

C Programlama Dili

Kurs Notları

Kaan ASLAN

C ve Sistem Programcıları Derneği

Güncelleme Tarihi: 12/07/2003

Bu kurs notları Kaan ASLAN tarafından yazılmıştır. Kaynak belirtilmek koşuluyla her türlü alıntı yapılabilir.

C Programlama Dilinin Tarihsel Gelişimi

C Programlama Dili 1970-1971 yıllarında AT&T Bell Lab.'ta UNIX İşletim Sisteminin geliştirilmesi sürecinde bir yan ürün olarak tasarlandı. AT&T o zamanlar Multics isimli bir işletim sistemi projesinde çalışıyordu. AT&T bu projeden çekildi ve kendi işletim sistemlerini yazma yoluna saptı. Bu işletim sistemine Multics'ten kelime oyunu yapılarak UNIX ismi verildi. UNIX DEC firmasının PDP-7 makinalarında ilk kez yazılmıştır.

O zamanlar işletimsistemleri sembolik Makine dilinde yazılıyordu. Ken Thompson (grup lideri) işleri kolaylaştırmak için B isimli bir programlama dilciği tasarladı. Sonra Dennis Ritchie bunu geliştirerek C haline getirdi. C Programlama dili UNIX proje ekibindeki Dennis Ritchie tarafından geliştirilmiştir.

UNIX İşletim Sistemi 1973 yılında sil baştan yeniden C ile yeniden yazılmıştır. O zamana kadar bir işletimsistemi yüksek seviyeli bir dilde yazılmış değildi. Bununla UNIX'in C'de yazılması bir devrim niteliğindedir. UNIX sayesinde C Programlama Dili 1970'lerde tanınmaya başladı. 1980-81 yıllarında IBM ilk kişisel bilgisayarı çıkarttı. C Programlama Dili kişisel bilgisayarlarda kullanılan en yaygın dil oldu.

1978 yılında Dennis Ritchie ve Brian Kernigan tarafından "The C Programming Language" kitabı yazıldı. Bu kitap tüm zamanların en önemli bilgisayar kitaplarından biridir. (Hatta HelloWorld programı ilk kez burada yazılmıştır.)

C Programlama dili ilk kez 1989 yılında ANSI tarafından standardize edildi. ISO 1990 yılında ANSİ standartları olarak (bölüm numaralandırmasını değiştirerek) ISO standardı olarak onayladı. Bu standardın resmi ismi ISO/IEC 9899: 1990'dı. Bu standartlar kısaca C90 ismiyle bilinmektedir. C 1999 yılında bazı özellikler eklenerek yeniden standardize edildi. Bu standartlara da ISO/IEC 9899: 1999 denilmektedir ve kısaca C99 olarak bilinir. C99 derleyici yazan firmalar ve kurumlar tarafından çok fazla destek görmedi. C'ye nihayet 2011 yılında bazı özellikler eklenmiştir. Bu son standartlar ISO/IEC 9899:2011 kod adıyla yayınlanmıştır. Buna da kısaca C11 denilmektedir. Bu kursta klasik C olan C90 ele alınmaktadır. C99 ve C11'deki yeni özellikler "Sistem Programlama Ve C İleri C Uygulamaları" kursunda ele alınacaktır. C denildiğinde default olarak akla C90 gelmelidir.

Anahtar Notlar: Bu kurs C Programlama Dilini her yönüyle anlatan bir kurstur. Bu kurstan sonra aşağıdaki kurslara katılınabilir:

- Sistem Programlama ve İleri C Uygulamaları (I) (kesinlikle tavsiye edilir)
- C++ (kesinlikle tavsiye edilir)
- Qt ile C++'ta Programlama (pencereli Windows ve Linux programları yazmak için)
- UNIX/Linux Sistem Programlama
- Windows Sistem Programlama
- Sistem Programlama ve İleri C Uygulamaları (II)

- PIC mikrodenetleyicileri ile programlama (biraz elektronik bilgisi tabanı gerekir).

C Programlama Dili şu an itibari ile Tiobe Index'e göre dünyanın en fazla kullanılan programlama dilidir.

Programlama Dillerinin Sınıflandırılması

Programlama dilleri teorisyenler tarafından genellikle üç biçimde sınıflandırılmaktadır:

- 1) Seviyelerine Göre Sınıflandırma
- 2) Kullanım Alanlarına Göre Sınıflandırma
- 3) Programlama Modeline Göre Sınıflandırma

Seviyelerine Göre Sınıflandırma

Seviye (level) bir programlama dilinin insan algısına yakınlığının bir ölçüsüdür. Yüksek seviyeli diller insana yakın yani kolay dillerdir. Alçak seviyeli diller makinaya yakın fakat zor öğrenilen dillerdir. Olabilecek en aşağı seviyeli dil saf makina dilidir (machine language). Bu dil yalnızca 1'lerden ve 0'lardan oluşur. Görsel diller, veritabanı dilleri vs. çok yüksek seviyeli dilledir. Java, C#, Pascal, Basic, PHP vs. diller yüksek seviyeli dillerdir. C Programlama Dili orta seviyeli bir dildir. Yani makinaya diğer dillerden daha yakındır.

Kullanım Alanlarına Göre Sınıflandırma

Programlama dilleri bazı konularda bazı iddialara sahiptir. Bazı dillerde web işlemleri yapmak daha kolayken, bazı dillerde veritabanı işlemleri yapmak daha kolaydır. İşte bir dilin hangi tür uygulamalarda kullanılabileceğine yönelik sınıflandırma bu.

Bilimsel ve Mühendislik Diller: Bu tür diller bilimsel ve mühendislik problemlerin çözülmesi için birincil olarak tercih edilen dillerdir. Örneğin C, C++, Fortran, Pascal, Java, C#, Matlab vs.

Veritabanı Dilleri: Bunlar veritabanlarının oluşturulması ve idaresinde tercih edilen dillerdir. SQL, Foxpro, Dbase vs. gibi...

Web Dilleri: İnteraktif web sayfalarını oluşturabilmek için tercih edilen dillerdir. Örneğin Java, C#, PHP, Python gibi...

Yapay Zeka Dilleri: İnsan düşüncesini eşitli boyutlarda taklit eden programlara yapay zeka programları denir. Bu programların yazılması için tercih edilen dillere yapay zeka dilleri denir. Örneğin C, C++, Java, C#, Lisp, Prolog vs. gibi

Görsel ve Animasyon Dilleri: Animasyon programları için kullanılan yüksek seviyeli script dilleridir. Action Script gibi...

Sistem Programlama Dilleri: Sistem Programlama yapmak için kullanılan dilledir. Tipik olarak C, C++ ve sembolik makina dilleri sistem programlama dilleridir.

Anahtar Notlar: Sistem programları bilgisayar donanımı ile arayüz oluşturan, uygulama programlarını çeşitli bakımlardan hizmet veren, aşağı seviyeli taban programlardır. Örneğin: İşletim sistemleri, editörler, derleyiciler ve bağlayıcılar, haberleşme programları vs.

Programlama Modeline Göre Sınıflandırma

Programlama modeli o dilde programı hangi teknikle yazdığımıza ilişkindir. Bazı dillerin bazı tekniklerin kullanılmasını açıkça desteklemektedir. Temel olarak (fakat daha fazla olabilir) programlama dillerini

programlama modellerine (programming paradigm) 4 biimde sınıflandırabiliriz:

1) Prosedürel Programlama Dilleri: Bu diller prosedürel programlama modelini destekleyen dillerdir. Örneğin C prosedürel bir programlama dilidir. Prosedürel teknikte program altprogramların (prosedürlerin) birbirlerini çağırmasıyla yazılır. Bunun dışında klasik Pascal, Basic, Fortran vs. gibi diller prosedürel dillerdir.

2) Nesne Yönelimli Programlama Dilleri: Bu diller nesne yönelimli programlama modelini (object oriented programming) uygulayabilecek yetenkte dillerdir. Bu modelde program sınıflar oluşturularak yazılır. C++, C'nin nesne yönelimli versiyonudur. Pek prosedürel dilin daha sonra nesne yönelimli versiyonları oluşturulmuştur. Örneğin Object Pascal, VB.NET. C#, Java temelde nesne yönelimli dillerdir.

3) Fonksiyonel Programlama Dilleri: Bu tür diller fonksiyonel programlama modelini (functional programming paradigm) açıkça destekleyen dillerdir. Örneğin Haskell, F#, R, Scheme, Erlang. Fonksiyonel teknikte programlar sanki formül yazarmış gibi yazılırlar.

4) Çok Modelli Programlama Dilleri: Bu dillerde birden fazla programlama modeli kullanılabilir. Örneğin C++'ta hem prosedürel hem de nesne yönelimli teknik kullanılabilir. C++ çok modellenli (multi paradigm) bir programlama dilidir. Yeni özelliklerle C# ve Java'da artık çok modellenli olmuştur.

Temel Kavramlar

İşletim Sistemi (Operating System): İşletim sistemi makinanın donanımını yöneten temel bir sistem yazılımıdır. Bir kaynak yöneticisi gibi çalışır. Yönettiği tipik donanımsal kaynaklar şunlardır: İşlemci, RAM, Disk, Yazıcı, Network kartı ve haberleşme sistemi. Eğer işletim sistemi olmasaydı bilgisayar hiç kolay kullanılamazdı. İşletim sistemi temel pek çok hizmeti bizim için sağlamaktadır. Kullanıcı ile makine donanımı arasında köprü kurmaktadır. İşletim sistemi insan vücudu gibi çeşitli alt sistemlere sahiptir. Bu alt sistemler karşılıklı etkileşim halindedir. İşletim sistemlerini kabaca çekirdek (kernel) ve kabuk (shell) olmak üzere iki katmana ayırabiliriz. Çekirdek makinanın donanımını yöneten motor kısımdır. Kabuk kullanıcı ile ilişki kurankısımdır. (Örneğin Windows'un masaüstü kabuk durumundadır). Tipik olarak (fakat her zaman değil) biz işletim sisteminin üzerinde çalışırız.

En çok kullanılan Masaüstü işletim sistemleri Windows, Linux (UNIX türevi sistemler), Mac OS X sistemleridir. En çok kullanılan mobil işleim sistemleri de Android, IOS, Windows Mobile sistemleridir.

Çevirici Programlar, Derleyiciler ve Yorumlayıcılar:

Bir programlama dilinde yazılmış olan programı eşdeğer olarak başka bir dile dönüştüren programlara çevirici programlar (translators) denilmektedir. Burada asıl programın diline kaynak dil (source language), dönüştürülecek dile hedef dil (target language) denilmektedir. Bir çevirici programda hedef dil alçak seviyeli bir dilse (sembolik Makine dili, ara kod ya da saf makine dili) böyle çevirici programlara derleyici (compiler) denir.

Yorumlayıcılar (interpreters) kaynak kodu doğrudan okuyup o anda çalıştırırlar. Bir hedef kod üretmezler. Dolayısıyla bunlar çevirici program durumunda değildirler. Bazı diller için yalnızca derleyiciler kullanılır (Örneğin C gibi). Bazıları için yalnızca yorumlayıcılar kullanılır (örneğin PHP, Perl gibi), Bazı dillerin derleyicileri de yorumlayıcıları da vardır (örneğin Basic gibi). Genel olarak yorumlayıcı yazmak derleyici yazmaktan daha kolaydır. Derleyiciler kodun daha hızlı çalışmasına yol açarlar. Yorumlayıcılarda kaynak kod gizlenemez.

Açık Kaynak Kod ve Özgür Yazılım

1980'li yılların ortalarında Richard Stallman, FSF (Free Software Foundation) isimli bir dernek kurdu ve özgür yazılım (free software) kavramını ortaya attı. Özgür yazılım kabaca, yazılan programların kaynak kodlarını sahiplenmemek, başkalarının da onu devam ettirmesini sağlamak anlamına gelir. Birisi özgür yazılım lisansına sahip bir ürünü değiştirdiği ya da özelleştirdiği zaman onu o da açmak zorundadır. Fakat alıcı varsa bunlar parayla da satılabilir. Açık kaynak kod (open source) aşağı yukarı (fakat tamamen değil) özgür yazılımla aynı anlama

gelmektedir.

Her yazılımın bir lisansı vardır. Fakat çeşitli yazılım akımları için çeşitli lisanslar kalıp olarak oluşturulmuştur. Örneğin Özgür yazılımın ana lisansı GPL(Gnu Public Licence)'dir. Fakat GPL'ye benzer pek çok lisans vardır (BSD, Apache, MIT, ...).

Richard Stallman FSF'yi kurduğunda GNU isimli bir proje başlattı. Bu projenin amacı bedava, açık kaynak kodlu bir işletim sistemi ve geliştirme araçları yazmaktır. Gerçekten de gcc derleyicisi, ld bağlayıcısı ve diğer pek çok utility program bu proje kapsamında yazıldı.

Linux işletim sistemi GNU malzemeleri kullanılarak Linus Torwalds'ın önderliğinde geniş bir ekip tarafından geliştirilmiştir ve sürdürülmektedir. Linux aslında bir çekirdek geliştirme projesidir. Bugün Linux diye dağıtılan sistemlerde binlerce açık kaynak kodlu yazılım bulunmaktadır. Yani Linux adeta GNU projesinin işletim sistemi gibi olmuştur (maalesef GNU projesinin öngörülen işletim sistemi bir türlü son haline getirilememiştir.) Bu nedenle bazı kesimler bu işletim sisteminin isminin Linux değil GNU/Linux olması gerektiğini söylemektedir.

Çok kullanılan C Derleyicileri

Pek çok derleyicisi olsa da bugün için en önemli iki derleyici Microsoft'un C derleyicileri ve gcc derleyicileridir. Ekiden Borland derleyicileri de çok kullanılıyordu. Intel firmasının da ayrı bir C derleyicisi vardır. UNIX/Linux sistemlerinde gcc derleyicileri çok yoğun kullanılırken, Windows sistemlerinde Microsoft C derleyicileri yoğun kullanılmaktadır. Gcc'nin Windows portuna MinGW denilmektedir.

IDE Kavramı

Normal olarak derleyiciler komut satırından çalıştırılan programlardır. Yani biz programı editör denilen bir ortamda yazarız. Bunu save ederiz. Sonra da komut satırından derleriz. Bu tarz çalışma biraz zahmetli olduğu için IDE (Integrated Development environment) denilen özel yazılımlardan faydalanılır. IDE'lerin editörleri vardır, menüleri vardır ve birtakım araçları vardır. IDE derleyici değildir. Derleyicinin dışındaki yazılım geliştirmeyi kolaylaştıran araçlar toplamıdır.

Bugün pek çok IDE seçenek olarak bulunmaktadır. Microsoft'un ünlü IDE'sinin ismi “Visual Studio”dur. Kursumuzda ağırlıklı bu IDE kullanılacaktır. Eclipse, Netbeans, MonoDevelop gibi open source IDE'ler de vardır. Apple'ın da X-Code denilen kendi bir IDE'si vardır.

IDE derleyici değildir. IDE'ye “derle” denildiği zaman IDE gerçek derleyiciyi çağırır. Biz bilgisayarımıza bir IDE yüklediğimizde yalnızca IDE değil, onun kullanacağı derleyici de yüklenecektir. Tabi IDE olmadan yalnızca derleyiciyi de yükleyebiliriz.

Visual Studio paralı bir IDE'dir. Ancak Microsoft “Express Edition” ismiyle bu IDE'nin bedava bir versiyonunu da oluşturmuştur. Bu kurs için ve genel olarak C programlama için Express Edition yeterlidir.

Visual Studio Windows sistemlerinde kullanılabilen bir IDE'dir. Oysa Eclipse ve Netbeans IDE'leri cross platform'dur. Linux sistemlerinde IDE olarak QtCreator, Eclipse, Netbeans ya da MonoDevelop tercih edilebilir. Şu anda Visual Studio IDE'sinin son versiyonu “Visual Studio 2013”tür.

Sayı Sistemleri

Biz günlük hayatımızda 10'luk sistemi kullanmaktayız. 10'luk sistemde sayıları ifade etmek için 10 sembol kullanılır:

0 1 2 3 4 5 6 7 8 9 (toplam 10 tane)

Aslında 10'luk sistemdeki sayıların her bir basamağı 10'un kuvvetlerini temsil eder. Örneğin:

$$123 = 3 * 10^0 + 2 * 10^1 + 1 * 10^2$$

10'luk sistemde her bir basamağ "digit" denilmektedir.

Makina ikilik sistemi kullanır. Çünkü sayıların elektriksel olarak ifade edilmeleri ikilik sistemde çok kolaydır ve standartlaşma böyle oluşmuştur. Elektroniğin bu alanına dijital elektronik denilmektedir.

İkilik sistemde toplam 2 sembol vardır:

0 ve 1 (toplam 2 tane)

İkilik sistemdeki bir sayı ikinin kuvvetleriyle çarpılarak elde edilir. Örneğin:

$$1010 = 0 * 2^0 + 1 * 2^1 + 0 * 2^2 + 1 * 2^3$$

İkilik sistemdeki her bir basamağa bit (binary digit'ten kısaltma) denilmektedir. Örneğin:

10100010 sayısı 8 bitlik bir sayıdır.

Bilgisayarımızın belleği de yalnızca bitleri tutar. Yani bellekteki herşey bit'lerden oluşmaktadır. Bit en düşük bellek birimidir.

8 bite byte denilmektedir. Kilo 1024luk sistemde 1000 katı anlamına gelir (örneğin 1 Kilometre 1000 metredir.) Fakat bilgisayar bilimlerinde kilo olarak 1000 anlamlı gözükmemektedir. Çünkü 1000 sayısı 10'un kuvvetidir. İşte bilgisayar bilimlerinde kilo 1024 katı anlamına gelir (1024, 2'nin 10'uncu kuvvetidir ve 1000'e de benzemektedir.). Mega da kilonun 1024 katıdır. Bu durumda 1 MB = 1024 * 1024 byte'tır. Giga da mega'nın 1024 katıdır.

Bilgisayarın belleğindeki her şey 2'lik sistemdeki sayılardan oluşur: Komutlar, yazılar, sayılar hep aslında ikilik sistemde bulunurlar.

Tamsayıların 2'lik Sistemde İfade Edilmesi

Tamsayılar için iki sistem söz konusudur: İşaretsiz sistem ve işaretli sistem. İşaretsiz (unsigned) sistemde tamsayı ya pozitif ya da sıfır olabilir. Fakat negatif yorumlanamaz. İşaretli (signed) sistemde tamsayı negatif de olabilir. Örneğin aşağıdaki 8 bit sayı işaretsiz tamsayı olarak yorumlanırsa kaçtır?

$$1000\ 1010 = 138$$

8 bitle işaretsiz tamsayı sisteminde ifade edilecek en küçük sayı 0, en büyük sayı 255'tir.

$$0000\ 0000 \ \rightarrow\ 0$$

$$\begin{array}{c} \dots\dots\dots \\ 1111\ 1111 \end{array} \ \rightarrow\ 255$$

2 byte (16 bit) ile işaretsiz sistemde ifade edilebilecek en küçük sayı 0, en büyük sayı 65535'tir.

$$0000\ 0000\ 0000\ 0000 \ \rightarrow\ 0$$

$$\begin{array}{c} \dots\ \dots\ \dots\ \dots \\ 1111\ 1111\ 1111\ 1111 \end{array} \ \rightarrow\ 65535$$

İşaretili tamsayıları ifade etmek için tarih boyunca 3 sistem denenmiştir. Bugün ikiye tümleme sistemi denilen sistem en iyi sistem olarak kullanılmaktadır. Bu sistemin özellikleri şöyledir:

- Sayının en solundaki bit işaret bitidir. Bu bit 1 ise sayı negatif, 0 ise pozitifdir.
- Negatif ve pozitif sayılar birbirlerinin ikiye tümleyenidirler.
- Bu sistemde bir tane sıfır vardır.

Bir sayının ikiye tümleyene sayının 1'e tümleyenine 1 eklenerek elde edilir (Sayının 1'e tümleyeni 1'lerin 0, 0'ların 1 yapılmasıyla elde edilir.) Örneğin:

Sayı: 1011 1010
1'e tümleyeni: 0100 0101
2'ye tümleyeni: 0100 0101 + 1 = 0100 0110

Bu işlemin kolay bir yolu vardır: Sayının sağında sola doğru ilk 1 görene kadar (ilk 1 dahil olmak üzere) aynı yazılarak ilerlenir. Sonra 0'lar 1, 1'ler 0 yazılarak devam edilir. Örneğin:

Sayı: 1010 1100
2'ye tümleyeni: 0101 0100

Örneğin:

Sayı: 1111 1111
2'ye tümleyeni: 0000 0001

Sayının 2'ye tümleyenininin 2'ye tümleyeni sayının kendisine eşittir. Örneğin:

Sayı: 1111 1111
2'ye tümleyeni: 0000 0001
2'ye tümleyenin 2'ye tümleyeni: 1111 1111

Bu sistemde negatif ve pozitif sayılar birbirlerinin 2'ye tümleyenidirler. Örneğin:

0000 1010 --> +10
1111 0110 --> -10

Bu sistemde negatif bir sayı yazmak istersek önce onun pozitifliğini yazıp sonra 2'ye tümleyenini alarak yazabiliriz. (Tabi doğrudan yazabiliyorsak ne mutlu bize.) Örneğin, bu sistemde -1 yazmak isteyelim:

0000 0001 --> +1
1111 1111 --> -1

Bu sistemde birisi bize “bu sayı kaç” diye sorduğunda şöyle yanıtı bulabiliriz: Önce sayının işaret bitine bakıp onun pozitif mi yoksa negatif mi olduğunu belirleriz. Sayı pozitif ise doğrudan hesaplarız. Negatif ise, sayının ikiye tümleyeni alırız. Ondan faydalanarak sayıyı belirleriz. Örneğin, aşağıdaki sayı işaretili tamsayı sisteminde kaçtır?

1111 0101

Sayı negatif, 2'ye tümleyenini alalım:

0000 1011

bu +11 olduğuna göre demek ki o sayı -11'dir.

Bu sistemde 1 tane sıfır vardır. Bu sistemde 2 tuhaf sayı vardır ki bunların 2'ye tümleyenleri yoktur (yani kendisiyle aynıdır). Bunlar tüm bitleri 0 olan sayı ve ilk biti 1, diğer bitleri 0 olan sayıdır:

0000 0000 (2'ye tümleyeni yok)

1000 0000 (2'ye tümleyeni yok)

İşte bu ilk sayı 0'dır, diğeri ise -128'dir. Yani 8 bit içerisinde işaretli tamsayı sisteminde yazılabilecek en büyük pozitif ve en küçük negatif sayılar şöyledir:

0111 1111 --> +127

1000 0000 --> -128

Peki 2 byte (16 bit) içerisinde yazılabilecek en büyük ve en küçük işaretli sayılar nelerdir?

0111 1111 1111 1111 --> +32767

1000 0000 0000 0000 --> -32768

Peki 4 byte (32 bit) içerisinde yazılabilecek en büyük ve en küçük işaretli sayılar nelerdir?

0111 1111 1111 1111 1111 1111 1111 1111 --> +2147483647

1000 0000 0000 0000 0000 0000 0000 0000 --> -2147483648

Soru: Aşağıdaki 1 byte'lık sayı işaretli ve işaretsiz tamsayı sistemlerinde kaçtır?

1111 1111

Cevap: İşaretsiz sistemde +255, işaretli sistemde -1'dir.

Peki bu sistemde örneğin 1 byte içerisindeki en büyük pozitif sayıya 1 toplarsak ne olur? Bir kere zaten sınırı aşmış oluruz. Peki aşınca ne olur?

0111 1111 --> +127

0000 0001

1000 0000 --> -128

Peki 2 toplasaydık ne olurdu? Yanıt: 1000 0001 = -127

Demek ki bu sistemde pozitif sınırı yanlışlıkla aşarsa kendimi negatif yüksek sayılarda buluruz. Buna programlamada üstten taşma (overflow) denilmektedir. Taşma alttan da (underflow) olabilir. Örneğin -128'den 1 çıkartırsak +127 elde ederiz.

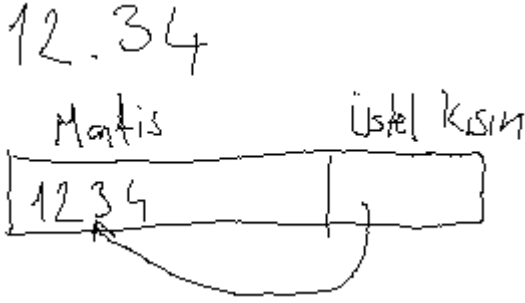
Gerçek Sayıların (Noktalı Sayıların) 2'lik Sistemde İfade Edilmesi

Noktalı sayıları 2'lik sistemde ifade edebilmek için tarih boyunca iki tür format grubu kullanılmıştır. Sabit noktalı (fixed point) formatlarda noktanın yeri sabittir. Onun solunda sayının tam kısmı, sağında noktadan sonraki kısım utulur. Örneğin 12.567 bu tür formatlarda şöyle tutulmaktadır:

Tabi sayının tam ve noktalı kısımları burada 10'luk sistemde değil, 2'lik sistemde tutulmaktadır.

Sabit noktalı formatlar pek verimli değildir. Çünkü dinamik değildir. Yani sayının tam kısmı büyük fakat noktalı kısmı küçükse ya da tersi durumda sayı ifade edilemez. Halbuki yeteri kadar yer vardır. Bunun için kayan noktalı

(floating point) formatlar geliştirilmiştir. Bugün bilgisayarlarımızda bu formatlar kullanılmaktadır. Kayan noktalı formatlarda sayı sanki noktası yokmuş gibi yazılır. Noktanın yeri ayrıca format içerisinde belirtilir. Örneğin:



Sayının noktası kaldırılmış olan haline mantis (manissa) denir. Noktanın yerini belirten kısma ise genellikle üstel kısım (exponential part) denilmektedir. Çeşitli kaynak noktalı formatlar olsa da bugün hemen tüm sistemlerde IEEE'nin 754 numaralı formatı kullanılmaktadır.

Tabi 2'lik sistemde sayının noktadan sonraki kısmı 2'nin negatif kuvvetleriyle çarpılarak oluşturulmaktadır. Bu da yuvarlama hatası (rounding error) bir hatanın oluşmasına zemin hazırlar. Yuvarlama hatası noktalı bir sayının tam olarak ifade edilemeyip ona yakın bir sayının ifade edilmesiyle oluşan bir hatadır. Yuvarlama hatası elimine edilemez. Olsa olsa onun etkisi azaltılabilir. Bunun için de sayının daha geniş olarak ifade edilmesi yoluna gidilir. Örneğin C, C# ve Java'da float türü 4 byte'lık gerçek sayı türünü, double türü 8 byte'lık gerçek sayı türünü temsil eder. Ve C programcıları bu nedenle en çok double türünü kullanırlar.

Anahtar Notlar: İlk elektronik bilgisayarlar 40'lı yıllarda yapıldı. Bunlar da vakum tüpler kullanılıyordu. Sonra transistör icad edildi. 50'li yıllarda bilgisayarlar transistörlerle yapılmaya başlandı. 70'li yıllarda entegre devre (integrated circuit) teknolojisi geliştirilince artık bilgisayarların işlem birimi entegre devre olarak yapılmaya başlandı. Bunlara mikroişlemci dediler. Kişisel bilgisayarlar bunlardan sonra gelişti.

Yuvarlama hataları her noktalı sayıda değil bazı sayılarda ortaya çıkar. Sayı ilk kez depolanırken de oluşabilir. İşlem sonucunda da oluşabilir. Bu nedenle programlama dillerinde gerçek sayıların "eşit mi", "eşit değil mi" biçiminde karşılaştırılması iyi bir teknik değildir.

Yazıların 2'lik Sistemde İfade Edilmesi

Yazılar karakterlerden oluşur. Yazının her karakteri bir byte kodlanabilir. Böylece aslında yazı 2'lik sistemde sayılarla ifade edilmiş olur. İşte hangi karakterin hangi sayıyla ifade edileceğini belirlemek için ASCII (American Standard Code Information Interchange) tablosu denilen bir tablo yaygın bir biçimde kullanılmaktadır. ASCII tablosu orijinal olarak 7 bitti. Sonra bu 8'bite çıkartıldı. ASCII tablosunun uzunluğu 256 satırdır. Bu 256 satır ile tüm karakterler ifade edilemez. (Japonlar ne yapsın?). ASCII tablosunun ilk 128 karakteri standarttır. Diğer 128 karakteri ülkelere göre değişmektedir. Böylece ASCII tablosunun çeşitli varyasyonları oluşmuştur. Bunlara code page denilmektedir. Ancak son 15 yıldır artık karakterlerin 2 byte ile kodlanması yaygınlık kazanmaya başlamıştır. UNICODE denilen tablo 2 byte'lık uluslararası kabul görmüş en önemli tablodur. Ve bugün artık en yoğun kullanılan tablo haline gelmiştir.

16'lık Sayı Sistemi (Hexadecimal system)

16'lık sayı sisteminde toplam 16 sembol vardır:

0 1 2 3 4 5 6 7 8 9 A B C D E F

16'lık sistem bilgisayar bilimlerinde 2'lik sistemin basit ve yoğun bir gösterimini sağlamak için kullanılır. 16'lık sistemdeki her hex digit 4 bit ile ifade edilebilir:

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

16'lık sistemdeki bir sayıyı 2'lik sisteme dönüştürmek için her hex digit için 4 bit yazılır. Örneğin:

C4A8 = 1100 0100 1010 1000

Tam ters işlem de yapılabilir. Örneğin:

1001 0101 1100 0010 = 95C2

8'lik Sayı Sistemi (Octal System)

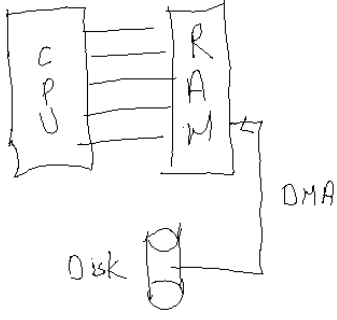
Sekizlik sistemde her bir octal digit 3 bitle açılır. Örneğin:

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

Örneğin 8'lik sistemdeki 756 sayısı 111 101 110 biçiminde 2'lik sisteme dönüştürülebilir.

Bilgisayarın Basit Mimarisi

Bir bilgisayar sisteminde üç önemli birim vardır: CPU, RAM ve Disk. İşlemleri yapan elektronik devrelerin bulunduğu bölüme CPU denir. Bugün CPU'lar entegre devre biçiminde üretilmektedir ve onlara mikroişlemci de denilmektedir. CPU ile elektriksel olarak bağlantılı olan ve CPU'nun çalışması sırasında sürekli kullanılan belleklere Ana Bellek (Main Memory) ya da Birincil Bellek (Primary Memory) denilmektedir. Bunlar RAM teknolojisiyle üretildiklerinden bunlara RAM de denir. Bir programın çalışması için ana belleğe yüklenmesi gerekir. Bilgisayarın güç kaynağı kesildiğinde ana bellekte bilgiler kaybolur. Bu durumda güç kaynağı kesilince bilgileri tutan birime ihtiyaç duyulmuştur. İşte onlara İkincil Bellekler (Secondary Storage Devices)" denilmektedir. Bugün kullandığımız diskler, flash EPROM'lar, EEPROM bellekler ves. hep ikincil belleklerdir.



Bir program diskte bulunur. Onu çalıştırmak istediğimizde işletim sistemi onu diskten alır, birincil belleğe yükler ve orada çalıştırır. Örneğin :

$a = b + c;$

gibi bir ifadeye a, b ve c RAM'dedir. CPU önce a ve b'yi RAM'den çeker. Toplama işlemini yapar, sonucu yeniden RAM'deki c'ye yazar.

Dil Nedir?

Dil olgusunun tek bir cümleyle tanımını yapmak kolay değildir. Fakat “iletişimde kullanılan semboller kümesi” biçiminde tanımlanabilir. Diller kabaca doğal diller ve yapay diller (ya da kurgusal diller) olmak üzere ikiye ayrılır. Doğal dillerde sentaks kesin ve açık olarak ifade edilemez. İstisnalar vardır ve bunlar kurala dönüştürülemezdir. Yapay diller ise insanlar tarafından tasarlanmış kurgusal dillerdir. Programlama dilleri yapay dillerdir.

Bir dilin bütün kurallarına gramer denir. Gramerin iki önemli alt alanı vardır: Sentaks ve semantik. Doğal dillerin fonetik gibi, morfoloji gibi başka kurallar kümesi olsa da sentaks ve semantik bütün dillerde olan en önemli olgudur. Bir olguya dil diyebilmek için kesinlikle sentaks ve semantik kuralların bulunuyor olması gerekir.

Bir dil aslında yalın elemanlardan oluşur. (örneğin doğal dillerdeki sözcükler). Buelemanlar gelişigüzel dizilemez. İşte sentaks dildeki öğelerin doğru yazılmasına ve doğru dizilmesine ilişkin kurallardır. Örneğin aşağıda bir sentaks hatası yapılmıştır:

i am school going to

Aşağıda yine bir sentaks hatası yapılmıştır:

```
if a == 2)(
```

Doğru yazılmış öğelerin ne anlam ifade ettiğine ilişkin kurallara semantik denilmektedir. Yani örneğin “I am going to school” sentaks olarak düzgün bir sünlemdir. Fakat ne denilmek istenmektedir. Bu semantiktir. Benzer biçimde:

```
if (a == 10)
    printf(“evet\n”);
```

Sentaks olarak geçerlidir. Semantik olarak “a 10'a eşitse ekrana evet yazdır” anlamına gelir.

Bilgisayar Dilleri (Computer Languages) ve Programlam Dilleri (Programming Languages)

Bilgisayar bilimlerinde kullanılan, insanlar tarafından tasarlanmış ve grameri matematiksel olarak ifade edilebilen dillere bilgisayar dilleri denir. Bilgisayar bilimlerinde kullanılan ve sentaks ve semantik yapıya sahip her olgu bir bilgisayar dilidir. (html, xml vs.). Bir bilgisayar dili özellikle akış ögesi de içeriyorsa bir programlama dilidir. Her programlama dili bilgisayar dilidir. Fakat tersi her zaman doğru değildir. (C, C++, Java, C# vs)”

C dili bir programlama dilidir.

Bir C Programını Oluşturmak

Bir C programının çalıştırılabilir program haline getirilmesi için genel olarak aşağıdaki adımlar uygulanır:

1. Kaynak dosyanın oluşturulması:

Kaynak dosya metin düzenleyici bir programla yazılır. C dilinin kurallarına göre yazılan dosyanın uzantısı geleneksel olarak “.c” dir.

2. Kaynak dosyanın derleyici program (compiler) tarafından derlenmesi:

Derleyici program kullanılarak .c uzantılı dosya derlenir. Derleme işlemi yazılan programın C dilinin kurallarına uygunluğunun belirlenmesi işlemidir. Derleme işlemi her zaman başarılı olmayabilir. Derleme işlemi başarılıysa derleyici program ismine amaç dosya (object file) denilen bir dosya üretir. Amaç dosyalar işletim sistemine göre değişiklik gösterebilir. Örneğin Windows sistemleri için amaç dosya uzantısı “.obj” dur. Unix/Linux sistemleri için amaç dosyanın uzantısı genel olarak “.o” dur.

Derleyiciler komut satırından kolayca çalıştırılabilen programlardır. Başka programlar tarafından çağrılabilmesi için basit bir kullanıma sahiptirler.

3. Amaç dosya(lar) bağlayıcı (linker) program tarafından birleştirilerek çalıştırılabilir (executable) dosya oluşturulur.

Anahtar Notlar: Windows sistemlerinde çalıştırılabilir dosyalar “.exe” uzantılıdır. Unix/Linux sistemlerinde uzantının önemi yoktur. Özel bir biti set edilmiş dosyalar Unix/Linux sistemleri için çalıştırılabilir dosya olarak algılanır.

Windows sistemlerinde derleme ve bağlama işlemi

Windows sistemlerinde genel olarak kullanılan Microsoft firmasının cl isimli derleyicisidir. Bu program bağlama işlemini de yapabilmektedir.

Anahtar Notlar: Windows sistemlerinde komut yorumlayıcı program üzerinde “cd” komutu ile dizin geçişleri yapılabilir. (cd c:\Users\csd\Desktop\Test)

cl derleyici programının hem derleme hem de bağlama işlemini yapabildiği basit bir kullanımı şu şekildedir:

```
cl hello.c
```

Üretilen hello.exe programı aşağıdaki gibi çalıştırılabilir.

```
hello
```

Unix/Linux Sistemlerinde Derleme ve Bağlama İşlemi

Unix/linux sistemlerinde derleyici program olarak çoğunlukla “gcc” kullanılmaktadır. Unix/Linux sistemleri için bu program parasız olarak kurulabilmektedir.

Anahtar Notlar: Bir çok Linux dağıtımı bulunmaktadır. Bunlardan lisanssız olarak kullanılabilen çeşitli dağıtımlar da mevcuttur. (Ubuntu, Mint, Debian, Open Suse vs). Bu sistemler sanal makine programlarına da kurulabilir. (Virtualbox vs.)

gcc programıyla derlemeve bağlama işlemi basit olarak aşağıdaki biçimde yapılabilir.

```
gcc -o hello hello.c
```

Burada “hello” isimli çalıştırılabilir bir dosya üretilenektir. Bu sistemlerde programın çalıştırılması terminal penceresinde aşağıdaki gibi yapılabilir.

```
./hello
```

Visual Studio IDE'sinde Derleme ve Çalıştırma İşlemleri

Visual Studio IDE'si ile bir programı derleyip çalıştırmak için sırasıyla şunların yapılması gerekir:

1) Öncelikle bir proje oluşturmak gerekir. Projeler tek başlarına değil “Solution” denilen kapların içerisinde bulunur. Dolayısıyla bir proje yaratırken aynı zamanda bir solution da yaratılır. Proje yaratmak için File/New/Project seçilir. Karşımıza “New Project” dialog penceresi çıkar. Burada template olarak “Visual C++” “Win32 Console Application” seçilir. Sonra Proje dizininin yaratılacağı dizin ve Proje ismi belirlenir. “Create Directory for Solution” seçen kutusu default çarpılmış durumda olabilir. Bunun çarpısı kaldırılabilir. Bu durumda solution bilgileri ile proje bilgileri aynı klasörde tutulacaktır. Bu dialog penceresi kapatılınca karşımıza yeni bir dialog penceresi gelir. Burada “Empty Project” seçilmelidir. SDL seçen kutusu da unchecked yapılabilir.

2) Solution yaratıldıktan sonra bununla ilgili işlem yapabilmek için “Solution Explorer” denilen bir pencere kullanılır. Solution Explorer View menüsü yoluyla,, araç çubuğu simgesiyle ya da Ctrl + Alt + L kısayol tuşuyla açılabilir. Solution Explorer “yuvalanabilir (dockable)” bir penceredir. Solution Explorer bir “treeview” kontrolü içermektedir.

3) Artık projeye bir kaynak dosyanın eklenmesi zamanı gelmiştir. Bunun için “Project/Add New Item” menüsü ya da Solution Explorer'da projenin üzerinde bağlam menüsünden “Add/New Item” seçilir. Karşımıza “Add New Item” dialog penceresi gelir. Burada kaynak dosyaya isim vererek onu yaratırız. Dosya uzantısının .c olması gerekmektedir. Artık programı dosyaya yazabiliriz.

4) Programı derlemek için “Build/Compile” seçilir. Link işlemi için “Build/Build Solution” ya da “Build/Build XXX” seçilir (Buradaki XXX projenin ismidir). Programı çalıştırmak için “Debug/Start Without Debugging” ya da Ctrl+F5 tuşlarına basılır. Sonraki bir aşama seçilirse zaten öncekiler de yapılır. Böylece programı derleyip, bağlay çalıştırabilmek için tek yapılacak şey Ctrl+F5 tuşlarına basmaktır.

5) Projeyi tekrar açabilmek için “File/Open/Project-Solution” seçilir. Projenin dizinine gelinir ve .sln dosyası seçilir.

6) IDE'den çıkmadan projeyi kapatmak için “File/Close Solution” seçilir.

Atom (Token) Kavramı

Bir programlama dilindeki anlamlı en küçük birime atom (token) denir. Program aslında atomların yan yana gelmesiyle oluşturulur. (Atomlar doğal dillerdeki sözcüklere benzetilebilirler). Örneğin “merhaba dünya” programını atomlarına ayıralım:

```
#include <stdio.h>

int main(void)
{
    printf(“Merhaba Dunya\n”);
}
```

```
    return 0;
}
```

Atomlar:

```
# include    <    stdio.h    >    int    main    (    void
) {    printf    (    “Merhaba Dunya”    )    ;
return    0    ;    }
```

Atomlar daha fazla parçaya ayrılamazlar. Atomları altı gruba ayırabiliriz:

1) Anahtar Sözcükler (Keywords/Reserved Words): Dil için özel anlamı olan, değişken olarak kullanılması yasaklanmış sözcüklerdir. Örneğin if, for, int, return gibi...

2) Değişkenler (Identifiers/Variables): İsmi bizim istediğimiz gibi verebildiğimiz atomlardır. Örneğin: x, y, count, printf gibi...

3) Sabitler (Literals/Constants): Bir değer ya b,ir değişkenin içerisinde bulunur ya da doğrudan yazılır. İşte doğrudan yazılan sayılara sabit denir. Örneğin:

```
a = b + 10;
```

ifadesinde a ve b birer değişken atom, 10 ise sabit atomdur.

4) Operatörler (Operators): Bir işleme yol açan, işlem sonucunda bir değer üretilmesini sağlayan atomlara operatör denir. Örneğin:

```
a = b + 10;
```

Burada + ve = birer operatördür.

5) String'ler (Strings): Programlama dillerinde iki tırnak içerisine yazılmış yazılar iki tırnaklarıyla tek bir atom belirtirler. Bunlara string denir. Örneğin: “Merhaba Dünya\n” gibi...

6) Ayıraçlar (Delimiters/Punctuators): Yukarıdaki grupların dışında kalan, ifadeleri ayırmak için kullanılan tüm atomlar ayıraç atomdur. Örneğin ; gibi, } gibi...

Sentaks Gösterimleri

Programlama dillerinin sentakslarını betimlemek için en çok kullanılan yöntem BNF notasyonu ve türevleridir. Ancak biz kurulumuzda açısız parantez, köşeli parantez tekniğini kullanacağız. Açısız parantezler içerisinde yazılanlar mutlaka bulunması zorunlu öğeleri, köşeli parantez içerisindekiler “yazılmasa da olur (optional)” öğeleri belirtir. Örneğin:

```
[geri dönüş değerinin türü] <fonksiyon ismi>([parametre bildirimi])
{
    /* ... */
}
```

Açısız ve köşeli parantezlerin dışındaki tüm atomlar aynı biçimde bulundurulmak zorundadır.

Merhaba Dünya Programının Açıklaması

C programlarının başında genellikle aşağıdaki satır bulunur:

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

Bu bir önışlemci (preprocessor) komutudur. Bu komut açısıl parantez içerisindeki dosyanın içeriğinin derleme sırasında komutun yerleştirildiği yere kopyalanacağını belirir. (Yani bu satırı silip onun yerine stdio.h isimli dosyanın içeriğini oraya yerleştirmek aynı etkiye yol açar). Neden bu dosyanın içeriğinin bulunmasına gereksinim duyulduğu ileride açıklanacaktır.

C programları kabaca fonksiyonlardan (functions) oluşur. Altprogramlara C'de fonksiyon denilmektedir. Bir fonksiyonun tanımlanması (definition) o fonksiyonun bizim tarafımızdan yazılması anlamına gelir. Çağırılması (call) onun çalıştırılması anlamına gelir. “Merhaba Dünya” programında main fonksiyonu tanımlanmıştır. Fonksiyon tanımlamanın genel biçimi şöyledir:

```
[geri dönüş değerinin türü] <fonksiyon ismi> ([parametre bildirimi])  
{  
    /* ... */  
}
```

Anahtar Notlar: Genel biçimlerdeki /* ... */ gösterimi “burada birşeyler var, fakat biz şimdilik onlarla ilgilenmiyoruz” anlamına gelmektedir.

İki küme parantezi arasındaki bölgeye blok denilmektedir. C'de her fonksiyonun bir ana bloğu vardır. C programları main isminde bir fonksiyondan çalışmaya başlar. main bitince program da biter. Örnek programımızda main fonksiyonunda printf isimli bir fonksiyon çağırılmıştır. Printf standart bir C fonksiyonudur. Standart C fonksiyonu demek, tüm C derleyicilerinde bulunmak zorunda olan, derleyicileri yazarlar tarafından zaten yazılmış olarak bulunan fonksiyonlar demektir. Bir fonksiyon çağırmanın genel biçimi şöyledir:

```
<fonksiyon ismi>([argüman listesi]);
```

printf fonksiyonu iki tırnak içerisindeki yazıları imlecin (cursor) bulundauğu yerden itibaren ekrana yazar. İmleç program çalıştığıında sol üst köşededir. printf imleci yazının sonunda bırakır. “\n” “imleci aşağıdaki satırın başına geçir” anlamına gelir.

return deyimi fonksiyonu sonlandırır. Eğer return yoksa fonksiyon kapanış küme parantezine geldiğinde sonlanır.

Bir C programında istenildiği kadar çok fonksiyon tanımlanabilir. Bunların sırasının bir önemi yoktur. Main fonksiyonunun da özel bir yerde bulunması gerekmez. Fakat program her zaman main fonksiyonundan çalışmaya başlar.

C'de iç içe fonksiyon tanımlanamaz. Her fonksiyon diğerinin dışında tanımlanmak zorundadır. Örneğin:

```
#include <stdio.h>  
  
foo()  
{  
    printf("I am foo\n");  
}  
  
main()  
{  
    foo();  
}
```

Anahtar Notlar: Örneklerimizde fonksiyon isimleri uydurulurken foo, bar, tar gibi, func gibi isimler kullanılacaktır. Bu isimlerin hiçbir özel anlamı yoktur.

İfade (Expression) Kavramı

Değişkenlerin, sabitlerin ve operatörlerin herbir birleşimine (kombinasyonuna) ifade denir. Örneğin:

```
x
30
x + 30
x + 30 - y
foo()
```

birer ifadedir. Görüldüğü gibi tekbaşına bir değişken ve sabit ifade belirtir, fakat tek başına bir operatör ifade belirtmez.

Nesne (Object) Kavramı

Bellekte yer kaplayan ve erişilebilir olan bölgeler nesne denir. Örneğin programlama dillerindeki değişkenler tipik birer nesnedir. Bir ifade ya bir nesne belirtir ya da nesne belirtmez. Örneğin:

100 bir ifadedir fakat nesne belirtmez. Örneğin:

x bir ifadedir ve nesne belirtir. Örneğin:

x + 10 bir ifadedir, nesne belirtmez.

Sol Taraf Değeri (Left Value) ve Sağ Taraf Değeri (Right Value)

Nesne belirten ifadelerle sol taraf değeri (lvalue), belirtmeyene ifadeleri sağ taraf değeri (rvalue) denilmektedir. Örneğin:

```
10 --> sağ taraf değeri
x --> sol taraf değeri
x + 10 --> sağ taraf değeri
```

Sol taraf değeri denmesinin nedeni atama operatörünün solunda bulunabilmesindedir. Sağ taraf değeri denmesinin sebebi atama operatörünün solunda bulunamamasındandır (tipik olarak sağında bulunduğu için).

C'nin Veri Türleri

Tür (type) bir nesnenin bellekte kaç byte yer kaplayacağını ve onun içerisindeki 0'ların ve 1'lerin nasıl yorumlanacağını anlatan temel bir kavramdır. Her nesnin C'de bir türü vardır. Ve bu tür programın çalışması sırasında değişmez. (Aslında yalnızca nesnelerin değil genel olarak her ifadenin bir türü vardır. Bundan ilerde bahsedilecektir.)

C'nin temel türleri aşağıdaki tabloda gösterilmektedir:

Tür	Belirten Anahtar Sözcükler	Uzunluk (byte) Windows (UNIX/Linux)	Sınır Değerler (Windows)
[signed] int		4 (4)	[-2147483648, +2147483647]

unsigned [int]	4 (4)	[0, +4294967295]
[signed] short [int]	2 (2)	[-32768, +32767]
unsigned short [int]	2 (2)	[0, +65535]
[signed] long [int]	4 (8)	[-2147483648, +2147483647]
unsigned long [int]	4 (8)	[0, +4294967295]
char	1 (1)	[-128, +127] [0, +255]
signed char	1 (1)	[-128, +127]
unsigned char	1 (1)	[0, +255]
float	4 (4)	[$\pm 3.6 \cdot 10^{-38}$, $\pm 3.6 \cdot 10^{+38}$]
double	8 (8)	[$\pm 1.8 \cdot 10^{-308}$, $\pm 1.8 \cdot 10^{+308}$]
long double	8 (8)	[$\pm 1.8 \cdot 10^{-308}$, $\pm 1.8 \cdot 10^{+308}$]
[signed] long long	8 (8)	[-9223372036854775808, +9223372036854775807]
unsigned long long (C99 ve C11 ve C++11)	8 (8)	[0, +18446744073709551615]
Bool (C99, C11)	1 (1)	true, false

- int türü işaretli bir tamsayı türüdür. int türünün uzunluğu sistemden sisteme değişebilir. Standartlarda en az 2 byte olması zorunlu tutulmuştur. Ancak derleyicileri yazarların isteğine bırakılmıştır.

- C'de her tamsayı türünün işaretli ve işaretsiz versiyonları vardır. int türünün işaretsiz biçimi unsigned int türüdür. Yalnızca unsigned demekle unsigned int demek aynı anlamdadır.

- Standartlara göre short türü ya int türü kadardır ya da int türünden küçüktür. Örneğin DOS'ta int türü de short türü de 2 byte uzunluktadır. Oysa Windows'ta ve UNIX/Linux sistemlerinde int türü 4 byte, short türü 2 byte uzunluktadır.

- short türünün işaretsiz biçimi unsigned short türüdür.

- long türü standartlara göre ya int türü kadar olmak zorundadır ya da ondan büyük olmak zorundadır. 32 ve 64 bit Windows sistemlerinde long türü int türü aynı uzunluktadır (4 byte). 32 bit UNIX/Linux sistemlerinde long türü 4 byte, 64 bit UNIX/Linux sistemlerinde 8 byte'tır.

- long türünün işaretsiz biçimi unsigned long türüdür.

- char türü standartlara göre her sistemde 1 byte olmak zorundadır. (Bu türün ismi belki byte olsaydı daha iyiydi). char ismi her ne kadar karakteri çağırıyor olsa da bunun karakterle bir ilgisi yoktur. char türü C'de 1 byte uzunlukta bir tamsayı türüdür. C'de yalnızca char denildiğinde bunun signed char mı, yoksa unsigned char mı olacağı derleyicileri yazarların isteğine bırakılmıştır. Örneğin Windows ve UNIX/Linux sistemlerindeki derleyicilerde char denildiğinde signed char anlaşılmaktadır (fakat bu durum ayarlardan da değiştirilebilmektedir).

- C'de gerçek sayı türlerinin işaretli ve işaretsiz biçimleri yoktur. Onlar zaten her zaman default işaretlidir.

- float türü en az 4 byte olması öngörülen bir gerçek sayı (noktalı sayı) türüdür. float türünün yuvarlama hatalarına direnci zayıftır. Bu nedenle float yerine programcılar daha çok double türünü tercih ederler.

- double türü standartlara göre en az float kadar olmak zorundadır. Yani float türüyle aynı duyarlılıkta olabilir ya da ondan daha geniş olabilir.
- long double türü en az double kadar olmak zorundadır. double ile aynı uzunlukta olabilir ya da ondan daha uzun olabilir.
- Birden fazla sözcükten oluşan türler için bu sözcüklerin yerleri değiştirilebilir (örneğin signed long int yerine int signed long denilebilir.)
- C'de en fazla kullanılan tamsayı türü int, en fazla kullanılan gerçek sayı türü double türüdür. Programcının öncelikle bunları tercih etmelidir. Özel durum varsa diğerlerini düşünmelidir.
- C99'da long long isimli bir tamsayı türü daha eklendi. Bu türün long türünden daha uzun olması öngörülmüştür. Standartlara göre bu tür ya long türüyle aynı uzunluktadır ya da ondan daha uzundur.
- C'ye C99'la birlikte nihayet bir bool türü de eklenmiştir. Fakat klasik C'de böyle bir tür yoktur.

Derleyicilerin Hata Mesajları

Derleyicilerin hata mesajları üçe ayrılmaktadır:

- 1. Uyarılar (Warnings):** Uyarılar gerçek hatalar değildir. Program içerisindeki program yapmış olabileceği olası mantık hatalarına dikkati çekmek için verilirler. Uyarılar derleme işleminin başarısızlığına yol açmazlar. Ancak programcıların uyarılara çok dikkat etmesi gerekir. Çünkü pek çok uyarıda derleyici haklı bir yere dikkat çekmektedir.
- 2. Gerçek Hatalar (Errors):** Bunlar dilin sentaks ve semantik kurallarına uyulmaması yüzünden verilirler. Bunların mutlaka düzeltilmesi gerekir. Bir programda bir tane bile “error” olsa program başarılı olarak derlenemez.
- 3. Ölümcül Hatalar (Fatal Errors):** Derleme işleminin bile devam etmesini engelleyen ciddi hatalardır. Normal olarak bir programda ne kadar hata olursa olsun tüm kod gözden geçirilir. Tüm hatalar en sonunda listelenir. Fakat bir ölümcül hata olduğunda artık derleme işlemi sonlandırılır. Ölümcül hatalar genellikle sistemdeki ciddi sorunlar yüzünden ortaya çıkmaktadır (örneğin diskte yeterli alan olmayabilir, ya da sistemde yeterli RAM bulunmuyor olabilir.)

Verilen hata mesajlarının metinleri derleyiciden derleyiciye değişebilir. Ayrıca bir hata durumunda bir derleyici buna bir mesaj verirken diğeri daha fazla mesaj verebilir.

C Programlarının Geçerliliği

Standartlar bir C programının geçerliliği ve derlenip derlenmeyeceği konusunda şunları söylemektedir:

1. Standartlarda belirtilen sentaks ve semantik kurallara uygun programlar kesinlikle derlenmek zorundadır.
2. Standartlara uygun bir program başarılı olarak derlendikten sonra birtakım mesajlar (diagnostics) da verilebilir. (Tabi bu durumda bu mesajlar uyarı mesajları olacaktır)
3. Sentaks ve semantik kurallara uyulmadığı her durumda derleyici bu durumu belirten bir mesaj (diagnostic) vermek zorundadır. Bu mesaj error ya da warning olabilir.
4. Geçersiz bir program yine de başarılı olarak derlenebilir. Ya da tersten şöyle düşünebiliriz: “Bir programın başarılı olarak derlenmesi onun geçerli olduğu anlamına gelmez”.

Özellikle 4. Madde C programcıları tarafından maalesef bilinmemektedir. Bazı programcılar sırf derleyici programlarını derledi diye programlarının geçerli olduğunu sanmaktadırlar. Böyle bir program başka C derleyicileri tarafından derlenmeyebilir.

Taşınabilirlik (Portability) Nedir?

Taşınabilirlik bir programlama dilinde yazılmış olan programın başka bir sisteme götürüldüğünde orada derlenerek sorunsuz çalışabilmesine denilmektedir. Taşınabilirlik bu anlamda bir standadizasyonun bulunmasını gerektirir. Çünkü derleyicileri yazanlar hep aynı kuralları kabul etmişlerse taşınabilirlik oluşabilir. C Programlama dili oldukça taşınabilir bir dildir. Burada söz konusu edilen taşınabilirlik kaynak kodun taşınabilirliğidir. Yoksa biz derlenip exe yapılmış bir programı başka bir sisteme götürüp orada çalıştıramayız. Ancak bunun mümkün olduğu ortamlar da vardır (.NET ve Java gibi). Bu ortamlarda derlenmiş olan kod yeniden derlenmeden başka sistemlere götürüldüğünde çalıştırılabilmektedir.

Bildirim ve Tanımlama Kavramları (Declaration & Definitions)

C, C++, C# ve Java gibi katı tür kontrolünün uygulandığı dillerde (strongly typed languages) bir değişken kullanılmadan önce derleyiciye tanıtılmak zorundadır. Kullanılmadan önce değişkenlerin derleyiciye tanıtılması işlemine bildirim (declaration) denilmektedir. Bir bildirim yapıldığında eğer derleyici bildirilen değişken için bellekte bir yer ayırıyorsa o bildirim aynı zamanda bir tanımlama (definition) işlemidir. Yani tanımlama derleyicinin yer ayırdığı bildirim işlemleridir. Bildirim daha geneldir. Her tanımlama bir bildirimdir fakat her bildirim bir tanımlama değildir. Kurulumuzda aksi belirtilmediği sürece bildirimler aynı zamanda tanımlama işlemi olarak kabul edilmelidir.

Bildirim işleminin genel biçimi şöyledir:

```
<tür> <değişken listesi>;
```

Örneğin:

```
int a;  
long b, c, d;  
double x, y;
```

Değişken listyesi bir'den fazla değişkenden oluşuyorsa onları ayırmak için ',' atomu kullanılır. Atomlar arasında istenildiği kadar boşluk karakteri bırakılabildiğine göre aşağıdaki bildirim d geçerlidir.

```
long b  
, c, d;
```

C'de (C90) bildirimler 3 yerde yapılabilir:

- 1) Blokların başlarında. Yani blok açıldığında henüz hiçbir fonksiyon çağrısı yapılmadan bildirimler yapılmalıdır. Blok başlarında bildirilen değişkenlere yerel değişkenler (local variables) de denilmektedir.
- 2) Tüm blokların dışında. C'de tüm blokların dışında bildirilen değişkenlere global değişkenler (global variables) de denilmektedir.
- 3) Fonksiyonların parametre parantezleri içerisinde. Böyle bildirilmiş olan değişkenlere “parametre değişkenleri (parameters)” denilmektedir.

Değişken isimlendirmede şu kurallar söz konusudur:

- C büyük harf-küçük harf duyarlılığı olan (case sensitive) bir programlama dilidir. Yani büyük harflerle küçük harfler tamamen farklı karakterlermiş gibi ele alınırlar.

