

# 1 . BÖLÜM : PROGRAMLAMA VE C

## Yazılım Nedir

Yazılım (software) programlama ve programlamayla ilgili konuların geneline verilen isimdir. Yazılım denince ilk olarak aklımıza programlama dilleri, bu diller kullanılarak yazılmış kaynak programlar ve çeşitli amaçlar için oluşturulmuş dosyalar gelir.

## Donanım Nedir

Donanım (hardware) : Bilgisayarın elektronik kısmı, yapısına verilen isimdir.

## Yazılımın Sınıflandırılması

Yazılımı uygulama alanlarına göre 5 gruba ayırabiliriz :

1. Bilimsel ve mühendislik yazılımları (scientific & engineering software).  
Bilimsel ve mühendislik konularındaki problemlerin çözülmesinde kullanılan programlardır. Bu tür programlarda veri miktarı görece olarak düşüktür ancak matematiksel ve istatistiksel algoritmalar yoğun olarak kullanılabilir. Tamamen hesaplama ağırlıklı işlemler içerir. Bu tür programlar ağırlıklı olarak bilgisayarın Merkezi İşlem Birimini (CPU) kullanırlar. Elektronik devrelerin çözümünü yapan programları, istatistik analiz paketlerini bu tür programlara örnek olarak verebiliriz.
2. Mesleki yazılımlar (Business software).  
Veri tabanı ağırlıklı yazılımlardır. Genel olarak verilerin yaratılması, işlenmesi ve dosyalarda saklanması ile ilgilidir. Bu tür programlara örnek olarak stok kontrol programları, müşteri takip programları, muhasebe programlarını verebiliriz.
3. Yapay zeka yazılımları (artificial intelligence software).  
İnsan davranışlarını taklit etmeyi amaçlayan 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 animasyon yazılımlarını verebiliriz. Bu yazılımlar ağırlıklı olarak bilgisayarın grafik arabirimini kullanırlar.
5. Sistem yazılımları (system software):  
Bilgisayarın elektronik yapısını yöneten yazılımlardır. Derleyiciler, haberleşme programları, işletim sistemi birer sistem yazılımıdır. Örneğin text editörü de bir sistem yazılımıdır. Uygulama programlarına göre daha düşük seviyeli işlem yaparlar.

## Programlama Dillerinin Sınıflandırılması

Programlama dillerini çeşitli açılardan sınıflandırabiliriz. En sık kullanılan sınıflandırmalar:

1. Seviyelerine göre sınıflandırma.
2. Uygulama alanlarına göre sınıflandırma.

## Bilgisayar Dillerinin Seviyelerine Göre Sınıflandırması ve Seviyelerine Göre Bilgisayar Dillerinin Gelişimi

Bir programlama dilinin seviyesi deyince o programlama dilinin insan algısına olan yakınlığının derecesini anlıyoruz. Bir programlama dili insan algısına ne kadar yakınsa o kadar yüksek seviyeli demektir (high level). Yine bir programlama dili bilgisayarın elektronik yapısına ve çalışma biçimine ne kadar yakınsa o kadar düşük seviyeli (low level) demektir. Yüksek seviyeli dillerle çalışmak programcı açısından kolaydır. Algoritma yoktur. Bu dillerde yalnızca nelerin yapılacağı programa bildirilir ama nasıl yapılacağı bildirilmez. Genel olarak programlama dilinin seviyesi yükseldikçe , o dilin öğrenilmesi ve o dilde program yazılması kolaylaşır.

Bir bilgisayar yalnızca kendi makine dilini doğrudan anlayabilir. Makine dili bilgisayarın doğal dilidir ve bilgisayarın donanımsal tasarımına bağlıdır. Bilgisayarların geliştirilmesiyle birlikte onlara iş yaptırmak için kullanılan ilk diller de makine dilleri olmuştur. Bu yüzden makine dillerine 1. kuşak diller de diyebiliriz.

Makine dilinin programlarda kullanılmasında karşılaşılan iki temel problem vardır. Makine dilinde yazılan kodlar doğrudan makinenin işlemcisine, donanım parçalarına verilen komutlardır. Değişik bir CPU kullanıldığında ya da bellek organizasyonu farklı bir şekilde yapıldığında artık program çalışmayacak ve programın tekrar yazılması gerekecektir. Çünkü makine dili yalnızca belirli bir CPU ya da CPU serisine uygulanabilir. Makine dili taşınabilir (portable) değildir. Diğer önemli bir problem 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 dili programlarını geliştirmek, daha büyütme 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 içerisinde makine dilinden, insan algılamasına çok yakın yüksek seviyeli dillere (4. kuşak diller) kadar uzanan bir süreç söz konusudur. Bu tarihsel süreci ana hatlarıyla inceleyelim :

1950 li yılların hemen başlarında makine dili kullanımının getirdiği problemleri ortadan kaldırmaya yönelik çalışmalar yoğunlaştı. Bu yıllarda makine dilleri bilgisayarın çok sınırlı olan belleğine yükleniyor ve programlar böyle çalıştırılıyordu. İlk önce makine dilinin algılanma ve anlaşılma zorluğunu kısmen de olsa ortadan kaldıran bir adım atıldı. Sembolik makine dilleri geliştirildi. Sembolik makine dilleri (Assembly languages) yalnızca 1 ve 0 dan oluşan makine dilleri yerine İngilizce bazı kısaltma sözcüklerden oluşuyordu. Sembolik makine dillerinin kullanımı kısa sürede yaygınlaştı. Ancak sembolik 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 sembolik dilin komutları, bilgisayar tarafından komut komut makine diline çevriliyor ve oluşan makine kodu çalıştırılıyordu. Yani bilgisayar, programı çalışma aşamasında önce yorumluyarak makine diline çeviriyor daha sonra makine diline çevrilmiş komutları icra ediyordu. Bu şekilde çalıştırılan programların hızı 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 yazılanların gerçek program deneceğini söylüyorlardı.

Bu sorunun da üstesinden gelindi. O zamanlar için çok parlak kabul edilebilecek fikir şuydu: Her defasında yazılan kod, kodun çalıştırılması sırasında makine diline çevireceğine, geliştirilecek bir başka program sembolik dilinde yazılan kodu bir kez makine diline çevirsin ve artık program ne zaman çalıştırılmak istense, bilgisayar yorumlama olmaksızın yalnızca makine kodunu çalıştırsın. Bu fikri geliştiren Grace Hopper isimli bir bayandı. 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 sembolik sözcüklerden oluşan Assembly programlarını kullanıyor. Yazdıkları programlar derleyici tarafından makine koduna dönüştürülüyor ve makine kodu eski hızından birşey kaybetmeksizin 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 bir çok komut gerekmesi, programlama prosesini daha hızlı bir hale getirmek için arayışları başlatmış, bunun sonucunda da daha yüksek seviyeli programlama dilleri geliştirilmeye başlanmıştır.

Tarihsel süreç içinde Assembly dillerinden daha sonra geliştirilmiş ve daha yüksek seviyeli diller 3. kuşak diller sayılmaktadır. Bu dillerin hepsi algoritmik dillerdir. Bugüne kadar geliştirilmiş olan yüzlerce yüksek seviyeli programlama dilinden yalnızca pek azı bugüne kadar varlıklarını sürdürebilmiştir:

**FORTRAN** dili (FORmula TRANslator) kompleks matematiksel hesaplamalar gerektiren mühendislik ve bilimsel uygulamalarda kullanılmak üzere 1954 - 1957 yılları arasında IBM firması için John Backus tarafından geliştirilmiştir. FORTRAN dili, yoğun matematik hesaplamaların gerektiği bilimsel uygulamalarda halen yaygın olarak kullanılmaktadır. FORTRAN dilinin FORTRAN IV ve FORTRAN 77 olmak üzere iki önemli versiyonu bulunmaktadır. Doksanlı yılların başlarında FORTRAN - 90 isimli bir versiyon için ISO ve ANSI standartları kabul edilmiştir. FORTRAN dili 3. seviye dillerin en eskisi kabul edilmektedir.

**COBOL** (COmmon Business Oriented Language) 1959 yılında, Amerika'daki bilgisayar üreticileri, özel sektör ve devlet sektöründeki bilgisayar kullanıcılarından oluşan bir grup tarafından geliştirilmiştir. COBOL'un geliştirilme amacı veri yönetimi ve işlemenin gerektiği ticari uygulamalarda kullanılacak taşınabilir bir programlama dili kullanmaktır. COBOL dili de halen yaygın olarak kullanılmaktadır.

**ALGOL** (The ALGOritmick Language) 1958 yılında Avrupa'da bir konsorsiyum tarafından geliştirilmeye başlanmıştır. IBM Firması FORTRAN dilini kendi donanımlarında kullanılacak ortak programlama dili olarak benimsediğinden, Avrupa'lılar da alternatif bir dil geliştirmek istemişlerdi. ALGOL dilinde geliştirilen bir çok prensip modern programlama dillerinin hepsinde kullanılmaktadır.

60'lı yılların başlarında programlama dilleri üzerinde yapılan çalışmalar yapısal programlama kavramını gündeme getirmiştir. Bu dillerden bazılarını kısaca göz atalım:

**PASCAL** dili 1971 yılında akademik çevrelere yapısal programlama kavramını tanıtmak için Profesör Niclaus Wirth tarafından geliştirilmiş (Dilin yaratıcısı, dile matematikçi ve filozof Blaise Pascal'ın ismini vermiştir.) ve bu dil kısa zaman içinde üniversitelerde kullanılan programlama dili haline gelmiştir.

Pascal dilinin ticari ve endüstriyel uygulamaları desteklemek için sahip olması gereken bir takım özelliklerden yoksun olması bu dilin bu uygulamalarda fazla kullanılmamasına yol açmıştır. Modula ve Modula-2 dilleri Pascal dili baz alınarak geliştirilmiştir.

**BASIC** dili 1960'lı yılların ortalarında John Kemeney ve Thomas Kurtz tarafından geliştirilmiştir. Her ne kadar BASIC isminin "Beginner's All-purpose Symbolic Instruction Code" sözcüklerinin baş harflerinden oluşturulduğu söylene de, bu sözcüklerin daha sonradan uydurulduğu açıktır. Yüksek seviyeli dillerin en eski ve en basit olanlarından biridir. Tüm basitliğine karşın, bir çok ticari uygulamada kullanılmıştır. BASIC dili de ANSI tarafından standartlaştırılmıştır. Ancak BASIC dilinin ilave özellikler içeren bir sürü versiyonu söz konusudur. Örneğin Microsoft firmasının çıkarttığı Visual Basic diline Nesne Yönelimli Programlamaya ilişkin birçok özellik eklenmiştir. Ayrıca BASIC dilinin bazı versiyonları uygulama programlarında (Örneğin MS Excel ve MS Word'de) kullanıcının özelleştirme ve otomatikleştirme amacıyla yazacağı makroların yazılmasında kullanılan programlama dili olarak da genel kabul görmüştür.

**ADA** dili ise Amerikan Savunma Departmanı (Department of Defence -DoD) desteği ile 70 li yıllar ve 80'li yılların başlarında geliştirilmiştir. Dod dünyadaki en büyük bilgisayar kullanıcılarından biridir. Bu kurum farklı yazılımsal gereksinimleri karşılamak için çok sayıda farklı programlama dili kullanıyordu ve tüm gereksinimlerini karşılayacak bir dil arayışına girdi. Dilin tasarlanması amacıyla uluslararası bir yarışma düzenledi. Yarışmayı kazanan şirket (CII-Honeywell Bull of France) Pascal dilini baz alarak alan çalışmalar sonucunda Ada dilini geliştirdi. Ada dilinin dökümanları 1983 yılında yayımlanmıştır. (Ada ismi, şair Lord Byron'un kızı olan Lady Ada Lovelace'ın isminden alıntıdır. Ada Lovelace delikli kartları hesap makinalarında ilk olarak kullanılan Charles Babbage'in yardımcısıydı. Charles Babbage hayatı boyunca "Fark makinası" (Difference Engine) ve "Analitik Makine" (Analytical Engine) isimli makinaların yapımı üzerinde çalıştı ama bu projelerini gerçekleştiremeden öldü. Yine de geliştirdiği tasarımlar modern bilgisayarların atası kabul edilmektedir. Ada Lovelace Charles Babbage'ın makinası için

delikli kartları ve kullanılacak algoritmaları hazırlıyordu. Bu bayanın 1800'lü yılların başında ilk bilgisayar programını yazdığı kabul edilmektedir.) Ada dili genel amaçlı bir dildir, ticari uygulamalardan roketlerin yönlendirilmesine kadar birçok farklı alanda kullanılmaktadır. Dilin önemli özelliklerinden bir tanesi gerçek zaman uygulamalarına (real-time applications / embedded systems) destek vermesidir. Başka bir özelliği de yüksek modüleritesi nedeniyle büyük programların yazımını kolaylaştırmasıdır. Ancak büyük ve karmaşık derleyicilere ihtiyaç 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 olarak kısaltılırlar. (fourth generation language). İnsan algısına en yakın dillerdir. **RPG** dili 4. kuşak dillerin ilki olarak kabul edilebilir. Özellikle küçük IBM makinalarının kullanıcıları olan şirketlerin, rapor üretimi için basit bir dil istemeleri üzerine IBM firması tarafından geliştirilmiştir.

Programlama dillerini seviyelerine göre 5 ana gruba ayırabiliriz:

1. Çok yüksek seviyeli diller ya da görsel diller (visual languages):  
Access, Foxpro, Paradox, Xbase, Visual Basic, Oracle Forms.
2. Yüksek seviyeli diller (Bunlara algoritmik diller de denir):  
Fortran, Pascal, Basic, Cobol.
3. Orta seviyeli programlama dilleri:  
Ada, C. Orta seviyeli diller daha az kayıpla makine diline çevrilebildiğinden daha hızlı çalışır.
4. Alçak seviyeli programlama dilleri:  
Sembolik makine dili (Assembly language).
5. Makine dili:  
En aşağı seviyeli programlama dili. (Saf makine dili tamamen 1 ve 0 lardan oluşuyor.)

### Uygulama Alanlarına Göre Sınıflandırma

1. Bilimsel ve mühendislik uygulama dilleri:  
Pascal, C (C programlama dili üniversitelerdeki akademik çalışmalarda da yoğun olarak kullanılıyor.), FORTRAN
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. Simulasyon dilleri  
GPSS, Simula 67
6. Makro Dilleri (Scripting languages)  
awk, Perl, Python, Tcl, JavaScript.
7. Sistem programlama dilleri:  
Sembolik 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 sembolik makine dili ile yazılmıştır. Bu işletim sistemi ilk olarak BELL laboratuvarlarında 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.

## 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 daha farklı kılan özellikler neler olabilir? Bir programlama dilini tanımlamak istesek hangi özellikleri kullanabiliriz? Programlama dilleri hakkındaki bu sorulara yanıt verebilmemiz için elimizde değerlendirme yapmamıza olanak sağlayacak ölçütler olmalıdır. Bu ölçütleri kısaca inceleyelim:

### Verimlilik (efficiency)

Bu özelliğe programın hızlı çalışma özelliği diyebiliriz. Programın çalışma hızı pek çok faktöre bağlıdır. Algoritmanın da hız üzerinde etkisi vardır. Çalışmanın yapıldığı bilgisayarın da doğal olarak hız üzerinde etkisi vardır. Verimliliği bir programlama dilinde yazılmış bir programın hızlı çalışması ile ilgili bir kavram olarak düşünebiliriz. Bu açıdan bakıldığında C verimli bir dildir.

### Veri türleri ve yapıları (data types and structures)

Çeşitli veri türlerini (tamsayı, gerçek sayı, karakter...) ve veri yapılarını (diziler, yapılar vs.) destekleme yeteneğidir. Veri yapıları, veri türlerinin oluşturduğu mantıksal birliklerdir. Örneğin C ve Pascal dilleri veri yapıları bakımından zengin dillerdir.

### Alt programlama yeteneği (Modularity)

Bir bütün olarak çözülmesi zor olan problemlerin parçalara ayrılması ve bu parçaların ayrı ayrı çözümlenmesinden sonra parçalar arasındaki koordinasyonun sağlanması programada sık başvurulan bir yöntemdir. Bir programlama dili buna olanak sağlayacak araçlara sahipse alt programlama yeteneği vardır diyebiliriz. Alt programlama yeteneği bir programlama dilinin, programı parçalar halinde yazmayı desteklemesi anlamına gelir. (C modüleritesi çok yüksek bir dildir)

Alt programlama Yapısal Programlama tekniği'nin de ayrılmaz bir parçasıdır. Alt programlamanın getirdiği bazı önemli avantajlar vardır. Alt programlar kodu küçültür. Çok tekrarlanan işlemlerin alt programlar kullanılarak yazılması çalışabilir programın kodunu küçültür. Çünkü alt programlar yalnızca bir kere çalışabilir kod içine yazılırlar. Ama program kodu alt programın olduğu yere atlatılarak bu bölgenin defalarca çalıştırılması sağlanabilir.

Alt programlama algılamayı kolaylaştırır, okunabilirliği artırır. Alt programlama kaynak kodun test edilebilirliğini artırır. Kaynak kodun daha kolay güncelleştirilmesi ve yeniden kullanılabilme olanağını artırır. Alt programlamanın en önemli avantajlarından biri de genel amaçlı kodlar yazarak bu yazılan kodları birden fazla projede kullanabilmektir. (reusability)

C alt programlama yeteneği yüksek bir dildir. C'de alt programlara fonksiyon denir. Fonksiyonlar C Dili'nin yapıtaşlarıdır.

### Yapısallık (structural programming support)

Yapısallık bir programlama tekniğidir. Bugün artık hemen hemen bütün programlama dilleri yapısal programlamayı az çok destekleyecek bir şekilde geliştirilmiştir. Yapısal Programlama fikri 1960'lı yıllarda geliştirilmiştir. Yapısal programlama tekniği dört ana ilke üzerine kurulmuştur :

1. Böl ve üstesinden gel (divide and conquer)  
Yapısal programlama tekniğinde, tek bir bütün olarak çözüm getirmek zor olan programlar, daha küçük ve üstesinden daha kolay gelinebilecek parçalara bölünürler. Bu parçalar fonksiyon, prosedür, subroutine, alt program vs. olarak isimlendirilir. Alt program yapısının getirdiği avantajlar modularite konusunda yukarıda açıklanmıştır.
2. Veri gizleme (Data hiding)

Yapısal programlama tekniğinde, programın diğer parçalarından ulaşılamayan, yalnızca belli bir faaliyet alanı olan, yani kodun yalnızca belli bir kısmında faaliyet gösterecek değişkenler tanımlanabilir. Bu tür değişkenler genel olarak "yerel değişkenler" (local variables) olarak isimlendirilirler. Değişkenlerin faaliyet alanlarının kısıtlanabilmesi hata yapma riskini azalttığı gibi, programların daha kolay değiştirilebilmesini ve program parçalarının başka programlarda tekrar kullanılabilmesini de sağlar. Alt programların, ve daha geniş şekliyle modüllerin, bir işi nasıl yaptığı bilgisi, o alt programın ya da modülün kullanıcılarından gizlenir. Kullanıcı için (client) 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 mekanizma programın yukarıdan aşağı olarak akışı ile uyum halindedir. Program parçalarına ancak tek bir noktadan girilebilir.

### 4. Döngüler ve diğer kontrol yapıları.

Artık hemen hemen kullanımda olan bütün programlama dilleri az ya da çok Yapısal Programlama tekniğini desteklemektedir. Zira bu teknik 60'lı yıllar için devrim niteliğindeydi.

## Esneklik (flexibility)

Esneklik programlama dilinin programcıyı kısıtlamaması anlamına gelir. Esnek dillerde birçok işlem, hata yapma riski artmasına karşın rağmen kullanıcı için serbest bırakılmıştır. Programcı bu serbestlikten ancak yetkin bir programcıysa bir fayda sağlayabilir. Fakat programcı deneyimsiz ise bu esneklikten zarar görebilir.

## Öğrenme ve öğretme kolaylığı (pedagogy)

Her programlama dilini öğrenmenin ve öğrenilen programlama dilinde uygulama geliştirebilmenin zorluğu aynı değildir. Genel olarak programlama dillerinin seviyesi yükseldikçe, öğrenme ve bu programlama dilini başkalarına öğretme kolaylaşır, öğrenme için harcanacak çaba ve zaman azalır. Bugün yaygın olarak kullanılan yüksek seviyeli programlı dillerinin bu derece popüler olmasının önemli bir nedeni de bu dillerin çok kolay öğrenilebilmesidir. Ne yazık ki C öğrenimi zor ve zahmetli bir dildir.

## Genellik (generality)

Programlama dillerinin çok çeşitli uygulamalarda etkin olarak kullanılabilmesidir. Örneğin COBOL mühendislik uygulamalarında tercih edilmez zaten ticari uygulamalar için tasarlanmıştır, Clipper ya da FOXPRO veri tabanı dilleridir. Oysa PASCAL, BASIC daha genel amaçlı dillerdir. C dili de bir sistem programlama dili olarak doğmasına karşın, güçlü yapısından dolayı, kısa bir süre içinde, genel amaçlı bir dil haline gelmiştir.

## Giriş / Çıkış (input / output, I / O facility) kolaylığı

Sıralı, indeksli ve rasgele dosyalara erişme, veritabanı kayıtlarını geri alma, güncelleştirme ve sorgulama yeteneğidir. Veritabanı programlama dillerinin (DBASE, PARADOX vs.) bu yetenekleri diğerlerinden daha üstündür ve bu dillerin en tipik özelliklerini oluşturur. Fakat C giriş çıkış kolaylığı kuvvetli olmayan bir dildir. C'de veri tabanlarının yönetimi için özel kütüphanelerin kullanılması gerekir.

## Okunabilirlik (readability)

Okunabilirlik, kaynak kodun çabuk ve iyi bir biçimde algılanabilmesi anlamına gelen bir terimdir. Kaynak kodun okunabilirliğinde sorumluluk büyük ölçüde programı yazan kişidedir. Fakat yine verimlilik de olduğu gibi dillerin bir kısmında okunabilirliği güçlendiren yapı ve mekanizmalar bulunduğu için bu özellik bir ölçüde dilin tasarımına da bağlıdır. En iyi program kodu, sanıldığı gibi "en zekice yazılmış fakat kimsenin anlayamayacağı" kod değildir.

Birçok durumda iyi programcılar okunabilirliği hiçbirşeye feda etmek istemezler. Çünkü okunabilir bir program kolay algılanabilme özelliğinden dolayı seneler sonra bile

güncelleştirmeye olanak sağlar. Birçok kişinin ortak kodlar üzerinde çalıştığı geniş kapsamlı projelerde okunabilirlik daha da önem kazanmaktadır. C de okunabilirlik en fazla vurgulanan kavramlardan biridir. Biz de kursumuz boyunca okunabilirlik konusuna sık sık değineceğiz ve C programlarının okunabilirliği konusunda bazı temel prensipleri benimseyeceğiz.

### **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. 1989 yılında standartlaştırma çalışmaları biten C Dili, diğer programlama dillerinden daha taşınabilir bir programlama dilidir.

### **Nesne Yönelimlilik (object orientation)**

Nesne yönelimlilik de bir programlama tekniğidir.

Yapısal programlama Tekniği 1960 yıllarında gündeme gelmişken, Nesne Yönelimli Programlama Tekniği 1980'li yıllarda popüler olmuştur.

Bu teknik kaynak kodların çok büyümesi sonucunda ortaya çıkan gereksinim yüzünden geliştirilmiştir. C dilinin geliştirildiği yıllarda, akla gelebilecek en büyük programlar ancak onbin satırlar mertebesindeydi, ancak kullanıcıların bilgisayar programlarından beklentilerinin artması ve grafik arayüzünün artık etkin olarak kullanılmasıyla, bilgisayar programlarının boyutu çok büyümüş, yüzbin satırlarla hatta milyon satırlarla ölçülebilir hale gelmiştir.

Nesne yönelimli programlama Tekniği, herşeyden önce büyük programların yazılması için tasarlanmış bir tekniktir. C dilinin yaratıldığı yıllarda böyle bir tekniğin ortaya çıkması söz konusu değildi, çünkü zaten programlar bugünkü ölçülere göre çok küçüktü.

Nesne yönelimli programlama Tekniğinin yaygın olarak kullanılmaya başlanmasıyla birlikte bir çok programlama dilinin bünyesine bu tekniğin uygulanmasını kolaylaştırıcı araçlar eklenek, yeni versiyonları oluşturulmuştur. Örneğin C'nin nesne yönelimli programlama tekniğini uygulayabilmek için Bjarne Stroustrup tarafından geliştirilmiş haline C++ denmektedir. C++ dili C dili baz olarak alınıp, geliştirilmiş yeni bir programlama dilidir. C++ dilini iyi öğrenebilmek için öncelikle C dilini çok iyi öğrenmek gerekir.

Pascal diline eklemeler yapılarak Delphi dili, Cobol dilinden yenilemesiyle OOCobol, Ada dilinin yenilenmesiyle ise ADA 95 dilleri geliştirilmiştir.

Bazı programlama dilleri ise doğrudan N.Y.P.T'ni destekleyecek şekilde tasarlanarak geliştirilmiştir. Örneğin JAVA dili C++ dilinin basitleştirilmiş biçimi olup daha çok Internet uygulamalarında kullanılmaktadır. Başka bir örnek olarak da Eiffel dili verilebilir.

### **C Nasıl bir Programlama Dilidir?**

Bütün bunlardan sonra yukarıda açıkladığımız kavramları da kullanarak C dilini aşağıdaki şekilde tanımlayabiliriz :

C orta seviyeli bir programlama dilidir. Yapısal diğer programlama dillerine göre C dilinin seviyesi daha düşüktür. C dili hem yüksek seviyeli dillerin, kontrol deyimleri, veri yapıları gibi avantajlarını bünyesinde barındırıyor, aynı zamanda bitisel operatörler gibi makine kodu deyimlerini yansıtan operatörlerle sahip. Yani hem makinaya yakın hem de insan algılamasına. Zaten çok tercih edilmesinin en önemli nedenlerinden biri de bu.

C bir sistem programlama dilidir. Sistem Programlama ne anlama geliyor? Donanımın yönetilmesi, kontrolü ve denetimi için yazılan, doğrudan donanımla ilişkiye giren programlara sistem programı diyoruz. Örneğin, işletim sistemleri, derleyiciler, yorumlayıcılar, aygıt sürücüler (device drivers), bilgisayarların iletişimine ilişkin programlar, otomasyon programları, sistem programlarıdır. Diğer uygulama programlarına destek veren yazılımlar da çoğunlukla sistem programları olarak ele alınırlar.

C'den önce sistem programları assembly dillerle yazılıyordu. Sistem programlarının yazılmasında hemen hemen alternatifsiz olduğunu söyleyebiliriz. Bugün cep telefonlarından, uçaklara kadar her yerde C kodları çalışmaktadır. Örneğin Boeing uçaklarında 100.000 satırdan fazla C kodu çalıştığı bilinmektedir.

C algoritmik bir dildir. C'de program yazmak için yalnızca dilin sentaks ve sementik 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. İfade gücü yüksek , okunabilirlik özelliği güçlü bir dildir.

C çok esnektir. Diğer dillerde olduğu gibi programcıya kısıtlamalar getirmez.

Güçlü bir dildir. Çok iyi bir biçimde tasarlanmıştır. C'ye ilişkin operatörlerin ve yapıların bir çoğu daha sonra başka programlama dilleri tarafından da benimsenmiştir.

C verimli bir dildir. Seviyesinden dolayı hızlı çalışır. Verimlilik konusunda assembly diller ile rekabet edebilir.

C doğal bir dildir. C bilgisayar sisteminin biçimiyle uyum içindedir.

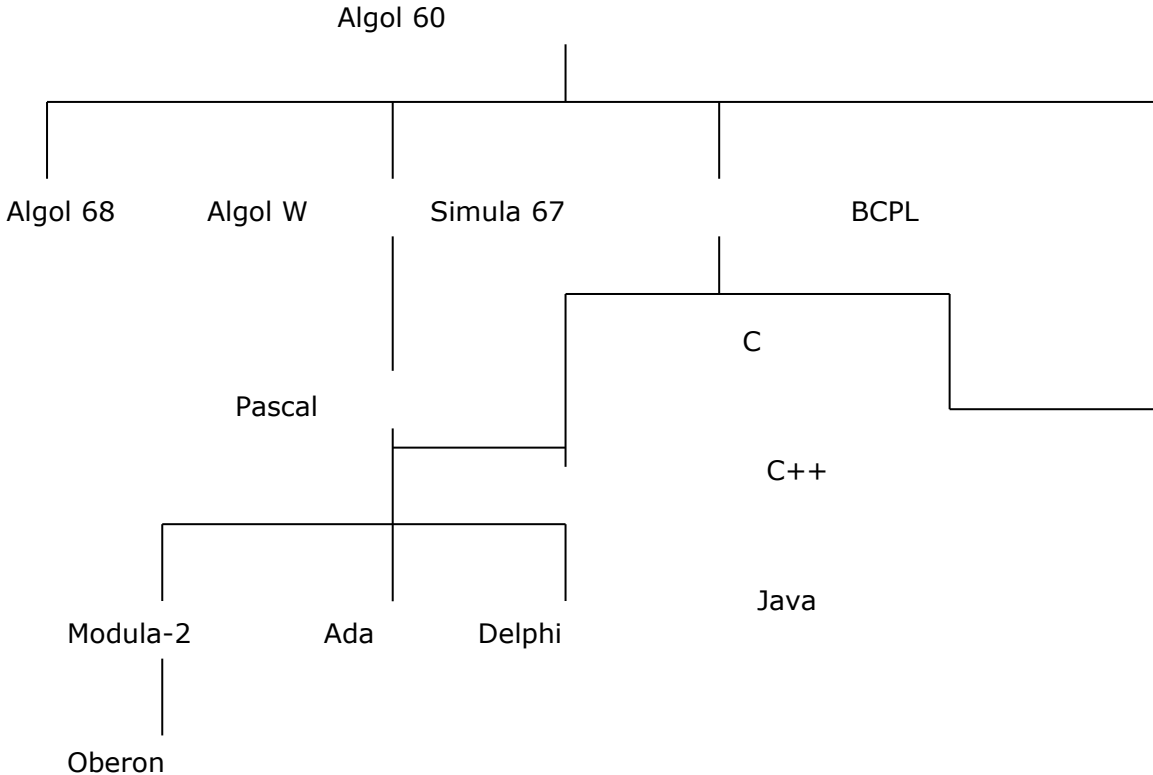
C küçük bir dildir. Yeni sistemler için derleyici yazmak zor değildir.

C'nin eğitimi diğer bilgisayar dillerine göre daha zordur.

## C Programlama Dili'nin Tarihi

C dilinin tarihini incelediğimizde C dilinin UNIX işletim sisteminin bir yan ürünü olarak doğduğunu söyleyebiliriz. UNIX işletim sisteminin orjinal ilk versiyonunu Bell Labaratuarları'nda çalışan Ken Thompson tek başına yazmıştı ve UNIX'in bu ilk versiyonu DEC PDP-7 isimli bilgisayarda çalışıyordu. DEC PDP-7 ilk mini bilgisayarlardan biriydi ve ana belleği yalnızca 16 K (16 MB değil!). Yıllardan 1969'du.

Zamanının diğer işletim sistemleri gibi UNIX de assembly dilinde yazılmıştı. Assembly dilinde yazılan programları geliştirmek çok zor ve zahmetli olduğundan, Thompson UNIX işletim sistemini daha geliştirebilmek için, makine dilinden daha yüksek seviyeli bir dile gereksinim duydu. Bu amaçla küçük bir programlama dili tasarladı. Kendi dilini tasarlariken 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 görülmektedir:





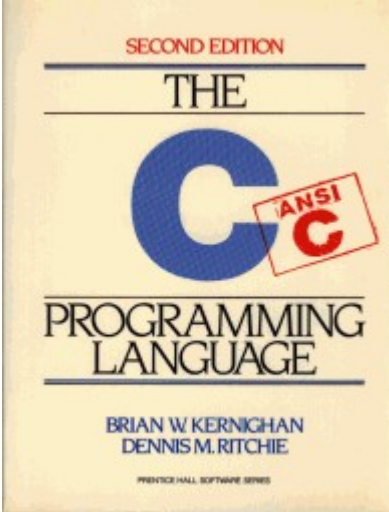
Thompson geliştirdiği bu dilin ismini B koydu. Dennis Ritchie UNIX projesine katılınca B dilinde programlamaya başladı. B dili daha da geliştirilmişti ve artık daha yeni teknoloji olan PDP-11 bilgisayarlar çalışıyordu. Thompson UNIX işletim sisteminin bir kısmını B dilinde tekrar yazdı. Artık 1971 yılına 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 versiyonunu 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 tekrar yazıldı.



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

C'nin evrimi ve gelişmesi 70'li yıllarda da devam etti. 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'nin standardize

edilmesine kadar olan dönemde bu kitap çoğunluğun benimsediği genel kabul gören gayriresmi bir standard vazifesi de görmüştür.



1970'li yıllarda C programcılarının sayısı azdı ve bunlardan çoğu UNIX kullanıcılarıydı. Ama artık 80'li yıllar gelince C'nin kullanımı UNIX sınırlarını aştı, ve farklı işletim sistemlerinde çalışan derleyiciler piyasaya çıktı, C dili de IBM PC'lerde yoğun olarak kullanılmaya başladı.

C'nin artan popülaritesi problemleri de beraberinde getirdi. Derleyici yazan kişiler, referans olarak Ritchie ve Kernighan'ın kitabını esas alıyorlardı ama söz konusu kitapta bazı noktalar çok da detaylı bir biçime 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 yazarların 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.

Artık C dilinin standardizasyonu neredeyse zorunlu bir hale gelmişti! C'nin standardizasyon çalışmaları 1983 yılında ANSI (American National Standards Institute) gözetiminde ve desteğinde başladı. Yapılan birçok değişiklikten sonra standart çalışmaları 1988 yılında sona erdi ve 1989 yılının Aralık ayında ANSI C standardı Jim Brodie başkanlığında X3.159 - 1989 numarasıyla resmi olarak onaylandı. 1990 yılında ise ISO/IEC 9899 - 1990 numarasıyla ISO (International Standards Organization) tarafından standardizasyonu kabul edildi. Standardizasyonu tamamlandıktan sonra C yüksek oranda taşınabilir bir sistem programlama dili haline gelmiştir. Günümüzde de sistem programlarının (derleyiciler, editörler, işletim sistemleri) çoğu C dili ile yazılmaktadır.



**Fotoğraflar**

**Dennis M. Ritchie**



## 2 . BÖLÜM : SAYI SİSTEMLERİ

Günlük hayatta 10'luk sayı sistemini kullanıyoruz. 10 luk sistemde bir sayının değeri aslında her bir basamak değerinin 10 sayısının ilgili kuvvetiyle çarpımlarının toplanmasıyla elde edilir.

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

Ancak bilgisayar sistemlerinde bütün bilgiler ikilik sistemde(binary system) ifade edilir.

Genel olarak sayı sistemi kaçlıksa o sayı sisteminde o kadar sembol bulunur.

Örneğin 10'luk sistemde 10 adet sembol vardır ve bu semboller 0, 1, 2, 3, 4, 5, 6, 7, 8, 9'dur.

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

Bir sayıyı başka bir sayı sisteminde ifade etmek o sayının değerini değiştirmez. Yalnızca sayının gösteriliş biçimi değişir. Örneğin onluk sayı sisteminde sayısal değeri 32 olan büyüklüğü çeşitli farklı sayı sistemlerinde farklı biçimlerde gösterebiliriz ama sayının büyüklüğünü değiştirmiş olmayız.

İkilik sistemde her bir basamağa 1 bit denir. Bit kelimesi **binary digit** sözcüklerinden türetilmiştir.

Örneğin 1011 sayısı 4 bittir. (Ya da 4 bit uzunluğundadır).

11011001 sayısı 8 bittir.

8 bitlik bir büyüklük bir byte olarak isimlendirilir.

1 kilobyte 1K = 1024 byte dır. (yani  $2^{10}$  byte)

1 mega byte 1 MB = 1024 Kilo byte dır. (yani  $2^{20}$  byte)

1 gigabyte 1 GB = 1024 MB dır. (yani  $2^{30}$  byte)

1 terabyte 1 TB = 1024 GB dır. (yani  $2^{40}$  byte)

1 petabyte 1PB = 1024 TB dır. (yani  $2^{50}$  byte)

1 exabyte 1EB = 1024 PB dır. (yani  $2^{60}$  byte)

1 zettabyte 1ZB = 1024 EB dır. ( yani  $2^{70}$  byte)

1 yottabyte 1YB = 1024 ZB dır.( yani  $2^{80}$  byte)

Kilo büyüklük olarak 1000 kat anlamına gelmektedir, ancak bilgisayar alanında Kilo 2'nin 1000'e en yakın kuvveti olan  $2^{10}$  yani 1024 kat olarak kullanılır.

4 bit 1 Nybble (Nibble şeklinde de yazılır)

8 bit 1 byte

16 bit 1 word

32 bit 1 double word

64 bit 1 quadro word

olarak da isimlendirilmektedir.

### ikilik sisteme ilişkin genel işlemler

i. İkilik sistemdeki bir sayının 10 luk sistemde ifade edilmesi:

ikilik sayı sisteminde ifade edilen bir sayının 10'luk sistemdeki karşılığını hesaplamak için en sağdan başlayarak bütün basamakları tek tek 2'nin artan kuvvetleriyle çarpılır. Örneğin :

$$1011 = 1 * 2^0 + 1 * 2^1 + 0 * 2^2 + 1 * 2^3 = 11$$

$$00101001 = (1 * 1) + (1 * 8) + (1 * 32) = 41$$

Bu arada sık kullanılan iki terimi de açıklayalım. İkilik sayı sisteminde yazılan bir sayının en solundaki bit, yukarıdaki örnekten de görüldüğü gibi en yüksek sayısal değeri katıyor. Bu bite en yüksek anlamlı bit (most significant digit) diyeceğiz ve bu bit için bundan sonra MSD kısaltmasını kullanacağız.

İkilik sayı sisteminde yazılan bir sayının en sağındaki bit, yine yukarıdaki örnekten de görüldüğü gibi en düşük sayısal değeri katıyor. Bu bite en düşük anlamlı bit (least significant digit) diyeceğiz ve bu bit için bundan sonra LSD kısaltmasını kullanacağız.

Örnek :

01011101 sayısı için

MSD = 0

LSD = 1

İkilik sayı sisteminde yazılan bir sayının belirli bir basamağından (bitinden) söz ettiğimizde, 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.

ii. 10'luk sistemdeki bir sayının 2'lik sistemde ifadesi :

Sayı sürekli olarak 2 ye bölünür. Her bölümden kalan değer( yani 1 ya da 0) oluşturulacak sayının 0. bitinden başlayarak, basamaklarını oluşturacaktır. Bu işleme 0 sayısı elde edilinceye kadar devam edilir. Örnek:

87 sayısını ikilik sayı sisteminde ifade etmek isteyelim:

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 10 luk sistemde ifade edilen sayıdan sürekli olarak 2'nin en büyük kuvvetini çıkarmaktır. 2'nin çıkarılan her bir kuvveti için ilgili basamağa 1 değeri yazılır. Bu işleme 0 sayısı elde edilene kadar devam edilir. Örnek:

Yine 87 sayısını ikilik sayı sisteminde ifade etmek isteyelim:

87'den çıkarılabilecek, yani 87'den büyük olmayan ikinin en büyük kuvveti nedir? Cevap 64. O zaman  $64 = 2^6$  olduğuna göre 6.bit 1 değerini alacak.

87 - 64 = 23. Şimdi 23'den çıkarılabilecek ikinin en büyük kuvvetini bulalım. Bu sayı 16'dır. Yani  $2^4$ 'dür. O zaman sayımızın 4. biti de 1 olacak.

23 - 16 = 7. 7'den çıkarılabilecek ikinin en büyük kuvveti 4'dür ve  $4 = 2^2$ 'dir. Sayımızın 2. biti de 1 olacak.

7 - 4 = 3.

3 - 2 = 1 ( $2 = 2^1$ ) Sayımızın 1. biti 1 olacak.

1 - 1 = 0 ( $1 = 2^0$ ) Sayımızın 0. biti 1 olacak.

1 değeri olmayan tüm bitleri 0 bitiyle doldurarak sayımızı ikilik sistemde ifade edelim:

87 = 0101 0111

iii. İkilik sistemde ifade edilen bir sayının 1'e tümleyeni. Sayının tüm bitlerinin tersinin alınmasıyla elde edilir. Yani sayıdaki 1'ler 0 ve 0'lar 1 yapılır.

Bir sayının 1'e tümleyeninin 1'e tümleyeni sayının yine kendisidir.

iv. İkilik sistemde ifade edilen bir sayının 2'ye tümleyeninin bulunması:

Önce sayının 1'e tümleyeni yukarıdaki gibi bulunur. Daha sonra elde edilen sayıya 1 eklenirse sayının 2'ye tümleyeni bulunmuş olur. 2'ye tümleyeni bulmak için daha daha pratik bir yol daha vardır : Sayının en solundan başlayarak ilk defa 1 biti görene kadar (ilk görülen 1 dahil) sayının aynısı yazılır, daha sonraki tüm basamaklar için basamağın tersi yazılır. (Yani 1 için 0 ve 0 için 1) Örneğin :

1110 0100 sayısının ikiye tümleyeni 0001 1100 dir.

0101 1000 sayısının ikiye tümleyeni 1010 1000 dir.

Bir sayının ikiye tümleyeninin ikiye tümleyeni sayının kendisidir. (Deneyiniz)

8 bitlik bir alana yazılacak en büyük tam sayı kaçtır?

1111 1111 = 255 dir. ( $1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255$ )

8 bitlik bir alana yazılacak en küçük tam sayı kaçtır?

0000 0000 = 0'dır.

**Negatif bir tamsayı ikilik sistemde nasıl gösterilir?**

Negatif tamsayıların da ifade edildiği ikilik sayı sistemine işaretli ikilik sayı sistemi (**signed binary system**) denir. İşaretli ikilik sayı sisteminde, negatif sayıları göstermek için hemen hemen tüm bilgisayar sistemlerinde aşağıdaki yol izlenir:

Sayının en yüksek anlamlı biti işaret biti (**sign bit**) olarak kabul edilir. Ve bu bit **1** ise sayı negatif, bu bit **0** ise sayı pozitif olarak değerlendirilir. İkilik sistemde bir negatif sayı aynı değerdeki pozitif sayının ikiye tümleyenidir. Örnek olarak, ikilik sistemde yazacağımız -27 sayısı yine ikilik sistemde yazılan 27 sayısının ikiye tümleyenidir.

Pozitif olan sayıların değerini tıpkı işaretsiz sayı sisteminde olduğu gibi elde ederiz:

0001 1110 işaretli sistemde pozitif bir sayıdır. (Decimal olarak 29 sayısına eşittir.)

Ancak negatif olan sayıların değerini ancak bir dönüşümle elde edebiliriz:

1001 1101 işaretli sistemde negatif bir sayıdır. (Çünkü işaret biti 1)

**2lik sistemde ifade edilen negatif bir sayının 10'luk sistemde hangi negatif sayıya eşit olduğunu nasıl bulunur?**

Sayının en yüksek anlamlı biti (MSD) işaret bitidir. Bu bit 1 ise sayı negatiftir. Sayının kaç'a eşit olduğunu hesaplamak için ilk önce sayının 2'ye tümleyeni bulunur. Ve bu sayının hangi pozitif sayıya karşılık geldiğini hesap edilir. Elde etmek istenen sayı, bulunan pozitif sayı ile aynı değerdeki negatif sayı olacaktır.

Örneğin 1001 1101 sayısının 10'luk sistemde hangi sayıya karşılık geldiği bulunmak istenirse:

Sayının en soldaki biti 1 olduğuna göre bu sayı negatif bir sayı olacaktır. Hangi negatif sayı olduğunu bulmak için sayının 2'ye tümleyenini alınır.

1001 1101 sayısının ikiye tümleyeni 0110 0011 sayıdır.

Bu sayının 10'luk sistemde hangi sayıya denk olduğu hesaplanırsa :

$$(1 * 1 + 1 * 2 + 0 * 4 + 0 * 8 + 0 * 16 + 1 * 32 + 1 * 64 = 99)$$

ilk yazılan sayının -99 olduğu anlaşılmış olur.

**10'luk sistemde ifade edilen negatif sayıların işaretli ikilik sistemde yazılması :**

Önce sayının aynı değerli fakat pozitif olanı ikilik sistemde ifade edilir : Daha sonra yazılan sayının ikiye tümleyenini alınarak, yazmak istenilen sayı elde edilir.

Örnek : İkilik sistemde -17 yazmak istenirse;

önce 17 yazılır.

0001 0001

bu sayının 2'ye tümleyeni alınır

1110 1111 sayısı elde edilir.

Sayı değeri aynı olan Negatif ve Pozitif sayılar birbirlerinin ikiye tümleyenleridir.

İkilik sistemde gösterilmiş olsa da aynı sayının negatifiyle pozitifinin toplamı 0 değerini verecektir. (Deneyiniz!)

Bir byte'lık (8 bitlik) bir alana yazabileceğimiz (işaret bitini dikkate almadan) en büyük sayı 255 (1111 1111) ve en küçük sayı ise 0'dır.(0000 0000). Peki işaret biti dikkate alındığında 1 byte'lık alana yazılabilecek en büyük ve en küçük sayılar ne olabilir?

En büyük sayı kolayca hesaplanabilir. işaret biti 0 olacak (yani sayı pozitif olacak) ve sayı değerini en büyük hale getirmek için diğer bütün bit değerleri 1 olacak, bu sayı 0111 1111 sayıdır. Bu sayıyı desimal sisteme dönüştürsek 127 olduğunu görürüz. Peki ya en küçük negatif sayı kaçtır ve nasıl ifade edilir?

0111 1111 sayısının ikiye tümleyenini alındığında -127 sayısını elde edilir.

1000 0001 (127) Bu sayıdan hala 1 çıkartabilir.

1000 0000 (-128) 1 byte alana yazılabilecek en küçük negatif sayıdır.

Burada dikkat edilmesi gereken iki önemli nokta vardır :

1 byte alana yazılabilecek en büyük sayı sınırı aşıldığında negatif bölgeye geçilir.

0111 1111 (en büyük pozitif tamsayı)

1 (1 toplarsak)

1000 0000 (-128 yani en küçük tamsayı)

yani 1 byte alana yazılabilecek en büyük tamsayıya 1 eklendiğinde 1 byte alana yazılabilecek en küçük tamsayı elde ederiz.

1 byte alana yazılabilecek en küçük tamsayıdan 1 çıkardığımızda da 1 byte alana yazılabilecek en büyük tamsayı elde ederiz.

Yukarıda anlattıklarımıza göre -1 sayısının işaretli ikilik sayı sisteminde 8 bitlik bir alanda aşağıdaki şekilde ifade edilecektir.

-1 = 1111 1111

Yani işaretli ikilik sayı sisteminde tüm bitleri 1 olan sayı -1'dir. İleride bu sayıyla çok işimiz olacak!

## 16'lık sayı sistemi (hexadecimal numbering system) ve 8'lik sayı sistemi (octal system)

Bilgisayarların tamamen 2'lik sistemde çalıştığını söylemiştik, ama yukarıda görüldüğü gibi 2'lik sistemde sayıların ifade edilmesi hem çok uzun hem de zahmetli. Bu yüzden, yazım ve algılama kolaylığı sağlamak için 16'lık ve 8'lik sayı sistemleri de kullanılmaktadır.

16'lık ve 8'lik sayı sistemlerinde sayılar daha yoğun olarak kodlanıp kullanılabilir.

Başta da söz edildiği gibi 10 luk sistemde 10, 2'lik sistemde ise 2 sembol bulunmaktadır. Bu durumda 16'lık sayı sisteminde de 16 sembol bulunur.

ilk 10 sembol 10'luk sistemde kullanılan sembollerle tamamen aynıdır :

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

Daha sonraki semboller

A = 10

B = 11

C = 12

D = 13

E = 14

F = 15

16'lık sayı sisteminde yazılmış bir sayıyı 10'luk sisteme çevirmek için, en sağdan başlayarak basamak değerleri 16'nın artan kuvvetleriyle çarpılır :

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

10'luk sistemde yazılmış bir sayıyı 16'lık sisteme çevirmek için 10 luk sistemden 2'lik sisteme yapılan dönüşümlerdekine benzer şekilde sayı sürekli 16 ya bölünerek, kalanlar soldan sağa doğru yazılır.

Pratikte 16 lık sayı sistemlerinin getirdiği önemli bir avantaj vardır. Bu avantaj 16 lık sayı sistemi ile 2'lik sayı sistemi arasındaki dönüşümlerin kolay bir şekilde yapılmasıdır.

16'lık sistemdeki her digit 2'lik sistemdeki 4 bit (1 Nibble) alan ile ifade edilebilir :

0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Örnek : 2ADFH sayısının (en sondaki H sayının hexadecimal olarak gösterildiğini anlatır yani sayıya ilişkin bir sembol değildir) 16'lık sistemde ifadesi :

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

Bu durumda 2ADFH = 0010 1010 1101 1111

2'lik sistemden 16'lık sisteme yapılacak dönüşümler de benzer şekilde yapılabilir :

Önce sayıları sağdan başlayarak dörder dörder ayırırız (en son dört eksik kalırsa sıfır ile tamamlarız.) Sonra her bir dördlük grup için doğrudan 16'lık sayı sistemindeki karşılığını yazarız.

1010 1110 1011 0001 = AEB1H  
0010 1101 0011 1110 = 2D3EH

soru : 16'lık sayı sisteminde 2 byte'lık bir alanda yazılmış olan 81AC H sayısı pozitif mi negatif midir?

cevap : Sayının yüksek anlamlı biti 1 olduğu için, işaretli sayı sistemlerinde sayı negatif olarak değerlendirilecektir. (1001 0001 1010 1100)

16 bitlik bir alanda ve işaretli sayı sisteminde -1 sayısını nasıl ifade edebiliriz :

Cevap : FFFF

### **8'lik sayı sistemi (octal numbering system)**

Daha az kullanılan bir sayı sistemidir.

8 adet sembol vardır. (0 1 2 3 4 5 6 7)

8'lik sayı sisteminin her bir digiti 2'lik sistemde 3 bit ile ifade edilir.

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

8'lik sayı sisteminin de kullanılma nedeni, 2'lik sayı sistemine göre daha yoğun bir ifade tarzı olması, ve ikilik sayı sistemiyle, 8'lik sayı sistemi arasında yapılacak dönüşümlerin çok kolay bir biçimde yapılabilmesidir.

### **GERÇEK SAYILARIN BELLEKTE TUTULMASI**

Sistemlerin çoğu gerçek sayıları **IEEE 754** standardına göre tutarlar. (**Institute of Electrical and Electronics Engineers**) Bu standarda göre gerçek sayılar için iki ayrı format belirlenmiştir:

#### **single precision format (tek hassasiyetli gerçek sayı formatı)**

Bu formatta gerçek sayı 32 bit (8 byte) ile ifade edilir.  
32 bit üç ayrı kısma ayrılmıştır.

1. İşaret biti (sign bit) (1 bit)

Aşağıda S harfi ile gösterilmiştir.  
İşaret biti 1 ise sayı negatif, işaret biti 0 ise sayı pozitifdir.

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

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

S	EEEEEEEE	FFFFFFFFFFFFFFFFFFFFFF
31	30-----23	22-----0

Aşağıdaki formüle göre sayının değeri hesaplanabilir :

V sayının değeri olmak üzere:

E = 255 ise ve F 0 dışı bir değer ise V = NaN (Not a number) bir gerçek sayı olarak kabul edilmez. Örnek :

0	11111111	000010000001000000000000	= Sayı değil
1	11111111	00010101010001001010101	= Sayı değil

E = 255 ise ve F = 0 ise ve S = 1 ise V = -sonsuz  
E = 255 ise ve F = 0 ise ve S = 1 ise V = +sonsuz

$0 < E < 255$  ise

$$V = (-1)^S * 2^{(E-127)} * (1.F)$$

Önce sayının fraction kısmının başına 1. eklenir. Daha sonra bu sayı  $2^{(E-127)}$  ile çarpılarak noktanın yeri ayarlanır. Noktadan sonraki kısım 2'nin artan negatif kuvvetleriyle çarpılarak elde edilecektir. Örnekler :

$$\begin{aligned} 0 \ 10000000 \ 000000000000000000000000 &= +1 * 2^{(128-127)} * 1.0 \\ &= 2 * 1.0 \\ &= 10.00 \\ &= 2 \end{aligned}$$

$$\begin{aligned} 0 \ 10000001 \ 101000000000000000000000 &= +1 * 2^{(129-127)} * 1.101 \\ &= 2^2 * 1.101 \\ &= 110.100000 \\ &= 6.5 \end{aligned}$$

$$\begin{aligned} 1 \ 10000001 \ 101000000000000000000000 &= -1 * 2^{(129-127)} * 1.101 \\ &= -2^2 * 1.101 \\ &= 110.100000 \\ &= -6.5 \end{aligned}$$

$$\begin{aligned} 0 \ 00000001 \ 000000000000000000000000 &= +1 * 2^{(1-127)} * 1.0 \\ &= 2^{-126} \end{aligned}$$

E = 0 ve F sıfır dışı bir değer ise

$$V = (-1)^S * 2^{(-126)} * (0.F)$$

Örnekler :

$$0 \ 00000000 \ 100000000000000000000000 = +1 * 2^{-126} * 0.1$$

$$0 \ 00000000 \ 000000000000000000000001 = +1 * 2^{-126} * 0.000000000000000000000001$$



$$= 2^{-149} \text{ (en küçük pozitif değer)}$$

$$E = 0 \text{ ve } F = 0 \text{ ve } S = 1 \text{ ise } V = -0$$

$$E = 0 \text{ ve } F = 0 \text{ ve } S = 0 \text{ ise } V = 0$$

### double precision format (çift hassasiyetli gerçek sayı formatı)

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

1. İşaret biti (sign bit) (1 bit)

Aşağıda S harfi ile gösterilmiştir.

İşaret biti 1 ise sayı negatif, işaret biti 0 ise sayı pozitiftir.

2. Üstel kısım (exponent) (11 bit)

Aşağıda E harfleriyle gösterilmiştir.

3. Ondalık kısım (fraction) (52 bit)

Aşağıda F harfleriyle gösterilmiştir.

S EEEEEEEEEEE FF  
63 62-----52 51-----0

Aşağıdaki formüle göre sayının değeri hesaplanabilir :

Aşağıdaki formüle göre sayının değeri hesaplanabilir :

V sayının değeri olmak üzere:

$E = 2047$  ise ve  $F = 0$  dışı bir değer ise  $V = \text{NaN}$  (Not a number) bir gerçek sayı olarak kabul edilmez.

$E = 2047$  ise ve  $F = 0$  ise ve  $S = 1$  ise  $V = -\text{sonsuz}$

$E = 2047$  ise ve  $F = 0$  ise ve  $S = 0$  ise  $V = +\text{sonsuz}$

$0 < E < 2047$  ise

$$V = (-1)^S * 2^{(E-1023)} * (1.F)$$

Önce sayının fraction kısmının başına 1. eklenir. Daha sonra bu sayı  $2^{(E-1023)}$  ile çarpılarak noktanın yeri ayarlanır. Noktadan sonraki kısım 2'nin artan negatif kuvvetleriyle çarpılarak elde edilecektir.

$E = 0$  ve  $F$  sıfır dışı bir değer ise

$$V = (-1)^S * 2^{(-126)} * (0.F)$$

$$E = 0 \text{ ve } F = 0 \text{ ve } S = 1 \text{ ise } V = -0$$

$$E = 0 \text{ ve } F = 0 \text{ ve } S = 0 \text{ ise } V = 0$$



---

## 3 . BÖLÜM : GENEL KAVRAMLAR

### ATOM KAVRAMI VE ATOM TÜRLERİ

Bir programlama dilinde yazılmış programı en küçük parçalara bölmeye çalışalım. Öyle bir noktaya geleceğiz ki, artık bu parçaları daha da bölmeye çalıştığımızda anlamsız parçalar oluşacak. İşte bir programlama dilinde anlam taşıyan en küçük birime **atom (token)** denir.

Atomlar daha fazla parçaya bölünemezler.

Yazdığımız kaynak kod (program) derleyici tarafından ilk önce atomlarına ayrılır. (**Tokenizing**). Atom yalnızca C diline ilişkin bir kavram değildir. Tüm programlama dilleri için atom kavramı söz konusudur, ama farklı programlama dillerinin atomları birbirlerinden farklı olabilir.

Atomları aşağıdaki gibi gruplara ayırabiliriz :

#### 1. Anahtar Sözcükler (keywords, reserved words)

Bu atomlar dil için belli bir anlam taşırlar. Değişken olarak kullanılmaları yasaklanmıştır. Yani programcı bu anahtar sözcükleri kendi tanımlayacağı değişkenlere isim olarak veremez.

Standard ANSI C dilinde 32 tane anahtar sözcük bulunmaktadır.(Derleyici yazan firmalar kendi yazdıkları derleyiciler için ilave anahtar sözcükler tanımlayabilmektedir.)

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

Bazı programlama dillerinde anahtar sözcüklerin küçük ya da büyük harf olması fark etmemektedir. Ama C'de bütün anahtar sözcükler küçük harf olarak tanımlanmıştır. C büyük harf küçük harf duyarlılığı olan bir dildir. (case sensitive) bir dildir. Ama diğer programlama dillerinin çoğunda büyük - küçük harf duyarlılığı yoktur. (case insensitive)

Örneğin, programcı olarak biz kullanacağımız bir değişkene "register" ismini vermeyiz. Çünkü bu bir anahtar sözcüktür. (C dili tarafından rezerve edilmiştir) Ama buna karşın biz istediğimiz bir değişkene REGISTER, Register, RegisTER vs. gibi isimler verebiliriz, çünkü bunlar artık anahtar sözcük sayılmazlar. Anahtar sözcük olan yalnızca tamamen küçük harf ile yazılan "register" dir.

#### 2. İsimlendirilenler (identifiers)

Değişkenlere, fonksiyonlara, makrolara, yapı ve birliklere vs. programlama dili tarafından belirlenmiş kurallara uyulmak şartıyla, istediğimiz gibi isim verebiliriz. Bu atomlar genellikle bellekte bir yer belirtirler.

C dilinde değişkenlerin isimlendirilmesine ilişkin kurallar vardır. Bunu ileride detaylı olarak göreceğiz.

#### 3. Operatörler (Operators)

Operatörler önceden tanımlanmış birtakım işlemleri yapan atomlardır.

Örneğin +, -, \*, /, >=, <= birer operatördür.

Programlama dillerinde kullanılan operatör sembolleri birbirinden farklı olabileceği gibi, operatör tanımlamaları da birbirinden farklı olabilir. Örneğin birçok programlama dilinde üs alma operatörü tanımlanmışken C dilinde böyle bir operatör yoktur. Üs alma işlemi operatör ile değil bir fonksiyon yardımıyla yapılabilir.

C dilinde bazı operatörler iki karakterden oluşmaktadır bu iki karakter bitişik yazılmalıdır aralarına space karakteri koyarsak operatör anlamını yitirir.

#### 4. Sabitler (Constants)

Doğrudan işleme sokulan değişken bilgi içermeyen atomlardır.

Örneğin SAYAC = SON + 10 gibi bir ifadede 10 sabiti doğrudan SON değişkeni ile toplanmaktadır.

#### 5. Stringler (String literals)

İki tırnak içindeki ifadeler string denir. Stringler programlama dillerinin çoğunda tek bir atom olarak alınırlar, daha fazla parçaya bölünemezler.

"STRİNGLER DE BİRER ATOMDUR" ifadesi bir stringdir.

## 6. Ayraçlar ya da noktalama işaretleri (Separators, Punctuators, Delimiters)

Yukarıda sayılan atom sınıflarının dışında kalan tüm atomları bu gruba sokabiliriz. Genellikle diğer atomları birbirinden ayırma amacıyla kullanıldıkları için ayraç olarak isimlendirilirler.

### örnek 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. Bu kaynak kodu atomlarına ayıralım. Amacımız söz konusu programı açıklamak değil, atomlar hakkında gerçek bir programdan örnek vermek.

```
#include <stdio.h>

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

    printf("lütfen bir sayı giriniz\n");
    scanf("%d", &number);
    for(k = 1; k<= number; ++k)
        total += k;
    printf("toplama = %d\n", toplam);
    return 0;
}
```

programda yer alan atomlardan

anahtar sözcükler

*include int for return*

isimlendirilenler (identifiers / variables)

*main n k toplam printf scanf*

operatörler

*= <= ++ +=*

sabitler

*0 1 0*

stringler

*("lütfen bir sayı giriniz\n" ) "%d" "toplama = %d\n"*

ayraçlar noktalama işaretleri

*< > ( ) , ; { }*

## NESNE (OBJECT)

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

$a = b + k$ ; örneğinde  $a$ ,  $b$  ve  $k$  birer nesnedir. Bu ifadede  $a$  nesnesine  $b$  ve  $k$  nesnelerine ait değerlerin toplamı atanmaktadır.

$sonuc = 100$ ;  $sonuc$  isimli nesneye 100 sabit değeri atanmaktadır.

nesnelerin bazı özelliklerinden söz edilebilir :

### İsimleri (name) :

Nesneyi temsil eden karakterlerdir. Nesnelere isimleri programcı tarafından verilir. Her dil için nesne isimlendirmede bazı kurallar söz konusudur.

VERGI = 20000; (Burada VERGI bir nesne ismidir.)

Nesne ile Değişken kavramları birbirine tam olarak eşdeğer değildir. Her değişken bir nesnedir ama her nesne bir değişken değildir. Değişkenler, programcının isimlendirdiği nesnelerdir. Peki programcının isimlendirmede de nesneler var mıdır? Evet, göstericiler konusunda da göreceğimiz gibi, değişken olmayan nesneler de vardır, nesne kavramı değişken kavramını kapsamaktadır.

## Değerleri (value) :

Nesnelerin 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 nesneler ise bir kez değer verildikten sonra bir daha değiştirilemezler.

## Türleri (Type) :

Nesnenin türü derleyiciye o nesnenin nasıl yorumlanacağı hakkında bilgi verir. Yine 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.

Türleri ikiye ayırabiliriz :

### 1. Önceden tanımlanmış veri türleri (default types)

Bu türler programlama dilinin tasarımında var olan veri türleridir. Örneğin C dilinde önceden tanımlanmış 11 ayrı veri türü vardır.

### 2. Programcı tarafından tanımlanan veri türleri (user defined types)

Programlama dillerinin çoğunda programcının tür tanımlamasına izin vermektedir. Örneğin C dilinde yapılar, birlikler, bit alanları, C++ dilinde de sınıflar programcı tarafından tanımlanan veri türleridir.

Programlama dillerindeki tür tanımlamaları birbirlerinden farklı olabilir. Örneğin bazı programlama dillerinde Boolean isimli (Mantıksal Doğru ya da Yanlış değerlerini alan) bir türdür tanımlanmıştır. Ama C dilinde böyle bir tür doğrudan tanımlanmamıştır.

## Faaliyet alanları (scope / visibility) :

Nesnenin, dilin derleyicisi ya da yorumlayıcısı tarafından tanınabildiği program alanıdır. (İleride detaylı inceleyeceğiz)

## Ömürleri (storage duration / lifespan) :

Programın çalıştırılması sırasında nesnenin varlığını sürdürdüğü zaman parçasıdır. (İleride detaylı inceleyeceğiz)

## Bağlantıları (linkage)

Nesnelerin programı oluşturan diğer modüllerde tanınabilme özelliğidir. (İleride detaylı inceleyeceğiz)

## İFADE (Expression)

Değişken, operatör ve sabitlerin kombinasyonlarına ifade denir.

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

geçerli ifadelerdir.

## DEYİM (statement)

Derleyicinin, bilgisayara bir iş yaptıracak şekilde kod üretmesine (yani icra edilebilecek bir kod üretmesine) yol açan ifadelere deyim denir.

Örneğin C dilinde

; ile sonlandırılmış ifadelere deyim diyoruz.

```
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 olacaktır.

Deyimleri ileride detaylı olarak inceleyeceğiz.

## SOL TARAF DEĞERİ (Left Value)

Nesne gösteren ifadelere denir. Bir ifadenin sol taraf değeri olabilmesi için mutlaka bir nesne göstermesi gerekir. Bir ifadenin Sol taraf değeri olarak isimlendirilmesinin nedeni o ifadenin atama operatörünün sol tarafına getirilebilmesidir.

Örneğin a ve b nesneleri tek başına sol taraf değerleridir. Çünkü bu ifadeler atama operatörünün 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 değeridirler.

sabitler sol taraf değeri olamazlar.

## SAĞ TARAF DEĞERİ (Rigth Value)

Daha az kullanılan bir terimdir. Nesne göstermeyen ifadeler sağ taraf değeri olarak isimlendirilirler. Tipik olarak, atama operatörünün sol tarafında bulunamayan yalnızca sağ tarafında bulunabilen ifadelerdir.

Sabitler 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 operatörünün sağ tarafına gelebilen her ifade sağ taraf değeri olarak isimlendirilmez.) Sağ taraf değeri, genellikle bir ifadenin nesne göstermediğini vurgulamak için kullanılır.

## 4 . BÖLÜM : VERİ TÜRLERİ

Nesne (Object) kavramını incelediğimiz zaman, nesnelerin en önemli özelliklerinden birinin nesnenin türü olduğunu belirtmiştik. Tür (type) nesnenin olmazsa olmaz bir özelliğidir ve türü olmayan bir nesneden söz etmek mümkün değildir. Derleyiciler nesnelerle ve verilerle ilgili kod üretirken, tür bilgisinden faydalanırlar. Tür bilgisinden, söz konusu veriyi bellekte ne şekilde tutacaklarını, verinin değerini ne şekilde yorumlayacaklarını, veriyi hangi işlemlere tabi tutabileceklerini öğrenirler.

Programlama dilleri açısından baktığımız zaman türleri iki ayrı gruba ayırabiliriz.

### 1. Önceden tanımlanmış veri türleri (Doğal veri türleri) (Basic types, default types, built-in types, primitive types)

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

### 2. Programcının tanımlanmış olduğu veri türleri (user defined types)

Programlama dillerinin çoğu, önceden tanımlanmış veri türlerine ek olarak, programcının da yeni türler tanımlanmasına izin vermektedir. Programcının tanımlayacağı bir nesne için önceden tanımlanmış veri türleri yetersiz kalıyorsa, programcı kendi veri türünü yaratabilir. C dilinde de programcı yeni bir veri türünü derleyiciye tanıtabilir ve 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 (integer types)

C dilinin toplam 4 ayrı tamsayı veri türü vardır ancak her birinin kendi içinde işaretli ve işaretsiz 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.

Bu türleri sırasıyla inceleyelim:

#### işaretli ve işaretsiz char veri türü :

Şüphesiz **char** sözcüğü İngilizce **character** sözcüğünden kısaltılmıştır ve Türkçe "**karakter**" anlamına gelmektedir. Ancak bu türün ismini, bundan sonraki derste C dilinin bir anahtar sözcüğü olduğunu öğreneceğimiz **char** sözcüğü ile özdeşleştirip, "char türü" (çar diye okuyunuz) diye söyleyeceğiz. İşaretli **char** türünden bir nesnenin bir byte'lık bir alanda tutulması C standartlarıncaya garanti altına alınmıştır.

1 byte'lık bir alanı işaretli olarak kullandığımızda yazabileceğimiz değerlerin **-128 / 127** değerleri arasında değişebileceğini sayı sistemleri dersimizden hatırlayalım.

İşaretsiz char veri 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.

#### işaretli ve işaretsiz short int veri türü (işaretli kısa tamsayı türü - işaretsiz kısa tamsayı türü) :

Yine bundan sonraki derste öğreneceğimiz gibi, **short** ve **int** sözcükleri C dilinin anahtar sözcüklerinden olduğu için bu türün ismini genellikle short int, ya da kısaca short türü olarak telaffuz edeceğiz.



işaretili ve işaretsiz short veri 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 alan kaplayacaktır. İşaretili short int veri türünden bir nesne -32768 - +32767 aralığındaki tamsayı değerlerini tutabilirken, işaretsiz short türü söz konusu olduğundan tutulabilecek değerler 0 - +65535 aralığında olabilir.

### **işaretili int (signed int) türü ve işaretsiz int (unsigned int) türü :**

işaretili ve işaretsiz int veri türünden bir nesne tanımlandığı zaman, nesnenin bellekte kaç byte yer kaplayacağı sistemden sisteme değişebilir. Çoğunlukla 16 bitlik sistemlerde, int veri , 32 bitlik sistemlerde ise int veri türü 4 byte yer kaplamaktadır.

16 bitlik sistem, 32 bitlik sistem ne anlama geliyor.

16 bitlik sistem demekle işlemcinin yazmaç (register) uzunluğunun 16 bit olduğunu anlatıyoruz.

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

### **işaretili ve işaretsiz long int veri türü (işaretili uzun tamsayı türü - işaretsiz uzun tamsayı türü)**

Bu türün ismini genellikle long int, ya da kısaca long türü olarak telaffuz edeceğiz.

işaretili ve işaretsiz long int veri türünden biriyle tanımlanan bir nesnenin bellekte kaç byte yer kaplayacağı sistemden sisteme değişebilir. Sistemlerin çoğunda, long int veri türünden yaratılan nesne bellekte 4 byte'lık bir alan kaplayacaktır. İşaretili long int veri türünden bir nesne -2147483648 - +2147483647 aralığındaki tamsayı değerlerini tutabilirken, işaretsiz long int türü söz konusu olduğundan tutulabilecek değerler 0 - +4.294.967.296 aralığında olur.

## **GERÇEK SAYI TÜRLERİ**

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 uyulmaktadır.

Sistemlerin hemen hemen hepsinde **float** veri türünden bir nesne tanımlandığı zaman bellekte 4 byte yer kaplayacaktır. 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 hassasiyet) olarak isimlendirilmiştir. Bu standartta 32 bitlik alan 3 bölüme ayrılmıştır.

1 bitlik alan (sign bit): gerçek sayının işaret bilgisini yani pozitif mi negatif mi olduğu bilgisini tutar.

8 bitlik alan (exponential part) :

23 bitlik alan (fraction part) : sayının ondalık kısmını tutar.

Sistemlerin hemen hemen hepsinde **double** veri türünden bir nesne tanımlandığı zaman bellekte 8 byte yer kaplayacaktır. Gerçek sayıların bellekte tutulması sistemden sisteme değişebilen özellikler içerebilir. Ancak sistemlerin çoğunda **IEEE 754** sayılı standarda uyulmaktadır.

**long double** veri türünden bir nesne tanımlandığı zaman bellekte 10 byte yer kaplayacaktır.

C dilinin doğal veri türlerine ilişkin bilgileri aşağıda bir tablo şeklinde veriyoruz:

## C DİLİNİN ÖNCEDEN TANIMLANMIŞ

### (DEFAULT ) VERİ TÜRLERİ

#### TAMSAYI TÜRLERİ (INTEGER TYPES)

TÜR İSMİ	UZUNLUK(byte) (DOS / UNIX)		SINIR DEĞERLERİ	
<b>signed char</b>	1		-128	127
<b>unsigned char</b>	1		0	255
<b>signed short int</b>	2		-32.768	32.767
<b>unsigned short int</b>	2		0	65.535
<b>signed int</b>	2	4	-32.768 -2.147.483.648	32.767 2.147.483.647
<b>unsigned int</b>	2	4	0 0	65.535 4.294.967.296
<b>long int</b>	4		-2.147.483.648	2.147.483.647
<b>unsigned long int</b>	4		0	4.294.967.296

#### GERÇEK SAYI TÜRLERİ (FLOATING TYPES)

TÜR İSMİ	UZUNLUK (byte)	SINIR DEĞERLERİ	
		<b>en küçük pozitif değer</b>	<b>en büyük pozitif değer</b>
<b>float</b>	4	$1.17 \times 10^{-38}$ (6 basamak hassasiyet)	$3.40 \times 10^{38}$
<b>double</b>	8	$2.22 \times 10^{-308}$ (15 basamak hassasiyet)	$1.17 \times 10^{38}$ (15 basamak hassasiyet)
<b>long double</b>	10	taşınabilir değil	

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

char türü 1 byte uzunluğunda olmak zorundadır.

short veri türünün uzunluğu int türünün uzunluğuna eşit ya da int türü uzunluğundan küçük olmalıdır. Yani

short <= int

long veri türünün uzunluğu int türüne eşit ya da int türünden büyük olmak zorundadır. Yani

long >= int

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 türü iken gerçek sayılar için double veri türüdür. Peki hangi durumlarda hangi veri türünü kullanmak gerekir. Bu sorunun cevabı olarak hazır bir reçete vermek pek mümkün değil, zira kullanacağımız bir nesne için tür seçerken bir çok faktör söz konusu olabilir, ama genel olarak şu bilgileri verebiliriz :

Gerçek sayılarla yapılan işlemler tam sayılarla yapılan işlemlere göre çok daha fazla 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 hızının belirli ölçüde yavaşlatılması anlamına gelecektir. Bir tamsayı türünün yeterli olması durumunda gerçek sayı türünün kullanılması programın okunabilirliğinin de azalmasına neden olacaktır.

# BİLDİRİM VE TANIMLAMA

Programlama dillerinin çoğunda nesneler kullanılmadan önceye derleyiciye tanıtılırlar.

Nesnelerin kullanılmalarından önce, özellikleri hakkında derleyiciye bilgi verme işlemlerine bildirim (declaration) denir. Bildirim işlemi yoluyla, derleyiciler nesnelerin hangi özelliklere sahip olduklarını anlarlar ve böylece bu nesneler için bellekte uygun bir yer tahsisatı yapabilirler. Yaratılacak nesne hakkında derleyiciye verilecek en önemli bilgi şüphesiz nesneye ilişkin tür (type) bilgisidir.

C dilinde eğer yapılan bir bildirim işlemi, derleyicinin bellekte bir yer ayırmasına neden oluyorsa bu işleme tanımlama (definition) denir. Tanımlama nesne yaratan bir bildirimdir.

Her tanımlama işlem aynı zamanda bir bildirim işlemidir ama her bildirim işlemi bir tanımlama olmayabilir. Başka bir deyişle, tanımlama nesne yaratan bir bildirim işlemidir.

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

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

## C Dilinde Bildirim İşleminin Genel Biçimi

C programlama Dili'nde bildirim işlemi aşağıdaki şekilde yapılmaktadır :

**<tür> <nesne ismi> <;>**

Burada noktalı virgül karakterine sonlandırıcı karakter diyoruz. Noktalı virgül ayıraç türünden bir atomdur ve C'de bütün ifadeler noktalı virgül ile birbirlerinden ayrılırlar.

ifadelerinde bulunan noktalı virgüller bunların ayrı birer ifade olduklarını gösterirler. Eğer bir tek noktalı virgül olsaydı derleyici iki ifadeyi tek bir ifade gibi yorumlayacaktı.

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

Tür belirten anahtar sözcükler, C dilinin önceden tanımlanmış veri türlerine ilişkin anahtar sözcüklerdir. Bu sözcükleri bildirim sentaksında kullanarak, daha önce öğrenmiş olduğumuz 11 temel veri türünden hangisinden değişken tanımlamak istediğimizi derleyiciye bildirmiş oluyoruz. 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 tabloda listelenen seçeneklerden biri olmalıdır. Köşeli parantez içerisindeki ifadeler kullanılması zorunlu olmayan, yani seçime bağlı olan anahtar sözcükleri göstermektedir. Aynı satırdaki tür belirten anahtar sözcükler tamamen aynı anlamda kullanılabilmektedir.

1	<b>char</b>	<b>[signed ] char</b>		
2	<b>unsigned char</b>			
3	<b>short</b>	<b>[signed] short</b>	<b>short [int]</b>	<b>[signed] short [int]</b>

4	<b>unsigned short</b>			
5	<b>int</b>	<b>[signed] int</b>	<b>signed</b>	
6	<b>unsigned int</b>	<b>unsigned</b>		
7	<b>long</b>	<b>[signed] long</b>	<b>long [int]</b>	<b>[signed] long [int]</b>
8	<b>unsigned long</b>	<b>unsigned long [int]</b>		
9	<b>float</b>			
10	<b>double</b>			
11	<b>long double</b>			

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

**char** a;                      **int** a;                      **long** a;  
**signed char** a;            **signed int** a;           **long int** a;  
                                 **signed a;**                      **signed long a;**  
   **signed long int a;**

Yukarıda aynı kolon üzerindeki bildirimlerin hepsi aynı türden nesne yaratır.

Bildirim işleminde nesne ismi olarak, C dilinin isimlendirme kurallarına uygun olarak seçilen herhangi bir isim kullanılabilir.

C dilinde isimlendirilenler (identifiers) kavramı 6 grubu içerir. Değişkenler (variable) bunlardan yalnızca bir tanesidir. Fonksiyonlar (functions), etiketler (labels), makrolar (macros), yapı ve birlik isimleri (structure and union tags), enum sabitleri (enum constants) isimlerini programcılardan alırlar.

## C Dilinin İsimlendirme Kuralları

İsimlendirmede yalnızca 63 karakter kullanılabilir.

Bunlar: İngiliz alfabesinde yer alan 26 karakter, (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, 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) ve alttire (underscore) karakteridir. (\_)

İsimlendirmelerde yukarıda belirtilen karakterlerin dışında başka bir karakterin kullanılması derleme zamanında hata oluşumuna yol açar. (öneğin boşluk karakterinin kullanılması Türkçe karakterlerin kullanılması, +, -, /, \*, & ya da \$ karakterinin kullanılması gibi).

Değişken isimleri rakam karakteriyle başlayamaz. Rakam karakteri dışında, yukarıda geçerli herhangi bir karakterle başlayabilir.

C'nin anahtar sözcükleri isimlendirme amacı ile kullanılamaz.

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

genel\_katsayi\_farki, square\_total, number\_of\_cards gibi.

Başka bir teknik de isimlendirmede her sözcüğün ilk harfini Büyük, diğer harfleri küçük yazmaktır.

GenelKatsayiFarki, SquareTotal, NumberOfCards gibi.

C dilinde yapılan isimlendirmelerde, isimlerin maksimum uzunluğu tanımlanmamıştır. Bu derleyicilere göre değişebilir. Ancak bir çok derleyicide 32 sayısı kullanılmaktadır. Eğer verilen isim 32 karakterden daha fazla karakter içeriyorsa, derleyici bu ismi budar, yani yalnızca ilk 32 karakterini algılar.

C dili büyük harf küçük harf duyarlılığı olan bir dil olduğu için (case sensitive) isimlendirmelerde de büyük harf ve küçük harfler farklı karakterler olarak ele alınacaktır :

---

**var, Var, VAr, VAR, vAR, vaR** deęişiklerinin hepsi ayrı deęişkenler olarak ele alınacaktır.

Bu noktaların hepsi C dilinin sentaksı açısından, hata oluşumunu engellemek için zorunlu durumları belirtmek için anlatılmıştır.

İsimlendirme yazılan programların okunabilirliği açısından da çok önemlidir. Kullanılan isimlerin legal olmalarının dışında, anlamlı olmalarına, kodu okuyacak kişiye bir fikir verecek şekilde seçilmelerine de dikkat edilmelidir.

Bildirim işlemi noktalı virgül ile sonlandırılmalıdır.

## Bildirim Örnekleri

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

Farklı türlere ilişkin bildirimler virgüllerle birbirinden ayrılamaz.

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

ile

tamamen aynı anlamdadır. Yine

ile

tamamen aynı anlamdadır. Ancak bu tür bir bildirimi tavsiye etmiyoruz, standartlar komitesi ileride bu özelliğin dilin kurallarından kaldırılabilceğini bildirmiştir. **(deprecated feature)**.

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

hepsi geçerli bildirimlerdir. Ama yukarıdaki bildirimde, seçimlik olan anahtar sözcükler özellikle kullanılmak isteniyorsa 1. yazım biçimi okunabilirlik açısından tercih edilmelidir.

## Bildirimlerin Kaynak Kod İçinde Yapılış Yerleri

C dilinde genel olarak 3 yerde bildirim yapılabilir :

1. Blokların içinde
2. Tüm blokların dışında.
3. Fonksiyon parametre deęişkeni olarak fonksiyon parantezlerinin içerisinde

Fonksiyon parametre parantezleri içerisinde yapılan bildirimler, başka bir sentaks kuralına uyarlar, bu bildirimler fonksiyonlar konusuna gelindiğinde detaylı olarak incelenecektir.

C dilinde eęer bildirim blokların içinde yapılacaksa, bildirim işlemi blokların ilk işlemi olmak zorundadır. Başka bir deyişle bildirimlerden önce başka bir ifade bulunmamalı ya da bildirimden önce bir fonksiyon çağırılmamalıdır. (Aksi halde derleme zamanı sırasında hata oluşur.)

Bildirimin mutlaka ana bloğun başında yapılması gibi bir zorunluluk yoktur. Eğer içiçe bloklar varsa içteki herhangi bir bloğun başında da (o bloğun ilk işlemi olacak şekilde) bildirim yapılabilir. Örnekler :

Yukarıdaki örnekte var1, var2, ch1, ch2, ch3 değişkenlerinin tanımlanma yerleri doğrudur. Ancak f değişkeni yanlış yerde bildirilmiştir. Çünkü bildirim işleminden önce başka bir işlem (deyim) yer almaktadır. Bu durum derleme aşamasında hata oluşumuna neden olur.

Aynı program parçası şu şekilde yazılmış olsaydı bir hata söz konusu olmazdı :

bu durumda artık **f** değişkeni de kendi bloğunun başında (ilk işlem olarak) tanımlanmıştır.

İleride de göreceğimiz gibi C dilinde tek başına bir noktalı virgöl, bir deyim oluşturur. C sentaksına göre oluşan bu deyim icra edilebilir bir deyimdir. Dolayısıyla aşağıdaki kod parçasında y değişkeninin tanımlaması derleme zamanında hata oluşturacaktır.

Aynı şekilde boş bir blok da C dilinde bir deyim gibi ele alınır. Bu yazım tamamen noktalı virgölün (sonlandırıcının) yalnız kullanılmasına eşdeğerdir. Dolayısıyla aşağıdaki kod parçası da hatalıdır:

Bir ya da birden fazla deyimin de blok içine alınması C dilinde bileşik deyim (compound statement) ismini alır ve bileşik deyimler de icra edilebilir deyim kategorisine girerler. Dolayısıyla aşağıdaki kod parçası da hatalıdır.

(C++ dilinde blok içinde bildirimi yapılan değişkenlerin, blokların ilk işlemleri olacak şekilde bildirilmeleri zorunlu değildir. Yani C++ da değişkenler blokların içinde herhangi bir yerde bildirilebilirler.)

---



## 5 . BÖLÜM : SABİTLER

Veriler ya nesnelerin içerisinde ya da doğrudan sabit biçiminde bulunurlar. Sabitler nesne biçiminde olmayan, programcı tarafından doğrudan girilen verilerdir. Sabitlerin sayısal değerleri derleme zamanında tam olarak bilinmektedir. Örneğin :

ifadesi bize a ve b içindeki sayıların toplanacağı ve c'ye aktarılacağını anlatır. Oysa

ifadesinde x değişkeni içinde saklanan değer ile 10 sayısı toplanmıştır. Burada 10 sayısı herhangi bir değişkenin içindeki değer değildir, doğrudan sayı biçiminde yazılmıştır. Nesnelerin türleri olduğu gibi sabitlerin de türleri vardır. Nesnelerin türleri daha önce gördüğümüz gibi bildirim yapılırken belirlenir. Sabitlerin türlerini ise derleyici, belirli kurallar dahilinde sabitlerin yazılış biçimlerinden tespit eder. Sabitlerin türlerini bilmek zorundayız, çünkü C dilinde sabitler, değişkenler ve operatörler bir araya getirilerek (kombine edilerek) ifadeler (expressions) oluşturulur. Daha sonra detaylı göreceğimiz gibi C dilinde ifadelerin de bir türü vardır ve ifadelerin türleri, içerdikleri sabit ve değişkenlerin türlerinden elde edilir. O halde sabit türlerini detaylı olarak inceleyelim :

### Tamsayı Sabitleri (integer constants)

#### İşaretli Tamsayı Sabitleri (signed int) :

Bunlar tipik olarak int türden değişkenlerine atanan ve tamsayı biçiminde olan sabitlerdir, yazılırken herhangi bir ek almazlar. C'de int türü sınırları içinde olan her tamsayı birer tamsayı sabit (ya da int türden sabit ) olarak ele alınır.

sayılarının hepsi işaretli tamsayı (**signed int**) sabiti olarak ele alınırlar, çünkü **int** türü sayı sınırları içinde bulunuyorlar ve sonlarında herhangi bir ek bulunmuyor.

**int** türü sistem bağımlıdır ve **int** sabitleri de sistemden sisteme değişebilir.

sistem	uzunluk	sınır değerler
DOS, WINDOWS 3.1	2 byte	- 32768, + 32767
UNIX WINDOWS 95	4 byte	-2147483648, +2147483647

Örneğin 425000 Dos'ta **int** sabiti değildir ama UNIX'te **int** sabittir.  
Uzun Tamsayı Sabitleri(Long integer constants)

### Uzun Tamsayı Sabitleri

#### İşaretli Uzun Tamsayı Sabitleri (signed long)

**long** türden sabitler iki türlü ifade edilirler :

1. **long int** türünün sayı sınırları içinde bulunan bir sayının sonuna L ya da l yazarak.  
Bu durumda derleyiciler ilgili sayı **int** sınırları içinde olsa da **long** sabit olarak ele alır.

~~22~~**long** sabittir. çünkü sonunda l eki var.  
~~long~~ sabitlerdir.

**long** sabit kullanımında algılanması daha kolay olduğu için L soneki tercih edilmelidir. l soneki 1 rakamıyla görünüm açısından çok benzediği için karışıklığa neden olabilir.

2. **int** türün sayı sınırlarını aşan fakat **long int** türü sayı sınırları içinde kalan her tamsayı doğrudan **long int** türden sabit olarak ele alınır. Bu durum doğal olarak, DOS gibi **int** ve **long** türlerinin birbirinden farklı olduğu sistemlerde anlamlıdır.

Örneğin DOS'da

**long** türden sabitlerdir. Oysa 32 bitlik sistemlerde **long** türünün uzunluğuyla **int** türün uzunluğu aynı (4 byte) olduğu için bu sayılar **int** sabiti olarak ele alınacaktır. Bu sistemlerde yukarıdaki sayıları **long** sabit olarak ele almak istersek sonlarına l ya da L eklememiz gerekmektedir.

## Karakter Sabitleri (char)

**char** sabitleri tipik olarak **char** türden nesnelere atanan sabitlerdir. (Böyle bir zorunluluk yok.) **char** türden sabitler C dilinde dört ayrı biçimde bulunabilirler.

1. İstenilen bir karakter tek tırnak (single quote) içerisinde kullanılırsa char türden sabit olarak ele alınır. Örnek :

Yukarıdaki gösterimlerin herbiri birer **char** türden sabitidir.

C'de tek tırnak içerisinde belirtilen char sabitleri, aslında o karakterin karakter setindeki (örneğin ASCII tablosundaki) sıra numarasını gösteren bir tamsayıdır.

Bu örnekte aslında ch isimli char türden değişkene a karakterinin ASCII tablosundaki sıra numarası olan 97 sayısı aktarılmaktadır. Tek tırnak içindeki karakter sabitlerini görünce aslında onların küçük birer tamsayı olduğunu bilmeliyiz. Çünkü bellekte karakter diye birşey yoktur herşey ikilik sistemde 1 ve 0 lardan oluşan sayılardır. Yukarıdaki örnekte istersek ch değişkenine aşağıdaki gibi bir atama yapabiliriz:

Bu durumda ch değişkenine sayısal olarak 100 değeri atanacaktır. Bu sayıya da ASCII tablosundaki 'd' karakteri karşılık gelir.

2. Önceden tanımlanmış ters bölü karakter sabitleri (escape sequences)  
Yukarıda tanımlanan yöntemde ekrana basılamayan yani ekranda görüntü oluşturmeyen (non printable) karakterleri ifade edemeyiz. Örneğin çan karakteri (çan sesi) ya da ters boşluk (backspace) karakteri ekrana basılamaz. Tek tırnak içindeki ters bölü (back slash) karakterinden sonra yer alan bazı karakterler çok kullanılan ancak basılamayan bazı karakterlerin yerlerini tutarlar. Bunların listesi aşağıda verilmiştir:

## Önceden Tanımlanmış Ters Bölü Karakter Sabitleri (Escape Sequences)

			Tanım	ASCII No
'\0'	'\x0'	'\0'	NULL karakter	0
'\a'	'\x7'	'\07'	çan sesi (alert)	7
'\b'	'\x8'	'\010'	geri boşluk (back space)	8
'\t'	'\x9'	'\011'	tab karakteri (tab)	9
'\n'	'\xA'	'\012'	aşağı satır (new line)	10
'\v'	'\xB'	'\013'	düşey tab (vertical tab)	11
'\f'	'\xC'	'\014'	sayfa ileri (form feed)	12
'\r'	'\xD'	'\015'	satır başı (carriage return)	13
'\"'	'\x22'	'\042'	çift tırnak (double quote)	34
'\\'	'\x5C'	'\134'	ters bölü (back slash)	92

kullanılışlarına bir örnek :

3. 16'lık (hexadecimal) sayı sisteminde tanımlanmış karakter sabitleri

Tek tırnak içinde tersbölü ve x karakterlerinden sonra bir hex sayı verilirse bu ASCII tablosundaki o sayısal değerin gösterdiği sıra numarasındaki karaktere işaret eden bir karakter sabitidir.

```
'\x41'      /* 41H numaralı ASCII karakteridir. */
'\xff'      /* FFH numaralı '2' karakter sabitidir. */
'\x1C'      /* 1C numaralı ASCII karakter sabitidir. */
```

Küçük "x" yerine büyük harfle "X" yazmak C'nin ilk klasik versiyonunda kabul ediliyordu şimdi artık geçerli değildir. Örnek :

Yukarıdaki örnekte harf isimli char türden değişkene 41H ASCII sıra no.lu karakter atanmıştır. Bu da desimal sistemdeki 65 sayısına eşittir. 65 sıra nolu ASCII karakteri 'A' karakteridir. Dolayısıyla harf isimli değişkene 'A' atanmıştır.

#### 4. 8'lik (octal) sayı sistemde tanımlanmış karakter sabitleri

Tek tırnak içinde tersbölü karakterinden sonra bir oktal sayı yazılırsa bu kullanılan karakter setindeki o sayısal değerin gösterdiği sıra numarasındaki karaktere işaret eden bir karakter sabitidir. Tek tırnak içindeki ters bölü karakterini izleyen sayı üç basamaktan uzun olmamalıdır. Sekizlik sayıların yazımında olduğu gibi sayının başında sıfır olma zorunluluğu yoktur. Bu şekilde yazılabilecek en büyük karakter sabiti '\377' dir.:

Program içinde kullanımına bir örnek:

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

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 sabiti şeklinde ifade edersek hangi sistem olursa olsun çan sesi karakterini verecektir. Ayrıca kodu okuyan kişi çan sesi karakterinin 7 numaralı ASCII karakteri olduğunu bilmeyebilir ama C programcısı olarak '\a' nın çan sesi karakteri olduğunu bilecektir.

Karakter sabitleri konusunu kapatmadan önce karakter setleri konusunda da biraz bilgi verelim:

Günümüzde kullanılan en popüler karakter seti ASCII karakter setidir. ASCII (American Standard Code for Information Interchange) sözcüklerinin başharflerinden oluşan bir kısaltmadır. ASCII setinin orjinal versiyonunda karakterler 7 bitlik bir alanda kodlanmıştır. Bazı bilgisayarlar ise 8 bit alana genişletilmiş ASCII seti kullanırlar ki bu sette 128 yerine 256 karakter temsil edilebilmektedir. Farklı bilgisayarlar farklı karakter setleri kullanabilmektedir. Örnek olarak IBM mainframe'leri daha eski bir set olan EBCDIC seti kullanırlar. Unicode ismi verilen daha geliştirilmiş bir karakter seti vardır ki karakterler 2 byte alanda temsil edildikleri için bu sette 65.536 farklı karakter yer alabilmektedir. Gelecekte bir çok makinanın bu karakter setini destekleyecek biçimde tasarlanacağı düşünülmektedir.

### İşaretsiz türlere ilişkin sabitler

İşaretsiz türlere ilişkin sabitler onların işaretli biçimlerinin sonuna u ya da U getirilmesiyle elde edilirler.

