str* işlevi adreslerini aldığı iki yazıyı takas ediyor.

display_names isimli işlev ise başlangıç adresini ve boyutunu aldığı iki boyutlu dizide yer alan isimleri ekrana yazdırıyor.
sort_names isimli işlev ise başlangıç adresini ve boyutunu aldığı iki boyutlu dizi içinde tutulan isimleri küçükten büyüğe doğru sıralıyor.

char * Türden Diziyle char Türden İki Boyutlu Dizi Arasındaki Farklar

Mantıksal ilişki içinde *n* tane yazı, bir gösterici dizisi yardımıyla tutulabileceği gibi iki boyutlu bir dizi içinde de tutulabilir:

```
char *pnames[10] = {"Ali", "Veli", "Hasan", "Deniz", "Ferda", "Murat",
"Ayca", "Erdem", "Kaan", "Gurbuz"};

char names[10][20] = {"Ali", "Veli", "Hasan", "Deniz", "Ferda", "Murat",
"Ayca", "Erdem", "Kaan", "Gurbuz"};
```

Elemanları dizgeleri gösteren bir gösterici dizisinin elemanları, yalnızca okuma amacıyla kullanılabilecek yazıların başlangıç adreslerini tutar. Dizgelerin yalnızca okuma amacıyla kullanılabilecek yazılar olduğunu anımsamalısınız.

Yukarıdaki örnekte, *pnames* dizisinin elemanı olan göstericilerin gösterdiği yazılar üzerinde değişiklik yapmak, doğru değildir. Ancak *names* dizisi içinde yer alan isimler istenirse değiştirilebilir. Aşağıdaki *main* işlevini inceleyin:

```
#include <string.h>

int main()
{
    int k;

    for (k = 0; k < 10; ++k) {
        strrev(pnames[k]);         /* Yanlış */
        strrev(names[k]);          /* Yanlış değil*/
    }

    return 0;
}
```

exit, abort atexit İşlevleri

Bir C programının çalışması *main* işlevinden başlar, *main* işlevinin sonunda, ya da bu işlevin geri dönüş değeri üretmesiyle, sonlanır. Program, eğer bir başka işlevin çalıştırılması sırasında sonlandırılmak istenirse, standart *exit* ya da *abort* işlevleri çağrılabilir.

exit İşlevi

Standart bu işlev *stdlib.h* başlık dosyası içinde bildirilmiştir:

```
void exit(int status);
```

İşlev, çalıştırılmakta olan programı sonlandırmadan önce aşağıdaki temizlik işlerini yapar:

i) *atexit* işleviyle daha önce kayıt edilmiş işlevleri kayıt edilmelerine göre ters sıra içinde çağırır.
ii) Yazma amaçlı açılmış tüm dosyaların tampon alanlarını (*buffer*) boşaltır (*flush*). Açık durumda olan tüm dosyaları kapatır.
iii) *tmpfile* işleviyle açılmış olan dosyaları siler.
iv) Kontrolu, programın çalıştırıldığı sisteme, başarı durumunu ileten bir bilgiyle geri verir.

İşleve gönderilen değer *0* ya da *EXIT_SUCCESS* simgesel değişmezi ise, işlev, sisteme programın başarı nedeniyle sonladırıldığı bilgisini iletir. İşleve gönderilen argüman *1* ya da *EXIT_FAILURE* simgesel değişmezi ise, işlev, sisteme programın başarısızlık nedeniyle sonlandırıldığı bilgisini iletir.

abort İşlevi

Standart bu işlev *stdlib.h* başlık dosyası içinde bildirilmiştir:

```
void abort(void);
```

İşlev bir C programını olağandışı biçimde sonlandırmak amacıyla çağrılır. Bir C programı *abort* işlevine yapılan çağrı ile sonlandırıldığında, *atexit* işleviyle daha önce kayıt edilmiş işlevler çağrılmaz. *abort* çağrısı ile programın sonlanmasından önce bazı temizlik işlemlerinin yapılıp yapılmayacağı derleyicinin seçimindedir.
Standart *assert* makrosu içinde de bu işlev çağrılır.

atexit İşlevi

Standart bu işlevin bildirimi yine *stdlib.h* başlık dosyası içindedir.

```
int atexit (void (*func)(void));
```

atexit işlevi ile, program sonlandığında ya da *exit* işlevi çağrıldığında, çağrılması istenen bir işlev kaydedilir. Program normal dışı yollarla, örneğin *abort* ya da *raise* işleviyle sonlandırıldığında, kaydedilen işlevler çağrılmaz.
atexit işleviyle en az *32* işlev kaydedilebilir.
İşlevin parametresi, kaydedilecek işlevin adresidir. İşlevin geri dönüş değeri işlemin başarı durumunu iletir. *0* geri dönüş değeri, işlemin başarısını, *0* dışı bir değer işlemin başarısızlığını gösterir.
Kaydedilen bir işlevi kayıttan çıkarmanın bir yolu yoktur.
Kaydedilen işlevler kaydedildikleri sıra ile ters sırada çağrılırlar. Bir işlev birden fazla kez kaydedilebilir. Bu durumda birden fazla kez çalıştırılır.

DİNAMİK BELLEK YÖNETİMİ

Dinamik Bellek Yönetimi Nedir

Çalıştırılabilen bir program bir bellek alanını kullanır. Nesneler programın kullandığı bellek alanındaki bloklardır. Bir ya da birden fazla nesnenin programın çalışma zamanında kaplayacağı yer iki ayrı biçimde elde edilebilir:
1. Nesne(ler) için derleyici derleme zamanında bellekte bir yer ayarlar.
2. Nesne(ler) için yer programın çalışma zamanında elde edilir.

Programın çalışma zamanı sırasında belli bir büyüklükte bitişik (*contigous*) bir bellek alanının çalışan program tarafından ayrılmasına , böyle bir alanın istenildiği zaman sisteme geri verilmesine olanak sağlayan yöntemlere *"dinamik bellek yönetimi"* (*dynamic memory management*) denir.

C dilinde bir değişken ya da bir dizi tanımlandığı zaman, bu değişken ya da dizinin programın çalışma zamanında kaplayacağı yer, derleme zamanında derleyici tarafından ayrılır:

```
int a[100];
```

Derleme sırasında yukarıdaki gibi bir dizi tanımı ile karşılaşan derleyici bellekte -eğer kullanılan sistemde *int* türü uzunluğunun *2* byte olduğu varsayılırsa- toplam *200* byte yer ayırır. Programın çalışması sırasında bir dizinin uzunluğunu değiştirmek mümkün değildir. Eğer bu dizi yerel ise otomatik ömürlüdür. Yani içinde tanımlanmış olduğu bloğun kodunun yürütülmesi süresince hayatını sürdürür. Dizi global ise statik ömürlüdür. Yani dizi, programın çalışma süresi boyunca bellekteki yerini korur.
Dinamik bellek yönetimi araçları kullanılarak bir ya da birden fazla nesnenin programın çalışma zamanında kaplayacağı yerin elde edilmesi derleme zamanından, çalışma zamanına geciktirilebilir.
Kullanılacak bellek bloğunun elde edilmesi neden derleme zamanından çalışma zamanına kaydırılsın? Bunun önemli bazı nedenleri vardır. İlerleyen sayfalarda bu nedenlerin çoğuna değinilecek.

Bellek alanının programın çalışma zamanında elde edilmesinin en önemli nedenlerinden biri gereksinim duyulan bellek bloğunun büyüklüğünün programın çalışma zamanında belli olmasıdır.
Dizi tanımlamalarında dizi boyutunu gösteren ifade yani köşeli ayracın içindeki ifade, değişmez ifadesi olmalıdır. Bu ifade değişken içeremez. Çünkü derleyicinin dizi için bellekte yer ayırabilmesi için, dizi boyutunu derleme zamanında bilmesi gerekir. Oysa birçok uygulamada kullanılması gereken dizinin boyutu programın çalışma zamanında belirlenir. Bir dizindeki dosyaların isimlerinin küçükten büyüğe doğru sıralanmak amacıyla geçici olarak bir dizide saklanması gerektiğini düşünün. Bu amaçla kullanılacak dizinin boyutu ne olmalıdır? Bu başlangıçta belli değildir. Çünkü dizin içinde kaç dosya olduğu belli değildir. Bu tip durumlara özellikle veri tabanı uygulamalarında sık rastlanır. Bazı uygulamalarda dizilerin gerçek uzunluğu programın çalışması sırasında, ancak birtakım olaylar sonucunda kesin olarak belirlenebilir. Bu durumda dizilerle çalışan programcı herhangi bir gösterici hatasıyla karşılaşmamak için dizileri en kötü olasılığı göz önünde bulundurarak mümkün olduğunca büyük tanımlamak zorundadır. Bu da belleğin verimsiz kullanılması anlamına gelir. Üstelik tanımlanan diziler yerel ise tanımlandıkları bloğun sonuna kadar, tanımlanan diziler global ise programın çalışmasının sonuna kadar bellekte tutulur. Oysa dizi ile ilgili işlem biter bitmez, dizi için ayrılan bellek bölgesinin boşaltılması verimli bellek kullanımı için gereklidir.

Belleğin kontrolu işletim sistemindedir. İşletim sistemlerinin başka bir amaçla kullanılmayan bir bellek alanını kullanıma sunmasına yarayan sistem işlevleri vardır. Şüphesiz bu işlevler doğrudan çağrılabilir. Ancak uygulamalarda çoğunlukla standart C

işlevleri kullanılır.Yazılan kaynak kodun taşınabilirliği açısından standart C işlevleri tercih edilmelidir.

Aşağıda dinamik bellek yönetiminde kullanılan standart C işlevlerini tek tek ayrıntılı olarak inceleniyor:

malloc işlevi

malloc işlevi programın çalışma zamanı sırasında bellekten dinamik bir blok elde etmek için kullanılır. İşlevin *stdlib.h* başlık dosyası içindeki bildirimi aşağıdaki gibidir:

```
void *malloc(size_t nbyte);
```

size_t türünün, *unsigned int* ya da *unsigned long* türlerinden birinin *typedef* ismi olduğunu biliyorsunuz.
İşlev, elde edilmek istenen bloğun *byte* olarak uzunluğunu alır. Ayrılan alanın bitişik *(contiguous*) olması güvence altına alınmıştır. *malloc* işlevinin geri dönüş değeri elde edilen bellek bloğunun başlangıç adresidir. Bu adres *void* türden olduğu için, herhangi bir türden göstericiye sorunsuz bir şekilde atanabilir. Bu adres herhangi bir türden göstericiye atandığı zaman artık elde edilen blok, başlangıç adresini tutan gösterici yardımıyla bir nesne ya da bir dizi gibi kullanılabilir. *malloc* işlevinin istenilen bloğu ayırması güvence altında değildir. *malloc* işlevi birçok nedenden dolayı başarısız olabilir. Bellekte elde edilmek istenen alan kadar boş bellek alanının bulunmaması sık görülen bir başarısızlık nedenidir.
malloc işlevi başarısız olduğunda *NULL* adresine geri döner. İşlev çağrısının başarısı mutlaka sınanmalıdır. *malloc* işlevi ile bellekte bir blok elde edilmesine ilişkin aşağıda bir örnek veriliyor:

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

int main()
{
    int *pd;
    int k, n;

    srand(time(0));

    n = rand() % 100 + 10;
    pd = (int *) malloc(n * sizeof(int));

    if (pd == NULL) {
        printf("cannot allocate memory\n");
        exit(EXIT_FAILURE);
    }
    printf("rastgele %d sayi\n", n);

    for (k = 0; k < n; ++k) {
        pd[k] = rand() % 100;
        printf("%d ", pd[k]);
    }
    /***/
}
```

Yukarıdaki main işlevinde, *n* değişkeninin değeri programın çalışma zamanında elde standart *rand* işlevine yapılan çağrı ile rastgele olarak elde ediliyor. Daha sonra standart *malloc* işleviyle *n* tane *int* türden nesnenin sığabileceği kadar büyüklükte bir bellek bloğu elde ediliyor.

malloc işlevinin başarısı bir *if* deyimiyle sınanıyor. Başarısızlık durumunda, yani *malloc* işlevinin geri dönüş değerinin *NULL* adresi olması durumunda standart *exit* işlevi çağrılarak program sonlandırılıyor.
Daha sonra elde edilen dinamik blok *int* türden bir dizi olarak kullanılıyor. Dinamik dizinin elemanlarına *0 - 100* aralığında rastgele değerler atanıyor.
Kaynak kodun taşınabilirliği açısından kullanılacak bellek bloğunun büyüklüğü *sizeof* işleciyle elde edilmelidir.
Şüphesiz, *malloc* işlevi başarısız olduğunda programı sonlandırmak zorunlu değildir.
Atama ile sınama bir defada da yapılabilir.

```
if ((pd = (int *) malloc(n * sizeof(int)) ==  NULL) {
    printf("cannot allocate memory\n");
    exit(EXIT_FAILURE);
}
```

malloc işlevinin başarısı mutlaka sınanmalıdır. İşlevin başarısız olması durumunda, geri dönüş değeri olan adresten okuma ya da yazma yapılması, *NULL* adresinin içeriğinin alınmasına neden olur. Bu da bir gösterici hatasıdır.
Programcılar çoğu kez, küçük miktarlarda çok sayıda bloğun ayrılması durumunda, sınamayı gereksiz bulma eğilimindedir. Oysa sınama işleminden vazgeçmek yerine daha kolaylaştırıcı yöntemler denenmelidir. Örneğin *p1, p2, p3, p4, p5* gibi 5 ayrı gösterici değişkenin her biri için *n byte* dinamik alan elde edilmek istensin. Bu durumda sınama mantıksal işleçler ile tek bir *if* deyimi ile yapılabilir.

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

int main()
{
    char *p1, *p2, *p3, *p4, *p5;

    p1 = (char *)malloc(n);
    p2 = (char *)malloc(n);
    p3 = (char *)malloc(n);
    p4 = (char *)malloc(n);
    p5 = (char *)malloc(n);

    if (!(p1 && p2 && p3 && p4 && p5)) {
        printf("cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
    /*....*/
}
```

malloc çağrılarından herhangi biri başarısız olursa

```
!(p1 && p2 && p3 && p4 && p5)
```

ifadesi "doğru" olarak yorumlanacağından *if* deyiminin doğru kısmı yapılır.
Bazen de malloc işlevi programcı tarafından tanımlanan başka bir işlev ile sarmalanır:

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

void *cmalloc(size_t n)
{
    void *pd = malloc(n);
```

```
    if (!pd) {
        printf("cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
    return pd;
}
```

Yukarıda tanımlanan *cmalloc* isimli işlevin parametrik yapısı *malloc* işleviyle aynı olduğunu görüyorsunuz. İşlev *malloc* işlevinden farklı olarak, başarısızlık durumunda programı sonlandırıyor. *malloc* ile yapılacak bir dinamik blok elde etme girişiminin başarısız olması durmunda program bir hata iletisi verilerek sonlandırılacaksa, kaynak kodun sürekli yinelenmesi yerine, *cmalloc* işlevi çağrılabilir.

malloc işlevinin geri dönüş değeri *void* türden bir adres olduğu için, bu adres sorunsuzca herhangi bir türden bir gösterici değişkene atanabilir. Ancak okunabilirlik açısından *malloc* işlevinin geri dönüş değeri olan adresin, tür dönüştürme işleci yardımıyla, kullanılacak gösterici değişkenin türüne dönüştürülmesi önerilir. Böylece *malloc* işlevi ile elde edilen bloğun hangi türden nesne ya da nesnelermiş gibi kullanılacağı bilgisi kodu okuyana verilmiş olur.
C dilinin standartlaştırılmasından önceki dönemde yani klasik C döneminde, *void* türden göstericiler olmadığı için, *malloc* işlevinin geri dönüş değeri *char* türden bir adresti. Bu durumda, geri dönüş değeri olan adresin, *char* türü dışında bir göstericiye atanması durumunda tür dönüşümü bir zorunluluktu.

malloc işlevi birden fazla kez çağrılarak birden fazla dinamik alan elde edilebilir. *malloc* işlevine yapılan farklı çağrılarla elde edilen blokların bellekte bitişik olması güvence altında değildir. Bu yüzden, her bir işlev çağrısının geri dönüş değeri olan adres mutlaka bir gösterici değişkende saklanmalıdır. Aşağıdaki kod parçası ardışık *malloc* çağrılarının bitişik bellek blokları elde edeceğini varsaydığı için yanlıştır:

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

int main()
{
    int *pd;
    int i;

    pd = (int *)malloc(sizeof(int) * 10);
    if (!pd) {
        printf("cannot allocate memory!\n");
        exit(EXIT_FAILURE);
    }
    if (malloc(sizeof(int) * 10) == NULL) {
        printf("cannot allocate memory!\n");
        exit(EXIT_FAILURE);
    }

    /* Yeni blok eski bloğun hemen altında olmak zorunda değildir */
    for (i = 0; i < 20; ++i)
        pd[i] = i;
    /*...*/

    return 0;
}
```

malloc işlevi ile elde edilen bloğun içinde çöp değerler vardır. Aşağıdaki kod parçası ile bu durumu sınayınız:

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

int main()
{
    int *pd;
    int i;

    pd = (int *) malloc(10 * sizeof(int));
    if (!pd) {
        printf("yetersiz bellek!..\n");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < 10; ++i)
        printf("pd[%d] = %d\n", i, pd[i]);

    return 0;
}
```

Dinamik bellek işlevlerinin kullanımına sunulan bellek bölgesi İngilizce'de *heap* olarak isimlendirilir. C++ dilinde bu alan *free store* olarak isimlendirilir. *heap* alanının büyüklüğü sistemden sisteme değişebilir ancak çalışabilir bir program söz konusu olduğunda, çalıştırılacak programın belleğe yüklenmesiyle, belirli büyüklükte bir *heap* alanı programın kullanımına ayrılır.
Sistemlerin çoğunda *malloc* işlevinin parametre değişkeni *unsigned int (size_t)* türünden olduğuna göre, *malloc* işlevi ile *DOS* altında en fazla *65535* byte yani *64KB* büyüklüğünde bir blok elde edilebilir. Oysa *UNIX, WINDOWS* gibi *32* bitlik sistemlerde *unsigned int* türü 4 byte uzunluğunda olduğuna göre, bu sistemlerde teorik olarak, *malloc* işlevi ile *4294967295 byte* (*4 MB*) uzunluğunda bitişik bir blok elde edilebilir. Tabi bu durum, dinamik yer ayırma işleminin güvence altında olduğu anlamına gelmez.
Heap alanı da sınırlıdır. Sürekli *malloc* işlevinin çağrılması durumunda, belirli bir noktadan sonra işlevler başarısız olarak *NULL* adresine geri döner.
Aşağıda, kullanılan *heap* alanının büyüklüğü bir kod ile elde edilmeye çalışılıyor:

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

#define BLOCK_SIZE     2048

int main()
{
    long total = 0L;
    void *pd;

    for (;;) {
        pd = malloc(BLOCK_SIZE);
        if (!pd)
            break;
        total += BLOCK_SIZE;
    }
    printf("toplam heap alanı = %ld byte\n", total);
    system("pause");

    return 0;
}
```

Yukarıdaki programı *DOS* işletim sistemi için önce *BLOCK_SIZE* simgesel değişmezinin 2048 değeri ile derleyerek çalıştırın. Daha sonra programı *BLOCK_SIZE* değişmezinin değerini *1* yaparak yeniden derleyin ve çalıştırın. Toplam değerin çok daha küçüldüğünü göreceksiniz.

malloc işlevi dinamik blok elde etme işlemini programın çalışma zamanında yerine getirir. Dinamik bellek işlevleriyle elde edilen blokların kontrol edilebilmesi için heap alanında uygun bir veri yapısı ile bir tablo tutulur. Tutulan tabloda elde edilmiş her bir dinamik bloğun başlangıç adresi ve boyutu vardır. Bu tablonun kendisi de *heap* alanındadır. Bir program çalıştırıldığında heap alanı iki ayrı biçimde tüketilir:
i) malloc/calloc/ realloc işlevleri tarafından elde edilen blokların kullanıma sunulmasıyla
ii) her bir yer ayırma işlemi için ilgili tabloya bir bilginin daha eklenmesi ile.

Elde Edilen Dinamik Bloğun Geri Verilmesi

Dinamik bellek işlevlerine yapılan çağrıdan elde edilen bir blok standart işlevlerinden *free* işlevine yapılan çağrı ile sisteme geri verilebilir. *free* işlevinin bildirimi de *stdlib.h* başlık dosyası içindedir:

```
void free(void *block);
```

free işlevine daha önce *malloc*, *calloc* ya da *realloc* işlevleriyle elde edilmiş olan bir bellek bloğunun başlangıç adresi geçilmelidir. *free* işlevi çağrısıyla bu blok *heap* alanına geri verilmiş olur. *heap* alanına geri verilen blok *malloc*, *calloc* ya da *realloc* işlevleri tarafından yeniden elde edilme potansiyeline girer.
Görüldüğü gibi *free* işlevi geri verilecek bellek bloğunun yalnızca başlangıç adresini alır, daha önce ayrılmış bellek bloğunun boyutunu istemez. Çalışma zamanında *heap* alanında tutulan tablolarda, zaten elde edilmiş bellek bloklarının boyutu değeri tutulmaktadır.
free işlevine argüman olarak daha önce dinamik bellek işlevleriyle elde edilmiş bir bellek bloğunun başlangıç adresi yerine başka bir adres gönderilmesi tanımsız davranıştır, yapılmamalıdır.

```
#include <stdlib.h>

int main()
{
    char *p1, *ptr;
    char s[100];

    ptr = (char *)malloc(20);

    free(ptr);          /* Geçerli ve doğru*/
    free(p1);           /* Yanlış */
    free(s);            /* Yanlış */

    return 0;
}
```

Yukarıdaki örnekte

```
free(p1)
```

çağrısı bir çalışma zamanı hatasına neden olur. Çünkü *p1* adresinde dinamik bellek işlevleri ile elde edilmiş bir alan yoktur.

```
free(s)
```

çağrısı da bir çalışma zamanı hatasına neden olur. Çünkü *s* dizisi için yer, dinamik bellek alanından ayrılmamıştır.
Daha önce dinamik olarak elde edilmiş bir blok *free* işlevi ile *heap* alanına geri verilir, ama bu bloğun başlangıç adresini tutan gösterici herhangi bir şekilde değiştirilmez. Bloğun başlangıç adresini tutan, *free* işlevine argüman olarak gönderilen gösterici, *free* işlevine yapılan çağrıdan sonra artık güvenli olmayan bir adresi gösterir. Geri verilen bir

bellek bloğuna ulaşmak sık yapılan bir gösterici hatasıdır. Aşağıdaki kodu derleyerek çalıştırın:

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

#define        ARRAY_SIZE     100

int main()
{
    char name_entry[ARRAY_SIZE];
    char *pd;

    printf("bir isim giriniz: ");
    gets(name_entry);
    pd = (char *)malloc(strlen(name_entry + 1));
    if (pd == NULL) {
        printf("dinamik bir blok elde edilemiyor!\n");
        exit(EXIT_FAILURE);
    }

    strcpy(pd, name_entry);
    printf("yazi = %s\n", pd);
    free(pd);
    printf("yazi = %s\n", pd);  /* Yanlış! */

    return 0;
}
```

Yukarıdaki *main* işlevi içinde önce klavyeden alınan bir isim *name_entry* isimli yerel diziye yerleştiriliyor. Daha sonra çağrılan *malloc* işlevi ile klavyeden alınan ismin uzunluğundan *1 byte* daha büyük bir bellek bloğu elde ediliyor. Elde edilen bellek bloğuna yerel dizideki isim kopyalanıyor. Daha sonra dinamik bloktaki isim ekrana yazdırılıyor. Yapılan

```
free(pd)
```

çağrısıyla dinamik blok *heap* alanına geri veriliyor. Ancak geri verme işleminden sonra yapılan *printf* çağrısıyla geri verilen bloğa ulaşılıyor. Bu bir gösterici hatasıdır.
Dinamik bellek işlevleri dinamik yer ayırma ve dinamik bloğu geri verme işlemlerini programın çalışma zamanında gerçekleştirirler.
free işlevine ilişkin tipik bir hata da daha önce elde edilen bir dinamik bloğun, *free* işleviyle küçültülmeye çalışılmasıdır. Aşağıdaki kodu inceleyelin:

```
#include <stdlib.h>

int main()
{
    char *pd;

    pd = (char *)malloc(1000);
    /****/
    free(pd + 500); /* Yanlis */
    /****/
    free(pd);
    return 0;
}
```

Yukarıdaki *main* işlevinde önce *malloc* işlevi ile *1000 byte'lık* bir blok elde edilerek bloğun başlangıç adresi *pd* gösterici değişkenine atanıyor. Daha sonra çağrılan *free* işlevine *pd + 500* adresinin argüman olarak gönderildiğini görüyorsunuz. Böyle bir çağrı ile *free* işlevinin daha önce ayrılan *1000 byte*'lık alanın *500 byte*'ını geri vermesi söz konusu değildir.
pd + 500 adresi bir dinamik alanın başlangıç adresi değildir. Bu yüzden işlev çağrısı tanımsız davranış özelliği gösterir. Bir *malloc* çağrısı ile elde edilen dinamik alan *free* işlevine yapılan bir çağrıyla küçültülemez.
free işlevine dinamik bir bloğun başlangıç adresi dışında başka bir adresin geçilmesi tanımsız davranıştır. Bu durumun tek istisnası *free* işlevine *NULL* adresinin geçilmesidir. İşleve *NULL* adresinin geçilmesi tanımsız davranışa neden olmaz, yalnızca bir işleme neden olmayan (*no operation*) çağrıdır:

```
if (ptr)
    free(ptr);
```

ptr bir gösterici değişken olmak üzere yukarıdaki *if* deyiminde *ptr* değişkeninin değeri *NULL* adresi değilse *free* işlevi çağrılıyor.
Bu *if* deyimi yerine *free* işlevi doğrudan çağrılsa da kodun anlamında fazlaca bir değişiklik olmazdı.

Dinamik Bir Dizinin Kullanılması

malloc işlevi ile elde edilen bir bellek bloğu boyutu programın çalışma zamanında elde edilen bir dizi olarak kullanılabilir. Aşağıdaki programı derleyerek çalıştırın:

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

void set_array(int *ptr, int size)
{
    int k;
    for (k = 0; k < size; ++k)
        ptr[k] = rand() % 100;
}

void display_array(const int *ptr, int size)
{
    int k;
    for (k = 0; k < size; ++k)
        printf("%2d ", ptr[k]);
    printf("\n");
}
```

```
int main()
{
    int n;
    int *pd;

    printf("kac tane tamsayi : ");
    scanf("%d", &n);
    pd = (int *)malloc(sizeof(int) * n);

    if (pd == NULL) {
        printf("cannot allocate memory!\n");
        exit(EXIT_FAILURE);
    }

    srand(time(0));
```

```
    set_array(pd, n);
    display_array(pd, n);
    free(pd);

    return 0;
}
```

Yukarıdaki programda programın çalışma zamanında kaç nesnenin istendiği bilgisi kullanıcıdan standart *scanf* işleviyle *n* değişkenine alınıyor. Daha sonra *malloc* işleviyle *n* tane *int* türden nesnenin sığacağı kadar büyüklükte bir dinamik bellek bloğu elde ediliyor. Elde edilen dinamik bellek bloğunun başlangıç adresinin *pd* isimli göstericiye atandığını görüyorsunuz. Artık başlangıç adresi *pd* ve boyutu *n* olan bir dizi söz konusudur, değil mi? *set_array* ve *display_array* işlevleri bu dizinin başlangıç adresi ve boyutu ile çağırılıyor.

calloc İşlevi

calloc işlevi *malloc* işlevine çok benzer. *stdlib.h* başlık dosyası içindeki bildirimi aşağıdaki gibidir.

```
void *calloc(size_t nblock, size_t block_size);
```

calloc işlevi *heap* alanından birinci parametresi ile ikinci parametresi çarpımı kadar *byte*'lık bir bellek bloğunu elde etmek için kullanılır. İşlevin geri dönüş değeri, yer ayırma işleminin başarılı olması durumunda, elde edilen bellek bloğunun başlangıç adresidir. Yer ayırma işlemi başarılı olmazsa *calloc* işlevi de *malloc* işlevi gibi *NULL* adresine geri döner. Elemanları *int* türden olan *100* elemanlı bir dizi için dinamik yer ayırma işlem *calloc* işlevi ile aşağıdaki gibi yapılabilir:

```
void func()
{
    int *pd;
    /***/
    pd = (int*) calloc(100, sizeof(int));
    if (pd == NULL) {
        printf("cannot allocate memory\n");
        exit(EXIT_FAILURE);
    }
    /***/
}
```

Şüphesiz yukarıdaki kod parçasında *calloc* işlevi şu şekilde de çağrılabilirdi:

```
pd = (int*) calloc(sizeof(int), 100);
```

calloc işlevinin *malloc* işlevinden başka bir farkı da elde ettiği bellek bloğunu sıfırlamasıdır. *calloc* işlevi tarafından elde edilen bloğunun tüm *byte*'larında sıfır değerleri vardır. *malloc* işlevi *calloc* işlevine göre çok daha sık kullanılır. Ayrılan blok sıfırlanacaksa *malloc* işlevi yerine *calloc* işlevi tercih edilebilir.
Yeri *heap* alanından elde edilen *n* elemanlı *int* türden bir dizinin elemanları sıfırlanmak istensin. Bu işlem *malloc* işlevi ile yapılırsa, dizinin elemanları ayrı bir döngü deyimi ile sıfırlanmalıdır:

```
int main()
{
    int *pd;
    int i;
    int n = rand() % 100 + 10;
    pd = (int *) malloc(sizeof(int) * n);
```

```
    /***/
    for (i = 0; i < n; ++i)
        ptr[i] = 0;
    /***/
}
```

Oysa *calloc* işlevi zaten sıfırlama işlemini kendi içinde yapar:

```
pd = calloc(sizeof(int) , n);
```

Derleyicilerin çoğunda *calloc* işlevi, bellekte yer ayırma işlemini yapması için, kendi içinde *malloc* işlevini çağırır.

Bellekte Sızıntı

Dinamik bellek işlevleriyle *heap* alanından elde edilen bir blok *free* işlevi çağrısına kadar tutulur. Dinamik bloğun başlangıç adresi kaybedilirse, artık programın sonuna kadar bu bloğu kullanma şansı olmadığı gibi blok *heap* alanına geri de verilemez. Bu durum bellekte sızıntı (*memory leak)* olarak isimlendirilir. Bir işlev içinde dinamik bir bloğun elde edildiğini ancak işlev içinde bu bloğun *heap* alanına geri verilmediğini düşünün:

```
void process()
{
    char *pd = (char *)malloc(1000);
    if (pd == NULL) {
        printf("dinamik bir blok elde edilemiyor!\n");
        exit(EXIT_FAILURE);
    }
    /***/
}
```

process isimli işlev her çağrıldığında *1000 byte*'lık bir alan elde ediliyor. Ancak işlev içinde elde edilen dinamik alan *heap* alanına geri verilmiyor. Bu durumda *process* işlevine yapılan her çağrı ile *heap* alanından *1000* byte'lık bir alan bloke edilmiş olur. *process* işlevin ana program içinde çok sık çağrılan bir işlev olduğu düşünülürse, belirli bir çağrı sayısından sonra *heap* alanı tümüyle tüketilebilir. Yani belirli bir süre sonra artık *malloc* işlevi yeni bir bloğu elde edemeyecek duruma gelir.

realloc işlevi

realloc işlevi daha önce *malloc* ya da *calloc* işlevleriyle elde edilmiş bir bellek bloğunu büyütmek ya da küçültmek için çağrılır. İşlevin bildirimi *stdlib.h* başlık dosyası içindedir.

```
void *realloc(void *block, size_t newsize);
```

realloc işlevine gönderilen birinci argüman, daha önce elde edilen bir bellek bloğunun başlangıç adresi, ikinci argüman ise yeni bloğun toplam uzunluğudur.
realloc işlevi ile dinamik bir blok büyütülmek istendiğinde, işlev, bloğu daha yüksek sayısal adrese doğru büyütmek zorunda değildir. *realloc* işlevi, daha önce elde edilmiş bloğun bittiği yerde, istenilen uzunlukta bir blok bulamazsa ya da dinamik alanı bu yönde büyütmek istemezse, bu kez bloğun tamamı için bellekte kullanılmayan başka bir yer araştırır. Eğer istenilen toplam uzunlukta ardışıl bir blok bulunursa burayı ayırır, eski bellek bloğundaki değerleri yeni yere taşır ve eski bellek bloğunu sisteme geri verir. İstenen uzunlukta sürekli bir alan bulunamazsa işlev *NULL* adresiyle geri döner. Aşağıdaki programda yapılan dinamik bellek işlemlerini inceleyin:

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

int main()
{
    int *pd;
    int k;
    int n, nplus;

    printf("kac tamsayı : ");
    scanf("%d", &n);
    pd = (int *) malloc(sizeof(int) * n);

    if (!pd) {
        printf("bellek yetersiz!..\n");
        exit(EXIT_FAILURE);
    }

    for (k = 0; k < n; ++k)
        pd[k] = k;

    printf("kac tamsayı eklenecek: ");
    scanf("%d", &nplus);

    pd = (int *) realloc(pd, sizeof(int) * (n + nplus));

    if (pd == NULL) {
        printf("bellek yetersiz!..\n");
        exit(EXIT_FAILURE);
    }

    for (; k < n + nplus; ++k)
        pd[k] = k;

    for (k = 0; k < n + nplus; ++k)
        printf("%d ", pd[k]);

    printf("\n");

    free(pd);

    return 0;
}
```

malloc işlevi ile *int* türden *n* tane nesnenin sığacağı kadar büyüklükte bir dinamik alan elde ediliyor. *malloc* işlevi başarılı olamazsa standart *exit* işleviyle program sonlandırılıyor.
Elde edilen dinamik alandaki *n* tane nesneye bir döngü yardımıyla atamalar yapılıyor. Daha sonra çağrılan *realloc* işleviyle elde edilen dinamik alan *nplus* tane nesneyi daha içine alabilecek biçimde genişletiliyor. Eğer *realloc* işlevi başarısız olursa program sonlandırılıyor. Ardından *realloc* işlevinin geri dönüş değeri olan adresindeki *n + nplus* adet nesneye sahip dizi yine bir döngü deyimiyle yazdırılıyor.

realloc işlevi başarılı olmuşsa iki olasılık söz konusudur:
1. *realloc* işlevi daha önce elde edilen dinamik alanın hemen altında yani daha yüksek sayısal adreste, ilk bloğu büyütecek biçimde boş alan bularak ek bellek alanını buradan sağlamıştır. Bu sistemlerin çoğunda pek karşılaşılan bir durum değildir. Bu durumda *realloc* işlevinin geri dönüş değeri olan adres işlevin birinci parametresine gönderilen adrestir.

2. *realloc* işlevi daha önce ayrılan bloğun altında boş yer bulamamış ya da bu yönde bir büyütme işlemini tercih etmemiş olabilir. Ve *heap* alanında toplam

```
(n + nplus) * sizeof(int)
```

kadar *byte'lık* başka bir boş blok ayarlamış olabilir. Bu durumda *realloc* işlevi daha önce elde edilmiş alandaki değerleri de bu alana kopyalayarak eski bloğu sisteme geri vermiştir.

realloc işlevinin bu davranışından dolayı geri dönüş değeri bir gösterici değişkene mutlaka atanmalıdır. Aşağıda sık yapılan tipik bir hata görülüyor:

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

int main()
{
    char *p;

    p = malloc(10);
    if (p == NULL) {
        printf("can not allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
    /***/
    if (realloc (p, 20) == NULL) {
        printf("can not allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
    gets(p);

    return 0;
}
```

realloc işlevinin başarılı olması daha önce elde edilen bloğun altına yani daha büyük sayısal adrese doğru ek blok sağladığı anlamına gelmez. *realloc* işlevi daha önce elde edilmiş dinamik alanı başka bir yere taşıyarak yeni bloğun başlangıç adresiyle geri dönmüş olabilir. Bu durumda eski blok serbest bırakılır, artık bu adresin bir güvenilirliği yoktur.
Yukarıdaki örnekte *realloc* işlevinin, daha önce elde edilen bloğu bulunduğu yerde büyüttüğü varsayılıyor. Eğer *realloc* işlevi bellek bloğunu başka bir yere taşıyarak büyütmüşse, yukarıdaki kodda bir gösterici hatası yapılmış olur, çünkü artık *p* göstericisinin gösterdiği adres güvenilir bir adres değildir.

Bu yüzden uygulamalarda genellikle *realloc* işlevinin geri dönüş değeri, *realloc* işlevine değeri gönderilen göstericiye atanır:

```
ptr = realloc(ptr, 100);
```

Ancak bu bir zorunluluk değildir. Eğer *realloc* işlevi ile yapılan yer ayırma işleminin başarılı olmaması durumunda program sonlandırılmayacaksa ve daha önce dinamik olarak elde edilen bölgedeki değerler bir dosyaya yazılacak ise, artık *realloc* işlevinin başarısız olması durumunda, *ptr* göstericisine *NULL* adresi atanacağı için *ptr* göstericisinin daha önce elde edilen blok ile ilişkisi kesilmiş olur.
Böyle bir durumda geçici bir gösterici kullanmak uygun olur:

```
temp = realloc(ptr, 100);
if (temp == NULL)
    printf("cannot allocate memory!..\n);
```

Bu durumda *ptr* göstericisi halen daha önce elde edilen bloğun başlangıç adresini gösteriyor.

C standartları *realloc* işlevi ile ilgili olarak aşağıdaki kuralları getirmiştir:

i) Daha önce elde edilen bir bellek bloğunu büyütmesi durumunda, *realloc* işlevi bloğa eklenen kısma herhangi bir şekilde değer vermez. Yani eski bloğa eklenen yeni blok içinde çöp değerler *(garbage values)* bulunur.
realloc işlevi eğer daha önce elde edilmiş bellek bloğunu büyütemez ise *NULL* adresi ile geri döner ancak daha önce elde edilmiş olan ve büyütülemeyen bellek bloğundaki değerler korunur.

ii) Eğer *realloc* işlevine gönderilen birinci argüman *NULL* adresi olursa, *realloc* işlevi tamamen *malloc* işlevi gibi davranır. Yani:

```
realloc(NULL, 100);
```

ile

```
malloc(100);
```

çağrıları tamamen aynı anlamdadır. Her ikisi de *100 byte'lık* bir dinamik alan elde etmeye çalışır.
Buna neden gerek görülmüştür? Yani neden *malloc(100)* gibi bir çağrı yapmak yerine *realloc(NULL, 100)* şeklinde bir çağrı tercih edilebilir?
Aşağıdaki programı inceleyin:

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

void display_array(const int *ptr, int size);
int get_max(const int *ptr, int size);
int get_min(const int *ptr, int size);
double get_ave(const int *ptr, int size);
double get_stddev(const int *ptr, int size);

int main()
{
    int *ptr = NULL;
    int ch, grade;
    int counter = 0;

    for (;;) {
        printf("not girmek istiyor msunuz? [e] [h]\n");
        while ((ch = toupper(getch())) != 'E' && ch != 'H')
            ;
        if (ch == 'H')
            break;
        printf("notu giriniz : ");
        scanf("%d", &grade);
        counter++;
        ptr = (int *) realloc(ptr, sizeof(int) * counter);
        if (ptr == NULL) {
            printf("cannot allocate memory!..\n");
            exit(EXIT_FAILURE);
        }
```

```
        ptr[counter - 1] = grade;
    }

    if (counter == 0) {
        printf("hiçbir not girişi yapmadınız!..\n");
        return 0;
    }
    printf("toplam %d tane not girdiniz\n", counter);
    printf("girdi§iniz notlar aşağıda listeleniyor :\n");
    display_array(ptr, counter);
    printf("\nen buyuk not : %d\n", get_max(ptr, counter));
    printf("en küçük not : %d\n", get_min(ptr, counter));
    printf("notların ortalaması : %lf\n", get_ave(ptr, counter));
    printf("notların standart sapması . %lf\n", get_stddev(ptr, counter));
    free(ptr);

    return 0;
}
```

Yukarıdaki programda:

```
ptr = (int *) realloc(ptr, sizeof(int) * counter);
```

deyiminde, döngünün ilk turunda *ptr* göstericisinin değeri *NULL* olduğu için *realloc* işlevi *malloc* gibi davranacak ve *int* türden *1* adet nesnelik yer ayırır. Ancak döngünün daha sonraki turlarında *realloc* işlevine gönderilen adres *NULL* adresi olmayacağından, daha önce elde edilen blok döngü içinde sürekli olarak büyütülmüş olur.
Eğer *realloc* işlevine gönderilen ikinci argüman 0 olursa, *realloc* işlevi tamamen *free* işlevi gibi davranır:

```
realloc(ptr, 0);
```

ile

```
free(ptr);
```

tamamen aynı anlamdadır. Buna neden gerek görülmüştür? Yani neden

```
free(ptr)
```

gibi bir çağrı yapmak yerine

```
realloc(ptr, 0)
```

şeklinde bir tercih edilsin?

Dinamik Olarak Elde Edilen Alanın Başlangıç Adresine Geri Dönen İşlevler

Dinamik olarak ayrılan bir blok, *free* işleviyle serbest bırakılarak sisteme geri verilene kadar güvenli olarak kullanılabileceğine göre, bir işlev böyle bir bloğun başlangıç adresine geri dönebilir.
Aşağıdaki işlevi inceleyin:

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

char *getname()
{
    char s[30];
    char *ptr;

    printf("ismi giriniz : ");
    gets(s);
    ptr = (char *) malloc(strlen(s) + 1);
    if (ptr == NULL) {
        printf("cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
    return strcpy(ptr, s);
}
```

getname işlevinde önce klavyeden bir isim yerel bir diziye alınıyor. Daha sonra *strlen* işlevi kullanılarak yerel diziye alınan ismin uzunluğu bulunuyor. *malloc* işlevi ile bu ismi ve sonuna gelecek sonlandırıcı karakteri içine alacak büyüklükte bir alan dinamik olarak elde ediliyor. Daha sonra da *strcpy* işlevi kullanılarak, isim dinamik olarak elde edilen alana kopyalanıyor ve bu bloğun başlangıç adresine geri dönülüyor.
Dinamik bir yer ayırma işlemi yapan işlevin çağrılması sonucunda ayrılan bloğun geri verilmesi, işlevi çağıranın kodun sorumluluğunda olur:

```
int main()
{
    char *p =  getname();
    /*****/
    free(p)
    /*****/
    return 0;
}
```

Bir işlev içinde dinamik olarak bir bellek bloğu ayırmak ve daha sonra bu bloğun başlangıç adresine geri dönmek C dilinde çok kullanılan bir tekniktir.
Aşağıda kaynak kodu verilen *strcon* işlevi birinci parametresinde başlangıç adresi tutulan yazının sonuna ikinci parametresinde başlangıç adresi tutulan yazıyı kopyalıyor; fakat her iki adresteki yazıları da bozmadan, birleştirilmiş yazının başlangıç adresi olan bir adrese geri dönüyor:

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

char *strcon(const char *s1, const char*s2)
{
    char *ptr = malloc (strlen(s1) + strlen(s2) + 1);
    /* Basari sinamasi */

    strcpy(ptr, s1);
    strcat(ptr, s2);

    return ptr;
}
```

İşlevimiz, adresleri gönderilen yazıları dinamik bir alana kopyalayarak, dinamik alanın başlangıç adresiyle geri dönüyor. *strcpy* ve *strcat* işlevleri, birinci paramatrelerinin değerleriyle geri döndüğünden

```
strcpy(ptr, s1);
strcat(ptr, s2);
return ptr;
```

deyimleri tek bir deyim olarak yazılabilirdi, değil mi?

```
return strcat(strcpy(ptr, s1), s2);
```

Birden Fazla Dinamik Alanın Bir Gösterici Dizisiyle Denetimi

Dinamik bellek işlevleriyle elde edilen farklı blokların başlangıç adreslerinin bir gösterici dizisinin elemanlarında tutulması çok karşılaşılan bir temadır. Aşağıdaki programı inceleyin:

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

char *get_name()
{
    char name_entry[40];
    char *pd;

    printf("ismi giriniz : ");
    gets(name_entry);
    pd = (char *) malloc(strlen(name_entry) + 1);
    if (pd == NULL) {
        printf("bellek yetersiz!..\n");
        exit(EXIT_FAILURE);
    }
    return strcpy(pd, name_entry);
}

int main()
{
    char *p[10];
    int k;
    int len_sum = 0;

    for (k = 0; k < 10; ++k)
        p[k] = get_name();

    printf("isimler : \n");
    for (k = 0; k < 10; ++k) {
        printf("(%s) ", p[k]);
        len_sum += strlen(p[k]);
    }
    printf("\n");
    printf("girilen isimlerin ortalama uzunlugu = %lf\n", (double)len_sum /
10);

    for (k = 0; k < 10; ++k)
        free(p[k]);
    return 0;
}
```

Bazı durumlarda gösterici dizisinin kendisi de dinamik olarak yaratılır. Aşağıdaki programda satır ve sütün sayısı klavyeden girilen bir matrisin elemanlarına önce *0 - 100* arasında rastgele değerler veriliyor sonra matris ekrana yazdırılıyor:

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

int main()
{
    int no_of_rows, no_of_cols;
    int **prows;
    int i, k;

    srand(time(0));
    printf("satir sayisi : ");
    scanf("%d", &no_of_rows);
    printf("sutun sayisi : ");
    scanf("%d", &no_of_cols);

    prows = (int **)malloc(sizeof(int *) * no_of_rows);
    if (prows == NULL) {
        printf("dinamik bellek elde edilemiyor!\n");
        exit(EXIT_FAILURE);
    }
    for (k = 0; k < no_of_rows; ++k) {
        prows[k] = (int *) malloc(sizeof(int) * no_of_cols);
        if (prows[k] == NULL) {
            printf("dinamik bellek elde edilemiyor!\n");
            exit(EXIT_FAILURE);
        }
    }
    /****/
    for (i = 0; i < no_of_rows; ++i)
        for (k = 0; k < no_of_cols; ++k)
            prows[i][k] = rand() % 100;

    /****/
    for (i = 0; i < no_of_rows; ++i) {
        for (k = 0; k < no_of_cols; ++k)
            printf("%2d ", prows[i][k]);
        printf("\n");
        }
    /****/
    for (i = 0; i < no_of_rows; ++i)
        free(prows[i]);
    free(prows);

    return 0;
}
```

Şimdi programı *inceleyin*. Matrisin satır ve sütun sayısının kullanıcı tarafından klavyeden girilmesi sağlanıyor. Matrisin satır sayısı *nrows* sütun sayısı *ncols'*dur*. Matris nrows* tane *ncols* uzunluğunda *int* türden dinamik diziden oluşur. Yani matrisin her bir satırı bir dinamik dizidir. Bu dinamik dizilerin başlangıç adresleri yine dinamik bir gösterici dizisinde tutulur. Önce *nrows* tane *int* * türünden nesne için dinamik bir alanın ayrıldığını görüyorsunuz:

```
prows = (int **)malloc(sizeof(int *) * no_of_rows);
```

Aşağıdaki döngü ile bu dinamik dizinin her bir elemanının *int* türden *ncols* uzunluğunda dinamik bir dizinin başlangıç adresini tutulması sağlanıyor.

```
for (k = 0; k < no_of_rows; ++k) {
    prows[k] = (int *) malloc(sizeof(int) * no_of_cols);
```

Yer ayırma işlemleri tamamlandıktan sonra

```
prows[i][k]
```

ifadesiyle matrisin *i* ve *k* indisli elemanına ulaşılabilir. Çünkü *prows[i] (int *)* türünden bir nesnedir. Bu nesnenin değeri dinamik bir dizinin başlangıç adresidir. Köşeli ayraç işlecinin işleç öncelik tablosunun birinci düzeyinde olduğunu, bu öncelik düzeyinin soldan sağa öncelik yönüne sahip olduğunu hatırlayın. Bu durumda önce daha soldaki köşeli ayraç işleci bir değer üretir. Soldaki köşeli ayraç işlecinin ürettiği *(int *)* türünden değer bu kez ikinci köşeli ayraç işlecine terim olur. Bu işleç de satırlardan herhangi birini oluşturan *int* türden dinamik dizinin bir elemanına ulaştırır. Bu durumda ulaşılan *int* türden nesne, matrisin *[i][k]* indisli elemanı olur.
Ayrılan dinamik alanların geri verilmesinde dikkatli olunmalıdır. Önce matrisin satırlarını oluşturan dinamik diziler *heap* alanına geri verilmeli, daha sonra ise dinamik dizilerin başlangıç adreslerini tutan gösterici dizisinin geri verilme işlemi gerçekleştirilmelidir:

```
for (i = 0; i < no_of_rows; ++i)
    free(prows[i]);
free(prows);
```

BELİRLEYİCİLER ve NİTELEYİCİLER

Belirleyiciler (*storage class specifiers)*, bildirimler yapılırken kullanılan ve nesnelerin özellikleri hakkında derleyicilere bilgi veren anahtar sözcüklerdir.
C'de kullanılan belirleyiciler şunlardır:
auto, register, static, extern ve typedef.

Tür niteleyicileri ise nesnelerin içindeki değerlerin değiştirilip değiştirilmeyeceğine ilişkin bilgi verir.
Tür belirleyicileri şunlardır:
const, volatile.
Yer belirleyici ve tür niteleyiciler C'nin anahtar sözcükleridir.

Yer Belirleyicilerle Bildirim İşlemi

Yer belirleyici, tür belirleyici ya da tür ifade eden anahtar sözcüklerin dizilimi herhangi bir biçimde olabilir:

```
auto const unsigned long int a;
const auto unsigned long int a;
unsigned long int auto const a;
int const long auto unsigned a;
```

Yukarıdaki bildirimlerin hepsi geçerlidir. Ancak okunabilirlik açısından yer belirleyici sözcüğün daha önce yazılması önerilir.
Eğer bir bildirimde belirleyicisi ya da tür niteleyicisi bir tür bilgisi olmadan kullanılırsa, önceden seçilmiş (*default*) olan *int* türü bildirimin yapıldığı kabul edilir.

```
register int a;
```

Yukarıdaki bildirim ile

```
register a;
```

bildirimi eşdeğerdir.

Ancak okunabilirlik açısından böyle bir bildirim önerilmez.

auto Belirleyicisi

auto yalnızca yerel değişkenler için kullanılabilecek bir yer belirleyicisidir. *auto* belirleyicisinin global değişkenlerin ya da işlevlerin parametre değişkenlerininin bildiriminde kullanılması geçersizdir.
Bu anahtar sözcük, nesnenin bilinirlik alanı bittikten sonra kaybolacağını, bellekte kapladığı yerin geçerliliği kalmayacağını gösterir. Yerel değişkenler otomatik ömürlüdür. Yani bulundukları bloğa ilişkin kod çalışmaya başlandığında yaratılır, söz konusu bloğun yürütülmesi bittikten sonra yok olurlar. İşte *auto* belirleyicisi bu durumu vurgulamak için kullanılır. Zaten bir yerel değişken, başka bir yer belirleyici anahtar sözcük kullanılmadığı sürece (*default olarak*) *auto* biçiminde ele alınır. Bu durumda *auto* yer belirleyicisinin kullanımı gereksizdir:

```
{
    auto int a;
    float b;
}
```

auto yer belirleyicisi global değişkenlerle ya da parametre değişkenleriyle birlikte kullanılmaz. Örneğin :

```
auto int a;            /* Geçersiz! */
function(auto int x)   /* Geçersiz! */
{
    /***/
}
```

auto anahtar sözcüğü bazı mikroişlemcilerde uyumu korumak için düşünülmüştür. Modern sistemlerde anlamlı bir kullanımı yoktur.

register Belirleyicisi

register belirleyicisi, değişkenin "bellekte değil de *CPU* yazmaçlarının içinde" tutulması isteğini derleyiciye ileten bir anahtar sözcüktür. Değişkenlerin bellek yerine doğrudan yazmaçlarda tutulması programın çalışmasını hızlandırır.
Yazmaç (*register*) nedir? Yazmaçlar *CPU (central processing unit)* içinde bulunan tampon bellek bölgeleridir. *CPU* içindeki aritmetik ve mantıksal işlemleri yapan birimin yazmaçlar ve belleklerle ilişkisi vardır. Genel olarak *CPU* tarafından yapılan aritmetik ve mantıksal işlemlerin her iki terimi de belleğe ilişkin olamaz. Örneğin bellekte bulunan *sayi1* ve *sayi2* isimli iki değişken toplanarak elde edilen değer *sayi3* isimli başka bir bellek bölgesine yazılmak istensin. Bu C'deki

```
sayi3 = sayi1 + sayi2;
```

işlemine karşılık gelir. *CPU* bu işlemi ancak *3* adımda gerçekleştirebilir:

1. adım : Önce *sayi1* bellekten *CPU* yazmaçlarından birine çekilir.
```
MOV reg, sayi1
```

2. adım : Yazmaç ile *sayi2* toplanır.
```
ADD reg, sayi2
```

3. adım: Toplam *sayi3* ile belirtilen bellek alanına yazılır.
```
MOV sayi3, reg
```

Belleğe yazma ve bellekten okuma işlemleri yazmaçlara yazma ve yazmaçlardan okuma işlemlerine göre daha yavaştır. Çünkü belleğe erişim için bir makine zamanı gerekir. *register* belirleyicisi derleyiciye yalnızca bir isteği iletir. Yani *register* belirleyicisi ile bildirilen değişkenin yazmaçta tutulacağının bir güvencesi yoktur. Derleyiciler böyle bir isteği yerine getirmeyebilir. *CPU* yazmaçları hangi sistem söz konusu olursa olsun sınırlı sayıdadır. Bu nedenle birkaç değişkenden fazlası *register* belirleyicisi ile tanımlanmış olsa bile yazmaçlarda saklanmayabilir. C derleyicileri yazmaçlarda saklayamayacakları değişkenler için genel olarak hata veya uyarı iletileri vermez. Yani derleyiciler, tutabilecekleri yazmaç sayısından fazla *register* belirleyicisine sahip değişkenlerle karşılaştıklarında bunlara ilişkin *register* belirleyicilerini dikkate almaz.
register belirleyicileri ancak yerel ya da parametre değişkenleri ile kullanılabilir. Global değişkenler ile kullanılamazlar. Örnekler:

```
register int g;        /* Geçersiz! */

int func (register int y)     /* Geçerli */
{
    register int x;    /* Geçerli */
}
```

En fazla kaç tane değişkenin yazmaçlarda saklanabileceği bilgisayar donanımlarına ve derleyicilere bağlıdır. Ayrıca, uzunluğu tamsayı *(int)* türünden büyük olan türler genellikle yazmaçlarda saklanamaz. Bu durumlarda da derleyicilerden hata veya uyarı iletisi beklenmemelidir.
Sonuç olarak *register* belirleyicisi hızın önemli olduğu çok özel ve kısa kodlarda ancak birkaç değişken için kullanılmalıdır. Modern derleyicilerin çoğu seçime bağlı olarak kod üzerinde eniyileme (optimizasyon) yaparak bazı değişkenleri yazmaçlarda saklar. Bu durum da çoğu zaman *register* anahtar sözcüğünün kullanılmasını gereksiz kılar.

static Belirleyicisi

static belirleyicisi ancak yerel ya da global değişkenlerin bildiriminde kullanılabilir. Bu belirleyici, parametre değişkenlerinin bildiriminde kullanılamaz.
static anahtar sözcüğünün global ve yerel değişkenlerin bildiriminde kullanılması farklı anlamlara gelir.

static Anahtar Sözcüğünün Yerel Değişkenlerin Bildiriminde Kullanılması

static yer belirleyicisi ile tanımlanmış yerel değişkenler ya da yerel diziler programın çalışması boyunca bellekte kalır. Başka bir deyişle, *static* anahtar sözcüğü yerel değişkenlerin ömrünü otomatik ömürden statik ömre yükseltir. Statik yerel değişkenler tıpkı global değişkenler gibi programın çalışmaya başlamasıyla yaratılır ve programın yürütülmesi bitene kadar da bellekte tutulur.
Statik yerel değişkenler programcı tarafından ilkdeğer verildikten sonra kullanılır. İlkdeğer verme işlemi programın çalışması sırasında değil, derleme zamanında derleyici tarafından yapılır. Derleyici bellekten yer ayrılmasına yol açacak makine kodunu oluşturur. Statik yerel değişken için bellekte yer programın yüklenmesi sırasında ayrılır. Derleme zamanında gerçekte bellekte yer ayrılmaz. Bellekte yer ayırma işini yapacak makine kodu üretilir. Statik yerel değişkenler ilkdeğerleriyle birlikte belleğe yüklenir.
Aşağıdaki programı derleyerek çalıştırın:

```
#include<stdio.h>

void func1()
{
    int x = 5;
    printf("x = %d\n", x);
    x++;
}

void func2()
{
    static int y = 5;
    printf("y = %d\n", y);
    y++;
}

int main()
{
    int k;

    printf("func1 işlevi 5 kez çağrılıyor!\n");
    for (k = 0; k < 5; ++k)
        func1();
    printf("\nfunc2 işlevi 5 kez çağrılıyor!\n");
    for (k = 0; k < 5; ++k)
        func2();
    return 0;
}
```

Yukarıdaki program içinde tanımlanan *func1* işlevi her çağrıldığında *x* değişkeni yaratılır ve işlevin çalışması sonlandığında *x* değişkeni bellekten boşaltılır. Her ne kadar işlevin sonlanmasından önce *x* değişkeninin değeri *1* artırılsa da, bunun bir sonraki çağrıya hiçbir etkisi olmaz. *func1* işlevinin bir sonraki çağrısında *x* değişkeni yine *5* değeri ile başlatılır. Oysa *func2* işlevi için durum değişiktir. *func2* işlevi içindeki *y* isimli yerel değişken blok bilinirlik alanına ait fakat statik ömürlüdür. *y* değişkenine ilkdeğer verme deyimi derleyici tarafından üretilen kodun bir parçasıdır. *y* değişkeni programın başında *5* değeri ile başlatılır ve programın sonuna kadar bellekteki yerini korur. İşlev kaç kez çağrılırsa çağırılsın, bellekteki yeri değişmeyen aynı *y* değişkeni kullanılır. Bu durumda işlevin ana bloğunun sonunda *y* değişkeninin değeri *1* artırıldığı için bir sonraki çağrıda artık *y* değişkeninin değeri *6* olur. Programa ilişkin ekran çıktısını aşağıda veriyoruz.