- Değişkenler boşluk içermez. Değişken isimleri sayısal karakterlerle başlatılamaz. Ancak alfabetik karakterlerle ya da _ ile başlatılıp sayısal devam ettirilebilirler.

- Değişken isimlendirmede yalnızca İngilizce büyük harfler, küçük harfler, sayısal karakterler ve alt tire karakterleri kullanılabilir. (Yani değişkenlere Türkçe isimler veremeyiz).

- C standartlarına göre değişken uzunlukları için derleyiciler minimum 32 karakteri dikkate almak zorundadır. Bu limit en az 32 olmak koşuluyla derleyiciden derleyiciye değişebilir. Fakat daha uzun değişken isimleri derleyici tarafından geçerli olarak değerlendirilir. ncak en az ilk 32 karakter ayırıcı olarak dikkate alınacaktır.

- Anahtar sözcükler de değişken ismi olarak kullanılamazlar

Değişkenlere İlkdeğer Verilmesi (Initialization)

Bildirim sırasında bildirim işleminin bir parçası olarak değişkenlere değer vermeye ilkdeğer verme (initialization) denilmektedir. Örneğin:

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

burada a ve c'ye ilkdeğer verilmemiştir, fakat b'ye verilmiştir. İlkdeğer vermekle ilk kez değer vermek çoğu zaman aynı etkiye yol açsa da gramatik olarak aynı şey değildir. Örneğin:

```
int a = 0;    /* ilkdeğer verme işlemi */
```

```
int b;  
b = 10;     /* ilkdeğer verme işlemi değil, ilk kez değer verme işlemi */
```

İçerisine heniz değer atanmamış yerel değişkenlerin içerisinde rastgele değerler vardır. C terminolojisinde buna çöp değer (garbage value) denilmektedir. Halbuki ilkdeğer verilmemiş global değişkenlerin içerisinde kesinlikle sıfır değeri bulunur.

Nesnelerin İçerisindeki Değerlerin printf Fonksiyonuyla Yazdırılması

printf fonksiyonunda iki tırnak içerisindeki karakterler ekrana yazdırılır. Fakat iki tırnağın içerisinde % karakteri varsa, printf bu % karakterini ve yanındaki bir ya da birkaç karakteri format karakteri olarak kabul eder. printf % karakterini ve onu izleyen formak karakterlerini ekrana yazdırmaz. Bunlar yer tutucudur. Bunların yerine iki tırnaktan sonraki ifadelerin değerleri yazdırılır. İki tırnak içerisindeki her format karakteri sırasıyla iki tırnaktan sonraki argümanlarla eşleştirilir. Formak karakterleri yerine bunların değerleri yazdırılır. Örneğin:

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

```
main()  
{  
    int a = 10, b = 20;  
  
    printf("a = %d, b = %d\n", a, b);    /* a = 10, b = 20 */  
    printf("a = %d, b = %d\n", b, a);    /* a = 20, b = 10 */  
    printf("%d%d\n", a, b);            /* 1020 */  
}
```

% karakterinin yanındaki format karakterleri nelerdir? Format karakterleri yazdırılacak ifadenin türüne bağlıdır. Ayrıca format karakterleri yazdırma işleminin nasıl yapılacağını da (örneğin kaçlık sistemde) belirler. Temel format karakterleri ve anlamları şöyledir:

Format Karakterleri	Anlamı
%d	int, short ve char türlerini 10'luk sistemde yazdırır
%ld	long int türünü 10'luk sistemde yazdırır
%x, %X	int, short ve char türlerini hex sistemde yazdırır
%lx, %lX	long int ve unsigned long int türlerini hex sistemde yazdırır
%u	unsigned int, unsigned short ve unsigned char türlerini 10'luk sistemde yazdırır
%lu	unsigned long int türünü 10'luk sistemde yazdırır
%f	float ve double türlerini 10'luk sistemde yazdırır
%c	char, short ve int türlerini karakter görüntüsü olarak yazdırır
%o	char, short ve int türlerini octal sistemde yazdırır
%lo	long ve unsigned long türlerini octal sistemde yazdırır

printf %f formatında default olarak noktadan sonra 6 basamak yazdırır. Yazdırılacak değer noktadan sonra 6 basamaktan fazlaysa yuvarlama yapılır. eğer printf ile noktadan sonra istediğimiz kadar basamak yazdırmak istiyorsak %.nf formatı kullanılmalıdır (burada n yerine bir sayı olmalıdır. Örneğin %.10f gibi).

scanf Fonksiyonuyla Klavyeden Okuma Yapılması

scanf fonksiyonunun kullanımı printf fonksiyonuna çok benzemektedir. Yine scanf fonksiyonunun bir format kısmı vardır. Bu format kısmını içerisine okunan değer yerleştirileceği nesne adresleri izler. scanf'te iki tırnak içerisinde yalnızca format karakterleri bulunmalıdır. Bu fonksiyon iki tırnak içerisindekileri ekrana yazdırmaz. scanf'te iki tırnak içerisindeki format karakterlerinin dışındaki karakterler tamamen başka anlama gelirler. scanf fonksiyonunda değer yerleştirileceği nesnelerin başında & operatörü bulunmalıdır. (Bu & operatörü bir adres operatörüdür. Nesnenin adresini elde etmekte kullanılır. Örneğin:

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

```
main()
{
    int a;

    printf("sayi giriniz:");
    scanf("%d", &a);
    printf("Girilen deger: %d\n", a);
}
```

scanf kullanırken şunlara dikkat etmek gerekir:

- İki tırnak içerisinde yalnızca format karakteri bulunmalıdır. Orada boşuk karakterleri ya da \n karakteri bulunmamalıdır.
- Değişkenlerin önündeki & operatörü unutulmamalıdır.

scanf fonksiyonuyla bir'den fazla değer girilirken girişler arasında istenildiği kadar boşluk karakterleri bırakılabilir. Örneğin:

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

```
main()
{
    int a;
```

```
int b;

printf("İki sayı giriniz:");
scanf("%d%d", &a, &b);
printf("a = %d, b = %d\n", a, b);
}
```

printf fonksiyonundaki format karakterleri scanf fonksiyonunda klavyeden girişi belirlemektedir. Yani örneğin biz bir int değeri printf ile %x kullanarak yazdırsak bu değer 16'lık sistemde ekrana yazdırılır. Fakat bir int değeri scanf ile %x ile okumak istersek klavyeden yaptığımız girişin 16'lık sistemde olduğu kabul edilir. Örneğin:

```
#include <stdio.h>

main()
{
    int a, b;
    printf("Bir sayı giriniz : ");
    scanf("%x", &a);
    printf("a = %d\n", a);
}
```

printf fonksiyonunda hem float hem de double türleri %f ile yazdırılır. Ancak scanf fonksiyonunda float %f ile, double %lf ile okunur. Örneğin:

```
#include <stdio.h>

main()
{
    float f;
    double d;

    printf("float bir deger giriniz:");
    scanf("%f", &f);

    printf("double bir deger giriniz:");
    scanf("%lf", &d);

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

Sınıf Çalışması

İsmi a, b, c olan double türden 3 değişken tanımlayınız. Sonra a ve b için klavyeden scanf fonksiyonuyla okuma yapınız. Bu ikisinin toplamını c'ye atayınız ve c'yi yazdırınız.

Çözümü

```
#include <stdio.h>

main()
{
    double a, b, c;

    printf("a degerini giriniz :");
    scanf("%lf", &a);

    printf("b degerini giriniz :");
    scanf("%lf", &b);

    c = a + b;
```

```
    printf("c = %f\n", c);  
}
```

Fonksiyonların Geri Dönüş Değerleri (return value)

Bir fonksiyon çağrıldığında akış fonksiyona gider. Fonksiyonun içerisindeki kodlar çalışır. Fonksiyonun çalışması bitince akış kalınan yerden devam eder. İşte fonksiyonun çalışması bittiğinde onu çağırana fonksiyona ilettiği değer geri dönüş değeri denilmektedir. Fonksiyonun geri dönüş değerinin de bir türü vardır. Bu tür tanımlama sırasında fonksiyonun isminin soluna yazılır. Örneğin:

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

```
long bar()  
{  
    /* ... */  
}
```

Fonksiyonun geri dönüş değeri yerine birşey yazılmazsa sanki int yazılmış gibi işlem görür. Yani örneğin:

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

ile

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

tamamen aynı anlamdadır. C99 ve C11'de geri dönüş değerinin yazılmaması seçeneği kaldırılmıştır. C'nin bu yeni versiyonlarında fonksiyonların geri dönüş değerlerinin türleri yazılmak zorundadır.

Örneğin:

```
result = foo() * 2;
```

Burada önce foo fonksiyonu çağrılır, geri dönüş değeri elde edilir, ikiyle çarpılır ve result değişkenine atanır.

Fonksiyonun geri dönüş değeri return deyimiyle oluşturulur. return deyiminin genel biçimi şöyledir:

```
return [ifade];
```

Programın akışı return anahtar sözcüğünü gördüğünde önce ifadenin değeri hesaplanır, sonra fonksiyon bu değerle sonlandırılır. Yani return deyiminin iki işlevi vardır:

- 1) Fonksiyonu sonlandırır
- 2) Geri dönüş değerini oluşturur

Örneğin:

```
#include <stdio.h>

int foo()
{
    printf("Foo\n");

    return 100;

    printf("test\n");    /* unreachable code! */
}

int main()
{
    int result;

    result = foo() * 2;
    printf("%d\n", result);
}
```

Anahtar Notlar: Linux Sanal makinaya nasıl kurulur? Öncelikle sanal makina programını bilgisayarımıza yüklememiz gerekir. Bunun için iki önemli seçenek vardır. Birincisi VMWare firmasının VMWare programı, ikincisi open source VirtualBox programı. VMWare normalde paralı bir programdır. Fakat bunun VMWare Player isminde bedava bir sürümü de vardır. Sanal makinaya Linux kurulur (örneğin Mint sürümü). Kurulum doğrudan ISO dosyasıyla yapılabilir. Kurulum sırasında bizden bir user name ve password istenecektir (Örneğin derste yaptığımız kurulumda user name = csd ve password = csd1993 verdir.)

C'de bir fonksiyonun geri dönüş değeri olduğu halde return ile belli bir değere geri dönülmezse geri dönüş değeri olarak çöp bir değer elde edilir. (Halbuki C# ve Java gibi bazı dillerde fonksiyonların geri dönüş değeri varsa return kullanmak zorunludur.)

C'de main fonksiyonun geri dönüş değeri int olmak zorundadır. Yani örneğin void, long vs. olamaz. Ayrıca main fonksiyonunda hiç return kullanmazsak sanki 0 ile geri dönmüş gibi işlem görür. Yani main fonksiyonunda return kullanmamakla ana bloğun sonunda 0 ile geri dönmek aynı anlamdadır. Örneğin:

```
#include <stdio.h>

int main()
{
    printf("I am main\n");

    return 0;    /* bu olmasaydı da aynı anlam oluşacaktı */
}
```

Fonksiyonun geri dönüş değeri yerine void anahtar sözcüğü yazılırsa bu durum fonksiyonun geri dönüş değerinin olmadığı anlamına gelir. Böyle fonksiyonlara void fonksiyonlar denir. void fonksiyonlarda return kullanılabilir fakat yanına ifade yazılamaz. void fonksiyonlarda return kullanılmamışsa fonksiyon ana blok sonlanınca sonlanır. Örneğin:

```
#include <stdio.h>

void foo()
{
    printf("foo\n");
}

int main()
{
```

```
printf("I am main\n");  
  
foo();  
  
return 0;  
}
```

Fonksiyonun geri dönüş değeri return işlemi sırasında önce geçici bir değişkene aktarılır, oradan alınarak kullanılır. Örneğin:

```
return 100;    ----> temp = 100;  
...  
result = foo() * 2----> result = temp * 2;
```

Aslında return işlemi geçici değişkene yapılan atama işlemidir. Fonksiyonun geri dönüş değerinin türü de geçici değişkenin türüdür. Geçici değişken derleyici tarafından return işlemi sırasında yaratılır, kullanım bittiğinde yok edilir. return işlemi de bir çeşit atama işlemidir.

Değişkenlerin Faaliyet Alanları (Scope)

Bir değişkenin derleyici tarafından tanınabildiği program aralığına faaliyet alanı (scope) denilmektedir. Değişkenler belirli faaliyet alanlarına sahiptir. C'de 3 çeşit faaliyet alanı vardır:

- 1) Blok faaliyet alanı (block scope)
- 2) Fonksiyon faaliyet alanı (function scope)
- 3) Dosya faaliyet alanı (file scope)

Bir fonksiyonun bir ana bloğu olmak zorundadır. O ana bloğun içerisinde istenildiği kadar çok iç içe ya da ayrık blok açılabilir. Örneğin:

```
void foo()  
{  
  {  
    ...  
    {  
      ...  
    }  
  }  
  {  
    ...  
  }  
}
```

C'de (C++, C# ve Java'da da öyle) iç içe fonksiyon bildirilemez. Örneğin:

```
void foo()  
{  
  ...  
  void bar() /* geçersiz */  
  {  
    ...  
  }  
  ...  
}
```

```
}
```

Blok faaliyet alanı yalnızca bir blokta ve o bloğun kapsadığı bloklarda tanınma alanıdır. Fonksiyon faaliyet alanı tek bir fonksiyonun her yerinde tanınma aralığıdır. Dosya faaliyet alanı bir dosyanın her yerinde tanınma aralığıdır.

Yerel Değişkenlerin Faaliyet Alanları

Yerel değişkenler blok faaliyet alanı kuralına uyarlar. Yerel değişken hangi blokta bildirilmiş yalnızca o blokta ve bloğun kapsadığı bloklarda kullanılabilir. Örneğin:

```
void foo()
{
    int a;

    {
        int b;
        /* a ve b burada kullanılabilir */
    }
    /* a burada kullanılabilir, b kullanılamaz */
}
```

Örneğin:

```
#include <stdio.h>

int main()
{
    int a;

    {
        int b;

        b = 20;
        a = 10;
        printf("a = %d, b = %d\n", a, b);    /* geçerli */
    }
    printf("a = %d\n", a);                  /* geçerli */
    printf("b = %d\n", b);                  /* error! */

    return 0;
}
```

C'de aynı faaliyet alanına sahip birden fazla aynı isimli değişken tanımlanamaz. Farklı faaliyet alanlarına sahip aynı isimli değişkenler tanımlanabilir. Bu durumda örneğin, iç içe yerel bloklarda aynı isimli değişkenler tanımlanabilir. Örneğin:

```
#include <stdio.h>

void foo()
{
    int a;
    {
        int a;    /* geçerli */
        /* ... */
    }
    /* ... */
}
```

```

int main()
{
    int a;    /* geçerli */

    return 0;
}

```

C'de bir blokta bir'den fazla değişken faaliyet gösteriyorsa o blokta o değişken ismi kullanıldığında dar faaliyet alanına sahip olan değişkene erişilir. Örneğin:

```

#include <stdio.h>

int main()
{
    int a;

    a = 10;

    {
        int a;

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

    return 0;
}

```

Global Değişkenlerin Faaliyet Alanı

Bildirimleri fonksiyonların dışında yapılan değişkenlere global değişkenler denir. Global değişkenler dosya faaliyet alanına (file scope) sahiptir. Yani tüm fonksiyonlarda tanınırlar.

```

#include <stdio.h>

int a;

void foo()
{
    a = 10;
}

int main()
{
    a = 20;
    printf("%d\n", a);    /* 20 */
    foo();
    printf("%d\n", a);    /* 10 */

    return 0;
}

```

Bir global değişkenle aynı isimli yerel değişkenler tanımlanabilir. Çünkü bunlar farklı faaliyet alanlarına sahiptir. Tabi ilgili blokta bu değişken ismi kullanıldığında dar faaliyet alanına sahip olana (yani yerel olana) erişilir.

Örneğin:

```
#include <stdio.h>

int a;

void foo()
{
    int a;

    a = 10;    /* yerel olan a */
}

int main()
{
    a = 20;    /* global olan a */
    printf("%d\n", a);    /* 20 */
    foo();
    printf("%d\n", a);    /* 20 */

    return 0;
}
```

C'de derleme işleminin bir yönü vardır. Bu yön yukarıdan aşağıya doğrudur. Derleyicinin önce değişkenin bildirimini görmesi gerekir. Bu nedenle bir global değişkeni aşağıda bildirip daha yukarıda kullanamayız. Örneğin:

```
#include <stdio.h>

void foo()
{
    a = 10;    /* geçersiz! */
}

int a;

int main()
{
    a = 10;    /* geçerli */
    printf("%d\n", a);    /* geçerli */

    return 0;
}
```

Bu durumda global değişkenler için en iyi bildirim yeri programın tepesidir.

Fonksiyonların Parametre Değişkenleri

Fonksiyonlar onları çağıran fonksiyonlardan değerler alabilirler. Bunlara fonksiyonların parametre değişkenleri (parameters) denilmektedir. Parametre değişkenlerini bildirmenin C'de iki yolu vardır: Eski biçim (old style) ve yeni biçim (modern style). Biz kursumuzda hiç eski biçimi kullanmayacağız. Bu eski biçim çok eskiden kullanılıyordu. Yeni biçim ortaya çıkınca programcılar bunu kullanmaz oldular. C standartları oluşturulduğunda her iki biçimi de destekledi. Eski biçim C++'ta C99'da ve C11'de desteklenmemektedir.

Eski biçimde parametre bildirimini yapılırken önce parametre parantezlerinin içerisine parametre değişkenlerinin isimleri aralarına virgül atomu getirilerek listelenir. Daha sonra henüz ana blok açılmadan bunların bildirimleri yapılır. Örneğin:

```
void foo(a, b, c)
    int a;
```

```
    long b, c;
{
    ...
}
```

Yeni biçimde parametre değişkenleri parametre parantezinin içerisinde tür belirtilerek bildirilir. Örneğin:

```
void foo(int a, long b, long c)
{
    ...
}
```

Yeni biçimde parametre değişkenleri aynı türden olsa bile her defasında tür belirten atomları yeniden belirtmek gerekir. Örneğin:

```
void foo(int a, b)    /* geçersiz! */
{
    /* ... */
}
```

bildirimini şöyle yapılması gerekirdi:

```
void foo(int a, int b)    /* geçerli */
{
    /* ... */
}
```

Parametrelili bir fonksiyon çağrılırken parametre sayısı kadar argüman kullanılır. Örneğin:

```
foo(10, 20);
```

Argümanlar ',' operatörü ile birbirlerinden ayrılmaktadır. Argüman olarak herhangi birer ifade (expression) kullanılabilir. Örneğin:

```
foo(a + b - 10, c + d + 20);
```

Anahtar Notlar: Bir fonksiyon çağrılırken parametre parantezinin içerisine yazılan ifadelere argüman (argument), fonksiyonun parametre değişkenlerine kısaca parametre (parameter) denilmektedir.

Parametrelili bir fonksiyon çağrıldığında önce argümanların değerleri hesaplanır. Sonra argümanlardan parametre değişkenlerine karşılıklı bir atama yapılır. Sonra programın akışı fonksiyona geçer. Örneğin:

```
#include <stdio.h>

void foo(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);
}

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

    foo(x, y);
    foo(x + 10, y + 20);
    foo(100, 200);

    return 0;
}
```

```
}
```

Fonksiyonların parametre değişkenleri faaliyet alanı bakımından sanki ana bloğun başında tanımlanmış değişkenler gibi işlem görür. Yani fonksiyonların parametreye değişkenleri o fonksiyonda tanınabilmektedir. Aynı faaliyet alanına sahip aynı isimli değişken tanımlanamayacağına göre aşağıdaki bildirim de geçerli değildir:

```
void foo(int a, int b)
{
    long a;    /* geçerli değil! */
    ...
}
```

Fakat:

```
void foo(int a, int b)
{
    ...
    {
        long a;    // geçerli */
        ...
    }
    ...
}
```

Bir fonksiyon parametreleriyle aldığı değere işlemler uygulayıp sonucu geri dönüş değeri olarak verebilir. Örneğin:

```
#include <stdio.h>

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

int mul(int a, int b)
{
    return a * b;
}

int main()
{
    int result;

    result = add(10, 20);
    printf("%d\n", result);

    result = mul(10, 20);
    printf("%d\n", result);

    return 0;
}
```

Bazı Matematiksel Standart C Fonksiyonları

C'de temel bazı matematik işlemleri yapan çeşitli standart C fonksiyonları vardır. Bu fonksiyonları kullanırken <math.h> dosyası include edilmelidir.

- sqrt fonksiyonu double bir parametreye sahiptir, parametresi ile aldığı değerin karekökünü geri dönüş değeri olarak verir.

```
double sqrt(double val);
```

Örneğin:

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

int main()
{
    double val;
    double result;

    printf("Lutfen bir sayi giriniz:");
    scanf("%lf", &val);

    result = sqrt(val);
    printf("%f\n", result);

    return 0;
}
```

pow fonksiyonu üs almak için kullanılır. Parametrik yapısı şöyledir:

```
double pow(double base, double e);
```

Bu fonksiyon birinci parametresiyle belirtilen değerin ikinci parametresiyle belirtilen kuvvetini alır ve onu geri dönüş değeri olarak verir. Örneğin:

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

int main()
{
    double result;

    result = pow(3, 4);
    printf("%f\n", result);

    return 0;
}
```

log fonksiyonu e tabanına göre log10 fonksiyonu 10 tabanına göre logaritma alır:

```
double log(double val);
double log10(double val);
```

Örneğin:

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

int main()
{
    double result;
```

```

    result = log10(1000);
    printf("%f\n", result);

    return 0;
}

```

sin, cos, asin, acos, tan, atan fonksiyonları trigonometrik işlemleri yapar. Bu fonksiyonlardaki açılar radyan cinsinden girilmelidir. Örneğin:

```

#include <stdio.h>
#include <math.h>

int main()
{
    double result;

    result = sin(3.141592653589793238462643 / 6);
    printf("%f\n", result);

    result = cos(3.141592653589793238462643 / 3);
    printf("%f\n", result);

    result = sin(3.141592653589793238462643 / 4) / cos(3.141592653589793238462643 / 4);
    printf("%f\n", result);

    result = tan(3.141592653589793238462643 / 2);
    printf("%f\n", result);

    result = tan(3.141592653589793238462643 / 4);
    printf("%f\n", result);

    return 0;
}

```

exp fonksiyonu e üzeri işlemi yapmaktadır.

Sabitler (Literals)

Program içerisinde doğrudan yazılan sayılara sabit denir. C'de yalnızca nesnelere değil, aynı zamanda sabitlerin de türü vardır. Sabitin türü onun niceliğiyle ve sonuna getirilen eklerle ilgilidir. Kurallar şöyledir:

1) Sayı nokta içermiyorsa ve sayının sonunda hiçbir ek yoksa sabit int, long ve unsigned long türlerinin hangisinin sınırları içerisinde ilk kez kalıyorsa o türdür. Örneğin:

```

0 ---> int
100 ---> int

```

long türünün 8 byte int türünün 4 byte olduğunu varsayalım:

```

123 ---> int
3000000000 ---> long

```

Eğer sayı 16'lık ya da 8'lik sistemde belirtilmişse sabit int, unsigned int, long ve unsigned long türlerinin hangisinin sınırları içerisinde ilk kez kalıyorsa sabit o türdür.

2) Sayı nokta içermiyorsa ve sayının sonuna küçük harf ya da büyük harf L getirilmişse sabit long ve unsigned long türlerinin hangisinin sınırları içerisinde ilk kez giriyorsa o türdür. Örneğin:

100L ---> long
0L ---> long

3) Sayı nokta içermiyorsa ve sayının sonunda küçük harf ya da büyük harf U varsa sabit unsigned int ve unsigned long türlerinin hangisinin sınırları içerisinde ilk kez kalıyorsa o türdendir. Örneğin:

123U ---> unsigned int
1000u ---> unsigned int

4) Sayı nokta içermiyorsa ve sayının sonunda küçük harf ya da büyük harf UL ya da LU varsa (UL, Ul, uL, LU, Lu, ul, lu) sabit unsigned long türündendir.

100ul ---> unsigned long
1LU ---> unsigned long

5) Sayı nokta içeriyorsa ve sayının sonunda hiçbir ek yoksa sabit double türdendir. Örneğin:

123.65 ---> double
0.5 ---> double

Pek çok programlama dilinde olduğu gibi C'de de noktanın solunda ya da sağında birşey yoksa sıfır olduğu varsayılır. Örneğin:

.10 ---> double
10 ---> int
10. ---> double

6) Sayı nokta içeriyorsa ve sayının sonunda küçük harf ya da büyük harf F varsa sabit float türdendir. Örneğin:

12.34F ---> float
10F ---> geçersiz
10.f ---> float

7) Sayı nokta içeriyorsa ve sonunda küçük harf ya da büyük harf L varsa sabit long double türdendir. Örneğin:

100.23L ---> long double
10.1 ---> long double
.10L ---> long double
10L ---> long

8) Tek tırnak içerisine bir karakter yerleştirilirse bu bir sayı belirtir. O karakterin karakter tablosundaki sıra numarasını belirtir. (Örneğin tipik olarak ASCII.) Örneğin 'a' bir sabit belirtir. Yani aslında 'a' bir sayıdır. ASCII tablosu kullanılıyorsa 97 anlamındadır. Örneğin:

```
x = 'A' + 1;
```

Burada x'e 66 atanmaktadır. 66 da 'B' nin ASCII kodudur.

Anahtar Notlar: ASCII tablosunda önce büyük harfler, sonra küçük harfler gelir. Büyük harflerle küçük harfler arasında 6 farklı karakter vardır. Bu özellikle 'a' - 'A' farkının 32 olması için düşünülmüştür.

Anahtar Notlar: UNICODE tablonun ilk 128 elemanı standart ASCII tablosunun aynıdır. ikinci 128'lik elemanı ASCII Latin-1 code page'i ile aynıdır.

C'de tek tırnak içerisinde int türünün byte uzunluğu kadar karakter yazılabilir (Örneğin şu andaki yaygın sistemlerde int türü 4 byte uzunluğundadır. Bu nedenle tek tırnak içerisinde 4 karakter yazılabilir.) Tek tırnak içerisinde bir'den fazla karakter yazılırsa bunun hangi sayı anlamına geleceği derleyicileri yazanların isteğine bırakılmıştır. Yani C'de örneğin 'ab' ifadesi geçerlidir. Fakat bunun hangi sayıyı belirttiği derleyiciden derleyiciye değişebilir. Genellikle tek tırnak içerisinde tek bir karakter yerleştiririz.

C'de tek tırnak içerisinde yerleştirilen sabitler int türündendir (halbuki C++'ta char türündendir.)

Karakter tablolarındaki tüm karakterler görüntülenebilir değildir. Bu tür görüntülenemeyen karakterleri (non-printable characters) ekrana basmak istediğimizde ekranda birşey göremeyiz. Bazı olaylar gerçekleşir. Tek tırnak içerisinde önce bir ters bölü ve sonra özel bazı karakterler bazı görüntülenemeyen özel karakterleri belirtir. Bunların listesi şöyledir:

Karakter	Anlamı
'\a'	Alert (ekrana gönderildiğinde beep sesi çıkar)
'\b'	Back Space (ekrana gönderildiğinde sanki geri tuşuna basılmış gibi imleç bir geriye gider)
'\f'	Form Feed (bu karakter yazıcıya gönderildiğinde bir sayfa atar)
'\n'	New Line (ekrana gönderildiğinde imleç aşağı satırın başına geçer)
'\r'	Carriage Return (ekrana gönderildiğinde imleç aynı satırın başına geçer)
'\t'	Tab (ekrana gönderildiğinde imleç bir tab atar)
'\v'	Vertical Tab (ekrana gönderildiğinde düşey zıplama yapar)

Bu ters bölü karakterleri iki tırnak içerisinde tek bir karakter belirtirler. Örneğin:

```
printf("ali\tveli\nselami\tayse\n");
```

Ters bölü karakterinin kendisi '\\' ile belirtilir. '\' ifadesi geçersizdir. Örneğin:

```
#include <stdio.h>

int main()
{
    printf("c:\temp\a.dat\n");
    printf("c:\\temp\\a.dat\n");

    return 0;
}
```

Tek tırnak karakterinin karakter sabiti '"' biçiminde yazılamaz. '\" biçiminde belirtilir. Fakat çift tırnak içerisinde tek tırnak soruna yol açmaz. Örneğin:

```
#include <stdio.h>

int main()
{
    char ch = '\"';

    printf("%c\n", ch);
    printf("Turkiye'nin baskenti Ankara'dir\n"); /* geçerli */
    printf("Turkiye\'nin baskenti Ankara\'dir\n"); /* geçerli */
}
```

```
    return 0;
}
```

İki tırnak karakterine ilişkin karakter sabiti \" ile belirtilir. Fakat tek tırnak içerisinde iki tırnak soruna yol açmaz. Ancak iki tırnak içerisinde iki tırnak soruna yol açar. Örneğin:

```
#include <stdio.h>

int main()
{
    char ch1 = '\\';      /* geçerli */
    char ch2 = '\"';     /* geçerli */

    printf("%c, %c\n", ch1, ch2);
    printf("\\Ankara\\n"); /* geçerli */

    return 0;
}
```

Karakter sabitleri tek tırnak içerisinde önce bir ters bölü sonra oktal digit'ler belirtilerek de yazılabilir. Örneğin '\\7' karakteri '\\a' karakteri ile ASCII tablosunda aynı değer sahiptir. Örneğin:

```
#include <stdio.h>

int main()
{
    char ch = '\\11';

    printf("Ankara%cIzmir\n", ch);

    return 0;
}
```

Karakter sabitleri tek tırnak içerisinde önce bir ters bölü sonra küçük harf bir x sonra da hex digit'ler belirtilerek de yazılabilir. Örneğin '\\x1B' gibi. Örneğin:

```
#include <stdio.h>

int main()
{
    printf("Ankara\\x9Izmir\n");

    return 0;
}
```

İki tırnak içerisinde oktal ve hex sistemde karakterleri yazarken dikkat etmek gerekir. Çünkü derleyici anlamlı en uzun karakter kümesinden ters bölü karakterlerini oluşturur. Yani örneğin "6Ankara\\x91Adana" stringinde ters bölü karakteri '\\x9' biçiminde değil '\\x91A' biçiminde ele alınır. Fakat "6Ankara\\x9\\x31\\x41\\x64\\x61na\\n" istediğimizi yapabilir.

9) C'de short, unsigned short, char, signed char ve unsigned char türünden karakter sabitleri yoktur. Örneğin:

```
#include <stdio.h>

int main()
{
    printf("6Ankara\\x9\\x31\\x41\\x64\\x61na\\n");
}
```

```
    return 0;  
}
```

Anahtar Notlar: C'de tek tırnak içerisinde yazılan sabitlere karakter sabitleri denir fakat bunlar int türündendir. Yani örneğin C'de 'a' bir karakter sabitidir fakat int türündendir. Oysa C++'ta tek tırnak içerisindeki karakterler (eğer tek tırnak içerisinde tek karakter varsa) char türündendir.

Tamsayıların 10'luk, 8'lik ve 16'lık Sistemde Belirtilmesi

C'de bir sayı yazdığımızda onu default olarak 10'luk sistemde yazdığımız anlaşılır. eğer sayıyı 0x ya da 0X ile başlatarak yazarsak sayının 16'lık sistemde yazılmış olduğukabul edilir. Örneğin:

```
int a, b;  
  
a = 100;  
b = 0x64;
```

Eğer sayıyı başına 0 getirerek yazarsak 8'lik sistem anlaşılır. Örneğin:

```
a = 0144;
```

Sabitin kaçlık sistemde yazıldığının sabit türüyle bir ilgisi yoktur. Örneğin 0x64 sabiti yine int türden 0x64L sabiti long türündendir. Ayrıca C90'da float, double ve long double türleri 16'lık ve 8'lik sistemde belirtilemezler. Ancak C99'da 16'lık sistemde belirtilebilirler.

Gerçek Sayıların Üstel Biçimde Belirtilmesi

float, double ve long double türden sabitler üstel biçimde yazılabilirler. Üstel biçimin genel biçimi şöyledir:

<noktalı ya da noktasız sayı><e ya da E>[±]<üs>;

Örneğin 1e20, 1.2e-50, -2.3e+15 gibi...

Sayıyı üstel biçimde belirtmişsek sayı sonuna ek almamışsa double türündendir. Örneği 1E20 sabiti double türündendir. Ancak 1E20F float, 1E20L long double türündendir.

Üsetel biçim özellikle çok büyük ve çok küçük sayıları ifad etmek için kullanılır.

C90'da ve C99'da ikilik sistemde sayılar belirtilememektedir.

Operatörler

Bir işleme yol açan ve işlem sonucunda bir değer üretmesini sağlayan atomlara operatör denir. Operatörlerin teknik olarak tanımlamak için onları sınıflandıracacağız. Operatörler üç biçimde sınıflandırılabilir:

- 1) İşlevlerine Göre Sınıflandırma
- 2) Operand Sayılarına Göre
- 3) Operatörün Konumuna Göre

1) İşlevlerine Göre Sınıflandırma: Bu sınıflandırma operatörün hangi tür işlem yaptığına yönelik bir sınıflandırmadır. Tipik olarak operatör şu sınıflardan birine ilişkin olabilir:

- Aritmetik Operatörler (Arithmetic Operators): Bunlar +, -, * ve / gibi dört işleme yönelik operatörlerdir.

- Karşılaştırma Operatörleri (Comparison / Relational Operators): Bunlar karşılaştırma için kullanılan operatörlerdir. Örneğin >, <, >=, <= gibi.
- Mantıksal Operatörler (Logical Operators): Bunlar AND, OR, NOT gibi mantıksal işlem yapan operatörlerdir.
- Bit Operatörleri (Bitwise Operators): Bunlar sayıların karşılıklı bitleri üzerinde işlem yapan operatörlerdir.
- Adres Operatörleri (Pointer Operators): Bunlar adres işlemi yapan operatörlerdir.
- Özel Amaçlı Operatörler (Special Purpose Operators): Bunlar çeşitli konulara ilişkin özel işlem yapan operatörlerdir.

2) Operand Sayılarına Göre Sınıflandırma: Operatörün işleme soktuğu ifadeler operand denilmektedir. Örneğin $a + b$ ifadesinde $+$ operatördür, a ve b bunun operandlarıdır. Operatörler tek operand, iki operand ve üç operand alabilir. Tek operand alan operatörlere İngilizce “unary”, iki operand alan operatörlere “binary” ve üç operand alan operatörler “ternary” operatör denilmektedir. Örneğin $!$ tek operandlı (unary bir operatördür ve $!$ biçiminde kullanılır. / iki operandlı bir operatördür.

3) Operatörün Konumuna Göre Sınıflandırma: Operatör operandlarının önünde, sonunda ya da arasında bulunabilir. Operandlarının önüne getirilerek kullanılan operatörlere “önek (prefix) operatörler, arasına getirilerek kullanılan operatörlere “araek (infix) operatörler” ve operandlarının sonuna getirilerek kullanılan operatörlere “sonak (postfix)” operatörler denilmektedir.

Bir operatörü teknik olarak tanımlayabilmek için öncelikle yukarıdaki üç sınıflandırma biçiminde de nereye düştüğünün belirtilmesi gerekir. Örneğin “ $!$ operatörü tek operandlı önek (unary prefix) mantıksal operatördür” gibi. Ya da “ $/$ operatörü iki operandlı araek (binary infix) aritmetik operatördür” gibi...

Operatörler Arasındaki Öncelik İlişkisi

Bir ifadedeki operatörler belli bir sırada seri olarak yapılır. Örneğin:

$$a = b + c * d;$$

$$\hat{I}1: c * d$$

$$\hat{I}2: b + \hat{I}1$$

$$\hat{I}3: a = \hat{I}2$$

Derleyiciler ifadedeki işlemi yapacak makine komutlarını seri olarak üretirler. Run time sırasında işlemci onları sırasıyla çalıştırır.

Operatörler arasındaki öncelik ilişkisi “Operatörlerin Öncelik Tablosu” denilen bir tabloyla betimlenmektedir. Operatörlerin Öncelik Tablosunun basit bir biçimi aşağıdaki gibidir:

()	Soldan-Sağa
* /	Soldan-Sağa
+ -	Soldan-Sağa
=	Sağdan-Sola

Tablo satırlardan oluşur. Üst satırdaki operatörler alt satırdaki operatörlerden daha yüksek önceliklidir. Aynı satırdaki operatörler eşit önceliklidir. Aynı satırdaki operatörler “Soldan-Sağa” ya da “Sağdan-Sola” eşit öncelikli

olabilirler. “Soldan-Sağa” öncelik demek, ifade içerisinde “o satırda solda olan önce yapılır” demektir. “Sağdan-Sola” ise “ifade içerisinde o satırda sağda olan önce yapılır” demektir.

Tablonun en yüksek öncelik grubunda fonksiyon çağırma operatörü ve öncelik parantezi vardır. Örneğin:

```
result = foo() + bar() * 2;
```

```
İ1: foo()
İ2: bar()
İ3: İ2 * 2
İ4: İ1 + İ3
İ5: result = İ4
```

Parantezler öncelik kazandırmak için kullanılabilirler. Örneğin:

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

```
İ1: b + c
İ2: İ1 * d
İ3 = a = İ2
```

***, /, + ve - Operatörleri**

Bu operatörler iki operandlı arak (binary infix) aritmetik operatörlerdir. Temel dört işlemi yaparlar.

% Operatörü

İki operandlı arak aritmetik operatördür. Bu operatör soldaki operandın sağdaki operanda bölümünden elde edilen kalan değerini verir. Öncelik tablosunda * ve / ile aynı gruptadır.

()	Soldan-Sağa
* / %	Soldan-Sağa
+ -	Soldan-Sağa
=	Sağdan-Sola

Örneğin:

```
#include <stdio.h>

int main()
{
    int result;

    result = 10 % 4 - 1;
    printf("%d\n", result);

    return 0;
}
```

% operatörünün her iki operandının da tamsayı türlerine ilişkin olması zorunludur. Örneğin 15.2 % 2 ifadesi geçerli değildir. Negatif sayının pozitif sayıya bölümünden elde edilen kalan negatiftir. (Örneğin -10 % 4 ifadesi -2 sonucunu verir).

İşaret - ve İşaret + Operatörleri

+ ve - sembolleri hem toplama ve çıkarma operatörleri için hem de işaret - ve işaret + işlemleri için kullanılmaktadır. Aynı sembollere sahip olsalar da bu operatörlerin birbirleriyle hiçbir ilgisi yoktur. Örneğin:

$$a = -3 - 5;$$

Burada soldaki - işaret eksi operatörü, sağdaki ise çıkartma operatörüdür. İşaret + ve işaret - operatörleri tek operandlı örnek aritmetik operatörlerdir. Öncelik tablosunun ikinci düzeyinde Sağdan-Sola grupta bulunurlar.

()	Soldan-Sağa
+ -	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
=	Sağdan-Sola

Örneğin:

$$a = -3 - 5;$$

İ1: -3

İ2: İ1 - 5

İ3: a = İ2

Örneğin:

$$a = - - b + 2;$$

İ1: -b

İ2: -İ1

İ3: İ2 + 2

İ4: a = İ3

İşaret - operatörü operandının negatif değerini üretir. İşaret + operatörü ise operandıyla aynı değeri üretir. (Yani aslında birşey yapmaz.). Örneğin:

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

```
int main()
```

```
{
```

```
    int result;
```

```
    result = - - - - - - - - - - - - - - - -3 - 5;
```

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

```
    return 0;
```

```
}
```

Tanımsız, Belirsiz ve Derleyiciye Bağlı Davranışlar

C standartlarında bazı kodların tanımsız davranışa yol açacağı (undefined behavior) belirtilmiştir. Tanımsız davranışa yol açan kodlamalar derleme aşamasında bir soruna yol açmazlar. Ancak programın çalışma zamanı

sırasında programın çökmesine, yanlış çalışmasına yol açabilirler. Bazen tanımsız davranışa yol açan kodlar hiçbir soruna yol amayabilirler. Programcının böylesi farkında olması ve bunlardan uzak durması gerekir.

Yine standartlarda bazı durumlar için belirsiz davranışın (unspecified behavior) oluşacağı söylenmiştir. Belirsiz davranışa yol açan kodlarda birkaç seçenek vardır. Derleyici yazarlar bu seçeneklerden birini tercih etmiş durumdadırlar. Ancak hangi seçeneği tercih ettiklerini belirtmek zorunda değillerdir. Derleyici hangi seçeneği seçmiş olursa olsun bu kodun çökmesine yol açmaz. Ancak belirsiz davranışa yol açan kodlar taşınabilirliği bozarlar. Yani bu programlar başka sistemlerde derlendiğinde farklı sonuçlar elde edilebilir. En iyisi belirsiz davranış sonucunda farklı sonuçların oluşacağı tarzda kodlardan kaçınılmaktadır.

Standartlarda bazı durumlardaki belirlemeler derleyiciyi yazarların isteğine bırakılmıştır (implementation defined behavior). Fakat derleyici yazarlar bunu referans kitaplarında açıkça belirtmek zorundadır.

Örneğin tahsis edilmiş bir alan erişilmesi tanımsız davranışa yol açan bir işlemdir. C'de fonksiyon çağrıldığında parametre aktarımının soldan sağa mı ya da sağdan sola mı yapılacağı belirsiz davranışa sahiptir. Büyük tamsayı türünden işaretli küçük tamsayı türüne dönüştürme yapıldığında bilgi kaybının nasıl olacağı derleyiciyi yazarların isteğine bırakılmıştır.

++ ve -- Operatörleri

Bu operatörler tek operandlı önek ve sonek kullanılabilen operatörlerdir. Yani ++a ya da a++ biçiminde kullanılabilirler. ++ operatörüne artırma (increment), -- operatörüne eksiltme (decrement) operatörü denilmektedir. ++ operatörü “operand içerisindeki değeri 1 artır”, -- operatörü “operand içerisindeki değeri 1 eksilt” anlamına gelir. Örneğin:

```
#include <stdio.h>

int main()
{
    int a;

    a = 3;
    ++a;
    printf("%d\n", a);    /* 4 */

    a = 3;
    --a;
    printf("%d\n", a);    /* 2 */

    return 0;
}
```

++ ve -- operatörleri öncelik tablosunun ikinci düzeyinde sağdan sola grupta bulunurlar.

()	Soldan-Sağa
+ - ++ --	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
=	Sağdan-Sola

Bu operatörlerin önek ve sonek kullanımları arasında farklılık vardır. Her iki kullanımda da artırma ya da eksiltme tablodaki öncelikte yapılır. Önek kullanımda sonraki işleme nesnenin artırılmış ya da eksiltilmiş değeri sokulurken, sonek kullanımda artırılmamış ya da eksiltilmemiş değeri sokulur. Örneğin:

```

#include <stdio.h>

int main()
{
    int a, b;

    a = 3;
    b = ++a * 2;
    printf("a = %d, b = %d\n", a, b);    /* a = 4, b = 8 */

    a = 3;
    b = a++ * 2;
    printf("a = %d, b = %d\n", a, b);    /* a = 4, b = 6 */

    return 0;
}

```

Örneğin:

```

#include <stdio.h>

int main()
{
    int a, b;

    a = 3;
    b = ++a;
    printf("a = %d, b = %d\n", a, b);    /* a = 4, b = 4 */

    a = 3;
    b = a++;
    printf("a = %d, b = %d\n", a, b);    /* a = 4, b = 3 */

    return 0;
}

```

Örneğin:

```

#include <stdio.h>

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

    a = 3;
    b = 2;
    c = ++a * b--;
    printf("a = %d, b = %d, c = %d\n", a, b, c); /* a = 4, b = 1, c = 8 */

    return 0;
}

```

Şüphesiz bu operatörler tek başlarına başka bir operatör olmaksızın kullanılıyorsa, bunların önek ve sonek biçimleri arasında bir fark oluşmaz. Yani örneğin:

++a;

ile

a++;

arasında bir fark oluşmaz.

C'de bir nesne bir ifadede ++ ya da -- operatörüyle kullanılmışsa artık aynı ifadede bir daha o nesnenin görülmemesi gerekir. Aksi halde tanımsız davranış oluşur. Örneğin aşağıdaki gibi ifadeleri kullanmamalıyız. Burada ne olacağı belli değildir:

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

++ ve -- operatörlerinin operandlarının nesne belirtmesi gerekir. Örneğin:

```
++3;
```

ifadesi geçerli değildir. Ya da örneğin:

```
b = ++(a - 2);
```

ifadesi de geçersizdir. Çünkü a - 2 bir nesne belirtmemektedir.

Bu operatörlerden elde edilen ürün nesne belirtmez. Örneğin:

```
b = ++a--;
```

gibi bir ifade geçersizdir. Burada a--'den elde edilen ürün nesne belirtmez.

Karşılaştırma Operatörleri

C'de 6 karşılaştırma operatörü vardır:

```
<, >, <=, >=  
==, !=
```

Bu operatörlerin hepsi iki operandlı arak (binary infix) operatörlerdir. Öncelik tablosunda karşılaştırma operatörleri aritmetik operatörlerden daha düşük önceliklidir:

()	Soldan-Sağa
+ - ++ --	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
< > <= >=	Soldan-Sağa
== !=	Soldan-Sağa
=	Sağdan-Sola

Bu operatörler önerme doğruysa 1 değerini, yanlışsa 0 değerini üretirler. Bu operatörlerin ürettiği bu değerler int türündendir ve istenirse başka işlemlere sokulabilirler. Örneğin:

```
#include <stdio.h>

int main()
{
    int a = 3, b;

    b = (a > 2) + 1;
    printf("%d\n", b);    /* 2 */

    return 0;
}
```

Karşılaştırma operatörleri kendi aralarında iki öncelik grubunda bulunmaktadır. Örneğin:

```
result = a == b > c;
```

```
İ1: b > c
İ2: a == İ1
İ3: result = İ2
```

Örneğin:

```
#include <stdio.h>

int main()
{
    int a = 3, b;

    b = a++ == 3;
    printf("a = %d, b = %d\n", a, b);

    return 0;
}
```

Örneğin:

```
#include <stdio.h>

int main()
{
    int a = 20, b;

    b = 0 < a < 10;    /* dikkat bu ifade a'nın 0 ile 10 arasında olduğuna bakamaz */
    printf("a = %d, b = %d\n", a, b);

    return 0;
}
```

Mantıksal Operatörler

C'de üç mantıksal operatör vardır:

```
! (NOT)
&&(AND)
|| (OR)
```

! operatörü öncelik tablosunun ikinci düzeyinde Sağdan-Sola grupta bulunur. && ve || operatörleri karşılaştırma operatörlerinden daha düşük önceliklidir:

()	Soldan-Sağa
+ - ++ -- !	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
< > <= >=	Soldan-Sağa
== !=	Soldan-Sağa
&&	Soldan-Sağa
	Soldan-Sağa
=	Sağdan-Sola

Anahtar Notlar: C'nin tüm tek operandlı (unary) operatörleri öncelik tablosunun ikinci düzeyinde Sağdan-Sola grupta bulunmaktadır.

! operatörü tek operandlı önek, && ve || operatörleri iki operandlı arak operatörlerdir.

AND, OR ve NOT işlemleri operand olarak Doğru, Yanlış Değerlerini alır. Bu mantıksal işlemler aşağıdaki gibi etki gösterirler:

A	B	A AND B	A OR B
Doğru	Doğru	Doğru	Doğru
Doğru	Yanlış	Yanlış	Doğru
Yanlış	Doğru	Yanlış	Doğru
Yanlış	Yanlış	Yanlış	Yanlış

A	NOT A
Doğru	Yanlış
Yanlış	Doğru

C'de bool türü yoktur. Peki Doğru ve Yanlış'lar nasıl ifade edilmektedir?

İşte !, && ve || operatörleri önce operandlarını Doğru ya da Yanlış olarak yorumlarlar. Sıfır dışı değerler (tamsayı da gerçek sayı türünden) C'de Doğru olarak, sıfır değeri Yanlış olarak ele alınır. Bu operatörler yine Doğru sonuç için 1 değerini, yanlış sonuç için 0 değerini üretirler. Örneğin:

```
-2.3 && 10.2    => 1
0 || 10         => 1
10.2 && 0       => 0
0.2 && 0.2     => 1
```

Tabi genellikle mantıksal operatörler doğrudan sayısal değerlerle değil, karşılaştırma operatörlerinin çıktıklarıyla işleme sokulurlar. Örneğin:

```
result = a > 10 && a < 20;
```

Burada iki koşul da doğruysa && operatörü 1 değerini üretecektir. Aksi halde 0 değeri üretilenektir. Örneğin:

```
#include <stdio.h>

int main()
{
    int a;
    int result;

    printf("a:");
    scanf("%d", &a);

    result = a > 10 && a < 20;
    printf("%d\n", result);

    return 0;
}
```

Anahtar Notlar: Microsoft'un C derleyicileri belirli bir versiyondan sonra bazı standart C fonksiyonları için "Unsafe" uyarısı vermektedir. Üstelik de proje yaratılırken SDL Check disable edilmemişse bu uyarı error'e dönüştürülmektedir. Bu "Unsafe" uyarıları ne anlama gelir? Kısaca bu uyarılarda "bu fonksiyonlar esasen standart C fonksiyonlarıdır fakat tasarımlarında küçük birtakım kusurlar vardır". Evet her ne kadar Microsoft bunda haklıysa da yine de bu fonksiyon çağrılarını Uyarı olarak değerlendirmesi hoş değildir. Eğer programcı bu uyarı mesajlarından kurtulmak istiyorsa iki yol deniyebilir. Birincisi Proje ayarlarından C/C++/Preprocessor kısmına gelip "Preprocessor Definitions" listesine `_CRT_SECURE_NO_WARNINGS` makrosunu eklemektir. İkincisi de Proje ayarlarından C/C++/General tabına gelip Uyarı "düzeylerini (Warning Level)" 2'ye çekmektir.

Anahtar Notlar: Eğer proje yaratılırken yanlışlıkla SDL Check çarpısı kaldırılmamışsa bu işlem sonradan da yapılabilir. Bunun için proje ayarlarına gelinir. C/C++/General tabından "SDL Checks" disable edilir.

Anahtar Notlar: Linux ortamında C ile program geliştirmek için hiç IDE kullanılmayabilir. Program iyi bir Editör'de (Örneğin Kate) yazılıp, komut satırında aşağıdaki gibi derlenebilir:

```
gcc -o sample sample.c
```

Şöyle çalıştırılabilir:

```
./sample
```

IDE olarak Linux sistemlerinde şu seçenler dikkate değerdir:

- MonoDevelop (Mono-core ve MonoDevelop paketleri kurulmalı)
- QtCreator
- Eclipse (C/C++ için)
- Netbeans (C/C++ için)

MonoDevelop çalışma biçimi olarak Visual Studio'ya en fazla benzeyen IDE'dir ve kullanımı çok basittir.

Anahtar Notlar: UNIX/Linux sistemlerinde komut satırı çalışması hala en yaygın çalışma biçimidir. Çünkü bu sistemler ağırlıklı olarak server amaçlı kullanılmaktadır (Server kullanım oranı %60-70 civarı, Client kullanım oranı %1-%2 civarındadır.) Linux komutlarının belli bir düzeyde öğrenilmesi tavsiye edilir. Bunun için pek çok kitap vardır.

Kısa devre özelliği "eğer işlemler hiç bu özellik olmadan yapılsaydı" ile aynı sonucu oluşturur. Yani kısa devre özelliği aslında öncelik tablosunda belirtilen sırada işlemlerin yapılması durumunda elde edilecek olan değer daha hızlı elde edilmesini sağlar.

`&&` ve `||` operatörleri aynı ifade içerisinde bulunuyorsa her zaman sol taraftaki operatörün sol tarafı önce yapılır. Duruma göre diğer taraflar yapılmaktadır. Örneğin:

```
result = foo() || bar() && tar();
```

Burada önce foo yapılır. foo sıfır dışı ise başka hiçbir şey yapılmaz ve sonuç 1 olarak elde edilir. Eğer foo sıfır ise

bu durumda bar yapılır. bar da sıfır ise tar yapılmaz. Örneğin:

```
result = foo() && bar() || tar();
```

Burada önce foo yapılır. foo sıfır ise bar yapılmaz fakat tar yapılır. foo sıfır dışı ise bar yapılır. bar da sıfır dışı ise tar yapılmaz.

Atama Operatörü

Atama operatörü iki operandlı araek özel amaçlı bir operatördür. Bu operatör sağdaki ifadenin değerini soldaki nesneye yerleştirir. Atama operatörü de bir değer üretmektedir. Eğer atama operatörü ifadenin son operatörü değilse ondan elde edilen değeri diğer operatörlere sokabiliriz. Atama operatörü sol taraftaki nesneye atanmış olan değeri üretir. Atama operatörünün öncelik tablousunda Sağdan-Sola grupta bulunduğu dikkat ediniz. Örneğin:

```
a = b = 10;
```

```
İ1: b = 10 => 10
```

```
İ2: a = İ1 => 10
```

Örneğin:

```
a = (b = 10) + 20;
```

```
İ1: b = 10
```

```
İ2: İ1 + 20
```

```
İ3: a = İ2
```

Atama operatörünün sol tarafındaki operand nesne belirtmek zorudur (yani LValue olmak zorundadır). Örneğin:

```
a + b = c;    /* geçersiz! */
```

```
10 = 20;     /* geçersiz! */
```

Aşağıdaki gibi bir ifade de geçerlidir:

```
foo(a = 10);
```

Burada önce a = 10 ifadesinin değeri hesaplanır, sonra bu değer foo fonksiyonunun parametre değişkenine atanır.

İşlemlili Atama Operatörleri (Compound Assignment Operators)

C'de bir grup +=, -=, *=, /=, %=, gibi işlemlili atama operatörü vardır. a <op>= b ifadesi ile a = a <op> b ifadesi tamamen eşdeğerdir. Örneğin:

```
a += 2;      /* a = a + 2; */
```

```
a *= 3;      /* a = a * 3; */
```

İşlemlili atama operatörlerinin hepsi öncelik tablosunda atama operatörüyle Sağdan-Sola aynı öncelik grubunda bulunmaktadır:

()	Soldan-Sağa
+ - ++ -- !	Sağdan-Sola
* / %	Soldan-Sağa

+ -	Soldan-Sağa
< > <= >=	Soldan-Sağa
== !=	Soldan-Sağa
&&	Soldan-Sağa
	Soldan-Sağa
= += -= *= /= %=, ...	Sağdan-Sola

Örneğin:

a *= 2 + 3;

İ1: 2 + 3;

İ2: a *= İ1;

Virgül Operatörü

Virgül atomu aynı zamanda iki operandlı araeek bir operatör belirtir. Virgül operatörü iki farklı ifadeyi tek ifade gibi ifade etmek için kullanılır. Örneğin:

a = 10, b = 20 tek bir ifadedir. Çünkü virgül de bir operatördür. Virgül operatörü özel bir operatördür. Klasik öncelik tablosu kuralına uymaz. Ancak öncelik tablosunda tablonun sonuna yerleştirilmiştir:

()	Soldan-Sağa
+ - ++ -- !	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
< > <= >=	Soldan-Sağa
== !=	Soldan-Sağa
&&	Soldan-Sağa
	Soldan-Sağa
= += -= *= /= %=, ...	Sağdan-Sola
,	Soldan-Sağa

Virgül operatörünün sağında ne olursa olsun önce onun sol tarafındaki ifade tamamen yapıp bitirilir. Sonra sağındaki ifade tamamen yapıp bitirilir. Virgül operatörü de bir değer üretir. Virgül operatörünün ürettiği değer sağ tafındaki ifadenin değeridir. Virgülün solundaki ifadenin değer üretmekte bir etkisi yoktur. Örneğin:

a = (b = 10, 20 + 30);

Burada parantezler olmasaydı virgül operatörünün sol tarafındaki operand a = b = 10 ifadesi olacaktı. Parantezler virgül operatörünü diğer operatörlerden ayırtmaktadır. Burada önce parantez içi yapılır. Parantez içerisinde virgül operatörü vardır. Onun da önce sol tarafı sonra sağ tarafı yapılır. Virgül operatörünün hepsinden elde edilen değer sağ tarafındaki ifadenin değeridir. Yani yukarıdaki örnekte a'ya 50 atanacaktır.

Bir'den fazla virgül operatörü de kombine edilebilir. Örneğin:

x = (foo(), bar(), tar());

```
i1: foo(), bar()
i2: i1, tar()
i3: x = i2
```

Her virgülden de virgül operatörü sanmamak gerekir. Örneğin:

```
int a, b, c;
```

Buradaki virgüller ayıraç görevindedir. Örneğin:

```
foo(a, b);
```

Buradaki virgül de argüman ayırıcı görevindedir. Ancak ekstra bir parantez virgülden operatör durumuna sokar. Örneğin:

```
foo((a, b), c);
```

Burada fonksiyonun iki parametre değişkeni vardır. Birinci parametre değişkenine b, ikincisine c kopyalanır.

Noktalı Virgülün İşlevi

Noktalı virgül ifadeleri birbirlerinden ayırarak onların bağımsız bir biçimde ele alınmasını sağlar. Örneğin:

```
a = 10 + 20;
b = 30 + 40;
```

Burada $a = 10 + 20$ ile $b = 30 + 40$ 'ın bir ilişkisi yoktur. Bunlar iki ayrı ifade belirtirler. İfadeleri ayırmak için kullanılan bu tür atomlara sonlandırıcı (terminator) denilmektedir. Eğer noktalı virgül unutulursa derleyici önceki ifadeyle sonraki ifadeyi tek bir bağımsız ifade olarak ele alır o da sentaks bakımından soruna yol açar. Örneğin:

```
a = 10 + 20    /* noktalı virgül unutulmuş */
b = 30 + 40;
```

Burada derleyiciye göre tek bir ifade vardır o da anlamsızdır.

Bazı dillerde sonlandırıcı olarak : bazılarında da `\n` kullanılabilir. Örneğin BASIC'te aslında `\n` karakteri (ENTER tuşuna basılınca elde edilen karakter) sonlandırıcı görevindedir. Tabii ki her satıra tek bir ifade yazılmak zorundadır.