```
-150    (signed) int sabit
150     (unsigned) int sabit.
120L    (signed) long sabit
120     (unsigned) long sabit.
```

Sonek olarak kullanılan l, L, u ve U harflerinin sırası önemli değildir.

Yukarıdaki hepsi geçerli birer uzun tamsayı (unsigned long int) sabittir.

### Tamsayı sabitlerinin 16'lık ve 8'lik sistemlerde gösterilmesi

C'de tamsayı sabitleri (char, int, long) 10'luk sistemin yanısıra 16'lık ve 8'lik sistemlerde de yazılabilirler. Bu sayı sistemleriyle yazılmış tamsayı sabit türleri için yukarıda verilen kurallar aynen geçerlidir. Çünkü bir sayıyı 16'lık ya da 8'lik sistemde yazmakla onun yalnızca görünümünü değiştirmiş oluruz. Sabit türleri gösterim biçimiyle değil nicelikle ilişkilidir. C dilinde ikilik sayı sisteminde sabitlerin yazılması söz konusu değildir.

16'lık sistemde gösterim 0Xbbb.. biçimindedir. (b karakterleri basamakları gösteriyor, 9'dan büyük basamak değerleri için A, B, C, D, E, F karakterleri ya da a, b, c, d, e, f karakterleri kullanılabilir.

8'lik sistemde ise 0bbb.. biçimindedir. (nadir olarak kullanılır). Örnekler:

0x12	sayısı hex gösterimli bir tamsayı ( <b>int</b> ) sabit.
0X12L	sayısı hex gösterimli bir uzun tamsayı ( <b>long</b> ) sabit.
0x1C205470	hex gösterimli bir uzun tamsayı ( <b>long</b> ) sabit. Çünkü (DOS'da) tamsayı sayı sınırını aşmaktadır.
0x1934ul	hex gösterimli işaretsiz uzun tamsayı ( <b>unsigned long</b> ) sabittir.
01234	oktal gösterimli tamsayı ( <b>int</b> ) sabit
0567L	oktal gösterimli uzun tamsayı ( <b>long</b> ) sabit
0777U	oktal gösterimli işaretsiz tamsayı ( <b>unsigned int</b> ) sabit
0452Lu	oktal gösterimli ( <b>unsigned long</b> ) sabit.

Sabitler yukarıda gösterildiği gibi her üç sayı sisteminde de yazılabilir, hatta bir ifade içinde kullanılan sabitler farklı sayı sistemlerinde de yazılmış olabilirler, bu derleme zamanında error oluşturacak bir neden olmayıp tamamen legaldir.

...

### Gerçek Sayı Sabitleri (floating Constants)

#### 1. **float** türden sabitler

Nokta içeren ve sonuna f ya da F getirilmiş sabitler **float** türden sabitler olarak ele alınırlar. Örneğin:

**float** türden sabitlerdir.

Not : Klasik C'de, yani C dilinin standartlaştırılmasından önceki dönemde **float** türden bir sabit elde etmek için, sayının sonuna f eki alması yeterliydi yani nokta içermesi gerekmiyordu ama standartlarda yapılan değişikliklerle artık **float** sabitlerin sonuna ek gelse de mutlaka nokta içermeleri gerekiyor. Yani derleyiciler 3f şeklinde bir yazımı derleme zamanında bir hata (error) mesajıyla bildiriyorlar.

#### 2. **double** türden sabitler

Sonuna f ya da F eki almamış nokta içeren sabitler ile **float** duyarlılığını aşmış sabitler **double** sabitler olarak değerlendirilirler. Örneğin :

- **double** türden sabittir.

#### 3. **long double** türden sabitler

long double türden sabitler noktalı ya da üstel biçimdeki sayıların sonuna l ya da L getirilerek elde edilirler :

**long double** türden sabitlerdir.

**Gerçek Sayı Sabitlerinin Üstel Biçimde Gösterilmesi**

Gerçek sayı sabitleri üstel biçimde de ifade edilebilirler, bunun için sayının sonuna e ya da E eki getirilerek bir tamsayı yazılır. Bu, sayının  $10^x$  gibi bir çarpanla çarpıldığını gösterir.

burada e 10'un kuveti anlamına gelmektedir:

aynı sabitlerdir.

## 6 . BÖLÜM : FONKSİYONLAR

C'de alt programlara fonksiyon denir. Fonksiyon sözcüğü burada matematiksel anlamıyla değil diğer programlama dillerinde kullanılan, "alt program", "prosedür", "subroutine" sözcüklerinin karşılığı olarak kullanılmaktadır.

Fonksiyonlar C dilinin temel yapı taşlarıdır. Çalıştırılabilen bir C programı en az bir C fonksiyonundan oluşur. Bir C programının oluşturulmasında fonksiyon sayısında bir kısıtlama yoktur.

Fonksiyonların onları çağıran fonksiyonlardan aldıkları girdileri ve yine onları çağıran fonksiyonlara gönderdikleri çıktıları vardır. Fonksiyonların girdilerine aktüel parametreler (actual parameters) ya da argumanlar (arguments) diyoruz. Fonksiyonların çıktılarına geri dönüş değeri (return value) diyoruz.

Bir fonksiyon iki farklı amaçla kullanılabilir :

1. Fonksiyon, icrası süresince belli amaçları gerçekleştirir. (Belli işlemleri yapar)
2. Fonksiyon icrası sonunda üreteceği bir değeri kendisini çağıran fonksiyona gönderebilir.

### Fonksiyonların Tanımlanması ve Çağırılması

Bir fonksiyonun ne iş yapacağını ve bu işi nasıl yapacağını C dilinin sentaks kurallarına uygun olarak anlatılmasına o fonksiyonun tanımlanması (definition) denir. Fonksiyon tanımlamaları aşağıda inceleneceği gibi birtakım sentaks kurallarına tabidir.

Bir fonksiyonun çağırılması ise o fonksiyonun yapacağı işi icraya davet edilmesi anlamına gelir. Fonksiyon çağırma ifadesi karşılığında derleyici, programın akışını ilgili fonksiyonun kodunun bulunduğu bölgeye aktaracak şekilde bir kod üretir. Programın akışı fonksiyonun kodu içinde akıp bu kodu bitirdiğinde, yani fonksiyon icra edildiğinde, programın akışı yine fonksiyonun çağırıldığı noktaya geri dönecektir. Fonksiyon çağırmaya ilişkin sentaks da yine aşağıda açıklanacaktır.

### Fonksiyonların Geri Dönüş Değerleri (return values)

Bir fonksiyonun yürütülmesi sonunda onu çağıran fonksiyona dönüşünde gönderdiği değere, fonksiyonun geri dönüş değeri (return value) denmektedir. Her fonksiyon bir geri dönüş değeri üretmek zorunda değildir. Fonksiyonların geri dönüş değerleri farklı amaçlar için kullanılabilir;

1. Bazı fonksiyonlar tek bir değer elde etmek amacıyla tasarlanmışlardır. Elde ettikleri değeri de kendilerini çağıran fonksiyonlara geri dönüş değeri olarak iletirler. Örneğin:

pow fonksiyonu standart bir C fonksiyonudur. Birinci parametresiyle belirtilen sayının ikinci parametresiyle belirtilen kuvvetini hesaplayarak, hesapladığı sayıyı geri dönüş değeri olarak kendisini çağıran fonksiyona iletir. Yukarıdaki örnekte 2 sayısının 3. kuvveti bu fonksiyon yardımıyla hesaplanarak bulunan değer y değişkenine atanmıştır.

2. Bazı fonksiyonların geri dönüş değerleri fonksiyonun yürütülmesi sırasında yapılan işlemlerin başarısı hakkında bilgi verir. Yani bu tür fonksiyonların geri dönüş değerleri test amacıyla kullanılmaktadır. Geri dönüş değerleri yapılması istenen işlemin başarılı olup olmaması durumunu açıklar. Örneğin :

ifadesiyle bellekte 200 byte uzunluğunda bir blok tahsis etmek isteyen programcı bu işlemin başarılı bir biçimde yerine getirilip getirilmediğini de test etmek zorundadır. Hemen arkasından p değişkeninin aldığı değeri kontrol edecek ve işlemin başarısı hakkında bir karara varacaktır. Dolayısıyla malloc fonksiyonunun geri dönüş değeri, fonksiyonun yapması gereken işin başarılı bir şekilde sonuçlanıp sonuçlanmadığını göstermektedir.

3. Bazı fonksiyonlar kendilerine gönderilen argümanları belirli bir kritere göre test ederler. Ürettikleri geri dönüş değerleri ise test sonucunu belirtir. Örneğin:

Burada isalpha fonksiyonu argüman olarak gönderilen karakterin bir harf karakteri olup olmadığını test eder. Eğer harf karakteriyse, isalpha fonksiyonu 0 dışı bir değere geri dönecek, eğer harf karakteri değilse 0 değerine geri dönecektir. Çağırılan fonksiyonda da geri dönüş değerine göre farklı işlemler yapılabilecektir.

4. Bazı fonksiyonlar hem belli bir amacı gerçekleştirirler hem de buna ek olarak amaçlarını tamamlayan bir geri dönüş değeri üretirler. Örneğin :

Burada printf fonksiyonu ekrana Merhaba Dünya yazısını yazmak için kullanılmıştır. Ancak ekrana yazdığı karakter sayısını da geri dönüş değeri olarak vermektedir.

Bir yazı içerisinde bulunan belirli bir karakteri silecek bir fonksiyon tasarladığımızı düşünelim. Fonksiyon işini bitirdikten sonra yazıdan kaç karakter silmiş olduğunu geri dönüş değeri ile çağırıldığı yere bildirilebilir.

5. Bazen geri dönüş değerlerine ihtiyaç duyulmaz. Örneğin yalnızca ekranı silme amacıyla tasarlanmış olan bir fonksiyonun geri dönüş değerine sahip olması gereksizdir.

clrscr fonksiyonu yalnızca ekranı siler, böyle bir fonksiyonun geri dönüş değerine ihtiyacı yoktur.

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

## Fonksiyonların Tanımlanması

Kendi yazdığımız fonksiyonlar için tanımlama (definition) terimini kullanıyoruz. C'de fonksiyon tanımlama işleminin genel biçimi şöyledir:

```
[Geri dönüş değerinin türü] <fonksiyon ismi> ([parametreler])
{
...
...
}
```

Yukarıdaki gösterimde açılabilir parantez içinde belirtilen ifadeler zorunlu olarak bulunması gerekenleri köşeli parantez içinde belirtilen ifadeler ise bulunması zorunlu olmayan, isteğe bağlı (optional) ifadeleri göstermektedir. Tanımlanan fonksiyonlar en az bir blok içerirler. Bu bloğa fonksiyonun ana bloğu denir. Ana blok içinde istenildiği kadar iç içe blok yaratılabilir. Aşağıdaki fonksiyon tanımlamasından fonk1 fonksiyonunun parametre almadığını ve geri dönüş değerinin de **double** türden olduğunu anlıyoruz.

```
double fonk1()
{
    ...
    ...
    ...
}
```

} Fonksiyonun ana bloğu

## void Anahtar Sözcüğü

Bir fonksiyonun parametre değişkeni ya da geri dönüş değeri olmak zorunda değildir. Bir fonksiyonun parametre değişkeni olmadığı iki şekilde belirtilebilir:

1. Fonksiyon parametre parantezinin içi boş bırakılır, yani buraya hiçbirşey yazılmaz.
2. Fonksiyon parametre parantezinin içine **void** anahtar sözcüğü yazılır.

Yukarıdaki tanımlamalar C'de aynı anlama gelmiyor. Fonksiyon prototipleri konusunu öğrenirken bu iki tanımlama arasındaki farkı da öğrenmiş olacağız. Şimdilik bu iki tanımlamanın aynı anlama geldiğini ve fonksiyonun parametre almadığını belirttiklerini varsayacağız.

Geri dönüş değerine ihtiyaç duyulmadığı durumlarda da geri dönüş değerinin türü yerine **void** anahtar sözcüğü yerleştirilir. Örneğin:

Yukarıda tanımlanan sample fonksiyonu parametre almamakta ve bir geri dönüş değeri de üretmemektedir.

Fonksiyon tanımlarken geri dönüş değeri yazılmayabilir. Bu durum geri dönüş türünün olmadığı anlamına gelmez. Eğer geri dönüş değeri yazılmazsa, C derleyicileri tanımlanan fonksiyonun int türden bir geri dönüş değerine sahip olduğunu varsayarlar. Örneğin :

Tanımlanan sample2 fonksiyonunun parametresi yoktur ama **int** türden bir geri dönüş değeri vardır.

C dilinde fonksiyon içinde fonksiyon tanımlanamaz!

Örneğin aşağıdaki durum error oluşturur, çünkü sample2 fonksiyonu sample1 fonksiyonunun içinde tanımlanmıştır:

tanımlamanın aşağıdaki şekilde yapılması gerekirdi :

## Fonksiyonların Çağırılması (function calls)

C dilinde fonksiyon çağırma operatörü olarak () kullanılmaktadır. Bir fonksiyon çağırıldığı zaman programın akışı fonksiyonu icra etmek üzere bellekte fonksiyonun kodunun bulunduğu bölgeye atlar, fonksiyonun icra edilme işlemi bittikten sonra da akış tekrar çağırılan fonksiyonun kalınan yerinden devam eder.

Bir fonksiyonun geri dönüş değeri varsa, fonksiyon çağırma ifadesi geri dönüş değerini üretir. Geri dönüş değeri bir değişkene atanabileceği gibi doğrudan aritmetik işlemlerde de kullanılabilir. Örneğin:

Burada hesapla fonksiyonunun çağırılma ifadesiyle üretilen geri dönüş değeri sonuc değişkenine atanmaktadır. Bir başka deyişle bir fonksiyon çağırma ifadesinin ürettiği değer, ilgili fonksiyonun ürettiği (eğer üretiyorsa) geri dönüş değeridir. Yukarıdaki örnekte önce hesapla() fonksiyonu çağırılacak daha sonra fonksiyonun icra edilmesiyle oluşan geri dönüş değeri sonuc değişkenine atanacaktır.

Fonksiyonların geri dönüş değerleri nesne değildir yani sol taraf değeri (L value) değildir. Yani C dilinde aşağıdaki gibi bir atama her zaman hata verecektir:

Fonksiyonların geri dönüş değerleri sağ taraf değeri (R value) dir.

gibi bir ifade geçerlidir. çağırılmış olan hesapla1 ve hesapla2 fonksiyonları icra edilerek üretilen geri dönüş değerleri ile x değişkeni içindeki değer ve 10 sabiti toplanacaktır. İfadeden elde edilen değer sonuc değişkenine atanacaktır.



Fonksiyonlar ancak tanımlanmış fonksiyonların içerisinde çağırılabilirler. Blokların dışından fonksiyon çağırılamaz.

Çağırılan fonksiyon ile çağırılan fonksiyonun her ikisi de aynı amaç kod içerisinde bulunmak zorunda değildir. Çağırılan fonksiyon ile çağırılan fonksiyon farklı amaç kodları içerisinde de bulunabilir. Çünkü derleme işlemi sırasında bir fonksiyonun çağırıldığını gören derleyici, amaç kod içerisinde (yani .obj içine) çağırılan fonksiyonun adını ve çağırılış biçimini yazmaktadır. Çağırılan fonksiyon ile çağırılan fonksiyon arasında bağlantı kurma işlemi, bağlama aşamasında, bağlayıcı program (linker) tarafından yapılır.

Bu nedenle tanımlanan bir fonksiyon içerisinde, var olmayan bir fonksiyon çağırılsa bile derleme aşamasında bir hata oluşmaz. Hata bağlama aşamasında oluşur. Çünkü bağlayıcı çağırılan fonksiyonu bulamayacaktır.

Bütün C programları çalışmaya main fonksiyonundan başlar. Programın başladığı nokta olma dışında main fonksiyonunun diğer fonksiyonlardan başka hiçbir farkı yoktur. main fonksiyonun icrası bitince program da sonlanır. Bir C programının çalışabilmesi için mutlaka bir main fonksiyonuna sahip olması gerekir. Eğer main fonksiyonu yoksa hata bağlama (linking) aşamasında bağlayıcı program tarafından bildirilecektir.

## Standart C Fonksiyonları

Standard C fonksiyonları, C dilinin standartlaştırılmasından sonra, her derleyicide bulunması zorunlu hale getirilmiş fonksiyonlardır. Yani derleyicileri yazarlar mutlaka standard C fonksiyonlarını kendi derleyicilerinde tanımlamak zorundadırlar. Bu durum C dilinin taşınabilirliğini (portability) artıran ana faktörlerden biridir.

Bir fonksiyonun derleyiciyi yazarlar tarafından tanımlanmış ve derleyici paketine eklenmiş olması, o fonksiyonun standart C fonksiyonu olduğu anlamına gelmez. Derleyiciyi yazarlar programcının işini kolaylaştırmak için çok çeşitli fonksiyonları yazarak derleyici paketlerine eklerler. Ama bu tür fonksiyonların kullanılması durumunda, oluşturulan kaynak kodun başka bir derleyicide derlenebilmesi yönünde bir garanti yoktur, yani artık kaynak kodun taşınabilirliği azalır. Örneğin printf fonksiyonu standart bir C fonksiyonudur. Yani printf fonksiyonu her derleyici paketinde aynı isimle bulunmak zorundadır.

Standart C fonksiyonları özel kütüphanelerin içerisinde bulunurlar. Başlık dosyaları içinde, yani uzantısı .h biçiminde olan dosyaların içinde standart C fonksiyonlarının prototipleri bulunmaktadır. Fonksiyon prototipleri konusu ileride detaylı olarak incelenecektir.

Kütüphaneler (libraries) derlenmiş dosyalardan oluşur. DOS'da kütüphane dosyalarının uzantısı .lib, UNIX'de ise .a (archive) biçimindedir. WINDOWS altında uzantısı .dll biçiminde olan dinamik kütüphaneler de bulunmaktadır.

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

Standart fonksiyonlarını kullanmak programların taşınabilirliğini artırdığı gibi proje geliştirme süresini de kısaltır. Bu yüzden iyi bir C programcısının C dilinin standart fonksiyonlarını çok iyi tanıması ve bu fonksiyonları yetkin bir şekilde kullanabilmesi gerekmektedir.

## Fonksiyonların Geri Dönüş Değerlerinin Oluşturulması

C dilinde fonksiyonların geri dönüş değerleri **return** anahtar sözcüğü ile oluşturulur. **return** anahtar sözcüğünün bir başka işlevi de içinde bulunduğu fonksiyonu sonlandırmasıdır.

Yukarıdaki örnekteki sample fonksiyonunda **return** anahtar sözcüğünün yanında yer alan  $x * y$  ifadesi sample fonksiyonunu sonlandırmakta ve sample fonksiyonunun geri dönüş değerini oluşturmaktadır. Fonksiyonun geri dönüş değeri, main fonksiyonu içinde c değişkenine atanmış ve daha sonra standart C fonksiyonu olan printf ile c fonksiyonunun değeri ekrana yazdırılmıştır. fonksiyonun geri dönüş değerini başka bir değişkene atamadan aşağıdaki ifade ile de doğrudan ekrana yazdırabilirdik :

Aynı örnekte main fonksiyonu içinde de bir **return** ifadesinin yer aldığı görülmektedir. main de bir fonksiyondur ve main fonksiyonunun da bir geri dönüş değeri olabilir. main fonksiyonun geri dönüş değeri programın icrası bittikten sonra işletim sistemine bildirilmektedir. main fonksiyonunun başına bir geri dönüş değer türü yazılmazsa derleyiciler main fonksiyonunun geri dönüş değerinin int türden olduğunu varsayarlar. Özellikle yeni derleyiciler, tanımlamalarında bir geri dönüş değeri üretecekleri belirtilen fonksiyonlarının, **return** anahtar sözcüğüyle geri dönüş değeri üretmemelerini bir uyarı (warning) mesajı ile bildirirler. Borland derleyicilerinde bu uyarı mesajı genellikle "warning : function should return a value..." şeklindedir. Bu uyarı mesajını kesmek için iki yol vardır:

1. main fonksiyonu da yukarıdaki örnekte olduğu gibi int türden bir geri dönüş değeri üretir. Geleneksel olarak bu değer 0 ise programın problemsiz bir şekilde sonlandırıldığı anlamına gelir.
2. main fonksiyonunun başına void anahtar sözcüğü yazılarak bu fonksiyonun bir geri dönüş değeri üretmeyeceği derleyiciye bildirilir. Bu durumda derleyici geri dönüş değeri beklemediği için bir uyarı mesajı göndermez.

**return** anahtar sözcüğünün kullanılması zorunlu değildir. Bir fonksiyon içinde **return** anahtar sözcüğü kullanılmamışsa fonksiyonun icrası, fonksiyonun ana bloğunun sonuna gelindiğinde otomatik olarak biter. Tabi bu tür bir fonksiyon anlamlı bir şekilde bir geri dönüş değeri üretemeyecektir. Bir geri dönüş değerine sahip olacak şekilde tanımlanmış fakat return ile geri dönüş değeri oluşturulmamış fonksiyonlar rastgele bir değer döndürürler.

**return** anahtar sözcüğünden sonra parantez kullanılabilir ama parantez kullanımı zorunlu değildir. Okunabilirlik açısından özellikle uzun **return** ifadelerinde parantez kullanımı tavsiye edilmektedir.

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

**return** 5; /\* return ifadesinin değişken içermesi bir zorunluluk değildir. Bir fonksiyon sabit bir değerle de geri dönebilir. \*/

**return** sample();

Bu örnekte **return** anahtar sözcüğünden sonra bir fonksiyon çağırma ifadesi yer almaktadır. Bu durumda önce çağırılan fonksiyon icra edilir, ve geri dönüş değeri elde edilir, daha sonra elde edilen geri dönüş değeri tanımlanması yapılan fonksiyonun da geri dönüş değeri yapılmaktadır.

Geri dönüş değeri olmayan fonksiyonlarda **return** anahtar sözcüğü yanında bir ifade olmaksızın tek başına da kullanılabilir :

**return**;

Bu durumda **return** içinde yer aldığı fonksiyonu geri dönüş değerini oluşturmadan sonlandırır.

C dilinde fonksiyonlar yalnızca bir geri dönüş değeri üretebilirler. Bu da fonksiyonların kendilerini çağırarak fonksiyonlara ancak bir tane değeri geri gönderebilmeleri anlamına gelmektedir. Ancak, fonksiyonların birden fazla değeri ya da bilgiyi kendilerini çağırarak

fonksiyonlara iletmeleri gerekiyorsa, C dilinde bunu sağlayacak başka mekanizmalar vardır ve bu mekanizmalar ileride detaylı olarak incelenecektir.

Fonksiyonların ürettiği geri dönüş değerlerinin kullanılması yönünde bir zorunluluk yoktur. Örneğin `fonk()` fonksiyonu **int** türden bir değeri geri dönen bir fonksiyon olsun:

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

yukarıdaki ifadeye `fonk` fonksiyonunun geri dönüş değeri `a` değişkenine atanmaktadır. Dolayısıyla biz bu fonksiyonu bir kez çağırmanıza karşın artık geri dönüş değerini `a` değişkeninde tuttuğumuz için, bu geri dönüş değerine fonksiyonu tekrar çağırmandan istediğimiz zaman ulaşabiliriz. Ancak:

```
fonk();
```

şeklinde bir fonksiyon çağırma ifadesinde artık geri dönüş değeri bir değişkende saklanmamaktadır. Bu duruma geri dönüş değerinin kullanılmaması denir. (discarded return value).

Örneğin standart bir C fonksiyonu olan `printf` fonksiyonun da bir geri dönüş değeri vardır (`printf` fonksiyonu ekrana bastırılan toplam karakter sayısına geri döner) ama bu geri dönüş değeri nadiren kullanılır.

## Fonksiyon Parametre Değişkenlerinin Tanımlanması

Bir fonksiyonun parametreleri ya da parametre değişkenleri fonksiyonların kendilerini çağıran fonksiyonlardan aldıkları girdileri tutan değişkenleridir. Bir fonksiyonun parametre sayısı ve bu parametrelerin türleri gibi bilgiler, fonksiyonların tanımlanması sırasında derleyicilere bildirilirler.

C dilinde fonksiyonların tanımlanmasında kullanılan 2 temel biçim vardır. Bu biçimler birbirlerinden fonksiyon parametrelerinin derleyicilere tanıtılma şekli ile ayrılırlar. Bu biçimlerden birincisi eski biçim (old style) ikincisi ise yeni biçim (new style) olarak adlandırılır.

Artık eski biçim hemen hemen hiç kullanılmamaktadır, ama C standartlarına göre halen geçerliliğini korumaktadır. Kullanılması tavsiye edilen kesinlikle yeni biçimdir ancak eski kodların ya da eski kaynak kitapların incelenmesi durumunda bunların anlaşılabilmesi için eski biçimin de öğrenilmesi gerekmektedir.

### Eski Biçim (old style)

Bu biçimde fonksiyonun parametre değişkenlerinin yalnızca ismi fonksiyon parantezleri içinde yazılır. (Eğer parametre değişkenleri birden fazla ise aralarına virgül koyulur. Daha sonra alt satıra geçilerek bu değişkenlerin bildirimi yapılır. Bu bildirimler daha önce öğrendiğimiz, C dilinin bildirim kurallarına uygun olarak yapılır. Örnek :

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

Yukarıda tanımlanan `alan` fonksiyonunun iki parametre değişkeni vardır ve bu parametre değişkenlerinin isimleri `x` ve `y`'dir. Her iki parametre değişkeni de `int` türündendir.

```
int sample(a, b, c)
int a;
double b;
long c;
{
```

```
}    ...  
}
```

Bu örnekte ise `sample` fonksiyonu üç parametre almaktadır. 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 biçimin dezavantajı gereksiz yere uzun oluşudur. Çünkü fonksiyon parantezlerinin içinde parametre değişkenlerinin ismi yer almakta sonra tekrar bu isimler alt satırlarda yeniden kullanılarak bildirim yapılmaktadır.

### Yeni Biçim (new style)

Yeni biçim eski biçime göre hem daha kısadır hem de okunabilmesi eski biçime göre çok daha kolaydır.

Yeni biçimde fonksiyon parametre değişkenlerinin bildirimi fonksiyon parantezlerinin içinde yalnızca bir kez yapılmaktadır. Bu biçimde fonksiyonun parantezlerinin içine parametre değişkenin türü ve yanına da ismi yazılır. Eğer birden fazla fonksiyon parametre değişkeni varsa bunlar virgüllerle ayrılır ancak her defasında tür bilgisi yeniden yazılır . Örnek :

```
int sample (int x, int y)  
{  
    ...  
}
```

```
int fonk(double x, int y)  
{  
    ...  
}
```

Bu biçimde dikkat edilmesi gereken önemli nokta, fonksiyon parametre değişkenleri aynı türden olsalar bile her defasında tür bilgisinin tekrar yazılması zorunluluğudur. Örneğin :

```
int sample (double x, y)           /* error */  
{  
    ...  
}
```

bildirimi hatalıdır. Doğru tanımlamanın aşağıdaki şekilde olması gerekir:

```
int sample (double x, double y)  
{  
    ...  
}
```

### Klavyeden Karakter Alan C Fonksiyonları

Sistemlerin hemen hemen hepsinde klavyeden karakter alan 3 ayrı C fonksiyonu bulunur. Bu fonksiyonların biri tam olarak standarttır ama diğer ikisi sistemlerin hemen hemen hepsinde bulunmasına karşın tam olarak standart değildir.

#### getchar Fonksiyonu

**int getchar(void);**

getchar standart bir C fonksiyonudur. Geri dönüş değeri klavyeden alınan karakterin ASCII tablosundaki sıra numarasını gösteren **int** türden bir sayıdır. getchar fonksiyonu klavyeden karakter almak için enter tuşuna ihtiyaç duyar.

Aşağıda yazılan programda önce klavyeden bir karakter alınmış daha sonra alınan karakter ve karakterin sayısal karşılıkları ekrana yazdırılmıştır. (getchar fonksiyonunun geri dönüş değeri klavyede basılan tuşa ilişkin karakterin sistemde kullanılan karakter seti tablosundaki sıra numarasıdır.)

```
#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 dosyasında bir makro olarak tanımlanmıştır. Makrolar konusunu daha ileride inceleyeceğiz.

### getch Fonksiyonu

**int getch(void);**

getchar fonksiyonu gibi bu fonksiyonda klavyede basılan tuşun kullanılan karakter setindeki sıra numarasıyla geri döner. getch fonksiyonundan iki farkı vardır.

1. Basılan tuş ekranda görünmez.
2. Enter tuşuna ihtiyaç duymaz.

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

Tam olarak standardize edilmemiştir ama neredeyse bütün sistemlerde bulunur. getch fonksiyonu özellikle tuş bekleme ya da onaylama amacıyla kullanılmaktadır:

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

Burada basılacak tuşun programcı açısından bir önemi olmadığı için fonksiyonun geri dönüş değeri kullanılmamıştır.

### getche Fonksiyonu

**int getche(void);**

getche İngilizce get char echo sözcüklerinden gelmektedir. getche fonksiyonu da basılan tuşun karakter setindeki sıra numarasıyla geri döner ve enter tuşuna gereksinim duymaz. Ama basılan tuşa ilişkin karakter ekranda görünür.

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

## Ekrana Bir Karakterin Görüntüsünü Yazan C Fonksiyonları

C dilinde ekrana karakter yazamakta iki fonksiyonun kullanıldığı görülür :

### putchar Fonksiyonu

***int putchar(int ch);***

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

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

```
main()
{
    char ch;

    ch = getchar();
    putchar (ch);
    return 0;
}
```

Burada putchar fonksiyonunun yaptığı işi printf fonksiyonuna da yaptırabilirdik;

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

putchar(ch) ile tamamen aynı işleve sahiptir.

putchar fonksiyonu ile '\n' karakterini yazdığımızda printf fonksiyonunda olduğu gibi imleç sonraki satırın başına geçer. putchar fonksiyonu ekrana yazılan karakterin ASCII karşılığı ile geri dönmektedir.

putchar fonksiyonu derleyicilerin çoğunda stdio.h dosyası içinde bir makro olarak tanımlanmıştır. Makrolar konusunu ileriki derslerde detaylı olarak öğreneceğiz.

### putch Fonksiyonu

***int putch(int ch);***

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

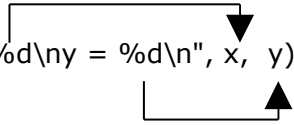
### printf Fonksiyonu

Değişkenlerin içerisindeki değerler aslında bellekte ikilik sistemde tutulmaktadır. Bir değişkenin içerisindeki değer ekrana, kağıt sistemde ve nasıl yazdırılacağı programcının isteğine bağlıdır. Değişkenlerin içerisindeki değerlerin ekrana yazdırılmasında printf fonksiyonu kullanılır. printf standart bir C fonksiyonudur.

printf aslında çok ayrıntılı özelliklere sahip bir fonksiyondur. Burada yalnızca temel özellikleri görsel bir biçimde açıklanacaktır. printf iki tırnak içerisindeki karakterleri ekrana yazar. Ancak iki tırnak içinde gördüğü % karakterlerini ekrana yazmaz. printf fonksiyonu % karakterlerini yanındaki karakter ile birlikte format karakteri olarak yorumlar. Format karakterleri iki tırnaktan sonra yazılan parametrelerle birebir eşleştirilir. Örnek:

```
int x, y;
```

```
x = 125;
y = 200;
printf("x = %d\n y = %d\n", x, y);
```



printf fonksiyonunun yukarıdaki şekilde çağırılmasıyla x ve y değişkeni içindeki değerler ekrana onluk sistemde yazdırılacaktır.

Format karakterleri yerine eşlenen değişkenlerin içerisindeki değerler ekrana yazılır. Format karakterleri sayıların ekrana nasıl yazılacağını belirtmekte kullanılır.

format karakteri	Anlamı
%d	int türünü desimal sistemde yazar.
%ld	long türünü desimal sistemde yazar
%x	unsigned int türünü hexadecimal sistemde yazar.
%X	unsigned int türünü hexadecimal sistemde yazar.(semboller büyük harfle)
%lx	unsigned long türünü hexadecimal sistemde yazar.
%u	unsigned int türünü decimal sistemde yazar.
%o	unsigned int türünü oktal sistemde yazar.
%f	float ve double türlerini desimal sistemde yazar.
%lf	double türünü desimal sistemde yazar.
%e	gerçek sayıları üstel biçimde yazar.
%c	char veya int türünü karakter görüntüsü olarak yazdırır.
%s	string olarak yazdırır.
%lf	long double türünü desimal sistemde yazdırır.

Yukarıdaki tabloda görüldüğü gibi **double** türü hem %f format karakteri hem de %lf format karakteri ile yazdırılabilmektedir. Ama %lf (okunabilirliği artırdığı için) daha çok tercih edilmektedir.

Yukarıdaki tabloya göre **unsigned int** türünden bir sayıyı aşağıdaki şekillerde yazdırabiliriz :

```
unsigned int u;
```

```
printf("%u", u);          /* u sayısını 10'luk sistemde yazar */
printf("%o", u);          /* u sayısını 8'lik sistemde yazar */
printf("%x", u);          /* u sayısını 16'lık sistemde yazar */
```

**short** bir sayıyı yazarken d o u ya da x karakterlerinden önce h karakterini kullanıyoruz :

```
short int sh;
```

```
printf("%hd", sh);        /* 10'luk sistemde yazar */
```

```
unsigned short int unsh;
```

```
printf("%hu", unsh);      /* 10'luk sistemde yazar */
printf("%ho", unsh);      /* 8'lik sistemde yazar */
printf("%hx", unsh);      /* 16'lık sistemde yazar */
```

**long** bir sayıyı yazarken d o u ya da x karakterlerinden önce l karakterini kullanıyoruz :

```
long int lo;
```

```
printf("%ld", lo);        /* 10'luk sistemde yazar */
```

```
unsigned long int unlo;
```

```
printf("%lu", unlo);      /* 10'luk sistemde yazar */
printf("%lo", unlo);      /* 8'lik sistemde yazar */
printf("%lx", unlo);      /* 16'lık sistemde yazar */
```

Yukarıdaki bilgilerde **unsigned** bir tamsayıyı printf fonksiyonuyla 8'lik ya da 16'lık sistemde yazdırabileceğimizi gördük. Peki **signed** bir tamsayıyı 8'lik ya da 16'lık sistemde yazdıramaz mıyız? Yazdırırsak ne olur? Söz konusu **signed** tamsayı pozitif olduğu sürece bir sorun olmaz. Sayının işaret biti 0 olduğu için sayının nicel büyüklüğünü etkilemez. Yani doğru sayı ekrana yazar, 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 olmayacaktır.

% karakterinin yanında önceden belirlenmiş bir format karakteri yoksa , % karakterinin yanındaki karakter ekrana yazılır.

%% (%) karakterini yaz anlamına gelir.

### scanf Fonksiyonu

scanf fonksiyonu klavyeden her türlü bilginin girişine olanak tanıyan standart bir C fonksiyonudur. scanf fonksiyonu da printf fonksiyonu gibi aslında çok detaylı, geniş kullanım özellikleri olan bir fonksiyondur. Ancak biz bu noktada scanf fonksiyonunu yüzeysel bir şekilde tanıyacağız.

scanf fonksiyonunun da birinci parametresi bir stringdir. Ancak bu string klavyeden alınacak bilgilere ilişkin format karakterlerini içerir. Bu format karakterleri önceden belirlenmiştir ve % karakterinin yanında yer alırlar. scanf fonksiyonunun kullandığı format karakterlerinin printf fonksiyonunda kullanılanlar ile aynı olduğunu söyleyebiliriz. Yalnızca gerçek sayılara ilişkin format karakterlerinde önemli bir farklılık vardır. printf fonksiyonu %f formatı ile hem **float** hem de **double** türden verileri ekrana yazabilirken scanf fonksiyonu %f format karakterini yalnızca **float** türden veriler için kullanır. **double** tür için scanf fonksiyonunun kullandığı format karakterleri %lf şeklindedir. scanf fonksiyonunun format kısmında format karakterlerinden başka bir şey olmamalıdır. printf fonksiyonu çift tırnak içindeki format karakterleri dışındaki karakterleri ekrana yazıyordu, ancak scanf fonksiyonu format karakterleri dışında string içine yazılan karakterleri ekrana basmaz, bu karakterler tamamen başka anlama gelecektir. Bu nedenle fonksiyonun nasıl çalıştığını öğrenmeden bu bölgeye format karakterlerinden başka bir şey koymayınız. Buraya konulacak bir boşluk bile farklı anlama gelmektedir.

```
int x, y;
```

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

Yukarıdaki örnekte x ve y sayıları için desimal sistemde klavyeden giriş yapılmaktadır. Giriş arasına istenildiği kadar boşluk karakteri konulabilir. Yani ilk sayıyı girdikten sonra ikinci sayıyı SPACE, TAB ya da ENTER tuşuna bastıktan sonra girebilirsiniz. Örneğin:

```
5      60
```

biçiminde bir giriş geçerli olacağı gibi;

```
5
60
```

biçiminde bir giriş de geçerlidir. scanf fonksiyonuna gönderilecek diğer argümanlar & operatörü ile kullanılmaktadır. & bir gösterici operatörüdür. Bu operatörü göstericiler konusunda öğreneceğiz.



scanf fonksiyonunun yalnızca giriş için kullanılır, ekrana yazmak için printf fonksiyonunu kullanmamız gerekir :

```
int number;
```

```
printf("bir sayi giriniz : ");  
scanf("%d", &number);
```

C++ dilinde bir fonksiyon tanımlanmasında fonksiyonun geri dönüş değerinin türü olarak **void** anahtar sözcüğü yazılmamışsa, fonksiyon **return** anahtar sözcüğü kullanılarak mutlaka bir geri dönüş değeri üretmelidir. Fonksiyonun geri dönüş değeri üretmemesi durumunda derleme zamanında hata oluşacaktır. Yani C dilinde olduğu gibi rasgele bir değer üretilmesi söz konusu değildir. Yine C++ dilinde geri dönüş değeri üretecek bir fonksiyonun tanımlanması içinde **return** anahtar sözcüğü yalın olarak kullanılamaz. **return** anahtar sözcüğünün yanında mutlaka bir ifade yer almalıdır.

## 7 . BÖLÜM : NESNELERİN FAALİYET ALANLARI VE ÖMÜRLERİ

Daha önce C dilinde nesnelerin 3 ayrı özelliğini görmüştük. Bunlar nesnelerin isimleri, değerleri ve türleriydi. Nesnelerin C dili açısından çok önem taşıyan iki özellikleri daha söz konusudur. Bunlar tanınabilirlik alanları (scope) ve ömürleridir (storage duration). Bu dersimizde bu iki kavramı detaylı olarak inceleyeceğiz.

### Tanınabilirlik Alanı (scope / visibility)

Tanınabilirlik alanı bir nesnenin ömrünü sürdürdüğü ve tanınabildiği program aralığıdır. Burada program aralığı demekle kaynak kodu kastediyoruz. Dolayısıyla tanınabilirlik 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 değişkeni tanımlıyor olmamız o değişkene kodun istediğimiz bir yerinde ulaşabilmemizi sağlamaz. Tanınabilirlik alanlarını 2 ayrı grupta toplayabiliriz :

Blok tanınabilirlik alanı. (Block scope): Bir değişkenin tanımlandıktan sonra, derleyici tarafından, yalnızca belirli bir blok içinde tanınabilmesidir.

Dosya tanınabilirlik alanı (File scope) : Bir değişkenin tanımlandıktan sonra tüm kaynak dosya içinde, yani tanımlanan tüm fonksiyonların hepsinin içerisinde tanınabilmesidir.

Değişkenleri de tanınabilirlik alanlarına göre ikiye ayırabiliriz :

Yerel değişkenler (local variables)

Global değişkenler (global variables)

C dili için çok önemli olan bu değişken tiplerini şimdi detaylı olarak inceleyeceğiz :

### Yerel Değişkenler (local variables)

Blokların içlerinde tanımlanan değişkenlere yerel değişkenler denir. Hatırlanacağı gibi C dilinde blokların içlerinde tanımlanan değişkenlerin tanımlama işlemleri blok içinde ilk işlem olarak yapılmalıydı. (C++ dilinde böyle bir zorunluluk yoktur) Yerel değişkenler blok içlerinde tanımlanan değişkenlerdir, iç içe blokların söz konusu olması durumunda hangi blok içerisinde tanımlanırlarsa tanımlansınlar bunları yerel değişken olarak adlandıracamız. Yani yerel değişken olmaları için en dış blok içinde tanımlanmaları gerekmektedir.

Yerel değişkenlerin tanınabilirlik alanı blok tanınabilirlik alanıdır. Yani yerel değişkenlere yalnızca tanımlandıkları blok içinde ulaşılabilir. Tanımlandıkları bloğun daha dışında bir blok içinde bu değişkenlere ulaşamayız. Örnek :

```
main ()
{
    float x;
    ...
    {
        int y;
        ...
        {
            int z;
            ...
            ...
        }
    }
}
```

x değişkeninin tanınabilirlik alanı

y değişkeninin tanınabilirlik alanı

z değişkeninin tanınabilirlik alanı

Yukarıdaki örnekte tanımlanan değişkenlerden hepsi yerel değişkenlerdir. Çünkü x y z değişkenleri blokların içlerinde tanımlanmışlardır. Bu değişkenlere yalnızca tanımlanmış oldukları blok içinde ulaşabiliriz. Tanımlandıkları blok dışında bunlara ulaşmaya çalışmak derleme aşamasında error ile neticelenecektir.

Dikkat etmemiz gereken bir nokta da şudur : Yukarıdaki örnekte bu değişkenlerin hepsi yerel değişkenler oldukları için blok tanınabilirlik alanı kuralına uyarlar, ancak bu tanınabilirlik alanlarının tamamen aynı olmasını gerektirmez. Yukarıdaki şekilden de görüldüğü gibi x değişkeni en geniş tanınabilirlik alanına sahipken y değişkeni daha küçük ve z değişkeni de en küçük tanınabilirlik alanına sahiptir.

Yukarıdaki örneği genişletelim :

```
main ()
{
    float x = 2.5;

    printf("x = %f\n", x); /* LEGAL BU ALANDA x'E ULAŞILABİLİR */
    printf("y = %d\n", y); /* ERROR BU ALANDA y'YE ULAŞILAMAZ. */
    printf("z = %ld\n", z); /* ERROR BU ALANDA z'YE ULAŞILAMAZ. */
    {
        int y = 1;

        printf("x = %f\n", x); /* LEGAL BU ALANDA x'E ULAŞILABİLİR */
        printf("y = %d\n", y); /* LEGAL BU ALANDA y'E ULAŞILABİLİR */
        printf("z = %ld\n", z); /* ERROR BU ALANDA z'YE ULAŞILAMAZ. */
        ...
        ...
    }

    {
        long z = 5;

        printf("x = %f\n", x); /* LEGAL BU ALANDA x'E ULAŞILABİLİR */
        printf("y = %d\n", y); /* LEGAL BU ALANDA x'E ULAŞILABİLİR */
        printf("z = %ld\n", z); /* LEGAL BU ALANDA x'E ULAŞILABİLİR */
    }
}
```

C dilinde aynı isimli birden fazla değişken tanımlanabilmektedir. Genel kural şudur: iki değişkenin scopları (tanınabilirlik alanları aynı ise) aynı isimi taşıyamazlar, aynı isim altında tanımlanmaları derleme zamanında hata oluşturur. İki değişkenin scoplarının aynı olup olmadıkları nasıl tespit edilecek? Standartlar bu durumu şöyle açıklamaktadır : İki değişkenin scopları aynı kapanan küme parantezinde sonlanıyorsa, bu değişkenlerin scopları aynı demektir.

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

Yukarıdaki program parçasının derlenmesi derleme aşamasında error ile neticelenir. Çünkü her iki a değişkeninin de tanınabilirlik alanı (scopları) aynıdır. (scopları aynı küme paranteziyle sonlanmaktadır.)

Bu durum error oluşturmasaydı, yukarıdaki örnekte derleyici hangi a değişkeninin yazdırılmak istendiğini nasıl anlayacaktı. Zira bu durum error ile engellenmeseydi printf fonksiyonunun çağırıldığı yerde her iki a değişkeni de tanınabilir olacaktı.

C dilinde farklı tanınabilirlik alanlarına sahip birden fazla aynı isimli değişken tanımlanabilir. Çünkü derleyiciler için artık bu değişkenlerin aynı isimli olması önemli değildir. Bunlar bellekte farklı yerlerde tutulurlar.

```
{
    int x = 100;

    printf("%d\n", x);
    {
        int x = 200;

        printf("%d\n", x);
        {
            int x = 300;

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

Yukarıdaki program parçasında bir hata bulunmamaktadır. Çünkü her üç x değişkeninin de tanınabilirlik alanları birbirlerinden farklıdır. Peki yukarıdaki örnekte içerideki bloklarda x ismini kullandığımızda derleyici hangi x değişkenini kast ettiğimizi nasıl anlayacak? Belirli bir kaynak kod noktasında, aynı isimli birden fazla değişkenin faaliyet alanı (scope) içindeysek, değişken ismini kullandığımızda derleyici hangi değişkene ulaşacaktır? Aynı isimli değişkenlere ulaşma konusundaki kural şudur : C dilinde daha dar faaliyet alanına sahip değişken diğer aynı isimli değişkenleri maskeler.

Konunun daha iyi anlaşılması için bir kaç örnek daha verelim :

```
{
    int a;
    char ch;
    long b;
    double a, f; /* hata aynı tanınabilirlik alanında ve aynı blok seviyesinde aynı isimli iki
                  değişken tanımlanmış */
}

{
    int var1;
    char var2;
    {
        int var1;
        char var2;
    }
}
```

Yukarıdaki kodda herhangi bir hata bulunmamaktadır. var1 ve var2 değişkenlerinin ismi ikinci kez içteki blokta tanımlama işleminde kullanılmıştır ama artık bu tanınabilirlik alanında farklılık yaratan ayrı bir bloktur, dolayısıyla bir hata söz konusu değildir.

```
void sample1(void)
{
    int k;
    ...
}

void sample2(void)
{
    int k;
    ...
}

void sample3(void)
{
    int k;
    ...
}
```

Yukarıdaki kodda da bir hata söz konusu değildir. Zira her üç fonksiyonunda da k isimli bir değişken tanımlanmış olsa da bunların tanınabilirlik alanları tamamen birbirinden farklıdır.

### Global Değişkenler (global variables)

C dilinde tüm blokların dışında da değişkenlerin tanımlanabileceğini söylemiştik. İşte bütün blokların dışında tanımlanan değişkenler global değişkenler olarak isimlendirilirler.

Bütün blokların dışı kavramını daha iyi anlamak için bir örnek verelim :

```
#include <stdio.h>

/* bu bölge tüm blokların dışı burada global bir değişken tanımlanabilir. */

int sample1()
{
    ...
}
/* bu bölge tüm blokların dışı burada global bir değişken tanımlanabilir. */

int sample2()
{
    ...
}
/* bu bölge tüm blokların dışı burada global bir değişken tanımlanabilir. */

int sample 3()
{
    ...
}
/* bu bölge tüm blokların dışı burada global bir değişken tanımlanabilir. */

main()
{
    ...
}
/* bu bölge tüm blokların dışı burada global bir değişken tanımlanabilir. */
```

Yorum satırlarının bulunduğu yerler global değişkenlerin tanımlanabileceği yerleri göstermektedir. Bu bölgeler hiçbir fonksiyon içinde değildir. Global değişkenler dosya tanınabilirlik alanı kuralına uyarlar. Yani global değişkenler programın her yerinde ve bütün

fonksiyonların içinde tanınabilirler. Burada bir noktayı gözden kaçırmamak gerekir, bir değişken yerel de olsa global de olsa tanımlaması yapılmadan önce bu değişkene ulaşılamaz. Derleme işleminin bir yönü vardır ve bu yön kaynak kod içerisinde yukarıdan aşağıya doğrudur. Bunu şu şekilde de ifade edebiliriz : Global değişkenler tanımlandıkları noktadan sonra kaynak kod içerisinde her yerde tanınabilirler.

```
#include <stdio.h>

int y;                /* y bir global değişken tanımlandıktan sonra her yerde tanınabilir */

void sample(void)
{
    y = 10;           /* başka bir fonksiyondan da y global değişkenine ulaşılabilir */
}

void main()
{
    y = 20;

    printf("y = %d\n", y);    /* y = 20 */
    sample();
    printf("y = %d\n", y);    /* y = 10 */
}
```

Yukarıdaki örnekte y değişkeni tüm blokların dışında tanımlandığı için (ya da hiçbir fonksiyonun içinde tanımlanmadığı için) global değişkendir. y değişkeninin tanınabilirlik alanı dosya tanınabilirlik alanıdır. yani y değişkeni tanımlandıktan sonra tüm fonksiyonların içinde tanınabilir. Yukarıdaki programın çalışması main fonksiyonundan başlayacaktır. y global değişkenine önce 20 değeri atanmakta ve daha sonra bu değer printf fonksiyonuyla ekrana yazdırılmaktadır. Daha sonra sample fonksiyonu çağırılmıştır. sample fonksiyonu çağırılınca kodun akışı sample fonksiyonuna geçer. sample fonksiyonu içinde de y global değişkeni tanınabilir. sample fonksiyonunda global y değişkenine 10 değeri atanmakta ve daha sonra bu değer yine printf fonksiyonuyla ekrana yazdırılmaktadır.

Peki bir global değişkenle aynı isimli yerel bir değişken olabilir mi? Kesinlikle olabilir! İki değişkenin tanınabilirlik alanları aynı olmadığı için bu durum bir hataya neden olmaz. Aynı isimli hem global hem de yerel bir değişkene ulaşılacak bir noktada, ulaşılan yerel değişken olacaktır, çünkü daha önce de söylediğimiz gibi aynı tanınabilirlik alanında birden fazla aynı isimli değişken olması durumunda o alan içinde en dar tanınabilirlik alanına sahip olanına erişilebilir. Aşağıdaki kodu çok dikkatli inceleyelim :

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

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

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

    g = 200;            /* yerel olan g değişkenine atama yapılıyor */
    printf("yerel g = %d\n", g);    /* yerel g yazdırılıyor. */
    sample();
    printf("yerel g = %d\n", g);    /* yerel g yazdırılıyor. */
    return 0;
}
```

Fonksiyonların kendileri de bütün blokların başlarında tanımlandıklarına göre global nesnelerdir. Gerçekten de fonksiyonlar kaynak kodun her yerinden çağırılabilirler. Aynı tanınabilirlik alanına ilişkin aynı isimli birden fazla değişken olamayacağına göre , aynı isme sahip birden fazla fonksiyon da olamaz. (C++ dilinde aynı isimli fakat farklı parametrik yapıya sahip fonksiyonlar tanımlamak mümkündür.)

Programcıların çoğu global değişkenleri mümkün olduğu kadar az kullanmak ister. Çünkü global değişkenleri kullanan fonksiyonlar başka projelerde kullanılamazlar. Kullanıldıkları projelerde de aynı global değişkenlerin tanımlanmış olması gerekecektir. Dolayısıyla global değişkenlere dayanılarak yazılan fonksiyonların yeniden kullanılabilirliği azalmaktadır.

### Parametre Değişkenleri (formal parameters)

Parametre değişkenleri, fonksiyon parametreleri olarak kullanılan değişkenlerdir. Parametre değişkenleri de blok tanınabilirlik alanı kuralına uyarlar. Yani parametresi oldukları fonksiyonun her yerinde tanınabilirler. Fonksiyon parametre değişkeninin scope'u fonksiyonun ana bloğunun kapanmasıyla sonlanacaktır. Yani fonksiyon parametre değişkeninin tanınabilirlik alanı fonksiyonun ana bloğudur.

```
function(int a, double b)
{
    // ...
    /* *a ve b bu fonksiyonun her yerinde tanınır. */
}
```

Başka bir örnek :

```
sample (int a, int b)
{
    int a;      /* hata aynı tanınırılık alanı içinde ve aynı seviyede aynı isimli değişken */
    ...
    {
        int b;  /* hata değil tanınabilirlik alanları farklı */
        ...
    }
}
```

Bu örnekte fonksiyonun ana bloğunun başında tanımlanmış olan a, "aynı tanınabilirlik alanında aynı blok seviyesinde aynı isimli birden fazla değişken olamaz " kuralına göre geçersizdir. Biri parametre değişkeni biri yerel değişken olmasına karşın, her iki a değişkeni aynı tanınabilirlik alanına sahiptir.

### Nesnelerin Ömürleri (storage duration / lifespan)

Ömür, nesnelerin faaliyet gösterdiği zaman aralığını anlatmak için kullanılan bir kavramdır. Bir kaynak kod içinde tanımlanmış nesnelerin hepsi program çalışmaya başladığında aynı zamanda yaratılmazlar. Ömürleri bakımından nesneleri iki gruba ayırabiliriz :

1. Statik ömürlü nesneler (static objects)
2. Dinamik ömürlü nesneler (dynamic / automatic objects)

### Statik Ömürlü Nesneler (static duration – static storage class)

Statik ömürlü nesneler, programın çalışmaya başlamasıyla yaratılırlar, programın çalışması bitene kadar varlıklarını sürdürürler, yani bellekte yer kaplarlar. Statik nesneler genellikle amaç kod (.obj) içerisine yazılırlar.

C dilinde statik ömürlü 3 nesne grubu vardır :

global değişkenler  
stringler(iki tırnak içerisindeki ifadeler)  
statik yerel değişkenler

stringler ve statik yerel değişkenleri daha sonra inceleyeceğiz.

statik ömürlü nesneler olan global değişkenlerin ömürleri konusunda şunları söyleyebiliriz :  
Global değişkenler programın çalışması süresince yaşayan, yani programın çalışması süresince bellekte yer işgal eden değişkenlerdir.

## Dinamik Ömürlü Nesneler

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

C dilinde dinamik ömürlü üç nense grubu vardır :

yerel değişkenler  
parametre değişkenleri  
dinamik bellek fonksiyonları ile tahsisat yapılarak yaratılmış nesneler

dinamik bellek fonksiyonları ile yaratılmış nesneleri daha sonra inceleyeceğiz.

Yerel değişkenler ve parametre değişkenleri dinamik ömürlü nesnelerdir. Tanımlandıkları bloğun çalışması başladığında yaratılırlar, bloğun çalışması bitince yok olurlar (ömürleri sona erer.) Tanınabilirlik alanları kendi bloklarının uzunluğu kadar olan yerel değişkenlerin ömürleri, program akışının bu bloğa gelmesiyle başlar ve bloğun çalışma süresi bitince de sona erer .  
Örnek :

```
{
    double x; /* programın akışı bu noktaya geldiğinde bellekte x için bir yer ayrılıyor. ve
               blok içinde a yaşamaya devam ediyor.*/
    ...
} /* programın akışı bloğun sonuna geldiğinde a değişkeninin de
   ömrü sona eriyor */
```

Fonksiyonların parametre değişkenleri de benzer biçimde fonksiyon çağırıldığında yaratılırlar, fonksiyon icrası boyunca yaşarlar, fonksiyonun icrası bitince yok olurlar.

Statik değişkenlerle dinamik değişkenler arasında ilk değer verme(initialization) açısından da fark bulunmaktadır. Statik olan global değişkenlere de yerel değişkenlerde olduğu gibi ilk değer verilebilir.

İlk değer verilmemiş ya da bir atama yapılmamış bir yerel değişkenin içinde rasgele bir değer bulunur. Bu değer o an bellekte o değişken için ayrılmış yerde bulunan rasgele bir sayıdır. (garbage value). Oysa ilk değer verilmemiş, ya da bir atama yapılmamış global değişkenler içinde her zaman 0 değeri vardır. Yani bu değişkenler 0 değeriyle başlatılırlar.

Aşağıdaki kısa programı derleyerek çalıştırınız :

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

```
int globx;
```

```
int main()
```

```
{
    int localy;
```



```

printf("globx = %d\n", globx);
printf("localy = %d\n", localy);

return 0;
}

```

Yerel değişkenler ile global değişkenler arasındaki başka bir fark da, global değişkenlere ancak sabit ifadeleriyle ilk değer verilebilmesidir. Global değişkenlere ilk değer verme işleminde kullanılan ifade (initializer), değişkenler ya da fonksiyon çağırma ifadeleri kullanılamazlar, ifade yalnızca sabitlerden oluşmak zorundadır. (C++ dilinde böyle bir zorunluluk bulunmamaktadır.)

Ancak yerel değişkenlere ilk değer verilme işleminde böyle bir kısıtlama bulunmamaktadır. Örnek :

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

```

int x = 5;           /* legal */
int y = x + 5;       /* error! ilk değer verme işleminde sabit ifadesi kullanılmamış */
int z = funk();      /* error! ilk değer verme işleminde fonksiyon çağırma ifadesi var */

```

```

int main()
{
    int a = b;
    int k = b - 2;    /* legal, k yerel değişken olduğu için ilk değer verme ifadesi değişken
çğerebilir */
    int l = funk(); /* legal, k yerel değişken olduğu için ilk değer verme ifadesinde fonksiyon
çğırma i          fadesi bulunabilir. */
    ...
}

```

```

int funk()
{
    ...
}

```

### Argumanların Parametre Değişkenlerine Kopyalanması

C dilinde bir fonksiyonun parametre değişkenlerinin tanımlandıkları fonksiyon içinde her yerde tanınabildiklerini söylemiştik. Peki bir fonksiyonun parametre değişkenleri ne işe yararlar?

Bir fonksiyonun parametre değişkenleri, o fonksiyonun çağırılma ifadesiyle kendisine gönderilen argumanları tutacak olan yerel değişkenlerdir. Örnek :

```

fonk(int a)
{
    ....
}

main()
{
    int x;
    ....
    fonk (x);
}

```

Yukarıdaki örnekte fonk fonksiyonu x argumanı ile çağırıldığında, programın akışı fonk fonksiyonunun kodunun bulunduğu yere sıçrar, fonk fonksiyonundaki a isimli parametre değişkeni için bellekte bir yer ayrılır ve a parametre değişkenine x değişkeninin değeri atanır. Yani

```
a = x;
```

işleminin otomatik olarak yapıldığını söyleyebiliriz.

Başka bir örnek verelim :

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

```
main()
```

```
{
```

```
    int x = 100, y = 200, z;
```

```
    z = add(x, y);
```

```
    printf("%d\n", z);
```

```
    return 0;
```

```
}
```

```
int add(int a, int b)
```

```
{
```

```
    return a + b;
```

```
}
```

add fonksiyonu çağırıldığında programın akışı bu fonksiyona geçmeden önce, x ve y değişkenlerinin içinde bulunan değerler add fonksiyonunun parametre değişkenleri olan a ve b'ye kopyalanırlar.

## 8 . BÖLÜM : OPERATÖRLER

Operatörler nesneler veya sabitler üzerinde önceden tanımlanmış birtakım işlemleri yapan atomlardır. Operatörler mikroişlemcinin bir işlem yapmasına neden olurlar ve bu işlem sonunda da bir değer üretilmesini sağlarlar. Programlama dillerinde tanımlanmış olan her bir operatör en az bir makine komutuna karşılık gelmektedir.

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

C programlama dilinde her ifade en az bir operatör içerir. (Sabitler, nesneler ve operatörlerin kombinezonlarına ifade denir.)

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

Her programlama dilinde operatörlerin birbirlerine göre önceliği söz konusudur. (Eğer öncelik kavramı söz konusu olmasaydı, operatörlerin neden olacağı işlemlerin sonuçları makinadan makineye, derleyiciden derleyiciye farklı olurdu.)

C' de toplam 45 operatör vardır, ve bu operatörler 15 ayrı öncelik seviyesinde yer alır. (bakınız operatör öncelik tablosu) . Bir öncelik seviyesinde eğer birden fazla operatör varsa bu operatörlerin soldan sağa mı sağdan sola mı öncelikle dikkate alınacağı da tanımlanmalıdır. Buna öncelik yönü diyebiliriz. (associativity)

Bir sembol, birden fazla operatör olarak kullanılabilir. Örneğin \* operatörü hem çarpma hem de içerik alma operatörü (göstericilerle ilgili bir operatör) olarak kullanılır. Yine bir C dilinde &(ampersand) hem bitisel ve hem de göstericilere ilişkin adres operatörü olarak kullanılmaktadır. Operatör öncelik listesinde gördüğümüz operatörlerin çoğunu bu ders tanıyacağız ama bir kısmı ileriki derslerimizde yer alacak. (örneğin gösterici operatörleri)

### Operatör Terminolojisi

Hem operatörler konusunun daha iyi anlaşılması hem de ortak bir dil oluşturmak amacıyla operatörler hakkında kullanabileceğimiz terimleri inceleyelim.

#### Operand

Operatörlerin değer üzerinde işlem yaptıkları nesne veya sabitlere denir. C'de operatörleri aldıkları operand sayısına göre 3 gruba ayırabiliriz.

1. Tek Operand Alan Operatörler (unary operators)  
Örneğin ++ ve -- operatörleri unary operatörlerdir. (C dilinde unary operatörler operatöröncelik tablosunun 2. seviyesinde bulunurlar.)
2. İki Operand Alan Operatörler. (binary operators)  
Aritmetik işlem operatörü olan + ve / operatörlerini örnek olarak verebiliriz.
3. Üç Operand Alan Operatör (ternary operator)  
Çoğunlukla kullanmıyoruz çünkü C'de 3 opererand olan tek bir operatör vardır. (koşul operatörü - conditional operator)

Operatörleri operandının ya operandlarının neresinde bulunduklarına göre de gruplayabiliriz:

1. Sonek Konumundaki Operatörler (postfix operators)  
Bu tip operatörler operandlarının arkasına getirilirler. Örneğin sonek ++ operatörü (x++)

2. Önek Konumundaki Operatörler (prefix operators)  
Bu tip operatörler operandlarının önüne getirilirler.  
Örneğin önek ++ operatörü (++x)

### C Dilinin Operatör Öncelik Tablosu

Seviye	Operatör	Tanım	Öncelik Yönü (associativity)
1	( ) [ ] . ->	öncelik kazandırma ve fonksiyon çağırma index operatörü (subscript) yapı elemanına ulaşım (structure access) yapı elemanına gösterici ile ulaşım	soldan sağa
2	+ - ++ -- ~ ! * & sizeof (tür)	işaret operatörü (unary) işaret operatörü (unary) 1 artırma (increment) 1 eksiltme (decrement) bitisel değil (bitwise not) mantıksal değil (logical not) içerik operatörü (indirection) adres operatörü (address of) sizeof operatörü tür dönüştürme (type cast operator)	sağdan sola
3	* / %	çarpma (multiplication) bölme (division) modulus (bölümünden kalan)	soldan sağa
4	+ -	toplama (addition) çıkarma (subtraction)	soldan sağa
5	<< >>	bitisel sola kaydırma (bitwise shift left) bitisel sağa kaydırma (bitwise shift right)	soldan sağa
6	< > <= >=	küçüktür (less than) büyüktür (greater than) küçük eşittir (less than or equal) büyük eşittir (greater than or equal)	soldan sağa
7	== !=	eşittir (equal) eşit değildir (not equal to)	soldan sağa
8	&	bitisel VE (bitwise AND)	soldan sağa
9	^	bitisel EXOR (bitwise EXOR)	soldan sağa
10		bitisel VEYA (bitwise OR)	soldan sağa
11	&&	mantıksal VE (logical AND)	soldan sağa
12		mantıksal VEYA (logical OR)	soldan sağa
13	?:	koşul operatörü (conditional operator)	sağdan sola
14	= += -= *= /= %= <<= >>= &=  = ^=	atama (assignment) işlemlili atama (assignment addition) işlemlili atama (assignment subtraction) işlemlili atama (assignment multiplication) işlemlili atama (assignment division) işlemlili atama (assignment modulus) işlemlili atama (assignment shift left) işlemlili atama (assignment shift right) işlemlili atama (assignment bitwise AND) işlemlili atama (assignment bitwise OR) işlemlili atama (assignment bitwise EXOR)	sağdan sola
15	,	virgül operatörü (comma)	

3. Araek Konumundaki Operatörler (infix operators)  
Bu tip operatörler operandlarının aralarına getirilirler.  
Örneğin aritmetiksel toplama operatörü (x + y)

Son olarak operatörleri yaptıkları işlere göre sınıflayalım :

1. Aritmetik operatörler (arithmetic operators)
2. İlişkisel operatörler (relational operators)
3. Mantıksal operatörler (logical operators)
4. Gösterici operatörleri (pointer operators)
5. Bitsel işlem yapan operatörler (bitwise operators)
6. Özel amaçlı operatörler (special purpose operators)

İlk 3 grup programlama dillerinin hepsinde vardır. Bitsel işlem yapan operatörler ve gösterici operatörleri yüksek seviyeli programla dillerinde genellikle bulunmazlar. Programlama dillerinden çoğu, kendi uygulama alanlarında kolaylık sağlayacak birtakım özel amaçlı operatörlere de sahip olabilir.

### Operatörlerin Değer Üretmeleri

Operatörlerin en önemli özellikleri bir değer üretmeleridir. Operatörlerin ana işlevi bir değer üretmeleridir. Programcı, bir ifade içinde operatörlerin ürettiği değeri kullanır ya da kullanmaz. (discard edebiliriz.) Operatörlerin ürettiği değeri nasıl kullanabiliriz:

Üretilen değeri başka bir değişkene aktararak kullanabiliriz.

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

Yukarıdaki örnekte  $y + z$  ifadesinin değer, yani  $+$  operatörünün ürettiği değer  $x$  değişkenine aktarılmaktadır.

Üretilen değeri başka bir fonksiyona arguman olarak gönderebiliriz.

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

Yukarıdaki örnekte `func` fonksiyonuna arguman olarak  $y + z$  ifadesinin değeri gönderilmektedir. (Yani  $+$  operatörünün ürettiği değer)

Üretilen değeri **return** anahtar sözcüğü ile fonksiyonların geri dönüş değerlerinin belirlenmesinde kullanabiliriz.

```
int sample(void)
{
    ...
    return (y + z)
}
```

yukarıda `sample` fonksiyonunun geri dönüş değeri  $y + z$  ifadesinin değeridir. (Yani  $+$  operatörünün ürettiği değer)

Operatörlerin elde ettiği değer hiç kullanılmaması C sentaksı açısından bir hataya neden olmaz. Ancak böyle durumlarda derleyiciler bir uyarı mesajı vererek programcıyı uyarırlar. Örneğin :

```
int x = 20;
int y = 10;
...
x + y;
```

Yukarıdaki ifadede  $+$  operatörü bir değer üretir. ( $+$  operatörünün ürettiği değer operandlarının toplamı değeri, yani 20 değeridir.) Ancak bu değer hiçbir şekilde kullanılmamıştır. Derleyici böyle bir işlemin büyük ihtimalle yanlışlıkla yapıldığını düşünecek ve bir uyarı mesajı verecektir. Borland derleyicilerinde verilen uyarı mesajı şu şekildedir :

warning : "code has no effect!"

Daha önce belirttiğimiz gibi C dilinde ifadelerin türleri ve değerleri söz konusudur. Bir ifadenin değerini derleyici şu şekilde tespit eder : İfade içindeki operatörler öncelik sıralarına göre değer üretirler, ve ürettikleri değerler, ifade içindeki önceliği daha az olan operatörlere operand olarak aktarılır. Bu işlemin sonunda tek bir değer elde edilir ki bu da ifadenin değeridir.

```
int x = 10;
```

```
int y = 3;
```

```
printf("%d\n", x % y / 2 + 7 && !x || y);
```

Yukarıdaki kod parçasında printf fonksiyonu çağırımıyla `x % y / 2 + 7 && !x || y` ifadesinin değeri yazdırılmaktadır. Yazdırılacak değer nedir? ifade içindeki operatörler sırayla değer üretecek ve üretilen değerler öncelik sırasına göre, değer üretecek operatörlere operand olacaktır. En son kalan değer ise ifadenin değeri, yani ekrana yazdırılan değer olacaktır.

### Operatörlerin Yan Etkileri (side effect of operators)

C dilinde operatörlerin ana işlevleri, bir değer üretmeleridir. Ancak bazı operatörler operandı olan nesnelerin değerlerini değiştirirler. Yani bu nesnelerin bellekteki yerlerine yeni bir değer yazılmasına neden olurlar. Bir operatörün kendilerine operand olan nesnenin değerlerini değiştirmesine operatörün yan etkisi diyeceğiz. Yan etki, bellekte yapılan eğer değişikliği olarak tanımlanmaktadır. Örneğin atama operatörünün, ++ ve -- operatörlerinin yan etkileri söz konusudur.

### Operatörler Üzerindeki Kısıtlamalar

Programlama dilinin tasarımında, bazı operatörlerin kullanılmalarıyla ilgili birtakım kısıtlamalar söz konusu olabilir. Örneğin ++ operatörünün kullanımında, operandın nesne gösteren bir ifade olması gibi bir kısıtlama söz konusudur. Eğer operand olan ifade bir nesne göstermiyorsa, yani sol taraf değeri değilse, derleme zamanında hata oluşacaktır.

Kısıtlama operatörün operand ya da operandlarının türleriyle de ilgili olabilir. Örneğin % operatörünün operandlarının tamsayı türlerinden birine ait olması gerekmektedir. % operatörünün operandları gerçek sayı türlerinden birine ait olamaz. Operandın gerçek sayı türlerinden birinden olması durumunda derleme zamanında hata oluşacaktır.

### Operatörlerin Detaylı Olarak İncelenmesi

#### Aritmetik Operatörler

Aritmetik operatörler dört işlemlerle ilgili işlem yapan operatörlerdir.

#### + ve - Operatörleri. (toplama ve çıkarma operatörleri)

İki operand alan aritmetik operatörlerdir. (binary infix) Diğer bütün programlama dillerinde oldukları gibi operandlarının toplamını ve farkını almak için kullanılırlar. Yani ürettikleri değer, operandlarının toplamı ya da farkı değerleridir. Operandları üzerinde herhangi bir kısıtlama yoktur. Öncelik tablosundan da görüleceği gibi genel öncelik tablosunun 4. sırasında yer alırlar ve öncelik yönleri soldan sağdır (left associative). Yan etkileri yoktur, yani operandlarının bellekte sahip oldukları değerleri değiştirmezler.

Toplama ve çıkarma operatörleri olan + ve - operatörlerini tek operand alan + ve - operatörleriyle karıştırmamak gerekir.

#### İşaret Operatörü Olan - ve + Operatörleri

Bu operatörler unary prefix (tek operand alan önek konumundaki) operatörlerdir. - operatörü operandı olan ifadenin değerinin ters işaretlisini üretir. Unary prefix - operatörünün ürettiği bir nesne değil bir değerdir. Aşağıdaki ifade matematiksel olarak doğru olmasına karşın C dili açısından hatalıdır, derleme zamanında hata (error) oluşumuna neden olur:

```
int x;
```

```
-x = 5;
```

x bir nesne olmasına karşı  $-x$  ifadesi bir nesne değil, x nesnesinin değerinin ters işaretlisi olan değerdir. Dolayısıyla bir sol taraf değeri değildir ve bu ifadeye bir atama yapılamaz.

+ operatörü yalnızca sentaks açısından C diline ilave edilmiş bir operatördür. Unary prefix bir operand olarak derleyici tarafından ele alınır. Operandı olan ifade üzerinde herhangi bir etkisi olmaz.

Unary prefix olan - ve + operatörleri operatör öncelik tablosunun 2. seviyesinde bulunurlar ve öncelik yönleri sağdan sola doğrudur. Aşağıdaki ifadeyi ele alalım :

```
int x = 5;
int y = -2;
```

```
-x - -y;
```

Yukarıdaki ifadede 3 operatör bulunmaktadır. Sırasıyla unary prefix - operatörü, binary infix çıkarma operatörü ve yine unary prefix - operatörü. İfadenin değerinin hesaplanmasında operatör önceliklerine göre hareket edilecektir. unary prefix olan - operatörü 2. öncelik seviyesinde olduğu için binary infix olan çıkarma operatörüne göre daha yüksek önceliklidir, ve kendi içinde öncelik yönü sağdan soladır. Önce -y ifadesi değer üretir. Üretilen değer 2 olacaktır. daha sonra -x ifadesi değer üretir. Üretilen değer -5 olacaktır. üretilen bu değerler çıkarma operatörüne operand olurlar ve ana ifadenin değeri  $-5 - 2$  yani  $-7$  değeri olarak hesaplanır.

### \* ve / Operatörleri

İki operand alan aritmetik operatörlerdir. (binary infix) \* operatörü operandları çarpmak / ise operandları bölmek amacıyla kullanılır. Yani üretilen değer operandlarının çarpım ya da bölüm değerleridir. Operandları herhangi bir türden olabilir. Öncelik tablosunda görülebileceği gibi, genel öncelik tablosunun 3. sırasında bulunurlar. Öncelik yönleri soldan sağdır. Her iki operatörün de bir yan etkisi yoktur.

/ operatörünün kullanımında dikkatli olmak gerekir. Eğer her iki operand da tamsayı türlerindense operatörün bulunduğu ifadenin türü de aynı tamsayı türünden olacaktır. (Bu konuya ileride detaylı değineceğiz.)

C programlama dilinde \* sembolü aynı zamanda bir gösterici operatörü olarak da kullanılır. Ama aynı sembol kullanılmasına karşın bu iki operatör hiçbir zaman birbirine karışmaz çünkü aritmetik çarpma operatörü olarak kullanıldığında binary infix durumundadır, ama gösterici operatörü olarak kullanıldığında unary prefix dir.

### % (modulus) Operatörü

İki operand alan aritmetik bir operatördür. (binary infix) Operatörlerin her ikisinin de türü de tamsayı türlerinden (**char, short, int, long**) biri olmak zorundadır. (Değilse bu durum derleme zamanında error ile neticelenir.) Ürettiği değer sol tarafındaki operandın sağ tarafındaki operanda bölümünden kalanıdır. Operatörün yan etkisi yoktur. Örneğin:

```
c = 15 % 4; /* burada c'ye 3 değeri atanacaktır */
```

```
int c = 13 - 3 * 4 + 8 / 3 - 5 % 2;
```

Burada c'ye 2 değeri atanacaktır. Çünkü işlem şu şekilde yapılmaktadır:

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

### Artırma ++ ve Eksiltme -- Operatörleri (increment and decrement operators)

Artırma (++) ve eksiltme (--) operatörleri C dilinde en çok kullanılan operatörlerdendir. Tek operand alırlar. Önek ya da sonek durumunda bulunabilirler. ++ operatörü operandı olan değişkenin içeriğini 1 artırmak -- operatörü de operandı olan değişkenin içeriğini 1 eksiltmek için kullanılır. Dolayısıyla yan etkileri söz konusudur. Operandları olan nesnenin bellekteki değerini değiştirirler. Bu iki operatör de diğer aritmetik operatörlerden daha yüksek önceliğe sahiptir. Operatör öncelik tablosunun 2. seviyesinde bulunurlar ve öncelik yönleri sağdan soladır. (right associative)

Yalın olarak kullanıldıklarında (başka hiçbir operatör olmaksızın) önek ya da sonek durumları arasında hiçbir fark yoktur. ++ bir artır ve -- bir eksilt anlamına gelir.

++c ve c++ ifadeleri tamamen birbirine denk olup  $c = c + 1$  anlamına gelirler.

--c ve c-- ifadeleri tamamen birbirine denk olup  $c = c - 1$  anlamına gelirler.

Diğer operatörlerle birlikte kullanıldıklarında (örneğin atama operatörleriyle) önek ve sonek biçimleri arasında farklılık vardır :

Önek durumunda kullanıldığında operatörün ürettiği değer artırma/eksiltme yapıldıktan sonraki değerdir. Yani operandın artırılmış ya da azaltılmış değeridir. Sonek durumunda ise operatörün ürettiği değer artırma / eksiltme yapılmadan önceki değerdir. Yani operandı olan nesnenin artırılmamış ya da azaltılmamış değeridir. Operandın değeri ifadenin tümü değerlendirildikten sonra artırılacak ya da eksiltilecektir.

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

Bu durumda:

++x  $\Rightarrow$  11  
ve y = 11 değeri atanır..

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

y = 10;  
++x  $\Rightarrow$  11

Aşağıdaki programı inceleyelim:

```
int main()
{
    int a, b;
    a = 10;
    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 ifadesi ekrana 11 11 yazdırırken ikinci printf ifadesi 11 10 yazdırır.

Bir örnek :

```
x = 10;
y = 5;
z = x++ % 4 * --y;
```



```
--y ⇒ 4
10 % 4 ⇒ 2
2 * 4 ⇒ 8
z = 8
x++ ⇒ 11
```

Bir örnek daha:

```
c = 20;
d = 10;
a = b = d + c--;
```

İşlem sırası:

```
10 + 20 ⇒ 30
b = 30
a = 30
c-- ⇒ 19
```

İfadenin sonunda değişkenler:

a = 30, b = 30, c = 19, d = 10 değerlerini alırlar.

Fonksiyon çağırma ifadelerinde bir değişken postfix ++ ya da – operatörü ile kullanılmışsa, ilk önce fonksiyon artırılmamış veya eksiltilmemiş değişken ile çağırılır, fonksiyonun çalışması bitince değişken 1 artırılır veya eksiltir.

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

```
int func(int x)
{
    printf("%d\n", x);
}
```

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

    return 0;
}
```

### **++ ve -- Operatörleriyle İlgili Şüpheli Kodlar (undefined behaviours)**

++ ve – operatörlerinin bilinçsiz bazı kullanımları derleyiciler arasında yorum farklılığına yol açarak taşınabilirliği bozarlar. Böyle kodlardan sakınmak gerekir.

3 tane + (++++) ya da 3 tane – (---) karakteri boşluk olmaksızın yan yana getirilmemelidir.

```
x = a++++b; /* şüpheli kod */
```

Burada derleyiciler  $x = a++ + b$ ; işlemlerini yapabileceği gibi  $x = a + ++b$ ; işlemlerini de yapabilirler.

Bir ifadede bir değişken ++ ya da – operatörleriyle kullanılmışsa, o değişken o ifadede bir kez daha yer almamalıdır. Örneğin aşağıdaki ifadelerin hepsi şüpheli kodlardır :

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

```

y = ++x + ++x;    /* şüpheli kod */
y = ++x + x       /* şüpheli kod */
a = ++a;          /* şüpheli kod */

```

Bir fonksiyon çağırılırken parametrelerden birinde bir değişken ++ ya da - ile kullanılmışsa bu değişken diğer parametrelerde gözükmemelidir.

Argumanların parametre değişkenlerine kopyalanma sırası standart bir biçimde belirlenmemiştir. Bu kopyalama işlemi bazı sistemlerde soldan sağa bazı sistemlerde ise sağdan soladır. örnek :

```
int a = 10;
```

```
fonk (a, a++);    /* şüpheli kod fonk fonksiyonuna 10, 10 değerleri mi 11, 10 değerleri mi
                  önderiliyor? */
```

```

void fonk(x, y)
{
...
}

```

### Karşılaştırma Operatörleri (ilişkisel operatörler)

C programlama dilinde toplam 6 tane karşılaştırma operatörü vardır:

<	küçüktür (less than)
>	büyüktür (greater than)
<=	küçük eşit (less than or equal)
>=	büyük eşit (greater than or equal)
==	eşittir. (equal)
!=	eşit değildir. (not equal)

Bu operatörlerin hepsi iki operand alan aritmetik operatörlerdir. (binary infix)

Operatör öncelik tablosuna bakıldığında, karşılaştırma operatörlerinin aritmetik operatörlerden daha sonra geldiği yani daha düşük öncelikli oldukları görülür. Karşılaştırma operatörleri kendi aralarında iki öncelik grubu biçiminde bulunurlar.

Karşılaştırma operatörleri bir önerme oluştururlar ve sonuç ya doğru ya da yanlış olur. Karşılaştırma operatörlerinden elde edilen değer , önerme doğru ise 1 önerme yanlış ise 0 değeridir ve **int** türündendir. Bu operatörlerin ürettiği değerler de tıpkı aritmetik operatörlerin ürettiği değerler gibi kullanılabilirler. örnekler :

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

```

int main()
{
    int x = 5;

    ...
    y = (x > 5) + 1;
    printf("%d\n", y);
    funk(y >= x);

    return 0;
}

```

Yukarıdaki örnekte ekrana 1 değeri yazdırılacaktır. Daha sonra da funk fonksiyonu 0 değeriyle çağırılacaktır.

Bazı programlama dillerinde  $x = (a < b) + 1$ ; gibi bir işlem hata ile sonuçlanır. Çünkü örneğin Pascal dilinde  $a < b$  ifadesinden elde edilen değer True ya da False dir. (yani BOOL ya da BOOLEAN türündendir.) Ama C doğal bir dil olduğu için karşılaştırma operatörlerinin ürettikleri değeri Boolean türü ile kısıtlamamıştır. C’de mantıksal veri türü yerine **int** türü kullanılır. Mantıksal bir veri türünün tamsayı türüyle aynı olması C’ye esneklik ve doğallık kazandırmıştır. Başka bir örnek :

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

Yukarıdaki deyim C dili için son derece doğal ve okunabilir bir deyimdir. Bu deyimin icrasıyla x değişkenine ya 1 ya da 0 değeri atanacaktır.

Karşılaştırma operatörünün kullanılmasında bazı durumlara dikkat etmemiz gerekmektedir. Örneğin

```
int x = 12;
```

```
5 > x > 9
```

Yukarıdaki ifade matematiksel açıdan doğru değildir. Çünkü 12 değeri 5 ve 9 değerlerinin arasında değildir. Ancak ifade C açısından doğru olarak değerlendirilir. Çünkü aynı seviyede olan iki operatöre (> ve >) ilişkin öncelik yönü soldan sağdır. Önce soldaki > operatörü değer üretecek ve ürettiği değer olan 0 sağdaki > operatörüne operand olacaktır. Bu durumda  $0 > 9$  ifadesi elde edilecek ve bu ifadeden de 0 yani yanlış değeri elde edilecektir.

### Mantıksal Operatörler (logical operators)

Bu operatörler operandları üzerinde mantıksal işlem yaparlar. Operandlarını Doğru (true) ya da Yanlış(false) olarak yorumladıktan sonra işleme sokarlar. C’de dilinde öncelikleri farklı seviyede olan (tabloya bakınız) üç mantıksal operatör vardır:

! değil operatörü (not)

&& Ve operatörü (and)

|| veya operatörü (or)

Tablodan da görüleceği gibi && ya da || operatörlerin aynı ifade içinde birden fazla kullanılması durumunda öncelik yönü soldan sağdır, 2. seviyede bulunan ! operatörü için ise öncelik yönü sağdan soladır.

C’de mantıksal veri türü yoktur. Mantıksal veri türü olmadığı için bunun yerine **int** türü kullanılır ve mantıksal doğru olarak 1 mantıksal yanlış olarak da 0 değeri kullanılır.

C dilinde herhangi bir ifade mantıksal operatörlerin operandı 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)

```
int x = 5;
```

```
x * x - 25
```

İfadesi mantıksal bir operatörün operandı olduğu zaman yanlış olarak yorumlanacaktır. Çünkü sayısal değeri 0’a eşittir.

### ! Operatörü

Değil operatörü tek operand alır ve her zaman önek durumundadır. (unary prefix) ! operatörü operandının mantıksal değerini değiştirir. Yani Doğru değerini yanlış değerini, yanlış değerini de Doğru değerine dönüştürür:

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 operatörü en düşük öncelikli operatördür)
```

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

İşlem sırası:

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

### && (ve / and) Operatörü

Bu operatör ilişkisel operatörlerin hepsinden düşük, || (veya / or) operatöründen yüksek önceliklidir. Operandlarının ikisi de Doğru ise Doğru (1), operandlardan bir tanesi yanlış ise yanlış (0) değerini üretir.

x	y	x&& y
yanlış	yanlış	yanlış
yanlış	Doğru	yanlış
Doğru	yanlış	yanlış
Doğru	Doğru	Doğru

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

&& operatörünün önce sol tarafındaki işlemler öncelik sırasına göre tam olarak yapılır. Eğer bu işlemlerde elde edilen sayısal değer 0 ise && operatörünün 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 operatörün sağ tarafı hiç icra edilmeyecek dolayısıyla da sqrt fonksiyonu çağırılmayacaktır. Dolayısıyla b değişkenine 0 değeri atanacaktır.

Uygulamalarda mantıksal operatörler çoğunlukla ilişkisel (karşılaştırma) operatörleriyle birlikte kullanılırlar :

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

Bu durumda y değişkenine ya 1 ya da 0 değeri atanacaktır. Eğer x 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 atanacaktır.

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

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

## || (veya / and) Operatörü

Önceliği en düşük olan mantıksal operatördür. İki operandından biri Doğru ise Doğru değerini üretir. İki operandı da yanlış ise yanlış değerini üretir.

x	y	x  y
yanlış	yanlış	yanlış
yanlış	Doğru	Doğru
Doğru	yanlış	Doğru
Doğru	Doğru	Doğru

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

|| operatörünün önce sol tarafındaki işlemler öncelik sırasına göre tam olarak yapılır. Eğer bu işlemlerden elde edilen sayısal değer 1 ise sağ tarafındaki işlemler yapılmaz. Örnekler:

```
a = 20;
b = 10;
y = a + b >= 20 || a - b <= 10;
```

```
a + b => 30
30 >= 20 => 1
```

Artık sağ taraftaki işlemlerin yapılmasına gerek kalmadan y değişkenine 1 değeri atanır.

Aşağıdaki ifadede ch değişkeninin içinde bulunan karakterin (Türkçe karakterler de dahil olmak üzere) küçük harf olup olmadığı bulunur:

```
ch = 'ç';
```

```
a = ch >= 'a' && ch <= 'z' || ch == 'ı' || ch == 'ü' || ch == 'ğ' || ch == 'ç' || ch == 'ö' || ch == 'ş';
```

Son olarak şunu da ilave edelim, mantıksal operatörler bir değer üretebilmek için operandlarını önce 1 ya da 0 (yani DOĞRU ya da YANLIŞ) olarak yorumlarlar, ama yan etkileri yoktur. Yani operandlarının nesne olması durumunda bu nesnelerin bellekteki değerlerini 1 ya da 0 olarak değiştirmezler.

## Özel Amaçlı Operatörler

Bu bölümde özel amaçlı operatörlerden atama, işlemli atama, öncelik ve virgül operatörlerini inceleyeceğiz.

### Atama Operatörü (assignment operator)

Atama operatörü C dilinde öncelik tablosunun en alttan ikinci seviyesinde bulunur ve yalnızca virgül operatöründen daha yüksek önceliklidir. Her operatör bir değer üretir ve atama operatörü de yaptığı atama işleminin yanısıra bir değer üretir. Atama operatörünün ürettiği değer sağ taraf değerinin kendisidir. Örneğin:

```
if (a = 20) {
    ...
}
```

Burada sırasıyla şunlar olur :

1. 20 sayısı x değişkenine atanır.
2. Bu işlemden 20 sayısı üretilir.
3. 20 sıfır dışı bir sayı olduğu için if deyimi doğru olarak değerlendirilir.

Atama operatörünün ürettiği değer nesne değildir. dolayısıyla;

$(b = c) = a;$  /\* ifadesi C’de geçerli bir deyim değildir. \*/

$b = c$  atamasından elde edilen değer c nesnesinin kendisi değil c nesnesinin sayısal değeridir.

Operatör tablosundan da göreceğimiz gibi atama operatörü kendi arasında sağdan sola önceliklidir. (right associative) Bu yüzden :

$var1 = var2 = var3 = 0;$  deyimi C’de geçerlidir. Bu deyimde önce var3 değişkenine 0 değeri atanacak ve üretilen 0 değeri var2 değişkenine atanacak ve yine üretilen 0 değeri son olarak var1 değişkenine atanacaktır.

### İşlemli Atama Operatörleri

Bir işlemin operandı ile işlem sonucunda üretilen değerın atanacağı nesne aynı ise işlemli atama operatörleri kullanılır.

$\langle nesne1 \rangle = \langle nesne1 \rangle \text{ işlem } \langle operand2 \rangle$  ile  
 $\langle nesne1 \rangle \text{ işlem} = \langle operand2 \rangle$  aynı anlamdadır.

İşlemli atama operatörleri atama operatörüyle sağdan sola eşit öncelike sahiptir. (operatör öncelik tablosuna bakınız)

İşlemli atama operatörleri hem okunabilirlik hem de daha kısa yazım için tercih edilirler. 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.

```
var1 = 3;
var2 = 5;
var1 += var2 *= 3; ifadesinde var2 = var2 * 3 işlemiyle önce var2 değişkenine 15 değeri
atanacak ve işlem sonucunda 15 değeri üretilecek var1 = var1 + 15 işlemiyle var1 değişkenine
18 değeri atanacaktır.
```

Özellikle += ve -= operatörlerinin yanlış yazılması, tespit edilmesi zor hatalara neden olabilir.

```
x += 5;
```

ifadesi x değişkeninin eski değerini 5 artırırken bu deyim yanlışlıkla aşağıdaki gibi yazıldığı takdirde

```
x =+ 5;
```

x değişkenine 5 değeri atanacaktır. Çünkü burada iki ayrı operatör söz konusudur. (atama operatörü olan = ve işaret operatörü olan +)

yukarıdaki örneklerden de görüldüğü gibi atama grubu operatörlerinin yan etkileri vardır. Yan etkileri operatörün sol operandının bellekteki değerinin değiştirilmesi (operatörün sağ tarafındaki operandı olan ifadenin değerinin sol tarafındaki nesneye aktarılması) şeklinde kendini gösterir.

### Virgül (,) Operatörü

İki ayrı ifadeyi tek bir ifade olarak birleştiren virgül operatörü C'nin en düşük öncelikli operatörüdür.

```
ifade1;  
ifade2;
```

ile

```
ifade1, ifade2;
```

aynı işleve sahiptir.

Virgül operatörünün önce sol tarafındaki ifade sonra sağ tarafındaki ifade tam olarak yapılır. Bu operatörün ürettiği değer sağ tarafındaki ifadenin ürettiği değerdir. Virgülün sol tarafındaki ifadenin ürettiği değer virgül operatörünün ürettiği değere bir etkisi yoktur. Örneğin :

```
x = (y *= 5, z = 100);
```

ifadesinde x değişkenine 100 değeri atanacaktır.

Aşağıdaki örnekte **if** değeri yanlış olarak değerlendirilecektir :

```
if (a =10, b = 0) {  
    ...  
}
```

Virgül operatörleri ile birden fazla ifade tek bir ifade olarak birleştirilebilir. Örneğin :

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

yerine

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

yazılabilir.

### Öncelik Operatörü ( )

Öncelik operatörü bir ifadenin önceliğini yükseltmek amacıyla kullanılmaktadır.

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

Öncelik operatörünü C nin en yüksek öncelikli operatörler grubundadır. Öncelik operatörü de kendi arasında soldan sağa öncelik kuralına uyar. Örneğin:

$a = (x + 2) / ((y + 3) * (z + 2) - 1);$

ifadesinde işlem sırası şöyledir :

i1 :  $x + 2$   
i2 :  $y + 3$   
i3 :  $z + 2$   
i4 :  $i2 * i3$   
i5 :  $i4 - 1$   
i6 :  $i1 / i5$   
i7 :  $a = i6$



## 9 . BÖLÜM : if DEYİMİ

C dilinde program akışını kontrol etmeye yönelik en önemli deyim **if** deyimidir. **if** deyiminin genel biçimi aşağıda verilmiştir:

```
if (ifade)
    deyim1;
else
    deyim2;
```

deyim1 ve deyim2 yalın deyim (simple statement) olabileceği gibi, bileşik bir deyim (compound statement) ya da başka bir kontrol deyimi de (control statement) olabilir.

**if** parantezi içindeki ifadeye kontrol ifadesi (control expression) denir.

**if** deyiminin icrası aşağıdaki gibi yapılır:

Derleyici önce kontrol ifadesinin sayısal değerini hesaplar. Hesapladığı sayısal değeri mantıksal doğru ya da yanlış olarak yorumlar. İfadenin hesaplanan değeri 0 ise yanlış, 0 dışında bir değer ise doğru olarak yorumlanır. (Örneğin kontrol ifadesinin hesaplanan değerinin -5 olduğunu düşünelim, bu durumda kontrol ifadesi doğru olarak değerlendirilecektir). Eğer ifadenin sonucu doğru ise **else** anahtar sözcüğüne kadar olan kısım, eğer ifadenin sonucu yanlış ise **else** anahtar sözcüğünden sonraki kısım icra edilir. Örnek :

```
{
    int var = 3;

    if (var * 5 < 50)
        deyim1;
    else
        deyim2;
    deyim3;
}
```

1. adım : **if** parantezi içindeki ifadenin sayısal değeri hesaplanacak :  
15 < 50      sonuç 1 (doğru)

2. adım : deyim1 icra edilecek.

3. adım : daha sonra **else** kısmı atlanarak deyim3 icra edilecek.

**if** parantezi içerisindeki ifadeler karmaşık yapıda da olabilir :

```
...
char ch = 'Q';

if (ch >= 'a' && ch <= 'z')
    printf("%c\n", ch);
else
    printf("küçük harf değil\n");
ch = 'A';
```

Yukarıdaki **if** deyimi ile ch karakterinin küçük harf olup olmadığı test edilmektedir. ch karakterinin küçük harf olması durumunda bu karakter ekrana basılacak, aksi durumda ekrana "küçük harf değil" mesajı yazılacaktır. Her iki durumdada programın akışı

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

atama deyimiyle devam edecektir.

**if** deyiminin doğru ve / veya yanlış kısmı birden fazla deyimden oluşuyorsa bloklama yapılmalıdır :

```
if (b * b - 4 * a * c < 0) {  
    deyim1;  
    deyim2;  
    deyim3;  
}  
else {  
    deyim4;  
    deyim5;  
    deyim6;  
}
```

Yukarıdaki örnekte kontrol ifadesinin doğru olması durumunda, deyim1, deyim2, deyim3 icra edilecektir. Kontrol ifadesinin yanlış olması durumunda ise deyim4, deyim5, deyim6 icra edilecektir.

Bir **if** deyiminin **else** kısmı olmayabilir:

```
if (result < 0) {  
    clrscr();  
    printf("sonuç 0'dan küçük\n");  
}  
++x;  
...
```

Bir **if** deyimi yalnızca **else** kısmına sahip olamaz. Bu durumda **if** deyiminin doğru kısmına boş deyim ya boş bileşik deyim yerleştirilmelidir.

```
if (ifade)  
    ;  
else  
    deyim1;
```

ya da

```
if (ifade)  
    { }  
else  
    deyim1;
```

Ama daha iyi teknik koşul ifadesini değiştirmektir:

```
if (!ifade)  
    deyim1;
```

**if** parantezindeki ifade değişken içermek zorunda değildir, sabit ifadesi de olabilir:

```
if (10)  
    deyim1  
...  
if (-1)  
    deyim2
```

Yukarıdaki kontrol ifadelerinin değeri her zaman doğru olacaktır. (0 dışı değer)

```
if ( 0)  
    deyim1;
```

Yukarıdaki kontrol ifadesi ise her zaman yanlış olacağından **if** deyiminin doğru kısmı hiçbir zaman icra edilmeyecektir.

```
if (x) {
    deyim1;
    deyim2;
    ....
}
```

Bu **if** deyiminde ise x değişkeninin değerinin 0 dışı bir değer olup olmamasına göre deyim1 ve deyim2 icra edilecektir.

Yukarıdaki yapıyla aşağıdaki yapı eşdeğerdir:

```
if (x != 0) {
    deyim1;
    deyim2;
    ....
}
```

Aşağıdaki örneği inceleyelim :

```
if (!x) {
    deyim1;
    deyim2;
    ....
}
```

Bu **if** deyiminde ise ancak x değişkeninin değerinin 0 olması durumunda deyim1 ve deyim2 icra edilecektir.

Yine yukarıdaki yapıyla aşağıdaki yapı eşdeğerdir:

```
if (x == 0) {
    deyim1;
    deyim2;
    ....
}
```

```
if ((ch = getchar()) == 'h')
    deyim1;
else
    deyim2;
```

Bu **if** deyiminin kontrol ifadesinde ise klavyeden bir değer alınmakta alınan değer ch değişkenine atanmakta ve daha sonra da ch değişkeni değerinin 'h' karakteri olup olmadığı test edilmektedir. Eğer klavyeden alınan değer 'h' ise deyim1 değil ise deyim2 yapılacaktır. kontrol ifadesi içindeki parantezler atama işlemine öncelik kazandırmak amacıyla kullanılmaktadır. Parantez kullanılmasaydı eşitlik karşılaştırma operatörünün (==) önceliği atama operatöründen daha yüksek olduğu için önce karşılaştırma işlemi yapılacak daha sonra üretilen 0 ya da 1 değeri ch değişkenine atanacaktı.

Yukarıdaki örnekte de görüldüğü gibi bir **if** deyiminin koşul ifadesi içinde atama operatörünün bir değer üretmesi fikrinden sıklıkla faydalanılır.

Fonksiyonların geri dönüş değerleri de sıklıkla **if** deyiminin kontrol ifadesi olarak kullanılır.