```
func1 işlevi 5 kez çağrılıyor!
x = 5
x = 5
x = 5
x = 5
x = 5

func2 işlevi 5 kez çağrılıyor!
y = 5
y = 6
y = 7
y = 8
y = 9
```

statik Yerel Nesnelerin Kullanılmasına İlişkin Ana Temalar

Statik yerel nesnelerin kullanılmasına ilişkin ana temelarda, bu nesnelerin ömürlerinin statik olmasından faydalanılır. Bu temalarda global değişkenler de pekala kullanılabilir. Ancak global değişkenler statik ömürlü olmalarıyla birlikte dosya bilinirlik alanına aittir. Oysa statik yerel değişkenler blok bilinirlik alanına aittir. Global değişkenlerin dosya bilinirlik alanına sahip olmalarından kaynaklanan dezavantajlardan etkilenmemek için bir çok durumda işlevlerin statik ömürlü nesne gereksinimi statik yerel değişkenlerle karşılanır.

İşlevlerin Statik Yerel Nesnelerin Adresleriyle Geri Dönmeleri

Bir adres değerine geri dönen işlevlerin, yerel değişkenlerin adreslerine geri dönmemeleri gerektiğini biliyorsunuz. Çünkü yerel değişkenler otomatik ömürlüdür ve işlevin çalışması sonunda bellekten boşaltılır. Ancak *static* anahtar sözcüğü ile tanımlanmış yerel değişkenler, programın sonlanmasına kadar bellekteki yerlerini koruyacağından, işlevin statik bir yerel değişkenin adresine geri dönmesinde bir sakınca yoktur:

```
#include <stdio.h>

char *get_name()
{
    static char entry[100];
    printf("bir isim giriniz : ");
    gets(entry);

    return entry;
}
```

Ancak statik bir yerel dizinin adresiyle geri dönmek, dinamik bir dizinin yani *malloc* işleviyle yeri elde edilmiş bir dizinin adresiyle geri dönmekten farklıdır. İşlev her çağırıldığında aynı adresi döndürür. Aşağıdaki *main* işlevini inceleyin:

```
#include <stdio.h>

int main()
{
    char *p[10];
    int k;

    for (k = 0; k < 10; ++k)
        p[k] = get_name();

    /*...*/
    for (k = 0; k < 10; ++k)
        printf("isim = (%s)\n", p[k]);

    return 0;
}
```

Yukarıdaki *main* işlevinde programcı *10* elemanlı bir gösterici dizisi tanımlanıyor. Gösterici dizisinin her bir elemanına *get_name* işlevinin çağrılarından elde edilen geri dönüş değerleri olan adresler atanıyor. Ancak *get_name* işlevi her defasında aynı statik yerel dizinin adresiyle geri dönüyor. Kullanıcı tarafından yapılan isim girişi her defasında bir önceki girişin üzerine yazılır. *main* işlevinin sonunda gösterici dizisinin tüm elemanları aynı yazıyı gösterir, değil mi?
Bazı standart C işlevleri de statik yerel dizilerin adreslerine geri döner. Örneğin *time.h* başlık dosyası içinde bildirimi yapılan *ctime* ve *asctime* işlevleri statik bir dizi içinde tutulan yazının başlangıç adresiyle geri döner.

Statik Yerel Dizilerin Kullanılmasıyla Sağlanan Verim Artışı

Diziler de *static* anahtar sözcüğüyle tanımlanabilirler.

```
void func()
{
    static char alphabet[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    /*…*/
}
```

Yukarıdaki örnekte, *func* işlevi içinde *alphabet* dizisi *static* anahtar sözcüğüyle tanımlandığı için, bir kez ilkdeğer verildikten sonra her çağrıldığında bu değerleri korur ve varlığını programın sonuna kadar sürdürür. Oysa *static* anahtar sözcüğü kullanılmasaydı, bu dizi *func* işlevine yapılan her çağrıda yeniden yaratılacak ve dizinin elemanlarına yapılan ilkdeğer atamaları her defasında yeniden yapılacaktı. Bu da işleve yapılan her çağrı için ek bir makine zamanının harcanmasına neden olacaktı.

İşlevlerin Daha Önceki Çağrılarıyla Mantıksal Bağ kurması

Bazı işlevler daha önceki çağrılarıyla mantıksal bir bağ oluşturarak işlerini gerçekleştirirler. Örneğin *LIMIT* pozitif bir tamsayı olmak üzere *0 – LIMIT* aralığında rastgele sayı üretecek bir işlev tanımlamak istediğimizi düşünelim. İşlev daha önce üretmiş olduğu bir sayıyı bir daha üretmesin. İşlev bir sayının daha önce üretilip üretilmemiş olduğu bilgisini statik bir yerel bayrak dizisinde saklayabilir. Aşağıdaki programı derleyerek çalıştırın:

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

#define    LIMIT      20
```

```
int urand()
{
    static char flags[LIMIT] = {0};
    int ret_val;

    if (!memchr(flags, 0, sizeof(flags)))
        return -1;

    while (flags[ret_val = rand() % LIMIT])
        ;
    flags[ret_val]++;

    return ret_val;
}

int main()
{
    int k;

    srand(time(0));
    for (k = 0; k < LIMIT; ++k)
       printf("%d ", urand());
    printf("\n");
    printf("%d\n", urand());

    return 0;
}
```

urand işlevi içinde *flags* isimli statik bir yerel dizinin elemanları *0* değeriyle başlatılıyor. Dizinin her bir elemanı ilgili değerin daha önce üretilip üretilmediğini gösteren bir bayrak olarak kullanılıyor. Örneğin işlev çağrıldığında *12* değerinin daha önce üretilip üretilmediği *flags* dizisinin *12* indisli elemanının değerinden anlaşılıyor. Bu elemanın değeri *0* ise *12* değeri üretilmemiş, *1* ise *12* değeri üretilmiş demektir.
İşlev önce standart *memchr* işleviyle dizinin içinde *0* değerinin olup olmadığına bakıyor. Dizinin hiçbir elemanının değeri *0* değilse, tüm değerler üretilmiş demektir. İşlev *-1* değerine geri dönerek bu durum hakkında bilgi iletiyor.
Bayrak dizisinde en az bir *0* değeri var ise, bir döngü içinde sürekli rastgele değer üretiliyor. Üretilen değer, ismi *ret_val* olan bir değişkene atanıyor. Bayrak dizisinin *ret_val* indisli elemanının değeri *1* oldukça döngü devam ediyor. Döngüden çıkılması için daha önce üretilmemiş bir değerin üretilmesi gerekiyor. Döngüden çıkıldıktan sonra *ret_val* değişkeninin değer, geri döndürülecek değerdir. Ancak bu değer geri döndürülmeden önce bayrak dizisinin *ret_val* indisli elemanının değeri *1* yapılıyor. Böylece bir sonraki çağrıda artık bu değeri yeniden üretilmesi engelleniyor.

Statik Yerel Değişkenler ile Global Değişkenlerin Karşılaştırılması

Statik yerel değişkenler ömür açısından global değişkenler ile aynı özellikleri gösterirler. Yani programın çalışmaya başlaması ile yaratılırlar ve programın çalışması bitince ömürleri sona erer. Ama bilinirlik alanı (*scope*) açısından farklılık gösterirler. Global değişkenler dosya bilinirlik alanına uyarken statik yerel değişkenler blok bilinirlik alanına uyar. Statik yerel değişkenlerin blok bilinirlik alanına sahip olması verilerin gizlenmesini *(data hiding)* kolaylaştırır ve kodun yeniden kullanılabilirliğini (*reusability*) artırır. Çünkü global değişkenlere bağlı olan işlevler bir projeden bir başka projeye kolayca taşınamazlar.

Modül Kavramı ve Nesnelerin Bağlantı Özelliği

Bir proje birbirinden bağımsız olarak derlenebilen birden fazla kaynak dosyadan oluşabilir. Projelerin bağımsız olarak derlenebilen her bir kaynak dosyasına *"modül*" denir.

Bir projeyi oluşturan farklı kaynak dosyalar birbirinden bağımsız olarak derlendikten sonra, hepsi birlikte bağlayıcı (*linker*) ile birleştirilerek tek bir çalıştırılabilir dosyayı oluştururlar.

Büyük Projelerin Modüllere Ayrılmasının Faydaları

Eğer bütün modüller tek bir kaynak kod içinde birleştirilirse en ufak bir değişiklikte tüm proje tekrar derlenmek zorundadır. Oysa modüllere ayrılmış projelerde, yalnız değişikliğin yapıldığı modülün derlenmesi yeterlidir. Çünkü diğer modüller zaten derlenmiştir ve onlar yalnızca bağlama aşamasında işlem görürler. Programları modüller halinde yazmanın bir diğer avantajı da grup çalışması yaparken ortaya çıkar. Bu durumda projelerin bağımsız parçaları (modülleri) ayrı kişiler tarafından hazırlanabilir.
Diğer taraftan modüller de işlevler gibi yeniden kullanılabilen yapılardır. Bir modül birden fazla projede kullanılabilir.

Bağlantı Kavramı

Projeyi oluşturan kaynak dosyaların birinde tanımlanan bir isme, projeyi oluşturan başka bir kaynak dosya içinde ulaşılabilir mi? Bu kaynak dosyalar en sonunda tek bir çalışır dosyayı oluşturacaklarına göre, bazı durumlarda bir modülde tanımlanan bir nesneye başka bir modülde ulaşmak istenmesi son derece doğaldır.
Bir modülde tanımlanmış bir değişkenin başka bir modülde kullanılıp kullanılmayacağını gösteren özelliğe o nesnenin bağlantı özelliği *(linkage)* denir. Bağlantı özelliği bilinirlik alanından farklıdır. Bilinirlik alanı derleyiciyi ilgilendiren yani derleme zamanına ilişkin bir kavramdır. Bir nesne bilinirlik alanı dışında kullanılırsa derleme zamanında hata oluşur. Ancak bağlantı, bağlayıcı programı ilgilendiren yani bağlama zamanına ilişkin bir kavramdır. Bir nesne, bağlantısının olmadığı bir modülde kullanılırsa bağlama zamanında hata oluşur.
C standartlarına göre nesneler bağlantı özellikleri açısından 3 ana gruba ayrılır.

1. Dış bağlantıya sahip nesneler *(external linkage)*
Eğer bir nesne tanımlandıktan sonra, bu nesneye hem kendi modülünün içinde her yerde hem de diğer modüllerde ulaşım mümkünse *"nesnenin dış bağlantısı vardır"* denir.

2. İç bağlantıya sahip nesneler *(internal linkage)*
Eğer bir nesne tanımlandıktan sonra bu nesneye kendi modülünün içinde her yerde ulaşılabiliyorsa, ancak projeyi oluşturan diğer modüllerde bu nesneye ulaşılamıyorsa, *"nesnenin iç bağlantısı vardır"* denir.

3. Bağlantısız Nesneler (*no linkage*)
Kendi modülünde ancak belirli bir blok içinde bilinen nesnelerdir.
Bir işlevin parametre değişkenleri ve bir işlev içinde tanımlanan yerel nesneler bağlantısız nesnelerdir. Bu nesnelere projeyi oluşturan diğer modüllerde ulaşmak mümkün değildir.
Global değişkenler normal olarak dış bağlantıya sahiptir. Yani bir global değişkene başka bir modül içinde ulaşılabilir. Peki projeye ait bir kaynak dosya içinde global bir değişken tanımlandığı zaman başka bir kaynak dosya içinde bu değişken nasıl kullanılabilir?

extern Belirleyicisi

extern belirleyicisi ile bir bildirim yapılır. Bir değişkenin *extern* belirleyicisi ile bildirilmesi derleyiciye nesnenin başka bir modülde tanımlandığını anlatır.

Bir proje *mod1.c* ve *mod2.c* isimli iki kaynak dosyadan oluşmuş olsun :

MOD1.C		MOD2.C

`int a;`
 	
`float func1()`
`{`
`  /***/`
`  a = 100;`
`  /***/`
`}`
		`int func2()`
		`{`
		`  ...`
		`  a = 300;`
		`/* HATA */`
		`}`

mod1.c dosyasında tanımlanmış olan *a* değişkeni global bir değişkendir ve dış bağlantıya sahip olduğu için normal olarak proje içindeki diğer kaynak dosyalarda da kullanılabilir. *mod2.c* dosyasında da bu değişkene ulaşılabilir. Fakat iki modülün ayrı ayrı derlendiği yukarıdaki örnekte problemli bir durum söz konusudur. Çünkü *mod2.c* dosyasının derlenmesi sırasında derleyici *a* değişkeninin *mod1.c* kaynak dosyası içinde global olarak tanımlandığını bilemez. *mod2.c* kaynak dosyasını derlerken *a* değişkeni hakkında bilgi bulamayan derleyici bu durumu hata olarak belirler. C dilinde bir değişken ancak önceden bildirilmişse kullanılabilir, değil mi? *extern* belirleyicisi derleyiciye, ilgili global değişkeninin kendi modülü içinde değil de bir başka modül içinde tanımlı olduğunu bildirme amacıyla kullanılır.
mod2.c modülünde *a* değişkeninin *extern* anahtar sözcüğü ile bildirilmesi durumunda söz konusu sorun ortadan kalkar.

```
mod2.c

extern int a;
/*extern bildirimiyle a değişkeninin başka bir modülde tanımlanmış olduğu
belirtiliyor*/

int func2()
{
    a = 300;
    /***/
}
```

extern bildirimini gören derleyici, değişkenin projeye ait başka bir modülde tanımlandığını varsayarak hata durumunu ortadan kaldırır. Ancak derleyici makine kodu üretirken *extern* anahtar sözcüğü ile bildirilmiş bir değişkenin bellekteki yerini saptayamayacağından, bu işlemi bütün modülleri gözden geçirecek olan bağlayıcı programa bırakır. Böylece değişkenin tanımlandığı modülü bulup *extern* olarak bildirilmiş olanlarla ilişkilendirme işlemi bağlayıcı program *(linker)* tarafından yapılır. Yani *extern* belirleyicisi ile programcı derleyiciye, derleyici ise bağlayıcıya bildirimde bulunur.
extern belirleyicisini gören derleyici bellekte bir yer ayırmaz. *extern* bildirimi bir tanımlama değildir, bildirimdir.
Aslında yalnız değişkenler için değil işlevler için de *extern* belirleyicisinin kullanılması söz konusudur. C derleyicileri kendi modülleri içinde tanımlanmadıkları halde çağrılan - standart C işlevleri gibi- işlevleri otomatik olarak *extern* kabul ederler. Bu nedenle işlev bildirimlerinde ayrıca *extern* belirleyicisini yazmaya gerek yoktur. Çünkü derleyici tarafından yazılmış varsayılır.

Örneğin yukarıda verilen örnekte *mod2c* modülünde bulunan *y1* işlevi içinde bulunan *x1* işlevi çağrılıyor olsun:

```
/****mod1.c  *****/
int a;

float x1()
{
     a = 100;
}

mod2.c

extern int a;
extern float x1(void);
int y1()
{
     float f;
     f = x1();
     a = 300;
    /***/
}
```

mod2.c modülünde *x1* işlevi için yazılmış olan prototip ifadesini inceleyin:

```
extern float x1(void);
```

Bu örnekte *x1* işlevi başka bir modülde tanımlı olduğu için bildiriminde *extern* anahtar sözcüğü kullanılıyor . Ancak *extern* belirteci eklenmeseydi derleyici zaten *extern* varsayacaktı.
Bir global değişken hiçbir modülde tanımlanmadan, bütün modüllerde extern olarak bildirilirse, tüm modüller hatasız bir şekilde derlenebilir. Hata bağlama aşamasında, bağlayıcının *extern* olarak bildirilen nesneyi hiçbir modülde bulamaması biçiminde ortaya çıkar.
extern belirleyicisinin tek bir modül söz konusu olduğunda "amaç dışı" bir kullanımı vardır. Aşağıdaki örnekte *main* işlevi içindeki global *x* değişkeni, tanımlamadan önce kullanıldığından hataya neden olur.

```
int main()
{
    /***/
    x = 100;
    /***/
}

int x;      /* x global bir değişken ama tanımından daha önce kullanılmış */
int func()
{
    x = 200;
    /***/
}
```

Yukarıdaki kod derlendiğinde, *main* işlevinin içindeki *x* değişkeninin bildiriminin bulunamadığını ifade eden bir hata iletisiyle karşılaşılır. Bu durumda, eğer bir global değişken tanımlamadan kullanılıyorsa, hata oluşmaması için daha önce *extern* bildiriminde bulunulmalıdır.

```
extern int x;

int main()
{
    /*…*/
    x = 100;
    /*…*/
}

int x;

int func()
{
    x = 200;
    /***/
}
```

extern bildirimini bu şekilde kullanmak yerine, global değişkeni programın en başında tanımlamak daha iyi bir tekniktir.

static Anahtar Sözcüğünün Global Değişkenler İle Kullanılması

static belirleyicisi global bir değişken ile birlikte kullanılırsa, bu değişkeni iç bağlantıya sahip yapar. Yani *static* anahtar sözcüğüyle tanımlanmış bir global değişken iç bağlantıya sahiptir. Böyle bir değişken *extern* bildirimiyle projeyi oluşturan diğer modüllerde kullanılmaz, yalnızca kendi modülünde kullanılabilir.
Global değişkenler için *static* tanımlamasının yalnızca bağlantı üzerinde etkili olduğuna, ömür üzerinde etkili olmadığına dikkat ediniz.
Bir de şu soru üzerinde düşünelim? Neden iç bağlantıya sahip bir global değişken kullanmak isteyelim? Bir global değişkeni kendi modülüyle sınırlamak ne gibi faydalar sağlayabilir?

İsim Kirlenmesi

Bir projeyi oluşturan dosyalar içinde dış bağlantıya sahip iki varlığın ismi aynı olamaz. Eğer iki global varlık aynı ismi paylaşıyorsa bağlama aşamasında hata oluşur.
Bu da özellikle büyük projeler söz konusu olduğunda tehlikeli bir durumdur. Zira büyük projelerin çoğunda hem çok sayıda programcı kod yazar, hem de başka firmalar tarafından yazılmış dosyalar da projede kullanılır. Bu durumda farklı kaynak dosyalar global varlıklara aynı isimler verilemez. Aynı isimlerin seçilmesi durumunda bağlama zamanında hata oluşur.

Başka Modüller Tarafından Değiştirilmesi İstenmeyen Global Değişkenler

Bir global değişken yalnızca bir modülü ilgilendiriyorsa, bu modülün hizmet verdiği modülleri ilgilendirmiyorsa, programın çalışma zamanında diğer modüller tarafından yanlışlıkla değiştirilebilir. Başka modüldeki global değişkenlerle isim çakışmasına yol açabilir yani isim kirlenmesine neden olabilir. Doğal olan böyle bir global değişkenin iç bağlantıya sahip olmasıdır. Böyle global değişkenler *static* anahtar sözcüğü ile tanımlanmalıdır.

Modüllerin Oluşturulması

Modül dışarıya bazı hizmetler verecek kodların oluşturduğu birimdir. C dilinde geleneksel olarak bir modüle ilişkin iki ayrı dosya oluşturulur. Bu dosyalardan biri modülün başlık dosyasıdır. Bir başlık dosyası bir modülün arayüzüdür *(interface).* Bir modülün arayüzü, o modülü kullanacak dosyaların derlenmesi sırasında, derleyicinin bilmesi gereken bilgileri sağlayan bildirimleri içerir. Modüller dışarıyla olan ilişkilerini arayüzleri ile kurar. Bir modüle ilişkin dışarıyı ilgilendiren bildirimler geleneksel olarak uzantısı *".h"* olarak

seçilmiş text dosyalarında toplanırlar. *"h"* uzantısı *"header"* sözcüğünün baş harfinden gelir.
Bir modülden faydalanan bir kaynak dosya o modülün başlık dosyasını *#include* önişlemci komutuyla eklerler. Böylece tüm bildirimleri derleyici görmüş olur.
O modüle ilişkin tanımlamalar ise *.c* uzantılı ayrı bir dosya içinde yer alır. Bu tanımlamalar global değişkenlerin tanımlamaları ve işlevlerin tanımlamalarıdır. Teknik olarak bu dosyaya *"kod dosyası" (implementation file)* denir. Projede çalışan bir programcı modülün ara yüzünü değiştirmeden, modülün kodunu yani kaynak dosyayı değiştirirse, projede bu modülü kullanan programcıların, kendi kodlarında bir değişiklik yapmaları gerekmez. Böylece önce modüller ilişkin arayüzler belirlenir ve bu arayüzlere bağlı olarak kaynak dosyalar yazılırsa, kaynak dosyada bir değişim olsa da, bu kaynak dosyadaki işlevleri çağıran modüllerde bir değişiklik yapılması gerekmez.

Static Anahtar Sözcüğünün İşlev Bildirimlerinde ya da Tanımlarında Kullanılması

İşlevler da global varlıklardır. Bir modüldeki bazı işlevler dışarıya hizmet vermek için tanımlanır. Bu işlevler modülün arayüzünde yani başlık dosyasında bildirilmelidir. Ancak modülde yer alan modülün kendi iç işleyişinde görev yapan işlevler, dışarıya doğrudan hizmet vermez. Bu işlevler, kodlama dosyasında *static* anahtar sözcüğü ile bildirilir ve tanımlanırl. Bu durumda *static* anahtar sözcüğü, işlevin geri dönüş değerinin türünden önce yazılır:

```
static int func(int);
```

Tür Niteleyicileri

C de *const* ve *volatile* olmak üzere iki tür niteleyicisi vardır.
[C99 standartlarıyla restrict anahtar sözcüğüyle yeni bir tür niteleyicisi eklenmiştir.]

const Belirleyicisi

const belirleyicisi, ilkdeğer atandıktan sonra nesnenin içeriğinin değiştirilemeyeceğini anlatır. *const* belirleyicisi yerel, global değişkenlerle ve işlevlerin parametre değişkenleriyle birlikte kullanılabilir. *const* anahtar sözcüğü ile tanımlanan bir nesneye atama işleci ile atama yapılması geçersizdir:

```
const double PI = 3.14159265   /* Geçerli. İlkdeğer verme. */

int main()
{
    const int i = 10;

    i = 100;               /* Geçersiz */

    return 0;
}
```

Yerel bir değişken *const* belirleyicisi ile tanımlanacaksa ilkdeğer verilmelidir, ilkdeğer verilmezse *const* belirleyicisi kullanmanın bir anlamı kalmaz. Aşağıdaki örneği inceleyin:

```
void func ()
{
    const int a;    /* anlamsız */
    /*…*/
}
```

Bu örnekte *a* yerel bir değişken olduğundan bir çöp değere sahiptir. Değişkenin değeri bir daha değiştirilemeyeceğine göre böyle bir tanımlamanın da bir anlamı olamaz.

const belirleyicisinin kullanım amacı nedir? Kullanıldığı yere göre *const* belirleyicisi aşağıdaki faydaları sağlayabilir:

1. Programın okunabilirliğini artırır. Bu yönüyle kodu okuyana yardımcı olur. Programı okuyanlar *const* anahtar sözcüğünü gördüklerinde nesnenin değerinin değiştirilmeyeceği bilgisini alırlar.

2. Yanlışlıkla nesnenin değerinin değiştirilmesi engellenir. Bu yönüyle kodu yazana yardımcı olur. *const* bir nesnenin değeri atama yoluyla değiştirlmesi durumunda derleme zamanında hata oluşur. Böylece değeri değiştirilmemesi gereken bir nesnenin değerinin yanlışlıkla atama yoluyla değiştirilmesi de engellenmiş olur.

3. *const* anahtar sözcüğü ile tanımlanan nesneler, derleyici tarafından salt okunan bellekte saklanabilirler. Derleyici *const* nesnenin değerinin değiştirilmeyeceğini bilmediğinden, bu nesnelerin kullanıldığı ifadeler için ek eniyileme(optimizasyon) işlemleri uygulayabilir. *const* anahtar sözcüğü bu yönüyle derleyiciye yardımcı olur.

const Nesnelerle Simgesel Değişmezlerin Karşılaştırılması

const nesneler değeri değişmeyecek değişmezler gibi kullanıldıklarına göre, bunların yerine çoğu zaman *#define* önişlemci komutuyla oluşturulmuş simgesel değişmezler de kullanılabilir.
Ancak *const* nesneler ile simgesel değişmezler arasında bazı farklılıklar vardır. İkisi arasında tercih yapılması durumunda bu farklılıklar rol oynar:
1. Simgesel değişmezler nesne değildir. Bu yüzden programın çalışma zamanında bellekte bir yer kaplamazlar. Nesne olmadıkları için bilinirlik alanları, ömürleri ve bağlantı özelliklerinden söz edilemez. Derleme aşamasına gelindiğinde derleyici simgesel değişmezlerin yerine, önişlemci tarafından yerleştirilen atomları görür.

2. Derleyiciler *const* nesneler için daha iyi *"debugging"* desteği verirler.

3. Derleyiciler derleme zamanında bir hata oluşması durumunda hatanın yerini işaretleme konusunda *const* nesneler için daha iyi destek verirler.

4. Bir dizi *const* anahtar sözcüğü ile tanımlanabilir. Bu durumda dizinin köşeli ayraç işleci ile ulaşılan herhangi bir elemanına atama yapılamaz. Böyle bir durumu önişlemci *#define* komutuyla kolaylıkla gerçekleştirmek mümkün değildir:

```
void foo
{
    const char alphabet[] = "abcdefghijklmnopqrstuvwxyz";
    alphabet[2] = 'X';     /* GEÇERSİZ*/
}
```

5. C'de *const* nesneler dış bağlantıya sahiptir. Bir *const* nesne, *extern* bildirimi yapılarak başka modüllerde kullanılabilir.
6. C dilinde *const* nesneler değişmez ifadesi olarak kullanılamazlar. Aşağıdaki örneği inceleyin:

```
#define    SIZE       100

int main()
{
    const int size = 100;
    int legal_array[SIZE];
    int illegal_array[size];   /* GEÇERSİZ size değişmez ifadesi değil */
    /***/
}
```

const Anahtar Sözcüğünün Gösterici Tanımlamalarında Kullanılması:

const anahtar sözcüğü sıklıkla gösterici nesnelerinin tanımlamalarında kullanılır. *const* anahtar sözcüğünün göstericilerle birlikte üç ayrı kullanılma biçimi vardır. Kullanılan her biçimde tanımlanan göstericinin özelliği değişir.

1. *const* anahtar sözcüğünün '*' atomundan önce kullanılması:
Böyle göstericilere gösterdiği yer *const* olan göstericiler *(pointer to const objects)* denir. Böyle bir göstericinin gösterdiği nesne, göstericinin içeriği alınarak elde edilen nesneye atama yapılması yoluyla değiştirilemez. Ancak göstericinin kendisine atama yapılabilir. Yani göstericinin bir başka nesneyi göstermesi sağlanabilir. Aşağıdaki örneği inceleyin:

```
int main()
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    double d1 = 2.5, d2 = 3.5;
    char str[] = "Necati";
    int x = 20;
    const int *iptr = a;
    double const *dptr = &d1;
    const char *cptr = str;
    /*
    *iptr = 100;    Geçersiz!
    *dptr = 1.23;   Geçersiz!
    cptr[1] = 'x';  Geçersiz!
    */
    iptr = &x;
    dptr = &d2;
    cptr = "Selami";

    return 0;
}
```

Yukarıdaki *main* işlevinde tanımlanmış, gösterdiği yer *const* olan göstericiler için yapılan

```
*iptr = 100;
*dptr = 1.23;
cptr[1] = 'x';
```

atamaları derleme zamanında hata oluşumuna neden olur. Ancak göstericilerin kendilerine yapılan

```
iptr = &x;
dptr = &d2;
cptr = "Selami";
```

atamalarının tamamen kurallara uygun olduğunu görüyorsunuz.

const anahtar sözcüğünün göstericilerle birlikte kullanılmasında en sık görülen biçim budur ve özellikle işlevlerin parametre değişkenleri olan göstericilerle bu biçim kullanılır:

```
void func(const char *str)
{
    /***/
}
```

Yukarıda *func* isimli işlevinin parametre değişkeni olan *str* göstericisi değerini işlevin çağrısıyla alır. İşlev çağrı ifadesindeki argüman işlevin çağrılmasıyla s göstericisine

kopyalanır. Ancak işlev içinde **s* nesnesine ya da *s[x]* nesnesine bir atama yapılamaz. Yani *s* göstericisinin gösterdiği yerdeki nesne (işleve adresi gönderilen nesne) değiştirilemez. Yukarıdaki örnekte *const* anahtar sözcüğünün kullanılması herşeyden önce okunabilirlik ile ilgilidir. Yukarıdaki kodu okuyan bir C programcısı *func* işlevinin dışardan adresi alınan nesnenin yalnızca değerinden faydalanılacağını, yani bu nesnenin değerini değiştirmeyeceğini anlar. Geleneksel olarak, *const* anahtar sözcüğünün, parametre değişkeni olan göstericilerin tanımlanmasında kullanılmaması okunabilirlik açısından bir ileti olarak kabul edilir:

```
void func(char *s)
{
    /***/
}
```

Yukarıdaki kodu okuyan bir C programcısı *func* işlevinin (aslında sentaks açısından bir zorunluluk bulunmasa da) kendisine adresi gönderilen nesneyi değiştireceğini anlar. Kodu okuyan kişi şöyle düşünür: Eğer *func* işlevi adresi gönderilen nesneyi değiştirmeyecek olsaydı, *s* göstericisi *const* anahtar sözcüğü ile tanımlanırdı.

2. *const* anahtar sözcüğünün `'*'` atomundan sonra kullanılması:
Bu şekilde tanımlanmış göstericilere kendisi *const* gösterici (*const pointer)* denir. Bu göstericilere ilkdeğer olarak verilen adres atama yoluyla değiştirilemez. Yani gösterici ömrü boyunca aynı nesneyi gösterir. Ancak göstericinin gösterdiği nesneyi, göstericinin içeriğini alarak değiştirmek mümkündür. *const* anahtar sözcüğünün bu kullanım biçiminde gösterici tanımlanırken ilkdeğer verilmelidir, yoksa *const* anahtar sözcüğünün kullanılmasının bir anlamı kalmaz. Aşağıdaki örneği inceleyin:

```
int main()
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    double d1 = 2.5, d2 = 3.5;
    char str[] = "Necati";
    int x = 20;

    int *const iptr = a;
    double  *const dptr = &d1;
    char *const cptr = str;

    *iptr = 100;
    *dptr = 1.23;
    cptr[1] = 'x';

    /*
    iptr = &x;          Geçersiz!
    dptr = &d2;         Geçersiz!
    cptr = "Selami";    Geçersiz!
    */

    return 0;
}
```

3. *const* anahtar sözcüğünün hem `'*'` atomundan önce hem de `'*'` atomundan sonra kullanılması:
Bu durumda hem kendisi *const* hem de gösterdiği yer *const* olan bir gösterici tanımlanmış olur. Atama işleciyle ne göstericiye ne de göstericinin içeriği alınarak elde edilen nesneye atama yapılabilir. *const* sözcüğünün bu biçimde kullanılması yine daha çok parametre değişkenlerinin tanımlanmasında görülür ise de bu uygulamalarda seyrek olan bir durumdur. Aaşağıdaki örneği inceleyin:

```
int main()
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    double d1 = 2.5, d2 = 3.5;
    char str[] = "Necati";
    int x = 20;

    const int *const iptr = a;
    const double  *const dptr = &d1;
    const char *const cptr = str;
    /*
    *iptr = 100;       Geçersiz!
    *dptr = 1.23;      Geçersiz!
    cptr[1] = 'x';     Geçersiz!

    iptr = &x;         Geçersiz!
    dptr = &d2;        Geçersiz!
    cptr = "Selami";   Geçersiz!

    return 0;
}
```

Göstericiyi gösteren bir göstericinin *const* anahtar sözcüğü ile bildirilmesinde *const* anahtar sözcüğünün kullanılması farklı olanaklar yaratır:

```
const char **ptr;
```
ptr nin gösterdiği göstericinin gösterdiği nesneye atama yapılamaz. *const* olan ***ptr* nesnesidir. *ptr* nesnesine ve **ptr* nesnelerine atama yapılabilir. (*pointer to pointer to const object*)

```
char *const *ptr;
```
**ptr* nin gösterdiği göstericiye atama yapılamaz. *const* olan **ptr* nesnesidir. *ptr* nesnesine ve ***ptr* nesnelerine atama yapılabilir.

```
char ** const ptr;
```
ptr ye atama yapılamaz. *const* olan *ptr* nesnesidir. ***ptr* nesnesine ve **ptr* nesnelerine atama yapılabilir.

```
const char *const *ptr;
```
yalnızca *ptr* nesnesine atama yapılabilir.

```
char *const * const ptr;
```
yalnızca ***ptr* nesnesine atama yapılabilir.

```
const char ** const ptr;
```
yalnızca **ptr* nesnesine atama yapılabilir.

Bir işlevin Geri Dönüş Değerinin const Adres Olması

Adrese geri dönen bir işlevin geri dönüş değeri *const* bir adres olabilir:

```
const char *get_name(void);
```

Yukarıda bildirimi yapılan *get_name* işlevinin geri dönüş değeri *const* bir adrestir. Peki bu ne anlama gelir? Bir işlevin geri dönüş değeri türünün aslında, geri dönüş değerini içinde tutacak geçici nesnenin türü olduğunu biliyorsunuz. *get_name* işlevinin geri dönüş değerini tutacak geçici nesnenin türü *const char* * türüdür. Yani geçici nesne gösterdiği yer *const* olan bir göstericidir. İşlev çağrısı, işlevin geri dönüş değerini, yani geçici nesnenin değerini gösterdiğine göre, böyle bir işleve yapılan çağrı içerik işlecine ya da

köşeli ayraç işlecine terim olduğunda, ulaşılan nesneye atama yapılamaz. Aşağıdaki örneği inceleyin:

```
char *get_name();
const char *get_fname();

int main()
{
    *get_name() = 'a';
    get_name()[2] = 'x';
    *get_fname() = 'a';    /* Geçersiz   */
    get_fname()[2] = 'x';  /* Geçersiz   */

    return 0;
}
```

Yukarıda *main* işlevi içinde çağrılan *get_name()* işlevinin geri döndürdüğü adresteki nesneye *'a'* değeri atanıyor. Yine *get_name()[2]* ifadesi ile işlevin döndürdüğü adresteki nesneden iki sonraki nesneye ulaşılarak bu nesneye de *'x'* değeri atanıyor. Atamalar geçerlidir. Ancak geri dönüş değeri *const char* * türden olan *get_fname* işlevi için benzer atamalar geçersizdir, derleme zamanında hata oluşturur.

volatile Belirleyicisi

Derleyiciler eniyileme amacıyla nesneleri geçici olarak yazmaçlarda tutabilir. Yazmaçlardaki bu çeşit geçici barınmalar *register* belirleyicisi kullanılmasa da derleyiciler tarafından yapılabilir. Örneğin:

```
int kare (int a)
{
    int b;

    b = a * a;
    return b;
}
```

Yukarıdaki işlevde *b* geçici bir değişkendir, dolayısıyla derleyici *b* 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 hızlandırır. Aşağıdaki kodu inceleyin:

```
int a;
int b;

/***/
a = b;
if (a == b) {
    /***/
}
```

Bu örnekte doğal olarak ilk adımda *b* değişkeni *a* değişkenine aktarılmak üzere yazmaçlardan birine çekilir. Ancak derleyici *if* ayracı içindeki ifadede *a == b* karşılaştırmasını yapmak için bellekteki *b* yerine yazmaçtaki *b*'yi kullanabilir. Verdiğimiz iki örnekte de derleyici bir takım eniyileme işlemleriyle programı işlevi değişmeyecek biçimde daha hızlı çalışır hale getirmek istemiştir. Ancak kimi uygulamalarda derleyicinin bu biçimde davranması hatalara neden olabilir. İkinci örnekte :

```
a = b;
/* Bir 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 eniyileme amacıyla geçici olarak yazmaçlarda tutulması arzu edilmeyen sonuçların oluşmasına yol açabilir. *volatile* " Değişkenleri eniyileme 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 olabilir. Bu belirleyicinin bilinçsizce kullanılmasının performansı kötü yönde etkileyebileceğini unutmayınız.

YAPILAR

Yapı nedir

Yapılar (*structures*) programcının birden fazla nesne yaratmasına izin veren bir araçtır. Yapıların kullanılmasıyla bellekte birbirini izleyecek şekilde yer alan birden fazla nesne yaratılabilir. Bellekte bitişik olarak yer alan nesnelerin dizi tanımlamalarıyla da yaratılabileceğini biliyorsunuz. Ancak yapıların dizilerden bazı farklılıkları vardır: Diziler aynı türden nesneleri içinde tutabilirken, yapılar farklı türlerden nesneleri tutabilir. Yapıların kullanılmasının ana nedeni budur. Çoğu zaman, türleri farklı bir takım nesneler, mantıksal olarak bir bütün oluşturabilir. İsim, yaş, cinsiyet, departman, ücret, öğrenim durumu gibi bilgileri farklı türden nesneler içinde saklanabilir. Bunların tamamı bir işyerinde çalışan bir kişiye ait bilgiler olabilir. Aralarında mantıksal ilişki olan farklı türden veriler yapılar içinde saklanabilir.

int türden *10* elemanlı bir dizi tanımlandığını düşünelim:

```
int a[10];
```

Kullanılan sistemde *int* türünün *4 byte* yer kapladığını düşünürsek bu dizi için bellekte *40 byte*'lık bir blok ayrılır değil mi? Programcı bu dizinin elemanlarına köşeli ayraç işleci ile ulaşarak, eleman olan nesneler üzerinde doğrudan işlemler yapabilir. Ancak dizinin tamamı bir nesne olarak kullanılamaz. Yani C dili bir dizinin tamamını bir nesne olarak görmez. Dizi isimlerinin, C'de dizinin ilk elemanı olan nesnenenin adreslerine dönüştürüldüklerini hatırlayalım. Yapılarda durum biraz daha farklıdır. Yapı kullanılmasıyla da birden fazla nesne için ardışıl bir blok ayrılır. Ancak bu kez bloğun tamamı da bir nesne olarak kullanılabilir. Yapının elemanı olan nesneler üzerinde işlemler yapabileceğimiz gibi, bu nesnelerin oluşturduğu bloğun tamamı üzerinde de doğrudan bazı işlemler yapabiliriz.

Programcının Tanımladığı Bir Tür Olarak Yapı

Yapıların kullanılmasıyla programcı yeni bir tür yaratabilir. C'nin var olan doğal veri türlerinin yanında, mantıksal bir anlamı soyutlayan yeni türler yaratılabilir. Örneğin değeri bir tarih bilgisi olan bir tür ya da değeri bir kompleks sayı olan bir tür oluşturulabilir. Böylece programlama ile bir çözüm oluşturmak istediğimiz problem düzlemi daha iyi modellenebilir. Yapıların iyi bir şekilde öğrenilmesi, ileride *nesne bazlı(object based)* ve *nesneye yönelimli (object oriented) programlama* tekniklerinin iyi bir biçimde anlaşılabilmesine yardımcı olur.

Yapı Bildirimi

Yapı ile programcının yeni bir tür yaratmasına olanak verilir. Yapıların kullanılabilmesi için yapılması gereken ilk işlem bu yeni türü derleyiciye tanıtmaktır. Tanıtma işlemi yapı bildirimi ile olur. Yapı bildirimini gören derleyici, bu yeni tür hakkında bilgi edinmiş olur. Bu bildirimle derleyiciye aşağıdaki bilgiler verilir:

- türün ismi
- türün bellekte ne kadar yer kapladığı
- elemanların isimleri

Derleyici bir yapı bildirimini gördükten sonra, bu türden bir nesne tanımlandığında nesne için bellekte ne kadar yer ayıracağını öğrendiği gibi, bu nesnenin programcı tarafından kullanımına ilişkin ifadelere ilişkin derleme zamanında bazı sınamalar yapabilir.
Yapı bildiriminin belirli bir sentaksı vardır.

Yapı Bildiriminin Genel Şekli

```
<struct > [yapı etiketi] {
    <tür> <eleman ismi>;
    <tür> <eleman ismi>;
    <tür> <eleman ismi>;
    ...
};
```

struct bir anahtar sözcüktür. Bildirimde mutlaka yer alması gerekir. *struct* anahtar sözcüğünü bir *yapı etiketi (structure tag)* izleyebilir. *Yapı etiketi* C dilinin isimlendirme kurallarına uygun olarak seçilmiş bir isimdir. Yapı etiketinden sonra bir blok yer alır. Bu blok bildirime açık bir bölge belirler. Bu blok içinde yapı elemanlarının bildirimleri yapılır. Örnek verelim:

```
struct Date {
    int day, month, year;
};

struct Point {
    int x, y;
};

struct Complex {
    double real;
    double imag;
};
```

Yapı etiketleri (*structure tags*) isimlendirme kurallarına uymalıdır. Ancak programcıların çoğu yapı etiketlerinin ilk harflerini büyük, diğer harflerini küçük seçerler.

Yapı bildiriminin yapılması bellekte derleyici tarafından bir yer ayrılmasına neden olmaz. Yani bir tanımlama (*definition*) söz konusu değildir. Bu bildirimle programcının yarattığı yeni bir veri türü hakkında derleyiciye bilgi verilir. Derleyici bu bilgiyi, bu türden yaratılacak nesneler için şablon olarak kullanır.

Yukarıdaki bildirimlerle yaratılmış olan türlerin ismi sırasıyla*, struct Sample*, *struct Date*, *struct Point* ve *struct Complex*'dir. Yani tür ismi *struct* anahtar sözcüğü ve yapı etiketinin birleştirilmesiyle elde edilir. Yapı etiketi tek başına bir tür bilgisi belirtmez. Ancak *struct* anahtar sözcüğü olmadan da bir tür ismi yaratılmasını sağlayan araçları da ileride göreceğimizi şimdiden belirtelim.

Yapı Bildirimlerinin Yeri

Tüm bildirimler gibi yapı bildirimleri de yerel ya da global düzeyde yapılabilir. Tüm bildirimler gibi yapı bildirimlerinin de bilinirlik alanı vardır. Bildirim bir blok içinde yapılırsa, bildirilen yapı yalnızca bildirimin yapıldığı blok içinde bilinir.
Yapı bildirimleri ancak çok özel durumlarda bir blok içinde yapılır. Kaynak dosya içindeki tüm işlevlerin yeni yaratılan türü kullanabilmeleri için, bildirim hemen her zaman global düzeyde yapılır. Tanımlanan işlevleri çağıracak diğer kaynak dosyalar da, çoğu zaman aynı yapıyı kullanmak zorunda olduğundan, yapı bildirimleri genellkle modüllerin başlık dosyalarına yerleştirilir. Böylece ilgili başlık dosyasını ekleyen bir kaynak dosya içinde de aynı bildirim yapılmış olur.

Yapı Türünden Değişken Tanımlaması

Doğal türlerden nesneler nasıl tanımlanıyorsa yapı türünden bir değişkenin tanımlanması da aynı sözdizimle olur. Yani önce tür belirten sözcükler daha sonra da yapı değişkeninin ismi yazılır:

```
struct Date bdate;
struct Point p1, p1;
```

Yukarıdaki tanımlamaların ilgili yapı bildirimlerinin görülür olduğu bir noktada yapıldığı kabul ediliyor. Yapı bildiriminin görülmediği bir kaynak kod noktasında yapı değişkeni tanımlanması geçersizdir.

```
struct Date mydate;
```

tanımlamasıyla ismi *mydate* olan türü *struct Date* olan bir değişken tanımlanıyor.

```
struct Point p1, p2, p3;
```

tanımlamasıyla türleri *struct Point* olan *p1*, *p2* ve *p3* isminde üç ayrı değişken tanımlanıyor.

Nesnelerin temel özellikleri nesnelerin türüne bağlı değildir. Şüphesiz yapı nesnelerinin de bilinirlik alanlarından, ömürlerinden, bağlantı özelliklerinden söz edilebilir. Bu özellikler doğal veri türleri için ne anlam taşıyorlarsa yapı nesneleri için de aynı anlamı taşır.

Global isim alanında tanımlanan bir yapı nesnesi

i) Dosya bilinirlik alanına aittir. Nesnenin tanımından sonra yer alan tüm işlevler bu yapı nesnesine ulaşabilirler.

ii) Statik ömürlüdür. Programın çalışma zamanının başlangıcından, programın sonlanmasına kadar bellekte yer kaplar.

iii) Statik ömürlü olduğu için ilkdeğer ataması yapılmadığı zaman, bellekte kapladığı bütün *byte* lar *0* bitleriyle doldurulur. Pratik olarak bunun anlamı yapının tüm elemanlarının değerinin *0* olmasıdır.

Yerel alanda tanımlanan bir yapı nesnesi

i) Blok bilinirlik alanına aittir. Yalnızca tanımlamasının yapıldığı blok içinde bu nesneye ulaşılabilir.
ii) Otomatik ömürlüdür. Programın akışı, nesnenin tanımlandığı bloğa girdiğinde nesne yaratılır. Programın akışı nesnenin tanımladığı bloğu terk ettiğinde nesne bellekten boşaltılır.

iii) Otomatik ömürlü olduğu için ilkdeğer ataması yapılmadığı zaman, bellekte kapladığı alanda çöp değerleri vardır. Pratik olarak bunun anlamı yapının tüm elemanlarının çöp değerleriyle başlatılmasıdır.

Yapı nesnelerinin bildirim ve tanımlamalarında belirleyiciler kullanılabilir.

Örneğin yerel ya da global bir yapı nesnesi *static* ya da *const* belirleyicisi ile tanımlanabilir. Bu belirleyicilerin bildirim ya da tanımlamalara kattığı anlamlar yapı nesneleri için bir farklılık taşımazlar.

Yapı değişkenleri bileşik nesnelerdir. Yani parçalardan, alt nesnelerden oluşurlar. Zaten yapı bildirimi ile yapılan bir işlem de bu parçaların isimleri ve türleri hakkında derleyiciye bilgi verilmesidir. Bir yapı değişkeni (nesnesi) için, yapı bildiriminde belirtilen elemanların toplam uzunluğu kadar (byte olarak) yer ayrılır. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

struct Sample {
     int i;
     long l;
     double d;
};

int main()
{
    struct Sample sam;
    printf("%d\n", sizeof(sam));

    return 0;
}
```

Yukarıdaki program *UNIX* altında çalıştığında ekrana *16* değerini yazdırır. Çünkü *struct Sample* türünden bir nesne olan *sam* nesnesinin bellekte kapladığı yer üç elemanının kapladığı uzunluğun toplamıdır. Bu aşağıdaki şekilde de ifade edilebilir:

```
sizeof(sam) == sizeof(int) + sizeof(long) + sizeof(double)
```

Hizalama (*alignment*) konusunda yeniden bu noktaya değininilecek.

Yapı Bildirimi İle Değişken Tanımlamasının Birlikte Yapılması

Bir yapı bildirimi sonlandırıcı atom ile sonlandırılmadan, yapı bildiriminin kapanan küme ayracından hemen sonra bir değişken listesi yazılırsa yapı bildirimi ile değişken tanımlaması bir arada yapılmış olur.

```
struct Date {
    int day, month, year;
}bdate, edate;
```

Yukarıdaki deyim ile *struct Date* türü bildirilirken bu yapı türünden *bdate* ve *edate* değişkenleri de tanımlanıyor. Bu durumda yapı değişkenlerinin bilinirlik alanları yapı bildiriminin yerleştirildiği yere bağlı olarak belirlenir. Yani söz konusu yapı değişkenleri yerel ya da global olabilir.
Yapı nesnelerinin hemen yapı bildiriminden sonra tanımlanması durumunda yapı ismi kullanma zorunluluğu da bulunmaz. Örneğin yukarıdaki bildirim aşağıdaki şekilde de yapılabilir.

```
struct  {
    int day, month, year;
} bdate, edate;
```

Burada *bdate* ve *edate* yukarıda bildirilen ama isimlendirilmeyen yapı türünden değişkenlerdir. Bu yöntemin dezavantajı artık aynı türden başka bir yapı değişkeninin tanımlanmasının mümkün olmayışıdır. Örneğin yukarıdaki kod parçasından sonra aynı yapı türünden bir yapı nesnesi daha tanımlamak istediğimizi düşünelim. Bu tanımlama

```
struct {
    int day, month, year;
}cdate;
```

biçiminde yapılsa bile artık derleyici, bildirimleri eşdeğer olan bu iki yapı türünü ayrı yapı türleri olarak ele alır. Dolayısıyla

```
cdate = bdate;
```

gibi bir atama yapı türlerinin farklı olması nedeniyle geçerli değildir.
Böyle bir tanımlamanın başka bir dezavantajı da, bu yapı türüyle ilgili bir işlev yazılamayışıdır. Çünkü yapı türünün bir ismi yoktur. Bir işlevin parametre değişkeni tür bilgisi olmadan tanımlanamaz.

Yapılar ve İsim Alanları

Yapı isimleri ve yapı elemanlarının isimleri ayrı isim alanlarında değerlendirilir:
Aşağıdaki kod geçerlidir:

```
void func()
{
    struct x {int x;}x;
}
```

Yapı Elemanlarına Erişim

Yapıların dizilerden önemli bir farkı da elemanlara erişim konusundadır. Dizi elemanlarına erişim, dizi ismi ve köşeli ayraç işleci *[ ]* (*index operator - subscript operator)* yoluyla yapılırken, yapı elemanlarına erişim için *nokta işleci* kullanılır.

Nokta işlecine (*dot operator*) ilişkin atom '.' dır. Nokta işleci, iki terimli araek konumunda (*binary infix*) bir işleçtir. Bu işlecin sol terimi bir yapı türünden nesne olmalıdır. İşlecin sağ terimi ise nesnenin ait olduğu ilgili yapı türününün elemanlarından biri olmak zorundadır. Nokta işleci işleç öncelik tablosunun en yüksek öncelikli düzeyinde bulunur.

Bir yapı nesnesi tanımlanarak, bu yapı nesnesinin elemanlarına nokta işleci ile erişildiğinde artık bu elemanların her biri ayrı bir nesne olarak ele alınır. Bu nesnelerin, yapı dışında tanımlanan nesnelerden herhangi bir farkı yoktur. Aşağıdaki kodu derleyerek çalıştırın:

```
#include <stdio.h>

struct Sample{
     int i;
     long l;
     double d;
};

int main()
{
    struct Sample sam;

    sam.i = 10;
    sam.l = 200000L;
    sam.d = 1.25;
    printf("sam.i = %d\nsam.l = %ld\nsam.d = %lf\n", sam.i, sam.l, sam.d);

    return 0;
}
```

Yukarıdaki örnekte görüldüğü gibi *sam.i, sam.l* ve *sam.d* yapı elemanları ayrı birer nesne özelliği gösterir. Bu elemanlara ayrı ayrı ulaşabilir, ayrı ayrı atamalar yapılabilir. Bu nesneler ++ ve -- işleçlerinin ya da adres işlecinin terimi olabilir.

Yapı Elemanlarının Bellekteki Yerleşimi

Yapı elemanları belleğe, bildirimde ilk yazılan eleman küçük sayısal adreste olacak biçimde, bitişik olarak yerleştirilir. Aşağıdaki programı derleyerek çalıştırın:

```
struct Sample{
    int i;
    long l;
    double d;
};
```

```
#include <stdio.h>

int main()
{
    struct Sample sam;

    printf("sam.i'nin adresi = %p\n", &sam.i);
    printf("sam.l'nin adresi = %p\n", &sam.l);
    printf("sam.d'nin adresi = %p\n", &sam.d);

    return 0;
}
```

Yapı Nesneleri Üzerinde Yapılabilecek İşlemler

Yapı nesnelerinin elemanları değil de kendileri söz konusu olduğunda, yani bir yapı nesnesi bir bütün olarak ele alındığında ancak dört işlecin terimi olabilir: *nokta* işleci, *sizeof* işleci, adres işleci, atama işleci. Bir yapı nesnesinin bunların dışında bir işlecin terimi olması geçersizdir.

sizeof İşleci ve Yapılar

Bir yapı nesnesi *sizeof* işlecinin terimi olabilir. Bu durumda *sizeof* işlecinin ürettiği değer, nesnenin ait olduğu türün bellekte kapladığı *byte* sayısıdır. Tabi *sizeof* işlecinin teriminin bir tür ismi de olabileceğini biliyorsunuz. Aşağıdaki örneği inceleyin:

```
struct Point {
    int x, y;
};

int main()
{
    struct Point p;

    printf("sizeof(struct Point) = %u\n", sizeof(struct Point));
    printf("sizeof(p) = %u\n ", sizeof(p));

    return 0;
}
```

Yukarıdaki *main* işlevini içinde yapılan *printf* çağrısında *sizeof* işlecinin terimi bir tür ismi (*struct Point*) iken, ikinci *printf* çağrısında ise *sizeof* işlecinin terimi *struct Point* türünden bir nesne olan *p* dir.

Adres İşleci ve Yapı Nesneleri

Bir yapı nesnesi adres işlecinin terimi olabilir. Bu durumda adres işlecinin ürettiği değer ilgili yapı türünden bir adres bilgisidir:
struct Date isimli bir yapının bildiriminin yapılmasından sonra aşağıdaki gibi bir nesne tanımlandığını düşünelim:

```
struct Date {
    int d, m, y;
};

struct Date mydate = {4, 3, 1964};

&mydate
```

ifadesi geçerli bir ifadedir. Bu ifade *(struct Date *)* türündendir.
Peki *&mydate* ile *&mydate.day* aynı adresler midir? Bu adreslerin sayısal bileşenleri aynı olmakla birlikte tür bileşenleri farklıdır. İki ifade, sayısal bileşenleri aynı olan farklı türden iki adres bilgisidir.

&mydate ifadesinin türü *(struct Date *)* iken *&mydate.day* ifadesinin türü *(int *)* türüdür.

Atama İşleci ve Yapı Nesneleri

Bir yapı nesnesi atama işlecinin sol ya da sağ terimi olabilir.
Bir yapı nesnesine atama işleci ile ancak aynı türden başka bir yapı nesnesi atanabilir. Atama işlecinin sağındaki ve solundaki türlerin farklı yapı türlerinden olması geçersizdir. Aynı türden iki yapı nesnesinin birbirine atanmasında yapı elemanları karşılıklı olarak birbirlerine atanır. Yani bir blok kopyalanması söz konusudur. Atama işlemi için kesinlikle iki yapı değişkeninin de aynı türden olması gerekir. İki yapı değişkeni de aynı türden değilse bu durum geçersizdir. İki yapının aynı türden olmaları için aynı yapı ismi ile tanımlanmış olmaları gerekir. Aşağıdaki iki yapı, elemanlarının türleri ve isimleri aynı olduğu halde farklı iki tür belirtirler ve bu türlerden nesneler birbirlerine atanamaz.

```
struct POINT_1 {
    int x, y;
};

struct  POINT_2{
    int x, y;
};

void func()
{
    struct POINT_1 a;
    struct POINT_2 b;

    b.x = 10;
    b.y = 20;
    a = b; /* Geçersiz */
}
```

İki yapı nesnesinin birbirine atanması, nesnelerin tüm elemanlarının karşılıklı olarak birbirlerine atanması sonucunu doğurur. Yani bir blok kopyalaması söz konusudur. Özellikle büyük yapılar söz konusu olduğunda atama işleci ile yapı nesnelerini birbirine atanmasının maliyetinin yüksek olduğu unutulmamalıdır. İlgili yapı türünün *sizeof* değeri büyüklüğünde bir bellek bloğu, adeta *memcpy* benzeri bir işlevle kopyalanır.
Aşağıdaki programı derleyerek çalıştırın:

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

struct Data{
     int i;
     long l;
     double d;
};
```

```
int main()
{
    struct Data d1, d2, d3, d4;

    d1.i = 10;
    d1.l = 200000L;
    d1.d = 1.25;

    d2 = d1;

    d3.i = d2.i;
    d3.l = d2.l;
    d3.d = d2.d;

    memcpy(&d4, &d3, sizeof(struct Data));

    printf("d1.i = %d\nd1.l = %ld\nd1.d = %lf\n\n", d1.i, d1.l, d1.d);
    printf("d2.i = %d\nd2.l = %ld\nd2.d = %lf\n\n", d2.i, d2.l, d2.d);
    printf("d3.i = %d\nd3.l = %ld\nd3.d = %lf\n\n", d3.i, d3.l, d3.d);
    printf("d4.i = %d\nd4.l = %ld\nsam4.d = %lf\n\n", d4.i, d4.l, d4.d);

    return 0;
}
```

Yukarıdaki *main* işlevinde *struct Sample* türünden *d1, d2, d3, d4* isimli *4* değişkenin tanımlandığını görüyorsunuz. *d2* değişkenine atama işleci kullanılarak *d1* değişkeninin değeri atanıyor. *d4* değişkenine de standart *memcpy* işlevi kullanılarak *d3* nesnesinden kopyalama yapılıyor.

Tür Dönüştürme İşleci ve Yapı Nesneleri

Bir yapı nesnesi tür dönüştürme işlecinin terimi olamaz. Yani aşağıdaki gibi bir kod da geçersizdir:

```
struct S1 {
    int a, b;
};

struct S2 {
    int a, b;
};

void func()
{
    struct S1 s1;
    struct S2 s2;

    s1.a = 10;
    s1.b = 20;
    s2 = (struct S2)s1;    /* Geçersiz! */
}
```

Yapı Elemanlarının Bellekteki Yerleşimi

Yapı elemanları belleğe, bildirimde ilk yazılan eleman küçük adreste olacak biçimde, bitişik olarak yerleştirilir. Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

struct Date {
    int day, mon, year;
};
```

```
int main()
{
    struct Date date;

    printf("date.day'in adresi = %p\n", &date.day);
    printf("date.mon'un adresi = %p\n", &date.mon);
    printf("date.year'in adresi = %p\n", &date.year);

    return 0;
}
```

Yapı Elemanı Olarak Dizilerin Kullanılması

Yapının bir elemanı herhangi türden bir dizi olabilir. Bu durumda da yapının dizi elemanının ismi bir nesne belirtmez, bir adres bilgisidir. Önce aşağıdaki örneği inceleyin:

```
#include <stdio.h>

#define    NAME_LEN       20

struct Person {
    char name[NAME_LEN];
    int no;
};

int main()
{
    struct Person per;

    gets(per.name);
    puts(per.name);

    putchar(per.name[3]);
    per.name++             /* Geçersiz. Dizi ismi nesne değildir. */
    return 0;
}
```

Bildirilen *struct Person* yapısının bir elemanı *char* türden *20* elemanlı bir dizidir. Bu durumda kaynak kodumuz *Unix* işletim sisteminde derlendiğinde, *sizeof(struct Person) 24* olur. Tanımlanan her *struct Person* türünden nesnenin içinde *20* elemanlı *char* türden bir dizi yer alır.

main işlevi içinde kullanılan *per.name* ifadesi, *per* yapı nesnesinin içinde yer alan *char* türden dizinin başlangıç adresidir. Bu ifade işleme sokulduğunda, derleyici tarafından otomatik olarak *char* * türüne dönüştürülür.

```
gets(per.name);
```

çağrısıyla *gets* işlevine, *per* nesnesi içindeki *char* türden *name* dizisinin başlangıç adresi geçiliyor. Böylece klavyeden alınan yazı nesnenin içindeki diziye yerleştiriliyor.

```
puts(per.name);
```

çağrısıyla da benzer biçimde bu dizinin içindeki yazı ekrana yazdırılıyor.

Ancak istenirse, nesnenin içindeki dizinin herhangi bir elemanına köşeli ayraç işleci ya da içerik işleci ile ulaşılabilir:

```
per.name[3]
*(per.name + 3)
```

ifadelerinin türü *char* türüdür. Bu ifadeler *per* nesnesinin içindeki *name* dizisinin *3* indisli elemanı olan *char* türden nesneye karşılık gelir.

struct Person türünden bir dizinin içinde tutulan yazının uzunluğu ne olursa olsun, *struct Person* türünün *sizeof* değeri değişmez. Bazı durumlarda, yapı nesneleri içinde yer alan diziler, yapı nesnesinin boyutunu gereksiz bir biçimde artırdığı için tercih edilmez.

Bir yapının elemanı neden bir dizi olur? Bu yolla bir yapı nesnesi içinde aynı türden belirli sayıda değer tutulabilir. Bir öğrenciye ilişkin bilgilerin, bir yapı türü ile modellendiğini düşünün. Böyle bir yapının içinde, öğrencinin notlarını içinde tutacak *int* türden bir dizi yer alabilir. Ancak uygulamalarda en çok görülen durum, yazı bilgilerinin yapı içinde yer alan *char* türden diziler içinde tutulmasıdır.

Aynı türden belirli sayıda değerin bir yapı nesnesine bağlı olarak tutulması için, bir başka yol da yapının bir elemanının bir gösterici olmasıdır.

Yapı Elemanı Olarak Gösterici Değişkenlerin Kullanılması

Yapının bir elemanı herhangi türden bir gösterici olabilir. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

struct Person {
    char *name_ptr;
    int no;
};

int main()
{
    struct Person per;
    per.name_ptr = "Necati Ergin";
    per.no = 125;
    printf("%s %d\n", per.name_ptr, per.no);

    return 0;
}
```

Yukarıdaki kod parçasında bu kez *struct Person* türünün bir elemanı *char* türden bir gösterici olarak seçiliyor. Bir gösterici kullanılan sistemde *2* ya da *4* byte yer kaplayacağına göre yukarıdaki *struct Person* yapısının *sizeof* değeri *UNIX* altında *8 byte* olur.