Deyimler (Statements)

Bir programdaki çalıştırma birimlerine deyim denir. Imperative dillerin teorisinde program deyimlerin çalışmasıyla çalışır. Deyim deyim içerebilir. Deyimler 5 gruba ayrılmaktadır:

1) Basit Deyimler (Simple Statements): Bir ifadenin sonuna ; getirilirse artık o atom grubu bir deyim olur. Bu tür deyimlere basit deyimler denir. Yani basit deyimler “ifade;” biçimindeki deyimlerdir. Örneğin:

```
x = 10;
foo();
```

Birer basit deyimdir. Görüldüğü gibi ifade kavramı noktalı virgülden içermemektedir. İfadenin sonuna noktalı virgül konulunca o deyim olur.

2) Bileşik Deyimler (Compound Statements): C'de bir bloğun içerisine sıfır tane ya da daha fazla deyim yerleştirilirse o bloğun tamamı da bir deyim olur. Bunlara bileşik deyim denilmektedir. Bileşik deyimi diğer deyimleri içeren deyimler gibi (yani büyük kutu gibi) düşünebiliriz. Örneğin:

```
ifade1;
{
    ifade2;
    {
        ifade3;
        ifade4;
    }
    ifade5;
}
ifade6;
```

Burada dışarıdan bakıldığında 3 deyim vardır: İki basit biri bileşik deyim. Bileşik deyim içinde 3 tane deyim bulunmaktadır.

3) Kontrol Deyimleri (Control Statements): if gibi, for gibi, while gibi akış üzerinde etkili olan özel deyimlere kontrol deyimleri denilmektedir. Zaten ilerleyen bölümde bu kontrol deyimleri tek tek ele alınıp incelenecektir.

4) Bildirim Deyimleri (Declaration Statements): Bildirim yapmakta kullanılan sentaks yapıları da aslında deyim belirlemektedir. Bunlara bildirim deyimleri denir. Örneğin:

```
int a, b;
int x;

a = 10;
b = 20;
```

Burada iki bildirim deyimini, iki de basit deyim vardır.

5) Boş Deyimler (Null Statements): Solunda ifade olmayan noktalı virgüllere boş deyim denilmektedir. Örneğin:

```
x = 10;;
```

Buradaki ilk noktalı virgül ifadeyi deyim yapan sonlandırıcıdır. Fakat ikinci noktalı virgölün solunda bir ifade yoktur. Boş deyimler “hiçbirşey;” gibi düşünülebilir. Boş deyim bir etkiye yol açmasa da yine de bir deyim olarak değerlendirilir. Yukarıdaki örnekte toplam iki deyim vardır.

Her deyim çalıştığında birşeyler olur:

1) Bir basit deyim çalıştırılması demek o deyim oluşturulan ifadenin yapılması demektir.

2) Bir bileşik deyim çalıştırılması demek onu oluşturan deyimlerin tek tek çalıştırılması demektir. Örneğin:

```
{
    ifade1;
    {
        ifade2;
        ifade3;
    }
}
```

```
}
 ifade4;
}
```

Burada bu bileşik deyim çalıştırıldığında önce ifade1; basit deyimi çalıştırılır. Sonra içerideki bileşik deyim çalıştırılır. Bu da ifade2; ve ifade3; basit deyimlerinin çalıştırılması anlamına gelir. Sonra da ifade4; basit deyimi çalıştırılır. Özetle bir bileşik deyim çalıştırıldığında onu oluşturan deyimler yukarıdan aşağıya doğru çalıştırılır. Aslında her fonksiyon ana bloğuyla bir bileşik deyim belirtir. Bir fonksiyon çağrıldığında o fonksiyonun ana bloğuna ilişkin bileşik deyim çalıştırılır. O halde her şey main fonksiyonun çağrılmasıyla başlar. Bir C programının çalıştırılması demek main fonksiyonunun çağrılması demektir. main fonksiyonu programı yükleyen işletim sistemi tarafından çağrılır.

3) Bir kontrol deyimi çalıştırıldığında nelerin olacağı izleyen bölümde ele alınmaktadır.

4) Bir bildirim deyimi çalıştırıldığında bildirilen değişkenler için yer ayrılır.

5) Boş deyim karşısında derleyiciler hiçbir şey yapmaz.

Kontrol Deyimleri

Programın akışı üzerinde etkili olan deyimler kontrol deyimleri denir. Burada C'deki kontrol deyimleri başlıklar halinde ele alınacaktır.

if Deyimi

if deyiminin genel biçimi şöyledir:

```
if (<ifade>)
    <deyim>
[
else
    <deyim>
]
```

if anahtar sözcüğünden sonra parantezler içerisinde bir ifadenin bulunması zorunludur. if deyimi doğruysa ve yanlışsa kısımlarından oluşur. Doğruysa ve yanlışsa kısımlarında tek bir deyim bulunmak zorundadır. Eğer programcı bu kısımlarda birden fazla deyim buldurmak istiyorsa onu bileşik deyim olarak ifade etmelidir (yani bloklamalıdır). if deyiminin yanlışsa kısmı bulunmak zorunda değildir. if deyiminin tamamı tek bir deyimdir.

if deyimi şöyle çalışır: Önce if parantezi içerisindeki ifadenin sayısal değeri hesaplanır. Bu değer sıfır dışı ise if deyiminin doğruysa kısmı, sıfır ise yanlışsa kısmı çalıştırılır. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int a;

    printf("Bir sayı giriniz:");
    scanf("%d", &a);

    if (a > 0)
        printf("pozitif\n");
    else
        printf("negatif ya da sıfır\n");
}
```

```

    printf("son\n");
    return 0;
}

```

Örneğin:

```

#include <stdio.h>
#include <math.h>

int main(void)
{
    double a, b, c;
    double delta;

    printf("a:");
    scanf("%lf", &a);

    printf("b:");
    scanf("%lf", &b);

    printf("c:");
    scanf("%lf", &c);

    delta = b * b - 4 * a * c;
    if (delta < 0)
        printf("Kok yok!\n");
    else {
        double x1, x2;

        x1 = (-b + sqrt(delta)) / (2 * a);
        x2 = (-b - sqrt(delta)) / (2 * a);

        printf("x1 = %f, x2 = %f\n", x1, x2);
    }

    return 0;
}

```

İf deyiminin doğruysa kısmı bulunmak zorundadır. Ancak yanlıssa kısmı bulunmak zorunda değildir. Örneğin:

```

if (ifade1)
    ifade2; ifade3;

```

Burada if deyiminin doğruysa kısmında yalnızca ifade; deyimi vardır. ifade3; deyimi if dışındadır. Örneğin:

```

if (ifade1) {
    ifade2;
    ifade3;
}
ifade4;

```

Burada ifade4; deyimi if dışındadır. Derleyici if deyiminin doğruysa kısmı bittiğinde else anahtar sözcüğünün olup olmadığına bakar. eğer else anahtar sözcüğü yoksa if deyimi bitmiştir. Örneğin:

```

#include <stdio.h>

int main(void)

```

```

{
    int a;

    printf("Bir sayi giriniz:");
    scanf("%d", &a);

    if (a > 0)
        printf("pozitif\n");
    printf("son\n");

    return 0;
}

```

if deyiminin yanlışlıkla boş deyim ile kapatılması sık rastalanan bir hatadır. Örneğin:

```

if (a > 0);          /* dikkat yanlışlıkla yerleştirilmiş boş deyim */
    printf("pozitif\n");

```

Boş deyim için bir şey yapılmıyor olsa da boş deyim yine bir deyimdir.

Yalnızca yanlışsa kısmı olan bir if olamaz. Fakat bunu aşağıdaki gibi yapay bir biçimde oluşturabiliriz:

```

if (ifade1)
    ;
else
    ifade2;

```

Tabi bunun yerine soruyu ters çevirmek daha iyi bir tekniktir:

```

if (!ifade1)
    ifade2;

```

Aşağıdaki if cümlesinde derleyici hatayı nasıl tespit edecektir?

```

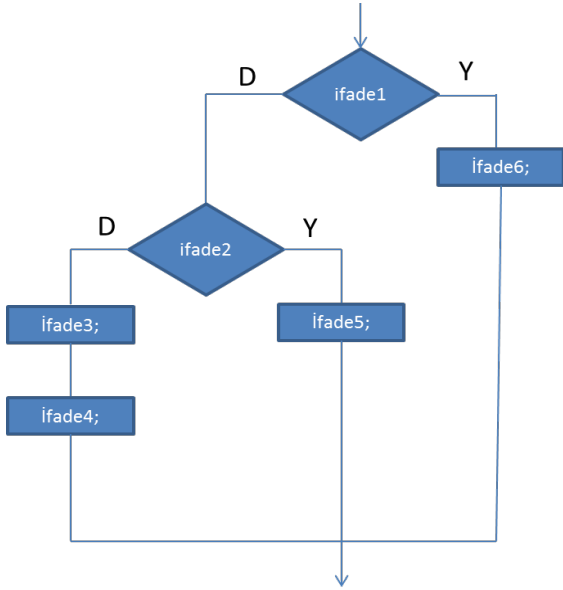
if (ifade1)
    ifade2;
    ifade3;
else
    ifade4;

```

Bu bize if deyiminin doğruysa kısmında birden fazla deyim olduğu halde bloklama yapılmamış hatası izlenimini vermektedir. Oysa derleyiciye göre ifade'den sonra if deyimi bitmiştir. Dolayısıyla if olmadan kullanılan bir else söz konusudur. Örneğin Microsoft derleyicileri bu durumda şu error mesajını vermektedir: "illegal else without matching if".

İç içe if Deyimleri (Nested if Statements)

Bir if deyiminin doğruysa ya da yanlışsa kısmında başka bir ifa deyimi bulunabilir. Örneğin:



Bu akış diagramının if cümlesi şöyle yazılabilir:

```

if (ifade1)
  if (ifade2) {
    ifade3;
    ifade4;
  }
  else
    ifade5;
else
  ifade6;

```

İki if için tek bir else varsa else hangi if'e ilişkindir. Örneğin:

```
if (ifade1) if (ifade2) ifade3; else ifade4;
```

Görünüşe bakılırsa burada iki durum da sanki mümkünmüş gibidir. else birinci if'in else kısmı da olabilir (yani ikinci if else'siz olabilir), ikinci if'in else'i de olabilir (yani birinci if else'sizdir). Bu duruma "dangling else" denilmektedir. Burada else içteki if'e ilişkindir. Deneyimli programcılar bile aşağıdaki gibi hatalı kod yazabilmektedir:

```

if (ifade1)
  if (ifade2)
    ifade3;
else
  ifade4;

```

Burada else'in gerçekten birinci if'e ilişkin olması isteniyorsa bilinçli bloklama yapılmalıdır:

```

if (ifade1) {
  if (ifade2)
    ifade3;
}
else
  ifade4;

```

Ayrık Koşullar

Bir grup koşuldan herhangi birisi doğruyken diğerlerinin doğru olma olasılığı yoksa bu koşullara ayrık koşullar

denilmektedir. Örneğin:

```
a > 0  
a < 0
```

koşulları ayrıktır. Örneğin:

```
a == 0  
a > 0  
a < 0
```

koşulları ayrıktır. Örneğin:

```
a == 1  
a == 2  
a == 3
```

koşulları ayrıktır. Fakat:

```
a > 0  
a > 10
```

koşulları ayrık değildir.

Ayrıık koşulların ayrıık if'lerle ifade edilmesi kötü bir tekniktir. Örneğin:

```
if (a > 0)  
    printf("pozitif\n");  
if (a < 0)  
    printf("negatif\n");
```

Burada $a > 0$ ise $a < 0$ olma olasılığı yoktur. Fakat gereksiz bir biçimde yine bu koşul yapılacaktır. Ayrıık koşulların else if'lerle ifade edilmesi gerekir. Örneğin:

```
if (a > 0)  
    printf("pozitif\n");  
else  
    if (a < 0)  
        printf("negatif\n");
```

Burada artık $a > 0$ ise gereksiz bir biçimde $a < 0$ işlemi yapılmayacaktır.

Bazen else-if durumu bir merdiven haline gelebilir. Örneğin:

```
if (a == 1)  
    printf("bir\n");  
else if (a == 2)  
    printf("iki\n");  
else if (a == 3)  
    printf("uc\n");  
else if (a == 4)  
    printf("dort\n");  
else  
    printf("hic biri\n");
```

else if merdivenlerinde olasılığı yüksek olanları yukarıya yerleştirmek daha iyi bir tekniktir.

Klavyeden Karakter Okumak

Klavyeden scanf fonksiyonuyla %c formatıyla karakter okuyabiliriz. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char ch;

    printf("bir karakter giriniz:");
    scanf("%c", &ch);

    printf("girilen karakter: %c\n", ch);

    return 0;
}
```

Fakat karakter okumak için getchar isimli standart bir C fonksiyonu da vardır:

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

Fonksiyon bir tuşa basılıp ENTER tuşuna basılana kadar bekler. Basılan tuşun karakter tablosundaki sıra numarasına geri döner. Geri dönüş değeri aslında bir byte'lık karakterin kod numarasıdır. Tipik olarak char türden bir nesneye atanabilir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char ch;

    ch = getchar();
    printf("girilen karakter: %c\n", ch);

    return 0;
}
```

stdin dosyasından okuma yapan fonksiyonlar tamponlu (buffered) bir çalışmaya sahiptir. Bu çalışma -bazı detayları olmakla birlikte- kabaca şöyledir: getchar fonksiyonuyla biz şeyler girdiğimizde girdiğimiz karakterler ve ilave olarak \n karakteri önce bir tampona aktarılır. Sonra oradan sırasıyla alınır. (En son '\n' alınacaktır.) Eğer tamponda karakter varsa getchar klavyeden birşey istemez. Tampondan alır. Ancak tamponda hiçbir karakter kalmamışsa getchar klavyeden değer ister. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char ch;

    ch = getchar();
    printf("girilen karakter: %c\n", ch);

    ch = getchar();
    printf("girilen karakter: %c\n", ch);

    return 0;
}
```

Burada birinci getchar'da biz k karakterine basıp ENTER tuşuna basmış olalım. Tamponda k\n karakterleri bulunacaktır. Birinci getchar k'yı alacak ikincisi hiç beklemeden '\n' karakterini alacaktır.

Sınıf Çalışması: Klavyeden getchar fonksiyonuyla bir karakter okuyunuz. Karakter 'a' ise ali, 'b' ise burhan, 'c' ise cemal 'd' ise demir, 'e' ise ercan ve bunların dışında bir karakterse hiçbiri yazısını bastırınız.

Çözüm:

```
#include <stdio.h>

int main(void)
{
    char ch;

    ch = getchar();

    if (ch == 'a')
        printf("ali\n");
    else if (ch == 'b')
        printf("burhan\n");
    else if (ch == 'c')
        printf("cemal\n");
    else if (ch == 'd')
        printf("demir\n");
    else if (ch == 'e')
        printf("ercan\n");
    else
        printf("hicbiri\n");

    return 0;
}
```

Anahtar Notlar: Standartlar asgariyi belirlemek için oluşturulmuştur. Standartlara uygun bir C derleyicisinin ekstra özellikleri olabilir. Bunlara eklenti (extension) denilmektedir. Eklentiler sentaksa ilişkin olabileceği gibi kütüphaneye ilişkin de olabilir. Eklenti özellikleri kullanırken dikkatli olmalıyız. Çünkü bunlar her derleyicide bulunmak zorunda olmayan özelliklerdir. Bu durumda kodumuzun taşınabilirliği (portability) zarar görür. Ancak bazı eklentiler çok yaygın bir biçimde desteklenmektedir. Hatta bu eklentileri bazı programcılar standart özellik dahi sanabilmektedir.

Klavyeden karakter okuyan diğer bir fonksiyon da getch fonksiyonudur. getch standart bir C fonksiyonu değildir. Pek çok C derleyicisinde bulunan bir eklenti fonksiyondur. Microsoft C derleyicilerinde ve gcc derleyicilerinde bu fonksiyon bulunmaktadır. Fonksiyonun parametrik yapısı şöyledir:

```
#include <conio.h>

int getch(void);
```

Bu fonksiyon ENTER tuşuna gereksinim duymaz ve tuşu basar basmaz alır. Basılan tuş görüntülenmez (biz istersek görüntüleriz). Örneğin:

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

int main(void)
{
    char ch;

    printf("bir karakter giriniz:");
    ch = getch();
```

```
    printf("\ngirilen karakter: %c\n", ch);
    return 0;
}
```

Bazen programcılar bu fonksiyonu bir tuşa basılana kadar akışı bekletmek için de kullanmaktadır. Örneğin:

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

int main(void)
{
    printf("Press any key to continue...\n");
    getch();
    printf("ok\n");

    return 0;
}
```

Diğer bir karakter okuyan eklenti fonksiyon getche fonksiyonudur. Bu fonksiyonun getch'tan farkı basıldığı zaman basılan tuşun da görüntülenmesidir.

putchar Fonksiyonu

putchar standart bir C fonksiyonudur. Bu fonksiyon parametresiyle aldığı sayıya karşılık gelen karakter tablosundaki karakter görüntüsünü ekrana yazdırır. Örneğin:

```
char ch = 'a';

putchar(ch);
```

Burada ekrana a karakteri çıkar. Yani:

```
putchar(ch);
```

ile:

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

aynı etkiyi oluşturur. Örneğin:

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

int main(void)
{
    char ch;

    while ((ch = getch()) != 'q')
        putchar(ch);

    return 0;
}
```

Örneğin:

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

```

int main(void)
{
    char ch;

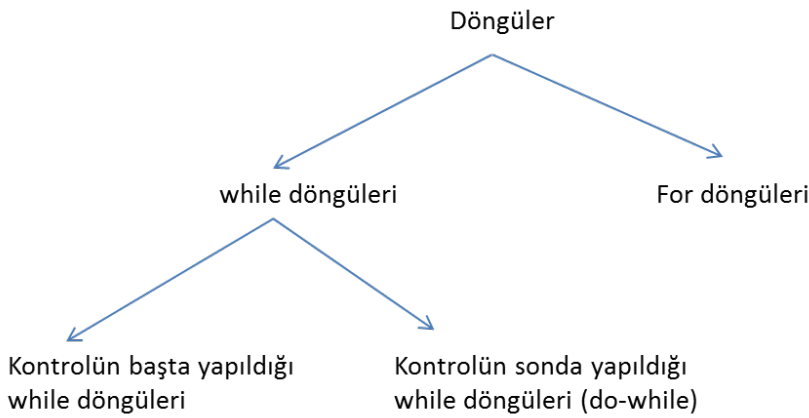
    while ((ch = getchar()) != '\n')
        putchar(ch);
    putchar('\n');

    return 0;
}

```

Döngü Deyimleri

Bir program parçasının yinelenmeli olarak çalıştırılmasını sağlayan kontrol deyimlerine döngü (loop) denir. C’de 2 döngü deyimi vardır: while döngüleri ve for döngüleri. while döngüleri de kendi aralarında “kontrolün başta yapıldığı while döngüleri” ve “kontrolün sonda yapıldığı while döngüleri (do-while döngüleri)” olmak üzere ikiye ayrılır.



Kontrolün başta yapıldığı while döngüleri

Bu biçimdeki while döngüleri pek çok programlama dilinde benzer biçimde bulunmaktadır. Genel biçimi şöyledir:

```

while (<ifade>)
    <deyim>

```

while anahtar sözcüğünden sonra parantezler içerisinde bir ifade bulunmak zorundadır. Döngü içerisinde tek bir deyim vardır. O yinelenen deyimdir.

while döngüsü şöyle çalışır: while parantezi içerisindeki ifadenin sayısal değeri hesaplanır. Bu değer sıfır dışı bir değerse doğru kabul edilir ve döngü deyimi çalıştırılıp başa dönlür. Bu ifade sıfır ise while deyimi sonlanır. while döngüleri parantez içerisindeki ifade sıfır dışı olduğu sürece yinelenen döngülerdir. Örneğin:

```

#include <stdio.h>

int main(void)
{
    int i = 0;

    while (i < 10) {
        printf("%d\n", i);
        ++i;
    }
}

```

```
    return 0;
}
```

Örneğin:

```
#include <stdio.h>

int main(void)
{
    int i = 10;

    while (i) {
        printf("%d\n", i);
        --i;
    }

    return 0;
}
```

Bir değeri atayıp, atanan değeri karşılaştırmak için parantezler kullanılmalıdır. Örneğin:

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

int main(void)
{
    char ch;

    while ((ch = getch()) != 'q')
        printf("%c", ch);

    return 0;
}
```

while parantezi içerisindeki ifadede virgül operatörü kullanılabilir. Örneğin:

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

int main(void)
{
    char ch;

    while (ch = getch(), ch != 'q')
        putchar(ch);
    putchar('\n');

    return 0;
}
```

Burada önce `ch = getch()` ifadesi yapılır, sonra `ch != 'q'` ifadesi yapılır. Test işlemine `ch != 'q'` ifadesi sokulur.

Anahtar Notlar: ENTER tuşuna basıldığında normal olarak “Carriage Return (CR)” karakterinin sayısal değeri elde edilir. CR karakterini ekrana yazdırdığımızda imleç bulunduğu satırın başına geçer. Fakat `getchar` fonksiyonunda karakter girdikten sonra basılan ENTER tuşu için `getchar` tampona CR değil “LF (Line Feed)” karakterini yerleştirmektedir. LF karakterini ekrana yazdırmak istediğimizde imleç aşağı satırın başına geçer. `getch` ve `getche` fonksiyonlarında ENTER tuşuna bastığımızda CR karakterini elde ediz. ENTER tuşu niyet olarak aşağı satırın başına geçmek amacıyla klavyeye yerleştirilmiş olsa da onun asıl kodu CR’dir. ENTER’a bastığımızda CR yerine LF karakterinin elde edilmesi girdi alan fonksiyonların yaptığı bir şeydir. CR karakteri (13 nolu ASCII karakter) C’de ‘r’ ile, LF karakteri de (10 numaralı ASCII karakteri) C’de ‘n’ ile temsil edilir.

while döngülerinin de yanlışlıkla boş deyim kapatılması durumuyla karşılaşmaktadır. Örneğin:

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

int main(void)
{
    char ch;

    while (ch = getch(), ch != 'q');
        putchar(ch);
        putchar('\n');

    return 0;
}
```

Burada artık döngünün içerisinde boş deyim vardır.

while parantezinin içerisinde ++ ve -- operatörlerinin kullanılması kafa karıştırabilmektedir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int i = 0;

    while (++i < 10)
        printf("%d ", i); /* 1'den 9'a kadar */
        putchar('\n');

    return 0;
}
```

while parantezinde önce bir ++ ya da -- varsa önce artırım ya da eksiltim uygulanır fakat sonraki operatöre nesnenin artırılmamış ya da eksiltilmemiş değeri sokulur. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int i = 0;

    while (i++ < 10)
        printf("%d ", i); /* 1'den 10'a kadar */
        putchar('\n');

    return 0;
}
```

Örneğin:

```
#include <stdio.h>

int main(void)
{
    int i = 10;

    while (--i)
```

```

        printf("%d ", i); /* 9'dan 1'e kadar */
        putchar('\n');

    return 0;
}

```

Örneğin:

```

#include <stdio.h>

int main(void)
{
    int i = 10;

    while (i--)
        printf("%d ", i); /* 9'dan 0'a kadar */
        putchar('\n');

    return 0;
}

```

Burada while parantezi içerisindeki ifadeyi while (i-- != 0) biçiminde değerlendirmek gerekir. Yani burada önce i bir eksiltilir fakat teste i'nin eksiltilmemiş değeri sokulur. Başka bir deyişle:

```

while (i--) {
    //...
}

```

ile,

```

while (i-- != 0) {
    //...
}

```

eşdeğer etkiye sahiptir.

while ile sonsuz döngü aşağıdaki gibi oluşturulabilir:

```

while (1) {
    /* ... */
}

```

Kontrolün Sonda Yapıldığı while Döngüleri (do-while Döngüsü)

Kontrolün sonda yapıldığı while döngüsünün genel biçimi şöyledir:

```

do
    <deyim>
while (ifade);

```

while parantezinin sonundaki noktalı virgül boş deyim belirtmez. Bu noktalı virgül sentaksın bir parçasıdır ve orada bulunmak zorundadır. Kontrolün sonda yapıldığı while döngüleri kontrolün başta yapıldığı while döngülerinin ters yüz edilmiş bir biçimdir. Ancak başına bir de do anahtar sözcüğü getirilmiştir. do-while döngülerinde döngü deyimi en az bir kez yapılmaktadır. Örneğin:

```

#include <stdio.h>

```

```

int main(void)
{
    int i;

    i = 0;
    do {
        printf("%d\n", i);    /* 0...9 */
        ++i;
    } while (i < 10);

    return 0;
}

```

Kontrolün başta yapıldığı while döngülerine kontrolün sonda yapıldığı while döngülerine göre çok fazla gereksinim duyulur. Fakat bazen kontrolün sonda yapıldığı while döngülerini kullanmak daha anlamlı olabilmektedir. Örneğin:

```

#include <stdio.h>

int main(void)
{
    char ch;

    ch = ' ';
    while (ch != 'e' && ch != 'h') {
        printf("(e)vet/(h)ayir?");
        ch = getch();
        printf("%c\n", ch);
    }

    if (ch == 'e')
        printf("evet secildi\n");
    else
        printf("hayir secildi\n");

    return 0;
}

```

Burada kontrolün sonda yapıldığı while döngüsü daha uygundur:

```

#include <stdio.h>
#include <conio.h>

int main(void)
{
    char ch;

    do
    {
        printf("(e)vet/(h)ayir?");
        ch = getch();
        printf("%c\n", ch);
    } while (ch != 'e' && ch != 'h');

    if (ch == 'e')
        printf("evet secildi\n");
    else
        printf("hayir secildi\n");

    return 0;
}

```


Karakter Test Fonksiyonları

C’de başı “is” ile başlayan isxxx biçiminde isimlendirilmiş bir grup standart fonksiyon vardır. Bunlara karakter test fonksiyonları denilmektedir. Bu fonksiyonların hepsi parametre olarak bir karakter alır ve int türden bir geri dönüş değeri verir. Bu fonksiyonlar parametreleriyle aldıkları karakterleri test ederler. Test doğrusa sıfır dışı bir değere yanlırsa sıfır değerine geri dönerler. Bu fonksiyonların listesi şöyledir:

- isupper (büyük harf mi?)
- islower (küçük harf mi?)
- isdigit (‘0’ ile ‘9’ arasındaki karakterlerden biri mi?)
- isxdigit (hex karakterlerden biri mi?)
- isalpha (alfabetik karakterlerden biri mi?)
- isalnum (alfabetik ya da nümerik (alfanümerik) karakterlerden biri mi?)
- isspace (boşluk karakterlerinden biri mi? (Space, tab, new line, carriage return, vertical tab))
- isascii (ASCII tablosunun ilk yarısındaki karakterlerden biri mi?)
- isctrl (İlk 32 kontrol karakterinden biri mi?)
- ispunct (noktalı virgül, nokta vs. gibi karakterlerden biri mi)

Bu fonksiyonları kullanırken <ctype.h> dosyası include edilmelidir.

Anahtar Notlar: Önce atama yapıp sonra atanan değer karşılaştırılmasını istiyorsak parantez kullanmalıyız. Örneğin:

```
if ((ch = getch()) != 'q') {  
    ...  
}
```

Örneğin:

```
#include <stdio.h>  
#include <conio.h>  
#include <ctype.h>  
  
int main(void)  
{  
    char ch;  
  
    while ((ch = getch()) != 'q') {  
        printf("%c\n", ch);  
        if (isupper(ch))  
            printf("buyuk harf\n");  
        else if (islower(ch))  
            printf("kucuk harf\n");  
        else if (isdigit(ch))  
            printf("digit karakter\n");  
        else if (isspace(ch))  
            printf("bosluk karakterlerinden biri\n");  
        else if (ispunct(ch))  
            printf("noktalama karakterlerinden biri\n");  
        else  
            printf("diger bir karakter\n");  
    }  
  
    return 0;  
}
```

Anahtar Notlar: UNIX/Linux sistemlerinde getch ve getche fonksiyonları yalnızca curses kütüphanesinde bulunmaktadır. Bu nedenle o kütüphanenin kurulması gerekir. Benzer biçimde gcc’nin Windows portunda da bu fonksiyonlar bulunmamaktadır.

Karakter test fonksiyonlarını biz de yazabiliriz. Örneğin isupper şöyle yazılabilir:

```

#include <stdio.h>

int myisupper(char ch)
{
    if (ch >= 'A' && ch <= 'Z')
        return 1;

    return 0;
}

int myislower(char ch)
{
    if (ch >= 'a' && ch <= 'z')
        return 1;

    return 0;
}

int myisalpha(char ch)
{
    if (myisupper(ch) || myislower(ch))
        return 1;

    return 0;
}

int main(void)
{
    char ch;

    printf("Karakter: ");
    ch = getchar();
    if (myisalpha(ch))
        printf("alfabetik\n");
    else
        printf("alfabetik degil\n");

    return 0;
}

```

Karakter test fonksiyonları ancak karakter 0-127 arasındaysa doğru çalışmaktadır. Bu fonksiyonlar ASCII karakterlerinin ilk 7 bitini dikkate almaktadır.

Büyük Harf Küçük Harf Dönüştürmesi Yapan Standart C Fonksiyonları

toupper fonksiyonu eğer parametresi ile belirtilen karakter küçük bir karakterse onun büyük karşılığıyla geri döner fakat küçük harf karakter değilse aynıysa i,le geri döner. tolower fonksiyonu da benzer biçimde parametresi ile aldığı karakter büyük harf ise onun küçük harf karşılığıyla değilse onun aynıysa ile geri dönmektedir. Bu fonksiyonları da kullanmadan önce <ctype.h> dosyasının include edilmesi gerekir. Örneğin:

```

#include <stdio.h>
#include <ctype.h>

int main(void)
{
    char ch;

    printf("Karakter giriniz:");
    ch = getchar();
    ch = toupper(ch);
    printf("%c\n", ch);
}

```

```
    return 0;
}
```

Anahtar Notlar: Karakter test fonksiyonları ve toupper ile tolower fonksiyonları C90 ekleriyle lokal spesifik özelliğe sahip olmuştur. Yani uygun lokal set edilmişse bunlar o dile göre de çalışabilir. Default lokal İngilizce olduğu için bunların da default davranışı standart İngilizce karakterler dikkate alınarak yapılır.

toupper ya da tolower kullanımına tipik bir örnek de şöyle olabilir:

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

int main(void)
{
    char ch;

    do
    {
        printf("(e)vet/(h)ayir?");
        ch = tolower(getch());
        printf("%c\n", ch);
    } while (ch != 'e' && ch != 'h');

    if (ch == 'e')
        printf("evet secildi\n");
    else
        printf("hayir secildi\n");

    return 0;
}
```

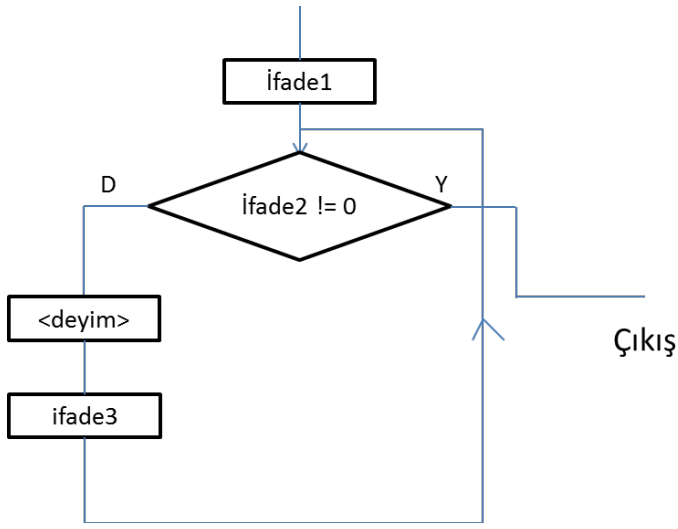
for Döngüleri

for döngüleri aslında while döngülerinin daha genel bir biçimidir. for döngülerinin genel biçimi şöyledir:

```
for ([ifade1];[ifade2];[ifade3])
    <deyim>
```

for anahtar sözcüğünden sonra parantezler içerisinde iki noktalı virgül bulunmak zorundadır. Bu for döngüsünü üç kısma ayırır. Bu üç kısımda da ifade tanımına uyan herhangi birer ifade kullanılabilir.

for döngüsü şöyle çalışır:



for döngüsünün birinci kısmındaki ifade döngüye girişte bir kez yapılır. Bir daha da yapılmaz. Döngü ikinci kısımdaki ifade sıfır dışı olduğu sürece yinelenir. İkinci kısımdaki ifade sıfır dışı ise önce döngü deyimi yapılır. Sonra üçüncü kısımdaki ifade yapılır ve yeniden kontrole girer.

for döngülerinin en çok kullanılan kalıbı şöyledir:

```
for (ilkdeğer; koşul; artırım)
  <deyim>
```

Örneğin:

```
#include <stdio.h>

int main(void)
{
  int i;

  for (i = 0; i < 10; ++i)
    printf("%d\n", i);

  return 0;
}
```

Örneğin:

```
#include <stdio.h>

int main(void)
{
  int i;

  for (i = 0; i < 10; ++i)
    printf("%d ", i);
  printf("\n");

  return 0;
}
```

Örneğin:

```
#include <stdio.h>

int main(void)
```

```

{
    int i;

    for (i = 0; i < 10; i += 2)
        printf("%d ", i);
    printf("\n");

    return 0;
}

```

Örneğin:

```

#include <stdio.h>
#include <math.h>

int main(void)
{
    double x;

    for (x = 0; x < 6.28; x += 0.1)
        printf("Sin(%.3f)=%.3f\n", x, sin(x));

    return 0;
}

```

Örneğin:

```

#include <stdio.h>

int main(void)
{
    int i;

    for (i = 10; i >= 0; --i)
        printf("%d ", i);
    printf("\n");

    return 0;
}

```

Şüphesiz üçüncü kısımda yapılan artırım ya da eksiltimlerin önek ya da sonek olması arasında bir fark yoktur.

1'den 100'e kadar sayıların toplamı şöyle bulunabilir:

```

#include <stdio.h>

int main(void)
{
    int i, total;

    total = 0;
    for (i = 1; i <= 100; ++i)
        total += i;

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

    return 0;
}

```

Bir döngünün döngü deyimi başka bir döngü olabilir. Bu duruma iç içe döngüler (nested loops) denilmektedir.

Örneğin:

```
#include <stdio.h>

int main(void)
{
    int i, k;

    for (i = 0; i < 3; ++i)
        for (k = 0; k < 3; ++k)
            printf("(%d, %d)\n", i, k);

    return 0;
}
```

for döngülerinin de yanlışlıkla boş deyim ile kapatılması durumuyla karşılaşilmektedir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int i;

    for (i = 0; i < 10; ++i);    /* Dikkat boş deyim */
        printf("%d\n", i);

    return 0;
}
```

for döngüsünün üç kısmına da ifade tanımına uyan her şey yerleştirilebilir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int i;

    i = 0;
    for (printf("birinci kisim\n"); i < 3; printf("ucuncu kisim\n")) {
        printf("dongu deyimi\n");
        ++i;
    }

    return 0;
}
```

for döngüsünün birinci kısmındaki ifade yazılmayabilir. Buradaki ifadeyi yukarı alsak da değişen birşey olmaz. Örneğin:

```
for (i = 0; i < 10; ++i) {
    ...
}
```

ile

```
i = 0;
for (; i < 10; ++i) {
    ...
}
```

tamamen işlevsel olarak eşdeğerdir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int i;

    i = 0;
    for (; i < 10; ++i)
        printf("%d ", i);
    printf("\n");

    return 0;
}
```

for döngüsünün üçüncü kısmı da yazılmayabilir. Aslında üçüncü kısımdaki ifade döngü değiminin sonuna yerleştirilirse değişen bir şey olmaz. Örneğin:

```
for (i = 0; i < 10; ++i) {
    ...
}
```

ile

```
i = 0;
for (; i < 10; ) {
    ...
    ++i;
}
```

işlevsel olarak eşdeğerdir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int i;

    i = 0;
    for (; i < 10;) {
        printf("%d ", i);
        ++i;
    }
    printf("\n");

    return 0;
}
```

Birinci ve üçüncü kısmı olmayan for döngüleri tamamen while döngüleriyle eşdeğerdir. Yani:

```
for (;ifade;) {
    ...
}
```

ile

```
while (ifade) {
    ...
}
```

tamamen eşdeğerdir.

Şüphesiz for döngüsü olmasaydı while döngüsünden for elde edebilirdik. Yani:

```
ifade1;
while (ifade2) {
    <deyim>
    ifade3;
}
```

aslında aşağıdaki for döngüsüyle eşdeğerdir:

```
for (ifade1; ifade2; ifade3)
    <deyim>
```

```
#include <stdio.h>

int main(void)
{
    int i;

    i = 0;
    while (i < 10) {
        printf("%d\n", i);
        ++i;
    }

    for (i = 0; i < 10; ++i)
        printf("%d\n", i);

    return 0;
}
```

for döngüsünün ikinci kısmı da boş bırakılabilir. Bu durumda döngü koşulunun her zaman sağlandığı kabul edilir. Yani döngünün ikinci kısmına birşey yazmamakla buraya sıfır dışı bir değer yazmak aynı anlamdadır. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int i;

    for (i = 0;; ++i) /* sonsuz döngü (infinite loop) */
        printf("%d\n", i);

    return 0;
}
```

for döngüsünün üç kısmında da virgül operatörü zenginlik katmak için kullanılabilir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int i, total;

    for (i = 1, total = 0; i <= 100; total += i, ++i)
        printf("%d\n", total);

    return 0;
}
```


Burada döngünün birinci kısmını $i = 1$, $total = 0$ ifadesi oluşturmaktadır. Üçüncü kısmını ise $total += i$, $++i$ ifadesi oluşturmaktadır.

Anahtar Notlar: Bir problemi kesin çözüme götüren adımlar topluluğuna algoritma (algorithm) denilmektedir. Algoritmaları hız bakımından ya da kaynak kullanımı bakımından karşılaştırabiliriz. Baskın ölçüt hızdır ve default olarak hız akla gelmektedir. Bazen bir problemin algoritması probleminin yapısı gereği çok uzun bilgisayar zamanı alabilmektedir. Bu durumda makul iyi bir çözümle yetinilebilir. Problemi kesin çözüme götürmeyen ancak iyi bir çözüm vaat eden adımlar topluluğuna sezgisel yöntem (heuristic) denilmektedir.

Önce C++'a sonra da C99 sokulmuş olan for döngülerinde önemli bir özellik vardır. C++'ta ve C99'da for döngülerinin birinci kısmında bildirim yapılabilir. Bu pratiklik sağlamaktadır. Örneğin:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 10; ++i)
        printf("%d\n", i);

    return 0;
}
```

Birden fazla değişkenin de aynı türden olmak koşuluyla bu biçimde bildirim yapılabilir. Örneğin:

```
for (int i = 0, k = 100; i + k >= 50; ++i, k -= 2) {
    ...
}
```

Bu biçimde bildirilen değişkenler ancak for döngüsünün içerisinde kullanılabilir. C++ ve C99'un standartlarına göre,

```
for (bildirim; ifade2; ifade3)
    <deyim>
```

ile,

```
{
    bildirim;
    for (;ifade2; ifade3)
        <deyim>
}
```

eşdeğerdir.

Biz kursumuzda C90 gördüğümüz için bu özelliği kullanmayacağız. Ancak bu özellik pek çok C90 derleyicisinde ekstra bir eklenti olarak bulunmaktadır.

break Deyimi

break deyiminin genel biçimi şöyledir:

```
break;
```

break deyimi yalnızca döngülerin ya da switch deyiminin içerisinde kullanılır. Programın akışı break deyimini gördüğünde döngü deyiminin kendisi sonlandırılır, akış sonraki deyimle devam eder. (Yani break adeta döngü dışına goto yapmak gibi etki gösterir.). Örneğin:

```

#include <stdio.h>

int main(void)
{
    int val;

    for (;;) {
        printf("Sayi giriniz:");
        scanf("%d", &val);
        if (!val)
            break;
        printf("%d\n", val * val);
    }

    return 0;
}

```

Asal sayı testi aşağıdaki gibi yapılabilir:

```

#include <stdio.h>

int isprime(int val)
{
    int i, result;

    result = 1;
    for (i = 2; i < val; ++i)
        if (val % i == 0) {
            result = 0;
            break;
        }

    return result;
}

int main(void)
{
    int i;

    for (i = 2; i < 1000; ++i)
        if (isprime(i))
            printf("%d ", i);
    printf("\n");

    return 0;
}

```

Yukarıdaki asallık testi aşağıdaki gibi iyileştirilebilir:

```

#include <stdio.h>
#include <math.h>

int isprime(int val)
{
    int i, result;
    double n;

    if (val % 2 == 0)
        return val == 2;

    n = sqrt(val);
    result = 1;
    for (i = 3; i <= n; i += 2)
        if (val % i == 0) {

```

```

        result = 0;
        break;
    }
}
return result;
}
int main(void)
{
    int i;

    for (i = 2; i < 1000; ++i)
        if (isprime(i))
            printf("%d ", i);
    printf("\n");

    return 0;
}

```

İç içe döngülerde break yalnızca kendi döngüsünden çıkar. Örneğin:

```

#include <stdio.h>
#include <conio.h>

int main(void)
{
    int i, k;

    for (i = 0; i < 10; ++i)
        for (k = 0; k < 10; ++k) {
            printf("(%d, %d)\n", i, k);
            if (getch() == 'q')
                break;
        }

    return 0;
}

```

Yukarıdaki örnekte 'q' tuşuna basıldığında her iki döngüden de çıkılması isteniyorsa dış döngü için de ayrıca break kullanılmalıdır. Örneğin:

```

#include <stdio.h>
#include <conio.h>

int main(void)
{
    int i, k;
    char ch;

    for (i = 0; i < 10; ++i) {
        for (k = 0; k < 10; ++k) {
            printf("(%d, %d)\n", i, k);
            if ((ch = getch()) == 'q')
                break;
        }
        if (ch == 'q')
            break;
    }

    return 0;
}

```

Sınıf Çalışması: Klavyeden int türden bir n değeri alınız ve aşağıdaki deseni çıkartınız:

```
*
**
***
****
*****
.....
*****.....***** (n tane)
```

Çözüm:

```
#include <stdio.h>

int main(void)
{
    int i, k, n;

    printf("Bir sayi giriniz:");
    scanf("%d", &n);

    for (i = 1; i <= n; ++i) {
        for (k = 1; k <= i; ++k)
            putchar('*');
        putchar('\n');
    }

    return 0;
}
```

Sınıf Çalışması: Klavyeden int türden bir sayı okuyunuz ve onun asal çarpanlarını yan yana yazdırınız. Örneğin:

```
Sayi giriniz: 28
2 2 7
```

İpucu: önce sürekli ikiye bölünerek ilerlenir. ikiye bölünmeyince bu sefer üçe bölünerek ilerlenir, bölünmediği zaman dörde bölünerek ilerlenir ve böyle devam ettirilir.

Çözüm:

```
#include <stdio.h>

int main(void)
{
    int n, i;

    printf("Bir sayi giriniz:");
    scanf("%d", &n);

    i = 2;
    while (n != 1) {
        if (n % i == 0) {
            printf("%d ", i);
            n /= i;
        }
        else
            ++i;
    }
    putchar('\n');

    return 0;
}
```

continue Deyimi

continue deyimi break deyimine göre daha seyrek kullanılır. Genel biçimi şöyledir:

```
continue;
```

Programın akışı continue anahtar sözcüğünü gördüğünde döngünün içindeki deyim sonlandırılır. Böylece yeni bir yenilemeye geçilmiş olur. break döngü deyiminin kendisini sonlandırırken, continue döngü içerisindeki yinelenen deyimi sonlandırır. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int i;

    for (i = 0; i < 10; ++i) {
        if (i % 2 == 0)
            continue;
        printf("%d\n", i);
    }

    return 0;
}
```

Burada ekrana yalnızca tek sayılar basılacaktır. continue deyimi yalnızca döngüler içerisinde kullanılabilir.

Sabit İfadeleri (Constant Expressions)

Yalnızca sabit ve operatörlerden oluşan ifadelere sabit ifadeleri (constant expressions) denilmektedir. Örneğin:

```
10
10 + 20
10 + 20 - 30 / 2
```

birer sabit ifadesidir. Aşağıdaki ifadeler sabit ifadesi değildir:

```
x
x + 10
10 - 20 - foo()
```

Sabit ifadelerinin net sayısal değerleri derleme aşamasında belirlenebilir. C'de kimi durumlarda sabit ifadeleri zorunludur. Örneğin global değişkenlere verilen ilkdeğerlerin sabit ifadesi olması gerekir. Örneğin:

```
int x = 10;           /* geçerli */
int y = x + 10;      /* geçersiz! */

void foo()
{
    int a = 10;
    int b = a + 10;   /* geçerli, b yerel */
    /* ... */
}
```

switch Deyimi

switch deyimi bir ifadenin çeşitli sayısal değerleri için çeşitli farklı işlemlerin yapılması amacıyla kullanılır. Genel

biçimi şöyledir:

```
switch (<ifade>) {  
    case <s.i>:  
        ....  
    case <s.i>:  
        ....  
    case <s.i>:  
        ....  
    [default:  
        ....  
    ]  
}
```

switch anahtar sözcüğünden sonra derleyici parantezler içerisinde bir ifade bekler. switch deyimi case bölümlerinden oluşur. case anahtar sözcüğünü bir sabit ifadesi ve sonra da ':' atomu izlemek zorundadır. switch deyiminin default bölümü olabilir. switch deyiminin tamamı tek bir deyimdir. switch deyimi şöyle çalışır: Önce switch parantezinin içerisindeki değer hesaplanır. Sonra bu değere eşit olan bir case bölümü var mı diye bakılır. Eğer böyle bir case bölümü varsa akış o case bölümüne aktarılır. eğer böyle case bölümü yoksa fakat switch deyiminin default bölümü varsa akış default bölüme aktarılır. default bölüm de yoksa akış switch deyimine girdiği gibi çıkar. Akış bir case bölümüne aktarıldığında artık diğer case etiketlerinin bir önemi kalmaz. Akış aşağıya doğru switch deyiminin sonuna kadar akar. (Buna "fall through" denilmektedir.). break deyimi döngülerin yanı sıra switch deyimini de kırmak için kullanılmaktadır. Genellikle her case bölümü break ile sonlandırılır. Böylece yalnızca o case bölümü yapılmış olur.

Örneğin:

```
#include <stdio.h>  
  
int main()  
{  
    int a;  
  
    printf("Bir sayi giriniz:");  
    scanf("%d", &a);  
  
    switch (a) {  
        case 1:  
            printf("bir\n");  
            break;  
        case 2:  
            printf("iki\n");  
            break;  
        case 3:  
            printf("uc\n");  
            break;  
        case 4:  
            printf("dort\n");  
            break;  
        default:  
            printf("hicbiri\n");  
            break;  
    }  
  
    printf("son\n");  
  
    return 0;  
}
```

case bölümlerinin sıralı olması ve default sonda olması zorunlu değildir. Bazen farklı case değerleri için aynı

işlemlerin yapılması istenebilir. Bunun tek yolu aşağıdaki gibidir. Daha pratik yolu yoktur:

```
case 1:  
case 2:  
    ifade;  
    break;
```

Örneğin:

```
#include <stdio.h>  
#include <conio.h>  
  
int main()  
{  
    char ch;  
  
    for (;;) {  
        printf("CSD>");  
        ch = getch();  
        printf("%c\n", ch);  
        if (ch == 'q')  
            break;  
        switch (ch) {  
            case 'r':  
            case 'd':  
                printf("delete command\n");  
                break;  
            case 'c':  
                printf("copy command\n");  
                break;  
            case 'm':  
                printf("move command\n");  
                break;  
            default:  
                printf("bad command!\n");  
                break;  
        }  
    }  
  
    return 0;  
}
```

Aynı değere ilişkin birden fazla case bölümü bulunamaz. Örneğin:

```
case 1 + 1:  
    ...  
    break;  
case 2:      /* geçersiz! */  
    ...  
    break;
```

Zaten case ifadelerinin sabit ifadesi olma zorunluluğu buradan gelmektedir. Eğer case ifadeleri sabit ifadesi olmasaydı bu durum derleme aşamasında denetlenemezdi.

case bölümlerine istenildiği kadar çok deyim yerleştirilebilir. İç içe switch deyimleri söz konusu olabilir.

case bölümlerinin hemen switch bloğunun içerisinde bulundurulması zorunlu değildir. Yani case bölümleri daha diplerde de bulunabilir.

```
switch (a) {  
    case 1:
```

```

    /*.... */
    if (ifade) {
        case 2:
            /* .... */
    }
    break;
}

```

switch parantezi içerisindeki ifade tamsayı türlerine ilişkin olmak zorundadır. case ifadeleri de gerçek sayı türünden sabit ifadesi olamaz. Tamsayı türlerine ilişkin olmak zorundadır.

Sınıf Çalışması: Klavyeden gün, ay ve yıl için üç değer isteyiniz. Ay bilgisi yazı ile olacak biçimde tarihi gün-ay-yıl biçiminde yazdırınız. Örneğin:

```

Gün:10
Ay: 12
Yıl: 1994

```

```
10-Aralik-1994
```

Çözüm:

```

#include <stdio.h>

int main()
{
    int day, month, year;

    printf("Gün:");
    scanf("%d", &day);

    printf("Ay:");
    scanf("%d", &month);

    printf("Yıl:");
    scanf("%d", &year);

    printf("%02d-", 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("Agustos");
            break;
    }
}

```



```

    case 9:
        printf("Eylul");
        break;
    case 10:
        printf("Ekim");
        break;
    case 11:
        printf("Kasim");
        break;
    case 12:
        printf("Aralik");
        break;
}
printf("-%02d\n", year);

return 0;
}

```

Sayıların printf Fonksiyonuyla Formatlanması

printf fonksiyonunda format karakterinden hemen önce bir sayı getirilirse, o sayı kadar alan ayrılır ve o alana sayı sağa dayılı olarak yazdırılır. Eğer o alana sayının sola dayalı olarak yazdırılması isteniyorsa sayının başına - karakteri getirilir. Örneğin:

```

#include <stdio.h>

int main()
{
    double f = 12.3;
    int a = 123;

    printf("%-10d%f\n", a, f);

    return 0;
}

```

Eğer format karakterinden önceki sayı, yazdırılacak sayının basamak sayısından küçükse bu durumda o sayının bir önemi kalmaz. Sayının hepsi yazdırılır.

Birtakım sayıların bıçakla kesilmiş gibi hizalanması için bu özellik kullanılmaktadır:

```

#include <stdio.h>

int main()
{
    int i;

    for (i = 1; i <= 100; ++i)
        printf("%-10d%d\n", i, i * i);

    return 0;
}

```

eğer format karakterinden önce + kullanılıyorsa (örneğin "%+10d" gibi, "%+d" gibi) sayının işareti her zaman yazdırılır. Örneğin:

```

#include <stdio.h>

int main()
{
    int i;

```

```

i = -123;
printf("%+d\n", i);    /* -123 */
i = 123;
printf("%+d\n", i);    /* +123 */

return 0;
}

```

Format karakterinden önceki sayının başına 0 getirilirse (örneğin "%010d" gibi) geri kalan boş alan sıfırla doldurulur. Örneğin:

```

day = 7;
month = 6;
year = 2009;

printf("%02d/%02d/%04d", day, month, year); /* 07/06/2009 */

```

float ve double türleri default olarak noktadan sonra 6 basamak yazdırılır:

```

#include <stdio.h>

int main()
{
    double d;

    d = 3.6;
    printf("%f\n", d);    /* 3.600000 */
    d = 3;
    printf("%f\n", d);    /* 3.000000 */

    return 0;
}

```

printf fonksiyonunda "%n.kf" "toplam n tane alan ayır, noktadan sonra k basamak yazdır" anlamına gelir. Eğer sayının tam kısmı için yeterli alan belirtilmemişse sayının hepsi yazdırılır. Şüphesiz noktadan sonraki k basamak için yuvarlama yapılmaktadır. Eğer sayının tam kısmı ile ilgilenilmiyorsa "%.kf" biçiminde format belirtilebilir. Örneğin:

```

printf("%.10f\n", f);    /* sayının tam kısmının hepsini yazdır, fakat noktadan sonra 10 basamak yazdır */

```

Örneğin:

```

#include <stdio.h>

int main()
{
    double d;

    d = 12345.6789;
    printf("%10.2f\n", d); /* 12345.68 */
    printf("%4.2f\n", d); /*12345.68 */

    d = 12.78;
    printf("%.0f\n", d);

    return 0;
}

```

goto Deyimi

goto deyimi programın akışını belli bir noktaya koşulsuz olarak aktarmak amacıyla kullanılır. Genel biçimi şöyledi:

```
goto <etiket>;
....
<etiket>:
```

programın akışı goto anahtar sözcüğünü gördüğünde akış koşulsuz olarak etiket ile belirtilen noktaya aktarılır. Etiket (label) isimlendirme kuralına uygun herhangi bir isim olabilir. Bazı programcılar okunabilirlik için goto etiketlerini büyük harflerle harflendirirler.

Eskiden ilk yüksek seviyeli diller makina dillerinin etkisi altındaydı. Bu dillerde neredeyse goto kullanmak zorunluydu. Sonraları goto'lar kodun takip edilebilirliğini bozması nedeniyle dillerden dışlanmaya başladılar. Pek çok deyim goto'suz blokları bir tasarıma sahip oldu. Bugün goto deyiminin kullanımı birkaç durum dışında tavsiye edilmemektedir. Fakat tipik olarak goto kullanılmasının anlamlığı olduğu birkaç durum vardır.

Aşağıdaki örnekte goto ile bir döngü oluşturulmuştur. Kesinlikle goto'lar döngü oluşturmak amacıyla kullanılmamalıdır. Bu nedenle aşağıdaki örnek goto kullanımının anlamlı olduğu bir durum için değil, onun çalışma mekanizmasını açıklamak için verilmiştir:

```
#include <stdio.h>

int main()
{
    int i = 0;
REPEAT:
    printf("%d\n", i);
    ++i;
    if (i < 10)
        goto REPEAT;

    return 0;
}
```

goto etiketi yalnızca goto işlemi sırasında etkili olur. Onun dışında bu etiketin bir işlevi yoktur. goto etiketinin bulundurulması ona goto yapılmasını zorunlu hale getirmez. (Ancak kendisine goto yapılmamış etiketler için derleyiciler uyarı verebilmektedir.)

goto etiketlerini bir deyim izlemek zorundadır. Örneğin:

```
void foo(void)
{
    ....
XX:      /* geçersiz! */
}
```

Fakat:

```
void foo(void)
{
    ....
XX:      /* geçerli */
    ;
}
```

goto etiketleri fonksiyon faaliyet alanına sahiptir. Yani aynı fonksiyon içerisinde aynı isimli tek bir goto etiketi bulunabilir.

goto deyimi ile başka bir fonksiyona atlama yapılamaz. Aynı fonksiyon içerisinde başka bir bölgeye atlama yapılabilir.

goto ile iç bir bloğa atlama yapılırken dikkat edilmelidir. Çünkü o bloktaki değişkenler için ilkdeğerleme işlemleri yapılmamış olabilir. Örneğin:

```
if (ifade)
    goto XX;
{
    int i = 10;

XX:
    ....
}
```

Burada atlanan noktada i çöp değere sahip olabilir. Başka bir dyleişle C'de iç bloğa goto yapıldığında o iç bloğun başında bildirilmiş değişkenler çöp değerlerdedir. Örneğin aşağıdaki programı çalıştırdığımızda ekrana çöp değer basılacaktır:

```
#include <stdio.h>

int main()
{
    goto TEST;

    {
        int i = 10;
    TEST:
        printf("%d\n", i);
    }

    return 0;
}
```

Şüphesiz aynı etikete fonksiyonun farklı yerlerinden birden fazla kez goto yapılabilir.

goto deyiminin aşağıdaki üç durumda kullanılması tavsiye edilmektedir:

1) İç içe döngülerden ya da döngü içerisindeki switch deyiminden tek hamlede çıkmak için. Örneğin:

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

int main(void)
{
    int i, k;

    for (i = 0; i < 10; ++i) {
        for (k = 0; k < 10; ++k) {
            printf("%d, %d\n", i, k);
            if (getch() == 'q')
                goto EXIT;
        }
    }
EXIT:
    return 0;
}
```

Örneğin:

```

#include <stdio.h>
#include <conio.h>

int main()
{
    char ch;

    for (;;) {
        printf("CSD>");
        ch = getch();
        printf("%c\n", ch);
        switch (ch) {
            case 'r':
            case 'd':
                printf("delete command\n");
                break;
            case 'c':
                printf("copy command\n");
                break;
            case 'm':
                printf("move command\n");
                break;
            case 'q':
                goto EXIT;
            default:
                printf("bad command!\n");
                break;
        }
    }
EXIT:
    return 0;
}

```

2) goto ters sırada kaynak boşaltımı amacıyla kullanılabilir:

3) Bazı özel durumlarda goto kullanılmazsa algortima çok çetrefil hale gelebilir. Yani goto bazı durumlarda kodu kısaltmakta ve okunabilirliği artırmaktadır.

Rastgele (Rassal) Sayı Üretimi

Bilgisayarda rassal sayılar aritmetik yöntemlerle üretilirler. Bu biçimde üretilmiş rassal sayılara sahte rassal sayılar (pseudo random numbers) denilmektedir. Sahte rassal sayı üretmek için pek çok yöntem bulunuyor olsa da bunların hemen hepsi bir başlangıç sayısı alınıp, onun üzerinde belirle işlem yapıp yeni sayı elde edilmesi ve aynı işlemlerin bu yeni sayı üzerinde devam ettirilmesi biçiminde elde edilmektedir. Örneğin bir sayıdan başlayıp onun karesini alıp ortadaki n basamağını alırsa bu n basamak rassaldır. Örneğin ilk sayı 123 olsun

123
512
621
856
....

Bu biçimde elde edilen sayılar rassaldır. Tabi burada ilk değer aynı ise her defasında aynı dizilim bulunur. Programın ger çalışmasında farklı bir dizilimin elde edilmesi için bu ilkdeğerin her çalışmada farklı alınması gerekir.