```

if (isupper(ch) != 0)
    deyim1;
else
    deyim2;

```

isupper parametresinin büyük harf olup olmadığını kontrol eden bir standart C fonksiyonudur. Yukarıdaki **if** deyiminde ch karakterinin büyük harf olup olmamasına göre deyim1 ya da deyim2 icra edilecektir.

Yukarıdaki koşul ifadesi yerine C programcılar genellikle aşağıdaki ifadeyi tercih ederler :

```

if (isupper(ch))
    deyim1;
else
    deyim2;

```

**if** deyiminin doğru ya da yanlış kısmı başka bir **if** deyimi de olabilir :

```

if (ifade1)
    if (ifade2) {
        deyim1;
        deyim2;
        deyim3;
    }
    deyim4;

```

Bu örnekte ikinci **if** deyimi birinci **if** deyiminin doğru kısmını oluşturmaktadır. 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 olacaktır:

```

if (ifade1)
    if (ifade2)
        deyim1;
else
    deyim2;

```

Yukarıdaki örnekte yazım tarzı olarak **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 olacaktır. (dangling else) Eğer **else** anahtar sözcüğünün birinci **if** deyimine ait olması isteniyorsa, birinci **if** deyiminin doğru kısmı blok içine alınmalıdır.

```

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;

```

Birinci **if** deyiminin doğru kısmı birden fazla deyimden oluştuğu için (bu deyimlerden birisi de yine başka bir **if** deyimidir) bloklama yapılmıştır. deyim5 birinci **if** deyiminin yanlış kısmını oluşturmaktadır.

Ayrık karşılaştırma ve **else if** merdivenleri :

Aşağıdaki **if** deyimlerini inceleyelim :

```

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 verilen yapıda olduğu gibi, eğer bir karşılaştırmının doğru olarak yorumlanması durumunda yapılan diğer karşılaştırmaların doğru olması söz konusu değilse bu tür karşılaştırmalara ayrık karşılaştırma denir. Ayrık karşılaştırmalarda ayrı ayrı **if** deyimlerinin kullanılması kötü tekniktir. Yukarıdaki örnekte m değişkeninin değerinin 1 olduğunu düşünelim. Bu durumda ekrana Ocak yazdırılacak fakat daha sonra yer alan **if** deyimleriyle m değişkeninin sırasıyla 2, 3, .... 12'ye eşit olup olmadığı test edilecektir. Ama x değişkeni 1 değerine sahip olduğundan bütün diğer **if** deyimleri içindeki kontrol ifadelerinin yanlış olarak değerlendirileceği bellidir.

Ayrık karşılaştırmalarda **else if** merdivenleri uygulamaları çok kullanılır :

```

if (ifade1)
    deyim1;
else
    if (ifade2)
        deyim2;
    else
        if (ifade3)
            deyim3;
        else
            if (ifade4)
                deyim4;
            else
                deyim5;

```

Bu yapıda artık herhangi bir **if** deyimi içerisindeki bir kontrol ifadesi doğru olarak değerlendirilirse programın akışı hiçbir zaman başka bir **if** deyimine gelmeyecektir. 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 tarzı okunabilirlik açısından tercih edilmelidir.

```
if (ifade1)
    deyim1;
else if (ifade2)
    deyim2;
else if (ifade3)
    deyim3;
else if (ifade4)
    deyim4;
else deyim5;
```

### **if Deyiminin Kullanılmasına İlişkin Sık Yapılan Hatalar**

**if** parantezinin sonuna yanlışlıkla ; yerleştirilmesi:

```
...
if (x > 5);
    printf("doğru!\n");
...
```

Yukarıdaki örnekte  $x > 5$  ifadesi doğru da yanlış da olsa printf fonksiyonu çağırılacaktır. Zira printf çağırımı **if** deyiminin dışındadır. **if** deyiminin doğru kısmını bir boş deyim (Null statement) oluşturmaktadır. Aynı nedenden dolayı aşağıdaki kod derleme zamanında error oluşumuna neden olacaktır :

```
...
if (x > 5);
    printf("doğru!\n");
else
    printf("yanlış!\n");
...
```

**if** anahtar sözcüğü olmayan bir **else** anahtar sözcüğü:

Tabi ki bir **if** deyiminin doğru ya da yanlış kısmını bir ; (boş deyim - null statement) oluşturabilir. Bu durumda okunabilirlik açısından bu boş deyim bir tab içeriden yazılmalıdır:

```
...
if (funk())
    ;
else
    x = 5;
```

**if** parantezi içerisinde karşılaştırma operatörü (==) yerine yanlışlıkla atama operatörünün (=) kullanılması

```
...
if (x == 5)
    printf("eşit!\n");
...
```

Yukarıdaki **if** deyiminde x değişkeninin değeri eğer 5'e eşitse printf fonksiyonu çağırılacaktır. Operatörler konusunda da değindiğimiz gibi karşılaştırma operatörünün yan etkisi yoktur, yani yukarıdaki **if** parantezi içerisinde x değişkeninin değeri yalnızca 5 sabiti ile karşılaştırılmakta, değişkenin değeri değiştirilmemektedir. Oysa karşılaştırma operatörünün yerine yanlışlıkla atama operatörü kullanılırsa:

```
...
```

```

if (x = 5)
    printf("eşit\n");
...

```

Atama operatörü atama operatörünün sağ tarafındaki ifadenin değerini üreteceğinden **if** parantezi içindeki ifadenin değeri 5 olarak hesaplanacak ve 5 de 0 dışı bir değer olduğundan printf fonksiyonu x değişkeninin değeri ne olursa olsun, çağırılacaktır. Tabi, atama operatörü yan etkisi olan bir operatör olduğundan x değişkeni de **if** deyiminin icrası sırasında 5 değerini alacaktır.

C derleyicilerinin çoğu **if** parantezi içindeki ifade yalın bir atama ifadesi ise, durumu şüpheyle karşılayarak, bir uyarı mesajı verirler. Örneğin Borland derleyicilerinde tipik bir uyarı mesajı aşağıdaki gibidir :

warning : possibly incorrect assignment! (muhtemelen yanlış atama!)

Oysa **if** parantezi içerisinde atama operatörü bilinçli olarak da kullanılabilir. Bilinçli kullanımda, uyarı mesajının kaldırılması için ifade aşağıdaki gibi düzenlenebilir :

```

...
if ((x = funk()) != 0)
    m = 20;
...

```

Yukarıdaki örnekte atama operatörünün ürettiği değer açık olarak bir karşılaştırma operatörüne operand yapıldığı için derleyiciler bu durumda bir uyarı mesajı vermezler.

**if** deyiminin doğru ya da yanlış kısmının birden fazla basit deyimden oluşması durumunda **if** deyiminin doğru ya da yanlış kısmı bileşik deyim haline getirilmelidir.

```

...
if ( x == 10)
    m = 12;
    k = 15;
...

```

Yukarıdaki **if** deyiminde sadece

m = 12;

deyimi **if** deyiminin doğru kısmını oluşturmaktadır.

k = 15;

deyimi **if** deyimi dışındadır. Bu durum, genellikle programcının **if** deyiminin doğru ya da yanlış kısmını önce basit bir deyimle oluşturmasından sonra, doğru ya da yanlış kısma ikinci bir basit deyim eklerken, bloklama yapmayı 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 derleme zamanında error oluşumuna neden olacaktır :

```
...
if ( x == 10)
    m = 12;
    k = 15;
else
    y = 20;
...
```

Bu tür yanlışlıklardan sakınmak için bazı C programcıları **if** deyiminin doğru ya da yanlış kısmı basit 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 basit deyimin ilave edilmesi durumunda bir yanlışlık ya da error oluşmayacaktır. Ancak biz yukarıdaki gibi bir kodu stil açısından beğenmiyoruz. Gereksiz bloklamadan kaçınmalıyız.

Bazen de **if** deyiminin yanlış kısmı unutulur :

```
...
if (x == 10)
    printf("x 10'a eşit!\n")
printf("x 10'a eşit değil!\n");
...
```

**if** parantezi içerisindeki ifadenin yanlış olması durumunda bir yanlışlık söz konusu değil ama ifade doğru ise ekrana ne yazılacak?

```
x 10'a eşit!
x 10'a eşit değil!
```

Tehlikeli bir bug da **if** parantezi içerisindeki ifadenin bir fonksiyon çağırma ifadesi olması durumunda yanlışlıkla fonksiyon çağırma operatörünün unutulmasıdır!

```
...
if (funk())
    m = 12;
...
```

yerine

```
...
if (funk)
    m = 12;
...
```

yazılırsa, **if** deyiminin her zaman doğru kısmı icra edilir. Zira C dilinde bir fonksiyon ismi, o fonksiyonun kodunun bellekteki yerine eşdeğer bir adres bilgisi olarak ele alınır. (Göstericiler konusunda göreceğiz.) Bu adres bilgisi de her zaman 0 dışı bir değer olacağından, koşul ifadesi her zaman doğru olarak değerlendirilecektir.



## if Deyiminin Kullanıldığı Örnek Uygulamalar

### islower Fonksiyonu

islower standart bir C fonksiyonudur. Parametresi olan karakter, küçük harf karakterlerinden biri ise Doğru (sıfır dışı bir değer), değilse Yanlış (sıfır) değerine geri döner. Bu fonksiyonu aşağıdaki şekilde yazabiliriz :

```
#include <stdio.h>

int _islower (int ch)
{
    if (ch >= 'a' && ch <= 'z')
        return ch;
    return 0;
}

main()
{
    char ch;

    ch = getchar();
    if (_islower(ch))
        printf("küçük harf\n");
    else
        printf("küçük harf değil\n");
}
```

Yukarıda yazılan \_islower fonksiyonunda önce parametre değişkeninin küçük harf olup olmadığı test edilmektedir. Parametre değişkeni eğer küçük harf ise **if** deyiminin Doğru kısmı yapılacaktır. Doğru kısmının yapılması ile fonksiyon ch'nin kendi değerine geri dönecektir. (Bu da 0 dışı bir değerdir. Pekala 1 gibi sabit bir değer ile de geri dönebilirdik ama ch parametre değişkeninin değeri ile geri dönmemiz fonksiyonun kullanılmasında bazı avantajlar getirebilecektir. Fonksiyonun yazımında dikkat edilmesi gereken bir nokta da şudur : **if** deyiminin Yanlış kısmı yoktur. (Yani **else** anahtar sözcüğü kullanılmamıştır. Test fonksiyonlarında bu durum çok sık görülür. Çünkü **if** deyiminin Doğru kısmında return anahtar sözcüğü ile fonksiyon yalnızca bir geri dönüş değeri üretmekle kalmamakta aynı zamanda sonlanmaktadır. Bu durumda **else** anahtar sözcüğüne gerek kalmaz. (Çünkü Doğru kısmının yapılmasından sonra Yanlış kısmının da yapılması mümkün değildir.) Bu tür durumlarda **else** anahtar sözcüğünün kullanılması kötü teknik olarak değerlendirilir.

### isalpha Fonksiyonu

isalpha fonksiyonu da standart bir C fonksiyonudur. Parametresi olan karakter , eğer alfabetik karakterse (yani büyük ya da küçük harf ise) Doğru (sıfır dışı bir değere), alfabetik bir karakter değilse Yanlış (sıfır değerine) geri döner.

```
#include <stdio.h>

int isalpha (char ch)
{
    if (ch >= 'a' && ch <= 'z' || ch >= 'A' && ch <= 'Z')
        return ch;
    return 0;
}

main()
{
    char ch;
```

```

ch = getchar();
if (isalpha(ch))
    printf("alfabetik karakter\n");
else
    printf("alfabetik karakter değil\n");

```

### tolower Fonksiyonu

tolower standart bir C fonksiyonudur. Parametresi olan karakter, eğer büyük harf ise, onun küçük harf karşılığıyla geri döner. tolower küçük harf olmayan karakterlere hiç dokunmaz, onları değiştirmeden geri dönüş değeri olarak verir :

```

#include <stdio.h>

int tolower (int ch)
{
    if (ch >= 'A' && ch <= 'Z')
        return ch - 'A' + 'a';
    return ch;
}

main()
{
    char ch;

    ch = getchar();
    printf("%c\n", tolower(ch));
}

```

### isdigit Fonksiyonu

isdigit standart bir C fonksiyonudur. Parametresi olan karakter, eğer bir rakam karakteri ise 0 dışı bir değer ile, rakam karakteri değilse 0 değeri ile geri döner.

```

int _digit (char ch)
{
    if (ch >= '0' && ch <= '9')
        return ch;
    return 0;
}

```

yukarıdaki örneklerde görüldüğü gibi, fonksiyonların parametreleri ve geri dönüş değerleri char türden olsa bile, int biçiminde gösterilir.

### Karakter Test Fonksiyonları

Karakter test fonksiyonları karakterler hakkında bilgi edinmemizi sağlayan fonksiyonlardır. Bu fonksiyonların hepsi ctype.h başlık dosyası içinde makro olarak bulunurlar. Bu nedenle karakter test fonksiyonları kullanılmadan önce kaynak koda mutlaka ctype.h dosyası dahil edilmelidir. Karakter test fonksiyonları ASCII karakter setinin ilk yarısı için geçerlidir, yani türkçe karakterler için kullanılması durumunda geri dönüş değerleri güvenilir değildir.

#### C dilindeki standart karakter test fonksiyonları:

fonksiyon	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	hex sayıları gösteren bir karakterse Doğru değilse Yanlış
isalnum	alfabetik ya da nümerik bir karakterse Doğru değilse Yanlış

isspace	Boşluk karakterlerinden biriye(space, carriage return, new line, vertical tab, form feed) Doğru değilse Yanlış
ispunct	Noktalama karakterlerinden biriye (kontrol karakterleri, alfanümerik karakterler ve boşluk karakterlerinin dışındaki karakterler) Doğru değilse Yanlış
isprint	Ekranda görülebilen (print edilebilen) bir karakterse (space karakteri dahil) Doğru, değilse Yanlış.
isgraph	Ekranda görülebilen bir karakterse (space dahil değil) Doğru değilse yanlış
isctrl	Kontrol karakteri ya da silme karakteri ise (İlk 32 karakter ya da 127 numaralı karakter) Doğru değilse Yanlış
isascii	ASCII tablosunun standart kısmı olan ilk 128 karakterden biriye Doğru değilse Yanlış

## Uygulamalar

Kendisine gönderilen 0 ile 15 arasındaki bir sayının hexadesimal sembol karşılığı karakter ile geri dönen get\_hex\_char fonksiyonunun yazılması :

```
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;
}
```

test kodu :

```
main()
{
    int number;

    printf("0 ile 15 arasında bir sayı giriniz : ");
    scanf("%d", &number);
    printf("hexadesimal digit karşılığı = %c\n", get_hex_char(number));
    return 0;
}
```

Kendisine gönderilen hex digit olan bir karakterin desimal sistemdeki değerine geri dönen get\_hex\_value(char digit) fonksiyonunun yazılması.

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

```
int get_hex_value(char digit)
{
    digit = toupper(digit);

    if (digit >= '0' && digit <= '9')
        return digit - '0';
    if (digit >= 'A' && digit <= 'F')
        return digit - 'A' + 10;
}
```

test kodu :

```
main()
```

```
{  
    char hex;  
  
    printf("hex digit gösteren bir karakter giriniz: ");  
    hex = getchar();  
    printf("girmiş olduğunuz hex basamağın desimal değeri %d\n", get_hex_value(hex));  
    return 0;  
}
```

Kendisine gönderilen karakter küçük harf ise büyük harfe dönüştüren, büyük harf ise küçük harfe dönüştüren, eğer harf karakteri değilse karakterin kendisiyle geri dönen `change_case` isimli bir fonksiyonun tasarlanması :

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

```
int change_case(int ch)  
{  
    if (ch >= 'A' && ch <= 'Z')  
        return ch - 'A' + 'a';  
    if (ch >= 'a' && ch <= 'z')  
        return ch - 'a' + 'A';  
    return ch;  
}
```

test kodu :

```
main()  
{  
    int kar;  
  
    printf("bir karakter giriniz : ");  
    kar = getchar();  
    kar = change_case(kar);  
    putchar(kar);  
    getch();  
    return 0;  
}
```

`change_case` fonksiyonunu standart C fonksiyonlarını kullanarak aşağıdaki şekilde de tanımlayabilirdik :

```
int change_case(int ch)  
{  
    if (isupper(ch))  
        return tolower(ch);  
    return toupper(ch);  
}
```

İkinci dereceden bir denklemin çözüm kümesini bulan bir program :

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

```
main()
```

```
{
    double a, b, c;
    double delta, kokdelta;

    printf("denklemin katsayılarını giriniz : ");
    scanf("%lf%lf%lf", &a, &b, &c);

    delta = b * b - 4 * a * c;
    if (delta < 0) {
        printf("denkleminizin gerçek kökü yok!\n");
        return 0;
    }
    if (delta == 0) {
        printf("denkleminizin tek gerçek kökü var\n");
        printf("kok = %lf\n", -b / (2 * a));
        return 0;
    }
    kokdelta = sqrt(delta);
    printf("denkleminizin 2 gerçek kökü var : ");
    printf("kök 1 = %lf\n", (-b + kokdelta) / (2 * a));
    printf("kök 2 = %lf\n", (-b - kokdelta) / (2 * a));
    return 0;
}
```

## 10 . BÖLÜM : FONKSİYON PROTOTİPLERİ

C programlama dilinde, bir fonksiyonun çağırılması durumunda derleyiciler fonksiyonun geri dönüş değerinin türünü bilmek zorundadır. C derleyicileri fonksiyonların geri dönüş değerlerini CPU yazmaçlarından (registers) alırlar ve aslında geri dönüş değeri türü, değerin hangi yazmaçtan alınacağını gösterir.

Eğer çağırılan fonksiyonun tanımlaması, fonksiyon çağırma ifadesinden daha önce yer alıyorsa, derleyici derleme işlemi sırasında fonksiyon çağırma ifadesine gelmeden önce, çağırılan fonksiyonun geri dönüş değeri türü hakkında zaten bilgi sahibi olacaktır. Çünkü derleme işlemi yukarıdan aşağı doğru yapılır.

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

```
float calculate(float x, float y)
{
    return x * y / (x + y);
}
```

```
int main()
{
    float a, b, c;

    c = calculate(a, b);
    printf("%f\n", c );
    return 0;
}
```

Yukarıdaki örnekte calculate fonksiyonu kendisini çağıran main fonksiyonundan daha önce tanımlanmıştır. Dolayısıyla çağırma ifadesine gelmeden önce derleyici, calculate fonksiyonunun geri dönüş değeri türünü zaten bilecektir.

Eğer çağırılan fonksiyonun tanımlaması çağıran fonksiyondan daha sonra yapılmışsa, derleyici fonksiyon çağırma ifadesine geldiğinde, söz konusu fonksiyonun geri dönüş değerinin türünü belirleyemez. Bu problemli bir durumdur.

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

```
int main()
{
    float a, b, c;
    c = calculate(a, b);
    printf("%f\n", c );
    return 0;
}
```

```
float calculate (float x, float y)
{
    return x * y / (x + y);
}
```

Yukarıda calculate fonksiyonu main içinde çağırılmıştır. Fakat calculate fonksiyonunun tanımlaması kaynak kod içinde main'den daha sonra yer almaktadır. Derleme akışı içerisinde calculate fonksiyonuna ilişkin çağırma ifadesine gelindiğinde, derleyici bu fonksiyonun geri dönüş değerini bilmemektedir.

C derleyicileri derleme işlemi sırasında bir fonksiyon çağırma ifadesi gördüklerinde, eğer fonksiyonun geri dönüş değeri türü hakkında henüz sahibi değillerse, söz konusu geri dönüş değerinin **int** türden olduğunu kabul ederler.

Yukarıdaki örnekte derleyici calculate fonksiyonunun geri dönüş değerinin **int** türden olduğunu varsayacak ve buna göre kod üretecektir. Daha sonra derleme akışı fonksiyonun tanımlama kısmına geldiğinde ise artık iş işten geçmiş olacaktır. Hedef kod oluşumunu engelleyen bu durumu derleyiciler bir hata mesajı ile bildirirler.

Bu hata mesajı Microsoft derleyicilerinde : 'calculate': redefinition

Borland derleyicilerinde ise : Type mismatch in redeclaration of 'calculate'

Çağırılan fonksiyonu çağıran fonksiyonun üstünde tanımlamak her zaman mümkün değildir. Büyük bir programda yüzlerce fonksiyon tanımlanabilir ve tanımlanan her fonksiyonun birbirini çağırması söz konusu olabilir. Bu durumda çağırılacak fonksiyonu çağıran fonksiyonun üzerinde tanımlanması çok zor olacaktır. Kaldı ki, C dilinde iki fonksiyon birbirini de çağırabilir. Bu tür bir fonksiyon tasarımı artık çağırılan fonksiyonun daha önce tanımlanması mümkün olamayacaktır :

```
double func1(void)
{
    ...
    func2();
    ...
}
```

```
double func2(void)
{
    ...
    func1();
    ...
}
```

Ayrıca standart C fonksiyonları da ancak bağlama aşamasına gelindiğinde bağlayıcı (linker) tarafından kütüphanelerden alınarak çalışabilen kod (.exe) içine yerleştirilirler. İşte bu gibi durumlarda derleyiciye çağırılan fonksiyonun geri dönüş tür bilgisi fonksiyon prototipleriyle verilir. Çağırılana kadar tanımlaması yapılmamış fonksiyonlar hakkında derleyicilerin bilgilendirilmesi işlemi fonksiyon prototip bildirimleri ile yapılır.

## Fonksiyon Prototip Bildirimlerinin Genel Biçimi

[geri dönüş değeri türü] <fonksiyon ismi> ([tür1], [tür2].....);

Örneğin **calculate** fonksiyonu için prototip aşağıdaki biçimde yazılabilir:

```
float calculate(float, float);
```

Derleyici böyle bir prototip bildiriminden calculate fonksiyonunun geri dönüş değerinin türünün **float** olduğunu anlayacaktır.

Birkaç örnek daha verelim:

```
int multiply (int, int);
double pow (double, double);
void clrscr(void);
```

Tıpkı fonksiyon tanımlamalarında olduğu gibi, fonksiyon prototip bildirimlerinde de, fonksiyonun geri dönüş değeri belirtilmemişse, derleyici bildirimin int türden bir geri dönüş değeri için yapıldığını anlayacaktır.

```
func(double);
```

Yukarıdaki prototip bildirimi örneğinde, derleyici func fonksiyonunun geri dönüş değerinin int türden olduğu bilgisini alır, fonksiyonun geri dönüş değerinin olmadığı bilgisini değil. Eğer tanımlanacak fonksiyon geri dönüş değeri üretmeyecekse, **void** anahtar sözcüğü kullanılmalıdır:

```
void func(double);
```

Fonksiyon prototipleri yalnızca derleyiciyi bildirme amacıyla kullanılır bir bildirimdir (declaration) bir tanımlama (definition) işlemi değildir, dolayısıyla yapılan bildirim sonucunda bellekte bir yer ayrılmaz.

Fonksiyon prototip bildirimlerinde parametre değişkenlerinin türlerinden sonra parametre isimleri de yazılabilir. Prototiplerdeki parametre isimlerinin faaliyet alanları yalnızca parametre parantezi ile sınırlıdır. Buraya yazılan parametre değişkenleri isimleri yalnızca okunabilirlik açısından faydalıdır. Buradaki değişken isimlerinin fonksiyonun gerçek tanımlamasında kullanılacak formal parametre değişkenlerinin isimleriyle aynı olması zorunluluğu yoktur. Yukarıdaki prototiplerini parametre isimleriyle tekrar yazalım.

```
float calculate(float a, float b);
int multiply(int number1, int number2);
double pow(double base, double exp);
```

Fonksiyon prototip bildirimlerinde parametre değişkenlerine isim verilmesi, bildirimleri okuyan kişilerin fonksiyonların tanımlarını görmeden, değişkenler için kullanılan isimleri sayesinde, fonksiyonların yaptığı iş konusunda daha fazla bilgi sahibi olmalarına yardımcı olur.

Aynı türden geri dönüş değerine sahip fonksiyonların bildirimi virgüllerle ayrılarak yazılabilir, ama bu genel olarak pek tercih edilen bir durum değildir :

```
double func1(int), func2(int, int), func3(float);
```

Yukarıdaki bildirimde func1, func2 ve func3 fonksiyonlarının hepsi **double** türden geri dönüş değerine sahip fonksiyonlardır.

Fonksiyon prototip bildirimleri değişken tanımlamalarıyla da birleştirilebilir. Bu da tercih edilen bir durum değildir.

```
long func1(int), long func2(void), x, y;
```

Yukarıdaki deyim ile **func1** ve **func2** fonksiyonlarının prototip bildirimi yapılmışken, **x** ve **y** değişkenleri tanımlanmıştır.

(C++ dilinde eğer çağrılan fonksiyon çağırılan fonksiyondan daha önce tanımlanmamışsa, fonksiyonun geri dönüş değeri **int** türden kabul edilmez. Bu durumda fonksiyon prototip bildiriminin yapılması zorunludur. Prototip bildiriminin yapılmaması durumunda derleme zamanında hata (error) oluşacaktır.)

## Fonksiyon Prototiplerinin Bildirim Yerleri

Fonksiyon prototiplerinin bildirimi programın herhangi bir yerinde yapılabilir. Prototipler bildirimleri global düzeyde yapılmışsa (yani tüm blokların dışında yapılmışsa) bildirildikleri yerden dosya sonuna kadar olan alan içinde geçerliliklerini sürdürürler. Önemli olan nokta söz konusu fonksiyon çağırılmadan bildiriminin yapılmış olmasıdır.

Ancak uygulamalarda çok az raslanmasına karşılık, fonksiyon prototipleri yerel düzeyde de yapılabilir. Bu durumda prototip bildirimi ile , yalnızca bildirimin yapılmış olduğu bloğa bilgi verilmiş olur. Başka bir deyişle prototip bildirimi de, değişken tanımlamaları gibi faaliyet alanı kuralına uyar.



Geleneksel olarak fonksiyon prototip bildirimleri programın en yukarısında ya da programcının tanımladığı başlık dosyalarının birinin içinde yapılır. Başlık dosyaları (header files) ileride detaylı olarak ele alınacaktır.

## Standart C Fonksiyonlarının Prototipleri

Standart C fonksiyonlarının prototipleri standart başlık dosyaları içine yerleştirilmiştir. Programcı, uygulamalarda standart bir C fonksiyonunun prototipini kendi yazmaz, bu prototip bildiriminin bulunduğu başlık dosyasını `#include` önilemci komutuyla (ileride detaylı göreceğiz) koda dahil eder.

Standart C fonksiyonlarının prototip bildirimlerinin yapılmaması durumunda hata ortaya çıkmaz. Eğer geri dönüş değeri türü **int** değilse ve geri dönüş değeri kullanılırsa, programda yanlışlık söz konusu olacaktır.

Derleme zamanında hata oluşumu söz konusu değildir, çünkü derleyici fonksiyonun geri dönüş değerinin türünü **int** türden kabul edecek fakat daha sonra kaynak kod içinde fonksiyonun tanımlamasını göremediğinden hata oluşmayacaktır. Ancak programın çalışma zamanı sırasında fonksiyonun geri dönüş değeri **int** türden bir değer olarak alınacağından yanlışlık söz konusu olacak ve program doğru çalışmayacaktır.

Standart C fonksiyonlarının prototipleri sonu `.h` uzantılı olan (header) başlık dosyalar içindedir. Önilemci komutuyla ilgili başlık dosyasının kaynak koda ilave edilmesiyle, aslında standart C fonksiyonunun da prototip bildirimi yapılmış olmaktadır. Zira önilemci modülünün çıktısı olan kaynak program artık derleyiciye verildiğinde, ekleme yapılmış bu dosyada fonksiyonun prototip bildirimi de bulunmaktadır. Şüphesiz, başlık dosyayı kaynak koda ilave etmek yerine standart C fonksiyonunun prototipini kendimiz de kaynak koda yazabiliriz, bu durumda da bir yanlışlık söz konusu olmayacaktır. Örnek :

Standart bir C fonksiyonu olan **pow** fonksiyonunun prototipini iki şekilde kaynak koda ekleyebiliriz:

1. Fonksiyon prototipinin bulunduğu başlık dosyasını kaynak koda bir önilemci komutuyla dahil ederek.

```
#include <math.h>
```

2. Fonksiyonun prototipini kendimiz yazarak.

```
double pow(double taban, double us);
```

Ancak tercih edilecek yöntem başlık dosyasını kaynak koda dahil etmek olmalıdır. Çünkü:

1. Programcı tarafından fonksiyonun prototipi yanlış yazılabilir.
2. Başlık dosyalarının kaynak koda dahil edilmesinin nedeni yalnızca fonksiyon prototip bildirimi değildir. Başlık dosyalarında daha başka bilgiler de vardır. (Makrolar, sembolik sabitler, tür tanımlamaları, yapı bildirimleri vs.)

## Fonksiyon Prototip Bildirimi İle Arguman-Parametre Uyumu Kontrolü

Fonksiyon prototiplerinin ana amacı yukarıda da belirtildiği gibi, derleyiciye fonksiyonun geri dönüş değeri türü hakkında bilgi vermektir. Ancak fonksiyon prototip bildirimlerinde fonksiyon parametrelerinin türleri belirtilmişse, derleyici prototip bildirimindeki parametre değişkeni sayısını fonksiyon çağırma ifadesindeki fonksiyona gönderilen arguman sayısı ile karşılaştırır. Örneğin:

```
float calculate(float, float);
```

biçiminde bir prototip yazıldığında eğer `calculate` fonksiyonu eksik ya da fazla parametre ile çağırılırsa derleme hatası oluşacaktır.

```
x = calculate(5.8);          /* hata eksik parametre ile çağırılmış */
y = calculate(4.6, 7.9, 8.0) /* hata fazla parametre ile çağırılmış */
```

Fonksiyon prototip bildiriminde parantezin içi boş bırakılırsa (buraya hiçbir parametre tür bilgisi yazılmazsa) bu durumun özel bir anlamı vardır. Bu durumda derleyiciden parametre kontrolü yapması istenmemiş olur. (Derleyici artık fonksiyon çağırma ifadesindeki argüman sayısı ile fonksiyonun formal parametrelerinin sayısının eşitliğini kontrol etmez.) Parametre kontrolü yapılmasının istenmemesi uygulamalarda sık görülen bir durum değildir. Parametre değişkenlerinin sayısı ile fonksiyona gönderilen argüman sayısının uyumu kontrolü, C diline standartlaştırma çalışmaları sırasında eklenmiştir. Klasik C diye adlandırdığımız, C dilinin standartlaştırılmasından önceki dönemde böyle bir kontrol söz konusu değildi ve prototip bildirimlerinde fonksiyon parantezlerinin içi boş bırakılırdı. Geriye doğru uyumun korunması ve eskiden yazılmış kaynak kodların da desteklenmesi amacıyla, fonksiyon prototip bildirimlerinde fonksiyon parantezlerinin içinin boş bırakılması durumu halen geçerli bir işlem olarak bırakılmıştır.

```
float calculate(); /* derleyiciden parametre kontrolü yapmaması isteniyor */
```

Yukarıdaki bildirimden calculate fonksiyonunun parametre almadığı anlamı çıkmaz. Yukarıdaki bildirimden sonra eğer calculate fonksiyonu aşağıdaki ifadelerle çağırılırsa derleme zamanı hatası oluşmayacaktır:

```
x = calculate (5.8);          /* derleme zamanında hata oluşmaz */
y = calculate (4.6, 7.9, 8.0) /* derleme zamanında hata oluşmaz */
```

Eğer fonksiyonun gerçekten parametre değişkeni yoksa, ve derleyicinin fonksiyonun çağırılması durumunda argüman parametre değişkeni kontrolü yapması isteniyorsa, prototip bildiriminde fonksiyon parantezi içerisine **void** anahtar sözcüğü yazılmalıdır.

```
float sample(void);
```

Burada sample fonksiyonunun parametre almadığı bildirilmiştir. Eğer fonksiyon

```
x = sample(20);          /* derleme hatası */
```

İfadesiyle çağırılırsa derleme hatası oluşur.

(C++ dilinde fonksiyon prototip bildiriminde, fonksiyon parametre parantezinin içinin boş bırakılması, fonksiyonun parametre değişkeni olmadığını göstermektedir. Başka bir deyişle, prototip bildirimi sırasında parantezlerin içini boş bırakmak, derleyiciden argüman sayısı kontrolü yapmasını istememek anlamına gelmez. C++'da argüman sayısı ile parametre değişkenlerinin sayısının uyumu daima derleyici tarafından kontrol edilir. Dolayısıyla derleyici, prototip bildiriminde, parametre parantezi içine bir şey yazılmadığını gördükten sonra, fonksiyonun tanımlanma ifadesinde parametre değişken(ler)inin varlığını görürse, bu durumu bir error mesajı ile bildirir.)

Fonksiyon prototip bildiriminin yapılmış olması o fonksiyonun tanımlanmasını ya da çağırılmasını zorunlu kılmaz.

Prototip bildirimi yapılan bir fonksiyonu tanımlamamak hata oluşturmaz.

Bir fonksiyonun prototip bildirimi birden fazla yapılabilir.. Bu durumda hata oluşturmaz. Ama yapılan bildirimler birbirleriyle çelişmemelidir.

Kaynak dosya içinde aynı fonksiyona ilişkin prototip bildirimlerinin farklı yerlerde ve aşağıdaki biçimlerde yapıldığını düşünelim :

```
int sample (int, int);
sample (int, int);
int sample(int x, int y);
```

```
sample(int number1, int number2);
```

Yukarıdaki bildirimlerinin hiçbirinde bir çelişki söz konusu değildir. Fonksiyon parametre değişkenlerinin isimleri için daha sonraki bildirimlerde farklı isimler kullanılması bir çelişki yaratmayacaktır. Çünkü bu isimlerin faaliyet alanı (name scope) yalnızca bildirimin yapıldığı parantezin içidir. Ancak aşağıdaki farklı bildirimler derleme zamanında error oluşturacaktır :

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

```
double func(int x, float y);
```

```
/* error! bildirimler arasında çelişki var */
```

```
long sample(double x);
```

```
sample (double x);
```

```
/* error! bildirimler arasında çelişki var. */
```

Fonksiyon prototiplerinde parametre değişkenlerinin türlerinin de belirtilmesi, argümanların parametre değişkenlerine aktarılmasında tür dönüşümüne olanak sağlamaktadır. Bu durum tür dönüşümleri konusunda ele alınacaktır.

## 11 . BÖLÜM : KOŞUL OPERATÖRÜ

Koşul operatörü (conditional operator) C dilinin 3 operand alan tek operatörüdür. (ternary operator) Koşul operatörünün 3 operandı, ifade tanımına uygun herhangi bir ifade olabilir. Koşul operatörünün genel sentaksı aşağıdaki gibidir:

ifade1 ? ifade2 : ifade3

Koşul operatörü yukarıdaki biçimden de görüldüğü gibi birbirinden ayrılmış iki atomdan oluşmaktadır. ? ve : atomları operatörün 3 operandını birbirinden ayırır.

Derleyici bir koşul operatörü ile karşılaştığını ? atomundan anlar ve ? atomunun solundaki ifadenin (ifade1) sayısal değerini hesaplar. Eğer ifade1'in değeri 0 dışı bir sayısal değerse, bu durum koşul operatörü tarafından doğru olarak değerlendirilir, ve bu durumda yalnızca ifade2'nin sayısal değeri hesaplanır.

Eğer ifade1'in değeri 0 ise bu durum koşul operatörü tarafından yanlış olarak değerlendirilir ve bu durumda yalnızca ifade3'ün sayısal değeri hesaplanır.

Diğer operatörlerde olduğu gibi koşul operatörü de bir değer üretir. Koşul operatörünün 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 hesaplanacaktır. Bu ifade 0 dışı bir değerse (yani doğru ise) koşul operatörü  $y + 5$  değerini üretecektir.  $x > 3$  ifadesinin değeri 0 ise (yani yanlış ise) koşul operatörü  $y - 5$  değerini üretecektir. Bu durumda m değişkenine  $x > 3$  ifadesinin doğru ya da yanlış olmasına göre  $y + 5$  ya da  $y - 5$  değeri atanacaktır.

Aynı işlem if deyimi ile de yapılabilir :

```
if (x > 3)
    m = y + 5;
else
    m = y - 5;
```

Koşul operatörü Operatör Öncelik Tablosunun 13. öncelik seviyesindedir. Bu seviye atama operatörünün hemen üstüdür. Aşağıdaki ifadeyi ele alalım:

$x > 3 ? y + 5 : y - 5 = m$

Koşul operatörünün önceliği atama operatöründen daha yüksek olduğu için, önce koşul operatörü ele alınır.  $x > 3$  ifadesinin DOĞRU olduğunu ve operatörün  $y + 5$  değerini ürettiğini düşünelim. Toplam ifadenin değerlendirilmesinde kalan ifade

$y + 5 = m$

olacak ve bu da derleme zamanı hatasına yol açacaktır. Çünkü  $y + 5$  ifadesi sol taraf değeri değildir, nesne göstermez. (Lvalue required).

Koşul operatörünün birinci kısmını (ifade1) parantez içine almak gerekmez. Ancak, okunabilirlik açısından genellikle parantez içine alınması tercih edilir.

$(x >= y + 3) ? a * a : b$

Koşul operatörünün üçüncü operandı (ifade3) konusunda dikkatli olmak gerekir. Örneğin :

$m = a > b ? 20 : 50 + 5$

$a > b$  ifadesinin doğru olup olmamasına göre koşul operatörü 20 ya da 55 değerini üretecek ve son olarak da  $m$  değişkenine koşul operatörünün ürettiği değer atanacaktır. 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 operatörünün 3 operandı da bir fonksiyon çağırma ifadesi olabilir, ama çağırılan fonksiyonların geri dönüş değeri üreten fonksiyonlar olması (**void** olmayan) gerekmektedir. Üç operandtan biri geri dönüş değeri **void** olan bir fonksiyona ilişkin fonksiyon çağırma ifadesi olursa koşul operatörü değer üretmeyeceğinden bu durum derleme zamanında hata oluşmasına neden olacaktır.

Aşağıdaki kod parçasını inceleyelim:

```
#include <stdio.h>

int func1(void);
int func2(void);
int func3(void);

int main()
{
    int m;

    m = func1() ? func2() : func3();
    return 0;
}
```

Yukarıda koşul operatörünün kullanıldığı ifadede  $m$  değişkenine, `func1` fonksiyonunun geri dönüş değerinin 0 dışı bir değer olması durumunda `func2` fonksiyonunun geri dönüş değeri, aksi halde `func3` fonksiyonunun geri dönüş değeri atanacaktır.

Koşul operatörünün ürettiği bir nesne değil bir değerdir. Koşul operatörünün ürettiği değer nesne göstermediği için bu değere bir atama yapılamaz. Aşağıdaki **if** deyimini inceleyelim:

```
if (x > y)
    a = 5;
else
    b = 5;
```

Yukarıdaki **if** deyimine  $x > y$  ifadesinin doğru olması durumunda  $a$  değişkenine, yanlış olması durumunda ise  $b$  değişkenine 5 değeri atanıyor. Aynı işi koşul operatörü kullanarak yapmak istenip de aşağıdaki ifade oluşturulursa:

$$(x > y) ? a : b = 5;$$

bu durum derleme zamanı hatasına yol açacaktır. Çünkü koşul operatörünün ürettiği  $a$  ya da  $b$  değişkenlerinin değeridir, nesnenin kendisi değildir. Böyle bir atama sol tarafın nesne gösteren bir ifade olmamasından dolayı derleme zamanında hata oluşturur.

Aynı nedenden dolayı aşağıdaki ifadenin değerlendirilmeside derleme zamanında hata oluşmasına neden olacaktır:

$$(x > 5 ? y : z)++;$$

Parantez içindeki ifade değerlendirildiğinde elde edilen y ya da z nesneleri değil bunların değerleridir. Yani postfix ++ operatörünün operandı nesne değildir. Bu durumda error oluşacaktır. (L value required!)

## Koşul Operatörünün Kullanıldığı Durumlar

Koşul operatörü her zaman **if** deyimine bir alternatif olarak kullanılmamalıdır. Koşul operatörünün kullanılmasının tavsiye edildiği tipik durumlar vardır ve bu durumlarda genel fikir, koşul operatörünün ürettiği değerden aynı ifade içinde faydalanmak, bu değeri bir yere aktarmaktır.

1. Bir önermenin doğru ya da yanlış olmasına göre farklı iki değerden birinin aynı değişkene aktarılması durumunda:

```
p = (x == 5) ? 10 : 20;
```

```
m = (a >= b + 5) ? a + b : a - b;
```

Yukarıdaki deyimleri **if else** yapısıyla da kurabilirdik:

```
if (x == 5)
```

```
    p = 10;
```

```
else
```

```
    p = 20;
```

```
if (a >= b + 5)
```

```
    m = a + b;
```

```
else
```

```
    m = a - b;
```

2. Fonksiyonlarda return ifadesini oluştururken:

```
return (x > y ? 10 : 20);
```

Bu örnekte  $x > y$  ifadesinin doğru olup olmamasına göre fonksiyonun geri dönüş değeri 10 ya da 20 olacaktır. Yukarıdaki ifade yerine aşağıdaki **if** yapısı da kullanılabilir:

```
if (x > y)
```

```
    return 10;
```

```
else
```

```
    return 20;
```

3. Fonksiyon çağırma ifadelerinde arguman olarak:

```
func(a == b ? x : y);
```

Bu ifadede func fonksiyonu a değişkeninin b değişkenine eşit olup olmaması durumuna göre x ya da y argumanlarıyla çağırılacaktır. **if** deyimiyile aşağıdaki şekilde karşılanabilir:

```
if (a == b)
```

```
    func(x);
```

```
else
```

```
    func(y);
```

4. **if** deyiminde koşul ifadesi olarak da koşul operatörünün kullanıldığı görülür:

```
if ( y == (x > 5 ? 10 : 20))
```

Yukarıdaki ifadede  $x > 5$  ifadesinin doğru olup olmamasına göre **if** parantezi içinde y değişkeninin 10 ya da 20'ye eşitliği test edilecektir.

Yukarıdaki durumlarda koşul operatörünün **if** deyimine tercih edilmesi iyi tekniktir. Bu durumlarda koşul operatörü daha okunabilir bir yapı oluşturmaktadır. (C'ye yeni başlayanlar için **if** yapısı daha iyi okunabilir ya da algılanabilir, ancak bir C programcisi için koşul operatörünün kullanılması daha okunabilir bir yapı oluşturur.)

Koşul operatörünün bilinçsizce kullanılmaması gerekir. Eğer koşul operatörünün ürettiği değerden doğrudan faydalanılmayacaksa koşul operatörü yerine **if** kontrol deyimi tercih edilmelidir. Örneğin:

```
x > y ? a++ : b++;
```

Deyiminde koşul operatörünün ürettiği değerden faydalanılmamaktadır. Burada aşağıdaki **if** yapısı tercih edilmelidir:

```
if (x > y)
    a++;
else
    b++;
```

Başka bir örnek:

```
x == y ? printf("eşit\n") : printf("eşit değil\n");
```

Bu örnekte printf fonksiyonunun bir geri dönüş değeri üretmesinden faydalanılarak koşul operatörü kullanılmıştır. Koşul operatörü  $x == y$  ifadesinin doğru olup olmamasına göre, 2. veya 3. ifade olan printf fonksiyonu çağırma ifadelerinden birinin değerini (geri dönüş değerini) üretecektir. (Bu da aslında ekrana yazılan karakter sayısıdır.) Ama ifade içinde koşul operatörünün ürettiği değer kullanılması söz konusu değildir. Burada da **if** kontrol deyimi tercih edilmelidir :

```
if (x == y)
    printf("eşit\n");
else
    printf("eşit değil\n");
```

Koşul operatörünün ikinci ve üçüncü operandlarının türleri farklı ise tür dönüştürme kuralları diğer operatörlerde olduğu gibi devreye girecektir:

```
long a;
int b;
```

```
m = (x == y) ? b : a;
```

Bu örnekte a nesnesi **long** türden b nesnesi ise **int** türündendir.  $x == y$  karşılaştırma ifadesi yanlış olsa bile tür dönüşümü gerçekleşecek ve **int** türden olan b long türe dönüştürülecektir.

Bazı durumlarda koşul operatörünün de **if** kontrol değiminin de kullanılması gerekmeyebilir.

```
if (x > 5)
    m = 1;
else
    m = 0;
```

Yukarıdaki **if** deyimi koşul operatörü ile aşağıdaki şekilde oluşturulabilirdi :

```
m = (x > 5) ? 1 : 0;
```

Ancak koşul operatörünün üreteceği değerlerin 1 veya 0 olabileceği durumlarda doğrudan karşılaştırma operatörünü kullanmak daha iyi teknik olarak değerlendirilmelidir.

```
m = x > 5;
```

Başka bir örnek :

```
return x == y ? 1 : 0;
```

yerine

```
return x == y;
```

yazabilirdik.

Aşağıda tanımlanan `is_leap` fonksiyonunu inceleyelim, fonksiyonun geri dönüş değerinin üretilmesinde koşul operatörü kullanılmayıp doğrudan mantıksal operatörlerin 0 ya da 1 değeri üretmeleri fikrinden faydalanılmıştır:

```
#define BOOL      int

BOOL is_leap(int year)
{
    return !(year % 4) && year % 100 || !(year % 400);
}
```

Koşul operatörünün öncelik yönü sağdan soladır. (right associative). Bir ifade içinde birden fazla koşul operatörü varsa önce en sağdaki değerlendirilecektir.

Aşağıdaki kod parçasını inceleyelim :

```
int x = 1;
int y = 1;

m = x < 5 ? y == 0 ? 4 : 6 : 8;
printf("m = %d\n", m);
```

Yukarıdaki kod parçasında `printf` fonksiyonu çağırımında `m` değişkeninin değeri 6 olarak yazdırılacaktır. İfade aşağıdaki gibi ele alınacaktır :

```
m = x < 5 ? (y == 0 ? 4 : 6) : 8;
```



## 12 . BÖLÜM : DÖNGÜ DEYİMLERİ

Bir program parçasının yinelenmeli olarak çalıştırılmasını sağlayan kontrol deyimlerine döngü denir. C dilinde 3 ayrı döngü deyimi vardır:

Kontrolün başta yapıldığı **while** döngüleri  
 Kontrolün sonda yapıldığı **while** döngüleri (**do while** döngüleri)  
**for** döngüleri

Bu döngü deyimlerinden en fazla kullanılanı **for** deyimidir. **for** deyimi yalnızca C dilini değil, tüm programlama dillerinin en güçlü döngü yapısıdır. Aslında **while** ya da **do while** döngüleri olmasa da bu döngüler kullanılarak yazılan kodlar, **for** döngüsüyle yazılabilir. Ancak okunabilirlik açısından **while** ve **do while** döngülerinin tercih edildiği durumlar vardır.

### Kontrolün Basta Yapıldığı while Döngüleri

Genel biçimi:

**while** (ifade)  
 deyim;

**while** anahtar sözcüğünü izleyen parantez içerisindeki ifadeye kontrol ifadesi (control statement) denir. **while** parantezini izleyen ilk deyim döngü gövdesi denir. Döngü gövdesi basit bir deyim olabileceği gibi, bloklanmış birden fazla deyimden de (bileşik deyim) oluşabilir.

**while** deyiminin icrası şu şekilde olur: Kontrol ifadesinin sayısal değeri hesaplanır. Kontrol ifadesinin sayısal değeri 0 dışı bir değerse, mantıksal olarak doğru kabul edilir ve döngü gövdesindeki deyim ya da deyimler çalıştırılır. Kontrol ifadesinin sayısal değeri 0 ise mantıksal olarak yanlış kabul edilir programın akışı döngünün dışındaki ilk deyimle devam eder. Yani **while** döngüsü, kontrol ifadesinin sayısal değeri 0 dışı bir değer olduğu sürece, döngü gövdesini oluşturan deyim(ler)in icrası ile devam eder.

Daha önce belirttiğimiz gibi C dilinde yalın bir deyimin olduğu yere bileşik bir deyim de yerleştirilebilir. Bu durumda **while** döngüsü aşağıdaki gibi de oluşturulabilir :

```
while (ifade) {
    ifade1;
    ifade2;
    ifade3;
    .....
}
```

Bu durumda kontrol ifadesinin sayısal değeri 0 dışı bir değer olduğu (doğru) sürece blok parantezleri arasında kalan tüm deyimler icra edilecektir. Örnek:

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

```
int main()
{
    int i = 0;

    while (i < 10) {
        printf ("%d\n", i)
        ++i;
    }
    return 0;
}
```

Başka bir örnek:

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

```
int main()
{
    char ch;

    while (ch = getch(), toupper(ch) != 'Q')
        putchar(ch);
    return 0;
}
```

Başka bir örnek daha:

```
int main()
{
    char ch;

    while (ch = getch(), isupper (ch))
        putchar(ch);
    return 0;
}
```

**while** parantezinin içindeki ifade yani koşul ifadesi, ifade tanımına uygun herhangi bir ifade olabilir.

```
while (1) {
    ....
}
```

Yukarıdaki **while** deyiminde kontrol ifadesi olarak bir sabit olan 1 sayısı kullanılmıştır. 1 değeri 0 dışı bir değer olduğundan ve kontrol ifadesi bir değişkene bağlı olarak değişemeyeceğinden, böyle bir döngüden çıkmak mümkün olmayacaktır. Bu tür döngülere sonsuz döngüler (infinite loops) denir. Sonsuz döngüler bir yanlışlık sonucu oluşturulabildiği gibi, bilinçli olarak da oluşturulabilir. Sonsuz döngülerden bazı yöntemlerle çıkılabilir.

### **break Anahtar Sözcüğü**

**break** anahtar sözcüğü ile bir döngü sonlandırılabilir. Kullanımı

**break;**

şeklinde. Programın akışı **break** anahtar sözcüğünü gördüğünde, döngü kırılarak döngünün akışı döngü gövdesi dışındaki ilk deyim ile devam eder. Yani koşulsuz olarak döngüden çıkılır.

```
int main (void)
{
    char ch;

    while (1) {
        ch = getch();
        if (ch == 'q')
            break;
        putchar(ch);
    }
    printf("döngüden çıkıldı!..\n");
    return 0;
}
```

Yukarıdaki programda bir sonsuz döngü oluşturulmuştur. Döngü içerisinde, döngünün her bir iterasyonunda `ch` değişkenine klavyeden bir değer alınmaktadır. Eğer klavyeden alınan karakter 'q' ise **break** anahtar sözcüğüyle programın akışı **while** döngü gövdesi dışındaki ilk deyimle devam edecektir.

### İççe Döngüler

Bir döngünün gövdesini başka bir kontrol deyimi oluşturabilir. Döngü gövdesini oluşturan kontrol deyimi bir **if** deyimi olabileceği gibi başka bir döngü deyimi de olabilir. (**while**, **do while**, **for** deyimleri)

```
int main()
{
    int i = 0;
    int k = 0;

    while (i < 10) {
        while (k < 10) {
            printf("%d %d", i, k);
            ++k;
        }
        ++i;
        return 0;
    }
}
```

İççe döngülerde içerideki döngüde **break** deyimi kullanıldığında yalnızca içerideki döngüden çıkılır, her iki döngüden birden çıkmak için **goto** deyimi kullanılmalıdır. (ileride göreceğiz)

**while** döngüsü bir bütün olarak tek deyim içinde ele alınır. Örnek:

```
while (1)
    while (1) {
        .....
        .....
        .....
    }
}
```

Burada ikinci **while** döngüsü tek bir kontrol deyimi olarak ele alınacağı için, bloklamaya gerek yoktur.

**while** döngüsünün yanlışlıkla boş deyim ile kapatılması çok sık yapılan bir hatadır.

```
int main()
{
    int i = 10;

    while (--i > 0); /* burada bir boş deyim var */
    printf("%d\n", i);
    return 0;
}
```

Döngü **while** parantezi içerisindeki ifadenin değeri 0 olana kadar devam eder ve boş deyim döngü gövdesi olarak icra edilir. Döngüden çıkıldığında ekrana 0 basılır.

Sonlandırıcı ; **while** parantezinden sonra konulursa herhangi bir sentaks hatası oluşmaz. Derleyici **while** döngüsünün gövdesinin yalnızca bir boş deyimden oluştuğu sonucunu çıkartır. Eğer bir yanlışlık sonucu değil de bilinçli olarak **while** döngüsünün gövdesinde boş deyim (null statement) bulunması isteniyorsa, okunabilirlik açısından, bu boş deyim **while** parantezinden hemen sonra değil, alt satırda ve bir tab içeriden yazılmalıdır.

## while Döngüsü İçerisinde Postfix ++ ya da -- Operatörünün Kullanılması

Bir postfix artırım ya da eksiltme işlemi yapıldığında önce döngüye devam edilip edilmeyeceği kararı verilir, sonra artırım ya da eksiltim uygulanır. Örnek :

```
int main()
{
    int i = 0;

    while (i++ < 100)
        printf("%d\n", i);
    printf("%d\n", i);    /* ekrana 101 değerini basar. */
    return 0;
}
```

Başka bir örnek:

```
...
int i = 10;

while (i-- > 0)
    printf("%d\n", i);
printf("%d\n", i);
```

n bir pozitif tam sayı olmak üzere **while** döngüsü kullanılarak n defa dönen bir **while** döngüsü oluşturmak için

```
while (n-- > 0)
```

ya da

```
while (n--)
```

kullanılabilir. Aşağıdaki içiçe döngü yapısı bir gecikme sağlamak için kullanılmıştır.

```
int main()
{
    int i = 0;
    long j;

    while (i++ < 10) {
        printf("%d\n", i);
        j = 1000000L;
        while (j-- > 0 )
            ;
    }
    return 0;
}
```

Bazen döngüler bilinçli bir şekilde boş deyimle kapatılmak istenebilir. Bu durumda boş deyim normal bir deyim gibi tablama kuralına uygun olarak yerleştirilmelidir.

## Kontrolün Sonda Yapıldığı while Döngüleri

Genel biçim;

1. **do**

```
    ifade 1;
    while (ifade 2);
```
2. **do** {

```
    ifade 1;
```

```

    ifade 2;
} while (ifade);

```

**do while** döngüsünde kontrol ifadesi sondadır. **while** parantezinden sonra sonlandırıcı ";" bulunmalıdır. Yani buradaki sonlandırıcı yanlışlık sonucu koyulmamıştır, deyimle ilişkin sentaksın bir parçasıdır. Döngü gövdesindeki deyim(ler) en az bir kere icra edilecektir. Örnek :

```

int main()
{
    int i = 0;
    do {
        ++i;
        printf("%d\n", i);
    } while (i < 10);
    return 0;
}

```

Başka bir örnek:

```

int main()
{
    char ch;

    do {
        printf("(e)vet / (h)ayıt?\n");
        ch = getch();
    } while (ch != 'e' && ch != 'h');
    printf("ok...\n");
    return 0;
}

```

## Uygulama

1'den 100'e kadar sayıları her satırda beş tane olacak biçimde ekrana yazan bir C programının yazılması:

```

#include <stdio.h>

int main()
{
    int i = 0;

    do {
        printf("%d ", ++i);
        if (i % 5 == 0)
            printf("\n");
    } while (i < 100);
    return 0;
}

```

Başka bir çözüm:

```

#include <stdio.h>

void main()
{
    int i = 0;

    while (i < 100) {
        printf ("%d", i);
    }
}

```

```
        if (i % 5 == 4)
            printf("\n");
        ++i;
    }
    return 0;
}
```

## Uygulama

Bir tamsayının basamak sayısını bulan program.

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

```
int main()
{
    int digits = 0;
    int number;

    printf("bir tamsayi girin: ");
    scanf("%d", &number);

    do {
        n /= 10;
        digits++;
    } while (n > 0);
    printf("the number has %d digit(s).\n", digits);
    return 0;
}
```

Aynı programı **while** döngüsü kullanarak yazalım.

```
...
while (n > 0) {
    n /= 10;
    digits++;
}
...
```

**do while** döngüsü yerine **while** döngüsü kullanıldığında, girilen sayının 0 olması durumunda döngü gövdesindeki deyimler hiç icra edilmeyecekti. Bu durumda ekranda:

The number has 0 digit(s)

yazısı çıkacaktı.

## for Döngüleri

**for** döngüleri yalnızca C dilinin değil, belki de tüm programlama dillerinin en güçlü döngü yapılarıdır. **for** döngülerinin genel biçimi şu şekildedir:

```
for (ifade1; ifade2; ifade3)
    deyim1;

for (ifade1; ifade2; ifade3)    {
    deyim1;
    deyim2;
    ...
}
```

Derleyici **for** anahtar sözcüğünden sonra bir parantez açılmasını ve parantez içerisinde iki noktalı virgül bulunmasını bekler. Bu iki noktalı virgül **for** parantezini üç kısma ayırır. Bu kısımlar yukarıda ifade1 ifade2 ve ifade 3 olarak gösterilmiştir.

**for** parantezi içinde mutlaka 2 noktalı virgül bulunmalıdır. **for** parantezi içinin boş bırakılması, ya da **for** parantezi içerisinde 1, 3 ya da daha fazla noktalı virgölün bulunması derleme zamanında hata oluşmasına yol açacaktır.

**for** parantezinin kapanmasından sonra gelen ilk deyim döngü gövdesini (loop body) oluşturur. Döngü gövdesi basit bir deyimden oluşabileceği gibi, bileşik deyimden de yani blok içine alınmış birden fazla deyimden de oluşabilir.

**for** parantezi içerisindeki her üç kısmın da ayrı ayrı işlevleri vardır.

**for** parantezinin 2. kısmını oluşturan ifadeye kontrol ifadesi denir. (control expression). Tıpkı **while** parantezi içindeki ifade gibi, döngünün devamı konusunda bu ifade söz sahibidir. Bu ifadenin değeri 0 dışı bir değer ise, yani mantıksal olarak doğru ise, döngü devam eder. Döngü gövdesindeki deyim(ler) icra edilir. Kontrol ifadesinin değeri 0 ise programın akışı **for** döngüsünün dışındaki ilk deyimle devam edecektir.

Programın akışı **for** deyimine gelince, **for** parantezinin 1. kısmı 1 kez icra edilir ve genellikle döngü değişkenine ilk değer verme amacıyla kullanılır. (Böyle bir zorunluluk yoktur).

**for** döngüsünün 3. kısım döngü gövdesindeki deyim ya da deyimler icra edildikten sonra, dönüşte çalıştırılır. Ve çoğunlukla döngü değişkeninin artırılması ya da azaltılması amacıyla kullanılır. (Böyle bir zorunluluk yok.)

```
for (ilk değer; koşul; işlem) {
    ...
    ...
    ...
}

int main()
{
    int i;

    for (i = 0; i < 2; ++i)
        printf("%d\n", i);
    printf("son değer = %d\n", i);
    return 0;
}
```

Yukarıdaki programı inceleyelim:

Programın akışı **for** deyimine gelince, önce **for** parantezi içindeki 1. ifade icra ediliyor. Yani i değişkenine 0 değeri atanıyor.

Şimdi programın akışı **for** parantezinin 2. kısmına yani kontrol ifadesine geliyor ve  $i < 2$  koşulu sorgulanıyor. Kontrol ifadesinin değeri 0 dışı bir değer olduğu için, ifade mantıksal olarak doğru kabul ediliyor ve programın akışı döngü gövdesine geçiyor. Döngü gövdesi bloklanmadığı için, döngü gövdesinde tek bir deyim var. (basit deyim). Bu deyim icra ediliyor. Yani ekrana i değişkeninin değeri yazılarak imleç alt satıra geçiriliyor.

Programın akışı bu kez **for** parantezinin 3. kısmına geliyor ve buradaki ifade bir deyimmiş gibi icra ediliyor, yani i değişkeninin değeri 1 artırılıyor. i değişkeninin değeri 1 oluyor.

2. ifade yeniden değerlendiriliyor ve  $i < 2$  ifadesi doğru olduğu için bir kez daha döngü gövdesi icra ediliyor.

Programın akışı yine **for** parantezinin 3. kısmına geliyor ve buradaki ifade bir deyimmiş gibi icra ediliyor, yani *i* değişkeninin değeri 1 artırılıyor. *i* değişkeninin değeri 2 oluyor.

Programın akışı yine **for** parantezinin 2. kısmına geliyor ve buradaki kontrol ifadesi tekrar sorgulanıyor. *i* < 2 ifadesi bu kez yanlış olduğu için programın akışı döngü gövdesine girmiyor ve programın akışı döngü gövdesi dışındaki ilk deyimle devam ediyor.  
Yani ekrana :

son değer = 2

yazılıyor.

## Uygulama

1'den 100'e kadar olan sayıların toplamını bulan program :

```
int main()
{
    int i;
    int total = 0;

    for (i = 0; i < 100; ++i)
        total += i;
    printf("Toplam = %d", total);
    return 0;
}
```

Döngü değişkeninin tamsayı türlerinden birinden olması gibi bir zorunluluk yoktur. Döngü değişkeni gerçek sayı türlerinden de olabilir.

```
int main()
{
    double i;

    for (i = 0; i < 6.28; i = i + 0.01)
        printf("%f\n", i);
    return 0;
}
```

Daha önce de söylendiği gibi **for** parantezi içindeki her 3 kısım da ifade tanımına uygun ifadeler içerebilir, yani bir döngü değişkenine ilk değer verilmesi, döngü değişkenine bağlı bir koşul ifadesi olması, döngü değişkeninin de azaltılması ya da artırılması bir zorunluluk değildir.

```
int main()
{
    char ch;

    for (ch = getch(); ch != 'p' ; ch = getch())
        putchar(ch);
    return 0;
}
```

Başka bir örnek:

```
int main()
{
    for (printf("1. ifade\n"); printf("2. ifade\n"), getch() != 'q'; printf("3. ifade\n"));
}
```



Virgül operatörü ile birleştirilmiş değerlerin soldan sağa doğru sırayla icra edileceğini ve toplam ifadenin üreteceği değerin en sağdaki ifadenin değeri olacağını hatırlayın.

**for** döngüsünün 1. kısmı hiç olmayabilir. Örneğin döngü dışında, programın akışı **for** deyiminin icrasına gelmeden önce, döngü değişkenine ilk değer verilmiş olabilir.

```
...
i = 0;
for (; i < 100; ++i)
    printf("%d\n", i);
```

**for** döngüsünün 3. kısmı da olmayabilir. Döngü değişkeninin artırılması ya da eksiltilmesi **for** parantezi içi yerine döngü gövdesi içerisinde gerçekleştirilebilir.

1. ve 3. kısmı olmayan (yalnızca 2. kısma sahip) bir for döngüsü örneği:

```
...
i = 0;

for (; i < 100; ) {
    printf("%d\n", i);
    ++i;
}
...
```

1.ve 3. kısmı olmayan **for** döngüleri tamamen **while** döngüleriyle eşdeğerdir. C’de **for** döngüleriyle **while** döngüleriyle yapabildiğimiz herşeyi yapabiliriz. O zaman şöyle bir soru aklımıza gelebilir: Madem **for** döngüleri **while** döngülerini tamamen kapsıyor, o zaman **while** döngülerine ne gerek var? **while** döngülerinin bazı durumlarda kullanılması for döngülerine göre çok daha okunabilir bir yapı yaratmaktadır.

**for** parantezinin 2. kısmı da hiç olmayabilir. Bu durumda kontrol ifadesi olmayacağı için döngü bir koşula bağlı olmaksızın sürekli dönecektir. Yani sonsuz döngü oluşturulacaktır. Ancak iki adet noktalı virgül yine parantez içinde mutlaka bulunmak zorundadır.

**for** parantezinin hiçbir kısmı olmayabilir. Ancak yine iki noktalı virgül bulunmak zorundadır:

```
...
i = 0;
for (;;) {
    printf("%d\n", i);
    ++i;
    if (i == 100)
        break;
}
...
```

**for** (;;) ile **while** (1) eşdeğerdir. İkisi de sonsuz döngü belirtir.

Sonsuz döngü oluşturmak için **for** (;;) biçimi **while** (1)’e göre daha çok tercih edilir. Bunun nedeni eski derleyicilerde while(1) ifadesi kullanıldığında döngünün her dönüşünde kontrol ifadesinin tekrar test edilmesidir. Ama yeni derleyicilerde böyle bir kontrol söz konusu değildir. Ama bazı programcılar **while** (1) ifadesini tercih ederler. (Tabi burada kontrol ifadesi 1 yerine 0 dışı herhangi bir değer de olabilirdi ama geleneksel olarak 1 ifadesi kullanılmaktadır.)

## Sonsuz Döngülerden Çıkış

1. **break** anahtar sözcüğü ile.

Bu durumda programın akışı döngü gövdesi dışındaki ilk deyime yönlenecektir. (eğer iç içe döngü varsa **break** anahtar sözcüğü ile yalnızca içteki döngüden çıkılacaktır.)

2. **return** anahtar sözcüğü ile  
bu durumda fonksiyonun (main de olabilir) icrası sona erecektir.
3. **goto** anahtar sözcüğüyle.  
İççe birden fazla döngü varsa en içteki döngü içinden en dıştaki döngünün dışına kadar çıkabiliriz. (**goto** anahtar sözcüğünün çok az sayıdaki faydalı kullanımından biri budur.)
4. exit fonksiyonu ile. (ileride göreceğiz)

### **continue Anahtar Sözcüğü**

**continue** anahtar sözcüğü de tıpkı **break** anahtar sözcüğü gibi bir döngü içerisinde kullanılabilir. Programın akışı **continue** anahtar sözcüğüne geldiğinde sanki döngü yinelemesi bitmiş gibi yeni bir yinelemeye geçilir. Eğer **for** döngüsü içerisinde kullanılıyorsa yeni bir yinelemeye geçmeden önce döngünün 3. kısmı yapılır. Örnek :

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

```
int main()
{
    int i, k;
    char ch;

    for (i = 0; i < 100; ++i) {
        if (i % 3 == 0)
            continue;
        printf("%d\n", i);
    }
    return 0;
}
```

**break** anahtar sözcüğü bir döngüyü sonlandırmak için, **continue** anahtar sözcüğü de bir döngünün o anda içinde bulunulan yinelemesini sonlandırmak için kullanılır.

**continue** anahtar sözcüğü özellikle, döngü içerisinde uzun **if** deyimlerini varsa, okunabilirliği artırmak amacıyla kullanılır.

```
for (i = 0; i < n; ++i) {
    ch = getch();
    if (!isspace(ch)) {
        ...
        ...
        ...
    }
}
```

Yukarıdaki kod parçasında döngü içinde, klavyeden getch fonksiyonu ile değer atanan ch değişkeni boşluk karakteri değilse, bir takım deyimlerin icrası istenmiş. Yukarıdaki durum **continue** anahtar sözcüğüyle daha okunabilir hale getirilebilir :

```
for (i = 0; i < n; ++i) {
    ch = getch();
    if (isspace(ch))
        continue;
    ...
    ...
    ...
}
```

n kere dönen for deyimi kalıpları

```
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)
```

Bir döngüden çıkmak için döngü değişkeni ile oynamak kötü bir tekniktir. Bu programları okunabilirlikten uzaklaştırır. Bunun yerine **break** anahtar sözcüğü ile döngülerden çıkılmalıdır.

## Uygulama

Basamaklarının küpleri toplamı kendisine eşit olan 3 basamaklı sayıları bulan program.

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

int main()
{
    int i, j, k;
    int number = 100;

    clrscr();
    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 sayısı şartları sağlıyor\n", number);
                number++;
            }
    return 0;
}
```

Başka bir çözüm:

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

int main()
{
    int i, b1, b2, b3;

    clrscr();
    for (i = 100; i <= 999; ++i) {
        b1 = i / 100;
        b2 = i % 100 / 10;
        b3 = i % 10;
        if (b1 * b1 * b1 + b2 * b2 * b2 + b3 * b3 * b3 == i)
            printf("%d sayısı şartları sağlıyor\n", i);
    }
    return 0;
}
```

## Uygulama

Kendisine gönderilen iki tamsayının obeb ve okek değerlerini hesaplayan fonksiyonlar.

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

```
int obeb(int number1, int number2);
int okek(int number1, int number2);

int main()
{
    int x, y;
    int n = 20;

    clrscr();
    while (n-- > 0) {
        printf("iki tamsayı giriniz : ");
        scanf("%d%d", &x, &y);
        printf("obeb = %d\n", obeb(x, y));
        printf("okek = %d\n", okek(x, y));
    }
    getch();
    return 0;
}

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;
}

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;
}
```

### Uygulama

Kendisine gönderilen **int** türden argumanın faktöriyel değerini hesaplayan fonksiyon.

```
long 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;
}
```

### Uygulama

Birinci parametre değişkeninde tutulan tamsayının ikinci parametre değişkeninde tutulan tamsayı kuvvetini hesaplayan fonksiyon.

```

long power(int base, int exp)
{
    long result = 1;
    int k;

    for (k = 1; k <= exp; ++k)
        result *= base;
    return result;
}

```

ya da

```

long power(int base, int exp)
{
    long result = 1;

    while (exp-- > 0)
        result *= base;
}

```

### Uygulama

Kendisine gönderilen sayının asal sayı olup olmadığını test eden isprime fonksiyonu.

```

int isprime(long 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;
}

```

### Uygulama

Bölenlerinin toplamına eşit olan sayılara mükemmel tamsayı (perfect integer) sayı denir. Örneğin 28 sayısı bir mükemmel tamsayıdır.

$$1 + 2 + 4 + 7 + 14 = 28$$

Kendisine gönderilen bir argumanın mükemmel tamsayı olup olmadığını test eden is\_perfect fonksiyonu.

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

```
int is_perfect(int number);
```

```

int main()
{
    int k;
}

```

```
    for (k = 1000; k <= 9999; ++k)
        if (isperfect(k)) {
            printf("%d perfect\n");
            return 0;
        }
    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;
}
```

### Uygulama

**int** türden bir sayıyı çarpanlarına ayıran ve çarpanları küçükten büyüğe ekrana yazdıran `display_factors` fonksiyonunu.

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

```
void factors(int number);
```

```
/* test kodu : 1111 ile 1200 arasındaki sayıları çarpanlara ayırıyor */
```

```
int main()
{
    int k;

    for (k = 1111; k < 1200; ++k) {
        printf("%d sayısının çarpanları = ", k);
        factors(k);
        putchar('\n');
        getch();
    }
    return 0;
}
```

```
void factors(int number)
{
    int temp = number;
    int k;

    for (k = 2; k <= number / 2; ++k)
        while (temp % k == 0) {
            printf("%d ", k);
            temp /= k;
        }
}
```

### Uygulama

Klavyeden alınan cümleyi ekrana yazan ve cümle "." karakteriyle sonlanınca yazılan toplam kelime sayısını ve ortalama kelime uzunluğunu bulan program.

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

int main()
{
    int ch;
    int wordlength = 0, total = 0, wordcount = 0;

    clrscr();

    while ((ch = getch()) != '.') {
        putchar(ch);
        if (isspace(ch)) {
            total += wordlength;
            if (wordlength)
                wordcount++;
            wordlength = 0;
        }
        else
            wordlength++;
    }
    wordcount++;
    total += wordlength;
    printf("\n\ntoplam kelime sayısı = %d\n", wordcount);
    printf("ortalama uzunluk = %f\n", (double) total / wordcount);
    return 0;
}
```

### Uygulama

Klavyeden sürekli karakter alınmasını sağlayan, alınan karakterleri ekranda gösteren, ancak arka arkaya "xyz" karakterleri girildiğinde sonlanan bir program.

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

main()
{
    char ch;
    int total = 0;

    while (total < 3) {
        ch = getch();
        putchar(ch);
        if (ch == 'ç' && total == 0)
            total++;
        else if (ch == 'ı' && total == 1)
            total++;
        else if (ch == 'k' && total == 2)
            total++;
        else total = 0;
    }
    return 0;
}
```

### Çalışma Soruları

Prototipi aşağıda verilen isprimex fonksiyonunu tanımlayınız.

**int** isprimex(**long** number);

isprimex fonksiyonuna gönderilen argumanın asal olup olmadığı test edilecek, eğer sayı asal ise bu kez sayının basamak değerleri toplanarak elde edilen sayının asal olup olmadığı test edilecektir. Bu işlem sonuçta tek basamaklı bir sayı kalana kadar devam edecektir. Eğer en son elde edilen tek basamaklı sayı dahil tüm sayılar asal ise isprimex fonksiyonu 0 dışı bir değere geri dönecektir. Eğer herhangi bir kademede asal olmayan bir sayı elde edilirse fonksiyon 0 değerine geri dönecektir.

Yazdığınız fonksiyonu aşağıdaki main fonksiyonu ile test edebilirsiniz :

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

**int** isprimex(**long** number);

```
int main()
{
    long k;

    clrscr();

    for (k = 19000; k <= 20000; ++k)
        if (isprimex(k))
            printf("%ld \n", k);
    return 0;
}
```



## 13 . BÖLÜM : TÜR DÖNÜŞÜMLERİ

Bilgisayarların aritmetik işlemleri gerçekleştirmesinde bir takım kısıtlamalar söz konusudur.

Bilgisayarların aritmetik bir işlemi gerçekleştirmesi için (genellikle) işleme sokulan operandların uzunluklarının aynı olması (yani bit sayılarının aynı olması) ve aynı şekilde belleğe yerleştirilmiş olmaları gerekmektedir. Örnek vermek gerekirse, bilgisayar 16 bit uzunluğunda iki tam sayıyı doğrudan toplayabilir ama 16 bit uzunluğunda bir tam sayı ile 32 bit uzunluğundaki bir tamsayıyı ya da 32 bit uzunluğunda bir tamsayı ile 32 bit uzunluğunda bir gerçek sayıyı doğrudan toplayamaz.

C programlama dili, değişik türlerin aynı ifade içerisinde bulunmalarına izin verir. Yani tek bir ifadede bir tamsayı türünden değişken, bir **float** sabit 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 bir **int** sayıyla 32 bitlik bir **int** sayıyı topladığımızda, derleyici önce 16 bitlik **int** sayıyı 32 bit uzunluğunda bir **int** sayıya dönüştürecek ve ondan sonra toplama işlemini gerçekleştirecektir. Yine 16 bitlik bir **int** sayıyla 64 bitlik bir **double** sayıyı çarpmak istediğimizde derleyici önce **int** sayıyı 64 bitlik bir **double** sayıya dönüştürecektir. Bu tür dönüşümü daha komplekstir çünkü **int** ve **double** sayılar bellekte farklı şekillerde tutulurlar.

Bu tür dönüşümleri programcının kontrolü dışında otomatik olarak gerçekleştirilirler. Bu tür dönüşümlerine otomatik tür dönüşümleri (implicit type conversions) diyeceğiz. C dili programcıya herhangi bir değişkenin ya da sabitin türünü, bir operatör kullanarak değiştirme olanağı da verir. Programcı tarafından yapılan bu tip tür dönüşümlerine bilinçli tür dönüşümleri (explicit type conversions/type casts) diyeceğiz.

Önce otomatik tür dönüşümlerini inceleyeceğiz. Otomatik tür dönüşümleri ne yazık ki karmaşık yapıdadır ve iyi öğrenilmemesi durumunda programlarda hatalar kaçınılmazdır. Zira C dilinde 11 ana tür vardır. Herhangi bir hataya düşmemek için bunların her türlü ikili kombinezonu için nasıl bir otomatik tür dönüşümü yapılacağını çok iyi bilmemiz gerekir.

Aşağıda belirtilen 4 durumda mutlaka otomatik bir tür dönüşümü yapılacaktır:

1. Aritmetik ya da mantıksal bir ifadedenin operandları aynı türden değilse:

```
...
int x, y;
float fl;

if (fl * ix / iy) {
    ...
}
```

Bu durumda yapılan tür dönüşümlerine genel aritmetik tür dönüşümleri diyeceğiz.

2. Atama operatörü kullanıldığında atama operatörünün sağ tarafıyla sol tarafı aynı türden değilse:

```
double toplam;
long sayi1, sayi2;

toplam = sayi1 + sayi2;
```

Bu durumda yapılan tür dönüşümlerine atama tür dönüşümleri diyeceğiz.

3. Bir fonksiyon çağırılması sırasında kullanılan bir argumanın türü ile fonksiyonun ilgili parametre değişkeni aynı türden değilse:

```

double sonuc;
int sayi1, sayi2;
...
sonuc = pow(sayi1, sayi2)

double pow (double taban, double us)
{
    ...
}

```

4. Bir **return** ifadesinin türü ile ilgili fonksiyonun geri dönüş değerinin türü arasında farklılık varsa:

```

double funktion (int para1, int para2)
{
    ...
    return (para1 + para2)
}

```

3. ve 4. durumlar da bir atama işlemi olarak düşünülebilir. Zira fonksiyon çağırma ifadesindeki argümanlar parametre değişkenlerine kopyalanarak (atanarak) geçirilir. Yine **return** ifadesi de aslında geçici bölgeye yapılan bir atama işlemidir.

## Otomatik Tür Dönüşümleri

Otomatik tür dönüşümleri iki operand alan operatörlerin bulunduğu ifadelerde operandların türlerinin farklı olması durumunda uygulanır. Ve otomatik tür dönüşümü sonucunda farklı iki tür olması durumu ortadan kaldırılarak operandların 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. (biri **int** diğeri **double** türden). Bu durumda i ve d nesnelerinin türleri otomatik olarak aynı yapılır. Peki **int** tür mü **double** türe dönüştürülecek yoksa **double** tür mü **int** türe dönüştürülecek? Eğer **double** tür **int** türe dönüştürülse bilgi kaybı söz konusu olurdu (Çünkü bu durumda gerçek sayının en az virgülden sonraki kısmı kaybedilir).

C'de otomatik tür dönüşümleri mümkünse bilgi kaybı olmayacak şekilde yapılır.

Bu durumda bilgi kaybını engellemek için genel olarak küçük tür büyük türe dönüştürülür. (istisnaları var) Bu duruma terfi (promotion) diyeceğiz.

Kuralları detaylı olarak öğrenmek için oluşabilecek durumları iki ana durum altında inceleyelim:

1. Operandlardan birinin türü gerçek sayı türlerinden biriye.

Operandlardan birinin **long double** değerinin farklı bir tür olması durumunda diğer operand **long double** türüne çevrilir.

Operandlardan birinin **double** değerinin farklı bir tür olması durumunda diğer operand **double** türüne çevrilir.

Operandlardan birinin **float** değerinin farklı bir tür olması durumunda diğer operand **float** türüne çevrilir.

## 2. Operandlardan hiçbirisi gerçek sayı türlerinden değilse:

Eğer ifade içindeki operandlardan herhangi biri **char**, **unsigned char**, **short** ya da **unsigned short** türden ise aşağıdaki algoritma uygulanmadan önce bu türler **int** türüne dönüştürülür. Bu duruma "tam sayıya terfi" (integral promotion) denir.

Daha sonra aşağıdaki kurallar uygulanır:

Operandlardan birinin **unsigned long** değerinin farklı bir tür olması durumunda (**long**, **unsigned int**, **int**) diğer operand **unsigned long** türüne çevrilir.

Operandlardan birinin **long** türden değerinin farklı bir türden olması durumunda (**unsigned int**, **int**) diğer operand **long** türüne çevrilir.

Operandlardan biri **unsigned int** türden değerinin farklı bir türden olması durumunda (**int**) ikinci operand **unsigned int** türüne çevrilir.

## İstisnalar:

Eğer operandlardan biri **long int** değeri **unsigned int** türünden ise ve kullanılan sistemde bu türlerin uzunlukları aynı ise (UNIX ve Win 32 sistemlerinde olduğu gibi) her iki tür de **unsigned long int** türüne dönüştürülür.

Eğer operandlardan biri **int** değeri **unsigned short** türünden ise ve kullanılan sistemde bu türlerin uzunlukları aynı ise (DOS'da olduğu gibi) her iki tür de **unsigned int** türüne dönüştürülür.

Otomatik tür dönüşümleri ile ilgili kuralları aşağıdaki tabloda özetleyelim:

operandlardan herhangi biri	yapılacak dönüşüm
<b>char, short, unsigned char, unsigned short</b>	ilgili operand <b>int</b> türüne dönüştürülecek.

operand 1	operand 2	yapılacak dönüşüm
<b>long double</b>	<b>double, float, unsigned long, long, int, unsigned int</b>	2. operand <b>long double</b> türüne çevrilecek.
<b>double</b>	<b>float, unsigned long, long, int, unsigned int</b>	2. operand <b>double</b> türüne çevrilecek.
<b>float</b>	<b>unsigned long, long, int, unsigned int</b>	2. operand <b>float</b> türüne çevrilecek.
<b>unsigned long</b>	<b>long, int, unsigned int</b>	2. operand <b>unsigned long</b> türüne çevrilecek.
<b>long</b>	<b>int, unsigned int</b>	2. operand <b>long</b> türüne dönüştürülecek. (istisnai duruma dikkat!)
<b>unsigned int</b>	<b>int</b>	2. operand <b>unsigned int</b> türüne dönüştürülecek.

Fonksiyon çağırma ifadeleri de, operatörlerle birlikte başka ifadeleri oluşturuyorsa, otomatik tür dönüşümlerine neden olabilir. Zira geri dönüş değerine sahip olan fonksiyonlar için fonksiyonun çağırılma ifadesi, fonksiyonun geri dönüş değerini temsil etmektedir. Örneğin:

```
int i = 5;
...
pow(2, 3) + i
```

ifadesinde pow fonksiyonunun geri dönüş değeri **double** türden olduğu için, **int** türden olan i değişkeni de, işlemin yapılabilmesi için **double** türüne çevrilerek işleme sokulacaktır.

## Atama Tür Dönüşümleri

Bu tür dönüşümlerin çok basit bir kuralı vardır: Atama operatörünün sağ tarafındaki tür, atama operatörünün sol tarafındaki türe otomatik olarak dönüştürülür.

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 yapılacak ve **double** hale getirilen 5 sabiti (5.) leftx değişkenine atanacaktır.

Aşağıda 16 bitlik sistemler için bazı örnekler verilmektedir :

TÜR	desimal	hex	dönüştürülecektür	hex	desimal
<b>int</b>	138	0x008A	<b>long int</b>	0x0000008A	138L
<b>char</b>	'd' (100)	0x64	<b>int</b>	0x0064	100
<b>int</b>	-56	0xFFC8	<b>long int</b>	0xFFFFFFFFC8	-56L
<b>char</b>	'\x95' (-07)	0x95	<b>int</b>	0xFF95	-107
<b>unsigned int</b>	45678	0xB26E	<b>long int</b>	0xFFFFB26EL	-19858L
<b>char</b>	'0' (48)	0x30	<b>long int</b>	0x00000030L	30L

Negatif olan bir tamsayı küçük türden büyük türe dönüştürüldüğünde sayının yüksek anlamlı bitleri negatifliğin korunması amacıyla 1 ile beslenmektedir. (Hex olarak F ile)

Derleyici tarafından yapılan otomatik, atama tür dönüşümlerinde, büyük türün küçük türe dönüştürülmesi durumunda bilgi kaybı söz konusu olabilir.

Aşağıdaki basit kuralları verebiliriz:

Eğer atama operatörünün her iki tarafı da tam sayı türlerinden ise (**char**, **short**, **int**, **long**), atama operatörünün sağ tarafının daha büyük bir türden olması durumunda bilgi kaybı olabilir. Bilgi kaybı ancak, atama operatörünün sağ tarafındaki değer, sol taraftaki türün sınırları içinde olmaması durumunda söz konusu olacaktır. Bilgi kaybı yüksek anlamlı byte'ların kaybolması şeklinde ortaya çıkar. Örnek:

```
long m = 0x12345678;
int y;
```

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

Yukarıdaki örnekte **int** türden olan y değişkenine **long** türden bir değişkenin değeri atanmıştır. (16 bitlik bir sistemde örneğin DOS altında çalıştığımızı düşünüyoruz.). DOS altında **int** türü için sayı sınırları -32768 +32767 değerleridir. Bu sayılarda iki byte'lık bir alan için işaretli olarak yazılabilecek en büyük ve en küçük sayılardır. Hex gösterimde her bir hex digit 4 bite ve her iki hex digit 1 byte alana tekabül edecektir. Dolayısıyla 0x12345678 sayısı 8 hex digit yani 4 byte uzunluğunda bir sayıdır. Oysa atamanın yapılacağı taraf **int** türündendir ve bu tür max. 4 hex digit (2 byte) uzunlukta olabilmektedir. Bu durumda m değişkenine ilişkin değerin yüksek anlamlı 2 byte'ı yani (4 hex digiti) kaybedilecektir. Atama işleminden sonra, printf fonksiyonuyla y değişkeninin değeri yazdırıldığında (hex gösterimli) ekrana 5678 değerinin yazdırıldığı görülecektir.

Atama operatörünün sağ tarafı gerçek sayı türlerinden bir türden ise (**float**, **double**, **long double**) ve sol tarafı ise tam sayı türlerinden birine ait ise, bu herşeyden önce "undefined behaviour" (şüpheli kod) durumudur ve bu durumun olduğu kodlardan kesinlikle kaçınmak gerekir. Ama derleyicilerin hemen hemen hepsi bu durumda aşağıdaki şekilde tür dönüşümü yaparlar:

Sağ taraftaki gerçek sayı türündeki değer nokta içeriyorsa, önce kesir kısmı kaybedilir. Kesir kısmı kaybedildikten sonra kalan değer eğer sol taraf türünün sınırları içinde kalıyorsa daha fazla bir bilgi kaybı olmaz, fakat sol taraf türünün sınırları aşıyorsa ilave bir bilgi kaybı daha olur ve yüksek anlamlı byte'lar kaybedilir.

Örnek:

```
double y = 234.12;
int x;
```

```
x = y;
printf("x = %d\n", x );           /* x değişkenine 234 değeri atanacaktır */
```

```
y = 32768.1;
x = y;
printf("x = %d\n", x );           /* x değişkenine -32768 değeri atanacaktır */
```

### Tam Sayıya Terfi (integral promotion)

Daha önce de açıklandığı gibi "integral promotion" bir ifade içinde bulunan **char**, **unsigned char**, **short**, **unsigned short** türlerinin, ifadenin derleyici tarafından değerlendirilmesinden önce, otomatik olarak **int** türe dönüştürülmeleri anlamına gelir.

Peki dönüşüm **signed int** türüne mi **unsigned int** türüne mi yapılacaktır, bunu nasıl bileceğiz?

Genel kural şudur: Tür dönüşümüne uğrayacak türün bütün değerleri **int** türü ile 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 terfi ettirilirken **unsigned int** türüne dönüştürülür.

Eğer tam sayıya terfi ettirilecek değer, **signed char**, **unsigned char** ya da **short (signed short)** türlerinden ise dönüşüm **signed int** türüne yapılacaktır.

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 buna 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 **long** türü sayı sınırları içinde kalmaktadır. Oysa bilgi kaybı atama işleminden önce gerçekleşir. 1000 ve 2000 **int** türden sabitlerdir, işleme sokulduklarında çarpma operatörünün de ürettiği değer **int** türden olacaktır. Bu durumda 2 byte uzunlukta olan **int** türü 2000000 sayısını tutamayacağı için yüksek anlamlı byte kaybedilecektir. 2000000 hex sistemde 0x1E8480 olarak gösterilebilir. Yüksek anlamlı byte kaybedilince işlem sonucu 0x8480 olarak bulunur. 0x8480 negatif bir sayıdır. İkiye tümleyenini alırsak

```
0x8480      1000 0100 1000 0000
ikiye tümleyenini  0111 1011 1000 0000 (0x7B80 = 31616)
```

Görüldüğü gibi işlem sonucu üretilecek değer -31616 dir. Bu durumda x değişkeninin türü **long** da olsa, atanacak değer -31616 olacaktır.

## Fonksiyon Çağırılarında Tür Dönüşümü

Daha önce söylendiği gibi bir fonksiyona gönderilecek argümanlarla, bu argümanları tutacak fonksiyonun parametre değişkenleri arasında tür farkı varsa otomatik tür dönüşümü gerçekleşecek ve argümanların türü, parametre değişkenlerinin türlerine dönüştürülecektir. Ancak bu tür dönüşümünün gerçekleşmesi için, derleyicinin fonksiyon çağırma ifadesine gelmeden önce fonksiyonun parametre değişkenlerinin türleri hakkında bilgi sahibi olması gerekecektir. Derleyici bu bilgiyi iki ayrı şekilde elde edebilir:

1. Çağrılan fonksiyon çağırılan fonksiyondan daha önce tanımlanmışsa, derleyici fonksiyonun tanımlanmasından parametre değişkenlerinin türünü belirler.
2. Fonksiyonun prototip bildirimi yapılmışsa derleyici parametre değişkenlerinin türü hakkında önceden bilgi sahibi olur.

Örnek:

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

```
double func(double x, double y)
{
    ...
}
```

```
int main()
{
    int a, b;
    ...
    func(a, b);
    return 0;
}
```

Yukarıdaki örnekte main fonksiyonu içinde çağırılan func fonksiyonuna argüman olarak **int** türden olan a ve b değişkenlerinin değerleri gönderilmiştir. Fonksiyon tanımlaması çağırma ifadesinden önce yer aldığı için **int** türden olan a ve b değişkenlerinin değerleri, **double** türe dönüştürülerek func fonksiyonunun parametre değişkenleri olan x ve y değişkenlerine aktarılırlar. func fonksiyonunun main fonksiyonundan daha sonra tanımlanması durumunda otomatik tür dönüşümünün yapılabilmesi için prototip 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 çağırılan fonksiyon çağırılan fonksiyondan daha sonra tanımlanmışsa, ve fonksiyon prototipi bildirimi yapılmamışsa (tabi bu durumda derleme zamanında error oluşmaması için

fonksiyonun **int** türden bir geri dönüş değerine sahip olması gerekecektir) tür dönüşümü gerçekleşebilecek mi?

Bu durumda fonksiyonun parametre değişkenlerinin türleri hakkında derleyici bilgi sahibi olamayacağı için, fonksiyona gönderilen argümanları, default argüman dönüşümü denilen dönüşüme tabi tutar. Default argüman dönüşümü şu şekilde olur:

**char** ya da **short** türünden olan argümanlar tamsayıya terfi ettirilir. (integral promotion). **float** türünden olan argümanlar **double** türüne dönüştürülür. Bunun dışındaki türlerden olan argümanların tür dönüştürülmesine tabi tutulmaz.

## 14 . BÖLÜM : switch DEYİMİ

**switch** deyimi ile bir ifadenin farklı değerleri için farklı işler yapmak mümkün hale gelir. Özellikle **else if** yapılarına bir alternatif olarak kullanılır.

Genel biçimi:

```
switch (ifade) {
    case ifade_1:
    case ifade_2:
    case ifade_3:
    .....
    case ifade_n:
    default:
}
```

**switch**, **case** ve **default** anahtar sözcüklerdir.

### switch Deyiminin Çalışması

Derleyici **switch** parantezi içerisindeki ifadenin sayısal değerini hesaplar. Bu sayısal değere eşit bir **case** ifadesi olup olmadığını araştırır. Bulursa programın akışı o **case** ifadesine geçirilir. Artık program buradan akarak ilerler. Eğer **switch** parantezi içindeki ifadenin sayısal değeri hiçbir **case** ifadesine eşit değilse akış **default** anahtar sözcüğünün bulunduğu kısma geçerilir. **default** anahtar sözcüğünün bulunması zorunlu değildir. Uygun bir **case** ifadesi yoksa ve **default** anahtar sözcüğü de yoksa programın akışı **switch** deyiminin içine girmez ve **switch** deyimi dışındaki ilk deyim ile devam eder.

Örnek:

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

```
int main()
{
    int a;

    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");
        default: printf("hiçbiri\n");
    }
    return 0;
}
```

Burada unutulmaması gereken noktayı tekrar edelim: Uygun bir **case** ifadesi bulunduğunda programın akışı buraya geçirilir. Yalnızca bulunan **case** ifadesini izleyen deyim ya da deyimlerin icra edilmesi söz konusu değildir. Uygun **case** ifadesini izleyen deyimlerin icrasından sonra programın akışı daha aşağıda bulunan **case** ifadelerini izleyen deyimlerin icra edilmesiyle devam eder.

Yukarıdaki örnekte scanf fonksiyonu ile a değişkenine 1 atandığını varsayalım. Bu durumda program çıktısı şu şekilde olacaktır :

```
bir
iki
```



üç  
dört  
hiçbiri

Eğer uygun **case** ifadesi bulunduğunda yalnızca bu ifadeye ilişkin deyim(ler)in icra edilmesini istersek **break** anahtar sözcüğünden faydalanırız. **break** anahtar sözcüğü döngülerden olduğu gibi **switch** deyimlerinden de çıkmamıza olanak verir.

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

```
int main()
{
    int a;

    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;
        default : printf("hiçbiri\n"); break;
    }
    return 0;
}
```

Uygulamalarda da **switch** deyiminde çoğunlukla her **case** ifadesi için bir **break** anahtar sözcüğünün kullanıldığını görürüz. (Tabi böyle bir zorunluluk yok!)

**case** ifadelerini izleyen ":" atomundan sonraki deyimler istenildiği kadar uzun olabilir. **case** ifadelerinin sıralı olması ya da **default** anahtar sözcüğünün en sonda olması gibi bir zorunluluk yoktur. **default** herhangi bir yere yerleştirilebilir. **default** ifadesi nerede olursa olsun, programın akışı ancak uygun bir **case** ifadesi bulunamazsa **default** içine girecektir.

Yukarıda da anlatıldığı gibi **switch** parantezi içerisindeki ifadenin sayısal değerine eşit bir **case** ifadesi bulunana kadar derleyici derleme yönünde (yani yukarıdan aşağıya doğru) tüm **case** ifadelerini sırasıyla kontrol eder. Bu yüzden en yüksek olasılığa ilişkin (biliniyorsa) **case** ifadesinin en başa (diğerlerinden önce) yerleştirilmesi iyi tekniktir.

**switch** kontrol deyimi belli ölçülerde **else if** (**else if** ladder - cascaded **if**) yapılarıyla karşılanabilir. Yani **switch** deyimi olmasaydı yapmak istediklerimizi **else if** deyimi ile yapabiliirdik. Ancak bazı durumlarda **else if** yapısı yerine **switch** deyimi kullanmak okunabilirliği artırmaktadır. Örneğin aşağıdaki iki kod işlevsel olarak birbirinin aynıdır:

<pre>if ( a == 1) {     ifade_1;     ifade_2; } else if (a == 2) {     ifade_3;     ifade_4; } else if (a == 4) {     ifade_5; } else {     ifade_6;     ifade_7; }</pre>	<pre>switch (a) {     case 1:         ifade_1;         ifade_2;     case 2:         ifade_3;         ifade_4;     case 4:         ifade_5;     default:         ifade_6;         ifade_7; }</pre>
---	---

örnek :

Bir tarih bilgisini (ay, gün, yıl) yazı ile gg - yazı - yyyy (12 - Ağustos - 2002 gibi) biçiminde ekrana yazan, prototipi **void** display\_date(**int** day, **int** month, **int** year) biçiminde olan fonksiyon. .