```
per.name_ptr = "Necati Ergin";
```

İfadesi ile *per* nesnesinin *name_ptr* isimli elemanına *"Necati Ergin"* dizgesi atanıyor. Dizgelerin birer adres bilgisi olduğunu, derleyici tarafından *char* * türünden ifadeler olarak değerlendirildiklerini biliyorsunuz.

```
per.name_ptr
```

char * türden nesne gösteren bir ifadedir. Bu ifade *per* nesnesi içinde yer alan *char* türden gösterici değişkene karşılık gelir.
Bu göstericinin gösterdiği yere yani

```
*per.name_ptr
per.name_ptr[n]
```

ifadelerine gösterici hatası oluşturmadan atama yapabilmek için, önce eleman olan göstericinin güvenilir bir yeri göstermesi gerekir.
Bir yapının elemanının bir gösterici olması çoğu zaman, yapı nesnesinin dinamik olarak elde edilen bir bellek alanını kullanması amacıyla istenir. Aşağıdaki kod parçasını inceleyin:

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

#define    ARRAY_SIZE     100

struct Person {
    char *name_ptr;
    int no;
};

int main()
{
    struct Person per;
    char name_entry[ARRAY_SIZE];

    printf("ismi giriniz : ");
    gets(name_entry);
    per.name_ptr = (char *)malloc(strlen(name_entry) + 1);
    if (per.name_ptr == NULL) {
        printf("bellek tahsis edilemiyor!\n");
        exit(EXIT_FAILURE);
    }
    strcpy(per.name_ptr, name_entry);
    printf("numarayi giriniz : ");

    scanf("%d", &per.no);
    printf("isim = %s\nNo : %d\n", per.name_ptr, per.no);
    /*...*/
    free(per.name_ptr);

    return 0;
}
```

main işlevi içinde yapılanları sırasıyla inceleyelim: Önce *struct Person* türünden *per* isimli bir değişkenin tanımlandığını görüyorsunuz. Klavyeden girilen isim, önce standart *gets* işleviyle yerel *name_entry* dizisine alınıyor. Yerel diziye alınmış ismin uzunluğunun 1 fazlası kadar byte'lık bir alan *malloc* işleviyle dinamik olarak elde ediliyor. Yazı uzunluğunun 1 fazlası kadar yer ayrılması yazının sonunda yer alacak sonlandırıcı karakter için de yer sağlanması amacı taşıyor. Dinamik alanın başlangıç adresinin *per* nesnesinin *name_ptr* elemanında saklandığını görüyorsunuz. Daha sonra standart *strcpy* işleviyle yerel dizideki isim ayrılan dinamik bloğa kopyalanıyor.

per nesnesi böylece kendi elemanı olan *name_ptr* göstericisi yoluyla dinamik bir alanı kontrol eder hale geliyor, değil mi? Bir *struct Person* nesnesinin tuttuğu isme ulaşmak için önce nesnenin *name_ptr* elemanına erişip bu elemanın değerinden de dinamik bloğa erişilebilir. Nesnenin ömrünün sona ermesinden önce, bellek sızıntısını (*memory leak)* engellemek amacıyla, dinamik bloğun geri verilmesi gerekir.

```
free(per.name_ptr);
```

Çağrısıyla geri verme işleminin yapıldığını görüyorsunuz.

Yapı Değişkenlerine İlkdeğer Verilmesi

Dizilere ilkdeğer vermeye ilişkin özel bir sözdizimi olduğunu biliyorsunuz. Yapı değişkenlerine de benzer bir sözdizimle ilkdeğer verilebilir. Verilen ilkdeğerler sırası ile yapı elemanlarına yerleştirilir. Daha az sayıda yapı elemanına ilkdeğer verilebilir. Bu durumda ilkdeğer verilmemiş yapı elemanları otomatik olarak *0* değeri alır.

```
#include <stdio.h>

struct Date {
    int day, month, year;
};

int main()
{
    struct Date x = {10, 12, 1999};
    struct Date y = {10};

    printf("%d %d %d\n", x.day, x.month, x.year);
    printf("%d %d %d\n", y.day, y.month, y.year);

    return 0;
}
```

İlkdeğer verme sözdiziminde, yapı nesnesinin isminden sonra yer alan atama işlecini bir blok izler. Bu bloğun içinde virgüllerle ayrılan bir liste ile yapı nesnesinin elemanlarına ilkdeğer verilir. İlk ifadenin değeri yapı nesnesinin ilk elemanına, ikinci ifadenin değeri yapı nesnesinin ikinci elemanına atanır.

Dizilerde olduğu gibi, yapılarda da bir yapı nesnesinin elemanlarından daha fazla sayıda elemana ilkdeğer vermek geçersizdir. Yapının içinde yazı tutmak için bildirilen *char* türden bir dizi var ise bu diziye de çift tırnak içinde ayrıca ilkdeğer verilebilir:

```
#include <stdio.h>

#define    MAX_NAME_LEN   20

struct Person {
    char name[MAX_NAME_LEN + 1];
    int no;
};

int main()
{
    struct Person per = {"Mustafa", 256};

    printf("isim : %s\nNo   : %d\n", per.name, per.no);

    return 0;
}
```

main işlevi içinde, *struct Person* türünden *per* değişkenine ilkdeğer veriliyor. Tanımlanan *per* değişkeninin *name* isimli *char* türden dizisine, ilkdeğer verme sözdizimiyle *Mustafa* yazısı yerleştiriliyor. Yine *per* değişkeninin *no* isimli elemanına da ilkdeğer verme sözdizimiyle *256* değeri atanıyor.

Yapı elemanı olan dizinin bütün elemanlarına değil de, belirli sayıda elemanına ilkdeğer vermek mümkündür. Aşağıdaki örneği derleyerek çalıştırın:

```
struct Sample {
    int x;
    int a[10];
    double d;
};

#include <stdio.h>

int main()
{
    struct Sample sam = {20, 1, 2, 3, 7.8};
    int k;

    printf("sam.x = %d\n", sam.x);

    for (k = 0; k < 10; ++k)
        printf("sam.a[%d] = %d\n", k, sam.a[k]);
    printf("sam.d = %lf\n", sam.d);

    return 0;
}
```

Programcının *sam.x* nesnesine *20*, *sam.a* dizisinin ilk üç elemanına sırasıyla *1, 2, 3* değerlerini atamak istediğini, *sam.a* dizisinin geri kalan elemanlarının otomatik olarak sıfırlanmasının beklediğini düşünelim. *sam.d* nesnesine de 7.8 ilkdeğerinin verilmek istendiğini düşünelim. Bildirimi yapılmış *struct Sample* türünden *sam* isimli nesneye aşağıdaki gibi ilkdeğer aktarılıyor:

```
struct Sample sam = {20, 1, 2, 3, 7.8};
```

Ancak yukarıdaki deyimle bu yapılamaz. Derleyici *20* değerini *sam.x* nesnesine yerleştirirken *sam.a* dizisinin ilk dört elemanına sırasıyla *1, 2, 3, 7.8* değerlerini yerleştirir. Tabi otomatik tür dönüşümüyle *double* türden olan *7.8* dizinin dördüncü elemanına atanırken *int* türüne dönüştürülür. *double* türden *per.d* nesnesine ise ilkdeğer verilmediğinden, bu eleman otomatik olarak *0* değeriyle başlatılır.
İlkdeğer verme deyiminde, ikinci bir blok kullanılarak eleman olan dizinin ilk *3* elemanına ilkdeğer ataması yapılması sağlanabilir. İlkdeğer verme deyimini aşağıdaki gibi değiştirerek kodu yeniden derleyip, çalıştırın:

```
struct Sample sam = {20, {1, 2, 3}, 7.8};
```

Böyle bir ilkdeğer verme işleminde, ikinci küme ayracı çifti içinde yazılan virgüllerle ayrılmış değerler, eleman olan dizinin ilk üç elemanına atanırken, kapanan küme ayracını izleyen *7.8* değeri ise yapı nesnesinin *d* isimli elemanına aktarılmış olur.

Yapı Türünden Adresler ve Göstericiler

Bir yapı nesnesinin adres işlecinin terimi olmasıyla, yapı nesnesinin adresi elde edilebilir. Bir yapı türünden gösterici değişkenler de tanımlanabilir. Bir yapı türünden gösterici değişkene, aynı yapı türünden bir adres atanmalıdır:

```
#include <stdio.h>

struct Point {
    double x, y;
};

int main()
{
    struct Point p;
    struct Point *ptr = &p;
    printf("sizeof ptr  = %d\n", sizeof(ptr));
    printf("sizeof *ptr = %d\n", sizeof(*ptr));

    return 0;
}
```

main işlevi içinde *struct Point* türünden *p* isimli bir değişken tanımlanıyor. Bu nesnenin adresi *struct Point* türünden bir gösterici değişken olan *ptr*'ye atanıyor. Bu atamadan sonra artık *p* değişkenine *ptr* gösterici değişkeni ile de ulaşılabilir, değil mi?
Bir gösterici değişken ne türden bir nesneyi gösterirse göstersin, göstericinin *sizeof* değeri *2* ya da *4 byte*'dır.

```
sizeof(ptr)
```

ifadesi ile *ptr* gösterici değişkeninin kendi *sizeof* değeri elde edilirken

```
sizeof(*ptr)
```

ifadesi ile *ptr*'nin gösterdiği nesnenin yani *p* değişkeninin *sizeof* değeri elde ediliyor.

Yapı Göstericisi İle Yapı Elemanlarına Erişim

ptr bir yapı türünden bir gösterici ve *mem* de o yapının bir elemanı olmak üzere *ptr*'nin gösterdiği nesnenin *mem* isimli elemanına aşağıdaki gibi erişilebilir:

```
(*p).mem
```

İçerik işleci işleç öncelik tablosunun ikinci düzeyindeyken, nokta işleci birinci öncelik düzeyindedir. Burada **p* ifadesinin öncelik ayracı içine alınması zorunludur. Eğer ifade

```
*p.mem      /* Geçersiz */
```

biçiminde yazılsaydı, önce nokta işleci değer üreteceğinden, *p.mem* ifadesi ele alınırdı.Nokta işlecinin sol terimi bir yapı nesnesi olmadığı için, ifade geçersiz olurdu.

Ok İşleci

Bir yapı türünden adres söz konusu olduğunda bu adresteki yapı nesnesinin belirli bir elemanına erişmek için üç ayrı işleç kullanılabilir: Öncelik işleci, içerik işleci, nokta işleci. Bu erişim, ismi *ok işleci* olan tek bir işleçle yapılabilir.

Ok işleci - ve > karakterlerinin yanyana getirilmesiyle oluşturulur. İki terimli araek konumunda (*Binary infix*) bir işleçtir. Ok işleci öncelik tablosunun en yüksek öncelik seviyesindedir. -> işlecinin sol terimi bir yapı türünden adres olmalıdır. İşlecin sağ terimi ise ilgili yapının bir elemanı olmalıdır. İşleç, sol terimi olan adresteki yapı nesnesinin, sağ terimi olan isimli elemanına erişmek için kullanılır.

ptr, bir yapı türünden bir nesnenin adresini tutuyor olsun. Aşağıdaki iki ifade eşdeğerdir.

```
(*ptr).mem
ptr->mem
```

Nokta ve ok işleçlerinin her ikisi de elemana erişmek için kullanılır. Ancak nokta işleci yapı değişkeninin kendisiyle, ok işleci ise adresiyle erişim sağlar.

Yapı Nesnelerinin İşlevlere Geçirilmesi

Bir işlevin değerle (*call by value*) ya da adresle (*call by reference*) çağrılabileceğini biliyorsunuz. Yapı nesneleriyle ilgili bir iş gören işlev de, değerle ya da adresle çağrılabilir.

Yapı Nesnesinin Değerinin İşleve Gönderilmesi

Bir işleve gönderilen argüman olan ifade, işlevin ilgili parametre değişkenine kopyalanır. Bu durumda bir işlevin parametre değişkeni bir yapı türünden ise, işlev aynı yapı türünden bir nesne ile çağrılabilir. Aynı türden yapı nesnelerinin atama işleciyle birbirine atanabileceğini biliyorsunuz. Böyle bir atama blok kopyalaması anlamına geldiği için, hem bellek hem de işlem zamanı açısından göreli bir kayba neden olur. Üstelik bu yöntemle, işlev kendisine gönderilen argümanları değiştiremez. Çünkü işlev değerle çağırılmaktadır. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

#define    MAX_NAME_LEN       16
#define    MAX_FNAME_LEN      24

struct Person {
    char name[MAX_NAME_LEN + 1];
    char fname[MAX_FNAME_LEN + 1];
    int no;
    double wage;
};

void display_person(struct Person per)
{
    printf("%d %s %s %.2lf\n", per.no, per.name, per.fname, per.wage);
}

int main()
{
    struct Person person = {"Necati", "ERGIN", 2345, 3.56};

    display_person(person);

    return 0;
}
```

display_person işlevinin parametre değişkeni *struct Person* türünden bir nesnedir. *main* işlevi içinde *struct Person* türünden ismi *person* olan bir nesnenin ilkdeğer verilerek tanımlandığını görüyorsunuz. *display_person* işlevine yapılan çağrıda argüman olarak *person* nesnesinin değeri kullanılıyor. İşlev çağrıldığı zaman, yaratılan işlevin parametre değişkeni olan *per* nesnesine, *main* bloğu içindeki *person* değişkeninin değeri kopyalanır. Bu bir blok kopyalama işlemidir. *sizeof(struct Person)* büyüklüğünde bir blok kopyalanır. *per* nesnesi *person* nesnesi ile aynı nesne değildir. *per* nesnesi değeri *person* nesnesinin değerine eşdeğer başka bir nesnedir. *display_person* işlevi içinde

```
per.wage = 5.60;
```

gibi bir atama yapılsaydı, şüphesiz bu atamadan *person* nesnesi etkilenmemiş olurdu.

Yerel bir yapı nesnesinin değeri böyle bir işlevle değiştirilemez, değil mi? İşlevin bir yapı nesnesinin değerini alması çoğu zaman gereksiz bir blok kopyalanmasına neden olur. İlgili yapı türünün *sizeof* değeri büyüdükçe, bellek ve işlemci zamanı kullanımı açısından verimsizlik artacağından, pek tercih edilen bir yöntem değildir. Küçük yapı nesneleri için kullanılabilir.

Yapı Nesnesinin Adresinin İşleve Gönderilmesi

Bir işlevin parametre değişkeni yapı türünden bir gösterici olursa, işlev de bu türden bir yapı nesnesinin adresi ile çağrılabilir. Böyle bir işleve yapılan çağrı ile, yalnızca bir gösterici nesnesinin *sizeof*'u kadar veri işleve gönderilir.
Bu yöntem çoğunlukla daha iyidir. Hemen her zaman bu yöntem kullanılmalıdır. Bu yöntemde yapı ne kadar büyük olursa olsun, aktarılan yalnızca bir adres bilgisidir. Üstelik bu yöntemde işlev kendisine adresi gönderilen yapı değişkenini değiştirebilir. Şüphesiz böyle bir aktarım işlemi, yapı nesnesinin bellekte tek bir blok olarak yer alması yüzünden mümkündür. Daha önce tanımlanan *display_person* isimli işlev, bu kez bir yapı nesnesinin adresini alacak biçimde yazılıyor:

```
void display_person(const struct Person *ptr)
{
    printf("%d %s %s %.2lf\n", ptr->no, ptr->name, ptr->fname, ptr->wage);}

int main()
{
    struct Person person = {"Necati", "ERGIN", 2345, 3.56};

    display_person(&person);

    return 0;
}
```

display_person isimli işlevin parametre değişkeninin

```
const struct Person *ptr
```

biçiminde bildirildiğini görüyorsunuz. İşlev dışarıdan *struct Person* türünden bir nesnenin adresini istiyor. Bildirimde kullanılan *const* anahtar sözcüğünün, adresi alınan yapı nesnesinde değişiklik yapılmayacağı bilgisini ilettiğini biliyorsunuz.

İşlevin kodu içinde kullanılan

```
ptr->no
```

ifadesi *int* türdendir. Bu ifade dışarıdan adresi alınan yapı nesnesinin *no* isimli elemanı olan nesneye karşılık gelir.

Bir Yapı Türüne Geri Dönen İşlevler

Bir işlevin geri dönüş değeri bir yapı türünden olabilir. Aşağıdaki örneği inceleyin:

```
struct Point {
    double m_x, m_y;
};

struct Point make_point(double x, double y);

int main()
{
    struct Point a;
```

```
    a = make_point(3, 5);
    /***/
    return 0;
}

struct Point make_point(double x, double y)
{
    struct Point temp;

    temp.m_x = x;
    temp.m_y = y;
    return temp;
}
```

İşlevlerin geri dönüş değerlerini geçici bir nesne yardımıyla dışarıya ilettiklerini biliyorsunuz. Geri dönüş değeri üreten bir işlev içinde kullanılan *return* ifadesi, işlevin geri dönüş değerini içinde tutacak geçici bir nesneye atanır. İşlev çağrı ifadesi de bu geçici nesnenin değerini temsil eder. Bu geçici nesne, işlevin kodunun çalışması *return* deyimine gelince yaratılır, işlev çağrısının yer aldığı ifadenin değerlendirilmesi bitince yok edilir. Bir işlevin geri dönüş değerinin türü, işlevin geri dönüş değerini içinde taşıyacak geçici nesnenin türüdür.
Yukarıdaki kod parçasında *make_point* işlevinin geri dönüş değeri, *struct Point* türündendir. Bu durumda işlevin geri dönüş değerinin aktarıldığı geçici nesne de *struct Point* türünden olur. Böyle bir işlevin geri dönüş değeri, aynı türden bir yapı değişkenine atanabilir.
Ancak böyle işlevler bellekte çok fazla bir yer kaplamayan küçük yapı nesneleri için kullanılmalıdır. Çünkü *return* ifadesiyle geçici bölgeye geçici bölgeden de geri dönüş değerinin saklanacağı değişkene atamalar yapılır. Bu da kötü bir teknik olmaya adaydır. Küçük yapılar için tercih edilebilir. Çünkü algısal karmaşıklığı daha azdır.

Yapı Türünden Bir Alanın Dinamik Olarak Elde Edilmesi

Nasıl bir dizi için bellek alanı dinamik olarak elde edilebiliyorsa bir yapı nesnesi için de dinamik bir bellek bloğu elde edilebilir. Aşağıdaki örneği derleyerek çalıştırın:

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

#define    MAX_NAME_LEN       15
#define    MAX_FNAME_LEN      23

struct Person {
    char name[MAX_NAME_LEN + 1];
    char fname[MAX_FNAME_LEN + 1];
    int no;
    double wage;
};

void set_person(struct Person *ptr, const char *name_ptr, const char
*fname_ptr, int n, double w)
{
    strcpy(ptr->name, name_ptr);
    strcpy(ptr->fname, fname_ptr);
    ptr->no = n;
    ptr->wage = w;
}

void display_person(const struct Person *ptr)
{
    printf("%d %s %s %.2lf\n", ptr->no, ptr->name, ptr->fname, ptr->wage);
```

```
}

int main()
{
    struct Person *ptr;

    ptr = (struct Person *)malloc(sizeof(struct Person));
    if (ptr == NULL) {
        printf("bellek tahsis edilemiyor!\n");
        exit(EXIT_FAILURE);
    }
    set_person(ptr, "Kaan", "Aslan", 2345, 4.80);
    display_person(ptr);

    free(ptr);

    return 0;
}
```

main işlevini inceleyin. *malloc* işleviyle *struct Person* türünden bir nesnenin sığabileceği büyüklükte bir bellek alanı dinamik olarak elde ediliyor. Dinamik bloğun başlangıç adresi *struct Person* türünden bir gösterici değişken olan *ptr*'de tutuluyor. *set_person* isimli işlevin, dışarıdan adresini aldığı yapı nesnesini, diğer parametrelerine aktarılan bilgilerle doldurduğunu görüyorsunuz. Daha önce tanımlanan *display_person* isimli işlev ise, yine adresini aldığı yapı nesnesinin tuttuğu bilgileri ekrana yazdırıyor. Bu işlevlere, yapı nesnesinin adresi olarak, elde edilen dinamik bloğun başlangıç adresinin geçildiğini görüyorsunuz.

Bir Yapı Türünden Adrese Geri Dönen İşlevler

Bir işlevin geri dönüş değeri, bir yapı türünden adres de olabilir. Bu durumda işlevin geri dönüş değerini içinde tutacak geçici nesne bir yapı türünden adrestir. İşlev çağrısı, işlevin geri dönüş değeri türünden bir göstericiye atanabilir. Aşağıdaki işlev tanımını inceleyin:

```
struct Person *create_person(const char *name_ptr, const char
*fname_ptr,int n, double w)
{
    struct Person *ptr;

    ptr = (struct Person *)malloc(sizeof(struct Person));
    if (ptr == NULL) {
        printf("bellek elde edilemiyor!\n");
        exit(EXIT_FAILURE);
    }
    set_person(ptr, name_ptr, fname_ptr, n, w);

    return ptr;
}
```

create_person işlevi dinamik olarak yerini ayırdığı bir nesneyi parametrelerine aktarılan bilgilerle dolduruyor. Daha sonra dinamik nesnenin adresiyle geri dönüyor. Bu işlevi kullanan bir kod parçası aşağıdaki gibi olabilir:

```
int main()
{
    struct Person *ptr;

    ptr = create_person("Kaan", "Aslan", 2345, 4.80);
    display_person(ptr);
```

```
    free(ptr);
    /***/
    return 0;
}
```

main işlevi içinde çağrılan *create_person* işlevinin geri döndürdüğü adres, *struct Person* türünden bir gösterici değişken olan *ptr*'de tutuluyor. İşlev, dinamik bir nesnenin adresini döndürdüğünden, dinamik alanı geri vermek, işlevi çağıran kod parçasının sorumluluğundadır.
Böyle bir işlev yerel bir yapı değişkeninin adresine geri dönemez. Global bir yapı değişkeninin adresine de geri dönmesinin çoğu zaman fazlaca bir anlamı yoktur.
İşlev, dinamik olarak yeri ayrılmış bir yapı nesnesinin adresi ile de geri dönebilir.
Elde edilen dinamik alanın *"heap"* bellek alanına geri verilmesi, işlevi çağıran kod parçasının sorumluluğundadır. *main* içinde *free* işlevine yapılan çağrı ile dinamik alanın geri verildiğini görüyorsunuz.
Bir işlevin yerel bir yapı nesnesinin adresine geri dönmesi tipik bir gösterici hatasıdır. Yerel yapı nesneleri de *static* anahtar sözcüğüyle bildirilebilir. Şüphesiz bir işlevin *static* bir yapı nesnesinin adresine geri dönmesi bir gösterici hatası değildir.

Bileşik Nesneler ve İçsel Yapılar

Bir yapının elemanı başka bir yapı türünden olabilir.
Bir yapının elemanının başka bir yapı türünden olabilmesi iki ayrı biçimde sağlanabilir:
Önce eleman olarak kullanılan yapı türü bildirilir. Bu bildirimin görülür olduğu bir yerde elemana sahip olan yapının bildirimi yapılır. Aşağıdaki örneği inceleyin:

```
struct Date {
    int day, month, year;
};

struct Student {
    char name[30];
    struct Date birth_date;
    int no;
};
```

Yukarıdaki örnekte önce *struct Date* yapısı bildiriliyor. Daha sonra yapılan *struct Student* türünün bildiriminde, *struct Date* türünden bir eleman kullanılıyor.
İkinci yöntemde, eleman olan yapı değişkeninin bildirimi, elemana sahip yapının bildirimi içinde yapılır:

```
struct Student {
    char name[30];
    struct Date {
        int day, month, year;
    }birth_date;
    int no;
};
```

Yukarıdaki örnekte yer alan *struct Date* yapısı gibi, bir yapının içinde bildirilen yapıya "içsel yapı" (*nested structure*) denir. Burada, içte bildirilen yapı da sanki dışarıda bildirilmiş gibi işlem görür. Yani içeride bildirilen yapı türünden değişkenler tanımlanabilir. Yukarıdaki kod parçası yalnızca bir bildirime karşılık gelir. *birth_date* isimli bir nesne tanımlanmış olmaz. *struct Student* türünden bir nesne tanımlandığında, bu yapı nesnesinin *struct Date* türünden *birth_date* isimli elemanı olur.
İki ayrı sözdiziminden herhangi birinden sonra *struct Student* türünden bir nesne tanımlanmış olsun:

```
struct Student s;
```

s yapı değişkeninin *struct Date* türünden olan *birth_date* isimli elemanına nokta işleciyle ulaşılabilir:

```
s.birth_date
```

Yukarıdaki ifade *struct Date* türünden bir nesne gösterir. İstenirse bu nesnenin *day*, *mon*, *year* isimli elemanlarına nokta işlecinin ikinci kez kullanılmasıyla erişilebilir:

```
s.birth_date.mon
```

ifadesi *int* türden bir nesne gösterir. Nokta işlecinin işleç öncelik tablosunun birinci öncelik düzeyinde olduğunu, bu öncelik düzeyine ilişkin öncelik yönünün soldan sağa olduğunu anımsayın.
Bir ya da birden fazla elemanı programcı tarafından bildirilen bir türden olan yapı nesnelerine bileşik nesne (*composite object*) denir. Yukarıdaki örnekte *struct Student* türünden olan *s* değişkeni bir bileşik nesnedir.

Bileşik Nesnelere İlkdeğer Verilmesi

Normal olarak ilkdeğer vermede elemanlar sırasıyla, içteki yapı da dikkate alınacak biçimde, yapı elemanlarına atanır. Ancak içteki yapının elemanlarına verilen değerlerin, ayrıca küme ayraçları içine alınması, okunabilirliği artırdığı için salık verilir:

```
struct Student s = {"Necati Ergin", {10, 10, 1967},  123};
```

Eğer içteki yapı ayrıca küme ayracı içine alınmışsa içteki yapının daha az sayıda elemanına ilkdeğer vermek mümkün olabilir:

```
struct Student s = {"Necati Ergin", {10, 10}, 123};
```

Burada *s* değişkeninin, *birth_date* elemanının *year* elemanına ilkdeğer verilmiyor. Derleyici bu elemana otomatik olarak *0* değeri yerleştirir. Burada içteki küme ayraçları kullanılmasaydı, *123* ilkdeğeri *year* elemanına atanır, *no* elemanına ise *0* değeri verilirdi. Aşağıdaki gibi bir bildirim de geçerlidir:

```
struct Data {
    int a[3];
    long b;
    char c;
} x = {{1, 2, 3}, 50000L, 'A'}, *p;
```

Bileşik Nesne Oluşturmanın Faydaları

Bir yapının başka bir yapı türünden elemana sahip olması durumunda, içerilen elemanın ait olduğu yapı türünün arayüzünden, yani dışarıya hizmet veren işlevlerinden faydalanma olanağı doğar.

Aşağıdaki gibi bir yapı bildirilmiş olsun:

```
struct Person {
    char name[16]
    char fname[20]
    struct Date bdate;
};
```

Yukarıdaki bildirimde *struct Person* yapısının bir elemanı *struct Date* türündendir.

```
struct Person per;
```

gibi bir nesne tanımlandığında bu nesnenin elemanı olan

```
per.bdate
```

ifadesi *struct Date* türünden bir nesneye karşılık gelir. *per* nesnesinin bu elemanı söz konusu olduğunda, *struct Date* türünün dışarıya karşı hizmet vermek üzere tanımlamış işlevleri çağrılabilir. Örneğin *struct Date* türüyle ilgili olarak aşağıdaki gibi bir işlevin bildirilmiş olsun:

```
void display_date(const struct Date *);
```

per nesnesinin içinde tuttuğu tarih bilgisini ekrana yazdırmak için doğrudan *display_date* işlevi çağrılabilir, değil mi?

```
void display_person(const struct Person *ptr)
{
    printf("isim          : %s\n", ptr->name);
    printf("soyisim       : %s\n", ptr->fname);
    printf ("doğum tarihi : ");
    display_date(&ptr->bdate);
}
```

display_person işlevi, kendisine gönderilen adresteki *struct Person* türünden nesnenin elemanı olan, *bdate* nesnesi içinde saklanan tarih bilgisini ekrana yazdırmak amacıyla *display_date* isimli işlevi çağırıyor. İşleve argüman olarak *bdate* elemanının adresini gönderiyor.

Yapıların Neden Kullanılır

Yapıların kullanılmasıyla bazı faydalar elde edilebilir:

Birbirleri ile ilişkili olan değişkenler yapı elemanları olarak bir yapı içinde toplanırsa algısal kolaylık sağlanır. Örneğin düzlemde bir nokta, bir tarih bilgisi, bir depoda bulunan mamüllere ilişkin özellikler bir yapı ile temsil edilebilir.

C dilinde bir işlev en fazla 8 - 10 parametre almalıdır. Daha fazla parametreye sahip olması kötü bir tekniktir. Bir işlev çok fazla parametrik bilgiye gereksinim duyuyorsa, bu parametrik bilgiler bir yapı biçiminde ifade edilmelidir. O yapı türünden bir değişken tanımlanmalı, bu değişkenin adresi işleve parametre olarak gönderilmelidir. Örneğin bir kişinin nüfus cüzdan bilgilerini parametre olarak alıp bunları ekrana yazdıracak bir işlev tasarlayacak olalım. Nüfus cüzdanı bilgilerinin hepsi bir yapı biçiminde ifade edilebilir ve yalnızca bu yapının adresi işleve gönderilebilir.

İşlevlerin tek bir geri dönüş değeri vardır. Oysa işlevlerin çok değişik bilgileri çağıran işleve iletmesi istenebilir. Bu işlem şöyle yapılabilir: İletilecek bilgiler bir yapı biçiminde ifade edilir. Sonra bu türden bir yapı değişkeni tanımlanarak adresi işleve gönderilir. İşlev de bu yapı değişkeninin içini doldurur. Yani işlev çıkışında bilgiler yapı değişkeninin içinde olur.

Yapı Dizileri

Yapılar da bir tür belirttiğine göre yapı türünden de diziler söz konusu olabilir. Bir yapı dizisi elemanları bellekte bitişik olarak yerleştirilen , elemanları aynı yapı türünden olan bir dizidir.
Yapı dizilerine de ilkdeğer verilebilir. İlkdeğer verme sırasında kullanılan içteki küme ayraçları okunabilirliği artırır. Örnek:

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

```
struct Date {
    int d, m, y;
};

int main()
{
    struct Date bdates[5] = {{10, 10, 1958}, {4, 3, 1964},
    {21, 6, 1967}, {22, 8, 1956}, {11, 3, 1970}};
    struct Date *pdate;
    int i;

    pdate = bdates;
    for (i = 0; i < 5; ++i) {
        printf("%02d / %02d / %04d\n", pdate->d, pdate->m, pdate->y);
        ++pdate;
    }
    return 0;
}
```

Bir yapı dizisi üzerinde işlem yapan işlev de tanımlanabilir. Böyle bir işlev, kendisini çağıran koddan, ilgili yapı dizisinin başlangıç adresi ile boyutunu almalıdır. Aşağıda, yapı dizileri ile ilgili işlemler yapan bazı işlevler tanımlanıyor. İşlev tanımlarını dikkatle inceleyin:

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

#define    NAME_LEN       16
#define    FNAME_LEN      20
#define    ARRAY_SIZE     200

struct Person {
    char m_name[NAME_LEN];
    char m_fname[FNAME_LEN];
    int m_no;
    double m_wage;
};

void swap_persons(struct Person *p1, struct Person *p2);
void set_person(struct Person *ptr, const char *name, const char *fname,
int no, double wage);
void set_person_random(struct Person *ptr);
void display_person(const struct Person *ptr);
void set_person(struct Person *ptr, const char *name, const char *fname,
int no, double wage);
void display_person_array(const struct Person *ptr, int size);
void sort_person_array(struct Person *ptr, int size);

char *name_array[20] = {"Ali", "Veli", "Hasan", "Necati", "Burcu", "Kaan",
"Selami", "Salah", "Nejla", "Huseyin", "Derya", "Funda", "Kemal", "Burak",
"Ozlem", "Deniz", "Nuri","Metin", "Guray", "Anil"};
char *fname_array[20] = {"Aslan", "Gencer", "Eker", "Ergin", "Serce",
"Kaynak", "Acar", "Aymir", "Erdin", "Doganoglu", "Avsar", "Ozturk",
"Yilmaz", "Tibet", "Arkin", "Cilasun", "Yildirim", "Demiroglu", "Torun",
"Polatkan"};
/*************************************************************************/
void swap_persons(struct Person *p1, struct Person *p2)
{
    struct Person temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
/*************************************************************************/
void set_person(struct Person *ptr, const char *name, const char *fname,
int no, double wage)
{
    ptr->m_no = no;
    ptr->m_wage = wage;
    strcpy(ptr->m_name, name);
    strcpy(ptr->m_fname, fname);
}
/*************************************************************************/
void set_person_random(struct Person *ptr)
{
    ptr->m_no = rand() % 5000;
    ptr->m_wage = (double)rand() / RAND_MAX + rand() % 5 + 2;
    strcpy(ptr->m_name, name_array[rand() % 100]);
    strcpy(ptr->m_fname, fname_array[rand() % 50]);
}
/*************************************************************************/
void display_person(const struct Person *ptr)
{
    printf("%-16s %-20s%-5d\t%4.2lf\n", ptr->m_name, ptr->m_fname, ptr-
```

```
>m_no, ptr->m_wage);
}
/*************************************************************************/
void set_person_array_random(struct Person *ptr, int size)
{
        int k;

        for (k = 0; k < size; ++k)
            set_person_random(ptr + k);
}
/*************************************************************************/
void display_person_array(const struct Person *ptr, int size)
{
    int k;

    for (k = 0; k < size; ++k)
        display_person(ptr + k);
}
/*************************************************************************/
void sort_person_array(struct Person *ptr, int size)
{
    int i, k;

    for (i = 0; i < size - 1; ++i)
        for (k = 0; k < size - 1 - i; ++k)
            if(strcmp(ptr[k].m_fname, ptr[k + 1].m_fname) > 0)
               swap_persons(ptr + k, ptr + k + 1);
}
/*************************************************************************/
int main()
{
    struct Person a[ARRAY_SIZE];

    srand(time(0));
    set_person_array_random(a, ARRAY_SIZE);
    printf("siralanmadan once : \n");
    display_person_array(a, ARRAY_SIZE);
    sort_person_array(a, ARRAY_SIZE);
    printf("siralanmadan sonra: \n");
    display_person_array(a, ARRAY_SIZE);

    return 0;
}
```

Aşağıdaki gibi bir işlev tanımlanmak istensin:

```
void search_display(const struct Person *ptr, int size, const char *name);
```

İşlev başlangıç adresini ve boyutunu aldığı dizi içinde *name* parametresine aktarılan isimli kişilerin bilgilerini, soyadlarına göre artan sırada ekrana yazdırsın. İşlevin tanımı ile sınama amacıyla yazılan yeni *main* işlevini inceleyin:

```
void search_display(const struct Person *ptr, int size, const char *name)
{
    int i,k;
    struct Person **pd = NULL;
    struct Person *temp;
    int count = 0;

    for (k = 0; k < size; ++k)
        if (!strcmp(ptr[k].m_name, name)) {
            pd = (struct Person **)realloc(pd, (count + 1) * sizeof(struct
Person *));
            if (pd == NULL) {
                printf("bellek blogu elde edilemiyor!\n");
                exit(EXIT_FAILURE);
            }
            pd[count++] = (struct Person *)(ptr + k);
        }

    if (!count) {
        printf("aranan isim dizide bulunamadi\n");
        return;
    }

    for (i = 0; i < count - 1; ++i)
        for (k = 0; k < count - 1 - i; ++k)
            if (strcmp(pd[k]->m_fname, pd[k + 1]->m_fname) > 0) {
                temp = pd[k];
                pd[k] = pd[k + 1];
                pd[k + 1] = temp;
            }

    for (k = 0; k < count; ++k)
        display_person(pd[k]);
    free(pd);
    printf("toplam %d kisi bulundu!\n", count);
}
```

```
int main()
{
    struct Person a[ARRAY_SIZE];
    char name_entry[20];

    srand(time(0));

    set_person_array_random(a, ARRAY_SIZE);
    display_person_array(a, ARRAY_SIZE);
    printf("aranan ismi giriniz : ");
    gets(name_entry);
    search_display(a, ARRAY_SIZE, name_entry);

    return 0;
}
```

Yapılara İlişkin Karmaşık Durumlar

Bir yapının elemanı başka bir yapı türünden gösterici olabilir. Örneğin:

```
struct Date {
    int day, month, year;
};

struct Person {
    char name[30];
    struct Date *bdate;
    int no;
};

struct Person per;
```

şeklinde bir tanımlama yapılmış olsun:

per.bdate ifadesi *struct Date* * türündendir. Bu ifade bir nesne belirtir.
per.bdate->day ifadesinin türü *int*'dir. Bu ifade de nesne gösterir.
&per.bdate->day ifadesinin *int* * türündendir.
Tabi bu örnekte, bir değer ataması yapılmamışsa, *per.bdate* ile belirtilen gösterici içinde rastgele bir adres vardır. Bu göstericinin bir gösterici hatasına neden olmadan kullanılabilmesi için güvenilir bir adresi göstermesi gerekir. Örneğin bu alan *malloc* işleviyle dinamik olarak elde edilebilir:

```
per.bdate = (struct Date *) malloc (sizeof(struct Date));
```

Yukarıdaki örnekte elimizde yalnızca *struct Person* türünden bir gösterici olduğunu düşünelim.

```
struct Person *ptr;
```

1. *ptr->bdate* ifadesini *struct Date* * türündendir.
2. *person->bdate->day* ifadesinin türü *int*'dir.

Bu örneklerde henüz hiçbir yer ayırma işlemi yapılmamıştır. Hem *ptr* hem de *ptr->date* gösterici değişkenleri için, dinamik bellek işlevleriyle yer elde edilmesi gerekir:

```
ptr = (struct Person *) malloc(sizeof(struct Person));
ptr->bdate = (struct Date *) malloc(sizeof(struct Date));
```

Burada dinamik olarak elde edilen alanlar ters sırada serbest bırakılmalıdır:

```
free(ptr->bdate);
free(ptr);
```

Bir yapının elemanı, kendi türünden bir yapı değişkeni olamaz. Örneğin:

```
struct Sample {
    struct Sample a; /* GEÇERSİZ */
};
```

Çünkü burada *struct Sample* yapısının uzunluğu belirlenemez. Ancak bir yapının elemanı kendi türünden bir gösterici olabilir. Örneğin:

```
struct Node {
    int val;
    struct Node *ptr;
};
```

Bir göstericinin tür uzunluğu neyi gösterdiğinden bağımsız olarak, belirli olduğundan yukarıdaki bildirimde derleyicinin *struct Node* türünün *sizeof* değerini saptamasına engel bir durum yoktur.

Bu tür yapılar özellikle bağlı liste ve ağaç yapılarını (algoritmalarını) gerçekleştirmek amacıyla kullanılabilir.

Bağlı Liste Nedir

Bağlı listenin ne olduğunu anlayabilmek için önce "dizi" veri yapısını hatırlayın.
Aynı türden nesnelerin bellekte tutulmak istendiğini düşünün. Bu nesneler bellekte birbirini izleyecek biçimde yani aralarında hiçbir boşluk olmayacak biçimde tutulabilir. C dilinin dizi aracında da dizinin elemanlarının bellekte bu şekilde tutulduğunu biliyorsunuz. Ancak nesneleri bellekte bitişik olarak yerleştirmek her zaman istene bir durum değildir. Bellekte nesnelerin bitişik yerleştirilmesi drumunda, nesnelerden herhangi birine ulaşım değişmez zamanda yapılabilir. Yani dizinin ya da dinamik dizinin herhangi bir elemanına ulaşma maliyeti, dizide tutuklan eleman sıyısı ile doğru orantılı değildir. Dizide 100 eleman da olsa dizide *1000* eleman da olsa herhangi bir elemana ulaşım maliyeti değişmez bir zaman olur. Neden? Zira dizinin elemanına aslında bir adres işlemiyle ulaşılır değil mi? Örneğin *pd* bir dizinin başlangıç adresini tutan bir gösterici ise
pd[n] gibi bir işlem

```
*(pd + n)
```

işlemine karşılık gelir. Bu işlemin de maliyetinin değişmez olduğu açıktır.

Bellekte elemanları ardışıl olarak bulunmayan listelere bağlı liste denir. Bağlı listelerde her eleman kendinden sonraki elemanın nerede olduğu bilgisini de tutar. İlk elemanın yeri ise ayrı bir göstericide tutulur. Böylece, bağlı listenin ilk elemanının adresi ile, bağlı listenin tüm elemanlarına ulaşılabilir. Bağlı liste dizisinin her elemanı bir yapı nesnesidir. Bu yapı nesnesinin bazı üyeleri bağlı liste elemanlarının değerlerini veya taşıyacakları diğer bilgileri tutarken, bir üyesi ise kendinden sonraki bağlı liste elemanı olan yapı nesnesinin adres bilgisini tutar. Bu şekilde elde edilen bir elemana "düğüm" ("*node*") denir. Örnek:

```
struct Node {
    int val;
    struct Node *next;
};
```

Amacımız int türden değerleri bellekte bir liste şeklinde tutmak olsun. Yukarıda *struct Node* isimli bir yapının bildirimini görüyorsunuz. Tutulacak *int* türden değerler yapımızın *val* isimli elemanının değeri olarak bellekte yer alacak. Yapının yine struct Node türünden olan gösterici elemanı ise kendisinden bir sonra gelen yapı nesnesinin adresini tutacak. Böylece bir *struct Node* nesnesinin adresi elimizdeyken, bu nesnenin içinde tutulan int türden veriye ulaşabileceğimiz gibi, nesnenin içinde yer alan *next* göstericisi yardımıyla da bir sonraki elemana ulaşabiliriz.

Bağlı Listelerle Dizilerin Karşılaştırılması

Bir dizinin herhangi bir elemanına değişmez bir zamanda erişilebilir. Zira bir elemana ulaşma bir adres bilgisine bir tamsayının toplanmasıyla olur. Oysa bağlı listelerde bir elemana erişebilmek için, bağlı listede ondan önce yer alan bütün elemanları dolaşmak gerekir. Bu durumda bir elemana ulaşmanın ortalama maliyeti ortadaki elemana ulaşmanın maliyetidir. Bu da bağlı listedeki eleman sayısının artmasıyla bir elemana ulaşma maliyetinin doğrusal biçimde artacağı anlamına gelir.

Dizilerde araya eleman ekleme ya da eleman silme işlemleri için blok kaydırması yapmak gerekir. Oysa bağlı listelerde bu işlemler çok kolay yapılabilir.

Diziler bellekte ardışıl bulunmak zorundadır. Bu durum belleğin bölünmüş olduğu durumlarda belirli uzunlukta dizilerin açılmasını engeller. Yani aslında istenilen toplam büyüklük kadar boş bellek vardır ama ardışıl değildir. İşte bu durumda bağlı liste tercih edilir.

Bağlı liste kullanımı sırasında eleman ekleme, eleman silme, bağlı listeyi gezme (traverse), vb. işlemler yapılır.

TÜR İSİMLERİ BİLDİRİMLERİ ve typedef BELİRLEYİCİSİ

C dilinde derleyicinin daha önce bildiği bir türe her bakımdan onun yerini tutabilen yeni isimler verilebilir. Bu *typedef* anahtar sözcüğü kullanılarak yapılan bir bildirim deyimi ile sağlanır.

typedef bildiriminin genel biçimi şöyledir:

```
typdef <isim> <yeni isim>;
```

Örnek:

```
typedef unsigned int UINT;
```

Bu bildirimden sonra *UINT* ismi derleyici tarafından *unsigned int* türünün yeni bir ismi olarak ele alınır. Yani kaynak kod içinde, *typedef* bildiriminin görülür olduğu bir noktada, *UINT* ismi kullanıldığında derleyici bunu *unsigned int* türü olarak anlamlandırır.

```
UINT x, y, z;
```

Bildiriminde artık x, y, z değişkenleri *unsigned int* türünden tanımlanmış olurlar. *typedef* bildiriminden sonra artık *UINT* ismi tür belirten isim gereken her yerde kullanılabilir:

```
printf("%d\n", sizeof(UINT));
```

typedef anahtar sözcüğü ile yeni bir tür ismi oluşturulması, bu türe ilişkin önceki ismin kullanılmasına engel olmaz. Yani yukarıdaki örnekte gösterilen *typedef* bildiriminin yapılmasından sonra

```
unsigned int result;
```

gibi bir bildirimin yapılmasına engel bir durum söz konusu değildir.

Şüphesiz *#define* önişlemci komutuyla da aynı iş yapılabilirdi:

```
#define UINT   unsigned int
```

Ancak *typedef* bildirimi derleyici tarafından ele alınırken, *#define* önişlemci komutu ile tanımlanan isimler önişlemci programı ilgilendirir. Yani yukarıdaki önişlemci komutunun kullanılmasından sonra, zaten derleyici *UINT* ismini görmez. Derleyiciye sıra geldiğinde, *UINT* isminin yerini *unsigned int* atomları almış olur.

Algılanması zor olan tür isimlerine, *typedef* bildirimleriyle algılanması daha kolay tür isimleri verilebilir. *typedef* bildirimleri için aşağıda verilen basit kural kolaylık sağlar: *typedef* anahtar sözcüğü, her tür bildirimin önüne gelebilir. *typedef* anahtar sözcüğü bir bildirimin önüne geldiğinde, *typedef* kullanılmamış olsaydı değişken ismi olacak isimler, *typedef* anahtar sözcüğü eklendiğinde artık ilgili türün ismi olur. Örneğin:

```
char *pstr;
```

biçiminde bildirilen *pstr char** türünden bir değişkendir.

```
typedef char *pstr;
```

Yukarıdaki deyim ise bir *typedef* bildirimidir. Artık *pstr* derleyici tarafından bir tür ismi olarak ele alınır. *pstr*, *char* türden bir adres türünün başka bir ismi olarak, geçerli bir türdür. Yani yukarıdaki *typedef* bildiriminden sonra

```
pstr p;
```

gibi bir tanımlama yapılabilir. Bu tanımlama

```
char *p
```

ile aynı anlama gelir.
Bir *typedef* bildirimi ile elde edilen fayda her zaman *#define* önişlemci komutuyla sağlanamayabilir:

```
#define pstr char*
```

gibi bir önişlemci ismi tanımlaması yapıldığında, önişlemci *pstr* ismini gördüğü yerde bunun yerine *char* * atomlarını yerleştirir.

```
char *str;
```

gibi bir bildirimin

```
pstr str;
```

olarak yazılmasında bir hata söz konusu olmaz. Önişlemci *pstr* yerine *char* * yerleştirdiğinde derleyiciye giden kod

```
char *str
```

haline gelir.

```
char *p1, *p2, *p3;
```

önişlemci *#define* komutunun kullanılarak yukarıdaki gibi bir bildirimin yapılmak istensin.

```
pstr p1, p2, p3;
```

yazıldığında, önişlemci yer değiştirme işlemini yaptıktan sonra derleyiciye verilen kod aşağıdaki biçime dönüşür:

```
char *p1, p2, p3;
```

Bu tanımlama yapılmak istenen tanımlamaya eşdeğer değildir. Yukarıdaki bildirimde yalnızca *p1 bir* gösterici değişkendir. *p2* ile *p3* gösterici değişkenler değildir. *char* türden değişkenlerdir.
Bir diziye ilişkin de yeni tür ismi bildirimi yapılabilir:

```
char isimdizi[20];
```

Yukarıdaki deyim ile *isimdizi* isimli *char* türden *20* elemanlı bir dizi tanımlanmış olur.

```
typedef char isimdizi[20];
```

Yukarıdaki deyim ise bir bildirimdir. Bu bildirim ile *isimdizi* artık *20* elemanlı *int* türden dizilerin tür ismidir. Bu *typedef* bildiriminden sonra eğer

```
isimdizi a, b, c;
```

Gibi bir tanımlama yapılırsa, artık *a, b, c* her biri *20* elemanlı *char* türden dizilerdir.

Bir *typedef* bildirimi ile birden fazla tür ismi yaratılabilir:

```
typedef unsigned int WORD, UINT;
```

Yukarıdaki bildirim deyiminden sonra, hem *WORD* hem de *UINT*, *unsigned int* türünün yerine geçen yeni tür isimleridir:

```
WORD x, y;
UINT k, l;
```

Artık *x, y, k, l unsigned int* türden değişkenlerdir.

10 elemanlı *char* türden gösterici dizisi için tür ismi bildirimi şöyle yapılabilir:

```
typedef char *PSTRARRAY[10];
```

Bu bildirim deyiminden sonra

```
PSTRARRAY s;
```

ile

```
char *s[10];
```

tamamen aynı anlama gelir.
Bir tür ismi başka tür isimlerinin bildiriminde de kullanılabilir:

```
typedef unsigned int WORD;
typedef WORD UINT;
```

Yeni oluşturulan tür isimleri, okunabilirlik açısından ya tamamen büyük harflerden seçilir, ya da bu isimlerin yalnızca ilk harfleri büyük harf yapılır.

typedef Bildirimlerinin Yapılar İçin Kullanımı

Bir yapı bildirimiyle yeni bir tür yaratılmış olur. Bu tür önce derleyiciye tanıtıldıktan sonra, bu türe ilişkin değişkenler tanımlanabilir:

```
struct Data {
    int a, b, c;
};
```

Yukarıdaki bildirimle yeni bir veri türü yaratılıyor. C dilinde bu veri türünün ismi *struct Data*'dır. Türün ismi *Data* değildir. Yani bu veri türünden bir nesne tanımlamak istendiğinde tür ismi olarak *struct Data* yazılmalıdır.
[C++ dilinde yapı isimleri (*structure tags*) aynı zamanda türün de ismidir. *struct* anahtar sözcüğü olmadan kullanıldığında da bu türün ismi olarak derleyici tarafından kabul görür.]

Yukarıdaki bildirimden sonra örneğin bir tanımlama yapılacak olsa

```
struct Data d;
```

biçiminde yapılmalıdır. C dilinde bu tanımlamanın

```
Data d;
```

biçiminde yapılması geçersizdir. Oysa C++ dilinde bu durum geçerlidir.
İşte *struct* anahtar sözcüğünün, yapı nesnesi tanımlamalarında yapı isminden önce kullanılma zorunluluğunu ortadan kaldırmak için programcılar, *typedef* bildirimiyle kendi bildirdikleri yapı türlerine ilişkin yeni tür isimleri oluştururlar.

Bir yapı türüne ilişkin yeni bir tür isminin oluşturulması üç ayrı biçimde yapılabilir.

1. Önce yapı bildirimi yapılır. Daha sonra bildirilen yapı türü için ayrı bir *typedef* bildirimi yapılır:

```
struct tagPerson {
     char name[30];
     int no;
};

typedef struct tagPerson Person;
Person per = {"Necati Ergin", 123};
```

Yukarıdaki örnekte önce *struct tagPerson* isimli bir tür yaratılıyor daha sonra *typedef* bildirimiyle *struct tagPerson* türüne yeni bir isim olarak *Person* ismi veriliyor. *typedef* bildiriminden sonra, hem *struct tagPerson* hem de *Person* isimleri, tür isimleri olarak kullanılabilir.

2. *typedef* bildirimi ile yapı bildirimi tek bir bildirim biçiminde birleştirilebilir:

```
typedef struct tagPerson {
     char name[30];
     int no;
} Person;

Person per;
```

Daha önce verilen kuralı anımsayın: Bu bildirimin başında *typedef* anahtar sözcüğü olmasaydı *Person* ismi *struct tagPerson* türünden bir nesnenin ismi olurdu, değil mi?
Yukarıdaki örnekte hem ismi *struct tagPerson* olan bir yapı bildiriliyor hem de *typedef* bildirimiyle bu yapıya yeni bir isim olarak *Person* ismi veriliyor. İlgili bildirimden sonra hem *struct tagPerson* hem de *Person* isimleri, tür isimleri olarak kullanılabilir.

3. Yapı ismi *(structure tag)* kullanılmadan yapılan bir yapı bildirimi ile *typedef* bildirimi birleştirilebilir:

```
typedef struct  {
     char name[30];
     int no;
}Person;

Person y;
```

Yukarıdaki örnekte yaratılan türün tek bir ismi vardır. Bu isim *Person* ismidir. Bu tür kullanılmak istendiğinde artık *struct* anahtar sözcüğü kullanılamaz.
Programcıların çoğu, yapı isimleriyle *(structure tag), typedef* bildirimiyle oluşturulacak tür isimleri için farklı isimler bulmak yerine birkaç karakter kullanarak aralarında ilişki kurarlar. Çok kullanılan kalıplardan biri, yapı isminin başına bir *"alt tire"* karakteri konularak tür isminden ayrılmasıdır:

```
typedef struct _Employee {
    int no;
    char name[30];
    double fee;
} Employee;
```

Şüphesiz aşağıdaki gibi bir bildirim de hata oluşturmazdı:

```
typedef struct Employee {
    int no;
    char name[30];
    double fee;
}Employee;
```

Türün eski ismi *Employee* değil *struct Employee* dir. Yani *struct Employee* türüne *typedef* bildirimiyle *Employee* ismi verilmiştir. Bu durum bir sakınca oluşturmaz.
Windows API programlarında yapı bildirimlerine ilişkin *typedef* bildirimlerinde aşağıdaki gibi bir kalıp da kullanılır:

```
typedef struct tagEmployee {
    int no;
    char name[30];
    double fee;
} Employee;
```

Yapı ismi *tag* önekiyle başlatılıyor, seçilen *typedef* isminde *tag* öneki kaldırılıyor.
Bir yapı bildirimi söz konusu olduğunda, yapı türüne isim vermek yerine o yapıya ilişkin adres türüne yeni isim verilebilir. Aşağıdaki örneği inceleyin:

```
struct {
    int a, b, c;
}*Hdata;
```

Yukarıdaki bildirimde, bildirimi yapılan yapının bir ismi yoktur. Ancak *typedef* bildirimiyle bu yapıya ilişkin adres türüne *Hdata* ismi veriliyor. Bu durumda *Unix, Windows* sistemlerinde

```
sizeof(Hdata)
```

ifadesinin değeri 4

```
sizeof(*Hdata)
```

ifadesinin değeri *12* dir.

Standart Başlık Dosyalarında Bulunan Bazı typedef Bildirimleri

Bazı standart işlevlerin bildiriminde, doğrudan doğal bir veri türünü kullanmak yerine daha önceden belirlenmiş bazı *typedef* isimleri kullanılır. Doğal türler sistemden sisteme farklı uzunlukta olabileceğinden, bazı işlevlerin ara yüzünde doğal tür isimlerini kullanmak yerine bir *typedef* isminin kullanılması, derleyiciyi yazanlara daha büyük bir esneklik sağlar. Örneğin standart *malloc* işlevinin *stdlib.h* başlık dosyası içindeki bildirimi aşağıdaki gibidir:

```
void *malloc(size_t ntypes);
```

Bu bildirimde *size_t* isminin bir tür ismi olarak kullanıldığını görüyorsunuz.
Derleyicilerin çoğunda bu tür isminin bildirimi *stddef.h, stdio.h, stdlib.h* başlık dosyalarında aşağıdaki gibi yapılır:

```
typedef unsigned int size_t;
```

Bu tür gerçekte ne olduğu derleyicileri yazanlara bırakılmış olan bir türdür. Derleyiciler ilgili başlık dosyalarında yapılan *typedef* bildirimleriyle işaretsiz bir tamsayı türlerinden birine *size_t* ismini verir. Standartlara göre *size_t* türü *sizeof* işlecinin ürettiği değerin türüdür. *ANSI* standartlarında bir çok işlevin bildiriminde *size_t* türü geçer. Örneğin *strlen* işlevinin gerçek bildirimi, *string.h* başlık dosyasının içinde

```
size_t strlen(const char *);
```

olarak yapılır.
Yani *malloc* işlevinin parametre değişkeni ya da *strlen* işlevinin geri dönüş değeri *size_t* türündendir. Bu türün gerçekte ne olduğu derleyicileri yazanlara bırakılır. Ancak hemen hemen bütün derleyicilerde *size_t* türü *unsigned int* türünün *typedef* ismi olarak belirlenir.

size_t türü gibi aslında ne olduğu derleyiciye bırakılmış olan, yani derleyici yazanların ilgili başlık dosyalarında *typedef* bildirimlerini yapacakları başka tür isimleri de C standartları tarafından tanımlanmıştır. Bu türlerden bazıları şunlardır:

time_t : standart *time* işlevinin geri dönüş değerinin türüdür. *time.h* başlık dosyası içinde, derleyiciyi yazanlar herhangi bir temel veri türüne *typedef* bildirimiyle bu ismi verirler. Bir zorunluluk olmasa da, *time_t* derleyicilerin hemen hepsinde *long* türünün *typedef* ismi olarak seçilir.

clock_t : standart *clock* işlevinin geri dönüş değerinin türüdür. *time.h* başlık dosyası içinde, derleyiciyi yazanlar herhangi bir temel veri türüne *typedef* bildirimiyle bu ismi verirler. Derleyicilerin hemen hepsinde *long* türünün *typedef* ismi olarak seçilir.

ptrdiff_t : Bir adres bilgisinden başka bir adres bilgisinin çıkartılmasıyla bir tamsayı elde edildiğini biliyorsunuz. İki adresin birbirinden farkı *ptrdiff_t* türündendir. Zaten bu türün ismi de *pointer difference* sözcüklerinden gelir. Bu tür işaretli tamsayı türlerinden biri olmak zorundadır. Derleyicilerin hemen hepsinde *int* türünün *typedef* ismi olarak seçilir.

fpos_t: *stdio.h* başlık dosyası içinde bildirilen *fgetpos* ve *fsetpos* işlevlerinin parametre değişkeni olan göstericilerin türüdür.

wchar_t: Bu türün bildirimin *stdlib.h* ve *stddef.h* başlık dosyalarının içinde yapılmıştır. Sistem tarafından desteklenen yöredeki (*locale)* en büyük genişletilmiş karakter setini temsil edebilecek bir türdür. Örneğin genişletilmiş karakter setinin tüm değerleri iki *byte'lık* bir alanda ifade edilebiliyorsa *wchar_t* türü *unsigned short int* türünün *typedef* ismi olabilir. *wchar_t* ismi *wide character* sözcüklerinden gelir. Geniş karakterler, geniş karakter değişmezleri, çoklu karakterler gibi noktalara "yerelleştirme" konusunda ayrıntılı olarak değinilecek.

div_t ve *ldiv_t*: Bu türlerin bildirimleri *stdlib.h* isimli başlık dosyasında yapılmıştır. *div_t stdlib.h* içinde bildirilen standart *div* işlevinin geri dönüş değeri olan yapı türüdür. *ldiv_t* türü de yine *stdlib.h* başlık dosyası içinde bildirilen *ldiv* isimli işlevin geri dönüş değeri olan yapı türüdür.

typedef ile Bildirimi Yapılan Tür İsimlerinin Bilinirlik Alanları

typedef bildirimleri için de bilinirlik alanı kuralları geçerlidir. Bir blok içinde tanımlanan bir *typedef* ismi, o blok dışında bilinmez.