C'de rassal sayı üretimi için iki standart fonksiyon vardır: srand ve rand. Fonksiyonların parametrik yapıları şöyledir:

```
#include <stdlib.h>
```

```
int rand(void);  
void srand(unsigned int seed);
```

rand fonksiyonu her çağrıldığında 0 ile RAND_MAX değeri arasında rastgele bir sayıyla geri dönmektedir. RAND_MAX değerinin kaç olacağı derleyicileri yazanların isteğine bırakılmıştır. Bu değer her derleyicide farklı olabilir (pek çok derleyicide 32767 ya da 2147483647'dir.) Elde böyle bir fonksiyon varsa herhangi bir aralıkta rassal sayı elde edilebilir. Örneğin:

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main()  
{  
    int i, val;  
  
    for (i = 0; i < 20; ++i) {  
        val = rand() % 10;  
        printf("%d ", val);  
    }  
    printf("\n");  
  
    return 0;  
}
```

Bu program her çalıştırıldığında aynı dizilimi bize verir. Aynı dizilim bize verilmesi sayıların rassal olmadığı anlamına gelmez. Tabi genellikle program her çalıştığında farklı bir dizilimin verilmesi istenir.

srand fonksiyonu rassal sayı üreticisinin kullandığı ilkdeğeri değiştirmek amacıyla kullanılır. Bu değere terminolojide tohum değer (seed value) denilmektedir. Programın her çalışmasında farklı bir dizilimin elde edilmesi için programın her çalışmasında srand'ın farklı bir değerle çağrılması gerekir. İşte bu bilgisayarın içerisindeki saatle sağlanır. İleride ele alınacak olan time fonksiyonu bize bilgisayarın saatine bakarak bize 01/01/1970'ten fonksiyonun çağrıldığı zamana kadar kaç saniye geçtiğini verir. Bu fonksiyon için <time.h> dosyasının include edilmesi gerekir. Fonksiyon sıfır parametresiyle çağrılmalıdır. Örneğin:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
  
int main()  
{  
    int i, val;  
  
    srand(time(0));  
    for (i = 0; i < 10; ++i) {  
        val = rand() % 10;  
        printf("%d ", val);  
    }  
    printf("\n");  
  
    return 0;  
}
```

srand(time(0)) çağrısı programın başında yalnızca bir kez yapılmaktadır. Bazen programcılar yanlışlıkla bu çağrıyı da döngüsünün içerisine yerleştirirler. Bu durum çoğu kez aynı hep aynı değer elde edilmesine yol açar. srand fonksiyonu hiç çağrılmazsa tohum değer hep aynı sayıdan başlar.

Olasılığın görelilik tanımı (büyük sayılar yasası): Yazı tura işlemindeki limit

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    double head, tail, headRatio, tailRatio;
    unsigned long i;
    unsigned long n;

    srand(time(0));
    head = tail = 0;
    n = 1000000000;
    for (i = 0; i < n; ++i)
        if (rand() % 2 == 0)
            ++head;
        else
            ++tail;

    headRatio = head / n;
    tailRatio = tail / n;

    printf("Head Ratio = %f, Tail Ratio = %f\n", headRatio, tailRatio);

    return 0;
}

```

Zarda 6 gelme olasılığı nedir?

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    double six, sixRatio;
    unsigned long i;
    unsigned long n;

    srand(time(0));
    six = 0;
    n = 1000000000;
    for (i = 0; i < n; ++i)
        if (rand() % 6 + 1 == 6)
            ++six;

    sixRatio = six / n;

    printf("Head Ratio = %f\n", sixRatio);

    return 0;
}

```

Sınıf Çalışması: Bir döngü içerisinde getch fonksiyonuyla karakter bekleyiniz. Her tuşa basıldığında "Ali", "Veli", "Selami", "Ayşe", "Fatma isimlerinden birini rastgele yazdırınız. 'q' tuşuna basıldığında program sonlansın.

Çözüm:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>

```

```

int main()
{
    srand(time(0));

    while (getch() != 'q') {
        switch (rand() % 5) {
            case 0:
                printf("Ali\n");
                break;
            case 1:
                printf("Veli\n");
                break;
            case 2:
                printf("Selami\n");
                break;
            case 3:
                printf("Ayse\n");
                break;
            case 4:
                printf("Fatma\n");
                break;
        }
    }

    return 0;
}

```

Sınıf Çalışması: Her tuşa basıldığında 16 büyük harf karakterden oluşan bir dizilimi ekrana bastırınız (Yalnızca İngilizce karakterler kullanılacaktır.). 'q' tuşuna basıldığında program sonlansın.

```

GDSUSLYERVFXWTER
KREYDSBSKAIEIRIT
...

```

Çözüm:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>

int main()
{
    int i;

    srand(time(0));

    while (getch() != 'q') {
        for (i = 0; i < 16; ++i)
            putchar('A' + rand() % 26);
        putchar('\n');
    }

    return 0;
}

```

Farklı Türlerin Birbirlerine Atanması

C'de tüm aritmetik türler birbirlerine atanabilir. Ancak atama işleminde bilgi kaybı söz konusu olabilir. Eğer bilgi kaybı söz konusu oluyorsa, bilginin nerisinin kaybedildiği standartlarda belirtilmiştir.

Bir atamada kaynak ve hedef tür vardır. Örneğin:

```
a = b;
```

Burada kaynak tür b'nin türüdür, hedef tür a'nın türüdür. Standartlarda farklı türlerin birbirlerine atanması bir tür dönüştürme faaliyeti gibi ele alınmıştır. Standartlara göre atama işleminde kaynak türle hedef tür birbirlerinden farklıysa önce kaynak tür hedef türe dönüştürülür, sonra atama yapılır. Yani farklı türlerin birbirlerine atanması aslında farklı türlerin birbirlerine dönüştürülmesiyle aynı anlamdadır.

Aşağıda farklı türlerin birbirlerine atanması sırasında ne olacağı tek tek açıklanmıştır (else-if gibi değerlendiriniz)

1) Eğer kaynak türün içerisindeki değer hedef türün sınırları içerisinde kalıyorsa bilgi kaybı söz konusu olmaz. Örneğin:

```
long a = 10;  
short b;
```

```
b = a;
```

2) Eğer kaynak tür bir tamsayı türünden (signed char, unsigned char, signed short, unsigned short, signed int, unsigned int, signed long, unsigned long) hedef tür de işaretli bir tamsayı türündense sayının yüksek anlamlı byte'ları atılır, düşük anlamlı byte'ları atanır. Örneğin:

```
#include <stdio.h>  
  
int main()  
{  
    long a = 12345678;    /* 0xBC614E */  
    unsigned short b;  
  
    b = a;  
    printf("%u\n", b);    /* 24910 = 0x614E */  
  
    return 0;  
}
```

3) Eğer kaynak tür bir tamsayı türünden (signed char, unsigned char, signed short, unsigned short, signed int, unsigned int, signed long, unsigned long) hedef tür de işaretli bir tamsayı türündense bilgi kaybının nasıl olacağını (yani nerenin atılacağı) derleyiciyi yazanların isteğine bırakılmıştır (implementation dependent). Ancak derleyicilerin hemen hepsi bu durumda yine sayının yüksek anlamlı byte değerlerini atar. Örneğin:

```
#include <stdio.h>  
  
int main()  
{  
    long a = 7654321;    /* 0x74CBB1 */  
    short b;  
  
    b = a;  
    printf("%d\n", b);    /* 0xCBB1 = -13391 */  
  
    return 0;  
}
```

4) Eğer kaynak ile hedef tür aynı tamsayı türünün işaretli ve işaretli olmayan biçimlerinden oluşuyorsa sayının bit kalıbı değişmez, yalnızca işaret bitinin anlamı değişir. Örneğin:

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

```

int main()
{
    int a = -1;
    unsigned int b;

    b = a;
    printf("%u\n", b); /* 4294967295 */

    return 0;
}

```

5) Eğer kaynak tür küçük işaretli bir tamsayı türü hedef tür de büyük işaretli bir tamsayı türü ise bu durumda dönüştürme iki aşamada yürütülür. Önce küçük işaretli tür, büyük türün işaretli biçimine dönüştürülür, sonra büyük türün işaretli biçiminden büyük türün işaretli biçimine dönüştürme yapılır. Örneğin:

```

#include <stdio.h>

int main()
{
    signed char a = -1;
    unsigned int b;

    b = a;
    printf("%u\n", b); /* 4294967295 */

    return 0;
}

```

6) Eğer kaynak tür bir gerçek sayı türündense (float, double, long double) hedef tür bir tamsayı türündense sayının noktadan sonraki kısmı atılır, tam kısmı elde edilir. Eğer sayının noktadan sonraki kısmı atıldıktan sonra elde edilen tam kısım hala hedef türün sınırları içerisinde kalmıyorsa tanımsız davranış (undefined behavior) oluşur. Örneğin:

```

#include <stdio.h>

int main()
{
    double a;
    int b;

    a = 3.99;
    b = a;
    printf("%d\n", b); /* 3 */

    a = -3.99;
    b = a;
    printf("%d\n", b); /* -3 */

    return 0;
}

```

7) Eğer kaynak tür bir tamsayı türü hedef tür gerçek sayı türüyse ve sayı tam olarak tutulamıyorsa, kaynak tür ile belirtilen değere en yakın büyük ya da en yakın küçük değer elde edilir. Örneğin:

```

#include <stdio.h>

int main()
{
    long a;
    float b;
}

```

```

a = 123456789;
b = a;
printf("%.0f\n", b);      /* 123456792 */

return 0;
}

```

8) Eğer kaynak tür ve hedef tür de gerçek sayı türündense bu durumda bilgi kaybının niteliğine bakılır. eğer basamksal bir kayıp (magnitute kaybı) varsa tanımsız davranış oluşur. Basamksal değil de mantis kaybı varsa yine kaynak tür ile belirtilen değere en yakın büyük ya da en yakın küçük değer elde edilir.

İşlem Öncesi Otomatik Tür Dönüştürmeleri

Yalnızca değişkenlerin ve sabitlerin değil, her ifadenin de bir türü vardır. Bir operatörün opendaları farklı türlerdence önce derleyici onları aynı türe dönüştürür, sonra işlemi yapar. İşlem sonucunda elde edilen türü bu dönüştürülen ortak tür türündendir. İşlem öncesi otomatik tür dönüştürmesinin özet kuralı "küçük tür büyük türe dönüştürülür, sonuç büyük türünden çıkar" biçimindedir. Örneğin:

```

int a;
long b;
...
a + b

```

$a + b$ işleminde a long türüne dönüştürülür, ondan sonra toplama yapılır. $a + b$ ifadesinin türü long olur.

İşlem öncesi otomatik tür dönüştürmeleri geçici nesne yoluyla yapılmaktadır. Yani dönüştürülmek istenen değer önce dönüştürülecek türden geçici bir nesneye atanır, işleme o sokulur, sonra o nesne yok edilir:

```

long temp = a;
temp + b
temp yok ediliyor

```

İşlem öncesi otomatik tür dönüştürmelerinin bazı ayrıntıları vardır:

1) Eğer bölme işleminde iki operand da tamsayı türündense sonuç tamsayı türünden çıkar. Bu durumda bölme yapılır, sayının noktadan sonraki kısmı atılır. Örneğin:

```

#include <stdio.h>

int main()
{
    double a;

    a = 10 / 4;

    printf("%f\n", a);      /* 2.0 */

    return 0;
}

```

Fakat örneğin:

```

#include <stdio.h>

int main()
{
    double a;

    a = 10 / 4.;
}

```

```
printf("%f\n", a);    /* 2.5 */
return 0;
}
```

2. Eğer operandlardan her ikisi de int türünden küçükse (örneğin char-char, char-short, short-short gibi) Bu durumda önce her iki operand da bağımsız olarak int türüne dönüştürülür, sonuç int türünden çıkar. Bu kurala "int türüne yükseltme kuralı (integral promotion)" denilmektedir. Örneğin:

```
#include <stdio.h>

int main()
{
    int a;
    short b = 30000;
    short c = 30000;

    a = b + c; /* 60000, sonuç int türden */
    printf("%d\n", a);

    return 0;
}
```

Bu kuralın şöyle bir ayrıntısı da vardır: Eğer ilgili sistemde short ile int aynı uzunluktaysa ve operandlardan biri unsigned short türündense dönüştürme int türüne doğru değil unsigned int türüne doğru yapılır.

3) Operandlardan biri tamsayı türünden, diğeri gerçek sayı türündense dönüştürme her zaman gerçek sayı türüne doğru yapılır. Örneğin long ile float işleme sokulursa sonuç float türden çıkar.

4) Operandlar aynı tamsayı türünün işaretli ve işaretsiz biçimine ilişkinse dönüştürme her zaman işaretsiz türe doğru yapılır (örneğin int ile unsigned int işleme sokulsa sonuç unsigned int türünden çıkar.)

İşlem öncesi tür dönüştürmeleri aşağıdaki gibi de açıklanabilir:

1. İşleme giren operandlardan en az biri gerçek sayı türlerindense (if-else if şeklinde düşünülmelidir): Eğer operandlardan biri long double türündense diğer operand long double türüne dönüştürülür. Eğer operandlardan bir tanesi double türündense diğer operand double türüne dönüştürülür. Eğer operandlardan bir tanesi float türündense diğer operand float türüne dönüştürülür. Bu tanımlamadan şu kural çıkartılabilir. İşleme giren operandlardan bir tanesi gerçek sayı türünden, diğeri tamsayı türünden ise tamsayı türünden operand o gerçek sayı türüne dönüştürülerek işlem yapılmaktadır.

2. İşleme giren operandlar tam sayı türlerindense: Operandlardan en az bir tanesi short int, unsigned short int, char, signed char, unsigned char türlerinden biri ise öncelikle değerler int türüne dönüştürülür. Buna tamsayıya yükseltme (integer/integral promotion) denilmektedir. Daha sonra algoritma şu şekildedir.

Eğer operandlardan bir tanesi unsigned long türündense diğer operanda unsigned long türüne dönüştürülür.

Eğer operandlardan bir tanesi signed long türündense diğeri long türüne dönüştürülür.

Eğer operandlardan bir tanesi unsigned int türündense diğeri unsigned int türüne dönüştürülür.

Tür Dönüştürme Operatörü

Bazen bir değişkeni işleme sokarken onun sanki başka tür olarak işleme girmesini isteyebiliriz. Örneğin:

```
int a = 10, b = 4;
double c;
```

```
c = a / b;
```

Burada kırılma olacak ve c'ye 2 değeri atanacaktır. Şüphesiz biz a ya da b'den en az birini double olarak bildirip sorunu çözebiliriz. Ancak a ve b'nin int olarak kalması da gerekiyor olabilir. İşte bunun için tür dönüştürme operatörü kullanılmaktadır. Tür dönüştürme operatörünün genel biçimi şöyledir:

(<tür>) operand

Buadaki parantez öncelik parantezi değildir, operatör görevindedir. Tür dönüştürme operatörü tek operandlı önek bir operatördür. Öncelik tablosunun ikinci düzeyinde sağdan sola grupta bulunur:

()	Soldan-Sağa
+ - ++ -- ! (tür)	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
< > <= >=	Soldan-Sağa
== !=	Soldan-Sağa
&&	Soldan-Sağa
	Soldan-Sağa
= += -= *= /= %=, ...	Sağdan-Sola
,	Soldan-Sağa

Örneğin:

```
a = (long) b * c;
```

```
İ1: (long)b
İ2: İ1 * c
İ3: a = İ2
```

Örneğin:

```
a = (long)(a * b);
```

```
İ1: a * b
İ2: (long)İ1
İ3: a = İ2;
```

Örneğin:

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

```
int main()
{
    int a = 10, b = 4;
    double c;

    c = (double)a / b;
    printf("%f\n", c);
}
```

```
    return 0;
}
```

Örneğin:

```
a = (double)(long) b * c;
```

```
İ1: (long)b
İ2:(double)İ1
İ3:İ2 * c
İ4: a = İ3
```

Tür dönüştürmesi yine geçici nesne yoluyla yapılmaktadır. Yani derleyici önce dönüştürülmek istenen tür türünden geçici bir nesne yaratır. Dönüştürülmek istenen ifadeyi oraya atar. Sonra işleme onu sokar. En sonunda da geçici nesneyi yok eder.

Örneğin:

```
#include <stdio.h>

int main()
{
    int n;
    int i, val, total;
    double avg;

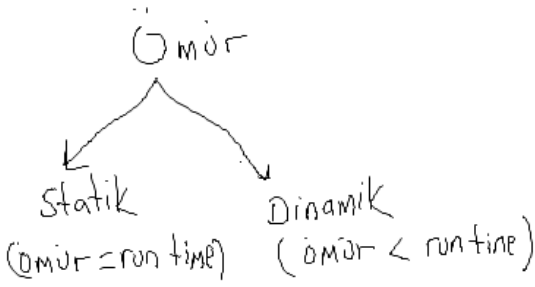
    printf("Kac sayi gireceksiniz:");
    scanf("%d", &n);

    total = 0;
    for (i = 1; i <= n; ++i) {
        printf("%d. Sayiyi giriniz:", i);
        scanf("%d", &val);
        total += val;
    }
    avg = (double)total / n;        /* dikkat! */
    printf("Ortalama = %f\n", avg);

    return 0;
}
```

Nesnelerin Ömürleri

Bir programda yaratılan nesnelerin hepsi program sonuna kadar bellekte kalmaz. Bazı nesneler programın çalışmasının belli bir aşamasında yaratılır, bir süre faaliyet gösterir, sonra yok edilir. Ömür (duration) bir nesnenin bellekte tutulduğu zaman aralığına denilmektedir. Bir nesne potansiyel en uzun ömür programın çalışma zamanı kadar (run time) olabilir. Bir nesnenin ömrü statik ya da dinamik olabilir. Statik ömür demek ömürün çalışma zamanına eşit olması demektir. Statik ömürlü nesneler program çalışmak üzere belleğe yüklendiğinde yaratılır, program sonlanana kadar bellekte kalır. Dinamik ömürlü nesnelere ömür programın çalışma zamanından küçüktür.



Global Nesnelerin Ömürleri

Global nesnelere statik ömürlüdür. Yani bunlar programın çalışma zamanı boyunca bellekte yer kaplarlar. Global nesnelere belleğin Data ve BSS denilen bölümlerinde tutulmaktadır.

Yerel Değişkenlerin Ömürleri

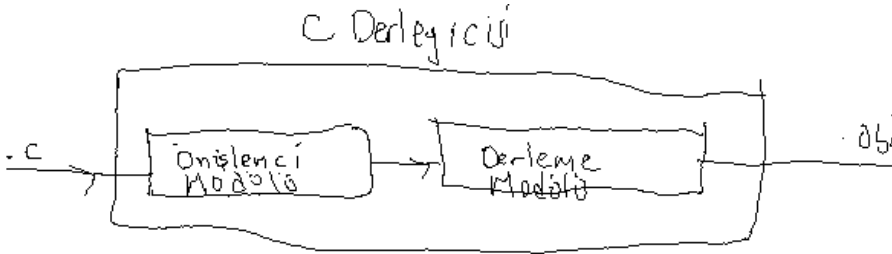
Yerel nesnelere dinamik ömürlüdür. Programın akışı nesnenin bildirildiği bloğa girdiğinde o blokta bildirilmiş bütün yerel nesnelere yaratılır, akış o bloktan çıktığında o blokta bildirilmiş bütün yerel nesnelere otomatik olarak yok edilir. Yerel değişkenlere belleği "stack" denilen bir bölümünde yaratılmaktadır. Stack'te yaratım ve yok edim çok hızlı yapılır.

Parametre Değişkenlerinin Ömürleri

Parametre değişkenleri de dinamik ömürlüdür. Fonksiyon çağrıldığında yaratılırlar, fonksiyon çalıştığı sürece bellekte kalırlar. Fonksiyonun çalışması bittiğinde yok edilirler. Parametre değişkenleri de stack denilen bölümde yaratılmaktadır.

Önişlemci Kavramı (Preprocessor)

Aslında bir C derleyicisi iki modülden oluşmaktadır: "Önişlemci Modülü" ve "Derleme Modülü". Önişlemci modülü kaynak kodu alır, onun üzerinde bazı işlemler uygular. Sonra onu derleme modülüne verir. Asıl derleme işlemi derleme modülü tarafından gerçekleştirilmektedir.



Pek çok programlama dilinde böyle bir önişlemci modülü yoktur.

C'de '#' karakteri ile başlayan satırlar önişlemciye ilişkindir. Yani önişlemci yalnızca başı '#' ile başlayan satırlarla ilgilenmektedir. '#' atomunu önişlemci komutu denilen bir anahtar sözcük izler. '#' ile bu anahtar sözcük arasında boşluk karakterleri bırakılabilir. Fakat önişlemci komutları tek bir satırda bulunmak zorundadır. Önişlemci komutu önişlemciye ne yapması gerektiğini anlatır. 20'ye yakın önişlemci komutu vardır. Ancak en çok kullanılanı #include ve #define komutlarıdır. Kursumuzun bu bölümünde bu iki komut ele alınacaktır. Diğerleri kursumuzun sonlarına doğru ele alınacaktır.

#include Komutu

#include komutunun açılma parantezleri içerisinde ya da iki tırnak içerisinde bir dosya ismi izlemek zorundadır. Örneğin:

```
#include <stdio.h>
#include "a.c"
```

#include komutu kaynak kodun tepesinde bulunmak zorunda değildir. Herhangi bir yerd bulunabilir. Fakat bir satırı sadece kendisi kaplamak zorundadır. Önışlemci #include komutunu gördüğünde dosyayı diskten okur ve onun içindekilerini komutun yazılı olduğu yere yapıştırır. Tabi bunu geçici bir dosya açarak yapmaktadır. Derleme modülüne de bu geçici dosyayı verir. Derleme modülü kodu aldığıında artık orada #include görmez. Onun içeriğini görür. #include komutu ile include ettiğimiz dosyanın içerisinde C'ce anlamlı şeyler olmalıdır. Örneğin:

```
/* test.c */

int a;

/* sample.c */

#include <stdio.h>

int main()
{
    #include "test.c"

    a = 10;
    printf("%d\n", a);

    return 0;
}
```

Eğer dosya açılmal parantezler içerisinde include edilirse bu durumda dosya derleyicinin belirlediği bir dizinde aranır. Derleyici install edilirken başlık dosyaları bir dizine çekilmektedir. Açılmal ile dosya ismi belirtilirken o dizine bakılır. C'nin standart başlık dosyalarının açılmal parantezler içerisinde include edilmesi uygundur. eğer dosya ismi iki tırnak içerisinde belirtilirse derleyiciler onu önce bulunulan dizinde (ya da projenin yüklü dizinde) arar eğer orada bulamazlarsa sani açılmal parantez ile belirtilmiş gibi derleyicinin belirlediği dizine de bakılır. Bu durumda bizim kendi dosyalarımız iki tırnak içerisinde include etmemiz daha uygundur. Çünkü onlar muhtemelen kendi dizinimizde bulunacaktır.

C standartları #include komutunun semantiğini ana hatlarıyla belirtip çoğu şeyi derleyicileri yazanların isteğine bırakmıştır. Yani dosya ismi açılmal parantez ile belirtildiğinde nerelere bakılacağı, iki tırnak içerisinde belirtildiğinde nerelere bakılacağı hep derleyicileri yazanların isteğine bırakılmıştır.

Pek çok derleyicide include dosyaları aranırken programcının istediği dizinlere de bakılması sağlanmıştır. Örneğin Visual Studio IDE'sinde Projeye ayarlarında "C++/General/Additional Include Directories" seçenekleri ile önışlemcinin bizim belirlediğimiz dizilere de bakması sağlanabilir. gcc derleyicilerinde -I seçeneği ile dizin eklenebilmektedir. Örneğin:

```
gcc -I /home/csd/Study/C -o sample sample.c
```

include edilen bir dosyanın içerisinde de #include komutları bulunabilir. Önışlemci bunların hepsini düzgün biçimde açabilmektedir. Örneğin biz pek çok dosyayı "myheaders.h" dosyası içerisinde include etmiş olalım:

```
/* myheaders.h */

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

Asıl programımızda yalnızca "myheaders.h" dosyasını include edebiliriz:


```
/* sample.c */
```

```
#include "myheaders.h"  
...
```

Fakat include işlemlerinde döngüsellik olamaz. include edilecek dosyanın ismi ve uzantısı herhangi bir biçimde olabilir. Ancak C'de gelenek uzantının .h biçiminde olmasıdır.

#define Komutu

#define komutu text editörlerdeki "find and replace" işlemine benzer bir işlem yapmaktadır. Komutun genel biçimi şöyledir:

```
#define STR1      STR2
```

Örneğin:

```
#define MAX      100  
#define MIN      100 - 20
```

#define anahtar sözcüğünden sonra boşluk karakterleri atılır ve ilk boşluksuz yazı kümesi elde edilir. Buna STR1 diyelim. Sonra önişlemci yine boşlukları atar ve satır sonuna kadarki tüm karakterleri elde eder. Buna da STR2 diyelim. Sonra kaynak kodda STR1 gördüğü yere STR2 yazısını yerleştirir. Sonucu derleme modülüne verir. Örneğin:

```
#include <stdio.h>  
  
#define MAX      100 - 50  
  
int main()  
{  
    int a;  
  
    a = MAX * 2;  
    printf("%d\n", a);  
  
    return 0;  
}
```

Burada #define komutunda STR1 MAX, STR2 de 100 - 50'dir. Önişlemci MAX atomlarını 100 - 50 ile yer değiştirir. Derleme modülüne kod aşağıdaki gibi verilir:

```
...stdio.h içeriği...  
  
int main()  
{  
    int a;  
  
    a = 100 - 50 * 2;  
    printf("%d\n", a);  
  
    return 0;  
}
```

Dolayısıyla bu program çalıştırıldığında ekrana 0 basılır.

Önişlemci hesp yapmaz. Bir yazıyı başka bir yazıyla yer değiştirir. Örneğin:

```

#include <stdio.h>

#define MAX    (100 - 50)

int main()
{
    int a;

    a = MAX * 2;
    printf("%d\n", a);

    return 0;
}

```

Burada ekrana 100 basılacaktır.

STR1 olarak yalnızca değişken ve anahtar sözcük atom kullanılır. Örneğin:

```

#define +        -        /* geçerli değil */
#define 100      200      /* geçerli değil */
#define beep()  putchar('\a') /* geçerli */
#define MAX     +        /* geçerli */

```

Aşağıdaki kod tamamen geçerlidir:

```

#include <stdio.h>

#define ana      main
#define tam      int
#define eger     if
#define yazf     printf
#define geridon  return

tam ana()
{
    tam a = 100;

    eger(a == 100)
        yazf("tamam\n");

    geridon 0;
}

```

Bir yazıya karşılık bir sayı karşı düşürülmesi durumunda bu yazıya sembolik sabit (symbolic constants) ya da makro (macro) denilmektedir. Örneğin:

```

#define MAX    100

```

Buarada MAX bir sembolik sabittir.

STR2'de başka makrolar bulundurulabilir. Önışlemci açımı özyinelemeli yapar. Yani açtığı makroyu yeniden açar. Ta ki açılacak birşey kalmayana kadar. Örneğin:

```

#include <stdio.h>

#define MAX    100
#define MIN    (MAX - 50)

int main()
{
    int a;

```

```

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

return 0;
}

```

Burada MIN yerine önce (MAX - 50) yerleştirilir. Sonra bu da yeniden önişlemciye sokulur. Böylece (100 - 50) elde edilir. Burada #define'ların sırası farklı olsaydı da değişen birşey olmayacaktı. Yani önişlemci #define satırları üzerinde değişiklik yazmaz.

#define önişlemci komutu ile iki tırnak içerisindeki yazılarda değişiklik yapamayız. Örneğin:

```

#include <stdio.h>

#define MAX 100

int main()
{
    printf("MAX");    /* MAX çıkar */

    return 0;
}

```

Örneğin:

```

#include <stdio.h>

#define CITY "Ankara"

int main()
{
    printf(CITY);    /* Ankara çıkar */

    return 0;
}

```

Aşağıdaki makro geçerlidir:

```

#define MYHEADER <stdio.h>
#include MYHEADER

```

#define komutu #include komutunda değişiklik yapabilmektedir.

Eğer STR2 yazılmazsa bu durumda STR1 görülen yere boşluk yerleştirilir. Yani STR1 silinir. Örneğin aşağıdaki kodda derleme sorunu oluşmaz:

```

#include <stdio.h>

#define in

int main()
{
    in
        in in in in

    return 0;
}

```

#define komutu nereye yerleştirilmişse onun aşağısında etkili olur. Örneğin:

```

#include <stdio.h>

int main()
{
    int i;

    for (i = 0; i < SIZE; ++i) {        /* geçersiz! */
        /* ... */
    }

#define SIZE    100

    for (i = 0; i < SIZE; ++i) {
        /* ... */
    }

    return 0;
}

```

#define komutunda yerel, global gibi bir kavram yoktur. Komut nereye yazılmışsa dosyanın sonuna kadarki bölgede geçerli olur.

Önişlemci #include komutu ile bir başlık dosyasını açtığında o dosyanın içeriğini de önişleme sokar. Yani oradaki #define'lar, #include'lar da etki gösterir. Böylece biz birtakım sembolik sabit tanımlamalarını bir başlık dosyasında yapabiliriz. Onu include ettiğimizde o makrolar etkili olur.

C standartlarına göre bir makro sabit ikinci kez farklı bir biçimde define edilirse tanımsız davranış oluşur. Örneğin:

```

#define MAX    100
...
#define MAX    200

```

Burada pek çok derleyici ikinci komuta kadar birincisini etkin yapar, ikincikomuttan sonra ikincisinin etkin olduğu düşünür. Fakat aslında derleyicinin burada ne yapacağı belli değildir. Bundan kaçınmak gerekir. Tabi makronun aynı biçimde birden fazla kez tanımlanması soruna yol açmaz.

Ayrıca #define komutunun parametrelili bir kullanımı da vardır. Bu konuyu kursumuzun sonlarına doğru ele alınacaktır.

Sembolik sabitler geleneksel olarak büyük harflerle isimlendirilmektedir. Bu onların program içerisinde ayırmsanmasını kolaylaştırmaktadır.

#define Komutuna Neden Gereksinim Duyulmaktadır?

#define komutu kaynak kod üzerinde bazı değişiklikler yapılmasına olanak sağlar. Fakat kursumuzun bu noktasında bu tür değişikliklerden nasıl fayda sağlanacağı açıklanmayacaktır. #define komutu okunabilirliği artırmak için sıkça kullanılır. Örneğin bir personel takip programında aşağıdaki gibi bir satırı buşunuyor olsun:

```

if (n > 732) {
    ...
}

```

Bu mu daha onunabilirdir (readable), yoksa aşağıdaki mi?

```

#define PERSONEL_SAYISI    732
...
if (n > PERSONEL_SAYISI) {
    ...
}

```

```
}
```

İşte programımızda çeşitli sayıları yazısal biçimde ifade edersek okunabilirliği artırmış oluruz. #define komutu kod üzerinde değişiklikleri daha az zahmetle yapmamızı sağlar. Örneğin bir programın pek çok yerinde 100 sayısı kullanılıyor olsun. Biz bunu 200 ile yer değiştirmek isteyelim. Eğer 100'ü define edersek bunu tek yerden yapabiliriz:

```
#include <stdio.h>

#define SIZE 100

int main()
{
    int i;

    for (i = 0; i < SIZE; ++i) {
        /* ... */
    }

    for (i = 0; i < SIZE; ++i) {
        /* ... */
    }
    /* ... */

    return 0;
}
```

Fonksiyon Prototipleri

Derleme işleminin bir yönü vardır ve bu yön yukarıdan aşağıya doğrudur. Derleyici çağrılan bir fonksiyonu gördüğünde çağrılma noktasına kadar bu fonksiyonun geri dönüş değerinin türünü tespit etmek zorundadır. Eğer çağrılan fonksiyon çağırılan fonksiyonun daha yukarısında tanımlanmışsa çağrılma noktasına kadar derleyici bu tespit yapmış olur. Fakat eğer çağrılan fonksiyon çağırılan fonksiyonun daha altında tanımlanmışsa bu durumda derleyici çağrılma noktasına kadar fonksiyonun geri dönüş değerinin türünü tespit edememiştir.

Eğer derleyici çağrılma noktasına kadar fonksiyonun geri dönüş değerinin türünü tespit edememişse bu durumda fonksiyonun int geri dönüş değerine sahip olduğunu varsayarak kodu üretir. Eğer fonksiyonunun daha sonra başka bir geri dönüş değerine sahip olduğunu derleyici görürse geri dönüp ürettiği kodu düzeltmez. Bu durum error oluşturur. Fakat derleyici daha sonra fonksiyonun int geri dönüş değerine sahip olarak tanımlandığı görürse sorun oluşmaz.

Örneğin:

```
#include <stdio.h>

int main()
{
    foo();    /* geçersiz */

    return 0;
}

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

Fakat örneğin:

```
#include <stdio.h>

int main()
{
    foo();    /* geçerli */

    return 0;
}

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

Örneğin:

```
#include <stdio.h>

int main()
{
    double result;

    result = div(10, 2.5);    /* error! */

    return 0;
}

double div(double a, double b)
{
    return a / b;
}
```

İşte derleyiciye bir fonksiyonun geri dönüş değeri ve parametreleri hakkında bilgi veren bildirimlere "fonksiyon prototipleri" denilmektedir. Eğer çağrılan fonksiyon çağırılan fonksiyonun daha aşağısında tanımlanmışsa biz çağırma işleminin yukarısına fonksiyon prototipini yerleştirmeliyiz. Prototip bildiriminin genel biçimi şöyledir:

[geri dönüş değerinin türü] <fonksiyon ismi>([parametre bildirimi]);

Örneğin:

```
double divide();
double divide(double a, double b);
divide(void);
```

geçerli prototip bildirimleridir. Prototipte parametre bildirimi yapmak iyi bir tekniktir. Çünkü derleyici bu durumda çağırılma ifadesinde parametre kontrolü ve uygun tür dönüştürmesini yapar. Prototip bildiriminde parametre değişkenlerinin yalnızca türleri belirtilebilir. Örneğin:

```
double divide(double, double);
```

Fakat parametre değişkenlerinin isimlerini yazmak okunabilirliği artırır. Örneğin aşağıdakilerden hangisi daha okunabilirdir?

```
double usal(double, double);
double usal(double taban, double us);
```

Parametre değişkenlerinin isimlerinin hiçbir önemi yoktur. Hatta istenirse bazı parametre değişkenlerine isim verili bazılarına verilmeyebilir. Tabi iyi teknik tüm parametre değişkenlerinin isimlendirilmesidir.

Protoipteki geri dönüş değeri ve parametre türleriyle tanımlamadakilerin tam olarak aynı olması gerekir. Aksi halde kod geçerli olmaz. Örneğin:

```
#include <stdio.h>

double divide(float x, double y);

int main()
{
    double result;

    result = divide(10, 2.5);
    printf("%f\n", result);

    return 0;
}

double divide(double a, double b)    /* geçersiz! */
{
    return a / b;
}
```

Fakat parametre değişkenlerinin isimlerinin aynı olması zorunluluğu yoktur.

Fonksiyon tanımlamasının ilk satırı alınıp kopyalanır ve sonuna noktalı virgül getirilirse prototip elde edilir. Örneğin:

```
double divide(double a, double b)
{
    return a / b;
}
```

Bu fonksiyonun prototipi şöyledir:

```
double divide(double a, double b);
```

Aynı protipi birden fazla kez bildirmek sorun oluşturmaz. Örneğin:

```
double div(double, double);
double div(double a, double b);
double div();
```

Bunların hepsi bir arada bulunabilir.

Küçük olmayan projelerde ve kodlarda fonksiyonlar nerede tanımlanmış olursa olsun onların prototiplerini kaynak kodun tepesine yerleştirmek ya da bir başlık dosyasına yerleştirip onu include etmek iyi bir tekniktir.

Eğer biz fonksiyonun prototini ya da tanımlamasını daha yukarıda oluşturmuşsak bu durumda çağrılma ifadesinde derleyici argümanları sayıca ve türce kontrol eder. Örneğin:

```
#include <stdio.h>

void foo(int a, int b);

int main()
{
    foo(100);    /* geçersiz! */

    return 0;
}
```

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

Prototip bir bildirimdir, tanımlama değildir. Yani prototip oluşturduğunuzda biz fonksiyonu yazmış (tanımlamış) olmayız. Yalnızca derleyiciye bir ön bildirimde bulunmuş oluruz.

Prototip bildiriminde parametre parantezinin boş bırakılmasıyla oraya void yazılması farklı anlamlara gelmektedir. eğer parametre parantezi boş bırakılırsa bu durum derleyicinin çağrılma ifadelerinde argüman kontrolü yapmayacağı anlamına gelir. Örneğin:

```
void foo();
```

Böyle bir bildirimde biz foo'yu istediğimiz kadar çok argümanla çağırabiliriz. Derleyici bir kontrol uygulamaz. Ancak parametre parantezinin içine void yazılırsa bu durum fonksiyonun parametre almadığı anlamına gelir. Örneğin:

```
void foo(void);
```

Biz artık foo'yu argümanla çağıramayız.

Anahtar Notlar: Aslında bu durumun tarihsel bazı gerekçeleri vardır. Eskiden C'de fonksiyon prototipleri yoktu. Zaten parametre parantezlerinin içi boş bırakılmak zorundaydı. Sonra C'ye prototipi kavramı eklendiği zaman eski kodlar geçerli olsun diye bu semantik muhafaza edildi. Eskiden yapılan parametre parantezinin içinin boş bırakıldığı bildirimlere prototip değil "fonksiyon bildirimi" deniliyordu.

Fonksiyon tanımlamasında parametre parantezinin içinin boş bırakılmasıyla void yazılması arasında bir farklılık yoktur. Her iki durum da fonksiyonun parametreye sahip olmadığı anlamına gelir. Yani:

```
void foo()
{
    ...
}
```

ile,

```
void foo(void)
{
    ...
}
```

tanımlamaları tamamen eşdeğerdir.

Fonksiyon prototip bildirimleri nerede yapılmalıdır? Prototip bildirimleri yerel ya da global düzeyde yapılabilir. Yine eğer yerel düzeyde yapılacaksa C90'da blokların başlarında yapılması zorunludur (Tabii C99 ve C++'ta blokların herhangi bir yerinde yapılabilir.) Eğer prototip bildirimi global düzeyde yapılırsa tüm dosya bundan etkilenir. Yerel düzeyde yapılırsa yalnızca o bloktaki kullanım bundan etkilenir. Yani başka bloklarda sanki bildirim hiç yapılmamış gibi etki gösterir. Örneğin:

```
#include <stdio.h>

void bar(void)
{
    void foo(int a, int b);
}
```



```

foo(10, 20);      /* prototip etki gösterir */
}

int main()
{
foo(10, 20);      /* geçersiz! sanki protip yokmuş gibi etki gösterir*/

return 0;
}

void foo(int a, int b)
{
/* ... */
}

```

Fakat hemen her zaman fonksiyon prototip bildirimini global düzeyde yapılmaktadır.

Derleyici bir fonksiyonun prototipini ya da tanımlamasını görmeden onun çağrıldığı görürse sanki onun prototipi aşağıdaki gibi yazılmış varsayar:

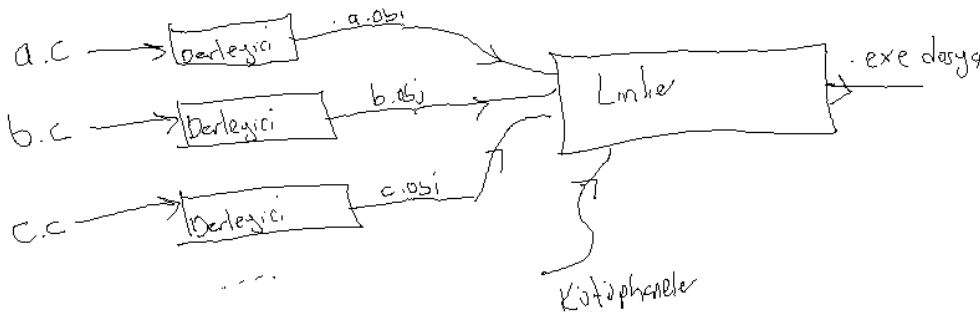
```
int some_function();
```

Bir fonksiyonun prototip bildirimini yazılmış olması onu çağırılmayı zorunlu hale getirmez.

Standart C Fonksiyonlarının Prototipleri

Kaynak programımızda olmayan bir fonksiyonu çağırarak bile derleme aşamasından başarıyla geçilir. Eğer prototip yoksa derleyici fonksiyonun geri dönüş değerinin türünü int varsayar. Tabi bu durumda argüman kontrolü yapmaz. Eğer prototip varsa derleyici geri dönüş değerinin türünü anlar. Her iki durumda da derleme işlemi başarıyla geçilir. Fakat derleyici object kod içerisine linker için bir mesaj yazar. Mesajda "sevgili linker, ben bu fonksiyonu bulamadım. Sen onu diğer objectg modüllerde ve kütüphane dosyalarında ara. Bulursan oradan al. Bulamazsan seninde elinden birşey gelmez. Ne yapalım...". Bu mesajı okuyan linker fonksiyonu diğer modüllerde ve kütüphanelerde arar.

Aslında link işlemi tek bir object modülle yapılmak zorunda değildir. Birden fazla object modül ve kütüphane dosyaları birlikte link edilip tek bir exe dosya elde edilebilir.



Uzantısı Windows'ta .lib ve .dll olan, UNIX/Linux sistemlerinde .a ve .so olan dosyalara kütüphane dosyaları denir. Kütüphaneler statik ve dinamik olmak üzere ikiye ayrılmaktadır. Uzantısı Windows'ta .lib, UNIX/Linux'ta .a olan dosyalara statik kütüphane dosyaları, uzantısı Windows'ta .dll, UNIX/Linux'ta .so olan dosyalara dinamik kütüphane dosyaları denilmektedir. Kütüphanelerin içerisinde derlenmiş halde fonksiyonlar bulunmaktadır.

Eğer bir fonksiyon linker tarafından statik kütüphanelerde bulunursa oradan çekilir ve .exe dosyanın içerisine yazılır. Eğer fonksiyon dinamik kütüphane içerisinde bulunursa linker fonksiyonu oradan çekip exe dosyaya yazmaz. Linker exe dosya içerisine işletim sistemi için şöyle bir not yazmaktadır: "Sevgili işletim sistemi bu programı çalıştırırken bu programın kullandığı dinamik kütüphaneleri de bul onları da belleğe yükle ki benim

fonksiyonların çalışsın". Görüldüğü gibi statik link işleminde exe dosya zaten her şeyi içermektedir. Programı artık başka bir makineye götürürken statik kütüphane dosyasını götürmememize gerek yoktur. Ancak dinamik link işleminde biz exe dosyasıyla birlikte onun kullandığı dinamik kütüphane dosyalarını da hedef makineye taşımamız gerekir.

C'nin standart fonksiyonları da kütüphaneler içerisinde derlenmiş bir biçimde bulunmaktadır. Dolayısıyla bunlar linker tarafından ele alınırlar. C derleyicileri standart C fonksiyonlarının farkında değildir. Yani printf fonksiyonuna bizim foo fonksiyonuna yaptığımızdan farklı bir muamele uygulamaz. Biz printf fonksiyonunu çağırdığımızda derleyici onu kaynak dosyada bulamayacağı için linker'a havale eder. Linker da onu baktığı kütüphane dosyalarında bulur. Aslında linker de standart C fonksiyonlarının farkında değildir. O yalnızca belirlenen kütüphanelere bakmaktadır. Standart C fonksiyonlarının yalnızca biz programcılar standart olduğunun farkındayızdır. Tabi onların kütüphane içerisinde bulunduğundan emin oluruz.

Standart C fonksiyonlarını derleyici tanımadığına göre onların geri dönüş değerlerinin türlerini ve parametrik yapılarını bilmez. Onlar çağrıldığında onların prototipini görmezse geri dönüş değerlerini int kabul eder. O halde standart C fonksiyonlarının da prototiplerinin bulundurulması gerekir. Örneğin aşağıdaki programda sqrt fonksiyonun prototipi bulundurulmadığı için program başarılı derlenip link edilmiştir. Fakat yazılış çalışır:

```
#include <stdio.h>

int main()
{
    double result;

    result = sqrt(10);
    printf("%f\n", result);

    return 0;
}
```

Halbuki prototip eklersek doğru çalışacaktır:

```
#include <stdio.h>

double sqrt(double);

int main()
{
    double result;

    result = sqrt(10);
    printf("%f\n", result);

    return 0;
}
```

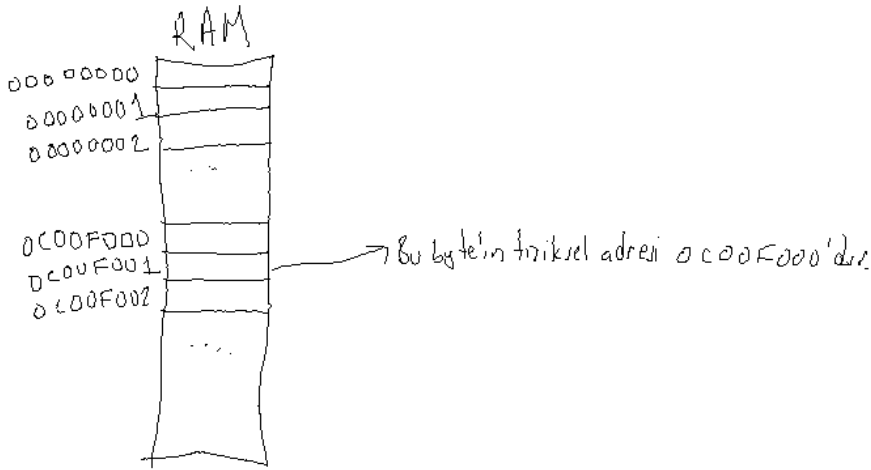
Buradan çıkan sonuç "standart fonksiyonlarının da prototiplerinin bulundurulması" gerektiğidir.

İşte standart C fonksiyonlarının prototipleri gruplanarak çeşitli başlık dosyalarına yerleştirilmiştir. Biz onları elle yazmak yerine o dosyaları include edebiliriz. Örneğin bütün matematiksel fonksiyonların prototipleri <math.h>, bütün klavye, ekran ve dosya fonksiyonlarının prototipleri <stdio.h>, genel amaçlı fonksiyonların prototipleri <stdlib.h> dosyası içerisinde yer almaktadır. Başlık dosyalarında fonksiyonların tanımlamaları yoktur. Yalnızca prototipleri vardır. Fonksiyonlar derlenmiş bir biçimde kütüphanede bulunmaktadır.

Adres Kavramı

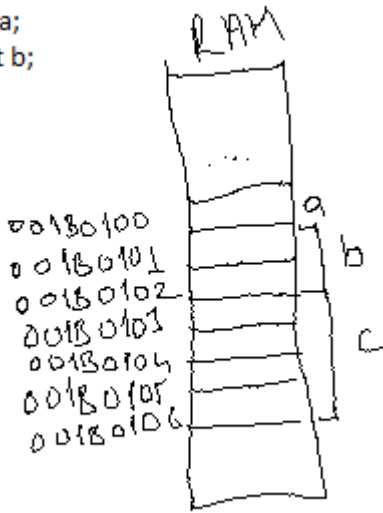
CPU'nun doğrudan bağlı olduğu ana bellek (yani RAM) byte'lardan oluşmaktadır. Her byte da 8 bitten oluşur. Belleğin her bir byte'ına ilk byte 0 olmak üzere bir numara verilmiştir. Buna ilgili byte'ın fiziksel adresi denir.

Byte'ların fiziksel adresleri geleneksel olarak 16'lık sistemde gösterilmektedir. Örneğin:



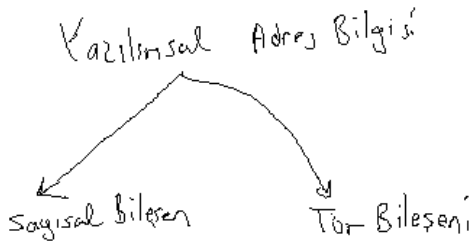
Bellekteki nesnelerin de adresleri vardır. Çünkü onlar da yaşam süresi içerisinde RAM'de bir yerlerde bulunmaktadır. Örneğin:

```
char a;  
short b;  
int c;
```

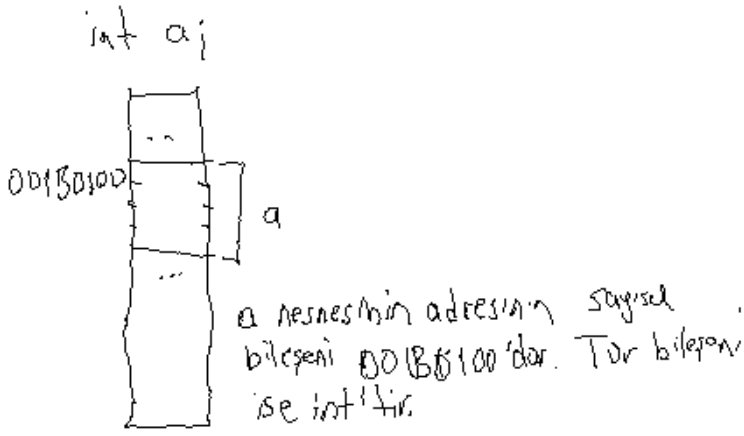


Her nesnenin bir fiziksel adresi vardır. Bir byte'tan büyük olan nesnelerin fiziksel adresleri onların en düşük adres numaralarıyla belirtilir. Örneğin b'nin fiziksel adresi 001B0101'dir, c'nin 001B0103'tür. Şüphesiz yerel nesneler dinamik ömürlü olduğu için, yerel nesneler için ayrılan alan onların ömürleri bittiğinde artık başka yerel nesneler için kullanılabilir.

Yazılımsal adres bilgisi iki bileşenli bir bilgidir. Adresin sayısal ve tür bileşenleri vardır:



Adresin sayısal bileşeni nesnenin bellekteki fiziksel adres numarasını belirtir. Tür bileşeni ise o fiziksel adresteki nesnenin türünü ifade eder. Örneğin:



Bundan sonra kursumuzda "adres" denildiğinde yazılımsal adres kavramı anlaşılmalıdır. (Donanımsal ya da fiziksel adres kavramı tek bileşenlidir ve yalnızca sayı belirtir.)

C'de adresler tamamen ayrı bir tür belirtmektedir. Nasıl int, long, double gibi türler varsa adresleri tutabilen türler de vardır. C'de adres türleri (pointer types) adresin tür bileşeni ile ifade edilir. Adres türleri C'de adresin tür bileşeni ile ifade edilir. Yani örneğin bu türe "adres" denmez de "int türden adres (pointer to int)", "long türden adres" denir.

C'de nasıl int türden, long türden, double türden sabit kavramı varsa adres sabiti kavramı da vardır. T türünden adres sabiti tür dönüştürme operatörü ile oluşturulur. Genel biçimi şöyledir:

(<tür bileşeni> *) <sayısal bileşen>

Adresin sayısal bileşeni fiziksel adres numarası belirtir. Tür bileşeni ise o fiziksel adresteki bilginin türünü belirtmektedir.

Örneğin:

(int *)0x001B1010

Bu ifade int türden adres sabiti belirtir. Örneğin:

(double *)0x001C2000

Bu ifade double türden adres sabiti anlamına gelir.

Genel biçimdeki yıldız adres anlamına gelir. Adres sabitlerini yazarken tür bileşeninin 16'lık sistemde belirtilmesi zorunlu değildir. Fakat gelenek bu yöndedir.

C'de nasıl int türünü tutan, long türünü tutan, double türünü tutan nesnelere bildirilebiliyorsa adres bilgilerini tutan da nesnelere bildirilebilir. Bunlara gösterici (pointer) denir.

Diziler (Arrays)

Elemanları aynı türden olan ve bellekte ardışıl bir biçimde bulunan veri yapılarına dizi denir (bu tanım ezberlenmelidir). Aralarında fiziksel ya da mantıksal ilişki bulunan birden fazla nesneni oluşturduğu topluluğa veri yapısı (data structure) denilmektedir. Dizi bir veri yapısıdır. Çünkü dizi dediğimizde birden fazla eleman (özel olarak bir eleman da olabilir) söz konusu edilir. Dizi elemanlarının hepsi aynı türdendir. Dizi elemanları bellekte ardışıl bir biçimde tutulur. Yani bir elemandan sonra hiç boşluk olmadan diğer elemana geçilir. Halbuki örneğin:

int a, b;

gibi bir bildirimde a ve b'nin bellekte ardışıl tutulacağını bir garantisi yoktur.

Dizi bildiriminin genel biçimi şöyledir:

```
<tür> <dizi ismi><[<uzunluk>]>;
```

Örneğin:

```
int a[10];
long b[20];
char c[5];
```

gibi.

Küme parantezi içerisindeki uzunluk belirten ifadenin tamsayı türlerine ilişkin sabit ifadesi olması zorunludur.

Örneğin:

```
int a[3 + 2];    /* geçerli */
int n = 5;
int b[n];        /* geçerli değil */
```

Derleyicinin dizi uzunluğunu daha derleme aşamasında bilmesi gerekir. Bu nedenle dizi uzunluklarının sabit ifadesi olması zorunlu tutulmuştur.

Bir dizinin ismi o dizinin tamamını temsil eder. Örneğin:

```
int a[10];
```

bildirimde a 10 elemanlı bir dizi nesnesidir. Dizi türleri sembolik olarak önce tür sonra bir köşeli parantez içerisinde uzunluk ifadesiyle temsil edilmektedir. Örneğin yukarıda a'nın türü int[10] biçiminde ifade edilir. int[10] "10 elemanlı int türden dizi" anlamına gelir. Örneğin:

```
char s[20];
```

Burada s, char[20] türündendir.

Dizi bildirimini ile normal nesne bildirimini birlikte yapılabilir. Örneğin:

```
int a[10], b, c[5];
Bu bildirim eşdeğeri şöyledir:
```

```
int a[10];
int b;
int c[5];
```

Dizi Elemanlarına Erişim ve [] Operatörü

Dizi elemanlarına [] operatörüyle erişilir. Köşeli parantezlerin içerisine tamsayı türlerine ilişkin bir indeks ifadesi yazılmak zorundadır. Bu ifade sabit ifade olmak zorunda değildir. Dizinin ilk elemanı sıfırıncı indeks'li elemanıdır. Bu durumda son eleman (n elemanlı bir dizide) n - 1'inci indeksteki eleman olur. Örneğin:

```
int a[10];
```

a[3] ifadesi int türündendir. a ifadesi ise int[10] türündendir.

Örneğin:

```
#include <stdio.h>

int main(void)
{
    int a[10];
    int i;

    for (i = 0; i < 10; ++i)
        a[i] = i * i;

    for (i = 0; i < 10; ++i)
        printf("%d ", a[i]);
    printf("\n");

    return 0;
}
```

Dizi Elemanlarına İlkdeğer Verilmesi

Bir dizinin elemanlarına değer atanmamışsa elemanların içerisinde ne vardır? Dizi yerel ise dizinin tüm elemanlarında çöp değer, globalse sıfır bulunur.

Dizi elemanlarına ilkdeğer küme parantezleri ile verilir. Küme parantezlerinin içerisinde değerler ',' atomu ile ayrılarak yazılır. Örneğin:

```
int a[3] = {1, 2, 3};
```

Verilen ilkdeğerler dizi elemanlarına sırasıyla yerleştirilmektedir:



Dizinin uzunluğundan fazla elemanına değer veremeyiz. Örneğin:

```
int a[3] = {1, 2, 3, 4};          /* geçersiz! */
```

Fakat dizinin az sayıda elemanına ilkdeğer verebiliriz. Bu durumda geri kalan elemanlar dizi ister yerel yerel olsun, ister global olsun derleyici tarafından sıfırlanır. Örneğin:

```
int a[5] = {1, 2 };              /* Dizinin 2, 3 ve 4'üncü indeksli elemanları sıfırdır */
```

İlkdeğer verirken küme parantezlerinin içi boş bırakılamaz. Örneğin:

```
int a[] = {};
```

Örneğin yerel bir dizinin tüm elemanlarını sıfırlamanın en kolay şöyledir:

```
int a[10] = {0};
```

C'de (C90 ve C99 ve C11'de) küme parantezleri içerisinde verilen ilkdeğerlerin sabit ifadesi olması zorunludur (ancak C++'ta sabit ifadesi olmayabilir). Örneğin:

```
int n = 10;
int a[3] = {5, n, 15};      /* geçersiz! */
```

Bazı derleyiciler bir eklenti olarak küme parantezleri içerisinde sabit ifadesi olmayan ilkdeğer ifadelerini kabul edebilmektedir. Fakat bu standart bir özellik değildir.

Diziye ilkdeğer verilirken küme parantezlerinin içi boş bırakılabilir. Bu durumda derleyici verilen ilkdeğerleri sayar ve dizinin o uzunlukta açılmış olduğunu kabul eder. Örneğin:

```
int a[] = {1, 2, 3};      /* geçerli */
```

Burada a dizisi 3 elemanlıdır.

Tabi diziye ilkdeğer verilmiyorsa uzunluk belirtmek zorunludur. Örneğin:

```
int a[];      /* geçersiz! */
```

C'de virgüllerle ayrılmış listelerde genel olarak (ileride başka konularda da karşılaşılabilecek) son elemandna sonra ',' atomu bulundurulabilir. Bu yasak değildir. Örneğin:

```
int a[3] = {1, 2, 3, };    /* geçerli */
int b[] = {1, 2, 3, };    /* geçerli */
```

C90'da dizinin istediğimiz bazı elemanlarına ilkdeğer veremeyiz. Böyle bir sentaks yoktur. Ancak C99'da bu aşağıdaki sentaksla mümkündür:

```
int a[10] = {10, [4] = 100, 200, [9] = 300};
```

C99'a göre burada dizinin sıfırıncı indeksli elemanında 10, 4'üncü indeksli elemanında 100, 5'inci indeksli elemanında 200 ve 9'uncu indeksli elemanında 300 vardır. Diğer elemanlarda sıfır bulunur. Bu özellik bazı C90 derleyicilerinde bir eklenti (extension) olarak desteklenmektedir.

Bir Dizi İçerisindeki En Büyük (ya da En Küçük) elemanın Bulunması Algoritması

Bu problemin tek bir algoritmik çözümü vardır. Dizinin ilk elemanı enbüyük kabul edilip bir nesnede saklanır. Sonra geri kalan tüm elemanlara bakılır, daha büyüğü varsa nesne içerisindeki değer yer değiştirilir. Örneğin:

```
#include <stdio.h>

#define SIZE      10

int main(void)
{
    int a[SIZE] = { 10, 4, 6, 34, 32, -34, 39, 21, 9, 22 };
    int i, max;

    max = a[0];

    for (i = 1; i < SIZE; ++i)
        if (a[i] > max)
            max = a[i];

    printf("Max = %d\n", max);
}
```

```
    return 0;
}
```

Sınıf Çalışması: Yukarıdaki dizide en büyük ve en küçük eleman arasındaki farkı ekrana yazdırınız

Çözüm:

```
#include <stdio.h>

#define SIZE      10

int main(void)
{
    int a[SIZE] = { 10, 4, 6, 34, 32, -34, 39, 21, 9, 22 };
    int i, max, min;

    min = max = a[0];

    for (i = 1; i < SIZE; ++i)
        if (a[i] > max)
            max = a[i];
        else if (a[i] < min)
            min = a[i];

    printf("Degisim Araligi = %d\n", max - min);

    return 0;
}
```

Dizilere İlişkin Çeşitli Algoritmik Örnekler

Bir dizinin eleman toplamı ve ortalaması şöyle bulunabilir:

```
#include <stdio.h>

#define SIZE      5

int main(void)
{
    int a[SIZE];
    int i, total;
    double avg;

    for (i = 0; i < SIZE; ++i) {
        printf("%d. indeksli elemani giriniz:", i);
        scanf("%d", &a[i]);
    }

    total = 0;
    for (i = 0; i < SIZE; ++i)
        total += a[i];

    avg = (double)total / SIZE;
    printf("%f\n", avg);

    return 0;
}
```

Sınıf Çalışması: int türden bir dizi içerisindeki en fazla yinelenen değeri bulunuz (mod değeri). Test için aşağıdaki diziyi kullanınız:

```
int a[SIZE] = { 3, 7, 3, 6, 6, 8, 7, 8, 6, 4 };
```


İpucu: İç içe iki döngü kullanılabilir. Her değer kaç tane tekrarlandığı bulunur ve bir değişkende saklanır, daha fazlası varsa yer değiştirilir. (Dizinin k'ncü elemanından kaç tane olduğuna bakmak için baştan başlamaya gerek yoktur. k'ncü indeksten başlamak yeterlidir. Çünkü zaten k'ncü indeksten önce o elemandan varsa biz onun sayısını bulmuş durumda oluruz).

Çözüm:

```
#include <stdio.h>

#define SIZE      10

int main(void)
{
    int a[SIZE] = { 3, 7, 3, 6, 6, 8, 7, 8, 6, 4 };
    int i, k;
    int count, max_count, max_count_val;

    max_count = 0;
    for (i = 0; i < SIZE; ++i) {
        count = 1;
        for (k = i + 1; k < SIZE; ++k)
            if (a[i] == a[k])
                ++count;
        if (count > max_count) {
            max_count = count;
            max_count_val = a[i];
        }
    }
    printf("Mod = %d\n", max_count_val);

    return 0;
}
```

Sınıf Çalışması: int türden bir diziyi ters yüz ediniz ve elemanları yazdırınız

Çözüm:

```
#include <stdio.h>

#define SIZE      10

int main(void)
{
    int a[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int i, temp;

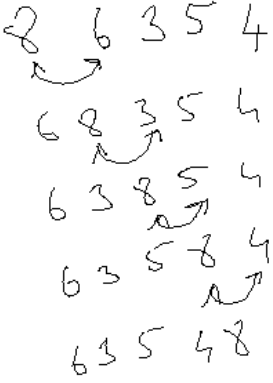
    for (i = 0; i < SIZE / 2; ++i) {
        temp = a[SIZE - i - 1];
        a[SIZE - i - 1] = a[i];
        a[i] = temp;
    }

    for (i = 0; i < SIZE; ++i)
        printf("%d ", a[i]);
    printf("\n");

    return 0;
}
```

Dizilerin sıraya dizilmesine İngilizce "sorting" denilmektedir ve 20'den fazla sıraya dizme yöntemi vardır. En basit yöntemlerden biri kabarcık sıralaması (bubble sort) yöntemidir. Bu yöntemde dizinin yan yana elemanları

karşılaştırılır, duruma göre yer değiştirilir. Tabi bu işlem bir kez yapılmaz. Bu işlem bir kez yapıldığında en büyük eleman sona gider. Örneğin:



Bu işlemi ikinci yaptığımızda sona kadar değil, bir önceye kadar gitmemiz yeterli olur. Böylece iç içe iki döngüyle algoritma gerçekleştirilebilir:

```
#include <stdio.h>

#define SIZE      10

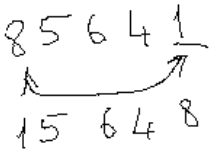
int main(void)
{
    int a[SIZE] = { 10, 23, -5, 34, 77, 100, 32, 87, 22, 44};
    int i, k;
    int temp;

    for (i = 0; i < SIZE - 1; ++i)
        for (k = 0; k < SIZE - 1 - i; ++k) {
            if (a[k] > a[k + 1]) {
                temp = a[k];
                a[k] = a[k + 1];
                a[k + 1] = temp;
            }
        }

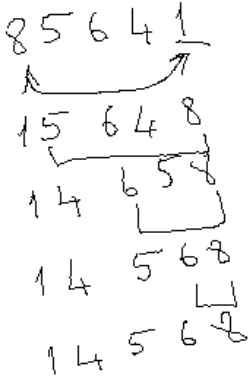
    for (i = 0; i < SIZE; ++i)
        printf("%d ", a[i]);
    printf("\n");

    return 0;
}
```

Seçerek Sıralama (Selection Sort) yönetminde dizinin en küçük elemanı bulunur, ilk elemanla yer değiştirilir. Örneğin:



Sonra dizi daraltılarak aynı işlem onun için de yapılır:



Algoritmada iki döngü bulunur. Dıştaki döngü diziyi daraltmakta kullanılır. İçteki döngü en küçük elemanı bulur:

```
#include <stdio.h>

#define SIZE      10

int main(void)
{
    int a[SIZE] = { 10, 23, -5, 34, 77, 100, 32, 87, 22, 44};
    int i, k;
    int min, minIndex;

    for (i = 0; i < SIZE - 1; ++i) {
        min = a[i];
        minIndex = i;
        for (k = i + 1; k < SIZE; ++k)
            if (a[k] < min) {
                min = a[k];
                minIndex = k;
            }
        a[minIndex] = a[i];
        a[i] = min;
    }

    for (i = 0; i < SIZE; ++i)
        printf("%d ", a[i]);
    printf("\n");

    return 0;
}
```

Yazıların char Türden Dizilerde Saklanması

Aslında yazı dediğimiz şey karakterlerden oluşan bir dizidir. Bilindiği gibi karakterler de aslında karakter tablosunda birer sayı belirtir. Bir yazının karakter numaralarını bir dizide saklarsak yazıyı saklamış oluruz. Programa dili ne olursa olsun yazılar arka planda hep böyle saklanmaktadır. Şüphesiz yazıları saklamak için en iyi tür char türüdür. char türü her sistemde 1 byte uzunluğundadır.

Genellikle yazılar char türden dizilerin başından başlarlar fakat onların sonuna kadar devam etmezler. Yani genellikle yazılar yerleştirildikleri diziden küçük olurlar. Bu durumda yazının sonunu belirlemek gerekir. İşte C'de bunun için null karakter kullanılmaktadır. null karakter '\0' ile temsil edilir. Karakter, tablonun sıfır numaralı karakteridir ve sayısal değeri de sıfırdır. null karakterin bir görüntü karşılığı yoktur, zaten tabloya böyle bir amaç için yerleştirilmiştir.

C'de bir char türden bir diziyeye yazı yerleştireceksek yazının sonuna (dizinin sonuna değil) '\0' karakteri koymalıyız. Çünkü C'de herkes tarafından yapılan böyle bir anlaşma vardır. Bu durumda n elemanlı charf türden bir diziyeye en

fazla n - 1 elemanlı bir yazı yerleştirilebilir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char s[10];
    int i;

    s[0] = 'a';
    s[1] = 'l';
    s[2] = 'i';
    s[3] = '\0';

    for (i = 0; s[i] != '\0'; ++i)
        putchar(s[i]);
    putchar('\n');

    return 0;
}
```

Aynı işlem ilşkedeğer verilerek de yapılabilirdi:

```
#include <stdio.h>

int main(void)
{
    char s[10] = { 'a', 'l', 'i', '\0' };
    int i;

    for (i = 0; s[i] != '\0'; ++i)
        putchar(s[i]);
    putchar('\n');

    return 0;
}
```

Burada diziye ilk değer verildiği için derleyici gerei kalan elemanları sıfırlar. Null karakter zaten sıfır olduğu için aşağıdaki iki bildirim eşdeğer olur:

```
char s[10] = { 'a', 'l', 'i', '\0' };
char s[10] = { 'a', 'l', 'i'};
```

char türden bir dizide null karakter görene kadar ilerlemek için şu kalıp kullanılabilir:

```
for (i = 0; s[i] != '\0'; ++i) {
    ...
}
```

gets Fonksiyonu

gets standart bir C fonksiyonudur. Aşağıdaki gibi kullanılır:

```
gets(<dizi ismi>);
```

gets fonksiyonu kalvyeden bir yazı girilip ENTER tuşuna basılana kadar bekler. Girilen yazının karakterlerini tek tek verdiğimiz char türden diziye yerleştirir, yazının sonuna '\0' karakterini de ekler. gets dizinin diğer elemanlarına hiç dokunmaz.

Örneğin:

```
#include <stdio.h>

int main(void)
{
    char s[100];
    int i;

    gets(s);

    for (i = 0; s[i] != '\0'; ++i)
        putchar(s[i]);
    putchar('\n');

    return 0;
}
```

puts Fonksiyonu

puts fonksiyonu bir dizinin ismini (yani adresini) parametre olarak alır '\0' görene kadar tüm karakterleri yan yana basar, en sonunda imleci aşağı satırın başına geçirerek orada bırakır. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char s[100];
    int i;

    gets(s);
    puts(s);

    return 0;
}
```

Sınıf Çalışması: Klavyeden gets fonksiyonuyla bir diziye bir yazı giriniz. Girilen yazının kaç karakterli olduğunu (yani yazının uzunluğunu) yazdırınız.

Çözüm:

```
#include <stdio.h>

int main(void)
{
    char s[100];
    int i;

    printf("Yazi giriniz:");
    gets(s);

    for (i = 0; s[i] != '\0'; ++i)
        ;
    printf("%d\n", i);

    return 0;
}
```

Sınıf Çalışması: Klavyeden gets fonksiyonuyla bir diziye bir yazı giriniz. Girilen yazıyı tersten yazdırınız.

Çözüm:

```

#include <stdio.h>

int main(void)
{
    char s[100];
    int i;

    printf("Yazi giriniz:");
    gets(s);

    for (i = 0; s[i] != '\0'; ++i)
        ;
    for (--i; i >= 0; --i)
        putchar(s[i]);
    putchar('\n');

    return 0;
}

```

Bir dizideki yazıyı büyük harfe dönüştüren örnek:

```

#include <stdio.h>
#include <ctype.h>

int main(void)
{
    char s[100];
    int i;

    printf("Yazi giriniz:");
    gets(s);

    for (i = 0; s[i] != '\0'; ++i)
        s[i] = toupper(s[i]);
    puts(s);

    return 0;
}

```

Sınıf Çalışması: char türden bir dizi ve bir de char türden değişken bildiriniz. gets fonksiyonu ile klavyeden diziye okuma yapınız. Sonra da getchar fonksiyonu ile char nesneye okuma yapınız. Yazı içerisinde o karakterden kaç tane olduğunu yazdırınız.

Çözüm:

```

#include <stdio.h>

int main(void)
{
    char s[100];
    char ch;
    int i, count;

    printf("Yazi giriniz:");
    gets(s);
    printf("Bir karakter giriniz:");
    ch = getchar();

    for (count = 0, i = 0; s[i] != '\0'; ++i)
        if (s[i] == ch)
            ++count;

    printf("%d\n", count);
}

```

```
    return 0;
}
```

printf Fonksiyonu İle Yazıların Yazdırılması

printf fonksiyonunda %s format karakterine karşılık char türden bir dizi ismi (aslında adres) gelmelidir. Bu durumda printf null karakter görene kadar karakterleri yazar. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char s[100];

    printf("Yazi giriniz:");
    gets(s);

    printf("Girilen yazi: %s\n", s);

    return 0;
}
```

Yani:

```
puts(s);
```

ile,

```
printf("%s\n", s);
```

eşdeğerdir.

char Türden Dizilere İki Tırnak İfadesi ile İlkdeğer Vermek

C'de char türden bir diziye iki tırnak ile ilkdeğer verilebilir. Örneğin:

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

Burada yazının karakterleri tek tek diziye yerleştirilir, sonuna null karakter eklenir. Tabi dizinin geri kalan elemanları da yine sıfırlanacaktır. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char s[100] = "ankara";

    printf("%s\n", s);

    return 0;
}
```

İki tırnak ile ilkdeğer verilirken dizi uzunluğu yine belirtilmeyebilir. Örneğin:

```
char s[] = "ankara";
```

Burada derleyici null karakteri de hesaba katar. Yani dizi 7 uzunlukta açılmış olacaktır. Aşağıdaki ilkdeğer verme geçersizdir:

```
char s[4] = "ankara";    /* geçersiz! */
```

Burada verilen ilkdeğerler dizi uzunluğundan fazladır. Aşağıdaki durum bir istisnadır:

```
char s[6] = "ankara";    /* geçerli, fakat null karakter eklenmeyecek */
```

Standartlara göre iki tırnak içerisindeki karakterlerin sayısı tam olarak dizi uzunluğu kadarsa bu durum geçerlidir. Ancak '\0' karakter derleyici tarafından bu istisnai durumda diziye eklenmez.

İki tırnağın içi boş olabilir. Bu durumda yalnızca null karakter ataması yapılır. Örneğin:

```
char s[] = "";    /* geçerli */
```

Dizi tek elemanlıdır ve o elemanda da '\0' karakter vardır.

Standartlara göre '\0' karakteri kesinlikle hangi karakter dönüştürme tablosu kullanılıyor olursa olsun o tablonun sıfır numaralı karakteridir. Yani null karakterin sayısal değeri sıfırdır. Başka bir deyişle:

```
s[n] = 0;
```

ile

```
s[n] = '\0';
```

tamamen aynı etkiyi yapar. Eğer char dizisi yazısal amaçlı kullanılıyorsa '\0' kullanmak iyi bir tekniktir.

char türden diziye iki tırnakla daha sonra değer atayamazız. Yalnızca ilkdeğer olarak bunu yapabiliriz. Örneğin:

```
char s[10];
```

```
s = "ankara";    /* geçersiz! */
```

Koşul Operatörü (Conditional Operator)

Koşul operatörü C'nin üç operandlı (ternary) tek operatörüdür. Operatör "? :" temsil edilir. Genel kullanımı şöyledir:

```
op1 ? op2 : op3
```

Burada op1, op2 ve op3 birer ifadedir. Koşul operatörü if deyimi gibi çalışan bir operatördür. Operatörün çalışması şöyledir: Önce soru işaretinin solundaki ifade (op1) yapılır. Bu ifade sıfır dışı bir değerse yalnızca soru işareti ile iki nokta üstü üste arasındaki ifade (op2) yapılır, sıfırsa iki nokta üst üstenin sağındaki ifade yapılır. Koşul operatörü bir operatör olduğu için bir değer üretir. Yani bu değer bir nesneye atanabilir, başka işlemlere sokulabilir. Koşul operatörünün ürettiği değer duruma göre op2 ya da op3 ifadelerinin değeridir. Örneğin:

```
result = val > 0 ? 100 + 200 : 300 + 400;
```

Burada val > 0 ise yalnızca 100 + 200 ifadesi yapılır ve koşul operatöründen bu değer elde edilir. Değilse 300 + 400 yapılır ve koşul operatöründen bu değer elde edilir. Burada yapılan aşağıdaki ile eşdeğerdir:

```
if (val > 0)
    result = 100 + 200;
else
    result = 300 + 400;
```


Örneğin:

```
#include <stdio.h>

int main(void)
{
    int val;
    int result;

    printf("Bir sayi giriniz:");
    scanf("%d", &val);

    result = val > 0 ? 100 + 200 : 300 + 400;
    printf("%d\n", result);

    return 0;
}
```

Koşul operatörü elde edilen değerin bir nesneye atanacağı ya da başka bir ifadede kullanılacağı durumlarda tercih edilmelidir. Aşağıdaki örnek koşul operatörünün kötü bir kullanımına işaret eder:

```
a > 0 ? foo() : bar();
```

Burada koşul operatörünün kullanılması kötü bir tekniktir. Koşul operatörünün üç durumda kullanılması tavsiye edilir:

1) Bir karşılaştırma sonucunda elde edilen değerin bir nesneye atanması durumunda. Örneğin:

```
max = a > b ? a : b;
```

Örneğin:

```
for (i = 1900; i < year; ++i)
    tdays += isleap(i) ? 366 : 365;
```

Burada isleap(i) sıfır dışı ise tdays'e 366 değilse 365 eklenmektedir. Aşağıda belli bir tarihin hangi güne karşı geldiğini bulan programı görüyorsunuz:

```
#include <stdio.h>

int isleap(int year)
{
    return year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
}

int totaldays(int day, int month, int year)
{
    int i;
    long tdays = 0;
    int montab[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    for (i = 1900; i < year; ++i)
        tdays += isleap(i) ? 366 : 365;

    montab[1] = isleap(year) ? 29 : 28;

    for (i = 0; i < month - 1; ++i)
        tdays += montab[i];

    tdays += day;
```

```

    return tdays;
}

void printday(int day, int month, int year)
{
    long tdays;

    tdays = totaldays(day, month, year);

    switch (tdays % 7) {
        case 0:
            printf("Pazar\n");
            break;
        case 1:
            printf("Pazartesi\n");
            break;
        case 2:
            printf("Salı\n");
            break;
        case 3:
            printf("Carsamba\n");
            break;
        case 4:
            printf("Persembe\n");
            break;
        case 5:
            printf("Cuma\n");
            break;
        case 6:
            printf("Cumartesi\n");
            break;
    }
}

int main(void)
{
    printday(10, 9, 1993);

    return 0;
}

```

2) Koşul operatörü fonksiyon çağrılarında argüman olarak kullanılabilir. Böylece elde edilen değer fonksiyonun parametre değişkenine atanmış olur. Örneğin:

```
foo(val % 2 == 0 ? 100 : 200);
```

Bunun if eşdeğeri şöyledir:

```

if (val % 2 == 0)
    foo(100);
else
    foo(200);

```

Örneğin:

```

#include <stdio.h>

int main(void)
{
    int val;

    printf("Bir sayı giriniz:");

```

```

scanf("%d", &val);

printf(val % 2 == 0 ? "Cift\n" : "Tek\n");

return 0;
}

```

3) Koşul operatörü return deyiminde de kullanılabilir. Örneğin:

```
return a % 2 == 0 ? 100 : 200;
```

Bu ifadenin eşdeğer if karşılığı şöyledir:

```

if (a % 2 == 0)
    return 100;
else
    return 200;

```

Koşul operatörü özel bir operatördür. Öncelik tablosundaki klasik kurallara tam uymaz. Ancak öncelik tablosunda hemen atama ve işlemler atama operatörlerinin yukarısında sağdan-sola grupta bulunur:

()	Soldan-Sağa
+ - ++ -- ! (tür)	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
< > <= >=	Soldan-Sağa
== !=	Soldan-Sağa
&&	Soldan-Sağa
	Soldan-Sağa
? :	Sağdan-Sola
= += -= *= /= %=, ...	Sağdan-Sola
,	Soldan-Sağa

Koşul operatörünün operandları şöyle ayrıştırılır: Soru işaretinden sola doğru gidilir. Koşul operatöründen daha düşük öncelikli operatör görülene kadarki ifade koşul operatörünün birinci operandını oluşturmaktadır. Soru işareti ile iki nokta üst üste arasındaki ifade koşul operatörünün ikinci operandını oluşturur. İki nokta üst üstenin sağındaki ifade koşul operatörünün üçüncü operandını oluşturmaktadır. Örneğin:

*a = (b * c > 10) ? 100 : 200;*

Op1 Op2 Op3

Parantezler koşul operatörünün operandlarını ayrıştırmakta kullanılabilir. Örneğin:

```
a = (b > 0 ? 100 + 200 : 300) + 400;
```

Burada artık üç operatör vardır: Atama, koşul ve +. Yani + 400 koşul operatörünün dışındadır. b > 0 koşulu sağlansa da sağlanmasa da 400 ile toplama yapılacaktır.

Koşul operatörü de iç içe kullanılabilir. Örneğin:

```
result = a > b ? a > c ? a : c : b > c ? b : c;
```

Burada üç sayının en büyüğü bulunmaktadır. Tabii bu ifade karmaşık gözüktüğü için parantezler gerekmiyor olsa da sadelik oluşturmak için kullanılabilir. Örneğin:

```
result = a > b ? (a > c ? a : c) : (b > c ? b : c);
```

sizeof Operatörü

sizeof bir derleme zamanı (compile time) operatörüdür. Parantezlerinden dolayı acemi C programcılarını onu fonksiyon sanabilmektedir. Operatörler derleme aşamasında derleyici tarafından kod üretilmesi için ele alınırlar. Oysa fonksiyonlar programın çalışma zamanı sırasında akışın oraya gitmesiyle işlem görürler.

sizeof operatörünün kullanım biçimleri şöyledir:

```
sizeof <ifade>  
sizeof(<tür>)  
sizeof <dizi ismi>
```

sizeof operatörünün operandı bir ifade ise, derleyici o ifadeyi yapmaz. Yalnızca o ifadenin türüne bakar. sizeof ifadenin türünün o sistemde kaç byte yer kapladığını belirtir. Derleyici sizeof operatörünü gördüğünde derleme aşamasında oraya sabit yerleştirir. Yani sanki sizeof yerine biz bir sabit yazmışız gibi işlem görür. sizeof operatörü tek operandlı önek bir operatördür ve öncelik tablosunun ikinci düzeyinde sağdan sola grupta bulunur:

()	Soldan-Sağa
+ - ++ -- ! sizeof	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
...	...

Dolayısıyla,

```
sizeof a + b
```

ifadesinde a'nın sizeof'u b ile toplanır. Fakat:

```
sizeof (a + b)
```

ifadesinde a + b'nin sizeof'u elde edilmektedir. Programcılar genellikle sizeof operatörünü hep parantezlerle kullanmaktadır. Örneğin:

```
#include <stdio.h>  
  
int main(void)  
{  
    int n;  
  
    n = sizeof 1.2 + 5;  
  
    printf("%d\n", n);    /* 13 */  
}
```

```
n = sizeof (1.2 + 5); /* 8 */
return 0;
}
```

sizeof operatörünü gören derleyici onun yerine o ifadenin türünün kaç byte olduğuna ilişkin sabit bir değer yerleştirir. Yerleştiği sabit size_t türündendir (tipik olarak unsigned int).

```
n = sizeof 1.2 + 5;
```

ifadesi derleyici tarafından şu biçime dönüştürülür:

```
n = 8u + 5;
```

Şüphesiz sizeof operatörünün operandı fonksiyon çağrısıysa fonksiyon işletilmez, onun geri dönüş değerinin türüne bakılır, o değer elde edilir:

```
#include <stdio.h>

double foo(void)
{
    printf("Im am foo\n");

    return 1500.2;
}

int main(void)
{
    int n;

    n = sizeof(foo());
    printf("%d\n", n);

    return 0;
}
```

Burada ekrana foo yazısı basılmaz, yalnızca 8 yazısı basılır.

void değerinin sizeof'u yoktur. Dolayısıyla sizeof operatörünün operandı void türden olamaz.

sizeof operatörünün operandı doğrudan bir tür ismi olabilir. Bu durumda parantezler kullanılmak zorundadır. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int n;

    n = sizeof(int);
    printf("%d\n", n); /* 4 */

    return 0;
}
```

sizeof operatörünün operandı bir dizi ismi olabilir. Bu durumda sizeof o dizinin bellekte kapladığı byte miktarını bize verir. Örneğin:

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

```

int main(void)
{
    int a[4] = { 1, 2 };
    char s[] = "ali";

    printf("%u\n", sizeof a);    /* 16 */
    printf("%u\n", sizeof s);    /* 4 */

    return 0;
}

```

Peki size of operatörüne neden gereksinim duyulmaktadır? Bilindiği gibi C'de char dışındaki tüm türlerin uzunlukları derleyiciye bağlı olarak değişebilmektedir. Kodun taşınabilirliğini sağlamak için bazı durumlarda size of operatörünü kullanmak zorunda kalabiliriz. size of operatörüne olan gereksinim ileriki konularda açıkça görülecektir.

Göstericiler (Pointers)

Adres bilgilerinin saklanması için kullanılan nesnelere gösterici (pointer) denir. Bir adres bilgisi C'de int, long türlerde tutulamaz. Ancak gösterici denilen türlerde tutulabilir. Benzer biçimde göstericiler de adi birer int, long türleri tutamazlar. Ancak adres tutarlar.

Gösterici bildiriminin genel biçimi şöyledir:

```
<tür> *<gösterici ismi>;
```

Örneğin:

```

int *p;
long *t;
double *d;

```

Buradaki * adres türünü temsil eder. Çarpma gibi bir anlama gelmez.

Bir gösterici her türden adres bilgisini tutamaz. Yalnızca tür bileşeni belirli olan adres bilgilerini tutar. Genel olarak:

```
T *p;
```

gibi bir bildirimde p göstericisi tür bileşeni T olan adresleri tutabilir. Örneğin:

```
int*pi;
```

Burada pi int türünden göstericidir. int türden gösterici demek, tür bileşeni int olan adresleri tutan gösterici demektir.

Anahtar Notlar: İngilizce T türünden gösterici "pointer to T" biçiminde yazılıp okunmaktadır.

Gösterici bildiriminde * farklı bir atomdur. Dolayısıyla öncesinde sonrasında birden fazla boşluk karakteri bulunabilir. Örneğin:

```
int * pi; /* geçerli */
```

Bazı programcılar * atomunu türe yapıştırmaktadır:

```
int* pi; /* geçerli */
```

Fakat gelenek (zaten daha mantıksal olduğu için) * atomunun gösterici ismine yapıştırılmasıdır:

```
int *pi;
```

* atomu deklaratora ilişkindir, türe ilişkin değildir. Örneğin aşağıdaki bildirim geçerlidir:

```
int *pi, a;
```

Burada pi int türden bir göstericidir fakat a, adi bir int nesnedir. Örneğin:

```
int *pi, *a;
```

Burada hem pi hem de a int türden göstericidir:

```
int a, b[10], *c;
```

Burada a adi bir int, b 10 elemanlı bir int dizi, c ise int türden bir göstericidir.

Bir göstericiye aynı türden bir adres bilgisi yerleştirilir:

```
int *pi;
```

```
pi = (long *) 0x1FC0; /* geçersiz! tür yanlış*/  
pi = 0x1FC0; /* geçersiz! adi bir int */
```

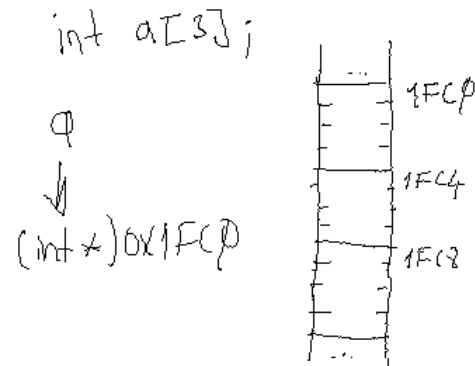
Benzer biçimde biz nümerik türlere adres bilgisi atayamazız. Örneğin:

```
int a;
```

```
a = (int *) 0x1FC0; /* geçersiz! */
```

Dizi İsimlerinin Otomatik Adreslere Dönüştürülmesi

C'de bir dizinin ismi dizinin tamamını temsil eder. Dizi isimleri işleme sokulduğunda otomatik olarak derleyici tarafından adrese dönüştürülür. Yani dizi isimleri aslında adres belirtmektedir. Dizi isimleriyle belirtilen adresin tür bileşeni dizinin aynı türden, sayısal bileşeni dizinin bellekteki başlangıcına ilişkin fiziksel adres numarasından oluşur:



Burada a ismini işleme soktuğumuzda otomatik olarak bu dizinin başlangıç adresine ilişkin adres sabiti gibi işleme girer.

Medemki dizi isimleri adres belirtmektedir, o halde bir dizi ismini biz aynı türden bir göstericiye atayabiliriz. Örneğin:

```
int *pi;
int a[3];

pi = a;    /* geçerli */
```

Örneğin:

```
int *pi;
char s[10];

pi = s;    /* geçersiz! s char türden adres belirtir */
```

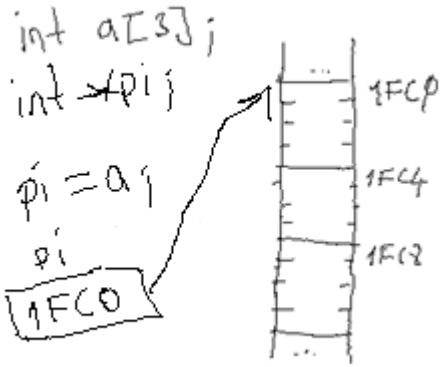
Örneğin:

```
int a[10];
int b;

b = a;    /* Geçersiz! a bir adres bilgisidir ve göstericiye yerleştirilebilir */
```

Bir göstericiye bir adres bilgisi atandığında aslında gösterici yalnızca adresin sayısal bileşenini tutar. (Tabi bu durum göstericiye adi bir sayı atılabileceğimiz anlamına gelmiyor).

Bir göstericiye bir adres adres bilgisi atandığında o göstericinin o adresi gösterdiği söylenir. Örneğin:



Burada pi göstericisi a dizisinin başlangıç adresini göstermektedir.

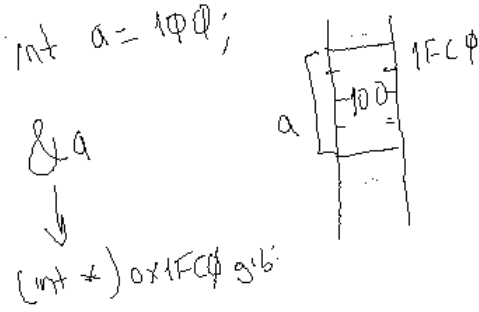
& (Address Of) Operatörü

& sembolü C'de "address of" isimli bir adres operatörü belirtir. & tek operandlı önek bir operatördür. Öncelik tablosunun ikinci düzeyinde sağdan sola grupta bulunmaktadır:

()	Soldan-Sağa
+ - ++ -- ! sizeof &	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
...	...

& operatörü operand olarak bir nesne alır. Operatör bize nesnenin bellekteki yerleşim adresini verir. & operatörü ile çekilen adresin sayısal bileşeni operand olan nesnenin bellekteki fiziksel adres numarasıdır. Tür bileşeni ise

nesnenin türüyle aynı türdendir. Örneğin:



Dizi isimleri zaten adres belirtmektedir. Bu nedenle yanlışlıkla dizi isimlerine & uygulamamak gerekir. Fakat adi nesnelerin adreslerini almak için & operatörü kullanmak gerekir.

& operatörüyle elde ettiğimiz adresler ancak aynı türde bir göstericiye atanabilir.

```
int a;  
int *pi;  
char *pc;
```

```
pi = &a;      /* geçerli */  
pc = &a;      /* geçersiz! */
```

& operatörünün operandı bir nesne olabilir. Nesne belirtmeyen bir ifade olamaz. Çünkü yalnızca nesnelerin adresleri vardır. Örneğin:

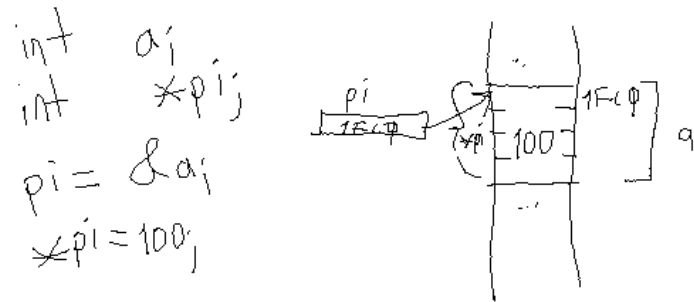
```
#define MAX      100
```

```
int *pi;
```

```
pi = &100;    /* geçersiz! */  
pi = &MAX;    /* geçersiz! */
```

* (Indirection) Operatörü

* tek operandlı örnek bir adres operatörüdür. * operatörünün operandı bir adres bilgisi olmak zorundadır. * operatörü operandı olan adresteki nesneye erişmekte kullanılır. * operatörüyle elde edilen nesnenin türü operanda olan adresin türüyle aynı türdendir. Örneğin:



pi göstericini kullandığımızda onun içerisindeki adres bilgisini kullanıyoruz. Ancak *pi ifadesi pi adresindeki nesneyi belirtir. Bu da şekilde görüldüğü gibi a'dır. Burada *pi ifadesi int türdendir. Çünkü pi adresi int türdendir. Örneğin:

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

```

int main(void)
{
    int a = 100;
    int *pi;

    pi = &a;
    printf("%d\n", *pi); /* 100 */
    *pi = 150;
    printf("%d\n", a); /* 150 */

    return 0;
}

```

Örneğin:

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

```

int main(void)
{
    char ch;
    char *pc;

    pc = &ch;
    *pc = 'x';
    putchar(ch);

    return 0;
}

```

Örneğin:

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

```

int main(void)
{
    int a[3] = { 10, 20, 30 };
    int *pi;

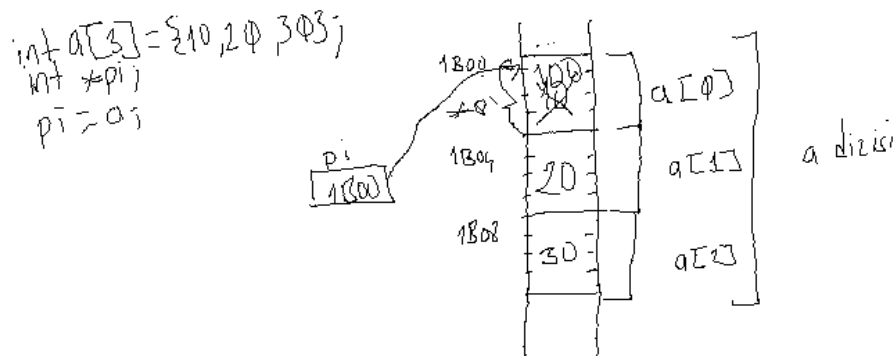
    pi = a;
    *pi = 100;

    printf("%d\n", a[0]);

    return 0;
}

```

Buradada yapılan işlemleri şekilsel olarak şöyle ifade edilebilir:



p bir adres belirtiyor olsun. Aşağıdaki gibi bir atamadan bellekteki kaç byte etkilenir:

*p = 0;

İşte p eğer char türdense p adresinden itibaren 1 byte etkilenir. p int türdense 4 byte, double türdense 8 byte etkilenecektir.

Aşağıdaki gibi bir gösterici bildirim yapılmış olsun:

```
int *pi;
```

Bu aradan iki şey anlaşılır:

1) Burada pi int * türündendir. (pi'yi kapatıp sola baktığımızda int * görüyoruz).



2) pi göstericisini * operatörü ile kullanırsak int bir nesne elde ederiz (*pi'yi kapatıp sola baktığımızda int görüyoruz).



C'de adres türleri sembolik olarak T * ile ifade edilir. Örneğin:

```
char *pc;
```

Burada pc'nin türü char * biçimindedir. *pc ise char türündendir.

* operatörü öncelik tablosunun ikinci düzeyinde sağdan sola grupta bulunur:

()	Soldan-Sağa
+ - ++ -- ! sizeof & *	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
...	...

Örneğin:

```
a = *pi * 50;
```

Burada:

İ1: *pi

İ2: İ1 * 50

İ3: a = İ2

* operatörünün operandı gösterici olmak zorunda değildir. Adres belirten herhangi bir ifade olabilir. Yani * operatörünün operandı şunlar olabilir.

1) Bir gösterici (*pi gibi)

2) Bir adres sabiti. Örneğin:

```
*(int *) 0x1FC0 = 100;
```

3) & operatörü elde edilmiş bir adres (*&a gibi)

4) Bir dizi ismi olabilir. Örneğin:

```
int a[] = {1, 2, 3};  
*a = 100; /* geçerli */
```

5) String ifadeleri (henüz görmedik)

& ve * operatörlerinin sağdan sola aynı grupta olduğuna dikkat ediniz. Örneğin:

```
int a = 100;  
  
*&a = 200;  
printf("%d\n", a);
```

Burada:

```
İ1: &a  
İ2: *İ1  
İ3: İ2 = 200
```

Örneğin:

```
#include <stdio.h>  
  
int main(void)  
{  
    int a = 100;  
  
    *&a = 200;  
    printf("%d\n", a);  
  
    return 0;  
}
```

Örneğin:

```
#include <stdio.h>  
  
int main(void)  
{  
    int a[3] = { 10, 20, 30 };  
    int i;  
  
    *a = 100;  
    for (i = 0; i < 3; ++i)  
        printf("%d\n", a[i]);  
  
    return 0;  
}
```

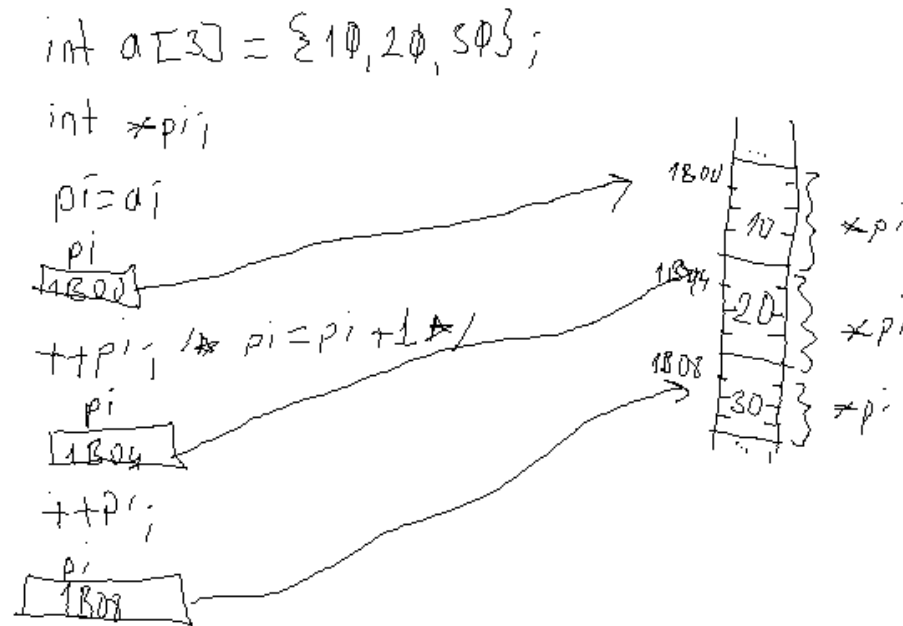
Adreslerin Artırılması ve Eksiltilmesi

Adres bilgileri tamsayı türlerine ilişkin değerlerle toplama ve çıkartma işlemine sokulabilir. Yani, p bir adres bilgisi n de bir tamsayı türünden olmak üzere $p + n$ ve $p - n$ ifadeleri geçerlidir (Ancak $n - p$ ifadesi geçerli değildir). Bir adres bilgisine bir tamsayıyı topladığımızda ya da bir adres bilgisinden bir tamsayı çıkarttığımızda elde edilen ürün yine aynı türden bir adres bilgisi olur. Örneğin p_i int türünden bir adres bilgisi (int^*) olsun:

$p_i + 1$

ifadesi int türden bir adres bilgisi belirtir. C'de bir adres bilgisi 1 artırıldığında ya da 1 eksiltildiğinde adresin sayısal bileşeni o adresin türünün uzunluğu kadar artar ya da eksilir. Yani örneğin int türden bir göstericiyi 1 artırdığımızda içindeki adres 32 bit sistemlerde 4 artacaktır. char türden bir gösterici bir artırdığımızda içindeki adres 1 artacaktır.

Bu durumda dizi elemanları ardışıl olduğuna göre biz dizinin başlangıç adresini aynı türden bir göstericiye yerleştirip sonra o göstericiyi artırarak dizinin tüm elemanlarına erişebiliriz:



Örneğin:

```
#include <stdio.h>

int main(void)
{
    int a[3] = { 10, 20, 30 };
    int *pi;
    int i;

    pi = a;

    printf("%d\n", *pi);      /* 10 */
    ++pi;
    printf("%d\n", *pi);      /* 20 */
    ++pi;
    printf("%d\n", *pi);      /* 30 */

    return 0;
}
```

* operatörünün operandı adres türünden olmak zorudur. Örneğin:

```
int a = 0x1Fc0;
*a = 10;      /* geçersiz! */
```

char türden bir dizinin adresini char türden bir göstericiye atadıktan sonra göstericiyi artıra artıra yazının tüm karakterlerine erişebiliriz. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char s[] = "ankara";
    char *pc;

    pc = s;

    while (*pc != '\0') {
        putchar(*pc);
        ++pc;
    }
    putchar('\n');

    return 0;
}
```

Köşeli parantez (Index) Operatörü

Köşeli parantez operatörü tek operandlı sonek bir operatördür. Köşeli parantezler içerisinde tamsayı türlerine ilişkin bir ifade bulunmak zorundadır. Köşeli parantez operatörü öncelik tablosunun tepe grubunda (..) operatörü ile soldan sağa aynı gruptadır:

() []	Soldan-Sağa
+ - ++ -- ! sizeof & *	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
...	...

Köşeli parantez operatörünün operandı bir adres bilgisi olmak zorundadır. p bir adres belirten ifade olmak üzere:

p[n] ile *(p + n) aynı anlamdadır. p[n] ifadesi p adresinden n ilerideki nesneyi temsil eder.

Dizi isimleri dizilerin başlangıç adresini belirttiği için dizi elemanlarına köşeli parantez operatörü ile erişilebilmektedir. Örneğin:

```
int a[3] = {10, 20, 30};
```

a[2] ifadesi a adresinden 2 ilerinin (yani 2 * sizeof(*a) kadar byte ilerinin) içeriği anlamına gelir. Yani a[2] ile *(a + 2) aynı anlamdadır.

Peki p[0] ne anlama gelir? Bu ifade p adresinden 0 ilerinin içeriği anlamına gelir. *(p + 0) ile ve dolayısıyla *p ile aynı anlamdadır.

Biz köşeli parantez operatörünü göstericilerle, dizi isimleriyle ve diğer adres belirten ifadelerle kullanabiliriz. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char s[] = "ankara";
    char *pc;
    int i;

    pc = s;

    for (i = 0; pc[i] != '\0'; ++i)
        putchar(pc[i]);
    putchar('\n');

    return 0;
}
```

Köşeli parantez içerisindeki ifade negatif bir değer de belirtebilir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int a[3] = { 10, 20, 30 };
    int *pi;

    pi = a + 2;
    printf("%d\n", pi[-2]);    /* 10 */

    return 0;
}
```

Göstericilere İlkdeğer Verilmesi

Göstericilere de bildirim sırasında '=' atomu ile ilkdeğer verebiliriz. Örneğin:

```
int *pi = (int *)0x1FC0;
```

Burada adres p'ye atanmaktadır. Bildirimdeki yıldız operatör görevinde değildir. Tür belirtmek için kullanılmaktadır. Başka bir örnek:

```
int a = 123;
int *pi = &a;
```

Örneğin:

```
#include <stdio.h>

int main(void)
{
    int a[3] = { 10, 20, 30 };
    int *pi = a;    // p'ye dizinin başlangıç adresi atanıyor
    int i;

    for (i = 0; i < 3; ++i)
        printf("%d ", pi[i]);
    printf("\n");

    return 0;
}
```

Fonksiyon Parametresi Olarak Göstericilerin Kullanılması

Bir fonksiyonun parametresi bir gösterici olabilir. Örneğin:

```
void foo(int *pi)
{
    ...
}
```

Bir fonksiyonun parametresi bir gösterici ise biz o fonksiyonu aynı türden bir adres bilgisi ile çağırmalıyız. Örneğin:

```
int a = 10;
...
foo(a);      // geçersiz!
foo(&a);     // geçerli
```

Örneğin:

```
#include <stdio.h>

void foo(int *pi)
{
    *pi = 200;
}

int main(void)
{
    int a = 123;

    foo(&a);
    printf("%d\n", a);

    return 0;
}
```

Burada foo fonksiyonuna a'nın içindeki değer değil, a'nın adresi geçirilmiştir. Böylece fonksiyon içerisinde *pi dediğimizde aslında bu ifade main fonksiyonundaki a'ya erişir.



C'de bir fonksiyonun başka bir fonksiyonun yerel değişkenini değiştirebilmesi için onun adresini alması gerekir. Bunun için de fonksiyonun parametre değişkeninin gösterici olması gerekir. (Aslında tüm dillerde böyledir. Çünkü makinanın çalışma prensibinde durum böyledir.)

İki yerel nesnenin içerisindeki değerleri değiştiren swap isimli bir fonksiyon yazmak isteyelim. Acaba bu fonksiyonu aşağıdaki gibi yazabilir miyiz?


```

#include <stdio.h>

void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main(void)
{
    int x = 10, y = 20;

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

    return 0;
}

```

Yanıt hayır! Çünkü burada swap x ve y'nin içindekilerini değiştirmiyor, parametre değişkeni olan a ve b'nin içindekileri değiştiriyor. Bunun sağlanabilmesi için yerel nesnelerin adreslerinin swap fonksiyonuna aktarılması gerekir. Bu durumda swap fonksiyonunun parametre değişkenleri birer gösterici olmalıdır:

```

#include <stdio.h>

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(void)
{
    int x = 10, y = 20;

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

    return 0;
}

```

Dizilerin Fonksiyonlara Parametre Yoluyla Aktarılması

Dizi elemanları bellekte ardışıl bir biçimde tutulduğuna göre biz bir diziyi yalnızca onun başlangıç adresi yoluyla fonksiyona aktarabiliriz. Böylece dizinin başlangıç adresini alan fonksiyon * ya da [] operatörü ile dizinin her elemanına erişebilir. Dizilerin fonksiyonlara parametre yoluyla aktarılmasında mecburen göstericilerden faydalanılmaktadır.

```

#include <stdio.h>

void foo(int *pi)
{
    int k;

    for (k = 0; k < 10; ++k) {
        printf("%d ", *pi);
        ++pi;
    }
    printf("\n");
}

```

```

void bar(int *pi)
{
    int k;

    for (k = 0; k < 10; ++k)
        printf("%d ", pi[k]);
    printf("\n");
}

int main(void)
{
    int a[10] = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };

    foo(a);
    bar(a);

    return 0;
}

```

Yukarıdaki fonksiyonlar dizinin uzunluğunun 10 olduğunu biliyor durumdadır. Biz bu fonksiyonlara ancak 10 elemanlı dizilerin başlangıç adreslerini yollayabiliriz. Eğer dizilerle çalışan fonksiyonların daha genel olmasını istiyorsak o fonksiyonlara ayrıca dizinin uzunluğunu geçirmemiz gerekir. Örneğin:

```

#include <stdio.h>

void foo(int *pi, int size)
{
    int k;

    for (k = 0; k < size; ++k) {
        printf("%d ", *pi);
        ++pi;
    }
    printf("\n");
}

void bar(int *pi, int size)
{
    int k;

    for (k = 0; k < size; ++k)
        printf("%d ", pi[k]);
    printf("\n");
}

int main(void)
{
    int a[10] = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };
    int b[5] = { 1, 2, 3, 4, 5 };

    foo(a, 10);
    bar(b, 5);

    return 0;
}

```

Örneğin:

```

#include <stdio.h>

int getmax(int *pi, int size)
{
    int max = pi[0];
    int i;

```

```

    for (i = 1; i < size; ++i)
        if (pi[i] > max)
            max = pi[i];

    return max;
}

int main(void)
{
    int a[5] = { 23, -4, 21, 34, 32 };
    int max;

    max = getmax(a, sizeof(a)/sizeof(*a));
    printf("%d\n", max);

    max = getmax(a, 3);
    printf("%d\n", max);

    return 0;
}

```

Burada getmax int bir dizinin en büyük elemanına geri dönmektedir.

double bir dizinin standart sapmasını hesaplayan fonksiyon şöyle yazılabilir:

```

#include <stdio.h>
#include <math.h>

double getavg(double *pd, int size)
{
    double total;
    int i;

    total = 0;
    for (i = 0; i < size; ++i)
        total += pd[i];

    return total / size;
}

double get_stddev(double *pd, int size)
{
    double avg, total;
    int i;

    avg = getavg(pd, size);

    total = 0;
    for (i = 0; i < size; ++i)
        total += pow(pd[i] - avg, 2);

    return sqrt(total / (size - 1));
}

int main(void)
{
    double a[5] = { 1, 1, 1, 1, 2 };
    double result;

    result = get_stddev(a, 5);
    printf("%f\n", result);

    return 0;
}

```

```
}
```

Örneğin bir diziyi kabarcık sıralaması (bubble sort) yöntemiyle sıraya dizen fonksiyon şöyle yazılabilir:

```
#include <stdio.h>

void bsort(int *pi, int size)
{
    int i, k;

    for (i = 0; i < size - 1; ++i)
        for (k = 0; k < size - 1 - i; ++k)
            if (pi[k] > pi[k + 1]) {
                int temp = pi[k];
                pi[k] = pi[k + 1];
                pi[k + 1] = temp;
            }
}

void disp_array(int *pi, int size)
{
    int i;

    for (i = 0; i < size; ++i)
        printf("%d ", pi[i]);
    printf("\n");
}

int main(void)
{
    int a[] = { 4, 8, 2, 21, 43, 10, -5, 87, 9, 68 };

    bsort(a, 10);
    disp_array(a, 10);

    return 0;
}
```

Sınıf Çalışması: 10 elemanlı int türden bir dizi tanımlayınız. Bu dizinin elemanlarını ters yüz eden aşağıdaki prototipe sahip fonksiyonu yazarak test ediniz:

```
void rev_array(int *pi, int size);
```

Çözüm:

```
#include <stdio.h>

void rev_array(int *pi, int size)
{
    int i, temp;

    for (i = 0; i < size / 2; ++i) {
        temp = pi[i];
        pi[i] = pi[size - i - 1];
        pi[size - i - 1] = temp;
    }
}

void disp_array(int *pi, int size)
{
    int i;

    for (i = 0; i < size; ++i)
```

```

        printf("%d ", pi[i]);
    printf("\n");
}

int main(void)
{
    int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    rev_array(a, 10);
    disp_array(a, 10);

    return 0;
}

```

Gösterici Parametrelerinin Alternatif Biçimi

C'de bir fonksiyonun parametre değişkeni bir gösterici ise bu alternatif olarak dizi sentaksıyla da belirtilebilmektedir. Bu alternatif sentaksta köşeli parantezlerin içi boş bırakılabilir. Fakat köşeli parantezlerin içerisine bir uzunluk yazılsa bile bunun hiçbir özel anlamı yoktur. Yani köşeli parantezlerin içinin boş bırakılmasıyla, onun içerisine bir uzunluk yazılması arasında bir fark yoktur. Örneğin:

```

void foo(int *pi)
{
    ...
}

```

bildirimi ile aşağıdaki tamamen eşdeğerdir:

```

void foo(int pi[])
{
    ...
}

```

bununla da aşağıdaki eşdeğerdir:

```

void foo(int pi[10])
{
    ...
}

```

Tabii bu eşdeğerlik yalnızca fonksiyon parametresi söz konusu olduğunda geçerlidir. Yani:

```

int pi[];        /* geçersiz! Dizi uzunluğu belirtilmemiş! */
int pi[10];     /* pi gösterici değil 10 elemanlı bir dizi */

```

Kaynak Kodun Okunabilirliği ve Anlaşılabilirliği

Kaynak kodun bakıldığında (ileride bizim tarafımızdan ya da başkaları tarafından) kolay anlaşılması önemlidir. Buna genel olarak okunabilirlik (readability) denilmektedir. Okunabilirliği sağlamak için bazı temel öğelere dikkat etmek gerekir:

1) Kaynak kodda bazı kritik noktalara açıklama (yorum) yerleştirilebilir. Örneğin:

```

if (delta >= 0) {        /* kök var mı? */
    ...
}

```

Çok fazla açıklama bilgisi de okunabilirliği azaltmaktadır. Uygun yerlere, çok uzun olmayan ve genellikle soru cümleleri biçiminde açıklama yapmak uygun olur.

2) Değişkenlere anlamlı ve telaffuz edilebilir isimler verilmelidir. İsimlendirme Türkçe olabilir. Ancak şu da unutulmamalıdır ki, maalesef uluslararası düzeyde herkes İngilizce isimlendirme beklemektedir. Fakat kodumuzu örneğin Internet ortamında uluslararası paylaymayacaksa isimlendirmeleri Türkçe yapabiliriz. Örneğin döngü değişkenleri için i, j, k, l, m, n gelenekseldir (Fortran zamanlarından gelme bir gelenek). Tabi bazen n, count gibi isimler kullanılabilir.

3) Global değişkenlerin özel bir önekle (örneğin g_xxx biçiminde) başlatılması uygun olabilir. Örneğin g_count gibi.

4) Değişken isimlendirmesinde harflendirme (capitalization) önemlidir. Harflendirme bir sözcük uzun olan isimlerin yazılış biçimlerini belirlemek anlamına gelir. Tipik olarak programda dillerinde üç harflendirme tarzı vardır:

a) Pascal Tarzı Harflendirme (Pascal casting): Burada her sözcüğün ilk harfi büyük yazılır (bu tarz Pascal dilinde çok kullanıldığı için Pascal tarzı denilmektedir). Örneğin:

```
CreateWindow  
GetWindowText  
NumberOfStudents  
...
```

b) Klasik C Tarzı (C Style): Bu tarzda isimlerde büyük harf kullanılmaz. Sözcükleri ayırmak için alt tire kullanılır. Örneğin:

```
create_window  
get_window_text  
number_of_students
```

Bu tarz UNIX/Linux sistemlerinde C programlama dilinde baskın olarak kullanılmaktadır.

c) Deve Notasyonu (Camel casting): Bu tarzda ilk sözcüğün tamamı küçük harflerle sonraki sözcüklerin yalnızca ilk harfleri büyük harflerle belirtilir. C++ QT framework'ünde, Java'da bu biçim tercih edilmektedir.

Biz kursumuzda global değişkenleri g_ ile başlatarak harflendireceğiz. Fonksiyon isimlerini Klasik C Tarzında, yerel ve parametre değişkenlerini deve notasyonunda harflendireceğiz.

Windows altında C programcıları fonksiyon isimlerini genel olarak Pascal tarzı harflendirmektedir.

5) Okunabilirliği sağlamada en önemli unsurlardan biri de kaynak kodun genel düzenidir. Bunun için birkaç tarz kullanılmaktadır. Biz kursumuzda Ritchie/Kernighan tarzını kullanıyoruz. Bu tarzın anahtar noktaları şunlardır:

a) Fonksiyonlar en soldaki sütuna dayalı olarak tanımlanırlar. Fonksiyon tanımlamaları arasında bir satır boşluk bırakılır.

b) Bildirim satırlarından sonra bir satır boşluk bırakılır. Örneğin:

```

int a;
long b;
→ a = 10;
  b = 20;

```

c) Üst üste birden fazla SPACE kullanılmaz. Bir SPACE yetmezse TAB kullanılır. Üst üste birden fazla TAB kullanılabilir.

d) Blok içleri bir TAB içeriden yazılır. Örneğin:

```

void foo(void)
{
→ int a, b;

→ {
→   int x, y;
→   }
→ }

```

e) İki operandlı operatörlerle operandlar arasında bir SPACE boşluk bırakılır. Ancak tek operandlı operatörlerle operand arasında boşluk bırakılmaz. (İstisna olarak iki operandlı nokta ve ok operatörleriyle operandlar arasında boşluk bırakılmaz.) Örneğin:

```

a ← = → b ← + → c;
++a;
foo(!c);

```

f) Anahtar sözcüklerden sonra 1 SPACE boşluk bırakılır. Ancak fonksiyon isimlerinden sonra bırakılmaz. Örneğin:

```
if ↓ (....)
for ↓ (....)
while ↓ (....)
foo(....);
```

g) Yazıdaki paragraf duygusu programlamada boş satır bırakılarak verilir. Yani konudan konuya geçerken bir satır boşluk bırakılmalıdır.

h) if deyiminin yazımı şöyledir:

Bloksuz biçim:

```
if ↓ (ifade1)
  → ifade2;
↓
else
  → ifade3;
```

Bloklı biçim:

```
if ↓ (ifade1) ↓ {
  → ifade2;
  ↓
  → ifade3;
}
else ↓ {
  → ifade4;
  → ifade5;
}
```

else-if Merdiveni:


```
if (ifade1)
  → ifade2;
else if (ifade3)
  → ifade4;
else if (ifades)
  → ifade6;
```

i) for deyimi şöyle yazılır:

Bloksuz biçim:

```
for (i = 0; i < 10; ++i)
  → ifade1;
```

Bloklı biçim:

```
for (i = 0; i < 10; ++i) {
  → ifade1;
  → ifade2;
}
```

j) while deyimi şöyle yazılır:

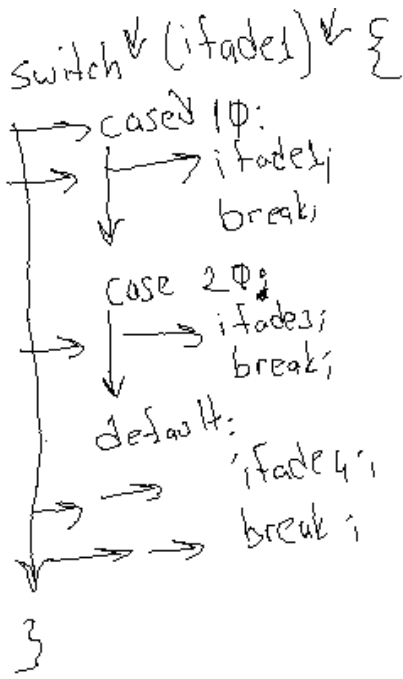
Bloksuz biçim:

```
while (ifade1)
  → ifade2;
```

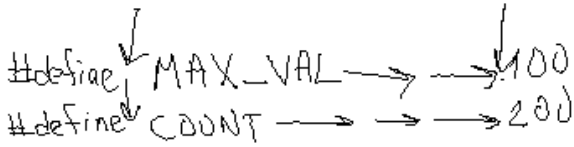
Bloklı biçim:

```
{ while (ifade1) {
  → ifade2;
  → ifade3;
}
```

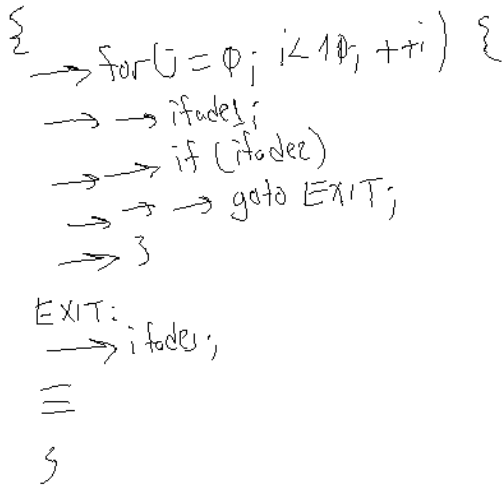
k) switch deyimi şöyle yazılır:



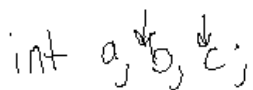
l) Sembolik sabitleri yazarken STR1'ler ve STR'ler alta alta gelmelidir. Bunun için yeterli tablama yapılabilir. Örneğin:



m) goto etiketleri büyük harflerle oluşturulmalıdır ve bir TAB geride bulunmalıdır. Örneğin:



n) Virgül atomundan önce boşluk bırakılmaz, sonra bir SPACE boşluk bırakılır. Örneğin:



Yazıların Fonksiyonlara Parametre Yoluyla Aktarılması

Yazıları fonksiyonlara parametre yoluyla aktarmak için yalnızca onların başlangıç adreslerinin fonksiyona

geçirilmesi yeterlidir. Ayrıca onların uzunluklarının fonksiyonlara aktarılması gereksizdir. Çünkü yazıların sonunda null karakter olduğu için yazının başlangıç adresini alan fonksiyon null karakter görene kadar ilerleyerek yazının tüm karakterlerini elde edebilir. Tabii bu durumda fonksiyonun parametre değişkeni char türden gösterici olmalıdır. null karakter görene kadar yazının tüm karakterlerini dolaşmak için iki kalıp kullanılabilir:

1)

```
while (*str != '\0') {  
    /* ... */  
    ++str;  
}
```

2)

```
for (i = 0; str[i] != '\0'; ++i) {  
    /* ... */  
}
```

Örneğin aslında puts fonksiyonu bizden char türden bir adres alıp null karakter görene kadar karakterleri ekrana yazdırmaktadır. Aynı fonksiyonu biz de şöyle yazabiliriz:

```
#include <stdio.h>  
  
void myputs(char *str)  
{  
    while (*str != '\0') {  
        putchar(*str);  
        ++str;  
    }  
    putchar('\n');  
}  
  
int main(void)  
{  
    char s[] = "ankara";  
  
    myputs(s);  
    puts(s);  
  
    myputs(s + 2);  
    puts(s + 2);  
  
    return 0;  
}
```

Sınıf çalışması: Başlangıç adresiyle verilen bir yazıyı tersten yazdıran putrev fonksiyonu yazınız:

```
void putsrev(char *str);
```

Çözüm:

```
#include <stdio.h>  
  
void putsrev(char *str)  
{  
    int i;  
  
    for (i = 0; str[i] != '\0'; ++i)  
        ;  
    for (--i; i >= 0; --i)  
        putchar(str[i]);  
}
```

```

    putchar('\n');
}

int main(void)
{
    char s[] = "ankara";

    putsrev(s);

    return 0;
}

```

Sınıf Çalışması: Bir yazı içerisinde belli bir karakterden kaç tane olduğunu bulan ve o sayıya geri dönen aşağıdaki prototipe sahip sonksiyonu yazınız:

```
int getch_count(char *str, char ch);
```

Çözüm:

```

#include <stdio.h>

int getch_count(char *str, char ch)
{
    int count = 0;

    while (*str != '\0') {
        if (*str == ch)
            ++count;
        ++str;
    }

    return count;
}

int main(void)
{
    char s[] = "ankara";
    int result;

    result = getch_count(s, 'a');
    printf("%d\n", result);

    return 0;
}

```

Fonksiyonların Geri Dönüş Değerlerinin Adres Olması Durumu

Bir fonksiyonun geri dönüş değeri bir adres bilgisi olabilir. Bu durumda T bir tür belirtmek üzere geri dönüş değeri T * biçiminde belirtilmelidir. Örneğin:

```

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

```

Burada foo fonksiyonu int türden bir adrese geri döner. Tabi böyle bir fonksiyonun geri dönüş değeri aynı türden bir göstericiye atanmalıdır. Örneğin:

```

int *pi;
...

```

```
pi = foo();
```

Örneğin bir dizideki en büyük elemanın adresiyle geri dönen `getmax_addr` isimli bir fonksiyonu yazacak olalım:

```
#include <stdio.h>

int *getmax_addr(int *pi, int size)
{
    int *pmax = &pi[0];
    int i;

    for (i = 1; i < size; ++i)
        if (pi[i] > *pmax)
            pmax = &pi[i];

    return pmax;
}

int main(void)
{
    int a[10] = { 34, 21, 87, 45, 32, 67, 93, 22, 9, 2 };
    int *pmax;

    pmax = getmax_addr(a, 10);
    printf("%d\n", *pmax);

    return 0;
}
```

Yukarıdaki örnekte dizinin en büyük elemanını değiştirmek isteyelim:

```
#include <stdio.h>

int *getmax_addr(int *pi, int size)
{
    int *pmax = &pi[0];
    int i;

    for (i = 1; i < size; ++i)
        if (pi[i] > *pmax)
            pmax = &pi[i];

    return pmax;
}

void disp_array(int *pi, int size)
{
    int i;

    for (i = 0; i < size; ++i)
        printf("%d ", pi[i])
        ;
    printf("\n");
}

int main(void)
{
    int a[10] = { 34, 21, 87, 45, 32, 67, 93, 22, 9, 2 };
    int *pmax;

    disp_array(a, 10);
    *getmax_addr(a, 10) = 1000;
    disp_array(a, 10);

    return 0;
}
```

```
}
```

Anahtar Notlar: C'de bir adresin sayısal bileşeni printf fonksiyonunda %p format karakteriyle yazdırılır. Örneğin:

```
char s[10];

printf("%p\n", s);
```

C'de Standart String Fonksiyonları

C'de ismi str ile başlayan strxxx biçiminde bir grup standart fonksiyon vardır. Bu fonksiyonlara string fonksiyonları denir. Bu fonksiyonlar bir yazının başlangıç adresini parametre olarak alırlar, onunla ilgili faydalı işlemler yaparlar. Bu fonksiyonların prototipleri <string.h> dosyası içerisinde bulunmaktadır.

strlen fonksiyonu

strlen fonksiyonu bir yazının karakter uzunluğunu bulmak için kullanılmaktadır:

```
unsigned int strlen(char *str);
```

Fonksiyon parametre olarak char türden bir yazının başlangıç adresini alır, geri dönüş değeri olarak bu yazının karakter uzunluğunu verir.

Anahtar Notlar: strlen fonksiyonun orijinal prototipi şöyledir:

```
size_t strlen(const char *str);
```

size_t türü ve const göstericiler ileride ele alınacaktır.

Örneğin:

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

int main(void)
{
    char s[100];
    unsigned n;

    printf("Bir yazi giriniz:");
    gets(s);

    n = strlen(s);
    printf("%u\n", n);

    return 0;
}
```

strlen fonksiyonu aşağıdaki gibi yazılabilir:

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

unsigned mystrlen(char *str)
{
    int i;

    for (i = 0; str[i] != '\0'; ++i)
        ;
}
```

```

    return i;
}

unsigned mystrlen2(char *str)
{
    int count = 0;

    while (*str != '\0') {
        ++count;
        ++str;
    }

    return count;
}

int main(void)
{
    char s[100];
    unsigned n;

    printf("Bir yazi giriniz:");
    gets(s);

    n = mystrlen(s);
    printf("%u\n", n);

    n = mystrlen2(s);
    printf("%u\n", n);

    return 0;
}

```

strcpy Fonksiyonu

Bu fonksiyon char türden bir dizi içerisindeki yazıyı başka bir diziye kopyalamakta kullanılır. Prototipi şöyledir:

```
char *strcpy(char *dest, char *source);
```

Fonksiyon ikinci parametresiyle belirtilen adresten başlayarak null karakter görene kadar (null karakter de dahil) tüm karakterleri birinci parametresi ile başlayan adresten itibaren kopyalar. Birinci parametresiyle verilen adresin aynına geri döner. Uygulamada geri dönüş değerine ciddi bir biçimde gereksinim duyulmamaktadır.

Örneğin:

```

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

int main(void)
{
    char s[32] = "ankara";
    char d[32];

    strcpy(d, s);
    puts(d);

    return 0;
}

```

Örneğin:

```

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

int main(void)
{
    char s[32] = "ankara";
    char d[32];

    strcpy(d, s + 2);
    puts(d);          /* kara */

    return 0;
}

```

strcpy fonksiyonu aşağıdaki gibi yazılabilir:

```

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

char *mystrcpy(char *dest, char *source)
{
    int i;

    for (i = 0; (dest[i] = source[i]) != '\0'; ++i)
        ;

    return dest;
}

char *mystrcpy2(char *dest, char *source)
{
    char *temp = dest;

    while ((*dest = *source) != '\0') {
        ++dest;
        ++source;
    }

    return temp;
}

int main(void)
{
    char s[32] = "ankara";
    char d[32];

    mystrcpy(d, s);
    puts(d);

    mystrcpy2(d, s);
    puts(d);

    return 0;
}

```

strcat Fonksiyonu

Bu fonksiyon bir yazının sonuna başka bir yazıyı eklemek için kullanılır. Prototipi şöyledir:

```
char *strcat(char *dest, char *source);
```

Fonksiyon ikinci parametresiyle belirtilen adresten başlayarak null karakter görene kadar tüm karakterleri (null karakter de dahil) birinci parametresiyle belirtilen yazının sonuna null karakter ekerek kopyalar. Fonksiyon birinci

parametre ile belirtilen adresin ayısına geri döner.

Örneğin:

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

int main(void)
{
    char s[32] = "ankara";
    char d[32] = "izmir";

    strcat(d, s);
    printf("%s\n", d);

    return 0;
}
```

strcat fonksiyonu şöyle yazılabilir:

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

char *mystrcat(char *dest, char *source)
{
    int i, k;

    for (i = 0; dest[i] != '\0'; ++i)
        ;
    for (k = 0; (dest[k + i] = source[k]) != '\0'; ++k)
        ;

    return dest;
}

char *mystrcat2(char *dest, char *source)
{
    char *temp = dest;

    while (*dest != '\0')
        ++dest;
    while ((*dest = *source) != '\0') {
        ++dest;
        ++source;
    }

    return temp;
}

int main(void)
{
    char s[32] = "ankara";
    char d[32] = "izmir";

    mystrcat(d, s);
    printf("%s\n", d);

    mystrcat2(d, s);
    printf("%s\n", d);    /* izmirankaraankara*/
    return 0;
}
```

Sınıf Çalışması: 1000 elemanlık char türden s imli bir dizi ve bunun yanı sıra 32 elemanlık k isimli bir dizi

tanımlayınız. Bir döngü içerisinde gets fonksiyonuyla k dizisine okuma yapınız. Oradan da okunanları s dizisinin sonuna ekleyiniz. Ta ki gets fonksiyonunda kullanıcı hiçbirşey girmeyip yalnızca ENTER tuşuna basana kadar. Döngüden çıkınca s dizisini yazdırınız.

Çözüm:

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

int main(void)
{
    char s[1000];
    char k[32];

    s[0] = '\0';
    for (;;) {
        printf("İsim giriniz:");
        gets(k);
        if (*k == '\0')
            break;
        strcat(s, k);
    }
    puts(s);

    return 0;
}
```

strcmp Fonksiyonu

strcmp fonksiyonu iki yazıyı karşılaştırmak amacıyla kullanılır. Karşılaştırma şöyle yapılır: Yazıların karakterleri eşit olduğu sürece devam edilir. İlk eşit olmayan karaktere gelindiğinde o karakterlerin hangisi karakter tablosunda daha büyük numaraya sahipse o yazı o yazıdan büyüktür. ASCII tablosu için şöyle örnekler verebiliriz:

- "ali" yazısı "Ali" yazısından büyüktür.
- "ali" yazısı "aliye" yazısından küçüktür.
- "almanya" yazısı "ali" yazısından büyüktür.

strcmp fonksiyonunun prototipi şöyledir:

```
int strcmp(char *s1, char *s2);
```

Fonksiyon birinci yazı ikinci yazıdan büyükse pozitif herhangi bir değer, küçükse negatif herhangi bir değer ve eşitse sıfır değerine geri döner.

Örneğin:

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

int main(void)
{
    char s[64];
    char passwd[] = "maviay";

    printf("Enter password:");
    gets(s);

    if (!strcmp(s, passwd))
        printf("Ok\n");
}
```

```

else
    printf("Invalid password\n");

return 0;
}

```

Fonksiyonu şöyle yazabiliriz:

```

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

int mystrcmp(char *s1, char *s2)
{
    while (*s1 == *s2) {
        if (*s1 == '\0')
            break;
        ++s1;
        ++s2;
    }

    return *s1 - *s2;
}

int main(void)
{
    char s[64];
    char passwd[] = "maviay";

    printf("Enter password:");
    gets(s);

    if (!mystrcmp(s, passwd))
        printf("Ok\n");
    else
        printf("Invalid password\n");

    return 0;
}

```

strcmp fonksiyonun büyük harf küçük harf duyarlılığı olmadan karşılaştırma yapan stricmp isimli bir versiyonu da vardır. Ancak stricmp standart bir fonksiyon değildir. Fakat Microsoft derleyicilerinde, Borland derleyicilerinde ve gcc derleyicilerinde bir eklenti biçiminde bulunmaktadır. stricmp fonksiyonu şöyle yazılabilir:

```

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

int mystricmp(char *s1, char *s2)
{
    while (tolower(*s1) == tolower(*s2)) {
        if (*s1 == '\0')
            break;
        ++s1;
        ++s2;
    }

    return tolower(*s1) - tolower(*s2);
}

int main(void)
{
    char s[64];
    char passwd[] = "maviay";

    printf("Enter password:");

```

```

gets(s);

if (!mystricmp(s, passwd))
    printf("Ok\n");
else
    printf("Invalid password\n");

return 0;
}

```

NULL Adres Kavramı (NULL Pointer)

NULL adres derleyici tarafından seçilmiş bir sayısal bileşene sahip olan özel bir adrestir. NULL adres C'de geçerli bir adres kabul edilmez. Başarısızlığı anlatmak için kullanılır. NULL adresin sayısal değeri standartlara göre derleyiciden derleyiciye değişebilir. Yaygın derleyicilerin hepsinde NULL adres 0 sayısal bileşenine sahip (yani belleğin tepesini belirten) adrestir. Fakat NULL adres standartlara göre farklı sistemlerde farklı biçimde olabilir.

C'de 0 değerini veren tamsayı türlerine ilişkin sabit ifadeleri NULL adres sabiti olarak kullanılır. Örneğin:

```

5 - 5
0
1 - 1

```

gibi ifadeler aynı zamanda NULL adres sabiti anlamına gelmektedir. Biz bir göstericiye bu değerleri atadığımızda derleyici o sistemde hangi sayısal bileşen NULL adres belirtiyorsa göstericiye onu atar. Yani buradaki 0, 0 adresini temsil etmez. O sistemdeki NULL adresi temsil eder. Örneğin:

```
int *pi = 0;
```

Burada pi göstericisine int bir sıfır atanmıyor. O sistemde NULL adres neyse o değer atanıyor.

NULL adres sabitini (yani sıfır değerini) her türden göstericiye atayabiliriz. Bu durumda o göstericinin içerisinde "NULL adres" bulunur. Örneğin:

```
char *pc = 0;
```

pc'nin içerisinde NULL adres vardır. NULL adres sabitinin türü yoktur. NULL adres sabiti (yani sıfır sayısı) her türden göstericiye atanabilir. Yukarıdaki örnekte pc göstericisi char türündendir. Fakat içerisinde NULL adres vardır.

Örneğin bir sistemde NULL adresin sayısal bileşeni 0xFFFF olsun. Biz bu sistemde göstericiye NULL adres atayabilmek için, göstericiye 0xFFFF atayamayız. Yine düz sıfır atarız. Bu düz sıfır zaten o sistemdeki NULL adres anlamına gelir. Örneğin:

```
char *pc = 0;
```

Burada pc'ye sıfır adresi atanmıyor, o sistemdeki NULL adres olan 0xFFFF sayısal bileşenine sahip olan adres atanıyor.

Anahtar Notlar: Standartlara göre NULL adres sabiti aynı zamanda sıfır değerinin void * türüne dönüşülmüş biçimi de olabilir. Yani (void *)0 da NULL adres anlamına gelir.

Bir adresin NULL olup olmadığı == ve != operatörleriyle öğrenilebilir. Örneğin:

```
if (pi == 0) {
```

```
    ...  
}
```

Burada pi'nin içerisindeki adresin sayısal bileşeninin sıfır olup olmadığına bakılmamaktadır. O sistemdeki NULL adres olup olmadığına bakılmaktadır. Örneğin ilgili sistemde NULL adres 0xFFFF sayısal değerine ilişkin olsun. Ve pi'nin içerisinde NULL adres olduğunu varsayalım:

```
if (pi == 0) {  
    ...  
}
```

Burada if deyimi doğrudan sapar.

if parantezinin içerisinde yalnızca bir adres ifadesi varsa bu adresin NULL adres olup olmadığına bakılır. Eğer adres NULL adres ise if deyimi yanlıştan sapar, NULL adres değilse doğrudan sapar. Örneğin:

```
if (pi) {  
    ...  
}  
else {  
    ...  
}
```

Bu örnekte ilgili sistemde NULL adresin 0xFFFF olduğunu düşünelim. Yukarıdaki if deyimi yanlıştan sapacaktır. Çünkü bu özel durumda adresin sayısal bileşeninin sıfır olup olmadığına değil, NULL adres olup olmadığına bakılmaktadır. Benzer biçimde C'de ! operatörünün operandı bir adres bilgisiyse bu operatör adres NULL adres ise 1 değerini, NULL adres değilse 0 değerini üretir.

* ya da köşeli parantez operatörleriyle NULL adresin içeriği elde edilmeye çalışılırsa bu durum tanımsız davranışa (undefined behavior) yol açar.

C'de NULL adres sabiti daha okunabilir ifade edilsin diye NULL isimli bir sembolik sabitle temsil edilmiştir:

```
#define NULL    0
```

Programcılar genellikle NULL adres sabiti için düz sıfır kullanmak yerine NULL sembolik sabitini kullanırlar. Örneğin:

```
if (pi == NULL) {  
    ...  
}
```

NULL sembolik sabiti <stdio.h> v e pek çok başlık dosyasında bulunmaktadır. NULL sembolik sabitini int türden bir sıfır olarak değil, NULL adres sabiti olarak kullanmalıyız.

strchr Fonksiyonu

Bu fonksiyon bir yazı içerisinde bir karakteri aramak için kullanılır. Eğer karakter yazı içerisinde bulunursa fonksiyon karakterin ilk bulunduğu yerin adresiyle geri döner. Eğer bulunamazsa NULL adresle geri döner. Bu fonksiyon null karakteri de arayabilmektedir.

```
char *strchr(char *str, char ch);
```

Örneğin:

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

int main(void)
{
    char s[] = "ankara";
    char *str;

    str = strchr(s, 'k');
    if (str == NULL)
        printf("karakter yok!..\n");
    else
        printf("Buldu:%s\n", str);

    return 0;
}
```

strchr fonksiyonu şöyle yazılabilir:

```
#include <stdio.h>

char *mystrchr(char *str, char ch)
{
    while (*str != '\0') {
        if (*str == ch)
            break;
        ++str;
    }

    if (*str == '\0' && ch != '\0')
        return NULL;

    return str;
}

int main(void)
{
    char s[] = "ankara";
    char *str;

    str = mystrchr(s, 'k');
    if (str == NULL)
        printf("karakter yok!..\n");
    else
        printf("Buldu:%s\n", str);

    return 0;
}
```

strrchr Fonksiyonu

Fonksiyon tamamen strchr fonksiyonu gibidir. Ancak ilk bulunan karakterin değil son bulunan karakterin adresiyle geri döner. Örneğin:

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

int main(void)
{
    char s[] = "izmir";
    char *str;
```

```

    str = strrchr(s, 'i');
    if (str == NULL)
        printf("karakter yok!..\n");
    else
        printf("Buldu:%s\n", str);

    return 0;
}

```

strrchr fonksiyonu şöyle yazılabilir:

```

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

char *mystrrchr(char *str, char ch)
{
    char *result = NULL;

    while (*str != '\0') {
        if (*str == ch)
            result = str;
        ++str;
    }

    if (ch == '\0')
        return str;

    return result;
}

int main(void)
{
    char s[] = "izmir";
    char *str;

    str = mystrrchr(s, 'i');
    if (str == NULL)
        printf("karakter yok!..\n");
    else
        printf("Buldu:%s\n", str);

    return 0;
}

```

strncpy Fonksiyonu

Bu fonksiyon bir yazının ilk n karakterini başka bir diziye kopyalar. Prototipi şöyledir:

```
char *strncpy(char *dest, char *source, unsigned n);
```

Bu fonksiyonda eğer n değeri strlen(source) değerinden küçük ya da eşitse fonksiyon null karakteri hedef diziye eklemeyiz. Örneğin:

```

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

int main(void)
{
    char s[] = "izmir";
    char d[32] = "ankara";
}

```

```

    strncpy(d, s, 3);
    puts(d);      /* izmara */

    return 0;
}

```

Eğer n değeri strlen(source) değerinden büyükse ya da eşitse hedefe null karakter de kopyalanır. Üstelik geri kalan miktar kadar null karakter kopyalanır. Örneğin:

```

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

int main(void)
{
    char s[] = "izmir";
    char d[32] = "ankara";

    strncpy(d, s, 30);
    puts(d);      /* izmir çıkacak fakat 24 tane null karakter d'ye eklenecek */

    return 0;
}

```

Fonksiyon yine strcpy de olduğu gibi, kopyalamanın yapıldığı hedef adrese geri döner.

strncpy fonksiyonu şöyle yazılabilir:

```

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

char *mystrncpy(char *dest, char *source, unsigned n)
{
    char *temp = dest;

    while (n-- > 0) {
        *dest = *source;
        if (*source != '\0')
            ++source;
        ++dest;
    }

    return temp;
}

int main(void)
{
    char s[] = "izmir";
    char d[32] = "ankara";

    mystrncpy(d, s, 30);
    puts(d);      /* izmir çıkacak fakat 24 tane null karakter d'ye eklenecek */

    return 0;
}

```

strncat Fonksiyonu

Bu fonksiyon bir yazının sonuna başka bir yazının ilk n karakterini ekler. Prototipi şöyledir:

```
char *strncat(char *dest, char *source, unsigned n);
```


Fonksiyon her zaman null karakteri ekler. Ancak $n > \text{strlen}(\text{source})$ ise null karakteri ekleyip işlemini sonlandırır. (Yani strcpy'de olduğu gibi geri kalan miktar kadar null karakter eklemeyiz.) Başka bir deyişle eğer $n > \text{strlen}(\text{source})$ ise fonksiyon tamamen strcat gibi çalışır. Fonksiyon yine birinci parametresiyle belirtilen adresin ayısına geri döner. Örneğin:

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

int main(void)
{
    char s[] = "izmir";
    char d[32] = "ankara";

    strncat(d, s, 3);
    puts(d);    /* ankaraizm */

    return 0;
}
```

Örneğin:

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

int main(void)
{
    char s[] = "izmir";
    char d[32] = "ankara";

    strncat(d, s, 100);
    puts(d);    /* ankaraizmır çıkar fakat dizi taşması olmaz */

    return 0;
}
```

strncat fonksiyonu şöyle yazılabilir:

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

char mystrncat(char *dest, char *source, unsigned n)
{
    char *temp = dest;

    while (*dest != '\0')
        ++dest;

    while ((*dest = *source) != '\0' && n > 0) {
        ++dest;
        ++source;
        --n;
    }

    /* if (n == 0) */
    *dest = '\0';

    return temp;
}

int main(void)
{
    char s[] = "izmir";
    char d[32] = "ankara";
}
```

```

    mystrncat(d, s, 50);
    puts(d);    /* ankaraizmir çıkar fakat dizi taşması olmaz */

    return 0;
}

```

strncmp Fonksiyonu

Bu fonksiyon iki yazının ilk n karakterinmi karşılaştırmakta kullanılmaktadır. Fonksiyonun prototipi şöyledir:

```
int strncmp(char *s1, char *s2, unsigned n);
```

Fonksiyonun üçüncü parametresi ilk kaç karakterin karşılaştırılacağını belirtir. n sayısı büyükse iki yazıdan hangisinde null karakter görülürse işlem biter. Fonksiyonun geri dönüş değeri yine strcmp fonksiyonunda olduğu gibidir.

Örneğin:

```

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

int main(void)
{
    char passwd[] = "maviay";
    char s[64];

    printf("Enter password:");
    gets(s);

    if (!strncmp(passwd, s, strlen(s)))
        printf("Ok\n");
    else
        printf("invalid password\n");

    return 0;
}

```

Adres Operatörleriyle ++ ve -- Operatörlerinin Birlikte Kullanılması

1) * Operatörüyle Kullanım

a) ++*p Kullanımı: Burada *p bir artırılır. Yani bu ifade *p = *p + 1 ile eşdeğerdir.

```

#include <stdio.h>

int main(void)
{
    int a = 10;
    int *pi = &a;

    ++*pi;

    printf("%d\n", a);    /* 11 */

    return 0;
}

```

b) *++p Durumu: Burada önce p bir artırılır sonra artırılmış adresin içeriğine erişilir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int a[2] = { 10, 20 };
    int *pi = a;

    *++pi = 30;

    printf("%d\n", a[1]);    /* 30 */

    return 0;
}
```

c) *p++ Durumu: Burada p bir artırılır ancak ++ sonrak durumunda olduğu için artırılmış adresin içeriğine erişilir. Örneğin:

```
#include <stdio.h>

char *mystrcpy(char *dest, char *source)
{
    char *temp = dest;

    while ((*dest++ = *source++) != '\0')
        ;

    return temp;
}

int main(void)
{
    char s[] = "ankara";
    char d[32];

    mystrcpy(d, s);

    puts(d);    /* ankara */

    return 0;
}
```

Görüldüğü gibi strcpy işlemi aşağıdaki gibi kompakt yazılabilmektedir:

```
while ((*dest++ = *source++) != '\0')
    ;
```

Burada her defasında *source değeri *dest değerine atanır ve source ile dest bir artırılmış olur.

2) [] Operatörü İle Kullanım

a) ++p[n] Durumu: Burada p[n] bir artırılır. Yani bu ifade $p[n] = p[n] + 1$ ile eşdeğerdir.

b) p[n]++ Durumu: Burada p[n] bir artırılır ancak sonraki operatöre p[n]'in artırılmamış değeri sokulur.

c) p[++n] Durumu: Burada n bir artırılır ve artırılmış indeksteki elemana erişilir.

d) p[n++] Durumu: Burada n bir artırılır fakat p[n] sonraki işleme sokulur. Örneğin:

3) & Operatörü İle Kullanım

& adres operatörü ile ++ ve -- operatörleri birlikte kullanılamazlar. Çünkü & operatörünün ürettiği değer nesne değildir. Örneğin:

```
int a;

++&a;    /* geçerli değil */
```

++ ve -- operatörleri nesne üretmezler. Yani ++a işleminde a bir artırılır, fakat ürün olarak a elde edilmez. a artırılmış değeri elde edilir. Benzer biçimde sonrek durumunda aynı şey söz konusudur. Örneğin:

```
a++ = b; /* geçerli değil */
```

Dolayısıyla &++a ve &a++ ifadeleri de anlamsızdır. Ancak nesnelerin adresleri alınabilir.

Farklı Türden Adresin Bir Göstericiye Atanması

C'de bir göstericiye farklı türden bir adres atanamaz. Örneğin:

```
char s[] = "ankara";
int *pi;

pi = s;    /* geçersiz */
```

Anahtar Notlar: Yukarıdaki atama C'de geçersizdir. Fakat bilindiği gibi geçersiz bir programı C derleyicileri bir mesaj vermek koşuluyla derleyebilir. Yani bir derleyicinin yukarıdaki kodu derlemesi onun insafına kalmıştır.

Fakat bir göstericiye farklı türden bir adres tür dönüştürme operatörüyle atanabilir. Örneğin:

```
char s[] = "ankara";
int *pi;

pi = (int *)s;    /* geçerli */
```

Örneğin:

```
#include <stdio.h>

int main(void)
{
    char s[] = "aaaa";
    int *pi;

    pi = (int *)s;

    printf("%d\n", *pi);    /* buradaki sayı kaçtır? */

    return 0;
}
```

Bir adresi başka türden bir göstericiye tür dönüştürme operatörüyle atamak çoğu zaman anlamsızdır. Örneğin yukarıdaki kodda artık *pi ifadesi dört tane 'a' karakterinin oluşturduğu int değer olarak yorumlanır. Böyle bir şeyi neden yapmak isteyebiliriz? Bazı durumlarda böylesi işlemlere gereksinim duyulabilmektedir.

Adres Olmayan Bir Bilginin Bir Göstericiye Atanması

Bazen elimizde bir tamsayı değeri vardır. Bu değeri sayısal bileşen anlamında bir göstericiye yerleştirmek

isteyebiliriz. Örneğin:

```
int *pi;
int a = 0x1FC0;

pi = a;      /* geçersiz! */
```

Bu tür atamalar da tür dönüştürme operatörüyle yapılabilmektedir:

```
int *pi;
int a = 0x1FC0;

pi = (int *)a;    /* geçerli! */
```

Örneğin:

```
int *pi;

pi = 0x1FC0;      /* geçersiz! */
pi = (int *)0x1FC0; /* geçerli! */
```

Dizi İsimleriyle Göstericilerin Karşılaştırılması

Dizi isimleriyle göstericiler bazen kullanım bakımından yeni öğreneleri tereddütte bırakabilmektedir. Aralarındaki benzerlikler ve farklılıklar şöyledir:

Açıklamalarda aşağıdaki iki bildirim göz önünde bulundurunuz:

```
int a[10];
int *pi;
```

- Dizi isimleri de göstericiler de birer adres belirtir. Yani ikisini de kullandığımızda biz adres bilgisi kullanmış oluruz. Fakat göstericiler bir nesne belirtir. Yani onun içerisine bir adres bilgisi atayabiliriz. Ancak dizi isimleri o dizilerin başlangıç adreslerini belirtir fakat nesne belirtmez. Biz bir dizi ismine birşey atayamayız. Örneğin:

```
char s[10];
char *pc;

s = pc;      /* geçersiz! */
pc = s;      /* geçerli! */
```

- Dizi isimleri dizilerin başlangıç adresi belirtir. Yani onların belirttiği adreste tahsis edilmiş bir alan vardır. Halbuki göstericilerin içerisindeki adresler değiştirilebilir. Göstericiler bellekte başka yerleri gösterebilir hale getirilebilir.

Gösterici Hataları

Bir gösterici bellekte potansiyel olarak her yeri gösterebilir. Fakat bir göstericiyi kullanarak * ya da [] operatörleriyle bellekte bizim için tahsis edilmemiş (yani bize ait olmayan) alanlara erişmek normal bir durum değildir. Bu durum C'de tanımsız davranışa (undefined behavior) yol açmaktadır. elimiz bir gösterici olsun biz parmağımızla başkasına ait bir araziye gösterebiliriz. Bu yasak değildir. Fakat oraya erişemeyiz (yani giremeyiz). Girsek başımıza ne geleceği belli değildir. Tabi biz kendi arazimizi göstererek oraya girebiliriz. İşte göstericiler

de tıpkı böyledir. Biz göstericilerle kendi tanımladığımız yani bizim için ayrılan nesnelere erişmeliyiz.

Peki bir yerin bizim için ayrıldığını nasıl anlarız? İşte tanımlama yoluyla oluşturduğumuz nesnelere bize tahsis edilmiştir. (Yani onların tapusu bizdedir.) Biz o alanlara erişebiliriz. Örneğin:

```
int a;  
char s[100];
```

Burada &a'dan itibaren 4 byte, s'ten itibaren 100 byte bize ayrılmıştır. Biz bu alanları istediğimiz gibi kullanabiliriz. **Anahtar Notlar:** Bir fonksiyona biz bir adres geçirelim. Fonksiyon onun tahsis edilmiş bir adres olup olmadığını anlayamaz. Bir adresin tahsis edilmiş olup olmadığını anlamamızın C'de resmi bir yolu yoktur.

Bizim için tahsis edilmemiş alanlara erişme işlemine "gösterici hatası" denilmektedir. Gösterici hataları maalesef deneyimli programcılar tarafından bile yanlışlıkla yapılabilmektedir. Gösterici hatalarının tipik ortaya çıkış biçimi şunlardır:

1) İlkdeğer Verilmemiş Göstericilerin Yol Açtığı Gösterici Hataları: Bir göstericinin içerisinde rastgele bir değer varsa onu * ya da [] parantez operatörleriyle kullanamayız. Örneğin:

```
#include <stdio.h>  
  
int main(void)  
{  
    int *pi;  
  
    *pi = 100;    /* 100 rastgele bir yere atanıyor */  
  
    return 0;  
}
```

Örneğin:

```
#include <stdio.h>  
  
int main(void)  
{  
    char *s;  
  
    gets(s);    /* Gösterici hatası! gets klavyeden girilen karakterleri nereye yerleştiriyor? */  
  
    return 0;  
}
```

2) Dizi Taşmalarından Doğan Gösterici Hataları: Bir dizi için tam o kadar yer ayrılmıştır. Dizinin gerisi de ötesi de bizim tahsis edilmemiştir. Oralara erişmek gösterici hatalarına yol açar. Örneğin:

```
int a[10];  
int i;  
...  
for (i = 0; i <= 10; ++i)    /* dikkat a[10] bizim için tahsis edilmemiş */  
    a[i] = 0;
```

Örneğin:

```
char s[] = "ankara";  
char d[] = "istanbul";  
  
strcat(d, s);    /* dikkat d dizisi taşıyor! */
```

Örneğin:

```
char s[10];  
gets(s);
```

Burada kullanıcı en fazla 9 karakter girmelidir. Yoksa dizi taşar.

3) Ömrü Biten Nesnelerin Yol Açtığı Gösterici Hataları: Bir fonksiyon yerel bir nesnenin ya da dizinin adresi ile geri dönmemelidir. Çünkü fonksiyon bitince o fonksiyonun yerel değişkenleri boşaltılır. Dolayısıyla fonksiyonun bize verdiği adres artık tahsis edilmiş bir alanın adresi olmaz. Örneğin:

```
#include <stdio.h>  
  
char *getname(void)  
{  
    char name[128];  
  
    printf("Adi Soyadi:");  
    gets(name);  
  
    return name;  
}  
  
int main(void)  
{  
    char *s;  
  
    s = getname();  
    printf("%s\n", s);  
  
    return 0;  
}
```

Burada getname fonksiyonu geri döndüğünde artık name isimli dizi yok edilecektir. Dolayısıyla main'de s göstericisine atanan adres tahsis edilmemiş bir alanın adresi olacaktır. Yukarıdaki fonksiyonda name global yapılırsa sorun kalmaz. Ya da fonksiyon şöyle düzenlenebilir:

```
#include <stdio.h>  
  
void getname(char *s)  
{  
    printf("Adi Soyadi:");  
    gets(s);  
}  
  
int main(void)  
{  
    char s[128];  
  
    getname(s);  
    printf("%s\n", s);  
  
    return 0;  
}
```

void Adresler ve void Göstericiler

C'de void türden bir değişken tanımlanamaz. Ancak gösterici tanımlanabilir. Örneğin:

```
void a;          /* geçersiz! */
void *pv;       /* geçerli */
```

void göstericiler türsüz göstericilerdir. void göstericiler * ya da [] operatörleriyle kullanılamazlar. Çünkü bu operatörler eriştiği yerdeki nesnenin türünü bilmek zorundadır. void göstericiler ya da void adresler artırılmazlar ve eksiltilemezler. Çünkü derleyici bu işlemlerde göstericinin sayısal bileşenini kaç artırıp kaç eksilteceğini bilememektedir. Peki void göstericiler ne işe yarar?

void bir göstericiye herhangi türden bir adres doğrudan atanabilir. Tür dönüştürme operatörüne hiç gerek yoktur. Örneğin:

```
void *pv;
char s[10];
int a[10];

pv = s;    /* geçerli */
pv = a;    /* geçerli */
```

Benzer biçimde void bir adres de herhangi bir türden göstericiye atanabilir. Örneğin:

```
void *pv;
int *pi;
char s[10];

pv = s;    /* geçerli */
...
pi = pv;   /* geçerli */
```

void göstericilere neden gereksinim duyulduğunu açıklayabilmek için memcpy fonksiyonu iyi bir örnek olabilir. memcpy fonksiyonu bir adresten bir adrese koşulsuz n byte kopyalamaktadır. Bu fonksiyon strcpy fonksiyonunu andırmakla birlikte ondan farklıdır. memcpy null karaktere bakmaz. Koşulsuz n byte kopyalar. Örneğin herhangi bir türden diziyeye aynı türden başka bir diziyeye memcpy ile kopyalayabiliriz. C'de başı memxxx biçiminde başlayan fonksiyonların prototipleri de <string.h> içerisinde.

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

int main(void)
{
    int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int b[10];
    int i;

    memcpy(b, a, 10 * sizeof(int));

    for (i = 0; i < 10; ++i)
        printf("%d ", b[i]);
    printf("\n");

    return 0;
}
```

Biz memcpy fonksiyonuyla herhangi türden iki diziyi kopyalayabiliriz:

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

int main(void)
{
    double a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

```



```

double b[10];
int i;

memcpy(b, a, 10 * sizeof(double));

for (i = 0; i < 10; ++i)
    printf("%f ", b[i]);
printf("\n");

return 0;
}

```

İşte memcpy gibi bir fonksiyonun bizden her türlü adresi kabul edebilmesi için onun parametrelerinin void gösterici olması gerekir. Gerçekten de memcpy fonksiyonunun prototipi şöyledir:

```
void *memcpy(void *dest, void *source, unsigned n);
```

Fonksiyon ikinci parametresiyle belirtilen adresten başlayarak birinci parametresiyle belirtilen adrese koşulsuz n byte kopyalar. Birinci parametresiyle belirtilen adresin aynısına geri döner.

Peki biz memcpy gibi bir fonksiyonu nasıl yazabiliriz? void göstericiler artırılıp azaltılmadığına göre onların türü belirli bir göstericiye atanması gerekir. Örneğin:

```

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

void *mymemcpy(void *dest, void *source, unsigned n)
{
    char *pcdest = dest;
    char *pcsource = source;

    while (n-- > 0)
        *pcdest++ = *pcsource++;

    return dest;
}

int main(void)
{
    double a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    double b[10];
    int i;

    mymemcpy(b, a, 10 * sizeof(double));

    for (i = 0; i < 10; ++i)
        printf("%f ", b[i]);
    printf("\n");

    return 0;
}

```

memcpy fonksiyonuyla yazı kopyalaması da yapabiliriz. Örneğin:

```
strcpy(d, s);
```

ile

```
memcpy(d, s, strlen(s) + 1);
```

işlevsel olarak eşdeğerdir. Örneğin:

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

int main(void)
{
    char s[30] = "ankara";
    char d[30];

    strcpy(d, s);
    memcpy(d, s, strlen(s) + 1);

    return 0;
}
```

memset isimli fonksiyon bir adresten başlayarak koşulsuz n byte'a belli bir değeri atar. Yani n tane byte'ı belli bir değerle doldurur. Örneğin:

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

int main(void)
{
    int a[10];
    int i;

    memset(a, 0, 10 * sizeof(int));

    for (i = 0; i < 10; ++i)
        printf("%d ", a[i]);
    printf("\n");

    return 0;
}
```

Örneğin:

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

int main(void)
{
    char s[200];

    memset(s, 'a', 199);
    s[199] = '\0';
    printf("%s\n", s);

    return 0;
}
```

memset fonksiyonunun prototipi şöyledir:

```
void *memset(void *dest, int val, unsigned n);
```

Fonksiyonun birinci parametresi doldurulacak yerin adresini, ikinci parametresi doldurulacak değeri ve üçüncü parametresi kaç adet değer doldurulacağını belirtir. memset fonksiyonunu şöyle yazabiliriz:

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

void *memset(void *dest, int val, unsigned n)
```

```

{
    unsigned char *pctest = dest;

    while (n-- > 0)
        *pctest++ = ch;

    return dest;
}

int main(void)
{
    char s[200];

    memset(s, 'a', 199);
    s[199] = '\0';

    printf("%s\n", s);

    return 0;
}

```

memset fonksiyonunun strset isimli bir kardeşi vardır. strset char türden bir diziyi null karakter görene kadar belli bir karakterle doldurur:

```
char *strset(char *dest, char ch);
```

Örneğin:

```

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

int main(void)
{
    char s[] = "ankara";

    strset(s, 'x');
    puts(s);

    return 0;
}

```

Anahtar Notlar: C'de strxxx biçiminde str ile başlayan fonksiyonlar char * türünden göstericileri parametre olarak alır ve null karakter görene kadar işlem yapar. Fakat memxxx biçiminde mem ile başlayan fonksiyonlar void * türünden parametre alır ve koşulsuz n byte için işlem yapar.

```

strcpy --- memcpy
strset --- memset
strchr --- memchr

```

strset fonksiyonu yerine memset kullanılabilir. Yani:

```
strset(s, ch);
```

ile,

```
memset(s, ch, strlen(s));
```

işlevsel olarak eşdeğerdir.

Yukarıda da belirtildiği gibi C'de void bir göstericiye her türden adres doğrudan atanabilir. Benzer biçimde void bir adres de her türlü göstericiye atanabilir. Ancak void adresin herhangi bir göstericiye atanması bazılarınca

eleştirilmektedir. Çünkü bu sayede aslında yasak olan birşeyi yapabilir duruma gelmekteyiz (yani hülle yapılabilir):

```
char *pc;  
int *pi;  
  
pi = pc;    /* yasak */
```

Fakat:

```
char *pc;  
int *pi;  
void *pv;  
  
pv = pc;    /* geçerli */  
pi = pv;    /* geçerli */
```

İşte C++'ta eleştirilen bu durum düzeltilmiştir. Şöyle ki: C++'ta yine void göstericiye her türden adres doğrudan atanabilir. Fakat void bir adres tür dönüştürmesi yoluyla başka türden bir göstericiye atanabilir. Böylece yukarıdaki kod C'de geçerli olduğu halde C++'ta geçerli değildir. Bazı C programcıları böylesi kodların her iki dilde de geçerli olmasını sağlamak için void adresleri atarken tür dönüştürmesi yaparlar. Örneğin:

```
pi = (int *)pv;    /* Hem C'de hem de C++'ta geçerli */
```

Özetle C'de void göstericiler bir adresin yalnızca sayısal bileşenini tutmak için ve tür dönüştürmesine programcıyı zorlamamak için kullanılmaktadır. Eğer memcpy fonksiyonunun parametreleri char * türünden olsaydı. O zaman onu biz şöyle kullanmak zorunda kalırdık:

```
int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
int b[10];  
int i;  
  
memcpy((char *)b, (char *)a, 10 * sizeof(int));
```

void göstericiler C'ye 80'li yılların ikinci yarısında katılmıştır.

Anahtar Notlar: Kursun bu noktasında koşul operatörü anlatılmıştır. Fakat notlarda "Diziler" konusundan sonraya eklenmiştir.

Göstericilerin Uzunlukları

Bir gösterici kaç byte uzunluktadır? Göstericilerin uzunlukları onların türleriyle ilgili değildir. Çünkü göstericilerin türleri onların gösterdikleri yerdeki nesnenin uzunluğuyla ilişkilidir. Bir göstericinin uzunluğu o sistemin adres alanına bağlıdır. Örneğin teorik olarak 32 bit işlemcilere 4GB RAM, 64 bit işlemcilere 16EB RAM takılabilir. Bu durumda 32 bit sistemlerdeki göstericiler 4 byte, 64 bit sistemlerdeki göstericiler 8 byte yer kaplarlar. Biz kursumuzda şekilleri genel olarak 32 bit mikroişlemcilere göre çizeriz. Yani örneğin sizeof(char *) ile, sizeof(double *) aynı değerlerdedir.

String İfadeleri

C'de derleyici ne zaman iki tırnak içerisinde bir yazı görse önce o yazıyı char türden bir dizinin içerisine yerleştirir, sonuna null karakteri ekler ve iki tırnak yerine o dizinin başlangıç adresini insert eder. Yani C'de iki tırnak ifadeleri char türden adres belirtmektedir. Böylece C'de iki tırnak ifadeleri doğrudan char türden bir göstericiye atanmalıdır. Örneğin:

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

```

int main(void)
{
    char *str;

    str = "ankara";
    printf("%s\n", str);

    return 0;
}

```

Eşdeğeri aşağıdaki gibi düşünülebilir:

```

#include <stdio.h>

static char compiler_generated_name[] = "ankara";

int main(void)
{
    char *str;

    str = compiler_generated_name;
    printf("%s\n", str);

    return 0;
}

```

Bu durumda örneğin biz char * parametrelili bir fonksiyonu string ifadesiyle çağırabiliriz. Örneğin:

```

#include <stdio.h>

int main(void)
{
    unsigned n;

    n = strlen("ankara");
    printf("%u\n", n);

    puts("ankara");

    return 0;
}

```

Örneğin char türden bir diziye sonradan (ilkdeğer vererek değil) bir yazı yerleştirecek olsak bunu karakter karakter yerleştirmek pek uygun olmaz. Bunun en pratik ve doğru yolu strcpy fonksiyonunu kullanmaktır:

```

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

int main(void)
{
    char s[32];

    strcpy(s, "ankara");
    puts(s);

    return 0;
}

```

char türden bir dizi ismine iki tırnak ifadesini atayamayız. Çünkü dizi isimleri nesne belirtmez. Göstericiler nesne belirtir. Örneğin:

```
char s[32];
```

```
char *ps;
```

```
s = "ankara";    /* geçersiz! */  
ps = "ankara";  /* geçerli */
```

C'de istisna olarak char türden bir diziye iki tırnak ifadesi ile ilkdeğer verilirken kullanılan iki tırnak bir adres belirtmemektedir. Bu iki tırnak içindeki karakterleri tek tek diziye yerleştirir ve sonuna null karakter ekler anlamına gelir. Yani:

```
char s[32] = "ankara";
```

bildiriminde bu iki tırnak ifadesi için derleyici bir adres insert etmeyecektir. Bu istisna durum aşağıdakiyle eşdeğerdir:

```
char s[32] = {'a', 'n', 'k', 'a', 'r', 'a', '\\0'};
```

Bunun dışındaki iki tırnak ifadelerinin hepsi birer string belirtir ve bunlar için derleyici birer adres insert eder. Örneğin:

```
char *ps = "ankara";
```

Burada "ankara" bir adres belirtmektedir.

C'de iki tırnak ifadeleri ile oluşturulan string'ler güncellenmeye çalışılmamalıdır. String karakterlerinin değiştirilmesi "tanımsız davranışa (undefined behavior)" yol açar. Örneğin:

```
#include <stdio.h>  
  
int main(void)  
{  
    char *str = "ankara";  
    char s[] = "ankara";  
  
    *str = 'x';    /* undefined behavior */  
    *s = 'x';     /* normal */  
  
    return 0;  
}
```

Örneğin:

```
#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
    char s[] = "ankara";  
    char *str = "ankara";  
  
    *strchr(s, 'k') = 'x';  
    puts(s);  
  
    *strchr(str, 'k') = 'x';    /* undefined behavior */  
    puts(s);  
  
    return 0;  
}
```

String'ler statik ömürlü nesnelerdir. Yani hangi fonksiyonda oluşturulmuş olursa olsunlar programın başından sonuna kadar bellekte kalırlar. Dolayısıyla bir fonksiyonun bir string'in başlangıç adresine geri dönmesi gösterici hatası değildir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char *str;

    str = getstr();
    printf("%s\n", str);

    return 0;
}
```

Aşağıdaki örnekte iki tırnak istisna olan duruma ilişkindir. Dolayısıyla burada dizinin adresiyle geri dönülmektedir. O da fonksiyon bittiğinde yok edilir.

```
#include <stdio.h>

char *getstr(void)
{
    char s[] = "ankara";

    return s;
}

int main(void)
{
    char *str;

    str = getstr(); /* dikkat! gösterici hatası */
    printf("%s\n", str);

    return 0;
}
```

Özdeş string'ler için aynı yerin mi kullanılacağı ya da farklı yerler mi ayrılacağı C'de belirsiz (unspecied) davranış olarak ele alınmaktadır. Yani derleyici bu ikisinden birini yapabilir. Bu durum derleyicileri yazanların isteğine bırakılmıştır. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char *s = "ankara";
    char *k = "ankara";

    printf(s == k ? "Evet\n" : "Hayir\n");

    return 0;
}
```

Burada derleyici bir tane "ankara" yazısı yerleştirip s ve k'ya onu atayabilir. Ya da iki farklı "ankara" yazısı yerleştirip s ve k'ya farklı adresler atayabilir. Bazı derleyiciler de derleyici ayarlarından programcı bunu değiştirebilmektedir.

Boş bir string oluşturulabilir. Bu durumda derleyici diziye yalnızca null karakteri yerleştirir. Örneğin:

```
char *s = "";    /* geçerli */
```

String'ler tek bir satır üzerine yazılmak zorundadır. Örneğin:

```
s = "this is a  
test";
```

Böyle bir yazım geçersizdir. Peki ya string bir satıra sığmıyorsa ne olacaktır? C'de yan yana iki string atomu (yani aralarında boşluk karakterlerinden başka bir karakter olmayan) derleme aşamasında derleyici tarafından birleştirilmektedir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char *s;

    s = "this is a"
        " test";
    printf("%s\n", s);

    return 0;
}
```

Yukarıdaki yazım tamamen geçerlidir.

Anahtar Notlar: C'de tek bir tes bölü ve hemen arkasından ENTER ('\n') karakteri "aşağıdaki satırı bu satırla birleştir" anlamına gelir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int count;

    cou\
nt = 10;

    printf("%d\n", count);

    return 0;
}
```

Yukarıdaki örnekte cou ve aşağısındaki satır birleştirilmiş ve count = 10 haline gelmiştir. Bu işlemin mümkün olması için '\n'den sonra hemen '\n' karakterinin gelmesi gerekir. Bu özelliği biz string'ler için de kullanabiliriz. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char *s;

    s = "ankara,\  
izmir";

    printf("%s\n", s);

    return 0;
}
```

String ifadeleri içerisinde '\ ' karakterleri '\ ' anlamına gelmez. Bunlar yanındak, karakterlerle başka bir karakter anlamına gelir. Dolayısıyla '\ ' karakterinin kendisi '\\ ' biçiminde ifade edilmelidir. Örneğin:

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



```
int main(void)
{
    char *path = "c:\\windows\\temp";

    printf("%s\n", path);

    return 0;
}
```

Adreslerin Karşılaştırılması

C'de adresler altı karşılaştırma operatörüyle de karşılaştırılabilir. Örneğin:

```
int *p, *q;

if (p > q) {
    ...
}
```

Ancak karşılaştırılan adreslerin aynı türden olması ya da birinin void türden olması gerekir. Fakat C'de rastgele iki adresin karşılaştırılması "tanımsız davranış" olarak değerlendirilir. İki adresin karşılaştırılabilmesi için bunların aynı dizinin ya da yapının elemanları olması gerekir. Örneğin:

```
#include <stdio.h>

void putsrev(char *s)
{
    char *end = s;
    while (*end != '\0')
        ++end;

    --end;
    while (end >= s) {
        putchar(*end);
        --end;
    }
    putchar('\n');
}

int main(void)
{
    char *s = "ankara";

    putsrev(s);

    return 0;
}
```

Gösterici Dizileri (Array of Pointers)

Her elemanı bir gösterici olan dizilere gösterici dizileri denir. Gösterici dizileri T bir tür belirtmek üzere,

```
T * <isim>[uzunluk];
```

biçiminde bildirilir. Örneğin:

```
int *a[10];
```

Bu dizinin her elemanı int türden bir göstericidir. Yani dizinin her elemanı int * türündendir. Örneğin:

```

#include <stdio.h>

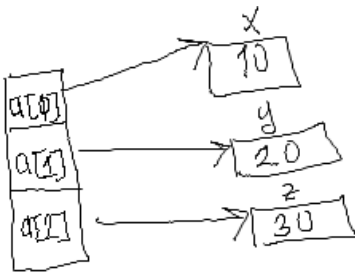
int main(void)
{
    int x = 10, y = 20, z = 30;
    int *a[3];
    int i;

    a[0] = &x;
    a[1] = &y;
    a[2] = &z;

    for (i = 0; i < 3; ++i)
        printf("%d\n", *a[i]);

    return 0;
}

```



C'de en fazla karşımıza çıkan char türden göstericileridir. Yazıların başlangıç adresleri char türden bir gösterici dizisine yerleştirilirse dizi yazıları tutar hale gelir. Örneğin:

```

#include <stdio.h>

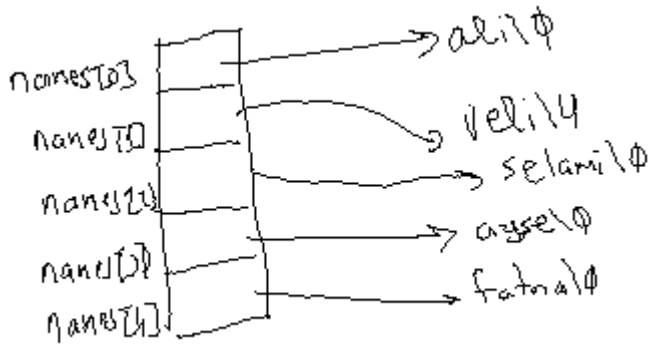
int main(void)
{
    char *names[5];
    int i;

    names[0] = "ali";
    names[1] = "veli";
    names[2] = "selami";
    names[3] = "ayse";
    names[4] = "fatma";

    for (i = 0; i < 5; ++i)
        printf("%s\n", names[i]);

    return 0;
}

```



Dizilere küme parantezleri içeriisinde ilkdeğer verebildiğimize göre aşağıdaki ilkdeğerleme de geçelidir:

```
#include <stdio.h>

int main(void)
{
    char *names[5] = { "ali", "veli", "selami", "ayse", "fatma" };
    int i;

    for (i = 0; i < 5; ++i)
        printf("%s\n", names[i]);

    return 0;
}
```

Bu dizinin sonuna NULL adres (Null karakter değil) yerleştirirsek, NULL adres görene kadar işlem yapabiliriz:

```
#include <stdio.h>

int main(void)
{
    char *names[] = { "ali", "veli", "selami", "siracettin", "ayse", "fatma", NULL };
    int i;

    for (i = 0; names[i] != NULL; ++i)
        printf("%s\n", names[i]);

    return 0;
}
```

Sınıf Çalışması: char türde bir gösterici dizisinin içerisine birtakım isimlerin adresleri yerleştirmiştir. Gösterici dizisinin sonunda NULL adres vardır. char türden geniş bir dizi açınız, bu isimleri aralarına ',' karakteri koyarak tek bir dizide (tabi sonu null karakter ile bitmeli) birleştiriniz. Sonra da bu diziyi puts ile yazdırınız.

Çözüm:

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

int main(void)
{
    char *names[] = { "ali", "veli", "selami", "siracettin", "ayse", "fatma", NULL };
    char str[100];
    int i;

    str[0] = '\0';

    for (i = 0; names[i] != NULL; ++i) {
        if (i != 0)
            strcat(str, ", ");
    }
}
```

```

        strcat(str, names[i]);
    }
    puts(str);

    return 0;
}

```

sprintf Fonksiyonu

sprintf fonksiyonu kullanım bakımından tamamen printf gibidir. Ancak ekrana yazmak yerine yazılacakları char türden bir dizinin içerisine yazar. Fonksiyonun birinci parametresi char türden dizinin adresidir. Fonksiyonun prototipi şöyledir:

```
int sprintf(char *s, char *format, ...);
```

Fonksiyonun birinci parametresi dizinin başlangıç adresini alır. Diğer parametreleri printf ile aynıdır. Örneğin:

```

#include <stdio.h>

int main(void)
{
    char s[100];
    int a = 10;
    double b = 12.345;

    sprintf(s, "a = %d, b = %f\n", a, b);
    puts(s);

    return 0;
}

```

Sınıf Çalışması: Üç basamaklı int türden bir sayıyı yazı biçiminde adresi aldığı char türden bir dizinin içerisine yerleştiren num_to_text isimli fonksiyonu yazınız.