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

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

```
void display_date(int day, int month, int year);
```

```
int main()
```

```
{
```

```
    int d, m, y;
```

```
    int k;
```

```
    for (k = 0; k < 10; ++k) {  
        scanf("%d%d%d", &d, &m, &y);  
        display_date(d, m, y);  
        putchar('\n');
```

```
    }
```

```
    getch();
```

```
    return 0;
```

```
}
```

```
void display_date(int day, int month, int year)
```

```
{
```

```
    printf("%d - ", day);
```

```
    switch (month) {
```

```
        case 1: printf("Ocak"); break;
```

```
        case 2: printf("Subat"); break;
```

```
        case 3: printf("Mart"); break;
```

```
        case 4: printf("Nisan"); break;
```

```
        case 5: printf("Mayis"); break;
```

```
        case 6: printf("Haziran"); break;
```

```
        case 7: printf("Temmuz"); break;
```

```
        case 8: printf("Ağustos"); break;
```

```
        case 9: printf("Eylul"); break;
```

```
        case 10: printf("Ekim"); break;
```

```
        case 11: printf("Kasim"); break;
```

```
        case 12: printf("Aralik"); break;
```

```
    }
```

```
    printf(" %d\n", year);
```

```
}
```

Birden fazla **case** ifadesi için aynı işlemlerin yapılması şöyle sağlanabilir.

```
case 1:
```

```
case 2:
```

```
case 3:
```

```
    ifade1;
```

```
    ifade2;
```

```
    ifade3;
```

```
    break;
```

```
case 4:
```

Bunu yapmanın daha kısa bir yolu yoktur. Bazı programcılar bu yapıyı şu şekilde de yazarlar:

**case 1: case 2: case 3: case 4: case 5:**

ifade1:

ifade2;

Her **switch** deyimini **else if** yapısıyla karşılayabiliriz, ama her **else if** yapısını **switch** deyimiyle karşılayamayız. **switch** değiminin parantezi içindeki ifadenin türü tamsayı olmak zarundadır. **case** ifadeleri de tamsayı sabit ifadesi olmak zarundadır. **switch** deyimi, tamsayı türünden bir ifadenin değerinin değişik tamsayı değerlerine eşitliğinin test edilmesi ve eşitlik durumunda farklı işlerin yapılması için kullanılır. Oysa **else if** yapısında 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** yapısını **switch** deyimiyle karşılayamayız.

Şöyle bir soru aklımıza gelebilir : Madem her **switch** yapısını **else if** yapısıyla karşılayabiliyoruz, o zaman **switch** deyimine ne gerek var? Yani **switch** deyiminin varlığı C diline ne kazandırıyor?

**switch** deyimi bazı durumlarda **else if** yapısına göre çok daha okunabilir bir yapı oluşturmaktadır, yani **switch** deyiminin kullanılması herşeyden önce kodun daha kolay okunabilmesi ve anlamlandırılması açısından bir avantaj sağlayacaktır.

**case** ifadelerinin tam sayı türünden (integral types) sabit ifadesi olması gerekmektedir. Bilindiği gibi sabit ifadeleri derleme aşamasında derleyici tarafından net sayısal değerlere dönüştürülebilir.

**case 1 + 3:** /\* legal \*/

mümkün çünkü 1 + 3 sabit ifadesi ama ,

**case x + 5:** /\* error \*/

çünkü sabit ifadesi değil. Derleyici derleme aşamasında sayısal bir değer hesaplayamaz.

**case 'a' :** /\* geçerli, bir sabit ifadesidir. \*/

**case 3.5 :** /\* gerçek sayı (double) türünden olduğu için geçerli değildir. \*/

**case** ifadelerini izleyen deyimlerin 15 - 20 satırdan uzun olması okunabilirliği zayıflatır. Bu durumda yapılacak işlemlerin fonksiyon çağırma biçimine dönüştürülmesi doğru bir tekniktir.

```
switch (x) {
    case ADDREC:
        addrec();
        break;
    case DELREC:
        delrec();
        break;
```

```

    case FINDREC:
        findrec();
        break;
}

```

gibi. (Bu örnekte **case** ifadelerindeki ADDREC, DELREC vs. daha önce tanımlanmış sembolik sabitlerdir.)

Sembolik sabitler derleme işleminden önce önışlemci tarafından değiştirilecekleri için, **case** ifadelerinde yer alabilirler :

```

#define TRUE          1
#define FALSE         0
#define UNDEFINED     2
...

```

```

case TRUE:
case FALSE:
case UNDEFINED:

```

Yukarıdaki ifadeler, sembolik sabitlerin kullanıldığı geçerli birer **case** ifadesidir.

**char** türünün de bir tamsayı türü olduğunu hatırlatarak **case** ifadelerinin doğal olarak karakter sabitleri de içerebileceğini belirtelim.

```

char ch;

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 olmaz. Bu durum derleme zamanında hata oluşturur.

**switch** deyimi başka bir **switch** deyiminin, **do while** ya da **for** deyiminin gövdesini oluşturabilir. Bu durumda dışarıdaki döngünün bloklanmasına gerek olmayacaktır, çünkü **switch** deyimi gövdeyi oluşturan tek bir deyim olarak ele alınacaktır :

```

...
for ((ch = getch()) != ESC)
    switch (rand() % 7 + 1) {
        case 1: printf("Pazartesi"); break;
        case 2: printf("Salı"); break;
        case 3: printf("Çarşamba"); break;
        case 4: printf("Perşembe"); break;
        case 5: printf("Cuma"); break;
        case 6: printf("Cumartesi"); break;
        case 7: printf("Pazar");
    }

```

Yukarıdaki kod parçasında **switch** deyimi dıştaki **for** döngüsünün gövdesini oluşturmaktadır, ve **switch** deyimi tek bir deyim (kontrol deyimi) olarak ele alınacağından, dıştaki **for** döngüsünün bloklanmasına gerek yoktur (tabi bloklama bir hataya neden olmayacaktır.) Ancak **case** ifadeleri içinde yer alan **break** anahtar sözcüğüyle yalnızca **switch** deyiminden çıkılır, **for** döngüsünün dışına çıkmak için **case** ifadesi içinde **goto** anahtar sözcüğü kullanılmalıdır.

## Uygulama

Kendisine gönderilen bir tarihi ingilizce (15<sup>th</sup> Aug. 2000 ) formatında ekrana yazan fonksiyonun.

```
void displaydate(int day, int month, int year);
```

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

```
void dispdate(int day, int month, int year);
```

```
int main()
{
    int day, month, year;
    int n = 20;

    clrscr();
    while (n-- > 0) {
        printf("gun ay yil olarak bir tarih giriniz : ");
        scanf("%d%d%d", &day, &month, &year);
        dispdate(day, month, year);
        putchar('\n');
    }
    return 0;
}
```

```
void dispdate(int day, int month, int year)
```

```
{
    printf("%2d", 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);
}
```

## Uygulama

Kendisine gönderilen bir tarihin (gün, ay, yıl ) o yılın kaçınıcı günü olduğunu bulan fonksiyon.

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

```
int isleap(int year);
int dayofyear(int day, int month, int year);
```

```
int main()
{
    int n = 20;
    int day, month, year;

    clrscr();
    while (n-- > 0) {
        printf("gun ay yil olarak bir tarih giriniz : ");
        scanf("%d%d%d", &day, &month, &year);
        printf("%d yilinin %d. gunudur.\n", year, dayofyear(day, month, year));
    }
    return 0;
}
```

```
int isleap(int year)
{
    return (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
}
```

```
int dayofyear(int day, int month, int year)
{
    int yearday = day;

    switch (month - 1) {
        case 11: yearday += 30;
        case 10: yearday += 31;
        case 9 : yearday += 30;
        case 8 : yearday += 31;
        case 7 : yearday += 31;
        case 6 : yearday += 30;
        case 5 : yearday += 31;
        case 4 : yearday += 30;
        case 3 : yearday += 31;
        case 2 : yearday += isleap(year) ? 29 : 28;
        case 1 : yearday += 31;
    }
    return yearday;
}
```

## Uygulama

01.01.1900 – 31.12.2000 tarihleri arasında geçerli rasgele tarih üreterek ekrana yazan fonksiyon.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>
```

```
int isleap(int year);
void randomdate(void);
```

```

int main()
{
    int n = 20;

    clrscr();
    srand(time(NULL));
    while (n-- > 0) {
        randomdate();
        putchar('\n');
    }
    return 0;
}

int isleap(int year)
{
    if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
        return year;
    return 0;
}

void randomdate(void)
{
    int day, month, year;
    int monthdays;

    year = rand() % 101 + 1900;
    month = rand() % 12 + 1;

    switch (month) {
        case 4:
        case 6:
        case 9:
        case 11: monthdays = 30; break;
        case 2: monthdays = isleap(year) ? 29 : 28; break;
        default : monthdays = 31;
    }
    day = rand() % monthdays + 1;

    printf("%02d - %02d - %d", day, month, year);
}

```

## Uygulama

**int** make\_char\_eng(**char** ch); fonksiyonu. Kendisine gönderilen karakter türkçe'ye özel karakterlerden birisi ise (ç, ı, ğ, ü vs.) bu karakterlerin ingilizce benzerine aksi halde karakterin kendisine geri dönen fonksiyon:

```

#include <stdio.h>
#include <conio.h>

int make_char_eng(char ch);

int main()
{
    int ch;

    while ((ch = getch()) != 'q')
        putchar(make_char_eng(ch));
    return 0;
}

```

```
}  
  
int make_char_eng(char ch)  
{  
    switch (ch) {  
        case 'ç': return 'c';  
        case 'ğ': return 'g';  
        case 'ı': return 'i';  
        case 'ö': return 'o';  
        case 'ş': return 's';  
        case 'ü': return 'u';  
        case 'Ç': return 'C';  
        case 'Ğ': return 'G';  
        case 'İ': return 'I';  
        case 'Ö': return 'O';  
        case 'Ş': return 'S';  
        case 'Ü': return 'U';  
        default : return ch;  
    }  
}
```

### Çalışma Sorusu:

İki basamaklı bir sayıyı yazı olarak ekrana basan num\_text isimli fonksiyonu yazınız. Fonksiyonun prototipi :

```
void num_text(int n);
```

şeklindedir.

Fonksiyonun parametre değişkeni yazılacak olan iki basamaklı sayıdır. Aşağıdaki programla test edebilirsiniz :

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

```
void num_text(int n);
```

```
int main()  
{  
    for (k = 10; k < 100; ++k) {  
        num_text(k);  
        putchar('\n');  
        getch();  
    }  
    return 0;  
}
```



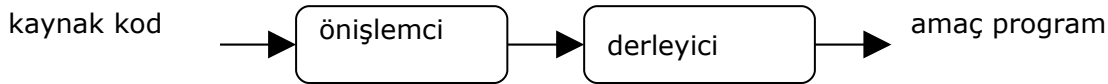
## 15 . BÖLÜM : ÖNİŞLEMCİ KAVRAMI

Şimdiye kadar tek bir yapı halinde ele aldığımız C derleyicileri aslında, iki ayrı modülden oluşmaktadır:

1. Önışlemci Modülü
2. Derleme Modülü

İsim benzerliği olsa da önışlemcinin bilgisayarın işlemcisi ya da başka bir donanımsal elemanıyla hiçbir ilgisi yoktur, önışlemci tamamen C derleyicisine ilişkin yazılımsal bir kavramdır.

Önışlemci, kaynak program üzerinde birtakım düzenlemeler ve değişiklikler yapan bir ön programdır. Önışlemci programının bir girdisi bir de çıktısı vardır. Önışlemcinin girdisi kaynak programın kendisidir. Önışlemcinin çıktısı ise derleme modülünün girdisini oluşturur. Yani kaynak program ilk aşamada önışlemci tarafından ele alınır. Önışlemci modülü kaynak programda çeşitli düzenlemeler ve değişiklikler yapar, daha sonra değiştirilmiş ve düzenlenmiş olan bu kaynak program derleme modülü tarafından amaç koda dönüştürülür.



C programlama dilinde # ile başlayan bütün satırlar önışlemciye verilen komutlardır. # karakterinin sağında bulunan sözcükler ki bunlara önışlemci komutları denir (preprocessor directives), önışlemciye ne yapması gerektiğini anlatır. Örneğin:

```
#include
#define
#if
#ifdef
#endif
```

hepsi birer önışlemci komutudur.

Önışlemci komutları derleyici açısından anahtar sözcük değildir # karakteriyle birlikte anlam kazanırlar, yani istersek include isimli bir değişken tanımlayabiliriz ama bunun okunabilirlik açısından iyi bir fikir olmadığını söyleyebiliriz. Önışlemci komutlarını belirten sözcükler ancak sol taraflarındaki # karakteriyle kullanıldıkları zaman özel anlam kazanırlar ki bunlara önışlemci anahtar sözcükleri de diyebiliriz.

Önışlemci amaç kod oluşturmaya yönelik hiçbir iş yapmaz. Önışlemci kaynak programı # içeren satırlardan arındırır. Derleme modülüne girecek programda artık # içeren satırlar yer almayacaktır.

Bir çok önışlemci komutu olmakla birlikte şimdilik yalnızca #include komutu ve #define önışlemci komutlarını inceleyeceğiz. Geriye kalan önışlemci komutlarını da ileride detaylı olarak inceleyeceğiz.

### #include önışlemci komutu

genel kullanım biçimi:

```
#include <dosya ismi>
```

ya da

```
#include "dosya ismi"
```

#include, ilgili kaynak dosyanın derleme işlemine dahil edileceğini anlatan bir önilemci komutudur. Bu komut ile önilemci belirtilen dosyayı diskten okuyarak komutun yazılı olduğu yere yerleştirir. (metin editörlerindeki copy – paste işlemi gibi)

#include komutundaki dosya ismi iki biçimde belirtilebilir:

1. Açıl parantezlerle:

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

2. Çift tırnak içerisinde:

```
#include "stdio.h"
#include "deneme.c"
```

Dosya ismi eğer açıl parantezler içinde verilmişse, sözkonusu dosya önilemci tarafından yalnızca önceden belirlenmiş bir dizin içerisinde aranır. Çalıştığımız 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. Genellikle standart başlık dosyaları önilemci tarafından belirlenen dizinde olduğundan, açıl parantezler ile kaynak koda dahil edilirler.

Dosya ismi iki tırnak içine yazıldığında önilemci ilgili dosyayı önce çalışılan dizinde (current directory), burada bulamazsa bu kez de sistem ile belirlenen dizinde arayacaktır. Örneğin:

```
C:\SAMPLE
```

dizinde çalışıyor olalım.

#include "strfunc.h" komutu ile önilemci strfunc.h dosyasını önce C:\SAMPLE dizinde arar. Eğer burada bulamazsa bu kez sistem ile belirlenen dizinde arar. Programcıların kendilerinin oluşturdukları başlık dosyaları genellikle sisteme ait dizinde olmadıkları için iki tırnak içinde kaynak koda dahil edilirler.

#include ile koda dahil edilmek istenen dosya ismi dosya yolu da (path) de içerebilir:

```
#include <sys\stat.h>
#include "c:\headers\myheader.h"
....
gibi.
```

Bu durumda açıl parantez ya da iki tırnak gösterimleri arasında bir fark yoktur. Her iki gösterimde de önilemci ilgili dosyayı yalnızca yolun belirttiği dizinde arar.

#include komutu ile yalnızca başlık dosyalarının koda dahil edilmesi gibi bir zorunluluk yoktur. Herhangi bir kaynak dosya da bu yolla kaynak koda dahil edilebilir.

prog.c

```
#include "topla.c"
```

```
int main()
{
    toplam = a + b;
    printf("%d", toplam);
    return 0;
}
```

topla.c

```
int a = 10;
int b = 20;
int toplam;
```

Yukarıdaki örnekte #include "topla.c" komutu ile önışlemci komutunun bulunduğu yere topla.c dosyasının içeriğini yerleştirecek böylece prog.c dosyası derlendiğinde herhangi bir hata oluşmayacaktır.

Çünkü artık derleme işlemine girecek dosya:

```
int a = 10;
int b = 20;
int toplam;

int main()
{
    toplam = a + b;
    printf("%d", toplam);
    return 0;
}
```

şeklinde olacaktır.

#include komutu kaynak programın herhangi bir yerinde bulunabilir. Fakat standart başlık dosyaları gibi, içerisinde çeşitli bildirimlerin bulunduğu dosyalar için en iyi yer kuşkusuz programın en tepesidir.

#include komutu içiçe geçmiş (nested) bir biçimde de bulunabilir. Örneğin çok sayıda dosyayı kaynak koda dahil etmek için şöyle bir yöntem izleyebiliriz.

<pre>ana.c  #include "project.h"  int main() {     ... }</pre>	<pre>project.h  #include &lt;stdio.h&gt; #include &lt;conio.h&gt; #include &lt;stdlib.h&gt; #include &lt;time.h&gt;</pre>
--	---

ana.c dosyası içerisine yalnızca project.h dahil edilmiştir. Önışlemci bu dosyayı kaynak koda dahil ettikten sonra yoluna bu dosyadan devam edecektir.

## #define Önışlemci Komutu

#define önışlemci komutu text editörlerindeki bul ve değıştir özelliği (find & replace) gibi çalışır. Kaynak kod içerisindeki bir yazıyı başka bir yazı ile değıştirmek için kullanılır.

Önişlemci define anahtar sözcüğünden sonra boşlukları atarak ilk boşluksuz yazı kümesini elde eder. (Buna STR1 diyelim.) Daha sonra boşlukları atarak satır sonuna kadar olan tüm boşlukların kümesini elde eder. (Buna da STR2 diyelim) Kaynak kod içerisinde STR1 yazısı yerine STR2 yazısını yerleştirir. (STR2 yazısı yalnızca boşluktan ibaret de olabilir. Bu durumda STR1 yazısı yerine boşluk yerleştirilir, yani STR1 yazısı silinmiş olur.) Örnekler :

```
#define SIZE    100
```

önişlemci komutuyla, önişlemci kaynak kod içerisinde gördüğü tüm SIZE sözcükleri yerine 100 yerleştirecektir. Derleme modülüne girecek kaynak programda artık SIZE sözcüğü hiç yer almayacaktır.

#define kullanılarak bir yazının bir sayısal değerle yer değiştirmesinde kullanılan yazıya "sembolik sabit" (symbolic constants) denir. Sembolik sabitler nesne değildir. Derleme modülüne giren kaynak kodda sembolik sabitler yerine bunların yerini sayısal ifadeler almış olur.

Sembolik sabitler basit makrolar (simple makro) olarak da isimlendirirler ama biz "sembolik sabit" demeyi yeğliyoruz.

Sembolik sabitlerin kullanılmasında 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 50 değeri atanır. Ancak ön işlemci komutunu

```
#define MAX    (10 + 20)
```

şeklinde yazsaydık, result değişkenine 60 değeri atanmış olurdu.

Bir sembolik sabit başka bir sembolik sabit tanımlamasında kullanılabilir. Örneğin:

```
#define MAX    100
#define MIN    (MAX - 50)
```

Önişlemci " " içerisindeki yazılarda değişiklik yapmaz. (stringlerin bölünemeyen atomlar olduğunu hatırlayalım.) Yer değiştirme işlemi büyük küçük harf duyarlılığı ile yapılır. Ancak sembolik sabitler geleneksel olarak büyük harf ile isimlendirilirler. Bunun nedeni, kodu okuyan kişinin değişkenlerle sembolik sabitleri ayırt edebilmesine yardımcı olmaktır. Bilindiği gibi geleneksel olarak değişken isimlendirmesinde C dilinde ağırlıklı olarak küçük harfler kullanılmaktadır.

#define komutu ile ancak değişkenler ve anahtar sözcükler yer değiştirilebilir. Sabitler ve operatörler yer değiştiremez.

Aşağıdaki #define önişlemci komutları geçerli değildir:

```
#define + -
#define 100 200
```

Sembolik sabitler C dilinin değişken isimlendirme kurallarına uygun olarak isimlendirilir:

```
#define BÜYÜK 10
```

tanımlamasında hata oluşur. (Hata tanımlama satırında değil sembolik sabitin kod içinde kullanıldığı yerlerde oluşur. Tanımlanan sembolik sabit kod içinde kullanılmazsa hata da oluşmayacaktır.)

Önişlemci `#include` komudu ile kaynak koda dahil edilen dosyanın içerisindeki önişlemci komutlarını da çalıştırır. İçinde sembolik sabit tanımlamaları yapılmış bir dosya `#include` komudu ile kaynak koda dahil edildiğinde, bu sembolik sabitler de kaynak kod içinde tanımlanmış gibi geçerli olur.

`#define` sembolik sabit tanımlamalarında stringler de kullanılabilir :

```
#define HATA_MESAJI "DOSYA AÇILAMIYOR \n"
```

```
...
printf(HATA_MESAJI);
...
```

Sembolik sabit tanımlamasında kullanılacak string uzunsa okunabilirlik açısından birden fazla satıra yerleştirilebilir. Bu durumda son satır dışında diğer satırların sonuna `"\"` karakteri yerleştirilmelidir.

Önişlemci değiştirme işlemini yaptıktan sonra `#define` komutlarını koddan çıkartır.

Okunabilirlik açısından tüm sembolik sabit tanımlamaları alt alta getirecek şekilde yazılmalıdır. Seçilen sembolik sabit isimleri kodu okuyan kişiye bunların ne amaçla kullanıldığı hakkında fikir vermelidir.

C'nin başlık dosyalarında bazı sembolik sabitler tanımlanmıştır. Örneğin `stdio.h` içerisinde

```
#define NULL 0
```

biçiminde bir satır vardır. Yani `stdio.h` dosyası koda dahil edilirse `NULL` sembolik sabiti 0 yerine kullanılabilir. `math.h` içinde de pek çok matematiksel sabit tanımlanmıştır.

Bir sembolik sabitin tanımlanmış olması kaynak kod içerisinde değiştirilebilecek bir bilginin olmasını zorunlu hale getirmez. Tanımlanmış bir sembolik sabitin kaynak kod içerisinde kullanılmaması hataya yol açmaz.

### **#define Önişlemci Komutu Neden Kullanılır**

1. Okunabilirliği ve algılabilişliliği artırır. Bir takım sabitlere onların ne amaçla kullanıldığını anlatan yazılar karşılık getirilirse programa bakan kişiler daha iyi anlamlandırır.

```
#define PERSONEL_SAYISI 750
```

```
int main()
{
    if (x == PERSONEL_SAYISI)
    ...
    return 0;
}
```

2. Bir sabitin program içerisinde pekçok yerde kullanıldığı durumlarda değiştirme işlemi tek yerden yapılabilir. Böylece söz konusu program sembolik sabite bağlı olarak yazılıp, daha sonra sembolik sabitin değiştirilmesiyle farklı parametrik değerler için çalıştırılabilir.

### 3. Sayısal sabitlerin kullanılmasında tutarsızlıkları ve yazım yanlışlarını engeller.

Örneğin matematiksel hesaplamalar yapan bir kodda sıklıkla pi sayısını kullandığımızı düşünelim. pi sayısı yerine

```
#define PI 3.14159
```

sembolik sabitini kullanabiliriz. Her defasında pi sayısının sabit olarak koda girersek, her defasında aynı değeri yazamayabiliriz. Örneğin bazen 3.14159 bazen 3.14156 girebileceğimiz gibi yanlışlıkla 3.15159 da girebiliriz. Sembolik sabit kullanımı bu tür hataları ortadan kaldırır.

### 4. Sembolik sabitler kullanılarak C sintaksında küçük değişiklikler yapılabilir.

```
#define FOREVER for(;;)
#define BEGIN {
#define END }
```

Böyle bir sembolik sabit kullanımını kesinlikle tavsiye etmiyoruz. Okunabilirliği artırması değil azaltması söz konusudur. Bir C programcısı için en okunabilir kod C'nin kendi sentaksının kullanıldığı koddur.

#define komutu kaynak kodun herhangi bir yerinde kullanılabilir. Ancak tanımlandığı yerden kaynak kodun sonuna kadar olan bölge içerisinde etki gösterir.

Ancak en iyi tanımlama yeri kaynak kodun tepesidir. Geleneksel olarak sembolik sabitler #include satırlarından (bir satır boşluk verildikten sonra) hemen sonra yer alırlar.

## Sembolik Sabitlerin Kullanılmasında Çok Yapılan Hatalar

### 1. Sembolik sabit tanımlamasında gereksiz yere = karakterini kullanmak.

```
#define N = 100 /* yanlış */
```

Bu durumda önişlemci N gördüğü yere = 100 yapıştıracaktır. Örneğin `int a[N]` gibi bir dizi tanımlanmışsa önişlemci bu tanımlamayı `a[N = 100]` yapar ki bu da derleme aşamasında hata (error) ile neticelenir.

### 2. Sembolik sabit tanımlama satırını ; ile sonlandırmak.

```
#define N 100; /* yanlış */
```

Bu durumda önişlemci N gördüğü yere `100;` yerleştirir. `int a[N];` tanımlaması `int a[100;]` haline gelir. Derleme aşamasında hata oluşur. (Borland derleyicileri hata mesajı :Array bounds missing ])Bu tür hatalarda derleyici sembolik sabit kaç yerde kullanılmışsa o kadar error verecektir.

### 3. Sembolik sabitlerle yer değiştirecek ifadelerin operatör içermesi durumunda bu ifadeleri paranteze almadan yerleştirmek.

```
#define SIZE 5 + 10
...
i = SIZE * 5 /* 5 + 10 * 5 = 55 (75 değil) */
...
```

## 16 . BÖLÜM : 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 yapılmaktadır.

**goto** deyiminin genel sentaksı aşağıdaki şekildedir.

```
<goto label;>
...
<label:>
    <statement;>
```

**goto** C dilinin 32 anahtar sözcüğünden biridir. (anahtar sözcüklerin küçük harfle yazıldıklarını tekrar hatırlatalım.) Label (etiket) programcının verdiği bir isimdir. Şüphesiz C dilinin isimlendirme kurallarına uygun olarak seçilmelidir. Programın akışı bu etiketin yerleştirilmiş olduğu yere yönlendirilecektir. Etiket, **goto** anahtar sözcüğünün kullanıldığı fonksiyon içinde, en az bir deyimden önce olacak şekilde, herhangi bir yere yerleştirilebilir. Etiket isminden sonra gelen : atomu sentaksı tamamlar. Etiketin **goto** anahtar sözcüğünden daha sonraki bir kaynak kod noktasına yerleştirilmesi zorunluluğu yoktur, **goto** anahtar sözcüğünden önce de tanımlanmış olabilir. Örneklerle açıklayalım:

```
int main()
{
    ...
    goto GIT;
    ...
    ...
GIT:
    printf("goto deyimi ile buraya gelindi);
    return 0;
}
```

```
int main()
{
    GIT:
    printf("goto deyimi ile gelinecek nokta);
    ...
    goto GIT;
    ...
    return 0;
}
```

**goto** etiketleri bir fonksiyon içerisinde, bir deyimden önce herhangi bir yere yerleştirilebilir. Yani aynı fonksiyon içinde bulunmak kaydıyla **goto** anahtar sözcüğünün yukarısına ya da aşağısına etiketi yerleştirebiliriz. Bu da aslında değişik bir faaliyet alanı tipidir. C dili standartlarında bu faaliyet alanı türüne "fonksiyon faaliyet alanı" denmiştir.

Yapısal programlama tekniğinde **goto** deyiminin kullanılması tavsiye edilmez. Çünkü **goto** deyiminin kullanılması birtakım dezavantajlar doğurur:

1. **goto** deyimi programların okunabilirliğini bozar. Kodu takip eden kişi **goto** deyimiyle karşılaştığında fonksiyonun içinde etiketi arayacak, ve programı bu noktadan takip etmeye koyulacaktır.
2. **goto** deyimlerinin kullanıldığı bir programda bir değişiklik yapılması ya da programın, yapılacak eklemelerle, geliştirilmeye çalışılması daha zor olacaktır. Programın herhangi bir yerinde bir değişiklik yapılması durumunda, eğer program içerisinde başka yerlerden kodun

akışı değişikliğin yapıldığı yere **goto** deyimleri ile atlatılmış ise, bu noktalarda da bir değişiklik yapılması gerekebilecektir.

Bu olumsuzluklara karşın, bazı durumlarda **goto** deyiminin kullanılması programın okunabilirliğini bozmak bir yana, diğer alternatiflere göre daha okunabilir bir yapının oluşmasına yardımcı olacaktır:

İç içe birden fazla döngü varsa, ve içteki döngülerden birindeyken, yalnızca bu döngüden değil, bütün döngülerden birden çıkılmak isteniyorsa **goto** deyimini kullanılmalıdır.

Aşağıdaki kod parçasında iç içe 3 döngü bulunmaktadır. En içteki döngünün içinde func fonksiyonu çağırılarak fonksiyonun geri dönüş değeri test edilmekte, fonksiyon eğer 0 değerine geri dönüyorsa programın akışı **goto** deyimiyile tüm döngülerin dışına yönlendirilmektedir:

```
int i, j, k;

...
for (i = 0; i < 100; ++i) {
    ...
    for (j = 0; j < 100; ++j) {
        ...
        for (k = 0 {
            ...
            if (!func())
                goto BREAK;
            ...
        }
    }
}
BREAK:
printf("döngü dışındaki ilk deyim\n");
...
```

Oysa **goto** deyimini kullanmasaydık, ancak bir bayrak (flag) kullanarak, ve her döngünün çıkışında bayrak olarak kullanılan değişkenin değerinin değiştirilip değiştirilmediğini kontrol ederek bunu başarabilirdik:

```
int i, j, k;

int flag = 0;

...
for (i = 0; i < 100; ++i) {
    ...
    for (j = 0; j < 100; ++j) {
        ...
        for (k = 0; k < 100; ++k) {
            ...
            if (!func()) {
                flag = 1;
                break;
            }
            ...
        }
        if (flag)
            break;
    }
    if (flag)
        break;
}
```



```

}
printf("döngü dışındaki ilk deyim\n");
...

```

Yine aşağıdaki kod parçasında **goto** deyimiyle hem **switch** deyimğinden hem de **switch** deyiminin içinde bulunduğu **for** döngüsünden çıkılmaktadır.

```

...
int main()
{
    int option;

    for (;;) {
        option = get_option();
        switch (option) {
            case ADDREC      :addrec();break;
            case LISTREC    :listrec();break;
            case DELREC      :delrec(); break;
            case SORTREC    :sortrec(); break;
            case EXITPROG   :goto EXIT;
        }
    }
EXIT:
    return SUCCESS;
}

```

Yukarıdaki kod parçasında option değişkeninin değeri EXITPROG olduğunda programın akışı goto deyimiyle sonsuz döngünün dışına gönderilmiştir. goto deyimi yerine break anahtar sözcüğü kullanılsaydı, yalnızca switch deyiminden çıkılmış olunacaktı.

## 17 . BÖLÜM : RASTGELE SAYI ÜRETİMİ

Rasgele sayı üretimi matematiğin önemli konularından biridir. Rasgele sayılar ya da daha doğru ifadeyle, rasgele izlenimi veren sayılar (sözde rasgele sayılar - pseudo random numbers) istatistik, ekonomi, matematik gibi pekçok alanda olduğu gibi programcılıkta da kullanılmaktadır.

Rasgele sayılar bir rasgele sayı üreticisi (random number generator) tarafından üretilirler. Rasgele sayı üreticisi aslında matematiksel bir fonksiyondur. Söz konusu fonksiyon bir başlangıç değeri alarak bir değer üretir. Daha sonra ürettiği her değeri girdi olarak alır ve tekrar başka bir sayı üretir . Üreticinin ürettiği sayılar rasgele izlenimi vermektedir.

C standartları rasgele tamsayı üreten bir fonksiyonun standart bir C fonksiyonu olarak tanımlanmasını zorunlu kılmıştır. Bu fonksiyonun prototip bildirimi aşağıdaki gibidir:

```
int rand(void);
```

C standartları rand fonksiyonunun rasgele sayı üretimi konusunda kullanacağı algoritma ya da teknik üzerinde bir koşul koymamıştır. Bu konu derleyiciyi yazarların seçimine bağlı (implementation dependent) bırakılmıştır. rand fonksiyonunun prototipi standart bir başlık dosyası olan stdlib.h içinde bildirilmiştir.

Bu yüzden rand fonksiyonunun çağırılması durumunda bu başlık dosyası "include" önışlemci komutuyla kaynak koda dahil edilmelidir.

```
#include <stdlib.h>
```

rand fonksiyonu her çağırıldığında [0, RAND\_MAX] aralığında rasgele bir tamsayı değerini geri döndürür. RAND\_MAX stdlib.h başlık dosyası içinde tanımlanan bir sembolik sabittir, derleyicilerin çoğunda RAND\_MAX sembolik sabiti 32767 olarak, yani 2 byte'lık **signed int** türünün maximum değeri olarak tanımlanmıştır.

Aşağıdaki program parçasında 0 ile 32767 arasında 10 adet rasgele sayı üretilerek ekrana yazdırılmaktadır :

```
#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 yazılacaktır. Örneğin yukarıdaki programı DOS altında Borland Turbo C 2.0 derleyicisi ile derleyip çalıştırdığımızda ekran çıktısı aşağıdaki gibi oldu:

```
346 130 10982 1090 11656 7117 17595 6415 22948 31126
```

Oluşturulan program her çalıştırıldığında neden hep aynı sayı zinciri elde ediliyor? rand fonksiyonu rasgele sayı üretmek için bir algoritma kullanıyor. Bu algoritma derleyiciden derleyiciye değişse de, rasgele sayı üretiminde kullanılan ana tema aynıdır. Bir başlangıç değeri ile işe başlanır. Buna tohum değer diyoruz. Bu değer üzerinde bazı işlemler yapılarak rasgele

bir sayı elde edilir. Tohum değer üzerinde yapılan işlem bu kez elde edilen rasgele sayı üzerinde tekrarlanır...

rand fonksiyonu rasgele sayı üretmek için bir başlangıç değeri kullanıyor. Rasgele sayı üreticilerinin (random number generator) kullandıkları başlangıç değerine tohum değeri (seed) denir. rand fonksiyonunu içeren programı her çalıştırdığımızda aynı tohum değerinden başlayacağı için aynı sayı zinciri elde edilecektir. İşte ikinci fonksiyon olan srand fonksiyonu, rasgele sayı üreticisinin tohum değerini değiştirmeye yarar:

**void** srand (**unsigned seed**);

srand fonksiyonuna gönderilen arguman rasgele sayı üreticisinin tohum değerini değiştir. srand() fonksiyonuna başka bir tohum değeri gönderdiğimizde fonksiyonun ürettiği rasgele sayı zinciri değişecektir. Yukarıdaki programa bir ilave yaparak yeniden çalıştırın.

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

```
int main()
{
    int k;

    srand(100);
    for (k = 0; k < 10; ++k)
        printf("%d ", rand());
    return 0;
}
```

1862 11548 3973 4846 9095 16503 6335 13684 21357 21505

Ancak bu kez oluşturduğumuz programı her çalıştırdığımızda yine yukarıdaki sayı zinciri elde edilecek. Zira rand fonksiyonunun kullanmakta olduğu önceden seçilmiş (default) tohum değerini kullanmasak da, rasgele sayı üretme mekanizması bu kez her defasında bizim srand fonksiyonuyla göndermiş olduğumuz tohum değerini kullanacak. Programı birkaç kere çalıştırıp aynı sayı zincirini ürettiğini test edin.

Bir çok durumda programın her çalıştırılmasında aynı rasgele sayı zincirinin üretilmesi istenmez. Programın her çalışmasında farklı bir sayı zincirinin elde edilmesi için programın her çalışmasında srand fonksiyonu başka bir değerle, rasgele sayı üreticisinin tohum değerini set etmelidir. Bu amaçla çoğu zaman time fonksiyonundan faydalanılır:

time fonksiyonu standart bir C fonksiyonudur, prototip bildirimi standart bir başlık dosyası olan time.h içindedir. time fonksiyonu, parametre değişkeni gösterici olan bir fonksiyon olduğundan ileride detaylı olarak ele alınacaktır. Şimdilik time fonksiyonunu işimizi görecektir kadar inceleyeceğiz. time fonksiyonu 0 argumanıyla çağırıldığında 01.01.1970 tarihinden fonksiyonun çağırıldığı ana kadar geçen saniye sayısını geri döndürür. Fonksiyonun geri dönüş değeri derleyicilerin çoğunda **long** türden bir değerdir. İçinde rasgele sayı üretilecek programda, srand fonksiyonuna arguman olarak time fonksiyonunun geri dönüş değeri arguman olarak gönderilirse, program her çalıştığında rasgele sayı üreticisi başka bir tohum değeriyle ilk değerini alacaktır, böyle programın her çalıştırılmasında farklı sayı zinciri üretilecektir.

```
srand(time(0));
```

srand fonksiyonunun bu şekilde çağırımı derleyicilerin çoğunda randomize isimli bir makro olarak tanımlanmıştır. Yukarıdaki fonksiyon çağırımı yerinde bu makro da çağırılabilir. Makrolar konusunu ileride detaylı olarak inceleyeceğiz :

```
randomize();
```

Yukarıdaki örnek programı her çalıştığında farklı sayı zinciri üretecek hale 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 rasgele sayı üretmek isteyebiliriz. Bu durumda mod operatörü kullanılabilir :

```
rand() % 2      Yalnızca 0 ya da 1 değerini üretir.
rand() % 6      0 - 5 aralığında rasgele bir değer üretir
rand() % 6 + 1  1 - 6 aralığında rasgele bir değer üretir.
rand() % 6 + 3  3 - 8 aralığında rasgele bir değer üretir.
```

Karmaşık olasılık problemleri, olasılığa konu olayın bir bilgisayar programı ile simule edilemesi yoluyla çözülebilir. İyi bir rasgele sayı üreticisi kullanıldığı takdirde, olasılığa konu olay bir bilgisayar programı ile simule edilir ve olay bilgisayarın işlem yapma hızından faydanlanılarak yüksek sayılarda tekrar ettirilir. Şüphesiz hesap edilmek istenen olaya ilişkin olasılık değeri, yapılan tekrar sayısına ve rasgele sayı üreticisinin kalitesine bağlı olacaktır. Aşağıdaki kod yazı tura atılması olayında tura gelme olasılığını hesap etmektedir.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

#define TIMES      100
#define HEADS      1

int main()
{
    long heads_counter = 0;
    long k;

    for (k = 0; k < TIMES; ++k)
        if (rand() % 2 == HEADS)
            heads_counter++;
    printf("tura gelme olasılığı = %lf", (double) heads_counter / TIMES);
    getch();

    return 0;
}
```

Yukarıdaki programı TIMES sembolik sabitinin farklı değerleri için çalıştırdığımızda 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
#define TIMES 10000000
tura gelme olasılığı = 0.500015
#define TIMES 20000000
tura gelme olasılığı = 0.500198
```

Aşağıdaki main fonksiyonunda uzunlukları 3 - 8 harf arasında değişen ingiliz alfabesindeki harfler ile oluşturulmuş rasgele 10 kelime ekrana yazdırılmaktadır:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

#define TIMES 20

void write_word(void);

int main()
{
    int k;

    srand(time(0));
    for (k = 0; k < TIMES; ++k) {
        write_word();
        putchar('_');
    }
    getch();
    return 0;
}

void write_word(void)
{
    int len = rand() % 6 + 3;

    while (len--)
        putchar('A' + rand() % 26);
}
```

Programın ekran çıktısı:

```
USPFY
KDUP
BUQJB
LMU
PKPBKFA
WQM
NQHPTXL
QCANSTEH
XZU
```

MNRI  
OXUTGISR  
XBNG  
BYOQ  
TFO  
MQUCSU  
OIHFPJ  
BLVD  
WDDEM  
OHMHJBZY  
ALIAQ

## 18 . BÖLÜM : DİZİLER

Bellekte ardışıl bir biçimde bulunan ve aynı türden nesnelerin oluşturduğu veri yapısına dizi (array) denir. Dizilerin kullanılmasını gerektiren iki önemli özellikleri vardır:

1. Dizi elemanları bellekte ardışıl (contiguous) olarak bulunurlar.
2. Dizi elemanları aynı türden nesnelerdir.

Diziler bileşik nesnelerdir. Yani bir dizinin tanımlanmasıyla birden fazla sayıda nesne birlikte tanımlanabilir. (Bileşik sözcüğü İngilizce "aggregate" sözcüğünün karşılığı olarak kullanılmıştır.) 10 elemanlık bir dizi tanımlamak yerine, şüphesiz farklı isimlere sahip 10 ayrı nesne de tanımlanabilir. Ama 10 ayrı nesne tanımlandığında bu nesnelerin bellekte ardışıl olarak yerleşmeleri garanti altına alınmış bir özellik değildir. Oysa dizi tanımlamasında, dizinin elemanı olan bütün nesnelerin bellekte ardışıl olarak yer almaları garanti altına alınmış bir özelliktir.

Dizilerde bir veri türü olduğuna göre dizilerin de kullanılmalarından önce tanımlanmaları gerekir.

### Dizilerin Tanımlanması

Dizi tanımlamalarının genel biçimi:

<tür> <dizi ismi> [<eleman sayısı>];

Yukarıdaki gösterimde köşeli parantez eleman sayısının seçimsel olduğunu değil, eleman sayısı bilgisinin köşeli parantez içine yazılması gerektiğini göstermektedir.

tür : Dizi elemanlarının türünü gösteren anahtar sözcüktür.  
dizi ismi : İsimlendirme kurallarına uygun olarak verilecek herhangi bir isimdir.  
eleman sayısı : Dizin kaç elemana sahip olduğunu gösterir.

### Örnek Dizi Bildirimleri:

```
double a[20]; /* a, 20 elemanlı ve elemanları double türden olan bir dizidir*/
float ave[10]; /* ave 10 elemanlı ve her elemanı float türden olan bir dizidir. */
unsigned long total[100]; /* total 100 elemanlı ve her elemanı unsigned long türden olan
                           bir dizidir */
char path[80]; /* path 80 elemanlı ve her elemanı char türden olan bir dizidir. */
```

Tanımlamada yer alan eleman sayısının mutlaka tamsayı türlerinden birinden sabit ifadesi olması zorunludur. (Sabit ifadesi [constant expression] tanımını hatırlayalım; değişken ve fonksiyon çağırımı içermeyen, yani yalnızca sabitlerden oluşan ifadelere, sabit ifadesi denir.)

```
int dizi[x]; /* x dizisinin bildirimi derleme zamanında hata oluşturur .*/
int dizi[5.]; /* gerçek sayı türünden sabit ifadesi olduğu için derleme zamanında hata
               oluşturur . */

int sample[10 * 20] /* sample dizisinin bildirimi geçerlidir. Eleman sayısını gösteren
                     ifade sabit ifadesidir. */
```

Dizi bildirimlerinde eleman sayısı yerine sıklıkla sembolik sabitler kullanılır:

```
#define MAXSIZE 100
...
int dizi[MAXSIZE]; /* geçerli bir bildirimdir */
...
```

Diğer değişken bildirimlerinde olduğu gibi, virgül ayırıcıyla ayrılarak, birden fazla dizi tek bir tür belirten anahtar sözcükle tanımlanabilir.

```
int x[100], y[50], z[10];
```

x, y ve z elemanları **int** türden olan dizilerdir.

Dizi tanımlamaları diğer değişken tanımlamaları ile kombine edilebilir.

```
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 index operatörüyle [] ulaşılabilir. Index operatörü bir gösterici operatördür. Göstericiler konusunda ayrıntılı bir şekilde ele alınacaktır.

Index operatörünün operandı dizi ismidir. (Aslında bu bir adres bilgisidir, çünkü dizi isimleri adres bilgisi belirtirler.) Köşeli parantez içinde dizinin kaçınıcı indisli elemanına ulaşacağımızı gösteren bir tamsayı ifadesi olmalıdır.

C dilinde dizilerin ilk elemanı sıfırıncı indisli elemandır.

a[n] gibi bir dizinin ilk elemanı a[0] son elemanı ise a[n - 1] dur.

Örnekler:

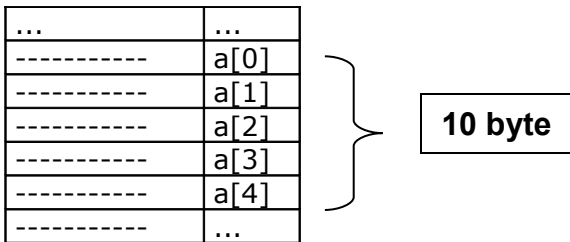
```
dizi[20] /* a dizisinin 20. indisli yani 21. sıradaki elemanı. */
ave[0]   /* ave dizisinin 0. indisli yani birinci sıradaki elemanı */
total[j] /* total dizisinin j indisli elemanı */
```

Görüldüğü gibi bir dizinin n. elemanı ve bir dizinin n indisli elemanı terimleri dizinin farklı elemanlarına işaret eder. Bir dizinin n indisli elemanı o dizinin n + 1 . elemanıdır.

Bir dizi tanımlaması ile karşılaşan derleyici, tanımlanan dizi için bellekte yer tahsis edecektir..Ayrılacak yer şüphesiz dizinin eleman sayısı \* bir elemanın bellekte kapladığı yer kadar byte olacaktır. Örneğin:

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

gibi bir dizi tanımlaması yapıldığını düşünelim. DOS işletim sisteminde çalışıyorsak, derleyici a dizisi için bellekte 2 \* 5 = 10 byte yer ayıracaktır. Bellekte bu dizinin yerleşimi aşağıdaki gibi olacaktır :



Dizi kullanımının getirdiği en büyük avantaj döngü deyimleri kullanarak tüm dizi elemanlarını kolay bir şekilde erişilebilmesidir.

Dizi indis ifadelerinde ++ ya da - operatörlerinin kullanıldığı sık görülür.

```
int a[20];
int k = 10;
int i = 5;
```



```
a[k++] = 100; /* a dizisinin 10 indisli elemanına yani 11. elemanına 100 değeri atanıyor. */
a[--i] = 200; /* a dizisinin 4 indisli elemanına yani 5. elemanına 200 değeri atanıyor. */
```

İndex operatörünün kullanılmasıyla artık dizinin herhangi bir elemanı diğer değişkenler gibi kullanılabilir.

Örnekler:

```
a[0] = 1; /* a dizisinin ilk elemanına 0 değeri atanıyor. */
printf("%d\n", sample[5]); /* sample dizisinin 6. elemanı yazdırılıyor. */
++val[3]; /* val dizisinin 4. elemanı 1 artırılıyor. */
val[2] = sample[2]; /* val dizisinin 3. elemanına sample dizisinin 3. elemanı atanıyor. */
```

Diziler üzerinde işlem yapmak için sıklıkla **for** ve **while** döngüleri kullanılır. Aşağıda SIZE elemanlı ve sample isimli bir dizi için **for** ve **while** döngülerinin kullanıldığı bazı kalıplar gösterilmektedir:

```
for (i = 0; i < SIZE; ++i)
    sample[i] = 0;
```

```
i = 0;
while (i < SIZE) {
    sample[i] = 0;
    ++i;
} /* sample dizisinin bütün elemanları sıfırlanıyor */
```

```
for (i = 0; i < SIZE; i++) {
    scanf("%d", &sample[i]);
    ++i;
}
```

```
i = 0;
while (i < SIZE) {
    scanf("%d", &sample[i]);
    ++i;
} /* sample dizisinin elemanlarına scanf fonksiyonuyla klavyeden değer alınıyor.*/
```

```
for (i = 0; i < SIZE; i++)
    total += sample[i];
```

```
i = 0;
while (i < SIZE) {
    total += sample[i];
    ++i;
} /* sample dizisinin elemanları toplanıyor. */
```

Bir dizi tanımlaması yapıldığı zaman derleyici tanımlaması yapılmış dizi için bellekte toplam dizi uzunluğu kadar yer ayırır. Örneğin :

```
double a[10];
```

gibi bir tanımlama yapıldığında dizi için bellekte ardışıl (contiguous) toplam 80 byte'lık bir yer ayrılacaktır.

Dizinin son elemanı a[9] olacaktır. Çok sık yapılan bir hata, dizinin son elemanı diye bellekte derleyici tarafından tahsis edilmemiş bir yere değer atamaktır.

a[10] = 5.;

deyimiyle bellekte rasgele 8 byte'lık bir alana (güvenli olmayan) bir yere 5. değeri yazılmaktadır.

Dizilere ilk değer verilmesi (array initialization)

Dizilere ilk değer verme işleminin genel şekli aşağıdaki gibidir :

<tür> <dizi ismi>[[uzunluk]] = {d1, d2, d3.....};

Örnekler :

```
double sample[5] = {1.3, 2.5, 3.5, 5.8, 6.0};
char str[4] = {'d', 'i', 'z', 'i'};
unsigned[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Dizilere yukarıdaki gibi ilk değer verildiğinde, verilen değerler dizinin ilk elemanından başlayarak dizi elemanlarına sırayla atanmış olur.

Dizilerin tüm elemanlarına ilk değer verme zorunluluğu yoktur. Dizinin eleman sayısından daha az sayıda elemana ilk değer verilmesi durumunda kalan elemanlara otomatik olarak 0 değeri atanmış olur. Bu kural hem yerel hem de global diziler için geçerlidir.

Bu durumda bütün dizi elemanlarına 0 değeri atanmak isteniyorsa bunun en kısa yolu :

```
int a[20] = {0};
```

yalnızca dizinin ilk elemanına 0 değeri vermektir. Bu durumda derleyici dizinin kalan elemanlarına otomatik olarak 0 değeri atayacaktır.

Dizi elemanlarına ilk değer verilmesinde kullanılan ifadeler, sabit ifadeleri olmalıdır.

```
int a[10] = {b, b + 1, b + 2};
```

gibi bir ilk değer verme işlemi derleme zamanı hatası oluşturur.

Bir diziye ilk değer verme işleminde dizi eleman sayısından daha fazla sayıda ilk değer vermek derleme zamanında hata oluşumuna neden olur :

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

yukarıdaki örnekte b dizisi 5 elemanlı olmasına karşın, ilk değer verme ifadesinde 6 değer kullanılmıştır.

dizi elemanlarına ilk değer verme işleminde dizi uzunluğu belirtilmeyebilir, bu durumda derleyici dizi uzunluğunu verilen ilk değerleri sayarak kendi hesaplar ve dizinin o uzunlukta açıldığını kabul eder. Örneğin :

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

derleyici yukarıdaki ifadeyi gördüğünde sample dizisinin 5 elemanlı olduğunu kabul edecektir. Bu durumda yukarıdaki gibi bir bildirimle aşağıdaki gibi bir bildirim eşdeğer olacaktır:

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

başka örnekler :

```
char name[ ] = {'N', 'e', 'c', 'a', 't', 'i', ' ', 'E', 'r', 'g', 'i', 'n', '\0'};
unsigned short count[ ] = {1, 4, 5, 7, 8, 9, 12, 15, 13, 21};
```

derleyici name dizisinin uzunluğunu 13, count dizisinin uzunluğunu ise 10 olarak varsayacaktır.

yerel ve global diziler

Bir dizi de, diğer nesneler gibi yerel ya da global olabilir. Yerel diziler blokların içlerinde tanımlanan dizilerdir. Global diziler ise tüm blokların dışında tanımlanırlar. Global bir dizinin tüm elemanları global nesnelerin özelliklerine sahip olacaktır. Yani dizi global ise dizi elemanı olan nesneler dosya faaliyet alanına (file scope) sahip olacak, ve ömür karakteri açısından da statik ömürlü (static storage duration) olacaklardır. Global bir dizi söz konusu olduğunda eğer dizi elemanlarına değer verilmemişse, dizi elemanları 0 değeriyle başlatılacaktır. Ama yerel diziler söz konusu olduğunda, dizi elemanı olan nesneler blok faaliyet alanına (block scope) sahip olacaklar, ömür açısından ise dinamik ömür karakterinde olacaklardır. (dynamic storage class) Değer atanmamış dizi elemanları içinde rasgele değerler (garbage values) bulunacaktır. Aşağıdaki örneği yazarak derleyicinizde yazarak deneyiniz :

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

```
int global[10];
```

```
main()
```

```
{
```

```
    int yerel[10], i;
```

```
    for (i = 0; i < 10; ++i)
```

```
        printf("global[%d] = %d\n", i, global[i]);
```

```
    for (i = 0; i < 10; ++i)
```

```
        printf("yerel[%d] = %d\n", i, yerel[i]);
```

```
}
```

dizilerin birbirine atanması

```
int a[SIZE], b[SIZE];
```

tanımlamasından sonra, a dizisi elemanlarına b dizisinin elemanları kopyalanmak istenirse

```
a = b;          /* hata */
```

yukarıdaki gibi bir atama derleme zamanı hatası verecektir. Çünkü a ve b dizi isimleridir. C dilinde dizi isimleri nesne göstermezler. Dizi isimleri dizilerin bellekte yerleştirildikleri alanın başlangıcını gösteren ve dizinin türü ile aynı türden adres değerleridir. (Dolayısıyla sabit değerlerdir.)

Dizileri ancak bir döngü kullanarak kopyalayabiliriz :

```
for (i = 0; i < SIZE; ++i)
```

```
    a[i] = b[i];
```

Yukarıdaki döngü deyimiyle b dizisinin her bir elemanının değeri a dizisinin ilgili indisli elemanına atanmıştır. Dizilerin kopyalanması için başka bir yöntem de bir standart C fonksiyonu olan memcpy (memory copy) fonksiyonunu kullanmaktır. Bu fonksiyon göstericiler konusundan sonra ele alınacaktır.

dizilerle ilgili uygulama örnekleri:

Dizilerin kullanılmasını teşvik eden temel nedenlerden biri aynı türden nesnelerin bir dizi altında tanımlanmasıyla, döngüler yardımıyla dizi elemanlarının çok kolay bir şekilde işleme tutulabilmesidir. Aşağıdaki örneklerde bu temalar işlenmiştir :

Uygulama 1 : Bir dizi elemanlarının toplamının bulunması

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

#define SIZE      10

main()
{
    int dizi[SIZE] = {12, 25, -34, 45, 67, 89, 24, -2, -15, 40};
    int toplam = 0;
    int k;

    clrscr();
    for (k = 0; k < SIZE; ++k)
        toplam += dizi[k];
    printf("dizi elemanları toplamı = %d\n", toplam);

    return 0;
}
```

Uygulama 2 : bir dizi içindeki en küçük sayıyı bulan program :

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

#define SIZE      10

main()
{
    int dizi[SIZE] = {12, 25, -34, 45, 67, 89, 24, -2, -15, 40};
    int min, k;

    clrscr();
    min = dizi[0];
    for (k = 1; k < SIZE; ++k)
        if (min > dizi[k])
            min = dizi[k];
    printf("en küçük eleman = %d\n", min);

    return 0;
}
```

eğer en küçük elemanın kendisiyle birlikte, dizinin kaçınıcı elemanı olduğu da bulunmak istenseydi :

```
int indis;
...
for (k = 1; k < SIZE; ++k)
    if (min > dizi[k]) {
min = dizi[k];
indis = k; /* en küçük eleman yenilendikçe indisi başka bir değişkende saklanıyor. */
    }
```

Uygulama 3: Bir diziye klavyeden değer alarak daha sonra dizi elemanlarını ekrana yazdıran program :

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

#define SIZE      10

main()
{
    int dizi[SIZE];
    int k;

    clrscr();
    for (k = 0; k < SIZE; ++k)
        scanf("%d", &dizi[k]);
    for (k = 0; k < SIZE; ++k)
        printf("dizi[%d] = %d\n", k, dizi[k]);
    return 0;
}
```

Uygulama 4 : Bir dizi içerisindeki elemanları küçükten büyüğe doğru bubble sort algoritmasıyla sıraya dizen program:

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

#define SIZE      10

main()
{
    int dizi[SIZE] = {12, 25, -34, 45, -23, 29, 12, 90, 1, 20};
    int i, k, temp;

    clrscr();
    for (i = 0; i < SIZE - 1; ++i)
        for (k = 0; k < SIZE - 1; ++k)
            if (dizi[k] > dizi[k + 1]) {
                temp = dizi[k];
                dizi[k] = dizi[k + 1];
                dizi[k + 1] = temp;
            }
    for (i = 0; i < SIZE; ++i)
        printf("dizi[%d] = %d\n", i, dizi[i]);
    return 0;
}
```

Bu algoritmanın performansı basit bir düzenleme ile artırılabilir. Dıştaki döngünün her dönüşünde en büyük sayı en sona gidecektir. Bu durumda içteki döngünün her defasında  $SIZE - 1$  kez dönmesine gerek yoktur. İçteki döngü  $SIZE - 1 - i$  kez dönebilir.

```
...
for (i = 0; i < SIZE - 1; ++i)
for (k = 0; k < SIZE - 1 - i; ++k)
...
```

Bubble sort algoritmasını sıraya dizme işlemi tamamlandığında dıştaki döngüyü sonlandıracak biçimde düzenleyebiliriz :

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

```
#include <conio.h>

#define SIZE          10
#define UNSORTED      0
#define SORTED        1

main()
{
    int dizi[SIZE] = {12, 25, -34, 45, -23, 29, 12, 90, 1, 20};
    int i, k, temp, flag;

    for (i = 0; i < SIZE - 1; ++i) {
        flag = SORTED;
        for (k = 0; k < SIZE - 1; ++k)
            if (dizi[k] > dizi[k + 1]) {
                temp = dizi[k];
                dizi[k] = dizi[k + 1];
                dizi[k + 1] = temp;
                flag = UNSORTED;
            }
        if (flag == SORTED)
            break;
    }
    for (i = 0; i < SIZE; ++i)
        printf("dizi[%d] = %d\n", i, dizi[i]);
    return 0;
}
```

bu yeni düzenleme diğerine göre daha iyi de olsa performans açısından çok ciddi bir farklılığı yoktur.

Aynı algoritmayı bir do while döngüsü kullanarak da yazabilirdik:

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

#define SIZE          10
#define UNSORTED      0
#define SORTED        1

main()
{
    int dizi[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 (dizi[k] > dizi[k + 1]) {
                temp = dizi[k];
                dizi[k] = dizi[k + 1];
                dizi[k + 1] = temp;
                flag = UNSORTED;
            }
    } while (flag == UNSORTED);

    for (i = 0; i < SIZE; ++i)
        printf("dizi[%d] = %d\n", i, dizi[i]);
    return 0;
}
```

```
}
```

Uygulama 5: Bir dizi içerisindeki elemanları küçükten büyüğe insertion sort algoritmasıyla sıraya dizen program:

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

#define SIZE    10

main()
{
    int dizi[SIZE] = {12, 25, -34, 45, -23, 29, 12, 90, 1, 20};
    int i, j, temp;

    clrscr();
    for (i = 1; i < SIZE; ++i) {
        temp = dizi[i];
        for (j = i; j > 0 && dizi[j - 1] > temp; --j)
            dizi[j] = dizi[j - 1];
        dizi[j] = temp;
    }
    for (i = 0; i < SIZE; ++i)
        printf("dizi[%d] = %d\n", i, dizi[i]);
    return 0;
}
```

sıralama yönünü küçükten büyüğe yapmak yerine büyükten küçüğe yapmak için içerideki döngüyü

```
for (j = i; j > 0 && dizi[j - 1] < temp; --j)
```

şeklinde değiştirmek yeterli olacaktır.

Uygulama 6 : Bir dizi içerisindeki elemanları küçükten büyüğe selection sort algoritmasıyla sıraya dizen program:

```
#include <stdio.h>

#define    SIZE    10

int a[SIZE] = { 12, 23, 45, 78, -23, ,56, 78, 3, 9, -4};

int main()
{
    int k, l, max, indis;

    for (k = 0; k < SIZE; ++k) {
        max = a[k];
        indis = k;
        for (l = k + 1; l < SIZE; ++l)
            if (a[l] > max) {
                max = a[l];
                indis = l;
            }
        a[indis] = a[k];
        a[k] = max;
    }
}
```

```

    }
    for (k = 0; k < SIZE; ++k)
        printf("%d\n", a[k]);
    return 0;
}

```

#### karakter dizileri

Karakter dizileri char türden dizilerdir. Karakter dizilerinin bazı ilave özellikler dışında diğer dizi türlerinden bir farkı yoktur. char türden diziler daha çok içlerinde yazı tutmak için tanımlanırlar.

```
char s[100];
```

yukarıdaki tanımlamada s dizisi bütün elemanları char türden olan 100 elemanlı bir dizidir. char türden bir dizi içinde bir yazı tutmak demek, dizinin herbir elemanına sırayla yazının bir karakterini atamak anlamına gelecektir. (Bu arada char türden bir dizinin içinde bir yazı tutmanın zorunlu olmadığını, böyle bir dizinin pekala küçük tamsayıları tutmak amacıyla da kullanılabileceğini hatırlatalım)

Yukarıda tanımlanan dizi içinde "Ali" yazısını tutmak isteyelim :

```

s[0] = 'A';
s[1] = 'l';
s[2] = 'i';

```

Şimdi şöyle bir problem ortaya çıkıyor. Dizi 100 karakterlik olmasına karşın biz dizi içinde 100 karakter uzunluğundan daha kısa olan yazılar da tutabiliriz. Peki biz yazıya tekrar ulaşmak istediğimizde bunu nasıl yapacağız? Yazının uzunluk bilgisini bilmiyoruz ki! Örneğin yazıyı ekrana yazdırmak istediğimizde, int türden dizilerde kullandığımız şekilde aşağıdaki gibi bir döngü kullanırsak :

```

for (k = 0; k < 100; ++k)
    putchar(s[k]);

```

bu döngü ile yalnızca ALi yazısı ekrana yazdırılmayacak dizinin diğer 97 elemanının da görüntüleri (rasgele değerler) ekrana yazdırılacak.

Programlama dilleri bu duruma bir çözüm bulma konusunda iki temel yaklaşım benimserler.

a. Dizinin 1. elemanına yazının uzunluk bilgisi yazılır. Böylece dizideki yazıya ulaşmak istendiğinde önce dizinin 1. elemanından yazının uzunluk bilgisi çekilir. Böylece döngü deyiminin iterasyon sayısı değeri elde edilmiş olur. Döngü bu değere göre oluşturulur.

```

s[0] = 3;
s[0] = 'A';
s[1] = 'l';
s[2] = 'i';

```

....

```

for (k = 1; k <= s[0]; ++k)
    putchar(s[k]);

```

C dilinde bu yaklaşım kullanılmaz.

b. Dizinin elemanlarına yazının karakterleri sırasıyla yerleştirilir. Yazı bittikten sonra dizinin sıradaki elemanına özel bir karakter yerleştirilir. Bu özel karakter doğal olarak normal şartlar altında bir yazının elemanı olamayacak bir karakter olarak seçilmelidir. Böylece char türden dizi içinde saklanmış yazıya tekrar ulaşmak istendiğinde, kullanılacak döngü deyiminde, dizinin



döngü değişkeni olan indisli elemanın seçilen özel karaktere eşit olmadığı sürece döngünün dönmesi şartı oluşturulur. C dilinin tasarımında bu yaklaşım tercih edilmiştir.

C dilinde karakterler üzerinde işlemlerin hızlı ve etkin bir biçimde yapılabilmesi için "sonlandırıcı karakter" (NULL KARAKTER) kavramından faydalanılmaktadır.

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. Bu karakter DOS ve UNIX sistemlerinde sonlandırıcı karakter olarak kullanılmaktadır. stdio.h içerisinde NULL sembolik sabiti 0 olarak tanımlandığı için sonlandırıcı karaktere NULL karakter de denir. NULL karakter '\0' karakteri ile karıştırılmamalıdır.

'0' karakterinin ASCII sıra numarası 48'dir. Dolayısıyla tamsayı değeri olarak 48 değerine sahiptir.

'\0' karakterinin ASCII sıra numarası 0'dır. Dolayısıyla tamsayı değeri olarak 0 değerine sahiptir.

```
...
printf("%d\n", '0');
printf("%d\n", '\0');
...
```

yukarıdaki ilk printf fonksiyonunun çağırılmasıyla ekrana 48 değeri yazdırılırken, ikinci printf fonksiyonunun çağırılmasıyla ekrana 0 değeri yazdırılmaktadır.

char türden dizilere ilk değer verilmesi

char türden dizilere ilk değer verme işlemi (initializing) aşağıdaki biçimlerde yapılabilir .

Diğer türden dizilerde olduğu gibi virgüllerle ayrılan ilk değerler küme parantezi içinde yer alır :

```
char name[12] = {'N', 'e', 'c', 'a', 't', 'i', ' ', 'E', 'r', 'g', 'i', 'n'};
```

Diğer dizilerde olduğu gibi dizi eleman sayısından daha fazla sayıda elemana ilk değer vermek derleme zamanında hata oluşumuna neden olur.

```
char name[5] = {'N', 'e', 'c', 'a', 't', 'i', ' ', 'E', 'r', 'g', 'i', 'n'};          /* error */
```

Dizi eleman sayısı kadar elemana ya da dizi eleman sayısından daha az sayıda elemana ilk değer verilebilir. Daha az sayıda elemana ilk değer verilmesi durumunda ilk değer verilmemiş elemanlar diğer dizilerde olduğu gibi 0 değeriyle başlatılacaklardır. 0 değerinin NULL karakter olduğunu hatırlayalım :

```
char name [10] = {'A', 'I', 'i'};
```

...	...
'A'	name[0]
'I'	name[1]
'i'	name[2]
'\0'	name[3]
'\0'	name[4]
'\0'	name[5]
'\0'	name[6]
'\0'	name[7]
'\0'	name[8]
'\0'	name[9]
...	...

Dizi elemanlarına ilk değer verilirken dizi boyutu belirtilmeyebilir. Bu durumda derleyici dizi boyutunu verilen ilk değerleri sayarak saptar ve diziyi bu boyutta açılmış varsayar.

```
char name[ ] = {'A', 'I', 'i'};
```

yukarıdaki tanımlama deyimiyle derleyici name dizisinin 3 elemanlı olarak açıldığını varsayacaktır.

Boyut bilgisi vermeden, char türden dizilere tek tek ilk değer verilmesi durumunda, ve boyut bilgisi verilerek dizinin toplam eleman sayısı kadar elemana tek tek ilk değer verilmesi durumunda, derleyici NULL karakteri dizinin sonuna otomatik olarak yerleştirmeyecektir. Bu durumda eğer sonlandırıcı karakter özelliğinden faydalanılmak isteniyorsa, NULL karakter de verilen diğer ilk değerler gibi programcı tarafından verilmelidir. Örnek :

```
char name[ ] = {'A', 'I', 'i', '\0'};
char isim[7] = {'N', 'e', 'c', 'a', 't', 'i', '\0'};
```

Bu şekilde ilk değer vermek zahmetli olduğundan, ilk değer vermede ikinci bir biçim oluşturulmuştur. Karakter dizilerine ilk değerler çift tırnak içinde de verilebilir :

```
char name[ ] = "Ali";
```

bu biçimin diğerinden farkı derleyicinin sonuna otomatik olarak NULL karakteri yerleştirmesidir.

...	...
'A'	name[0]
'I'	name[1]
'i'	name[2]
'\0'	name[3]

yukarıdaki örnekte derleyici diziye 4 elemanlı olarak açılmış varsayacaktır. Dizi eleman sayısından daha fazla sayıda elemana ilk değer vermeye çalışmak, bu biçimin kullanılması durumunda da derleme zamanında hata oluşumuna neden olacaktır.

```
char city[5] = "İstanbul";          /* error */
```

Bu durumun bir istisnası vardır. Eger tam olarak dizi eleman sayısı kadar dizi elemanına çift tırnak içinde ilk değer verilirse bu durum ERROR oluşturmaz. Derleyici bu durumda NULL karakteri dizinin sonuna yerleştirmez. Bu durum C dili standartlarına yöneltilen eleştirilerden biridir. (C++ dilinde son kabul edilen standartlara göre bu durum da error oluşturmaktadır.)

```
char name[3] = "Ali";               /* C'de hata değil, C++'da hata */
```

...	...
'A'	name[0]
'I'	name[1]
'i'	name[2]
...	buraya bu durumda null karakter 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 gerekmektedir. Ama char türden diziler söz konusu olduğunda artık dizi uzunluğunu bilmemiz gerekmez, çünkü yazının sonunda '\0' karakter bulunacağından, kontrol ifadelerinde bu durum test edilerek yazının sonuna gelinip gelinmediği anlaşılır. Ancak '\0' karakterin, karakter dizilerinde yazıların son elemanı olarak kullanılmasının dezavantajı da

diziye fazladan bir karakter yani '\0' karakteri eklemek zorunluluğudur. Bu nedenle SIZE elemanlı bir diziye en fazla SIZE – 1 tane karakter girilmelidir.

klavyeden karakter dizisi alan ve ekrana karakter dizisi yazan standart C fonksiyonları

Daha önce gördüğümüz getchar, getch, ve getche fonksiyonları klavyeden tek bir karakter alıyorlardı. Yine putchar fonksiyonu ise tek bir karakteri ekrana yazıyordu. C dilinde birden fazla karakteri (bir stringi) klavyeden alan ya da birden fazla karakteri ekrana yazan standart fonksiyonlar bulunmaktadır.

gets fonksiyonu

gets fonksiyonu klavyeden karakter dizisi almakta kullanılan standart bir C fonksiyonudur. Kullanıcı karakterleri girdikten sonra enter tuşuna basmalıdır. Klavyeden girilecek karakterlerin yerleştirileceği dizinin ismini parametre olarak alır. daha önce de belirtildiği gibi dizi isimleri aslında bir adres bilgisi belirtmektedir. gets fonksiyonunun da parametresi aslında char türden bir adrestir. Ancak göstericilerle ilgili temel kavramları henüz öğrenmediğimiz için şimdilik gets fonksiyonunun char türden bir dizi içine klavyeden girilen karakterleri yerleştirdiğini kabul edeceğiz. Örneğin :

```
char name[20];
```

```
gets(name);
```

ile klavyeden enter tuşuna basılana kadar girilmiş olan tüm karakterler name dizisi içine sırayla yerleştirilirler. Klavyeden Necati yazısının girildiğini kabul edelim.

...	...
'N'	name[0]
	]
'e'	name[1]
	]
'c'	name[2]
	]
'a'	name[3]
	]
't'	name[4]
	]
'i'	name[5]
	]
'\0'	name[6]
	]
...	...

gets fonksiyonu klavyeden girilen karakterleri diziye yerleştirdikten sonra dizinin sonuna NULL karakter (sonlandırıcı karakter) diye isimlendirilen 0 numaralı ASCII karakterini (ya da kullanılan karakter setinin 0 numaralı karakterini) koyar.

gets fonksiyonu dizi için hiç bir şekilde sınır kontrolü yapmaz. gets fonksiyonu ile dizi eleman sayısından fazla karakter girilirse, d, z, taşacağı için beklenmeyen sonuçlarla karşılaşılabilir. Bu tür durumlar göstericiler konusunda (gösterici hataları) detaylı olarak incelenecektir.

gets fonksiyonu '\0' karakterini dizinin sonuna eklediği için SIZE uzunluğunda bir dizi için gets fonksiyonuyla alınacak karakter sayısı en fazla SIZE – 1 olmalıdır. Çünkü sonlandırıcı karakterde diğer karakterler gibi bellekte bir yer kaplamaktadır. Örnek :

```
char isim[12];
```

```
gets(isim);
```

ile klavyeden Necati Ergin isminin girildiğini düşünelim.

...	...
'N'	isim[0]
'e'	isim[1]
'c'	isim[2]
'a'	isim[3]
't'	isim[4]
'i'	isim[5]
' '	isim[6]
'E'	isim[7]
'r'	isim[8]
'g'	isim[9]
'i'	isim[10]
]	
'n'	isim[11]
]	
'\0'	taşma
...	

isim dizisinin tanımlanmasıyla derleyici bu dizi için bellekte 12 byte yer ayıracaktır. isim[0] ...isim[11]

gets fonksiyonu bu durumda '\0' karakterini derleyicinin dizi için tahsis etmediği bir bellek hücresine yazacaktır. Bu tür durumlara dizinin taşırılması (off by one) denmektedir.

taşma durumuyla ilgili olarak ortaya çıkacak hatalar derleme zamanına değil çalışma zamanına (run time) ilişkindir.

puts fonksiyonu

puts fonksiyonu bir karakter dizisinin içeriğini ekrana yazdırmak için kullanılır. İçeriği yazdırılacak olan karakter dizisinin ismini parametre olarak alır. puts fonksiyonu karakter dizisini ekrana yazdıktan sonra imleci sonraki satırın başına geçirir.

```
char name[20];
```

```
gets(name);
puts(name);
```

yukarıdaki örnekte gets fonksiyonu ile klavyeden alınan karakter dizisi puts fonksiyonu ile ekrana yazdırılmaktadır.

karakter dizilerini ekrana yazdırmak için printf fonksiyonu da kullanılabilir.

```
printf("%s\n", name);
```

ile

```
puts(name);
```

aynı işi yapmaktadır. Ancak printf fonksiyonu dizi içeriğini ekrana yazdırdıktan sonra imleci alt satıra taşımaz.

```
puts(name);
```

deyimi yerine aşağıdaki kod parçasını da yazabilirdik.

```
for (i = 0; name[i] != '\0'; ++i)
    putchar(s[i]);
putchar('\n');
```

puts fonksiyonu ve %s format karakteriyle kullanıldığında printf fonksiyonu, sonlandırıcı karakter görene kadar bütün karakterleri ekrana yazar. Bu durumda, herhangi bir şekilde NULL karakter ezilirse her iki fonksiyon da ilk sonlandırıcı karakteri görene kadar yazma işlemine devam edecektir.

Örneğin :

```
char city[ ] = "Ankara";
```

```
city[6] = '!';
```

deyimi ile sonlandırıcı karakter ortadan kaldırılırsa (ezilirse) :

```
puts(name);
```

şeklinde puts fonksiyonu çağırıldığında ekrana Ankara! yazıldıktan sonra tesadüfen ilk NULL karakter görülene kadar ekrana yazmaya devam edilir. puts ve printf fonksiyonları karakter dizilerini yazarken yalnızca NULL karakteri dikkate alırlar. karakter dizilerinin uzunluklarıyla ilgilenmezler.

sizeof operatörünün dizilerle kullanılması

sizeof operatörü operand olarak bir dizi ismi aldığıda byte olarak o dizinin toplam uzunluğunu değer olarak üretir.

```
double sample[10];
```

```
sizeof(sample)
```

ifadesi 80 değerini üretecektir.

sizeof operatörü operand olarak dizinin bir elemanını aldığıda ürettiği değer, dizi hangi türden ise o türün kullanılan sistemdeki byte olarak uzunluğu olacaktır. yani yukarıdaki örnekte

```
sizeof(sample[0])
```

 ifadesi, 8 değerini üretecektir.

sizeof operatörünü bir dizinin uzunluk bilgisi gerektiği yerde de kullanabiliriz :

```
sizeof(sample) / sizeof(sample[0])
```

ifadesi dizi uzunluğunu verecektir. Örnek :

```
for (i = 0; i < sizeof(a) / sizeof(a[0]); ++i)
    a[i] = 0;
```

karakter dizileriyle ilgili bazı küçük uygulama örnekleri

Uygulama 7: Karakter dizisi içerisindeki yazının uzunluğunu bulan program:

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

```
main()
```

```
{
```

```
    int k = 0,
    char s[50];
```

```
    gets(s);
```

```
    while (s[k] != '\0')
        ++k;
    printf("uzunluk = %d\n", k);
}
```

yukarıda while döngüsü yerine for döngüsü kullanılabilirdi :

```
for (k = 0; s[k] != '\0'; ++k)
    ;
```

Uygulama 8: karakter dizisini tersten yazdıran program :

```
#include <stdio.h>

#define SIZE 100

main()
{
    char s[SIZE];
    int k;

    printf("bir yazı giriniz :");
    gets(s);
    for (k = 0; s[k + 1] != '\0'; ++k)
        ++k;
    for (; k >= 0; --k)
        putchar(s[k]);
    return 0;
}
```

Uygulama 9: Karakter dizisindeki, büyük harfleri küçük harfe küçük harfleri büyük harfe çeviren bir program :

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

#define SIZE 100

main()
{
    char s[SIZE];
    int k;

    printf ("bir yazı giriniz : ");
    gets(s);
    for (k = 0; s[k] != '\0'; ++k)
        s[k] = isupper(s[k]) ? toupper(s[k]) : tolower(s[k]);
    printf("dönüştürülmüş yazı \n");
    puts(s);
    return 0;
}
```

Uygulama 10 : Karakter dizisini tersyüz eden program :

```
#include <stdio.h>

main()
```

```

{
    char s[50];
    int n, j, temp,

    gets(s);
    for (n = 0; s[n] != '\0'; ++n)
        ;
    for (j = 0; j < n / 2; ++j) {
        temp = s[n - j - 1];
        s[n - j - 1] = s[j];
        s[j] = temp;
    }
    puts(s);
}

```

Uygulama 11 : bir karakter dizisi içindeki harfleri küçükten büyüğe doğru sıralayan bir program

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

```
#define SIRALI 1
#define SIRASIZ 0
```

```

main()
{
    char s[100];
    int flag, k, temp;

    clrscr();
    printf("bir yazı giriniz :");
    gets(s);
    do {
        flag = SIRALI;
        for (k = 0; s[k + 1] != '\0'; ++k)
            if (s[k] > s[k + 1]) {
                temp = s[k];
                s[k] = s[k + 1];
                s[k + 1] = temp;
                flag = SIRASIZ;
            }
    } while (flag == SIRASIZ);
    printf("sıralanmış yazı :\n");
    puts(s);
    getch();
}

```

## 19 . BÖLÜM : GÖSTERİCİLER

### Adres Kavramı

Adres kavramı hem donanıma hem de yazılıma ilişkindir. Donanımsal açıdan adres bellekte yer gösteren bir sayıdan ibarettir. Mikroişlemci bellekte bir bölgeye ancak o bölgenin adres bilgisiyle erişilebilir. Bellekte (RAM'de) her byte (8 bit) diğerlerinden farklı bir adresle temsil edilir. Sıfır sayısından başlayarak her byte'a artan sırada bir karşılık getirerek elde edilen adresleme sistemine doğrusal adresleme sistemi (linear addressing), bu sistem kullanılarak elde edilen adreslere de doğrusal adresler denilmektedir. Donanımsal olarak RAM'deki her bir byte'a okuma ya da yazma yapma amacıyla ulaşılabilir.

Örneğin, 64 K'lık bir belleğin doğrusal olarak adreslenmesi aşağıdaki biçimde yapılabilir :

BELLEK	DOĞRUSAL ADRES
	0 (0000)
	1
	2
	3
	4
.....	.....
.	
	65533 (FFFD)
	65534 (FFFE)
	65535 (FFFF)

Bilgisayara ilişkin pekçok nitelikte olduğu gibi doğrusal adreslemede de 10'luk sistem yerine 16'lık sistemdeki gösterimler tercih edilir. Bundan böyle yalnızca adres dendiğinde doğrusal adres anlaşılmalıdır.

### Nesnelerin Adresleri

Her nesne bellekte yer kapladığına göre belirli bir adrese sahiptir. Nesnelerin adresleri, sistemlerin çoğunda, derleyici ve programı yükleyen işletim sistemi tarafından ortaklaşa olarak belirlenir. Nesnelerin adresleri program yüklenmeden önce kesin olarak bilinemez ve programcı tarafından da önceden tespit edilemez. Programcı, nesnelerin adreslerini ancak programın çalışması sırasında (run time) öğrenebilir. Örneğin:

**char** ch;

biçiminde bir tanımlamayla karşılaşan derleyici bellekte ch değişkeni için 1 byte yer ayıracaktır. Derleyicinin ch değişkeni için bellekte hangi byte'ı ayıracağını önceden bilemeyiz. Bunu ancak programın çalışması sırasında öğrenebiliriz. Yukarıdaki örnekte ch yerel bir değişkendir. ch değişkeni ilgili blok icra edilmeye başlandığında yaratılıp, bloğun icrası bittiğinde de yok olmaktadır. ch değişkeninin 1A03 adresinde olduğu varsayılarak aşağıdaki şekil çizilmiştir:

	1A00
	1A01
	1A02
ch	1A03
	1A04
	1A05

Tanımlanan nesne 1 byte'dan daha uzunsa, o zaman nesnenin adresi nasıl belirlenir?

**int** b;

	1C00
--	------



	1C01
	1C02
x	1C03
	1C04
	1C05
	1C06
	1C07

1 byte'dan uzun olan nesnelerin adresleri, onların ilk byte'larının (düşük anlamlı byte'larının) adresleriyle belirtilir. Yukarıdaki örnekte x değişkeninin adresi 1C03'tür. Zaten x değişkeninin tamsayı türünden olduğu bilindiğine göre diğer parçasının 1C04 adresinde olacağı da açıktır.

Benzer biçimde **long** türünden olan y değişkeninin bellekteki yerleşiminin aşağıdaki gibi olduğu varsayılırsa, adresinin 1F02 olduğunu söyleyebiliriz:

	1F00
	1F01
x	1F02
	1F03
	1F04
	1F05
	1F06
	1F07

Yerel ya da global olsun, ardışıl (contiguous) bir biçimde tanımlanmış nesnelerin bellekte de ardışıl bir biçimde tutulacağının bir garantisi yoktur. Örneğin:

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

Buradaki ch değişkeninin x değişkeninden 2 byte sonra bulunacağının bir garantisi yoktur.

## Ayrı Bir Tür Olarak Adres

Yazılımsal açıdan RAM'de belirli bir byte'dan ne amaçla bahsetmek isteyebiliriz? O byte içinde yer alan nesnenin değerini tespit etmek için! Eğer tüm nesnelerimiz 1 byte uzunluğunda olsalardı, o zaman yazılımsal olarak adres ile fiziksel adres arasında herhangi bir fark bulunmayacaktı. Oysa RAM'deki belirli bir byte'dan söz ettiğimizde, o byte'ı (belki onu izleyen bazı byte'larla birlikte) herhangi türden bir nesne kaplıyor olabilir. Yazılımsal olarak adres bilgisi yalnızca bellekte yer gösteren bir sayıdan ibaret değildir; aynı zamanda o adresin gösterdiği yerdeki bilginin ne biçimde yorumlanacağını belirten bir tür bilgisini de içermektedir.

Örneğin yazılımsal olarak 1250 adresindeki değerden söz etmek isteseydik aşağıdaki soruların cevabı açıkta kalacaktı.

Söz konusu değer yalnızca 1250 adresindeki byte da mı saklanmış yoksa bunu izleyen (1251, 1252 vs.) bytelar da nesnenin değerinin tespit edilmesinde kullanılacak mı? Söz konusu değer hangi formatta yorumlanacak? (Yani işaretli tamsayı aritmetiğine göre mi, işaretli tam sayı aritmetiğine göre mi, gerçek sayı formatına göre mi, vs.)

yazılımsal olarak bir adres bilgisinin iki bileşeni vardır:

1. Bellekte yer gösteren sayısal bir değer
2. Adresin türü

## Adreslerin Türleri

Bir adresin türü demekle o adrese ilişkin bellek bölgesinde bulunan bilginin derleyici tarafından yorumlanış biçimi anlaşılmalıdır. Örneğin:

**char** ch;

gibi bir bildirim sonrasında ch değişkeninin belleğe aşağıdaki biçimde yerleşmiş olduğunu kabul edelim:

	1A00
	1A01
	1A02
ch	1A03
	1A04
	1A05

Yukarıdaki şekle baktığımızda 1A03 sayısının bir adres belirttiğini anlıyoruz. 1A03 adresinde bulunan bilgi **char** türündendir. Çünkü derleyici 1A03 adresinde bulunan bilgiyi **char** türden bir nesne olarak yorumlamaktadır. Adres yalnız başına sayısal bileşeniyle bir anlam taşımaz, gösterdiği türle birlikte belirtilmesi gerekmektedir.

C'de adres bilgisi ayrı bir veri/nesne türüdür. Daha önce gördüğümüz 11 temel veri türüne ilaveten 11 ayrı adres türünün de varlığından söz edebiliriz.

## Adres Sabitleri

Adresler tamsayı görünümünde olsalar da tamsayı sabitleri gibi belirtilmezler. Çünkü tür bilgilerinin de belirtilmesi gerekir. Adres sabitleri, tamsayı türlerindeki sabitler üzerinde bilinçli tür dönüşümü yapılarak elde edilirler. Bir tamsayı sabitini adres türüne çevirmek için tür dönüştürme operatörü kullanılır.

(<tür> \*) <tamsayı sabiti>

Tür dönüştürme operatörünün içindeki \* adres ya da göstericiyi temsil etmektedir. Örneğin

0x1F00

hexadesimal olarak gösterilmiş **int** türden bir sabittir. Ancak:

(**int** \*) 0x1F00

**int** türünden bir adres sabitidir.

0x16C0L

**long** türden bir tamsayı sabittir. Ancak:

(**long** \*) 0x16C0L

**long** türden bir adres sabitidir. Bunun anlamı şudur: Derleyici 16C0 adresiyle belirtilen yerdeki bilgiyi **long** olarak yorumlayacaktır.

Adres türüne bilinçli dönüşüm yalnız sabitlerle yapılmayabilir, örneğin:

(**char** \*) var

var isimli değişken hangi türden olursa olsun, yukarıdaki ifade ile **char** türden bir adres haline dönüştürülmüştür.

Bir adres bilgisini bir nesne içinde tutmak isteyelim. (Şu an için neden böyle bir şey isteyebileceğimiz sorusu akla gelebilir, ama ileride bunun çok doğal ve gerekli bir işlem olduğunu göreceğiz.) Bu işi nasıl gerçekleştireceğiz? İlk olarak aklımıza şu gelebilir: **int** türden bir nesne tanımlayalım ve adresin sayısal bileşenini tanımladığımız **int** türden nesne içinde tutalım. Yani adresin sayısal bileşeni olan tamsayıyı **int** türden bir nesneye atayalım. Bu durumda iki önemli sakınca ortaya çıkacaktır:

Söz konusu **int** nesnenin değerine programın herhangi bir yerinde tekrar ulaştığımızda, bunun bir adres bilgisinin (yani iki bileşenli bir bilginin) sayısal bileşeni mi yoksa **int** türden normal bir değer mi olarak yorumlanacağını nereden bileceğiz? Peki örneğin değişkenin isminden, değerinin bir adres bilgisinin sayısal bileşeni olduğu sonucunu çıkardığımızı düşünelim, bu durumda adresin tür bileşeninin ne olduğunu nereden anlayacağız?

O zaman belki de şu önerilecektir: **int** türden bir adres bilgisini **int** türden bir nesnede, **char** türden bir adres bilgisini **char** türden bir nesnede, **double** türden bir adres bilgisini ise **double** türden bir nesnede saklayalım, vs. Ama örneğin 64K'lık bir adres alanının söz konusu olduğunu düşünelim. toplam 65535 byte söz konusu olabileceğine göre bize 2byte uzunluğunda bir nesne gerekecek. Örneğin 1 byte uzunlukta bir nesne içinde ancak ilk 255 byte'a ilişkin sayısal bileşenleri tutarken, 8 byte uzunluğunda bir nesne içinde de gerekmediği halde çok büyük bir alana ilişkin adres bilgisini tutabiliriz.

**int** sabitleri **int** türden değişkenlere, gerçek sayı sabitleri gerçek sayı türünden değişkenlere doğal olarak atanıyorlar. Yani her sabit türünün doğal olarak atanabileceği türden bir değişken tanımlayabiliyoruz. İşte adres de ayrı bir tür olduğuna göre adres türünün de doğal olarak atanabileceği değişkenler tanımlanabilir. Bu değişkenlere (nesnelere) gösterici (pointer) denir.

```
'a'          /* char türden bir sabittir */
char a;      /* a char türden bir değişkendir */
int b;       /* b int türden bir değişkendir */
2000L       /* long türden bir sabittir */
long l;      /* l long türden bir değişkendir */
(<tür> *) 0x1B00 /* adres sabiti */
pointer     /* içinde adres tutan bir değişken */
```

Gösterici (pointer) içinde adres bilgisi tutan bir değişkendir (nesnedir). Göstericiler de nesne oldukları için bellekte bir yer kaplarlar. Göstericilerin bellekte kapladıkları yerdeki 1'ler ve 0'ların bir tamsayı olarak yorumlanır, ve yorumlanan bu değer bir adres bilgisinin sayısal bileşenini gösterir.

Peki adres bilgisinin iki bileşenli bir bilgi olduğunu söylemiştik. Göstericinin değeri adresin sayısal bileşeni olarak yorumlanacaksa adresin tür bileşeni nasıl elde edilecek? Zira bellekte yalnızca 1 ler ve 0 lar var. Göstericilerin tuttuğu adres bilgilerinin sayısal bileşenleri göstericiler içinde saklanan tamsayının değeridir. Adres bilgisinin tür bileşeni ise göstericinin tanımlanması sırasında bildirilen türdür.

## Göstericilerin Bildirimleri

Göstericiler adres bilgilerini saklamak ve adreslerle ilgili işlemler yapmak için kullanılan nesnelerdir. Göstericilerin içlerinde adres bilgileri bulunur. Bu nedenle gösterici ile adres hemen hemen eş anlamlı olarak düşünülebilir. Ancak gösterici deyince bir nesne, adres deyince bir tür akla gelmelidir.

Gösterici bildirimlerinin genel biçimi şöyledir:

```
<tür> *<gösterici ismi>;
```

<tür> göstericinin (içerisindeki adresin) türüdür. **char**, **int**, **float**... gibi herhangi bir tür olabilir.

Burada \* göstericiyi ya da adresi temsil etmektedir.

Örnek gösterici bildirimleri:

```
float *f;  
char *s;  
int *dizi;  
unsigned long *PDWORD;
```

Gösterici bildirimlerinin diğer türlere ilişkin bildirimlerden \* atomu ile ayrılmaktadır.

```
char s;
```

bildiriminde s **char** türden bir değişken iken

```
char *s;
```

bildiriminde s **char** türden bir göstericidir. İçerisine **char** türden bir adres konulmalıdır. Bu bildirimden derleyici şu bilgileri çıkaracaktır:

s bir nesnedir, yani bellekte bir yer kaplar. s nesnesi için bellekte ayrılan yerdeki 1 ler ve 0 lar **char** türden bir adresin sayısal bileşeni olarak yorumlanırlar. Bu adresin tür bileşeni ise **char** türüdür.

Burada \* bir operatör değildir. Sentaks olarak nesnenin bir gösterici olduğunu anlatmaktadır. Ritchie stilinde, okunabilirlik açısından \* ile gösterici ismi bitişik yazılmalıdır. Gösterici bildirimleri ile normal bildirimler bir arada yapılabilir. Örneğin :

```
int *p, a;
```

Burada p **int** türden bir göstericidir ama a **int** türden bir normal bir değişkendir.

Aynı türden birden fazla göstericinin bildirimi yapılacaksa araya virgül konularak, her gösterici değişkenin bildirimi \* atomu ile yapılmalıdır.

```
char *str, *ptr;
```

Yukarıdaki bildirimde str ve ptr **char** türden göstericilerdir.

```
long *p1, *p2, l, k[20];
```

Yukarıdaki bildirimde p1 ve p2 **long** türden göstericiler, l **long** türden bir değişken ve k ise **long** türden 20 elemanlı bir dizidir.

## Göstericilerin Uzunlukları

Bir gösterici tanımlamasıyla karşılaşan derleyici –diğer tanımlamalarda yaptığı gibi- bellekte o gösterici için bir yer tahsis eder. Derleyicilerin göstericiler için tahsis ettikleri yerlerin uzunluğu donanım bağımlı olup, sistemden sisteme değişebilmektedir. 32 bit sistemlerde (örneğin UNIX ve Windows 95 sistemlerinde) göstericiler 4 byte uzunluğundadır. 8086 mimarisinde ve DOS altında çalışan derleyicilerde ise göstericiler 2 byte ya da 4 byte olabilirler. DOS'ta 2 byte uzunluğunda ki göstericilere yakın göstericiler (near pointer), 4 byte uzunluğundaki göstericilere ise uzak göstericiler (far pointer) denilmektedir. Biz uygulamalarımızda şimdilik göstericilerin 2 byte olduğunu varsayacağız.

Göstericilerin uzunlukları türlerinden bağımsızdır. Örneğin:

```
char *str;  
int *p;  
float *f;
```

DOS altında str, p ve f isimli göstericilerin hepsi de bellekte 2 byte ya da 4 byte yer kaplarlar. Çünkü göstericilerin türü yalnızca içinde tuttıkları adres bilgisinin hangi tür olarak yorumlanacağı ile ilgilidir.

Bir göstericiye aynı türden bir adres bilgisi yerleştirilmelidir. Örneğin :

```
int *p;
```

```
p = 100;
```

Burada p göstericisine adres olmayan bir bilgi atanmaktadır. C dilinin kurallarına göre, derleme zamanında bir hata oluşumuna yol açmaz ama yanlıştır. (Derleyiciler bu durumu bir uyarı mesajı ile bildirirler.) Bu durum ileride detaylı olarak ele alınacaktır.

```
int *p;
```

```
p = (char *) 0x1FC0;
```

Burada **int** türden p göstericisine **char** türden bir adres bilgisi atanmaktadır. C dilinin kurallarına göre, derleme zamanından bir hata oluşumuna yol açmaz ama yanlıştır. (Derleyiciler bu durumu bir uyarı mesajı ile bildirirler.)

```
int *p;
```

```
p = (int *) 0x1FC5; /* geçerli ve uygun bir atamadır */
```

Bir adres bilgisi göstericiye atandığında adresin sayısal bileşeni gösterici içerisine yerleştirilir.