```
void func()
{
    typedef int  WORD;
}

void foo()
{
    WORD x;   /*Geçersiz*/
}
```

Yukarıdaki programda

```
WORD x;
```

deyimi geçersizdir. Zira *WORD* türü yalnızca *func* işlevinin ana bloğu içinde bilinen bir veri türüdür. Bu bloğun dışında bilinmez.

C dilinde blok içinde yapılan bildirimlerin blokların başında yapılması zorunludur. *typedef* bildirimiyle blok içinde yapılan yeni tür ismi bildirimleri de blokların başında yapılmak zorundadır.
Ancak hemen her zaman *typedef* bildirimleri global düzeyde yapılır. Uygulamalarda *typedef* bildirimleri genellikle, ya kaynak dosyanın başında ya da başlık dosyaları içinde yapılır. Çünkü *typedef* isimleri çoğunlukla dışarıya hizmet veren bir modülün arayüzüne aittir.
Aynı *typedef* ismi farklı iki türün yeni ismi olarak bildirilemez:

```
typedef int WORD;
/*...*/
typedef long WORD; /* Geçersiz! */
```

typedef Bildirimlerinin Amacı Nedir

Okunabilirliği artırmak için *typedef* bildirimleri yapılabilir. Bazı türlere onların kullanım amaçlarına uygun isimler verilirse kaynak kod daha kolay okunur daha kolay anlamlandırılır. Örneğin *char* türü genelde karakter değişmezlerinin atandığı bir türdür. *char* türü yalnızca bir *byte'lık* bir veri olarak kullanılacaksa, yani yazı işlemlerinde kullanılmayacak ise aşağıdaki gibi bir tür tanımlaması yerinde olur:

```
typedef char BYTE;
/*...*/
BYTE x;
```

C89 standartlarında *bool* türü doğal bir veri türü değildir. Ancak bir *typedef* bildirimiyle *int* türüne *bool* ismi verilebilir:

```
typedef int bool;
```

typedef bildirimleri yazım kolaylığı sağlar. Karmaşık pek çok tür ismi *typedef* bildirimi kullanılarak kolay bir biçimde yazılabilir. Programı okuyanlar tür bilgisine karşılık gelen karmaşık atomlar yerine onu temsil eden yalın bir isimle karşılaşırlar. Aşağıda önce bir işlev adresine ilişkin türe *typedef* bildirimiyle yeni bir isim veriliyor, daha sonra bu türden bir nesne tanımlanıyor:

```
typedef struct Person * *Fpper)(struct Person *, int);
/*...*/
Fpper  fptr;
```

3. *typedef* bildirimleri bazen de taşınabilirliği artırmak amacıyla kullanılır. *typedef* bildirimlerinin kullanılmasıyla, yazılan işlevlere ilişkin veri yapıları değişse bile kaynak

programın değişmesi gerekmez. Örneğin, bir kütüphanede birtakım işlevlerin geri dönüş değerleri *unsigned int* türünden olsun. Daha sonraki uygulamalarda bu işlevlerin geri dönüş değerlerinin türünün *unsigned long* türü olarak değiştirildiğini düşünelim. Eğer programcı bu bu işlevlere ilişkin kodlarda *typedef* bildirimleri kullanmışsa, daha önce yazdığı kodları değiştirmesine gerek kalmaz, yalnızca *typedef* bildirimlerini değiştirmesi yeterli olur. Örneğin:

```
typedef unsigned int HANDLE;
/***/
HANDLE hnd;
hnd = GetHandle();
```

Burada *GetHandle* işlevinin geri dönüş değerinin türü sonraki uyarlamalarda değişerek *unsigned long* yapılmış olsun. Yalnızca *typedef* bildiriminin değiştirilmesi yeterli olur:

```
typedef unsigned long HANDLE;
```

Bir C programında değerleri *0* ile *50000* arasında değişebilecek olan sayaç amacıyla kullanılacak değişkenler kullanılmak istensin. Bu amaç için *long int* türü seçilebilir, çünkü *long int* türü *Windows* ya da *Unix* sistemlerinde *2.147.483.647* ye kadar değerleri tutabilir. Ama *long int* türü yerine *int* türünü kullanmak, aritmetik işlemlerin daha hızlı yapılabilmesi açısından tercih edilebilir. Ayrıca *int* türden olan değişkenler bazı sistemlerde bellekte daha az yer kaplayabilir.
int türünü kullanmak yerine bu amaç için yeni bir tür ismi yaratılabilir:

```
typedef int SAYAC;
SAYAC a, b, c;
```

Kaynak kodun *int* türünün *16* bit uzunluğunda olduğu bir sistemde derlenmesi durumunda *typedef* bildirimi değiştirilebilir:

```
typedef long  SAYAC;
```

Bu teknikle taşınabilirliğe ilişkin bütün sorunların çözülmüş olacağı düşünülmemelidir. Örneğin *SAYAC* türünden bir değişken *printf* ya da *scanf* işlevlerine yapılan çağrılarda argüman olan ifadenin türü olarak kullanılmış olabilir:

```
#include <stdio.h>

typedef int SAYAC;

int main()
{
    SAYAC a, b, c;
    /***/
    scanf("%d%d%d", &a, &b. &c);
    /***/
    printf("%d %d %d", a, b, c);
    /***/
    return 0;
}
```

Yukarıdaki deyimlerde *a, b, c* değişkenleri *SAYAC* türünden yani *int* türden tanımlanıyor. *printf* ile *scanf* işlevlerine yapılan çağrılarda da bu değişkenlere ilişkin format karakterleri olarak *%d* seçiliyor. Ancak *SAYAC* türünün *long* türü olarak değiştirilmesi durumunda *printf* ve *scanf* işlevlerinde bu türden değişkenlerin yazdırılmasında kullanılan format karakterlerinin de *%d* yerine *%ld* olarak değiştirilmesi gerekir.

typedef İle Adres Türlerine İsim Verilmesi

typedef bildirimi ile gösterici türlerine yeni isim verilmesinde dikkat edilmesi gereken bir nokta vardır. Tür niteleyicileri konusunda ele alındığı gibi C dilinde aşağıdaki gibi bir tanımlama yapıldığında

```
const int *ptr;
```

ptr gösterdiği yer *const* olan bir göstericidir. Yani *ptr* değişkeninin gösterdiği yerdeki nesne değiştirilemez:

```
*ptr = 10;
```

gibi bir atama geçersizdir. Ancak tanımlama

```
int *const ptr;
```

biçiminde yapılırsa, *ptr* kendisi *const* olan bir göstericidir. *ptr* göstericisinin gösterdiği nesnenin değeri değiştirilebilir, ama *ptr* göstericisinin içindeki adres değiştirilemez, yani

```
ptr = (int *) 0x1F00;
```

gibi bir atama yapılması geçersizdir.

```
typedef int *IPTR;
```

gibi bir bildirimden sonra

```
const IPTR p;
```

biçiminde bir tanımlama yapıldığında, *p* göstericisinin değeri değiştirilemez, *p* göstericisinin gösterdiği yerdeki nesnenin değeri değiştirilebilir. Yani **p* nesnesine atama yapılabilir. Başka bir deyişle

```
const IPTR ptr;
```

deyimi ile

```
int *const ptr;
```

deyimi eşdeğerdir.
Windows işletim sistemi altında çalışacak C ya da C++ programlarının yazılmasında *typedef* bildirimleri sıklıkla kullanılır. *windows.h* isimli başlık dosyasında temel veri türlerinin çoğuna *typedef* bildirimleriyle yeni isimler verilmiştir. *Windows API* programlamada *windows.h* başlık dosyası kaynak koda eklenmelidir. Bu dosyanın içinde *API* işlevlerinin bildirimleri, çeşitli yapı bildirimleri, *typedef* isimleri, önemli simgesel değişmezler bulunur.

windows.h İçinde Tanımlanan typedef İsimleri

```
typedef  int BOOL;
```

Bu türle ilişkili iki simgesel değişmez de tanımlanmıştır.

```
#define FALSE       0
#define TRUE        1
```

BOOL türü, özellikle işlevlerin geri dönüş değerlerinde kullanılır. Bu durum işlevin başarılıysa *0* dışı bir değere, başarısızsa 0 değerine geri döneceği anlamına gelir. Başarı kontrolü, *1* değeriyle karşılaştırılarak yapılmamalıdır.
Aşağıdaki *typedef* isimleri, işaretsiz *1* byte, *2* byte ve *4* byte tam sayıları simgeler.

```
typedef unsigned char BYTE;
typedef unsigned short WORD;
typedef unsigned long int DWORD;
typedef unsigned int UINT;
```

Göstericilere ilişkin *typedef* isimleri *P* harfiyle başlar. *LP* uzak göstericileri belirtmek için ön ek olarak kullanılır. *Win16* sistemlerinde uzak ve yakın gösterici kavramları vardı. Dolayısıyla o zamanlar, *P* önekli göstericiler yakın göstericileri, *LP* önekli göstericiler ise uzak göstericileri temsil ediyordu. Fakat *Win32* sistemlerinde yakın ve uzak gösterici kavramları yoktur. Bu durumda, *P* önekli göstericilerle *LP* önekli göstericiler arasında hiçbir fark yoktur. Ancak, *Win16*'daki alışkanlıkla hala *LP* önekli *typedef* isimleri kullanılır. *Windows.h* içinde her ihtimale karşı -*Win16* programları çalışabilsin diye- *near* ve *far* sözcükleri aşağıdaki gibi silinmiştir.

```
#define far
#define near
typedef char near *PSTR;
typedef char far *LPSTR;
```

PSTR ya da *LPSTR Win32* sistemlerinde tamamen aynı anlama gelir ve *char** türünü belirtir.

```
typedef char *PSTR;
typedef char *LPSTR;
```

Göstericilerde *const*'luk *P* ya da LP'den sonra *C* ekiyle belirtilir. Örneğin;

```
typedef const char *PCSTR;
typedef const char *LPCSTR;
```

Klasik *typedef* isimlerinin hepsinin gösterici karşılıkları da vardır. Bütün gösterici türleri, *Win16* uyumlu olması için *P ve L*P önekleriyle ayrıca bildirilmiştir.

```
typedef BYTE *PBYTE;
typedef WORD *PWORD;
typedef const BYTE *PCBYTE;
typedef const DWORD *LPCDWORD;
```

C'nin doğal türlerinin hepsinin büyük harf normal, adres ve *const* adres biçimleri vardır.

```
typedef long LONG;
typedef int INT;
typedef char CHAR;
```

Windows programlamada *H* ile başlayan, handle olarak kullanılan pek çok *typedef* ismi vardır. Bu *typedef* isimlerinin hepsi *void* * türündendir. Örneğin:

```
typedef  void   *HWND;
typedef  void   *HICON;
```

Tarih ve Zaman İle İlgili İşlem Yapan Standart İşlevler

Standart başlık dosyalarından *time.h* içinde bildirilen, tarih ve zaman bilgileriyle ilgili faydalı işler yapmak için tanımlanan bazı standart işlevler vardır. Bu işlevlerin bazılarının parametre değişkenleri ya da geri dönüş değerleri bir yapı türünden adres bilgileridir. Aşağıda bu işlevler açıklanıyor:

time İşlevi

```
time_t time (time_t *timer);
```

İşlevin geri dönüş değeri standart bir *typedef* türü olan *time_t* türüdür. Derleyicilerin çoğunun bu türü *long* türünün *typedef* ismi olarak bildirir. İşlevin parametre değişkeni de bu türden bir adrestir.
İşlev adresi gönderilen nesneye belirli bir tarihten (çoğu sistemde bu tarih *01.01.1970* tarihidir) işlev çağrısına kadar geçen saniye sayısı değerini yazar. Bu değer standartlarda *"takvim zamanı" ("calender time")* olarak geçer. Tarih ve zaman üzerinde işlem yapan diğer bazı işlevler işlerini yapabilmek için bu değere gereksinim duyar.

İşlev, bu değeri aynı zamanda geri dönüş değeri olarak da dışarıya iletir. Eğer *takvim* zamanı bilgisi elde edilemiyorsa işlev *(time_t)-1* değerine geri döner.
Eğer işleve *NULL* adresi gönderilirse, işlev bu değeri özel bir ileti olarak algılar, hiçbir nesneye yazma yapmadan, saniye bilgisini yalnız geri dönüş değeri ile dışarıya aktarır.
Aşağıda işlevin kullanılmasıyla ilgili basit bir kod parçası veriliyor:

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

int main()
{
    time_t timer1, timer2, timer3;

    timer1 = time(&timer2);
    printf("devam etmek için bir tuşa basın : ");
    getchar();
    timer3 = time(NULL);

    printf("timer1 = %ld\n", (long)timer1);
    printf("timer2= %ld\n", (long)timer2);
    printf("timer3= %ld\n", (long)timer3);

    return 0;
}
```

localtime İşlevi

01.01.1970'den geçen saniye sayısı yani takvim zamanı doğrudan kullanılabilecek bir zaman bilgisi değildir. *localtime* işlevi bu bilgiyi alarak faydalı parçalara ayırır. İşlevin *time.h* başlık dosyası içindeki bildirimi aşağıdaki gibidir:

```
struct tm *localtime(const time_t *timer);
```

İşlevin geri dönüş değeri, *time.h* içinde bildirilen bir yapı olan *struct tm* türünden bir adrestir. Bu yapı aşağıdaki gibi bildirilmiştir:

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

Yapının ismi *tm* önekiyle başlayan tüm elemanları *int* türdendir. Her bir eleman tarih ya da zaman ile ilgili faydalı bir veriyi tutar. Aşağıda yapının elemanları hakkında bilgi veriliyor:

tm_sec : saniye değeri (0 - 60)
tm_min : dakika değeri (0 - 60)
tm_hour : saat değeri (0 – 24)
tm_mday : Ayın günü (1 – 31)
tm_mon : Ay değeri (0 Ocak, 1 Şubat, 2 Mart…)
tm_year : Yıl değerinin 1900 eksiği
tm_wday : Haftanın günü (0 Pazar, 1 Pazartesi, 2 Salı…)
tm_yday : Yılın günü (1 Ocak için 0)
tm_isdst :Gün ışığı tasarruf modu ile ilgili bilgi. Bu elemanın değerinin pozitif ise tasarruf modunda olunduğu bilgisi iletilmiş olur. Bu değer *0* ise tasarruf modu değildir. Elemanın değerinin negatif olması durumunda bu bilgi elde edilemiyor demektir.

struct tm yapısı ile tutulan zaman bilgisine "ayrıştırılmış zaman bilgisi" *(broken-down time)* denir.
localtime işlevi statik ömürlü bir *struct tm* nesnesinin adresi ile geri döner. İşlevin geri dönüş değeri olan adresteki yapı nesnesinin değeri kullanılmadan ya da başka bir nesneye aktarılmadan, işlev bir kez daha çağrılırsa, daha önceki çağrı ile ilgili olarak üretilen değerin üstüne yazılmış olur. İşlevin parametre değişkeni *time_t* türünden bir nesneyi gösteren göstericidir. İşlev adresini aldığı bu nesneden takvim zamanı bilgisini alır. Aşağıda *localtime* işlevini kullanan örnek bir kod veriliyor:

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

char *months[12] = {"Ocak", "Subat", "Mart", "Nisan", "Mayis", "Haziran",
"Temmuz", "Agustos", "Eylul","Ekim", "Kasim", "Aralik"};
char *days[7] = {"Pazar","Pazartesi", "Sali", "Carsamba", "Persembe",
"Cuma", "Cumartesi"};

int main()
{
    time_t timer;
    struct tm *tptr;

    time(&timer);
    tptr = localtime(&timer);
    printf("Tarih : %02d %s %04d %s\n",   tptr->tm_mday,months[tptr-
>tm_mon],  ptr->tm_year + 1900, days[tptr-   >tm_wday]);
    printf("Saat:%02d:%02d:%02d\n", tptr->tm_hour, tptr->tm_min,
    tptr->tm_sec);
    printf("bugun %d yilinin %d. gunu\n", tptr->tm_year + 1900,
    tptr->tm_yday);
    if (tptr->tm_isdst < 0)
```

```
        printf("gun isigi tasarruf modu bilgisi elde edilemiyor!\n");
    else if (tptr->tm_isdst > 0)
        printf("gun isigi tasarruf modundayiz!\n");
    else
        printf("gun isigi tasarruf modunda degiliz!\n");

    return 0;
}
```

ctime İşlevi

Bu işlev, takvim zamanı bilgisini girdi olarak alarak bu bilgiyi bir yazıya dönüştürür. İşlevin bildirimi aşağıdaki gibidir:

```
char *ctime(const time_t *time);
```

İşlevin parametre değişkeni *time_t* türünden bir adrestir. İşlev adresini aldığı nesneden takvim zamanı bilgisini okur. İşlevin geri dönüş değeri *26* karakterlik bir yazının başlangıç adresidir. Bu yazı ayrıştırılmış zaman bilgilerini içeren özel olarak formatlanmış bir yazıdır:

F	r	i		F	e	b		2	3		1	2	:	2	5	:	5	4		2	0	0	4	\n	'\0'

```
Fri Feb 23 12:25:12 2004
```

İşlev statik ömürlü bir dizinin adresiyle geri döner. İşlevin geri dönüş değeri olan adresteki yazı kullanılmadan ya da başka bir diziye aktarılmadan, işlev bir kez daha çağrılırsa, daha önceki çağrı ile ilgili olarak üretilen yazının üstüne yazılmış olur. Aşağıda işlevin kullanımına ilişkin örnek bir kod parçası yer alıyor:

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

int main()
{
    char *ptr;
    time_t timer;

    timer = time(NULL);
    ptr = ctime(&timer);
    printf("%s", ptr);

    return 0;
}
```

asctime İşlevi

ctime işlevinin yaptığı işin aynısını yapar. Ancak girdi olarak takvim zamanını değil ayrıştırılmış zaman bilgisini alır:

```
char *asctime (const struct tm *tblock);
```

İşlevin parametre değişkeni *struct tm* yapısı türünden bir adrestir. İşlev, adresini aldığı nesneden, ayrıştırılmış zaman bilgilerini okur. Bu bilgiyi bir yazıya dönüştürerek, özel bir formatta *26* karakterlik statik ömürlü bir dizi içine yazar. İşlev, ilgili yazının başlangıç adresine geri döner. İşlevin geri dönüş değeri olan adresteki yazı kullanılmadan ya da başka bir diziye aktarılmadan, işlev bir kez daha çağrılırsa, daha önceki çağrı ile ilgili olarak üretilen yazının üstüne yazılmış olur. Aşağıda işlevin kullanımına ilişkin örnek bir kod parçası yer alıyor:

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

int main()
{
    time_t timer;

    timer = time(NULL);
    printf("%s", asctime(localtime(&timer)));
    return 0;
}
```

Yukarıdaki kod parçasında, *localtime* işlevinin geri dönüş değeri olan *struct tm* türünden adres doğrudan *asctime* işlevine argüman olarak gönderiliyor. *asctime* işlevinin geri dönüş değeri olan adres de *printf* işlevine geçiliyor.

clock İşlevi

İşlevin geri dönüş değeri *time.h* başlık dosyası içinde bildirilen standart bir *typedef türü* olan *clock_t* türüdür. Derleyicilerin hemen hepsinde bu tür *long* türünün *typedef* ismi olarak bildirilir:

```
clock_t clock(void);
```

İşlev, programın çalışmaya başlamasıyla işlev çağrısına kadar geçen süreye geri döner. Ancak işlevin geri dönüş değerine ilişkin birim saniye değil işlemcinin zaman devresinin tick sayısıdır. İşlevin geri dönüş değerini saniyeye dönüştürmek için, geri dönüş değeri işlemcinin saniyedeki tick sayısına bölünmelidir. Zaman devresinin bir saniyedeki *tick* sayisi *time.h* içinde *CLOCKS_PER_SEC* isimli bir simgesel değişmez olarak tanımlanmıştır.

```
#define    CLOCKS_PER_SEC    1000
```

Kodun taşınabilirliği açısından bu simgesel değişmez kullanılmalıdır.
Derleyicilerin çoğu daha kısa bir simgesel değişmezin kullanılmasını sağlamak üzere *CLK_TCK* isimli bir simgesel değişmez daha tanımlar:

```
#define    CLK_TCK               CLOCKS_PER_SEC
```

Ancak taşınabilirlik açısından *CLOCKS_PER_SEC* simgesel değişmezi kullanılmalıdır.
clock işleviyle ilgili aşağıdaki örnek programı inceleyin:

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

int main()
{
    clock_t clock1, clock2;
    long k;

    clock1 = clock();
    for (k = 1; k < 10000000; ++k)
        sqrt(sqrt(k) + sqrt(k + 1));
    clock2 = clock();
    printf("dongu %lf saniye surdu!\n", (double)(clock2 - clock1) /
CLK_TCK);

    return 0;
}
```

Aşağıda *clock* işlevinden faydalanan bir geciktirme işlevi yazılıyor:

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

#define    SENSITIVITY        0.1

void delay(double sec)
{
    double total = 0.;
    clock_t tstart = clock();
    clock_t tend;
    double duration;

    for (;;) {
        tend = clock();
        duration = (double)(tend - tstart) / CLOCKS_PER_SEC;
        if  (fabs(duration - sec) < SENSITIVITY)
            return;
    }
}
```

difftime İşlevi

Standart *difftime* işlevi takvim zamanı cinsinden verilen iki zaman bilgisi arasındaki saniye farkını bulmak için kullanılabilir. İşlev bildirimi aşağıdaki gibidir:

```
double difftime(time_t timer2, time_t timer1);
```

İşlevin geri dönüş değeri *timer1* değeri ile *timer2* değeri arasında geçen saniye sayısıdır.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
int main()
{
    time_t start,finish;
    long i;
    double result, elapsed_time;

    printf("20000000 kez kare kok aliniyor.\n");
    time(&start);

    for (i = 1; i <= 20000000; i++)
            result = sqrt(i);

    time(&finish);
    elapsed_time = difftime(finish, start);
    printf("\nToplam sure = %lf saniye.\n", elapsed_time);
    return 0;
}
```

mktime İşlevi

Bu standart işlev ayrıştırılmış zaman bilgisini (*broken-down time*) takvim zamanına (*calender time*) dönüştürür. İşlevin bildirimi aşağıdaki gibidir:

```
time_t mktime(struct tm *tptr):
```

işlevin parametresi ayrıştırılmış zaman bilgisinin tutan *struct tm* türünden nesnenin adresidir. İşlevin parametresinde kullanılan adresin *const* olmadığına dikkat edin. Buradan işlevin bu adrese yazma yapacağı sonucunu çıkarabilirsiniz. İşlevin geri dönüş değeri takvim zamanı değeridir. Eğer ilgili sistemde takvim zamanı değeri elde edilemiyorsa işlev *-1* değerine geri döner.
İşlevin şöyle bir yan etkisi de vardır. Eğer parametresine aldığı adresteki yapı nesnesinin elemanları olması gereken değerleri aşıyorsa, *mktime* işlevi değerlerin fazla kısmını bir sonraki elemana ekler.
Bu amaçla önce *tm_sec* isimli elemana bakılır. Buradaki fazlalıklar *tm_min* elemanına verilir. Burada da bir fazlalık oluşur ise sırasıyla yapı nesnesinin *tm_hour*, *tm_mday*, *tm_mon* ve *tm_year* elemanları değiştirilir. Bu elemanları değeri alındıktam sonra yapı nesnesinin *tm_wday* ve *tm_yday* isimli elemanları sahip olması gereken değerlere getirilir.

Aşağıdaki kod, işlevin bu özelliğini kullanarak bir tarihin ne gününe geldiğini buluyor:

```
#include <stdio.h>
#include <time.h>
int main()
{
    struct tm t;
    const char *days[] = {"Pazar", "Pazartesi", "Sali", "Carsamba",
"Persembe", "Cuma","Cumartesi", "bilinmiyor"};
    t.tm_mday = 11;
    t.tm_mon = 2;
    t.tm_year = 2005 - 1900;
    t.tm_sec = 1;
    t.tm_min = 0;
    t.tm_hour = 0;
    t.tm_isdst = -1;

    if (mktime(&t) == 1)
        t.tm_wday = 7;
    printf("gun = %s\n", days[t.tm_wday]);

    return 0;
}
```

Yapılarla İlgili Uygulamalar

Aşağıdaki programda bir tarih bilgisini tutmak amacıyla *Date* isimli bir yapı bildiriliyor. Bu yapı türünü kullanarak hizmet veren işlevler tanımlanıyor.

```
/********** date.h *************************/
typedef struct {
    int m_d;
    int m_m;
    int m_y;
}Date;

#define    YEAR_BASE  1000
#define    false      0
#define    true       1

typedef int bool;

void set_date(Date *ptr, int d, int m, int y);
void set_today(Date *ptr);
void set_random(Date *ptr);
void display_date(const Date *ptr);
void inc_date(Date *p);
void dec_date(Date *p);
Date ndays(const Date *p, int n);
int date_cmp(const Date *p1, const Date *p2);
int get_day(const Date *p);
int get_month(const Date *p);
int get_year(const Date *p);
int get_yearday(const Date *ptr);
int get_weekday(const Date *ptr);
bool isweekend(const Date *ptr);
bool isleap(int y);

/********** date.c *************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <assert.h>

#define    PUBLIC
#define    PRIVATE    static
#define    FACTOR      2

PRIVATE int daytabs[2][13] = {
{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
{0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
};

PRIVATE bool check_date(int d, int m, int y);
PRIVATE int totaldays(const Date *p);
PRIVATE Date revdate(int totaldays);

PRIVATE bool check_date(int d, int m, int y)
{
    if (y < YEAR_BASE)
        return false;

    if (m < 1 || m > 12)
        return false;
```

```
    if (d < 1 || d > daytabs[isleap(y)][m])
        return false;

    return true;
}
/*************************************************************************/
PRIVATE int totaldays(const Date *ptr)
{
    int sum = 0;
    int k;
    for (k = YEAR_BASE; k < ptr->m_y; ++k)
        sum += 365 + isleap(k);

    return sum + get_yearday(ptr);
}
/*************************************************************************/
PRIVATE Date revdate(int totaldays)
{
    Date ret_val;
    int val;
    int index;

    ret_val.m_y = YEAR_BASE;
    while (totaldays > (val = isleap(ret_val.m_y) + 365)) {
        totaldays -= val;
        ret_val.m_y++;
    }

    ret_val.m_m = 1;
    index = isleap(ret_val.m_y);
    while (totaldays > daytabs[index][ret_val.m_m])
        totaldays -= daytabs[index][ret_val.m_m++];

    ret_val.m_d = totaldays;
    return ret_val;
}
/*************************************************************************/
PUBLIC int get_yearday(const Date *ptr)
{
    int sum = ptr->m_d;
    int k;
    int index = isleap(ptr->m_y);
    for (k = 1; k < ptr->m_m; ++k)
        sum += daytabs[index][k];

    return sum;
}
/*************************************************************************/
PUBLIC int dayofweek(const Date *ptr)
{
    return totaldays(ptr) % 7;
}
/*************************************************************************/
PUBLIC bool isleap(int y)
{
    return y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
}
/*************************************************************************/
PUBLIC void set_date(Date *ptr, int d, int m, int y)
{
    assert(check_date(d, m, y));
```

```
    ptr->m_d = d;
    ptr->m_m = m;
    ptr->m_y = y;
}
/*************************************************************************/
PUBLIC void set_today(Date *ptr)
{
    time_t timer = time(NULL);
    struct tm *tptr = localtime(&timer);
    ptr->m_d = tptr->tm_mday;
    ptr->m_m = tptr->tm_mon + 1;
    ptr->m_y = tptr->tm_year + 1900;
}
/*************************************************************************/
PUBLIC void set_random(Date *ptr)
{
    ptr->m_y = rand() % 50 + 1960;
    ptr->m_m = rand() % 12 + 1;
    ptr->m_d = rand() % daytabs[isleap(ptr->m_y)][ptr->m_m] + 1;
}
/*************************************************************************/
PUBLIC void display_date(const Date *ptr)
{
    static const char *days[] = {"Pazar", "Pazartesi", "Sali", "Carsamba",
"Persembe", "Cuma",    "Cumartesi"};
    static const char *mons[] = {"", "Ocak", "Subat", "Mart", "Nisan",
"Mayis", "Haziran",    "Temmuz", "Agustos", "Eylul","Ekim", "Kasim",
"Aralik"};
    printf("%02d %s %4d %s\n", ptr->m_d, mons[ptr->m_m], ptr->m_y,
    days[get_weekday(ptr)]);
}
/*************************************************************************/
PUBLIC int date_cmp(const Date *p1, const Date *p2)
{
    if (p1->m_y != p2->m_y)
        return p1->m_y - p2->m_y;
    if (p1->m_m != p2->m_m)
        return p1->m_m - p2->m_m;

    return p1->m_d - p2->m_d;
}
/*************************************************************************/
PUBLIC void inc_date(Date *p)
{
    *p = revdate(totaldays(p) + 1);
}
/*************************************************************************/
PUBLIC void dec_date(Date *p)
{
    *p = revdate(totaldays(p) - 1);
}
/*************************************************************************/
PUBLIC Date ndays(const Date *p, int n)
{
    return revdate(totaldays(p) + n);
}
/*************************************************************************/
PUBLIC int get_weekday(const Date *ptr)
{
    return (totaldays(ptr) + FACTOR) % 7;
}
```

```
/*************************************************************************/
PUBLIC bool isweekend(const Date *ptr)
{
    int day = get_weekday(ptr);
    return day == 6 || day == 0;
}
/*************************************************************************/
int main()
{
    Date today;
    Date ndaysafter;

    set_today(&today);
    ndaysafter = ndays(&today, 10);

    while (date_cmp(&today, &ndaysafter)) {
        display_date(&today);
        inc_date(&today);
    }

    return 0;
}
```

Aşağıdaki programda bir tekli bağlı liste oluşturuluyor.

```
/********** datelist.h *************************/
#include "date.h"
typedef struct tagNode {
    Date date;
    struct tagNode *pnext;
}Node;

typedef struct {
    Node *pstart;
    Node *pend;
    size_t size;
}*ListHandle;

ListHandle openlist(void);
void closelist(ListHandle);
void push_front(ListHandle handle);
void push_back(ListHandle handle);
void display_list(ListHandle handle);
void pop_front(ListHandle handle);
void pop_back(ListHandle handle);
void remove_date(const Date *);
void clear_list(ListHandle handle);
size_t get_size(ListHandle handle);

/********** datelist.c *************************/
PRIVATE Node *create_node(void);
PRIVATE void free_nodes(Node *p);
/*************************************************************************/
PRIVATE void free_nodes(Node *p)
{
    Node *temp;
    while (p) {
        temp = p;
        p = p->pnext;
        free(temp);
    }
}
/*************************************************************************/
PRIVATE Node *create_node(void)
{
    Node *pd = (Node *)malloc(sizeof(Node));
    if (!pd) {
        printf("cannot allocate memory!\n");
        exit(EXIT_FAILURE);
    }
    return pd;
}
/*************************************************************************/
PUBLIC ListHandle openlist(void)
{
    ListHandle pd = (ListHandle)malloc(sizeof(*pd));

    if (!pd) {
        printf("cannot allocate memory!\n");
        exit(EXIT_FAILURE);
    }
    pd->pstart = pd->pend = NULL;
    pd->size = 0;
```

```
    return pd;
}
/*************************************************************************/
PUBLIC void closelist(ListHandle handle)
{
    clear_list(handle);
    free(handle);
}
/*************************************************************************/
PUBLIC void push_front(ListHandle handle)
{
    Node *pnew = create_node();
    set_random(&pnew->date);
    handle->size++;
    if (handle->pstart == NULL) {
        handle->pstart = handle->pend = pnew;
        pnew->pnext = NULL;
        return;
    }
    pnew->pnext = handle->pstart;
    handle->pstart = pnew;
}
/*************************************************************************/
PUBLIC void push_back(ListHandle handle)
{
    Node *pnew = create_node();
    set_random(&pnew->date);
    pnew->pnext = NULL;
    handle->size++;
    if (handle->pstart == NULL) {
        handle->pstart = handle->pend = pnew;
        return;
    }
    handle->pend->pnext = pnew;
    handle->pend = pnew;
}
/*************************************************************************/
PUBLIC void display_list(ListHandle handle)
{
    Node *cur = handle->pstart;

    if (!handle->size) {
        printf("empty list!\n");
        return;
    }
    while (cur) {
        display_date(&cur->date);
        cur = cur->pnext;
    }
}
/*************************************************************************/
PUBLIC void clear_list(ListHandle handle)
{
    free_nodes(handle->pstart);
    handle->pstart = handle->pend = NULL;
}
/*************************************************************************/
PUBLIC size_t get_size(ListHandle handle)
{
    return handle->size;
}
```

```
/*************************************************************************/
PUBLIC void pop_front(ListHandle handle)
{
    Node *temp;

    if (!handle->size) {
        printf("liste bos!\n");
        return;
    }
    handle->size--;
    if (handle->pstart == handle->pend) {
        free(handle->pstart);
        handle->pstart = handle->pend = NULL;
        return;
    }
    temp = handle->pstart;
    handle->pstart = handle->pstart->pnext;
    free(temp);
}
/*************************************************************************/
PUBLIC void pop_back(ListHandle handle)
{
    Node *temp, *cur;

    if (!handle->size) {
        printf("liste bos!\n");
        return;
    }
    handle->size--;
    if (handle->pstart == handle->pend) {
        free(handle->pstart);
        handle->pstart = handle->pend = NULL;
        return;
    }
    temp = handle->pend;

    for (cur = handle->pstart; cur->pnext != handle->pend; cur = cur-
>pnext)
        ;
    cur->pnext = NULL;
    handle->pend = cur;

    free(temp);
}
/*************************************************************************/
void display_menu()
{
    printf("[1] PUSH FRONT\n");
    printf("[2] PUSH BACK\n");
    printf("[3] DISPLAY LIST\n");
    printf("[4] POP FRONT\n");
    printf("[5] POP BACK\n");
    printf("[6] EMPTY LIST\n");
    printf("[7] EXIT\n");

    printf("seciminiz : ");
}
/*************************************************************************/
int get_option()
{
    int option;
```

```
    display_menu();
    scanf("%d", &option);
    if (option < 1 || option > 7)
        option = 0;
    return option;
}
/*************************************************************************/
int main()
{
    int option;
    ListHandle handle = openlist();
    for (;;) {
        option = get_option();
        switch (option) {
        case 1: push_front(handle); break;
        case 2: push_back(handle); break;
        case 3: display_list(handle);break;
        case 4: pop_front(handle);break;
        case 5: pop_back(handle);break;
        case 6: clear_list(handle); break;
        case 7: goto END;
        case 8: printf("invalid entry!\n");
        }
    }
END:
    closelist(handle);
    printf("end of program!\n");

    return 0;
}
```

BİRLİKLER

Programcı tarafından yeni bir tür yaratmasına olanak veren bir başka araç da "birlikler"dir (*unions).*
Birlikler yapılara çok benzer. Bir birliğin kullanılabilmesi için, yani bir birlik türünden nesnelerin tanımlanabilmesi için önce birliğin bildirimi yapılmalıdır. Birlik bildirimleri aynı yapı bildirimleri gibi yapılır. Tek fark *struct* anahtar sözcüğü yerine *union* anahtar sözcüğünün kullanılmasıdır.
Aşağıdaki örneği inceleyin:

```
union Dword {
    unsigned char byte;
    unsigned int word;
    unsigned long dword;
};
```

Yukarıdaki deyimle, ismi *union Dword* olan yeni bir tür bildirilmiş olur. Bu bildirimin görülür olduğu yerlerde, bu tür kullanılabilir. Bir *typedef* bildirimi yapılarak, bu türün isminin, yalnızca *Dword* olması da sağlanabilir:

```
typedef union {
    unsigned char byte;
    unsigned int word;
    unsigned long dword;
}Dword;
```

Birlik Türünden Değişkenlerinin Tanımlanması

Birlik değişkenleri aynı yapı değişkenleri gibi tanımlanır. Birliklerde de, bellekte yer ayırma işlemi yapı bildirimi ile değil, yapı nesnesinin tanımlanması ile yapılır.

```
Dword a, b;
```

deyiminden sonra, *a* ve *b, Dword* türünden iki değişkendir.

Yine yapılarda olduğu gibi, birliklerde de bildirim ile tanımlama işlemi birlikte yapılabilir:

```
union Double {
    double d;
    unsigned char s[8];
} a, b, c;
```

Bu durumda *a*, *b* ve *c* değişkenlerinin bilinirlik alanları, birlik bildiriminin yapıldığı yere bağlı olarak, yerel ya da global olabilir.

Birlik elemanlarına da yapı elemanlarında olduğu gibi nokta işleciyle erişilir. Örneğin yukarıdaki tanımlamadan sonra *a.d* birliğin *double* türünden ilk elemanını belirtir. Benzer biçimde birlik türünden nesneleri gösterecek, gösterici değişkenler de tanımlanabilir. *Ok* işleci ile yine yapılarda olduğu gibi birliklerde de, adres yoluyla birlik elemanlarına ulaşılabilir:

```
union Dword *p;
p->word = 100;
```

p->word ifadesi ile, *p* adresindeki birlik nesnesinin *word* isimli elemanına erişilir.

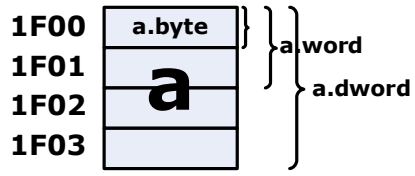
Birlik Nesnelerinin Bellekteki Yerleşimi

Birlik nesneleri için birliğin en uzun elemanı kadar yer ayrılır. Birlik elemanlarının hepsi aynı sayısal adresten başlayacak şekilde belleğe yerleşir. Örneğin:

```
union Dword {
    unsigned char byte;
    unsigned short word;
    unsigned long dword;
};

union Dword a;
```

bildirimi ile, *Dword* türüden *a* değişkeni için bellekte *Unix, Windows* sistemlerinde *4 byte* yer ayrılır. Çünkü *a* değişkeninin *dword* isimli elemanı, *4 byte* ile birliğin en uzun elemanıdır.

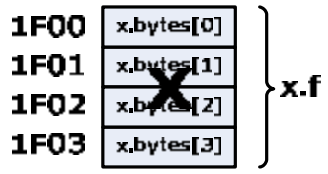


Birlik bir dizi içeriyorsa dizi tek bir eleman olarak alınır. Örneğin:

```
typedef union {
    float f;
    unsigned char bytes[4];
}Float;

Float x;
```

tanımı ile *x* değişkeni için ne kadar yer ayrılır? *Float* birliğinin iki elemanı vardır. Birincisi *4 byte* uzunluğunda *float* türden bir değişken, ikincisi *de 4 byte* uzunluğunda bir karakter dizisidir. İki uzunluk da aynı olduğuna göre *x* nesnesi için *4* byte yer ayrılacağı söylenebilir.



Buna göre

```
Float x;

x.f = 10.2F;
```

ile *x.bytes[0], x.bytes[1], x.bytes[2], x.bytes[3]* sırasıyla *float* türden elemanın *byte* değerleridir.
Aşağıdaki programı inceleyin:

```
#include <stdio.h>

typedef union {
    char ch;
    int i;
    double d;
    char s[4];
}Data;
```

```
int main()
{
    Data data;

    printf("&data    = %p\n", &data);
    printf("&data.ch = %p\n", &data.ch);
    printf("&data.i  = %p\n", &data.i);
    printf("&data.d  = %p\n", &data.d);
    printf("data.s   = %p\n", data.s);

    return 0;
}
```

Yukarıdaki programın derlenip çalıştırılmasıyla ekrana hep aynı adres yazdırılır.

Birlik elemanlarının aynı orjinden, yani aynı sayısal adresten başlayarak yerleştirilmesi bir elemanın değeri değiştirildiğinde, diğer elemanların da değerlerinin değişeceği anlamına gelir. Zaten birliklerin kullanılmasının asıl amacı da budur.

Birlik Nesnelerine İlkdeğer Verilmesi

C standartlarına göre, birlik nesnelerinin yalnızca ilk elemanına ilkdeğer verilebilir. Bir birlik nesnesinin birden fazla elemanına ilkdeğer vermek geçersizdir:

```
union Dword {
    unsigned char byte;
    unsigned int word;
    unsigned long dword;
} x = {'m'};

union Dword  y = {'a', 18, 24L};          /* Geçersiz */
```

Birlikler Neden Kullanılır

Birlikler başlıca üç amaç için kullanılabilir. Birinci amaç bellekte yer kazanmaya yöneliktir. Birlik kullanarak farklı zamanlarda kullanılacak birden fazla değişken için ayrı ayrı yer ayırma zorunluluğu ortadan kaldırılır. Örneğin bir hediyelik eşya kataloğu ile üç değişik ürünün satıldığını düşünelim: kitap, tshort ve saat. Her bir ürün için bir stok numarası, fiyat bilgisi ve ürün tip bilgisinin dışında ürüne bağlı olarak başka özelliklerin de tutulmak zorunda olduğunu düşünelim:

kitaplar : isim, yazar, sayfa sayısı.
t-short : desen, renk, boyut
saat : model

```
typedef struct {
    int stok_no;
    float fiyat;
    int urun_tipi;
    char kitapisim[20];
    char yazar[20];
    int sayfa_sayisi;
    char desen[20];
    int renk;
    int boyut;
    char saatisim[20];
    int model;
}Katalog;
```

Yukarıdaki bildirimde *Katalog* yapısının *urun_tipi* isimli elemanı ürünün ne olduğu bilgisini tutar. Bu elemanın değeri yalnızca *KITAP*, *TSHORT* ya da *SAAT* olabilir. Bunların simgesel değişmezler olarak tanımlandığını düşünelim. Yukarıda bildirilen yapı ürünlerin bütün özelliklerini tutabilir ancak şöyle bir sakıncası vardır:

Eğer ürün tipi *KITAP* değil ise *isim[20], yazar[30]* ve *sayfa_sayisi* isimli elemanlar hiç kullanılmaz. Yine ürün tipi *TSHORT* değil ise *desen[20]*, *renk*, *boyut* elemanları hiç kullanılmaz.
Unix ya da *Windows* sistemlerinde *byte hizalama* altında yukarıda bildirilen *Katalog* türünün *sizeof* değeri *108*'dir. Yani *Katalog* türünden bir nesne tanımlandığında bu nesne bellekte *108* byte yer kaplar.

```
#include <stdio.h>

int main()
{
    Katalog katalog;

    printf("sizeof(katalog) = %d\n", sizeof(katalog));
    return 0;
}
```

Ama *Katalog* yapısının bir elemanının bir birlik türünden olmasıyla yer kazancı sağlanabilir:

```
typedef struct {
    char isim[20];
    char yazar[20];
    int sayfa_sayisi;
}Kitap;

typedef struct {
    char desen[20];
    int renk;
    int size;
}Tshirt;

typedef struct {
    char isim[20];
    int model;
}Saat;

typedef union {
    Kitap kitap;
    Tshirt tshirt;
    Saat saat;
}Data;

typedef struct {
    int stok_no;
    float fiyat;
    int urun_tipi;
    Data data;
}Katalog;
```

Yukarıdaki bildirimleri *inceleyin*. Önce kitap, tshirt ve saate ilişkin bilgileri tutmak amacıyla *Kitap*, *Tshirt* ve *Saat* isimli yapılar bildirilmiş. *Unix*, *Windows* sistemlerinde *Kitap*, *Tshirt* ve *Saat* yapılarının *sizeof* değerleri sırasıyla *44*, *28* ve *24*'dür. Daha sonra *Data* isimli bir birliğin bildirildiğini görüyorsunuz. Bu birliğin elemanları *Kitap*, *Tshirt* ve *Saat* türlerinden olduğuna göre bu birliğin *sizeof* değeri en uzun elemanının yani *Kitap* yapısı türünden olan elemanının *sizeof* u kadardır, yani *44* dür.
Katalog isimli yapının bildiriminde ise *Data* birliği türünden *data* isimli bir eleman yer alıyor. Bu durumda *Katalog* yapısının *sizeof* değeri yalnızca *56* olur. Birliğin kullanılmasıyla *Katalog* yapısının *sizeof* değeri *108* den *56*'ya düşürülüyor.

```
#include <stdio.h>

int main()
{
    printf("sizeof(Katalog) = %u\n", sizeof(Katalog));

    return 0;
}
```

Katalog yapısıyla ilgili işlem yapan kod parçaları, *Katalog* yapısı türünden nesnenin önce *urun_tipi* isimli elemanının değerine bakarak, ürünün cinsi bilgisini elde ettikten sonra, duruma göre, *data* isimli elemanları farklı biçimde kullanabilir:

```
Katalog katalog;
/***/
if (katalog.urun_tipi == KITAP)
    puts(katalog.data.kitap.isim)
```

Birlik kulanımının ikinci nedeni herhangi bir veri türünün parçaları üzerinde işlem yapmak ya da parçalar üzerinde işlem yaparak bir bütünü oluşturmaktır. Aşağıdaki örneği inceleyin:

int türünün *2 byte* olduğu bir sistemde aşağıdaki bildirimlerin yapıldığını düşünelim:

```
typedef struct {
    unsigned char low_byte;
    unsigned char high_byte;
}Word;

typedef union {
    unsigned int i;
    Word w;
}Wordform;
```

bildirimlerinden sonra, *Wordform* türünden bir nesne tanımlanırsa bu birlik nesnesi, alçak (*low_byte*) ve yüksek (*high_byte*) anlamlı *byte*'larına ayrı ayrı erişilebilen bir tamsayı olarak kullanılabilir:

```
#include <stdio.h>

int main()
{
    Wordform wf;

    wf.w.low_byte = 0x12;
    wf.w.high_byte = 0x34;
    printf("%x\n", wf.i);

    return 0;
}
```

Yani I*ntel* işlemcilerinin bulunduğu *16* bit sistemlerde yukarıdaki *main* işlevinin derlenip çalıştırılmasıyla:

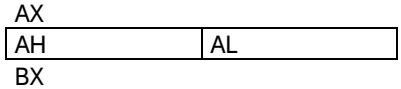
```
printf("%x\n", wf.i);
```

çağrısı ile *3412* sayısı ekrana yazdırır. *Motorola* işlemcilerinde (*big endian)* düşük sayısal adreste düşük anlamlı düşük anlamlı *byte* değeri olacağına göre sayının ters olarak görülmesi gerekir.

Benzer bir tema, işlemcilerin yazmaçlarının yazılımsal olarak temsil edilmesinde karşımıza çıkar.

8086 işlemcilerinde toplam 14 yazmaç vardır.
- 4 tane genel amaçlı yazmaç*: AX, BX, CX, DX*
- 4 tane segment yazmacı: *CS, DS, CS, ES*
- 2 tane indeks yazmacı: *SI, DI*
- 3 tane gösterici yazmacı: *IP, BP, SP*
- 1 tane bayrak yazmacı: *F*

Bütün yazmaçlar *16* bit uzunluğundadır. Genel amaçlı yazmaçlar olan *AX, BX, CX, DX* yazmaçları 8'er bitlik iki parçaya ayrılır. Genel amaçlı yazmaçlar aşağıdaki gibi bağımsız 8 bitlik yazmaçlar gibi de kullanılabilir.



BH	BL

CX

CH	CL

DX

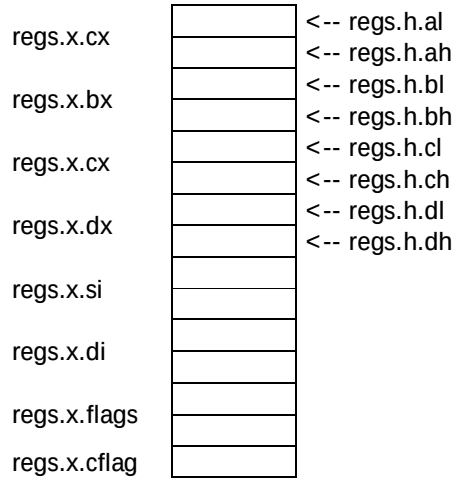
DH	DL

Bu yazmaçlar aynı zamanda bütünün parçalarıdır, yani *AH* ve *AL* yazmaçlarına ayrı ayrı değerler yazıp *AX* yazmacı bütünsel bir değer olarak çekilebilir. Yazmaçlar bellekte bir *birlik (union)* ile temsil edilebilirler.

```
struct BYTEREGS {
    unsigned char al, ah, bl, bh, cl, ch, dl, dh;
};

struct WORDREGS {
    unsigned ax, bx, cx, dx, si, di, flags, cflag;
};

union REGS {
    struct BYTEREGS h;
    struct WORDREGS x;
};
```



Birliklerin Karışık Veri Yapılarında Kullanılması

Birlikler kullanılarak, elemanları farklı türden olan diziler oluşturulabilir:
Bir dizinin elemanları *char, int, double* veya programcı tarafından yaratılan bir tür olan *Complex* türünden olabilsin. Dizinin herhangi bir elemanı bu dört türden herhangi birinden olabilecek şekilde kullanılabilsin.
Önce *Complex* isimli bir tür yaratalım:

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

typedef struct _Complex {
    double m_r, m_i;
}Complex;

void set_cr(Complex *p)
{
    p->m_r = (rand() % 2 ? 1:-1) * ((double)rand() / RAND_MAX + rand()% 9);
    p->m_i = (rand() % 2 ? 1:-1) * ((double)rand() / RAND_MAX + rand()% 9);
}

void display_c(const Complex *p)
{
    printf("(%.2lf %+.2lfi)", p->m_r, p->m_i);
}
```

Yukarıda tanımlanan *set_cr* işlevi adresini aldığı bir *Complex* nesnesini rastgele bir değerle set ediyor. *display_c* isimli işlev ise adresini aldığı bir *Complex* nesnesinin değerini ekrana yazdırıyor.

Önce okunabilirliği artırmak için bazı simgesel değişmezler tanımlayalım:

```
#define     CHAR       0
#define     INT        1
#define     DOUBLE     2
#define     COMPLEX    3
```

Şimdi ismi *Data* olan bir birlik türü yaratalım.

```
typedef union{
    char ch;
    int i;
    double d;
    Complex c;
}Data;
```

Data türünün *sizeof* değeri en uzun elemanı olan *Complex* türden *c* isimli elemanın *sizeof* değeri kadar olur değil mi? *Unix* ya da *Windows* sistemlerinde *Data* türünün *sizeof* değeri *16 byte* olur. Şimdi de ismi *Element* olan bir yapı türü yaratalım:

```
typedef struct {
    char type;
    Data data;
}Element;
```

Element türünün *sizeof* değeri elemanlarının *sizeof* değerlerinin toplamı kadar olduğuna göre (*byte alignment*'ın etkin olduğu varsayılıyor) *UNIX* ve *Windows* sistemlerinde *Element* türünden bir nesne bellekte *17* byte yer kaplar. Oluşturulacak dizinin elemanları *Element* türünden olur. *Element* türünün elemanı olan *type Element* türden bir nesnenin elemanı olan *data*'nın hangi elemanının kullanıldığı bilgisini tutar. Şimdi *Element* türden bir nesneyi rastgele bir değerle set edecek bir işlev tanımlayalım:

```
void set_elem_random(Element *ptr)
{
    switch (rand() % 4) {
    case CHAR      :   ptr->type = CHAR; ptr->data.ch = rand() % 26 + 'A';
                           break;
    case INT       :   ptr->type = INT; ptr->data.i = rand(); break;
    case DOUBLE :   ptr->type = DOUBLE;
                ptr->data.d = (double)rand() / RAND_MAX + rand() % 10;
break;
    case COMPLEX:  ptr->type = COMPLEX; set_cr(&ptr->data.c);
    }
}
```

Şimdi de *Element* türünden bir nesnenin değerini ekrana yazdıran bir işlev tanımlayalım:

```
void display_elem(const Element *ptr)
{
    switch (ptr->type) {
    case CHAR      : printf("(%c)", ptr->data.ch); break;
    case INT       : printf("(%d)", ptr->data.i); break;
    case DOUBLE : printf("(%.2lf)", ptr->data.d); break;
    case COMPLEX   : display_c(&ptr->data.c);
    }
}
```

Şimdi de *Element* türünden bir diziyi rastgele değerlerle set edecek ve dizinin elemanlarını ekrana yazdıracak işlevler yazalım:

```
void display_array(const Element *ptr, int size)
{
    int k;

    for (k = 0; k < size; ++k) {
        if (k && k % 5 == 0)
            printf("\n");
        display_elem(ptr + k);
    }
}

void set_array(Element *ptr, int size)
{
    int k;

    for (k = 0; k < size; ++k)
        set_elem_random(ptr + k);
}
```

Şimdi aşağıdaki *main* işlevini inceleyin:

```
#define    ARRAY_SIZE     50

int main()
{
    Element a[ARRAY_SIZE];

    srand(time(0));

    set_array(a, ARRAY_SIZE);
    display_array(a, ARRAY_SIZE);

    return 0;
}
```

main işlevi içinde *Element* türünden *a* isimli bir dizi tanımlanıyor.

```
set_array(a, ARRAY_SIZE);
```

çağrısıyla *a* dizisinin elemanlarına rastgele değerler atanıyor. Tabi bu durumda dizinin elemanları rastgele bir biçimde *char*, *int*, *double* ya da *Complex* türlerinden seçiliyor.

```
display_array(a, ARRAY_SIZE);
```

çağrısıyla dizinin elemanlarının değerleri ekrana yazdırılıyor. Aşağıda programın çalıştırılmasıyla elde edilen örnek bir ekran çıktısı veriliyor:

```
(1.07)(4.49)(19755)(1.69 -8.82i)(0.91)
(4.45)(9.35)(1.41)(V)(5.57 +5.51i)
(K)(T)(11331)(7.78)(22316)
(26066)(28923)(2.36 -2.31i)(C)(8.39)
(9951)(8.79)(28301)(7.50)(18583)
(20564)(7.02)(-0.23 -5.46i)(28931)(-4.76 +2.32i)
(5850)(14891)(J)(-5.40 +8.65i)(H)
(7.50 -3.32i)(-3.81 +3.17i)(D)(-5.38 -6.72i)(2.15 +2.47i)
(-4.04 +2.10i)(4737)(-0.21 -0.61i)(-9.00 -3.72i)(4276)
(12552)(-6.95 -0.11i)(E)(0.71)(-4.72 +7.21i)
```

NUMARALANDIRMALAR

Yazılan birçok programda, yalnızca belirli anlamlı değerler alabilen değişkenler kullanma gereksinimi duyulur. Örneğin bir "*Boolean*" değişkeni, yalnızca iki değere sahip olabilir: "Doğru" ve "Yanlış". Haftanın bir gününü tutacak bir değişken, haftanın yedi günü olduğuna göre yedi farklı değerden birini alabilir.
Başka bir örnek olarak bir oyun kağıdının rengini tutacak bir değişkeni verilebilir. Böyle bir değişken yalnızca 4 değişik değer alabilir: *Sinek, Karo, Kupa, Maça*.
Böyle değişkenler tamsayı türlerinden tanımlanabilir:

```
int renk;
renk = 1;
```

Yukarıdaki örnekte renk isimli değişken bir oyun kağıdının renk bilgisini tutuyor. Bu değişkene *1* değerinin atanması renk değerinin *Karo* olduğunu gösteriyor.

Böyle bir teknik uygulamalarda pekala kullanılabilir. Ancak bu tekniğin bazı sakıncaları vardır.

1. Kodu okuyan *renk* isimli değişkene yapılan atamalarda kullanılan tamsayı değişmezlerinin neyi temsil ettiği konusunda doğrudan bilgi alamaz. Kodun okunabilirliği azalır. Örneğin

```
renk = 1;
```

şeklinde bir atama doğrudan *renk* değişkenine *"karo"* değerinin verilmiş olduğu biçiminde de anlamlandıramaz.

2. Derleyici *renk* değişkeni ile ek bir kontrol yapamaz. Derleyici *renk* değişkeninin yalnızca *4* farklı değerden birini alması gerektiğinin farkında değildir. Derleyiciye böyle bir bilgi verilmemiştir. Örneğin böyle bir değişkene *10* gibi bir değerin atanması anlamsızdır. Ancak derleyici derleme sırasında bu yanlışlığı bulamaz, bir hata ya da bir uyarı iletisiyle veremez.

C dili, isimlendirilmiş belirli tamsayı değerlerini alabilen, bir tür yaratılmasına olanak veren bir araca sahiptir. Bu araca *numaralandrırma (enumaration)* ve bu araçla ilişkili kullanılan değişmezlere (*enumaration constants*) "numaralandırma değişmezleri" denir.

Bir numaralandırma türünün ve bir numaralandırma türüne ilişkin değişmezlerin bildirimi aşağıdaki gibi yapılır:

```
enum [enum türünün isimi] {değişmez1,değişmez2, ......};
```

enum bir anahtar sözcüktür. Derleyici küme ayraçları arasında isimlendirilmiş değişmezleri *0* değerinden başlayarak artan tamsayı değerleriyle eşler. Örnek :

```
enum Renk {Sinek, Karo, Kupa, Maca};
```

Yukarıdaki bildirimle *Sinek* değişmezi *0*, *Karo* değişmezi *1*, *Kupa* değişmezi *2*, *Maca* değişmezi ise *3* değerini alır.

```
enum Bool {TRUE, FALSE};
```

burada *TRUE* değişmezi *0 FALSE* değişmezi ise *1* değerini alır.

```
enum Months {January, February, March, April, May, June, July, August,
September, Oktober, November, December};
```

Bu bildirim ile numaralandırma değişmezlerine
January = 0, February = 1, ... *December = 11* biçiminde artan değerler verilir.

Numaralandırma değişmezlerine atama işleci ile küme ayraçları içinde değerler verilebilir, bu biçimde değer verilmiş bir numaralandırma değişmezinden sonraki değişmezin değeri önceki değerden bir fazlasıdır.

```
enum Months {January = 1, February, March, April, May, June, July,
August,September, Oktober, November, December};
```

Bu bildirim ile numaralandırma değişmezlerine *January = 1*, *February = 2*, ... *December = 12* biçiminde artan değerler verilmiş olur.

enum değişmezlerine *int* türü sayı sınırları içinde *pozitif* ya da *negatif* değerler verilebilir:

```
enum RTVals {
    RT1 = -127,
    RT2,
    RT3,
    RT4 = 12,
    RT5,
    RT6,
    RT7 = 0,
    RT8,
    RT9 = 90
};
```

Yukarıdaki bildirim ile numaralandırma değişmezlerinin alacağı değerler aşağıdaki gibi olur:

```
RT1 = -127, RT2 = -126, RT3 = -125, RT4 = 12, RT5 = 13,  RT6 = 14, RT7 = 0,
RT8 = 1,    RT9 = 90
```

Bir numaralandırma bildiriminde kullanılan farklı numaralandırma değişmezlerinin değeri aynı olabilir. Aşağıdaki bildirim geçerlidir:

```
enum Bool {YANLIS, FALSE = 0, DOGRU, TRUE = 1};
```

Bir numaralandırma değişmezinin ismi, aynı bilinirlik alanında kullanılan tüm isimlerden farklı olmalıdır. Bir numaralndırma değişmezi ismi ile aynı bilinirlik alanında bildirilen bir değişkenin isminin ya da bir başka numaralandırma değişmezinin isminin aynı olması geçersizdir.

Numaralandırma isimleri (*enum tag*), yapı ve birlik isimleri gibi bilinirlik alanı kurallarına uyar. Bu isimler aynı bilinirlik alanındaki diğer numaralandırma, yapı ve birlik isimlerinden farklı olmalıdır.

Bildirimlerde numaralandırma ismi (*enumeration tag*) hiç belirtilmeyebilir. Bu durumda genellikle küme ayracının kapanmasından sonra o numaralandırma türünden nesne(ler) tanımlanarak deyim sonlandırıcı atom ile bitirilir.