```
char *num_to_text(int val, char *s);
```

Fonksiyon birinci parametresiyle belirtilen değeri yazıya dönüştürerek ikinci parametresi ile aldığı diziye yerleştirir. Fonksiyon ikinci parametresiyle aldığı adresin aynısına geri döner.

```

char s[100];

num_to_text(983, s);
puts(s);    /* dokuz yüz seksen üç */

```

Çözüm:

```

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

char *num_to_text(unsigned long val, char *s);

int main(void)
{
    char s[100];
    unsigned long val;

    for (;;) {
        printf("Sayı:");
        scanf("%lu", &val);
    }
}

```

```

        if (val == 123)
            break;
        num_to_text(val, s);
        puts(s);          /* dokuz yüz seksen üç */
    }

    return 0;
}

```

```

char *num_to_text(unsigned long val, char *s)
{
    char *ones[] = { "bir", "iki", "uc", "dort", "bes", "alti", "yedi", "sekiz", "dokuz" };
    char *tens[] = { "on", "yirmi", "otuz", "kirk", "elli", "altmis", "yetmis", "seksen", "doksan" };
    int one, ten, hundred;

    *s = '\0';
    hundred = val / 100;
    ten = val % 100 / 10;
    one = val % 10;

    if (!val) {
        strcat(s, "sifir");
        return;
    }

    if (hundred) {
        if (hundred != 1) {
            strcat(s, ones[hundred - 1]);
            strcat(s, " ");
        }
        strcat(s, "yuz");
    }
    if (ten) {
        if (hundred)
            strcat(s, " ");
        strcat(s, tens[ten - 1]);
    }
    if (one) {
        if (ten)
            strcat(s, " ");
        strcat(s, ones[one - 1]);
    }
}

```

Soruyu genelleştirerek de çözebiliriz:

```

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

char *num_to_text(unsigned long val, char *s);

int main(void)
{
    char s[100];
    unsigned long val;

    for (;;) {
        printf("Sayi:");
        scanf("%lu", &val);
        if (val == 123)
            break;
        num_to_text(val, s);
        puts(s);          /* dokuz yüz seksen üç */
    }
}

```

```

    return 0;
}

char *num_to_text(unsigned long val, char *s)
{
    char *ones[] = { "bir", "iki", "uc", "dort", "bes", "alti", "yedi", "sekiz", "dokuz" };
    char *tens[] = { "on", "yirmi", "otuz", "kirk", "elli", "altmis", "yetmis", "seksen", "doksan" };
    char *others[] = { "bin", "milyon", "milyar", "trilyon", "katrilyon", "katrilyar" };
    int one, ten, hundred;
    int digits3[5], temp;
    int i;

    *s = '\0';

    temp = val;
    for (i = 0; val; ++i) {
        digits3[i] = val % 1000;
        val /= 1000;
    }
    val = temp;
    if (!val) {
        strcat(s, "sifir");
        return;
    }
    while (--i >= 0) {
        hundred = digits3[i] / 100;
        ten = digits3[i] % 100 / 10;
        one = digits3[i] % 10;

        if (hundred) {
            if (hundred != 1) {
                strcat(s, ones[hundred - 1]);
                strcat(s, " ");
            }
            strcat(s, "yuz");
        }
        if (ten) {
            if (hundred)
                strcat(s, " ");
            strcat(s, tens[ten - 1]);
        }
        if (one) {
            if (ten)
                strcat(s, " ");
            if (i != 1 || digits3[i] != 1)
                strcat(s, ones[one - 1]);
        }

        if (i != 0) {
            strcat(s, " ");
            strcat(s, others[i - 1]);
            strcat(s, " ");
        }
    }
}

```

exit Fonksiyonu

exit prototipi <stdlib.h> dosyasında olan standart bir C fonksiyonudur. exit çağrıldığında program sonlanır. Yani sanki main bitmiş gibi bit etki oluşur. Bu durumda biz bir programı sonlandırmak için illa da akışın main'e geri dönmesini beklemek zorunda değiliz. Herhangi bir zaman herhangi bir fonksiyonda exit fonksiyonunu çağırırsak program sonlanır. Fonksiyonun prototipi şöyledir:

```
#include <stdlib.h>
```

```
void exit(int code);
```

Fonksiyon parametre olarak programın exit kodunu alır. İşletim sistemlerinde her sonlanan programın bir exit kodu vardır. Bu exit kodunun kaç olduğunun bir önemi yoktur. Bu kod işletim sistemine iletilir. İşletim sistemi de birisi (genellikle üst proses) isterse bunu ona verir. Ancak geleneksel olarak başarılı sonlanmalar için sıfır değeri, başarısız sonlanmalar için sıfır dışı değerler kullanılmaktadır. Hatta okunabilirliği artırmak için <stdlib.h> içerisinde iki sembolik sabit bildirilmiştir:

```
#define EXIT_SUCCESS    0
#define EXIT_FAILURE    1
```

Her ne kadar C standartlarında EXIT_SUCCESS ve EXIT_FAILURE sembolik sabitlerinin değerleri 0 ve 1 olarak kesin bir biçimde belirtilmemişse de derleyicilerin hemen hepsi EXIT_SUCCESS için 0 değerini, EXIT_FAILURE için 1 değerini kullanmaktadır.

main fonksiyonu sonlandığında program sonlanmaktadır. O halde main fonksiyonun da return işlemi programın sonlanmasına yol açar. İşte C standartlarına göre main bittiğinde main fonksiyonunun geri dönüş değeri alınarak exit çağrılır. Yani program aslında her zaman exit ile sonlandırılır. O halde bir C programının şöyle çalıştırıldığı düşünülmelidir:

```
exit(main());
```

Ayrıca C'de main fonksiyonuna özgü olmak üzere, main fonksiyonunda hiç return kullanılmazsa sanki return 0 kullanılmış gibi işlem görür.

Prosesin Adres Alanı Ve Bölümleri

İşletim sistemi çalıştıracağı programı ikincil bellekten alarak fiziksel belleğe yükler. Bu program için bellekte belli bir yer ayırmaktadır. Buna prosesin adres alanı (address space) denir. Prosesin adres alanı içerisinde şu bölümler vardır:

Stack Bölümü: Yerel ve parametre değişkenlerinin yaratıldığı ve yok edildiği alana stack denir. Stack'te nesnelere çok hızlı yaratılır ve yok edilir. Bloktan çıkıldığında o blok içerisinde tanımlanmış olan nesnelere hepsi stack'ten atılır.

Data Bölümü: İlkdeğer verilmiş global nesnelere yaratıldığı bölümdür. Data bölümü programın başından sonuna kadar kalır. Burada yaratım ve boşaltım yapılmaz. Program çalışırken artık bu bölümde tahsisat yapılamaz.

BSS Bölümü: İlkdeğer verilmemiş global nesnelere yaratıldığı bölümdür. BSS bölümü programın başından sonuna kadar kalır. Burada yaratım ve boşaltım yapılmaz. Program çalışırken artık bu bölümde tahsisat yapılamaz.

Heap Bölümü: Dinamik bellek fonksiyonlarıyla tahsisat yapılan bölümdür. Heap'te alan tahsisatı programın çalışma zamanı sırasında yapılır ve yok edilir. Tahsisat yapılırken ve serbest bırakılırken nerelerin boş ve dolu olduğu sorgulandığı için stack'teki tahsisata göre çok yavaştır. Ancak bu bölümde yapılan tahsisatlar bloktan çıkıldığında otomatik geri bırakılmaz. Yani yaşamaya devam eder.

Sistemlerde en küçük alan stack olma eğilimindedir. Bunu daha sonra data ve bss alanları sonra da heap alanı izler. En büyük alan heap olma eğilimindedir.

Dinamik Bellek Yönetimi (Dynamic Memory Management)

Bilindiği gibi C'de dizi uzunlukları sabit ifadesi biçimde verilmek zorundadır. Fakat bazen bir dizinin hangi uzunlukta açılacağı işin başında belli değildir. Ancak program çalışırken birtakım olaylar sonucunda dizinin uzunluğu bilinmektedir. İşte bu tür durumlarda programın çalışma zamanı sırasında dizi açmaya gereksinim duyulur. Programın çalışma zamanı sırasında bellek tahsis etme işlemine "dinamik bellek yönetimi" denilmektedir.

C'de dinamik bellek yönetimi ismine "dinamik bellek fonksiyonları" denilen bir grup standart C fonksiyonuyla gerçekleştirilir. Bunlar malloc, calloc, realloc ve free fonksiyonlarıdır.

Dinamik bellek fonksiyonları bellekte ismine "heap" denilen bir alanda tahsisatlarını yaparlar. Heap alanının büyüklüğü sistemden sisteme değişebilmektedir. C standartlarında söylenmese de modern sistemlerde her programın heap'i ayrıdır. Yani bir orgramdaki dinamik tahsisatların başka bir programa etkisi yoktur. Program bitinci modern sistemlerde o programın kullandığı heap alanı da sisteme iade edilir.

malloc Fonksiyonu

malloc programın çalışma zamanı sırasında parametresiyle belirtilen sayıda byte kadar ardışıl alanı tahsis eder. Tahsis ettiği alanın başlangıç adresiyle geri döner. Prototipi şöyledir:

```
#include <stdlib.h>
```

```
void *malloc(unsigned n);
```

malloc parametresiyle belirtilen miktarda byte kadar ardışıl alanı heap'te tahsis eder. Tahsis ettiği alanın başlangıç adresine geri döner. Eğer heap doluysa malloc tahsisatı yapamaz. Bu durumda NULL adrese geri döner. malloc ile tahsis edilen blokta çöp değerler vardır. Her ne kadar Windows gibi Linux gibi işletim sistemlerinde prosesin heap alanı çok büyükse de yine de her bellek tahsisatının başarısının kontrol edilmesi iyi bir tekniktir.

malloc fonksiyonunun bize verdiği adres herhangi türden bir dizi olarak kullanılabilir. Örneğin biz malloc ile biz 32 byte bir alan tahsis etmiş olalım. Orayı 4 elemanlı double bir dizi gibi de, 8 elemanlı int bir dizi gibi de, 32 elemanlı char türden bir dizi gibi de kullanabiliriz. Ne de olsa tüm diziler bellekte ardışıl tutulmaktadır. Örneğin:

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

int main(void)
{
    int n, i;
    int *pi;

    printf("Kac elemanli bir int dizi olusturmak istiyorsunuz?");
    scanf("%d", &n);

    pi = (int *)malloc(n * sizeof(int));
    if (pi == NULL) {
        printf("cannot allocate memory!\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < n; ++i) {
        printf("%d. elemani giriniz:", i + 1);
        scanf("%d", &pi[i]);
    }

    for (i = 0; i < n; ++i)
        printf("%d ", pi[i]);
    printf("\n");

    return 0;
}
```



```
}
```

calloc Fonksiyonu

Aslında calloc fonksiyonu taban bir fonksiyon değildir. Pek çok kütüphanede calloc fonksiyonu malloc fonksiyonunu çağırarak biçimde yazılmıştır. Prototipi şöyledir:

```
#include <stdlib.h>
```

```
void *calloc(unsigned count, unsigned size);
```

calloc iki parametresinin çarpımı kadar arşılı alan tahsis eder. Geleneksel olarak birinci parametre veri yapısının eleman sayısı, ikinci parametre ise bir elemanın byte uzunluğu olarak girilir. Örneğin biz 20 elemanlı bir int dizi tahsis edecek olsak birinci parametreye 20, ikinci parametresizeof(int) biçiminde girilir. calloc fonksiyonu malloc fonksiyonundan farklı olarak tahsis ettiği alanı sıfırlar. Halbuki malloc Yine fonksiyon başarı durumunda tahsis edilen alanın başlangıç adresine, başarısızlık durumunda NULL adrese geri döner. Örneğin:

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

int main(void)
{
    int *pim;
    int *pic;
    int i;

    if ((pim = (int *)malloc(10 * sizeof(int))) == NULL) {
        printf("cannot allocate memory!\n");
        exit(EXIT_FAILURE);
    }

    if ((pic = (int *)calloc(10, sizeof(int))) == NULL) {
        printf("cannot allocate memory!\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i)
        printf("%d ", pim[i]);
    printf("\n");

    for (i = 0; i < 10; ++i)
        printf("%d ", pic[i]);
    printf("\n");

    return 0;
}
```

calloc fonksiyonunu malloc kullanarak biz de yazabiliriz:

```
void *mycalloc(unsigned count, unsigned size)
{
    void *ptr;

    if ((ptr = malloc(count * size)) == NULL)
        return NULL;

    return memset(ptr, 0, count * size);
}
```

realloc Fonksiyonu

realloc daha önce tahsis edilmiş bir alanı büyütme ya da küçültme için kullanılır. Fonksiyonun prototipi şöyledir:

```
#include <stdlib.h>
```

```
void *realloc(void *ptr, unsigned newsize);
```

Fonksiyonun birinci parametresi daha önce tahsis edilmiş alanın başlangıç adresini alır. İkinci parametre toplam yeni uzunluğu belirtir. Fonksiyon tipik olarak şöyle çalışmaktadır: eğer blok büyütülüyorsa realloc önce tahsis edilmiş alanın hemen altında istenilen kadar ilave yer var mı diye bakar. Eğer varsa orayı da tahsis eder. Böylece blok yer değiştirmemiş olur. Eğer önceki alanın hemen altında toplam yeni uzunluğu karşılayacak kadar boş yer yoksa realloc heap alanının başka bir yerinde toplam yeni uzunluk kadar yeni bir yer arar. Bulursa burayı tahsis eder. Eski alandaki bilgileri buraya kopyalar. Eski alanı free hale getirir ve yeni alanın başlangıç adresiyle geri döner. Her ne kadar standartlarda realloc'un önce eski bloğun aşağısında yer arayıp aramayacağı belirtilmemişse de tipik çalışma böyledir. Örneğin realloc aslında eski bloğun altında boş yer olsa bile yine de bloğu taşıyabilir. Tabi realloc yine de başarısız olabilir. Yani ne eski yerin altında ne de herap'in başka bir yerinde toplam yeni uzunluğu karşılayacak kadar yer olmayabilir. Bu durumda realloc NULL adrese geri döner. Ayrıca bloğun küçültülmesi durumunda da blok yer değiştirebilir. Küçültme durumunda başarısızlık olmaz fakat blok yer değiştirebilir.

O halde realloc fonksiyonunun geri dönüş değeri her zaman değerlendirilmelidir. Örneğin:

```
p = malloc(10);  
...  
realloc(p, 20); /* hata! */
```

Burada realloc bloğun yerini değiştirebilir. Bu durumda p hala eski bloğu gösterir durumda olacaktır. Oysa şöyle yapılmalıydı:

```
p = malloc(10);  
...  
p = realloc(p, 20); /* doğru, fakat başarısızlık durumuna dikkat */
```

realloc başarısız olduğunda eski bloğu free hale getirmemektedir. eğer programa devam edilecekse eski bloğun kaybedilmemesi uygun olur:

```
p = malloc(10);  
...  
temp = realloc(p, 20); /* doğru */  
  
if (temp == NULL) {  
    ...  
}  
else  
    p = temp;  
...
```

realloc ile bloğun büyütüldüğü durumda bloğa ek yapılan alanda rastgele (çöp) değerler vardır. Yani burası da tıpkı malloc fonksiyonunda olduğu gibi sıfırlanmaz.

realloc fonksiyonunun birinci parametresi NULL girilirse realloc malloc gibi davranır. Yani:

```
p = malloc(n);
```

ile,

```
p = realloc(NULL, n);
```

aynı işleve sahiptir. realloc kullanılması için daha önce kesinlikle bloğun dinamik bellek fonksiyonlarıyla tahsis edilmiş olması gerekir. Normal bir diziyi biz realloc ile büyütüp küçültemeyiz. Örneğin:

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

#define BLOCK_ELEM_SIZE          5

int main(void)
{
    int *pi = NULL;
    int val, count, i;

    count = 0;
    for (;;) {
        printf("Sayı giriniz:");
        scanf("%d", &val);

        if (val == 0)
            break;

        if (count % BLOCK_ELEM_SIZE == 0) {
            pi = (int *)realloc(pi, (count + BLOCK_ELEM_SIZE) * sizeof(int) );
            if (pi == NULL) {
                printf("cannot allocate memory!..\n");
                exit(EXIT_FAILURE);
            }
        }
        pi[count++] = val;
    }

    for (i = 0; i < count; ++i)
        printf("%d ", pi[i]);
    printf("\n");

    return 0;
}
```

aynı işleve sahiptir. realloc kullanılması için daha önce kesinlikle bloğun dinamik bellek fonksiyonlarıyla tahsis edilmiş olması gerekir. Normal bir diziyi biz realloc ile büyütüp küçültemeyiz.

Sınıf Çalışması: Bir döngü içerisinde sürekli isim isteyiniz. Girilen isimleri dinamik olarak büyütülen char türden bir diziyeye aralarına ',' koyarak ekleyiniz. "exit" yazıldığında döngüden çıkınız ve tüm yazıyı tek bir puts ile yazdırınız.

Çözüm:

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

int main(void)
{
    char *names;
    char name[64];
    int count;

    count = 1;
    names = (char *)malloc(1);
```

```

if (names == NULL) {
    printf("cannot allocate memory!..\n");
    exit(EXIT_FAILURE);
}
*names = '\0';
for (;;) {
    printf("Isim giriniz:");
    gets(name);
    if (!strcmp(name, "exit"))
        break;
    if (count != 1)
        strcat(names, ", ");
    count += strlen(name) + 2;
    names = (char *)realloc(names, count);
    if (names == NULL) {
        printf("cannot allocate memory!\n");
        exit(EXIT_FAILURE);
    }
    strcat(names, name);
}

puts(names);

return 0;
}

```

Önce malloc kullanmadan realloc fonksiyonunu malloc gibi kullanmak isteyebiliriz. Ancak bu durumda ilk tahsisattan önce names dizisini başında null karakter bulunması gerekir. Şöyle çözüm söz konusu olabilir:

```

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

int main(void)
{
    char *names = NULL;
    char name[64];
    int count, len;

    count = 1;

    for (;;) {
        printf("Isim giriniz:");
        gets(name);
        if (!strcmp(name, "exit"))
            break;
        if (count != 1)
            strcat(names, ", ");
        len = strlen(name) + 2;
        names = (char *)realloc(names, count + len);
        if (names == NULL) {
            printf("cannot allocate memory!\n");
            exit(EXIT_FAILURE);
        }
        if (count == 1)
            *names = '\0';
        count += len;

        strcat(names, name);
    }

    puts(names);

    return 0;
}

```

free Fonksiyonu

free fonksiyonu daha önce malloc, calloc ya da realloc fonksiyonlarıyla tahsis edilmiş alanı serbest bırakmak için kullanılır. Yani free'den sonra artık o alan sani hiç tahsis edilmemiş gibi olur. Fonksiyununun prototipi şöyledir:

```
#include <stdlib.h>
```

```
void free(void *ptr)
```

Fonksiyon daha önce tahsis edilmiş olan bloğun başlangıç adresini alır ve o bloğun tamamını serbest bırakır. Bloğun bir kısmının serbest bırakılması söz konusu değildir.

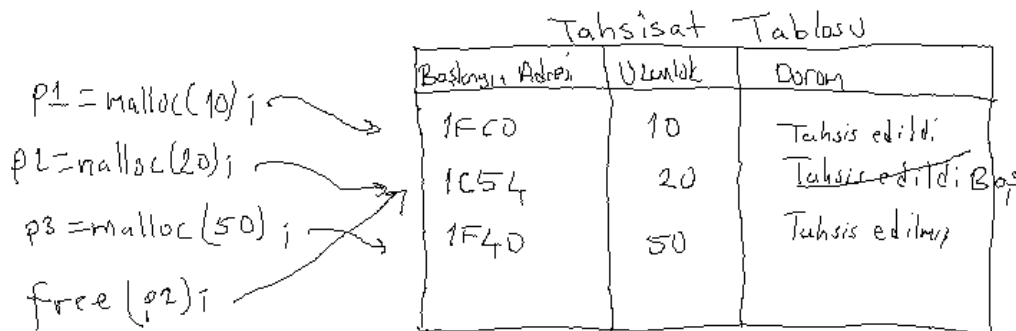
Modern sistemlerin hepsinde çalışan programların (proseslerin) heap alanları birbirlerinden farklıdır. Yani bir programdaki dinamik tahsisatlar diğer programın heap alanını azaltmazlar. Ancak yine de C standartlarında proseslerin heap alanlarının farklı olduğu belirtilmemiştir. Yani bazı küçük kapasiteli sistemlerde programdan çıkılsa bile heap tahsisatları kalıyor olabilir. Fakat mevcut sistemlerde bir program sonlandığında onun bütün tahsisatları otomatik free hale getirilmektedir.

Örneğin:

```
if ((ptr = malloc(100)) == NULL) {  
    printf("cannot allocate memory!..\n");  
    exit(EXIT_FAILURE);  
}  
...  
free(ptr);
```

Heap Organizasyonu Nasıl Yapılmaktadır?

Heap alanının hangi büyüklükte olacağı ve heap olarak belleğin neresinin kullanılacağı sistemden sisteme değişebilmektedir. Ancak dinamik bellek fonksiyonları kendi aralarında hangi bölgelerin tahsis edilmiş olduğunu bir tahsisat tablosu yoluyla tutmaktadır. Pek çok sistemde tahsisat algoritması olarak "boş blok bağlı listeleri" kullanılıyorsa da çok çeşitli tahsisat algoritmaları vardır. Burada bir fikir oluşsun diye bu fonksiyonların aşağıdaki gibi bir tahsisat tablosu oluşturduğunu düşünebiliriz:



Yapılar (Structures)

Elemanları bellekte ardışıl biçimde tutulan fakat farklı türlerden olabilen veri yapılarına "yapı (structure)" denilmektedir. Dizi ile yapılar birbirlerine çok benzerler. Her ikisinde de elemanları ardışıl olarak bellekte tutulmaktadır. Fakat dizi elemanları aynı türden olduğu halde yapı elemanları farklı türlerden olabilmektedir.

Yapılarla çalışırken önce yapıların bir şablonu bildirilir. Buna "yapı bildirimi" denir. Yapı bildirimi bellekte yer kaplamaz. Yani tanımlama değildir. Yapı bildirimini gören derleyici yapı elemanlarının türlerini ve isimlerini öğrenir. Daha sonra bu yapı türünden gerçek nesnel tanımlanır. Yapı bildirimini genel biçimi şöyledir:

```
struct <isim> {  
    <eleman bildirimleri>  
};
```

Örneğin:

```
struct DATE {  
    int day, month, year;  
};
```

```
struct COMPLEX {  
    double real;  
    double imag;  
};
```

```
struct SAMPE {  
    int a;  
    long b;  
    double c;  
};
```

Yapı isimlerini bazı programcılar büyük harflerle bazıları küçük harflerle harflendirmektedir. Biz kursumuzda yapı isimlerini büyük harflerle harflendireceğiz. Yapı bildirimi ile derleyici yapı elemanlarının isimlerini ve türlerini öğrenir.

Yapı bildirimleri yerel ya da global düzeyde yapılabilir. Eğer yapı bildirimi yerel düzeyde yapılmışsa o yapı ismi ancak o blokta kullanılabilir. Global olarak yapılmışsa her yerde kullanılabilir. Yapı bildirimleri genellikle global düzeyde yapılmaktadır. Çünkü çoğu kez yapıların farklı fonksiyonlarda kullanılması gerekmektedir.

Her yapı bildirimi aynı zamanda bir tür de belirtmektedir. Yani bir yapı bildirimi ile biz bir tür de oluşturmuş oluruz. Yapı bildirimleriyle oluşturulan türler struct anahtar sözcüğü ve yapı ismiyle ifade edilir. Örneğin struct DATE gibi, struct SAMPLE gibi.

Yapı bildirimi yapıldıktan sonra artık o yapı türünden nesnel tanımlanır. Örneğin:

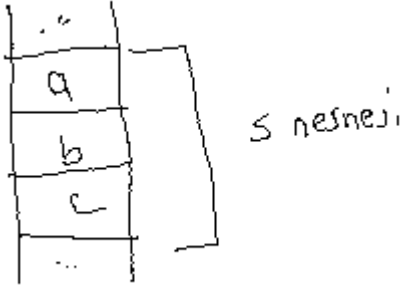
```
struct DATE d;  
struct SAMPLE s;
```

Burada d "struct DATE" türünden, s ise "struct SAMPLE" türündendir.

Yapılar bileşik nesnelerdir. Yani parçalardan oluşmaktadır. Örneğin:

```
struct SAMPLE {  
    int a;  
    long b;  
    double c;  
};
```

```
struct SAMPLE s;
```



Burada s a, b, ve c parçalarından oluşan nesne'nin bütünü temsil eder.

Yapı nesnelere genellikle bütünsel olarak kullanılmaz. Onların parçalarına erişilip parçaları kullanılır. Yapı elemanlarına erişmek için nokta operatörü kullanılmaktadır.

Nokta operatörü iki operandlı aritmetik bir operatördür. Nokta operatörünün sol tarafındaki operand yapı nesnesinin bütünü, sağ taraftaki operand onun bir elemanı olmak zorundadır. Bu operatör sol taraftaki yapının sağ taraftaki elemanına erişmekte kullanılır. Örneğin:

```
#include <stdio.h>

struct SAMPLE {
    int a;
    long b;
    double c;
};

int main(void)
{
    struct SAMPLE s;

    s.a = 10;
    s.b = 20;
    s.c = 12.4;

    printf("%d, %ld, %f\n", s.a, s.b, s.c);

    return 0;
}
```



Burada s nesnesi struct SAMPLE türündendir. s.a ifadesi int türündendir. s.a ifadesi "s nesnesinin a parçası" anlamına gelir. s.b ifadesi long türden, s.c ifadesi ise double türündendir.

C standartlarına göre yapı bildiriminde ilk yazılan eleman düşük adreste bulunacak biçimde eleman ardı şılığı oluşturulur. Yani örneğimizde s'in a parçası en düşük adrestedir. Bunu b parçası izler onu da c parçası izler.

Nokta operatörü öncelik tablosunun en yüksek düzeyinde soldan sağa grupta bulunur

() [] .	Soldan-Sağa
+ - ++ -- ! sizeof & *	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
...	...

Örneğin &s.a gibi bir ifadede s.a'nın adresi alınmaktadır. Çünkü nokta operatörü & operatöründen daha yüksek önceliklidir. Örneğin:

```
#include <stdio.h>

struct SAMPLE {
    int a;
    long b;
    double c;
};

int main(void)
{
    struct SAMPLE s;

    printf("%p, %p, %p\n", &s.a, &s.b, &s.c);

    return 0;
}
```

Anahtar Notlar: Anımsanacağı gibi Kernighan & Ritchie yazım tarzında iki operandlı operatörlerle operandlar arasında birer boşluk karakteri bırakılıyordu. Fakat nokta ve -> operatörleri iki operandlı olmasına karşın bunlarla operandlar arasında boşluk karakteri bırakılmamaktadır.

Bir yapı nesnesinin elemanlarına henüz değer atanmadıysa içerisinde ne vardır? İşte eğer yapı nesnesi yerelse onun tüm elemanlarında çöp değerler, global ise sıfır değerleri bulunur.

Bir yapı nesnesine küme parantezleri içerisinde ilkdeğer verebiliriz. (Tıpkı dizilerde olduğu gibi) Örneğin:

```
#include <stdio.h>

struct SAMPLE {
    int a;
    long b;
    double c;
};

int main(void)
{
    struct SAMPLE s = { 10, 20, 20.5 };

    printf("%d, %ld, %f\n", s.a, s.b, s.c);

    return 0;
}
```

Tıpkı dizilerde olduğu gibi yapının az sayıda elemanına ilkdeğer verebiliriz. Bu durumda geri kalan elemanlar derleyici tarafından sıfırlanır. Fakat yapının fazla sayıda elemanına ilkdeğer vermeye çalışırsak derleme aşamasında error oluşur.

Aynı türden iki yapı nesnesi birbirlerine atanabilir. Bu durumda yapının karşılıklı elemanları birbirlerine atanmaktadır. Örneğin:


```

#include <stdio.h>

struct SAMPLE {
    int a;
    long b;
    double c;
};

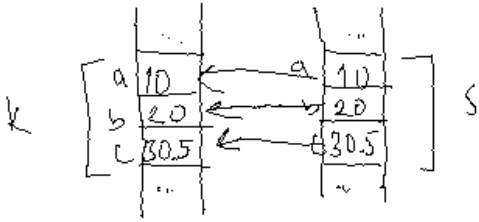
int main(void)
{
    struct SAMPLE s = { 10, 20, 30.5 };
    struct SAMPLE k;

    k = s;

    printf("%d, %ld, %f\n", s.a, s.b, s.c);
    printf("%d, %ld, %f\n", k.a, k.b, k.c);

    return 0;
}

```



Atama işleminde yapıların türlerinin aynı olması gerekir. Tür ise isme bağlıdır. İçi aynı olan farklı isimli yapıları birbirlerine atayamayız.

Yapı Elemanı Olarak Diziler

Bir dizi bir yapının elemanı olabilir. Bu durumda yapının dizi elemanına erişildiğinde bu ifade dizinin tamamını temsil eder, ifade içerisinde kullanıldığında ise yapı içerisindeki dizinin başlangıç adresini belirtir. Örneğin:

```

#include <stdio.h>

struct PERSON {
    char name[32];
    int no;
};

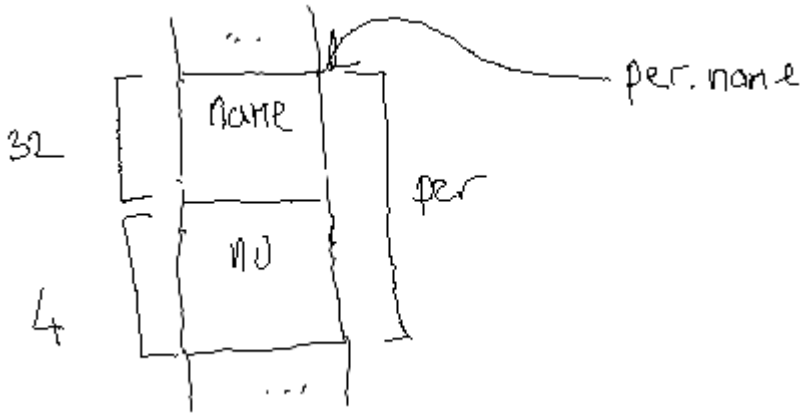
int main(void)
{
    struct PERSON per;

    printf("Adi soyadi:");
    gets(per.name);
    printf("No:");
    scanf("%d", &per.no);

    printf("%s, %d\n", per.name, per.no);

    return 0;
}

```



Böyle yapı nesnesine ilkdeğer de verebiliriz:

```
struct PERSON per = {"Kaan Aslan", 123};
```

Tabi buradaki iki tırnak ifadesi adres anlamına gelmez. Karakterlerin name dizisine kopyalanacağı anlamına gelir.

Nokta operatörle [] operatörünün aynı grupta soldan sağa öncelikli olduğuna dikkat ediniz. Örneğin:

```
per.name[3]
```

ifadesi per.name dizisinin 3. indeksli elemanı anlamına gelir. Yani:

per -----> struct PERSON türündendir

per.name -----> char * türündendir

per.name[3] -----> char türündendir

Örneğin:

```
#include <stdio.h>

struct PERSON {
    char name[32];
    int no;
};

int main(void)
{
    struct PERSON per = { "Kaan Aslan", 123 };
    char ch;

    ch = per.name[3];
    putchar(ch);

    return 0;
}
```

Yapı Elemanı Olarak Göstericilerin Kullanılması

Bir yapının elemanı bir gösterici olabilir. Örneğin:

```
struct PERSON {
    char *name;
    int no;
```

```
};
```

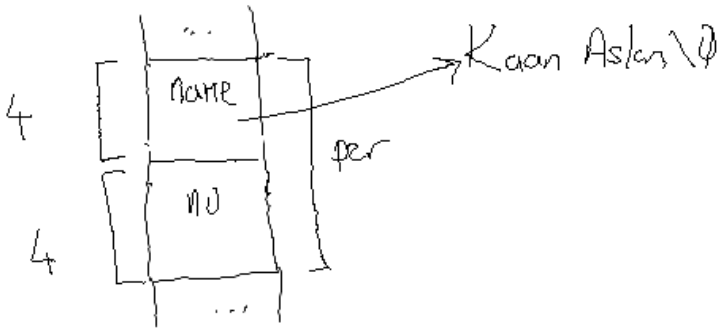
```
struct PERSON per;
```

Burada name elemanı bir göstericidir ve bu göstericinin tahsis edilmiş bir alanı gösteriyor olması gerekir. Örneğin:

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

```
struct PERSON {  
    char *name;  
    int no;  
};
```

```
int main(void)  
{  
    struct PERSON per;  
  
    per.name = "Kaan Aslan";  
    per.no = 123;  
  
    printf("%s, %d\n", per.name, per.no);  
  
    return 0;  
}
```

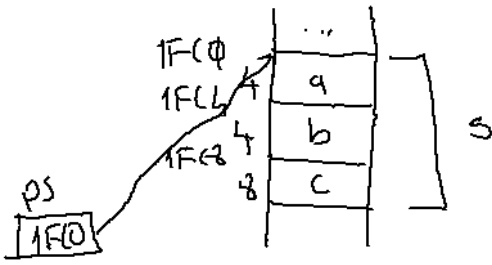


Yapı Türünden Adresler ve Göstericiler

Bir yapı nesnesinin adresi alınabilir. Bu durumda elde edilen adresin sayısal bileşeni tüm yapı nesnesinin bellekteki başlangıç adresi, tür bileşeni ise po yapı türündendir. Bir yapı nesnesinin adresi aynı türden bir yapı göstericisine atanabilir. Örneğin:

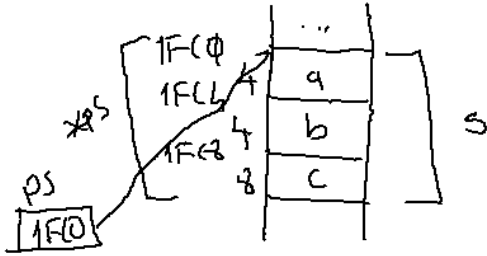
```
struct SAMPLE {  
    int a;  
    long b;  
    double c;  
};  
struct SAMPLE s;  
struct SAMPLE *ps;
```

```
ps = &s;
```



```
struct SAMPLE s;
struct SAMPLE *ps;
ps = &s;
```

Bir yapı göstericisi ya da bir yapı türünden adres * operatörüyle kullanılırsa yapı nesnesinin tamamına erişilir. Yani yukarıdaki örnekte *ps ile s aynıdır. Burada ps struct SAMPLE * türünden *ps ise struct SAMPLE türündedir.



```
struct SAMPLE s;
struct SAMPLE *ps;
ps = &s;
```

Yapı Göstericisi Yoluyla Yapı elemanlarına Erişilmesi

p bir yapı türünden adres a da bu yapının bir elemanı olmak üzere bu adresin gösterdiği yerdeki yapı nesnesinin a elemanına erişmek için (*p).a ifadesi kullanılır. *p.a ifadesi geçersizdir. Çünkü nokta operatörü * operatöründen daha yüksek öncelikli olduğu için bu ifadede önce p.a yapılmaya çalışılır ki nokta operatörünün solundaki operand geçersi olur. Çünkü nokta operatörünün solundaki operand yapı nesnesinin kendisi olmalıdır, adresi olmamalıdır.

Örneğin:

```
#include <stdio.h>

struct SAMPLE {
    int a;
    long b;
    double c;
};

int main(void)
{
    struct SAMPLE s;
    struct SAMPLE *ps;

    ps = &s;

    (*ps).a = 10;
    (*ps).b = 20;
    (*ps).c = 30;

    printf("%d, %ld, %f\n", (*ps).a, (*ps).b, (*ps).c);

    return 0;
}
```

Ok (Arrow) Operatörü

Ok operatörü -> karakterleriyle elde edilir. Ok operatörü iki operandlı aralık bir operatördür. Ok operatörünün

solundaki operand bir yapı türünden adres, sağındaki operand o yapının bir elemanı olmak zorundadır. On operatörü sol taraftaki operandla belirtilen adresteki yapı nesnesinin sağ taraftaki operandla belirtilen elemanına erişmekte kullanılır. Yani:

p->a

ile

(*p).a

tamamen eşdeğerdir.

Nokta operatörüyle ok operatörünün her ikisi de yapı elemanlarına erişmekte kullanılır. Nokta operatörü nesnenin kendisiyle ok operatörü adresiyle erişim sağlar. Ok operatörü öncelik tablosunun en üst düzeyinde soldan sağa grupta bulunur:

() [] . ->	Soldan-Sağa
+ - ++ -- ! sizeof & *	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
...	...

Örneğin:

```
#include <stdio.h>

struct SAMPLE {
    int a;
    long b;
    double c;
};

int main(void)
{
    struct SAMPLE s;
    struct SAMPLE *ps;

    ps = &s;

    ps->a = 10;
    ps->b = 20;
    ps->c = 30;

    printf("%d, %ld, %f\n", ps->a, ps->b, ps->c);

    return 0;
}
```

Örneğin:

```
#include <stdio.h>

struct POINT {
    int x, y;
};

int main(void)
```

```

{
    struct POINT pt;
    struct POINT *ppt;

    ppt = &pt;

    ppt->x = 10;
    ppt->y = 20;

    printf("(%d, %d)\n", ppt->x, ppt->y);

    return 0;
}

```

s bir yapı türünden nesne a da bu yapının bir elemanı olmak üzere &s->a ifadesi geçersizdir. Çünkü & operatörü -> operatöründen daha düşük önceliklidir. Ancak (&s)->a ifadesi geçerlidir ve s.a ile eşdeğerdir.

p bir yapı türünden adres ve a da bu yapının bir elemanı olmak üzere &p->a ifadesi geçerlidir ve bu ifade p göstericisinin gösterdiği yerdeki nesnenin a parçasının adresi anlamına gelir.

Örneğin:

```

#include <stdio.h>

struct PERSON {
    char name[32];
    int no;
};

int main(void)
{
    struct PERSON per = { "Ali Serce", 234 };
    struct PERSON *pper;

    pper = &per;

    printf("%s, %d\n", pper->name, pper->no);

    printf("%c\n", pper->name[2]);

    return 0;
}

```

Yapıların Fonksiyonlara Parametre Yoluyla Aktarılması

Yapıların fonksiyonlara aktarılmasında iki teknik kullanılır. Bunlardan birincisi nesnenin kopyalama yoluyla aktarılması tekniği, diğeri ise adres yoluyla aktarma tekniğidir. Nesnenin kendisinin aktarılması genel olarak kötü bir tekniktir. Adrtes yoluyla aktarma iyi bir tekniktir. Gerçekten de C'de yapı nesnelere hemen her zaman fonksiyonlara adres yoluyla aktarılır.

2) Yapı Nesnelerinin Fonksiyonlara Kopyalama Yoluyla Aktarılması: Bu yöntemde fonksiyonun parametre değişkeni bir yapı türünden yapı nesnesidir. Fonksiyon da aynı yapı türünden bir nesneyle çağrılır. Aynı türden iki yapı nesnesi atanabildiğine göre bu çağırma geçerlidir. Ancak burada aktarım kopyalama yoluyla yapılmaktadır. Örneğin:

```

#include <stdio.h>

struct PERSON {
    char name[32];
    int no;
};

```

```

void foo(struct PERSON per)
{
    printf("%s, %d\n", per.name, per.no);
}

int main(void)
{
    struct PERSON x = { "Ali Serce", 234 };

    foo(x);

    return 0;
}

```

Bu yöntem genel olarak kötü bir tekniktir. Çünkü büyük bir yapının bu yöntemde tüm elemanlarını tek tek aktarım sırasında fonksiyona kopyalanır. Üstelik bu yöntemde fonksiyon içerisinde artık biz orijinal nesneye erişemeyiz. Tabi eğer yapı çok küçükse bu teknik kötü bir teknik olmaz. Bu durumda kullanılabilir.

2) Yapı Nesnelerinin Fonksiyonlara Adres Yoluyla Aktarılması: Bu yöntemde fonksiyonun parametre değişkeni yapı türünden gösterici olur. Fonksiyon da aynı türden bir yapı nesnesinin adresiyle çağrılır. Bu kullanılması gereken doğru tekniktir. Örneğin:

```

#include <stdio.h>

struct PERSON {
    char name[32];
    int no;
};

void foo(struct PERSON *per)
{
    printf("%s, %d\n", per->name, per->no);
}

int main(void)
{
    struct PERSON x = { "Ali Serce", 234 };

    foo(&x);

    return 0;
}

```

Örneğin:

```

#include <stdio.h>

struct DATE {
    int day;
    int month;
    int year;
};

void get_date(struct DATE *date)
{
    printf("Gun:");
    scanf("%d", &date->day);

    printf("Ay:");
    scanf("%d", &date->month);

    printf("Yil:");
    scanf("%d", &date->year);
}

```

```

}

void disp_date(struct DATE *date)
{
    printf("%02d/%02d/%04d\n", date->day, date->month, date->year);
}

int main(void)
{
    struct DATE date;

    get_date(&date);
    disp_date(&date);

    return 0;
}

```

Örneğin:

```

#include <stdio.h>

struct COMPLEX {
    double real, imag;
};

void get_comp(struct COMPLEX *comp)
{
    printf("Gercek Kisim:");
    scanf("%lf", &comp->real);

    printf("Sanal Kisim:");
    scanf("%lf", &comp->imag);
}

void disp_comp(struct COMPLEX *comp)
{
    printf("%.0f+%.0fi\n", comp->real, comp->imag);
}

int main(void)
{
    struct COMPLEX z;

    get_comp(&z);
    disp_comp(&z);

    return 0;
}

```

Yapılara Neden Gereksinim Duyulmaktadır?

1) Yapılar olgular için mantıksal bir kap oluşturmaktadır. Yani tarih gibi, sanal sayılar gibi, şahıs bilgileri gibi birbirleriyle ilgili çok sayıda nesne bir yapıyla ifade edilirse daha kolay bir temsil yeteneği el edilir. Gerçekten de mantıksal bakımdan birbirleriyle bağlantılı nesnelerin yapılarla temsil edilmesi okunabilirliği ve anlaşılabilirliği artırmaktadır.

2) Bir fonksiyona çok sayıda parametre aktarılacaksa onların tek tek aktarılmaları hem yazılımsal olarak zordur. Hem anlaşılır olmaktan çıkar hem de yavaştır. Bunun yerine çok sayıda parametre bir yapıda toplanım tek bir parametre biçiminde fonksiyona geçirilebilir.

3) Bir fonksiyonun tek bir geri dönüş değeri vardır. Eğer fonksiyon dış dünyaya çok sayıda değer iletecekse bu yapılarla sağlanabilir. Örneğin iletilecek değerler bir yapıyla ifade edilir. Fonksiyona yapı nesnesinin adresi

geçirilir. Fonksiyon da bu nesnenin içeriğini doldurur.

Fonksiyonların Geri Dönüş Değerlerinin Yapı Olması Durumu

Bir fonksiyonun geri dönüş değeri bir yapı türünden olabilir. Bu durumda return ifadesi de aynı türden bir yapı nesnesi olmalıdır. Aslında bu yöntem de C'de çoğu kez (yani yapı büyükse) iyi teknik kabul edilmez. Çünkü burada return işlemi sırasında geçici nesneye bir kopyalama yapılmakta ve geri dönüş değerinin atanması sırasında da aynı sorun oluşmaktadır. Örneğin:

```
#include <stdio.h>

struct COMPLEX {
    double real, imag;
};

void disp_comp(struct COMPLEX *comp)
{
    printf("%.0f+%.0fi\n", comp->real, comp->imag);
}

struct COMPLEX add_comp(struct COMPLEX *z1, struct COMPLEX *z2)
{
    struct COMPLEX result;

    result.real = z1->real + z2->real;
    result.imag = z1->imag + z2->imag;

    return result;
}

int main(void)
{
    struct COMPLEX z1 = { 3, 2 };
    struct COMPLEX z2 = { 1, 4 };
    struct COMPLEX result;

    result = add_comp(&z1, &z2);
    disp_comp(&result);

    return 0;
}
```

Genellikle programcılar çoklu bilgiyi böyle almaktansa bir nesne verip fonksiyonun onun içerisine yerleştirme yapmasını tercih ederler. Örneğin:

```
#include <stdio.h>

struct COMPLEX {
    double real, imag;
};

void disp_comp(struct COMPLEX *comp)
{
    printf("%.0f+%.0fi\n", comp->real, comp->imag);
}

void add_comp(struct COMPLEX *z1, struct COMPLEX *z2, struct COMPLEX *result)
{
    result->real = z1->real + z2->real;
    result->imag = z1->imag + z2->imag;
}
```

```

int main(void)
{
    struct COMPLEX z1 = { 3, 2 };
    struct COMPLEX z2 = { 1, 4 };
    struct COMPLEX result;

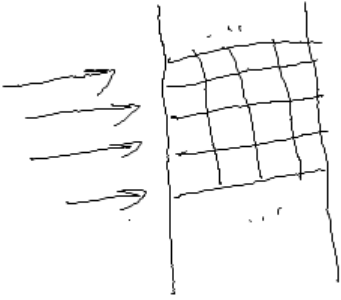
    add_comp(&z1, &z2, &result);
    disp_comp(&result);

    return 0;
}

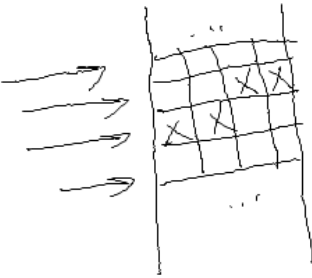
```

Yapılarda Hizalama (Alignment Kavramı)

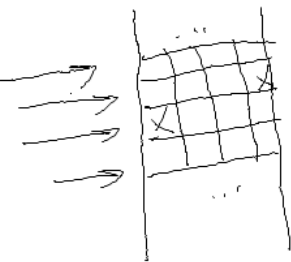
Modern 32 işlemcilerde bellek bağlantısı bir hamlede 4 byte bilgiyi çekecek biçimde yapılmıştır. Benzer biçimde 64 bit işlemcilerde de bellek bağlantısı bir hamlede 8 byte bilgiyi çekecek biçimde yapılmıştır. Böylece örneğin 32 bit işlemciler aslında 32 adres yoluna değil 30 adres yoluna sahiptir. Tabii makina komutları yine aynı biçimde byte ardeslemeli çalışmaktadır. Aşağıda 32 bit bir işlemcinin bellek erişimi resmedilmiştir:



Bu sistemlerde eğer 4 byte'lık bir bilgi (örneğin int türden bir bilgi) 4'ün katlarında değilse makina komutları görel olarak daha yavaş çalışmaktadır. Çünkü bu 4 byte'ı işlemci iki bus erişimiyle elde eder:



Tabii bir byte bir bilgiye kaçın katlarında olursa olsun tek bus hareketiyle erişilebilmektedir. Peki 2 byte'lık bilgiler? Bunların da 2'nin katlarında olması gerekir. Örneğin 2'nin katlarında olmayan 2 byte'lık (örneğin short türden bir nesne) bir nesne aşağıdaki gibi gösterilebilir:



İşte derleyiciler işlemcilerin böyle çalıştıklarını bildiği için makina komutları daha hızlı çalışsın diye 4 byte'lık

nesneleri 4'ün katlarına, 8 byte'lık nesneleri 8'in katlarına, 2 byte'lık nesneleri 2'nin katlarına, 1 byte'lık nesneleri 1'in katlarına yerleştirmektedir. Derleyiciler bunlara yerel değişkenler için ve global değişkenler için dikkat ederler. Yapı elemanları bellekte ardışıl olacağından ve ilk yazılan elemanın düşük adreste olması gerekeceğinden bu hizalama (alignment) yapılar için nasıl gerçekleştirilecektir? İşte derleyiciler yapı elemanlarının arasına boşluklar koyarak o elemanların belli adres katlarında olmasını sağlayabilmektedir. Örneğin:

```
struct SAMPLE {
    char a;
    int b;
    char c;
    int d;
};
struct SAMPLE s;
```

32 bit bir işlemcide bu yapı nesnesinin bellekte 10 byte yer kaplayacağı düşünülebilir. Ancak derleyiciler a ile b arasına ve c ile d arasına 3'er byte boşluk bırakarak int olan kısımların 4'ün katlarına gelmesini sağlayabilmektedir. Böylece bu yapı nesnesinin sizeof değerinin 16 çıkması programcıyı şaşırtmamalıdır. Örneğin:

```
#include <stdio.h>

struct SAMPLE {
    char a;
    int b;
    char c;
    int d;
};

int main(void)
{
    struct SAMPLE s;

    printf("%u\n", sizeof(s));    /* 16 */

    return 0;
}
```

Peki yukarıdaki yapıyı aşağıdaki gibi düzenleseydik ne olurdu?

```
#include <stdio.h>

struct SAMPLE {
    char a;
    char b;
    int c;
    int d;
};

int main(void)
{
    struct SAMPLE s;

    printf("%u\n", sizeof(s));    /* 12 */

    return 0;
}
```

Hizalama pek çok derleyicide derleyici seçeneklerinden yönetilebilmektedir. Örneğin Microsoft C derleyicilerinde hizalama Proje ayarlarında C-C++/Code Generation/Struct Member Alignment ile değiştirilebilmektedir. Eğer hizalama "1 byte hizalama moduna çekilirse derleyici yapı elemanlarını 1'in katlarına yerleştirmeye çalışır.

Dolayısıyla elemanlar arasında hiç boşluk bırakmaz.

C standartlarına hizalama kavramı bir kurala bağlanmamıştır. Standartlarda yalnızca derleyicinin yapı elemanları arasında boşluk bırakabileceği belirtilmiştir.

Peki derleyicinin yapı elemanları arasında boşluk bırakabilmesi erişimde bir soruna yol açar mı? Yanıt hayır. Çünkü derleyici nerelere boşluk bıraktığını bildiği için ona göre erişimi yapmaktadır. Örneğin aşağıdaki yapı için derleyici 4 byte hizalama kullanmış olsun:

```
struct SAMPLE {
    char a;
    int b;
    char c;
    int d;
};
```

Şimdi p göstericisinin bu yapıyı gösterdiğini düşünelim. Artık derleyici p->b ifadesiyle p adresinden 1 byte sonraya değil 4 byte sonraya erişir. Çünkü araya boşluk bıraktığını zaten kendisi bilmektedir. Yani yukarıdaki yapıyı biz şöyle düşünebiliriz:

```
struct SAMPLE {
    char a;
    char temp1, temp2, temp3;
    int b;
    char c;
    char temp4, temp5, temp6;
    int d;
};
```

Yapılar İçin Dinamik Tahsisat Yapılması

Mademki yapı elemanları bellekte ardışıl bir biçimde tutulmaktadır. O halde yapı nesnelere için de heap'te dinamik tahsisat yapılabilir. Dinamik tahsisat yaparken hizalama olasılığını göz önüne almak gerekir. Bu nedenle yapının byte uzunluğunun sizeof operatörü ile elde edilmesi uygun olur. sizeof operatörü derleyicinin o anda uyguladığı hizalamayı da hesaba katmaktadır. Örneğin:

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

struct PERSON {
    char name[32];
    int no;
};

int main(void)
{
    struct PERSON *per;

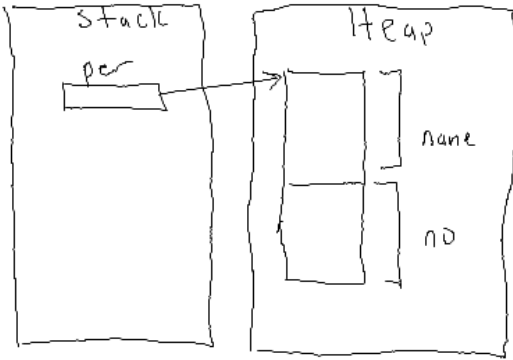
    per = (struct PERSON *)malloc(sizeof(struct PERSON));
    if (per == NULL) {
        printf("cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
    printf("Adi soyadi:");
    gets(per->name);
    printf("No:");
    scanf("%d", &per->no);

    printf("%s, %d\n", per->name, per->no);
}
```

```

free(per);
return 0;
}

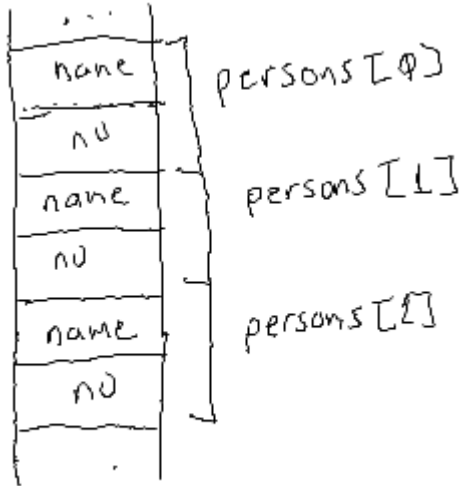
```



Yapı Dizileri

Her elemanı bir yapı nesnesi olan dizilere yapı dizileri (array of structures) denilmektedir. Yapı dizileri de normal dizilerde olduğu gibi bildirilir. Örneğin:

```
struct PERSON persons[3];
```



Örneğin:

```

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

struct PERSON {
    char name[32];
    int no;
};

int main(void)
{
    struct PERSON persons[3];
    int i;

    strcpy(persons[0].name, "Kaan Aslan");
    persons[0].no = 123;
}

```

```

strcpy(persons[1].name, "Ali Serce");
persons[1].no = 234;

strcpy(persons[2].name, "Necati Ergin");
persons[2].no = 678;

for (i = 0; i < 3; ++i)
    printf("%s, %d\n", persons[i].name, persons[i].no);

return 0;
}

```

Anımsanacağı gibi dizi isimleri tüm diziyi temsil etmektedir. Ancak bunlar işleme sokulduğunda derleyici tarafından dizinin başlangıç adresine dönüştürülürler. Yani biz dizi isimlerini kullandığımızda aslında o dizilerin başlangıç adreslerini kullanıyor oluruz. Dizi isimleri kullanıldığında nesne belirtmezler.

Biz bir yapı dizisinin ismini aynı yapı türünden bir yapı göstericisine atayabiliriz. Örneğin:

```

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

struct PERSON {
    char name[32];
    int no;
};

int main(void)
{
    struct PERSON persons[3], *per;
    int i;

    strcpy(persons[0].name, "Kaan Aslan");
    persons[0].no = 123;

    strcpy(persons[1].name, "Ali Serce");
    persons[1].no = 234;

    strcpy(persons[2].name, "Necati Ergin");
    persons[2].no = 678;

    per = persons;

    for (i = 0; i < 3; ++i) {
        printf("%s, %d\n", per->name, per->no);
        ++per;
    }

    return 0;
}

```

Bir yapı göstericini bir artırdığımızda göstericinin içerisindeki adres yapı uzunluğu kadar artmaktadır.

Yapı dizilerine de küme parantezleri ile ilkdeğer verilebilir. Bu durumda eleman olan her yapı nesnesi için de ayrı bir küme parantezi kullanılır. Örneğin:

```

struct PERSON persons[3] = {
    { "Kaan Aslan", 123 }, { "Ali Serce", 345 }, { "Necati Ergin", 654 }
};

```

Aslında bu tür durumlarda ,çteki küme parantezleri zorunlu değildir. Örneğin yukarıdaki ilk değer verme şöyle yapılabilirdi:

```
struct PERSON persons[3] = {
    "Kaan Aslan", 123 , "Ali Serce", 345, "Necati Ergin", 654
};
```

Fakat bu biçimde ilkdeğer verme hem okunabilirliği azaltmakta hem de aradaki bir değer unutulduğunda diğer tüm değerlerin yanlış elemanlara atanması gibi bir soruna yol açabilmektedir. Örneğin:

```
struct POINT {
    int x;
    int y;
};
struct POINT points[5] = {
    { 1, 2 }, { 4, 7 }, { 6, 8 }, { 7, 9 }, { 3, 4 }
};
```

Burada 2 değerinin yazılmadığını düşünelim:

```
struct POINT points[5] = {
    { 1 }, { 4, 7 }, { 6, 8 }, { 7, 9 }, { 3, 4 }
};
```

Artık dizinin ilk elemanının y değeri sıfır olacaktır. Ancak:

```
struct POINT points[5] = {
    1, 4, 7 , 6, 8, 7, 9, 3, 4
};
```

Burada tamamen bir kaydırma oluşmaktadır.

Örneğin:

```
#include <stdio.h>

struct POINT {
    int x;
    int y;
};

int main(void)
{
    struct POINT points[5] = {
        { 1, 2 }, { 4, 7 }, { 6, 8 }, { 7, 9 }, { 3, 4 }
    };
    int i;

    for (i = 0; i < sizeof(points) / sizeof(*points); ++i)
        printf("(%d, %d)\n", points[i].x, points[i].y);

    return 0;
}
```

Bir yapı dizisini de fonksiyona parametre yoluyla aktarabiliriz. Bunun için yine onun başlangıç adresini ve uzunluğunu fonksiyona geçiririz. Örneğin:

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

struct PERSON {
    char name[32];
    int no;
};
```

```

void sort_persons_byname(struct PERSON *per, int size);
void sort_persons_byno(struct PERSON *per, int size);
void disp_persons(struct PERSON *per, int size);

int main(void)
{
    struct PERSON persons[] = {
        { "Selami Hakyemez", 123 }, { "Ahmet Hamdi Tanpınar", 523 }, { "Hulusi Sen", 323 },
        { "Siracettin Bilyap", 654 }, { "Ali Ipek", 234 }
    };

    sort_persons_byname(persons, 5);
    disp_persons(persons, 5);
    printf("-----\n");
    sort_persons_byno(persons, 5);
    disp_persons(persons, 5);

    return 0;
}

void sort_persons_byname(struct PERSON *per, int size)
{
    int i, k;
    struct PERSON temp;

    for (i = 0; i < size - 1; ++i)
        for (k = 0; k < size - 1 - i; ++k)
            if (strcmp(per[k].name, per[k + 1].name) > 0) {
                temp = per[k];
                per[k] = per[k + 1];
                per[k + 1] = temp;
            }
}

void sort_persons_byno(struct PERSON *per, int size)
{
    int i, k;
    struct PERSON temp;

    for (i = 0; i < size - 1; ++i)
        for (k = 0; k < size - 1 - i; ++k)
            if (per[k].no > per[k + 1].no) {
                temp = per[k];
                per[k] = per[k + 1];
                per[k + 1] = temp;
            }
}

void disp_persons(struct PERSON *per, int size)
{
    int i;

    for (i = 0; i < size; ++i)
        printf("%s, %d\n", per[i].name, per[i].no);
}

```

Yapı Bildirimiyle Nesne Tanımlamasının Birlikte Yapılması

Bir yapı bildirilirken bildirim ';' ile kapatılmayıp bir değişken listesi de yazılırsa aynı zamanda o yapı türünden nesnelere de tanımlanmış olur. Örneğin:

```

struct POINT {
    int x, y;
}

```



```
} pt1, pt2;
```

Bu bildirim aşığıdakinden hiçbir farkı yoktur.