```
int *p;
```

```
p = (int *) 0x1FC3;
```

Burada bellekte p gösterici değişkeninin tutulduğu yere 0x1FC3 sayısal değeri yerleştirilecektir.

...
0001
1111
1100
0011
...

p p göstericisinin bellekte bulunduğu yere 1FC3 değeri yazılıyor.

Göstericilere kendi türlerinden bir adres bilgisi atanmasının gerektiğini söylemiştik. Şimdiye kadar adres bilgilerini yalnızca adres sabitleri şeklinde gördük ve örneklerimizde de göstericilere adres sabitlerini atadık. Peki adres sabitleri dışında adres bilgileri taşıyan başka ifadeler mevcut mudur? Evet. Örneğin diziler konusunu incelerken dizi isimlerinin, nesne göstermediğini, bir adres bilgisi olduğunu söylemiştik, şimdi bu konuyu daha detaylı olarak inceleyeceğiz:

Dizi isimleri bir adres bilgisi belirtir. Adres bilgisinin iki bileşeni olduğuna göre örneğin:

```
char s[10];
```

gibi bir dizi tanımlaması yapıldığında dizi ismi olan s bir adres bilgisi belirtecektir. Peki bu bilginin tür bileşeni ve sayısal bileşenleri nelerdir?

Dizi isimleri, türleri dizinin türleriyle aynı ve sayısal bileşenleri dizi için bellekte ayrılan bloğun başlangıç yerini gösteren bir adres bilgisi olarak ele alınırlar. Örneğin yukarıdaki örnekte dizinin bellekte aşağıdaki şekilde yerleştirildiğini düşünelim:

```
s[0] 1C00
```

s[1]	1C01
s[2]	1C02
s[3]	1C03
s[4]	1C04
s[5]	1C05
s[6]	1C06
s[7]	1C07
s[8]	1C08
s[9]	1C09

Bu durumda dizi ismi olan **s**, **char** türden 1C00 adresine eşdeğerdir.

Göstericilere kendi türlerinden bir adres bilgisi atamak gerektiğine göre aşağıdaki atamaların hepsi legal 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 göstericinin içine aynı türden bir dizinin ismi atanmalıdır. Örneğin:

```
int *p;
char s[] = "Necati";
```

```
p = s;
```

Yukarıdaki örnekte **int** türden bir göstericiye **char** türden bir adres bilgisi atanmıştır. Derleme zamanında error oluşumuna neden olmaz ama yanlıştır, ileride detaylı olarak incelenecektir.

C dilinde hiçbir değişkenin ya da dizinin tahsisat yeri 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 error oluşturur. Çünkü dizi isimleri nesne göstermezler.

Özetle, adres belirten 3 tür ifade ele alındı.

1. Adres sabitleri.
2. Göstericiler.
3. Dizi isimleri.

Göstericiler içlerinde adres bilgileri taşıdıklarına göre bir göstericiye aynı türden başka bir göstericinin değerinin atanması da tamamen uygundur.

```
int *p, *q;
```

```
p = (int *) 0x1AA0;
q = p;
```

Yukarıdaki atama ile q göstericisine p göstericisinin değeri atanmıştır. Yani bu atama deyiminden sonra q göstericisinin de içinde (**int** \*) 0x1AA0 adresi bulunacaktır.

...	
0001 1010	p p göstericisine (int *) 1AA0 adresi atanıyor.
1010 0000	
...	
...	
...	
0001 1010	q q göstericisine de p göstericisinin içindeki değer atanıyor
1010 0000	

```
int k;
```

gibi bir tanımlama yaptığımızda k değişkeni **int** türündendir, içindeki değer **int** olarak yorumlanacaktır.

```
20
```

gibi, tek bir sabitten oluşan bir ifade de **int** türündendir, çünkü 20 **int** türden bir sabittir. 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. (sağ taraf değeridir)

```
int *ptr;
```

gibi bir tanımlama yapıldığında ptr nesnesinin türü nedir? ptr içinde **int** türden bir adres sakladığına göre ptr nesnesinin türü **int** türden bir adrestir.

```
int a[100];
```

Yukarıdaki tanımlamadan sonra

```
a
```

gibi bir ifade kullanılırsa bu ifadenin türü de "int türden bir adres" dir. Ancak ptr ifadesi nesne gösteren bir ifadeyken, yani bir sol taraf değeriye, a ifadesi nesne gösteren bir ifade değeridir. Sol taraf değeri olarak kullanılamaz.

Yine bir adres sabitinden oluşan

```
(int *) 0x1A00
```

ifadesinin türü de **int** türden bir adrestir. Ancak bu ifade de sol taraf değeri değildir.

Görüldüğü gibi göstericiler, belirli bir adres türünden, nesnelerdir. (sol taraf değerleridir.)

## Gösterici Operatörleri

C dilinde toplam 4 tane gösterici operatörü vardır. Bu operatörler göstericiler ve adres bilgileriyle kullanılabilirler.

Gösterici operatörleri şunlardır :

*	içerik operatörü	indirection operator (dereferencing operator)
&	adres operatörü	address of operator
[ ]	köşeli parantez operatörü	index operator (subscript operator)
->	ok operatörü	arrow operator

Bu operatörlerden ok operatörü yapı türünden adreslerle kullanıldığı için yapılar konusunda detaylı olarak incelenecektir.

## Gösterici Operatörlerinin Ayrıntılı İncelenmesi

### & Adres Operatörü (address of operator)

Adres operatörü tek operand alan önek konumunda bir operatördür. (unary prefix). Operatör öncelik tablosunun ikinci seviyesinde yer alır. Bu operatörün ürettiği değer operandı olan nesnenin adresidir. Yani operatörün ürettiği adres bilgisinin sayısal bileşeni nesnenin bellekteki fiziksel adres numarası, tür bileşeni ise nesnenin türü ile aynı türdür. & operatörünün operandı mutlaka bir nesne olmalıdır. Çünkü yalnızca nesnelerin (sol taraf değerlerinin) adres bilgilerine ulaşılabilir. & operatörüne operand olarak nesne olmayan bir ifade gönderilirse bu durum derleme zamanında hata oluşumuna neden olacaktır. (Borland derleyicileri bu durumda şöyle bir error mesajı verirler : "must take address of memory location")

**int** k;

&k

ifadesini ele alalım. Bu ifadenin ürettiği değer **int** türden bir adres bilgisidir. Adres bilgilerinin iki bileşeni olduğunu söylemiştik. Yukarıdaki ifadenin ürettiği bilginin sayısal bileşeni k nesnesinin bellekte yerleştirildiği yerin başlangıç adresi, tür bileşeni ise **int** türüdür. Zira k değişkeni **int** türden bir değişkendir.

& operatörü diğer tek operand alan (unary) operatörler gibi, operatör öncelik tablosunun 2. seviyesinde bulunur. Bilindiği gibi bu öncelik seviyesinin öncelik yönü "sağdan sola"dır.

Adres operatörü ile elde ettiğimiz adres aynı türden bir göstericiye atanmalıdır. Örneğin aşağıdaki programda bir göstericiye farklı türden bir adres atanmıştır:

```
char ch = 'x';
int *p;
```

```
p = &ch;          /* error değil ama yanlış */
```

Bu durumda C'de adres belirten 4 tür ifade görmüş olduk:

1. Adres sabitleri.
2. Göstericiler.
3. Dizi isimleri.
4. & operatörü ile oluşturulmuş ifadeler.

Tabi bu operatörün ürettiği adres bilgisi nesne değildir. Örneğin:



```
int x;
```

```
++&x      /* error */
```

gibi bir işlem error ile neticelenir. ++ operatörünün operandı nesne olmalıdır. Yukarıdaki ifadede ++ operatörüne operand olan &x ifadesi bir nesne göstermemektedir. Yalnızca bir adres değeridir.

### \* İçerik Operatörü (indirection operator)

İçerik operatörü de önek konumunda bulunan ve tek operand alan bir operatördür (unary prefix). Bir gösterici, \* operatörünün operandı olursa, elde edilen ifade p göstericisinin içerisindeki RAM adresinde bulunan, nesneyi temsil eder. Dolayısıyla, \* operatörü ile oluşturulan bir ifade bir nesneyi temsil etmektedir, ve sol taraf değeri olarak kullanılabilir.

```
int a;
```

gibi bir bildirimde a nesnesinin türü **int**'dir. Çünkü a nesnesi içerisinde **int** türden bir bilgi tutulmaktadır.

```
int *p;
```

bildiriminde p nin **türü** int türden bir adrestir.

```
char *ptr;
```

gibi bir bildirimden iki şey anlaşılır:

ptr **char** türden bir göstericidir. İçerisine **char** türden bir adres bilgisi yerleştirilir. ptr göstericisi \* operatörü ile birlikte kullanıldığında elde edilen nesne **char** türdendir. Yani \*ptr **char** türden bir nesnedir.

Örneğin:

```
int *p;
```

```
p = (int *) 0x1FC3;  
*p = 100;
```

Burada \*p'nin türü **int**'dir. Dolayısıyla \*p = 100 gibi bir işlemde yalnızca 0x1FC3 byte'ı değil, 0x1FC3 ve 0x1FC4 bytelerinden her ikisi birden etkilenir.

Göstericinin içerisindeki adresin sayısal bileşeni nesnenin düşük anlamlı byte'ının adresini içerir. Bu durumda bir göstericinin içerisine RAM'deki herhangi bir bölgenin adresi atanabilir. Daha sonra \* operatörü ile o RAM bölgesine erişilebilir.

\* operatörünün operandı bir adres bilgisi olmak zorundadır. Yani operand adres sabiti olabilir. Dizi ismi olabilir. Bir gösterici olabilir. Adres operatörü ile elde edilmiş bir adres ifadesi olabilir.

\* operatörü yalnız göstericilerle değil, her tür adres bilgisi ile (adres sabitleri ve dizi isimleri vs.) de kullanılabilir. Bu operatör operandı ile belirtilen adresteki nesneye erişmekte kullanılır. Bu operatör ile elde edilen değer, operandı olan adreste bulunan değerdir.

\* operatörü bir adrese uygulandığında ifade bir nesne belirtir. Nesnenin türü operand olarak kullanılan adresin türü ile aynı türdendir. Örneğin :

```
int main()  
{  
    char s[] = "Balıkesir";
```

```
    putchar(*s);

    return 0;
}
```

\* operatörü operatör öncelik tablosunun 2. düzeyinde sağdan sola öncelikli bulunmaktadır. Örneğin :

```
*s + 1;
```

ifadesinde önce \*s yapılır. Sonra + operatörü yapılır. Oysa ifade \*(s + 1) biçiminde olsaydı önce + operatörü yapılırdı.

Derleyiciler \* operatörünün çarpma operatörü mü yoksa adres operatörü mü olduğunu ifade içerisindeki kullanımına bakarak anlar. Çarpma operatörü iki operand alırken adres operatörü önek konumundadır ve tek operand alır. Örnek :

```
*s * 2
```

ifadesinde 1. \* adres operatörü iken 2. \* aritmetik çarpma operatörüdür.

### **[ ] Köşeli Parantez (index) Operatörü :**

Dizi elemanlarına erişmekte kullandığımız köşeli parantez aslında unary prefix bir gösterici operatörüdür.

[ ] içerisine tamsayı türünden bir ifade yazılır. (İfadenin değeri pozitif olmak zorunda değildir.)

[ ] operatörünün operandı dizi ismi olmak zorunda değildir. Bir adres bilgisi olmak zorundadır.

[ ] operatörünün operandı bir dizi ismi olabilir. Gösterici olabilir. Diğer adres belirten ifadeler olabilir.

p[n] ile \*(p + n) tamamen eşdeğerdir.

Yani köşeli parantez operatörü bir adresten n ilerisinin içeriğini almak için kullanılır. [ ] operatörü ile elde edilen nesnenin türü operandı olan adresin türü ile aynı türdendir. Örneğin:

```
#include <stdio.h>

int main()
{
    char s[] = "İstanbul";
    char *p;

    p = s + 1;
    putchar(p[2]);
    return 0;
}
```

[ ] operatörü öncelik tablosunun en yüksek düzeyinde bulunur. Örneğin:

```
&p[n]
```

ifadesinde önce [ ] yapılır, nesne elde edilir daha sonra nesnenin adresi alınır.

[ ] içerisindeki ifadenin sayısal değeri negatif olabilir. Örneğin p[-2] geçerli bir ifadedir.

```
#include <stdio.h>

int main()
```

```
{
    char ch = '0';

    (&ch)[0] = 'b';
    putchar(ch);
    return 0;
}
```

&ch ifadesi parantez içine alınmasaydı [] operatörünün önceliği söz konusu olacağından önce ch[0] yapılırdı. Buradan da bir nesne elde edilemeyeceğinden (& operatörünün operandının nesne olması gerektiğinden) hata oluşurdu.

## Adreslerle İşlemler / Adreslerin Artırılması ve Eksiltilmesi (gösterici aritmetiği)

C dilinde adreslerle tamsayı türünden sayılar toplama ve çıkarma işlemlerine sokulabilir. İki adres aynı türden ise karşılaştırma operatörleri ile işleme sokulabilir. Burada adreslerin sayısal bileşenleri işleme sokulmaktadır.

C dilinde bir adres bilgisi tamsayı türleri ile artırılabilir veya eksiltilebilir. Sonuç aynı türden bir adres bilgisi olur. Bir adres bilgisi 1 artırıldığında adresin sayısal bileşeni adresin türünün uzunluğu kadar artmaktadır. Bu durumda örneğin DOS'da **char** türden bir göstericiyi, 1 artırdığımızda adresin sayısal bileşeni 1 **int** türden bir göstericiyi 1 artırdığımızda ise adresin sayısal bileşeni 2 artar. Bu durumda p[n] ifadesi p adresinden n byte ilerisinin içeriği değil, p adresinden n \* p göstericisinin türünün uzunluğu kadar byte ilerinin içeriği anlamına gelir.

İki adres bilgisinin toplanması faydalı bir işlem olmadığı gerekçesiyle yasaklanmıştır. Ancak iki adres bilgisi birbirinden çıkartılabilir. İki adres birbirinden çıkartılırsa sonuç **int** türden olur. İki adres birbirinden çıkartıldığında önce adreslerin sayısal bileşenleri çıkartılır, sonra elde edilen değer adresin türünün uzunluğuna bölünür. Örneğin a **int** türden bir adres olmak üzere:

&a[2] - &a[0] ifadesinden elde edilen değer 2'dir.

```
#include <stdio.h>

int main()
{
    int a[5] = {1, 2, 3, 4, 5};
    int *p;

    p = a;
    printf("%p adresindeki değer = %d\n", p, *p);
    ++p;
    printf("%p adresindeki değer = %d\n", p, *p);
    return 0;
}
```

## ++ ve -- operatörlerinin gösterici operatörleriyle birlikte kullanılması

### ++\*p durumu

```
x = ++*p;
```

\*p = \*p + 1; anlamına gelir. \*p bir nesne gösterdiği için nesnenin değeri 1 artırılabilecektir.

Yukarıdaki deyimle x değişkenine \*p nesnesinin artırılmış değeri atanacaktır.

### \*++p durumu

p göstericisinin değeri 1 artırılır. Artırılmış adresteki nesneye ulaşılmış olur.

```
x = *++p;
```

deyimi ile x değişkenine artırılmış adresteki bilgi atanır.

### **\*p++ durumu**

++ operatörü ve \* operatörünün ikisi de 2. öncelik seviyesindedir ve bu öncelik seviyesine ilişkin öncelik yönü sağdan soladır. Önce ++ operatörü ele alınır ve bu operatör ifadenin geri kalan kısmına p göstericisinin artmamış değerini gönderir. Bu adresteki nesneye ulaşılır ve daha sonra p göstericisinin değeri 1 artırılır.

x = \*p++;

deyimi ile x değişkenine \*p nesnesinin değeri atanır daha sonra p göstericisinin değeri 1 artırılır.

### **&x++ /\* error \*/**

x nesnesinin artmamış değeri ifadenin geri kalanına gönderilir. (Adres operatörüne operand olur.) Bu da bir nesne olmadığı için error oluşur. (Adres operatörünün operandı nesne olmak zorundadır.)

### **&++x /\* error \*/**

x nesnesinin artmış değeri ifadenin geri kalanına gönderilir. (Adres operatörüne operand olur.) Bu da bir nesne olmadığı için error oluşur. (Adres operatörünün operandı nesne olmak zorundadır.)

### **++&x /\* error \*/**

x nesnesinin adresi ++ operatörüne operand olur. ++ operatörünün operandı nesne olmak zorunda olduğu için derleme zamanında error oluşacaktır.

& operatörü ile ++ ya da -- operatörlerinin her türlü kombinasyonu derleme zamanında hata oluşmasına neden olur.

### **++p[i] Durumu**

İndex operatörü 1. öncelik seviyesinde, ++ operatörü ise 2. öncelik seviyesindedir. Bu durumda önce derleyici tarafından indeks operatörü ele alınır. p[i] ifadesi bir nesne gösterir. Dolayısıyla ++ operatörüne operand olmasında bir sakınca yoktur. Söz konusu ifade

p[i] = p[i] + 1;

anlamına gelmektedir. Yani p[i] nesnesinin değeri 1 artırılacaktır.

### **p[i]++ Durumu**

x = p[i]++;

Önce p[i] nesnesinin artmamış değeri üretilir, ifadenin geri kalanına 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 1 artırılır. Daha sonra artırılmış adresteki bilgi x değişkenine atanır.

### **p[i++] Durumu**

```
x = p[i++];
```

Burada p[i] nesnesinin değeri x değişkenine atanır. Daha sonra i değişkeni 1 artırılır.

### **p++[i] durumu**

```
x = p++[i];
```

Pek tercih edilen bir ifade değildir. Burada önce p[i] x değişkenine atanır. Sonra p göstericisinin değeri 1 artırılır.

## **Göstericilere İlk Değer Verilmesi**

Diğer türden değişkenlerde olduğu gibi göstericilere de tanımlanmaları sırasında ilk değer verilebilir. Göstericilere ilk değer 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;
```

## **Fonksiyon Parametre Değişkeni Olarak Göstericilerin Kullanılması**

Bir fonksiyonun parametre değişkeni herhangi bir türden gösterici olabilir.

```
void func(int *p)
{
    ...
}
```

Bir fonksiyonun parametre değişkeni bir gösterici ise fonksiyon da aynı türden bir adres bilgisi ile çağırılmalıdır.

Bir fonksiyonun başka bir fonksiyonun yerel değişkenini değiştirebilmesi için o fonksiyonun yerel değişkeninin adresini parametre olarak alması gerekir.

```
#include <stdio.h>

void func(int *p)
{
    *p = 20;
}

int main()
{
    int a = 10;

    func(&a);
    printf("%d\n", a);
    return 0;
}
```

Bir fonksiyon bir değer elde edip, çağırılan fonksiyona bu değeri iletmek isterse 2 yöntem kullanılabilir:

1. Elde edilen değer çağırılan fonksiyon tarafından geri dönüş değeri olarak üretilir.

2. Elde edilen değer çağırılan fonksiyonun göndermiş olduğu adrese yerleştirilir. Tabi bunun için çağırılan fonksiyonun parametre değişkeninin bir gösterici olması gerekir. Bir örnekle gösterelim:

Kendisine gönderilen bir sayının faktoriyelini hesaplayana ve bu değeri parametre olarak gönderilen adrese kopyalayan bir fonksiyon yazalım.

```
void factorial(int n, long *p);

#include <stdio.h>

int main()
{
    long a;

    factorial(7, &a);
    printf ("%d! = %ld", 7, a);

    return 0;
}

void factorial(int n, long *p);
{
    if (n == 0 || n == 1)
        *p = 1;
    for (*p = 1; n > 0; n--)
        *p *= n;
}
```

a bir yerel değişken olsun. C dilinde bir fonksiyon

```
func(a);
```

biçiminde çağırılmışsa, çağırılan bu fonksiyonun, a değişkenini değiştirme şansı yoktur. (Bu tür fonksiyon çağırımına "değer ile çağırma" (call by value) denmektedir. Fonksiyonun a değişkenini değiştirebilmesi için

```
func(&a);
```

biçiminde çağırılması gerekir. Örneğin scanf fonksiyonuna & operatörü ile bir nesnenin adresinin arguman olarak yollanmasının nedeni budur. Bu şekilde fonksiyon çağırmaya C'de "adres ile çağırma" (call by reference) denmektedir.

int türden iki yerel nesnenin değerlerini değiştirmek (swap etmek) istediğimizi düşünelim. Bunu bulunduğumuz fonksiyon içerisinde aşağıdaki gibi yapabiliriz:

swap işleminin bir fonksiyon tarafından yapılmasını istersek, aşağıdaki gibi bir fonksiyon işimizi görür müydü?

Yukarıdaki program çalışıldığında ekrana

```
a = 10
b = 20
```

yazacaktır. Zira swap fonksiyonu a ve b değişkenlerinin değerlerini değiştirmemiştir. Zaten yerel nesneler olan a ve b değişkenlerinin değerleri ancak adresleri bir fonksiyona gönderilerek değiştirilebilirdi. Oysa bizim yukarıdaki swap fonksiyonumuz a ve b değişkenlerinin değerlerini parametre değişkenleri olan x ve y değişkenlerine kopyalıyor. Yani değerleri değiştirilen

parametre değişkenleri  $x$  ve  $y$ 'nin değerleri. İsteddiğimiz amacı gerçekleştirecek fonksiyon dışarıdan adres alacağı için gösterici parametre değişkenlerine sahip olmalı:

## Dizilerin Fonksiyonlara Göstericiler Yoluyla Geçirilmesi

Bir diziyi fonksiyona geçirebilmek için dizinin başlangıç adresinin ve uzunluğunun geçirilmesi yeterlidir. Dizinin başlangıç adresini alacak parametre değişkeninin aynı türden bir gösterici olması gerekir. Fonksiyonun 2. parametresi dizinin uzunluğunu tutacak **int** türden bir değişken olabilir. Bir dizinin başlangıç adresini parametre olarak alan fonksiyon dizi elemanlarına köşeli parantez operatörü ya da içerik operatörü ile erişebilir. Ancak dizi elemanlarının kaç tane olduğu bilgisi fonksiyon tarafından bilinemez. Bu nedenle dizi uzunluğunu ikinci bir parametre olarak fonksiyona yolluyoruz. Örnek:

Dizi uzunluğu hakkında fonksiyona bilgi vermek amacı ile başka teknikler de kullanılır. Dizi uzunluğunu dizinin birinci parametresinde saklayabiliriz. Yukarıdaki fonksiyonu, dizi uzunluğunu birinci dizi elemanında saklayarak yeniden yazalım:

Bir dizi içerisinde, çözülen problemin koşullarına bağlı olarak, belirli bir değer bulunması olanaksızsa, bu değer dizi sonunu belirten bir işaret olarak kullanılabilir. Bir dizi içinde öğrencilerin aldıkları notları sakladığımızı varsayalım. Öğrencilerin negatif bir not almaları mümkün olmadığına göre dizi içerisinde negatif bir sayı bulunamaz. Bu durumda biz -1 gibi bir değeri dizi sonunu belirtmek amacıyla kullanabiliriz. Yukarıdaki örneği yeniden yazalım:

$n$  elemanlı **int** türden bir dizinin aritmetik ortalamasını bulan getavg fonksiyonunu tasarlanması:

```
#include <stdio.h>

double getavg(int *p, int size);

int main()
{
    int s[10] = {1, 23, 45, -4, 67, 12, 22, 90, -3, 44};
    double average;

    average = getavg(s, 10);
    printf("dizinin aritmetik ortalaması = %lf\n", average);
    return 0;
}

double getavg(int *p, int size)
{
    int i;
    double total = 0;

    for (i = 0; i < size; ++i)
        total += p[i];
    return total / size;
}
```

size elemanlı **int** türden bir dizi içerisindeki en büyük sayıya geri dönen getmax isimli fonksiyonun yazılması:

```
#include <stdio.h>

int getmax(int *p, int size);

int main()
{
    int s[10] = {1, 23, 45, -4, 67, 12, 22, 90, -3, 44};
```

```

    printf("%d\n", getmax(s, 10));
    return 0;
}

int getmax(int *p, int size)
{
    int i, max;

    max = p[0];
    for (i = 1; i < size; ++i)
        if (max > p[i])
            max = p[i];

    return max;
}

```

Bubble sort yöntemiyle **int** türden bir diziyi küçükten büyüğe sıraya dizen fonksiyon örneği:

```

#include <stdio.h>

#define SIZE 10

void bsort(int *p, int size);

void bsort(int *p, int size)
{
    int i, k, temp;

    for (i = 0; i < size - 1; ++i)
        for (k = 0; k < size - 1 - i; ++k)
            if (p[k] > p[k+1]) {
                temp = p[k];
                p[k] = p[k+1];
                p[k+1] = temp;
            }
}

int main()
{
    int a[SIZE] = {3, 4, 5, 8, 78, 12, -2, 11, 41, -34};
    int i;

    bsort(a, SIZE);
    for (i = 0; i < SIZE; ++i)
        printf("a[%d] = %d\n", i, a[i]);
    return 0;
}

```

## Geri Dönüş Değeri Adres Türünden Olan Fonksiyonlar

Bir fonksiyon parametre değişkeni olarak bir gösterici alabildiği gibi, adres türlerinden bir geri dönüş değerine de sahip olabilir. Geri dönüş değerinin adres olduğu, fonksiyon tanımlanırken \* operatörü ile aşağıdaki biçimde belirtilmektedir.

```

<adresin türü> *<fonksiyon ismi> ([parametreler])
{
    ...
}

```

Örneğin **int** türden bir adrese geri dönen ve parametre değişkeni almayan sample isimli fonksiyon aşağıdaki gibi tanımlanabilir:



```
int *sample(void)
{
    ...
}
```

Yukarıdaki fonksiyon tanımlama ifadesinden \* atomu kaldırılırsa fonksiyon **int** türden bir değer döndürür.

Bir adres türüne geri dönen bir fonksiyonun çağırılma ifadesi, aynı türden bir göstericiye atanmalıdır.

```
int *ptr;
```

```
ptr = sample();
```

gibi.

Benzer şekilde:

```
char *str;
```

```
char *func(void) {
    ...
}
```

```
str = func();
```

```
...
```

Adrese geri dönen fonksiyonlara C programlarında çok rastlanır. Standart C fonksiyonlarından bir çoğu, adres türünden bir değer döndürür.

Bir dizinin en büyük elemanını bulup bu elemanın değerine geri dönen getmax fonksiyonunu daha önce tasarlamıştık. Şimdi aynı fonksiyonu en büyük elemanın adresine geri dönecek şekilde tasarlayalım:

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

```
int *getmax(int *p, int size);
```

```
int main()
{
    int s[10] = {1, 23, 45, -4, 67, 12, 22, 90, -3, 44};

    printf("%d\n", *getmax(s, 10));
    return 0;
}
```

```
int *getmax(int *p, int size)
{
    int *pmax, i;

    pmax = p;
    for (i = 1; i < size; ++i)
        if (*pmax < p[i])
            pmax = p + i;
    return pmax;
}
```

## Göstericilere İlişkin Uyarılar ve Olası Gösterici Hataları

### Bir Göstericiye Farklı Türden Bir Adres Atanması:

Bir göstericiye farklı türden bir adres atandığında, C derleyicileri durumu şüpheyle karşılayarak bir uyarı mesajı verirler. Yani bu işlem hata (error) değil uyarı gerektirmektedir. Ancak derleyici yine de farklı türden adresin sayısal bileşenini hedef göstericiye atar. Borland derleyicileri bu durumda aşağıdaki uyarı mesajını verirler:

warning : suspicious pointer conversion in function .....

Bu işlemin faydalı bir gerekçe ile yapılma ihtimali zayıftır. Ancak bazan bilinçli olarak da yapılabilir. O zaman bilinçli olarak yapıldığı konusunda derleyiciyi ikna etmek gerekmektedir.

```
int *p;
char s[ ] = "Ankara";

p = s;          /* uyarı */
```

Uyarı bilinçli tür dönüştürmesiyle kesilebilir.

```
p = (int *) s;
```

Tür dönüştürme operatörü kullanılarak **char** türden bir adres **int** türden bir adrese dönüştürülmüştür.

Bir göstericiye farklı türden bir adres atanması durumunda, otomatik tür dönüşümü sonucunda, atama operatörünün sağ tarafındaki ifadenin türü, atama operatörünün sol tarafındaki nesnenin türüne dönüştürülecek, ve dönüştürülmüş olan adres bilgisi nesneye atanacaktır.

Örneğin **int** türden bir göstericiye **char** türden bir adres atadığımızı düşünelim:

```
int main()
{
    char s[20] = "Necati Ergin";
    int *p;

    p = s;
    ...
    return 0;
}
```

Bu durumda **int** türden p göstericisini içerik operatörü ya da indeks operatörüyle kullandığımızda elde ettiğimiz nesne **char** türden değil **int** türden olacaktır. Böyle bir durumun bilinçli olarak yapılmış olma ihtimali azdır, ve bilinçli olarak yapılması durumunda tür dönüştürme operatörü kullanılarak, derleyicinin vereceği uyarı mesajı kesilmelidir. (derleyici ikna edilmelidir)

C++ dilinde bir göstericiye farklı türden bir adres atanması durumunda, derleme zamanında hata oluşacaktır. Yani bu durum uyarı seviyesinden error seviyesine yükseltilmiştir.

### Bir Göstericiye Adres Olmayan Bir Değerin Atanması

Bu da bilinçli olarak yapılma ihtimali çok az olan bir işlemdir. C derleyicileri şüpheli olan bu durumu bir uyarı mesajı ile programcıya bildirirler. Örneğin bu uyarı mesajı Borland derleyicilerinde:

"nonportable pointer conversion"

şeklinde. Peki bir göstericiye adres bilgisi olmayan bir değer atarsak ne olur? Yine otomatik tür dönüşümü söz konusudur. Atama operatörünün sağ tarafındaki ifadenin türü, atama operatörünün sol tarafında bulunan nesne gösteren ifadenin türüne çevrilerek, atama yapılacaktır. Dolayısıyla, atanan değer göstericinin türünden bir adrese çevrilecek ve göstericiye atanacaktır. Örneğin:

```
int main()
{
    int k, *ptr;

    k = 1356;
    ptr = k;    /* uyarı */
    ...
    return 0;
}
```

Yukarıdaki örnekte `ptr = k;` atama deyiminden sonra bellekte `ptr` göstericisi için ayrılan yere 1356 yazılacaktır. Yani `*ptr` ifadesi ile **int** türden bir nesneye ulaşılacaktır ki bu nesne de 1356 ve 1357 adreslerinde bulunan nesnedir.

Bu durumun bilinçli olarak yapılması durumunda yine tür dönüştürme operatörü kullanılarak derleyici ikna edilmeli ve uyarı mesajı kesilmelidir:

```
int main()
{
    int k, *ptr;

    k = 1356;
    ptr = (int *) k; /* uyarı yok */
    ...
    return 0;
}
```

## Yazıların Fonksiyonlara Parametre Olarak Geçirilmesi

Yazılar karakter dizilerinin içerisinde bulunurlar. Bir yazıyı fonksiyona parametre olarak geçirebilmek için yazının yalnızca başlangıç adresini fonksiyona geçirmek yeterlidir. Yani fonksiyon yazının (karakter dizisinin) başlangıç adresi ile çağırılır. Yazıyı içinde tutan **char** türden dizinin uzunluk bilgisini fonksiyona geçirmeye gerek yoktur. Çünkü yazıları saklamak amacıyla kullanılan karakter dizilerinin sonu null (ASCII karakter tablosunda 0 sıra numaralı karakter.) karakterle işaretlenmiştir. Karakter dizileri üzerinde işlem yapan algoritmalar dizinin sonunu null karakter yardımıyla belirlerler.

Yazılarla işlem yapan fonksiyon **char** türden bir gösterici ile üzerinde işlem yapacağı karakter dizisinin başlangıç adresini alır. Fonksiyon, null karakter görene kadar bir döngü ile yazının bütün karakterlerine erişir.

`str` **char** türünden bir gösterici olmak üzere yazı üzerinde null karakter görene kadar işlem yapabilecek döngüler şöyle oluşturulabilir:

```
while (*str != '\0') {
    ...
    ++str;
}

for (i = 0; str[i] != '\0'; ++i) {
    ...
}
```

Örneğin `puts` fonksiyonunun parametre değişkeni **char** türünden bir göstericidir. `puts` fonksiyonunu `myputs` ismi ile yeniden yazalım:

```
#include <stdio.h>

void myputs(char *str)
{
    while (*str != '\0') {
        putchar(*str);
        ++str;
    }
    putchar('\n');
}

int main()
{
    char s[] = "Deneme";

    myputs(s);
    return 0;
}
```

## Yazılarla İlgili İşlem Yapan Fonksiyonlar

Bir grup standart C fonksiyonu vardır ki, bu fonksiyonlar bir yazının başlangıç adresini parametre olarak alarak yazı ile ilgili birtakım faydalı işlemler yaparlar. Bu fonksiyonlara string fonksiyonları denir. String fonksiyonlarının prototipleri string.h dosyası içindedir. Bu fonksiyonlardan bazılarını inceleyelim :

### strlen Fonksiyonu

Bu fonksiyon bir yazının karakter uzunluğunu (kaç karakterden oluştuğu bilgisini) elde etmek için kullanılır.

Fonksiyonun prototipi:

**unsigned int** strlen(**char** \*str);

şeklinde. Fonksiyonun parametre değişkeni uzunluğu hesaplanacak yazının başlangıç adresidir. Fonksiyon null karakter görene kadar karakterlerin sayısını hesaplar.

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

int main()
{
    char s[100];

    printf("bir yazı giriniz : ");
    gets(s);
    printf("%d\n", strlen(s));
    return 0;
}
```

standart C fonksiyonu olan strlen fonksiyonunu kendimiz yazsaydık aşağıdaki biçimlerde yazabilirdik:

```
#include <stdio.h>

unsigned  strlen1 (char *str);
unsigned  strlen2(char *str);
unsigned  strlen3(char *str);
```

```
int main()
{
    char s[100];

    printf("bir yazı giriniz : ");
    gets(s);
    printf("yazının uzunluğu : %d\n", strlen1(s));
    printf("yazının uzunluğu : %d\n", strlen1(s));
    printf("yazının uzunluğu : %d\n", strlen1(s));

    return 0;
}

unsigned int  strlen1(char *str)
{
    unsigned int length = 0;

    while (*str != '\0') {
        ++length;
        ++str;
    }
    return length;
}

unsigned int  strlen2(char *str)
{
    unsigned int len;

    for (len = 0; str[len] != '\0'; ++len)
        ;
    return len;
}

unsigned int  strlen3(char *str)
{
    char *ptr = str;

    while (*str != '\0')
        str++;
    return str - ptr;
}
```

strlen2 fonksiyonunda len değişkeni hem döngü değişkeni hem de sayaç olarak kullanılmıştır. null karakter '\0' sayısal değer olarak 0 değerine eşit olduğu için yukarıdaki döngülerin koşul ifadelerini aşağıdaki şekilde yazabilirdik:

```
while (*str)
for (i = 0; str[i]; ++i)
```

Yukarıdaki koşul ifadelerinde de \*str ya da str[i] 0 değerine eşit olduğunda kodun akışı döngü dışındaki ilk deyimle devam ederdi. Ancak okunabilirlik açısından null karakterle karşılaştırmamızın açıkça yapılması daha uygundur.

## strchr fonksiyonu

Fonksiyonun ismi olan strchr "string character" sözcüklerinin kısaltılarak birleştirilmesinden elde edilmiştir. strchr fonksiyonu bir karakter dizisi içinde belirli bir karakteri aramak için kullanılan standart bir C fonksiyonudur. Prototipi string.h dosyası içinde bulunmaktadır.

strchr fonksiyonunun prototipi aşağıdaki gibidir:

```
char *strchr(char *str, int ch);
```

Bu fonksiyon, 2. parametresi olan `ch` karakterini, 1. parametresi olan `str` adresinden başlayarak null karakter görene kadar arar. (Aranan karakter null karakterin kendisi de olabilir.) Fonksiyonun geri dönüş değeri, `ch` karakterinin yazı içinde bulunabilmesi durumunda ilk bulunduğu yerin adresidir. Eğer `ch` karakteri yazı içinde bulunamazsa, fonksiyon `NULL` adresine geri dönecektir.

`strchr` fonksiyonunu kendimiz yazsaydık aşağıdaki şekilde yazabilirdik:

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

int main()
{
    char s[100];
    char *p, ch;

    printf("bir yazı giriniz : ");
    gets(s);
    printf("yazı içinde arayacağınız karakteri giriniz : ")
    scanf("%c", &ch);
    p = strchr(s, ch);
    if (p == NULL) {
        printf("aranan karakter bulunamadı\n");
    }
    else
        puts(p);
    return 0;
}

char *my_strchr(char *str, int ch)
{
    while (*str != '\0') {
        if (ch == *str)
            return str;
        ++str;
    }
    if (ch == '\0')
        return str;

    return NULL;
}
```

Yukarıda verilen `main` fonksiyonunda `strchr` fonksiyonunu çağırdığımız yerde kendi yazdığımız `my_strchr` fonksiyonunu çağırarak fonksiyonun doğru çalışıp çalışmadığını test edebiliriz.

```
if (ch == '\0')
    return str;
```

deyimleri aranan karakterin null karakter olup olmadığını test etmek için eklenmiştir. **while** döngüsü yerine **for** döngüsü de kullanabilirdik;

```
...
int i;
for (i = 0; str[i] != '\0'; ++i)
    if (ch == str[i])
        return (str + i);
...
```

## strcpy fonksiyonu

fonksiyonun ismi olan strcpy, string ve copy sözcüklerinin kısaltılarak birleştirilmesinden elde edilmiştir. Fonksiyon ikinci parametresinde tutulan adresten başlayarak, NULL karakter görene kadar, (NULL karakter dahil olmak üzere) tüm karakterleri sırasıyla birinci parametresinde tutulan adresten başlayarak sırayla yazar. Fonksiyonun prototipi string.h dosyası içindedir. Fonksiyonun prototipi aşağıdaki gibidir:

```
char *strcpy(char *dest, char *source);
```

Fonksiyonun geri dönüş değeri kopyalamanın yapılmaya başlandığı adrestir. (Yani dest adresi)

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

int main()
{
    char dest[100] = "C öğreniyoruz!";
    char source[100];

    printf("kopyalanacak yazıyı giriniz : ");
    gets(source);
    printf("kopyalama yapılmadan önce kopyalamanın yapacağı yerde bulunan yazı : \n");
    puts(dest);
    strcpy(dest, source);
    printf("kopyalamadan sonra kopyalamanın yapıldığı yerde bulunan yazı : \n");
    puts(dest);
    return 0;
}
```

strcpy fonksiyonunu kendimiz yazmak isteseydik aşağıdaki şekilde yazabilirdik :

```
char *_strcpy(char *dest, char *source)
{
    int i;

    for (i = 0; (dest[i] = source[i]) != '\0'; ++i)
        ;
    return dest;
}
```

fonksiyon içinde kullanılan for döngüsünde önce atama yapılmış, daha sonra atama ifadesinin değeri (k, bu da atama operatörünün sağ tarafında bulunan değerdir) NULL karakter ile karşılaştırılmıştır. Böylece ilgili adrese NULL karakter de kopyalandıktan sonra döngüden çıkmıştır.

fonksiyonu aşağıdaki gibi de yazabilirdik :

```
...
for (i = 0; source[i] != '\0'; ++i)
    dest[i] = source[i];
dest[i] = '\0';
...
```

for döngüsünde indeks operatörü kullanıldığı için, birinci parametre değişkenine kopyalanan dest adresi değiştirilmemiş ve fonksiyonun sonunda bu dest adresi ile geri dönmüştür. Fonksiyonun tasarımında while döngüsü kullansaydık, ve dest içindeki adresi değiştirseydik, fonksiyonun dest göstericisinin ilk değeriyle geri dönebilmesini sağlayabilmek için, dest göstericisindeki değeri değiştirmeden önce, bu değeri başka bir gösterici içinde saklamak gerekecekti :

```
char *_strcpy(char *dest, char *source)
```

```
{
    char *temp = dest;

    while ((*source++ = *dest++) != '\0')
        ;
    return temp;
}
```

Yazılan fonksiyonların doğru çalışıp çalışmadıklarını aynı main fonksiyonu ile test edebiliriz.

### strcat fonksiyonu

fonksiyonun ismi string ve concatenate sözcüklerinin kısaltılarak birleştirilmesiyle elde edilmiştir.

strcat fonksiyonu bir karakter dizisinin sonuna başka bir karakter dizisinin kopyalanması amacıyla kullanılmaktadır. Fonksiyonun prototipi string.h dosyası içindedir. Fonksiyonun prototipi aşağıdaki gibidir:

```
char *strcat(char *s1, char *s2);
```

strcat fonksiyonu eklemenin yapılacağı ve başlangıç adresi s1 birinci parametre değişkeninde tutulan yazının sonundaki NULL karakteri ezerek, başlangıç adresi ikinci parametre değişkeninde tutulan yazıyı birinci yazının sonuna (NULL karakter de dahil olmak üzere) eklemektedir. Yani işlem sonunda s1 dizisi s2 dizisi kadar büyümektedir.

Fonksiyonun geri dönüş değeri, sonuna eklemenin yapıldığı yazının başlangıç adresidir. (Yani s1 adresi)

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

int main()
{
    char s1[100], s2[100];

    printf("sonuna ekleme yapılacak yazıyı giriniz : ");
    gets(s1);
    printf("girdiğiniz yazının uzunluğu = %d\n", strlen(s1));
    printf("eklemek istediğiniz yazıyı giriniz : ");
    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 fonksiyonunu kendimiz yazsaydık aşağıdaki şekilde yazabilirdik :

```
char *_strcat(char *s1, char *s2)
{
    char *temp = s1;

    while (*s1 != '\0')
        ++s1;
    while ((*s1++ == *s2++) != '\0')    /* strcpy(s1, s2); */
        ;
    return temp;
}
```



Yazılan fonksiyonun doğru çalışıp çalışmadığını aynı main fonksiyonu ile test edebiliriz.

### strset fonksiyonu

Standart olmayan bu fonksiyon derleyicilerin çoğunda bulunur. Fonksiyonun ismi string ve set sözcüklerinin kısaltılarak birleştirilmesinden elde edilmiştir. Bir karakter dizisinin belirli bir karakterle doldurulması amacıyla kullanılmaktadır. Prototipi string.h dosyası içinde bulunmaktadır. Fonksiyonun prototipi aşağıdaki gibidir :

```
char *strset(char *str, int ch);
```

Fonksiyon birinci parametre değişkeninde başlangıç adresi olan yazıyı NULL karakter görene kadar ikinci parametre değişkeninde tutulan karakterle doldurur. (yazının sonundaki NULL karaktere dokunmaz) .

Fonksiyonun geri dönüş değeri yine doldurulan yazının başlangıç adresidir.

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

int main()
{
    char s[100];
    int ch;

    printf("bir yazı giriniz :");
    gets(s);
    printf("yazıyı hangi karakterle doldurmak istiyorsunuz : ");
    ch = getchar();
    printf("\nyazının %c karakteriyle doldurulduktan sonraki hali : %s\n", ch,
        strset(s, ch));

    return 0;
}
```

strset fonksiyonunu kendimiz yazsaydık aşağıdaki şekillerde yazabilirdik :

```
#include <stdio.h>

char *_strset(char *str, int ch)
{
    int i;

    for (i = 0; str[i] != '\0'; ++i)
        str[i] = ch;
    return str;
}

char *_strset2(char *str, int ch)
{
    char *temp = str;

    while (*str != '\0') {
        *str = ch;
        ++str;
    }
    return temp;
}
```

### strcmp fonksiyonu

Standart bir C fonksiyonudur. Fonksiyonun ismi string ve compare sözcüklerinin kısaltılarak birleştirilmesinden elde edilmiştir. Fonksiyon 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 tablosunda 'd' karakterinden sonra gelmektedir.

ankara yazısı ANKARA yazısından daha büyüktür. Çünkü küçük harfler ASCII tablosunda büyük harflerden sonra gelmektedir.

kalem yazısı kale yazısından daha büyüktür.

strcmp fonksiyonunun string.h içerisindeki prototipi aşağıdaki gibidir :

```
int strcmp(char *s1, char *s2);
```

fonksiyon 1. parametre değişkeninde başlangıç adresi tutulan yazı ile, ikinci parametre değişkeninde başlangıç adresi tutulan yazıları karşılaştırır.

fonksiyonun geri dönüş değeri

- 1. yazı 2.yazıdan daha büyükse pozitif bir değere
- 1. yazı 2. yazıdan daha küçükse negatif bir değere
- 1.yazı ve 2. yazı birbirine eşit ise 0 değerine geri döner.

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

int main()
{
    char s[20];
    char password[ ] = "Mavi ay";

    printf("parolayı giriniz : ");
    gets(s);
    if (!strcmp(s, password))
        printf("Parola doğru!..\n");
    else
        printf("Parola yanlış!..\n");
    return 0;
}
```

strcmp fonksiyonunu kendimiz yazsaydık aşağıdaki şekillerde yazabilirdik :

```
int _strcmp(char *s1, char *s2)
{
    while (*s1 == *s2) {
        if (*s1 == '\0')
            return 0;
        ++s1;
        ++s2;
    }
    return *s1 - *s2;
}
```

strrev fonksiyonu

Standart olmayan bu fonksiyon derleyicilerin çoğunda bulunur. Fonksiyonun ismi ingilizce string ve reverse sözcüklerinin kısaltılarak birleştirilmesinden elde edilmiştir. Karakter dizilerini ters çevirmek amacıyla kullanılır. string.h içerisinde yer alan prototipi aşağıdaki gibidir :

```
char *strrev(char *str);
```

fonksiyon parametre değişkeninde başlangıç adresi tutulan yazıyı tersyüz eder. Fonksiyonun geri dönüş değeri tersyüz edilen yazının başlangıç adresidir.

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

main()
{
    char s[100];

    printf("ters çevirilecek yazıyı giriniz : ");
    gets(s);
    printf("yazınızın ters çevrilmiş hali : \n");
    puts(strrev(s));
    return 0;
}
```

strrev fonksiyonunu kendimiz yazsaydık aşağıdaki şekilde yazabilirdik :

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

char *strrev(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;
}
```

### strncpy fonksiyonu

Standart bir C fonksiyonudur. Fonksiyonun ismi ingilizce, string number copy sözcüklerinin kısaltılarak birleştirilmesinden elde edilmiştir. Fonksiyon bir yazının (karakter dizisinin) ilk n karakterini başka bir yazıya(karakter dizisine) kopyalamakta kullanılır. Fonksiyonun string.h içerisindeki prototipi aşağıdaki gibidir :

```
char *strncpy(char *dest, char *source, int n);
```

fonksiyon 1. parametre değişkeninde başlangıç adresi tutulan yazıya, ikinci parametre değişkeninde adresi tutulan yazıdan, üçüncü parametresinde tutulan sayıda karakteri kopyalar. Fonksiyonun geri dönüş değeri kopyalamanın yapılacağı adrestir. (Yani dest adresi)

üçüncü parametre olan n sayısı eğer kopyalanacak yazının uzunluğundan daha küçük ya da eşit ise fonksiyon kopyalama sonunda NULL karakteri birinci dizinin sonuna eklemeyiz.

n <= strlen(source) ise NULL karakter eklenmiyor.

üçüncü parametre olan n sayısı eğer kopyalanacak yazının uzunluğundan daha büyük ise fonksiyon kopyalama sonunda NULL karakteri birinci dizinin sonuna ekler.

`n > strlen(source)` ise NULL karakter ekleniyor.

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

int main()
{
    char dest[100], source[100];
    int n;

    printf("birinci yazıyı giriniz : ");
    fflush(stdin);
    gets(dest);
    printf("ikinci yazıyı giriniz : ");
    gets(source);
    printf("ikinci yazıdan 1. yazıya kaç karakter kopyalamak istiyorsunuz : ");
    scanf("%d", &n);
    strncpy(dest, source, n);
    printf("kopyalamadan sonra 1. yazının yeni şekli : ");
    puts(dest);
    return 0;
}
```

`strncpy` fonksiyonunu kendimiz yazsaydık aşağıdaki şekilde yazabilirdik :

```
#include <stdio.h>

char *_strncpy(char *dest, char *source, int n)
{
    int i;

    for (i = 0; i < n && source[i] != '\0'; ++i)
        dest[i] = source[i];
    if (n > i)
        dest[i] = '\0';
    return dest;
}
```

### `strncat` fonksiyonu

Standart bir C fonksiyonudur. Fonksiyonun ismi ingilizce string number concatenate sözcüklerinin kısaltılarak birleştirilmesiyle elde edilmiştir. Bir karakterden dizisinin sonuna başka bir karakter dizisinden belirli bir sayıda karakteri kopyalamak amacıyla kullanılır. `string.h` içinde bulunan prototipi aşağıdaki gibidir :

```
char *strncat(char *s1, char *s2, int n);
```

fonksiyon 1. parametre değişkeni içinde başlangıç adresi verilen yazının sonuna (NULL karakteri ezerek), 2. parametresinde başlangıç adresi tutulan karakter dizisinden, 3. parametresinde tutulan tamsayı adedi kadar karakteri kopyalar.

fonksiyonun geri dönüş değeri sonuna ekleme yapılacak yazının başlangıç adresidir. (yani `s1` adresi)

fonksiyonun çalışmasını açıklayacak bir örnek aşağıda verilmiştir :

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

int main()
```

```

{
    char dest[100], source[100];
    int n;

    printf("birinci yazıyı giriniz : ");
    fflush(stdin);
    gets(dest);
    printf("ikinci yazıyı giriniz : ");
    gets(source);
    printf("ikinci yazıdan 1. yazının sonuna kaç karakter kopyalamak istiyorsunuz : ");
    scanf("%d", &n);
    strncat(dest, source, n);
    printf("eklemeden sonra 1. yazının yeni şekli : ");
    puts(dest);
    return 0;
}

```

### strncmp fonksiyonu

Standart bir C fonksiyonudur. Fonksiyonun ismi İngilizce string number compare sözcüklerinin kısaltılarak birleştirilmesiyle elde edilmiştir. strcmp fonksiyonuna benzer, ancak bu fonksiyon iki yazının tümünü değil de, belirli bir sayıda karakterlerini karşılaştırma amacıyla kullanılır. fonksiyon 1. parametre değişkeninde başlangıç adresi tutulan yazı ile, ikinci parametre değişkeninde başlangıç adresi tutulan yazıların, üçüncü parametresinde tutulan sayıdaki karakterlerini karşılaştırır.

### fonksiyonun geri dönüş değeri

1. yazının n karakteri 2.yazının n karakterinden daha büyükse pozitif bir değere  
 1. yazının n karakteri 2.yazının n karakterinden daha küçükse negatif bir değere  
 1.yazı ve 2. yazının n karakteri birbirine eşit ise 0 değerine geri döner.  
 /\* büyük harf küçük harf duyarlılığı ile (case sensitive) bir yazı içinde başka bir yazıyı arayan mystrstr fonksiyonu. Fonksiyon aranan yazıyı aramanın yapılacağı yazı içinde bulursa bulunduğu yazının başlangıç adresine, bulamazsa NULL adresine geri dönmektedir.  
 \*/

```

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

char *mystrstr(char *s1, char *s2);

int main()
{
    char s1[100];
    char s2[100];
    char *ptr;

    clrscr();
    printf("aramanın yapılacağı yazıyı girin : ");
    gets(s1);
    printf("aranacak yazıyı girin :");
    gets(s2);
    ptr = mystrstr(s1, s2);
    if (ptr)
        puts(ptr);
    else
        puts("aradığınız yazı bulunamadı\n");
    getch();

    return 0;
}

```

```

}

char *mystrstr(char *s1, char *s2)
{
    int i, j;
    int lens1, lens2;

    lens1 = strlen(s1);
    lens2 = strlen(s2);

    for (i = 0; lens1 - i >= lens2; ++i, ++s1) {
        for (j = 0; s1[j] == s2[j]; ++j)
            if (s2[j + 1] == '\0')
                return s1;
    }
    return NULL;
}

```

### strupr ve strlwr fonksiyonları

Standart C fonksiyonları olmamalarına karşın hemen hemen her derleyicide bulunurlar. İsimleri string upper ve string lower kelimelerinin kısaltılarak birleştirilmesinden elde edilmiştir. Bu fonksiyonların bir yazının tüm karakterleri için büyük harf küçük harf dönüştürmesi yaparlar. Fonksiyonların geri dönüş değerleri parametresi ile verilen adresin aynısıdır. Geri dönüş değerlerine genellikle ihtiyaç duyulmaz. Her iki fonksiyon da ingiliz alfabesinde olan harfler için dönüşüm yaparlar. Türkçe karakterler için de dönüşüm yapacak bir fonksiyon kullanmak istiyorsak kendimiz yazmalıyız.

```

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

int main()
{
    char s[ ] = "C programcısı olmak için çok çalışmak gerekir!";

    strupr(s);
    puts(s);
    return 0;
}

```

strupr ve strlwr fonksiyonlarını kendimiz de aşağıdaki şekilde

```

#include <stdio.h>,
#include <ctype.h>

char *_strupr(char *str)
{
    char *temp = str;

    while (*str != '\0') {
        if (islower(*str)) /* if (*str >= 'a' && *str <= 'z') */
            *str = toupper(*str); /* *str = *str - 'a' + 'A'; */
        ++str;
    }
    return temp;
}

char *_strlwr(char *str)
{
    char *temp = str;

```

```
while (*str != '\0') {
    if (isupper(*str)) /* if (*str >= 'A' && *str <= 'Z') */
        *str = tolower(*str); /* *str = *str - 'A' + 'a'; */
    ++str;
}
return temp;
}
```

strlwr vestrupr fonksiyonlarının türkçe versiyonlarını aşağıdaki şekilde yazabiliriz :

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

char *struprtrk(char *str)
{
    char trklower[ ] = "çğıöşü";
    char trkupper[ ] = "ÇĞİÖŞÜ";
    int index;
    char *p, *temp;

    temp = str;
    while (*str != '\0') {
        p = strchr(trklower, *str);
        if (p) {
            index = p - trklower;
            *str = trkupper[index];
        }
        else
            *str = toupper(*str);
        ++str;
    }
    return temp;
}

char *strlwrtrk(char *str)
{
    char trklower[ ] = "çğıöşü";
    char trkupper[ ] = "ÇĞİÖŞÜ";
    int index;
    char *p, *temp;

    temp = str;
    while (*str != '\0') {
        p = strchr(trkupper, *str);
        if (p) {
            index = p - trkupper;
            *str = trklower[index];
        }
        else
            *str = tolower(*str);
        ++str;
    }
    return temp;
}
```

char türden bir göstericiyi NULL karaktere öteleme

Bir göstericiyi NULL karakteri gösterir hale getirmek için 3 yöntem kullanılabilir :

char \*p;

- i. `p += strlen(p);`
- ii. `p = strchr(p, '\0');`
- iii. `while (*p != '\0')`  
`++ptr;`

Yukarıdakiler içinde en hızlı çalışabilecek kod üçüncüsüdür. 2. biçim tercih edilebilir ama 1. yöntem pek tercih edilmez. Aşağıdaki döngüden çıktıktan sonra p göstericisi NULL karakterden bir sonraki adresi göstermektedir :

```
while (*p++ != '\0')
    ;
```

## GÖSTERİCİ HATALARI

Göstericileri kullanarak RAM üzerinde herhangi bir bölgeye erişebiliriz. 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 çıkabilecektir. Bu bozukluk tüm sistemi olumsuz yönde etkileyebilir.

Kim tarafından kullanıldığını bilmediğimiz bellek bölgelerine güvenli olmayan bölgeler denir. Güvenli olmayan bölgelere veri aktarılmasına da "gösterici hataları" denir.

Gösterici hataları yapıldığında sistem kilitlenebilir, programlar yanlış çalışabilirler. Gösterici hataları sonucundaki olumsuzluklar hemen ortaya çıkmayabilir.

Gösterici hataları güvenli olmayan bölgelere erişildiğinde değil oralara veri aktarıldığında oluşur.

Gösterici hataları derleme sırasında derleyici tarafından tespit edilemezler. Programın çalışma zamanı sırasında olumsuzluklara yol açarlar. tanımlama yöntemiyle tahsis 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

1) ilk değer 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 rasgele değerlerle başlatılırlar. (garbage value)

Yerel bir gösterici tanımlandıktan sonra, herhangi bir şekilde bu göstericiye bir değer ataması yapılmaz ise göstericinin içinde rasgele bir değer bulunacağından, bu gösterici \*operatörü ile ya da [ ] operatörü ile kullanıldığında, bellekte rasgele bir yerde bulunan bir nesneye ulaşılacaktır. Dolayısıyla, elde edilen nesneye bir atama yapıldığı zaman, bellekte, bilinmeyen rasgele bir yere yazılmış olunur. Bu durum tipik bir gösterici hatasıdır. Örnekler :

```
main()
{
    int *p;          /* p göstericisinin içinde rasgele bir adres var */

    *p = 25;         /* p göstericisinin içindeki adreste buluna nesneye 25 değeri atanıyor */
}
```

Yukarıdaki örnekte rasgele bir yere (güvenli olmayan bir yere) veri aktarılmaktadır. Verinin aktarıldığı yerde işletim sisteminin, derleyicinin ya da bellekte kalan başka bir programın



(memory resident) kodu bulunabilir. Verinin aktarıldığı yerde programın kendi kodu da bulunabilir.

İlk değer verilmemiş global göstericilerin içinde (ya da statik yerel göstericilerin içerisinde) sıfır değeri bulunmaktadır. Sıfır sayısı (NULL adresi) göstericilerde test amacıyla kullanılmaktadır. Bu adrese bir veri aktarılması durumunda , derleyicilerin çoğunda isteğe bağlı olarak bu hata çalışma zamanı (runtime) sırasında kontrol edilmektedir. Örnek :

```
char *ptr;

main()
{
    *p = 'm';           /* NULL pointer assignment */
}
```

Yukarıdaki kodun çalıştırılmasında "NULL pointer assignment" şeklinde bir çalışma zamanı hatasıyla karşılaşılabilir. Bu kontrol derleyicinin alılabilen program içerisine yerleştirdiği "kontrol kodu" sayesinde yapılmaktadır.

İlk değer verilmemiş göstericilerin neden olduğu hatalar fonksiyon çağırımlarıyla da ortaya çıkabilir :

```
main()
{
    char *ptr;

    gets(ptr);   /* ????? */
}
```

Yukarıdaki kod parçasında gets fonksiyonu ile klavyeden alınan karakterler, bellekte rasgele bir yere yazılacaktır. gets fonksiyonu klavyeden alınan karakterleri kendisine arguman olarak gönderilen adresten başlayarak yerleştirdiğine göre, daha önceki örnekte verilen hata klavyeden girilen bütün karakterler için söz konusudur.

## 2) Güvenli olmayan ilk değerlerin neden olduğu gösterici hataları

Bir göstericiye ilk değer verilmesi , o göstericinin güvenli bir bölgeyi gösterdiği anlamına gelmeyecektir. 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 içerisinde belli amaçlar için kullanılan güvenli bölgeler varsa da 1FC5 böyle bir bölgeyi göstermemektedir.

## 3) dizi taşmalarından doğan gösterici hataları

Bilindiği gibi bir dizi tanımlaması gördüğünde derleyici, derleme sırasında dizi için bellekte toplam dizi uzunluğu kadar yer ayırır. C'nin seviyesi ve felsefesi gereği dizi taşmaları derleyici tarafından tespit edilmemektedir.

```
main()
{
    int a[10], k;
```

```

    for (k = 0; k <= 10; ++k)
        a[k] = 0;
    ...
}

```

Yukarıdaki örnekte döngü k değişkeninin 10 değeri için de sürecektir. Oysa a[10] elemanı için bir tahsisat yapılmamıştır. Dizinin son elemanı a[9] elemanıdır. Derleyici a dizisi için a[0]'dan başlayarak a[9] elemanı için bellekte toplam 10 elemanlık yani 20 byte'lık bir yer tahsis edecektir. Oysa tahsis edilmemiş olan a[10] bölgesinin kim tarafından ve ne amaçla kullanıldığı hakkında herhangi bir bilgi bulunmamaktadır.

(Bu konuda standartlar tarafından bir zorunluluk bulunmamakla birlikte derleyicilerin çoğu, ardışıl olarak tanımlanan elemanları bellekte ardışıl olarak (contiguous) olarak yerleştirirler. Ama bu garanti altına alınmış bir özellik değildir. Yani bunun garanti altına alındığı düşünülerek kod yazılması problemlere yol açacaktır. Yukarıdaki örnekte a[10] bölgesi derleyicilerin çoğunda a dizisinden hemen sonra tanımlanan ve döngü değişkeni olarak kullanılan k değişkenine ayrılacaktır. Dolayısıyla a[10] elemanına 0 değerini vermekle aslında k değişkenine 0 değeri verilecek ve bu da döngünün sonsuz bir döngüye dönüşmesine yol açabilecektir.)

Bilindiği gibi indeks operatörü de bir gösterici operatörüdür, bu operatörü kullanarak dizi için tahsis edilen alanın dışına atama yapılabilir. C dili bu konuda bir kontrol mekanizması getirmemektedir. Ayrıca a dizisi için a[-1], a[-2].. gibi ifadelerin de sentaks açısından geçerlidir ve buraya yapılacak atamalar da gösterici hatalarına yol açacaktır.

Bazan dizi taşmalarına gizli bir biçimde fonksiyonlar da neden olabilirler. Örneğin :

```

char str[12];
...
printf("adı soyadı : ");
gets(str);

```

yukarıdaki kod parçasında str dizisi için toplam 12 karakterlik (12 byte) yer tahsis edilmiştir. gets fonksiyonu klavyeden alınan karakterleri kendisine gönderilen adresten başlayarak belleğe yerleştirdikten sonra, NULL karakteri de eklemektedir. O halde yukarıdaki örnekte programın çalışması sırasında 11 karakterden fazla giriş yapılması gösterici hatasına neden olacaktır. Sonlandırıcı karakter olan NULL karakter de ('\0') bellekte bir yer kaplayacağı için tahsis edilmiş bir bölge içinde bulunması gerekir. Örneğin klavyeden girilen ad ve soyad :

Necati Ergin

olsun. gets fonksiyonu bu karakterleri aşağıdaki gibi yerleştirecektir :

...	
s[0]	'N'
s[1]	'e'
s[2]	'c'
s[3]	'a'
s[4]	't'
s[5]	'i'
s[6]	' '
s[7]	'E'
s[8]	'r'
s[9]	'g'
s[10]	'i'
s[11]	'n'
	'\0'
	sonlandırıcı karakter güvenli olmayan bir yere yazılıyor.
...	

Görüldüğü gibi sonlandırıcı karakter, dizi için tahsis edilen bölgenin dışına yerleştirilmiştir. Bu örnekte girilen isim daha uzun olsaydı, tahsis edilmemiş bölgeye daha fazla karakter yazılacaktı. Bu tür hatalarla karşılaşmamak için dizi yeteri kadar uzun olmalı ya da standart bir

C fonksiyonu olan gets fonksiyonu yerine, dizi uzunluğundan daha fazla sayıda eleman yerleştirilmesine izin vermeyecek özel bir giriş fonksiyonu yazılmalı ve kullanılmalıdır.

stringlerle ilgili işlemler yapan standart C fonksiyonlarından strcpy, strcat, strncpy, strncat fonksiyonlarının yanlış kullanılması da benzer hataları oluşturabilecektir. örneğin :

```
char s[10] = "Eskişehir";
...
strcpy(s, "Kahramanmaraş");          /* gösterici hatası */
strcat(s, "li");                      /* gösterici hatası */
strncpy(s, "Kahramanmaraş", 15)      /* gösterici hatası */
strncat(s, "liyim", 5);               /* gösterici hatası */
```

yukarıdaki fonksiyonlardan hepsi tahsis edilmeyen bir bölgeye yazma yaptıkları için gösterici hatasına neden olacaktır.

## void Göstericiler

void göstericiler herhangi bir türden olmayan göstericilerdir. Bildirimlerinde void anahtar sözcüğü kullanılır.

```
void *ptr;
void *str;
```

void göstericilerin tür bilgisi yoktur. void göstericilerde adreslerin yalnızca sayısal bileşenleri saklanır. Bu yüzden void göstericilerle diğer türden göstericiler (adresler) arasında yapılan atamalarda uyarı söz konusu değildir. void göstericileri kullanırken tür dönüştürme operatörünü kullanmaya gerek yoktur.

```
char *str;
void *p;
```

```
....
```

```
str = p;          /* herhangi bir uyarı mesajı alınmaz */
```

```
p = str;          /* herhangi bir uyarı mesajı alınmaz */
```

Benzer biçimde fonksiyon çağırma ifadelerinden argümanların fonksiyon parametre değişkenlerine kopyalanmasında da, void göstericiler söz konusu olduğunda, adres türlerinin farklılığından dolayı uyarı mesajı alınmaz.

```
funk1(void *p)
{
    ...
}

main()
{
    int *ptr;
    ...
    funk1(ptr); /* p = ptr gibi bir atama işlemi yapılmaktadır. Uyarı mesajı alınmaz. */
}

funk2(int * p)
{
    ...
}
```

```
int main()
{
    void *ptr;
    ...
    funk2(ptr); /* p = ptr gibi bir atama işlemi yapılmaktadır. Uyarı mesajı alınmaz. */
}
```

void göstericiler belirli bir türe ait olmadıkları için, tür bilgisine sahip olan göstericiler üzerinde uygulanan bazı işlemler void türden göstericilere uygulanamazlar:

1. void türden göstericilerin \* ya da [ ] operatörleriyle kullanılmaları derleme zamanında error oluşturacaktır. Çünkü bu operatörler nesneye erişmek için tür bilgisine ihtiyaç duyarlar.

```
long l[50];
void *vptr;

vptr = l;          /* normal bir işlem. void türden göstericiye long türden bir adres atanıyor. */
...
*p = 10L;          /* hata! void gösterici * operatörüyle kullanılamaz. */
...
p[2] = 25L;        /* hata! void gösterici [ ] operatörüyle kullanılamaz. */
```

2. void türden bir göstericiden bir tamsayının toplanması, ya da void türden bir göstericiden bir tamsayının çıkartılması derleme zamanında hata oluşturur. Çünkü pointer aritmetiğine göre bir göstericiyi n kadar artırdığımızda, gösterici içindeki adresin sayısal bileşeni n \* göstericinin türünü uzunluğu kadar artar. void göstericilerin türleri olmadığı için bu durumda sayısal bileşenin ne kadar artacağı da bilinemez.

void türden göstericiler ++ ve -- operatörlerine operand olamazlar.

```
++ptr;
```

ifadesi

```
ptr = ptr + 1;
```

ifadesine eşdeğerdir.

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
void *ptr = &k;
void *p;
p = ptr + 2; /* error! void türden bir göstericiye bir tamsayı toplanıyor! */
++ptr;      /*error ! */
--ptr;      /* error! */
ptr += 2;   /*error */
ptr -= 3;   /* error */
```

3. Benzer şekilde, void türden iki adres birbirinden çıkartılamaz. (Diğer türden adresler için, aynı türden iki adresin birbirinden çıkartılmasının tamamen legal olduğunu ve ifadenin değerinin pointer aritmetiğine göre bir tamsayı olduğunu hatırlayalım!)

```
void *p1, *p2;
int k;
...
k = p1 - p2; /* error! void türden adresler birbirinden çıkarılamaz! */
```

void göstericiler adreslerin sayısal bileşenlerini geçici olarak saklamak amacıyla kullanılırlar. Diğer tür göstericiler arasındaki atama işlemlerinde uyarı ya da hata oluşturmadıklarından

dolayı, türden bağımsız adres işlemlerinin yapıldığı fonksiyonlarda parametre değişkeni biçiminde de bulunabilirler. Örneğin:

```
void sample(void *p)
{
    ...
}
```

sample isimli fonksiyonun parametre değişkeni void türden bir gösterici olduğundan bu fonksiyona arguman olarak herhangi türden bir adres bilgisi gönderilebilir. Yani sample fonksiyonu herhangi türden bir adres ile çağırılabilir. Bu durumda derleme zamanında bir hata oluşmadığı gibi bir uyarı mesajı da alınmaz.

C dilinde fonksiyonlar void türden adreslere de geri dönebilirler. Void türden adresler derleme zamanında bir hata oluşmaksızın, ve bir uyarı mesajı alınmadan herhangi türden bir göstericiye atanabildiği için, bu fonksiyonların geri dönüş değerleri herhangi türden bir göstericiye atanabilir:

```
main()
{
    int *p;
    char * str;

    p = sample();    /* hata oluşmaz ve uyarı mesajı alınmaz. */
    ...
    str = sample();  /* hata oluşmaz ve uyarı mesajı alınmaz. */
}

void *sample();
{
    ...
}
```

#### void göstericilere ilişkin uygulamalar

void göstericilerin kullanımını en iyi açıklayan örnek memcpy fonksiyonudur. memcpy fonksiyonu 2. parametresiyle belirtilen adresten başlayarak n sayıda byte'ı birinci parametresiyle belirtilen adresten başlayarak kopyalar. Fonksiyonun NULL karakterle ya da yazılarla bir ilişkisi yoktur, koşulsuz bir kopyalama yapar. Yani bir blok kopyalaması söz konusudur.

Uygulamada en fazla, aynı türden herhangi iki dizinin kopyalanması amacıyla kullanılır. Fonksiyonun prototipi string.h başlık dosyası içindedir.

#### Örnek :

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

#define      SIZE      10

void main()
{
    int a[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int b[SIZE];
    int i;

    memcpy(b, a, 2 * SIZE);
    for (i = 0; i < SIZE; ++i)
```

```
        printf("%d\n", b[i]);
    }
```

aşağıdaki fonksiyon çağırışı da eşdeğerdir.

```
char s[50] = "Ali";
char d[50];
```

```
strcpy(d, s);
memcpy(d, s, strlen(s) + 1);
```

memcpy fonksiyonunun gösterici olan parametreleri void türündendir. Çünkü hangi türden adres geçirilirse geçirilsin, derleyicilerden herhangi bir uyarı mesajı alınmaz.

```
void memcpy(void *dest, void *source, int n);
```

ancak memcpy fonksiyonunun çakışık blokların kopyalanmasında davranışı standart değildir. Çakışık blokların kopyalanmasında ayrı prototipe sahip olan memmove fonksiyonu tercih edilmelidir. Özellikle dizi elemanlarının kaydırılması gibi uygulamalarda çakışık blokların kopyalanması söz konusu olabilir. Bu durumda memmove fonksiyonu tercih edilmelidir. Standart C'de başı mem ile başlayan mem.... biçiminde bir grup fonksiyon vardır. Bu fonksiyonlar türden bağımsız olarak byte sayısı ile işlem yaparlar.

memset fonksiyonu

prototipi :

```
memset(void *p, char ch, int n);
```

Bu fonksiyon 1. parametresiyle belirtilen adresten başlayarak n tane byte'ı , 2. parametresiyle belirtilen bilgiyle doldurulur. Bu fonksiyon herhangi türden bir diziyi sıfırlamak amacıyla kullanılabilir. Örneğin :

```
double d[100];
```

```
memset(d, 0, sizeof(d));
```

void gösterici parametresine sahip olan fonksiyonların yazılması :

Böyle bir fonksiyon yazılırken parametre değişkeni olan void gösterici önce türü belirli bir göstericiye atanır, daha sonra işlemler bu göstericiyle gerçekleştirilir.

Örnek 1 :

```
void *memcpy (void *dest, void *source, int n)
{
    char *_dest = (char *) dest;
    char *_source = (char *) source;
    int i;

    for (i = 0; i < n; ++i)
        _dest[i] = _source[i];
    return dest;
}
```

aynı fonksiyonu şu şekilde de yazabilirdik :

```
void *memcpy (void *dest, void *source, int n)
{
    char *_dest = (char *) dest;
    char *_source = (char *) source;
```

```
    while (n-- > 0)
        *_dest++ = *_source++;
    return dest;
}

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]);
}
```

### memset fonksiyonunun yazılması

```
#include <stdio.h>

void *memset(void *p, int x, int n);

int main()
{
    int a[10];
    int i;

    memset(a, 0, 20);

    for (i = 0; i < 10; ++i)
        printf("%d\n", a[i]);
return 0;
}

void *memset(void *p, int x, int n);
{
    char *_p = (char *)p;

    while (n-- > 0)
        *_p++ = x;
    return p;
}
```

## Göstericilerle İlgili Örnek Uygulamalar

Uygulama 1 : long türden bir sayıyı verilen bir adrese virgüllerle ayırarak string olarak yazan longtoa fonksiyonunun yazılması. Fonksiyon gönderilen adrese geri dönecektir.

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

#define NEGATIVE (-1)
#define POSITIVE 1

char *longtoa(long val, char *str)
{
    int i, sign;
    char digit;

    if (val < 0) {
        sign = NEGATIVE;
        val = -val;
    }
```

```
    }
    else
        sign = POSITIVE;

    i = 0;

    do {
        digit = val % 10;
        str[i++] = digit + '0';
        val /= 10;

        if (i % 4 == 3 && val)
            str[i++] = ',';
    } while (val);
    str[i] = '\\0';
    strrev(str);

    return str;
}

int main()
{
    char str[80];
    long val;

    printf("bir sayı giriniz : ");
    scanf("%ld", &val);
    puts(longtoa(val, str));

    return 0;
}
```

**Uygulama 2:** büyük harf küçük harf duyarlılığı ile (case sensitive) bir yazı içinde başka bir yazıyı arayan `mystrstr` fonksiyonu. Fonksiyon aranan yazıyı aramanın yapılacağı yazı içinde bulursa bulunduğu yazının başlangıç adresine, bulamazsa NULL adresine geri dönmektedir.

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

char *mystrstr(char *s1, char *s2);

int main()
{
    char s1[100];
    char s2[100];
    char *ptr;

    clrscr();
    printf("aramanın yapılacağı yazıyı girin : ");
    gets(s1);
    printf("aranacak yazıyı girin :");
    gets(s2);
    ptr = mystrstr(s1, s2);
    if (ptr)
        puts(ptr);
    else
        puts("aradığınız yazı bulunamadı\\n");
    getch();
}

char *mystrstr(char *s1, char *s2)
{
```



```
int i, j;
int lens1, lens2;

lens1 = strlen(s1);
lens2 = strlen(s2);

for (i = 0; lens1 - i >= lens2; ++i, ++s1) {
    for (j = 0; s1[j] == s2[j]; ++j)
        if (s2[j + 1] == '\0')
            return s1;
}
return NULL;
}
```

**Uygulama 3:** klavyeden girilen bir yazı içerisindeki türkçe karakterlerin sayısını bulan fonksiyon.

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

int gettrknum(char *str);

void main()
{
    char s[100];
    printf("YAZIYI GİRİNİZ : ");

    gets(s);
    printf("\ntürkçe karakter sayısı = %d", gettrknum(s));
}

int gettrknum(char *str)
{
    int counter = 0;
    int i;

    char trk[ ] = "çğıöşüçĞİÖŞÜ";
    for (i = 0; str[i] != '\0'; ++i)
        if (strchr(trk, str[i]))
            ++counter;
    return counter;
}
```

**Uygulama 4:** Büyük harf küçük harf duyarlılığı olmadan iki stringi karşılaştıran fonksiyon.

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

int mystricmp(char *s1, char *s2);

void main()
{
    char s1[100];
    char s2[100];
    int result;
    int n = 10;

    while (n-- > 0) {
```

```

        clrscr();
        printf("1. yazıyı girin : ");
        gets(s1);
        printf("2. yazıyı girin :");
        gets(s2);
        result = mystricmp(s1, s2);
        if (result == 0)
            puts("s1 = s2");
        else if (result > 0)
            puts("s1 > s2");
        else
            puts("s1 < s2");
        getch();
    }
}

int mystricmp(char *s1, char *s2)
{
    while (toupper(*s1) == toupper(*s2)) {
        if (*s1 == '\0')
            return 0;
        ++s1;
        ++s2;
    }
    return toupper(*s1) - toupper(*s2);
}

```

**Uygulama 5 :** Kendisine başlangıç adresi gönderilen bir yazının içindeki boşlukları atan `rem_space` fonksiyonu. Fonksiyon kendisine gönderilen adrese geri dönecektir.

```

#include <stdio.h>

char *rem_space(char *ptr)
{
    int k;
    int indis = 0;

    for (k = 0; ptr[k] != '\0'; ++k)
        if (ptr[k] != ' ' && ptr[k] != '\t')
            ptr[indis++] = ptr[k];
    ptr[indis] = '\0';
    return ptr;
}

main()
{
    char s[100];

    printf("bir yazı giriniz ");
    gets(s);
    printf("bosluksuz yazı : %s", rem_space(s));
    return 0;
}

```

### ***göstericilerle ilgili örnek uygulamalar***

```

/*
mystrstr.c
büyük harf küçük harf duyarlılığı ile (case sensitive) bir yazı içinde başka bir yazıyı arayan mystrstr
fonksiyonu. Fonksiyon aranan yazıyı aramanın yapılacağı yazı içinde bulursa bulunduğu yazının başlangıç
adresine, bulamazsa NULL adresine geri dönmektedir.
*/

```

```

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

char *mystrstr(char *s1, char *s2);

int main()
{
    char s1[100];
    char s2[100];
    char *ptr;

    printf("aramanın yapılacağı yazıyı girin : ");
    gets(s1);
    printf("aranacak yazıyı girin :");
    gets(s2);
    ptr = mystrstr(s1, s2);
    if (ptr)
        puts(ptr);
    else
        puts("aradığınız yazı bulunamadın");
    getch();
    return 0;
}

char *mystrstr(char *s1, char *s2)
{
    int i, j;
    int lens1, lens2;

    lens1 = strlen(s1);
    lens2 = strlen(s2);

    for (i = 0; lens1 - i >= lens2; ++i, ++s1) {
        for (j = 0; s1[j] == s2[j]; ++j)
            if (s2[j + 1] == '\0')
                return s1;
    }
    return NULL;
}

/*
trknum.c
klavyeden girilen bir yazı içerisindeki türkçe karakterlerin
sayısını bulan fonksiyon
*/

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>

int gettrknum(char *str);

int main()
{
    char s[100];
    printf("YAZIYI GİRİNİZ : ");

    gets(s);
    printf("ntürkçe karakter sayısı = %d", gettrknum(s));
    return 0;
}

int gettrknum(const char *str)
{

```

```

int counter = 0;
int i;

char trk[ ] = "çğıöşüçğİÖŞÜ";
for (i = 0; str[i] != '_; ++i)
    if (strchr(trk, str[i]))
        ++counter;
return counter;
}

```

/\*

**stricmp.c**

Büyük harf küçük harf duyarlılığı olmadan iki stringi karşılaştıran  
bir fonksiyonun tasarımı fonksiyon

\*/

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>

```

```

int mystricmp(char *s1, char *s2);

```

```

int main()
{

```

```

    char s1[100];
    char s2[100];
    int result;
    int n = 10;

    while (n-- > 0) {
        printf("1. yazıyı girin : ");
        gets(s1);
        printf("2. yazıyı girin :");
        gets(s2);
        result = mystricmp(s1, s2);
        if (result == 0)
            puts("s1 = s2");
        else if (result > 0)
            puts("s1 > s2");
        else
            puts("s1 < s2");
        getch();
    }
    return 0;
}

```

```

int mystricmp(char *s1, char *s2)
{
    while (toupper(*s1) == toupper(*s2)) {
        if (*s1 == '_')
            return 0;
        ++s1;
        ++s2;
    }
    return toupper(*s1) - toupper(*s2);
}

```

/\*

**nowords.c**

adresi verilen bir yazıdaki kelime sayısını bulan

int no\_of\_words(char \*str);

fonksiyonunun tasarlanması

\*/

```

#include <stdio.h>
#include <ctype.h>

```

```

#define OUT 0

```

```
#define      IN                1
#define      MAX_LEN          200

int wcount(const char *str)
{
    int w_count = 0;
    int state_flag = OUT;

    while (*str != '\0') {
        if (isspace(*str) || ispunct(*str))
            state_flag = OUT;
        else if (state_flag == OUT) {
            state_flag = IN;
            ++w_count;
        }
        ++str;
    }
    return w_count;
}

int main()
{
    char s[MAX_LEN];

    printf("bir yazı giriniz : ");
    gets(s);
    printf("yazınızda %d sözcük var!.\n", wcount(s));
    return 0;
}
```

## 20 . BÖLÜM : STRINGLER

C programa dilinde iki tırnak içerisindeki ifadeler string ifadeleri ya da kısaca stringler denir. Örneğin:

```
"Kaan Aslan"
"x = %d\n"
"lütfen bir sayı giriniz : "
```

ifadelerinin hepsi birer stringdir.

Stringlerin tek bir atom olarak ele alındığını önceki derslerden hatırlıyoruz. C'de stringler aslında **char** türden bir adres olarak ele alınmaktadır. C derleyicileri, derleme aşamasında bir stringle karşılaştığında, önce bu stringi belleğin güvenli bir bölgesine yerleştirir, sonuna NULL karakteri ekler ve daha sonra string yerine yerleştirildiği yerin başlangıç adresini koyar. Bu durumda string ifadeleri aslında stringlerin bellekteki başlangıç yerini gösteren **char** türden birer adrestir. Örneğin:

```
char *p;
...
p = "Nuri Yılmaz";
```

gibi bir kodun derlenmesi sırasında, derleyici önce "Nuri Yılmaz" stringini belleğin güvenli bir bölgesine yerleştirir; daha sonra yerleştirdiği yerin başlangıç adresini string ifadesi ile değiştirir.

Stringler **char** türden bir adres olarak ele alındığına göre **char** türden göstericilere atanmalarında error ya da uyarı gerektiren bir durum söz konusu değildir.

### Stringlerin Fonksiyonlara Arguman Olarak Gönderilmesi

Parametre değişkeni **char** türden bir gösterici olan fonksiyonu **char** türden bir adres ile çağırmak gerektiğini biliyoruz. Çünkü doğal olarak **char** türden bir göstericiye **char** türden bir adres atanmalıdır.

Derleyiciler açısından stringler de **char** türden bir adres belirttiklerine göre, parametre değişkeni **char** türden gösterici olan bir fonksiyonu bir string ile çağırmak son derece doğal bir durumdur, ve C dilinde bu yapı çok kullanılır:

```
puts("Necati Ergin");
```

Burada derleyici "Necati Ergin" stringini belleğe yerleştirip sonuna NULL karakteri koyduktan sonra artık bu stringi, karakterlerini yerleştirdiği bellek bloğunun başlangıç adresi olarak görecektir. puts fonksiyonunun parametre değişkenine de artık **char** türden bir adres kopyalanacaktır. puts fonksiyonu parametre değişkeninde tutulan adresten başlayarak NULL karakteri görene kadar tüm karakterleri ekrana yazmaktadır. Bu durumda ekranda

Merhaba

yazısı çıkacaktır. Başka bir örnek:

```
char str[20];
...
strcpy(str, "Kaan Aslan");
```

Bu örnekte de "Kaan Aslan" stringi str adresinden başlayarak kopyalanmaktadır. String ifadelerinin bulunduğu yerde **char** türden bir adresin bulunduğu düşünülmelidir. Şimdi aşağıdaki ifadeyi yorumlayalım:

```
strcpy("Ali", "Kaan");
```

Derleyici derleme aşamasında iki stringi de bellekte güvenli bir bölgeye yerleştirir. Çalışma zamanı sırasında strcpy fonksiyonu "Ali" stringini gösteren adresten başlayarak "Kaan" stringini gösteren adrese NULL karakter görene kadar kopyalama yapacağına göre, bu ifadenin faydalı hiçbir anlamı olmayacaktır. Üstelik bu ifade, ikinci string birinciden uzun olduğu için bir gösterici hatasına da neden olur. Şimdi aşağıdaki ifadeyi inceleyelim:

```
p = "Kaan" + "Necati";
```

Yukarıdaki örnekte aslında "Kaan" stringinin başlangıç adresi ile "Necati" stringinin başlangıç adresi toplanmaktadır. Peki bu toplamın yararlı bir sonucu olabilir mi? Derleyiciler anlamsızlığından dolayı iki adresin toplanmasına izin vermezler. Fakat bir göstericiden başka bir göstericinin değerini çıkarmanın yararlı gerekçeleri olabilir. Bu yüzden göstericilerin birbirinden çıkartılması derleyicilerin hepsinde geçerli bir işlemdir. Gösterici aritmetiğine göre bir adresten aynı türden bir başka bir adres çıkartıldığı zaman elde edilen sonuç bir tamsayıdır, ve bu sayı iki adresin sayısal bileşenlerinin farkı değerinin adresin tür uzunluğuna bölünmesiyle elde edilir:

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

```
int main()
{
    char *p = (char *) 0x1AA8;
    char *q = (char *) 0x1AA0;
    long *lp = (long *) 0x1AA8;
    long *lq = (long *) 0x1AA0;

    printf("fark1 = %d\n", p - q);
    printf("fark2 = %d\n", lp - lq);
    return 0;
}
```

Yukarıdaki programın çalıştırılmasıyla ekrana:

```
fark1 = 8
fark2 = 2
```

yazılacaktır.

C derleyicileri kaynak kodun çeşitli yerlerinde tamamen özdeş stringlere rastlasa bile bunlar için farklı yerler ayırabilirler. Derleyici özdeş stringlerin sadece bir kopyasını da bellekte saklayabilir. Özdeş stringlerin nasıl saklanacağı bazı derleyicilerde programcının seçimine bırakılmıştır. Derleyicinin ayarları arasında bu konuyla ilgili bir seçenek bulunur.

Örneğin bir programın içerisinde:

```
printf("Dosya açılmıyor!..\n");
...
printf("Dosya açılmıyor!..\n");
...
printf("Dosya açılmıyor!..\n");
```

ifadelerinin bulunduğunu varsayalım. Farklı yerlerde "Dosya açılmıyor!..\n" gibi özdeş stringler bulunsa bile derleyici bunlar için tekrar ve farklı yerler ayırabilir. Bu da büyük uygulamalar için belleğin verimsiz kullanılmasına yol açabilir.

Bellekte yer ayırma işlemi derleme aşamasında yapılmaktadır. Aşağıdaki kod parçasının çalıştırılmasıyla ekrana hep aynı adres basılacaktır :

```
char *p;
```

```
int k;
...
for (k = 0; k < 10; ++k) {
    p = "Necati Ergin";
    printf("%p\n", p);
}
...
```

Eğer yer ayırma işlemi çalışma zamanı sırasında yapılsaydı, o zaman farklı adresler basılabilirdi.

Stringlerin doğrudan karşılaştırılması yanlış bir işlemdir.

```
...
if ("Ankara" == "Ankara")
    printf("dogru\n");
else
    printf("yanlış\n");
...
```

Yukarıdaki kodun çalıştırılması durumunda büyük bir olasılıkla ekrana "yanlış" yazdırılacaktır. Çünkü derleyicinin **if** parantezi içindeki karşılaştırma ifadesindeki iki "Ankara" stringinin bellekte ayrı yerlere yerleştirilmesi durumunda, bu iki string birbirinden farklı iki ayrı **char** türden adres olarak değerlendirilir. Benzer bir yanlışlık aşağıdaki kod parçasında da yapılmıştır.

```
char *str = "Mavi ay";
char s[20];

printf("parolayı giriniz : ");
gets(s);
if (str == s)
    printf("dogru parola\n");
else
    printf("yanlış parola\n");
```

s bir dizi ismidir ve derleyici tarafından dizinin yerleştirildiği bloğun başlangıcı olan **char** türden bir adres sabiti gibi ele alınır. str ise char türden bir göstericidir. str göstericisine "Mavi ay" stringi atandığında, derleyici önce "Mavi ay" stringini bellekte güvenli bir yere yerleştirir daha sonra stringin yerleştirildiği yerin başlangıç adresini str göstericisine atar. programı kullananın parola olarak "Mavi ay" girdiğini varsayalım. Bu durumda **if** deyimi içinde yalnızca s adresiyle str göstericisinin içindeki adresin eşit olup olmadığı kontrol edilmektedir. Bu adresler de eşit olmadıkları için ekrana "yanlış parola" yazılacaktır. İki yazının birbirine eşit olup olmadığı strcmp fonksiyonu ile kontrol edilmeliydi:

```
char *str = "Mavi ay";
char s[20];

printf("parolayı giriniz : ");
gets(s);
if (!strcmp(str, s))
    printf("dogru parola\n");
else
    printf("yanlış parola\n");
```

## Stringlerin Ömürleri

Stringler statik ömürlü nesnelerdir. Tıpkı global değişkenler gibi programın yüklenmesiyle bellekte yer kaplamaya başlarlar, programın sonuna kadar bellekte kalırlar. Dolayısıyla stringler çalışabilen kodu büyütür. Birçok sistemde statik verilerin toplam uzunluğunda belli bir sınırlama söz konusudur.



Stringler derleyici tarafından .obj modüle linker tarafından da .exe dosyasına yazılırlar. Programın yüklenmesiyle hayat kazanırlar.

Bir exe program 3 bölümden oluşmaktadır:

kod  
data  
stack

Kod bölümünde, fonksiyonların makine kodları vardır. Data bölümünde, statik ömürlü nesneler bulunmaktadır, global değişkenler ve stringler bu bölümde bulunurlar. Stack bölümü yerel değişkenlerin saklandığı bellek alanıdır. Stack bölümü her sistemde sınırlı bir alandır. Örneğin DOS işletim sisteminde 64K büyüklüğünde bir stack söz konusudur. Yani hiç bir zaman yerel değişkenlerin bellekte kapladığı alan 64K değerini geçemez. WINDOWS işletim sisteminde default stack sınırı 1 MB'dır. Ancak bu sınır istenildiği kadar büyütülebilir.

Daha önceki derslerimizde, bir fonksiyonun yerel bir değişkenin adresi ile geri dönmemesi gerektiğini söylemiştik. Çünkü fonksiyon sonlandığında yerel değişkenler bellekten boşaltılacağı için, fonksiyonun geri döndürdüğü adres güvenli bir adres olmayacaktır. Örneğin aşağıdaki program hatalıdır:

```
char *getname(void)
{
    char s[100];

    printf("adı ve soyadı giriniz : ");
    gets(s);
    return s;
}

int main()
{
    char *ptr;

    ptr = getname();
    puts(ptr);
    return 0;
}
```

Fonksiyon bir stringin adresiyle geri dönebilir, bu durumda bir yanlışlık söz konusu olmayacaktır. Çünkü stringler programın çalışma süresi boyunca bellektedir. Örneğin aşağıdaki programda hatasız olarak çalışır:

```
char *getday(int day)
{
    char *ptr;

    switch (day) {
        case 0: p = "Sunday"; break;
        case 1: p = "Monday"; break;
        case 2: p = "Tuesday"; break;
        case 3: p = "Wednesday"; break;
        case 4: p = "Thursday"; break;
        case 5: p = "Friday"; break;
        case 6: p = "Saturday";
    }
    return p;
}
```

## Stringlerin Birleştirilmesi

Daha önce söylendiği gibi stringler tek bir atom olarak ele alınmaktadır. Bir string aşağıdaki gibi parçalanamaz:

```
char *ptr;
...
ptr = "Necati Ergin'in C Dersi
Notlarını okuyorsunuz";           /* error */
```

Ancak string ifadeleri büyüdükçe bunu tek bir satırda yazmak zorlaşabilir. Ekrandaki bir satırlık görüntüye sığmayan satırlar kaynak kodun okunabilirliğini bozar. Uzun stringlerin parçalanmasına olanak vermek amacıyla C derleyicileri yanyana yazılan string ifadelerini birleştirirler. Örneğin:

```
ptr = "Necati Ergin'in C Dersi"
"Notlarını okuyorsunuz";
```

geçerli bir ifadedir. Bu durumda iki string birleştirilerek ve aşağıdaki biçime getirilecektir.

```
ptr = "Necati Ergin'in C Dersi Notlarını okuyorsunuz";
```

Derleyicinin iki stringi birleştirmesi için, stringlerin arasında hiçbir operatörün bulunmaması gerekmektedir. :

```
p = "Necati " "Ergin";
```

ifadesi ile

```
p = "Necati Ergin";
```

ifadesi eşdeğerdir.

Birleştirmenin yanı sıra, bir ters bölü karakteri ile sonlandırılarak sonraki satıra geçiş sağlanabilir. Örneğin:

```
ptr = "Necati Ergin'in C Dersi \
Notlarını okuyorsunuz";
```

ifadesi ile

```
ptr = "Necati Ergin'in C Dersi Notlarını okuyorsunuz";
```

ifadesi eşdeğerdir. Tersbölü işaretinden sonra string aşağıdaki satırın başından itibaren devam etmelidir. Söz konusu string aşağıdaki şekilde yazılırsa:

```
ptr = "Necati Ergin'in C Dersi \
    Notlarını okuyorsunuz";
```

satır başındaki boşluk karakterleride stringe katılır ve sonuç aşağıdaki ifadeye eşdeğer olur:

```
ptr = "Necati Ergin'in C Dersi   Notlarını okuyorsunuz";
```

Ancak ters bölü karakteri ile sonraki satırın başından devam etme standart olarak her C derleyicisinde geçerli olmayabilir. Çatışmalı bir durumla karşılaştığınızda çalıştığınız derleyicilerin referans kitaplarına başvurmalısınız.

## Stringlerde Ters Bölü Karakter Sabitlerinin Kullanılması

Stringler içerisinde ters bölü karakter sabitleri de (escape sequence) kullanılabilir. Derleyiciler stringler içerisinde bir ters bölü karakteri gördüklerinde, onu yanındaki karakter ile birlikte tek bir karakter olarak ele alırlar. Örneğin:

```
char *p;
```

```
p = "Necati\tErgin";
```

ifadesinde \t bir karakterdir. (9 numaralı ASCII karakteri olan tab karakteri)

yani

```
printf("stringdeki karakter sayısı = %d\n", strlen(p));
```

ifadesi ile ekrana

stringdeki karakter sayısı = 12

yazdırılacaktır.

String ifadelerinde doğrudan çift tırnak ya da ters bölü karakterleri kullanılamaz, çünkü bu karakterlerin özek işlevi var. Çift tırnak karakter sabitinin kendisini ifade etmek için çift tırnak karakterinden önce bir ters bölü karakteri kullanılır. Örneğin:

```
p = "\"Necati Ergin\"";
```

gibi bir string ifadesi geçerlidir. Bu durumda

```
puts(p);
```

ile ekrana

"Necati Ergin"

yazısı basılacaktır.

## Stringlerle Göstericilere İlkdeğer Verilmesi

Stringler kullanılarak **char** türden göstericilere ilkdeğer verilebilir. Örneğin:

```
char *p = "İstanbul";
```

```
char *err = "Bellek yetersiz";
```

```
char *s = "Devam etmek için bir tuşa basınız";
```

```
...
```

String ifadeleri aslında **char** türden bir adres olduğuna göre ilk değer verilen göstericilerin de **char** türden göstericiler olması gerekir. Dizilere iki tırnak içerisinde ilk değer vermeyeyle göstericilere stringlerle ilk değer verme arasındaki ayırımı dikkat etmek gerekir.