```
enum  {Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday}day1, day2, day3;
```

Yukarıdaki örnekte *day1, day2, day3* bildirimi yapılmış numaralandırma türünden (belirli bir tür ismi seçilmemiştir) yaratılmış değişkenlerdir. Böyle bir bildirim + tanımlama işleminin dezavantajı, artık bir daha aynı numaralandırma türünden başka bir değişkenin yaratılmamasıdır.

Bildirilen bir numaralandırma türünden nesne tanımlamak da zorunlu değildir:

```
enum {LOW = 10, HIGH = 20};
```

Yukarıdaki bildirim geçerlidir. Bildirilen türün bir ismi yoktur ama bu bildirimin görülür olduğu her yerde *LOW* ve *HIGH* numaralandırma değişmezleri kullanılabilir.

Numaralandırma Türlerinin sizeof Değerleri

Bir numaralandırma türünden tanımlanmış bir nesne için derleyici, kullanılan sistemde *int* türünün uzunluğu kadar bir yer ayırır. Derleyici için numaralandırma türünden bir nesne ile *int* türden bir nesne arasında fiziksel olarak herhangi bir fark bulunmaz.
örneğin *Unix* altında :

```
sizeof(enum Days) == sizeod(int) ==  4
```

ifadesi doğrudur.

Tabi bu kuralın bir kötü tarafı *int* türünün *2* byte olduğu bir sistemde kullanılabilecek en büyük numaralandırma değişmezinin değeri *32767* olabilir.

Numaralandırmalar ve typedef Bildirimi

Bir numaralandırma türü için de bir *typedef* bildirimi yapılabilir. Böylece türün kullanımında artık *enum* anahtar sözcüğünün kullanılmasına gerek kalmaz:

```
typedef enum tagRenk{SINEK, KARO, KUPA, MACA} Renk;
```

Bildiriminden sonra artık *enum* anahtar sözcüğü kullanılmadan

```
Renk kart1, kart2, kart3, kart4;
```

gibi bir tanımlama yapılabilir. *typedef* bildiriminden sonra artık *Renk* bir tür ismi belirtir. Şimdi de aşağıdaki örneğe bakalım:

```
typedef enum {FALSE, TRUE} Bool;
```

Artık *Bool* ismi *FALSE* ve *TRUE* değerlerini alabilen bir türün ismidir.

[C++ dilinde yapı, birlik ve numaralandırma türlerine ilişkin isimler (tag) aynı zamanda türün genel ismi olarak kullanılabileceği için yukarıdaki gibi bir *typedef* bildirimine gerek yoktur.]

Numaralandırma Türlerinin Kullanımı

enum bildirimi ile yeni bir tür yaratıldığına göre, bu türden nesne ya da gösterici tanımlanabilir. Tanımlanacak işlevlerin geri dönüş değerleri ve/veya parametre değişkenleri de böyle bir türden olabilir:

```
typedef enum {Sunday, Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday}Days;
typedef struct {int d, m, y}Date;
```

Bir yapı türünün bildiriminde, elemanlar bir numaralandırma türünden seçilebilir:

```
typedef struct {
    Days day;
    Months month;
    int year;
}Date;
```

Yukarıdaki örnekte *Date* isimli yapı türünün *day* ve *month* isimli elemanları *Days* ve *Months* isimli numaralandırma türlerindendir.

Numaralandırma Değişmezleri Değişmez İfadesi Olarak Kullanılabilir

Bir numaralandırma değişmezi bir değişmez ifadesi (*constant expression*) oluşturur. Bu yüzden numaralandırma değişmezleri değişmez ifadesi gereken yerlerde kullanılabilir. Aşağıdaki kodu inceleyin:

```
enum {MIN_SIZE = 100, AVERAGE_SIZE = 500, MAX_SIZE = 1000};

void func()
{
    int a[MIN_SIZE];
    int b[AVERAGE_SIZE];
    int c[MAX_SIZE];
    /***/
}
```

Yukarıdaki *main* işlevinin içinde tanımlanan *a, b* ve *c* isimlerinin boyutları numalandırma değişmezleri ile belirlenmiştir. Numaralandırma değişmezlerin *switch* deyiminin *case* ifadeleri olarak kullanılmaları çok sık karşılaşılan bir durumdur:

```
Renk renk;
/***/
switch (renk) {
case SINEK: /***/
case KARO : /***/
case KUPA : /***/
case MACA : /***/
}
```

Numaralandırma Türleri ile #define Değişmezlerinin Karşılaştırılması

#define önişlemci komutu önişlemci programa ilişkindir. Fakat numaralandırma değişmezleri derleme aşamasında ele alınır.
Bir *#define* simgesel değişmezi için bilinirlik alanı diye bir kavram söz konusu değildir, çünkü bilinirlik alanı derleyicinin anlamlandırdığı bir kavramdır. Ancak numaralandırma türleri ve değişmezleri için bilinirlik alanı kuralları geçerlidir. Eğer bir numaralandırma türü, bir işlev içinde bildirilmişse bu türe ilişkin numaralandırma değişmezleri söz konusu işlevin dışında bilinmez.

Birden fazla *#define* simgesel değişmezi, mantıksal bir ilişki içinde kullanılsa bile, derleyici bu ilişkinin farkında olmaz. Bu yüzden, derleyici programın, bu simgesel değişmezlerin yanlış kullanımına ilişkin bir uyarı iletisi üretme şansına sahip olmaz. Ancak derleyici aynı tür kapsamında bildirilen numaralandırma değişmezlerinin arasındaki mantıksal ilişkinin farkındadır. Bunların yanlış kullanılması durumunda uyarı iletisi üretebilir.

Numaralandırma değişmezleri de *#define* komutuyla tanımlanmış simgesel değişmezler gibi, nesne belirtmez. Örneğin :

```
enum METAL {Demir, Bakir, Kalay, Cinko, Kursun};
/***/
Kalay = 3;      /* Geçersiz */
```

Numaralandırma değişmezleri, mantıksal ilişki içinde bulunan belirli sayıda tanımlamalar için tercih edilir. Mantıksal ilişki içindeki bazı isimlerin bir tür kapsamı altında ele alınması okunabilirliği artırır. Örneğin:

```
enum Gun {Pazartesi, Sali, Carsamba, Persembe, Cuma, Cumartesi, Pazar};
```

Derleyicilerin sistemlere yönelik sağladığı başlık dosyalarında, birçok numaralandırma türü ve bu türlere ilişkin numaralandırma değişmezleri tanımlanır. Örneğin aşağıdaki numaralandırma bildirimi *80x86* sistemlerinde çalışan bir *Borland* derleyicisinin *graphics.h* isimli başlık dosyasından alınmıştır:

```
enum COLORS {
    BLACK,
    BLUE,
    GREEN,
    CYAN,
    RED,
    MAGENTA,
    BROWN,
    LIGHTGRAY,
    DARKGRAY,
    LIGHTBLUE,
    LIGHTGREEN,
    LIGHTCYAN,
    LIGHTRED,
    LIGHTMAGENTA,
    YELLOW,
    WHITE
};
```

Numaralandırma Türlerine İlişkin Tür Dönüşümleri

C dilinde bir numaralandırma türü derleyici tarafından fiziksel olarak *int* türü biçiminde ele alınır. Numaralandırma türleri ile diğer doğal türler arasında otomatik tür dönüşümü söz konusudur. Örneğin bir numaralandırma değişkenine *int* tüden bir değer atanabilir. Ya da *int* türden bir değişkene bir numaralandırma değişmezi atanabilir:

```
typedef enum {OFF, HOLD, STANDBY, ON}Position;

void func()
{
    Position pos = 2;  /* Geçerli */
    int x = ON;        /* Geçerli */
    /***/
}
```

Ancak numaralandırma değişkenlerine diğer doğal türlerden atama yapıldığında tür dönüştürme işlecinin kullanılması programın okunabilirliğini artırır:

```
pos = (Position)2;
```

Numaralandırma Değişmezlerinin Ekrana Yazdırılması

Bir numaralandırma değeri ekrana standart *printf* işleviyle *%d* format karakteriyle eşlenerek yazdırılabilir. Numaralandırma değişmezinin tamsayı değerini değil de ismini ekrana yazdırmanın doğrudan bir yolu yoktur. Bu amaçla bir işlev yazılabilir:

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

typedef enum {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday}Days;

void print_day(Days day)
{
    static const char *days[] = {"Sunday", "Monday", "Tuesday",
"Wednesday", "Thursday", "Friday", "Saturday"};

    printf("%s ", days[day]);
}

int main()
{
    Days day;
    int k;
    for (k = 0; k < 10; ++k) {
        day = (Days)(rand() % 7);
        print_day(day);
    }

    return 0;
}
```

BİTSEL İŞLEÇLER

Bitsel işleçler *(bitwise operators*), bir tamsayının bitleri üzerinde işlemler yapar. Daha çok sistem programlarında ya da düşük seviyeli kodlarda kullanılırlar. Bitsel işleçlerin ortak özellikleri, işleme soktukları tamsayıları bir bütün olarak değil, bit bit ele almalarıdır. Bu işleçlerin terimleri tamsayı türlerinden olmak zorundadır. Terimlerinin gerçek sayı türlerinden olması geçersizdir.

C dilinde toplam *11* bitsel işleç vardır. Aşağıdaki tabloda bitsel işleçler kendi aralarındaki öncelik sırasına göre listeleniyor:

öncelik seviyesi	simge	işleç
2	~	Bitsel değil *(bitwise not)*
5	<< >>	Bitsel sola kaydırma *(bitwise left shift)* Bitsel sağa kaydırma *(bitwise right shift)*
8	&	Bitsel ve *(bitwise and)*
9	^	Bitsel özel veya *(bitwise exor)*
10	|	Bitsel veya *(bitwise or)*
14	<<= >>= &= ^= |=	Bitsel işlemli atama işleçleri *(bitwise compound assignment operators)*

Yukarıdaki işleçler içinde, yalnızca "Bitsel değil" (*bitwise not*) işleci, tek terimli önek konumunda (*unary prefix*) bir işleçtir. Diğerleri iki terimli (*binary*) araek konumunda (*infix*) bulunan işleçlerdir.

Bitsel Değil İşleci

Bitsel değil işleci (*bitwise not*), diğer tüm tek terimli işleçler gibi işleç öncelik tablosunun ikinci seviyesindedir.
Bu işleç, terimi olan tamsayının bitleri üzerinde *1*'e tümleme işlemi yaparak bir değer elde eder. Yani terimi olan tamsayı değerinin *1* olan bitlerini *0*, *0* olan bitlerini *1* yaparak değer üretir. İşlecin yan etkisi (*side effect*) yoktur. İşlecin terimi bir nesne ise, bu nesnenin değeri değiştirilmez.
Aşağıda programı inceleyin:

```
#include <stdio.h>

int main()
{
    unsigned int x = 0x1AC3;
    unsigned int y;

    y = ~x;

    printf("y = %X\n", y);


    return 0;
}
```

Yukarıdaki programın, *int* türünün *4 byte* olduğu bir sistemde derlendiğini düşünelim. *x* değişkenine atanan *0x1AC3* değeri, ikilik sayı sisteminde aşağıdaki biçimde ifade edilir:

```
0000 0000 0000 0000 0001 1010 1100 0011
```

Bu durumda *y* değişkenine atanan değer

```
1111 1111 1111 1111 1110 0101 0011 1100
```

olur. *printf* işlevi ekrana

```
y = FFFFE53C
```

yazar. Şimdi de aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    int x = 0;
    int y = ~x;
    printf("y = %d\n", y);

    return 0;
}
```

İşaretli ikilik sayı sisteminde, *0*'ın bire tümleyeni *-1* değeridir, değil mi?

Bitsel Kaydırma İşleçleri

İki tane bitsel kaydırma işleci *(bitwise shift operator)* vardır:

Bitsel sağa kaydırma işleci >> *(bitwise right shift)*
Bitsel sola kaydırma işleci << *(bitwise left shift)*

Her iki işleç de, öncelik tablosunun 5. seviyesinde bulunur. Dolayısıyla bu işleçlerin önceliği tüm aritmetik işleçlerden daha düşük, fakat karşılaştırma işleçlerinden daha yüksektir. Bitsel kaydırma işleçleri iki terimli araek konumundaki *(binary infix)* işleçlerdir. Bitsel sola kaydırma işleci, soldaki terimi olan tamsayının, sağdaki terimi olan tamsayı kadar sola kaydırılmasından elde edilen değeri üretir. Sınır dışına çıkan bitler için, sayının sağından *0* biti ile besleme yapılır. Örnek:

```
#include <stdio.h>

int main()
{
    unsigned int x = 52;       /* x = 0000 0000 0011 0100 */
    unsigned int y = 2;
    unsigned int z;

    z  = x << y;               /* z = 0000 0000 1101 0000  */
    printf("z = %u\n", z);     /* z = 208  */

    return 0;
}
```

Bir tamsayıyı, sola bitsel olarak *1* kaydırmakla o tamsayının ikiyle çarpılmış değeri elde edilir.
Bitsel sağa kaydırma işleci, soldaki terimi olan tamsayının, sağdaki terimi olan tamsayı kadar sağa kaydırılmasından elde edilen değeri üretir. Sol terim işaretsiz *(unsigned)* bir tamsayı türünden ise, ya da işaretli *(signed)* bir tamsayı türünden ancak pozitif değere sahip ise, sınır dışına çıkan bitler yerine, sayının solundan besleme *0* biti ile yapılır. Sağa kaydırılacak ifadenin işaretli bir tamsayı türünden negatif değerli olması durumunda sınır

dışına çıkan bitler için soldan yapılacak beslemenin *0* ya da *1* bitleriyle yapılması derleyiciye bağlıdır *(implementation specified).* Yani derleyiciler bu durumda sayının işaretini korumak için soldan yapılacak beslemeyi *1* biti ile yapabilecek bir kod üretebilecekleri gibi, sayının işaretini korumayı düşünmeksizin *0* biti ile besleyecek bir kod da üretebilirler. İşaretli negatif bir tamsayının bitsel sağa kaydırılması taşınabilir bir özellik değildir.

Bir tamsayıyı sağa bitsel olarak *1* kaydırmakla, o sayının ikiye bölünmüş değeri elde edilir:

```
#include <stdio.h>

int main()
{
    unsigned int x = 52;       /* x = 0000 0000 0011 0100 */
    unsigned int y = 2;
    unsigned int z = x >> y;   /* z = 0000 0000 0000 1101 */

    printf("z = %u\n", z);     /* z = 13  */

    return 0;
}
```

Bitsel kaydırma işleçlerinin yan etkileri yoktur. Yani sol terimleri bir nesne ise, bu nesnenin bellekteki değeri değişmez. Kaydırma işlemi ile sol terim olan nesnenin değeri değiştirilmek isteniyorsa, bu işleçlerin işlemli atama biçimleri kullanılmalıdır.

Kaydırma işleçlerinin sağ terimi, sol terimin ait olduğu türün toplam bit sayısından daha küçük olmalıdır. Bu koşul sağlanmamış ise oluşan durum tanımsızdır *(undefined behaviour).* Örneğin *Windows* sistemlerinde *int* türden bir değerin *32* ya da daha fazla sola ya da sağa kaydırılması tanımsızdır. Bu durumdan kaçınılmalıdır.

Bitsel kaydırma işleçlerinin öncelik yönü soldan sağadır:

```
x << 4 >> 8
```

x, 2 byte'lık işaretsiz bir tamsayı değişken olsun. Yukarıdaki ifade derleyici tarafından

```
x << 4) >> 8
```

biçiminde ele alınır. Bu ifade ile *x* değişkeninin ortadaki *8* biti elde edilir.

Bitsel ve İşleci

"Bitsel ve" işleci (*bitwise and*), işleç öncelik tablosunun 8. seviyesindedir. Bu seviyenin öncelik yönü soldan sağadır. İşlecin yan etkisi yoktur. Terimleri nesne gösteren bir ifade ise, bu terimlerin bellekteki değeri değişmez. Değer üretmek için terimi olan tamsayıların karşılıklı bitlerini *"ve"* işlemine sokar. "Ve" işlecine ilişkin işlem tablosu tablosu aşağıda yeniden veriliyor:

x	y	x&y
1	1	1
1	0	0
0	1	0
0	0	0

"Bitsel ve" işlecinin ürettiği değer, terimlerinin karşılıklı bitlerinin "ve" işlemine sokulmasıyla elde edilen değerdir:

```
#include <stdio.h>

int main()
{
    unsigned int x = 0x1BC5;   /* x = 0001 1011 1100 0101 */
    unsigned int y = 0X3A0D;   /* y = 0011 1010 0000 1101 */
    unsigned int z = x & y;    /* z = 0001 1010 0000 0101 */
    printf("z = %X\n", z);     /* z = 0x1A05 */

    return 0;
}
```

1 biti "bitsel ve" işleminde etkisiz elemandır.
0 biti "bitsel ve" işleminde yutan elemandır.

"Mantıksal ve" işleci yerine yanlışlıkla "*bitsel ve"* işlecini kullanmak sık yapılan bir hatadır. Aşağıdaki kodu dikkatli bir şekilde inceleyin:

```
#include <stdio.h>

int main()
{
    int x = 85;
    int y = 170;

    if (x && y)
        printf("dogru!\n");
    else
        printf("yanlis!\n");

    if (x & y)
        printf("dogru!\n");
    else
        printf("yanlis!\n");

    return 0;
}
```

Yukarıdaki programda "mantıksal ve" (&&) işleci yerine yanlışlıkla "bitsel ve" (&) işleci kullanılıyor. Hem "mantıksal ve" hem de "bitsel ve", iki terimli, araek konumunda işleçlerdir. Derleyiciler yukarıdaki kod için bir hata ya da uyarı iletisi üretmez. Yukarıdaki örnekte "mantıksal ve" işlecinin kullanılması durumunda, mantıksal "*doğru*" biçiminde yorumlanacak olan ifade, bitsel ve işlecinin kullanılmasıyla *0* değeri üretir, mantıksal "*yanlış*" olarak yorumlanır.

Bitsel Özel Veya İşleci

Bitsel "özel veya" işleci (*bitwise exor*) işleç öncelik tablosunun 9. seviyesindedir. Öncelik yönü soldan sağadır. Yan etkisi yoktur, terimleri olan nesnelerin bellekteki değeri değişmez. Değer üretmek için, terimi olan tamsayıların karşılıklı bitlerini özel veya (*exclusive or*) işlemine sokar. Bitsel "özel veya" işlecine ilişkin işlem tablosu aşağıda veriliyor:

x	y	x ^ y
1	1	0
1	0	1
0	1	1
0	0	0

Yukarıdaki tablo şöyle özetlenebilir: Terimlerinden ikisi de aynı değere sahip ise, üretilen değer *0*, terimlerden biri diğerinden farklı ise üretilen değer *1* olur.

```
#include <stdio.h>

int main()
{
    unsigned int x = 0x1BC5;   /* x = 0001 1011 1100 0101 */
    unsigned int y = 0X3A0D;   /* y = 0011 1010 0000 1101 */
    unsigned int z = x ^ y;    /* z = 0010 0001 1100 1000 */
    printf("z = %X\n", z);     /* z = 21C8 */
    return 0;
}
```

Bir tamsayı, arka arkaya aynı değerle bitsel özel veya işlemine sokulursa, tamsayının kendi değeri elde edilir:

```
#include <stdio.h>

int main()
{
    unsigned int a = 0X1BC5;       /* a = 0001 1011 1100 0101 */
    unsigned int b = 0X3A0D;       /* b = 0011 1010 0000 1101 */

    a ^= b;                        /* a = 0010 0001 1100 1000*/
    a ^= b;                        /* b = 0001 1011 1100 0101 */
    printf("a = %X\n", a);         /* a = 0X1BC5 */

    return 0;
}
```

Bazı şifreleme algoritmalarında "özel veya" işleminin bu özelliğinden faydalanılır.

Bitsel Veya İşleci

"Bitsel veya" (*bitwise or operator*) işleci, işleç öncelik tablosunun *10.* seviyesindedir. Öncelik yönü soldan sağadır. Yan etkisi yoktur, terimleri nesne gösteren bir ifade ise bellekteki değerlerini değiştirmez. Değer üretmek için terimi olan tamsayıların karşılıklı bitlerini "veya" işlemine sokar. Bitsel veya işlecine ilişkin işlem tablosu aşağıda veriliyor:

x	y	x | y
1	1	1
1	0	1
0	1	1
0	0	0

Bitsel veya işleci, terimlerinin karşılıklı bitlerinin "veya" işlemine sokulmasıyla elde edilen değeri üretir:

```
#include <stdio.h>

int main()
{
    unsigned int x = 0x1BC5;   /* x = 0001 1011 1100 0101 */
    unsigned int y = 0X3A0D;   /* y = 0011 1010 0000 1101 */
    unsigned int z = x | y;    /* z = 0011 1011 1100 1101 */
    printf("z = %X\n", z);     /* z = 0x3BCD */

    return 0;
}
```

0 biti bitsel veya işleminde etkisiz elemandır. *1* biti bitsel ve işleminde yutan elemandır.

Aşağıdaki programda bitsel ve, özel veya, veya işleçlerinin ürettikleri değerler ikilik sayı sisteminde ekrana yazdırılıyor. Programı derleyerek çalıştırın:

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

void bit_print(int val)
{
    char bits[sizeof(val) * 8 + 1];

    itoa(val, bits, 2);
    printf("%10d %032s\n", val, bits);
}

int main()
{
    int x, y;

    printf("iki sayi giriniz ");
    scanf("%d%d", &x, &y);
    bit_print(x);
    bit_print(y);
    printf("bitsel ve islemi\n");
    bit_print(x & y);
    printf("*********************************************************\n");
    printf("bitsel ozel veya islemi\n");
    bit_print(x ^ y);
    printf("*********************************************************\n");
    printf("bitsel veya islemi\n");
    bit_print(x | y);

    return 0;
}
```

Bitsel İşleçler Kısa Devre Davranışı Göstermez

Bitsel işleçler kısa devre davranışına sahip değidir. Yani bu işleçlerin her iki terimi de mutlaka işlenir.

Bitsel İşlemli Atama İşleçleri

Bitsel değil işlecinin dışında, tüm bitsel işleçlere ilişkin işlemli atama işleçleri vardır. Daha önce de söylendiği gibi bitsel işleçlerin yan etkileri (*side effect*) yoktur. Bitsel işleçler terimleri olan nesnelerin bellekteki değerlerini değiştirmez. Eğer terimleri olan nesnelerin değerlerinin değiştirilmesi isteniyorsa bu durumda işlemli atama işleçleri kullanılabilir:

```
x = x << y  yerine      x <<= y
x = x >> y  yerine      x >>= y
x = x & y   yerine      x &= y
x = x ^ y   yerine      x ^= y
x = x | y   yerine      x |= y
```

kullanılabilir.
Bitsel özel veya işlemli atama işleci, tamsayı türlerinden iki değişkenin değerlerinin, geçici bir değişken olmaksızın takas *(swap)* edilmesinde de kullanılabilir:

```
#include <stdio.h>

int main()
{
    int x, y;

    printf("iki sayi giriniz ");
    scanf("%d%d", &x, &y);
    printf("x = %d\ny = %d\n", x, y);
    x ^= y ^= x ^= y;
    printf("x = %d\ny = %d\n", x, y);

    return 0;
}
```

Yukarıdaki programda, *x* ve *y* değişkenlerinin değerleri bitsel özel veya işlecinin kullanılmasıyla takas ediliyor.

Bitsel İşleçlerin Kullanılmasına İlişkin Bazı Temalar

Bitsel işleçlerin kullanılmasına daha çok sistem programlarında rastlanır. Sistem programlarında bir tamsayının bitleri üzerinde bazı işlemler yapılması sıklıkla gerekli olur. Aşağıda bitsel düzeyde sık yapılan işlemler açıklanıyor:

Bir Tamsayının Belirli Bir Bitinin Birlenmesi

Buna tamsayının belirli bir bitinin "set edilmesi" de denebilir. Bir tamsayının belirli bir bitini birlemek için, söz konusu tamsayı, ilgili biti *1* olan ve diğer bitleri 0 olan bir sayıyla "bitsel veya" işlemine sokulmalıdır. Çünkü bitsel veya işleminde *1* yutan eleman 0 ise etkisiz elemandır.
Aşağıdaki örnekte bir sayının *5.* biti birleniyor:

```
#include <stdio.h>

int main()
{
    int ch = 0x0041;          /* ch = 65  (0000 0000 0100 0001)  */
    int mask = 0x0020;        /* mask = 32 (0000 0000 0010 0000) */

    ch |= mask;               /* ch = 97 (0000 0000 0110 0001)   */
    printf("ch = %d\n", ch);  /* ch = 97 */

    return 0;
}
```

x bir tamsayı *k* da bu sayının herhangi bir bit numarası olmak üzere bir tamsayının *k.* bitini birleyecek bir ifade şu biçimde yazılabilir:

```
x |= 1 << k
```

Böyle işlemleri gerçekleştirmek için kullanılan *1 << k* gibi ifadelere *"bitsel maske"* (*bitmask*) ya da yalnızca "*maske*" denir.

Bir Tamsayının Belirli Bir Bitinin Sıfırlanması

Bir tamsayının belirli bir bitini sıfırlamak(*clear)* için, söz konusu tamsayı, ilgili biti *0* olan ve diğer bitleri *1* olan bir maskeyle "bitsel ve" işlemine sokulmalıdır. Çünkü "bitsel ve" işleminde, *0* yutan eleman *1* ise etkisiz elemandır. Aşağıdaki örnekte bir tamsayının *5*. biti sıfırlanıyor:

```
#include <stdio.h>

int main()
{
    int ch = 0x0061;              /* ch = 97 (0000 0000  0110 0001)   */
    int mask = ~0x0020;           /* mask = ~32 (1111 1111  1101 1111)*/
    ch &= mask;                   /* ch = 65 (0000 0000 0100 0001)    */
    printf("ch = %d\n", ch);      /* ch = 65 */

    return 0;
}
```

x, bir tamsayı, *k* da bu sayının herhangi bir bit numarası olmak üzere, bir sayının *k.* bitini sıfırlayan bir ifade aşağıdaki gibi genelleştirilebilir:

```
x  &= ~(1 << k);
```

Bir Tamsayının Belirli Bir Bit Değerinin Elde Edilmesi (0 mı 1 mi)

Bir tamsayının belirli bir bitinin *0* mı *1* mi oldugunun öğrenilmesi için, söz konusu tamsayı, ilgili biti *1* olan ve diğer bitleri 0 olan bir maskeyle "bitsel ve" işlemine sokularak mantıksal olarak yorumlanmalıdır. Çünkü "*bitsel ve"* işleminde *0* yutan eleman, *1* ise etkisiz elemandır. İfade mantıksal olarak "*dogru*" biçiminde yorumlanırsa, ilgili bit *1*, yanlış olarak yorumlanırsa ilgili bit *0* demektir.
x bir tamsayı, *k* da bu sayının herhangi bir bit numarası olmak üzerebir sayının *k.* bitinin *1* ya da *0* olduğunu sınayan bir deyim aşağıdaki biçimde yazılabilir:

```
if (x & 1 << k)
    /* k. bit 1 */
else
    /* k. bit 0 */
```

Bir pozitif tamsayının tek sayı olup olmadığı "bitsel ve" işleciyle sınanabilir. Bir tamsayı tek sayı ise sayının *0.* biti *1* dir.

```
#include <stdio.h>

int main()
{
    int x;

    printf("bir sayi giriniz ");
    scanf("%d", &x);
    if (x & 1)
        printf("%d tek sayi\n", x);
    else
        printf("%d cift sayi\n", x);

    return 0;
}
```

Bir Tamsayının Belirli Bir Bitini Ters Çevirmek

Bazı uygulamalarda bir tamsayının belirli bir bitinin değerinin ters çevrilmesi (*toggle - flip*) gerekir. Yani söz konusu bit *1* ise *0*, *0* ise *1* yapılmalıdır. Bu amaçla "bitsel özel veya" işleci kullanılır. Bitsel özel veya işlecinde *0* biti etkisiz elemandır. Bir sayının *k.* bitinin değerini değiştirmek için, sayı, *k.* biti *1* diğer bitleri *0* olan bir maske ile "bitsel özel veya" işlemine sokulur.
x, bir tamsayı, *k* da bu sayının herhangi bir bit numarası olmak üzere, bir sayının *k.* bitini ters çeviren bir ifade şu şekilde yazılabilir:

```
x ^= 1 << k;
```

Birden Fazla Bit Üzerinde İşlem Yapmak

Bir tamsayının belirli bitlerini sıfırlamak için ne yapılabilir? Örneğin *int* türden bir tamsayının *7., 8.* ve *9.* bitlerini, diğer bitlerin değerlerini değiştirmeden sıfırlamak isteyelim. Bunu gerçekleştirmek için, tamsayı *7., 8.* ve 9. bitleri *0* olan diğer bitleri *1* olan bir maske ile bitsel ve işlemine sokulabilir. Örneğin *int* türünün *16* bit olduğu bir sistemde bu sayı aşağıdaki bit düzenine sahip olur:

```
1111 1100 0111 1111
```

Bu değerin onaltılık sayı sisteminde gösterimi *0xFC7F* biçimindedir, değil mi?

```
x &= 0xFC7F;
```

Tamsayıların belirli bitleri üzerinde yapılan işleri, işlevlere yaptırmaya ne dersiniz?

```
void clearbits(int *ptr, size_t startbit, size_t nbits);
```

clearbits işlevi adresi gönderilen ifadenin *startbit* nolu bitinden başlayarak *nbits* tane bitini sıfırlar.
Örneğin *x* isimli *unsigned int* türden bir değişkenin *7. 8. 9*. bitlerini sıfırlamak için işlev aşağıdaki biçimde çağrılır:

```
clearbits(&x, 7, 3);
```

```
void clearbits(unsigned int *ptr, size_t startbit, size_t nbits)
{
    size_t k;

    for (k = 0; k < nbits; ++k)
        *ptr &= ~(1 << nbits + k);
}
```

Benzer biçimde *setbits* işlevi de yazılabilir. *x* isimli *unsigned int* türden bir değişkenin *7. 8. 9*. bitlerini birlemek için işlev aşağıdaki biçimde çağrılır:

```
clearbits(&x, 7, 3);
```

```
void setbits(unsigned int *ptr, size_t startbit, size_t nbits)
{
    size_t k;

    for (k = 0; k < nbits; ++k)
        *ptr |= ~(1 << nbits + k);
}
```

Peki örneğin *32* bitlik bir alan birden fazla değeri tutacak şekilde kullanılabilir mi?

Negatif sayıların kullanılmadığı düşünülürse *4* bitlik bir alanda *0 – 15* aralığındaki değerler tutulabilir. Bir değişkenin değerinin *0 – 15* aralığında değiştiğini varsayalım: Bazı durumlarda *4* bitle ifade edilebilen bir değerin *2* ya da *4 byte* yer kaplayan bir tamsayı türünde tutulması istenmeyebilir. *32* bitlik bir alan içinde aslında *4* ayrı değer tutulabilir, değil mi?
Bir sayının belirli bir bit alanında bir tamsayı değerini tutmak için ne yapılabilir? Önce sayının ilgili bitleri sıfırlanır. Bu amaç için yukarıda yazılan *clearbits* gibi bir işlev çağrılabilir. Daha sonra sayı, uygun bir değerle *bitsel veya* işlemine sokulabilir. Aşağıdaki işlev tanımını inceleyin:

```
void putvalue(unsigned int*ptr, size_t startbit, size_t nbits, int value)
{
    clearbits(ptr, startbit, nbits);
    *ptr |= value << startbit;
}
```

putvlaue işlevi adresi gönderilen nesnenin *startbit* nolu bitinden başlayarak *nbits* bitlik alanına *value* değerini yerleştirir. İşlev aşağıdaki gibi çağrılabilir:

```
putvalue(&x, 4, 3, 7);
```

Bir tamsayı değişkenin belirli bit alanları içinde saklanmış değer nasıl elde edilebilir? Şüphesiz bu iş de bir işleve yaptırılabilir:

```
unsigned getvalue(unsigned int x, size_t startbit, size_t nbits);
```

getvalue işlevi birinci parametresine aktarılan tamsayı değerin, *startbit* numaralı bitinden başlayarak *nbits* bitlik alanında saklanan değerle geri döner:

```
unsigned int getvalue(unsigned int number, size_t startbit, size_t nbits)
{
    number <<= sizeof(int) * 8 - startbit - nbits;
    return number >>= sizeof(int) * 8 - nbits;
}
```

Bitsel İşleçlerin Kullanımına İlişkin Bazı Örnek Uygulamalar

Aşağıda *int* türden bir değeri, ekrana ikilik sayı sisteminde yazdıran *showbits* isimli bir işlev tanımlanıyor:

```
#include <stdio.h>

void showbits(int x)
{
    int i = sizeof(int) * 8 - 1;
    for (; i >= 0; --i)
        putchar (x >> i & 1 ? '1' : '0');
}

int main()
{
    int val;

    printf("bir sayi giriniz : ");
    scanf("%d", &val);
    showbits(val);
    return 0;
}
```

Aşağıda aynı işi değişik bir biçimde yapan *showbits2* isimli bir işlevin tanımı yer alıyor:

```
void showbits2(int x)
{
    unsigned int i = (~((unsigned)~0 >> 1));

    while (i) {
        putchar (x & i ? '1' : '0');
        i >>= 1;
    }
}
```

Aşağıda tamımlanan *reverse_bits* isimli işlev, *int* türden bir değerin bitlerini ters çeviriyor:

```
#include <stdio.h>

int reverse_bits(int number)
{
    int k;
    int no_of_bits = sizeof(int) * 8;
    int rev_num = 0;

    for (k = 0; k < no_of_bits; ++k)
        if (number & 1 << k)
            rev_num |= 1 << (no_of_bits - 1 - k);
    return rev_num;
}
```

Aşağıda tamımlanan *reverse* isimli işlev, *unsigned char* türden bir değerin bitlerini ters çeviriyor:

```
#include <stdio.h>

unsigned char reverse(unsigned char byte)
{
    unsigned char dest;

    dest = (byte << 4) | (byte >> 4);
    dest = ((dest << 2) & 0xCC) | ((dest >> 2) & 0x33);

    return ((dest << 1) & 0xAA) | ((dest >> 1) & 0x55);
}
```

Aşağıda tanımlanan *no_of_setbits* isimli işlev, kendisine gönderilen *int* türden bir değerin kaç tane bitinin *1* olduğu bilgisi ile geri dönüyor:

```
#include <stdio.h>

int no_of_setbits(unsigned int value)
{
    int counter = 0;
    int k;

    for (k = 0; k < sizeof(int) * 8; ++k)
        if (value & 1<<k)
            counter++;

   return counter;
}

int main()
```

```
{
    int number;

    printf("bir sayı girin : ");
    scanf("%d", &number);
    printf("sayınızın %d biti 1n", no_of_setbits(number));

    return 0;
}
```

Şimdi de çok daha hızlı çalışacak bir işlev tasarlayalım:

```
int no_of_setbits(unsigned int value)
{
    static int bitcounts[] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3,
4};
    int counter = 0;

    for (; value != 0; value >>= 4)
        counter += bitcounts[value & 0x0F];

    return counter;
}
```

Yukarıda tanınımlanan işlevde yer alan *for* döngüsü içinde, döngünün her turunda *value* değişkeninin düşük anlamlı *4* bitindeki birlenmiş bitlerin sayısı, *4* bitin sayısal değerinin *bitcounts* isimli diziye indis yapılmasıyla elde ediliyor. Örneğin *4* bitlik alanda ifade edilen tamsayının değeri *11* olsun:

```
11 = 1011
```

Bu sayının toplam *3* biti *1*'dir.

bitcounts dizisinin *11* indisli elemanın değeri *3*'tür. Döngünün bir sonraki turuna geçmeden önce *value* değişkeni sağa *4* kez kaydırılarak bu kez *value*'nun bir sonraki *4* bitlik alanındaki bitlere bakılıyor.

Aşağıda işlem hızını daha da artıran bir işlev tanımlanıyor:

```
const static char bit_array[] = {
0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8
};

int count1bits(unsigned long x)
{
```

```
    return  bit_array[x & 0xff] + bit_array[(x >> 8) & 0xff] +
    bit_array[(x >> 16) & 0xff] + bit_array[(x >>24) & 0xff];
}
```

count1bits işlevi 32 bitlik bir tamsayının değeri *1* olan bitlerini sayıyor. İşlev bu işi yaparken *8* bitlik bir gruptaki *1* olan bitlerin sayısını *bit_array* dizisini kullanarak buluyor.

Aşağıda tanımlanan *gcd_b* işleviyle, iki tamsayının ortak bölenlerinden en büyüğü hesaplanıyor. İşlevin tanımı içinde kalan işlecinin kullanılmadığına, bitsel işleçlerin kullanıldığına dikkat edin:

```
unsigned int gcd_b(unsigned int x, unsigned int y)
{
    unsigned int temp;
    unsigned int cpof = 0;

    if (x == 0)
        return y;
    if (y == 0)
        return x;

    while (((x | y) & 1) == 0) {
        x >>= 1;
        y >>= 1;
        ++cpof;
    }

    while ((x & 1) == 0)
        x >>= 1;

    while (y) {
        while (!(y & 1))
            y >>= 1;
        temp = y;
        if (x > y)
            y = x - y;
        else
            y -= x;
       x = temp;
    }
    return x << cpof;
}
```

BİT ALANLARI

Bir yapının elemanı bit seviyesinde tutulan bir tamsayı değişken olabilir. Yapıların böyle elemanlarına "bit alanı" *(bit fields)* denir.
Bit alanları, C'nin bit seviyesinde erişime olanak veren bir aracıdır. Bit seviyesinde erişim amacıyla şüphesiz daha önceki konuda anlatılan bitsel işleçler kullanılabilir. Ancak bit alanları programcıya böyle ulaşımlar için daha kolay bir arayüz sunar.

Bir yapı içinde bir bit alanı oluşturmak için özel bir sözdizim kuralına uymak gerekir. Elemanın yapı içindeki bildiriminde eleman ismini *':'* atomu izler. Bu atomdan sonra bit alanının kaç bit yer kaplayacağı bilgisini gösteren bir tamsayı değişmezi yazılır. Aşağıdaki bildirimi inceleyin:

```
struct Data {
    unsigned int a: 5;
    unsigned int b: 4;
    unsigned int c: 7;
};
```

struct Data isimli yapının *a* isimli elemanı, yapı içinde *5* bit yer kaplarken, *b* isimli elemanı *4* bit, *c* isimli elemanı *7* bit yer kaplar. Yapının elemanları *unsigned int* türünden olduğuna göre bu türden bir nesnenin

a elemanı *0 - 31*
b elemanı *0 -15*
c elemanı *0 - 127*

arasında tamsayı değerler tutabilir.

Bit Alanı Elemanların Türleri

Bir bit alanı eleman, işaretli ya da işaretsiz *int* türlerden olabilir. Bir bit alanı eleman gerçek sayı türlerinden olamaz. C standartları bit alanı elemanlarının *char, short, long* türlerinden olamayacağını belirtse de C derleyicilerin çoğu seçimlik olarak bu duruma izin verir. Taşınabilirlik açısından bit alanı elemanın bildiriminde *signed* ya da *unsigned* sözcüklerinin kullanılması yerinde olur. Çünkü derleyici yalnızca *int* anahtar sözcüğüyle bildirilen bir bit alanının türünü işaretli ya da işaretsiz *int* türü olarak alabilir.
Bit alanı eleman bir dizi olamaz.

Bit Alanları Neden Kullanılır:

Bazı dışsal aygıtlar verileri bitsel düzeyde iletiyor olabilir.
Bazı sıkıştırma işlemlerinde bir tamsayının belirli sayıda biti kullanılıyor olabilir.
Bazı şifreleme işlemlerinden bir tamsayının belirli bit alanlarında yapılan kodlama yapıyor olabilir.
Çok sayıda bayrak değişkeni bit alanları olarak tutulabilir.

Şüphesiz bu işlemleri yapmak için bitsel işleçler de kullanılabilir. Ancak bit alanları bu işlerin yapılması için programcıya çok daha kolay bir arayüz sunar, programcının işini kolaylaştırır.
Örneğin DOS işletim sistemi dosyalara ilişkin tarih ve zaman bilgilerini *16* bitlik alanlarda tutar:

```
struct file_date {
    unsigned int day: 5;
    unsigned int mon: 4;
    unsigned int year: 7;
};
```

Yukarıdaki bildirimden de görüldüğü gibi, tarih bilgisinin yılı için 7 bitlik bir alan ayrılmıştır. *7* bitlik bir alanda tutulabilecek en büyük değer *127* dir. *DOS* işletim sisteminde böyle bir bit alanında yıl değerinin *1980* fazlası tutulur.
Aşağıdaki işlev *file_date* yapısı içinde tutulan tarih bilgisini ekrana yazdırıyor:

```
void display_file_date(const file_date *ptr)
{
    printf("%02u/%02u/%u", ptr->day, ptr->mon, ptr->year);
}
```

Bir yapının bit alanı elemanlarına, yine nokta ya da ok işleçleriyle erişilir. Şüphesiz, derleyicinin ürettiği kod bitsel işlemlerin yapılmasını sağlar.
Bu kez zaman bilgisini tutmak için elemanları bit alanları olan bir yapı bildiriliyor:

```
struct file_time {
    unsigned int hour: 5;
    unsigned int min: 6;
    unsigned int sec: 5;
};
```

Zaman bilgisinin saniye değeri için *5* bitlik bir alan ayrılıyor. *5* bitlik bir alanda tutulabilecek en büyük değer *31*'dir. DOS işletim sisteminde, böyle bir bit alanında gerçek saniye değerinin yarısı tutulur.
Aşağıdaki işlev *file_time* yapısı içinde tutulan zaman bilgisini ekrana yazdırıyor:

```
void display_file_date(const file_time *ptr)
{
    printf("%02u:%02u%u", ptr->hour, ptr->min, 2 * ptr->sec);
}
```

Bitalanı Elemanların Adresleri

Bir bitalanı elemanın adresi alınamaz. Bir bit alanı elemanın adres işlecinin terimi olması geçersizdir:

```
#include <stdio.h>

void func();
{
    struct file_date fdate;

    scanf("%d", &fdate.day);   /* Geçersiz */
    /***/
}
```

Bit Alanlarının Bellekte Yerleşimi

Bir bit alanının bellekte yerleşimi konusunda derleyiciler geniş ölçüde serbest bırakılmıştır. Bit alanlarının bellekte yerleşimi "saklama birimi" *(storage unit)* isimli bir terimle açıklanır. Saklama birimi, belirli bir *byte* uzunluğudur ve derleyicinin seçimine bağlıdır. Örneğin derleyicinin seçtiği saklama birimi *1, 2, 4* vs. *byte* olabilir.
Bit alanlarının bildirimiyle karşılaşan derleyici bit alanlarını, aralarında boşluk bırakmadan tek bir saklama birimine yerleştirmeye çalıştırır. Saklama biriminde bir bit alanı içinde yeteri kadar yer kalmaz ise bir sonraki saklama birimine geçilir. Sığmayan elemanın her

iki saklama biriminde mi tutulacağı yoksa tamamen yeni saklama biriminde mi yer alacağı derleyicinin seçimine bırakılmıştır. Bir elemanın ilgili bit alanı içinde yerleşiminin soldan sağa ya da sağdan sola olması da yine derleyicinin seçimine bırakılmıştır.
Bir bit alanı uzunluğu saklama biriminin kendi uzunluğundan daha büyük olamaz. Örneğin saklama birimi *8* bit ise *9* bitlik bir bit alanı eleman kullanılmaz.
Bit alanı elemanların yerleşimi üzerinde daha fazla denetimin sağlanması için, elemana isim vermeme olanağı da getirilmiştir. Bir bit alanı elemana isim verilmeyebilir. İsmi olmayan bir bit alanı elemanı içi derleyici yine gerekli yeri ayırır. Böylece programcı kullanacağı elemanların içsel yerleşimini kendine göre ayarlayabilir.

```
typedef struct {
    int         : 5;
    int hour    : 5;
    int min     : 6;
}Time;
```

Yukarıdaki örnekte *Time* yapısının isim verilmeyen birinci elemanı için, derleyici yine *5* bit ayırır. Yani yapının *hour* isimli elemanı *5*. bitten başlayarak yerleştirilir.
Bir başka olanak da, bit alanı elemanın uzunluğunu *0* olarak belirlemektir. Bunun önceden belirlenmiş özel bir anlamı vardır. Bir elemanın uzunluğunun *0* olduğunu gören derleyici bir sonraki elamanı yeni bir saklama biriminden başlatır:
Aşağıdaki ilgili sistemin saklama birimi uzunluğu bit sayısı olarak elde ediliyor:

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

typedef struct {
    int : 1;
    int : 0;
    int : 1;
}StorageCheck;

int main()
{
    printf("Saklama birimi = %u bit\n", sizeof(StorageCheck) / 2 *
CHAR_BIT);

    return 0;
}
```

Yukarıdaki örnekte *StorageCheck* isimli yapı tanımlanıyor. Bu yapının isim verilmeyen ilk bitalanı elemanı için *1* bit yer ayrıldığını görüyorsunuz. İkinci eleman için ise uzunluk *0* olarak belirleniyor. Son elemanın uzunluğu yine *1* bitdir. Derleyici 3. eleman için gereken yeri bir sonraki saklama biriminden ayırır. Bu durumda eğer saklama birimi *8* bit ise yapının toplam uzunluğu *16* bit, saklama birimi *16* bit ise yapının uzunluğu *32* bit nihayet saklama birimi *32* bit ise yapının uzunluğu *64* bit olur.

KOMUT SATIRI ARGÜMANLARI

Bir program çalıştırıldığı zaman, programın çalışmasını yönlendirecek ya da değiştirecek bazı parametreler, çalışacak programa gönderilmek istenebilir. Programa gönderilecek bir parametre örneğin bir dosya ismi olabileceği gibi, programın değişik biçimlerde çalışmasını sağlayacak bir seçenek de olabilir. *UNIX* işletim sistemindeki *ls* komutunu düşünelim. Eğer *ls* programı

```
ls
```

yazarak çalıştırılırsa, bulunulan dizin içindeki dosyaların isimleri listelenir. Ama *ls* yerine

```
ls -l
```

yazılarak program çalıştırılırsa bu kez yalnızca dosya isimleri değil, dizindeki dosyaların büyüklüğünü, dosyaların sahiplerini, yaratılma tarih ve zamanlarını vs. gösteren daha ayrıntılı bir liste ekranda görüntülenir. *ls* programı yine

```
ls -l sample.c
```

yazarak çalıştırılırsa, yalnızca *sample.c* dosyasına ilişkin bilgiler görüntülenir. İşte bir programı çalıştırırken program isminin yanına yazılan diğer parametrelere komut satırı argümanları *(command line arguments)* denir.

Komut satırı argümanları yalnızca işletim sistemi komutları için geçerli değildir. Tüm programlar için komut satırı argümanları kullanılabilir. Komut satırı argümanlarına C dili standartlarında program parametreleri *(program parameters)* denir.
C programlarının çalışmaya başladığı *main* işlevi de, isteğe bağlı olarak iki parametre değişkenine sahip olabilir. Bu parametre değişkenleri geleneksel olarak *argc* ve *argv* olarak isimlendirilir.

```
int main(int argc, char *argv[])
{
    /***/
}
```

argc "argument count" sözcüklerinden kısaltılmıştır. Komut satırı argümanlarının sayısını gösterir. Bu sayıya programın ismi de dahildir. *argv* ise *"argument vector"* sözcüklerinden kısaltılmıştır. *argv*, dizgeler şeklinde saklanan komut satırı argümanlarını gösteren, *char* türden bir gösterici dizisinin adresini alan göstericidir. Yani *argv* göstericisinin gösterdiği yerde bir gösterici dizisi vardır. Bu gösterici dizisinin her elemanı, komut satırı argümanı olan yazıları tutar. Bu durumda *argv[0]* göstericisi programın ismi olan yazının başlangıç adresini tutar. *argv[1]*'den *argv[argc - 1]* e kadar olan göstericiler ise program ismini izleyen diğer komut satırı argümanı olan yazıların başlangıç adreslerini tutar.
argv[argc] göstericisi ise *NULL* adresi değerine sahiptir.
Yukarıdaki örnekte kullanıcı *ls* programını

```
ls -l sample.c
```

şeklinde çalıştırdığında

argc, *3* değerini alır.
argv[0], program ismini gösterir.
argv[0] = "ls";

argv[1] program ismini izleyen birinci argümanı gösterir.
argv[*1*] = "-l";

argv[2] program ismini izleyen ikinci argümanı gösterir.
argv[2] = "sample.c";
argv[argc] yani *argv[3]* ise *NULL* adresini gösterir.

Bir işlevin, göstericiyi gösteren gösterici (*pointer to pointer*) parametresinin iki ayrı biçimde tanımlanabileceğini anımsayın:

```
void func(char **ptr);
void foo(char *ptr[]);
```

Her iki işlevin de parametre değişkeni *char* türden göstericiyi gösteren bir göstericidir. Bu durumda, komut satırı argümanlarını işleyecek bir *main* işlevinin de göstericiyi gösteren gösterici olan *argv* parametresi, iki farklı biçimde tanımlanabilir:

```
int main(int argc, char **argv)
{
    /***/
}

int main(int argc, char **argv)
{
    /***/
}
```

Komut satırı argümanları boşluklarla birbirinden ayrılmış olmalıdır. Yukarıdaki örnekte program

```
ls -lsample.c
```

biçiminde çalıştırılırsa

argc, *2* değerine sahip olurdu.

Komut satırı argümanları, işletim sistemi tarafından komut satırından alınarak derleyicinin ürettiği giriş kodu yardımıyla *main* işlevine gönderilir.
Aşağıda komut satırı argümanlarını ekrana basan örnek bir program görüyorsunuz:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    for (i = 0; i < argc; ++i)
        printf("argv[%d] : %s\n", i, argv[i]);

    return 0;
}
```

main işlevinde yer alan döngü deyimi aşağıdaki biçimde de yazılabilirdi:

```
for (i = 0; argv[i] != NULL ; ++i)
    printf("argv[%d] : %s\n", i, argv[i]);
```

Komut satırından çalıştırılan programlar genellikle, önce girilen argümanları yorumlar ve sınar. Örneğin bir dosyanın kopyasını çıkarmak üzere, ismi *kopyala* olan bir programın, komut satırından çalıştırılmak istendiğini düşünelim. Program, komut satırından kopyalanacak dosyanın ismini ve yeni yaratılacak dosyanın ismini istesin:

```
kopyala deneme.c test.c
```

Program komut satırından yukarıdaki gibi çalıştırıldığında, ismi *deneme.c* olan bir dosyanın ismi *test.c* olan bir kopyasını oluştursun. Bu durumda *argc 3* degerini alır, değil mi?

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

int main(int argc, char *argv[])
{
    /***/
    if (argc != 3) {
        printf("kullanim : <kopyala> <kaynak dosya ismi> <yeni dosya
ismi>\n");
        exit(EXIT_FAILURE);
    }
    /***/
    return 0;
}
```

Yukarıdaki *main* işlevinde, program *3* komut satırı argümanı verilerek çalıştırılmadıysa, yani eksik ya da fazla argümanla çalıştırıldıysa, program bir hata iletisiyle sonlandırılıyor. Komut satırından argümanlar doğru girilmediği zaman bir program sonlandırılmak zorunda değildir. Dışarıdan doğru olarak girilemeyen argümanların bu kez, programın çalışması durumunda programı kullanan kişi tarafından girilmesi istenebilir. Aşağıdaki örneği inceleyin:

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

#define    NAME_LEN   80

int main(int argc, char *argv[])
{
    char source_name[NAME_LEN];
    char dest_name[NAME_LEN];

    if (argc != 3) {
        printf("kaynak dosyanin ismini giriniz : ");
        gets(source_name);
        printf("kopya dosyanin ismini giriniz : ");
        gets(dest_name);
    }
    else {
        strcpy(source_name, argv[1]);
        strcpy(dest_name, argv[2]);
    }
    /***/
    return 0;
}
```

DOS işletim sisteminde olduğu gibi, bazı sistemlerde *main* işlevi üçüncü bir parametre alabilir. Üçüncü parametre sistemin çevre değişkenlerine ilişkin bir karakter türünden göstericiyi gösteren göstericidir.

```
int main(int argc, char *argv[], char *env[])
{
    /***/
}
```

Aşağıdaki programda, komut satırından girilen sözcükler ters sırada ve tersten yazdırılıyor:

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int k, i;

    for (k = argc - 1; k > 0; --k) {
        for (i = 0; argv[k][i] != '\0'; ++i)
            ;
        for (i--; i >= 0; --i)
            putchar(argv[k][i]);
        putchar('\n');
    }

    return 0;
}
```

Aşağıda, komut satırından çalıştırılacak basit bir hesap makinesi programının kodu veriliyor:

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

int main(int argc, char *argv[])
{
    char ch;
    int op1, op2;

    if (argc != 4) {
        printf("usage : cal Op1 Operator Op2\n");
        exit(EXIT_FAILURE);
    }

    op1 = atoi(argv[1]);
    op2 = atoi(argv[3]);
    ch = argv[2][0];

    printf(" = ");

    switch (ch) {
        case '+' : printf("%d\n", op1 + op2); return 0;
        case '-' : printf("%d\n", op1 - op2); return 0;
        case '/' : printf("%lf\n", (double)op1 / op2); return 0;
        case '*' : printf("%d\n", op1 * op2); return 0;
        case '%' : printf("%lf\n", (double)op1 * op2 / 100); return 0;
        case 'k' : printf("%lf\n", pow(op1, op2)); return 0;
        default  : printf("hatalı işleç\n");
    }
    return 0;
}
```

Aşağıdaki program, komut satırından gün, ay ve yıl değerleri girilen bir tarihin haftanın hangi gününe geldiğini ekrana yazdırıyor:

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

char *days[] = {"Pazar", "Pazartesi", "Sali", "Carsamba", "Persembe",
"Cuma", "Cumartesi"};

int day_of_week(int d, int m, int y)
{
    static int t[] = {0, 3, 2, 5, 0, 3, 5, 1, 4,6, 2, 4};
    y -= m < 3;
    return (y + y / 4 - y / 100 + y / 400 + t[m - 1] + d) % 7;
}

int main(int argc, char **argv)
{
    int day, mon, year;

    if (argc != 4) {
        printf("gun ay ve yil degerini giriniz : ");
        scanf("%d%d%d", &day, &mon, &year);
    }
    else {
        day = atoi(argv[1]);
        mon = atoi(argv[2]);
        year = atoi(argv[3]);
    }
    printf("%s\n", days[day_of_week(day, mon, year)]);

    return 0;
}
```

DOSYALAR

İkincil belleklerde tanımlanmış bölgelere dosya denir. Dosya işlemleri tamamen işletim sisteminin kontrolü altındadır. Her dosyanın bir ismi vardır. Ancak dosyaların isimlendirme kuralları sistemden sisteme göre değişebilir.
İşletim sistemi de bir programdır. Bu program da ayrı ayrı yazılmış işlevlerin birbirlerini çağırması biçiminde çalışır. Örneğin komut satırında bir programın isminin yazılarak çalıştırılması aslında birkaç sistem işlevinin çağrılması ile yapılır. Komut satırından yazılan yazıyı alan, diskte bir dosyayı arayan, bir dosyayı belleğe yükleyen, bellekteki programı çalıştıran işlevler düzenli olarak çağrılır.

İşletim sisteminin çalışması sırasında kendisinin de çağırdığı, sistem programcısının da dışarıdan çağırabildiği işletim sistemine ait işlevlere sistem işlevleri denir. Bu tür işlevlere Windows sisteminde *API (Application Programming Interface)* işlevleri, *UNIX* işletim sisteminde ise sistem çağrıları (*system calls*) denir.
Aslında bütün dosya işlemleri, hangi programlama dili ile çalışılırsa çalışılsın, işletim sisteminin sistem işlevleri tarafından yapılır. Sistem işlevlerinin isimleri ve parametrik yapıları sistemden sisteme değişebilir.

Dosyanın Açılması

Bir dosya üzerinde işlem yapmadan önce dosya açılmalıdır. Dosya açabilmek için işletim sisteminin *"dosya aç"* isimli bir sistem işlevi kullanılır. Dosyanın açılması sırasında dosya ile ilgili çeşitli ilk işlemler işletim sistemi tarafından yapılır.

Bir dosya açıldığında, dosya bilgileri, ismine *"Dosya Tablosu" (File table)* denilen ve işletim sisteminin içinde bulunan bir tabloya yazılır. Dosya tablosunun biçimi sistemden sisteme değişebilir. Örneğin tipik bir dosya tablosu aşağıdaki gibi olabilir:

Dosya tablosu

Sıra No	Dosya ismi	Dosyanın Diskteki Yeri	Dosyanın Özellikleri	Diğerleri
0				
1				
...				
12	AUTOEXEC.BAT	...	...	...
...				

İşletim sisteminin sistem işlevlerinin de parametre değişkenleri, geri dönüş değerleri vardır. "Dosya aç" sistem işlevinin parametresi açılacak dosyanın ismidir.İşlev, dosya tablosunda dosya bilgilerinin yazıldığı sıra numarası ile geri döner ki bu değere *"file handle"* denir. Bu *handle* değeri diğer dosya işlevlerine parametre olarak geçirilir. Dosyanın açılması sırasında buna ek olarak başka önemli işlemler de yapılır.

Dosyanın Kapatılması

Dosyanın kapatılması açılması sırasında yapılan işlemlerin geri alınmasını sağlar. Örneğin dosyanın kapatılması sırasında, işletim sisteminin dosya tablosunda bulunan bu dosyaya ilişkin bilgiler silinir. Açılan her dosya kapatılmalıdır. Bir dosyanın kapatılmaması çeşitli sorunlara yol açabilir.

Dosyaya Bilgi Yazılması ve Okunması

İşletim sistemlerinin dosyaya *n byte* veri yazan ve dosyadan *n byte* veri okuyan sistem işlevleri vardır. Yazma ve okuma işlemleri bu işlevler kullanılarak yapılır.

Dosya Konum Göstericisi

Bir dosya *byte*'lardan oluşur. Dosyadaki her bir *byte* a *0*'dan başlayarak artan sırada bir sayı karşılık getirilir. Bu sayıya ilgili *byte* ın *ofset* numarası denir. Dosya konum göstericisi içsel olarak tutulan *long* türden bir değişkendir, bir ofset değeri yani bir *byte* numarası belirtir. Dosyaya yazan ve dosyadan okuma yapan standart işlevler, bu yazma ve okuma işlemlerini her zaman dosya konum göstericisinin gösterdiği yerden yapar. Bu işlevler, dosyanın neresine yazılacağını ya da dosyanın neresinden okunacağını gösteren bir değer istemez. Örneğin dosya konum göstericisinin gösterdiği yer *100. ofset* olsun. Dosyadan *10* byte bilgi okumak için bir sistem işlevi çağrıldığında, 100. ofsetden başlayarak *10* byte bilgi okunur. İşletim sisteminin dosya göstericisini konumlandıran bir sistem işlevi vardır. Dosya ilk açıldığında dosya konum göstericisi 0. ofseti gösterir. Örneğin bir dosyanın *100. byte* ından başlayarak *10 byte* okunmak istenirse sırası ile şu işlemlerin yapılması gerekir:

İlgili dosya açılır
dosya konum göstericisi *100.* offset'e konumlandırılır
Dosyadan 10 byte okunur
Dosya kapatılır.

C dilinde dosya işlemleri iki biçimde yapılabilir :
1. İşletim sisteminin sistem işlevleri doğrudan çağrılarak.
2. Standart C işlevleri kullanılarak.

Bildirimleri *stdio.h* içinde olan standart dosya işlevlerinin hepsinin ismi *f* ile başlar. Standart C işlevleri da işlemlerini yapabilmek için aslında işletim sisteminin sistem işlevlerini çağırır. İşletim sisteminin sistem işlevleri taşınabilir değildir. İsimleri ve parametrik yapıları sistemden sisteme değişebilir. Bu yüzden standart C işlevlerinin kullanılması tavsiye edilir.

fopen İşlevi

```
FILE *fopen (const char *fname, const char *mode);
```

İşlevin birinci parametresi açılacak dosyanın ismidir. İkinci parametre açış modu bilgisidir. Her iki bilgi de bir yazı olarak işleve iletilir. Dosya ismi yol bilgisi de içerebilir. Dizin geçişleri için '/' karakteri de kullanılabilir. Bir dosya belirli modlarda açılabilir. Açış modu bilgisi, açılacak dosya ile ilgili olarak hangi işlemlerin yapılabileceğini belirler. Yine açış modu bilgisi, açılmak istenen dosyanın var olup olmaması durumunda işlevin nasıl davranacağını belirler. Aşağıdaki tabloda, *fopen* işlevine açış modu bilgisini iletmek üzere geçilmesi gereken yazılar listelenmiştir:

Mod	Anlamı
"w"	Dosya yazmak için açılır. Dosyadan okuma yapılamaz. Dosyanın var olması zorunlu değildir. Dosya yok ise verilen isimde bir dosya yaratılır. Dosya var ise dosya sıfırlanır. Var olan bir dosyayı bu modda açmak dosyanın kaybedilmesine neden olur.
"w+"	Dosyayı hem yazma ve okuma için açar. Dosyanın var olması zorunlu değildir. Dosya yok ise veriln isimde bir dosya yaratılır. Dosya var ise sıfırlanır. Var olan bir dosyayı bu modda açmak dosyanın kaybedilmesine neden olur.
"r"	Dosya okumak için açılır. Dosyaya yazılamaz. Dosya yok ise açılamaz.
"r+"	Dosya hem okuma hem yazma için açılır. Dosya yok ise açılamaz.
"a"	Dosya sona ekleme için açılır. Dosyadan okuma yapılamaz. Dosyanın var olması zorunlu değildir. Dosya yok ise verilen isimde bir dosya yaratılır. Dosya var ise dosya sıfırlanmaz.
"a+"	Dosyayı sonuna ekleme ve dosyadan okuma için açar. Dosyanın var olması zorunlu değildir. Dosya yok ise verilen isimde bir dosya yaratılır. Dosya var ise sıfırlanmaz.

İşlevin geri dönüş değeri *FILE* yapısı türünden bir adrestir. İşlev, açılan dosyaya ilişkin birtakım bilgileri *FILE* yapısı türünden bir nesnenin elemanlarında saklar ve bu nesnenin adresini geri döndürür. İşlevin geri dönüş değerine ilişkin *FILE* yapısı *stdio.h* başlık dosyası içinde bildirilmiştir. Bu yapının elemanları standart değildir. Sistemden sisteme değişiklik gösterebilir. Zaten programcı bu yapının elemanlarına gereksinim duymaz. *fopen* işlevi işletim sisteminin dosya aç sistem işlevini çağırarak dosyayı açar ve dosyaya ilişkin bazı bilgileri *FILE* yapısı türünden bir nesnenin elemanlarına yazarak bu nesnenin başlangıç adresini geri döndürür. Örneğin *"file handle"* değeri de bu yapının içerisindedir. Tabi *fopen* işlevinin geri verdiği *FILE* türünden adres güvenli bir adrestir. Dosya çeşitli sebeplerden dolayı açılamayabilir. Bu durumda *fopen* işlevi *NULL* adresine geri döner. İşlevin geri dönüş değeri kesinlikle kontrol edilmelidir. Tipik bir sınama işlemi aşağıdaki gibi yapılabilir:

```
/*…..*/
FILE *f;

if ((f = fopen("mektup.txt", "r")) == NULL) {
     printf("cannot open file"...\n);
     exit(EXIT_FAILURE);
}
```

Yukarıdaki örnekte ismi *mektup.txt* olan bir dosya okuma amacıyla açılmaya çalışılıyor. Dosya açılmaz ise ekrana bir hata iletisi verilerek, standart *exit* işlevi ile program sonlandırılıyor. Yukarıdaki kod parçasının atama işlecinin ürettiği değerin nesneye atanan değer olmasından faydalandığını görüyorsunuz. Şüphesiz *fopen* işlevi ile açılmak istenen bir dosyanın açılamaması durumunda programın sonlandırılması zorunlu değildir. Ancak bundan sonra verilecek kod örneklerinde, bir dosyanın açılmaması durumunda şimdilik program sonlandırılacak.

Dosya ismi dosyanın yeri hakkında sürücü, yol gibi bilgi içerebilir. Dosya ismi bir dizge ile veriliyorsa yol bilgisi verirken dikkatli olmak gerekir. Yol bilgisi `'\'` (ters bölü) karakteri içerebilir. dizge içinde `'\'` karakterinin kullanılması, `'\ '`karakterininin onu izleyen karakterle birlikte, önceden belirlenmiş ters bölü karakter değişmezi (*escape sequence*) olarak yorumlanmasına yol açabilir. Örneğin :

```
fopen("C:\source\new.dat", "r");
```

Yukarıdaki işlev çağrısında derleyici *'\n'* karakterini *"newline"* karakteri olarak yorumlar *'\s'* karakterini ise *"undefined"* kabul eder. Bu sorundan sakınmak `'\'` karakteri yerine `'\\'` kullanılmasıyla mümkün olur:

```
fopen("C:\\source\\new.dat", "r");
```

Sona ekleme modları *("a", "a+")* çok kullanılan modlar değildir. Dosyaya yazma durumunda *"w"* modu ile *"a"* modu arasında farklılık vardır. *"w"* modunda dosyada olan *byte* ın üzerine yazılabilir. "a" modunda ise dosya içeriği korunarak sadece dosyanın sonuna yazma işlemi yapılabilir.

Bir dosyanın hem okuma hem de yazma amacıyla açılması durumunda yani açış modunu belirten dizgede *'+'* karakterinin kullanılması durumunda dikkatli olmak gerekir. Okuma ve yazma işlemleri arasında mutlaka ya dosya konum göstericisinin konumlandırılması (mesela *fseek* işlevi ile) ya da dosyaya ilişken tampon bellek alanının (*buffer*) boşaltılması gerekir. Bu konuya ileride ayrıntılı bir şekilde değinilecek.

Bir dosyanın açılıp açılamayacağı aşağıdaki küçük programla sınanabilir:
Program komut satırından

```
canopen dosya.dat
```

şeklinde çalıştırıldığında ekrana "*dosya.dat* dosyası açılabilir" ya da "*dosya.dat* dosyası açılamaz" yazar.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    FILE *fp;

    if (argc != 2) {
        printf("usage: canopen filename\n");
        return 1;
    }
    if ((fp = fopen(argv[1], "r")) == NULL) {
        printf("%s dosyası açılamaz\n", argv[1]);
        return 2;
    }
    printf("%s dosyası açılabilir\n", argv[1]);
    fclose(fp);
    return 0;
}
```

stdio.h başlık dosyası içinde *FOPEN_MAX* isimli bir simgesel değişmez tanımlanıyor. *FOPEN_MAX* simgesel değişkeni aynı zamanda açılabilecek en büyük dosya sayısıdır.

fclose İşlevi

Standart bu işlevin *stdio.h* başlık dosyası içindeki bildirimi aşağıdaki gibidir:

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

Bu işlev açılmış ola bir dosyayı kapatır. İşlev *fopen* ya da *fropen* işlevinden elde edilen *FILE* yapısı türünden adresi parametre olarak alır ve açık olan dosyayı kapatır. İşlevin geri dönüş değeri *0* ise dosya başarılı olarak kapatılmış demektir. İşlevin geri dönüş değeri *EOF* ise dosya kapatılamamıştır.
EOF , *stdio.h* başlık dosyası içinde tanımlanan bir simgesel değişmezdir, derleyicilerin çoğunda *(-1)* olarak tanımlanmıştır:

```
#define    EOF    (-1)
```

İşlevin başarısı ancak şüphe altında sınanmalıdır. Normal koşullar altında dosyanın kapatılmaması için bir neden yoktur.

```
#include <stdio.h>

int main()
{
    FILE *f;
    if ((f = fopen("mektup.txt", "w")) == NULL) {
        printf("mektup.txt dosyası yaratılamıyor!\n");
        exit(EXIT_FAILURE);
    }
    fclose(f);
    printf("mektup.txt dosyasi kapatildi!\n");
    return 0;
}
```

Derleyicilerin çoğunda bulunmasına karşılık standart olmayan *fcloseall* isimli bir işlev de vardır:

```
int fcloseall(void);
```

Bu işlev çağrıldığında açık olan dosyaların hepsi kapatılır. İşlevin geri dönüş değeri, kapatılan dosya sayısıdır.