```
char *p = "Deneme";
```

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

ifadeleri tamamen farklıdır.

Göstericilere ilkdeğer verildiğinde derleyici bunu bir string ifadesi olarak ele almaktadır. Yani string 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 kullandığımız iki tırnak ifadeleri adres belirtmezler (string değildireler). Dizi

elemanlarına tek tek **char** türden sabitlerle ilk değer verme işlemi zahmetli olduğu için, programcının işini kolaylaştırmak amacı ile dile bu ilk değer verme sentaksı eklenmiştir.

Yani

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

aslında

```
char s[10] = {'D', 'e', 'n', 'e', 'm', 'e', '\\0'};
```

ile aynı anlamdadır.

### İçinde Yazı Tutan **char** Türden Dizilerle Bir Stringi Gösteren **char** Türden Göstericilerin Karşılaştırılması

Şimdiye kadar almış olduğumuz bilgileri değerlendirdiğimizde, C dilinde bir yazıyı saklamak iki ayrı şekilde sağlanabilir:

1. Yazıyı **char** türden bir dizi içinde saklamak:

```
...
char s1[100] = "Necati Ergin";
char s2[100];
char s3[100];

gets(s2);           / * Diziye yazının karakterleri klavyeden alınıyor */
strcpy(s2, s3);      /* s3 dizisinin elemanları s2 dizisi içine kopyalanıyor */
```

2. Yazıyı bir string olarak saklayarak **char** türden bir göstericinin bu stringi göstermesini sağlamak .

```
char *str = "Necati Ergin";
```

İki yöntem birbirinin tamamen alternatifi değildir ve aşağıdaki noktalara dikkat edilmesi gerekmektedir:

stringler statik nesneler oldukları için programın sonlanmasına kadar bellekte yer kaplarlar. Yani bir göstericinin bir stringi gösterirken daha sonra başka bir stringi gösterir duruma getirilmesi , daha önceki stringin bellekten silinmesi anlamına gelmeyecektir.

```
char *s = "Bu string programın sonuna kadar bellekte kalacak.";
```

```
s = "artık yukarıdaki string ile bir bağlantımız kalmayacak...";
```

Yazının **char** türden bir dizi içinde tutulması durumunda bu yazıyı değiştirmemiz mümkündür. Dizi elemanlarına yeniden atamalar yaparak yazıyı istediğimiz gibi değiştirebiliriz ama stringlerin değiştirilmesi taşınabilir bir özellik değildir. Stringler içinde bulunan yazılar değiştirilmemelidir. Aşağıdaki kod parçaları taşınabilir değildir:

```
char *s = "Metin";
char *str = "Ankara";
```

```
s[1] = 'Ç';
strcpy(str, "Bolu");
```

Derleyicilerin özdeş stringleri aynı adreste saklamaları konusunda bir garanti yok. Özdeş stringler aynı adreste tek kopya olarak ya da farklı adreslerde tutuluyor olabilir. Daha önce söylediğimiz gibi derleyiciler bu konuda birbirinden farklı davranabilirler. Yani yukarıdaki örneklerde biz "Metin" ya da "Ankara" stringini değiştirdiğimizde, bu stringleri program içinde

baska yerlerde de kullanmışsak, bunların hepsinin değiştirilmiş olması gibi bir tehlike söz konusudur.

```
char *s = "dosya açılmıyor"; /* birinci string */  
char a[] = "dosya açılmıyor";
```

```
...  
printf("%s", s);  
...  
printf("dosya açılmıyor"); /* ikinci string */  
...  
strcpy(s, "Dosya Açıldı");  
...
```

"dosya açılmıyor" stringinin derleyici tarafından aynı adreste tek kopya olarak saklandığını farsayalım. s adresinde bulunan stringe yeni bir yazı kopyalanınca hem birinci string hemde ikinci string değişmiş olur.

## 21 . BÖLÜM : GÖSTERİCİ DİZİLERİ

Göstericiler de birer nesne olduğuna göre gösterici dizileri de tanımlanabilmelidir.

Elemanları göstericilerden oluşan dizilere “gösterici dizileri” (pointer arrays) denir. Bildirimleri aşağıdaki şekilde yapılır :

<tür> \*<dizi ismi [dizi\_uzunluğu];>

Örneğin :

```
char *s[10];
int *sample[50];
float *kilo[10];
```

Gösterici dizilerinin bildirimleri normal dizi bildirimlerinden farklı olarak \* operatörü ile yapılmaktadır. Örneğin:

```
char str[100];
```

bildiriminde, str 100 elemanlı char türden bir diziye

```
char *str[100];
```

bildiriminde ise, str 100 elemanlı char türden bir gösterici dizisidir.

Derleyici bir gösterici dizisi tanımlaması ile karşılaşınca diğer dizilerde yaptığı gibi, bellekte belirtilen sayıda gösteriyi tutabilecek kadar ardışıl yer tahsis eder. Örneğin:

```
char *p[10];
```

bildirimi ile derleyici, s dizisinin 10 elemanlı ve her elemanın char türden bir gösterici olduğunu anlar. Kullanılan sistemde göstericilerin uzunluğunun 2 byte olduğunu kabul edersek, 20 byte sürekli bir bölge tahsis edecektir. Bu durumda;

```
p[0], p[1], p[2], ...p[9]
```

dizi elemanlarının her biri char türden bir göstericidir. Elemanlarından her biri diğerinden bağımsız olarak kullanılabilir ve her bir eleman bir nesnedir. Aşağıdaki örneği inceleyiniz :

```
main()
{
    char *s[10];
    int k;

    for (k = 0; k < 10; ++k)
        s[k] = (char *) malloc(30);
        if (s[k] == NULL) {
            printf("cannot allocate memory!..\n");
            exit(EXIT_FAILURE);
        }
        printf("ismi giriniz : ");
        gets(s[k]);
}
```

Bu örnekte her elemanı char türden bir gösterici olan, 10 elemanlı bir gösterici dizisi açılmış ve dizinin her elemanı için, döngü içerisinde malloc fonksiyonu ile 30 byte büyüklüğünde bir bellek bloğu tahsis edilmiştir. Tahsis edilen blokların başlangıç adresleri gösterici dizisinin elemanlarına yerleştirildikten sonra, gets fonksiyonu ile klavyeden girilen karakterlerin tahsis edilen bloklara aktarılmıştır. s isimli dizi yerel bir dizi olduğuna göre, dizi elemanları içerisinde rasgele (garbage values) değerler olduğunu belirtelim. Dolayısıyla malloc fonksiyonu ile yer ayırmadan yapılan veri aktarımları gösterici hatalarına yol açacaktır. Dinamik olarak tahsis edilen alanlar, gerektiği zaman yine bir döngü yardımıyla serbest bırakılabilir :

```
...
for (k = 0; k < 10; ++k)
    free(s[k]);
...
```

### **gösterici dizilerine ilk değer verilmesi**

Normal dizilere ilkdeğer nasıl veriliyorsa gösterici dizilerine de ilk değer aynı biçimde verilir. meğin:

```
int *p[] = {
    (int *) 0x1FC0,
    (int *) 0x1FC2,
    (int *) 0x1FC4,
    (int *) 0x1FC6,
    (int *) 0x1FC8
}
```

Kuşkusuz ki, gösterici dizisine atanan adreslerin bir amaç doğrultusunda kullanılabilmesi için güvenli bölgeleri göstermesi gerekir. Bu nedenle uygulamada karakter gösterici dizisi dışındaki gösterici dizilerine ilk değer verilmesi durumuna pek raslanmaz. Gösterici dizilerine ilk değer verme işlemi genellikle karakter gösterici dizilerine string ifadeleriyle ilk değer verilmesi biçiminde karşımıza çıkmaktadır.

### **char türden gösterici dizileri**

Uygulamalarda en sık görülen gösterici dizileri char türden olan gösterici dizilerdir.

Daha önce stringler konusunda da gördüğümüz gibi, stringler C derleyicisi tarafından char türden adresler olarak ele alındığına göre, char türden bir gösterici dizisine stringler aracılığıyla ilk değer verilebilir:

```
char *aylar[] = {"Ocak",
    "Şubat",
    "Mart",
    "Nisan",
    "Mayıs",
    "Haziran",
    "Temmuz",
    "Ağustos",
    "Eylül",
    "Ekim",
    "Kasım",
    "Aralık"
};
```

aylar dizisinin her bir elemanı char türden bir göstericidir. Ve bu gösterici dizisine stringler ile ilk değer verilmiştir. Başka bir deyişle char türden bir gösterici dizisi olan aylar dizisinin her bir elemanı bir yazıyı göstermektedir.

### **char türden gösterici dizilerinin kullanılma nedenleri**

1. Program içinde sık kullanılacak yazıların her defasında yazılması yerine bir gösterici dizisinin elemanlarında saklanması. Aşağıdaki örnekte err dizisinin her bir elemanında hata mesajlarına ilişkin yazılar saklanmıştır.

```
char *err[] = {"Bellek Yetersiz!", "Hatalı şifre", "Dosya bulunamadı", "Belirtilen dosya zaten var", "sürücü hazır değil", "Okunacak dosya açılmıyor", "yazılacak dosya açılmıyor!..", "Belirlenemeyen hata!"};
```

Artık programın herhangi bir yerinde yazılardan biri yazdırılmak istenirse, yazdırılacak olan yalnızca gösterici dizisinin herhangi bir elemanında başlangıç adresi tutulan yazıdır :

```
...
```

```
if (fp == NULL) {
    printf("%s\n", err[5]);
    return 5;
}
....
```

## 2. Bazı yazıların bu şekilde bir algoritmaya bağlı olarak kullanılması :

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

```
int dayofweek(int day, int month, int year);
```

```
char *aylar[] = {"Ocak",
                "Şubat",
                "Mart",
                "Nisan",
                "Mayıs",
                "Haziran",
                "Temmuz",
                "Ağustos",
                "Eylül",
                "Ekim",
                "Kasım",
                "Aralık"
};
```

```
char *gunler[] = {"Pazar",
                  "Pazartesi",
                  "Salı",
                  "Çarşamba",
                  "Perşembe",
                  "Cuma",
                  "Cumartesi"
};
```

```
void display_date(int day, int mon, int year)
{
    printf("%02d ", day);
    puts(aylar[mon - 1]);
    printf(" %d ", year);
}
```

Bir yazının belirli bir yazıyla aynı olup olmadığını, strcmp fonksiyonuyla saptayabiliriz. Peki bir yazının bir grup yazı içinde var olup olmadığını nasıl tespit edebiliriz? Aşağıda tanımlanan **getmonth** fonksiyonu kendisine başlangıç adresi gönderilen bir yazının ingilizce ay isimlerinden biri olup olmadığını test etmekte, eğer böyle ise bu ayın kaçınıcı ay olduğu bilgisiyle geri dönmektedir. (1 - 12) Fonksiyon, kendisine gönderilen argumanda başlangıç adresi tutulan yazı bir ay ismine karşılık gelmiyor ise 0 değeriyle geri dönmektedir.

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

```
int getmonth(char *str)
{
    char *months[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
    int k;

    for (k = 0; k < 12; ++k)
        if (!strcmp(months[k], str))
            return k + 1;
    return 0;
}
```



```
}  
  
main()  
{  
    char s[20];  
    int n = 20;  
    int result;  
  
    while (n-- > 0) {  
        printf("bir ay ismi giriniz .");  
        gets(s);  
        result = getmonth(s);  
        if (result)  
            printf("%s yılın %d. ayıdır\n", s, result);  
        else  
            printf("%s geçerli bir ay ismi değildir\n");  
    }  
    return 0;  
}
```

strcmp fonksiyonunun iki yazının karşılaştırmasını büyük harf küçük harf duyarlılığı olmadan yapması dışında, strcmp fonksiyonundan başka bir farkı bulunmadığını hatırlayalım.

Gösterici dizileri de tıpkı diğer diziler gibi yerel ya da global olabilecektir. Eğer dizinin global olması durumunda dizi elemanlarının hepsinin içinde 0 değerleri olurken, yerel bir gösterici dizisinin içinde rasgele değerler olacaktır. Dizinin her bir elemanı içinde bir adres bilgisi olduğuna göre, atama yapılmamış global gösterici dizilerinin her bir elemanı içinde 0 adresi (NULL adresini) bulunurken, atama yapılmamış yerel gösterici dizilerinin elemanları içinde rasgele değerler bulunmaktadır.

Bir gösterici hatasına neden olmamak için önce gösterici dizisi elemanlarına güvenli adresler yerleştirmek gerekmektedir. Stringler statik nesneler gibi ele alındıkları için bir gösterici dizisi elemanlarına stringlerle değer vermek bir gösterici hatasına enden olmayacaktır. Zira stringler, daha önce de belirtildiği gibi önce derleyici tarafından bellekte güvenli bir bölgeye yerleştirilirler, daha sonra ise yerleştirildikleri bloğun başlangıç adresi olarak ele alınırlar.

## **22 . BÖLÜM : GÖSTERİCİYİ GÖSTEREN GÖSTERİCİLER**

**konu eklenecek.....**

## 23 . BÖLÜM : DİNAMİK BELLEK YÖNETİMİ

C dilinde bir dizi tanımlandığı zaman, bu dizi için derleyici tarafından bellekte yer ayrılır. Örneğin:

```
int a[100];
```

Derleme sırasında yukarıdaki gibi bir dizi tanımlaması ile karşılaşan derleyici bellekte (eğer kullanılan sistemde **int** türü uzunluğunun 2 byte olduğu varsayılırsa) toplam 200 byte yer ayıracaktır. Programın çalışması sırasında bir dizinin uzunluğunu değiştirmek mümkün değildir. Diziler konusunda açıklandığı gibi, dizi tanımlama ifadelerinde dizi boyutunu gösteren ifade (köşeli parantezin içerisindeki ifade) sabit ifadesi olmalıdır, değişken içeremez. Çünkü derleyicinin bellekte yer ayırması için, dizi boyutunu bilmesi gerekir. Oysa pratikte birçok uygulamada açılması gereken dizinin boyutu programın çalışması sırasında (runtime) belirlenmektedir.

Birtakım sayıların kullanılmak üzere klavyeden girildiğini düşünelim. Kullanıcının kaç tane sayı gireceğinin belli olmadığını düşünelim. Kullanıcının girdiği sayıları tutmak için açılan bir dizinin boyutu ne olmalıdır? Ya da bir dizindeki dosyaları sıraya göre dizmek için dosya bilgilerini geçici olarak bir dizide saklayacağımızı düşünelim. Dizinin uzunluğu ne olmalıdır? Bu başlangıçta belli değildir. Çünkü dizin içinde kaç dosya olduğu belli değildir. Bu tür örnekleri çoğaltabiliriz. Bu tip durumlara özellikle veri tabanı uygulamalarında sıklıkla raslarız. 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 gözönünde bulundurarak açmak zorundadır. Bu da belleğin verimsiz bir şekilde kullanılması anlamına gelir. Üstelik açılan diziler yerel ise ilgili blok sonlanana kadar, global ise programın çalışmasının sonuna kadar bellekte tutulacaktır. Oysa, dizi ile ilgili işlem biter bitmez, dizi için ayrılan bellek bölgesinin boşaltılması verimli bellek kullanımı için gereklidir.

Programın çalışma zamanı sırasında belli büyüklükte ardışıl (contiguous) bir bellek bölgesinin programcı tarafından tahsis edilemesine ve istenildiği zaman serbest bırakılmasına olanak sağlayan yöntemlere dinamik bellek yönetimi denir. C dilinde dinamik bellek yönetimi dinamik bellek fonksiyonlarıyla yapılmaktadır. Dinamik bellek yönetiminde kullanılan standart C fonksiyonları hakkında aşağıda detaylı bilgi verilmektedir. Şüphesiz bu fonksiyonların dışında, ticari derleyici paketlerinin kütüphanesinde, standart olmayan dinamik bellek fonksiyonları da bulunabilir. Ancak yazılan kaynak kodun taşınabilirliği açısından standart C fonksiyonları tercih edilmelidir.

Şimdi dinamik bellek yönetiminde kullanılan standart C fonksiyonlarını tek tek detaylı olarak inceleyelim:

### malloc fonksiyonu

malloc fonksiyonu programın çalışma zamanı sırasında bellekten dinamik tahsisat yapmak için kullanılır. Fonksiyonun prototipi aşağıdaki gibidir :

```
void *malloc(size_t nbyte);
```

size\_t türünün derleyiciye yazarların seçimine bağlı olarak **unsigned int** ya da **unsigned long** türlerinden birinin yeni tür ismi olarak tanımlanması gerektiğini, ve sistemlerin hemen hemen hepsinde size\_t türünün **unsigned int** türü olduğunu hatırlayalım.

Fonksiyona gönderilecek arguman tahsis edilmek istenen bloğun byte olarak uzunluğudur. Tahsis edilen alanın sürekli (contiguous) olması garanti altına alınmıştır. malloc fonksiyonunun geri dönüş değeri tahsis 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 tahsis edilen blok gösterici yardımıyla bir dizi gibi kullanılabilir. malloc fonksiyonunun istenilen bloğu tahsis etmesi garanti altında değildir. Birçok

nedeninden dolayı malloc fonksiyonu başarısız olabilir. Bellekte tahsis edilmek istenen alan kadar boş bellek bulunmaması sık görülen başarısızlık nedenidir. malloc fonksiyonu başarısız olduğunda NULL adresine geri döner. malloc fonksiyonu bellekten yer ayırdıktan sonra işleminin başarılı olup olmadığı mutlaka kontrol edilmelidir. malloc fonksiyonu ile bellekte bir blok tahsis edilmesine ilişkin aşağıda bir örnek verilmiştir.

```
char *ptr;
```

```
ptr = (char *) malloc(30);    /* bellekte 30 byte'lık bir yer tahsis edilmek isteniyor */
```

```
if (ptr == NULL) {
    printf("cannot allocate memory\n");
    exit(EXIT_FAILURE);
}
/* başarısızlık durumunda program sonlandırılıyor */
printf("please enter the name : ");
gets(ptr);    /* tahsis edilen alana klavyeden giriş yapılıyor */
...
```

Şüphesiz, malloc fonksiyonu başarısız olduğunda programı sonlandırmak zorunlu değildir.

Atama ve kontrol bir defada da yapılabilir.

```
if ((ptr = (char *) malloc(30)) == NULL) {
    printf("cannot allocate memory\n");
    exit(EXIT_FAILURE); /* başarısızlık durumunda program sonlandırılıyor */
}
```

malloc fonksiyonu ile yapılan dinamik tahsisatın başarılı olup olmadığı mutlaka kontrol edilmelidir. Fonksiyonun başarısız olması durumunda, geri dönüş değerinin aktarıldığı gösterici aracılığı ile belleğe birşey yazılmaya çalışılırsa, atama NULL adrese yapılmış olur. (NULL pointer assignment). Programın yanlış çalışması kaçınılmazdır.

Programcılar çoğu kez, küçük miktarlarda ve çok sayıda bloğun tahsis edilmesi durumunda, kontrolü gereksiz bulma eğilimindedir. Oysa kontrolden vazgeçmek yerine daha kolaylaştırıcı yöntemler denenmelidir. Örneğin p1, p2, p3, p4, p5 gibi 5 ayrı gösterici için 10'ar byte alan tahsis edilmek istensin. Bu durumda kontrol mantıksal operatörler ile tek hamlede yapılabilir.

```
char *p1, *p2, *p3, *p4, *p5;
```

```
...
p1 = (char *)malloc(10);
p2 = (char *)malloc(10);
p3 = (char *)malloc(10);
p4 = (char *)malloc(10);
p5 = (char *)malloc(10);

if (!(p1 && p2 && p3 && p4 && p5)) {
    printf("cannot allocate memory!..\n");
    exit(EXIT_FAILURE);
}
```

malloc fonksiyonu blokların herhangi birinde başarısız olursa bu durum **if** deyimi içinde tespit edilmektedir.

malloc fonksiyonunun geri dönüş değeri **void** türden bir adres olduğu için, bu adres sorunsuzca her türden göstericiye atanabilir. Ancak okunabilirlik açısından malloc fonksiyonunun geri dönüş değeri olan adresin, tür dönüştürme operatörü yardımıyla, kullanılacak göstericinin türüne dönüştürülmesi tavsiye edilir. Böylece malloc fonksiyonu ile

tahsis edilen bloğun ne türden bir diziymiş gibi kullanılacağı bilgisi de açıkça okuyucuya verilmiş olabilir.

C dilinin standartlaştırılmasından önceki dönemde **void** türden göstericiler olmadığı için, malloc fonksiyonunun geri dönüş değeri **char** türden bir adrestir ve bu durumda, geri dönüş değeri olan adresin, **char** türü dışında bir göstericiye atanması durumunda tür dönüşümü bir zorunluluktur.

**int** türden bir dizi için dinamik olarak tahsisatı yaptığımız düşünelim. Örneğin bir kullanıcının klavyeden **int** türden sayılar gireceğini düşünelim. Kullanıcıdan, kaç tane sayı gireceği bilgisi alınsın ve istenilen miktardaki sayıyı saklayabilecek bir alan dinamik olarak tahsis edilsin.

```
int main()
{
    int number;
    int i;
    int *ptr;

    printf("kaç tane sayı girmek istiyorsunuz? ");
    scanf("%d", &number);
    ptr = (int *) malloc(number * 2);
    ...
    return 0;
}
```

Yukarıdaki kod parçasında **int** türü uzunluğunun 2 byte olduğu varsayılmıştır. Kaynak kod **int** türü uzunluğunun 4 byte olduğu bir sistemde (örneğin UNIX) derlenirse problemler ortaya çıkacaktır. Bu taşınabilirlik problemi **sizeof** operatörünün kullanılmasıyla çözülür.

```
ptr = (int *) malloc(number * sizeof(int));
```

Artık çalışılan sistem ne olursa olsun number sayıda tamsayı eleman için alan tahsis edilmiş olacaktır.

malloc fonksiyonu birden fazla çağırılarak birden fazla alan tahsis edilebilir. malloc fonksiyonun farklı çağırımlarıyla tahsis edilen blokların bellekte ardışıl olması garanti altına alınmış değildir. Bu yüzden fonksiyonun geri dönüş değeri olan adres mutlaka bir göstericide saklanmalıdır. Aşağıdaki kod parçası ardışık olarak çağırılan malloc fonksiyonlarının ardışık bellek blokları tahsis edeceğini varsaydığı için yanlıştır.

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

int main()
{
    int * ptr;
    int i;

    ptr = malloc(sizeof(int) * 10);
    malloc(sizeof(int) * 10); /* tahsis edilen alan daha önce tahsis edilen alanın hemen
                                altında olmak zorunda değildir */

    for (i = 0; i < 20; ++i)
        ptr[i] = 5;
    return 0;
}
```

malloc fonksiyonu ile tahsis edilen bloğun içinde rasgele değerler vardır. Aşağıdaki kod parçası ile bunu test edebiliriz.

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

int main()
{
    int *ptr;
    int i;

    ptr = (int *) malloc(10 * sizeof(int));
    if (ptr == NULL) {
        printf("yetersiz bellek!..\n");
        exit(1);
    }
    for (i = 0; i < 10; ++i)
        printf("ptr[%d] = %d\n", i, ptr[i]);
    return 0;
}
```

Dinamik bellek fonksiyonları kullanılarak tahsis edilebilecek bellek bölgesi heap olarak isimlendirilmiştir. heap alanı donanımsal bir kavram olmayıp sistemden sisteme değişebilmektedir. (C++ dilinde bu alan free store olarak isimlendirilmektedir.)

malloc fonksiyonunun parametre değişkeni **unsigned int** (size\_t) türünden olduğuna göre, DOS altında en fazla 65535 byte (64K) ardışıl tahsisat yapılabilir. Oysa UNIX, WINDOWS ve diğer 32 bitlik sistemlerde **unsigned int** türü 4 byte uzunluğund olduğuna göre, bu sistemlerde teorik olarak, malloc fonksiyonu ile 4294967295 byte (4 MB) uzunluğunda ardışıl bir tahsisat yapılabilir. Tabi bu durum, tahsisatın yapılabileceği anlamına gelmez.

## calloc fonksiyonu

calloc fonksiyonu malloc fonksiyonuna çok benzer. Prototipi aşağıdaki gibidir.

```
void *calloc(size_t nblock, size_t block_size);
```

calloc fonksiyonu kedisine gönderilen birinci arguman ile ikinci arguman çarpımı kadar ardışıl byte'ı heap alanından tahsis etmek için kullanılır.

Elemanları **int** türden olan 100 elemanlı bir dizi için dinamik bellek tahsisatı calloc fonksiyonu ile aşağıdaki şekilde yapılabilir:

```
int *ptr;
...
ptr = (int*) calloc(100, sizeof(int));
if (ptr == NULL) {
    printf("cannot allocate memory\n");
    exit(EXIT_FAILURE);
}
...
```

Şüphesiz yukarıdaki kod parçasında calloc fonksiyonu şu şekilde de çağırılabilirdi:

```
ptr = (int*) calloc( sizeof(int), 100);
```

calloc fonksiyonunun malloc fonksiyonundan başka bir farkı da tahsis ettiği bellek bloğunu sıfırlanmasıdır. calloc fonksiyonu tarafından tahsis edilen bloğunun tüm byte'larında sıfır değerleri vardır. malloc fonksiyonu calloc fonksiyonuna göre çok daha sık kullanılır. Tahsis edilen blok sıfırlanacaksa malloc fonksiyonu yerine calloc fonksiyonu tercih edilmelidir.

Örneğin 50 elemanlı **int** türden bir dizi için bellekten bir bloğu dinamik olarak tahsis ettikten sonra sıfırlamak istediğimizi düşünelim. Bu işlemi malloc fonksiyonu ile yaparsak, diziyi ayrıca döngü içerisinde sıfırlamamız gerekecektir :

```
int *ptr;
int i;
...
ptr = (int *) malloc(sizeof(int) * 100);
if (ptr == NULL) {
    printf("cannot allocate memory!..\n");
    exit(EXIT_FAILURE);
}
for (i = 0; i < 100; ++i)
    ptr[i] = 0;
...
```

Oysa calloc fonksiyonu zaten sıfırlama işlemini kendi içerisinde yapmaktadır.

calloc fonksiyonu, bellek tahsisatını yapabilmek için, kendi içinde malloc fonksiyonunu çağırır.

### realloc fonksiyonu

realloc fonksiyonu daha önce malloc ya da calloc fonksiyonlarıyla tahsis edilmiş bellek bloğunu büyütmek ya da küçültmek amacıyla kullanılır. Prototipi diğer dinamik bellek fonksiyonlarında olduğu gibi stdlib.h başlık dosyası içindedir.

```
void *realloc(void *block, unsigned newsize);
```

realloc fonksiyonuna gönderilen birinci arguman, daha önce tahsis edilen bellek bloğunun başlangıç adresi, ikinci arguman ise bloğun toplam yeni uzunluğudur.

realloc fonksiyonu daha önce tahsis edilmiş bloğun hemen altında sürekliliği bozmayacak şekilde tahsisat yapmaya çalışır. Eğer daha önce tahsis edilmiş bloğun aşağısında istenilen uzunlukta sürekli yer bulamazsa realloc, bu sefer bloğun tamamı için bellekte başka boş yer araştırır. Eğer istenilen toplam uzunlukta ardışıl (contiguous) bir blok bulunursa burayı tahsis eder, eski bellek bloğundaki değerleri yeni yere taşır ve eski bellek bloğunu işletim sistemine iade eder. İstenen uzunlukta sürekli bir alan bulunamazsa NULL adresiyle geri döner.

Aşağıdaki programda yapılan dinamik bellek tahsisatı işlemlerini inceleyelim:

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

int main()
{
    int *ptr;

    ptr = (int *) malloc(sizeof (int) * 5);    /* 1 */
    if (ptr == NULL) {
        printf("bellek yetersiz!..\n");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < 5; ++i)                /* 2 */
        ptr[i] = i;
    ptr = realloc(ptr, sizeof(int) * 10); /* 3 */
    if (ptr == NULL) {
        printf("bellek yetersiz!..\n");
        exit(EXIT_FAILURE);
    }
    return 0;
```

}

1. malloc fonksiyonu ile 5 **int** lik yer tahsis ediliyor. Eğer tahsisat işlemi başarılı değilse exit fonksiyonuyla program sonlandırılıyor.

1A00	?	<--ptr
1A01	?	
1A02	?	
1A03	?	
1A04	?	
1A05	?	
1A06	?	
1A07	?	
1A08	?	
1A09	?	

2. Bir döngü yardımıyla tahsis edilen alana atamalar yapılıyor.

1A00	0	<--ptr
1A01		
1A02	1	
1A03		
1A04	2	
1A05		
1A06	3	
1A07		
1A08	4	
1A09		

3. realloc fonksiyonuyla tahsis edilen alan 10 int sayılık örneğin DOS altında çalışıyorsak 20 byte'a çıkartılıyor.

Eğer realloc fonksiyonu başarısız olursa program sonlandırılıyor. realloc fonksiyonu başarılı olmuşsa iki ihtimal söz konusudur.

1. realloc fonksiyonu daha önce tahsis edilen alanın altında boşalan bularak ilave tahsisatı buradan yapmıştır :

ptr----- >	1A00	0
	1A01	
	1A02	1
	1A03	
	1A04	2
	1A05	
	1A06	3
	1A07	
	1A08	4
	1A09	
	1A0A	?
	1A0B	
	1A0C	?
	1A0D	
	1A0E	?
	1A0F	
	1A10	?
	1A11	
	1A12	?
	1A13	

2. realloc fonksiyonu daha önce tahsis edilen bloğun altında boş yer bulamamış ve heap alanında toplam 20 bytelik başka bir boş yer bulmuş olabilir. Bu durumda realloc fonksiyonu daha önce tahsis edilmiş alandaki değerleri de bu alana kopyalamıştır :



ptr----->

1F00	0
1F01	
1F02	1
1F03	
1F04	2
1F05	
1F06	3
1F07	
1F08	4
1F09	
1F0A	?
1F0B	
1F0C	?
1F0D	
1F0E	?
1F0F	
1F10	?
1F11	
1F12	?
1F13	

realloc fonksiyonunun bu davranışından dolayı mutlaka geri dönüş değeri bir göstericiye atanmalıdır. Aşağıda sık yapılan çok tipik bir hata görülmektedir:

```
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 fonksiyonunun başarılı olması tahsis edilen bloğun altında ilave tahsisat yapıldığı anlamına gelmez. (realloc fonksiyonu daha önce tahsis edilmiş alanı başka bir yere taşıyarak yeni bloğun başlangıç adresiyle geri dönmüş olabilir. Bu durumda eski blok serbest bırakılır ve artık bu adresin bir güvenilirliği yoktur.

Yukarıdaki örnekte realloc fonksiyonunun tahsisatı daha önce tahsis edilen bloğu büyüterek yaptığı varsayılmıştır. Eğer realloc fonksiyonu bir taşıma yapmışsa, yukarıdaki kodda bir gösterici hatası yapılmış olacaktır, çünkü artık p göstericisinin gösterdiği adres güvenilir bir adres değildir.

Bu yüzden uygulamalarda genellikle realloc fonksiyonunun geri dönüş değeri, realloc fonksiyonuna gönderilen 1. adresi içinde tutan göstericiye atanır.

```
ptr = realloc(ptr, 100);
```

gibi. Ancak bu bir zorunluluk değildir. Eğer realloc fonksiyonu ile yapılan tahsisatın başarılı olmaması durumunda program sonlandırılmayacaksa (örneğin artık daha fazla tahsisat yapılamaması durumunda) daha önce dinamik olarak tahsis edilen bölgedeki değerler bir dosyaya yazılacak ise, artık realloc fonksiyonunun başarısız olması durumunda, ptr göstericisine NULL adresi atanacağı için ptr göstericisinin daha önce tahsis edilen blok ile ilişkisi kesilmiş olacaktır.

Böyle bir durumda geçici bir gösterici kullanmak uygun olacaktır:

```
temp = realloc(ptr, 100);
if (temp == NULL)
    printf("cannot allocate memory!..\n");
```

Bu durumda ptr göstericisi halen daha önce tahsis edilen bloğun başlangıç adresini göstermektedir.

C standartları realloc fonksiyonu ile ilgili olarak aşağıdaki kuralları getirmiştir:

Bir bellek bloğunu genişletmesi durumunda, realloc fonksiyonu bloğa ilave edilen kısma herhangi bir şekilde değer vermez. Yani eski bloğa eklenen yeni blok içinde rasgele değerler (garbage values) bulunacaktır.

realloc fonksiyonu eğer daha önce tahsis edilmiş bellek bloğunu büyütemez ise NULL adresi ile geri dönecektir ancak tahsis edilmiş olan (büyütülemeyen) bellek bloğundaki değerler korunacaktır.

Eğer realloc fonksiyonuna gönderilen birinci arguman NULL adresi olursa, realloc fonksiyonu tamamen malloc fonksiyonu gibi davranır. Yani:

```
realloc(NULL, 100);
```

ile

```
malloc(100);
```

tamamen aynı anlamdadır.

(Buna neden gerek görülmüştür? Yani neden malloc(100) gibi bir çağırımı yapmak yerine realloc(NULL, 100) şeklinde bir çağırımı tercih edelim?

Aşağıdaki programı inceleyelim :

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <stdlib.h>
```

```
void display_array(int *ptr, int size);
int find_max(int *ptr, int size);
int find_min(int *ptr, int size);
double find_ave(int *ptr, int size);
double find_stddev(int *ptr, int size);
```

```
int main()
{
    int *ptr = NULL;
    int ch, grade;
    int counter = 0;

    clrscr();
    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("girdiğiniz notlar aşağıda listelenmektedir :\n");
display_array(ptr, counter);
printf("\nen büyük not : %d\n", find_max(ptr, counter));
printf("en küçük not : %d\n", find_min(ptr, counter));
printf("notların ortalaması : %lf\n", find_ave(ptr, counter));
printf("notların standart sapması . %lf\n", find_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 fonksiyonu malloc gibi davranacak ve int türden 1 adet nesnelik yer tahsis edecektir. Ancak döngünün daha sonraki turlarında realloc fonksiyonuna gönderilen adres NULL adresi olmayacağından, daha önce tahsis edilen blok döngü içinde sürekli olarak büyütülmüş olacaktır.

Tahsis edilen bloğun serbest bırakılması

malloc ya da diğer dinamik bellek fonksiyonları "heap" diye isimlendirilen bir bellek bölgesinden tahsisat yaparlar. Ancak heap alanı da sınırlıdır ve sürekli bu fonksiyonların çağırılması durumunda, belirli bir noktadan sonra fonksiyonlar başarısız olarak NULL adresine geri dönecektir. heap alanının büyüklüğünü aşağıdaki kod ile test edebiliriz :

```

#include <stdio.h>
#include <stdlib.h>

#define BLOCK_SIZE    512

int main()
{
    long total = 0;
    char *ptr;

    for (;;) {
        ptr = (char *) malloc(BLOCK_SIZE);
        if (ptr == NULL)
            break;
        total += BLOCK_SIZE;
    }
    printf("toplam heap alanı = %ld byte\n", total);
    return 0;
}

```

```
}
```

Dinamik bellek fonksiyonlarıyla tahsis edilen bir blok free fonksiyonu kullanılarak sisteme iade edilebilir. free fonksiyonunun prototipi de diğer dinamik bellek fonksiyonlarınkiler gibi stdlib.h başlık dosyası içindedir:

```
void free(void *ptr);
```

free fonksiyonuna gönderilecek olan arguman, daha önce malloc, calloc ya da realloc fonksiyonlarıyla tahsis edilmiş olan bellek bloğunun başlangıç adresidir. Bu blok heap alanına iade edilmiş olur. Böylece malloc, calloc ya da realloc fonksiyonunun bundan sonraki bir çağırımında iade edilen blok, tekrar tahsis edilme potansiyelindedir.

free fonksiyonuna arguman olarak daha önce tahsis edilen bir bellek bloğunun başlangıç adresi yerine başka bir adres gönderilmesi şüpheli kod oluşturur (undefined behaviour) yapılmamalıdır.

```
char *p1;
char s[100];
```

```
ptr = (char *)malloc(20);
```

```
....
```

```
free(ptr) /* Legal */
```

```
free(p1) /* Bu adreste dinamik bellek fonksiyonları ile tahsis edilmiş bir alan yok. Hata. */
```

```
free(s) /* Hata s dizisi için tahsisat dinamik bellek fonksiyonları ile yapılmamıştır. */
```

free fonksiyonu ile daha önce tahsis edilen blok sisteme iade edilir, ama bu bloğun başlangıç adresini tutan gösterici herhangi bir şekilde değiştirilmez. Bloğun başlangıç adresini tutan, ve free fonksiyonuna arguman olarak gönderilen gösterici, free fonksiyonun çağırılmasından sonra artık güvenli olmayan bir adresi göstermektedir.

```
char *ptr = malloc(100);
```

```
...
```

```
free(ptr);
```

```
...
```

```
strcpy(ptr, "Necarti Ergin"); /* yanlış, bir gösterici hatası!! */
```

Eğer realloc fonksiyonuna gönderilen ikinci arguman 0 olursa, realloc fonksiyonu tamamen free fonksiyonu gibi davranır.

```
realloc(ptr, 0);
```

ile

```
free(ptr);
```

tamamen aynı anlamdadır.

(Buna neden gerek görülmüştür? Yani neden free(ptr) gibi bir çağırım yapmak yerine realloc(ptr, 0) şeklinde bir çağırımı tercih edelim?

## **Dinamik Olarak Tahsis Edilen Bir Alanın Başlangıç Adresine Geri Dönen Fonksiyonlar**

Dinamik olarak tahsis edilen bir blok, free fonksiyonuyla serbest bırakılarak sisteme iade edilene kadar güvenli olarak kullanılabileceğine göre , bir fonksiyon böyle bir bloğun başlangıç adresine geri dönebilir. Aşağıdaki fonksiyon tasarımını inceleyelim :

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

```
#include <stdlib.h>
```

```
char *getname(void)
{
    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);
    }
    strcpy(ptr, s);
    return ptr;
}
```

Yukarıdaki fonksiyonda kullanıcıdan isim önce kullanıcıdan, yerel bir diziye alınıyor. Daha sonra strlen fonksiyonu kullanılarak ismin uzunluğu tespit ediliyor ve malloc fonksiyonu ile bu ismi ve sonuna gelecek NULL karakteri içine alacak büyüklükte bir alan dinamik olarak tahsis ediliyor. Daha sonra da strcpy fonksiyonu kullanılarak, isim dinamik olarak tahsis edilen alana kopyalanıyor ve bu bloğun başlangıç adresine geri dönülüyor.

Dinamik tahsisat fonksiyonun çağırılması neticesinde yapılacak ve tahsis edilen bloğun serbest bırakılması fonksiyonu çağırmanın sorumluluğunda olacaktır :

```
int main()
{
    char *p;
    ...
    p = getname();
    ...
    free(p)
    ...
    return 0;
}
```

Bir fonksiyon içinde dinamik olarak bir bellek bloğu tahsis etmek ve daha sonra bu bloğun başlangıç adresine geri dönmek C dilinde çok kullanılan bir tekniktir:

Aşağıda tasarımı verilen strcon fonksiyonu birinci parametresinde başlangıç adresi tutulan yazının sonuna ikinci parametresinde başlangıç adresi tutulan yazıyı kopyalayacak fakat her iki adresteki yazıları da bozmadan, birleştirilmiş yazının başlangıç adresi olan bir adrese geri dönecektir. Bu dinamik bellek fonksiyonlarının kullanılmasıyla mümkündür :

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

char *strcon(const char *s1, const char*s2)
{
    char *ptr;

    ptr = malloc (strlen(s1) + strlen(s2) + 1);
    if (ptr == NULL) {
        printf(cannot allocate memory..\n");
        exit(EXIT_FAILURE);
    }
}
```

```
    strcpy(ptr, s1);  
    strcat(ptr, s2);  
    return ptr;  
}
```

## 24 . BÖLÜM : BELİRLEYİCİLER

Belirleyiciler (specifiers), bildirimler yapılırken kullanılan ve nesnelerin ikincil özellikleri hakkında derleyicilere bilgi veren anahtar sözcüklerdir. Belirleyicileri yer belirleyicileri (storage class specifiers) ve tür belirleyicileri (type qualifiers) olarak iki grupta inceleyeceğiz.

Yer belirleyicileri genel olarak nesnelerin tutuldukları yerler hakkında bilgi verirler. Tür belirleyicileri ise nesnelerin içindeki değerlerin değiştirilip değiştirilmeyeceğine ilişkin bilgi verirler. C'de 4 tanesi yer belirleyici (**auto**, **register**, **static**, **extern**) ve 2 tane tür belirleyici (**const**, **volatile**) vardır. Bu belirleyici isimleri C dilinin birer anahtar sözcüğüdür.

### Yer ve Tür Belirleyicileriyle Bildirim İşlemi

Bildirimin genel biçimi:

[yer belirleyici] [tür belirleyici] <tür> nesne1, [nesne2], [nesne3], ...

**auto**                      **const**  
**register**            **volatile**  
**static**  
**extern**

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 geçerli bildirimlerdir. Ancak okunabilirlik açısından bildirimlerin yukarıdaki genel biçime uygun olarak yapılması tavsiye edilir.

Eğer bir bildirimde yer ya da tür belirleyicileri bir tür bilgisi olmadan kullanılırsa, önceden seçilmiş (default) olarak **int** türü bildirimin yapıldığı kabul edilir.

**const a; /\* const int a; bildirimi ile eşdeğerdir. \*/**

Şimdi yer ve tür belirleyicilerini tek tek detaylı olarak inceleyelim :

### auto Belirleyicisi

**auto** yalnızca yerel değişkenler için kullanılabilecek bir yer belirleyicisidir. **auto** belirleyicisinin global değişkenlerin ya da fonksiyonların parametre değişkenlerinin bildiriminde kullanılması derleme zamanında hata oluşturur.

Bu anahtar sözcük, nesnenin faaliyet alanı bittikten sonra kaybolacağını, bellekte kapladığı yerin geçerliliği kalmayacağını gösterir. Yerel değişkenler bulundukları blok icra edilmeye başlandığında yaratılıyorlar, söz konusu bloğun icrası bittikten sonra yok oluyorlardı. İşte **auto** belirleyicisi bu durumu vurgulamak için kullanılmaktadır. Zaten bir yerel değişken, başka bir yer belirleyici anahtar sözcük kullanılmadığı sürece (default olarak) **auto** biçiminde ele alınır. 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;      /* hata çünkü global bir değişken auto olarak bildirilemez */
function(auto int x) /* hata parametre değişkeni auto biçiminde bildirilemez */
{
    auto int var; /* yereldeğişken auto olabilir */
    int x;        /* yerel değişken auto belirleyicisi kullanılsa da auto olarak ele alınır */
    ...
}

```

**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 içerisinde" tutulacağını belirten bir anahtar sözcüktür. Değişkenlerin bellek yerine yazmaçlar içerisinde tutulması programın çalışmasını hızlandırır.

Yazmaç (**register**) nedir? Yazmaçlar CPU (central processing unit) içerisinde bulunan tampon bellek bölgeleridir. CPU içerisindeki aritmetik ve mantıksal işlemleri yapan birimin yazmaçlar ve belleklerle ilişkisi vardır. Genel olarak CPU tarafından yapılan aritmetik ve mantıksal işlemlerin her iki operandı da belleğe ilişkin olamaz. Örneğin bellekte bulunan sayi1 ve sayi2 ile gösterdiğimiz 2 sayiyi toplayarak sayi3 ile gösterdiğimiz başka bir bellek bölgesine yazmak isteyelim. Bu C'deki

```
sayi3 = sayi1 + sayi2;
```

işlemine karşılık gelmektedir. 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 data3 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 gerekmektedir. CPU yazmaçları hangi sistem söz konusu olursa olsun sınırlı sayıda. Bu nedenle birkaç değişkenden fazlası **register** belirleyicisi ile tanımlanmış olsa bile yazmaçlarda saklanamayabilir. C derleyicileri yazmaçlarda saklayamayacakları değişkenler için genel olarak hata veya uyarı mesajları vermezler. Yani derleyiciler, tutabilecekleri yazmaç sayısından fazla **register** belirleyicisine sahip değişkenlerle karşılaştıklarında bunlara ilişkin **register** belirleyicilerini dikkate almazlar.

**register** belirleyicileri ancak yerel ya da parametre değişkenleri ile kullanılabilir global değişkenler ile kullanılamazlar. Örnekler

```
register int x; /* hata x değişkeni global register belirleyicisi ile kullanılamaz */
```

```

int sample (register int y)      /* hata değil */
{
    register float x; /* hata değil */
    ...
}

```

Ne kadar değişkenin yazmaçlarda saklanabileceği bilgisayar donanımlarına ve derleyicilere bağlıdır. Ayrıca, uzunluğu tamsayı (**int**) türünden büyük olan türler genellikle yazmaçlarda saklanamazlar bu durumlarda da derleyicilerden hata veya uyarı mesajı beklenmemelidir.



Sonuç olarak **register** belirleyicisi hızın önemli olduğu çok özel ve kısa kodlarda ancak birkaç değişken için kullanılmalıdır. Modern derleyicilerin çoğu (seçime bağlı) kod optimizasyonu yaparak bazı değişkenleri yazmaçlarda saklayabilirler. Bu durum da çoğu zaman **register** anahtar sözcüğünün kullanılmasını gereksiz kılar.

## static Belirleyicisi

**static** belirleyicisine sahip değişkenler programın çalışma süresince bellekten kaybolmazlar. Bir bakıma **static** belirleyicisi **auto** belirleyicisinin zıt anlamıdır. **static** belirleyicisi ancak yerel ya da global değişkenlere birlikte kullanılabilirler. **static** belirleyicisi parametre değişkenleriyle kullanılmazlar.

**static** anahtar sözcüğünün global ve yerel değişkenlerle birlikte kullanılması farklı anlamlara gelir bu yüzden bu durumları ayrı ayrı inceleyeceğiz:

## static Anahtar Sözcüğünün Yerel Değişkenlerle Kullanılması

**static** yer belirleyicisine sahip olan yerel değişkenler programın icrası boyunca bellekte kalırlar. Başka bir deyişle, **static** anahtar sözcüğü yerel değişkenlerin ömrünü uzatmaktadır. Statik yerel değişkenler tıpkı global değişkenler gibi programın çalışmaya başlamasıyla yaratılırlar ve programın icrası bitene kadar da bellekte tutulurlar.

Statik yerel değişkenler programcı tarafından ilk değer verildikten sonra kullanılırlar. İlk değer verme işlemi programın çalışması sırasında değil, derleme zamanında derleyici tarafından yapılır. (Derleyici bellekten yer ayrılmasına yol açacak makine kodunu oluşturur ve statik değişken için bellekte yer programın yüklenmesi sırasında ayrılır. "Derleyici tarafından bellekte yer ayrılır." derken bunu kastediyoruz. Derleme zamanında gerçekten bellekte yer ayrılmıyor, bellekte yer ayırma işini yapacak makine kodu yaratılıyor.) Statik yerel değişkenler ilk değerleriyle birlikte belleğe yüklenirler. Örnek :

```
int statfunc(void)
{
    static int x = 12;    /* bu kısım yalnızca derleme sırasında ve bir kez işlem görür */
    ++x;                 /* fonksiyonun her çağırılışında yeniden yaratılmaz ve değerini korur */
    return x;
}

void main()
{
    printf("a = %d\n", statfunc()); /* a = 13 */
    printf("a = %d\n", statfunc()); /* a = 14 */
}
```

Yukarıda verilen örnekte statfunc() fonksiyonunun içerisindeki yerel x değişkeninin değeri fonksiyonun her çağırılışında bir artırılır. Çünkü x statik yerel değişken olduğu için, fonksiyonun her çağırılışında yeniden yaratılmayacak, en son aldığı değeri koruyacaktır.

Daha önce söylendiği gibi, bir adres değerine geri dönen fonksiyonlar yerel değişkenlerin adreslerine geri dönmemelidir. Ancak yerel değişkenin statik olarak tanımlanması durumunda, statik bir yerel değişken programın sonlanmasına kadar bellekteki yerini koruyacağından, fonksiyonun statik bir yerel değişkenin adresine geri dönmesinde bir sakınca yoktur:

```
char * funk(void)
{
    static char ch;
    ...
    return &ch;
}
```

Diziler bir ya da daha fazla nesnenin bir arada tanımlandığı veri yapıları olduğuna göre, yerel diziler de **static** anahtar sözcüğüyle tanımlanabilirler.

```
func(void)
{
    static aylar[12] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    ...
}
```

Yukarıdaki örnekte, func() fonksiyonu içinde aylar dizisi **static** anahtar sözcüğüyle tanımlandığı için, bir kez ilk değer verildikten sonra her çağırıldığı zaman bu değerleri koruyacak ve varlığını programın sonlanmasına kadar devam ettirecektir. Oysa **static** anahtar sözcüğü kullanılmasaydı, bu dizi func() fonksiyonun her çağırılmasıyla yeniden yaratılacak ve tüm ilk değer atamaları her defasında yeniden yapılacaktı. Bu da fonksiyonun her çağırılmasında belirli bir makine zamanının tekrar harcanmasına neden olacaktı.

### Statik Yerel Değişkenler İle Global Değişkenlerin Karşılaştırılması

Statik yerel değişkenler ömür açısından global yerel değişkenler ile aynı özellikleri gösterirler. Yani programın çalışmaya başlaması ile yaratılırlar ve programın çalışması bitince ömürleri sona erer. Ama faaliyet alanı (scope) açısından farklılık gösterirler. Global değişkenler dosya faaliyet alanına uyarken statik yerel değişkenler blok faaliyet alanına uyarlar. Statik yerel değişkenlerin faaliyet alanlarının dar olması (blok faaliyet alanına sahip olması) soyutlamayı (abstraction) ve yeniden kullanılabilirliği (reusability) kuvvetlendirmektedir. Çünkü global değişkenlere bağlı olan fonksiyonlar projeden projeye taşınamazlar ve kendi içlerinde bir bütünlük oluşturmazlar.

### static Anahtar Sözcüğünün Global Değişkenler İle Kullanılması

**static** belirleyicisi global bir değişken ile birlikte kullanılırsa, global değişkenin faaliyet alanını, yalnızca tanımlandığı modülü kapsayacak biçimde daraltır. Yani global değişkenleri kendi modülleri içine hapseder. Statik global değişkenler yalnızca tanımlandıkları modül içerisinde faaliyet gösterebilirler. Statik global değişkenleri daha iyi anlayabilmek için modül kavramının ne olduğunu bilmek zorundayız:

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 dosyaları birbirinden bağımsız olarak derlendikten sonra, hepsi birlikte bağlayıcı (linker) ile birleştirilerek tek bir .exe dosyası oluşturulur.

### Büyük Projelerin Modüllere Ayrılmasının Ne Gibi Faydaları Vardır

Eğer bütün modüller tek bir kaynak kod içerisinde 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.

Global bir değişken normal olarak tüm modüller içerisinde faaliyet gösterebilirken (Bu durumda, bir sonraki konuda açıklayacağımız gibi diğer, modüllerde **extern** bildiriminin yapılmış olması gerekmektedir.) statik global değişkenler yalnızca tanımlandıkları modül içerisinde faaliyet gösterirler.

Global değişkenler için statik tanımlamasının yalnızca faaliyet alanı üzerinde etkili olduğuna, ömür üzerinde etkili olmadığına dikkat ediniz.

Farklı modüllerde aynı isimli iki statik global değişken tanımlanabilir. "C'de aynı faaliyet alanına sahip aynı isimli birden fazla değişken tanımlanamaz" kuralını anımsayalım. İki farklı modülde tanımlanan aynı isimli iki statik global değişkenin faaliyet alanı farklıdır.

## extern Belirleyicisi ve Bağlantı Kavramı (Linkage)

**extern** belirleyicisini açıklamadan önce linkage (bağlantı) kavramı üzerinde durmak istiyoruz. İlk derslerimizde nesnelerin özelliklerinden birinin de bağlantı (linkage) özelliği olduğunu söylemiştik. C standartları üç ayrı bağlantı sınıfı tanımlamıştır :

### 1. Dış Bağlantı (external linkage)

Bir modülde tanımlanan bir nesneye programı oluşturan başka modüllerde de ulaşılabilmesi özelliğidir. Eğer bir değişkene başka bir modülde de tanınabiliyorsa o değişkenin dış bağlantısı var denir.

### 2. iç bağlantı (internal linkage)

Bir modülde tanımlanan bir nesneye yalnızca kendi modülü içerisinde ulaşılabilmesi özelliğidir. Eğer bir değişken ancak kendi modülü içerisinde tanınabiliyorsa, diğer modüllerde tanınamıyorsa, o değişkenin iç bağlantısı var denir.

### 3. bağlantısız (no linkage)

Bir modülde tanımlanan bir nesneye, o modül içinde yalnızca belirli bir blok içinde ulaşılabilmesi özelliğidir. Eğer bir değişken ancak kendi modülü içerisinde belirli bir blok dahilinde tanınabiliyor, bu blok dışında kendi modülü içinde tanınmıyorsa, o değişkenin bağlantısı yoktur denir.

Global değişkenlerin dış bağlantısı varken, statik global değişkenlerin iç bağlantısı vardır. Yerel değişkenlerin bağlantısı yoktur.

Bütün belirleyiciler içinde belki de anlaşılması en güç olanı extern belirleyicisidir. extern belirleyicisi genel olarak, derleyiciye nesnenin başka bir modülde tanımlandığını bildirmek için kullanılır

Bir proje MOD1.C ve MOD2.C biçiminde iki modülden oluşmuş olsun :

```
MOD1.C          MOD2.C
int a;          int fonk2()
               {
float fonk1()    ...
{               a = 300;    /* HATA */
    ...        }
    a = 100;
    ...
}

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

MOD1.C modülünde tanımlanmış olan a değişkeni global bir değişkendir ve dosya faaliyet alanına uyduğu için normal olarak proje içindeki diğer modüllerde de faaliyet gösterebilir. MOD2.C modülünde de faaliyet gösterebilir. Fakat iki modülün ayrı ayrı derlendiği yukarıdaki örnekte problemli bir durum söz konusudur. Çünkü MOD2.C modülünün derlenmesi sırasında derleyici a değişkeninin MOD1.C modülü içerisinde global olarak tanımlandığını bilemez. MOD2.C modülünü derlerken a değişkeni hakkında bilgi bulamayan derleyici bu durumu hata (error) olarak belirler. (C dilinde bir değişken ancak önceden bildirilmişse kullanılabilir.) extern belirleyicisi derleyiciye, ilgili global değişkeninin kendi modülü içerisinde değil de bir başka modül içerisinde tanımlı olduğunu bildirme amacıyla kullanılır.

MOD2.C modülündeki a değişkeninin extern olarak bildirilmesi durumunda söz konusu problem ortadan kalkar.

#### MOD2.C

extern int a; /\*extern bildirimiyle a değişkeninin başka bir modülde tanımlanmış olduğu belirtiliyor\*/

```
int fonk2()
{
    ...
    a = 300;
    ...
}
```

extern bildirimini gören derleyici değişkenin projeye ait başka bir modülde tanımlandığını varsayarak hata durumunu ortadan kaldırır. Ancak derleyici makine kodu üretirken extern olarak bildirilmiş bir değişkenin bellekteki yerini tespit edemeyeceğinden , bu işlemi bütün modülleri gözden geçirecek olan bağlayıcı programa bırakır. Böylece değişkenin tanımlandığı modülü bulup, extern olarak bildirilmiş olanlarla ilişkilendirme işlemi bağlayıcı (linker) tarafından yapılmaktadır. Yani extern belirleyicisi ile programcı derleyiciye, derleyici ise bağlayıcıya bildirimde bulunmaktadır.

extern belirleyicisini gören derleyici bellekte bir yer ayırmaz. extern bildirimi bir tanımlama değildir, bildirimdir.

Aslında yalnız değişkenler için değil fonksiyonlar için de extern belirleyicisinin kullanılması söz konusudur. C derleyicileri kendi modülleri içerisinde tanımlanmadıkları halde çağırılan (standart C fonksiyonları gibi) fonksiyonları otomatik olarak extern kabul ederler. Bu nedenle fonksiyon prototiplerinde ayrıca extern belirleyicisini yazmaya gerek yoktur. Çünkü derleyici tarafından yazılmış varsayılır.

Örneğin yukarıda verilen örnekte MOD2.c modülünde bulunan y1 fonksiyonu içerisinde bulunan x1 fonksiyonu çağırılıyor olsun:

MOD1.c	MOD2.c
<pre>int a;  float x1() {     ...     a = 100;     ... }  main() {     ... }</pre>	<pre>extern int a; extern float x1(void);  int y1() {     float f;     ...     f = x1();     a = 300;     ... }</pre>

MOD2.c modülünde x1 fonksiyonu için yazılmış olan prototip ifadesini inceleyiniz:

```
extern float x1(void);
```

Bu örnekte x1 fonksiyonu başka bir modülde tanımlı olduğu için prototip ifadesine extern belirleyicisi konmuştur. Ancak konulmasaydı derleyici zaten extern varsayacaktı. (Tıpkı yerel değişkenler için auto belirleyicisini varsaydığı gibi.)

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 çıkacaktır.

extern belirleyicisinin tek bir modül söz konusu olduğunda "amaç dışı" bir kullanımı vardır. Aşağıdaki örnekte main fonksiyonu içerisindeki global x değişkeni, tanımlanmadan önce kullanıldığından hataya neden olmaktadır.

```
void main()
{
    ...
    x = 100;
}

int x;          /* x global bir değişken ama tanımlanmasından önce yukarıda kullanılmış */

int fonk()
{
    ....
    x              =                200;
    ...
}
```

yukarıdaki kod derlendiğinde , main fonksiyonunun içerisindeki x değişkeninin bildiriminin bulunamadığını ifade eden bir hata mesajıyla karşılaşılır. Bu durumda, eğer bir global değişken tanımlamadan önce kullanılıyorsa, hata oluşmaması için daha önce extern bildiriminde bulunulmalıdır.

```
extern int x;

void main()
{
    ...
    x = 100;
}

int x;

int fonk()
{
    ....
    x              =                200;
    ...
}
```

extern bildirimini bu şekilde kullanmak yerine, global değişkeni programın en başında tanımlamak daha iyi bir tekniktir.

### extern bildirimlerinin yapılaş yerleri

extern bildirimleri kaynak kodun herhangi bir yerinde yapılabilir. Global ya da yerel olması sözkonusu değildir. Bildirimin yapıldığı yerden dosya sonuna kadar olan bölge için geçerlidir. Ancak extern bildirimlerinin programın tepesinde ya da programcıya ait bir başlık dosyasının içinde yapılması daha uygun olur.

## const belirleyicisi

const ilk değer atandıktan sonra nesnenin içeriğinin değiştirilemeyeceğini anlatan tür belirleyici, bir anahtar sözcüktür. Yerel, global ve parametre değişkenleriyle birlikte kullanılabilir. Örneğin :

```
const double PI = 3.14159265    /*Geçerli */

main()
{
    const int i = 10;

    i = 100;                /* hata */
}
```

Bir değişken const belirleyicisi ile tanımlanacaksa ilk değer verilmelidir. Aksi halde const belirleyicisi kullanmanın bir anlamı kalmaz. Aşağıdaki örneği inceleyiniz.

```
void sample (void)
{
    const int a; /* anlamsız */
}
```

Bu örnekte a yerel bir değişkendir, dolayısıyla rastgele bir değere sahiptir. İçeriğini bir daha değiştiremeyeceğimize göre tanımlanmasının da bir anlamı olamaz.

const belirleyicisinin kullanım amacı ne olabilir diye düşünebilirsiniz? Sıklıkla şu merak edilir:

"Eğer const belirleyicisi koruma amaçlı olarak kullanılıyorsa kim kime karşı korunuyor? const belirleyicisinin iki yararlı işlevi vardır.:

Okunabilirliği artırır. Çünkü programı inceleyen bir kişi const belirleyicisine sahip değişkenin değerinin bir daha değiştirilemeyeceğini düşünerek daha fazla bilgi edinir.

Yanlışlıkla nesnenin değerinin değiştirilmesi engellenir.

const belirleyicisi değeri hiç değişmeyecek sabitler için kullanılmalıdır. const bildirimlerinin nesne yarattığına nesnenin yalnızca okunabildiğine (read only) dikkat ediniz.

NOT : C++'da const belirleyicisi zorunluluk olmadıkça nesne yaratmaz. (#define önişlemci komutu gibidir, ancak derleme aşamasında işlem görür)

const anahtar sözcüğünün gösterici tanımlamalarında kullanılması :

const anahtar sözcüğünün göstericilerle birlikte üç ayrı kullanıma biçimi vardır. Kullanılan her biçimde tanımlanan göstericinin özelliği değişecektir :

1. const anahtar sözcüğünün gösterici tanımlamasında ilk sözcük olarak kullanılması. Örnekler :

```
double dizi[20];
int k;
const char *str;
const int *ptr;
const double *dptr;
```

Bu durumda göstericinin gösterdiği nesne (yani gösterici içindeki adreste yer alan nesne) değiştirilemez ancak göstericinin kendi değeri değiştirilebilir. Bu tür bir tanımlamadan sonra

göstericinin gösterdiği yerdeki nesnenin değiştirilmeye çalışılması derleme zamanı hatası ile neticelenecektir:

```
*str = 'A';           /* error */
ptr[2] = 25;          /* error */
*dptr = 4.5;          /* error */
str = (char *)0x1FFF0; /* legal */
ptr = &k;             /* legal */
dptr = dizi;          /* legal */
```

const anahtar sözcüğünün göstericilerle birlikte kullanılmasında en sık görülen biçim budur ve özellikle fonksiyonların parametre değişkenleri olan göstericilerle bu biçim kullanılır:

```
void func(const char *s)
{
...
}
```

Yukarıda func fonksiyonunun parametre değişkeni olan s göstericisi değerini fonksiyonun çağırmasıyla alacaktır. Fonksiyon çağırma ifadesindeki arguman fonksiyonun çağırılmasıyla s göstericisine kopyalanacaktır. Ancak fonksiyon içinde \*s nesnesine ya da s[x] nesnesine bir atama yapılamaz. Yani s göstericisinin gösterdiği yerdeki nesne (fonksiyona adresi gönderilen nesne) değiştirilemez. Yukarıdaki örnekte const anahtar sözcüğünün kullanılması herşeyden önce okunabilirlik ile ilgilidir. Yukarıdaki kodu okuyan bir C programcısı func fonksiyonunun dışarıdan adresi alınan nesnenin yalnızca değerinden istifade edeceğini, yani bu nesnenin değerini değiştirmeyeceğini anlar. Geleneksel olarak, const anahtar sözcüğünün, parametre değişkeni olan göstericilerin tanımlanmasında kullanılmasında okunabilirlik açısından bir mesaj olarak kabul edilir:

```
void func(char *s)
{
...
}
```

Yukarıdaki kodu okuyan bir C programcısı func fonksiyonunun (aslında sentaks açısından bir zorunluluk bulunmasa da) kendisine adresi gönderilen nesneyi değiştireceğini anlar. Kodu okuyan kişi şöyle düşünecektir : Eğer func fonksiyonu adresi gönderilen nesneyi değiştirmeyecek olsaydı, s göstericisi const anahtar sözcüğü ile tanımlanırdı.

2. const anahtar sözcüğünün gösterici isminden hemen önce kullanılması . Örnekler :

```
...
char ch;
int m;
float f;
float fdizi[20];
char *const str = &ch ;
int *const ptr = (int *) 0x 1AC3;
float *const fptr = &f;
```

Bu durumda göstericinin gösterdiği yerdeki nesne değiştirilebilir ama göstericinin içeriği değiştirilemez:

```
str = (char *) 0x1FC0; /* error */
ptr = &m;             /* error */
fptr = fdizi;         /* error */

*str = 'A';           /*legal */
*ptr = 15;            /*legal */
fptr[2] = 12.3f;      /*legal */
```

const anahtar sözcüğünün bu kullanım biçiminde gösterici tanımlanırken ilk değer verilmelidir, yoksa const anahtar sözcüğünün kullanılmasının bir anlamı kalmaz.

3. const anahtar sözcüğünün hem tanımlama ifadesinden önce hem de gösterici isminden önce kullanılması. Örnekler :

```
...
char ch;
int k;
const char *const str = (char *) 0x1AAA;
const char *const ptr = (int *) 0x1FFF;
```

Bu durumda ne göstericinin içeriği değiştirilebilir ne de göstericinin gösterdiği yerdeki nesne değiştirilebilir. Her iki durum da derleme zamanında error oluşumuna yol açacaktır :

```
str = &ch;          /* error */
*str = 'A';         /* error */
ptr = &k;           /* error */
*ptr = 12;          /* error */
```

const sözcüğünün bu biçimde kullanılması yine daha çok parametre değişkenlerinin tanımlanmasında görülür ise de bu uygulamalarda seyrek olan bir durumdur.

## volatile belirleyicisi

Derleyiciler optimizasyon amacıyla nesneleri geçici olarak yazmaçlarda tutabilir. Yazmaçlardaki bu çeşit geçici barınmalar register belirleyicisi kullanılsa da derleyiciler tarafından yapılabilir. Örneğin:

```
int kare (int a)
{
    int x;

    x = a * a;
    return x;
}
```

Yukarıdaki fonksiyonda x geçici bir değişkendir, dolayısıyla derleyici x değişkenini bellekte bir yerde saklayacağına, geçici olarak yazmaçlarından birinde saklasa da işlevsel bir farklılık ortaya çıkmaz. Bu çeşit uygulamalarda derleyicinin değişkenleri geçici olarak yazmaçlarda saklaması işlemleri hızlandırmaktadır. Aşağıdaki kodu inceleyiniz:

```
int a;
int b;

...
a = b;
if (a == b) {
    ...
}
```

Bu örnekte doğal olarak ilk adımda b değişkeni a değişkenine aktarılacak üzere yazmaçlardan birine çekilecektir. Ancak derleyici if içerisindeki ifadede a == b karşılaştırmasını yapmak için bellekteki b yerine yazmaçtaki b'yi kullanabilir. Verdiğimiz iki örnekte de derleyici birtakım optimizasyonlarla programı işlevi değişmeyecek biçimde daha hızlı çalışır hale getirmek istemiştir. Ancak kimi uygulamalarda derleyicinin bu biçimde davranması hatalara neden olabilmektedir. 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 optimizasyon amacıyla geçici olarak yazmaçlarda tutulması arzu edilmeyen sonuçların oluşmasına yol açabilmektedir. volatile "Değişkenleri optimizasyon amacıyla yazmaçlarda bekletme, onları bellekteki gerçek yerlerinde kullan!" anlamına gelen bir tür belirleyicisidir. Bu anlamıyla volatile belirleyicisini register belirleyicisi ile zıt anlamlı olarak düşünebiliriz. Yukarıdaki örnekte b değişkenini volatile olarak bildirerek anlattığımız gibi bir problemin çıkması engellenebilir.

```

int a;
volatile int b;
...

```

volatile çok özel uygulamalarda kullanılabilen bir belirleyicidir.

Yerel, parametre ya da global değişkenlerle birlikte kullanılabilen volatile belirleyicisi ancak çok özel uygulamalarda önemli olabilmektedir. Bu belirleyicinin bilinçsizce kullanılmasının performansı kötü yönde etkileyebileceğini unutmayınız.

Belirleyicilerin kullanılışlarını gösteren bir örnek :

```

#include <stdio.h>

int a;
extern int b;
static int c;

void func(int d, register int e)
{
    auto int g;
    int h;
    static int i;
    extern int j;
    register int k;
}

```

yukarıdaki programda tanımlanan değişkenlerin faaliyet alanı, ömür ve bağlantı açısından özellikleri aşağıdaki tabloda belirtilmektedir :

(isim) name	(ömür) storage duration	(faaliyet alanı) scope	(bağlantı) linkage
a	statik (static)	dosya (file)	dış (external)
b	statik (static)	dosya (file)	dış (external) *
c	statik (static)	dosya (file)	dış (external)
d	otomatik (automatic)	blok (block)	yok (none)
e	otomatik (automatic)	blok (block)	yok (none)
g	otomatik (automatic)	blok (block)	yok (none)
h	otomatik (automatic)	blok (block)	yok (none)
i	statik (static)	blok (block)	yok (none)
j	statik (static)	blok (block)	dış (external) *

k	otomatik (automatic)	blok (block)	yok (none)
---	----------------------	--------------	------------

\* b ve j değişkenleri kendi modülleri içinde tanımlanan global değişkenler de olabilir. Bu durumda bu değişkenle static anahtar sözcüğüyle tanımlanmış olmaları durumunda iç bağlantıya sahip olacak, static anahtar sözcüğü olmadan tanımlanmışlar ise dış bağlantıya sahip olacaklardır.

belirleyicilerin hangi tür değişkenlerle kullanılabileceklerini gösteren tablo :

belirleyici	global değişken	yerel değişken	parametre değişkeni
auto		√	
static	√	√	
register		√	√
extern	√	√	
volatile	√	√	√
const	√	√	√

## 25 . BÖLÜM : YAPILAR

Yapılar (structures) da diziler gibi birden fazla nesneyi içlerinde tutabilen bir veri türüdür. Yapıların da elemanları bellekte ardışıl (contiguous) bir biçimde bulunur. Fakat yapıların dizilerden temel farkı şudur: Diziler aynı türden nesneleri içinde tutabilirken, yapılar farklı türlerden nesneleri tutabilirler. Yapıların kullanılmasının ana nedeni budur. Çoğu zaman, türleri farklı bir takım nesneler, mantıksal olarak bir bütün oluşturabilirler. İsim, yaş, departman, ücret, öğrenim durumu gibi bilgileri farklı türden nesneler içinde saklayabiliriz ve bunların tamamı çalışan bir personele ait bilgiler olabilir. Bu gibi aynı konu ile ilgili farklı türden veriler yapılar içinde saklanır.

### Yapı Bildiriminin Genel Şekli

```
<struct> [yapı ismi] {
    <tür> <m1>;
    <tür> <m2>;
    <tür> <m3>;
    ...
};
```

Yukarıdaki gösterimde **struct** bir anahtar sözcüktür. Bildirimde mutlaka yer alması gerekmektedir. Yapı ismi (structure tag) C dilinin isimlendirme kurallarına uygun olarak seçilmiş bir isimdir.

Örnek bildirimler:

```
struct SAMPLE {
    int a;
    long b;
    char ch;
};
```

```
struct DATE {
    int day, month, year;
};
```

```
struct POINT {
    int x, y;
};
```

Yapı isimleri (structure tags) geleneksel olarak büyük harfle yazılır. Bu bir zorunluluk değildir.

Yapı bildiriminin yapılması bellekte derleyici tarafından bir yer ayrılmasına neden olmaz. Yani bir tanımlama (definition) söz konusu değildir. Bu bildirimle (declaration) programcının yarattığı yeni bir veri türü hakkında derleyiciye bilgi verilmektedir.

Yapı bildiriminin yapılmasından sonra artık bildiri yapılmış yapı türünden nesneler tanımlanabilir. Yapı bildiriminin yapılması ile yeni bir veri türü yaratılmıştır. Derleyici artık bu tür hakkında bilgi sahibi olduğundan, bu yeni veri türünden, nesneler tanımlanabilir. Yeni veri türünden nesnelerin tanımlanması durumunda artık derleyici bu nesneler için bellekte ne kadar bir alan ayırması gerektiği bilgisine sahip olacaktır.

### Yapı Türünden Nesne Tanımlaması

```
<struct> <yapı ismi> <nesne ismi>;
```

Örnekler :

```
struct DATE x;          /* x DATE yapısı türünden bir nesnedir. */
```

```
struct POINT p1, p2; /* p1 ve p2 POINT yapısı türünden nesnelerdir. */
struct SAMPLE sample; /* sample SAMPLE yapısı türünden bir nesnedir. */
```

Yapı değişkenleri bileşik nesnelerdir. Yani parçalardan oluşurlar. Zaten yapı bildirimlerinin yapılmasının amacı da bu parçaların isimleri ve türleri hakkında derleyiciye bilgi vermektir. Bir yapı bildirimini gören derleyici, daha sonra bildirimi yapılan yapı türünden bir değişken tanımlanması durumunda, bu değişken için bellekte ne kadar yer ayracağını, ve ayırdığı yeri ne şekilde organize edeceğini bilir.

Bir yapı değişkeni (nesnesi) için, yapı bildiriminde belirtilen elemanların toplam uzunluğu kadar (byte olarak) yer ayrılır.

```
struct SAMPLE {
    int a;
    long b;
    char ch;
};

void main()
{
    struct SAMPLE x;

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

Yukarıdaki program parçasında ekrana (DOS altında) 7 sayısı yazdırılacaktır. Çünkü yapı nesnesi olan x nesnesinin bellekte kapladığı yer üç parçasının kapladığı uzunluğun toplamıdır. Bunu aşağıdaki şekilde de ifade edebiliriz:

```
sizeof(x) == sizeof(int) + sizeof(long) + sizeof(char)
```

(Hizalama [alignment] konusuna geldiğimizde bu durumu daha detaylı olarak inceleyeceğiz.)

## Yapı Elemanlarına Erişim

Yapıların dizilerden önemli bir farkı da elemanlara erişim konusundadır. Dizi elemanlarına erişim, dizi ismi (aslında bir adres bilgisidir) ve indis operatörü [ ] (index operator - subscript operator) yoluyla yapılırken, yapı elemanlarına erişim doğrudan yapı nesnesinin ve elemanın isimleriyle olur. Yapı elemanlarına erişimde nokta operatörü kullanılır.

Nokta operatörü (.) iki operand alan aritmetik konumunda (binary infix) bir operatördür. Bu operatörün sol tarafındaki operand bir yapı türünden değişken olmalıdır. Sağ tarafındaki operand ise ilgili yapı türünün (yani yapı bildiriminde önceden belirlenmiş) bir üyesi olmak zorundadır. Nokta operatörü, operatör öncelik tablosunun en yüksek düzeyinde bulunur.

Yapı nesneleri de yerel ya da global olabilir. Diğer nesnelerde olduğu gibi yapı nesneleri içinde faaliyet alanı (scope) kavramı söz konusudur. Tüm blokların dışında tanımlanan yapı nesneleri global iken blokların içlerinde tanımlanan yapı değişkenleri yereldir. Global yapı nesneleri diğer türden global nesneler gibi statik ömür karakterine sahiptir ve dosya faaliyet alanı kuralına uyarlar. Yerel yapı nesneleri ise dinamik ömürlüdür ve blok faaliyet alanı kuralına uyarlar.

Yapı değişkenlerine programcı tarafından değer verilmemişse, yapı değişkeni yerel (local) ise tüm elemanlarında rasgele değerler (garbage value) bulunur. Yapı nesnesi global ise tüm elemanlarında 0 değeri bulunur.

Bir yapı nesnesi tanımlanarak, bu yapı nesnesinin elemanlarına nokta operatörü ile ulaşıldığında artık bu elemanların her biri ayrı bir nesne olarak ele alınır. Bu nesnelerin yapı dışında tanımlanan nesnelerden herhangi bir farkı yoktur, nesnelerin tüm özelliklerine sahiptirler.

Örnek:

```
struct SAMPLE {
    int a;
    long b;
    char c;
};

int main()
{
    struct SAMPLE x;

    x.a = 10;
    x.b = 200000;
    x.c = 'M';
    printf("x.a = %d\nx.b = %ld\nx.c = %c\n", x.a, x.b, x.c);
    return 0;
}
```

Yukarıdaki örnekte görüldüğü gibi x.a, x.b ve x.c yapı elemanları ayrı birer nesne özelliği gösterirler. Bu elemanlara ayrı ayrı ulaşabilir ve ayrı ayrı atamalar yapabiliriz. Bu nesneleri ++ ve -- operatörlerine ya da & operatörüne operand yapabiliriz.

## Yapı Bildirimlerinin Yapılış Yerleri

Yapı bildirimi global ya da yerel olarak yapılabilir. Eğer yerel olarak yapılırsa yalnızca bildirimin yapıldığı blokta o yapı türünden nesne tanımlanabilir. Bildirim global olarak yapılırsa her yerde o yapı türünden değişken tanımlanabilir.

Uygulamalarda hemen hemen her zaman yapı bildirimleri programın tepesinde global olarak yapılır. (Yapı bildirimi başlık dosyalarının içinde de yapılabilir. Bu durumu ileride detaylı olarak inceleyeceğiz.)

Yapı bildiriminde yer alan yapı elemanlarının faaliyet alanı yapı bildirim bloğuyla sınırlıdır. Yani yapı bildirimi bloğu dışında, yapı elemanı ile aynı isimli bir değişken tanımlanabilir. Yapı ismi (structure tag) ile aynı isimli değişkenler de tanımlanabilir.

Aşağıdaki program parçasında isimlendirme açısından bir hata yok. Okunabilirlik açısından iyi bir uygulama değil.

```
struct SAMPLE {
    int sample;
    long x;
};

int main()
{
    int sample, SAMPLE;
    ...
    return 0;
}
```

## Yapı Elemanlarının Bellekteki Yerleşimi

Yapı elemanları belleğe, bildirimde ilk yazılan eleman küçük adreste olacak biçimde, ardışıl olarak yerleştirilir.

Örnek:

```
struct SAMPLE {
    int a;
```

```

    long b;
    char c;
};

int main()
{
    struct SAMPLE x;

    printf("x.a adresi = %p\nx.b adresi = %p\n x.c adresi = %p\n", &x.a, &x.b, &x.c);
}

```

## Yapı Değişkenlerine İlk Değer Verilmesi (initialization)

Yapı değişkenlerine küme parantezleri içerisinde ilk değer verilebilir. Bu durumda verilen ilk değerler sırası ile yapı elemanlarına yerleştirilir. Daha az sayıda yapı elemanına ilk değer verilebilir, bu durumda ilk değer verilmemiş yapı elemanları 0 değeri alırlar.

```

struct DATE {
    int day, month, year;
};

int main()
{
    struct DATE x = {10, 12, 1999};
    struct DATE y = {10, 12};

    printf("%d %d %d\n", x.day, x.month, x.year);
    printf("%d %d %d\n", y.day, y.month, y.year);
}

```

Dizilerde olduğu gibi, yapılarda da yapı elemanlarından daha fazla sayıda elemana ilk değer vermek derleme zamanında hata oluşumuna neden olacaktır.

## Yapı Elemanı Olarak Göstericilerin Kullanılması

Yapının bir elemanı herhangi türden bir gösterici olabilir. Bu durumda bu yapı elemanına ulaşıldığında, bu gösterici de yapı elemanı olmayan göstericiler gibi değerlendirilir

Örnek:

```

struct PERSON {
    char *name;
    int no;
};

int main()
{
    struct PERSON x;

    x.name = "Necati";
    x.no = 125;
    printf("%s %d\n", x.name, x.no);
    return 0;
}

```

Yukarıdaki örnekte yapı elemanlarına ilk değer verme sentaksıyla da (initialization) değer atanabilirdi:

```

....
struct PERSON x = {"Necati", 125};
...

```

## Yapı Elemanı Olarak Dizilerin Kullanılması

Yapının bir elemanı herhangi türden bir dizi olabilir. Dizilerde olduğu gibi, bu durumda da yapının dizi elemanının ismi nesne belirtmez. (adres bilgisi belirtir).

Örnek:

```
struct PERSON {
    char name[30];
    int no;
};

int main()
{
    struct PERSON x;
    gets(x.name);
    puts(x.name);
    putchar(x.name[3]);
    x.name++ /* error. dizi ismi nesne değildir. */
    return 0;
}
```

## Yapı Nesneleri Üzerinde Yapılabilecek İşlemler

Yapı değişkenleri bir bütün olarak aritmetik operatörlerle ve karşılaştırma operatörleri ile işleme sokulamaz. Yapı nesneleri üzerinde şu işlemler yapılabilir:

1. Bir yapı değişkeninin adresi alınabilir.
2. Aynı türden iki yapı değişkeni birbirine atanabilir.
3. **sizeof** operatörü ile bir yapı nesnesinin bellekte kapladığı alan bulunabilir.

Aynı türden iki yapı değişkeninin 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 gerekmektedir. İki yapı değişkeni de aynı türden değilse bu durum derleme aşamasında hata oluşturur. İki yapının aynı türden olmaları için aynı yapı ismi ile tanımlanmış olmaları gerekir. Aşağıdaki iki yapı, elemanlarının türleri ve dizilişleri aynı olduğu halde birbirleri ile aynı değildir ve birbirlerine atanamazlar.

```
struct POINT_1 {
    int x, y;
};
```

```
struct POINT_2{
    int x, y;
};
```

```
...
struct POINT_1 a;
struct POINT_2 b;
```

```
b.x = 10;
b.y = 20;
```

```
a = b /* derleme zamanında hata. a ve b yapıları aynı türden değil. */
```

## Yapı Türünden Adresler ve Göstericiler

Bir yapı değişkeninin adresi alınabilir. Bu durumda elde edilen adresin sayısal bileşeni yapının bellekteki başlangıç adresi, tür bileşeni ise yapı ile aynı türden adrestir. Bir yapı türünden göstericiler de tanımlanabilir. Yapı türünden göstericilere aynı yapı türünden bir adres atanmalıdır.

Örnek:

```
struct POINT_1 *p = &a;
```

## Göstericiler Yapı Elemanlarına Erişim

p yapı türünden bir gösterici ve elm de o yapının bir elemanı olmak üzere erişim şu biçimde yapılır:

(\*p).elm

Burada öncelik operatörünün kullanılması zorunludur. Kullanılmazsa önce p.elm ele alınır daha sonra \* operatörü işleme sokulurdu. Bu durum da derleme zamanında hata oluşumuna neden olurdu.

## Yapıların Fonksiyonlara Parametre Olarak Geçirilmesi

Yapıları fonksiyonlara geçirebilmek için iki yöntem kullanılabilir:

1. Yapının kendisinin fonksiyona parametre olarak geçirilmesi.  
Aynı türden yapı nesnelerinin birbirlerine atanabilmesi özelliğinden faydalanılır. Bu yöntemde fonksiyonun parametre değişkeni bir yapı değişkeni olur. Fonksiyon da aynı yapı türünden bir yapı değişkeni ile çağırılır. Aynı türden iki yapı değişkeninin birbirine atanması geçerli olduğuna göre bu işlem de geçerlidir. Bu işlem bir blok kopyalaması gerektirdiği için hem bellek hem de zaman açısından görece bir kayıba neden olur. Üstelik bu yöntemle, fonksiyon kendisine gönderilen argümanları değiştiremez. Yani fonksiyon değerle çağırılmıştır. (call by value)

Örnek :

```
struct PERSON {
    char name[30];
    int no;
};

void disp(struct PERSON y)
{
    printf("%s %d\n", y.name, y.no);
}

int main()
{
    struct PERSON x = {"Necati ERGIN", 125};
    disp(x);
}
```

2. Yapı değişkenlerinin adreslerinin geçirilmesi.  
Bu yöntemde fonksiyonun parametre değişkeni yapı türünden bir gösterici olur. Fonksiyon da bu türden bir yapı değişkeninin adresi ile çağırılır. Bu yöntem iyi bir tekniktir. Hemen her zaman bu yöntem kullanılmalıdır. Bu yöntemde yapı ne kadar büyük olursa olsun aktarılan yalnızca bir adres bilgisidir. Üstelik bu yöntemde fonksiyon kendisine adresi gönderilen yapı değişkenini değiştirebilir. Şüphesiz böyle bir aktarım işleminin mümkün olabilmesi yapı elemanlarının bellekteki ardışıklık özelliğinden faydalanılmaktadır. Örnek :

```
struct PERSON {
    char name[30];
    int no;
};

void disp(struct PERSON *p)
{
```



```
printf("%s %d\n", (*p).name, (*p).no);
}
```

```
int main()
{
    struct PERSON x = {"Necati Ergin", 156};

    disp(&x);
}
```

## Yapıların Kullanım Yerleri

Yapılar temel olarak 3 nedenden dolayı kullanılırlar.

1. Birbirleri ile ilişkili olan değişkenler yapı elemanları olarak bir yapı içerisinde toplanırsa algısal kolaylık sağlanır. Örneğin düzlemde bir nokta, bir tarih bilgisi, bir depoda bulunan mamüllere ilişkin özellikler bir yapı ile temsil edilebilir.
2. C dilinde bir fonksiyon en fazla 8 - 10 parametre almalıdır. Daha fazla parametreye sahip olması kötü bir tekniktir. Bir fonksiyon çok fazla parametrik bilgiye gereksinim duyuyorsa, bu parametrik bilgiler bir yapı biçiminde ifade edilmelidir. O yapı türünden bir değişken tanımlanmalı, bu değişkenin adresi fonksiyona parametre olarak gönderilmelidir. Örneğin bir kişinin nüfus cüzdan bilgilerini parametre olarak alıp bunları ekrana yazdıracak bir fonksiyon tasarlayacak olalım. Nüfus cüzdanı bilgilerinin hepsi bir yapı biçiminde ifade edilebilir ve yalnızca bu yapının adresi fonksiyona gönderilebilir.
3. C dilinde fonksiyonların tek bir geri dönüş değeri vardır. Oysa fonksiyonların çok değişik bilgileri çağıran fonksiyona iletmesi istenebilir. Bu işlem şöyle yapılır: İletilecek bilgiler bir yapı biçiminde ifade edilir. Sonra bu türden bir yapı değişkeni tanımlanarak adresi fonksiyona gönderilir. Fonksiyon da bu yapı değişkeninin içeriğini doldurur. Yani fonksiyon çıkışında bilgiler yapı değişkeninin içinde olur.
4. C dilinin tasarımında belirlenmiş veri türleri yalnızca temel türleri kapsayacak şekildedir. Örneğin tarihler ya da karmaşık sayılar için C dilinde belirlenmiş temel bir tür yoktur (default veri tipi yoktur.) Yapıları bu gibi veri türlerini oluşturmak için kullanıyoruz.

## Ok Operatörü

OK operatörü - ve > karakterlerinin yanyana getirilmesiyle oluşturulur. İki operand alan aritmetik konumunda bir operatördür. (Binary infix ) Ok operatörü öncelik tablosunun en yüksek öncelik seviyesindedir. -> operatörünün sol tarafındaki operand yapı türünden bir adres olmalıdır. -> operatörünün sağ tarafındaki operand ise ilgili yapının bir elemanı olmalıdır. Sol tarafındaki operand ile belirtilen yapının (o adresteki yapının) sağ tarafında belirtilen elemanına ulaşmak için kullanılır.

p, yapı türünden bir nesnenin adresini tutuyor olsun. Aşağıdaki iki ifade eşdeğerdir.

```
(*)p.a          p->a
```

Yani nokta ve ok operatörlerinin her ikisi de elemana erişmek için kullanılır. Ancak nokta operatörü yapı değişkeninin kendisiyle ok operatörü ise adresiyle erişim sağlar. Okunabilirlik açısından ok operatörünün solunda ve sağında boşluk bırakılmamasını tavsiye ediyoruz.

&p->a ifadesiyle p'nin gösterdiği yapının a elemanının adresi alınır. (Ok operatörü adres operatörüne göre daha önceliklidir.)

## Bir Yapı Türüne Geri Dönen Fonksiyonlar

Bir fonksiyonun geri dönüş değeri bir yapı türünden olabilir.

Örnek:

```
struct POINT {  
    int x, y;  
};  
  
struct POINT make_point(int x, int y);  
  
int main()  
{  
    struct POINT a;  
  
    a = make_point(3, 5);  
    ...  
    return 0;  
}  
  
struct POINT make_point(int x, int y)  
{  
    struct POINT temp;  
  
    temp.x = x;  
    temp.y = y;  
    return temp;  
}
```

Bu durumda geri dönüş değerinin aktarıldığı geçici bölge de **struct** POINT türünden olur. Böyle bir fonksiyonun geri dönüş değeri aynı türden bir yapı değişkenine atanabilir.

Ancak böyle fonksiyonlar küçük yapılar için kullanılmalıdır. Çünkü **return** ifadesiyle geçici bölgeye geçici bölgeden de geri dönüş değerinin saklanacağı değişkene atamalar yapılacaktır. Bu da kötü bir teknik olmaya adaydır. Küçük yapılar için tercih edilebilir. Çünkü algısal karmaşıklığı daha azdır.

`make_point` fonksiyonunun parametre değişkenleri ile `POINT` yapısının üyeleri aynı isimde oldukları halde faaliyet alanları farklıdır.

## Yapı Türünden Bir Alanın Dinamik Olarak Tahsis Edilmesi

Nasıl bir dizi için bellek alanı dinamik olarak tahsis edilebiliyorsa bir yapı için de böyle bir tahsisat yapılabilir. Aşağıdaki örnekte `DATE` türünden bir yapı için dinamik tahsisat yapılmıştır:

```
struct DATE {  
    int day, month, year;  
};  
  
int main()  
{  
    struct DATE *p;  
  
    p = (struct DATE *) malloc (sizeof(struct DATE));  
    p->day = 10;  
    p->month = 12;  
    p->year = 2000;  
    printf("%d %d %d\n", p->day, p->month, p->year);  
    free(p);  
    return 0;  
}
```

## Bir Yapı Adresine Geri Dönen Fonksiyonlar

Bir fonksiyonun geri dönüş değeri bir yapı türünden adres de olabilir.

Örneğin:

```
struct POINT {
    int x, y;
};

struct POINT *dump_point(struct POINT *p);

int main()
{
    struct POINT *p;
    struct POINT a;

    a.x = 10;
    b.x = 20;
    p = dump_point(&a);
    return 0;
}

struct POINT dump_point(struct POINT *p)
{
    struct POINT *retval;

    if ((retval = (struct POINT *) malloc(sizeof(struct POINT))) == NULL) {
        printf("not enough memory");
        exit(EXIT_FAILURE);
    }

    *retval = *p;
    return retval;
}
```

Tabi böyle bir fonksiyon yerel bir yapı değişkeninin adresine geri dönemez. Global bir yapı değişkeninin adresine de geri dönmesinin bir anlamı yoktur. (Statik bir yerel yapı nesnesinin adresiyle geri dönülmesinde hiçbir sakınca yok. Statik yerel değişkenler konusundaki bilgilerinizi hatırlayınız.) En iyi tasarım, içeride dinamik olarak tahsis edilmiş bir yapının adresine geri dönmektir.

Tahsis edilen alanın "heap" bellek alanına iade edilmesi, fonksiyonu çağıranın sorumluluğundadır.

## Yapı Bildirimi İle Değişken Tanımlamasının Birlikte Yapılması

Bir yapı bildirimi noktalı virgül ile kapatılmayıp, yapı bildiriminin kapanan blok parantezinden hemen sonra bir değişken listesi yazılırsa yapı bildirimi ile değişken tanımlaması bir arada yapılmış olur.

```
struct DATE {
    int day, month, year;
} bdate, edate;
```

Bu tanımlamada **struct** DATE türünden bir yapının bildirimi ile birlikte bu yapı türünden bdate ve edate değişkenleri de tanımlanmıştır. Bu durumda yapı değişkenlerinin faaliyet 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 bulunmamaktadır. 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;
```

şeklinde yapılırsa bile artık derleyici, eleman yapısı ve elemanların türleri aynı olsa bile bu yapıyı ayrı bir yapı türü olarak ele alacaktır. 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 fonksiyon yazılamayıdır. Çünkü yapı türünün bir ismi yoktur, ve bir fonksiyonun parametre değişkeni tür bilgisi olmadan tanımlanamaz.

## İççe Yapılar (nested structures)

Bir yapının elemanı başka bir yapı türünden yapı değişkeni ise bu duruma iççe yapılar denir. İççe yapı bildirimini iki biçimde yapılabilir :

1. Önce eleman olarak kullanılan yapı bildirilir. Onun aşağısında diğer yapı bildirimini yapılır. Örnek:

```
struct DATE {
    int day, month, year;
};

struct PERSON {
    char name[30];
    struct DATE birthdate;
    int no;
};
```

Bu durumda içteki yapıya ulaşmak için 2 tane nokta operatörü kullanmak gerekir.

```
int main()
{
    struct PERSON employee;

    employee.birthdate.day = 10;
    employee.birthdate.month = 12;
    employee.birthdate.year = 2000;
    strcpy(employee.name, "Necati Ergin");
    employee.no = 1472;
    printf("%d  %d  %d  -  %s  -  %d\n",    employee.birthdate.day,
    employee.birthdate.month,
    employee.birthdate.year, employee.name, employee.no);
    return 0;
}
```

2. Bu yöntemde eleman olan yapı değişken tanımlaması ile birlikte elemana sahip yapının içerisinde bildirilir.

Örnek:

```
struct PERSON {
    char name[30];
    struct DATE {
        int day, month, year;
    } birthdate;
    int no;
};
```

Burada içte bildirilen yapı da sanki dışarıda bildirilmiş gibi işlem görür. yani içeride bildirilen yapı türünden değişkenler tanımlanabilir. Burada dikkat edilmesi gereken bir noktada içiçe yapı bildiriminin yapılmasına rağmen bir değişken tanımlamasının yapılmamış olmasıdır. Yani birthdate bir nesne değildir. Ancak **struct** PERSON türünden bir değişken tanımlandığında, bu yapı değişkeninin bir alt elemanı olacaktır:

```
struct PERSON X;
```

```
X.birthdate.day = 20;
```

### İçiçe Yapılara İlk Değer Verilmesi

Normal olarak ilk değer vermede elemanlar sırasıyla, içteki yapı da dikkate alınacak biçimde, yapı üyelerine atanır. Ancak içteki yapının ayrıca küme parantezleri içerisine alınması okunabilirliği artırdığı için tavsiye edilir. Örnek:

```
struct PERSON per = {"Necati Ergin", {10, 10, 1967}, 123};
```

Eğer içteki yapı ayrıca küme parantezi içine alınmışsa içteki yapının daha az sayıda elemanına ilk değer vermek mümkün olabilir. Örnek :

```
struct PERSON per = {"Necati Ergin", {10, 10}, 123};
```

Burada doğum tarihinin year elemanına ilk değer verilmemiştir. Derleyici bu elemana otomatik olarak 0 değeri yerleştirecektir. Ancak burada içteki küme parantezleri kullanılmasaydı 123 ilk değeri year elemanına verilmiş olacak, no elemanına otomatik olarak 0 değeri verilmiş olacaktı. Aynı durum yapı içinde dizi bildirimlerinde de söz konusudur. Örnek :

```
struct SAMPLE {
    int a[3];
    long b;
    char c;
};
```

```
struct SAMPLE x = {{1, 2, 3}, 50000L, 'x'};
```

Aşağıdaki gibi bir bildirim de geçerlidir :

```
struct SAMPLE {
    int a[3];
    long b;
    char c;
} x = {{1, 2, 3}, 50000L, 'A', *p;
```

## İççe Yapı Kullanımının Avantajları

Aşağıdaki gibi bir yapı bildiriminin yapıldığını düşünelim :

```
struct ISIM {
    char ad[ADUZUNLUK + 1];
    char soyad[SADUZUNLUK + 1];
};
```

İsim yapısını daha büyük bir yapının parçası olarak kullanabiliriz.

```
struct OGRENCI {
    struct ISIM isim;
    int no, yas;
    char cinsiyet;
} ogrenci1, ogrenci2;
```

Bu durumda ogrenci1 degiskeninin ad ve soyad'ına ulaşmak için iki kez nokta operatörü kullanılacaktır:

```
strcpy(ogrenci1.isim.ad, "Hakan");
```

Yukarıdaki örnekte OGRENCI yapısının elemanı olarak ISIM yapısını kullanmak yerine, doğrudan ad[ADUZUNLUK + 1] ve char soyad[SADUZUNLUK + 1] dizilerini OGRENCI yapısının elemanları yapabildik. Ama isim yapısını doğrudan bir eleman olarak kullanırsak bazı işlemleri daha kolay yapabiliriz. Örneğin öğrencilerin ad ve soyadını yazan bir fonksiyon yazacağımızı düşünelim. Bu durumda yazılacak fonksiyona 2 arguman göndermek yerine yalnızca 1 arguman gönderilecektir.

```
isim_yaz(ogrenci1.isim);
```

Benzer şekilde isim türünden bir yapı elemanından öğrencinin ad ve soyad bilgilerini OGRENCI yapısı türünden bir değişkenin alt elemanlarına kopyalama daha kolay yapılacaktır :

```
struct isim yeni_isim;
...
ogrenci1.isim = yeni_isim;
```

## Yapı Dizileri

Yapılar da bir tür belirttiğine göre yapı türünden de diziler söz konusu olabilir. Yapı dizilerine de normal dizilere ilk değer verilmesine benzer şekilde ilk değer verilebilir. İlk değer verme sırasında kullanılan, içteki küme parantezleri okunabilirliği artırır. Örnek:

```
struct DATE {
    int day, month, year;
};

int main()
{
    struct DATE birthday[5] = {{10, 10, 1958}, {04, 03, 1964},
                               {21, 6, 1967}, {22, 8, 1956},
                               {11, 3, 1970}};

    struct DATE *pdate;
    int i;

    pdate = birthday;
    for (i = 0; i < 5; ++i) {
        printf("%02d / %02d / %04d\n", pdate->day, pdate->month, pdate->year);
        ++pdate;
    }
```

```
    return 0;
}
```

Program parçasını şu şekilde de yazabilirdik:

```
int main()
{
    ...
    for (i = 0; i < 5; ++i)
        disp(&birthdate[i]);
    return 0;
}

void Disp(struct DATE *p)
{
    printf("%02d / %02d / %04d\n", p->day, p->month, p->year);
}
```

## Yapılara İlişkin Karmaşık Durumlar

Bir yapının elemanı başka bir yapı türünden yapı göstericisi olabilir. Örneğin:

```
struct DATE {
    int day, month, year;
};

struct PERSON {
    char name[30];
    struct DATE *bdate;
    int no;
};
```

**struct** PERSON x; şeklinde bir tanımlamanın yapıldığını düşünelim.

1. x.bdate ifadesinin türü **struct** DATE türünden bir adrestir ve nesne belirtir. (Nesne belirtir çünkü bdate bir gösterici değişkendir.)
2. x.bdate->day ifadesinin türü **int** dir ve nesne belirtir.
3. &x.bdate->day ifadesinin türü **int** türden bir adrestir.