Şimdi de açılmış bir dosyadan okuma yapan ya da açılmış bir dosyaya yazma yapan standart C işlevlerini inceleyelim:

fgetc İşlevi

```
int fgetc(FILE *f);
```

C'nin standart yazma ve okuma yapan işlevleri yazılan ve okunan ofset sayısı kadar dosya konum göstericisini ilerletirler. *fgetc* işlevi dosya göstericisinin gösterdiği yerdeki *byte* ı okur ve bu *byte* ın tamsayı değerini geri dönüş değeri olarak verir. İşlev başarısız olursa, yani okuma işlemi yapılamaz ise *stdio.h* dosyası içinde simgesel değişmez olarak tanımlanmış *EOF* değerine geri döner.

fgetc işlevinin geri dönüş değerini *char* türden bir değişkene atamak yanlış sonuç verebilir, bu konuda dikkatli olunmalı ve işlevin geri dönüş değeri *int* türden bir değişkende saklanmalıdır.

```
char ch;
ch = fgetc(fp);
```

Yukarıda dosyadan okunan karakterin *255* numaralı *ASCII* karakteri *(0x00FF)* olduğunu düşünelim. Bu sayı *char* türden bir değişkene atandığında yüksek anlamlı *byte* ı kaybedilerek *ch* değişkenine *0xFF* değeri atanır. Bu durumda *ch* değişkeni işaretli *char* türden olduğundan *ch* değişkeni içinde negatif bir tamsayının tutulduğu anlamı çıkar.

```
if (ch == EOF)
```

gibi bir karşılaştırma deyiminde, *if* ayracı içindeki karşılaştırma işleminin yapılabilmesi için otomatik tür dönüşümü yapılır, yani *ch* tam sayıya yükseltilir (*integral promotion*). Bu otomatik tür dönüşümünde işaretli *int* türüne çevrilecek *ch* değişkeni negatif olduğu için *FF byte*'ı ile beslenir. Bu durumda eşitlik karşılaştırması doğru sonuç verir, yani dosyanın sonuna gelindiği (ya da başka nedenden dolayı okumanın yapılamadığı) yorumu yapılır.

Oysa *ch* değişkeni *int* türden olsaydı, *ch* değişkenine atanan değer *0x00FF* olurdu. Bu durumda karşılaştırma yapıldığında *ch* değişkeni ile *EOF* değerinin (*0xFFFF*) eşit olmadığı sonucuna varılırdı.

fgetc işlevi kullanılarak okuma amacıyla açılmış bir dosya karakter karakter okunabilir. Aşağıdaki programda klavyeden isimi alınan bir dosyanın içeriği ekrana yazdırılıyor:

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

#define    FILENAME_LEN       256

int main()
{
    FILE *f;
    char file_name[FILENAME_LEN];
    int ch;

    printf("Yazdirilacak dosyanın ismi : ");
    gets(file_name);
    if ((f = fopen(file_name, "r")) == NULL) {
        printf("cannot open file...\n");
        exit(EXIT_FAILURE);
    }
    while ((ch = fgetc(f)) != EOF)
        putchar(ch);
    fclose(f);

    return 0;
}
```

Yukarıdaki kodu inceleyin. Önce yazdırılacak dosyanın ismi klavyeden alınarak *file_name* isimli diziye yerleştiriliyor. İsmi *file_name* dizisine alınan dosya *fopen* işlevine yapılan çağrı ile okuma amacıyla açılıyor. *fopen* işlevinin başarısız olması durumunda program sonlandırılıyor.

```
while ((ch = fgetc(f)) != EOF)
     putchar(ch);
```

döngü deyimiyle, *fgetc* işlevi *EOF* değerini döndürünceye kadar dosyadan bir karakter okunuyor ve okunan karakterin görüntüsü standart *putchar* işleviyle ekrana yazdırılıyor. Okuma işlemi tamamlanınca standart *fclose* işleviyle dosya kapatılıyor.

Şimdi de komut satırından çalıştırılacak aşağıdaki programı inceleyin. İşlev komut satırından

```
<say> <dosya ismi> <karakter>
```

biçiminde çalıştırılır. Program ekrana ismi verilen dosyada üçüncü komut satırı argümanı olarak verilen karakterden kaç tane bulunduğu bilgisini yazar. Programı önce inceleyin daha sonra derleyerek çalıştırın:

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

#define    FILENAME_LEN       256

int main(int argc, char **argv)
{
    FILE *f;
    char file_name[FILENAME_LEN];
    int ch, cval;
    int char_counter = 0;

    if (argc != 3) {
        printf("Dosya ismi : ");
        gets(file_name);
        printf("sayilacak karakter : ");
        cval = getchar();
    }
    else {
        strcpy(file_name, argv[1]);
        cval = *argv[2];
    }
    if ((f = fopen(file_name, "r")) == NULL) {
        printf("%s dosyasi acilamiyor!\n", file_name);
        exit(EXIT_FAILURE);
    }
    while ((ch = fgetc(f)) != EOF)
        if (ch == cval)
            char_counter++;

    fclose(f);
    printf("%s dosyasinda %d adet %c karakteri var!\n", file_name,
    har_counter, ch);

    return 0;
}
```

Son olarak şunu da ekleyelim. *fgetc* bir işlevdir. Ancak *stdio.h* başlık dosyası içinde *getc* isimli bir de makro tanımlanmıştır. Derleyicilerin çoğu *getc* makrosuyla aynı isimli bir de işlev tanımlar. Ancak *fgetc* bir makro değil işlevdir. Makrolar konusuna ilerdeki bölümlerde değinilecek.

fputc İşlevi

```
int fputc(int ch, FILE *p);
```

Bu işlev dosya konum göstericisinin bulunduğu yere *1* byte bilgiyi yazar. İşlevin birinci parametresi yazılacak karakter, ikinci parametresi ise yazılacak dosyaya ilişkin *FILE* yapısı adresidir. İşlevin geri dönüş değeri *EOF* ise yazma işlemi başarısız olmuş demektir. Yazma işlemi başarılı olmuşsa işlev yazılan karakterin sıra numarası ile geri döner.
Sık yapılan bir hata işleve gönderilecek argumaların sırasını karıştırmaktır. Örneğin *f FILE* yapısı türünden bir nesneyi gösteren gösterici olsun. *fputc* işlevi

```
fputc('A', f)
```

yerine yanlışlıkla

```
fputc(f, 'A');
```

biçiminde çağrılırsa *f* adresi *int* türüne ve *'A'* tamsayı değeri de *FILE* yapısı türünden bir adrese dönüştürülür. Bu durumda şüphesiz yazma işlemi başarısız olur.
Aşağıda *fgetc* ve *fputc* işlevlerini kullanarak bir dosyanın kopyasını çıkartan bir programın kodunu görüyorsunuz. Program komut satırından

```
<kopyala> <kaynak dosya ismi> <hedef dosya ismi>
```

şeklinde çalıştırılır. Programın çalıştırılmasıyla ikinci komut satırı argümanıyla ismi verilen dosyanın üçüncü komut satırı argümanıyla verilen isimli bir kopyası çıkartılır. Programı dikkatle inceleyin:

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

#define MAX_FILE_NAME_LEN     256

int main(int argc, char **argv)
{
    FILE *fsource, *fdest;
    char source_name[MAX_FILE_NAME_LEN];
    char dest_name[MAX_FILE_NAME_LEN];
    int ch;
    int byte_counter = 0;

    if (argc != 3) {
        printf("kopyalanacak dosyanın ismi : ");
        gets(source_name);
        printf("kopya dosyanın ismi : ");
        gets (dest_name);
    }
    else {
        strcpy(source_name, argv[1]);
        strcpy(dest_name, argv[2]);
    }
    if ((fsource = fopen(source_name, "r")) == NULL) {
        printf("%s dosyasi acilamiyor\n", source_name);
        exit(EXIT_FAILURE);
    }
    printf("%s dosyasi acildi!\n", source_name);

    if ((fdest = fopen(dest_name,  "w")) == NULL) {
        printf("%s dosyasi yaratilamiyor\n", dest_name);
        fclose(fsource);
        exit(EXIT_FAILURE);
        }
    printf("%s dosyasi yaratildi!\n", dest_name);
    while ((ch = fgetc(fsource)) != EOF) {
        fputc(ch, fdest);
        byte_counter++;
    }
    fclose(fsource);
    printf("%s dosyasi kapatildi!\n", source_name);
    fclose(fdest);
    printf("%s dosyasi kapatildi!\n", dest_name);
    printf("%d uzunlugunda %s dosyasinin %s isimli kopyasi cikarildi!\n",
    byte_counter,  source_name, dest_name);

    return 0;
}
```

fputc bir işlevdir. Ancak *stdio.h* başlık dosyası içinde *getc* isimli bir de makro tanımlanmıştır. Derleyicilerin çoğu *putc* makrosuyla aynı isimli bir de işlev tanımlar. Ancak *fgetc* bir makro değil işlevdir.

fprintf İşlevi

```
int fprintf(FILE *, const char *, …);
```

Bu işlev tıpkı *printf* işlevi gibidir. Ancak ilk parametresi yazma işleminin hangi dosyaya yapılacağını belirtir. Diğer parametreleri *printf* işlevin de olduğu gibidir. *printf* işlevi ekrana yazarken, *fprintf* işlevi birinci parametre değişkeninde belirtilen dosyaya yazar. Aşağıdaki örneği inceleyin:

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

int main()
{
    FILE *f;
    int i, ch;

    if ((f = fopen("data.txt", "w")) ==NULL) {
        printf("cannot open file...\n");
        exit(EXIT_FAILURE);
    }
    printf("data.txt dosyasi yaratıldı!\n");
    for (i = 0; i < 10; ++i)
        fprintf(f, "sayi = %d\n", i);
    fclose(f);
    printf("data.txt dosyası kapatildı!\n");
    if ((f = fopen("data.txt", "r")) == NULL) {
        printf("cannot open file...\n");
        exit(EXIT_FAILURE);
    }
    printf("data.txt dosyasi okuma amacıyla açıldı!\n");
    while ((ch = fgetc(f)) != EOF)
        putchar(ch);
    fclose(f);

    return 0;
}
```

Programın ekran çıktısı:

```
data.txt dosyasi yaratildı!
data.txt dosyasi kapatildı!
data.txt dosyasi okuma amacıyla açıldı!
sayi = 0
sayi = 1
sayi = 2
sayi = 3
sayi = 4
sayi = 5
sayi = 6
sayi = 7
sayi = 8
sayi = 9
```

fscanf İşlevi

Dosyadan okuma yapan bir işlevdir. *scanf* işlevine çok benzer.

Nasıl *fprintf* işlevi, ekrana yazmak yerine, yazma işlemini birinci parametresiyle belirlenen dosyaya yapıyorsa, *fscanf* işlevi de, okumayı klavyeden yapmak yerine belirtilen bir dosyadan yapar. *fscanf* işlevi *scanf* işlevinden farklı olarak, birinci parametre değişkenine *FILE* türünden bir adres alır. İşlevin bildirimi aşağıdaki gibidir:

```
int fscanf(FILE *, const char *, …);
```

İşlevin geri dönüş değeri, dosyadan okunarak bellekteki alanlara yazılan değer sayısıdır. Hiçbir alana yazma yapılmadıysa, işlev *0* değerine geri döner. Eğer ilk alana atama yapılamadan dosyanın sonuna gelinmişse, ya da bir hata oluşmuşsa işlev *EOF* değerine geri döner. İkinci parametresine geçilecek yazıda kullanılacak format karakterleri *scanf* işlevi ile aynıdır. Aşağıdaki programı dikkatle inceleyin:

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

#define    SIZE              100
#define    PASS_GRADE        60

char *name[50] = {"Ali", "Veli", "Hasan", "Necati", "Ayşe", "Kaan",
"Selami","Salah","Nejla", "Huseyin", "Derya", "Funda", "Kemal", "Burak",
"Ozlem", "Deniz", "Nuri", "Metin", "Guray",  "Anil", "Umut", "Selda",
"Belgin", "Figen", "Korhan", "İhsan", "Ufuk", "Necmettin", "Taylan",
"Abdullah", "Perihan", "Soner", "Can", "Ata", "Berk", "Melahat", "Zarife",
"Yelda", "Ertan", "Mustafa", "Gizem", "Funda", "Aleyna", "Simge", "Damla",
"Kaan", "Kerim", "Cumali", "Ferda", "Sami"};
char *fname[30] = {"Aslan", "Ozkan", "Eker", "Ergin", "Serce", "Kaynak",
"Acar",  "Aymir", "Erdin", "Doganoglu", "Avsar", "Ozturk", "Yılmaz",
"Tibet", "Arkın", "Cilasun",  "Yildirim", "Demiroglu", "Torun", "Polatkan",
"Burakcan", "Kale", "Nergis", "Kayserili",  "Duman", "Tansel",  "Kurt",
"Tonguc", "Melek", "Mungan"};

int main()
{
    FILE *fgrades, *fpass, *ffail;
    int no_of_lines, k;
    char name_entry[SIZE];
    char fname_entry[SIZE];
    int grade;
    int pass_counter = 0;
    int fail_counter = 0;

    srand(time(0));
    fgrades = fopen("notlar.txt", "w");

    if (fgrades == NULL) {
        printf("notlar.txt dosyasi yaratilamiyor!\n");
        exit(EXIT_FAILURE);
    }

    printf("notlar.txt dosyasi yaratildi!\n");
    no_of_lines = rand() % 2000 + 1000;
    for (k = 0; k < no_of_lines; ++k)
        fprintf(fgrades, "%s %s %d\n", name[rand() % 50], fname[rand() %
30], rand() % 101);
    printf("notlar.txt dosyasina %d satir kayıit yazildi!\n", no_of_lines);
    fclose(fgrades);
    printf("notlar.txt dosyasi kapatildi!\n");
    fgrades = fopen("notlar.txt", "r");
    if (fgrades == NULL) {
```

```
        printf("cannot open notlar.txt!\n");
        exit(EXIT_FAILURE);
    }
    printf("notlar.txt dosyasi acildi!\n");
    fpass = fopen("gecen.txt", "w");
    if (fpass == NULL) {
        printf("gecen.txt dosyasi yaratilamiyor!\n");
        exit(EXIT_FAILURE);
    }

    printf("gecen.txt dosyasi yaratildi!\n");
    ffail = fopen("kalan.txt", "w");
    if (ffail == NULL) {
        printf("kalan.txt dosyasi yaratilamiyor!\n");
        exit(EXIT_FAILURE);
    }

    printf("kalan.txt dosyasi yaratildi!\n");
    while (fscanf(fgrades, "%s%s%d", name_entry, fname_entry, &grade) !=
EOF) {
        if (grade >= PASS_GRADE) {
            fprintf(fpass, "%s %s %d\n", name_entry, fname_entry, grade);
            pass_counter++;
        }
        else {
            fprintf(ffail, "%s %s %d\n", name_entry, fname_entry, grade);
            fail_counter++;
        }
    }
    fprintf(fpass, "TOPLAM GECEN OGRENCI SAYISI = %d\n", pass_counter);
    fprintf(ffail, "TOPLAM KALAN OGRENCI SAYISI = %d\n", fail_counter);
    if (fcloseall() != 3) {
        printf("dosya kapatilamiyor!\n");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

Programda her satırı rastgele bir isim soyisim ve not değerinden oluşan *"notlar.txt"* isimli bir dosya yaratılıyor. Daha sonra bu dosya okuma amacıyla açılarak, dosyanın her satırında bulunan isim ile soyisimler *char* türden dizilere, not değerleri ise *int* türden bir değişkene okunuyor. Okunan not değerinin *PASS_GRADE* değerinden yüksek olup olmamasına göre ilgili satır, yazma amacıyla açılmış *"gecen.txt"* ya da *"kalan.txt"* isimli dosyalara yazdırılıyor. Böylece *"notlar.txt"* dosyasından *"gecen.txt"* ve *"kalan.txt"* isimli iki farklı dosya oluşturuluyor.

fscanf işlevinde kullanılan format karakterlerine ilişkin önemli bir ayrıntıya değinelim: Dosyadan yapılan her okumanın dönüştürülerek mutlaka bellekte bir alana yazılması zorunlu değildir. Boşluk karakterleriyle ayrılan bir karakter grubu bir yere atanmadan dosya tampon alanından çıkarılmak isteniyorsa, format karakter grubunda % karakterinden sonra '*' karakteri kullanılır. Örneğin bir metin dosyasında dosya konum göstericisinin gösterdiği yerde

```
1376  ----------   4567
```

gibi bir satırın bulunduğunu düşünelim. Yapılacak okuma sonucunda yalnızca *1376* ve *4567* değerlerinin *x* ve *y* değişkenlerine aktarılması gerektiğini düşünelim. Bunun için aşağıdaki gibi bir çağrı yapılabilir:

```
fscanf(f, "%d%*s%d", &x, &y);
```

fgets İşlevi

```
char *fgets(char *buf, int n, FILE *f);
```

Bu işlev dosya konum göstericisinin gösterdiği yerden bir satırlık bilgiyi okur. İşlev dosyadan *'\n'* karakterini okuyunca onu da birinci parametresinde verilen adrese yazarak işlemini sonlandırır.

İşlevin birinci parametresi okunacak bilginin bellekte yerleştirileceği yerin adresidir. İkinci parametresi ise okunacak maksimum karakter sayısıdır. *fgets* işlevi en fazla *n – 1* karakteri okur. Okuduğu karakterlerin sonuna sonlandırıcı karakteri ekler ve işlemini sonlandırır. Eğer satır üzerindeki karakter sayısı *n - 1*'den az ise tüm satırı okur ve işlemini sonlandırır. Örneğin bu parametrenin *10* olarak girildiğini düşünelim. Satır üzerinde *20* karakter olsun. İşlev *9* karakteri okuyarak diziye yerleştirir, sonuna sonlandırıcı karakteri ekler. Ancak satır üzerinde *\n* dahil olmak üzere *5* karakter olsaydı işlev bu *5* karakteri de okuyarak sonuna da sonlandırıcı karakteri ekleyerek işlemini sonlandırırdı. İşlevin ikinci parametresine *char* türden bir dizinin ya da dinamik olarak elde edilen bir bloğun boyutunu geçmek taşma hatalarını doğrudan engeller. Zira *fgets* işlevi en fazla, dizinin boyutundan bir eksik sayıda karakteri okuyarak diziye yazar, dizinin son elemanına da sonlandırıcı karakterin değerini yazar.
İşlevin geri dönüş değeri, en az *1* karakter okunmuş ise birinci parametresi ile belirtilen adresin aynısı, hiçbir karakter okunmamışsa *NULL* adresidir.

Bir döngü içinde *fgets* işlevi sürekli olarak çağrılarak bütün dosya okunabilir.

fgets işlevi ile bir dosyayı satır satır ekrana yazdıran aşağıdaki programı inceleyin:

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

#define    MAX_FILE_NAME_LEN         256
#define    BUFFER_SIZE               100

int main()
{
    FILE *f;
    char file_name[MAX_FILE_NAME_LEN];
    char buf[BUFFER_SIZE];
    printf("Dosya ismi : ");
    gets(file_name);
    if ((f = fopen(file_name, "r")) == NULL) {
        printf("cannot open the file %s\n", file_name);
        exit(EXIT_FAILURE);
    }
    while (fgets(buf, BUFFER_SIZE, f) != NULL)
        printf("%s", buf);
    fclose(f);

    return 0;
}
```

```
while (fgets(buf, BUFFER_SIZE, f) != NULL)
    printf("%s", buf);
```

Döngüsüyle *f* ile gösterilen dosyadan satır satır okuma yapılarak okunan karakterler *buf* dizisine yazılıyor. Eğer dosyadan okunacak bir karakter kalmadıysa *fgets* işlevi *NULL* adresine geri döner.

fputs İşlevi

İşlevin bildirimi aşağıdaki gibidir:

```
int fputs(const char *str, FILE *stream);
```

İşlev, birinci parametresine geçilen adresteki yazıyı ikinci parametresine geçilen *FILE* türünden adresle ilişkilendirilen dosyaya yazar. Yazının sonunda yer alan sonlandırıcı karakter işlev tarafından dosyaya yazılmaz. İşlev, yazma işleminde başarısız olursa *EOF* değerine geri döner. Başarı durumunda işlevin geri dönüş değeri negatif olmayan bir tamsayıdır. Aşağıdaki programda, *fputs* ve *fgets* işlevleri kullanılarak bir metin dosyasının kopyası çıkarılıyor:

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

#define FILE_NAME_LEN      256
#define BUFFER_SIZE        100

int main()
{
    char source_file_name[FILE_NAME_LEN];
    char dest_file_name[FILE_NAME_LEN];
    char buffer [BUFFER_SIZE];
    FILE *fs, *fd;

    printf("kopyalanacak dosya ismi: ");
    gets(source_file_name);
    printf("kopya dosya ismi: ");
    gets(dest_file_name);
    fs = fopen(source_file_name, "r");
    if (fs == NULL) {
        printf("%s dosyasi acilamiyor!\n", source_file_name);
        exit(EXIT_FAILURE);
    }
    fd = fopen(dest_file_name, "w");
    if (fd == NULL) {
        printf("%s dosyasi yaratilamiyor!\n", dest_file_name);
        fclose(fd);
        exit(EXIT_FAILURE);
    }
    while (fgets(buffer, BUFFER_SIZE, fs))
        fputs(buffer, fd);
    printf("kopyalama basari ile tamamlandi!\n");
    fclose(fs);
    fclose(fd);

    return 0;
}
```

Metin Dosyaları ve İkilik Dosyalar

Bir dosya *text* modunda ya da *binary* modda açılabilir. Varsayılan (*default)* açış modu text modudur. Yani dosyanın hangi modda açıldığı açık bir şekilde belirtilmezse dosyanın text modunda açıldığı varsayılır. Dosyayı *binary* modda açabilmek için açış mod yazısına *'b'* eklemek gerekir. Aşağıda bir dosyayı *binary* modda açabilmek için *fopen* işlevine gönderilebilecek geçerli dizgeler veriliyor:

```
"rb", "r+b", "rb+", "w+b", "wb+","a+b", "ab+"
```

DOS ve *WINDOWS* gibi bazı işletim sistemlerinde bir dosyanın *text* modu ya da *binary* modda açılması arasında bazı önemli farklar vardır:

DOS işletim sisteminde bir dosya yazdırıldığında bir karakter aşağı satırın başında görünüyorsa bunu sağlamak için o karakterden önce *CR (carriage return)* ve *LF (line feed)* karakterlerinin bulunması gerekir. *CR* karakteri C'de '\r' ile belirtilir. *13* numaralı *ASCII* karakteridir. *LF* karakteri C'de *n* ile belirtilir. *10* numaralı *ASCII* karakteridir. Örneğin bir dosya yazdırıldığında görüntü

```
a
b
```

şeklinde olsun. Dosyadaki durum *a\r\nb* şeklindedir. Oysa *UNIX* tabanlı sistemlerinde aşağı satırın başına geçebilmek için sadece *LF* karakteri kullanılır. *UNIX* işletim sisteminde

```
a
b
```

görüntüsünün dosya karşılığı

```
a\nb biçimindedir.
```

DOS işletim sisteminde LF karakteri bulunulan satırın aşağısına geç *CR* karakteri ise bulunulan satırın başına geç anlamındadır. Örneğin *DOS işletim sisteminde* bir bir dosyanın içeriği *a\nb* biçiminde ise dosya yazdırıldığında

```
a
b
```

görüntüsü elde edilir. Eğer dosyanın içeriği *a\rb* biçiminde ise dosya yazdırıldığında

```
b
```

görüntüsü elde edilir.

printf İşlevinda *n* ekranda aşağı satırın başına geçme amacıyla kullanılır. Aslında *printf* işlevinin 1. parametresi olan dizgenin içine *n* yerleştirildiğinde *UNIX*'de yalnızca \ni DOS işletim sisteminde ise \r ve\n ile bu geçiş sağlanır.
Text dosyaları ile rahat çalışabilmek için dosyalar *text* ve *binary* olarak ikiye ayrılmıştır. Bir dosya text modunda açıldığında dosyaya \n karakteri yazılmak istendiğinde dosya işlevleri otomatik olarak \r ve \n karakterlerini dosyaya yazar. Benzer bir biçimde dosya text modda açılmışsa dosya göstericisi \r\n çiftini gösteriyorsa dosyadan yalnızca /n karakteri okunur. *DOS* işletim sisteminde text ve binary dosyalar arasındaki başka bir fark da, *CTRL Z* (*26* numaralı *ASCII* karakterinin) dosyayı sonlandırdığının varsayılmasıdır. Oysa dosya binary modda açıldığında böyle bir varsayım yapılmaz.

UNIX işletim sisteminde, text modu ile *binary* mod arasında hiçbir fark yoktur. Yani *UNIX* işletim sisteminde dosyanın binary mod yerine text modunda açılmasının bir sakıncası olmaz. Ancak *DOS* altında text dosyası olmayan bir dosyanın binary mod yerine text modunda açılmasının sakıncaları olabilir. Örneğin *DOS* altında bir *exe* dosyanın binary mod yerine text modunda açıldığını düşünelim. Bu dosyada *10* numaralı ve *13* numaralı *ASCII* karakterleri yanyana bulunduğunda dosyadan yalnızca *1* byte okunur. Aynı şekilde dosyadan *26* numaralı *ASCII* karakteri okunduğunda dosyadan artık başka bir okuma yapılamaz. Dosyanın sonuna gelindiği varsayılır.
Aşağıdaki program text modu ile *binary* mod arasındaki farkı gösteriyor:

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

int main()
{
    FILE *fp;
    int k, ch;
    fp = fopen("deneme", "w");
    if (fp == NULL) {
        printf("dosya açılamıyor\n");
        exit(EXIT_FAILURE);
    }
/* dosyaya 5 tane \n karakteri yazdırılıyor */
    for (k = 0; k < 5; ++k)
        fputc('\n', fp);
    fclose(fp);
    printf("\ndosya binary modda açılarak yazdırılıyor\n");
    fp = fopen("deneme", "rb");
    if (fp == NULL) {
        printf("dosya açılamıyor\n");
        exit(EXIT_FAILURE);
    }
    while ((ch = fgetc(fp)) != EOF)
        printf("%d ", ch);
/* ekran çıktısı
13 10 13 10 13 10 13 10 13 10
*/
    fclose(fp);
    printf("\ndosya kapatıldı. Şimdi dosya text modunda açılarak
    yazdırılıyor .\n");
    fp = fopen("deneme", "r");
    if (fp == NULL) {
        printf("dosya açılamıyor\n");
        exit(EXIT_FAILURE);
    }
    while ((ch = fgetc(fp)) != EOF)
        printf("%d ", ch);
/* ekran çıktısı
10  10  10  10  10
*/
    fclose(fp);
/* simdi '\x1A' karakterinin text modunda dosyayı sonlandırması özelliği
sınanıyor*/
    fp = fopen("deneme", "w");
    if (fp == NULL) {
        printf("dosya açılamıyor\n");
        exit(EXIT_FAILURE);
    }

/* dosyaya 5 tane 'A' karakteri yazdırılıyor */
    for (k = 0; k < 5; ++k)
        fputc('A', fp);

/* dosyaya '\x1A' karakteri yazdırılıyor */
    fputc('\x1a', fp);
/* dosyaya 10 tane 'A' karakteri yazdırılıyor. */
    for (k = 0; k < 5; ++k)
        fputc('A', fp);
    fclose(fp);
    printf("\ndosya binary modda açılarak yazdırılıyor :\n");
    fp = fopen("deneme", "rb");
```

```
    if (fp == NULL) {
        printf("dosya açılamıyor\n");
        exit(EXIT_FAILURE);
    }
    while ((ch = fgetc(fp)) != EOF)
        printf("%d ", ch);

/* ekran çıktısı
65 65 65 65 65 26 65 65 65 65 65
*/
    printf("\ndosya kapatıldı, Şimdi dosya text modunda açılarak
    yazdırılıyor\n");
    fp = fopen("deneme", "r");
    if (fp == NULL) {
        printf("dosya açılamıyor\n");
        exit(EXIT_FAILURE);
    }
    while ((ch = fgetc(fp)) != EOF)
        printf("%d ", ch);
/* ekran çıktısı
65 65 65 65 65 26 65 65 65 65 65
*/
    fclose(fp);

    return 0;
}
```

EOF Durumu

Dosyanın sonunda hiçbir özel karakter yoktur. İşletim sistemi dosyanın sonuna gelinip gelinmediğini dosyanın uzunluğuna bakarak anlayabilir. *EOF (end of file)* durumu dosya konum göstericisinin dosyada olmayan son karakteri göstermesi durumudur. *EOF* durumunda dosya konum göstericisinin *offset* değeri dosya uzunluğu ile aynı değerdedir. *EOF* durumunda dosyadan okuma yapılmak istenirse dosyadan okuma yapan işlevler başarısız olur. Ancak açış modu uygunsa dosyaya yazılabilir ve bu durumda dosyaya ekleme yapılır.

Daha önce belirtildiği gibi C dilinde açılan bir dosya ile ilgili bilgiler *FILE* türünden bir yapı nesnesi içinde tutulur. Bu yapının elemanları dosyanın özellikleri hakkında bilgi verir. C programcısı bu yapının elemanlarının değerleri ile doğrudan ilgilenmez, zira *fopen* işlevinin geri dönüş değeri bu yapı nesnesini gösteren *FILE* yapısı türünden bir göstericidir ve C dilinin dosyalarla ilgili işlem yapan işlevleri çoğunlukla bu adresi parametre olarak alarak, istenilen dosya ile ilgili bilgilere ulaşır.
Söz konusu *FILE* yapısının elemanlarından biri de, bayrak olarak kullanılan *EOF* bayrağıdır. Aslında derleyicilerin çoğunda *int* türden bir bayrağın yalnızca belirli bir bitidir. C dilinin dosyalarla ilgili işlem yapan bazı işlevleri *EOF* bayrağının değerini değiştirir. Yani *EOF* bayrağını birler ya da sıfırlarlar. Okuma yapan işlevler okumadan önce bu bayrağın değerine bakar. EOF bayrağı set edilmişse okuma başarılı olmaz. Başarılı bir okuma yapılabilmesi için *EOF* bayrağının yeniden sıfırlanması gerekir. Dosya açan işlevler *FILE* yapısındaki *EOF* bayrağını sıfırlar. Bu işlevler dışında dosya konum göstericisinin değerini değiştiren işlevler *(fseek, rewind, fsetpos)* ile *clearerr* işlevleri de *EOF* bitini sıfırlar.

Formatlı ve Formatsız Yazım

Bir dosyaya *int* türden değerlerin yazılacağını düşünelim. Dosyaya *int* türden bir değer yazmak ne anlama gelir? Örneğin *int* türünün *4* byte olduğu bir sistemde, dosyaya yazılacak tamsayı değeri *1234567890* olsun. Bu değer bir dosyaya yazılmak ve o dosyadan daha sonra geri okunmak istensin.
Bu işlem *fprintf* işlevi ile yapılabilir, değil mi?

f yazma amacıyla açılmış bir dosyanın göstericisi olmak üzere

```
fprintf(f, "%d", 1234567890);
```

Yukarıdaki çağrıyla dosya konum göstericisinin gösterdiği yerden başlanarak dosyaya *10 byte* eklenmiş olur. Dosyaya yazılan *byte*'lar *1234567890* sayısının basamak değerlerini gösteren rakamların sıra numaralarıdır. Sistemde *ASCII* karakter kodlamasının kullanıldığını düşünelim. Dosyaya aşağıdaki *byte*'lar yazılmıştır:

```
49 50 51 42 53 54 55 56 57 59 48
```

Oysa 32 bitlik bir sistemde *1234567890* gibi bir değer *4 byte'lık* alanda ifade edilir değil mi? Örneğin *1234567890* sayısının bellekteki görüntüsü aşağıdaki gibidir:

```
0100 1001 1001 0110 0000 0010 1101 0010
```

Bu *4 byte RAM*'de olduğu gibi dosyaya yazılamaz mı? İşte dosyaya bu şekilde yazım formatsız yazımdır. C'nin iki önemli standart işlevi *RAM*'den dosyaya dosyadan *RAM*'e belirli sayıda *byte*'ı formatsız biçimde aktarır. Şimdi bu işlevleri inceleyeceğiz:

fread ve fwrite İşlevleri

Bu iki işlev C'de en çok kullanılan işlevlerdir. Genel olarak dosya ile *RAM* arasında aktarım (*transfer*) yaparlar. Her iki işlevin de bildirimi aynıdır:

```
size_t fread(void *adr, size_t block_size, size_t n_blocks, FILE *);
size_t fwrite(const void *adr, size_t block_size, size_t n_blocks, FILE *);
```

size_t türünün sistemlerin hemen hemen hepsinde *unsigned int* ya da *unsigned long* türünün *typedef* edilmiş yeni ismi olduğunu anımsayın.

fread işlevi dosya konum göstericisinin gösterdiği yerden, ikinci ve üçüncü parametresine kopyalanan değerlerin çarpımı kadar *byte*'ı , bellekte birinci parametresinin gösterdiği adresten başlayarak kopyalar. Genellikle işlevin ikinci parametresi veri yapısının bir elemanının uzunluğunu, üçüncü parametresi ile parça sayısı biçiminde girilir. İşlevin geri dönüş değeri belleğe yazılan ya da bellekten dosyaya yazılan parça sayısıdır.

Bu işlevler sayesinde diziler ve yapı nesneleri tek bir çağrı ile bir dosyaya aktarılabilirler. Örneğin *10* elemanlı *int* türden bir dizi aşağıdaki gibi tek bir çağrıyla dosyaya yazılabilir.

```
int a[5] = {3, 4, 5, 7, 8};
fwrite (a, sizeof(int), 5, f);
```

Yukarıdaki örnekte, dizi ismi olan *a* int türden bir adres bilgisi olduğu için, *fwrite* işlevine 1. argüman olarak gönderilebilir. *FILE* türünden *f* göstericisi ile ilişkilendirilen dosyaya bellekteki *a* adresinden toplam *sizeof(int) * 5 byte* yazılır.

Aşağıdaki kod parçasında bir yapı nesnesi bellekten dosyaya aktarılıyor:

```
typedef struct {
    char name[20];
    char fname[20];
    int no;
}Person;

int main()
{
    FILE *f;
    Person per = {"Necati", "Ergin", 325);

    f = fopen("person.dat", "wb");
    /*******************************/
    fwrite(&per, sizeof(Person), 1, f);
    /****/
}
```

fwrite işlevi sayıları bellekteki görüntüsü ile dosyaya yazar. Yani *fprintf* işlevi gibi formatlı yazmaz. Örneğin *DOS* işletim sisteminde:

```
int i = 1535;
fwrite(&i, sizeof(int), 1, f);
```

Burada dosya yazdırılırsa 2 byte uzunluğunda rastgele karakterler görünür. Çünkü DOS işletim sisteminde *int* türü *2* byte uzunluğundadır. Bizim gördüğümüz ise *1525*'in rastgele olan byte'larıdır. Bilgileri *ASCII* karşılıkları ile dosyaya yazmak için *fprintf* işlevi kullanılabilir.

fread ve *fwrite* işlevleri bellekteki bilgileri aktardığına göre dosyaların da *binary* modda açılmış olması uygun olur.

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

int main()
{
    FILE *f;
    int i;
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int b[10];

    if ((f = fopen("data", w+b")) == NULL) {
        printf("cannot open file...\n");
        exit(EXIT_FAILURE);
    }
    fwrite (a, sizeof(int), 10, f);
    fseek(f, 0, SEEK_SET);
    fread(b, sizeof(int), 10, f);
    for (i = 0; i < 10; ++i)
        printf("%d\n", b[i]);

    fclose(f);

    return 0;
}
```

fread ve *fwrite* işlevlerinin geri dönüş değerleri üçüncü parametresi ile belirtilen okunan ya da yazılan parça sayısıdır. Örneğin

```
n = fread(a, sizeof(int), 10, f);
```

çağrısı ile *f* ile gösterilen dosyadan *sizeof(int) * 10* kadar byte okunarak *RAM*'da *a* adresine yazılmak istenmiştir. Yani dosya konum göstericisinin gösterdiği yerden itibaren bütün sayıları okunabildiyse işlev *10* değerine geri döner. Eğer dosyadaki kalan byte sayısı okunmak istenen sayıdan az ise işlev bütün byte'ları okur ve geri dönüş değeri okunan *byte* sayısı 2. parametresi ile belirtilen değer olur. Örneğin *DOS* işletim sistemi altında çalışıyor olalım. Dosyada *10 byte* bilgi kalmış olsun.

```
n = fread(a, sizeof(int), 10, f);
```

ile işlev *5* değerine geri döner.

Aşağıdaki iki çağrıyı inceleyelim:

```
fread(str, 100, 1, f);
fread(str, 1, 100, f);
```

Her iki işlev çağrısı da *RAM*'deki *str* adresine *FILE* türünden *f* göstericisi ile ilişkilendirilen dosyadan *100* byte okumak amacıyla kullanılabilir. Ancak birinci çağrıda geri dönüş değeri ya *0* ya *1* olabilecekken, ikinci işlev çağrısında geri dönüş değeri *0 100*(dahil) aralığında herhangi bir değer olabilir.

fread ve fwrite İşlevleriyle Blok Blok Kopyalama

Aşağıdaki örnekte bir grup byte *fread* işlevi ile bir dosyadan okunuyor, *fwrite* işlevi ile diğer bir dosyaya yazılıyor:

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

#define   BLOCK_SIZE          1024
#define   MAX_NAME_LEN        80

int main()
{
    FILE *fs, *fd;
    char source_file_name[MAX_NAME_LEN];
    char dest_file_name[MAX_NAME_LEN];
    unsigned int n;
    unsigned char buf[BLOCK_SIZE];

    printf("kaynak dosya ismini giriniz : ");
    gets(source_file_name);
    printf("yeni dosya ismini giriniz : ");
    gets(dest_file_name);
    if ((fs = fopen(source_file_name, "rb")) == NULL) {
        printf("%s dosyası açılamıyor!\n", source_file_name);
        exit(EXIT_FAILURE);
    }
    printf("%s dosyası açıldı!\n", source_file_name);
    if ((fd = fopen(dest_file_name, "wb")) == NULL) {
        printf("%s dosyası yaratılamıyor!\n", source_file_name);
        fclose(fs);
        exit(EXIT_FAILURE);
    }
    printf("%s dosyası yaratılamıyor!\n", dest_file_name);
    while ((n = fread(buf, 1, BLOCK_SIZE, fs))  != 0)
        fwrite(buf, 1, n, fd);
    fclose(fs);
    printf("%s dosyası kapatıldı!\n", source_file_name);
    fclose(fd);
    printf("%s dosyası kapatıldı!\n", dest_file_name);
    printf("kopyalama başarıyla tamamlandı\n");

    return 0;
}
```

Yukarı programı inceleyin. Kopyalama aşağıdaki döngü deyimiyle yapılıyor:

```
while ((n = fread(buf, 1, BLOCK_SIZE, fs))  != 0)
        fwrite(buf, 1, n, fd);
```

fread işlevi ile *fs* ile gösterilen dosyadan, *1 byte'lık* bloklardan *BLOCK_SIZE* kadar okunmaya çalışılıyor. *fread* işlevinin geri dönüş değeri *n* isimli değişkende saklanıyor. *n* değişkenine atanan değer *0* olmadığı sürece döngünün devamı sağlanıyor. Başka bir deyişle *while* döngüsü dosyadan en az *1 byte* okuma yapılabildiği sürece dönüyor. *while* döngüsünün gövdesinde yer alan *fwrite* işlevi çağrısı ile kaynak dosyadan okunan *n* byte hedef dosyaya yazılıyor.

Şimdi de aşağıdaki programı inceleyin. Bu programda komut satırından girilen iki değer arasındaki tüm asal sayılar, ismi komut satırından ismi girilen dosyaya formatsız olarak yazılıyor. Program komut satırından aşağıdaki gibi çalıştırılabilir:

```
<asalyaz> <deger1> <deger2> <dosya ismi>
```

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

```
#include <string.h>

int isprime(int val);

int main(int argc, char *argv[])
{
    int lower_bound, upper_bound;
    int k, temp;
    int prime_counter = 0;
    FILE *f;


    if (argc != 4) {
        printf("<asalyaz> <deger1> <deger2> <dosya ismi>\n");
        exit(EXIT_FAILURE);
    }

    lower_bound = atoi(argv[1]);
    upper_bound = atoi(argv[2]);

    if (lower_bound > upper_bound) {
        temp = lower_bound;
        lower_bound = upper_bound;
        upper_bound = temp;
    }
    f = fopen(argv[3], "wb");
    if (f == NULL) {
        printf("%s dosyasi yaratilamiyor!\n", argv[3]);
        exit(EXIT_FAILURE);
    }
    printf("%s dosyasi yaratildi!\n", argv[3]);

    for (k = lower_bound; k <= upper_bound; ++k)
        if (isprime(k)) {
            fwrite(&k, sizeof(int), 1, f);
            prime_counter++;
        }
    printf("%s dosyasina %d adet asal sayi yazildi!\n", argv[3],
    prime_counter);
    fclose(f);
    printf("%s dosyasi kapatildi!\n", argv[3]);

    return 0;
}
```

Dosya Konum Göstericisi İle İlgili İşlevler

Okuma ya da yazma yapan işlevler, okuma ya da yazma işlemini dosya konum göstericisinin değeri olan konumdan yapar. Bir dosyanın istenilen bir konumundan okuma ya da yazma yapabilmek için önce dosya konum göstericisi konumlandırılmalıdır. Dosya konum göstericisi standart C işlevleri ile konumlandırılabilir:

fseek İşlevi

Bu işlev dosya konum göstericisini istenilen bir *offset* değerine konumlandırmak amacıyla çağrılır. Bu işlevin çağrılmasıyla, açılmış bir dosyanın istenilen bir yerinden okuma yapmak ya da istenilen bir yerine yazmak mümkün hale gelir. İşlevin bildirimi aşağıdaki gibidir:

```
int fseek(FILE *f, long offset, int origin);
```

İşlevin birinci parametresi hangi dosyanın dosya konum göstericisinin konumlandırılacağını belirler. İşlevin ikinci parametresi konumlandırma işleminin yapılacağı *offset* değeridir. İşlevin üçüncü parametresi konumlandırmanın hangi noktadan itibaren yapılacağını belirler. Bu parametreye *stdio.h* içinde bildirilen *SEEK_SET, SEEK_CUR, SEEK_END* simgesel değişmezlerden biri geçilmelidir. Derleyiciler bu simgesel değişmezleri aşağıdaki gibi tanımlar:

```
#define SEEK_SET   0
#define SEEK_CUR   1
#define SEEK_END   2
```

Son parametreye *SEEK_SET* değeri geçilirse, konumlandırma dosya başından itibaren yapılır. Bu durumda ikinci parametre >= *0* olmalıdır. Örneğin:

```
fseek(f, 10L,  SEEK_SET);
```

ile dosya göstericisi *10.* offsete konumlandırılır. Ya da

```
fseek(f, 0L, SEEK_SET);
```

İşlevin üçüncü parametre değişkenine geçilen değer *SEEK_CUR* ise, konumlandırma dosya göstericisinin en son bulunduğu yere göre yapılır. Bu durumda ikinci parametre pozitif ya da negatif değere sahip olabilir. Pozitif bir değer ileri, negatif bir değer geri anlamına gelir. Örneğin dosya göstericisi *10*. *byte*'ı gösteriyor olsun.

```
fseek(f, -1, SEEK_CUR);
```

çağrısı ile dosya göstericisi *9. offset* e konumlandırılır.

İşlevin üçüncü parametre değişkenine geçilen değer *SEEK_END* ise konumlandırma *EOF* durumundan itibaren yani dosya sonundan itibaren yapılır. Bu durumda ikinci parametreye geçilen değer <= *0* olmalıdır. Örneğin dosya göstericisini dosyanın sonuna konumlandırmak için

```
fseek(f, 0, SEEK_END);
```

çağrısını yapmak gerekir.

```
fseek(f, -1, SEEK_END);
```

çağrısı ile dosya konum göstericisi son karakterin bulunduğu yere çekilir. İşlevin geri dönüş değeri işlemin başarısı hakkında bilgi verir. Geri dönüş değeri *0* ise işlem başarılı, *0* dışı bir değer ise işlem başarısızdır. Yalnızca sorunlu durumlarda geri dönüş değerinin sınanması salık verilir.

Yazma ve okuma işlemleri arasında dosya göstericisinin bir konumlandırma işlevi ile konumlandırılması gerekir. Ya da *fflush* işlevine yapılan çağrı ile dosya tampon alanı boşaltılmaldır. Gerekirse boş bir *fseek* çağrısı ile, konumlandırma bulunulan yere yapılabilir:

```
fseek(0, 0L, SEEK_CUR).
```

Örneğin dosyadan bir karakter okunup, bir sonraki karaktere okunmuş karakterin *1* fazlası yazılacak olsun:

```
ch = fgetc(f);
fputc(ch + 1, f);
```

işlemi hatalıdır. Durum çalışma zamanına ilişkin tanımlanmamış bir davranış (*undefined behaviour)* özelliği gösterir. Yani çalışma zamanında herşey olabilir. Yazmadan okumaya, okumadan yazmaya geçişte dosya göstericisi konumlandırılmalıdır. Bu durumun tek istisnası, son yapılan okuma ile dosyanın sonuna gelinmesi durumudur. Bu durumda konumlandırma yapılmadan dosyanın sonuna yazılabileceği güvence altındadır.

rewind İşlevi

Bu işlev ile dosya konum göstericisi dosyanın başına konumlandırılır. İşlevin bildirimi aşağıdaki gibidir:

```
void rewind(FILE *fp);
```

```
rewind(f);
```

çağrısı ile

```
(void) fseek(f, 0L, SEEK_SET);
```

çağrısı aynı anlamdadır.

ftell İşlevi

Bu işlev dosya konum göstericisinin değerini elde etmek için çağrılır. İşlevin bildirimi aşağıdaki gibidir:

```
long ftell(FILE *);
```

İşlevin geri dönüş değeri dosya konum göstericisinin değeridir. Bu değer dosyanın başından itibaren kaçıncı *byte* olduğu bilgisi olarak verilir. Bir hata durumunda işlev *-1L* değerine geri döner.
İkili (*binary)* bir dosyanın uzunluğu *fseek* ve *ftell* işlevlerine yapılan çağrılarla elde edilebilir:

```
fseek(f, 0, SEEK_END);
length = ftell(f);
```

fgetpos ve fsetpos İşlevleri

Bu işlevler, dosya konum göstericisinin değerini elde etmek ya da değerini değiştirmek için, birbirleriyle ilişkili olarak kullanılır. Bu işlevlerle dosya içindeki bir nokta işaretlenerek, daha sonra okuma ya da yazma amacıyla dosya konum göstericisi aynı noktaya konumlandırılabilir. *fgetpos* işlevi ile dosya konum göstericisinin değeri elde edilir, daha sonra bu konuma geri dönebilmek için *fsetpos* işlevi kullanılır.

```
int fgetpos(FILE *, fpos_t *pos);
```

İşlev birinci parametresine geçilen adresle ilişkilendirilen dosyanın dosya konum göstericisinin değerini elde ederek ikinci parametresine geçilen adrese yazar. *fpos_t* türü *stdio.h* başlık dosyası içinde bildirilen standart bir *typedef* türüdür. Bu tür yalnızca *fgetpos* ile *fsetpos* işlevlerinde kullanılır. İşlevin geri dönüş değeri işlemin başarısını gösterir. İşlev, başarı durumunda *0* değerine, başarısızlık durumunda sıfırdan farklı bir değere geri döner.

```
int fsetpos(FILE *, const fpos_t *pos);
```

İşlev, birinci parametresine geçilen adresle ilişkilendirilen dosyanın dosya konum göstericisini ikinci parametresine geçilen adresten okunan değere konumlandırır. İşlevin ikinci parametresine *fgetpos* işlevinden elde edilen bir değeri taşıyan nesnenin adresi geçilmelidir.

Özellikle büyük dosyalar söz konusu olduğunda *fsetpos/fgetpos* işlevleri *fseek* işlevine tercih edilmelidir. Çünkü *fseek* işlevinin ikinci parametresi *long* türden iken, *fgetpos/fsetpos* işlevlerinde kullanılan *fpos_t* türü çok büyük dosyaların konum bilgilerini tutabilecek büyüklükte olan, içsel olarak tanımlanmış bir yapı türü olabilir.

Dosyalarla İlgili İşlem Yapan Diğer Standart İşlevler

remove İşlevi

Bu işlev bir dosyayı siler. İşlevin bildirimi

```
int remove (const char *filename);
```

biçimindedir. İşleve argüman olarak silinecek dosyanın ismi gönderilir. İşlevin geri dönüş değeri, dosyanın başarılı bir şekilde silinebilmesi durumunda *0,* aksi halde yani dosya silinememişse sıfır dışı bir değerdir. Açık olan bir dosyanın silinebilmesi sisteme bağlı olduğundan, yazılan kodun taşınabilirliği açısından, silinecek bir dosya açık ise önce kapatılmalıdır.

rename İşlevi

Bu işlev bir dosyanın ismini değiştirmek için kullanılır. İşlevin bildirimi:

```
int rename (const char *old, const char *new);
```

biçimindedir.

İşleve birinci argüman olarak dosyanın eski ismi ikinci argüman olarak ise dosyanın yeni ismi gönderilmelidir. İşlevin geri dönüş değeri, isim değiştirmen işleminin başarılı olması durumunda *0*, aksi halde yani dosyanın ismi değiştirilemmesi durumunda *0* dışı bir değerdir. Sistemlerin çoğunda açık olan bir dosyanın isminin değiştirilmeye çalışılması durumunda işlev başarısız olur ve *0* dışı bir değere geri döner.

tmpfile İşlevi

İşlev geçici bir dosya açmak amacıyla kullanılır. İşlevin bildirimi :

```
FILE * tmpfile(void);
```

tmpfile işlevi açtığı geçici dosyayı *"wb"* modunda açar. Açılan dosya *fclose* işlevi ile kapatıldığında ya da dosya kapatılmazsa program sona erdiğinde otomatik olarak silinir. İşlevin geri dönüş değeri, açılan geçici dosya ile ilişki kurulmasına yarayacak, *FILE* yapısı türünden bir adrestir. Herhangi bir nedenle dosya geçici dosya açılamıyorsa işlev *NULL* adresine geri döner.
stdio.h başlık dosyası içinde tanımlanan *TMP_MAX* simgesel değişmezi *tmpfile* işleviyle yaratılabilecek maksimum geçici dosya sayısını gösterir. Yazılan bir kaynak kodda, aynı anda açılmış olan geçici dosyaların sayısı *TMP_MAX* değerinden daha büyük olmamalıdır.

tmpnam İşlevi

Geçici olarak kullanılacak bir dosya için bir isim üretilmesi amacıyla kullanılır. İşlevin *stdio.h* başlık dosyası içindeki bildirimi:

```
char *tmpnam(char *s);
```

biçimindedir. İşlev, ürettiği dosya ismini kendisine gönderilen *char* türden adrese yazar. İşlev, aynı zamanda aynı dosya ismini statik ömürlü bir dizinin içine yazarak, bu dizinin başlangıç adresini geri döndürür. Eğer işleve argüman olarak *NULL* adresi gönderilirse, işlev yalnızca statik ömürlü dizinin adresinin döndürür. İşleve *char* türden bir dizinin

adresi gönderildiğinde bu dizinin boyutu ne olmalıdır? Başka bir deyişle *tmpnam* işlevi kaç karakter uzunluğunda bir dosya ismi üretir? İşte bu değer *stdio.h* dosyası içinde tanımlanan *L_tmpnam* simgesel değişmeziyle belirtilir.
tmpnam işlevinin ürettiği dosya isminin çalışılan dizin içinde daha önce kullanılmayan bir dosya ismi olması güvence altına alınmıştır. Yani üretilen dosya ismi bulunulan dizin içinde tektir. Bir programda daha sonra silmek üzere bir dosya açılacağını ve bu dosyaya birtakım bilgilerin yazılmasından sonra dosyanın silineceğini ya da dosyaya başka bir isim verileceğini düşünelim. Bu durumda ilgili dosya yazma modunda açılacağına göre, bu dosyaya var olan bir dosyanın ismi verilmemelidir. Eğer var olan bir dosyanın ismi verilirse, var olan dosya sıfırlanacağı için dosya kaybedilir. Bu riske girmemek için, geçici olarak kullanılacak dosya *tmpfile* işlevi kullanılarak açılmalıdır. Ancak *tmpfile* işlevinin kullanılması durumunda, açılan dosya kalıcı hale getirilemez. Yani herhangi bir nedenden dolayı geçici dosyanın silinmemesi istenirse, dosya kalıcı hale getirilmek istenirse dosya *fopen* işleviyle açılmalıdır. İşte bu durumda geçici dosya başka bir dosyayı riske etmemek için *tmpnam* işlevinin ürettiği isim ile açılmalıdır.
Peki *tmpnam* işleviyle en fazla kaç tane güvenilir dosya ismi üretilebilir? İşte bu değer *stdio.h* içinde tanımlanan *TMP_MAX* simgesel değişmezi ile belirlenmiştir.

Akımlar

C dilinde bazı giriş ve çıkış birimleri (klavye, ekran gibi) doğrudan bir dosya gibi ele alınır. Herhangi bir giriş çıkış birimini akım (*stream)* olarak isimlendirir. Bir akım, bir dosya olabileceği gibi, dosya olarak ele alınan bir giriş çıkış birimi de olabilir. Örneğin küçük programlar genellikle girdilerini genellikle tek bir akımdan alıp (örneğin klavyeden) çıktılarını da tek bir akışa (örneğin ekrana) iletir.

stdout stdin ve stderr Akımları

Bir programa yapılan girdilerin programa doğru akan bir *byte* akımından (*input stream*) geldiği kabul edilir. Yine bir programın çıktısı da programdan dışarıya doğru akan *byte* lar olarak (*output stream*) düşünülür. Dosyalarla ilgili okuma yazma yapan işlevler doğrudan giriş akımından okuma yapıp, çıkış akımına yazabilirler.
Bir C programı çalıştırıldığında 3 akımı gösteren dosyanın otomatik olarak açıldığı kabul edilir. Bu akımlar birer dosya olarak kullanılabilirler ve önceden tanımlanmış *FILE ** türünden değerlerle ilişkilendirilmişlerdir:

stdin : Standart giriş birimini temsil eder. Bu akım normal olarak klavyeye bağlanmıştır.
stdout : Standart çıkış akımını temsil eder. Bu akım normal olarak ekrana bağlanmıştır.
stderr : Standart hata akımını temsil eder. Bu akım da normal olarak ekrana bağlanmıştır.

Daha önce klavyeden girdi alan işlevler (*getchar, gets, scanf*) olarak öğrendiğimiz işlevler aslında *stdin* akımından okuma yapan işlevlerdir.
Daha önce ekrana yazan işlevler (*putchar, puts, printf*) olarak öğrendiğimiz işlevler aslında *stdout* akımına yazan işlevlerdir.

Dosyadan okuma yapan işlevlere *stdin*, yazma yapan işlevlere ise *stdout* ve *stderr FILE ** türünden değerler olarak geçilebilir. Örneğin

```
fprintf(stdout, "Necati Ergin");
```

çağrısı ekrana *Necati Ergin* yazısını yazdırır. Benzer biçimde

```
fputc('A', stdout);
```

Ekrana '*A*' karakteri bastırır.