Tabi bu örnekte, bir değer ataması yapılmamışsa, x.bdate ile belirtilen gösterici içerisinde rasgele bir adres vardır. (garbage value) Bu göstericinin kullanılabilmesi için tahsis edilmiş bir alanı göstermesi gerekir. Örneğin bu alan dinamik olarak tahsis edilebilir.

x.bdate = (**struct** DATE \*) malloc (sizeof(**struct** DATE));

Yukarıdaki örnekte elimizde yalnızca **struct** PERSON türünden bir gösterici olduğunu düşünelim.

**struct** PERSON \*person;

1. person->bdate ifadesinin türü **struct** DATE türünden bir adrestir.
2. person->bdate->day ifadesinin türü **int** dir.

Bu örneklerde henüz hiçbir yer için bilinçli bir tahsisat yapılmamıştır. Hem person göstericisi için hem de person->date göstericisi için, dinamik bellek fonksiyonlarıyla bellekte yer tahsisatlarının yapılması gerekir:

```
person = (struct PERSON *) malloc(sizeof(struct PERSON));
person->bdate = (struct DATE *) malloc(sizeof(struct DATE));
```

Burada dinamik olarak tahsis edilen alanlar ters sırada iade (free) edilmelidir.

```
free(person->bdate);
free(person);
```

Bir yapının elemanı kendi türünden bir yapı değişkeni olamaz. Örneğin:

```
struct SAMPLE {
    struct SAMPLE a;
}; /* hata */
```

Çünkü burada SAMPLE yapısının uzunluğu belirlenemez. Ancak bir yapının elemanı kendi türünden bir gösterici olabilir. Örneğin:

```
struct LLIST {
    int val;
    struct LLIST *next;
};
```

Bu tür yapılar özellikle bağlı liste ve ağaç yapılarını (algoritmalarını) gerçekleştirmek amacıyla kullanılabilir. (Bu konu detaylı olarak bağlı listeler dersinde ele alınacaktır.)

## Bağlı Liste Nedir

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 yapı türünden bir göstericide tutulur. Böylece 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. Örnek:

```
struct LLIST {
    int val;
    struct LLIST *next;
};
```

Bağlı listenin ilk elemanının adresi global bir göstericide tutulabilir. Son elemanına ilişkin gösterici ise NULL adresi olarak bırakılır. Bağlı liste elemanları malloc gibi dinamik bellek fonksiyonları ile oluşturulur.

## Bağlı Listelerle Dizilerin Karşılaştırılması

Dizilerde herhangi bir elemana çok hızlı erişilebilir. Oysa bağlı listelerde bir elemana erişebilmek için, bağlı listede ondan önce yer alan bütün elemanları dolaşmak gerekir.

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.



## 26 . BÖLÜM : C DİLİNİN TARİH VE ZAMAN İLE İLGİLİ İŞLEM YAPAN STANDART FONKSİYONLARI

time_t time (time_t *timer); 01.01.1970 tarihinden itibaren geçen saniye sayısını bulur. Bu değer bazı tarih ve zaman fonksiyonlarında girdi olarak kullanılmaktadır.	geri dönüş değeri 01.01.1970 den çağırıldığı zamana kadar geçen saniye sayısıdır. Bu değeri aynı zamanda kendisine gönderilen argumandaki adrese de yazar. Arguman olarak NULL adresi gönderilirse saniye değerini yalnızca geri dönüş değeri olarak üretir.
struct tm localtime(const time_t *timer); localtime fonksiyonu çağırıldığı andaki sistem tarihinin ve zamanının elde edilmesi amacıyla kullanılır. 01.01.1970 tarihinden geçen saniye sayısını (yani time fonksiyonunun çıktısını) parametre olarak alır, bunu o andaki tarih ve zaman değerine dönüştürerek statik olarak tahsis edilmiş bir yapı içerisinde saklar.	Geri dönüş değeri struct tm türünden bir yapı adresidir. Bu yapı içinde ayrıştırılmış tarih ve zaman bilgileri bulunmaktadır.
char *ctime(const time_t *time); Bu fonksiyon time fonksiyonunun çıktısını parametre olarak alır, 26 karakterlik NULL ile biten bir diziye yerleştirerek dizinin başlangıç adresiyle geri döner. Tarih ve zaman bilgisinin dizi içerisindeki durumu şöyledir: DDD MMM dd hh:mm:ss YYYY Mon Oct 20 11:31:54 1975\n\0 DDD : İngilizce günlerin ilk üç harfinden elde edilen kısaltma MMM : İngilizce ayların ilk üç harfinden elde edilen kısaltma dd : Sayısal olarak ayın hangi günü olduğu hh : Saat mm : Dakika ss : Saniye YYYY : Yıl	fonksiyon kendi içerisinde statik olarak tahsis ettiği dizinin adresine geri döner. (Dolayısıyla hep aynı adrese geri dönmektedir.)
char *asctime (const struct tm *tblock); ctime fonksiyonuna benzer bir işlevi yerine getirir. Parametre olarak ctime fonksiyonundan farklı olarak struct tm türünden bir gösterici alır. Tıpkı ctime fonksiyonunda olduğu gibi tarihi ve zamanı 26 karakterlik bir diziye yerleştirir.	fonksiyon kendi içerisinde statik olarak tahsis ettiği dizinin adresine geri döner. (Dolayısıyla hep aynı adrese geri dönmektedir.)
clock_t clock(void) Programın başlangıcından itibaren geçen saat ticklerinin sayısını verir. clock fonksiyonu iki olay arasındaki zamansal farkı bulmak için kullanılır. Elde edilen zaman değerini saniyeye çevirmek için, bu değeri CLK_TCK sembolik sabitine bölmek gerekir.	Başarı durumunda programın başlangıcından beri geçen işlemci zamanı değerine geri döner. Başarısızlık durumunda işlemci zamanı elde edilemiyorsa, ya da değeri elde edilemiyorsa) -1 değerine geri döner.
double difftime(time_t time2, time_t time1); İki zaman arasındaki farkı saniye olarak hesaplar.	Geri dönüş değeri time1 ve time2 zamanları arasındaki saniye olarak farktır.

### Uygulama

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

```
void showtime(void);
void delay(unsigned long n);
double keeptime(void);
double chrono(void);

int main()
{
    int i = 10;

    clrscr();
    chrono();
    while (i-- > 0) {
        delay(3000000ul);
        printf("%lf\n", chrono());
    }
    return 0;
}

void showtime(void)
{
    time_t timer;
    struct tm *tptr;

    while (!kbhit() || getch() != 27) {
        time(&timer);
        tptr = localtime(&timer);
        printf("%02d:%02d:%02d\r", tptr->tm_hour, tptr->tm_min, tptr->tm_sec);
    }
}

double keeptime(void)
{
    static int flag = 0;
    static clock_t start, end;

    if (!flag) {
        start = clock();
        flag = 1;
        return -1.;
    }
    end = clock();
    flag = 0;
    return (end - start) / CLK_TCK;
}

double chrono(void)
{
    static int flag = 0;
    static clock_t start, end;

    if (flag == 0) {
        start = clock();
        flag = 1;
        return -1.;
    }
    end = clock();
    return (end - start) / CLK_TCK;
}
```

```
void delay(unsigned long n)
{
    while (n-- > 0)
        ;
}
```

## Uygulama

Tarihler ile ilgili İşlem Yapan fonksiyonlar.

```
/* INCLUDE FILES */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>

/* symbolic constants */

#define VALID      1
#define INVALID   0

#define FALSE      0
#define TRUE       1

#define SUN        0
#define MON        1
#define TUE        2
#define WED        3
#define THU        4
#define FRI        5
#define SAT        6

/* type definitions */

typedef int BOOL;
typedef int DAYS;

typedef struct DATE {
    int day, month, year;
}DATE;

/* function prototypes */

void setDate(int day, int month, int year, DATE *date);
void displayDate(const DATE *date);
void displayDate2(const DATE *date);

long totaldays(const DATE *date);
int dayofyear(const DATE *date);
int Datecmp(const DATE *date1, const DATE *date2);
int datedif(const DATE *date1, const DATE *date2);

BOOL isleap(int year);
BOOL validDate(const DATE *date);
BOOL isweekend(const DATE *date);

DAYS Dateday(const DATE *date);

DATE nDate(const DATE *date, int n);
DATE randomDate(void);
DATE totaltoDate(long totaldays);
DATE randomDate(void);
```

```
/* global variables */

char *days[] = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};

char *months[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
                  "Aug", "Sep", "Oct", "Nov", "Dec"};

/* function definitions */

int main()
{
    /* type your test code here */
}

/* Gönderilen argumanın artık yıl olup olmadığını test eder. Artık yıl ise 1
degil ise 0 değerine geri döner. */

BOOL isleap(int year)
{
    if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
        return TRUE;
    return FALSE;
}

/* Gönderilen adresteki DATE türünden yapı nesnesine ilk 3 argumanındaki ay,
gün, yıl değerlerini yerleştirir. */

void setDate(int day, int month, int year, DATE *date)
{
    date->day = day;
    date->month = month;
    date->year = year;
}

/* Gönderilen adresteki yapı nesnesinin elemanlarını ekrana yazdırır. */

void displayDate(const DATE *date)
{
    printf("%02d %02d %4d", date->day, date->month, date->year);
}

/* gönderilen adresteki yapı nesnesinin sakladığı tarihi ingilizce format ile
ekrana yazar. Örnek:
3 rd Sep 2000 Sunday */

void displayDate2(const DATE *date)
{
    printf("%02d", date->day);
    switch (date->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 ");
    }
    printf("%s ", months[date->month - 1]);
    printf("%d ", date->year);
    printf("%s", days[Dateday(date)]);
}
```

```
/* 01.01.1900 tarihinden adresi verilen tarihe kadar geçen gün sayısına geri döner. */

long totaldays(const DATE *date)
{
    long result = 0;
    int year;

    for (year = 1900; year < date->year; ++year)
        result += 365 + isleap(year);
    result += dayofyear(date);
    return result;
}

/* adresi gönderilen yapı nesnesindeki tarihin ilgili yılın kaçınıcı günü olduğunu bulur. */

int dayofyear(const DATE *date)
{
    int result = date->day;

    switch (date->month - 1) {
        case 11: result += 30;
        case 10: result += 31;
        case 9: result += 30;
        case 8: result += 31;
        case 7: result += 31;
        case 6: result += 30;
        case 5: result += 31;
        case 4: result += 30;
        case 3: result += 31;
        case 2: result += 28 + isleap(date->year);
        case 1: result += 31;
    }
    return result;
}

/* Adresleri gönderilen yapı nesnelerinde saklanan tarihler arasındaki gün farkını bulur. */

int datedif(const DATE *date1, const DATE *date2)
{
    return abs(totaldays(date1) - totaldays(date2));
}

/* Parametre değişkenindeki gün sayısını DATE türünden bir nesneye dönüştürerek geri dönüş değeri olarak üretir. */

DATE totaltoDate(long totaldays)
{
    int months[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    DATE result;
    int i = 0;

    result.year = 1900;

    while (totaldays > (isleap(result.year) ? 366 : 365)) {
        totaldays -= isleap(result.year) ? 366 : 365;
        result.year++;
    }
    months[1] += isleap(result.year);
    while (totaldays > months[i]) {
        totaldays -= months[i];
        ++i;
    }
}
```

```

    }
    result.month = i + 1;
    result.day = totaldays;
    return result;
}

/* Adresi gönderilen yapı nesnesindeki tarihten n gün sonraki ya da önceki
tarihin ne olduğunu hesaplayarak bu tarihe geri döner. */

DATE nDate(const DATE *date, int n)
{
    return totaltoDate(totaldays(date) + n);
}

/* adresi gönderilen yapı nesnesindeki tarihin hangi gün olduğunu hesap eder.
*/

DAYS Dateday(const DATE *date)
{
    switch (totaldays(date) % 7) {
        case 0: return SUN;
        case 1: return MON;
        case 2: return TUE;
        case 3: return WED;
        case 4: return THU;
        case 5: return FRI;
        case 6: return SAT;
    }
    return 0;
}

/* Adresleri gönderilen yapı nesnelerindeki tarihleri karşılaştırır. */

int Datecmp(const DATE *date1, const DATE *date2)
{
    if (date1->year != date2->year)
        return date1->year - date2->year;
    if (date1->month != date2->month)
        return date1->month - date2->month;
    if (date1->day != date2->day)
        return date1->day - date2->day;
    return 0;
}

/* Adresi gönderilen yapı adresindeki tarihin geçerli bir tarih olup olmadığını
test eder. */

BOOL validDate(const DATE *date)
{
    int months[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    months[1] = 28 + isleap(date->year);

    if (date->year < 1900)
        return INVALID;
    if (date->day > months[date->month - 1] || date->day <= 0)
        return INVALID;
    if (date->month < 1 || date->month > 12)
        return INVALID;
    return VALID;
}

/* Tarihin hafta sonu olup olmadığını test eder. */

BOOL isweekend(const DATE *date)

```

```
{
    DAYS result = Dateday(date);

    if (result == SAT || result == SUN)
        return TRUE;
    return FALSE;
}

/* 01.01. 1900 ve 31.12.2000 tarihleri arasında rasgele bir tarih üretir. */
DATE randomDate()
{
    int months[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

    DATE random;

    random.year = rand() % 101 + 1900;
    months[1] += isleap(random.year);
    random.month = rand() % 12 + 1;
    random.day = rand() % months[random.month - 1] + 1;
    return random;
}
```

## Uygulama

Tekli bağlı liste oluşturulmasına ilişkin bir program.

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

#define ADD_BEGIN          1
#define ADD_END            2
#define DELETE_NAME       3
#define DELETE_NO         4
#define DISPLAY            5
#define EXIT_PROG         6

typedef struct _PERSON {
    char name[20];
    int no;
    struct _PERSON *next;
} PERSON;

PERSON *start = NULL;

void add_begin(void);
void display_list(void);
void add_end(void);
void delete_name(void);
void delete_no(void);
int get_option(void);

int main()
{
    int option;

    for (;;) {
        option = get_option();
        switch (option) {
            case ADD_BEGIN : add_begin(); break;
            case ADD_END   : add_end(); break;
```

```

        case DELETE_NAME : delete_name(); break;
        case DELETE_NO : delete_no(); break;
        case DISPLAY      : display_list(); break;
        case EXIT_PROG    : goto EXIT;
        default            : printf("geçersiz seçenek!..\n");
    }
}
EXIT:
    exit(EXIT_SUCCESS);
return 0;
}

/*****/
int get_option(void)
{
    int option;

    printf("\n[1] bağlı listenin başına ekle.\n");
    printf("[2] bağlı listeye sondan ekle.\n");
    printf("[3] bağlı listeden isme göre sil.\n");
    printf("[4] bağlı listeden no'ya göre sil.\n");
    printf("[5] bağlı listeyi listele.\n");
    printf("[6] programdan çık.\n");
    option = getch() - '0';
    return option;
}

/*****/
void add_begin(void)
{
    PERSON *new;

    new = (PERSON *)malloc (sizeof(PERSON));
    if (new == NULL) {
        printf("cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
    fflush(stdin);
    printf("ismi giriniz : ");
    gets((char *)new);
    printf("No'yu giriniz : ");
    scanf("%d", &new->no);
    new->next = start;
    start = new;
}

/*****/
void add_end(void)
{
    PERSON *new, *cur;

    new = (PERSON *)malloc(sizeof(PERSON));
    if (new == NULL) {
        printf("cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
    fflush(stdin);
    printf("ismi giriniz : ");
    gets((char *)new);
    printf("No'yu giriniz : ");
    scanf("%d", &new->no);
    if (start == NULL) {
        start = new;
        new->next = NULL;
    }
}

```



```
        return;
    }
    for (cur = start; cur->next != NULL; cur = cur->next)
        ;
    cur->next = new;
    new->next = NULL;
}

/*****/
void display_list(void)
{
    PERSON *cur;

    printf("\n bilgiler listeleniyor : \n\n");
    for (cur = start; cur != NULL; cur = cur->next)
        printf("%-20s\t%05d\n", cur->name, cur->no);
}

/*****/
void delete_name(void)
{
    PERSON *cur, *prev;
    char nametemp[30];

    printf("silinecek ismi girin : ");
    fflush(stdin);
    gets(nametemp);

    for (prev = NULL, cur = start; cur != NULL && strcmp(cur->name, nametemp);
        prev = cur, cur = cur->next)
        ;
    if (cur == NULL) {
        printf("silinecek kay^t bulunamad^ \n");
        return;
    }
    if (prev == NULL)
        start = start->next;
    else
        prev->next = cur->next;
    free(cur);
    printf("kayıt silindi! \n");
}

/*****/
void delete_no(void)
{
    PERSON *cur, *prev;
    int no;

    printf("silinecek No'yu girin : ");
    scanf("%d", &no);

    for (prev = NULL, cur = start; cur != NULL && cur->no != no;
        prev = cur, cur = cur->next)
        ;
    if (cur == NULL) {
        printf("kayıt bulunamadı \n");
        return;
    }
    if (prev == NULL)
        start = start->next;
    else
        prev->next = cur->next;
    free(cur);
}
```

```
    printf("kayıt silindi!..\n");
}
Uygulama
Sıralı bağlı liste oluşturulmasına ilişkin bir program.
```

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

typedef struct _PERSON{
    char name[20];
    int no;
    struct _PERSON *next;
}PERSON;

PERSON *start = NULL;

int get_option(void);
void add(void);
void display(void);
void delete_name(void);
void fill(void);
PERSON *find_name(char *name);
void del_list(void);
void search(void);

int main()
{
    int option;
    for (;;) {
        option = get_option();
        switch (option) {
            case 1: add(); break;
            case 2: display(); break;
            case 3: delete_name(); break;
            case 4: fill(); break;
            case 5: search(); break;
            case 6: del_list(); break;
            case 0: goto EXIT;
            default: printf("make your choice\n");
        }
    }
    EXIT:
    return 0;
}

/*****/
int get_option(void)
{
    int option;

    printf("\n[1] add a new name\n");
    printf("[2] display list\n");
    printf("[3] delete name\n");
    printf("[4] fill the link list with test data\n");
    printf("[5] find name\n");
    printf("[6] delete list\n");
    printf("[0] exit\n");
    option = getch() - '0';
    return option;
}
```

```
/*******/
void add(void)
{
    PERSON *prev, *cur, *_new;
    char name[50];
    fflush(stdin);
    printf("name:");
    gets(name);

    for (prev = NULL, cur = start; cur != NULL && strcmp(name, cur->name) > 0;
        prev = cur, cur = cur->next)
        ;
    if (cur != NULL && !strcmp(name, cur->name)) {
        printf("name already exists\n");
        return;
    }

    if ((_new = (PERSON *) malloc(sizeof(PERSON))) == NULL) {
        printf("not enough memory");
        exit(EXIT_FAILURE);
    }
    strcpy(_new->name, name);
    printf("number:");
    scanf("%d", &_new->no);
    _new->next = cur;

    if (prev == NULL)
        start = _new;
    else
        prev->next = _new;
}
/*******/
void display(void)
{
    PERSON *cur;

    putchar('\n');

    for (cur = start; cur != NULL; cur = cur->next)
        printf("%-20s %d\n", cur->name, cur->no);
    printf("press any key to continue\n");
    getch();
}

/*******/
void delete_name(void)
{
    PERSON *cur, *prev;
    char name[50];
    fflush(stdin);
    printf("name: ");
    gets(name);

    for (prev = NULL, cur = start;
        cur != NULL && strcmp(cur->name, name);
        prev = cur, cur = cur->next)
        ;
    if (cur == NULL) {
        printf("couldn't find\n");
        return;
    }
    if (prev == NULL)
        start = cur->next;
    else
```

```

        prev->next = cur->next;
        printf("%s was deleted\n", name);
        free(cur);
    }

/*****/
void fill(void)
{
    int i;
    PERSON *_new, *prev, *cur;
    PERSON a[7] = {{ "ali",      -3, NULL}, {"erdem",    7,  NULL},
{"kaan",    -1,    NULL}, {"necati",  5,  NULL},
{"gurbuz",  9,  NULL}, {"damla",   20, NULL},
{"metin",   15, NULL} };

    for (i = 0; i < 7; i++) {
        if ((_new = (PERSON *) malloc(sizeof(PERSON))) == NULL) {
            printf("not enough memory");
            exit(EXIT_FAILURE);
        }
        *_new = a[i];

        for (prev = NULL, cur = start;
             cur != NULL && strcmp(_new->name, cur->name) > 0;
             prev = cur, cur = cur->next)
            ;

        if (cur != NULL && !strcmp(_new->name, cur->name)) {
            printf("name already exists\n");
            return;
        }
        _new->next = cur;

        if (prev == NULL)
            start = _new;
        else
            prev->next = _new;
    }
}

/*****/
void search(void)
{
    char name[50];
    PERSON *p;

    fflush(stdin);
    printf("name: ");
    gets(name);

    if ((p = find_name(name)) == NULL)
        printf("could not find\n");
    else
        printf("name: %-20s number: %d\n", p->name, p->no);
}

/*****/
void del_list(void)
{
    PERSON *cur = start, *prev;

    while (cur != NULL) {
        prev = cur;
        cur = cur->next;
    }
}

```

```
        free(prev);
    }
    start = NULL;
}

/*****/
PERSON *find_name(char *name)
{
    PERSON *p;

    for (p = start; p != NULL && strcmp(name, p->name) > 0; p = p->next)
        ;
    if (p != NULL && !strcmp(name, p->name))
        return p;
    return NULL;
}
```

## 27 . BÖLÜM : TÜR TANIMLAMALARI ve typedef ANAHTAR SÖZCÜĞÜ

C dilinde bir türe her bakımdan onun yerini tutabilen alternatif isimler verilebilir. Bu işlemler **typedef** anahtar sözcüğü ile yapılır.

Genel biçimi:

**typedef** <isim> <yeni isim>;

Örnek :

**typedef unsigned int** UINT;

Bu bildirimden sonra UINT ismi derleyici tarafından **unsigned int** türünün yeni bir ismi olarak ele alınacaktır. Yani kaynak kod içerisinde UINT sözcüğü kullanıldığında derleyici bunu **unsigned int** olarak anlamlandıracaktır.

UINT x, y, k;

Bildiriminde artık x, y, k değişkenleri **unsigned int** türünden tanımlanmış olurlar.

printf("%d\n", **sizeof**(UINT));

**typedef** anahtar sözcüğü ile yeni bir tür ismi yaratılması 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 önilemci komutuyla da aynı iş yapılabilirdi.

#define UINT**unsigned int**

Ancak **typedef** anahtar sözcüğü derleyici tarafından ele alınırken #define ile tanımlanan sembolik sabitler önilemciyi ilgilendirmektedir.

Karışık tür tanımlamalarında aşağıdaki kural sentaks açısından hata yapılmasını önelleyecektir.

**typedef** anahtar sözcüğü, her tür bildirimin önüne gelir. **typedef** anahtar sözcüğü bir bildirimin önüne geldiğinde **typedef** kullanılmamış olsaydı nesne olacak isimler, **typedef** anahtar sözcüğü eklendiğinde artık tür ismi olurlar. Örnekler:

**char** \* pstr;

bildirimi ile pstr **char** türden bir gösteri olur.

**typedef char** \*pstr;

bir tür tanımlamasıdır ve artık pstr bir tür olarak ele alınır. pstr **char** türden bir adres türünün başka bir ismi olarak, geçerli bir türdür.

Yani yukarıdaki tür tanımlamasından sonra

pstr p;

gibi bir bildirim yapılabilir. Bildirim **char** \*p ile aynı anlama gelir.

Bu örnekte yapılan tür tanımlaması #define önişlemci komutuyla, yani sembolik sabit tanımlamasıyla yapıldığında kodda yanlışlık olabilirdi.

```
#define pstr char *
```

gibi bir sembolik sabit tanımlaması yapıldığında önişlemci pstr gördüğü yerde bunun yerine **char** \* yerleştirecektir.

```
char *str;
```

gibi bir bildirimin

```
pstr str;
```

olarak yazılmasında bir hata söz konusu olmaz çünkü önişlemci pstr yerine **char** \* yerleştirdiğinde derleyiciye giden kod **char** \*str haline gelir.

```
char *p1, *p2, *p3;
```

pstr önişlemci sabitini kullanarak yukarıdaki gibi bir bildirimin yapılmak istendiğini düşünelim.

```
pstr p1, p2, p3;
```

yazıldığında, önişlemci yer değiştirme işlemini yaptıktan sonra derleyiciye verilen kod aşağıdaki biçime dönüşecektir :

```
char *p1, p2, p3;
```

Bu tanımlama yapılmak istenen tanımlamaya eşit değildir. Yukarıdaki bildirimde yalnızca p1 **char** türden bir göstericidir. p2 ve p3 **char** türden göstericiler değil **char** türden nesnelerdir. Bu örnekte görüldüğü gibi, **typedef** anahtar sözcüğünün işlevi #define önişlemci komutuyla her zaman yerine getirilemiyor.

Bir diziye ilişkin tür tanımlaması da yapılabilir.

```
char isimdizi[20];          /* isimdizi char türden 20 elemanlı bir dizidir. */  
typedef isimdizi[20]; /* isimdizi 20 elemanlı char türden bir dizi türüdür. */
```

```
isimdizi a, b, c;          /* a, b ve c herbiri 20 elemanlı char türden dizilerdir. */
```

Bir **typedef** ile birden fazla tür ismi tanımlanabilir.

```
typedef unsigned int WORD, UINT;
```

Bu tür tanımlamasıyla hem WORD hem de UINT **unsigned int** türünün yerine geçebilir.

```
WORD x, y;
```

```
...
```

```
UINT k, l;
```

10 elemanlı **char** türden gösterici dizisi için tür tanımlamasını şöyle yapabiliriz:

```
typedef char *PSTRARRAY[10];
```

Bu tür tanımlamasından sonra

```
PSTRARRAY s;
```

ile

```
char *s[10];
```

tamamen aynı anlama gelecektir.

Bir tür ismi başka tür isimlerinin tanımlanmasında da kullanılabilir.

```
typedef unsigned int WORD;  
typedef WORD UINT;
```

Tür isimleri geleneksel olarak büyük harfle yazılır ama bu C sentaksı açısından bir zorunluluk değildir.

## **typedef Tür Tanımlamalarının Yapılarla Birlikte Kullanımı**

C dilinde yapılar ayrı bir veri türüdür. Bu tür önce derleyiciye tanıtıldıktan sonra, bu türe ilişkin değişkenler (nesneler) tanımlanabilir. Örneğin:

```
struct SAMPLE {  
    int a, b, c;  
};
```

Yukarıdaki örnekte yeni bir veri türü yaratılmıştır. Bu veri türünün ismi **struct** SAMPLE dir. (SAMPLE değil) Yani bu veri türünden bir nesne tanımlamak istersek tür ismi olarak **struct** SAMPLE yazılmalıdır. (Oysa C++ dilinde yapı isimleri (structure tags) aynı zamanda türün de ismidir ve **struct** anahtar sözcüğü olmadan kullanıldığında bu türün ismi olarak derleyici tarafından kabul görür.)

Yukarıdaki bildirimden sonra örneğin bir tanımlama yapılacak olsa

```
struct SAMPLE x;
```

şeklinde yapılmalıdır. C dilinde bu tanımlamanın

```
SAMPLE x;
```

şeklinde yapılması derleme zamanında hata oluşturacaktır. (Oysa C++ dilinde bu durum bir hata oluşturmaz.)

İşte **struct** anahtar sözcüğünün, yapı tanımlamalarında yapı isminden önce kullanılma zorunluluğunu ortadan kaldırmak için programcılar, **typedef** anahtar sözcüğüyle kendi bildirdikleri yapı (birlik, bit alanları, enum türleri ) türlerine ilişkin yeni tür isimleri oluştururlar.

Bir yapı türüne ilişkin yeni bir tür isminin tanımlanması 2 ayrı şekilde yapılabilir.

1. Önce yapı bildirimini yapmak, daha sonra bu yapı türünden bir tür ismi tanımlamak.

```
struct PERSON {  
    char name[30];  
    int no;  
};  
  
typedef struct PERSON PER;  
  
int main()  
{  
    PER x = {"Necati Ergin", 123};  
  
    printf("%s%d\n", x.name, x.no);
```



```
return 0;
}
```

2. **typedef** bildirimi ile yapı bildirimini bir arada yapmak.

```
typedef struct PERSON {
    char name[30];
    int no;
} PER;
```

```
PER x;
```

Yapı ismini (tag) hiç belirtmeden de typedef bildirimi yapılabilirdi.

```
typedef struct {
    char name[30];
    int no;
} PER;
```

```
PER y;
```

Ancak yukarıdaki son durumda artık değişken bildiriminin yeni tanımlanan tür ismiyle yapılması zorunluluk haline gelir.

Programcıların çoğu yapı isimleriyle (tag) tür isimleri için farklı isimler bulmak yerine birkaç karakter kullanarak aralarında ilişki kurarlar. Çok kullanılan yöntemlerden biri, yapı isminin başına bir alt tire konularak tür isminden ayrılmasıdır :

```
typedef struct _EMPLOYEE {
    int no;
    char name[30];
    double fee;
} EMPLOYEE;
```

(Yukarıdaki örnekte tamamen aynı isim [mesela EMPLOYEE] hem yapı ismi hem de yeni tür ismi olarak kullanılabilir. Bu durum derleme zamanında bir hataya yol açmaz. )

```
typedef struct EMPLOYEE {
    int no;
    char name[30];
    double fee;
} EMPLOYEE;
```

Çünkü türün eski ismi EMPLOYEE değil **struct** EMPLOYEE dir. Yani **struct** EMPLOYEE türünün ismi EMPLOYEE olarak **typedef** edilmiştir ki bu durum bir sakınca oluşturmaz.

windows.h içerisinde yapı ismi ile yeni tür isminin, yapı ismine tag sözcüğü eklenerek birbirinden ayrıldığını görüyoruz.

```
typedef struct tagEMPLOYEE {
    int no;
    char name[30];
    double fee;
} EMPLOYEE;
```

## Başlık Dosyalarında Standart Olarak Bulunan Bazı typedef Bildirimleri

Temel C Kurdu boyunca bildirimi bizim tarafımızdan yapılmamış olan bazı tür isimlerini kullandık. typedef anahtar sözcüğü ile ilgili öğrendiklerimizden sonra bu tür isimlerinin ne

olduklarını daha iyi anlayabiliriz. `size_t` tür ismi ile dinamik bellek fonksiyonları ile çalışırken karşılaşmıştık. Bu tür isminin bildirimi `stdlib.h`, `alloc.h` de diğer bazı başlık dosyaları içinde aşağıdaki şekilde yapılmıştır.

```
typedef unsigned size_t;
```

Bu tür aslında gerçekte ne olduğu derleyicileri yazanlara bırakılmış olan bir türdür. `size_t` türü **sizeof** operatörünün ürettiği değerin türüdür. ANSI standartlarında bir çok fonksiyonun parametrik yapısı içerisinde `size_t` türü geçer. Örneğin `malloc` fonksiyonunun gerçek prototipi `alloc.h`, `mem.h` başlık dosyalarının içinde

```
void *malloc (size_t size);
```

biçiminde yazılmıştır.

Yani `malloc` fonksiyonunun parametresi `size_t` türündendir ama bu türün gerçekte ne olduğu derleyicileri yazanlara bırakılmıştır. Ancak hemen hemen bütün derleyicilerde `size_t` **unsigned int** olarak belirlenmiştir.

`size_t` türü gibi aslında ne oldukları derleyiciye bırakılmış olan (yani derleyici yazanların **typedef** bildirimlerini yapacakları) başka tür isimleri de C standartları tarafından tanımlanmıştır.

`size_t` (`sizeof` operatörünün ürettiği değerin türü.) Derleyiciyi yazanlar **unsigned int** ya da **unsigned long** türüne **typedef** ederler. (Bir zorunluluk olmasa da derleyicilerin hemen hemen hepsinde **unsigned int** türü olarak tanımlanmıştır.)

`time_t` (`time` fonksiyonunun geri dönüş değerinin türü, derleyiciyi yazanlar herhangi bir scalar (birleşik olmayan) default veri türüne **typedef** edebilirler. (Bir zorunluluk olmasa da derleyicilerin hemen hemen hepsinde **long** türü olarak tanımlanmıştır.)

`clock_t` (`clock` fonksiyonunun geri dönüş değerinin türü, derleyiciyi yazanlar herhangi bir skalar (bileşik olmayan) default veri türüne **typedef** edebilirler. (Bir zorunluluk olması da derleyicilerin hemen hemen hepsinde **long** türü olarak tanımlanmıştır.)

`ptrdiff_t` (gösterici değerlerine ilişkin fark değeri türü, (Bir zorunluluk olması da derleyicilerin hemen hemen hepsinde **int** türü olarak tanımlanmıştır.)

## **typedef ile Bildirimi Yapılan Tür İsimlerinin Faaliyet Alanları**

**typedef** isimleri için de faaliyet alanı kuralları geçerlidir. Blok içerisinde tanımlanan bir **typedef** ismi blok dışında tanınmaz.

```
func()
{
    typedef int WORD;
    ...
}

int main()
{
    WORD x; /* hata */
    ...
    return 0;
}
```

Yukarıdaki programda

```
WORD x;
```

tanımlamasını gören derleyici bir hata mesajı üretecektir. Zira WORD türü yalnızca func fonksiyonunun ana bloğu içinde tanınan ve anlamlandırılan bir veri türüdür. Bu bloğun dışında tanınmamaktadır.

C dilinde blok içinde yapılan bildirimler, blokların ilk işlemi olmak zorunda olduğundan **typedef** anahtar sözcüğüyle blok içinde yapılan tür bildirimleri de blokların ilk işlemi olmak zorundadır. Aksi halde derleme zamanında hata oluşacaktır.

Hemen her zaman **typedef** bildirimleri global olacak yapılmaktadır. Uygulamalarda **typedef** bildirimleri genellikle, ya kaynak dosyanın başında ya da başlık dosyaları içinde tanımlanır.

Bir **typedef** isminin aynı faaliyet alanı içinde (örneğin global düzeyde) özdeş olmayacak şekilde iki kere tanımlanması derleme zamanında hata oluşturacaktır :

```
typedef int WORD;
...
typedef long WORD; /* error */
```

### **typedef tanımlamaları ne amaçla kullanılır**

1. Okunabilirliği artırmak için. Bazı türlere onların kullanım amaçlarına uygun isimler verilirse kaynak kodu inceleyen kişiler kodu daha kolay anlamlandırır. Örneğin **char** türü genelde karakter sabitlerinin atandığı bir türdür. Yazılacak bir kodda **char** türü yalnızca bir byte'lık bir alan olarak kullanılacaksa, yani karakter işlemlerinde kullanılmayacaksa aşağıdaki gibi bir tür tanımlaması yerinde olacaktır.

```
typedef char BYTE;
...
BYTE x;
```

2. Yazım kolaylığı sağlar. Karmaşık pek çok tür ifadesi **typedef** sayesinde kolay bir biçimde yazılabilmektedir. Programı inceleyen kişi karmaşık operatörler yerine onu temsil eden yalın bir isimle karşılaşır.

```
typedef struct PERSON EMPLOYEE[100];
...
EMPLOYEE manager;
```

3. Tür tanımlamaları taşınabilirliği artırmak amacıyla da kullanılırlar. Tür tanımlamaları sayesinde kullandığımız fonksiyonlara ilişkin veri yapıları değişse bile kaynak programın değişmesi gerekmez. Örneğin, kullandığımız kütüphanede birtakım fonksiyonların geri dönüş değerleri **unsigned int** olsun. Daha sonraki uygulamalarında bunun **unsigned long** yapıldığını düşünelim. Eğer programcı bu bu fonksiyonlara ilişkin kodlarda tür tanımlaması kullanmışsa daha önce yazdığı kodları değiştirmesine gerek kalmaz, yalnızca tür tanımlamasını değiştirmek yeterlidir. Örneğin:

```
typedef unsigned int HANDLE;
...
HANDLE hnd;
hnd = GetHandle();
```

Burada GetHandle fonksiyonunun geri dönüş değerinin türü sonraki uyarlamalarda değişerek **unsigned long** yapılmış olsun. Yalnızca tür tanımlamasının değiştirilmesi yeterli olacaktır:

```
typedef unsigned long HANDLE;
```

Başka bir örnek :

Bir C programında değerleri 0 ile 50000 arasında değişebilecek olan sayaç değişkenler kullanacağımızı düşünelim. Bu amaç için **long int** türünü seçebiliriz, çünkü **long int** türü bilindiği gibi 2.147.483.647 ye kadar değerleri tutabilir. Ama **long** türü yerine **int** türünü kullanmak, aritmetik işlemlerin daha hızlı yapılabilmesi açısından tercih edilecektir. Ayrıca **int** türden olan değişkenler bellekte daha az yer kaplayabilecektir.

**int** türünü kullanmak yerine bu amaç için kendi tür tanımlamamızı yapabiliriz :

```
typedef int SAYAC;
```

```
SAYAC a, b, c;
```

Kaynak kodun **int** türünün 16 bit uzunluğunda olduğu bir sistemde derlenmesi durumunda tür tanımlama ifadesini değiştirebiliriz:

```
typedef long SAYAC;
```

Bu teknikle taşınabilirliğe ilişkin bütün problemlerimizin çözülmüş olacağını düşünmemeliyiz. Örneğin SAYAC türünden bir değişken printf ya da scanf fonksiyonu çağırımlarında arguman olarak kullanılmış olabilir :

```
typedef int SAYAC;
```

```
...
SAYAC a, b, c;
...
scanf("%d%d%d", &a, &b, &c);
...
printf("%d %d %d", a, b, c);
```

Yukarıdaki ifadelerde a. b. c değişkenleri SAYAC türünden (**int** türden) tanımlanmışlardır. printf ve scanf fonksiyonlarının çağırma ifadelerinde de bu değişkenlere ilişkin format karakterleri doğal olarak %d seçilmiştir. Ancak SAYAC türünün **long** türü olarak değiştirilmesi durumunda printf ve scanf fonksiyonlarında bu türden değişkenlerin yazdırılmasında kullanılan format karakterlerinin de %d yerine %ld olarak değiştirilmesi gerekecektir.

## **typedef İle Gösterici Türlerine İsim Verilmesi**

typedef anahtar sözcüğü ile gösterici türlerine isim verilmesinde dikkat etmemiz gereken bir nokta var. Belirleyiciler (specifiers) konusunda incelediğimiz gibi C dilinde aşağıdaki gibi bir tanımlama yapıldığında

```
const int *ptr;
```

ptr göstericisinin gösterdiği yerdeki nesne değiştirilemez. Yani

```
*ptr = 10;
```

gibi bir atama yapılması durumunda derleme zamanında hata oluşacaktır.

Ancak tanımlama

```
int *const ptr;
```

şeklinde yapılırsa ptr göstericisinin gösterdiği yerdeki nesnenin değeri değiştirilebilir, ama ptr göstericisinin içindeki adres değiştirilemez, yani

```
ptr = (int *) 0x1F00;
```

gibi bir atama yapılması durumunda derleme zamanında hata oluşur.

```
typedef int * IPTR;
```

gibi bir tür tanımlanarak

```
const IPTR p;
```

şeklinde bir tanımlama yapıldığında, p göstericisinin değeri değiştirilemez, p göstericisinin gösterdiği yerdeki nesnenin değeri değiştirilebilir. (yani \*p nesnesine atama yapılabilir.) Başka bir deyişle

```
const IPTR p;
```

deyimi ile

```
int *const ptr;
```

deyimi eşdeğerdir.

Windows işletim sistemi altında çalışacak C ya da C++ programlarının yazılmasında **typedef** sıklıkla kullanılmaktadır. windows.h isimli başlık dosyasında temel veri türlerinin çoğuna **typedef**'le yeni isimler verilmiştir. Windows programlamada Windows.h dosyası kaynak koda dahil edilmelidir. Bu dosyanın içerisinde API fonksiyonlarının prototipleri, çeşitli yapı bildirimleri, **typedef** isimleri, önemli sembolik sabitler bulunmaktadır.

## windows.h İçerisinde Tanımlanan Typedef İsimleri

```
typedef int BOOL
```

Bu türle ilişkili 2 sembolik sabit de tanımlanmıştır.

```
#define FALSE 0
#define TRUE 1
```

BOOL türü, özellikle fonksiyonların geri dönüş değerlerinde karşımıza çıkar. Bu durum fonksiyonun başarılıysa 0 dışı bir değere, başarısızsa 0 değerine geri döneceği anlamına gelir. Başarı kontrolü, 1 değeriyle karşılaştırılarak yapılmamalıdır.

Aşağıdaki **typedef** isimleri, işaretli 1 byte, 2 byte ve 4 byte tam sayıları temsil eder.

```
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 kullanılmaktadır. Win16 sistemlerinde uzak ve yakın gösterici kavramları vardı. Dolayısıyla o zamanlar, P ön ekli 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 kullanılmaktadır. Windows.h içerisinde 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** türden adres 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 LP örnekleriyle 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 versiyonları vardır.

```
typedef long LONG;  
typedef int INT;  
typedef char CHAR;
```

Windows programlamada H ile başlayan, handle olarak kullanılan pek çok **typede** ismi vardır. Bu **typedef** isimlerinin hepsi **void \*** türündendir. Örneğin:

```
typedef void *HWND;  
typedef void *HICON;
```

## 28 . BÖLÜM : BİRLİKLER

Programcı tarafından tanımlanan (user defined) ve bileşik bir veri yapısı olan birlikler yapılara çok benzer. Birliklerin de 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. Birlik bildirimlerinin genel şekli şöyledir:

```
union [birlik ismi] {
    tür <birlik elemanı>;
    tür <birlik elemanı>;
    tür <birlik elemanı>;
    ...
};
```

Örneğin;

```
union DWORD {
    unsigned char byte;
    unsigned int word;
    unsigned long dword;
};
```

Diğer bir örnek:

```
union DBL_FORM {
    double n;
    unsigned char s[8];
};
```

### Birlik Değişkenlerinin Tanımlanması

Birlik değişkenlerinin tanımlanması aynı yapı değişkenlerinde olduğu gibidir. Tıpkı yapılarda olduğu gibi birliklerde de bellekte yer ayırma işlemi bildirim ile değil tanımlama işlemi ile yapılmaktadır. Birlik değişkeni tanımlamalarının yapı bildirimi tanımlamalarından tek farkı **struct** anahtar sözcüğü yerine **union** anahtar sözcüğünün kullanılmasıdır. Örnekler :

```
union DWORD a, b;
```

ile a ve b **union** DWORD türünden iki değişken olarak tanımlanmışlardır.

```
union DBL_FORM x, y;
```

x ve y **union** DBL\_FORM türünden iki değişkendir. Yine tıpkı yapılarda olduğu gibi birliklerde de bildirim ile tanımlama işlemi birlikte yapılabilir :

```
union DBL_FORM {
    double n;
    unsigned char s[8];
} a, b, c;
```

Bu durumda a, b ve c değişkenlerinin faaliyet alanları birlik bildiriminin yapıldığı yere bağlı olarak yerel ya da global olabilir.

Birlik elemanlarına da yapı elemanlarında olduğu gibi nokta operatörüyle erişilir. Örneğin yukarıdaki tanımlama dikkate alınırsa a.n birliğin double türünden ilk elemanını belirtmektedir. Benzer biçimde birlik türünden göstericiler de tanımlanabilir. Ok operatörü ile yine yapılarda olduğu gibi birliklerde de gösterici yoluyla birlik elemanlarına ulaşılabilir. Örneğin:

```
union DWORD *p;
```

```
...
```

```
printf("%d", p->word);
```

p->word ifadesi ile p adresinde birlik nesnesinin word elemanına erişilmektedir.

Genel olarak şunları söyleyebiliriz: Yapılarla birliklerin bildirilmesi, tanımlanması ve yapı değişkenlerinin kullanılması tamamen aynı biçimde yapılmaktadır. İkisi arasında tek fark yapı ve birlik elemanlarının bellekteki organizasyonunda ortaya çıkar. Birlik elemanlarının bellekteki organizasyonu konusuna değinmeden önce bir byte'dan daha uzun bilgilerin bellekteki görünüşleri üzerinde duralım.

## Sayıların Bellekteki Yerleşimleri

Bir byte'dan daha büyük olan sayıların belleğe yerleşim biçimi kullanılan mikroişlemciye göre değişebilir. Bu nedenle sayıların bellekteki görünüşleri taşınabilir bir bilgi değildir. Mikroişlemciler iki tür yerleşim biçimi kullanabilirler:

### Düşük anlamlı byte değerleri belleğin düşük anlamlı adresinde bulunacak biçimde. (little endian)

80x86 ailesi Intel işlemcileri bu yerleşim biçimini kullanır. Bu işlemcilerin kullanıldığı sistemlere örneğin

```
int x = 0x1234;
```

biçimindeki bir x değişkeninin bellekte 1FC0 adresinden başlayarak yerleştiğini varsayalım:

	...	
1A00	34	düşük anlamlı byte
1A01	12	yüksek anlamlı byte
	...	

Şekilden de görüldüğü gibi x değişkenini oluşturan sayılar düşük anlamlı byte değeri (34H) düşük anlamlı bellek adresinde (1A00H) olacak biçimde yerleştirilmiştir. Şimdi aşağıdaki kodu inceleyiniz :

```
...
```

```
unsigned long a = 0x12345678;
```

```
unsigned int *b;
```

```
b = (unsigned int*) &a;
```

```
printf("%x\n", *b);
```

ile ekrana hex sistemde 5678 sayıları basılır. Aşağıdaki şekli inceleyiniz. b değişkeninin 1B10 adresinden başlayarak yerleştirildiği varsayılmıştır :

	...			
1B10		78	1B10	*b
1B11		56		
1B12		34		
1B13		12		
	...			

### Düşük anlamlı byte değerleri, belleğin yüksek anlamlı adresinde bulunacak şekilde (big endian)

Motorola işlemcileri bu yerleşim biçimini kullanır. Örneğin yukarıdaki kod Motorola işlemcilerinin kullanıldığı bir sistemde yazılmış olsaydı :



```

unsigned long a = 0x12345678;
unsigned int *b;
b = (unsigned int*) &a;
printf("%x\n", *b);

```

ekrana 1234 basılırdı.

	...			
1B10		12	1B10	*b
1B11		34		
1B12		56		
1B13		78		
	...			

Aşağıdaki kod kullanılan sistemin little endian ya da big endian olduğunu test etmektedir :

```

int x = 1;

if (*(char *)&x == 1)
    printf("little-endian\n");
else
    printf("big-endian\n");

```

## Birlik Elemanlarının Bellekteki Organizasyonu

Birlik değişkenleri için birliğin en uzun elemanı kadar yer ayrılır. Birlik elemanlarının hepsi aynı orjinden başlayacak şekilde belleğe yerleşirler. Örneğin :

```

union DWORD {
    unsigned char byte;
    unsigned int word;
    unsigned long dword;
};

```

...

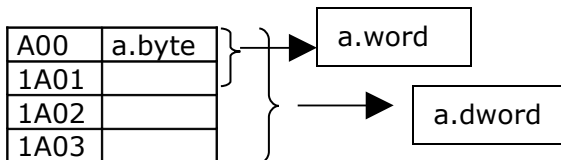
```

union DWORD a;

```

bildirimi ile a değişkeni için 4 byte yer ayrılacaktır. Çünkü a değişkeninin dword elemanı 4 byte ile birliğin en uzun elemanıdır.

Birlik bir dizi içeriyorsa dizi tek bir eleman olarak alınır. Örneğin



```

union DBL_FORM {
    double n;
    unsigned char s[8];
};

```

...

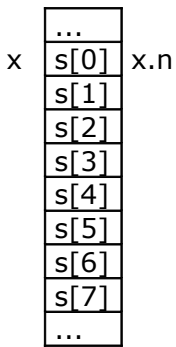
```

union DBL_FORM x;

```

tanımlaması ile x değişkeni için ne kadar yer ayrılacaktır? DBL\_FORM birliğinin iki elemanı vardır. Birincisi 8 byte uzunluğunda bir karakter dizisi, ikincisi de 8 byte uzunluğunda **double**

türden bir değişkendir. İki uzunluk da aynı olduğuna göre x için 8 byte yer ayrılacağını söyleyebiliriz:



Buna göre DBL\_FORM X;

...

X.n = 10.2;

ile x.s[0], x.s[1], x.s[2], ....x.s[7] sırasıyla double elemanın byte değerlerini göstermektedir.

Birlik elemanlarının aynı orjinden, yani aynı adresten başlayarak yerleştirilmesi bir elemanın değiştirince diğer elemanlarının da içeriğinin değişeceği anlamına gelir. Zaten birliklerin kullanılmasının asıl amacı da budur. Birlik elemanı olarak yapıların kullanılması uygulamada en sık karşılaşılan durumdur. Örneğin :

```
struct WORD {
    unsigned char low_byte;
    unsigned char high_byte;
};
```

```
union WORD_FORM {
    unsigned int x;
    struct WORD y;
};
```

bildirimlerinden sonra union WORD\_FORM türünden bir birlik tanımlanırsa :

```
union WORD_FORM wf;
```

bu birliğin alçak (low\_byte) ve yüksek (high\_byte) anlamlı byte değerlerine ayrı ayrı erişebiliriz ya da onu bir bütün olarak kullanabiliriz :

Yani intel işlemcilerinin bulunduğu 16 bit sistemlerde :

```
wf.y.low_byte = 0x12;
wf.y.high_byte = 0x34;
```

işlemlerden sonra;

```
printf("%x\n", wf.x);
```

ile 3412 sayısını ekrana yazdıracaktır. (Motorola işlemcilerinde belleğin düşük anlamlı bölgesinde düşük anlamlı byte değeri olacağına göre sayının ters olarak görülmesi gerekir. Birlik elemanlarının bellekte aynı orjinden başlanarak yerleştirildiğini aşağıdaki kod parçasından da rahatlıkla görebiliriz :

## Birlik Nesnelere İlk Değer Verilmesi

ANSI ve ISO standartlarına göre birlik nesnelerinin yalnızca ilk elemanlarına değer verilebilir. Eğer bir birlik nesnesinin birden fazla elemanına ilk değer verilmeye çalışılırsa derleme zamanında error oluşacaktır:

```
union DWORD {
    unsigned char byte;
    unsigned int word;
    unsigned long dword;
} x = {'m'};
```

```
union DWORD y = {'a', 18, 24L}; /* error */
```

## Birlikler Neden Kullanılır

Birlikler iki amaçla kullanılabilir. Birincisi yer kazancı sağlamak içindir. Birlik kullanarak farklı zamanlarda kullanılacak birden fazla değişken için ayrı ayrı yer ayırma zorunluluğu ortadan kaldırılır. Örneğin bir hediyelik eşya kataloğu ile 3 değişik ürünün satıldığını düşünelim: kitap, tshort ve saat. Her bir ürün için bir stok numarası, fiyat bilgisi ve tip bilgisinin dışında ürüne bağlı olarak başka özelliklerine de sahip olduğunu düşünelim:

kitaplar : isim, yazar, sayfa sayısı.  
t-short : desen, renk, size.  
saat : model

```
struct katalog {
    int stok_no;
    float fiyat;
    int urun_tipi
    char kitapisim[20];
    char yazar[20];
    int sayfa_sayisi;
    char desen[20];
    int renk;
    int size;
    char saatisim[20];
    int model;
};
```

Yukarıdaki bildirimde urun\_tipi elemanı yalnızca KITAP, TSHORT ya da SAAT olabilecektir. (Bunların sembolik sabit olarak tanımlandığını düşünelim.) Yukarıda bildirilen yapı ürünlerin bütün özelliklerini tutabilmekle birlikte şöyle bir dezavantajı vardır :

Eğer ürün tipi KITAP değil ise isim[20], yazar[30] ve sayfa\_sayisi elemanları hiç kullanılmayacaktır. Yine ürün tipi TSHORT değil ise desen[20], renk, size elemanları hiç kullanılmayacaktır.

Ama katalog yapısının içine birlikler yerleştirirsek yer kazancımız olacaktır:

```
struct katalog {
    int stok_no;
    float fiyat;
    int urun_tipi
    union {
        struct {
            char isim[20];
            char yazar[20];
            int sayfa_sayisi;
        } kitap;
```

```

    struct {
        char desen[20];
        int renk;
        int size;
    } tshort;
    struct {
        char isim[20];
        int model;
    } saat;
} cins;
};

```

Birlik kullanımının ikinci nedeni herhangi bir veri türünün parçaları üzerinde işlem yapmak ya da parçalar üzerinde işlem yaparak bir veri türünden bütünü oluşturulmasıdır. Bunun için `int` türden bir sayının byte değerlerini ayrıştırma örneğini yineleyebiliriz :

```

struct WORD {
    unsigned char low_byte;
    unsigned char high_byte;
};

union WORD_FORM {
    unsigned int x;
    struct WORD y;
};

union word_form wf;

```

Tanımlamalarından sonra `wf.x` elemanına değer atadığımızda `wf.y.low_byte` ve `wf.y.high_byte` ile yüksek ve alçak anlamlı byte değerleri üzerine kolaylıkla işlem yapabiliriz.

## Birliklerin Karışık Veri Yapılarında Kullanılması

Birlikleri kullanarak, elemanları farklı türden olan diziler oluşturabiliriz.

Belirli değerleri bir dizi içinde tutmak zorunda olduğumuzu düşünelim. Ancak dizi elemanları **int**, **long**, **double** türlerinden olabilsin. Örneğin dizinin, herhangi bir elemanına ulaştığımızda bu eleman **int** ise onu **int** olarak, **long** ise **long** olarak ve **double** ise **double** olarak (bilgi kaybı olmaksızın) kullanabilelim.

Diziyi **double** türü ile tanımlarsak, dizi elemanlarına atama yaptığımız zaman, otomatik tür dönüşümü sonucunda bütün elemanlar **double** türüne çevrileceğinden dizi elemanlarının tür bilgisini kaybetmiş olurduk.

Ancak aşağıdaki şekilde bir birlik oluşturabiliriz :

```

union DEGER {
    int i;
    long l;
    double d;
};

```

**union** DEGER türünden her bir nesne bellekte, en uzun elemanı kadar, yani 8 byte yer kaplayacak, ve her bir eleman (member) aynı adresten başlayacaktır.

Şimdi **double** türden bir dizi açmaktansa **union** DEGER türünden bir dizi açabiliriz. Dizinin her bir elemanının, istediğimiz alt elemanına değer ataması yapabiliriz :

```

main()
{
    union DEGER s[100];

    s[0].i = 3;           /* s[0] int türden bir değeri tutuyor */
    s[1].d = 6.           /* s[1] double türden bir değeri tutuyor */
    s[2].l = 12L          /* s[2] long türden bir değeri tutuyor */
    ...
}

```

Hangi türden bir değer tutmak istiyorsak, bu değeri s dizisinin elemanı olan bir **union** DEGER nesnesinin ilgili alt elemanında tutuyoruz. Ancak hala şöyle bir eksiğimiz var : Örneğin dizinin 17. elemanını gerçek türüyle ve değeriyle kullanmak istiyoruz. 17. eleman nesnesininin hangi türden olduğunu nereden bileceğiz? Hangi dizi elemanının hangi türden değer için kullanıldığı bilgisini de bir şekilde saklamalıyız.

Bunun için en iyi yöntem bir yapı bildirimi yapmak, ve DEGER birliğini de bu yapının alt elemanı olarak bildirmektir. Yapının başka bir elemanı da birliğin hangi elemanının kullanılacağı bilgisini tutacaktır :

```

#define INT      0
#define LONG     1
#define DOUBLE   2
...

struct KARTIP {
    char type;
    union DEGER deger;
};

```

Şimdi **struct** KARTIP türünden bir dizi tanımlayabiliriz:

```

...
struct KARTIP s[100];
...
s[0].type = INT;
s[0].deger.i = 3;
s[1].type = DOUBLE;
s[1].deger.d = 6.;
s[2].type = LONG;
s[2].deger.l = 12L;

```

## 29 . BÖLÜM : BİTSEL OPERATÖRLER

Bitsel operatörler (Bitwise operators) bir tamsayının bitleri üzerinde işlem yapan operatörlerdir, daha çok sistem programlarında kullanılırlar. Bitsel operatörlerin ortak özellikleri operandları olan tamsayıları bir bütün olarak değil, bit bit ele alarak işleme sokmalarıdır. Bitsel operatörlerin operandları tamsayı türlerinden olmak zorundadır. Operandları tamsayı türlerinden birinden değilse derleme zamanında hata oluşur.

C dilinde toplam 11 tane bitsel operatör vardır. Bu operatörler, kendi aralarındaki öncelik sırasına göre, aşağıdaki tabloda verilmiştir:

2	~	bitsel değil	bitwise not
5	<< >>	bitsel sola kaydırma bitsel sağa kaydırma	bitwise left shift bitwise right shift
8	&	bitsel ve	bitwise and
9	^	bitsel özel veya	bitwise exor
10		bitsel veya	bitwise or
14	<<= >>= &= ^=   =	bitsel işlemleri atama operatörleri	bitwise compound assignment operators

Yukarıdaki operatörler içinde yalnızca bitsel Değil (bitwise not) operatörü örnek konumunda tek operand alan (unary prefix) bir operatördür. Diğer bütün bitsel operatörler iki operand alırlar ve operandlarının arasında bulunurlar. (binary infix)

Şimdi bu operatörleri tek tek tanıyalım :

### Bitsel Değil Operatörü

Diğer tüm tek operand alan operatörler gibi operatör öncelik tablosunun ikinci seviyesindedir. Yan etkisi yoktur. (side effect) Operandı bir nesne ise, bu nesnenin bellekteki değerini değiştirmez. Ürettiği değer, operandı olan tamsayının bitleri üzerinde 1'e tümleme işlemi yapılarak elde edilen değerdir. Yani operandı olan ifade değerinin 1 olan bitleri 0, 0 olan bitleri 1 yapılarak değer üretilir.

```
#include <stdio.h>

int main()
{
    unsigned int x = 0x1AC3;          /* x = 0001 1010 1100 0011 */
    unsigned int y;

    y = ~x;                          /* y = 1110 0101 0011 1100 */
    printf("y = %x\n", y);          /* y = 0xE53C */
    return 0;
}
```

### Bitsel Kaydırma Operatörleri (bitwise shift operators)

İki adet bitsel kaydırma operatörü vardır:

Bitsel sağa kaydırma operatörü >> (bitwise right shift)

Bitsel sola kaydırma operatörü << (bitwise left shift)

Her iki operatör de operatör öncelik tablosunun 5. seviyesinde bulunmaktadır. (Dolayısıyla bu operatörlerin önceliği tüm aritmetik operatörlerden daha aşağıda fakat karşılaştırma operatörlerinden daha yukarıdadır. Bitsel kaydırma operatörleri binary infix operatörlerdir. (İki operand alan aritmetik konumundaki operatörlerdir.)

Bitsel sola kaydırma operatörünün ürettiği değer soldaki operandı olan tamsayının sağdaki operandı olan tamsayı kadar sola kaydırılmış biçimidir. Menzil 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;

    z = x << y;              /* z = 0000 0000 1101 0000 */
    printf("z = %d\n", z)    /* z = 208 */
    return 0;
}
```

Bir sayıyı sola bitsel olarak 1 kaydırmakla o sayının ikiyle çarpılmış değeri elde edilir. Bir sayıyı sağa bitsel olarak 1 kaydırmakla o sayının ikiye bölünmüş değeri elde edilir.

Bitsel kaydırma operatörlerinin yan etkileri yoktur. Yani soldaki operandları bir nesne ise, bu nesnenin bellekteki değeriniği değiştirmezler. Kaydırma sonucu soldaki nesnenin değeri değiştirilmek isteniyorsa daha sonra inceleyeceğimiz bitsel işlemli atama operatörleri kullanılmalıdır. (>>= ya da <<=)

Bitsel sağa kaydırma operatörünün ürettiği değer soldaki operandı olan tamsayının sağdaki operandı olan tamsayı kadar sağa kaydırılmış biçimidir. Soldaki operand işaretli (signed) ya da pozitif işaretli (signed) bir tamsayı ise menzil dışına çıkan bitler yerine sayının solundan besleme 0 biti ile yapılması C standartları tarafından garanti altına alınmıştır. Sağa kaydırılacak ifadenin negatif değerli, işaretli bir tamsayı türünden olması durumunda menzil dışına çıkan bitler için soldan yapılacak beslemenin 0 ya da 1 bitleriyle yapılması uygulama bağımlıdır. (implementation depended). Yani derleyiciyi yazarlar bu durumda sayının işaretini korumak için soldan yapılacak beslemeyi 1 biti ile yapabilecekleri gibi, sayının işaretini korumayı düşünmeksizin 0 biti ile de yapabilirler. İşaretli negatif bir tamsayının bitsel sağa kaydırılması taşınabilir bir özellik olmadığından dikkatli olunmalıdır.

```
#include <stdio.h>

int main()
{
    unsigned int x = 52;    /* x = 0000 0000 0011 0100 */
    unsigned int y = 2;

    z = x >> y;              /* z = 0000 0000 0000 1101 */
    printf("z = %d\n", z)    /* z = 13 */
    return 0;
}
```

## Bitsel Ve Operatörü (bitwise and)

Operatör öncelik tablosunun 8. seviyesindedir. Öncelik yönü soldan sağadır. Yan etkisi yoktur, operandları nesne gösteren bir ifade ise bellekteki değerlerini değiştirmez. Değer üretmek için operandı olan tamsayıların karşılıklı bitlerini Ve işlemine tabi tutar. Ve işlemine ait doğruluk tablosu aşağıda tekrar verilmektedir:

x	y	x & y
1	1	1
1	0	0
0	1	0
0	0	0

Bitsel ve operatörünün ürettiği değer operandlarının 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;

    z = x & y;                        /* z = 0001 1010 0000 0101 */

    printf("z = %xn", Z);            /* z = 0x1A05 */
    return 0;
}
```

1 biti bitsel ve işleminde etkisiz elemandır.

0 biti bitsel ve işleminde yutan elemandır.

### Bitsel Özel Veya Operatörü (bitwise exor)

Operatör öncelik tablosunun 9. seviyesindedir. Öncelik yönü soldan sağdır. Yan etkisi yoktur, operandları nesne gösteren bir ifadeyse bellekteki değerlerini değiştirmez. Değer üretmek için operandı olan tamsayıların karşılıklı bitlerini özel veya (exclusive or) işlemine tabi tutar. Bitsel özel veya işlemine ilişkin doğruluk tablosu aşağıda verilmektedir:

x	y	x ^ y
1	1	0
1	0	1
0	1	1
0	0	0

Yukarıdaki tablo şöyle özetlenebilir. Operandlardan ikisi de aynı değere sahip ise üretilen değer 0 olacak, operandlardan biri diğerinden farklı ise üretilen değer 1 olacaktır.

```
#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;

    z = x ^ y;                        /* z = 0010 0001 1100 1000 */
    printf("z = %xn", z);            /* z = 0x21C8 */
    return 0;
}
```

Bir sayıya arka arka arkaya aynı değerle özel veya işlemi yapılırsa aynı değer elde edilir:

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

```
int main()
{
    unsigned int x = 0x1BC5;          /* x = 0001 1011 1100 0101 */
    unsigned int y = 0X3A0D          /* y = 0011 1010 0000 1101 */
    x = x ^ y;                        /* x = 0010 0001 1100 1000   x = 0x21C8 */
    x = x ^ y;                        /* x = 0001 1011 1100 0101   x = 0x1BC5 */
    printf("x = %xn", x);            /* x = 0x0x1BC5 */
    return 0;
}
```



Bazı şifreleme algoritmalarında özel veya işleminin bu özelliğinden faydalanılmaktadır.

bitsel exor operatörü tamsayı türlerinden iki değişkenin değerlerinin, geçici bir değişken olmaksızın swap edilmesinde (değerlerinin değiştirilmesinde) de kullanılabilir. Aşağıdaki programda x ve y değişkenlerini değerleri bitsel exor operatörünün kullanılmasıyla swap edilmektedir :

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

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

    x ^= y ^= x ^= y;
    printf("x = %d\n y = %d\n", x, y);
    return 0;
}
```

### Bitsel Veya Operatörü (bitwise or operator)

Operatör öncelik tablosunun 10. seviyesindedir. Öncelik yönü soldan sağdır. Yan etkisi yoktur, operandları nesne gösteren bir ifade ise bellekteki değerlerini değiştirmez. Değer üretmek için operandı olan tamsayıların karşılıklı bitlerini veya işlemine tabi tutar. Veya işlemine ait doğruluk tablosu aşağıda verilmektedir:

x	y	x   y
1	1	1
1	0	1
0	1	1
0	0	0

Bitsel veya operatörünün ürettiği değer operandlarının karşılıklı bitlerinin veya işlemine sokularak 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;

    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.

### Bitsel Operatörlere İlişkin İşlemler Atama Operatörleri

Bitsel değil operatörünün dışında tüm bitsel operatörlere ilişkin işlemler atama operatörleri vardır. Daha önce de söylendiği gibi bitsel operatörlerin yan etkileri (side effect) yoktur. Bitsel operatörler operandları olan nesnelerin bellekteki değerlerini değiştirmezler. Eğer operandları olan nesnelerin değerlerinin değiştirilmesi isteniyorsa bu durumda işlemler atama operatörleri kullanılmaktadır.

```

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 Operatörlerin Kullanılmasına İlişkin Bazı Temalar

Bitsel operatörlerin kullanılmasına daha çok sistem programlarında raslanır. Sistem programlarında bir sayının bitleri üzerinde bazı işlemler yapılması sıklıkla gerekli olmaktadır. Aşağıda bitsel düzeyde yapılan işlemlerden örnekler verilmektedir.

### Bir Sayının Belirli Bir Bitinin 1 Yapılması (birlenmesi) (set a bit / turn a bit on)

Buna sayının belirli bir bitinin set edilmesi de denebilir. Bir sayının belirli bir bitini set etmek için, söz konusu sayı, ilgili biti 1 olan ve diğer bitleri 0 olan bir sayıyla veya işlemine tabi tutulmalıdır. Çünkü bitsel veya işleminde 1 yutan eleman 0 ise etkisiz elemandır.

Aşağıdaki örnekte bir sayının 5. biti set edilmektedir.

```

#include <stdio.h>
#include <conio.h>

int main()
{
    int ch = 0x0041;          /* ch = 65      0000 0000 0100 0001 */
    int mask = 0x0020;       /* mask = 32    0000 0000 0010 0000 */

    ch |= mask;              /* ch =        0000 0000 0110 0001 */
    printf("ch = %d\n", ch); /* ch = 97 */
    getch();
    return 0;
}

```

x bir tamsayı olmak üzere, bir sayının herhangi bir bitini set edecek bir ifadeyi şu şekilde yazabiliriz:

$x |= 1 \ll k$

### Bir Sayının Belirli Bir Bitinin 0 Yapılması (sıfırlanması) (clear a bit / turn the bit off)

Buna sayının belirli bir bitinin temizlenmesi de denebilir. Bir sayının belirli bir bitini sıfırlamak için, söz konusu sayı, ilgili biti 0 olan ve diğer bitleri 1 olan bir sayıyla bitsel ve işlemine tabi tutulmalıdır. Çünkü bitsel ve işleminde 0 yutan eleman 1 ise etkisiz elemandır. Bir bitin sıfırlanması için kullanılacak bu sayıya maske (mask) denir.

Aşağıdaki örnekte bir sayının 5. biti sıfırlanmaktadır.

```

#include <stdio.h>
#include <conio.h>

int main()
{
    int ch = 0x0061;          /* ch = 97      0000 0000 0110 0001 */
    int mask = ~0x0020;       /* mask = ~32    1111 1111 1101 1111 */
    ch &= mask;              /* ch          0000 0000 0100 0001 */
    printf("ch = %d\n", ch); /* ch = 65 */
    getch();
}

```

```
    return 0;
}
```

x bir tamsayı olmak üzere, bir sayının herhangi bir bitini set edecek bir ifadeyi şu şekilde yazabiliriz

```
x &= ~(1 << k);
```

### Bir Sayının Belirli Bir Bitinin Değerinin Test Edilmesi (0 mı 1 mi)

Bir sayının belirli bir bitinin 0 mı 1 mi olduğunun öğrenilmesi için, söz konusu sayı, **if** deyiminin koşul ifadesi olarak (**if** parantezi içinde) ilgili biti 1 olan ve diğer bitleri 0 olan bir sayıyla bitset ve işlemine tabi tutulmalıdır. Çünkü bitset ve işlemine 0 yutan eleman 1 ise etkisiz elemandır. programın akışı **if** deyiminin doğru kısmına giderse, ilgili bitin 1, yanlış kısmına gider ise ilgili bitin 0 olduğu sonucu çıkarılacaktır.

x bir tamsayı olmak üzere, bir sayının herhangi bir bitinin 1 ya da 0 olduğunu anlamak için aşağıdaki ifadeyi yazabiliriz.

```
if (x & 1 << k)
    /* k. bit 1 */
else
    /* k. bit 0 */
```

### Bir Sayının Belirli Bir Bitini Ters Çevirmek (toggle)

Bazı uygulamalarda bir sayının belirli bir bitinin değerinin değiştirilmesi istenebilir. Yani söz konusu bir 1 ise 0 yapılacak, söz konusu bit 0 ise 1 yapılacaktır. Bu amaçla bitset özel veya operatörü kullanılır. Bitset özel veya operatöründe 0 biti etkisiz elemandır. Bir sayının k.bitinin değerini değiştirmek için, sayı, k.bit 1 diğer bitleri 0 olan bir sayı ile bitset özel veya işlemine tabi tutulur.

x bir tamsayı olmak üzere, bir sayının herhangi bir bitinin değerini değiştirmek için aşağıdaki ifadeyi yazabiliriz.

```
x ^= 1 << k;
```

Bir tamsayının belirli bitlerini sıfırlamak için ne yapabiliriz? Örneğin int türden bir nesnenin 7., 8. ve 9.bitlerini sıfırlamak isteyelim (tabi diğer bitlerini değiştirmeksizin). 7., 8. ve 9. bitleri 0 olan diğer bitleri 1 olan bir sayı ile bitset ve işlemine tabi tutarız. Örneğin 16 bitlik **int** sayıların kullanıldığı bir sistemde bu sayı aşağıdaki bit düzenine sahip olacaktır.

```
1111 1100 0111 1111    (bu sayı 0xFC7F değildir?)
```

```
x &= 0xFC7F;
```

Bu gibi tamsayıların bitleri üzerinde yapılan işleri fonksiyonlara yaptırmaya ne dersiniz?

```
void clearbits(int *ptr, int startbit, int endbit);
```

clearbits fonksiyonu adresi gönderilen ifadenin startbit ve endbit aralığındaki bitlerini sıfırlayacaktır.

Örneğin x isimli **int** türden bir nesnenin 7. 8. 9. bitlerini sıfırlamak için fonksiyon aşağıdaki şekilde çağırılacaktır:

```
clearbits(&x, 7, 9);
```

```
void clearbits(int *ptr, int startbit, int endbit)
{
    int k;
```

```

    for (k = startbit; k <= endbit; ++k)
        *ptr &= ~(1 << k);
}

```

Benzer şekilde setbits fonksiyonunu da yazabiliriz :

```

void setbits (int *ptr, int startbit, int endbit)
{
    int k;

    for (k = startbit; k <= endbit; ++k)
        *ptr |= 1 << k;
}

```

Peki örneğin 16 bitlik bir alanın içindeki belirli bitleri birden fazla değeri tutacak şekilde kullanabilir miyiz?

Örneğin 4 bitlik bir alan içerisinde (negatif sayıları kullanmadığımızı düşünürsek 0 – 15 aralığındaki değerleri tutabiliriz. O zaman yalnızca bu aralıktaki değerleri kullanmak istiyorsak ve bellek (ya da dosya) büyüklüğü açısından kısıtlamalar söz konusu ise 16 bitlik bir alan içinde aslında biz 4 ayrı değer tutabiliriz değil mi?

Bir sayının belirli bir bit alanında bir tamsayı değerini tutmak için ne yapabiliriz? Önce sayının ilgili bitlerini sıfırlar (örneğin yukarıda yazdığımız clearbits fonksiyonuyla) daha sonra sayıyı, sıfırlanmış bitleri yerleştireceğimiz sayının bit paternine eşit diğer bitleri 0 olan bir sayı ile bitsel veya işlemine tabi tutarız, değil mi? Aşağıdaki fonksiyon prototip bildirimine bakalım:

**void putvalue(int \*ptr, int startbit, int endbit, int value);**

putvlaue fonksiyonu adresi gönderilen nesnenin startbit – endbit değerleri arasındaki bitlerine value sayısını yerleştirecektir. Fonksiyon aşağıdaki gibi çağırılabilir :

```

putvalue(&x, 0, 3, 8);
putvalue(&x, 4, 7, 12);
putvalue(&x, 8, 11, 5);
putvalue(&x, 12, 15, 3);

```

```

void putvalue(int *ptr, int startbit, int endbit, int value);
{
    int temp = value << startbit;

    clearbits(ptr, startbit, endbit);
    *ptr |= temp;
}

```

Peki yukarıdaki fonksiyonlar ile örneğin 16 bitlik bir sayının belirli bit alanları içinde saklanmış değerleri nasıl okuyacağız. Bu işi de bir fonksiyona yaptıralım :

**int getvalue(int x, int startbit, int endbit);**

getvalue fonksiyonu startbit ve endbit ile belirlenen bit alanına yerleştirilmiş değer ile geri dönecektir.

```

int getvalue(int number, int startbit, int endbit)
{
    int temp = number >>= startbit;

    clearbits(&temp, endbit + 1, sizeof(int) * 8 - 1);
    return temp;
}

```

## Uygulama

**int** türden bir sayının bit bit ekrana yazdırılması

```
include <stdio.h>

void showbits(int x)
{
    int i = sizeof(int) * 8 - 1;

    for (; i >= 0; --i)
        if (x >> i & 1 == 1)
            putchar('1');
        else
            putchar('0');
}
```

Bu fonksiyonu aşağıdaki şekilde de yazabilirdik:

```
#include <stdio.h>

void showbits2(int x)
{
    unsigned i = (~(unsigned)~0 >> 1));

    while (i) {
        if (x & i)
            putchar('1');
        else
            putchar('0');
        i >>= 1;
    }
}
```

## Uygulama

**int** türden bir sayının bitlerini ters çeviren reverse\_bits fonksiyonunun yazılması:

```
#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;
}
```

## Uygulama

**int** türden bir değerin kaç adet bitinin set edilmiş olduğu bilgisine geri dönen no\_of\_setbits fonksiyonunun yazılması :

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

```
int no_of_setbits(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;
}
```

Daha hızlı çalışacak bir fonksiyon tasarlamaya ne dersiniz?

```
#include <stdio.h>

int no_of_setbits(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;
}
```

## **30 . BÖLÜM : BİT ALANLARI**

**konu eklenecek**

## 31 . BÖLÜM : KOMUT SATIRI ARGUMANLARI

Bir programı çalıştırdığımız zaman, programın çalışmasını yönlendirecek ya da değiştirecek birtakım parametre(leri) çalışacak programa göndermek isteyebiliriz. Bu parametreler bir dosya ismi olabileceği gibi, programın değişik biçimlerde çalışmasını sağlayacak bir seçenek de olabilir. UNIX işletim sistemindeki ls komutunu düşünelim. Eğer ls programını

ls

yazarak çalıştırsak, bulunulan dizin içindeki dosyaların isimleri listelenir. (DOS'daki dir komutu gibi). 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 programını yine

ls -l sample.c

yazarak çalıştırsak, yalnızca sample.c dosyasına ilişkin bilgiler görüntülenecektir. İşte bir programı çalıştırırken program isminin yanına yazılan diğer parametrelere komut satırı argümanları (command line arguments) denilmektedir.

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) denmektedir.

C programlarının çalışmaya başladığı main fonksiyonu da isteğe bağlı olarak iki parametre alabilir. Bu parametreler geleneksel olarak argc ve argv olarak isimlendirilirler.

```
int main(int argc, char *argv[])
{
    ...
}
```

argc (argument count) komut satırı argümanlarının sayısını gösterir. Bu sayıya programın ismi de dahildir. argv ise , stringler şeklinde saklanan komut satırı argümanlarını gösteren, **char** türden bir gösterici dizisidir. Bu durumda argv[0] göstericisini programın ismini tutan stringi, argv[1] den argv[argc - 1] e kadar olan göstericiler ise program ismini izleyen diğer argümanların tutuldukları stringleri gösterirler. argv[argc] ise her zaman bir NULL göstericiyi göstermektedir.

Yukarıdaki örnekte kullanıcı ls programını

ls -l sample.c şeklinde çalıştırdığında

argc 3 değerini alır.

argv[0] program ismini gösterir. argv[1] = "ls";

argv[1] program ismini izleyen 1. argümanı gösterir. argv[1] = "-l";

argv[2] program ismini izleyen 2. argümanı gösterir. argv[2] = "sample.c";

argv[argc] yani argv[3] ise NULL adresini gösterecektir.

komut satırı argümanları boşluklarla birbirinden ayrılmış olmalıdır. Yukarıdaki örnekte program

ls -lsample.c şeklinde çalıştırılırsa

argc = 2 olurdu.



Komut satırı argümanlarının ikincisi karakter türünden bir göstericiyi gösteren göstericidir. Yani `argv[0]`, `argv[1]`, `argv[2]`... herbiri **char** türden bir göstericidir. Komut satırı argümaları NULL ile sonlandırılmış bir biçimde bu adreslerde bulunurlar. `argv[0]` sürücü ve izin dahil olmak üzere (full path name) programın ismini vermektedir. Diğer argümanlar sırasıyla `argv[1]`, `argv[2]`, `argv[3]`, ... adreslerindedir.

Komut satırı argümanları işletim sistemi tarafından komut satırından alınır ve derleyicinin ürettiği giriş kodu yardımıyla `main` fonksiyonuna parametre olarak kopyalanır.

Aşağıda komut satırı argümanlarını 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;
}
```

Şüphesiz bu kodu aşağıdaki şekilde de yazabilirdik .

```
int main(int argc, char *argv[])
{
    int i;

    for (i = 0; argv[i] != NULL ; ++i)
        printf("argv[%d] : %s\n", i, argv[i]);
    return 0;
}
```

Komut satırı argümanlarının isimleri `argc` ve `argv` olmak zorunda değildir. Bunlar yerine herhangi iki isim de kullanılabilir. Ancak `argc` ve `argv` isimleri "argument counter" ve "argument vector" sözcüklerinden kısaltılmıştır. Bu isimler programcılar tarafından geleneksel olarak kullanılmaktadır.

Komut satırı argümanlarını alan programlar genellikle, önce girilen argümanları yorumlar ve test ederler. Örneğin :

```
int main(int argc, char *argv[])
{
    if (argc == 1) {
        printf("lütfen bir argümanla çalıştırınız!..\n");
        exit(1);
    }

    if (argc == 2) {
        printf("lütfen bir argümanla çalıştırınız!..\n");
        exit(1);
    }
    return 0;
}
```

Yukarıdaki örnekte program bir komut satırı argümanı verilerek çalıştırılmadıysa (yani eksik argümanla çalıştırıldıysa) bir hata mesajıyla sonlandırılıyor. Benzer biçimde fazla sayıda argüman için de böyle bir kontrol yapılmıştır.

DOS'ta olduğu gibi bazı sistemlerde `main` fonksiyonu üçüncü bir parametre alabilir. Üçüncü parametre sistemin çevre değişkenlerine ilişkin bir karakter türünden göstericiyi gösteren göstericidir.

```
int main(int argc, char *argv[], char *env[])
{
    ...
    return 0;
}
```

main fonksiyonuna tüm parametreleri geçilmek zorunda değildir. Örneğin :

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

gibi yalnızca birinci parametrenin kullanıldığı bir tanımlama geçerlidir. Ancak yalnızca ikinci parametre kullanılamaz. Örneğin:

```
int main(char *argv[])
{
    ...
    return 0;
}
```

## Komut Satırı Argumanlarının Kullanılmasına Bir Örnek

Komut satırından çalışan basit bir hesap makinası

```
#include <stdio.h>
#include <conio.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 operatör\n");
    }
    return 0;
}
```

## 32 . BÖLÜM : DOSYALAR

İkincil bellekte tanımlanmış bölgelere dosya denir. Her dosyanın bir ismi vardır. Ancak dosyaların isimlendirme kuralları sistemden sisteme göre değişebilmektedir. Dosya işlemleri tamamen işletim sisteminin kontrolü altındadır.

İşletim sistemi de ayrı ayrı yazılmış fonksiyonların birbirlerini çağırması biçiminde çalışır. Örneğin komut satırında ismi yazılmış olan bir programın çalıştırılması birkaç sistem fonksiyonunun çağırılması ile yapılmaktadır. Komut satırından yazıyı alan, diskte bir dosyayı arayan, bir dosyayı RAM'e yükleyen, RAM'deki programı çalıştıran fonksiyonlar düzenli olarak çağırılmaktadır.

İşletim sisteminin çalışması sırasında kendisinin de çağırdığı, sistem programcısının da dışarıdan çağırabildiği işletim sistemine ait fonksiyonlara sistem fonksiyonları denir. Bu tür fonksiyonlara Windows sisteminde API (Application Programming Interface) fonksiyonları, UNIX işletim sisteminde ise sistem çağırımları (system calls) denir.

Aslında bütün dosya işlemleri, hangi programlama dili ile çalışılırsa çalışılsın, işletim sisteminin sistem fonksiyonları tarafından yapılır. Sistem fonksiyonlarının isimleri ve parametrik yapıları sistemden sisteme değişebilmektedir.

### Dosyalara İlişkin Temel Kavramlar

#### Dosyanın Açılması

Bir dosya üzerinde işlem yapmadan önce dosya açılmalıdır. Dosya açabilmek için işletim sisteminin "dosya aç" isimli bir sistem fonksiyonu 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 içerisinde bulunan bir tabloya yazılır. Dosya tablosunun biçimi sistemden sisteme değişebilir. Örneğin tipik bir dosya tablosu aşağıdaki gibi olabilir :

**Dosya tablosu (File table)**

Sıra No	Dosya ismi	Dosyanın Diskteki Yeri	Dosyanın Özellikleri	Diğerleri
0				
1				
...				
12	AUTOEXEC.BAT	...	...	...
...				

Sistem fonksiyonlarının da parametreleri ve geri dönüş değerleri vardır. "Dosya aç" sistem fonksiyonunun parametresi açılacak dosyanın ismidir. Fonksiyon, dosya bilgilerinin yazıldığı sıra numarası ile geri döner ki bu değere "file handle" denir. Bu handle değeri diğer dosya fonksiyonlarına parametre olarak geçirilir. Dosyanın açılması sırasında buna ek olarak başka önemli işlemler de yapılmaktadır.

#### 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. Kapatılmaması çeşitli problemlere yol açabilir.

#### Dosyaya Bilgi Yazılması ve Okunması

İşletim sistemlerinin dosyaya n byte veri yazan ve dosyadan n byte veri okuyan sistem fonksiyonları vardır. Yazma ve okuma işlemleri bu fonksiyonlar kullanılarak yapılır.

## Dosya pozisyon göstericisi (file pointer)

Bir dosya byte'lardan oluşur. Dosyadaki her bir byte'a 0'dan başlayarak artan sırada bir sayı karşılık getirilmiştir. Bu sayıya ilgili byte'ın ofset numarası denir. Dosya pozisyon göstericisi **long** türden bir sayıdır ve bir ofset değeri belirtir. Dosyaya yazan ve dosyadan okuma yapan fonksiyonlar bu yazma ve okuma işlemlerini her zaman dosya pozisyon göstericisinin gösterdiği yerden yaparlar. Örneğin dosya göstericisinin gösterdiği yer 100 olsun. Dosyadan 10 byte bilgi okumak için sistem fonksiyonu çağırıldığında, 100. ofsetden itibaren 10 byte bilgi okunur. İşletim sisteminin dosya göstericisini konumlandıran bir sistem fonksiyonu vardır. Dosya ilk açıldığında dosya göstericisi 0. ofsetdedir. Örneğin bir dosyanın 100. Ofsetinden itibaren 10 byte okunmak istenirse sırası ile şu işlemlerin yapılması gerekir:

İlgili dosya açılır.

Dosya pozisyon göstericisi 100. Offsete konumlandırılır.

Dosyadan 10 byte okunur.

Dosya kapatılır.

C dilinde dosya işlemleri 2 biçimde yapılabilir :

1. İşletim sisteminin sistem fonksiyonları doğrudan çağırılarak.
2. Standart C fonksiyonları kullanılarak.

Prototipleri stdio.h içerisinde olan standart dosya fonksiyonlarının hepsinin ismi f ile başlar. Tabi standart C fonksiyonları da işlemlerini yapabilmek için aslında işletim sisteminin sistem fonksiyonlarını çağırmaktadır. İşletim sisteminin sistem fonksiyonları taşınabilir değildir. İsimleri ve parametrik yapıları sistemden sisteme değişebilir. Bu yüzden standart C fonksiyonlarının kullanılması tavsiye edilir.

## fopen fonksiyonu

FILE \*fopen (const char \*fname, const char \*mode)

Fonksiyonun 1. Parametresi açılacak dosyanın ismidir. 2. Parametre açış modudur. Dosya ismi path içerebilir. Dizin geçişleri için / de konulabilir. Açış modu şunlar olabilir :

Mode	Anlamı
"w"	Dosyayı yazmak için açar. Dosyadan okuma yapılamaz. Dosyanın mevcut olması zorunlu değildir. Dosya mevcut değilse yaratılır. Dosya mevcut ise sıfırlanır.
"w+"	Dosyayı yazma ve okuma için açar. Dosyanın mevcut olması zorunlu değildir. Dosya mevcut değilse yaratılır. Dosya mevcut ise sıfırlanır.
"r"	Dosyayı okuma için açar. Dosyaya yazma yapılamaz. Dosya mevcut değilse açılmaz.
"r+"	Dosyayı okuma ve yazma için açar. (Dosyanın pozisyon göstericisi dosyanın başındadır). Dosya mevcut değilse açılmaz.
"a"	Dosyayı sonuna ekleme için açar. Dosyadan okuma yapılamaz. Dosyanın mevcut olması zorunlu değildir. Dosya mevcut değilse yaratılır.
"a+"	Dosyayı sonuna ekleme ve dosyadan okuma için açar. Dosyanın mevcut olması zorunlu değildir. Dosya mevcut değilse yaratılır.

Fonksiyonun geri dönüş değerine ilişkin FILE yapısı stdio.h içerisinde bildirilmiştir. Fonksiyonun geri dönüş değeri FILE yapısı türünden bir adrestir. Bu yapının elemanları standart değildir. Sistemden sisteme değişiklik gösterebilir. Zaten programcı bu yapının elemanlarına gereksinim duymaz. fopen fonksiyonu işletim sisteminin dosya aç sistem fonksiyonunu çağırarak dosyayı açar ve dosyaya ilişkin bazı bilgileri bu FILE yapısı içerisine yazarak bu yapının başlangıç adresini geri döndürür. Örneğin "file handle" değeri de bu yapının içerisinde. Tabi fopen fonksiyonunun geri verdiği FILE türünden adres güvenli bir adrestir. Dosya çeşitli sebeplerden dolayı açılmayabilir. Bu durumda fopen NULL gösterici döndürür. Fonksiyonun geri dönüş değeri kesinlikle kontrol edilmelidir. Tipik bir kontrol işlemi aşağıdaki gibi yapılabilir:

```
if ((f = fopen("data", "r")) == NULL) {
```

```
printf("cannot open file"...\n);
exit(1);
}
```

Dosya ismi dosyanın yeri hakkında bilgi (sürücü, path gibi) içerebilir. Dosya ismi string ile veriliyorsa path bilgisi verirken dikkatli olmak gerekecektir. path bilgisi \ karakteri içerebilir. string içinde \ karakterinin kullanılması, \ karakterinin onu izleyen karakterle birlikte önceden belirlenmiş ters bölü karakter sabiti olarak yorumlanmasına yol açabilecektir. Örneğin :

```
fopen("C:\source\new.dat", "r");
```

fonksiyon çağırımında derleyici \n karakterini "newline" karakteri olarak yorumlayacak \s karakterini ise "undefined" kabul edecektir. Bu problemten sakınmak \ yerine \\ kullanılmasıyla mümkün olur:

```
fopen("C:\\source\\new.dat", "r");
```

Append modları ("a", "a+") çok kullanılan modlar değildir. Dosyaya yazma durumunda "w" modu ile "a" modu arasında farklılık vardır. "w" modunda dosyada olan ofsetin 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 stringde '+' karakterinin kullanılması durumunda dikkatli olmak gerekir. Okuma ve yazma işlemleri arasında mutlaka ya dosya pozisyon göstericisinin konumlandırılması (mesela fseek fonksiyonu ile) ya da dosyaya ilişkin tampon bellek alanının (buffer) tazelenmesi gerekecektir.

Bir dosyanın açılıp açılmayacağını aşağıdaki şekilde test edebiliriz:

Program komut satırından

```
canopen dosya.dat
```

şeklinde çalıştırıldığında ekrana "xxxxx dosyası açılabilir" ya da "xxxxx dosyası açılmaz" yazacaktır.

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

```
int main(int argc, char *argv[])
{
    FILE *fp;

    if (argc != 2) {
        printf("usage: canopen filename\n");
        return 2;
    }
    if ((fp = fopen(argv[1], "r")) == NULL) {
        printf("%s dosyası açılmaz\n", argv[1]);
        return 1;
    }
    printf("%s dosyası açılabilir\n", argv[1]);
    fclose(fp);
    return 0;
}
```

### **fclose fonksiyonu**

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

Bu fonksiyon açılmış olan bir dosyayı kapatır. Fonksiyon fopen ya da fopen fonksiyonundan elde edilen FILE yapısı türünden adresi parametre olarak alır ve açık olan dosyayı kapatır.

Fonksiyonun geri dönüş değeri 0 ise dosya başarılı olarak kapatılmıştır. Fonksiyonun geri dönüş değeri EOF ise dosya kapatılamamıştır. EOF stdio.h içinde tanımlanan bir sembolik sabittir ve derleyicilerin çoğunda (-1) olarak tanımlanmıştır. Fonksiyonun başarısı ancak şüphe altında test edilmelidir. Normal şartlar altında dosyanın kapatılmaması için bir neden yoktur.

```
int main()
{
    FILE *f;

    if ((f = fopen("data", "w")) == NULL) {
        printf("cannot open file...\n");
        exit(1);
    }
    fclose(f);
    return 0;
}
```

### **fgetc fonksiyonu**

```
int fgetc(FILE *f);
```

İşletim sisteminin dolayısıyla C'nin yazma ve okuma yapan fonksiyonları yazılan ve okunan miktar kadar dosya göstericisini ilerletirler. fgetc fonksiyonu dosya göstericisinin gösterdiği yerdeki byte'ı okur ve geri dönüş değeri olarak verir. Fonksiyon başarısız olursa , stdio.h dosyası içerisinde sembolik sabit olarak tanımlanmış EOF değerine geri döner. Pek çok derleyici de EOF sembolik sabiti aşağıdaki gibi tanımlanmıştır.

```
#define EOF (-1)
```

fgetc fonksiyonunun geri dönüş değerini **char** türden bir değişkene atamak yanlış sonuç verebilir, bu konuda dikkatli olunmalı ve fonksiyonun 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 kaybedilecek ve ch değişkenine 0xFF değeri atanacaktır. Bu durumda ch değişkeni işaretli **char** türden olduğundan ch değişkeni içinde negatif bir sayının tutulduğu anlamı çıkar.

```
if (ch == EOF)
```

gibi bir karşılaştırma deyiminde, **if** parantezi içerisindeki karşılaştırma işleminin yapılabilmesi için otomatik tür dönüşümü yapılır. Bu otomatik tür dönüşümünde işaretli **int** türüne çevrilecek ch değişkeni FF byte'ı ile beslenecektir. (negatif olduğu için). Bu durumda eşitlik karşılaştırması doğru netice verecek, yani dosyanın sonuna gelindiği (ya da başka nedenden dolayı okumanın yapılamadığı) yorumu yapılacaktır.

Oysa ch değişkeni **int** türden olsaydı, ch değişkenine atanan değer 0x00FF olacak ve bu durumda karşılaştırma yapıldığında ch değişkeni ile EOF değerinin (0xFFFF) eşit olmadığı sonucuna varılacaktır.

Bir dosyanın içeriğini ekrana yazdıran örnek program:

```
#include <stdio.h>

int main()
{
    FILE *f;
    char fname[MAX_PATH];
    int ch;

    printf("Dosya ismi : ");
    gets(fname);
    if (f = fopen(fname, "r")) == NULL) {
        printf("cannot open file...\n");
        exit(1);
    }
    while (ch = fgetc(f)) != EOF)
        putchar(ch);
    fclose(f);
    return 0;
}
```

not : maximum path uzunluğu DOS'da 80 UNIX ve WINDOWS'da 256 karakteri geçemez.

### fputc Fonksiyonu

**int fputc(int ch, FILE \*p);**

Bu fonksiyon dosya göstericisinin bulunduğu yere 1 byte bilgiyi yazar. Fonksiyonun 1. parametresi yazılacak karakter, 2. parametresi ise yazılacak dosyaya ilişkin FILE yapısı adresidir. Fonksiyonun geri dönüş değeri EOF ise işlem başarısızdır. Değilse fonksiyon yazılan karakter ile geri döner.

fgetc ve fputc fonksiyonları kullanılarak bir dosyayı kopyalayan örnek program:

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

#define MAX_PATH 80

int main()
{
    FILE *fsource, *fdest;
    char source[MAX_PATH], dest[MAX_PATH];
    int ch;

    printf("kopyalanacak dosya : ");
    gets(source);
    printf("kopya dosya : ");
    gets(dest);
    if ((fsource = fopen(source, "r")) == NULL) {
        printf("cannot open file...\n");
        exit(EXIT_FAILURE);
    }
    if ((fdest = fopen(dest, "w")) == NULL) {
        printf("cannot create file...\n");
        exit(EXIT_FAILURE);
    }
    while ((ch = fgetc(fsource)) != EOF)
        fputc(ch, fdest);
    fclose(fsource);
    fclose(fdest);
    printf("1 file copied...\n");
    return 0;
}
```

## fprintf Fonksiyonu

Bu fonksiyon tıpkı printf fonksiyonu gibidir. Ancak ilk parametresi yazma işleminin hangi dosyaya yapılacağını belirtir. Diğer parametreleri printf fonksiyonun da olduğu gibidir. Özetle printf fonksiyonu ekrana yazar ancak fprintf fonksiyonu 1. parametre değişkeninde belirtilen dosyaya yazar.

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

#define MAX_PATH 80

int main()
{
    FILE *f;
    int i;

    if ((f = fopen("data", "w")) == NULL) {
        printf("cannot open file...\n");
        exit(1);
    }
    for (i = 0; i < 10; ++i)
        fprintf(f, "sayi = %d\n", i);
    fclose(f);

    return 0;
}
```

## fgets fonksiyonu

**char \*fgets(char \*buf, int n, FILE \*f);**

Bu fonksiyon dosya göstericisinin gösterdiği yerden 1 satırlık bilgiyi okur. Fonksiyon dosyadan '\n' karakterini okuyunca onu da birinci parametresinde verilen adrese yazarak işlemini sonlandırır.

Fonksiyonun 1. parametresi okunacak bilginin RAM'de yerleştirileceği yerin adresidir. 2. parametresi ise okunacak maksimum karakter sayısıdır. fgets en fazla n -1 karakteri okur. Okuduğu karakterlerin sonuna null karakteri ekler ve işlemini sonlandırır. Eğer satır üzerindeki karakter sayısı n- 1'den az ise tüm satırı okur ve işlemini sonlandırır. Örneğin bu parametreyi 10 olarak girdiğimizi düşünelim. Satır üzerinde 20 karakter olsun. Fonksiyon 9 karakteri okur, sonuna NULL karakteri ekler. Ancak satır üzerinde \n dahil olmak üzere 5 karakter olsaydı fonksiyon bu 5 karakteri de okuyarak sonuna da NULL karakter ekleyerek işlemini sonlandıracaktı. Bir döngü içerisinde fgets fonksiyonu sürekli olarak çağırılarak bütün dosya okunabilir.

Fonksiyonun geri dönüş değeri, en az 1 karakter okunmuş ise 1. parametresi ile belirtilen adresin aynısı, hiç okunmamışsa NULL adresidir.

**Örnek :** Bir dosyayı fgets fonksiyonu ile satır satır olarak ekrana yazan program:

```
#define MAX_PATH 80
#define MBUFSIZE 100

int main()
{
    FILE *f;
    char s[MAX_PATH];
    char buf[BUFSIZE];

    printf("Dosya : ");
    gets(s);
}
```



```

    if ((f = fopen(s, "r")) == NULL) {
        printf("cannot open file...\n");
        exit(1);
    }
    while (fgets(buf, BUFSIZE, f) != NULL)
        printf("%s", buf);
    fclose(f);
    return 0;
}

```

## Text ve Binary Dosya Kavramları

DOS VE WINDOWS işletim sistemlerinde bir dosya text ve binary modda açılabilir. Varsayılan açış modu textdir. Yani dosyanın hangi modda açıldığı açık bir şekilde belirtilmezse dosyanın text modda açıldığı varsayılacaktır. Dosyayı binary modda açabilmek için açış modunun sonuna b eklemek gerekir. Örneğin :

f = fopen("data", "r") text modda  
f = fopen("data", "rb") binary modda

DOS'da bir dosya type edildiğinde 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 type edildiğinde görüntü

```

a
b

```

şeklinde olsun. Dosyadaki durum a\r\nb şeklindedir. Oysa UNIX sistemlerinde aşağı satırın başına geçebilmek için sadece LF karakteri kullanılmaktadır. UNIX de

```

a
b

```

görüntüsünün dosya karşılığı

a\nb biçimindedir.

DOS'da LF karakteri bulunulan satırın aşağısına geç CR karakteri ise bulunulan satırın başına geç anlamındadır. Örneğin DOS da bir dosyanın içeriği a\nb biçiminde ise dosya type edildiğinde

```

a
b

```

görüntüsü elde edilir. Eğer dosyanın içeriği a\rb biçiminde ise dosya type edildiğinde

```

b
görüntüsü elde edilir.

```

printf fonksiyonunda \n ekranda aşağı satırın başına geçme amacıyla kullanılır. Aslında biz printf fonksiyonunun 1. parametresi olan stringin içine \n yerleştirdiğimizde UNIX'de yalnızca \n DOS'da 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 fonksiyonları otomatik olarak \r ve \n karakterlerini dosyaya yazarlar. Benze r 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 olmayacaktır. 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 modda açıldığını düşünelim. Bu dosyada 10 numaralı ve 13 numaralı ASCII karakterleri yanyana bulunduğu dosyadan yalnızca 1 byte okunacaktır. Aynı şekilde dosyadan 26 numaralı ASCII karakteri okunduğunda dosyadan artık başka bir okuma yapılamayacaktır. (Dosyanın sonuna gelindiği varsayılacaktır.)

```
/*
textmode.c programı
DOS altında text modu ile binary mod arasındaki
farkı gösteren bir program
*/

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

main()
{
    FILE *fp;
    int k, ch;

    clrscr();
    fp = fopen("deneme", "w");
    if (fp == NULL) {
        printf("dosya açılmıyor\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 açılmıyor\n");
        exit(EXIT_FAILURE);
    }
    while ((ch = fgetc(fp)) != EOF)
        printf("%d ", ch);

    /* ekran çıktısı
13 10 13 10 13 10 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 açılmıyor\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
test ediliyor */

fp = fopen("deneme", "w");
if (fp == NULL) {
    printf("dosya açılmıyor\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 açılmıyor\n");
    exit(EXIT_FAILURE);
}
while ((ch = fgetc(fp)) != EOF)
    printf("%d ", ch);

/* ekran çıktısı
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 açılmıyor\n");
    exit(EXIT_FAILURE);
}
while ((ch = fgetc(fp)) != EOF)
    printf("%d ", ch);

/* ekran çıktısı
65 65 65 65 26 65 65 65 65 65
*/

fclose(fp);

return 0;
}
```

## EOF (END OF FILE) DURUMU

Dosyanın sonunda hiçbir özel karakter yoktur. İşletim sistemi dosyanın sonuna gelinip gelinmediğini dosyanın uzunluğuna bakarak anlayabilir. EOF durumu dosya pozisyon göstericisinin dosyada olmayan son karakteri göstermesi durumudur. EOF durumunda dosya pozisyon göstericisinin offset değeri dosya uzunluğu ile aynı değerdedir. EOF durumunda dosyadan okuma yapılmak istenirse dosya fonksiyonları başarısız olur. Ancak açış modu uygunsa dosyaya yazma yapılabilir ve bu durumda dosyaya ekleme yapılır.

Daha önce söylendiği gibi C dilinde açılan bir dosya ile ilgili bilgiler FILE türünden bir yapı nesnesi içinde tutulur. Bu yapının her bir elemanı dosyanın bir özelliği hakkında bilgi vermektedir. C programcısı bu yapının elemanlarının değerleri ile doğrudan ilgilenmez, zira fopen fonksiyonunun geri dönüş değeri bu yapı nesnesini gösteren FILE yapısı türünden bir göstericidir ve C dilinin dosyalarla ilgili işlem yapan fonksiyonları çoğunlukla bu adresi parametre olarak alarak, istenilen dosyaya ulaşırlar.

Söz konusu FILE yapısının elemanlarından biri de flag olarak kullanılan EOF indikatörüdür. (aslında int türünden bir flag'in yalnızca belirli bir bitidir.) C dilinin dosyalarla ilgili işlem yapan bazı fonksiyonları EOF indikatörünün değerini değiştirirler. (set ya da clear ederler. Set edilmesi bu indikatöre 1 değerinin atanması clear edilmesi ise 0 değerinin atanmasıdır. Bu indikatörün set edilmesi demek dosya pozisyon göstericisinin dosyanın sonunu göstermesi demektir.

Dosya açan fonksiyonlar FILE yapısındaki EOF indikatörünü sıfırlarlar. Bu fonksiyonlar dışında fseek fonksiyonu ve clearerr fonksiyonları da EOF indikatörünü sıfırlarlar. (clear ederler).

fseek fonksiyonu :

Bu fonksiyon dosya pozisyon göstericisini istenilen bir offsete konumlandırmak amacıyla kullanılır. Bu fonksiyonun kullanılmasıyla, açılmış bir dosyanın istenilen bir yerinden okuma yapmak ya da istenilen bir yerine yazmak mümkün hale gelir. Prototipi :

```
int fseek(FILE *f, long offset, int origin);
```

Fonksiyonun ikinci parametresi konumlandırma işleminin yapılacağı offset değeridir.

Fonksiyonun 3. parametresi 0, 1, veya 2 olabilir. Bu değerler stdio.h dosyasında

```
#define SEEK_SET    0
#define SEEK_CUR    1
#define SEEK_END    2
```

biçiminde sembolik sabitlerle tanımlanmıştır ve fseek fonksiyonun çağırılmasında daha çok bu sembolik sabitler kullanılmaktadır.

son parametre 0 ise, konumlandırma dosya başından itibaren yapılır. Bu durumda 2. parametre  $\geq 0$  olmalıdır. Örneğin:

```
fseek(f, 10L, 0);
```

ile dosya göstericisi 10. offsete konumlandırılır. Ya da

```
fseek(f, 0, 0);
```

ile dosya göstericisi dosyanın başına konumlandırılır. Dosya pozisyon göstericisinin dosyanın başına konumlandırılması için rewind fonksiyonu da kullanılabilir:

```
void rewind(FILE *fp);
```

```
rewind(f);
```

fonksiyonun 3. parametre değişkenine geçilen değer 1 ise (SEEK\_CUR) , 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 gelecektir. Örneğin dosya göstericisi 10. offsette olsun.

```
fseek(f, -1, SEEK_CUR);
```

çağırması ile dosya göstericisi 9. offset'e konumlandırılır.

fonksiyonun 3. parametre değişkenine geçilen değer 2 ise (SEEK\_END), konumlandırma EOF durumundan itibaren yani dosya sonunu referans alınarak yapılır. Bu durumda ikinci parametre  $\leq 0$  olmalıdır. Örneğin dosya göstericisini EOF durumuna çekmek için :

```
fseek(f, 0, SEEK_END);
```

çağırmasını yapmak gerekir. Ya da başka bir örnek:

```
fseek(f, -1, SEEK_END);
```

çağırması ile dosya göstericisi son karakterin offsetine çekilir. Fonksiyonun geri dönüş değeri işlemin başarısı hakkında bilgi verir. Geri dönüş değeri 0 ise işlem başarılıdır. Geri dönüş değeri 0 dışı bir değer ise işlem başarısızdır. Ancak problemlili durumlarda geri dönüş değrinin test edilmesi tavsiye edilir.

Yazma ve okuma işlemleri arasında dosya göstericisinin fseek fonksiyonu ile konumlandırılması gerekir. Konumlandırma gerekirse boş bir fseek çağırması ile yapılabilir. Örneğin dosyadan bir karakter okunup , bir sonraki karaktere bu karakterin 1 fazlasını yazacak olalım.

```
ch = fgetc(f);  
fputc(ch + 1, f);
```

işlemi hatalıdır. Yazmadan okumaya, okumadan yazmaya geçişte dosya göstericisi konumlandırılmalıdır.

feof fonksiyonu :

bu fonksiyon dosya göstericisinin EOF durumunda olup olmadığını (EOF indikatörünün set edilip edilmediğini) test etmek amacıyla kullanılır. Prototipi:

```
int feof(FILE *f);
```

Eğer dosya göstericisi dosya sonunda ise (EOF indikatörü set edilmişse) fonksiyon 0 dışı bir değere , değilse (EOF indikatörü set edilmemişse) 0 değeri ile geri döner.

Ancak feof fonksiyonunun 0 dışı bir değere geri dönebilmesi için dosya göstericisinin dosyanın sonunda olmasının yanı sıra en son yapılan okuma işleminin de başarısız olması gerekir. daha önce söylendiği gibi bazı fonksiyonlar (fopen, fseek, rewind, clearerr) EOF indikatörünü clear ederler, yani EOF indikatörünün 0 değerinde olması (clear edilmiş olması) dosyanın sonunda olunmadığının bir garantisi değildir.

fread ve fwrite fonksiyonları

Bu iki fonksiyon C dilinde en çok kullanılan dosya fonksiyonlarıdır. Genel olarak dosya ile RAM arasında transfer yaparlar. Her iki fonksiyonun da prototipleri aynıdır.

```
size_t fread(void *adr, size_t size, size_t n, FILE *);  
size_t fwrite (const void *adr, size_t size, size_t n, FILE *);
```

size\_t türünün derleyiciyi yazarlar tarafından unsigned int ya da unsigned long türünün typedef edilmiş yeni ismi olduğunu hatırlayalım.

fread fonksiyonu dosya pozisyon göstericisinin gösterdiği yerden, 2. ve 3. parametresine kopyalanan değerlerin çarpımı kadar byte'ı , RAM'de 1. parametresinin gösterdiği adresten başlayarak kopyalar. Geleneksel olarak fonksiyonun 2. parametresi veri yapısının bir elemanının uzunluğunu, 3. parametresi ile parça sayısı biçiminde girilir.

Bu fonksiyonlar sayesinde diziler ve yapılar tek hamlede dosyaya transfer edilebilirler. Örneğin 10 elemanlı bir dizi aşağıdaki gibi tek hamlede 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 fonksiyonuna 1. arguman olarak gönderilebilir. FILE türünden f göstericisi ile ilişkilendirilen dosyaya RAM'deki a adresinden toplam sizeof(int) \* 5 byte yazılmaktadır.

Ancak tabi fwrite fonksiyonu sayıları bellekteki görüntüsü ile dosyaya yazar. (yani fprintf fonksiyonu gibi formatlı yazmaz.) Örneğin:

```
int i = 1535;
```

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

Burada dosya type edilirse 2 byte uzunluğunda rasgele karakterler görünür. Çünkü DOS'da int türü 2 byte uzunluğundadır. Bizim gördüğümüz ise 1525'in rasgele olan byteleridir. Bilgileri ASCII karşılıkları ile dosyaya yazmak için fprintf fonksiyonu kullanılabilir..

fread ve fwrite fonksiyonları bellekteki bilgileri transfer ettiğine göre dosyaların da binary modda açılmış olması uygun olacaktır.

```
#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]);
}
```

fread ve fwrite fonksiyonlarının geri dönüş değerleri 3. parametresi ile belirtilen okunan ya da yazılan parça sayısıdır. Örneğin

```
n = fread(a, sizeof(int), 10, f);
```

ifadesinde fonksiyon bütün sayıları okuyabildiyse 10 sayısına geri döner. Eğer dosyadaki kalan byte sayısı okunmak istenen sayıdan az ise fonksiyon bütün byte'ları okur ve geri dönüş değeri

okunan byte sayısı 2. parametresi ile belirtilen sayı olur. Örneğin DOS altında çalışıyor olalım. Dosyada 10 byte bilgi kalmış olsun.

```
n = fread(a, sizeof(int), 10, f);
```

ile fonksiyon 5 sayısına geri dönecektir.

Aşağıdaki iki ifadeyi inceleyelim:

```
fread(str, 100, 1, f);
fread(str, 1, 100, f);
```

her iki fonksiyon çağırma ifadesi de RAM'deki str adresine FILE türünden f göstericisi ile ilişkilendirilen dosyadan 100 byte okumak amacıyla kullanılabilir. Ancak birinci çağırma geri dönüş değeri ya 0 ya 1 olabileceken, ikinci fonksiyon çağırmasında geri dönüş değeri 0 100(dahil) herhangi bir değer olabilecektir.

Blok blok kopyalama işlemi

Aşağıdaki örnekte bir grup byte fread fonksiyonu ile bir dosyadan okunmuş ve fwrite fonksiyonu ile diğer bir dosyaya yazılmıştır.

```
#define BLOCK_SIZE    1024
#define MAX_PATH      80

int main()
{
    FILE *fs, *fd;
    char s[MAX_PATH], d[MAX_PATH];
    unsigned n;

    printf("kaynak dosya : ");
    gets(s);
    printf("amaç dosya : ");
    gets(d);
    if ((fs = fopen(s, "rb")) == NULL) {
        printf("cannot open file ...\n");
        exit(EXIT_FAILURE);
    }
    if ((fd = fopen(d, "wb")) == NULL) {
        printf("cannot open file ...\n");
        fclose(fs);
        exit(EXIT_FAILURE);
    }
    while ((n = fread(buf, 1, BLOCK_SIZE, fs)) > 0)
        fwrite(buf, 1, n, fd);
    fclose(fs);
    fclose(fd);
    printf("1 file copied...\n");
    return 0;
}
```

remove fonksiyonu

Bu fonksiyon bir dosyayı silmek için kullanılır. Fonksiyonun prototipi :

```
int remove (const char *filename);
```

şeklindedir. Fonksiyona arguman olarak silinecek dosyanın ismi gönderilir. Fonksiyonun geri dönüş değeri, dosyanın başarılı bir şekilde silinebilmesi durumunda 0 aksi halde (yani dosya silinmemişse) 0 dışı bir değerdir. Açık olan bir dosyanın silinmesi "implementation depended" (derleyiciye bağlı) olduğundan, yazılan kodun taşınabilirliği açısından, silinecek bir dosya açık ise önce kapatılmalıdır.

rename fonksiyonu

Bu fonksiyon bir dosyanın ismini değiştirmek için kullanılır. Fonksiyonun prototipi :

```
int rename (const char *old, const char *new);
```

şeklindedir.

Fonksiyona 1. arguman olarak dosyanın eski ismi ikinci arguman olarak ise dosyanın yeni ismi gönderilmelidir. Fonksiyonun geri dönüş değeri, isim değiştirmen işleminin başarılı olması durumunda 0, aksi halde (dosyanın ismi değiştirilemiyorsa 0 dışı bir değerdir. (Örneğin açık olan bir dosyanın isminin değiştirilmeye çalışılması başarılı olamayacağından fonksiyon bu durumda 0 dışı bir değere geri dönecektir.)

tmpfile fonksiyonu

Fonksiyon geçici bir dosya açmak amacıyla kullanılır. Fonksiyonun prototipi :

```
FILE * tmpfile(void);
```

tmpfile fonksiyonu açtığı geçici dosyayı "wb" modunda açar. Açılan dosya fclose fonksiyonu ile kapatıldığında ya da (kapatılmazsa), program sona erdiğinde otomatik olarak silinecektir. Fonksiyonu geri dönüş değeri, açılan geçici dosya ile ilişki kurulmasına yarayacak, FILE yapısı türünden bir adrestir. Herhangi bir nedenle dosya geçici dosya açılmıyorsa fonksiyon NULL adresine geri dönecektir.

tmpnam fonksiyonu

Geçici olarak kullanılacak bir dosya için bir isim üretilmesi amacıyla kullanılır. Fonksiyonun prototipi :

```
char *tmpnam(char *s);
```

şeklindedir. Fonksiyon ürettiği dosya ismini kendisine gönderilen char türden adrese yerleştirir. Eğer fonksiyona arguman olarak NULL adresi gönderilirse, fonksiyon üretilen dosya ismini statik bir dizi içinde tutarak bu dizinin adresiyle geri dönecektir. Fonksiyona char türden bir dizinin adresi gönderildiğinde bu dizinin boyutu ne kadar olmalıdır. Başka bir deyişle tmpnam fonksiyonu kaç karakter uzunluğunda bir dosya ismi üretecektir. İşte bu değer stdio.h dosyası içinde tanımlanan L\_tmpnam sembolik sabitiyle belirtilmektedir.

tmpnam fonksiyonunun ürettiği dosya isminin daha önce kullanılmayan bir dosya ismi olması garanti altına alınmıştır. Yani üretilen dosya ismi tektir. (unique file name) Bir programda daha sonra silmek üzere bir dosya açacağımızı ve bu dosyaya birtakım bilgileri yazacağımız düşünelim. Bu durumda dosyayı yazma modunda açacağımıza göre, açacağımız dosyaya olan bir dosyanın ismini veremeyiz. Eğer verirsek , var olan dosya sıfırlanacağı için bu dosyayı kaybederiz. Bu riske girmemek için, geçici olarak kullanılacak dosya tmpfiel fonksiyonu kullanılarak açılmalıdır. Ancak tmpfile fonksiyonunun kullanılması durumunda, açılan dosya kalıcı hale getirilemez. Yani herhangi bir nedenden dolayı geçici dosyanın silinmemesini istersek, (dosyayı kalıcı hale getirmek istersek) dosyayı fopen fonksiyonuyla açmalıyız. İşte bu durumda geçici dosyayı başka bir dosyayı riske etmemek için tmpnam fonksiyonunun ürettiği isim ile açmalıyız. Peki tmpnam fonksiyonuyla en fazla kaç tane "unique file name " üretebiliriz. İşte bu sayı stdio.h içinde tanımlanan TMP\_MAX sembolik sabiti ile belirlenmiştir.



C dilinde bazı giriş ve çıkış birimleri(klavye, ekran gibi) doğrudan bir dosya gibi ele alınırlar. C standartları herhangi bir giriş çıkış birimini "stream" olarak isimlendirmektedir. Bir stream bir dosya olabileceği gibi, dosya olarak ele alınan bir giriş çıkış birimi de olabilir. Örneğin küçük programlar genellikle girdilerini genellikle tek bir stream,den alıp (mesela klavye) çıktılarını da tek bir streame (mesela ekran) iletirler.

freopen fonksiyonu

freopen fonksiyonu daha önce açılmış bir dosyayı, fopen fonksiyonu ile açılan dosyaya yönlendirir. Fonksiyonun prototipi :

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

şeklindedir.

Uygulamalarda daha çok standart dosyaların (stdin, stdout, stderr) başka dosyalara yönlendirilmesinde kullanılır. Örneğin bir programın çıktılarının data.dat isimli dosyaya yazılmasını istersek :

```
if (freopen("data.dat", "w", stdout) == NULL) {
    printf("data.dat dosyası açılmıyor\n");
    exit(EXIT_FAILURE);
}
```

yukarıdaki fonksiyon çağırımıyla stdout dosyasının yönlendirildiği dosya kapatılarak (bu yönlendirme işlemi komut satırından yapılmış olabileceği gibi, freopen fonksiyonunun daha önceki çağırımı ile de yapılmış olabilir.) stdout dosyasının data.dat dosyasına yönlendirilmesi sağlanır.

freopen fonksiyonunun geri dönüş değeri fonksiyona gönderilen üçüncü arguman olan FILE yapısı türünden göstericidir. freopen dosyası yönlendirmenin yapılacağı dosyayı açamazsa NULL adresine geri döner. Eğer yönlendirmenin yapıldığı eski dosya kapatılamıyorsa, freopen fonksiyonu bu durumda bir işaret vermez.

dosya buffer fonksiyonları

İkincil belleklerle (disket, hard disk vs.) yapılan işlemler RAM'de 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 performansı bufferlama yoluyla artırılmaktadır. Bir dosyaya yazılacak data ilk önce bellekteki bir buffer alanında saklanır. Bu buffer alanı dolduğunda ya da yazılmanın yapılacağı dosya kapatıldığında bufferdaki data alanında ne varsa dosyaya yazılır. Buna bufferın boşaltılması (to flush the buffer) denir.

Giriş dosyaları da benzer şekilde bufferlanabilir. Giriş biriminden alınan data (örneğin klavyeden) önce buffera yazılır.

dosyaların bufferlaması erimlilikte çok büyük bir artışa neden olur. Çünkü bufferdan (RAM'den) bir karakter okunması ya da buffera bir karakter yazılması ihmal edilecek kadar küçük bir zaman içinde yapılır. Buffer ile dosya arasındaki transfer şüphesiz yine vakit alacaktır ama bir defalık blok transferi, küçük küçük transferlerin toplamından çok daha kısa zaman alacaktır.

stdio.h başlık dosyası içinde prototipi bildirimi yapılan ve dosyalarla ilgili işlem yapan fonksiyonlar tamponlamayı otomatik olarak gerçekleştirirler. Yani dosyaların tamponlanması için bizim birşey yapmamıza gerek kalmadan bu iş geri planda bize sezdirilmeden yapılmaktadır. Ama bazı durumlarda tamponlama konusunda programcı belirleyici durumda olmak isteyebilir. İşte bu durumlarda programcı dosya tamponlama fonksiyonlarını (fflush, setbuf, setvbuf) kullanacaktır:

fflush fonksiyonu

Bir program çıktısını bir dosyaya yazarken (örneğin stdout dosyasına) yazılan dosya ilk önce RAM'deki tamponlama alanına gider. Dosya kapatıldığında ya da tamponlama alanı dolduğunda, tamponlama alanı boşaltılarak dosyaya yazılır. fflush fonksiyonunun kullanılmasıyla, dosyanın kapatılması ya da tamponlama alanının dolması beklenmeksizin, tamponlama alanı boşaltılarak dosyaya yazılır. Bu işlem istenilen sıklıkta yapılabilir. Fonksiyonun prototipi :

```
int fflush (FILE *stream);
```

şeklindedir.

```
fflush(fp);
```

çağırımı ile FILE yapısı türünden fp göstericisi ile ilişkilendirilen dosyanın tamponlama alanı (buffer) boşaltılır. Eğer fflush fonksiyonuna NULL adresi gönderilirse, açık olan bütün dosyaların tamponlama alanları boşaltılacaktır.

Tamponlama alanının boşaltılması işlemi başarılı olursa fflush fonksiyonu 0 değerine geri dönecek aksi halde EOF değerine geri dönecektir.

setvbuf fonksiyonu

setvbuf fonksiyonu bir dosyanın tamponlanma şeklinin değiştirilmesi ve tampon alanının yerinin ve boyutunun değiştirilmesi amacıyla kullanılır. Fonksiyonun prototipi aşağıdaki şekildedir :

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Fonksiyona gönderilen üçüncü arguman tamponlama şeklini belirler. Üçüncü argumanın değeri stdio.h başlık dosyası içinde tanımlanan sembolik sabitlerle belirlenir.

\_IOFBF (full buffering - tam tamponlama)

data 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 dosyaya yazma tamponlama olmadan doğrudan yapılır.

setvbuf fonksiyonuna gönderilen ikinci arguman RAM'de tamponlamanın yapılacağı bloğun başlangıç adresidir. Tamponlamanın yapılacağı alan statik ya da dinamik ömürlü olabileceği gibi, dinamik bellek fonksiyonlarıyla da tahsis edilebilir.

Fonksiyona gönderilen son arguman tamponlama alanında tutulacak bytelerin sayısıdır.

setvbuf fonksiyonu dosya açıldıktan sonra, fakat dosya üzerinde herhangi biri işlem yapılmadan önce çağırılmalıdır. Fonksiyonun başarılı olması durumunda fonksiyon 0 değerine geri dönecektir. Fonksiyona gönderilen üçüncü argumanın geçersiz olması durumunda ya da fonksiyonun ilgili tamponlamayı yapamaması durumunda, geri dönüş değeri 0 dışı bir değer olacaktır.

Fonksiyona gönderilen buffer alanının geçerliliğinin bitmesinden önce (ömrünün tamamlanmasından önce) dosya kapatılmamalıdır.

dosya işlemleri ile ilgili örnek uygulamalar:

Uygulama 1 : Dosya üzerinde sıralı veri tabanı tutulması

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

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

/* symbolic constants */

#define MAX_PATH      80
#define ADDREC        1
#define LISTREC        2
#define FINDREC        3
#define DELREC         4
#define EDITREC        5
#define PACKRECS       6
#define SORTREC        7
#define EXITPROG       8
#define DELETED        0
#define NORMALREC      1
#define INVALID        0

/* structure declarations */

typedef struct _PERSON {
    char name[30];
    int no;
    int delflag;
} PERSON;

/* function prototypes */

int  GetOption(void);
void AddRec(void);
void ListRec(void);
void FindRec(void);
void DelRec(void);
void EditRec(void);
void SortRec(void);
void Pack(void);

/* global variables */

FILE *f;
char fname[MAX_PATH];

/* function definitions */

void AddRec(void)
{
    PERSON per;

    printf("Adı soyadı : ");
    fflush(stdin);
    gets(per.name);
    printf("No : ");
    scanf("%d", &per.no);
    per.delflag = NORMALREC;
    fseek(f, 0, SEEK_END);
    fwrite(&per, sizeof(PERSON), 1, f);
}

void FindRec(void)
{
    PERSON per;
```

```

    char name[30];

    printf("l\u00fctfen kayıt ismini giriniz :");
    fflush(stdin);
    gets(name);
    fseek(f, 0, SEEK_SET);
    while (fread(&per, sizeof(PERSON), 1, f) > 0) {
        if (per.delflag != NORMALREC)
            continue;
        if (!strcmp(per.name, name)) {
            printf("\n%s    %d\n\n", per.name, per.no);
            return;
        }
    }
    printf("\nKayıt bulunamadı...\n\n");
}

void ListRec(void)
{
    PERSON per;

    putchar('\n');
    fseek(f, 0L, SEEK_SET);
    while (fread(&per, sizeof(PERSON), 1, f) > 0) {
        printf("\n%20s    %5d", per.name, per.no);
        if (per.delflag == DELETED)
            printf("\tDELETED");
    }
    putchar('\n\n');
}

void DelRec(void)
{
    char name[30];
    PERSON per;

    printf("Silinecek kaydın adı ve soyadı : ");
    fflush(stdin);
    gets(name);
    fseek(f, 0, SEEK_SET);
    while (fread(&per, sizeof(PERSON), 1, f) > 0) {
        if (!strcmp(per.name, name)) {
            per.delflag = DELETED;
            fseek(f, -(long)sizeof(PERSON), 1);
            fwrite(&per, sizeof(PERSON), 1, f);
            printf("Record deleted!...\n");
            return;
        }
    }
    printf("silinecek kayıt bulunamadı");
}

void EditRec(void)
{
    char name[30];
    PERSON per;

    printf("Değiştirilecek kaydın adı ve soyadı : ");
    fflush(stdin);
    gets(name);
    fseek(f, 0, SEEK_SET);
    while (fread(&per, sizeof(PERSON), 1, f) > 0) {
        if (per.delflag == NORMALREC && !strcmp(per.name, name)) {
            printf("Adı soyadı : ");

```

```

        fflush(stdin);
        gets(per.name);
        printf("No : ");
        scanf("%d", &per.no);
        fseek(f, -(long)sizeof(PERSON), 1);
        fwrite(&per, sizeof(PERSON), 1, f);
        printf("Record updated!...\n");
        return;
    }
}
printf("değiştirilecek kayıt bulunamadı\n");
}

int GetOption(void)
{
    int option;

    printf("\n1) Kayıt Ekle\n");
    printf("2) Kayıt Listele\n");
    printf("3) Kayıt Bul\n");
    printf("4) Kayıt Sil\n");
    printf("5) Kayıt değiştir\n");
    printf("6) Pack\n");
    printf("7) Sırala\n");
    printf("8) Çık\n");
    printf("\nSeçiminiz :");
    scanf("%d", &option);
    if (option < 0 || option > 8)
        return INVALID;
    return option;
}

void SortRec(void)
{
    PERSON per[2], tmp;
    int i, count, chgFlag;

    fseek(f, 0, SEEK_END);
    count = ftell(f) / sizeof(PERSON);

    do {
        chgFlag = 0;
        for (i = 0; i < count - 1; ++i) {
            fseek(f, (long)i * sizeof(PERSON), SEEK_SET);
            if (fread(per, sizeof(PERSON), 2, f) != 2) {
                printf("cannot read from the file!...\n");
                exit(EXIT_FAILURE);
            }
            if (per[0].no > per[1].no) {
                chgFlag = 1;

                tmp = per[0];
                per[0] = per[1];
                per[1] = tmp;

                fseek(f, (long)i * sizeof(PERSON), SEEK_SET);
                if (fwrite(per, sizeof(PERSON), 2, f) != 2) {
                    printf("cannot read from the file!...\n");
                    exit(EXIT_FAILURE);
                }
            }
        }
    }
    while (chgFlag);
}

```

```

}

void Pack(void)
{
    FILE *fnew;
    PERSON per;

    if ((fnew = fopen("temp", "wb")) == NULL) {
        printf("cannot create temporary file!..\n");
        exit(EXIT_FAILURE);
    }
    fseek(f, 0l, SEEK_SET);
    while (fread(&per, sizeof(PERSON), 1, f) > 0) {
        if (per.delflag == NORMALREC)
            fwrite(&per, sizeof(PERSON), 1, fnew);
    }
    fclose(fnew);
    fclose(f);
    if (unlink(fname)) {
        printf("Fatal Error : Cannot open database file!..\n");
        exit(EXIT_FAILURE);
    }
    if (rename("temp", fname)) {
        printf("fatal Error: cannot delete database file!..\n");
        exit(EXIT_FAILURE);
    }
    if ((f = fopen(fname, "r+b")) == NULL) {
        printf("Fatal Error : Cannot open database file!..\n");
        exit(EXIT_FAILURE);
    }
    printf("Pack operation succesfully completed!..\n");
}

void main()
{
    char dfname[MAX_PATH];
    int option;

    printf("Data File : ");
    gets(dfname);
    if ((f = fopen(dfname, "r+b")) == NULL)
        if ((f = fopen(dfname, "w+b")) == NULL) {
            printf("Cannot open database file!..\n");
            exit(EXIT_FAILURE);
        }
    strcpy(fname, dfname);
    for (;;) {
        option = GetOption();
        switch (option) {
            case ADDREC : AddRec(); break;
            case LISTREC : ListRec(); break;
            case FINDREC : FindRec(); break;
            case DELREC : DelRec(); break;
            case EDITREC : EditRec(); break;
            case PACKRECS : Pack(); break;
            case SORTREC : SortRec(); break;
            case EXITPROG : goto EXIT;
            case INVALID : printf("Geçersiz Seçenek!..\n");
        }
    }
    EXIT:
    fclose(f);
}

```

## Uygulama 2: developer's back up programı (G. Aslan)

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <io.h>
#include <errno.h>

#define GOOD          0
#define FAIL          (-1)

#define FBUFSIZ        (63*512)          /* Buffer size */

char    *Buffer, *in_buf, *out_buf;
char    CmpFile[128], OutFile[128];

int filecmp(FILE *fi, FILE *fo)
{
    int c1, c2;
    long l1, l2;

    l1 = filelength(fileno(fi));
    l2 = filelength(fileno(fo));
    if (l1 != l2)
        return FAIL;

    rewind(fi);
    rewind(fo);
    for (;;) {
        c1 = getc(fi);
        c2 = getc(fo);

        if (c1 != c2) {
            return FAIL;
        }

        if (c1 == EOF)
            break;
    }

    return GOOD;
}

int filecopy(FILE *fi, FILE *fo)
{
    int c, nbytes;

    rewind(fi);
    rewind(fo);
    for (;;) {
        c = fread(Buffer, 1, FBUFSIZ, fi);
        if (c == 0) {
            break;
        }

        nbytes = fwrite(Buffer, 1, c, fo);
        if (nbytes != c) {
            return FAIL;
        }
    }
    return GOOD;
}

```

```
int main(int argc, char *argv[])
{
    FILE *fi, *fo;
    char *ptr;
    int i;

    printf("*GA Developer's Backup Utility. Version 1.0\n"
           "(C) *GA, 1995\n\n");

    if (argc != 2) {
        fprintf(stderr, "Usage: BK <filename>\n");
        return 1;
    }

    if ((Buffer = malloc(FBUFSIZ)) == NULL ||
        (in_buf = malloc(FBUFSIZ)) == NULL ||
        (out_buf = malloc(FBUFSIZ)) == NULL) {
        fprintf(stderr, "Not enough memory\n");
        return 2;
    }

    if ((fi = fopen(argv[1], "rb")) == NULL) {
        ptr = argv[1];
    OPN_ERR:
        fprintf(stderr, "File could not be opened: '%s'\n", ptr);
        return 3;
    }
    setvbuf(fi, in_buf, _IOFBF, FBUFSIZ);

    strcpy(CmpFile, argv[1]);
    ptr = strchr(CmpFile, '.');
    if (ptr == NULL)
        ptr = strchr(CmpFile, '\\0');

    for (i = 1; i <= 999; ++i) {
        sprintf(ptr, "%.03d", i);
        if (access(CmpFile, 0))
            break;
    }

    if (i == 1000) {
        fprintf(stderr, "Backup operation failed: File limit!\n");
        return 3;
    }

    strcpy(OutFile, CmpFile);

    if (i > 1) {
        sprintf(ptr, "%.03d", i-1);

        if ((fo = fopen(CmpFile, "rb")) == NULL) {
            ptr = CmpFile;
            goto OPN_ERR;
        }
        setvbuf(fo, out_buf, _IOFBF, FBUFSIZ);

        if (!filecmp(fi, fo)) {
            printf("No differences encountered: '%s'\n", CmpFile);
            return 0;
        }
        fclose(fo);
    }

    printf("File being copied: %s ---> %s\n", argv[1], OutFile);
```



```

    if ((fo = fopen(OutFile, "wb")) == NULL) {
        ptr = OutFile;
        goto OPN_ERR;
    }
    setvbuf(fo, out_buf, _IOFBF, FBUFSIZ);

    if (filecopy(fi, fo)) {
        fprintf(stderr, "File copy error!\n");
        return 4;
    }

    fcloseall();

    return 0;
}

```

### Uygulama 3 : bol.c ve bir.c programları :

bir dosyanın belirkli byte büyüklüğünde n kadar sayıda dosyaya bölünmesi ve daha sonra başka bir programla bu dosyaların tekrar birleştirilmesi.

/\*\*\*\*\*bol.c \*\*\*\*\*/

```

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

#define      MAX_LEN      80

int main(int argc, char **argv)
{
    FILE *fs, *fd;
    char fsname[MAX_LEN];
    char fdname[MAX_LEN] = "dos0001.xxx";
    int chunk;
    long no_of_chars = 0L;
    int no_of_files = 0;
    int ch;

    if (argc != 3) {
        printf("bolunecek dosyanin ismini giriniz : ");
        gets(fsname);
        printf("kac byte'lik parcalara bolmek istiyorsunuz?");
        scanf("%d", &chunk);
    }
    else {
        strcpy(fsname, argv[1]);
        chunk = atoi(argv[2]);
    }

    printf("%s dosyasi %d uzunlugunda dosyalara bolunecek!\n", fsname, chunk);

    fs = fopen(fsname, "rb");
    if (fs == NULL) {
        printf("%s dosyasi acilamiyor!\n", fsname);
        exit(EXIT_FAILURE);
    }
    fd = NULL;

    while ((ch = fgetc(fs)) != EOF) {

```

```

        if (fd == NULL) {
            fd = fopen(fdname, "wb");
            if (fd == NULL) {
                printf(" %s dosyasi yaratilamiyor!\n", fdname);
                exit(EXIT_FAILURE);
            }
            no_of_files++;
            printf("%s dosyasi yaratildi!\n", fdname);
        }
        fputc(ch, fd);
        no_of_chars++;
        if (no_of_chars % chunk == 0) {
            fclose(fd);
            printf("%s dosyasi kapatildi!\n", fdname);
            fd = NULL;
            sprintf(fdname, "dos%04d.xxx", no_of_files + 1);
        }
    }
    fclose(fs);
    if (no_of_chars % chunk != 0) {
        fclose(fd);
        printf("%s dosyasi kapatildi!\n", fdname);
    }

    printf("%ld uzunlugunda %s dosyasi %d uzunlugunda %d adet dosyaya
bolundu!\n",
        no_of_chars, fsname, chunk, no_of_files);

    return 0;
}

/*****bir.c *****/

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

#define MAX_LEN 80

int main(int argc, char **argv)
{
    FILE *fs, *fd;
    char fdname[MAX_LEN];
    char fsname[MAX_LEN] = "dos0001.xxx";
    int ch;
    int no_of_files = 0;
    long no_of_chars = 0L;
    int k;

    if (argc != 2) {
        printf("birlestirilecek dosyanin ismini giriniz : ");
        gets(fdname);
    }
    else {
        strcpy(fdname, argv[1]);
    }

    fd = fopen(fdname, "wb");
    if (fd == NULL) {
        printf("%s dosyasi yaratilamiyor!\n", fdname);
        exit(EXIT_FAILURE);
    }
}

```

```
printf("%s dosyasi yaratildi!\n", fdname);
while (fs = fopen(fsname, "rb")) {
    no_of_files++;
    printf("%s dosyasi acildi!\n", fsname);

    while ((ch = fgetc(fs)) != EOF) {
        fputc(ch, fd);
        no_of_chars++;
    }
    fclose(fs);
    printf("%s dosyasi kapatildi!\n", fsname);
    sprintf(fsname, "dos%04d.xxx", no_of_files + 1);
}
fclose(fd);
printf("%s dosyasi kapatildi!\n", fdname);
printf("%d adet dosya %ld uzunlugunda %s isimli dosya altinda\
birlestirildi!\n", no_of_files, no_of_chars, fdname);

for (k = 1; k <= no_of_files; ++k) {
    sprintf(fsname, "dos%04d.xxx", k);
    remove(fsname);
    printf("%s dosyasi silindi!\n", fsname);
}

return 0;
}
```

## 33 . BÖLÜM : enum TÜRÜ VE enum SABİTLERİ

Yazılan bir çok programda, yalnızca belirli anlamlı değerler alabilen değişkenler kullanma ihtiyacı duyulur. Örneğin bir "Boolean" değişkeni, yalnızca iki değere sahip olabilmektedir (doğru ve yanlış değerleri) C dilinde Boolean diye bir veri tipinin olmadığını hatırlayalım. Başka bir örnek olarak bir oyun kağıdının rengini tutacak bir değişkeni verebiliriz. Böyle bir değişken yalnızca 4 değişik değer alabilecektir : Sinek, Karo, Kupa, Maça. Uygulamalarda yapılacak olan, değişkene tamsayı değerleri vermek ve her tamsayıyı değişkenin alacağı değerle ilişkilendirmektir. örneğin:

```
int renk;          /* renk değişkeni iskambil kağıdının renk bilgisini tutacak */
```

```
renk = 1;          /* 1 tamsayı sabitinin karoğu temsil ettiğini varsayalım */
```

Böyle bir teknik uygulamalarda pekala kullanılabilir. Ancak bu tekniğin dezavantajı, algılanabilmesinin ve okunabilmesinin zor olmasıdır. Programı okuyacak kişi

```
int renk;
```

tanımlama ifadesini gördüğü zaman, renk değişkeninin yalnızca 4 farklı değer alabileceğini bilemediği gibi,

```
renk = 1;
```

şeklinde bir atama yapıldığını gördüğünde de renk değişkenine "karo" değerinin atanmış olduğunu anlayamaz.

Sembolik sabitlerin kullanılması ile okunabilirlik büyük ölçüde artırılabilir. Yukarıdaki örneğimiz için aşağıdaki sembolik sabitlerin tanımlanmış olduğunu düşünelim:

```
#define KARTRENK    int
#define SINEK       1
#define KARO        2
#define KUPA        3
#define MACA        4
```

```
KARTRENK renk;
```

```
renk = KARO;
```

Artık ifade daha okunabilir bir hale getirilmiştir. Bu yöntem ilk kullanılan teknikten şüphesiz daha iyidir, ama yine de en iyi çözüm olduğu söylenemez. Yukarıdaki sembolik sabit tanımlamalarının aynı türe ait olduğunu programı okuyan kişiye gösterecek bir ibare yoktur. Türün alabileceği değer sayısı daha fazla sayıda ise her biri için bir sembolik sabit tanımlamak zahmetli olacaktır. KARO, KUPA vs. olarak isimlendirdiğimiz sembolik sabitler programın derlenmesinden önce, önışlemci aşamasında tamsayı sabitlerle yer değiştireceğinden, hata ayıklama (debug) aşamasında artık bu sabitlere ulaşamayacaktır.

C dili, tasarımı yapılan tür için yalnızca belirli değerlerin alınabilmesini sağlayan bir türe sahiptir. Bu tür **enum** anahtar sözcüğüyle belirtilir. Türün alabileceği belirli değerleri gösteren isimlere ise **enum** sabitleri (enumeration constants) denir.

### enum Türünün ve enum Türüne İlişkin enum Sabitlerinin Bildirimi

```
enum [türün ismi] {esabit1, esabit2, .....};
```

**enum** bir anahtar sözcüktür. Derleyici küme parantezleri arasında isimlendirilmiş sembolik sabitlere 0 değerinden başlayarak artan sırada bir tamsayı karşılık getirir. Örnek:

**enum** RENK {Sinek, Karo, Kupa, Maca};

gibi bir bildirim yapıldığında Sinek **enum** sabiti 0, Karo **enum** sabiti 1, Kupa **enum** sabiti 2, Maca **enum** sabiti ise 3 değerini alır.

**enum** BOOL {FALSE, TRUE};

burada TRUE **enum** sabiti 1 FALSE **enum** sabiti ise 0 değerini alacaktır.

**enum** MONTHS {January, February, March, April, May, June, July, August, September, October, November, December};

Bu bildirimde January = 0, February = 1, ... December = 11 olacak şekilde **enum** sabitlerine 0 değerinden başlatarak ardışıl olarak değer verilir.

Eğer **enum** sabitlerine atama operatörü ile küme parantezleri içinde değerler verilirse, bu şekilde değer verilmiş **enum** sabitinden sonraki sabitlerin değerleri birer artarak otomatik olarak verilmiş olur:

**enum** MONTHS {January = 1, February, March, April, May, June, July, August, September, Oktober, November, December};

Artık aynı örnek için **enum** sabitlerinin değerleri January = 1, February = 2, ... December = 12 olmuştur.

**enum** sabitlerine **int** türü sayı sınırları içerisinde pozitif ya da negatif değerler verilebilir :

```
enum Sample {
    RT1 = -127,
    RT2,
    RT3,
    RT4 = 12,
    RT5,
    RT6,
    RT7 = 0,
    RT8,
    RT9 = 90
};
```

Yukarıdaki tanımlamada sabitlerin alacağı değerler aşağıdaki gibi olacaktır :

RT1 = -127, RT2 = -126, RT3 = -125, RT4 = 12, RT5 = 13, RT6 = 14, RT7 = 0, RT8 = 1, RT9 = 90

Görüldüğü gibi bir **enum** sabitine atama operatörü ile atama yapıldığı zaman onu izleyen **enum** sabitlerinin alacağı değerler, bir sonraki atamaya kadar, ardışıl olarak değerleri 1'er artarak biçimde oluşturulur.

Yukarıdaki ifadelerin hepsi bildirim ifadeleridir, tanımlama işlemi değildir. Bir başka deyişle derleyici bir nesne yaratmamakta ve dolayısıyla bellekte bir yer ayırmamaktadır.

**enum** bildirimi tür bildirmektedir. Bildirilen tür **enum** türünün yaratılmasında kullanılan enum tür ismidir. (enumeration tag). Tıpkı yapılarda ve birliklerde olduğu gibi bu türden bir nesne de tanımlanabilir:

```
enum Sample rst;
enum MONTHS this_month;
enum RENK kart1, kart2, kart3;
enum BOOL flag1, openflag, endflag, flag2;
```

Bildirimlerde **enum** tür ismi (enumeration tag) hiç belirtilmeyebilir. Bu durumda genellikle küme parantezinin kapanmasından sonra o **enum** türünden nesne(ler) tanımlanarak ifade noktalı virgül ile sonlandırılır.

```
enum {Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday} day1, day2, day3;
```

Yukarıdaki örnekte day1, day2, day3 bildirimi yapılmış **enum** türünden (belirli bir **enum** tür ismi yok) yaratılmış değişkenlerdir. Böyle bir tanımlama işleminin dezavantajı, artık bir daha aynı **enum** türünden başka bir değişkenin yaratılmamasıdır.

**enum** türünden bir nesne için derleyici, kullanılan sistemde **int** türünün uzunluğu ne kadar ise, o uzunlukta bir yer ayırır. Derleyici için **enum** türden bir nesne ile **int** türden bir nesne arasında herhangi bir fark bulunmamaktadır.

örneğin DOS altında:

```
sizeof(rst) == sizeod(rst) == sizeof(this_month) == sizeof(kart1) == sizeof(flag1) == 2 olacaktır.
```

**enum** ile bir tür tanımlandığına göre, bu türden nesne yaratılabileceği gibi fonksiyonların geri dönüş değerleri ve / veya parametre değişkenleri de **enum** ile yaratılmış türden olabilirler. Örneğin :

```
RENK kartbul (RENK kart1, RENK kartt2);
```

Kartbul fonksiyonu **enum** RENK türünden iki parametre almakta ve **enum** RENK türünden bir değere ger dönmektedir.

Aşağıdaki program parçasını inceleyelim:

```
...
```

```
enum RENK {Sinek = 0, Karo, Kupa, Maca};
```

```
RENK kartbul(RENK kart1, RENK kart2);
```

```
...
```

```
{
    RENK kartx, karty, kartz;
    ....
    kartx = kartbul(karty, kartz);
    if (kartx == Sinek){
        ...
    }
}
```

kartx, karty, kartz değişkenleri **enum** RENK türündendir. Prototipini gördüğümüz kartbul fonksiyonu da **enum** RENK türünden iki parametre almakta ve **enum** KART türünden bir değere geri dönmektedir. Dolayısıyla program parçası içinde kartx değişkenine **enum** RENK türünden bir değer atanmış olacaktır.

**enum** sabitleri ve #define önilemci komutuyla tanımlanmış sembolik sabitler nesne belirtmezler. Örneğin:

```
enum METAL {Demir, Bakir, Kalay, Cinko, Kursun};
```

```
...
```

```
Bakir = 3; /* hata oluşturur. Çünkü enum sabitleri sol taraf değeri değillerdir. */
```

#define komutu önışlemciye ilişkindir fakat **enum** sabitlerini ele alarak işleme sokmak derleme modülüne ilişkindir. Yani #define ve **enum** farklı aşamalarda ele alınırlar. **enum** sabitleri çok sayıda ve ardışıl tanımlamalar için tercih edilirler ve bir tür ismi olarak da okunabilirliği artırır. Örneğin:

```
BOOL isprime(int x);
```

gibi bir prototipi gören programcı isprime fonksiyonunun yalnızca doğru ya da yanlış değerlerinden birini ürettiğini, böylelikle fonksiyonun test amacıyla yazıldığını anlar.

**enum** sabitleri C dilinin faaliyet alanı kurallarına uyarlar. Eğer bir **enum** türü bir fonksiyon içinde bildirilmişse bu türe ilişkin **enum** sabitleri söz konusu fonksiyonun dışında tanınmazlar. Derleyicilerin standart başlık dosyalarında bir çok **enum** türü ve bu türlere ilişkin sembolik sabitler tanımlanmıştır. Örneğin aşağıdaki enum bildirimi 80x86 sistemlerinde çalışan bir Borland derleyicisinin GRAPHICS.H dosyasından alınmıştır.

```
enum COLORS {  
    BLACK,  
    BLUE,  
    GREEN,  
    CYAN,  
    RED,  
    MAGENTA,  
    BROWN,  
    LIGHTGRAY,  
    DARKGRAY,  
    LIGHTBLUE,  
    LIGHTGREEN,  
    LIGHTCYAN,  
    LIGHTRED,  
    LIGHTMAGENTA,  
    YELLOW,  
    WHITE  
};
```

**enum** tür isimleri de yapı ve birliklerde olduğu gibi, **typedef** anahtar sözcüğü yardımıyla tip isimlerine dönüştürülebilirler. Örneğin:

```
typedef enum {Sinek, Karo, Kupa, Maca} renk;
```

bildiriminden sonra artık **enum** anahtar sözcüğü kullanılmadan

```
renk kart1, kart2, kart3, kart4;
```

gibi bir tanımlama yapılabilir. artık renk bir tip ismi belirtmektedir. **enum** türü için çok kullanılan bir **typedef** örneği de:

```
typedef enum {FALSE, TRUE} BOOL;
```

(C++ dilinde **struct**, **union** ve **enum** türüne ilişkin isimler aynı zamanda türün genel ismi olarak kullanılabileceği için yukarıdaki gibi bir **typedef** bildirimine gerek kalmayacaktır. )

### **enum Türünün ve enum Sabitlerinin Kullanımına İlişkin Uyarılar**

**enum** türü ile bildirilen tür ismi kapsamında bildirimi yapılan sembolik sabitlerin tamsayı olarak derleme aşamasında değerlendirildiği söylenmişti. enum türünden tanımlanan bir değişkene enum şablonunda belirtilen sembolik sabitler yerine tamsayılar ile atama yapılabilir.

```
enum BOOL {TRUE, FALSE} flag1;
```

tanımlamasında

```
flag1 = FALSE;
```

yerine

```
flag1 = 1; şeklinde atama yapabiliriz?
```

Verilen örnekte `flag1` **enum** `BOOL` türünden bir değişkendir, ve derleyici **enum** türünden bir değişken ile **int** türünden bir değişken arasında fark gözetmez. Ancak derleyicilerin çoğu, programcının bunu bilinçsizce ya da yanlışlıkla yaptığını düşünerek bir uyarı mesajı verirler. Yukarıda verilen örnek için verilebilecek tipik bir uyarı mesajı : "assigning **int** to `BOOL`" şeklinde olacaktır.

**enum** türünden bir değişkene şablonda belirtilen sınırlar dışında kalan değerleri de atayabiliriz. Yukarıdaki örnek için:

```
flag1 = 10;
```

şeklinde atama yapabiliriz. (Bu durumda doğal olarak yine aynı uyarı mesajını alırız.)

Dahası **enum** türünden bir değişkene başka türünden bir değişken ya da sabit de atayabiliriz. Bu durumda tür dönüşümü kuralları gereği atamadan önce otomatik tür dönüşümü yapılarak sağ taraf ifadesi **int** türüne dönüştürülecektir.

```
double x = 5.6;
flag1 = x;
```

Bu durumda `x` değişkenine 5 değeri atancaktır.

Unutulmaması gereken şudur: `enum` türünün kullanılmasının nedeni öncelikle okunabilirliği artırmaktır. Yukarıdaki atamaları yapmamız teknik olarak mümkün olmakla birlikte, bu örnekler okunabilirliği kötü yönde etkilediklerinden ancak bilinçsiz kullanıma örnek olabilir. İdeal durum, **enum** türünden nesnelere ilgili **enum** türü için önceden belirlenmiş **enum** sabitlerini atamaktır.

Bir **enum** nesnesini operatörler işleme sokabiliriz. Örneğin:

Derleyici **enum** türünden nesneleri aynı **int** türü gibi ele alacağından **enum** türünden nesneler de **int** türünden nesneler gibi operatörler yardımıyla yeni ifadelerin oluşturulmasında kullanılabilirler. `enum` sabitlerinin kendisi sol taraf değeri gerektiren durumlarda kullanılamazlar.

```
enum {Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday} day1, day2, day3;
```

Yukarıdaki tanımlamadan sonra

```
++Sunday;
```

gibi bir ifade geçersiz olacaktır. Çünkü **enum** sabitleri bir soltaraf değeri değildir.

```
++day1;
```

ifadesi ise geçerlidir. `++day1` ifadesi `day1 + 1` anlamına geleceğinden, sonuçta `day1`'e (**enum** türünden değişken) **int** türünden bir sayı atanmış olacağından bazı derleyiciler bu durum için bir uyarı verebilirler.



## 34 . BÖLÜM : MAKROLAR

Önişlemci komutlarından, **#define** önişlemci komutunun sembolik sabit tanımlamalarında kullanılışını daha önce görmüştük. Makrolar **#define** önişlemci komutunun parametrelili kullanımı ile elde edilen yapılardır.

**#define** önişlemci anahtar sözcüğünü izleyen isim parametrelili olarak kullanılabilir. Bu durum sentaks olarak isime bitişik olarak yazılan açılan parantez ile belirtilir. Bu parantezin içinde tanımlanan makroya ilişkin parametre(ler) yer alır. Kapanan parantezi izleyen yazı ise makroya ilişkin değiştirme listesidir. Önişlemci makro parametrelerine ilişkin değiştirme listesini bir şablon olarak kabul eder ve kaynak kod içinde makronun çağırıldığını tespit ettiğinde makro çağırımında kullanılan argümanları, şablona uygun olarak açar.

Basit bir örnekle başlayalım:

```
#define Alan(x, y) ((x) * (y))
```

Yukarıdaki makro tanımlamasında x ve y makroya ilişkin parametrelerdir. Önişlemci yukarıdaki satırı gördükten sonra kaynak kod içinde bu makronun çağırıldığını tespit ederse, makroyu yukarıdaki şablona göre açacaktır. Örneğin kaynak kod içerisinde

```
a = Alan(b, c);
```

şeklinde bir deyim yer aldığını düşünelim. Bu durumda önişlemci Alan makrosunun çağırıldığını anlar. Ve ifadeyi aşağıdaki şekilde açar :

```
a = ((b) * (c));
```

Makro açılımının sadece metinsel bir yer değiştirme olduğunu bilmeliyiz. Yani yukarıdaki deyim örneğin :

```
a = Alan(7 + 3, 8 + 1);
```

şeklinde olsaydı, önişlemci bunu

```
a = ((7 + 3) * (8 + 1));
```

şeklinde açardı. Yer değiştirme işlemcinin derlemenin ön modülü olan önişlemci tarafından derleme öncesi yapıldığını ve derleyicinin object kod oluşturmak için ele aldığı kaynak kodun önişlemci modülünün çıktısı olduğunu hatırlatalım. Yani derleyicinin ele aldığı kodda artık makro çağırımları değil makroların önişlemci tarafından açılmış biçimleri bulunacaktır.

Başka bir makroyu inceleyelim :

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

```
#define Kare_fark(x, y) (((x) - (y)) * ((x) + (y)))
```

```
int main()
{
    int a = 10;
    int b = 5;
    int c;

    c = Kare_fark(a, b);
    printf("c = %d\n", c);
    return 0;
}
```

Yukarıdaki kaynak kodu alan önışlemci işini bitirdikten sonra derleyicinin ele alacağı kaynak kod aşağıdaki şekilde olacaktır :

stdio.h dosyasının içeriği

```
int main()
{
    int a = 10;
    int b = 5;
    int c;

    c = (((a) - (b)) * ((a) + (b)));
    printf("c = %d\n", c);
    return 0;
}
```

Bir yılın artıkyıl olup olmadığını test etmek için kullanılacak başka bir makro:

```
#define is_leap(y) ((y) % 4 == 0 && (y) % 100 != 0 || (y) % 400 == 0)
```

## Makrolar Ne Amaçla Tanımlanır

Makrolar fonksiyonlara bir alternatif olarak kullanılmaktadır. Yukarıdaki örnekleri tekrar inceleyelim. Alan makrosu yerine:

```
int alan(int en, int boy)
{
    return en * boy;
}
```

Kare\_fark makrosu yerine:

```
int Kare_fark(int a, int b);
{
    return (a - b) * (a + b);
}
```

is\_leap makrosu yerine:

```
int is_leap(int year)
{
    return year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
}
```

fonksiyonları tanımlanabilirdi.

Başka bir örnek olarak da toupper fonksiyonunun makro biçiminde tanımlanmış olması verilebilir. (toupper derleyicilerin çoğunda fonksiyon olarak değil makro biçiminde tanımlanmıştır.)

```
#define TOUPPER(c) ((c) >= 'a' && (c) <= 'z' ? (c) - 'a' + 'A' : (c))
```

Aşağıda fonksiyon tanımlamalarına alternatif olacak bazı basit makrolar tanımlanmaktadır:

```
#define ISUPPER(c) ((c) >= 'A' && (c) <= 'Z')
#define ISLOWER(c) ((c) >= 'a' && (c) <= 'z')
#define ISDIGIT(c) ((c) >= '0' && (c) <= '9')
#define ISEVEN(x) ((x) % 2 == 0)
```

Bir makronun parametresi olmayabilir. Örneğin:

```
#define getchar()    getc(stdin)
```

Yukarıdaki makro yerine sembolik sabit de kullanılabilirdi. Ancak fonksiyon yapısına benzetmek amacıyla parametresi olmayan bir makro olarak tanımlanmıştır.

## Makrolarla Fonksiyonlar Arasındaki Farklılıklar Nelerdir

Makrolar fonksiyonlara alternatif olarak kullanılmalarına karşılık, makrolar ile fonksiyonlar arasında çok önemli farklar bulunmaktadır:

1. Makrolar kaynak kodu dolayısıyla da çalışabilir (exe) kodu büyütürler. Makrolar ön işlemci aşamasında değerlendirilir. Örneğin `is_leap` makrosunun **#define** ön işlemci komutuyla tanımlandıktan sonra kaynak kod içinde yüz kere çağırıldığını düşünelim. Ön işlemci kaynak kodu ele aldığı anda yüz tane çağırılma ifadesinin her biri için makroyu açacaktır. Derleyici modülü kaynak kodu ele aldığı anda artık makroları değil makroların açılmış şeklini görecektir. Makro açılımları kaynak kodu büyütecektir. Kaynak kodun büyümesinden dolayı çalışabilen dosyanın boyutu da büyüyecektir. Oysa makro yerine bir fonksiyon tanımlansaydı, Fonksiyon çağırılması durumunda yalnızca çağırılma bilgisi kayanaka koda yazılmaktadır.
2. Makroların en büyük avantajı, fonksiyon çağırma işlemindeki görelili yavaşlıktan kaynaklanır. Fonksiyon çağırması sembolik makine dili (assembler) düzeyinde ele alındığında, bazı ilave makine komutları da icra edilmekte ve bu makine komutlarının icrası ve yapılan bazı ilave işlemler icra süresini uzatacaktır.

Fonksiyon çağırılırken `CALL` ve geri dönüşte `RET` makine komutları icra edilmektedir. Fonksiyon çağırımı için ayrıca parametre aktarımı ve stack düzenlemesi işlemleri yapılmaktadır. Makroların açılması metinsel bir yer değiştirme olduğu için, yukarıda anlatılan ilave işlemler yapılmaz. Böylece programın hızı görece olarak artacaktır.

3. Makrolar türden bağımsızdır. (generic). Makrolar metinsel bir yer değiştirme olduğundan fonksiyonlar gibi belirli bir türe bağlı değildirler. Örneğin iki sayıdan büyüğüne geri dönen bir fonksiyon yazmak istersek bu fonksiyonun parametre değişkenleri belirli bir türden olmak zorundadır:

```
int max_of_two(int number1, int number2)
{
    return (number1 > number2 ? number1 : number2);
}
```

Yukarıdaki fonksiyon **int** türden değerler için yazılmıştır. Başka türden değerler fonksiyona argüman olarak gönderildiğinde tür dönüşümü kuralları işleyecek ve argümanların türü parametre değişkenlerinin türlerine dönüştürülecektir. Oysa yukarıdaki fonksiyon yerine bir makro tanımlanmış olması durumunda bu makro türden bağımsız olacaktır :

```
#define max_of_two(a, b)    ((a) > (b) ? (a) : (b))
```

Yukarıdaki makro aşağıdaki şekillerde çağırılabilir:

```
int x, y, z;
char c1, c2, c3;
double d1, d2, d3;
...
x = max_of_two(y, z);
c1 = max_of_two(c2, c3);
d1 = max_of_two(d2, d3);
```

4. Fonksiyonlar kendilerine gönderilen argümanları bir kez ele alırlar ancak makrolar argümanlarını birden fazla ele alabilirler:

Aşağıdaki makroları göz önüne alalım:

```
#define KARE(a)      ((a) * (a))
```

```
int x = 10, y;  
y = KARE(x++)
```

ifadesi makro tanımlamasına uygun olarak açıldığında

```
y = ((x++) * (x++))
```

ifadesi oluşacaktır ki bu ifade de, operatörler konusunda gördüğümüz gibi "undefined behavior" özelliği gösterecektir. Oysa KARE bir fonksiyon olsaydı bu fonksiyonun

```
KARE(x++);
```

şeklinde çağırılması bir probleme neden olmayacaktı. `x++` ifadesi `x` değişkeninin (artmayan) değerini üreteceğinden, KARE fonksiyonu 10 değeri ile çağırılmış olacaktı.

5. Makrolarda makroların argümanları ile parametreleri arasında tür dönüştürülmesi söz konusu değildir.

Bir fonksiyon çağırılma ifadesini gördüğünde derleyici fonksiyona gönderilen argüman ile fonksiyonun ilgili parametre değişkeni arasındaki tür uyumunu kontrol eder. Eğer bu türler arasında bir farklılık varsa, mümkünse, tür dönüştürme kuralları gereği argüman olan ifadenin türünü parametre değişkeninin türüne çevirir. Makro argümanlarının önışlemci tarafından makronun parametrelerine göre tür uyumunun kontrol edilmesi ya da tür dönüştürme işlemine tabi tutulması söz konusu değildir.

C++ dilinde makroların kullanımı C diline göre çok daha azalmıştır. Zira C++ dilinde bir çok durumda makrolar yerine inline fonksiyonlar kullanılmaktadır.

6. Bir gösterici bir makroyu gösteremez.

İleride göreceğimiz gibi C dilinde bir fonksiyonu gösteren gösterici tanımlanabilir. Ve bu göstericilerden C programcılığında çeşitli biçimlerde faydalanmak mümkündür. Ancak makrolar önışlemci aşamasında ele alındığından bir göstericini bir makroyu göstermesi söz konusu değildir.

## # ve ## önışlemci operatörleri

Makro tanımlamaları iki özel operatör içerebilirler. Bu operatörler derleyiciyi değil önışlemciyi ilgilendirmektedir. Yani önışlemcinin çıktısı derleyici modülüne verildiğinde artık kaynak kod bu operatör atomlarından arındırılmış olacaktır.

# atomu makronun yer değiştirme listesinde (replacement list) yer alabilecek bir operatördür. Operandı makro parametresidir. Makronun açılımı sırasında #operandına alınan argüman string haline getirilir. Yani çift tırnak içine alınır. Bu yüzden bu atoma stringe dönüştürme operatörü de denir. (stringizing operator)

#atomunun kullanılabileceği bir kaç tema aşağıda verilmektedir. Aşağıdaki makroyu inceleyelim :

```
#define PRINT_INT(x)      printf("#x " = %d\n", x)
```

örneğin kaynak kod içerisinde

```
int result = 5;  
PRINT_INT(result);
```

şeklinde makronun çağırıldığını düşünelim. Makro açıldığında derleyiciye giden kaynak kod aşağıdaki şekilde olacaktır :

```
int result = 5;
printf("result" " = %d\n", result);
```

Aralarında boşluk karakterleri dışında başka bir karakter bulunmayan stringlerin derleyici tarafından tek bir string olarak birleştirildiklerini anımsarsak, yukarıdaki ifadenin derleyici açısından aşağıdaki şekilde yorumlanacağını görebiliriz :

```
printf("result = %d\n", result);
```

Böylece yukardaki deyimın icra edilmesiyle ekrana aşağıdaki şekilde bir yazı yazdırılabilecektir :

```
result = 5;
```

`##` önişlemci operatörü iki operand alan aralık konumunda bir operatördür. Makronun açılması sırasında Operandı olan atomları birleştirerek tek bir atom haline getirir. Bu yüzden atom birleştirme operatörü diye de isimlendirilir. (Token pasting operator), Atom birleştirme operatörünün operandlarından biri eğer bir makro parametresi ise, birleştirme işlemi parametre ilgili argümanla yer değiştirdikten sonra yapılacaktır. Atom birleştirme operatörü `##`, string yapma atomu kadar sık kullanılan bir operatör değildir. Sadece çok özel durumlarda kullanılmaktadır.

### Makro tanımlamalarında nelere dikkat edilmelidir?

Makro tanımlamalarında en sık yapılan hatalar makro açılımı neticesinde, operatör öncelikleri yüzünden açılmış kodun beklenilmeyen şekilde değer üretmesidir. Örneklerle açıklayalım :

```
#define ALAN(a)      a * a
```

Yukarıdaki makro

```
x = ALAN(y);
```

şeklinde çağırılırsa önişlemci kodu

```
x = y * y;
```

şeklinde açacağından bir problem oluşmayacaktır. Ancak makronun

```
int y = 1;
x = ALAN(y + 5)
```

şeklinde çağırıldığını düşünelim. Bu durumda derleyici kodu

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

şeklinde açacaktır. Ve bu durumda `y` değişkenine 36 yerine 11 değeri atanacaktır.

Bu tür problemlerin önlenmesi için makro tanımlamalarında makro açılım ifadesinde yer alan makro parametreleri parantez içine alınmalıdır.

Yukarıdaki makro

```
#define ALAN(a)      (a) * (a)
```

şeklinde tanımlansaydı makro çağırımı önişlemci tarafından açıldığında

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

ifadesi oluşacaktır ki, deyimin icrası sonucunda x değişkenine 36 değeri atanacaktır.

Peki ya makro aşağıdaki şekilde çağırılsa

```
x = 72 / ALAN(y + 5);
```

x değişkenine 2 değerinin atanması gerekirken, makro aşağıdaki gibi açılacağından

```
x = 72 / (y + 5) * (y + 5) ;
```

x değişkenine 72 değeri atanacaktır.

Makro tanımlamalarında makro açılım ifadeleri en dışarıdan parantez içerisine alınmalıdır.

Söz konusu makro

```
#define ALAN(a)      ((a) * (a))
```

şeklinde tanımlanmış olsaydı

```
x = 72 / ((y + 5) * (y + 5) );
```

makro önışlemci tarafından yukarıdaki gibi açılacaktı ve bu kodun icrası ile x değişkenine gerçekten 2 değeri atanacaktı.

Makrolarla ilgili olarak sık yapılan başka bir hata da makro tanımlamasında makro ismi ile makro parametre parantezi arasında boşluk bırakmaktır. ALAN makrosunun yanlışlıkla aşağıdaki gibi tanımlandığını düşünelim.

```
#define ALAN (a)      ((a) * (a))
```

Önışlemci artık yukarıdaki tanımlamayı bir makro değil bir sembolik sabit olarak ele alacaktır. Önışlemci kaynak kodun geri kalan kısmında gördüğü ALAN yazılarını (a) ((a) \* (a)) ile yer değiştirecektir. Örneğin :

```
ALAN(x + 3);
```

gibi bir ifadenin var olduğunu düşünelim. Önışlemci bu kodu aşağıdaki şekilde açacaktır :

```
(a)      ((a) * (a))(x + 3);
```

aynı isimli makro ve fonksiyonlar

Bir makro ile aynı isimli bir fonksiyon kaynak kod içinde yer alabilir. Bu durumda bu isimle bir çağırım yapıldığında ne olacaktır. makro mu fonksiyon mu çağırılmış olacaktır?

```
#define max(x, y)      ((x) > (y) ? (x) : (y))
```

gibi bir makronun tanımlandığını ve kaynak kod içerisinde aşağıdaki fonksiyon tanımının da yer aldığını düşünelim.

```
int max(int number1, int number2)
{
    return x > y ? x : y;
}
```

Kaynak kod içerisinde örneğin

`c = max(a, b);`

şeklinde bir deyim yer aldığı düşünelim. Bu durumda ne olacaktır?

Tabi ki önışlemci makroyu açacaktır. Zira makroların açılması, yani metinsel yer değıştirme işleminin yapılması, daha derleyici kaynak kodu ele almadan önışlemci tarafından yapılacağından, derleyici modülüne sıra geldiğinde derleyici artık yukarıdaki çağırım ifadesi yerinde

`c = a > b ? a : b;`

ifadesini görecektir.

Aynı isimli makro ve fonksiyon tanımlamalarının var olması durumunda, makro açılımın yapılması yerine fonksiyonunu çağırılması isteniyorsa ne yapılabilir?

Bu durumda fonksiyon çağırma ifadesinde fonksiyon ismi parantez içine alınırsa, önışlemci artık bir yer değıştirme yapmaz ve derleyici fonksiyon çağırma kodu üretir.

### uzun makroların yazılması

Daha işlevsel makroların yazılmasında virgöl operatörü kullanılabilir. Aşağıdaki makroyu düşünelim:

```
#define ECHO(s)    (gets(s), puts(s))
```

Fonksiyon çağırımları da birer ifade olduğundan, fonksiyon çağırımı ifadelerinin virgöl operatörüyle birleştirilmesi tamamen legaldir. Yukarıdaki makroyu bir fonksiyonmuş gibi çağırabiliriz:

```
char *ptr = "Ahmet";
ECHO(str);
```

Önışlemci makroyu aşağıdaki şekilde açacaktır :

```
char *ptr = "Ahmet";
(gets(str), puts(str));
```

Ancak bir makronun bir seri deyim içerdığını düşünelim. İfade yerine belirli sayıda deyim olması, bazı durumlarda virgöl operatörünün kullanılmasına izin vermeyebilir. Çünkü virgöl operatörünün operandları deyim değil ifadelerdir.

Uzun ve işlemsel makroların yazılmasında virgöl operatörünün kullanılması yerine makro değıştirme listesini en dıştan küme parantezleri içine de alabilirdik :

```
#define ECHO(s)    { gets(s); puts(s); }
```

Makro yukarıdaki gibi tanımlandığında bazı problemler oluşabilir. Örneğin makronun aşağıdaki gibi bir if deyiminde çağırıldığını düşünelim :

```
if (echo_flag)
    ECHO(str);
else
    gets(str);
```

Önışlemci makroyu aşağıdaki gibi açacaktır :

```
if (echo_flag)
    { gets(str); puts(str); };
else
    gets(str);
```

Kod derleyici modülüne geldiğinde aşağıdaki gibi yorumlanacağından

```
if (echo_flag)
    { gets(str); puts(str); }
;
else
    gets(str);
```

if kısmı olmayan bir else olması gerekçesiyle derleme zamanında bir error oluşacaktır. Oluşan problemi nasıl çözebiliriz? Makronun çağırılması durumunda sonlandırıcı ";" at0munu koymayarak? Bu çirkin bir tarz olurdu...

Bu durumda iki yöntem kullanabiliriz :

1. Makro açılımında do while döngü yapısını kullanmak:

```
#define ECHO(s)      \
    do {             \
        gets(s);      \
        puts(s);      \
    } while (0)
```

(#define önışlemci komutunun kullanılmasında satır sonunda yer alan "\" karakterinin komutun bir sonraki satırda devam edeceğini gösterdiğini hatırlayalım)

artık yukarıdaki makro

```
ECHO(str);
```

şeklinde çağırıldığında, makro önışlemci tarafından açıldıktan sonra, çağırma ifadesinin sonundaki sonlandırıcı (;) do while deyiminin sentaksında bulunan while parantezini izleyen sonlandırıcı olacaktır.

2. Makro açılımında if deyimi kullanmak:

```
#define ECHO(s)      \
    if (1) {          \
        gets(s);      \
        puts(s);      \
    }                 \
    else
```

yukarıdaki makro

```
ECHO(str);
```

şeklinde çağırılıp önışlemci tarafından açıldığında makro çağırma ifadesinin sonundaki sonlandırıcı (;) if deyiminin yanlış kısmına ilişkin (else kısmına ilişkin) boş deyim olacaktır.



## 35 . BÖLÜM : DİĞER ÖNİŞLEMCİ KOMUTLARI

Daha önceki derslerimizde önışlemci kavramını incelemiş ve önışlemciye ilişkin iki komutu incelemiştik. Bu komutlar **#include** ve **#define** önışlemci komutlarıydı. Önışlemciye ilişkin diğer komutları da bu desimizde ele alacağız.

### Koşullu derlemeye ilişkin önışlemci komutları

Koşullu derlemeye ilişkin komutlar **#if** **#ifdef** ve **#ifndef** olmak üzere üç tanedir. Koşullu derleme komutuyla karşılaşan önışlemci, koşulun sağlanması durumunda belirlenmiş program bloğunu derleme işlemine dahil eder, koşul sağlanmıyorsa ilgili program bloğunu çıkartarak derleme işlemine dahil etmez.

#### **#if Önışlemci Komutu**

Genel biçimi şöyledir:

```
#if <sabit ifadesi>
...
...
#else
...
..
#endif
```

Sabit ifadesinin sayısal değeri sıfır dışı bir değer ise (sabit ifadesi doğru olarak değerlendirilerek) **#else** anahtar sözcüğüne kadar olan kısım, eğer sabit ifadesinin sayısal değeri 0 ise (sabit ifadesi yanlış olarak değerlendirilerek) **#else** ile **#endif** arasındaki kısım derleme işlemine dahil edilir. Örnek :

```
#include <stdio.h>

int main()
{
    #if 1
        printf("bu bölüm derleme işlemine sokulacak\n");
    #else
        printf("bu bölüm derleme işlemine sokulmayacak\n");
    #endif
}
```

Önışlemci, programı **#** içeren satırlardan arındırarak aşağıdaki biçimde derleme modülüne verecektir:

```
[stdio.h dosyasının içeriği ]
void main()
{
    printf("bu bölüm derleme işlemine sokulacak\n");
}
```

**#if** komutunun sağındaki sabit ifadesi operatör içerebilir ancak değişken içeremez. Örneğin:

```
#define MAX    100
...
#if MAX > 50
...
#else
...
#endif
```

Tıpkı C dilinde olduğu gibi önışlemci komutlarında da **#else** - **#if** merdivenlerine sık raslanır.

```
#define MAX    100
...
#if MAX > 50
...
#else
    #if MAX < 1
    ...
    #endif
#endif
```

yerine

**#elif** önişlemci komutu da kullanılabilir. Ancak bu durumda **#elif** önişlemci komutuna ilişkin **#endif** ile dışarıdaki **#if** komutuna ilişkin **#endif** ortak olacaktır.

```
#define MAX 100
...
#if max > 50
...
#elif MAX > 30
...
#else
...
#endif
```

Koşullu derleme işlemi kaynak kodun belirli bölümlerini belli durumlarda derleme işlemine sokmak için kullanılır. Şüphesiz bu durum programcı tarafından da kodun eklenip silinmesiyle sağlanabilirdi. Ancak **#if** komutu buna daha kolay olanak sağlamaktadır.

### **#ifdef ve #ifndef Önişlemci Komutları**

Belli bir sembolik sabitin tanımlandığı durumda **#else** anahtar sözcüğüne kadar olan kısım, tanımlanmadığı durumda **#else** ve **#endif** arasındaki kısım derleme işlemine dahil edilir. Genel biçimi aşağıdaki gibidir :

```
#ifdef <sembolik sabit>
...
#else
...
#endif
```

Örneğin:

```
#include <stdio.h>

#define SAMPLE

void main()
{
    #ifdef SAMPLE
        printf("SAMPLE tanımlanmış");
    #else
        printf("SAMPLE tanımlanmamış");
    #endif
}
```

Burada **SAMPLE** **#define** ile tanımlandığı için **#ifdef** doğru olarak ele alınır ve derleme işlemine ilk **printf()** dahil edilir. **#ifdef** ile sorgulama yapılırken sembolik sabitin ne olarak tanımlandığının bir önemi yoktur.

**#ifdef** önışlemci komutu aynı başlık dosyasını kullanan ve birkaç modülden oluşan programlarda yaygın olarak kullanılır. Örneğın dosya1.c, dosya2.c ve dosya3.c modüllerinden oluşan bir projede project.h isimli bir başlık dosyası ortak olarak kullanılıyor olsun. Bu durumda global değışkenlerin yalnızca bir modülden global değıerlerinde **extern** olarak kullanılması gerekecektir. Bu aşığıdaki gibi sağlanabilir.

```
/* project.h*/

#ifdef GLBVAR
#define EXTRN
#else
#define EXTRN extern
#endif

/* Global değışkenler */

EXTRN int x;
EXTRN int y;
...
```

Şimdi modüllerin birinde, bu başlık dosyasının include edilmesinden önce, GLBVAR adlı bir sembolik sabit tanımlanırsa, önışlemci EXTRN sözcüğünü sileceğinden (onun yerine boşluk karakteri kopyalayacağından) o modül için global tanımlama yapılmış olur. Değilse EXTRN yerine **extern** anahtar sözcüğü önışlemci tarafından yerleştirilecektir. **#ifdef** önceden tanımlanmış sembolik sabitlerle taşınabilirliği sağlamak amacıyla da sıklıkla kullanılmaktadır.

**#ifndef** komutu **#ifdef** komutunun tam tersidir. Yani sembolik sabit tanımlanmamışsa **#else** kısmına kadar olan bölüm, tanımlanmışsa **#else** ile **#endif** arasındaki kısım derleme işlemine dahil edilir. Örneğın :

```
#ifndef BLINK
#define BLINK 0x80
#endif
```

Burada BLINK sembolik sabiti ancak daha önce tanımlanmamışsa tanımlanması yapılmıştır. Daha önce tanımlanmışsa tanımlanma işlemi yapılmamıştır.

### **defined() Önışlemci Operatörü**

**defined** bir sembolik sabitin tanımlanıp tanımlanmadığını anlamak için kullanılan bir önışlemci operatördür. Parantez içerisindeki sembolik sabit tanımlanmışsa 1 değeri tanımlanmamışsa 0 değeri üretir. Örneğın

```
#if defined(MAX)
...
#endif
```

Burada MAX tanımlanmışsa **defined** önışlemci operatörü 1 değeri üretecektir. Bu değeri de **#if** tarafından doğru olarak kabul edilecektir. Burada yapılmak istenen **#ifdef** ile aşığıdaki gibi elde edilebilirdi:

```
#ifdef MAX
...
#endif
```

## Koşullu Derlemeye İlişkin Önışlemci Komutları Nerelerde Kullanılır

### 1. Debug Amacıyla

Bir programı yazarken, programın doğru çalışıp çalışmadığını test etmek amacıyla birtakım ilave kodlar yazabiliriz. Bu tür test kodlarının biz programı test ettiğimiz sürece çalışmasını ancak programınızı yazıp bitirdikten sonra, yapacağımız son derlemeye, bu test kodlarının derleme işlemine sokulmamasını isteriz. Test amacı ile yazılan kod parçacıkları derleme işlemine DEBUG adlı bir sembolik sabitin değerine bağlı olarak aşağıdaki şekilde derleme işlemine katılır ya da katılmaz.

```
#if  DEBUG
...
#endif
```

Eğer DEBUG önışlemci sabitinin değeri 0 ise #if ile #endif arasındaki kod parçalı derlemeye işlemine katılmaz.

### 2. Birden Fazla Makinede ya da İşletim Sisteminde Çalışacak Programların Yazılmasında

```
#if defined(WINDOWS)
...
#elif defined(DOS)
...
#elif defined(OS2)
...
#endif
```

### 3. Bir Programın Farklı Derleyicilerde Derlenmesi Durumunda

```
#ifdef  __STDC__
...
#else
...
#endif
```

Yukarıda önışlemci komutları ile derleyicinin standart C derleyicisi olup olmadığına bakılarak belirli kaynak kod parçaları derlemeye dahil ediliyor.

### 4. Bir Başlık Dosyasının Bir Kereden Daha Fazla Kaynak Koda Dahil Edilmesini Önlemek Amacıyla. (multiple inclusion)

Aynı başlık dosyası kaynak koda **#include** önışlemci komutuyla iki kez eklenmemelidir. Bir başlık dosyası bir kaynak koda birden fazla eklenirse ne olur? Başlık dosyaları içinde ne olduğunu hatırlayalım :

1. Fonksiyon prototip bildirimleri  
Hatırlayacağımız gibi fonksiyon prototip bildirimlerinin özdeş olarak birden fazla yapılması bir error oluşumuna neden olmaz.
2. Sembolik sabit ve makro tanımlamaları  
Özdeş sembolik sabit ve makro tanımlamaları kaynak kodun önışlemci tarafından ele alınması sırasında bir hataya neden olmaz.
3. typedef bildirimleri (tür tanımlamaları)  
typedef anahtar sözcüğü ile yapılan tür tanımlamaları özdeş olmaları durumunda yine derleme zamanında bir hata oluşumuna neden olmazlar.
4. Yapı, birlik, bit alanı, enum türü bildirimleri  
Bir özdeş olarak bile birden fazla yapılırsa derleme zamanında hata oluşumuna neden olur. Bu yüzden bir başlık dosyası kaynak koda birden fazla dahil edilirse ve o başlık

dosyası içinde bir yapının, birliğin, bit alanının ya da enum trünün bildirimi yapılmış ise derleme zamanında error oluşacaktır. Dolayısıyla başlık dosyasının ikinci kez kaynak koda eklenmesi engellenmelidir.

Bir başlık dosyasının kaynak koda birden fazla eklenmesi aşağıdaki gibi engellenebilir:

```
/* GENERAL. H   BİR BAŞLIK DOSYASI ÖRNEĞİ */
```

```
#ifndef  GENERAL_H
...
#define  GENERAL_h
...
#endif
```

Yukarıdaki kod parçasında başlık dosyası **#ifndef** ve **#endif** önışlemci komutlarının arasına alınmıştır. Eğer GENERAL\_H sembolik sabiti tanımlanmışsa, bu general.h başlık dosyasının kaynak koda daha önce dahil edildiği anlamına gelecek ve **#ifndef** ile **#endif** arasındaki kaynak kod parçası kaynak koda eklenmeyecektir. Eğer GENERAL\_H sembolik sabiti tanımlanmış ise, **#ifndef** ile **#endif** arasındaki kaynak kod parçası kaynak koda eklenecektir. Eklenen kod parçası içinde **#define** GENERAL\_h önışlemci komutu da bulunmaktadır.

Bu standart yöntemle, yani başlık dosyalarının bir sembolik sabitin tanımlanmasına bağlı olarak kaynak koda eklenip eklenmemesi sağlanır böylece bir kaynak kodun birden fazla kaynak koda eklenmesi engellenmiş olur.

Koşullu derlemeye ilişkin önışlemci komutlarının kullanılmasına ilişkin bir program parçası örneği aşağıda verilmiştir :

```
#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 bulunmaktadır. DLEVEL sembolik sabitinin 5'den büyük olup olmasına göre değerine göre önışlemci tarafından doğru kısım ya da yanlış kısım ele alacaktır.

#**elif** önışlemci komutunun da yer aldığı ikinci #**if** bloğunda ise DLEVEL sembolik sabitinin değerine göre 4 seçenektan biri ele alınacaktır. DLEVEL sembolik sabitinin değerine bağılı olarak STACK sembolik sabiti 0, 100 ya da 200 olarak tanımlanmaktadır. DLEVEL sembolik sabitinini değerinin 5'den büyük olması durumunda ise

```
display(debugptr);
```

deyimi derleyiciye verilmekte ve bu durumda STACK sembolik sabiti tanımlanmamaktadır.

## Genel Önışlemci Komutları

### #**undef** önışlemci komutu

Bir sembolik sabitin ilki ile özdeş olmayan bir biçimde ikinci kez tanımlanması C önışlemcisi tarafından "uyarı " olarak değerlendirilir. Bu durumda ilk tanımlananın mı ikinci tanımlananın mı geçerli olacağı taşınabilir bir özellik değildir. (Derleyiciler genellikle ikinci tanımlanma noktasına kadar ilkinin, ikinci tanımlanma noktasından sonra ise ikincinin geçerliliğini kabul ederler.)

Örneğin aşağıdaki gibi bir tanımlama işlemi uyarı gerektirir.

```
#define MAX100
...
#define MAX200
```

Bir sembolik sabitin ilki ile özdeş olarak tanımlanmasında herhangi bir problem çıkmayacaktır. Bir sembolik sabit ikinci kez tanımlanmak isteniyorsa önce eski tanımlamayı ortadan kaldırmak gerekmektedir. Bu işlem #**undef** önışlemci komutu ile yapılır. #**undef** önışlemci komutunun yanına geçerliliği ortadan kaldırılacak sembolik sabitin ya da makronun ismi yazılmalıdır. Örneğin :

```
#undef MAX
#define MAX200
```

Önce MAX sembolik sabitinin tanımlanması iptal edilmiş, sonra 200 olarak yeniden tanımlanmıştır. #**undef** ile tanımlanması kaldırılmak istenen sembolik sabit, daha önce tanımlanmış olmasa bile bu durum bir probleme yol açmaz. Örneğin yukarıdaki örnekte MAX tanımlanmamış olsaydı bile bir uyarı ya da hataya yol açmazdı.

### #**error** önışlemci komutu

Önışlemci #**error** komutunu görünce bu komutu izleyen mesajı basarak derleme işlemibne son verir. Örneğin:

```
#ifdef __TINY__
#error Bu program tiny modelde derlenmez!..
#endif
```

```
#error direktifi
```

**error** önışlemci komutunun yanında (boşluk karakteri ile ayrılmış) bir hata mesajı yer alır. Derleyici **error** direktifini görünce bu hata mesajını yazarak deleme işlemine son verecektir.

```
#ifdef __TINY__
#error bu program tiny modelde derlenemez
#endif
```

## ÖNCEDEN TANIMLANMIŞ SEMBOLİK SABİTLER

Önişlemci bu sabit yerine kaynak koddaki o anda bulunan satır numaralarını yerleştirir. Bu özel durumlar test edilerek çeşitli seçenekler değerlendirilmektedir. Bu kısımda önceden tanımlanmış sembolik sabitlerin standart olanlarından bir kısmını inceleyeceğiz.

\_\_LINE\_\_

Önişlemci bu sabit yerine kaynak koddaki o anda bulunulan satır numarasını yerleştirir. #include dosyalarının içi bu işleme dahil edilmez. Aşağıdaki kodu inceleyiniz:

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

```
int main()
{
    printf("satır no . %d\n", __LINE__)
    return 0;
}
```

\_\_FILE\_\_

Önişlemci bu sabit yerine iki tırnak içerisinde kaynak dosyanın ismini yazar. Aşağıdaki örnekte kaynak dosyanın ismi ekrana yazdırılıyor. string ifadelerinin karakteri gösteren birer adres olduğunu anımsayınız.

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

```
int main()
{
    printf("dosya ismi : %s\n", __FILE__);
    return 0;
}
```

\_\_DATE\_\_ ve \_\_TIME\_\_ : Önişlemci bu sabitlerin yerine derlemenin yapıldığı tarih ve zamanı yazar. \_\_DATE\_\_ "aaa gg yyyy", \_\_TIME\_\_ ise ss:dd:ss biçiminde yazılmaktadır.

Taşınabilirliğe ilişkin sembolik sabitler

\_\_STDC\_\_

C'de kullandığımız kimi anahtar sözcükler tam anlamıyla standart değildir. Bu anahtar sözcükler sistemler arası farklılıklara karşılık verebilmek için kullanılmaktadır. Örneğin 8086 sistemlerinde kullandığımız far ve near standart olarak her sistemde bulunmayan iki anahtar sözcüktür. Derleyiciniz eğer yalnızca C'nin anahtar sözcüklerini destekliyorsa

\_\_STDC\_\_

sembolik sabiti tanımlanmış varsayılır.

İçerisinde standart olmayan anahtar sözcükler geçen programlar bu sembolik sabit kullanılarak daha taşınabilir bir hale getirilebilir :

```
#ifdef      __STDC__
#define      far
#endif
```

Yukarıdaki örnekte eğer yalnızca standart C'nin anahtar sözcüklerini kullanan bir derleyici ile çalışıyorsanız \_\_STDC\_\_ tanımlanmış olacağından far anahtar sözcüğü silinecektir.

\_\_TINY\_\_  
\_\_SMALL\_\_  
\_\_MEDIUM\_\_

\_\_COMPACT\_\_  
\_\_LARGE\_\_  
\_\_HUGE\_\_

Borland derleyicilerinde aktif olarak kullanılan bellek modeline göre bu sembolik sabitlerden bir tanesi tanımlanmış sayılır. Örneğin o anda LARGE modelde çalışılıyorsa yalnızca \_\_LARGE\_\_ SEMBOLİK SABİTİ TANIMLANMIŞ SAYILIR.

\_\_MSDOS\_\_

DOS ile UNIX arasındaki taşınabilirlik problemlerini çözmek için kullanılır. Eğer MSDOS altındaki bir derleyici ile çalışıyorsanız \_\_MSDOS\_\_ sembolik sabiti tanımlanmış kabul edilmektedir.

```
#ifndef    __MSDOS__  
#define    far  
#endif
```

Yukarıdaki örnekte eğer MSDOS ile çalışılmıyorsa far anahtar sözcüğü silinmektedir.

\_\_TURBOC\_\_

Borland firmasının turbo C uyarlamalaerından bir tanesinde çalışıyorsanız bu sembolik sabit derleyiciniz tarafından tanımlanmış varsayılır. Sembolik sabitin tanımlanan değeri (test için önemli olamaz) uyarlamadan uyarlamaya değişebilmektedir.

\_\_BORLANDC\_\_

Borland firmasının BORLAND C uyarlamalaerından bir tanesinde çalışıyorsanız bu sembolik sabit derleyiciniz tarafından tanımlanmış varsayılır. Sembolik sabitin tanımlanan değeri (test için önemli olamaz) uyarlamadan uyarlamaya değişebilmektedir.

\_\_cplusplus

C++ ismiyle piyasaya çıkan derleyiciler standart C yi de destekler. Bu durumda derleyicilerin hangi C uyarlamasında çalıştıkları bu sembolik sabit yardımıyla öğrenilebilir.