```
fscanf(stdin, "%d", &val);
```

çağrısı klavyeden alınan değeri *val* değişkenine atar.

Akımların Yönlendirilmesi

İşletim sistemlerinin çoğu giriş ve çıkış akımlarının başka dosyalara yönlendirilmesine izin verir. Örneğin ismi *asalyaz* olan bir programın çalıştırıldığında ekrana *1 - 1000* aralığında asal sayıları yazdırdığını düşünelim. Programın kaynak kodunda yazdırma işlemi için *printf* işlevi çağırılmış olsun. Bu program *DOS / WINDOWS* işletim sistemlerinde komut satırından bu program

```
asalyaz
```

biçiminde çalıştırıldığında

çıktısını ekrana yazar.

Komut satırından *asalyaz* programı,

```
asalyaz > asal.txt
```

biçiminde çalıştırılırsa *axsalyaz* programının *stdout* akımına yani ekrana gönderdiği her şey *asal.txt* dosyasına yazılır. Burada kullanılan '>' simgesine "yönlendirme" *(redirection)* simgesi denir.
Benzer biçimde '<' yönlendirme simgesi de *stdin* akımı için kullanılır. Örneğin, ismi *process* olan bir program

```
process < numbers.txt
```

biçiminde çalıştırılırsa, normal olarak klavyeden alınacak her bilgi *numbers.txt* dosyasından alınır.
Her iki yönlendirme simgesi de bir arada kullanılabilir:

```
process < numbers.txt > prime.txt
```

process işlevi yukarıdaki gibi çalıştırıldığında girdisini *numbers.txt* dosyasından alacak çıktısını ise *prime.txt* dosyasına yazar.

Bunların dışında *UNIX* işletim sisteminde '|' biçiminde başka bir yönlendirme simgesi daha vardır.

```
x | y
```

gibi bir işlemde *x* ve *y* iki program olmak üzere, bu iki program aynı anda çalıştırılır, *x* programının ekrana yazdığı her şey *y* programında klavyeden giriliyormuş gibi işlem görür. Bu yönlendirme işlemine *pipe* adı verilir.

stderr dosyası da normal olarak ekrana yönlendirilir. Yani standart *fprintf* işlevi ile *stderr* dosyasına bir yazma işlemi yapılırsa, yazılanlar yine ekrana çıkar. Ancak *stdout* akımı başka bir dosyaya yönlendirilirse, *stderr* akımı yönlendirme işleminden etkilenmez, halen ekrana bağlı kalır. Yani komut satırından yönlendirme yapılmış olsa bile

```
fprintf(stderr, "hata!\n");
```

gibi bir çağrı sonucunda *"Hata"* yazısı, dosyaya değil ekrana yazılır. Hata iletileri *printf* işlevi ile ekrana yazdırılmak yerine *fprintf* işlevi ile *stderr* akımına yazdırılmalıdır.

fgets İşlevinin stdin Değeri İle Çağrılması:

fgets işlevinin üçüncü parametresin *FILE* * türünden bir değer istediğini biliyorsunuz:

```
fgets(char *str, int n, FILE *f);
```

İşlevin bu parametresine *stdin* değeri geçildiğinde, diziye yerleştirilecek satır klavyeden alınır. Böyle bir çağrı *gets* işlevine yapılan bir çağrıya tercih edilmelidir, çünkü gösterici hatası riski ortadan kaldırılmış olur.

```
char str[20];
gets(s);
```

Yukarıda *gets* işlevine yapılan çağrıda, klavyeden *20* ya da daha fazla karakter girilmesi bir gösterici hatası oluşmasına neden olur. Ancak çağrı

```
fgets(str, 20, stdin);
```

biçiminde yapıldığında diziye en fazla *19* karakter yazılır. Yani gösterici hatası oluşması riski yoktur. Yalnız burada dikkat edilmesi gereken nokta *fgets* işleviyle diziye *'\n'* karakterinin de yazılması olasılığıdır. Aşağıdaki kodu inceleyin:

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

int main()
{
    char str[20];

    printf("bir isim giriniz : ");
    fgets(str, 20, stdin);

    if (!strcmp(str, "NECATI"))
        printf("esit\n");
    else
        printf("esit degil\n");

    return 0;
}
```

Yukarıdaki program çalıştırıldığında klavyeden *NECATI* ismi girildiğinde ekrana *"eşit degil"* yazısı yazdırılır. Zira *fgets* işlevi *NECATI* girişi yapıldıktan sonra girilen *newline* karakterini de diziye yerleştirir. Yani aslında *strcmp* işlevi

```
N E C A T I
N E C A T I \n
```

yazılarını karşılaştırır.

Karşılaştırma işleminden önce aşağıdaki gibi bir *if* deyimi kullanılabilirdi:

```
if ((ptr = strchr(str, '\n')) != NULL)
    *ptr = '\0';
```

Yukarıdaki deyimde *strchr* işleviyle *str* adresindeki yazının içinde *'\n'* karakteri aranıyor. Yazının içinde '\n' karakteri bulunursa, bu karakterin yerine sonlandırıcı karakter yazılıyor. Yani yazının sonunda *'\n'* karakteri varsa yazıdan siliniyor.

freopen İşlevi

Bu işlev daha önce açılmış bir dosyayı, kendi açtığı dosyaya yönlendirir. İşlevin bildirimi:

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```
biçimindedir.
Uygulamalarda daha çok standart dosyaların (*stdin, stdout, stderr)* başka dosyalara yönlendirilmesinde kullanılır. Örneğin bir programın çıktılarının *data.dat* isimli dosyaya yazılması istenirse :

```
if (freopen("data.dat", "w", stdout) == NULL) {
    printf("data.dat dosyası açılamıyor\n");
    exit(EXIT_FAILURE);
}
```

Yukarıdaki işlev çağrısıyla *stdout* dosyasının yönlendirildiği dosya kapatılarak *stdout* dosyasının *data.dat* dosyasına yönlendirilmesi sağlanır. Bu yönlendirme işlemi komut satırından yapılmış olabileceği gibi, *freopen* işlevine daha önce yapılan bir çağrı ile de gerçekleştirilmiş olabilir.
freopen işlevinin geri dönüş değeri işleve gönderilen üçüncü argüman olan *FILE* yapısı türünden göstericidir. *freopen* dosyası yönlendirmenin yapılacağı dosyayı açamazsa *NULL* adresine geri döner. Eğer yönlendirmenin yapıldığı eski dosya kapatılamıyorsa, *freopen* işlevi bu durumda bir işaret vermez.

feof İşlevi

Dosya konum göstericisi dosyanın sonunu gösterdiğinde bir dosyadan okuma yapılırsa okuma işlemi başarılı olmaz. Bu durumda içsel olarak tutulan bir bayrak set edilir. Dosyadan okuma yapan işlevler önce bu bayrağın değerine bakar. Bayrak set edilmişse okuma yapmazlar. *feof* işlevi bu bayrağın değerini alır:

```
int feof(FILE *);
```

Dosya konum göstericisi dosya sonunu gösterirken dosyadan okuma yapılmışsa işlev sıfır dışı bir değere döner. Aksi halde işlev *0* değerine geri döner.

feof işleviyşe ilgili yapılan tipik bir hata, işlevin dosya konum göstericisinin dosyanın sonunu gösterip göstermediğini sınadığını sanmaktır:

```
if (feof(f))
    /*****/
```

deyimi ile böyle bir sınama yapılamaz. Çünkü *feof* işlevi dosya konum göstericisi dosya sonunu göstermesine karşın bu konumdan daha hiç okuma girişiminde bulunulmamışsa *0* değerine geri döner. Bir metin dosyasının içeriğini ekrana yazdırmak isteyen aşağıdaki *C* kodu hatalıdır. Neden hatalı olduğunu bulmaya çalışın:

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

#define FILE_NAME_LEN      256
#define BUFFER_SIZE        20

int main()
{
    char source_file_name[FILE_NAME_LEN];
    char buffer [BUFFER_SIZE];
    FILE *fs;

    printf("yazdirilacak dosya ismi: ");
    gets(source_file_name);

    fs = fopen(source_file_name, "r");
    if (fs == NULL) {
        printf("%s dosyasi acilamiyor!\n", source_file_name);
        exit(EXIT_FAILURE);
    }
    while (!feof(fs)) {
        fgets(buffer, BUFFER_SIZE, fs);
        printf("%s", buffer);
    }

    fclose(fs);

    return 0;
}
```

ferror İşlevi

Okuma ya da yazma yapan işlevler, okuma ya da yazma işleminde bir hata olduğunda içsel olarak çoğunlukla bitsel bir alanda tutulan bir bayrağı birlerler. Okuma ya da yazma yapan işlevler önce bu bayrağın değerine bakar. Bayrak set edilmişse yeni bir okuma/yazma işlemi yapılamaz. Önce bayrağın tekrar sıfırlanması gerekir. *ferror* işlevi *hata* bayrağının birlenip birlenmediğini sınar:

```
int ferror(FILE *);
```

hata bayrağı set edilmişse işlev sıfır dışı bir değere geri döner. *Hata* bayrağı set edilmemişse işlev *0* değerine geri döner.

clearerr İşlevi

```
void clearerr(FILE *stream );
```

Okuma ya da yazma işleminde bir hata olduğunda hata bayrağının set edildiğini, dosya sonundan okuma yapma girişiminde de *EOF* bayrağının set edildiğini söylemiştik. Bu bayraklar set edilmiş durumdayken bir okuma ya da yazma işlemi gerçekleştirilemez. Yeniden bir okuma ya da yazma işleminin yapılabilmesi için önce bayrakların sıfırlanması gerekir. Bu sıfırlama işlemi için *clearerr* işlevi çağrılabilir.
İşlev ilgili dosyaya ilişkin *FILE* * türünden değeri alır ve bu dosyanın *EOF* ve *Error* bayraklarını sıfırlar.

ungetc İşlevi

Bu işlev dosyadan okunan karakteri, dosyanın tampon alanına geri koyar.

```
int ungetc(int c, FILE *f);
```

ungetc işlevi için dosyanın okuma modunda açılmış olması gerekir. İşlevin çağrılmasından sonra yapılan ilk okumada *c* değeri okunur. İşlev başarılı olursa, *c* değerine geri döner. Başarısızlık durumunda dosyanın tampon alanında bir değişiklik olmaz ve *EOF* değeri döndürülür. İşlev, karakteri tampon alanına yerleştirdikten sonra *EOF* bayrağını da sıfırlar. Dosya konum göstericisi yeniden konumlandırılmadan *ungetc* işlevi arka arkaya çağırılmamalıdır.

Dosya Tamponlama İşlevleri

İkincil belleklerle (disket, hard disk vs.) yapılan işlemler bellekte yapılan işlemlere göre çok yavaştır. Bu yüzden bir dosyadan bir karakterin okunması ya da bir dosyaya bir karakterin yazılması durumunda her defasında dosyaya doğrudan ulaşmak verimli bir yöntem değildir.
İşlemin verimi tamponlama (*buffering*) yoluyla artırılır. Bir dosyaya yazılacak veri ilk önce bellekteki bir tampon alanında saklanır. Bu tampon alanı dolduğunda ya da yazma işleminin yapılacağı dosya kapatıldığında tampondaki alanda ne veri varsa dosyaya yazılır. Buna tampon alanının boşaltılması (*flushing*) denir.
Giriş dosyaları da benzer şekilde tamponlanabilir. Giriş aygıtından örneğin klavyeden alınan *byte*'lar önce tampona yazılır.
Dosyaların tamponlanması verimlilikte çok büyük bir artışa neden olur. Çünkü tampondan yani bellekten bir karakter okunması ya da tampona bir karakter yazılması ihmal edilecek kadar küçük bir zaman içinde yapılır. Tampon ile dosya arasındaki transfer, şüphesiz yine zaman alır ama bir defalık blok aktarımı , küçük küçük aktarımların toplamından çok daha kısa zaman alır.

stdio.h başlık dosyası içinde bildirimi yapılan ve dosyalarla ilgili işlem yapan işlevler tamponlamayı otomatik olarak gerçekleştirir. Yani dosyaların tamponlanması için programcının birşey yapmasına gerek kalmadan bu iş geri planda programcıya sezdirilmeden yapılır. Ama bazı durumlarda tamponlama konusunda programcı belirleyici durumda olmak isteyebilir. İşte bu durumlarda programcı dosya tamponlama işlevlerini (*fflush, setbuf, setvbuf)* kullanır:

fflush İşlevi

Dosyalar üzerindeki giriş çıkış işlemleri, çoğunlukla tamponlama yoluyla yapılır. Dosyaya yazma işlemi gerçekleştiren bir işlev çağrıldığında, eğer tamponlama yapılıyorsa, işlev yazma işlemini bellekteki bir tampon alanına yapar. Dosya kapatıldığında ya da tamponlama alanı dolduğunda, tamponlama alanı boşaltılarak dosyaya yazılır.
fflush işlevinin çağrılmasıyla, dosyanın kapatılması ya da tamponlama alanının dolması beklenmeksizin, tamponlama alanı boşatılarak dosyaya yazılır. Bu işlem istenilen sıklıkta yapılabilir. İşlevin bildirimi:

```
int fflush (FILE *stream);
```

biçimindedir.

```
fflush(fp);
```

çağrısı ile *FILE* yapısı türünden *fp* göstericisi ile ilişkilendirilen dosyanın tamponlama alanı (*buffer*) boşaltılır. Eğer *fflush* işlevine *NULL* adresi gönderilirse, açık olan bütün dosyaların tamponlama alanları boşaltılır.
Tamponlama alanının boşaltılması işlemi başarılı olursa *fflush* işlevi *0* değerine geri döner, aksi halde *EOF* değerine geri döner.

setvbuf İşlevi

setvbuf işlevi bir dosyanın tamponlanma biçiminin değiştirilmesi ve tampon alanının yerinin ve boyutunun değiştirilmesi amacıyla kullanılır. İşlevin bildirimi aşağıdaki şekildedir:

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

İşleve gönderilen üçüncü argüman tamponlama şeklini belirler. Üçüncü argümanın değeri *stdio.h* başlık dosyası içinde tanımlanan simgesel değişmezlerle belirlenir.

```
_IOFBF (full buffering - tam tamponlama)
```

Veri dosyaya tamponlama alanı dolduğunda yazılır. Ya da giriş tamponlaması söz konusu ise dosyadan okuma tamponlama alanı boş olduğu zaman yapılır.

```
_IOLBF (line buffering - satır tamponlaması)
```

Tamponlama alanı ile dosya arasındaki okuma ya da yazma işlemi satır satır yapılır.

```
_IONBF (no buffering - tamponlama yok)
```

Dosyadan okuma ya da soyaya yazma tamponlama olmadan doğrudan yapılır.

setvbuf işlevine gönderilen ikinci argüman RAM'de tamponlamanın yapılacağı bloğun başlangıç adresidir. Tamponlamanın yapılacağı alan statik ya da dinamik ömürlü olabileceği gibi, dinamik bellek işlevleriyle de elde edilebilir.
İşleve gönderilen son argüman tamponlama alanında tutulacak *byte*'ların sayısıdır.

setvbuf işlevi dosya açıldıktan sonra, fakat dosya üzerinde herhangi biri işlem yapılmadan önce çağrılmalıdır. İşlevin başarılı olması durumunda işlev *0* değerine geri döner. İşleve gönderilen üçüncü argümanın geçersiz olması durumunda ya da işlevin ilgili tamponlamayı yapamaması durumunda, geri dönüş değeri sıfırdan farklı bir değer olur. İşleve gönderilen tampon alanının geçerliliğinin bitmesinden önce, yani ömrünün tamamlanmasından önce dosya kapatılmamalıdır.

Dosyalarla İlgili İşlem Yapan İşlevlerin Yazımı

Bir dosya ile ilgili işlem yapan bir işlev iki ayrı biçimde tasarlanabilir:

1. İşlev çağrıldığında dosya açıktır. Bu durumda işlevin bir parametresi *FILE* * türünden olur. İşlev çağrılmadan önce, en son okuma işlemi mi yazma işlemi mi yapıldığı bilinemeyeceğinden, böyle bir işlev ilk işlem olarak dosya konum göstericisini konumlandırmalıdır. Aşağıdaki işlevleri inceleyin:

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

#define    SORTED         1
#define    NOT_SORTED     0

void swap(int *p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

void add_ints_to_file(FILE *f)
{
    int k, val, number;

    fseek(f, 0L, SEEK_END);
    number = rand() % 100 + 300;
    for (k = 0; k < number; ++k) {
        val = rand();
        fwrite(&val, sizeof(int), 1, f);
    }
}

void print_file(FILE *f)
{
    int val;
    int counter = 0;

    rewind(f);
    while (fread(&val, sizeof(int), 1, f)) {
        if (counter && counter % 10 == 0)
            printf("\n");
        printf("%5d ", val);
        counter++;
    }
    printf("\n*************************************************\n");
}

void sort_file(FILE *f)
{
    int a[2];
    int k, temp, sort_flag, number_of_ints;

    fseek(f, 0L, SEEK_END);
    number_of_ints = ftell(f) / sizeof(int);

    do {
        sort_flag = SORTED;
        for (k = 0; k < number_of_ints - 1; ++k) {
            fseek(f, sizeof(int) * k, SEEK_SET);
            fread(a, sizeof(int), 2, f);
            if (a[0] > a[1]) {
                swap(a, a + 1);
                fseek(f, sizeof(int) * k, SEEK_SET);
                fwrite(a, sizeof(int), 2, f);
                sort_flag = NOT_SORTED;
            }
        }
```

```
    } while (sort_flag == NOT_SORTED);
}

int main()
{
    FILE *f;

    f = fopen("sayilar.dat", "w+b");
    if (f == NULL) {
        printf("dosya yaratilamiyor!\n");
        exit(EXIT_FAILURE);
    }
    srand((unsigned int)(time(0)));
    add_ints_to_file(f);
    print_file(f);
    sort_file(f);
    print_file(f);
    fclose(f);

    return 0;
}
```

main işlevi içinde çağrılan işlevlerden *add_ints_to_file* işlevi aldığı *binary* dosyanın sonuna rastgele tamsayıları formatsız olarak yazıyor. Daha sonra çağrılan *print_file* işlevi, binary dosyaya yazılan tüm sayıları ekrana yazıyor. *sort_file* işlevi ile dosyadaki tamsayılar küçükten büyüğe doğru sıralandıktan sonra *print_file* işleviyle yeniden yazdırılıyor.

2. İşlev çağrıldığında dosya kapalıdır. İşlev, çağıran kod parçasından dosya ismini ister. Bu durumda işlev işini gerçekleştirmek için önce dosyayı açar. İşini gerçekleştirdikten sonra dosyayı kapatır. Aşağıda bir metin dosyasının içeriğini ekrana yazdıran bir işlev tanımlanıyor:

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

void ftype(const char *file_name)
{
    int ch;

    FILE *f = fopen(file_name, "r");
    if (f == NULL) {
        printf("cannot open %s\n", file_name);
        exit(EXIT_FAILURE);
    }
    while ((ch = fgetc(f)) != EOF)
        putchar(ch);
    fclose(f);
}
```

MAKROLAR

Yapısal programlama alt programlamaya dayanır. Bir problem çözümü daha kolay adımlara ayrılır, bu adımlar için işlevler tanımlanır. Programın kullanacağı veri işlevler tarafından işlenir. Tanımlanan işlevlerden bazılarının kaynak kodu çok kısa olabilir. Aşağıdaki gibi bir işlev tanımlandığını düşünelim:

```
int get_max(int a, int b)
{
    return  a > b ? a : b;
}
```

get_max işlevi kendisine gönderilen iki sayıdan daha büyük olanına geri dönüyor. Şimdi de şu soruyu soralım: Bu kadar kısa kaynak koda sahip bir işlev tanımlanmalı mıdır? İşlev tanımlamak yerine işlevin kodunu, doğrudan işlev çağrısının bulunduğu yere yerleştirmek daha iyi olmaz mı?

Yazılan bir kod parçasında iki sayıdan büyüğünün bulunması gerektiğini düşünelim:

```
x = get_max(y, z);
```

gibi bir deyim yerine

```
x = y > z ? y : z;
```

gibi bir deyim yazılamaz mı?

Bir işlev tanımlayıp tanımlanan işlev mi çağırılmalı yoksa işlevin kodu doğrudan mı yazılmalıdır?
Önce, işlev tanımlamayı destekleyen argümanlar üzerinde duralım:

1. İşlev çağrısı, özellikle işlevin ismi iyi seçilmiş ise, çoğu zaman okunabilirliği daha yüksek bir kod oluşturur. Aşağıdaki örneği inceleyin:

```
a = get_max(ptr->no, per.no);
a = ptr->no > per.no ? ptr->no : per.no;
```

İlk deyimde *get_max* isimli bir işlev çağırılmış ikinci satırda iki ifadeden değeri daha büyük olan, bir işlev çağrılmadan bulunmuştur. Hangi deyimin okunabilirliği daha iyidir?

2. İşlev çağrısı çoğu durumda kaynak kodu küçültür. Özellikle, işleve konu işlem kaynak kod içinde sık yineleniyorsa, bir işlevin tanımlanarak çeşitli yerlerde çağrılması kaynak kodu küçültür.

3. İşlev çağrısı kaynak kodda değişiklik yapmayı daha kolay hale getirir:
Kaynak kod içinde bir çok yerde, iki değerden daha büyüğünün bulunarak kullanıldığını düşünelim. Daha sonra kaynak kodun bu noktalarında değişiklik yapılarak, aynı noktalarda iki değerden daha küçüğünün bulunarak kullanılması gerektiğini düşünelim. Eğer işlev çağrısı yerine kod açık bir şekilde yazılmışsa, değişiklik yapmak çok daha zor olur.

4. İşlev çağrısı, debug etme olanağını artırır. Debugger programlar genel olarak işlev çağrıları için, kodun açık olarak yazılmasına göre daha iyi destek verir.

5. İşlev çağrısı derleyiciye bazı sınamalar yapma olanağını verir. İşleve gönderilen argümanların toplam sayısının işlevin parametre değişkenlerinin toplam sayısına eşit olup

olmadığı derleyici tarafından sınanır. Ayrıca işleve gönderilen argümanların türü ile işlevin ilgili parametre değişkeninin türlerinin uyumlu olup olmadığı da derleyici tarafından sınanır.
Şimdi de işlev çağrısı yapmak yerine işlevin kodunu doğrudan yazmak durumunda elde edilebilecek avantajların ne olabileceğine bakalım:
Bir işleve yapılan çağrının işlemci zamanı ve bellek kullanımı açısından ek bir maliyeti vardır. İşleve giriş ve işlevden çıkış kodlarının yürütülmesi için ek makina komutlarının yürütülmesi gerekir. Varsa işlevin parametre değişkenleri de, işlev çağırıldığında bellekte yer kaplar. Özellikle kaynak kodu çok kısa, kaynak dosya içinde çok çağrılan işlevler için, çoğu durumda bu maliyetin ödenmesi istenmez.
İşlev çağrısı yapmak yerine kod açık bir biçimde yazılırsa, derleyici kodun bulunduğu yere bağlı olarak daha verimli bir eniyileme (*optimizasyon*) gerçekleştirebilir.

Makro Nedir

Makro, işlev tanımlamadan işlev çağrısının getirdiği bazı avantajlardan yararlanılmasını sağlayan bir araçtır. Ortada gerçek bir işlev tanımı söz konusu olmadığından, işlev için ödenen bir ek gider de söz konusu değildir.

Makrolar, önişlemci programın *#define* komutu ile tanımlanır. Bir makro tanımı, önişlemci programa verilen parametrik bir yer değiştirme komutudur.

```
#define MAX(X, Y)       ((x )> (y) ? (x) :(y))
```

MAX, tanımlanan makronun ismidir. Bu ismi "açılan ayraç" karakteri izlemelidir.
Makro ismini izleyen ayraç içinde, virgüllerle ayrılarak bildirilen isimlere "makro parametreleri" denir. Kapanan ayracı izleyen boşluktan sonra gelen atomlar "makro yer değiştirme listesi"dir. Yer değiştirme listesinde, makro parametreleri istenildiği kadar istenildiği sıra ile kullanılabilir.
Önişlemci program kaynak kodun izleyen kesiminde *MAX* ismi ile karşılaştığında, ayraç içindeki ifadeleri makro argümanı olarak kabul eder. Makro tanımında bulunan yer değiştirme listesinde makro parametreleri nasıl kullanılmışsa, makro çağrısının bulunduğu yerlerde benzer bir yer değiştirme işlemi yapılır. Aşağıdaki kodu inceleyin:

```
#include <stdio.h>

#define     ISEVEN(a)       (!((a) & 1))
#define     MAX(a, b)       ((a) > (b) ? (a) : (b))

int main()
{
    int x, y;

    printf("iki sayi giriniz :");
    scanf("%d%d", &x, &y);

    if (ISEVEN(x))
       printf("%d cift sayi!\n", x);
    else
       printf("%d tek sayi!\n", x);

    printf("%d ve %d sayilarindan buyugu = %d\n", x, y, MAX(x, y));

    return 0;
}
```

Yukarıdaki *main* işlevinde, ismi *ISEVEN* ve *MAX* olan iki makro tanımlanıyor. Önişlemci makro tanımının üzerinden geçtikten sonra

```
if (ISEVEN(x))
```

satırı ile karşılaştığında, *ISEVEN* isminin daha önce tanımlanan makroya ait olduğunu ve makro argümanı olarak *x* atomunun kullanıldığını anlar. Önişlemci program ilgili yer değiştirme işlemini yaptığında bu satır aşağıdaki biçime dönüşür:

```
if ((!((x) & 1)))
```

Benzer şekilde

```
printf("%d ve %d sayilarindan buyugu = %d\n", x, y, MAX(x, y));
```

satırı da *MAX* isimli makronun önişlemci program tarafından açılması sonucu

```
printf("%d ve %d sayilarindan buyugu = %d\n", x, y, ((x) > (y) ? (x) :
(y)));
```

haline getirilir.

Güvenli Makro Yazmak

Aşağıdaki makroyu inceleyin:

```
#define Kare(x)    x * x
```

Kaynak kod içinde bu makro aşağıdaki gibi kullanılmış olsun:

```
void func()
{
    int a = 10;
    int b;

    b = Kare(a + 5);
    /***/
}
```

Kare eğer bir işlev olsaydı, işleve gönderilen argüman olan *a + 5* ifadesinin önce değeri hesaplanırdı. Argüman olan ifadenin değeri *15* olduğu için çağrılan işlev *225* değerine geri dönerdi. Ancak önişlemci program aşağıdaki gibi bir yer değiştirme işlemi yapar:

```
b = a + 5 * a + 5;
```

Çarpma işleci toplama işlecine göre daha yüksek öncelikli olduğu için burada *b* değişkenine atanacak değer 65 olur, değil mi? Bu tür öncelik sorunlarını çözmek amacıyla makro parametreleri, açılım listesinde ayraç içine alınmalıdır:

```
#define Kare(x)    (x) * (x)
```

Bu kez aynı ifade önişlemci tarafından aşağıdaki gibi açılır:

```
b = (a + 5) * (a + 5);
```

b değişkenine atanan değer *225* olur. Ancak şimdi de makronun aşağıdaki gibi kullanıldığını düşünün:

```
b = 100 / Kare(a);
```

Kare işlev olsadı *b* değişkenine *1* değeri atanırdı değil mi? Ancak yukarıdaki makro önişlemci tarafından

```
b = 100 / (a) *( a);
```

Burada *b* değişkenine atanan değer *100* olur.

Bu tür öncelik sorunlarını çözmek amacıyla makro açılım listesi dıştan öncelik ayracı içine alınmalıdır.

```
#define Kare(x)     ((x) *(x))
```

Bu kez aynı ifade önişlemci tarafından aşağıdaki gibi açılır:

```
b = 100 / ((a) * (a));
```

Bu kez *b* değişkenine atanan değer *1* olur.

Makro açılım listesini dıştan ayraç içine almak ve açılım listesinde yer alan makro parametrelerinin her birini ayraç içine almak makroları güvenilir kılmaya yetmez. Zaten makrolarla ilgili en büyük sorun güvenilirlik sorunudur. Aşağıdaki kodu inceleyin:

```
#include <stdio.h>

#define Kare(x)     ((x) * (x))

int func(int val);

int main()
{
    int x = 10;
    int y = Kare(x++);
    int z = Kare(func(y));
    /*********/
    return 0;
}
```

```
y = Kare(x++);
```

deyiminde eğer *Kare* bir işlev olsaydı bu işlev *10* değeri ile çağrılmış olurdu. Sonek konumundaki ++ işlecinin yan etkisi nedeniyle, *x* nesnesinin değeri *11* olurdu. Ancak makro önişlemci tarafından açıldığında derleyicinin göreceği kod

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

biçiminde olur ki, bu durum "tanımsız davranıştır" *(undefined behaviour).*

```
z = Kare(func(y));
```

deyimini de önişlemci aşağıdaki gibi açar:

```
z = ((func(y)) * (func(y)));
```

Derlenen kod, *func* işlevinin iki kez çağrılmasına neden olur. Oysa *Kare* bir işlev olsaydı, çağrılan *func* işlevinin üreteceği geri dönüş değeri, bu kez *Kare* işlevine argüman olarak gönderilmiş olurdu.

Örnek Makrolar

Aşağıda bazı faydalı makro tanımlamalarına örnekler verilmiştir. Örnekleri inceleyin:

```
#define GETRANDOM(min, max)    ((rand()%(int)(((max) + 1)-(min)))+ (min))
#define ISLEAP(y)      ((y) % 4 == 0 && (y) % 100 != 0 || (y) % 400 == 0)
#define SWAPINT(x, y)  ((x) ^= (y) ^= (x) ^= (y))
#define XOR(x, y)      (!(x) ^ !(y))
```

Yukarıda tanımlanan *getrandom* isimli makro *min* ve *max* değerleri arasında bir rastgele tamsayının üretilmesine neden olur.
isleap makrosu önişlemci tarafından açıldığında makro argümanı olan ifadenin değerinin artık yıl olması durumunda *1* değeri aksi halde *0* değeri üretilir. Bu durumda makro bir sınama işlevi gibi kullanılır.
swap_int makrosu argümanı olan tamsayı değişkenlerin değerlerini takas eden bir ifadeye açılır.

Bir makronun parametreye sahip olması gerekmez. Aşağıda tanımlanan *randomize* makrosunu inceleyin:

```
#define randomize()    srand((unsigned int)time(NULL))
```

Önişlemci Dizge Yapma İşleci

Bir makro tanımının açılım listesinde yer alan # atomu önişlemci programın bir işlecidir. # işleci önek konumunda tek terimli bir işleçtir. Bu işlemcinin terimi makro parametrelerinden biri olmalıdır. Parametresiz makrolarda bu işleç kullanılamaz. Önişlemci makroyu açtığında makro parametresine karşılık gelen makro argümanını çift tırnak içine alır:
Aşağıdaki kodu inceleyin:

```
#include <stdio.h>

#define    printint(x)    printf(#x " = %d\n", x);

int main()
{
    int a = 10;
    int b = 20;

    printint(a);
    printint(a + b);

    return 0;
}
```

Yukarıdaki örnekte, *printint* makrosu açılım listesinde, makro parametresi olan *x,* dizge yapma işlecinin terimi yapılıyor. Önişlemci program, *main* işlevinin kodunu aşağıdaki biçime dönüştürür.

```
int main()
{
    int a = 10;
    int b = 20;

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

    return 0;
}
```

Boşluk karakterleriyle birbirinden ayrılan dizgelerin, derleyici tarafından otomatik olarak birleştirilerek tek dizge haline getirildiğini anımsayın. Program derlenerek çalıştırıldığında ekran çıktısı aşağıdaki gibi olur:

```
a = 10
a + b = 30
```

Makro argümanı olarak bir dizge verilirse dizgenin başında ve sonunda yer alan çift tırnak (") karakterleri ve dizgenin içinde yer alan ters bölü (\) ve çift tırnak (") karakterinin önüne makro açılımında otomatik olarak ters bölü karakteri yerleştirilir. Aşağıdaki kodu inceleyin:

```
#include <stdio.h>

#define print(x)   printf(#x)

int main()
{
    print(Ekranda bu yazi gorulecek\n);
        print("Ekranda bu yazi cift tirnak icinde gorunecek"\n);
    print("\""\n);

    return 0;
}
```

Program derlenip çalıştırıldığında ekran çıktısı aşağıdaki gibi olur:

```
Ekranda bu yazi gorulecek
"Ekranda bu yazi cift tirnak icinde gorunecek"
"\""
```

Önişlemci Atom Birleştirme İşleci

Önişlemci programın ikinci işleci "##" atom birleştirme işelecidir (*tokenpasting operator*). Atom birleştirme işleci iki terimlidir araek konumundadır. Parametreli ya da parametresiz makrolarda kullanılabilir. Terimlerinin, makro parametrelerinden biri olması zorunlu değildir. Önişlemci, atom birleştirme işlecinin terimlerini birleştirerek, terimlerinden tek bir atom yapar. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

#define    paste(x, y)        x##y

int main()
{
    int paste(a,1);
    a1 = 10;
    printf("a1 = %d\n", a1);

    return 0;
}
```

Yukarıdaki örnekte, önişlemci program tarafından *paste* isimli makronun açılması sonucu *a* ve *1* makro argümanları birleştirilerek *a1* atomu elde edilir.

Bir Makro İle Bir İşlevin Aynı İsmi Paylaşması

Bazı kütüphanelerde belirli işlevler aynı isimle hem makro hem de işlev olarak tanımlanırlar. Bundan amaçlanan, programcının isteğe bağlı olarak makro kullanımını ya da işlev çağrısını seçmesine olanak vermektir. Hem *kare* isimli bir makro tanımlanmış hem de *kare* isimli bir işlev bildirilmiş olsun:

```
int kare(int);

#define    kare(a)    ((a) * (a))
```

Kaynak kodda aşağıdaki gibi bir deyim olsun:

```
x = kare(y);
```

Şüphesiz, böyle bir deyimde makro kullanılmış olur. Çünkü önişlemci, program kodunu derleyici programdan daha önce ele alır. Derleyiciye sıra geldiğinde, önişlemci zaten ilgili makro için yer değiştirme işlemini tamamlamış olur. Ancak makro yerine işlev çağrısını seçmek için iki yöntem kullanılabilir.

1. *#undef* önişlemci komutu kullanılarak makro tanımı ortadan kaldırılabilir:

```
#undef kare
x = kare(y);
```

Önişlemci program *#undef* konutuyla karşılaştığında makro tanımını ortadan kaldırır ve izleyen satırlarda *kare* ismini gördüğünde bir yer değiştirme işlemi yapmaz.

2. *kare* ismi ayraç içine alınarak makro devre dışı bırakılabilir

```
x = (kare)(y);
```

Bu durumda *kare* ismini izleyen ilk atom, açılan ayraç *"("* karakteri olmadığından, önişlemci program yer değiştirme işlemi yapmaz. Ancak işlev çağrısını engelleyen bir durum söz konusu değildir. İşlev isminin bir adres belirttiğini biliyorsunuz. Yukarıdaki ifade ile işlev adresine dönüştürülecek olan *kare* ismi öncelik ayracı içine alınmıştır.

3. Bir işlev, bir işlev göstericisi kullanılarak çağrılabilir:

```
int kare(int);

void func()
{
    int (*fp)(int) = &kare;
    fp();
}
```

Derleyiciler, standart kütüphaneye ilişkin işlevleri aynı zamanda makro olarak da tanımlayabilirler. Örneğin derleyicilerin çoğu *cytpe.h* başlık dosyasında bildirilen karakter test işlevlerini aynı zamanda makro olarak tanımlar.

Makrolar ile İşlevler Arasındaki Farklar

1. Makrolar çoğu zaman aynı işi yapan işleve göre daha hızlı çalışan bir kodun üretilmesine neden olurlar. Zaten makroların kullanıldığı durumlarda hedeflenen de budur.

2. Makrolar, her kullanıldığı yerde önişlemci tarafından açıldığı için, çoğunlukla kaynak kodu büyütür. Kaynak kodun büyüdüğü durumların çoğunda, çalıştırılabilir dosyanın boyutu da büyür. Ancak bir işlev kaç kez çağrılırsa çağrılsın, o işlevin tanımı kaynak kod içinde bir kez yer alır.

3. Makro paremetreleri makro açılımı içinde birden fazla kullanılıyor ise makro açılımı sonucu istenmeyen durumlar oluşabilir. Makro argümanı olarak kullanılan ifadenin bir yan etkiye sahip olması durmunda bu yan etki birden fazla kez oluşur. Ancak işlevler söz konusu olduğunda böyle bir risk söz konusu değildir.

4. Bir işlevin adresi işlev göstericilerinde tutularak bazı işlemler gerçekleştirilebilir. Ancak bir makronun adresi olmaz.

5. Bir makro söz konusu olduğunda önişlemci program, makroya verilen argümanların sayısı ile makro parametrelerinin sayısının uyumunu kontrol edebilir. Ancak önişlemci

program makro argümanlarıyla makro parametrelerinin tür açısından uyumunu kontrol edemez, makrolar türden bağımsızdır. Ancak işlevler türlere ilişkin yazıldığından derleyicinin yapacağı tür kontrollerine tabidir. Derleyici işleve gönderilen argümanlarla işlev parametrelerinin türlerinin uyumsuz olması durumunda hata ya da uyarı iletisi üretebilir.

Makro Açılımlarının Alt Satırdan Devam Etmesi

Bir makro açılım listesinin yazımı makronun tanımlandığı satırda sonlanmak zorunda değildir. Makro açılım listesinin yazımı alt satırdan devam edebilir. Bunun için alt satıra geçmeden önce ters bölü karakteri '\' kullanılır.

Makrolar ile İşlev Kalıbı Oluşturulması

Makrolar kullanılarak işlev kalıbı hazırlanabilir. Aşağıdaki örnekte bir dizinin en büyük elemanını bulan bir işlev kalıbı hazırlanıyor:

```
#include <stdio.h>

#define generic_max(T)      T getmax_##T(const T *ptr, size_t size) {\
                           int k; T max = *ptr; for (k = 1; k < size; ++k)\
                            if (ptr[k] > max) max = ptr[k]; return max;}

generic_max(int)
generic_max(double)

int main()
{
    int a[10] = {1, 4, 5, 7, 8, 9, 10, 2, 3, 6};
    double b[10] = {1.2, 3.5, 7.8, 2.4, 4.4, .7, 3.2, 4.8, 2.9, 1.};

    printf("a dizisinin en buyuk elemani = %d\n", getmax_int(a, 10));
    printf("b dizisinin en buyuk elemani = %lf\n", getmax_double(b, 10));

    return 0;
}
```

generic_max isimli makronun tanımında makro parametresi olarak *T* isminin kullanıldığını görüyorsunuz. Bu isim bir tür bilgisi olarak kullanılıyor. Önişlemci atom birleştirme işleciyle, makro açılımı sonucunda farklı işlev isimleri elde edilir.

Makro Açılımında Makro İsminin Kullanılması

Bir makro ismi ya da bir simgesel değişmez, değiştirme listesinde yer alabilir. Bu duruma İngilizcede *"self referential macro"* denmektedir. Eğer açılım listesinde makronun kendi ismi yer alırsa önişlemci özyinelemeli bir yer değiştirme işlemi yapmaz.

Aşağıdaki örneği inceleyin:

```
#define max     (4 + max)
```

max isminin programın içinde kullanılan bir başka değişkenin ismi olduğunu düşünelim.

```
x = max;
```

Önişlemci program yukarıdaki deyimin üzerinden geçtiğinde bu deyimi

```
x = 4 + max;
```

biçimine dönüştürür. Ve daha sonra artık *max* ismini yine yer değiştirme işlemine sokmaz.

Standart offsetof Makrosu

Bu makro *stddef.h* başlık dosyası içinde tanımlanmıştır. Makronun iki parametresi vardır. Birinci parametreye bir yapı tür ismi, ikinci parametreye yapının bir elemanı verilir. Makronun açılması sonucu oluşan değişmez ifadesinin değeri, bir yapı elemanının yapı nesnesi içindeki konumunu gösteren *offset* değeridir. Yapının ilk elemanı için bu değer *0*'dır.
Aşağıdaki örneği inceleyin:

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

struct Date {
    int day, mon, year;
};

int main()
{
    printf("%d\n", offsetof(struct Date, day));
    printf("%d\n", offsetof(struct Date, mon));
    printf("%d\n", offsetof(struct Date, year));

    return 0;
}
```

offsetof makrosu derleyicilerin çoğunda aşağıdaki gibi tanımlanır:

```
#define offsetof(type, member)     ((size_t)&((type *)0)member)
```

ÖNİŞLEMCİ KOMUTLARI (2)

Koşullu Derleme Nedir?

Koşullu derleme, derleme işleminin bir koşula ya da bazı koşullara göre yapılmasıdır. Yazılan kaynak kodun belirli kısımları önişlemci program tarafından derleyiciye verilirken bazı kısımları derleyiciye verilmez. Derleyiciye verilecek kaynak kod önişlemci program tarafından seçilir.

Koşulu Derleme Önişlemci Komutları

Koşullu derlemede amacıyla kullanılan önişlemci komutları

```
#if, #else, #elif, #ifdef, #ifndef, #endif
```

komutlarıdır.

#if ve #endif Önişlemci Komutları

#if önişlemci komutunun argümanı tamsayı türünden bir değişmez ifadesi olmalıdır. *if* sözcüğünü izleyen ifadenin değeri sıfır dışı bir değer ise *#if* komutu ile *#elif* ya da *#else* yada *#endif* komutları arasında kalan kısım derleyiciye verilir. Eğer *#if* ifadesi yanlış ise yani ifadenin değeri sıfır ise *#if* komutu ile #else arasında kalan kısım derleyiciye verilmez. Aşağıdaki programı derleyerek çalıştırın:

```
int main()
{
#if 1
    printf("bu yazi ekranda gorunecek\n");
#endif

#if 0
    printf("bu yazi ekranda gorunmeyecek\n");
#endif
    return 0;
}
```

#if komutunda kullanılacak ifade tamsayı türünden olmalıdır. Bu ifade içinde

Tamsayı değişmezleri kullanılabilir. Önişlemci program kullanılan tüm sayı değişmezlerini işaretli *long* ya da işaretsiz *long* türünden varsayar.
Karakter değişmezleri kullanılabilir.
Aritmetik işleçler (+, -, *, /, %), karşılaştırma işleçleri (<, <=, >, >=, ==, !=) , bitsel işleçler(*~, &, ^,* |) ve mantıksal işleçler (*!,* && ve ||)kullanılabilir. && ve || işleçlerinin kullanımında kısa devre davranışı geçerlidir.
#if ifadesinin değerinin hesap edilmesinden önce simgesel değişmezlerle ve makro tanımlarına ilişkin bütün yer değiştirme işlemleri yapılır.

Simgesel değişmez ve makro ismi olmayan tüm isimler *0* değerine sahip kabul edilirler. *#if* ifadesinde *sizeof* işleci yer alamaz. Numaralandırma türlerinden değişmezler temsil ettikleri değerler olarak ele alınmaz. *#if* ifadesinde bir numaralandırma değişmezi kullanılırsa, bu değişmez makro ismi olmayan tüm isimler gibi *0* değeri olarak ele alınır.

#if komutu ile *#endif* komutu aralığındaki alanda başka önişlemci komutları yer alabilir. Bu alanda yer alan önişlemci komutları ancak *#if* ifadesi doğru ise yerine getirilir.
#if komutunu izleyen ifade bu *#if* komutuyla ilişkilendirilen *#endif* komutunu izleyen bir yorum satırı içinde yinelenirse, kodun okunabilirliği artar. Bu durum özellikle *#if* komutlarının iç içe bulunması durumunda fayda sağlar. Aşağıdaki örneği inceleyin:

```
#if MAX > 100
#if MIN < 20
#define LIMIT 100
#endif /* MIN < 20 */
#define UPPER_BOUND 2000
#endif /* MAX > 100  */
```

#else Önişlemci Komutu

 # if komutunu bir *#else* komutu izleyebilir. Bu durumda *#if* ifadesinin değeri *0* dışı bir değer ise yani ifade doğru ise *#if* ve *#else* komutları arasında kalan kaynak kod parçası derleyiciye verilir. Eğer bu ifade yanlış ise *#else* ve *#endif* komutları arasındaki kısım derleyici programa verilir. Böylece bir tamsayı değişmez ifadesinin doğru veya yanlış olmasına göre derleyiciye farklı kod parçaları verilebilir. Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

#define    MAX        100

int main()
{
#if    MAX > 10
    printf("%d\n", MAX);
#else
    printf("**********\n");
#endif /*MAX > 100 */
    return 0;
}
```

#elif Önişlemci Komutu

elif sözcüğü *else if* sözcüklerinin birleştirilerek kısaltılmasından elde edilmiştir.
Tıpkı *C* dilinin if deyimi gibi bir *#if* komutunun doğru ya da yanlış kısmında başka bir *#if* komutu yer alabilir. *#if* ve *#else* komutlarıyla bir merdiven oluşturulabilir.
Bu amaçla kullanılabilecek bir komut *#elif* komutudur.
#elif komutunu yine bir tamsayı değişmez ifadesi izler. Eğer bu tamsayı değişmez ifadesi *0* dışı bir değere sahipse doğru olarak yorumlanır ve *#elif* komutu ile bunu izleyen bir başka *#elif* ya da *#endif* komutu arasındaki kısım derleyiciye verilir. Aşağıdaki örneği derleyerek çalıştırın:

```
#include <stdio.h>

#define    MAX        60

int main()
{
#if MAX > 10 && MAX <= 20
    printf("(if)%d\n", MAX);
#elif MAX > 20 && MAX <= 50
    printf("(elif1)%d\n", MAX);
#elif MAX > 50 && MAX < 100
    printf("(elif2)%d\n", MAX);
#else
    printf("(else)%d\n", MAX);
#endif
    return 0;
}
```

defined Önişlemci İşleci

Makrolar konusunda önce önişlemci programın dizge yapma (#) ve atom birleştirme (##) işleçlerini görmüştük. Önişlemci programın 3. ve son işlei *defined* işlecidir. *defined* önişlemci işlecini bir isim izler. Bu isim ayraç içine de alınabilir:

```
defined MAX
defined (MAX)
```

Eğer *defined* işlecinin terimi olan isimli daha önce tanımlanmış bir simgesel var ise, *defined* işleci *1* değeri üretir. Bu isimli bir simgesel değişmez yoksa *defined* işleci *0* değeri üretir.
Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

#define    MAX    100

int main()
{
#if defined MAX && !defined   MIN
    printf("max = %d\n", MAX);
#endif
    return 0;
}
```

#ifdef ve ifndef Önişlemci Komutları

#if önişlemci komutunun kullanıldığı her yerde *#ifdef* ve *#ifndef* önişlemci komutları kullanılabilir:

#ifdef önişlemci komutunu bir isim(*identifier*) izler. Eğer bu isim daha önce tanımlanmış bir simgesel değişmeze ilişkinse ilk *#else*, *#elif* ya da *#endif* öneişlemci komutlarına kadar olan kısım derleyiciye verilir. Eğer *#ifdef* önişlemci komutunu izleyen isimde bir simgesel değişmez tanımlı değil ise ilk *#else #elif #endif* komutuna kadar olan kısım derleyiciye verilmez. Bu komut ile yapılan iş #if önişlemci komutu ile defined işlecinin birlikte kullanılmasıyla da yapılabilir:

```
#ifdef  ISIM
```

gibi bir önişlemci komutu kullanımı ile

```
#if defined (ISIM)
```

gibi bir önişlemci komutu aynı anlamdadır.

#ifndef önişlemci komutunu da bir isim(*identifier*) izler. Eğer bu isimde bir isim daha önce #define önişlemci komutuyla tanımlanmamış ise, ilk *#else*, *#elif* ya da *#endif* öneişlemci komutuna kadar olan kısım derleyiciye verilir.
#ifndef önişlemci komutunu izleyen isimde bir simgesel değişmez tanımlanmış ise *#else #elif #endif* komutlarına kadar olan kısım derleyiciye verilmez.

```
#ifndef    SYSTEM
```

gibi bir önişlemci komutu ile

```
#if !defined (SYSTEM)
```

gibi bir önişlemci komutu aynı anlamdadır.
Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

#define ISIM1

int main()
{
#ifdef ISIM1
    printf("bu yazi ekranda cikacak!\n");
#endif

#ifndef ISIM1
    printf("bu yazi ekranda cikmayacak!\n");
#endif
    return 0;
}
```

#error Önişlemci Komutu

#error önişlemci komutu koşullu derlemede kullanılır.
Önişlemci program bu komut ile karşılaştığında derleme işlemini daha önişleme aşamasında sonlandırır. Kullanımı aşağıdaki gibidir:

```
#error yazı
```

error önişlemci komutunun yanında -boşluk karakteri ile ayrılmış- bir hata iletisi yer alır. Önişlemci #error komutunu görünce bu hata iletisini ekrana yazarak derleme işlemine son verir. Hata yazısının ekranda nasıl gösterileceği derleyiciye göre değişebilir.

```
#ifndef __STDC__
#error Bu program yalnızca C derleyicisinde derlenir.
#endif
```

Yukarıdaki örnekte *#ifndef* komutu ile, derleme işlemini yapacak derleyicinin standart C derleyicisi olup olmadığı sınanıyor, derleyici eğer standart C derleyicisi değil ise derleme işlemi önişlemci aşamasında sonlandırılıyor. Derleme işlemi sonlandırıldığında ekrana:

```
Bu program yalnızca C derleyicisinde derlenir.
```

yazısı yazdırılır.

```
#if UINT_MAX  < 65535
#error unsigned int turu yeterli buyuklukte degil
#endif
```

Yazılan programın *int* türünün *2 byte* dan daha küçük olan bir sistem için derlenmeye çalışılması durumunda ekrana

```
Error directive: unsigned int turu yeterli buyuklukte degil
```

Biçiminde bir yazı yazdırılarak derleme işlemine son verilir.

#undef Önişlemci Komutu

Bir simgesel değişmezin ilki ile özdeş olmayan bir biçimde ikinci kez tanımlanması tanımsız davranıştır (*undefined behaviour).*
Örneğin aşağıdaki gibi bir tanımlama işlemi yanlıştır:

```
#define    MAX 100
#define    MAX 200
```

Bir simgesel değişmezin ilki ile özdeş olarak tanımlanmasında herhangi bir sorun çıkmaz. Bir simgesel değişmez ikinci kez tanımlanmak istenirse önce eski tanımlamayı ortadan kaldırmak gerekir. Bu işlem *#undef* önişlemci komutuyla yapılır.
#undef önişlemci komutunun yanına geçerliliği ortadan kaldırılacak simgesel değişmezin ya da makronun ismi yazılmalıdır:

```
#undef      MAX
#define     MAX     20
```

Yukarıdaki önişlemci komutlatıyla önce *MAX* simgesel değişmezinin tanımı ortadan kaldırılıyor, sonra *MAX* simgesel değişmezi *200* olarak tanımlanıyor. *#undef* ile tanımlanması kaldırılmak istenen simgesel değişmez, daha önce tanımlanmış olmasa bile bu durum bir soruna yol açmaz. Yukarıdaki örnekte, *MAX* simgesel değişmezi daha önce tanımlanmamış olsaydı, bu durum bir hataya yol açmazdı.

Önceden Tanımlanmış Simgesel Değişmezler

Standart C dilinde 5 tane simgesel değişmez önceden tanımlanmış kabul edilir. Herhangi bir başlık dosyası içinde bu simgesel değişmezler *#define* önişlemci komutuyla tanımlanmış olmamasına karşın kaynak kodun derleyici tarafından ele alınmasından önce bir yer değiştirme işlemine sokulurlar. Bu simgesel değişmezler çoğunlukla hata arama amacıyla yazılan kodlarda kullanılırlar:

__LINE__ öntanımlı simgesel değişmezi
Bu simgesel değişmez kaynak kodun kaçıncı satırında kullanılmış ise, o satırın numarasını gösteren bir tamsayı değişmezi ile yer değiştirilir.

__FILE__ öntanımlı simgesel değişmezi
Bu simgesel değişmez hangi kaynak dosya içinde kullanılmış ise, o kaynak dosyanın ismini gösteren bir dizge ifadesiyle yer değiştirilir.

__DATE__ öntanımlı simgesel değişmezi
Bu simgesel derleme tarihini gösteren bir dizge ifadesiyle yer değiştirilir. Tarih bilgisini içeren yazının formatı aşağıdaki gibidir:

```
Aaa gg yyyy (ay, gün, yıl)
```

__TIME__ öntanımlı simgesel değişmezi
Bu simgesel değişmez derleme zamanını gösteren bir dizge ifadesiyle yer değiştirilir. Zaman bilgisini içeren yazının formatı aşağıdaki gibidir:

```
sa:dd:ss (saat, dakika, saniye)
```

__STDC__ öntanımlı simgesel değişmezi
Eğer derleyici standart C derleyicisi ise bu simgesel değişmez tanımlı kabul edilir. Derleyici standart C derleyicisi değil ise bu simgesel değişmez tanımlanmamış kabul edilir.
Aşağıdaki programı derleyerek çalıştırın:

```
# include <stdio.h>

int main()
{
    printf("kaynak dosya ismi : %s\n", __FILE__);
    printf("derleme tarihi = %s\n", __DATE__);
    printf("derleme zamani = %s\n", __TIME__);
    printf("bu satirin numarasi = %d\n", __LINE__);
#ifdef __STDC__
    printf("standart C derleyicisi\n");
```

```
#else
    printf("standart C derleyicisi degil\n");
#endif
    return 0;
}
```

[C++ derleyiclerinde de __cplusplus öntanımlı simgesel değişmezi, tanımlanmış kabul edilir.]

#line Önişlemci Komutu

Bu önişlemci komutuyla derleyicinin kaynak koda ilişkin tuttuğu satır numarası ve dosya ismi değiştirilebilir. Bu komut iki ayrı argüman alabilir.
Komutun alabileceği birinci argüman bir tamsayı olarak satır numarasıdır. Komutun seçimlik olarak alabileceği ikinci argüman dosya ismini gösteren dizgedir.
Bu önişlemci komutu kaynak kod üreten programlar tarafından kullanılabilir.

#line önişlemci komutu size anlamsız gelebilir. Neden derleyicinin vereceği hata iletisi örneğin *20*. satırı değil de *25*. satır göstersin? Neden derleyici hata iletisinde derlediği kaynak dosyanın ismini değil de bir başka dosyanın ismini yazdırsın?
#line komutu programcılardan çok, çıktı olarak C kaynak kodu üreten programalar tarafından kullanılır. Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

int main()
{
    printf("%s dosyasının %d. satırı\n", __FILE__, __LINE__);
#line   100     "aaaaa.c"
    printf("%s dosyasının %d. satırı\n", __FILE__, __LINE__);

    return 0;
}
```

Bir Başlık Dosyasının Birden Fazla Kez Kaynak Dosyaya Eklenmesi

Projeler çoğunlukla farklı kaynak dosyalardan ve başlık dosyalarından oluşur. Hizmet alan kodlar *(client codes*) hizmet veren kodların (*server codes*) başlık dosyalarını eklerler. Başlık dosyasını eklemenin *#include* önişlemci komutuyla yapıldığını biliyorsunuz. Bir kodlama dosyasında iki ayrı başlık dosyasının eklendiğini düşünelim:

```
/*** file1.c  *****/
#include "header1.h"
#include "header2.h"
```

Başlık dosyaları içinde başka başlık dosyalarının eklenmesine sık rastlanır. Örneğin *header1* ve *header2* isimli başlık dosyalarının her ikisinde de *header3.h* isimli bir başlık dosyasının eklendiğini düşünelim:

```
/*** header1.h  *****/
#include "header3.h"
/*** header2.h  *****/
#include "header3.h"
```

Bu durumda *file1.c* isimli kaynak dosya içine *header3.h* isimli başlık dosyası iki kez eklenmiş olur. Bu duruma İngilizcede *"multiple inclusion*" denir. Peki bu durumun bir sakıncası var mıdır?
Başlık dosyaları içinde bildirimler bulunur. Bir başlık dosyası iki kez eklenirse bu başlık dosyası içindeki bildirimler iki kez yapılmış olur. Bazı bildirimlerin özdeş olarak yinelenmesinin bir sakıncası yoktur. Örneğin bir işlev bildirimi daha önceki bir bildirimle çelişmemek kaydıyla yinelenebilir. Benzer durum *extern* bildirimleri ve *typedef* bildirimleri

için de geçerlidir. Ancak programcı tarafından yapılan bir tür bildiriminin yinelenmesi, bildirimler özdeş olsa bile geçersizdir:

```
struct A {
    /****/
};
struct A {
    /****/
};
```

Yukarıdaki örnekte *struct A* isimli yapının bildirimi ikinci kez yapıldığında derleyici program bir hata iletisi verir. Bir örnek de standart kütüphaneden verelim:
time.h isimli standart başlık dosyasında *struct tm* isimli bir yapının bildirildiğini biliyorsunuz. Eğer bir kodlama dosyası içine *time.h* başlık dosyasının içeriği iki kez boşaltılırsa, kodlama dosyaısnın derlenmesi aşamasında hata oluşur. Çünkü *struct tm* yapısı iki kez bildirilmiş olur.

Proje geliştirilmesi süresinde böyle hataların giderilmesi fazladan zaman kaybına yol açar. Önişlemci koşullu derleme komutlarıyle bu konuda bir önlem alınabilir. C ve C++ dillerinde başlık dosyaları çoğunlukla aşağıdaki biçimde hazırlanır:

```
//header1.h

#ifndef _HEADER1_H_
#define _HEADER1_H_

#endif
```

Bu başlık dosyası bir kaynak dosya tarafından ilk kez eklendiğinde *_HEADER1_H_* simgesel değişmezi henüz tanımlanmış olmadığından *#endif* önişlemci komutuna kadar olan kısım derleyiciye verilir. Ancak daha sonra bu başlık dosyası bir kez daha kaynak koda eklenmek istenirse artık *_HEADER1_H* simgesel değişmezi tanımlanmış olduğundan *#endif* önişlemci komutuna kadar olan kısım artık kaynak koda verilmez.

Koşullu Derlemeye İlişkin Önişlemci Komutları Nerelerde Kullanılır?

1. Koşullu derleme komutları sıklıkla debug amacıyla kullanılır.
Program yazılırken debug amacıyla programa bazı kodlar eklenir. Ancak programın son sürümünde hata aramaya yönelik kodların bulunması istenmez. Çünkü bu kodların programın çalışma zamanında getireceği ek maliyet istenmez.
Program içindeki belirli kod parçalArı yalnızca debug sürümüde derlenir:

```
#if DEBUG
    /***/
#endif
```

2. Bazı programların birden fazla işletim sisteminde çalışması istenebilir. Kaynak kodun belirli kısımları önişlemci program tarafından işletim sistemine göre seçilerek derleyiciye verilir:

```
#if defined(WINDOWS)
/***/
#elif  defined (DOS)
/***/
#elif  defined(OS2)
/***/
#endif
```

3. Bazen de aynı program farklı derleyiciler ile derlenir.

```
#ifdef __STDC__
    /***/
#else
    /***/
#endif
```

Koşullu derlemeye ilişkin önişlemci komutlarının kullanılmasına ilişkin bir program parçası örneği aşağıda veriliyor:

```
#if DLEVEL > 5
    #define SIGNAL 1
    #if STACKUSE == 1
        #define STACK  200
    #else
        #define STACK  50
    #endif
#else
    #define SIGNAL 0
    #if STACKUSE == 0
        #define STACK  200
    #else
        #define STACK  50
    #endif
#endif
#if DLEVEL == 0
    #define STACK      0
#elif DLEVEL == 1
    #define STACK      100
#elif DLEVEL > 5
    display(debugptr);
#else
    #define STACK 200
#endif
```

Yukarıdaki örnekte birinci *#if* bloğunun altında iki ayrı *#if #else* yapısı bulunur. *DLEVEL* simgesel değişmezinin *5*'den büyük olup olmamasına göre değerine göre önişlemci tarafından doğrı kısım ya da yanlış kısım ele alınır.
#elif önişlemci komutunun da yer aldığı ikinci *#if* bloğunda ise *DLEVEL* simgesel değişmezinin değerine göre 4 seçenekten biri ele alınır. *DLEVEL* simgesel değişmezinin değerine bağlı olarak *STACK* simgesel değişmezi 0, 100 ya da 200 olarak tanımlanır. *DLEVEL* simgesel değişmezinin değerinin 5'den büyük olması durumunda ise

```
display(debugptr);
```

deyimi derleyiciye verilir. Bu durumda *STACK* simgesel değişmezi tanımlanmaz.

assert Makrosu

assert, program geliştirirken programcının geliştirme süreci içinde bir takım olası böcekleri farketmesi için kullanılan bir makrodur. C'nin standart başlık dosyalarından *assert.h* içinde tanımlanmıştır. *assert* makrosu, makroya argüman olarak bir ifadenin verilmesiyle kullanılır. Örneğin:

```
assert(p != NULL);
```

Ayraç içindeki ifade, doğru olduğu ya da doğru olması gerektiği düşünülen bir durum belirtir. *assert* makrosuna verilen ifadenin sayısal değeri sıfır dışı bir değerse, yani sınanan önerme doğruysa, *assert* makrosu hiçbir şey yapmaz. Ancak makro ifadesinin değeri sıfırsa, yani ifadeye konu önerme mantıksal olarak yanlış ise *assert* makrosu standart *abort()* işlevini çağırarak programı sonlandırır.
abort işlevi tıpkı standart *exit()* işlevi gibi çağırıldığı zaman programı sonlandırır. Ancak *exit* işlevinin yaptığı bir takım geri alma işlemlerini *abort* işlevi yapmaz. *abort* işlevi programı sonlandırdığında *stdout* akımına "*Abnormal program termination*" yazısını basar. *assert* makrosu başarısız olduğunda *abort* işlevini çağırmadan önce doğruluğu sınanan ifadeyi ve *assert* işleminin hangi dosyada ve satırda olduğu bilgisini ekrana yazar:

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

int main()
{
    int x, y;

    printf("iki sayi giriniz : ");
    scanf("%d%d", &x, &y);
    assert(y != 0);
    printf("%d / %d = %d\n", x, y, x / y);

    return 0;
}
```

Yukarıdaki programı inceleyin. Klavyeden y değişkenine girilen değer *0* ise, *assert* makrosuna verilen ifade yanlış olduğundan, program *abort* işlevinin çağrılmasıyla sonlandırılır. Programı çalıştırarak, program sonlandırıldığında ekrana basılan yazıyı inceleyin.

assert, *assert.h* dosyası içinde aşağıdakine benzer biçimde tanımlanmış bir makrodur.

```
#ifndef NDEBUG
#define    assert(exp) if(!(exp)){fprintf(stderr,"Assertion failed: %s,file
%s, line %d\n",\
       #exp, __FILE__, __LINE__);abort();}
#else
#define assert(exp)
#endif
```

Bu makro şunları yapar:

Eğer programcı *assert.h* dosyasının içeriğinin boşaltıldığı yerden daha yukarıda olan kaynak kod alanı içinde *NDEBUG* simgesel değişmezini tanımlamamışsa bütün *assert* makroları yerine kontrol kodları yerleştirilir. Kontrol kodu görüldüğü gibi ifadenin sayısal değeri sıfır ise *abort* işlevi ile program sonlandırılır. Eğer programcı *assert.h* dosyasının eklenmesinden önce *NDEBUG* simgesel değişmezini tanımlamış ise görüldüğü gibi bütün

assert makroları kaynak koddan silinir. Bütün *assert* makrolarını kaynak koddan silmek için programın kaynak kodun tepesine

```
#define NDEBUG
```

komutu yerleştirilmelidir.

assert Makrosu Neden Kullanılır?

assert makrosu programın geliştirilme aşamasında olası hataları yakalamak için programcı tarafından kaynak koda eklenen bir makrodur. Programcı kodunu yazarken bir böcek karşısında oluşabilecek durumlar için *assert* mokrolarını yerleştirir. Programın geliştirilme sürecinde denemeler sırasında *assert* mokrosu tarafından program sonlandırılırsa programcı olmaması gereken bir durumun olduğunu anlar ve bu durumun bir böcek nedeni ile oluştuğunu düşünür. Şüphesiz böcek tam olarak *assert* mokrosunun bulunduğu yerde olmayabilir. Başka yerde yapılan hatalar dolayısı ile böcek oluşmuş olabilir. Bu durum *assert* makrosu tarafından bir ipucu olarak ele geçirilir. *assert* makrosuna yakalanıldığında programcı kodunu akışa göre incelemeli ve böceğin nereden kaynaklandığını saptamaya çalışır. *assert* makrosu ile yapılan kontroller aslında düzgün çalışan bir program için gereksiz olan kontrollerdir. İşte programın *assert* makrolarının kontrol kodları yerleştirdiği (yani *NDEBUG* simgesel değişmezinin tamımlanmadığı) versiyonuna "hata arama sürümü" (*debug versiyonu*) denir. Hata arama sürümü *assert* makroları ile kaynak kodun şişmiş olduğu sürümdür. Programcı kodunda hiçbir böcek olmadığından emin ise programı teslim etmeden önce *NDEBUG* simgesel değişmezini tanımlayarak programı son kez derler. Buna programın "sürüm biçimi" (*release version*) denir. Büyük programlar debug versiyonunda oluşturulmalı, son aşamada *release* versiyonu elde edilmelidir.

assert Makrosu Nerelerde Kullanılır?

assert makroları programın ticari sürümlerinde ya da kodda kalması gereken durumlar için kullanılmamalıdır. Örneğin *malloc* ve *fopen* işlevlerinin geri dönüş değerleri programın hem debug sürümünde hem de release sürümünde kontrol edilmelidir. *assert* makroları şu durumlarda kullanılabilir:
İşlev parametrelerinin olmaması gereken anormal değerleri alması durumu *assert* makrosu ile kontrol edilebilir. Özellikle işlev parametresi olan göstericilerin hemen ana bloğun girişinde *NULL* adresi olma durumu kontrol edilebilir, çünkü böceklerden dolayı göstericilerin *NULL* değerine düşmesi sık rastlanılan bir durumdur.
Örneğin sıralama işlemi yapan bir işlevin tanımı içinde *assert* makrosu şöyle kullanılabilir:

```
void sort_array(int *pArray, size_t size)
{
    int i, k, temp;
    /*…*/
    assert(pArray != NULL);
    /*…*/
}
```

Program içinde programın düzenlenme sistemine göre asla olmaması gereken bir durum oluşuyorsa, bir böcekten şüphelenilmesi gerektiği durumlarda *assert* makrosu kullanılmalıdır.

assert ifadeleri çok uzun tutulmamalıdır.
Çünkü bu durumda *assert* ifadesinin başarısızlığı durumunda neden anlaşılamayabilir.
Örneğin:

```
assert(row1 < 25 && col1 < 80 && row2 < 25 && col2 < 80);
```

yerine aşağıdaki kodun kullanılması daha anlamlıdır:

```
assert(row1 < 25 && col1 < 80);
assert(row2 < 25 && col2 < 80);
```

İŞLEV GÖSTERİCİLERİ

Çalıştırılabilen bir programın diskteki görüntüsüne *program* denir. Bir program çalıştırılmak üzere belleğe yüklendiğinde, yani çalışır duruma getirildiğinde *process* ya da *task* biçiminde isimlendirilir. Yani *program* diskteki durumu, *process* ise çalışmakta olan bir programı anlatır.

Çalıştırılabilen bir program içerik olarak üç kısımdan oluşur:

Çalıştırılabilen bir dosyanın işletim sistemi tarafından çalıştırılabilmesi için, önce diskten *RAM*'e yüklenmesi gerekir. İşletim sisteminin programı diskten alıp belleğe yükleyip çalıştıran kısmına *yükleyici (loader)* denir. Normal olarak bir programın belleğe yüklenmesi için, programın toplam büyüklüğünün bellekteki boş alandan küçük olması gerekir. Ancak UNIX, Windows gibi sistemlerde programlar küçük bir bellek altında çalıştırılabilir.

Bu özelliğe "sanal bellek" *(virtual memory)* özelliği denir. Sanal bellek kullanımı ile önce programın küçük bir kısmı *RAM*'e yüklenir. İşletim sistemi, program çalışırken programın akışının kod ya da data bakımından *RAM*'de olmayan bir kısma geçtiğini anlar. Programın *RAM*'deki kısmını boşaltarak bu kez gereken kısmını *RAM*'e yükler. Yani program parçalı olarak çalıştırılır. *DOS* işletim sistemi sanal bellek kullanımı yoktur. *DOS* sisteminin program çalıştırmaktaki bellek miktarı en fazla *640KB* kadardır.

C'deki tüm statik ömürlü varlıklar, yani global değişkenler, statik yerel değişkenler, dizgeler çalıştırılabilir dosyanın "veri" (*data)* bölümüne yerleştirilir. Statik ömürlü değişkenler bu bölüme ilkdeğerleriyle yerleştirilir. Yerel değişkenler ile parametre değişkenleri çalıştırılabilir dosyanın "yığın" (*stack*) bölümünde geçici olarak yaratılırlar. Programın tüm işlevleri derlenmiş olarak "kod" *(code)* bölümünde bulunur.

Nesnelerin nasıl adresleri varsa işlevlerin de adresleri vardır. Bir işlevin adresi, o işlevin makine kodlarının bellekteki başlangıç adresidir. İşlevlerin başlangıç adresleri işlev göstericileri denilen özel göstericilere yerleştirilebilir. Bir işlev göstericisine geri dönüş değeri ve parametreleri belirtilen türden olan bir işlevin adresi atanabilir.

İşlev Göstericisi Değişkenlerin Tanımlanması

Bir işlev göstericisi özel bir sözdizimi ile tanımlanır:

```
int (*fp)(void);
```

Yukarıdaki deyim ile ismi *fp* olan bir işlev gösterici değişkeni tanımlanıyor. *fp* gösterici değişkeni, geri dönüş değeri *int* türden olan, parametre değişkeni olmayan bir işlevi gösterebilir. Bir başka deyişle, *fp* gösterici değişkenine geri dönüş değeri *int* türden olan, parametre değişkeni olmayan bir işlevin adresi atanabilir.

Tanımlamada '*' atomu ve bunu izleyen gösterici isminin ayraç içine alındığını görüyorsunuz. Bu ayracın soluna işlev göstericisinin göstereceği işlevin geri dönüş değerinin türü yazılır. Ayracın sağındaki ayraç içinde, işlev göstericisinin göstereceği işlevin parametre değişkenlerinin türü yazılır.
Bir başka örnek:

```
int (*fptr) (int, int);
```

Yukarıda tanımlanan *fptr* isimli gösterici değişkene geri dönüş değeri *int* türden olan, parametreleri *(int, int)* olan bir işlevin adresi atanabilir.

Tanımlamada gösterici değişkenin ismi eğer ayraç içine alınmazsa, bir işlev göstericisi tanımlanmış olmaz. Bir adres türüne geri dönen bir işlev bildirilmiş olur:

```
int *f(int, int);
```

Yukarıda, ismi *f* olan geri dönüş değeri (*int* *) türü olan parametreleri (*int, int*) olan bir işlev bildirilmiştir.

İşlev İsimleri

Bir dizi isminin, bir işleme sokulduğunda, otomatik olarak dizinin ilk elemanının adresine dönüştürüldüğünü biliyorsunuz. Benzer şekilde bir işlev ismi de, bir işleme sokulduğunda derleyici tarafından otomatik olarak ilgili işlevin adresine dönüştürülür. İşlev isimlerine, yazılımsal olarak işlevlerin adresleri gözüyle bakılabilir.
Bir işlev ismi adres işlecinin de terimi olabilir. Bu durumda yine işlevin adresi elde edilir. Yani *func* bir işlev ismi olmak üzere

```
func
```

ile

```
&func
```

eşdeğer ifadeler olarak düşünülebilir. Her iki ifade de *func* işlevinin adresi olarak kullanılabilir.

İşlev Çağrı İşleci

işlev çağrı işleci tek terimli sonek konumunda bir işleçtir. İşleç öncelik tablosunun en üst düzeyindedir. Bu işleçin terimi bir işlev adresidir. İşleç, programın akışını o adrese yöneltir, yani o adresteki kodu çalıştırır. Örneğin:

```
func()
```

Burada işlev çağrı işlecinin terimi *func* adresidir. İşleç, *func* adresinde bulunan kodu, yani *func* işlevini çalıştırır. İşlev çağrı işlecinin ürettiği değer, çağrılan işlevin geri dönüş değeridir. Örneğin:

```
a = add(10, 20);
```

Burada *add* ifadesinin türü geri dönüş değeri *int* parametresi (*int, int*) olan bir işlev adresidir. Yani *add* ifadesinin

```
int (*)(int, int)
```

türüne dönüştürülür.

```
add(10, 20)
```

gibi bir işlev çağrısının oluşturduğu ifade ise *int* türdendir.

İşlevlerin İşlev Göstericileri ile Çağrılması

Bir işlev göstericisinin gösterdiği işlev iki ayrı biçimde çağrılabilir. *pf* bir işlev göstericisi değişken olmak üzere, bu göstericinin gösterdiği işlev

```
pf()
```

biçiminde, ya da

```
(*pf)()
```

biçiminde çağrılabilir.
Birinci biçim daha doğal görünmekle birlikte, *pf* isminin bir gösterici değişkenin ismi mi, yoksa bir işlevin ismi mi olduğu çok açık değildir.
İkinci biçimde, **pf* ifadesinin öncelik ayracı içine alınması zorunludur. Çünkü işlev çağrı işlecinin öncelik seviyesi, içerik işlecinin öncelik seviyesinden daha yüksektir.
Aşağıdaki programı derleyerek çalıştırın:

```
#include <stdio.h>

void func()
{
    printf("func()\n");
}

int main()
{
    void (*pf)(void) = func;

    pf();
    (*pf)();

    return 0;
}
```

main işlevi içinde tanımlanan *pf* isimli işlev göstericisi değişkene *func* isimli işlevin adresi atanıyor. Daha sonra *pf*'nin gösterdiği işlev, yani *func* işlevi iki ayrı biçimle çağrılıyor.

İşlev Göstericileri ve Gösterici Aritmetiği

Bir işlev göstericisi gösterici aritmetiği kurallarına uymaz. İşlev adresleri diğer adresler gibi tamsayılarla toplanamaz. İşlev göstericileri ++ ya da -- işlecinin terimi olamaz. Zaten gösterici aritmetiğinin işlev adresleri için bir anlamı da olamazdı. Bir işlev adresine bir tamsayı toplayıp bellekte bir sonraki işlevin adresine ulaşmak nasıl mümkün olurdu? İşlev adresleri içerik işlecinin terimi olabilir. Böyle ifadeler ile, yalnızca işlev çağrısı yapılabilir. Bu göstericiler karşılaştırma işleçlerinin de terimi olabilir. Örneğin, iki işlev göstericisinin aynı işlevi gösterip göstermediği == ya da != işleçleriyle karşılaştırılabilir.

İşlev Göstericilerinin İşlev Parametreleri Olarak Kullanılması

Bir işlevin parametre değişkeni işlev göstericisi olabilir. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

void func(void (*pf)(void))
{
    pf();
}

void f1()
{
    printf("f1()\n");
}

void f2()
{
    printf("f2()\n");
}
```

```
int main()
{
    func(f1);
    func(f2);

    return 0;
}
```

Yukarıdaki programda, *func* isimli işlevin parametre değişkeni bir işlev göstericisidir. *func* işlevinin parametre değişkeni olan gösterici, geri dönüş değeri ve parametre değişkeni olmayan bir işlevi gösterebilir. İşlevlerin parametre değişkenleri, değerlerini işlev çağrılarındaki argümanlardan aldığına göre *func* işlevine argüman olarak, geri dönüş değeri ve parametre değişkeni olmayan bir işlev adresi gönderilmelidir. *func* işlevi içinde, işlevin parametresi olan *fp* isimli göstericinin gösterdiği işlev çağrılır.
f1 ve *f2* geri dönüş değeri ve parametre değişkeni olmayan işlevlerdir. *main* işlevi içinde *func* işlevinin önce *f1* işlevin adresi ile daha sonra da *f2* işlevinin adresi ile çağrıldığını görüyorsunuz. Bu durumda

```
func(f1);
```

çağrısıyla önce *func* işlevi çağrılır. *func* işlevi içinde de *f1* işlevin çağrılması sağlanır.

İşlev Göstericileri Niçin Kullanılır

İşlev göstericileri, bir işlevin belirli bir kısmının dışarıdan değiştirilmesine olanak sağlar. Yani işlev, bir amacı gerçekleştirecek belirli işlemleri yapmak için, dışarıdan adresini aldığı bir işlevi çağırabilir. Böylece işlevi çağıran kod parçası, çağırdığı işlevin işinin belirli bir kısmının, kendi istediği gibi yapılmasını sağlayabilir.
Böylece yazılan kaynak kod miktarı da azalabilir. Kaynak kodlarının büyük bir kısmı aynı olan çok sayıda işlev yazmak yerine, işlev göstericisiyle çağrılan tek bir işlev yazmak mümkün olabilir. İşlev göstericileri ile çağrılan işlevler, genelleştirilmiş işlevlerdir. Dışarıdan adreslerini aldıkları işlevlerin çağrılmalarıyla işlevleri daha özel hale getirilmiş olur.

Türden Bağımsız İşlem Yapan İşlevler

Bazı işlevler, özellikle gösterici parametresi alan işlevler, belirli bir türe dayalı olarak çalışır. Örneğin, elemanları *int* türden olan bir diziyi sıralayan bir işlev, elemanları *double* türden olan bir diziyi sıralayamaz. Bu tür durumlarda aynı kodu, farklı türlere göre yeniden yazmak gerekir. Türden bağımsız olarak tek bir işlevin yazılabilmesi mümkün olabilir. Türden bağımsız işlem yapan işlevlerin gösterici parametreleri *void* * türünden olmalıdır. Ancak parametrelerin *void* * türünden olması yalnızca çağrı açısından kolaylık sağlar. İşlevi yazacak olan programcı, yine de işleve geçirilen adresin türünü saptamak zorundadır. Bunun için işleve bir tür parametresi geçirilebilir. Örneğin herhangi bir türden diziyi sıralayacak işlevin bildirimi aşağıdaki gibi olsun:

```
void *gsort(void *parray, size_t size, int type);
```

Şimdi işlevi yazacak kişi *type* isimli parametreyi bir *switch* deyiminin ayracı içine alarak, dışarıdan adresi alınan dizinin türünü saptayabilir. Bu yöntem C'nin doğal türleri için çalışsa da programcının kendi oluşturduğu türden diziler için doğru çalışmaz. Böyle genel işlevler ancak işlev göstericileri kullanılarak yazılabilir.
Şimdi bir dizinin en büyük elemanın adresiyle geri dönen bir işlevi türden bağımsız olarak yazmaya çalışalım. Dizi türünden bağımsız olarak işlem yapan işlevlerin, genel parametrik yapısı genellikle aşağıdaki gibi olur:
Dizinin başlangıç adresini almak için, *void* türden bir gösterici.
Dizinin eleman sayısını alan, *size_t* türünden bir parametre.

Dizinin bir elemanın *byte* uzunluğunu yani *sizeof* değerini alan *size_t* türünden bir parametre.
Dizinin elemanlarını karşılaştırma amacıyla kullanılacak bir işlevin başlangıç adresini alan, işlev göstericisi parametre.
Bir dizinin en büyük elemanının adresini bulan genel işlevin parametrik yapısı şöyle olabilir:

```
void *get_max(const void *pArray,  size_t size, size_t width, int
(*cmp)(const void *, const void *));
```

Bu tür işlevlerin tasarımındaki genel yaklaşım şudur:
İşlev herbir dizi elemanının adresini, gösterici aritmetiğinden faydalanarak bulabilir. Ancak dizi elemanlarının türü bilinmediğinden, işlev dizinin elemanlarının karşılaştırma işlemini yapamaz. Bu karşılaştırmayı, işlev göstericisi kullanarak işlevi çağıran kod parçasına yaptırır. İşlevi çağıracak programcı, karşılaştırma işlevinin dizinin herhangi iki elemanının adresiyle çağrılacağını gözönüne alarak, karşılaştırma işlevini şöyle yazar:
Karşılaştırma işlevi, karşılaştırılacak iki nesnenin adresini alır. İşlevin birinci parametresine adresi alınan nesne, ikinci parametreye adresi alınan nesneden daha büyükse, işlev *pozitif* herhangi bir değere, küçükse *negatif* herhangi bir değere, bu iki değer eşitse *sıfır* değerine geri döner. Bu, standart *strcmp* işlevinin sunduğu anlaşmadır.
Aşağıdaki programı inceleyin:

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

#define   MAX_NAME_LEN  20

typedef struct tag_Person {
    char name[MAX_NAME_LEN];
    int no;
}Person;

typedef unsigned char Byte;

void *get_max(const void *parray, size_t size, size_t width, int
(*fp)(const void *, const void *))
{
    Byte *pb = (Byte *)parray;
    void *pmax = (void *)parray;
    size_t k;

    for (k = 1; k < size; k++)
        if (fp(pb + k * width, pmax) > 0)
            pmax = pb + k * width;
    return pmax;
}

int cmp_int(const void *vp1, const void *vp2)
{
    return *(const int *)vp1 - *(const int *)vp2;
}

int cmp_person_name(const void *vp1, const void *vp2)
{
    return strcmp(((const Person *)vp1)->name, ((const Person *)vp2)-
>name);
}
```

```
int cmp_person_no(const void *vp1, const void *vp2)
{
    return ((const Person *)vp1)->no - ((const Person *)vp2)->no;
}

int main()
{
    int a[10] = {3, 8, 4, 7, 6, 9, 12, 1, 9, 10};

    Person per[10] = { {"Ali Serce", 123}, {"Kaan Aslan", 563}, {"Ahmet
    Adlori", 312},{"Necati Ergin", 197}, {"Guray Sonmez", 297}, {"Erdem
    Eker", 144}, {"Nuri Yilmaz", 765}, {"Tayfun Tan", 117}, {"Demir Kerim",
    222},{"Can Mercan", 12}};

    Person *p_person;
    int *iptr;

    iptr = (int *)get_max(a, 10, sizeof(int), cmp_int);
    printf("int dizinin en buyuk elemani %d\n", *iptr);

    p_person = (Person *) get_max(per, 10, sizeof(Person),
cmp_person_name);
    printf("per dizisinin en buyuk elemani (isme gore) %s %d\n",
    p_person->name, p_person->no);
    p_person = (Person *) get_max(per, 10, sizeof(Person), cmp_person_no);
    printf("per dizisinin en buyuk elemani (numaraya gore) %s %d\n",
    p_person->name,    p_person->no);

    return 0;
}
```

Yukarıdaki programda *get_max* isimli işlev bir dizinin başlangıç adresini, boyutunu, dizinin bir elemanının uzunluğu ile dizinin elemanlarını karşılaştırma işleminde kullanılacak işlevin adresini alıyor. İşlev çağrısıyla dizi adresi *void* türden bir göstericiye alınıyor. Ancak işlev içinde gösterici aritmetiğinden faydalanmak amacıyla bu adres *char* türden bir göstericiye aktarılıyor. Bir elemanın boyutu bilindiğine göre bu değer dizinin başlangıç adresine eklenerek dizinin bir sonraki elemanının adresi bulunabilir değil mi? Karşılaştırma işlevi, karşılaştırma işlemini yapabilmek için iki nesnenin adresini alıyor. Böylece *get_max* işlevi içinde, karşılaştırma işlevi, parametre olan işlev göstericisi ile doğru argümanlarla çağırılabiliyor. Dizinin en büyük elemanının adresini bulmak için yine bilinen algoritma kullanılıyor. Ancak dizinin iki elemanının değerini karşılaştırmak için dışarıdan adresi alınan işlev çağrılıyor.

Daha sonra üç ayrı karşılaştırma işlevinin tanımlanmış olduğunu görüyorsunuz. Tüm karşılaştırma işlevlerinin parametre değişkenleri (*const void *, const void *)* dır. Böylece, bu karşılaştırma işlevlerine *get_max* işlevi iki adres gönderdiğinde bir tür uyumsuzluğu sorunu ortaya çıkmaz.

main işlevi içinde bir *int* türden bir de *Person* türünden dizi tanımlanıyor. Bu dizilere ilkdeğer verilmiş olduğunu görüyorsunuz. Daha sonra *get_max* işlevi çağrılarak dizilerin en büyük elemanlarının adresleri bulunuyor. *cmp_person_name* işlevi *Person* türünden iki nesneyi isimlerine göre karşılaştırırken, *cmp_person_no* işlevi ise *Person* türünden iki nesneyi, numaralarına göre karşılaştırıyor.

Türden Bağımsız Sıralama

Her türlü sıralama algoritmasında karşılaştırma ve değer takas etme söz konusudur. Bir dizinin türü bilinmese bile dizinin iki elemanı takas edilebilir. Karşılaştırma işleminini yapacak işlevin adresi çağıran kod parçasından alınırsa, türden bağımsız sıralama işlemi gerçekleştirilebilir. Aşağıdaki örneği inceleyin:

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

#define     SIZE    10

typedef unsigned char BYTE;

void gswap(void *vp1, void *vp2, size_t nbytes)
{
    unsigned char *p1 = (unsigned char *)vp1;
    unsigned char *p2 = (unsigned char *)vp2;
    unsigned char temp;

    while (nbytes--) {
        temp = *p1;
        *p1++ = *p2;
        *p2++ = temp;
    }
}

void bsort(void *vparray, size_t nelem, size_t width, int (*compare)(const
void *, const void *))
{
    size_t i, j;
    BYTE *base = (BYTE *)vparray;

    for (i = 0; i < nelem - 1; i++)
        for (j = 0; j < nelem - 1 - i; j++)
            if (compare(base + j * width, base + (j + 1) * width) > 0)
                gswap(base + j * width, base + (j + 1) * width, width);
}

int cmp_int(const void *vp1, const void *vp2)
{
    return *(const int *)vp1 - *(const int *)vp2;
}

int cmp_double(const void *vp1, const void *vp2)
{
    const double *dp1 = (const double *) vp1;
    const double *dp2 = (const double *) vp2;

    if (*dp1 < *dp2)
        return -1;

    return *dp1 > *dp2;
}

int main()
{
    int k;
    double ad[SIZE] = {4.3, 7.4, 2.7, 8.88, 66.99, 4.8, 90.67, 23.87, 7.89,
10.87};
    int ai[SIZE] = {12, 4, 56, 45, 23, 60, 17, 56, 29, 1};

    bsort(ad, SIZE, sizeof(double), cmp_double);
    bsort(ai, SIZE, sizeof(int), cmp_int);

    for (k = 0; k < SIZE; ++k)
        printf("%.2lf ", ad[k]);
```

```
    printf("\n");

    for (k = 0; k < SIZE; ++k)
        printf("%d ", ai[k]);

    return 0;
}
```

g_swap isimli işlev, başlangıç adreslerini aldığı belirli uzunlukta iki bellek bloğunun içeriğini takas ediyor. Bu işlev, türden bağımsız sıralama işlemini gerçekleştiren *bsort* isimli işlev tarafından çağrılıyor. *bsort* işlevinin parametrik yapısının, daha önce yazılan *get_max* isimli işleve benzer olduğunu görüyorsunuz. İşlevin birinci parametresi sıralanacak dizinin adresi, ikinci parametresi dizinin eleman sayısı, üçüncü parametresi dizinin bir elemanının uzunluğu ve dördüncü parametresi ise dizinin elemanlarını karşılaştırma işlemini gerçekleştirecek işlevin adresidir.
bsort işlevi içinde *"kabarcık sıralaması"* algoritması kullanılıyor. Dizinin ardışık iki elemanının büyüklük-küçüklük ilişikisi dışarıdan adresi alınan işlevin, parametre değişkeni olan gösterici ile çağrılmasıyla elde ediliyor. Eğer dizinin bu iki elemanı doğru yerde değilse bu elemanlar daha önce tanımlanan *gswap* işlevinin çağrılmasıyla takas ediliyor. Böyle bir işlevle her türden dizi sıraya sokulabilir. *bsort* işlevi yalnızca karşılaştırma işlemini gerçekleştirecek uygun bir işlevin adresine gereksinim duyar. Çünkü bu işlev dışarıdan adresi gelen dizinin elemanlarının türünü bilemez.
İşlev göstericileri tanımlarken parametre ayracının içinin boş bırakılması ile parametre ayracının içerisine *void* yazılması farklı anlamlardadır. Parametre ayracının içi boş bırakılırsa, böyle tanımlanan işlev göstericisi değişkenlere, geri dönüş değeri uygun herhangi bir parametre yapısına sahip olan işlevin adresi yerleştirilebilir. Oysa *void* yazılması, parametresi olmayan işlevlerin adreslerinin yerleştirilebileceği anlamına gelir. Parametre ayracının içi boş bırakılarak, çağrılacak işlevlerin parametrik yapısı dışarıdan istendiği gibi düzenlenebilir.

[C++ dilinde parametre ayracının içinin boş bırakılması, işlev göstericisi değişkenin, göstereceği işlevin parametreye sahip olmadığı bilgisini iletir. C++'da ayracın içine void yazılmasıyla, buranın boş bırakılması arasında bir anlam farkı yoktur.]

Standart qsort İşlevi

Standart *qsort* işlevi, yukarıda tanımlanan *bsort* işleviyle aynı parametrik yapıya sahiptir. Yazdığımız *bsort* işlevi kabarcık sıralaması algoritmasını kullanıyordu. Standart *qsort* işlevi ise "*quick sort* " algoritmasını kullanır.
qsort işlevinin *stdlib.h* başlık dosyası içindeki bildirimi aşağıdaki gibidir:

```
void qsort(void *vp, size_t size, size_t width, int (*cmp)(const void *,
const void *));
```

İşlevin birinci parametresi sıraya dizilecek dizinin başlangıç adresi, ikinci parametresi dizinin eleman sayısı, üçüncü parametresi dizinin bir elemanın uzunluğu ve son parametresi de karşılaştırma işlevinin adresidir. Karşılaştırma işlevi, birinci parametresine adresi gönderilen elemanın değeri, ikinci parametresine adresi gönderilen elemandan büyükse *pozitif* herhangi bir değere, küçükse *negatif* herhangi bir değere, eşitse *sıfıra* geri dönecek biçimde yazılırsa küçükten büyüğe sıraya dizme, ters yazılırsa büyükten küçüğe sıraya dizme gerçekleşir.

Standartlar öncesi C derleyicilerinde *qsort* işlevinin bildiriminde karşılaştırma işlevinin adresini alacak göstericinin parametre ayracının içi boş bırakılmıştır. Bu durum karşılaştırma işlevini yazmayı kolaylaştırır. İstenirse *qsort* İşlevinda böyle bir kolaylık elde etmek için *stdlib.h* başlık dosyası içinde bulunan bildirimde işlev gösterici ayracı boşaltılabilir:

```
void qsort(void *parray, size_t size, size_t width, void(*fp)());
```

qsort işlevi ile bir gösterici dizisinin sıralanmasına dikkat edilmelidir.
Aşağıdaki programı inceleyin ve sıralama işleminin neden yapılamadığını anlamaya çalışınız:

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

#define     SIZE 20

int cmp(const void *vp1, const void *vp2)
{
    const char *p1 = (const char *)vp1;
    const char *p2 = (const char *)vp2;

    return strcmp(p1, p2);
}

int main()
{
    char *names[SIZE] = {"Ali", "Veli", "Hasan", "Necati", "Burcu", "Kaan",
    "Selami", "Salah", "Nejla", "Huseyin", "Derya", "Funda", "Kemal",
    "Burak", "Ozlem", "Deniz", "Nuri","Metin", "Guray", "Anil"};

    int k;

    qsort(names, SIZE, sizeof(char *), cmp);

    for (k = 0; k < SIZE; ++k)
        printf("%s ", names[k]);

    return 0;
}
```

Hatayı görebildiniz mi? Karşılaştırma işlemini gerçekleştirecek *cmp* isimli işlev yanlış yazılmış. Bu işleve *char* * türünden iki nesnenin adresi gönderileceğine göre, işlev içinde yapılan tür dönüştürme işleminde, hedef tür *char** türü değil *char*** türü olmalıydı. *(char *)* türünden bir nesnenin adresi olan bilgi *(char **)* türündendir.
İşlev aşağıdaki gibi yazılmalıydı:

```
int cmp(const void *vp1, const void *vp2)
{
    const char * const *p1 = (const char * const *)vp1;
    const char * const *p2 = (const char * const *)vp2;

    return strcmp(*p1, *p2);
}
```

İşlev Göstericisi Dizileri

Elemanları işlev göstericileri olan diziler de tanımlanabilir:

```
void (*fparray[10])(void);
```

Yukarıdaki tanımlama ile *fparray,* her bir elemanı, geri dönüş değeri ve parametre değişkeni olmayan bir işlevi gösteren *10* elemanlı bir dizidir.

```
fparray[2]
```

ifadesiyle bu dizinin üçüncü elemanına ulaşılır ki bu ifadenin türü

```
void(*)(void)
```

türüdür. Bu göstericinin gösterdiği işlev

```
fparray[2]();
```

biçiminde ya da

```
(*fparray[2])() ;
```

biçiminde çağrılabilir. Böyle bir dizinin elemanlarına da ilkdeğer verilebilir:

```
void (*fparray[5])(void) = {f1, f2, f3, f4, f5};
```

Böylece, dizinin her bir elemanının bir işlevi göstermesi sağlanabilir.
Artık bir döngü deyimi yardımıyla gösterici dizisinin elemanlarına ulaşılarak dizinin elemanlarının gösterdiği işlevler çağrılabilir. Aşağıdaki kodu derleyerek çalıştırın:

```
#include <stdio.h>

void f1() {printf("f1()\n");}
void f2() {printf("f2()\n");}
void f3() {printf("f3()\n");}
void f4() {printf("f4()\n");}
void f5() {printf("f5()\n");}

int main()
{
    void (*fparray[5])(void) = {f1, f2, f3, f4, f5};
    int k;

    for (k = 0; k < 5; ++k)
        fparray[k]();

    return 0;
}
```

Bir İşlevin Bir İşlev Adresini Geri Döndürmesi

Bir işlevin geri dönüş değeri de bir işlev adresi olabilir. Aşağıdaki örneği inceleyin:

```
#include <stdio.h>

void foo()
{
    printf("foo()\n");
}

void (*func(void))(void)
{
    void (*fp)(void) = foo;

    return fp;
}

int main()
{
    func()();
    return 0;
}
```

Yukarıdaki programda tanımlanan *func* isimli işlevin parametre değişkeni yok. *func* işlevinin geri dönüş değeri, -geri dönüş değeri ve parametre değişkeni olmayan- bir işlev adresidir. Yani *func* işlevi, kendisini çağıran kod parçasına böyle bir işlevin adresini iletiyor. *func* işlevi içinde tanımlanan yerel *fp* gösterici değişkeni, *foo* isimli işlevi gösteriyor. *main* işlevi içinde yer alan

```
func()();
```

deyiminde önce *func* işlevi çağrılır. Bu çağrıdan elde edilen geri dönüş değeri, *foo* işlevinin adresi olur. İkinci işlev çağrı işleci de *foo* işlevin çağrılmasını sağlar.

İşlevler İçin Yapılan typedef Bildirimleri

Bir işlev türü için *typedef* bildirimi yapılabilir:

```
typedef double FuncDouble(int);
```

Yukarıdaki bildirimden sonra *FuncDouble*, geri dönüş değeri *double* türden olan, tek parametresi *int* türden olan bir işlev türüdür.

```
FuncDouble *fptr, *fpa[10];
```

Yukarıdaki tanımlama ile *fptr*, geri dönüş değeri *double* ve parametresi *int* türden olan bir işlevi gösteren gösterici değişkendir. *fpa* ise *fptr* gibi göstericilerin oluşturduğu *10* elemanlı bir dizidir. *fpa* dizisinin herbir elemanı bir işlev göstericisidir.

Elemanları *FuncDouble* türden olan bir değişken tanımlanamaz. C

```
FuncDouble func;  /* Geçersiz */
```

Ancak bu türden bir işlevin bildiriminde bir sorun yoktur:

```
extern FuncDouble func;
```

Yukarıdaki deyimle, geri dönüş değeri *double* türden olan, parametre değişkeni *int* türden olan bir işlev bildiriliyor.

Uygulamalarda daha çok bir işlev adresi türü için *typedef* bildirimi yapılır:

```
typedef int (*CmpFunc)(const void *, const void *);
```

Yukarıdaki bildirimden sonra *CmpFunc*, geri dönüş değeri *int* türden olan, iki parametresi de *const void* * türünden olan bir işlev adresi türüdür:

```
CmpFunc fcmp1, fcmp2, fcmpa[10];
```

Yukarıdaki tanımlama ile *fcmp1* ve *fcmp2*, geri dönüş değeri *int* türden olan, iki parametresi de *const void* * türünden olan bir işlevi gösterecek değişkenlerdir. *fcmp1* ve *fcmp2* işlev göstericisi değişkenlerdir. *fcmpa* ise, her bir elemanı *fcmp* değişkeninin türünden olan, *10* elemanlı bir dizidir.

Bir işlev adresi türünün, bir işlevin parametre değişkeninin türü ya da işlevin geri dönüş değerinin türü olması bildirimleri oldukça karışık hale getirir. Okuma ve yazma kolaylığı açısından işlev adreslerine ilişkin türlere *typedef* bildirimleriyle yeni isimler verilir. Aşağıdaki örneği inceleyin:

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

typedef char *(*FPTR)(const char *, int);

FPTR func(FPTR ptr)
{
    return ptr;
}

int main()
{
    char str[] = "Necati Ergin";

    puts(func(strchr)(str, 'E'));

    return 0;
}
```

Yukarıdaki programın çalıştırılmasıyla ekrana *Ergin* yazısı yazdırılır.

ÖZYİNELEMELİ İŞLEVLER

Bir işlev başka bir işlevi çağırdığı gibi kendisini de çağırabilir. Bu tür işlevlere özyinelemeli işlev (*recursive functions*) denir. Algoritmalar bu özelliğe göre üç ayrı gruba ayrılabilir:

1- Özyinelemeli olmayan işlevlerle yazılabilen algoritmalar.
2- Hem özyinelemeli olmayan hem de özyinelemeli işlevlerle yazılabilen algoritmalar.
3- Yalnızca özyinelemeli işlevlerle yazılan algoritmalar.

Özyinelemeli işlevlerle yazılması gereken tipik algoritmalar şunlardır:

Dizin ağacını dolaşan algoritmalar
Parsing algoritmaları
Ağaç algoritmaları
Özel amaçlı pek çok algoritma

Bir algoritmanın kendi kendisini çağıran işlevlerle yazılıp yazılamayacağının ölçütü nedir? Algoritmada ilerlendiğinde yine ilk baştakine benzer bir durumla karşılaşılıyorsa büyük olasılıkla bu algoritma kendi kendini çağıran bir işlevle yazılabilir.

Bir işlev kendini çağırdığında işlevin bütün yerel değişkenleri yeniden yaratılır. Aynı durum işlevin parametre değişkeni için de söz konusudur. Bu yüzden özyinelemeli işlevler genellikle yığın alanını diğer işlevlerle karşılaştırıldığında daha çok kullanır.
Özyinelemeli işlevler işlemi gerçekleştirmek amacıyla bir süre kendi kendini çağırır. Daha sonra işlevlerden çıkış işlemleri başlar, yani çağrılan tüm işlevlerden geri çıkılmaya başlanır.

Özyinelemeli İşlevlere Tipik Örnekler

Aşağıda faktöriyel hesaplayan döngüsel (*iterative)* bir işlev yazılıyor:

```
int fact(int n)
{
    int result = 1;

    while (n)
        result *= n--;
    return result;
}
```

Aynı işlev özyinelemeli olarak da yazılabilir:

```
int fact(int n)
{
    if (n == 0 || n == 1)
        return 1;
    return  n * factorial(n - 1);
}
```

İşlevin akışı şöyledir:

```
n = 5 için val = 5 * factorial(4);
n = 4 için val = 4 * factorial(3);
n = 3 için val = 3 * factorial(2);
n = 2 için val = 2 * factorial(1);
n = 1 için return 1;
```

Döngüsel yapıyla yazılan faktöriyel işlevinin daha verimli olduğu söylenebilir.
Aşağıda bir tamsayıyı ikilik sayı sisteminde yazdıran döngüsel yapılı bir işlev tanımlanıyor:

```
#include <stdio.h>

void bitprint(int i)
{
    int k;

    for (k = sizeof(int) * 8 - 1; k >= 0; --k)
        putchar((i >> k & 1) + '0');
}
```

İşlev özyinelemeli olarak aşağıdaki biçimde tanımlanabilir:

```
#include <stdio.h>

void bitprint(int i)
{
    if (i == 0)
        return;
    bitprint(i >> 1);
    putchar(i & 1 + '0');
}
```

Yine özyinelemeli olmayan biçimin daha verimli olduğu söylenebilir.
Aşağıda bir yazıyı tersten yazdıran döngüsel yapılı bir işlev tanımlanıyor:

```
#include <stdio.h>

void putsrev(const char *str)
{
    int i;

    for (i = strlen(str) - 1; i >= 0; --i)
        putchar(str[i]);

    putchar('\n');
}
```

İşlev özyinelemeli olarak aşağıdaki biçimde tanımlanabilir:

```
#include <stdio.h>

void putsrev(const char *str)
{
    if (*str == '\0')
        return;
    putsrev(str + 1);
    putchar(*str);
}
```

Aşağıda tanımı verilen *ters_yaz* isimli işlev klavyeden girilen bir yazıyı tersten yazdırıyor:

```
#include <stdio.h>

void ters_yaz()
{
    char ch;
```

```
    if ((ch = getchar())!='\n')
        ters_yaz();
    putchar(ch);
}
```

Aşağıda bir tamsayıyı yalnızca *puthar* işlevini kullanarak yazdıran döngüsel yapılı bir işlev tanımlanıyor:

```
#include <stdio.h>

#define    SIZE       100

void printd(int i)
{
    char s[SIZE];
    int k = 0;
    int fnegative = 0;

    if (i < 0) {
        fnegative = 1;
        i = -i;
    }

    do {
        s[k++] = i % 10 + '0';
        i /= 10;
    } while(i);

    if(fnegative)
        s[k++] = '-';

    for (k--; k >= 0; --k)
        putchar(s[k]);
}
```

İşlev özyinelemeli olarak aşağıdaki biçimde tanımlanabilir:

```
#include <stdio.h>

void print(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        print(n / 10);
    putchar(n % 10 + '0');
}
```

Aşağıda bir tamsayının belirli bir tamsayı üssüne geri donen *power* isimli işlev özyinelemeli olarak yazılıyor:

```
int power(int base, int exp)
{
    if (exp == 0 )
        return 1;

    return base * power(base, exp - 1);
}
```

Aşağıda iki tamsayının ortak bölenlerinden en büyüğüne geri dönen *get_gcd* isimli işlev özyinelemeli olarak tanımlanıyor:

```
int get_gcd(int a, int b)
{
    if (a >= b && a % b == 0)
        return b;

    if (a < b)
        return(get_gcd(b, a));

    return get_gcd(b, a % b);
}
```

Aşağıda tanımlanan *fibonacci* isimli işlev *fibonacci* serisinin *n*. terimine geri dönüyor:

```
int fibonacci(int n)
{
    if (n == 1 || n == 2)
        return 1;

    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Aynı işi yapan döngüsel yapıda bir işlev çok daha verimli olurdu:

```
int fibonacci(int n)
{
    int x = 1;
    int y = 1;
    int k, result;

    for (k = 2; k < n; ++ k) {
        result = x + y;
        x = y;
        y = result;
    }
    return y;
}
```

Aşağıda özyinelemeli olarak tanımlanan *rakam_yaz* işlevi bir tamsayının her bir basamağını ekrana yazı olarak basıyor:

```
#include <stdio.h>

void rakam_yaz(int n)
{
    static const char *rakamlar[ ] = { "Sifir", "Bir", "Iki", "Uc", "Dort",
"Bes", "Alti", "Yedi", "Sekiz", "Dokuz" };

    if (n > 9)
        rakam_yaz(n / 10);
    printf("%s ", rakamlar[n % 10]);
}
```

Aşağıda bir tamsayıyı ekrana onaltılık sayı sisteminde yazdıran *to_hex* isimli özyinelemeli bir işlev tanımlanıyor:

```
void to_hex(int n)
```

```
{
    static const char *htab[ ] = { "0", "1", "2", "3", "4", "5", "6", "7",
"8", "9", "A", "B", "C", "D", "E", "F" };

    if (n > 15)
        to_hex(n / 16);
    printf("%s", htab[n % 16]);
}
```

Aşağıda bir yazıyı ters çeviren *strrev_r* isimli bir işlev tanımlanıyor.

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

#define SIZE    100

void swapstr(char *str, int l, int r)
{
    char temp;

    if (l >= r)
        return;
    temp = str[l];
    str[l] = str[r];
    str[r] = temp;

    swapstr(str, l + 1, r - 1);
}

char *strrev_r(char *str)
{
    swapstr(str, 0, strlen(str) - 1);

    return str;
}

int main()
{
    char str[SIZE];

    printf("bir yazi girin : ");
    gets(str);
    printf("yazi = (%s)\n", str);
    strrev_r(str);
    printf("yazi = (%s)\n", str);

    return 0;
}
```

Şüphesiz yazının ters çevrilmesi özyinelemesiz bir işlevle de gerçekleştirilebilir. Özyinelemeli yazılan *strrev_r* işlevinin üç parametresi olduğunu görüyorsunuz: Yazının başlangıç adresi, takas edilecek ilk ve son elemanın indis değerleri.
İşlevin *strrev* işlevinde olduğu gibi yazının yalnızca başlangıç adresini alması bir sarma işlev yazılarak sağlanıyor. Sarma işlev (*wrapper function*) işin kendisini yapan işlev değildir. Sarma işlev işi yapan işlevi çağıran küçük işlevlere denir.
Aşağıdaki programda tanımlanan *revprint* isimli işlev adresini aldığı bir yazının sözcüklerini ters sırayla ekrana yazdırıyor:

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

#define MAX_WORD_LEN 100
#define ARRAY_SIZE 1000

void revprint(const char *str)
{
    char s[MAX_WORD_LEN + 1] = "";
    int index = 0;
    static const char seps[] = " \n\t,.:;!?";

    while (*str && strchr(seps, *str))
        str++;
    if (!*str)
        return;
    while (!strchr(seps, *str))
        s[index++] = *str++;
    s[index] = '\0';

    if (*str == '\0') {
        printf("%s ", s);
        return;
    }

    revprint(str);
    printf("%s ", s);
}

int main()
{
    char str[ARRAY_SIZE];

    printf("bir cumle girin : ");
    gets(str);

    revprint(str);
    return 0;
}
```

DEĞİŞKEN SAYIDA ARGÜMAN ALAN İŞLEVLER

C'de değişken sayıda argüman alan işlevler tanımlanabilir. Değişken sayıda argüman alan bir işlevin bildirimi aşağıdaki gibidir:

```
[İşlevin geri dönüş değeri]işlev ismi(parametre bildirimleri, ...);
```

üç nokta "..." atomunun ismi ingilizce de "*elipsis*"tir. Bir işlevin değişken sayıda argüman alabilmesi için son parametresinin üç nokta atomuyla bildirilmesi zorunludur. Böyle işlevler istenen sayıda argümanla çağrılabilir.
Böyle işlevlerin soldan en az bir paramteresi bilinen bir türden olmalıdır.
Aşağıda bazı değişken sayıda argüman alan işlevlerin bildirimleri görülüyor:

```
int printf(const char *, ...);
int scanf(const char *, ...);
double get_mean(int, ...);
```

İşlevlerin tanımı da aynı biçimde yapılmalıdır. İşlev tanımlarında işleve gönderilen argümanlara bazı standart makrolar kullanılarak ulaşılır. Bu makrolar *stdarg.h* isimli standart başlık dosyası içinde tanımlanmıştır.
Değişken sayıda argüman alan işlev içinde, türü ve ismi belirtilmiş parametre değişkenlerinin değerine diğer işlevlerde olduğu gibi isimleriyle ulaşılır. Seçimlik olan argümanlara, yani kaç tane olduğu bilinmeyen argümanların her birine ulaşım bazı standart makrolar kullanılmasıyla gerçekleştirilir:
Önce bu makroları tanıyalım:

va_start Makrosu
Bu makro kullanılarak *vs_list* türünden bir değişkene ilkdeğeri verilir.
va_list standart <*stdarg.h*> başlık dosyası içinde bildirilen bir *typedef* ismidir. Seçeneğe bağlı argümanları dolaşacak gösterici bu türdendir. Önce bu türden bir değişken tanımlanmalıdır. Bu göstericinin argüman listesini dolaşma işlemini yapabilmesi için, önce bu değişkene *va_start* makrosu ile ilkdeğeri verilmelidir.
va_start makrosunun aşağıdaki gibi bir işleve karşılık geldiğini düşünebilirsiniz:

```
void va_start (va_list ap, belirli son parametre);
```

Bu makroya birinci argüman olarak *va_list* türünden değişken gönderilir. Makroya gönderilmesi gereken ikinci argüman işlevin türü ve ismi belirli olan son parametre değişkeninin değeridir.

```
#include <stdarg.h>

int add_all(int val, …)
{
    va_list va;
    va_start(va, val);
    /***/
}
```

add_all isimli işlev kendisine gönderilen değişken sayıda pozitif tamsayıların toplam değeriyle geri dönecek. İşlevin ilk parametresinin int türden olduğunu görüyorsunuz. Bu işleve en az bir argüman gönderilmelidir. İşlev çağrısında kullanılacak toplam argüman sayısı isteğe bağlı olarak ayarlanabilir. Ancak son argümanın değeri *-1* olduğunda, başka

bir argüman göderilmediği anlaşılırt. Yani işleve gönderilen son değerin *-1* olması gerektiği işlevin arayüzünün bir parçası oluyor.
İşlev tanımı içinde önce *va_list* türünden va isimli bir değişkenin tanımlandığını görüyorsunuz. *va_start* makrosuna *va* değişkeni ve işlevin parametre değişkeni *val* argüman olarak gönderiliyor. Bu çağrıya karşılık gelen içsel kodun çalışmasıyla, va değişkeni argüman listesinde uygun argümanı gösterecek şekilde konumlandırıldığını düşünebilirsiniz. Böylece artık işleve gönderilen argümanların değeri elde edilebilir.

va_arg Makrosu

va_arg makrosu işleve gönderilen argümanların değerlerini elde etmek için kullanılır.

Bu makro bir işlev gibi düşünüldüğünde aşağıdaki gibi gösterilebilir:

```
tür va_arg (va_list ap, tür);
```

va_list türünden değişkenimiz *va_start* makrosu ile ilkdeğerini aldıktan sonra bu makro seçneğe bağlı tüm argümanları elde etmek için çağrılır. Bu makrodan elde edilen seçeneğe bağlı argümanın değeridir. Makronun çalışması sonucu argüman listesi göstericisi de bir sonraki argümanı gösterecek biçimde konumlandırılmış olur.
Bu makroya yapılan bir sonraki çağrı ile bir sonraki seçimlik argümanın değeri elde edilir. Makroya gönderilecek ikinci bilgi, değeri elde edilecek argümanın türüdür. Örneğin seçimlik argüman *int* türdense makronun ikinci argümanına

int

geçilmelidir. Yeniden *add_all* işlevine dönüyoruz:

```
int add_all(int val, ...)
{
    va_list va_p;
    int sum = 0;

    va_start(va_p, val);

    while (val != -1) {
        sum += val;
        val = va_arg(va_p, int);
    }
    /***/
}
```

işlevin tanımında val değişkeni -1 olmadığı sürece dönen bir döngü oluşturuluyor. Döngünün her turunda val değişkeninin değeri *sum* değişkenine katılıyor ve val değişkenine seçimlik argümanlardan bir sonrakinin değeri aktarılıyor.

```
val = va_arg(va_p, int);
```

va_arg makrosunun ilk defa kullanılmasıyla seçimlik ilk argüman, 2. kez kullanılmasıyla seçimlik 2. argüman vs. elde edilir. Eğer belirli bir argümandan sonrası elde edilmek istenmiyorsa *va_arg* çağrıları sonuna kadar yapılmak zorunda değildir.
İşleve çağrı ile gönderilen argüman sayısından daha fazla sayıda argüman elde etmeye çalışılırsa, yani *va_arg* makrosu olması gerekeneden daha fazla çağrılmaya çalışılırsa çöp değerler elde edilir.

va_end Makrosu

Argüman listesi göstericisinin işi bittiğinde *va_end* makrosu çağrılır.

```
void va_end (va_list ap)
```

Bu makro *va_list* türünden değişkenle ilgili son işlemleri yapar.

```
int add_all(int val, ...)
{
    va_list va_p;
    int sum = 0;

    va_start(va_p, val);

    while (val != -1) {
        sum += val;
        val = va_arg(va_p, int);
    }
    va_end(va_p);
    return sum;
}
```

Sistemlerin çoğunda *va_end* makrosu karşılığında bir şey yapılmaz. Yani uygulamalarda bu makro çağırılmasa da yapılan işlemde değişiklik olmaz.
Bu durum *GNU* derleyicisinde her zaman doğrudur.
Yine de okunabilirlik, güvenlik ve taşınabilirlik açılarından bu makronun çağrılması doğru tekniktir.

İstenirse birden fazla argüman listesi göstericisi kullanılabilir. Bu durumda argüman göstericilerinin her birine *va_start* makrosu ile ayrı ayrı ilkdeğerleri verilmelidir. Göstericilerden herhangi biri ile işleve gönderilen argümanlar elde edilebilir.

Değişken Sayıda Argüman Alan İşlevlere Yapılan Çağrılar

Değişken sayıda argüman alan işlevlere yapılan çağrılar derleyici tarafından özel biçimde değerlendirilir. *int* altı türlerden olan argümanlar otomatik olarak *int* türüne yükseltilir. *float* türünden olan argümanlar *double* türüne yükseltilir:

```
#include <stdio.h>

int add_all(int val, ...);

int main()
{
    short sh = 5;

    printf("toplam1 = %d\n", add_all(23, 12, 30, 14, -1));
    printf("toplam2 = %d\n", add_all('A', 10, sh, 20, 30, 40, -1));

    return 0;
}
```

Şimdi kendisine adresleri gönderilen yazıları birleştirecek ve değişken sayıda argümanla çağırılabilecek bir işlev tanımlanıyor:

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

#define      BUFFER_SIZE            1000

char *con_strings(const char *ptr, ...)
{
    va_list ap;
```

```
    char *pd;
    int k;
    char buffer[BUFFER_SIZE] = {'\0'};
    va_start (ap, ptr);

    while (ptr){
        strcat(buffer, ptr);
        ptr = va_arg(ap, const char *);
    }

    pd = (char *)malloc(strlen(buffer) + 1);
    if (pd == NULL) {
        printf("bellek yetersiz!\n");
        exit(EXIT_FAILURE);
    }
    va_end(ap);
    return strcpy(pd, buffer);
}

int main ()
{
    char *pstr = con_strings("C ", "ve ", "Sistem ", "Programcilari ",
"Dernegi", NULL);
    puts(pstr);
    free(pstr);

    return 0;
}
```

con_strings işlevinin tanımını inceleyin:
va_list türünden *ap* değişkeni *va_start* makrosuyla ilkdeğerini alıyor.

```
va_start (ap, ptr);
```

Parametre değişkeni olan *ptr* nin değeri *NULL* adresi olmadığı sürece dönen bir *while* döngüsü oluşturulduğunu görüyorsunuz. Döngünün her turunda işleve gönderilen argümanlardan her birinin değeri *va_arg* makrosuyla elde ediliyor ve bu değer *ptr* göstericisine atanıyor. *ptr* göstericisinin gösterdiği yazı önce yerel *buffer* isimli dizideki yazının sonuna standart *strcat* işleviyle ekleniyor. İşleve çağrı ile gönderilen son argümanın *NULL* adresi olması gerekiyor. *ptr* göstericisinin değeri *NULL* adresi olduğunda *buffer* dizisindeki yazı dinamik bir bellek bloğuna kopyalanıyor. İşlev içinde *va_end* makrosu çağırıldıktan sonra işlev dinamik bloğun adresiyle geri dönüyor.

Kaynaklar

1.American National Standard for Programming Languages - C
ANSI/ISO 9989-1990

2. C - A Reference Manual.
Samuel P. Harbison III - Guy L. Steele Jr.
Prentice-Hall, Inc 2002
ISBN 0-13-089592x

3. The Standard C Library
P. J. Plaguer
Prentice Hall P T R
ISBN 0-13-131509-9

4. C Programming FAQ's
Steve Summit
Addison Wesley