```
struct POINT {
    int x, y;
};

struct POINT pt1, pt2;
```

Örneğin:

```
struct PERSON {
    char name[32];
    int no;
} per = {"Kaan Aslan", 123}, *pper, persons[] = { {"Ali Serce", 123}, {"Ahmet
Altintarti", 345} };
```

Burada "per" struct PERSON türünden bir nesnedir, "pper" struct PERSON türünden bir göstericidir. persons ise struct PERSON türünden bir dizidir. Bu biçimde bildirilmiş değişkenler yapı global olarak bildirildiyse global, yerel olarak bildirildiyse yerel biçimdedir.

C standartlarına göre eğer yapı bildirimini ile aynı zamanda o yapı türünden n esne tanımlanıyorsa bu durumda yapıya isim verilmeyebilir. Örneğin:

```
struct {
    char name[32];
    int no;
} x, y;
```

Ancak eğer yapı türündne değişken tanımlanmıyorsa yapıya isim verilmek zorudur. Örneğin:

```
struct { /* geçersin */
    char name[32];
    int no;
};
```

İsimsiz yapıların hepsi farklı bir tür kabul edilir. Yani örneğin:

```
struct {
    char name[32];
    int no;
} x;
```

```
struct {
    char name[32];
    int no;
} y;
```

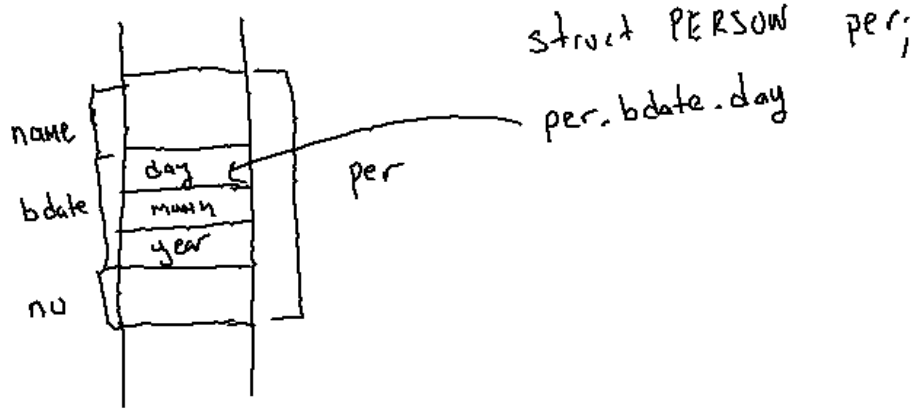
Burada x ve y derleyici tarafından aynı türden kabul edilmemektedir. Bu nedenle biz örneğin bunları birbirlerine atayamayız.

İç İç Yapı Bildirimleri

Bir yapının elemanları başka yapı türünden olabilir. Bu tür durumlara "iç içe yapı bildirimleri" denilmektedir. Örneğin:

```
struct DATE {  
    int day, month, year;  
};
```

```
struct PERSON {  
    char name[32];  
    struct DATE bdate;  
    int no;  
};
```



Örneğin:

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

```
struct DATE {  
    int day, month, year;  
};
```

```
struct PERSON {  
    char name[32];  
    struct DATE bdate;  
    int no;  
};
```

```
int main(void)
```

```
{  
    struct PERSON per = { "Sacit Bayram", { 12, 10, 1970 }, 2000 };  
  
    printf("%s, %d/%d/%d, %d\n", per.name, per.bdate.day, per.bdate.month, per.bdate.year, per.no);  
  
    return 0;  
}
```

İç içe yapılarda ilkdeğer verilirken iç yapı ayrıca küme parantezine alınabilir ya da alınmayabilir. Ancak okunabilirlik bakımından iç yapıya ilişkin değerlerin küme parantezleri içerisine alınması tavsiye edilir.

İç içe yapıların alternatif bildirim biçimi de vardır. Bu biçimde iç yapı dış yapının fiziksel olarak içerisinde bildirilir. Örneğin:

```
struct PERSON {  
    char name[32];  
    struct DATE {  
        int day, month, year;  
    } bdate;  
    int no;  
};
```

Burada dış yapı bildirimini içerisinde hem iç yapı bildirilmiş hem de o yapı türünden değişken bildirim yapılmıştır. Bu biçimdeki bildirimde içeride bildirilen yapı (örneğimizde struct DATE) ayrıca dış yapıdan bağımsız olarak da kullanılabilir. Örneğin:

```
struct PERSON per;  
struct DATE date; /* geçerli */
```

Yani bu iki iç içe yapı bildirim biçimi semantik olarak tamamen eşdeğerdir. Genel olarak birinci biçimin daha okunabilir olduğu söylenebilir.

typedef Bildirimleri

typedef anahtar sözcüğü bir tür isminin tam olarak yerini tutabilen alternatif isimler oluşturmak için kullanılır. typedef standartlarda bildirimdeki bir yer belirleyici (storage class specifier) biçiminde tanımlanmıştır. Bir bildirimde typedef anahtar sözcüğü eklenirse bildirimdeki değişken ismi o değişkenin türünü belirten tür ismi haline gelir. Örneğin:

```
int I;
```

Burada I bir değişken ismidir ve int türündendir. Şimdi bildirimde typedef ekleyelim:

```
typedef int I;
```

Burada I artık int türünü temsil eder. Örneğin:

```
int a, b, c;
```

ile,

```
I a, b, c;
```

tamamen eşdeğerdir. Örneğin:

```
char *STR;
```

Burada STR char * türünden bir değişkendir. Şimdi bildirim başına typedef yerleştirelim:

```
typedef char *STR;
```

Artık STR char * türünü temsil etmektedir. Yani:

```
char *s;
```

ile

```
STR s;
```

aynı anlamdadır.

Örneğin:

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

```

typedef int I;
typedef char *STR;

int main(void)
{
    I a;
    STR s = "Ankara";

    a = 10;
    printf("%d\n", a);
    printf("%s\n", s);

    return 0;
}

```

Örneğin:

```
int A, B, C;
```

Burada A, B ve C int türden değişken isimleridir. Bildirimin başına typedef getirelim:

```
typedef int A, B, C;
```

Burada artık, A, B ve C int türünü temsil eden tür ismidir. Örneğin:

```
int I, *PI;
```

Burada I int türünden, PI da int * türündendir. Bildirimin başına typedef getirelim:

```
typedef int I, *PI;
```

Burada artık I int türünü, PI da int * türünü temsil eder. Örneğin:

```

struct PERSON {
    char name[32];
    int no;
};

```

```
struct PERSON PER;
```

Burada PER struct PERSON türündendir. Şimdi bildirimin başına typedef getirelim:

```
typedef struct PERSON PER;
```

Artık PER struct PERSON türünü belirten bir tür ismi haline gelmiştir. Yani örneğin:

```
struct PERSON per;
```

ile,

```
PER per;
```

aynı anlamdadır. Örneğin:

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

```
struct PERSON {
    char name[32];
    int no;
};

typedef struct PERSON PER;

int main(void)
{
    PER per = { "Ali Serce", 123 };

    printf("%s, %d\n", per.name, per.no);

    return 0;
}
```

Örneğin aynı işlem şöyle de yapılabilirdi:

```
typedef struct PERSON {
    char name[32];
    int no;
} PER;

PER per = {"Ali Serce", 123};
```

Örneğin:

```
struct {
    char name[32];
    int no;
} PER;
```

Burada PER bildirilen yapı türünden bir nesnedir. Başına typedef getirelim:

```
typedef struct {
    char name[32];
    int no;
} PER;
```

Burada PER bu yapıyı temsil eder. Örneğin:

```
PER x;
PER y;
```

x ve y aynı türdendir.

Örneğin:

```
int ARY[3];
```

Burada ARY int[3] türündedir. Şimdi başına typedef getirelim:

```
typedef int ARY[3];
```

Burada artık ARY 3 elemanlı bir int dizi türü ismidir. Yani örneğin:

```
int a[3];
```

ile,

```
ARY a;
```

aynı anlamdadır. Örneğin:

```
#include <stdio.h>

typedef int ARY[3];

int main(void)
{
    ARY a = { 1, 2, 3 };
    int i;

    for (i = 0; i < 3; ++i)
        printf("%d\n", a[i]);

    return 0;
}
```

Örneğin:

```
void FOO(int a);
```

Burada FOO geri dönüş değeri void parametresi int olan bir fonksiyon prototipidir. Başına typedef getirelim:

```
typedef void FOO(int a);
```

Artık Foo geri dönüş değeri void parametresi int olan bir fonksiyon türünü temsil eder. Yani:

```
FOO x, y;
```

ile,

```
void x(int a);
void y(int a);
```

aynı anlamdadır.

typedef anahtar sözcüğünün bildirimde başa gelmesi zounlu değildir. Fakat değişken isminin solunda bulunmak zorudur. Örneğin:

```
int typedef I;          /* geçerli */
typedef int I;          /* geçerli */
int I typedef;          /* geçersiz */
```

typedef bildirimleri yerel ya da global düzeyde yapılabilir. Eğer yerel düzeyde yapılırsa o typedef ismi yalnızca o blokta kullanılabilir. Eğer global düzeyde yapılırsa her yerde kullanılabilir. typedef bildirimleri genellikle global düzeyde yapılmaktadır. Hatta çoğu kez programcılar typedef bildirimlerini başlık dosyalarının içerisine yerleştirir.

Bazı typedef bildirimleri ile yaratılmak istenen etki #define önişlemci komutuyla ya yaratılabilmektedir. Örneğin:

```
#define I int
```

```
I a, b, c;
```

Fakat pek çok tür #define ile oluşturulamaz. Örneğin:

```
typedef int ARY[3];
```

Bunun karşılığını #define ile oluşturamayız. Ya da örneğin:

```
#define I int *
```

```
I a, b;
```

Burada yalnızca a gösterici olur. Halbuki:

```
typedef int *I;
```

```
I a, b;
```

Burada hem a hem de b göstericidir. Karmaşık türler #define ile zaten oluşturulamaz. Ayrıca #define önişlemci aşamasına ilişkindir. Halbuki typedef derleme modülü tarafından değerlendirilir.

typedef Bildirimlerine Neden Gereksinim Duyulmaktadır?

1) typedef karmaşık türlerin daha kolay yazılmasını sağlar. Örneğin:

```
char *PARY[5];
```

```
PARY names = {"ali", "veli,", "selami", "ayse", "fatma"};
```

2) typedef okunabilirliği artırmak için de kullanılabilir. Yani bir türe onun kullanım amacına uygun bir isim verirsek kodu inceleyen kişiler onu daha iyi anlamlandırabilirler. Örneğin:

```
typedef int BOOL;
```

```
BOOL foo()
```

```
{  
    ...  
}
```

Burada biz foo'nun başarı ya da başarısızlık biçiminde bir geri dönüş değerine sahip olduğunu anlıyoruz.

3) typedef taşınabilirliği artırmak için kullanılabilir. Örneğin bir kütüphane oluşturan ekip bazı türlerin duruma göre sistemden sisteme değişebileceğini gördüğünden bunlara typedef ismi atayabilir.

```
/* lib.h */
```

```
typedef int HANDLE;
```

```
HANDLE foo(void);
```

Burada HANDLE int türündendir. Programcı fonksiyonu şöyle kullanır:

```
HANDLE h;  
  
h = foo();
```

Burada kütüphaneyi yazarlar başka bir sistem için HANDLE değerini long olarak typedef edebilirler. Biz de int yerine HANDLE kullandığımızda boşuna kodumuzu değiştirmek zorunda kalmayız. Yani değişiklikten kodumuz etkilenmemiş olur. Gerçekten de C/C++'ta yazılmış framework'ler ve kütüphanelerde typedef ile oluşturulmuş tür isimlerine çok sık rastlanır.

C'de Standart Olarak Bildirilen Bazı typedef İsimleri

C'de de bazı başlık dosyalarında bildirilmiş olan çeşitli typedef isimleri vardır. Bunlar taşınabilirlik sağlamak amacıyla oluşturulmuştur. Tabii bunları kullanmak için ilgili dosyasının include edilmesi gerekir. C'nin tüm standart typedef isimleri <stddef.h> isimli bir dosyada bildirilmiştir. Bu standart typedef isimlerinin hepsinin sonu xxx_t ile biter. Fakat bazı typedef isimleri ayrıca başka başlık dosyalarında da bildirilmiş durumdadır. Bunların bazıları aşağıda açıklanmaktadır:

size_t türü: Standartlara göre size_t işaretli bir tamsayı türü olarak typedef edilmek zorundadır. size_t tipik olarak ilgili sistemdeki bellek büyüklüğünü ifade edecek biçimde derleyicileri yazarlar tarafından uygun bir türe typedef edilir. Bazı sistemlerde size_t unsigned int olarak bazı sistemlerde de unsigned long int olarak karşımıza çıkabilmektedir. size_t C'de bazı fonksiyonların prototiplerinde kullanılmıştır. Örneğin malloc fonksiyonunun standartlarda belirtilen prototipi şöyledir:

```
void *malloc(size_t size);
```

Yani malloc fonksiyonunun parametresi o sistemde derleyiciyi yazarlar size_t türünü nasıl typedef etmişlerse o türdür. Örneğin strlen fonksiyonunun da orijinal prototipi şöyledir:

```
size_t strlen(const char *str);
```

C programcıları da kendi programlarında genellikle dizi uzunluklarını size_t ile ifade ederler. size_t türü <stddef.h> dosyasının yanı sıra <stdio.h>, <stdlib.h> gibi temel başlık dosyalarında typedef edilmiştir.

ptrdiff_t Türü: C'de iki adres toplanamaz. Ancak aynı türden iki adres çıkartılabilir. İşlem sonucu bir tamsayı türündendir. Öyle ki, önce adreslerin sayısal bileşenleri çıkartılır, sonuç adresin türünün uzunluğuna bölünür. Yani C'de aynı türden iki adresi çıkarttığımızda sonuç sanki aradaki eleman sayısını vermektedir. Örneğin:

```
int a[10];
```

Burada &a[5] - &a[0] ifadesi bize 5'i verir. Aslında a dizisi hangi türden olursa olsun bu işlem 5 değerini verecektir. Aynı türden iki adresi çıkarttığımızda sonuç hangi türdür? İşte standartlar sonucun ptrdiff_t türünden olacağını söyler. ptrdiff_t işaretli bir tamsayı türü olarak typedef edilmelidir. Pek çok derleyicide ptrdiff_t int ya da long türdür.

C'de standart olan birkaç tür daha ileride ele alınacaktır.

C'de Bildirim İşleminin Genel Biçimi

Daha önce biz bildirim genel biçimini şöyle görmüştük:

```
<tür> <değişken listesi>;
```


Aslında bildirimde genel biçimi şöyledir:

[yer belirleyici anahtar sözcükler] [tür niteleyici anahtar sözcükler] [tür belirten sözcükler] <değişken listesi>;

Görüldüğü gibi bildirimde türün yanı sıra yer belirleyiciler (storage class specifiers) ve tür niteleyiciler (type qualifiers) de kullanılmaktadır. Örneğin:

static const int a = 10, b = 20;

↑ yer belirleyici; tür niteleyici tür ↑ değişken listesi

Genel olarak bildirimde yer belirleyicileri, tür niteleyicileri ve tür belirten sözcükler yer değiştirebilir. Örneğin aşağıdaki iki bildirim eşdeğerdir:

```
static const int a = 10;
const int static a = 10;
```

Fakat genellikle programcılar önce yer belirleyici, sonra tür niteleyici en sonra da tür belirten anahtar sözcükleri yazmaktadır.

C'de yer belirleyici (storage class Specifiers) olarak aşağıdaki 5 anahtar sözcük kullanılabilir:

```
auto
static
register
extern
typedef
```

Aslında typedef anahtar sözcüğünün yer belirleyici işlevi yoktur. Tasarımcılar typedefi başka bir yere yerleştiremeyince buraya yerleştirmişlerdir. Bildirimde en fazla 1 tane yer belirleyici anahtar sözcük bulundurulabilir. C'de ayrıca iki de tür niteleyici (type qualifier) anahtar sözcük vardır:

```
const
volatile
```

C90'da eğer bildirimde tür belirten sözcük kullanılmamışsa fakat yer belirleyici ya da tür niteleyici anahtar sözcüklerden en az biri kullanılmışsa bu durumda default olarak tür belirten sözcük int kabul edilmektedir. Yani örneğin:

```
const a = 10;
```

bildirimi geçerlidir. Burada a'nın türü int'tir. Ancak bildirimde yalnızca değişken ismi bulundurulamaz. Örneğin:

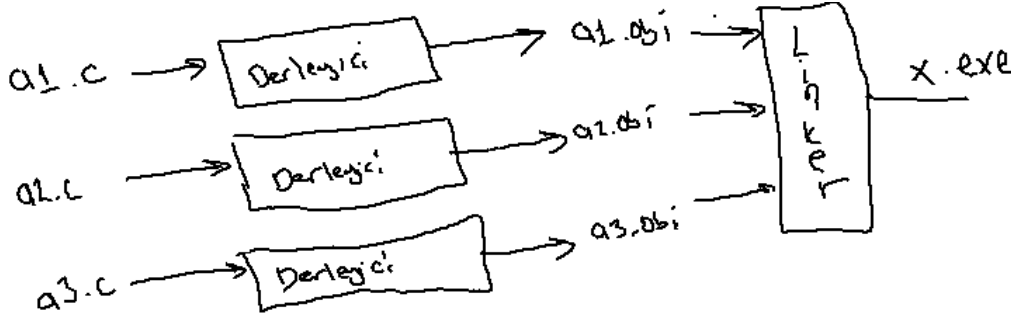
```
a, b = 20; /* geçersiz */
```

Böyle bir bildirim geçersizdir. Halbuki C99 ve C11'de bildirimde mutlaka tür belirten sözcük bulundurulmak zorundadır.

Yer Belirleyici Anahtar Sözcüklerin İşlevleri

auto Belirleyicisi: auto aslında işlevi olmayan bir anahtar sözcük durumundadır. auto belirleyicisi yerel değişkenlerle ya da parametre değişkenleriyle kullanılabilir. Global değişkenlerle kullanılamaz. auto değişkenin ilgili blok bittiğinde bellekten atılacağını (yani stack'te yer ayrılacağını) vurgulamaktadır.

extern Belirleyicisi: Aslında biz tek bir .c dosyasını derlemek zorunda değiliz. Bir proje çok uzun olabilir ve programcı bunu çeşitli dosyalara yaymak isteyebilir. Bu durumda .c dosyaları bağımsız olarak derlenir. Oluşan object modüller birlikte link edilerek çalıştırılabilen dosya (executable) oluşturulur. Örneğin projemiz a1.c a2.c a3.c biçiminde üç kaynak dosyadan oluşuyor olsun:



Görüldüğü gibi linker aslında birden fazla object modülü çalıştırılabilen dosyaya dönüştürebilmektedir. C standartlarında projeyi oluşturan kaynak dosyalara "translation unit" denilmektedir. Programcılar ise genellikle bu dosyalara "modül" demektedir.

Peki bir programı neden birden fazla kaynak dosyaya bölerek yazmak isteriz? Birinci neden karmaşıklığı azaltmaktır. Büyük bir projenin birden fazla dosyaya bölünmesi yönetilebilirliği artırır. Örneğin farklı kişiler farklı dosyalar üzerinde çalışabilirler. Sonra bunları birleştirebilirler. İkinci neden derleme zamanının azaltılmasıdır. Şöyle ki, örneğin 100000 satırlık bir projeyi tek bir kaynak dosya olarak organize etmiş olalım. Proje içerisindeki bir satırı bile değiştirsek tüm projeyi yeniden derlememiz gerekir. Halbuki bunu 10000 satırlık 10 dosyaya ayırsak yalnızca değişikliğin yapıldığı modülü derlememiz yeterli olur. Tabi hep berber her defasında bir link işlemi yine yapılmak zorundadır. Tabi link işlemi derleme işlemine göre daha kısa bir işlemdir.

Birden fazla modülle çalışma nasıl yapılmaktadır? Derleyicilerin komut satırlı biçimlerinde önce her modülü derleyip sonra birlikte link işlemi yapılabilir. gcc derleyicisinde -c seçeneği Microsoft'un cl derleyicisinde /c seçeneği yalnızca derleme yapar link etme işlemi yapmaz. Daha sonra yalnızca derlenmiş bu modüller link edilebilir. gcc programına object modülleri verirsek zaten o link işlemi yapmaktadır. Microsoft'un linker programı link.exe isimli programdır. Örneğin:

```
gcc -c a1.c
gcc -c a2.c
gcc -c a3.c
```

Buradan a1.o, a2.o ve a3.o dosyaları elde edilecektir. Sonra bunlar aşağıdaki gibi link edilebilirler:

```
gcc -o app a1.o a2.o a3.o
```

Bunun yerine aynı işlem şöyle de yapılabilir:

```
gcc -o app a1.c a2.c a3.c
```

Biz gcc'ye birden fazla .c dosyası verirsek gcc onları önce bağımsız olarak derler sonra object modülleri link

ederek çalıştırılabilir (executable) dosyayı oluşturur.

IDE'lerde projeye birden fazla kaynak dosya eklenirse bunlar otomatik olarak derlenir ve object modüller link edilerek çalıştırılabilir dosya elde edilir.

Birden fazla dosya ile çalışırken bir dosyadaki global değişkenin diğer dosyalardan da erişilebilmesi gerekir. Ancak her dosya bağımsız olarak derlendiği için diğer dosyada global olarak tanımlanmış değişkenleri derleyici tanımaz. Yani derleyici bir dosya derlenirken hangi dosyaların projenin parçalarını oluşturduğunu bilmemektedir. Projeyi oluşturan her kaynak dosya bağımsız olarak derlenir. İşte başka bir mütldde global olarak tanımlanmış olan bir değişken kullanılacaksa onun için extern bildirimini yapılması gerekir. Örneğin:

```
extern int g_x;
```

Global değişkenin yalnızca bir modülde global olarak tanımlanması fakat onun kullanıldığı tüm modüllerde extern olarak bildirilmesi gerekir. Global tanımlamanın hangi modülde yapıldığının bir önemi yoktur.

extern bir tanımlama değil bildirim işlemidir. extern belirleyicisini gören derleyici değişkenin başka bir modülde tanımlandığını anlar.. Ona göre kod üretir. Object modül içerisine linker için şöyle bir mesaj yazar: "Sayın linker bu değişken başka bir modülde tanımlanmış. Ben durumu idare ettim. Fakat sen proje link edilirken bu değişkenin hangi modülde tanımlandığını bul, bunu onla ilişkilendir. Bu değişken o değişkendir." Linker da link işlemine sokulan tüm modüllere bakarak bu birleştirmeyi yapar.

Burada dikkat edilmesi gerek durum global değişkenin yalnızca tek bir modülde global tanımlanmasının yapılması gerekliliğidir. eğer global değişken birden fazla modülde global tanımlanırsa her modül sorunsuz derlenir fakat link aşamasında aynı isimli birden fazla global tanımlama yapılmış olması nedeniyle error oluşur. Benzer biçimde global değişken her modülde extern bildirilirse yine her modül başarılı olarak derlenir ancak link aşamasında global tanımlamaya rastlanmadığından error oluşur.

Bir değişkenin başka modüllerden kullanılabilirliğine o değişkenin bağlama durumu (linkage) denilmektedir. Değişkenin bağlama durumu üç biçimde olabilir:

1) Dışsal Bağlama Durumu (External Linkage): Burada değişken başka modüllerden kullanılabilir. Global değişkenler default olarak dışsal bağlamaya sahiptir. Global değişkenleri extern ile bildirirsek de dışsal bağlama oluşur. Fakat extern bir tanımlama değildir. Global değişkenin tek bir tanımlanmasını olması gerektiği için tek bir modülde global tanımlama yapılmalı diğer modüllerde extern bildirilmelidir.

2) İçsel Bağlama Durumu (Internal Linkage): İçsel bağlamaya sahiğ değişkenler yalnızca bir modülün her yerinde kullanılabilirler. Ancak farklı modüllerden kullanılamazlar. Bir global değişken static anahtar sözcüğü ile bildirilirse onun bağlaması içsel bağlama olur. Artık diğer modüllerden o kullanılamaz.

Özetle C'de bir global değişken default olarak dışsal bağlamaya sahiptir. Ancak onu static anahtar sözcüğüyle bildirirsek içsel bağlamaya sahip olur.

3) Bağlamasız Durum (No Linkage): C'de yerel değişkenlerin ve parametre değişkenlerinin bağlaması yoktur.

Bir global değişken C'de aynı kaynak dosyada hem global olarak tanımlanmış olabilir hem de extern olarak bildirilmiş olabilir. Bu durum hata oluşturmaz. Değişkenin bağlama durumu dışsal bağlamadır. Bu durumda değişken için yer ayrılır. Yani extern bildirimini bir etkisi olmaz. Örneğin:

```
int g_a;          /* global tanımlama */
```

```
extern int g_a;  /* Gglobal bildirim */
```

Buradaki etki aşağıdakiyle eşdeğerdir:

```
int g_a;
```

Bir global değişken extern belirleyicisi kullanılarak bildirilmiş olsun ve ilkdeğer de verilmiş olsun. Örneğin:

```
extern int g_a = 10;
```

C standartlarına göre artık bu bildirim aynı zamanda bir tanımlamadır. Yani değişken için yer ayrılır. Başka bir deyişle yukarıdaki bildirim aşağıdaki ile eşdeğerdir:

```
int g_a = 10;
```

extern bildirimini yerel bir blokta yapılabilir. Tabi bu durumda o değişken yerel bir değişken olmaz. Başka modüldeki global bir değişken anlamındadır. Fakat bu isim yalnızca o blokta kullanılabilir. Örneğin:

```
void foo(void)
{
    extern int a;
    ...
}
```

```
void bar(void)
{
    a = 10;    /* geçersiz! */
}
```

Burada yerel bloktaki extern belirleyicisi ile bildirilmiş değişken aslında yerel bir değişken değildir. Dışsal bağlamaya sahip muhtemelen başka modülde tanımlanmış bir global değişkendir. Ama biz bu ismi yalnızca foo bloğunda kullanabiliriz.

C standartlarına göre farklı modüllerde aynı global değişkenin ilkdeğer verilmeden tanımlanması yapılabilir. Buna standartlarda "tentative declaration" denilmektedir. Bu durumda her ne kadar global değişkenin her modül için tanımlanması object modüle yazılsa da linker bunların tek bir kopyasını çalıştırılabilen dosyaya yazmaktadır. Dolayısıyla aşağıdaki kod geçerlidir:

```
a1.c          a2.c
-----
int g_a;      int g_a;
```

Ancak ilkdeğer verme durumunda bu geçerlilik kaybolur. Aşağıdaki derlemelerde link aşamasında error oluşur:

```
a1.c          a2.c
-----
int g_a=10;   int g_a=10;
```

Peki bir modülde tanımlanmış bir fonksiyonu diğer modüllerden çağırabilir miyiz? Fonksiyonlar teknik olarak global değişkenler gibidir. Fonksiyonların da default bağlama biçimleri dışsal bağlamadır. Yani onlar başka bir modülden kullanılabilirler. Ancak kullanmadan önce bir fonksiyonun tanımlanmasının ya da prototip bildirim derleyici tarafından görülmesi gerekir. Bu nedenele biz bir modülde tanımladığımız fonksiyonu diğer modülden kullanmadan önce fonksiyon prototipini bulundurmalıyız. eğer prototip bulundurmadafoo gibi bir fonksiyonu çağırırsak derleyici onun aşağıdaki gibi bir prototipe sahip olduğunu varsayar:

```
int foo();
```

Örneğin:

```
/* a1.c */

#include <stdio.h>

extern int g_x;

void foo(void);

int main(void)
{
    g_x = 10;
    foo();
    printf("%d\n", g_x);

    return 0;
}

/* a2.c */

#include <stdio.h>

int g_x;

void foo()
{
    g_x = 100;
}
```

Prototiplerde extern belirleyicisinin kullanılmasına gerek yoktur. Fakat kullanılsa da sorun oluşmaz.

Çok modüllü derlemelerde önerilen durum. Proje için ortak bir başlık dosyası hazırlamak ve tüm modüllerden bunu include etmektir. Bu başlık dosyasında yer kaplayan hiçbir tanımlama bulunmamalıdır. Yalnızca #'li önışlemci bildirimleri, extern bildirimler, fonksiyon prototipleri, yapı bildirimleri vs. bulunmalıdır. Global değişkenlerin ayrıca bir dosyada global tanımlaması yapılmalıdır. Örneğin:

```
/* project.h */

#ifndef PROJECT_H_
#define PROJECT_H_

void foo(void);
void bar(void);

extern int g_a;
extern int g_b;

#endif
```

```

/* a1.c */

#include <stdio.h>
#include "project.h"

int g_a;
int g_b;

int main(void)
{
    foo();
    bar();

    printf("%d, %d\n", g_a, g_b);

    return 0;
}

/* a2.c */

#include <stdio.h>
#include "project.h"

void foo()
{
    g_a = 10;
}

/* a3.c */

#include <stdio.h>
#include "project.h"

void bar()
{
    g_b = 20;
}

```

static Belirleyicisi: static belirleyicisi yer değişkenlerle ya da global değişkenlerle kullanılabilir. Fakat parametre değişkenleriyle kullanılamaz. Static yerel ve static global değişkenler tamamen farklı anlamlara gelirler.

static belirleyicisi yerel değişkenlerle kullanılırsa bu durum o değişkenin ömrünün static ömürlü olduğunu gösterir. Yani blok bittiğinde değişken yok edilmez. Program çalışmaya başladığında bellekte yer kaplar, program sonlanana kadar bellekte kalır. Örneğin:

```

#include <stdio.h>

void foo(void)
{
    static int count = 1;

    printf("%d\n", count);

    ++count;
}

int main(void)
{
    foo();
    foo();
    foo();
}

```

```
    return 0;
}
```

static yerel değişkene ilkdeğer verilmemişse içerisinde sıfır bulunur. Verilen ilkdeğer derleme aşasında değerlendirilir. Program yüklendiğinde static yerel değişken o değerle başlar. Artık bloğa her girişte bu ilkdeğer tekrar atanmaz.

Peki static yerel değişkenin global değişkenden ne farkı vardır? Bunların ömürleri aynı olmasına karşın faaliyet alanları farklıdır. static yerel değişkenler blok faaliyet alanı kuralına uyarlar. Global değişkenler ise dosya faaliyet alanına sahiptir.

Bir fonksiyonun static yerel bir nesnenin ya da dizinin adresiyle geri dönmesinde bir sakınca yoktur. Çünkü programdan çıkılsa bile o nesne yaşamaya devam etmektedir. Örneğin:

```
#include <stdio.h>

char *getname(void)
{
    static char name[64];

    printf("Adi soyadi:");
    gets(name);

    return name;
}

int main(void)
{
    char *nm;

    nm = getname();
    puts(nm);

    return 0;
}
```

static anahtar sözcüğü global değişkenlerle kullanılırsa bu durumda global değişkenin bağlama durumu "içsel bağlama (internal linkage)" olur. Yani böyle değişkenler diğer modüllerden hiçbir biçimde kullanılamazlar. Başka bir modülde aynı isimli global nesne olsa bile bu durum soruna yol açmaz.

Bir global değişken aynı modülde hem static hem de extern olarak bildirilebilir mi? Tabi bu durum saçmadır. Böyle bir şeyden kaçınmak gerekir. Ancak standartlara göre bir global değişken önce static ile sonra extern ile bildirilirse nesnenin içsel bağlamaya sahip olduğu kabul edilir. Fakat önce extern sonra static ile bildirilirse bu durum tanımsız davranışa yol açar.

register Belirleyicisi: register belirleyicisi global değişkenlerle kullanılamaz. Yerel ya da parametre değişkenleri ile kullanılabilir. Aslında uzunca bir süredir bu belirleyicinin ciddi bir işlevi kalmamıştır. Çünkü derleyicilerin kod optimizasyonları son derece gelişmiş durumdadır. Dolayısıyla bu belirleyicinin sağlayacağı fayda tartışmalıdır.

CPU içerisinde ismine "yazmaç (register)" denilen küçük bellek bölgeleri vardır. CPU ile RAM bağlantılıdır. CPU aritmetik ya da mantıksal ya da karşılaştırma işlemlerini yapmak için operandları yazmaçlardan alır. Dolayısıyla örneğin:

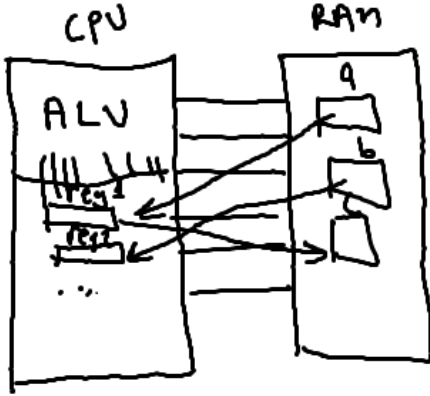
```
c = a + b;
```

gibi bir işlem aşağıdaki gibi makine komutlarıyla yapılmaktadır:

```

mov  reg1, a
mov  reg2, b
add  reg1, reg2
mov  c, reg1

```



işte register anahtar sözcüğü söz konusu değişkenin RAM yerine CPU içerisindeki yazmaçlarda tutulmasını istemek için kullanılmaktadır. Böylece bu değişkenler işleme sokulurken boşuna yeniden yazmaçlara alınmak zorunda kalınmaz. Şüphesiz çok yoğun işlem gören (örneğin döngü değişkenleri gibi) nesnelerin yazmaçlarda tutulması anlamlı olur.

register belirleyicisi bir emir değildir. Rica niteliğindedir. Yani biz bir nesneyi aşağıdaki gibi tanımlamış olalım:

```
register int x;
```

Derleyici bunu yazmaçlarda tutmak zorunda değildir. İsterse bunu normal nesnelere olduğu gibi yine RAM'de tutabilir. Bunun için de programcıya herhangi bir bildirimde (uyarı gibi) bulunmaz. Zaten modern derleyiciler gerekli gördükleri nesnelere gerektiği kadar yazmaçlarda tutacak biçimde kod optimizasyonu uygulamaktadır. Bu belirleyiciyi kullanmak yerine derleyicinin optimizasyon mekanizmasına güvenmek daha uygun gözükmektedir.

CPU'lardaki yazmaç miktarları CPU'dan CPU'ya değişebilmektedir. Genel olarak RISC tabanlı mimarilere sahip olan işlemcilerde CISC tabanlı mimarilere sahip olan işlemcilere göre daha fazla yazmaç bulunma eğilimindedir. Bir nesnenin programcının isteği ile yazmaçta tutulması eldeki yazmaç sayısını azaltabilir. Bunu farkedenden derleyici bu belirleyiciyi hiç dikkate almayabilir.

Tür Niteleyici Anahtar Sözcükler

C'de tür niteleyici iki anahtar sözcük vardır: const ve volatile. Bunların her ikisi de bildirimde kullanılabilir.

const Belirleyicisi

const bir nesne ilkdeğer verilerek bildirilir. Artık bundan sonra bu nesneye değer atanamaz. Eğer const bir nesneye değer atanmak istenirse derleme aşamında "error" oluşur. Örneğin:

```
const int x = 10;
```

```
printf("%d\n", x);    /* geçerli */
x = 20;              /* geçersiz! */
```

const bir nesnenin ilkdeğer verilmeden tanımlanması C++'ta geçersizdir. C'de geçerli olsa da anlamsızdır. (İlkdeğer verilmediğine göre ve bundan sonra değer de atanamayacağından böyle bir tanımlamanın bir anlamı olamaz.)

const anahtar sözcüğü deklarator ilişkin değildir. Türe ilişkin kabul edilir. Yani:

```
int const x = 10, y = 20;
```

Burada hem x hem de y const durumdadır.

Peki const belirleyicisinin ne faydası vardır? const temel olarak okunabilirliği ve anlaşılabilirliği kuvvetlendirmek için kullanılır. Örneğin:

```
const int personel_sayisi = 150;
```

gibi bir tanımlamada koda bakan kişi program içerisinde personel sayısının değişmeyeceğini anlayabilir. const belirleyicisi kodu yazanın yanlışlıkla bir nesneye atama yapmasını da engelleyebilmektedir. Bu durumda derleme aşamasında hatayla karşılaşan programcı hatasını anlayabilir. const belirleyicisi bazı durumlarda derleyicilere bazı optimizasyonları yapmaya olanak sağlayabilir.

const belirleyicisi ile #define sembolik sabitleri benzer amaçlarla kullanılabilir. Örneğin:

```
#define personel_sayisi 150
```

```
const int personel_sayisi = 120;
```

Fakat sembolik sabitler gerçekten bir sabittir. Halbuki const nesnelere gerçekten birer nesnedir. Örneğin biz const nesnelere adreslerini alabiliriz. Fakat sembolik sabitlerin adreslerini alamayız.

Bir dizinin tamamı const olabilir. Bu durumda biz dizinin hiçbir elemanını değiştiremeyiz. Örneğin:

```
const int a[5] = {10, 20, 30, 40, 50};
```

```
a[3] = 100;    /* geçersiz! */
```

const bir dizinin tüm elemanları const bir nesne gibidir.

Bir yapı nesnesi de const olabilir. Bu durumda yapının hiçbir elemanını değiştiremeyiz. Örneğin:

```
struct COMPLEX {  
    double real, imag;  
};
```

```
const struct COMPLEX z = {3, 2};
```

```
z.real = 5;    /* geçersiz! */
```

```
z.imag = 7;    /* geçersiz! */
```

Yapı bildiriminde belli elemanlar const yapılabilir. Bu durumda yapı nesnesinin tamamı const olmasa bile bu elemanları daha sonra değiştiremeyiz. Örneğin:

```
struct SAMPLE {  
    int a;  
    const int b;
```

```
};  
  
struct SAMPLE s = {10, 20};  
  
s.a = 30; /* geçerli */  
s.b = 40; /* geçerli değil! */
```

Genellikle programcılar yapı elemanlarını const yapmazlar. Çünkü bu durum karmaşıklığa yol açmaktadır.

const belirleyicisi en çok göstericilerle kullanılır. Böyle göstericilere const göstericiler denilmektedir.

const Göstericiler

Bir gösterici bildiriminde iki nesne söz konusu olur: Göstericinin kendisi ve onun gösterdiği yer. Bunların hangisinin const olacağına göre const göstericiler üçe ayrılır:

- 1) Kendisi değil gösterdiği yer const olan const göstericiler
- 2) Gösterdiği yer değil kendisi const olan const göstericiler
- 3) Hem kendisi hem de gösterdiği yer const olan const göstericiler

En çok kullanılan ve okunabilirlik bakımından en faydalı olan const göstericiler kendisi değil gösterdiği yer const olan const göstericilerdir. Bir const göstericinin hangi gruba girdiği const anahtar sözcüğünün yerine bakılarak karar verilir. Eğer const anahtar sözcüğü * atomunun solundaysa bu gösterdiği yer const olan const göstericidir. Örneğin:

```
const int *pi; /* int const *pi ile aynı */
```

Burada pi'nin kendisi const değildir. *pi yani pi'nin gösterdiği yer const durumdadır.

Eğer const anahtar sözcüğü * atomunun sağına getirilirse bu durumda göstericinin kendisi const olur. Örneğin:

```
int * const pi = ptr;
```

Burada const pi'yi nitelemektedir. Yani burada pi const durumdadır. *pi const değildir. Nihayet const anahtar sözcüğü iki yerde de bulunabilir:

```
const int * const pi = ptr;
```

Burada hem pi hem de *pi const durumdadır.

Kendisi Değil Gösterdiği Yer const Olan const Göstericiler

Bu tür göstericilerin bildirimlerinde const anahtar sözcüğü * atomunun soluna getirilir. Örneğin:

```
const int *pi;
```

Tabi aşağıdaki bildirim de eşdeğerdir:

```
int const *pi;
```

Burada göstericinin kendisi const değildir. Yani onun içerisine istediğimiz bir adresi istediğimiz zaman atayabiliriz. Ancak onun gösterdiği yere atama yapamayız. Örneğin:

```

int a = 10, b = 20;
const int *pi;

pi = &a;      /* geçerli, pi const değil */
*pi = 30;     /* Geçersiz göstericinin gösterdiği yer const */

pi = &b;      /* geçerli pi const değil */
*pi = 40;     /* Geçersiz göstericinin gösterdiği yer const */

```

Bu tür const göstericilerde tam olarak göstericinin gösterdiği yer const değil o gösterici ile erişilen her yer const biçimindedir. Örneğin:

```

int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
const int *pi;

```

```

pi = a;      /* geçerli */
pi[3] = 20;  /* geçersiz*/
pi[1000] = 30; /* Dizi taşması var ama derleme aşamasında yiner error oluşur */
pi += 500;   /* geçerli */
*pi = 40;    /* geçersiz! */

```

Gösterdiği yer const olan const göstericiler en çok karşımıza fonksiyon parametresi olarak çıkarlar. Örneğin:

```

void foo(const int *pi)
{
    /* ... */
}

```

Burada foo bizden int türden bir adres alır. Parametre const bir gösterici olduğuna göre foo aldığı adresteki nesneyi değiştirmez. Onu kullanabilir ama değiştirmez. Böylece fonksiyona bakan kişi fonksiyonun adresini verdiği nesneyi değiştirip değiştirmeyeceğini anlar. Gösterici parametrelerdeki const'luk çok önemlidir. Örneğin strlen fonksiyonunun orijinal prototipinde parametre const bir göstericidir:

```

size_t strlen(const char *str);

```

Bu prototipiten biz şunu anlarız: Biz strlen fonksiyonuna char türden bir dizinin başlangıç adresini vereceğiz. Fonksiyon bu adresteki yazıyı değiştirmeyecek. Yalnızca onu okuyacak.

Göstericilerde const'luk çok kararlı bir biçimde kullanılmalıdır. Böylece bir fonksiyonun parametresinin const olmayan bir gösterici olduğunu gördüğümüzde artık biz onun verdiğimiz adresteki bilgiyi değiştireceğini anlarız. (Çünkü değiştirmeyecek olsaydı zaten onu yazan göstericiyi const yapardı.). Çeşitli örnekler verelim:

- strcpy fonksiyonunun orijinal prototipinde birinci parametre const olmayan bir gösterici, ikinci parametre const bir göstericidir:

```

char *strcpy(char *dest, const char *source);

```

Bu prototipe baktığımızda bizim fonksiyona iki char türden adres göndereceğimiz anlaşılır. Fonksiyon birinci argümanla gönderdiğimiz adresteki bilgiyi değiştirecektir. Ancak ikinci argümanla gönderdiğimiz adresteki bilgiyi değiştirmeyecektir.

- strchr fonksiyonunun parametresi const bir göstericidir:

```
char *strchr(const char *s, int c);
```

- strcat fonksiyonunun birinci parametresi const olmayan bir gösterici ikinci parametresi const bir göstericidir:

```
char *strcat(char *dest, const char *source);
```

- Bir diziyi sıraya dizen sort isimli bir fonksiyon yazacak olsak onun parametresi const gösterici olamaz:

```
void sort(int *pi, size_t size);
```

- puts fonksiyonun parametresi const bir gösterici, gets fonksiyonun parametresi const olmayan bir göstericidir:

```
void puts(const char *str);  
char *gets(char *str);
```

Yapı nesnelere adresini alan fonksiyonların gösterici parametreleri const ise fonksiyonlar bizden aldıkları yapı nesnesinin elemanlarını kullanırlar fakat değiştirmezler. Fakat bu fonksiyonların gösterici parametreleri const olmayan bir göstericiyse bu durumda fonksiyon yapı nesnesinin içeriğini doldurmaktadır. Yani eğer fonksiyonun yapı gösterici parametresi const ise nesnenin içeriğini biz doldururuz o kullanır. eğer const olmayan bir gösterici ise o doldurur biz kullanırız. Çeşitli örnekler verelim:

- Örneğin sistemin zamanını alan GetSystemTime isimli bir fonksiyon olsun. Bu fonksiyon bizden TIME isimli bir yapı nesnesinin adresini alsın onun içerisine sistem zamanını yerleştiresin. Bu durumda fonksiyonun gösterici parametresi const olmayan bir gösterici olmalıdır:

```
struct TIME {  
    int hour, minute, second;  
};
```

```
void GetSystemTime(struct TIME *t);  
...  
struct TIME t;
```

```
GetSystemTime(&t);
```

- Sistem zamanını bizim belirlediğimiz zaman olacak biçimde değiştiren SetSystemTime fonksiyonunun yapı gösterici parametresi const gösterici olmalıdır:

```
void SetSystemTime(const TIME *t);
```

- Veritabanından birisini ismiyle arayıp bulursa bulunduğu kişinin bilgilerini PERSON isimli bir yapı nesnesine yerleştiren fonksiyonun parametrelerinin const'luk durumu şöyledir:

```
BOOL FindPerson(const char *name, struct PERSON *per);
```

- Seri portun setting değerlerini alıp bize veren fonksiyonun yapı gösterici parametresi const olmayan bir gösterici olmalıdır:

```
void GetSerialPortSettings(struct SERIAL *serial);
```

- Bunun tam tersi işlemi yapan fonksiyonun parametresi const gösterici olmalıdır:

```
void SetSerialPortSettings(const struct SERIAL *serial);
```

Fonksiyonların gösterici parametrelerindeki const'luga çok önem verilmelidir. Maalesef acemi C programcılarını bunlara önem vermezler ve acemilikleri hemen anlaşılır.

Fonksiyonun gösterici olmayan parametrelerinin const'luginun hiçbir önemi yoktur. Bunların const yapılması da amaç bakımından faydasız ve saçmadır. Örneğin:

```
void foo(const int x);
```

Burada bildirim geçersiz değildir. Ancak böyle const bildirim kimseye hiçbir faydası yoktur. Şöyle ki: Fonksiyonun parametre değişkenini değiştirip değiştirmeyeceği bizi ilgilendirmez. Önemli olan bizden aldığı nesnelerin değerlerini değiştirip değiştirmeyeceğidir. Bu nedenle parametrelerdeki const'luk yalnızca göstericiler söz konusu olduğunda anlamlı olur.

const Nesnelerin Adresleri ve const Adres Dönüştürmeleri

const bir nesnenin adresini bir göstericiye atayıp o nesneyi derleyiciyi kandırarak değiştirebilir miyiz? Örneğin:

```
const int a = 10;
int *pi;

pi = &a;    /* geçersiz */
*pi = 20;
```

İşte bu durumu engellemek için C'ye şöyle bir kural eklenmiştir: "const bir nesnenin adresi ancak gösterdiği yer const olan const bir göstericiye atanabilir." O halde yukarıdaki örnekte zaten biz a'nın adresini pi'ye atayamayız. Peki pi'yi const yapalım:

```
const int a = 10;
const int *pi;

pi = &a;    /* geçerli */
*pi = 20;  /* geçersiz */
```

Bir fonksiyonun adresini aldığı nesneyi değiştirmedeği halde onun parametresinin const gösterici yapılmamasının önemli bir dezavantajı vardır. Biz artık o fonksiyonu const bir nesnenin adresiyle çağıramayız. Örneğin bir int dizi içerisindeki sayıları ekrana yazdıran disp isimli fonksiyonun normal olarak gösterici parametresinin const olması gerekir. Fakat programcının kötü teknik uygulayarak bunu const yapmadığını düşünelim:

```
void disp(int *pi, size_t size);
```

Biz artık bu fonksiyonu const bir adresle çağıramayız:

```
const int a[] = {1, 2, 3};

disp(a, 3);    /* geçersiz */
```

const nesnelerinin adresleri const adreslerdir. const bir adres ancak gösterdiği yer const olan göstericilere atanabilir.

Tabi const olmayan bir adresin const bir göstericiye atanmasının hiçbir sakıncası yoktur. Örneğin biz const olmayan x nesnesinin adresini const bir göstericiye atayabiliriz. Çünkü zatan x'in değiştirilmemesi konusunda bir çekince yoktur. const anahtar sözcüğü türe ilişkin olduğu için başka bir deyişle T bir tür belirtmek üzere const T * türünden T * türüne otomatik dönüştürme yoktur. Fakat T * türünden const T * türüne otomatik dönüştürme vardır.

Bazen const bir adresin const'luğunu kaldırmak gerekebilir. Bu durumda const T * türünden T * türüne tür dönüştürme operatörü ile dönüştürme yapabiliriz. Yani const T * türünden T * türüne otomatik dönüştürme yoktur fakat tür dönüştürme operatörü ile dönüştürme yapılabilir.

C'de (ve tabii C++'ta) const bir nesnenin adresini tür dönüştürme operatörüyle const olmayan bir göstericiye atayıp orayı değiştirmeye çalışırsak bu durum tanımsız davranışa (undefined behavior) yol açar. (Gerçekten de statik ömürlü const nesnelerin Windows ve Linux'ta güncellenmeye çalışılması programın çökmesine yol açmaktadır.)
Örneğin:

```
#include <stdio.h>

int main(void)
{
    const int a = 20;
    int b = 30;
    int *pi;
    const int *pci;

    pi = &a;          /* geçersiz */
    pi = (int *)&a;  /* geçerli */
    *pi = 40;        /* geçerli ama tanımsız davranışa yol açar */

    pci = &b;        /* geçerli */
    pi = pci;       /* geçersiz */
    pi = (int *)pci; /* geçerli */
    *pi = 50;       /* normal yani tanımsız davranış değil */

    return 0;
}
```

Örneğin:

```
#include <stdio.h>

const int g_a = 10;

int main(void)
{
    int *pi;

    pi = (int *)&g_a; /* geçerli */
    *pi = 20;         /* tanımsız davranış, Windows'ta ve Linux'ta program çöker */
    printf("%d\n", g_a);

    return 0;
}
```

Gösterdiği Yer Değil Kendisi const Olan const Göstericiler

Daha önce de belirtildiği gibi böyle const göstericiler seyrek kullanılmaktadır. Bunların bildirimlerinde const anahtar sözcüğü * atomunun sağına getirilir. Örneğin:

```
char s[50];
char d[50];

char * const pc = s;

*pc = 'x';          /* geçerli */
pc = d;            /* geçersiz */
```

Hem Gösterdiği Yer Hem de Kendisi const Olan const Göstericiler

Bunlar hepten seyrek kullanılırlar. Burada const belirleyicisi hem * atomunun soluna hem de sağına getirilir. Örneğin:

```
char s[50];
char d[50];

const char * const pc = s;

pc = d;          /* geçerli değil */
*pc = 'x';       /* geçerli değil */
```

Derleyicilerin Kod Optimizasyonları ve volatile Belirleyicisi

Kodun çalışmasında hiçbir değişikliğin olmaması koşuluyla derleyiciler programcının yazdığı kodu yeniden düzenleyebilirler. Bundan amaç kodun daha hızlı çalışması (speed optimization) ya da daha az yer kaplamasıdır (size optimization). Derleyicinin kodun daha hızlı çalışması ya da daha az yer kaplaması için kodu yeniden düzenlemesine kod optimizasyonu denilmektedir.

Pek çok kod optimizasyon teması vardır. Örneğin "ortak alt ifadelerin elimine edilmesi (common subexpression elimination)" denilen teknikte derleyici ortak alt ifadeleri yeniden yapmamak için bir yerde (muhtemelen bir yazmaçta) toplar. Örneğin:

```
x = a + b + c;
y = a + b + d;
z = a + b + e;
```

deyimlerinde derleyici her defasında a + b işlemini yapmamak için bunun toplamını bir yazmaçta saklayabilir ve her defasında onu kullanabilir:

```
reg = a + b;
x = reg + c;
y = reg + d;
z = reg + e;
```

Derleyici kullanılmayan nesnelere sanki tanımlanmamış gibi elemine edebilir. Döngü içerisindeki gereksiz ifadeleri döngünü dışına çıkartabilir vs.

Derleyiciler nesnelere ikide bir yeniden yazmaçlara çekmemek için belli büre yazmaçlarda tutabilirler. Örneğin:

```
x = g_a + 10;
y = g_a + 20;
z = g_a + 30;
```

Burada derleyici her defasında a'yı yeniden RAM'den yazmaca çekmeyebilir. Onu bir kez çekip sonraki erişimlerde o yazmacı kullanabilir. Fakat bzen özellikle çok thread'li uygulamalarda ya da kesme (interrupt) uygulamalarında bu istenmeyebilir. Şöyle ki:

$x = g - a + 10i$
 $y = g - a + 20i$
 $z = g - a + 30i$

Oksirden
 g_a deęiştirilmiř olsun

Derleyicinin ürettięi kod

```

mov reg4, g-a
add reg2, reg4
mov x, reg2
mov reg2, reg4
add reg2, 20
mov y, reg2
mov reg4, reg4
add reg2, 30
mov z, reg2
  
```

Bizden
 g_a deęiştirilmiř dięer

Okla gösterilen noktada başka bir kaynak tarafından g_a deęiştirildięinde derleyicinin ürettięi kod bu deęiřiklięi anlayamaz. Halbuki bazı durumlarda programcı bu deęiřiklięin kod tarafından anlaşılmasını isteyebilir. İřte volatile belirleyicisi buna saęlamak için kullanılmaktadır:

```
volatile int g_a;
```

volatile bir nesne kullanıldıęında derleyici her zaman onu yeniden belleęe bařvurarak alır. Yani onu optimizasyon amaçlı yazmaçlarda bekletmez. Örneęin ařaęıdaki bir döngü söz konusu olsun:

```

while (g - flog) {
    //
    //
    //
}
  
```

Başka bir akıřın (örneęin bir thread'in) bu g_flag nesnesini 0 yaparak döngüyü sonlandırabilme olanaęı olduęunu düşünelim. eęer derleyici g_flag nesnesini bir kez yazmaca çekip hep onu yazmaçtan kullanırsa dięer thread g_flag nesnesini sıfıra çekse bile döngüden çıkılmaz. Bunu engellemek için g_flag volatile yapılmalıdır:

```
volatile int g_flag;
```

Nesnenin volatile olduęunu gören derleyici artık o nesnenin deęerini her zaman belleęe bařvurarak elde eder.

volatile bir gösterici söz konusu olabilir. Aslında tıpkı const'luk durumunda olduęu gibi volatile belirleyicisinde de gösterici üç biçimde olabilir:

- 1) Kendisi deęil gösterildięi yer volatile olan volatile göstericiler
- 2) Gösterildięi yer deęil kendisi volatile olan volatile göstericiler
- 3) Hem kendisi hem de gösterildięi yer volatile olan volatile göstericiler

Yine en çok kullanılan volatile göstericiler "kendisi deęil gösterildięi yer volatile olan volatile göstericilerdir". Yine

volatile anahtar sözcüğünün getirildiği yere göre bunlar değişebilmektedir:

```
volatile int *pi;  
int * volatile pi;  
volatile int * volatile pi;
```

Gösterdiği yer volatile olan bir göstgericinin anlamı nedir?

```
volatile int *pi;
```

Burada *pi ya da pi[n] gibi ifadelerle erişen nesnelere geçici süre yazmaçlarda bekletilmezler. Bunlar her defasında yeniden belleğe başvurularak oradan alınırlar. volatile bir nesnenin adresi ancak gösterdiği yer volatile olan bir göstericiye atanabilir. Örneğin:

```
volatile int a;  
int *pi;  
volatile int *piv;
```

```
pi = &a;    /* geçerli değil */  
piv = &a;   /* geçerli */
```

volatile olma durumu const olma durumuyla aynı mantığa sahiptir. Yani T bir tür belirtmek üzere volatile T * türünden T * türüne otomatik dönüştürme yoktur ancak T * türünden volatile T * türüne doğrudan dönüştürme vardır. Örneğin:

```
int a;  
volatile int *piv;
```

```
piv = &a;    /* geçerli */
```

Tabii tür dönüştürme operatörüyle volatile'lığı atarak volatile T * türünden T * türüne dönüştürme yapılabilir.

const ve volatile belirleyicilerine kısaca İngilizce "cv qualifiers" denilmektedir. const ve volatile birlikte kullanılabilir. Örneğin:

```
const volatile int g_a = 10;
```

enum Türleri ve Sabitleri

enum türleri C'de bir grup sembolik sabiti kolay bir biçimde oluşturmak için kullanılmaktadır. enum bildiriminin genel biçimi şöyledir:

```
enum [isim] {<enum sabit listesi>;
```

enum sabit listesi ',' atomuyla ayrılan isimlerden oluşur. Örneğin:

```
enum COLORS {Red, Green, Blue, Yellow};  
enum DAYS {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
```

İlk enum sabitinin değeri sıfırdır. Sonraki her sabit öncekinden bir fazladır. Örneğin:

ENUM COLORS { Red, Green, Blue, Yellow};
↑ ↑ ↑ ↑
0 1 2 3

Enum sabitlerine İngilizce "enumerator" da denilmektedir. Enum sabitleri nesne belirtmezler. Derleyici enum sabitleri kullanıldığında onlar yerine koda gerçekten o sabitin değerini yerleştirir. Yani enum sabitleri #define oluşturulmuş sembolik sabitlere benzetilebilir. Ancak #define önişlemciye ilişkindir. Halbuki enum türleri ve sabitleri derleme aşamasına ilişkindir.

Enum bildirimi global ya da yerel düzeyde yapılabilir. Bu durumda enum sabitlerinin de faaliyet alanı global ya da yerel olmaktadır. Örneğin:

```
#include <stdio.h>

enum {Red, Green, Blue};

int main(void)
{
    int color, city;
    enum {Adana, Ankara, Eskisehir};

    color = Green;          /* geçerli */
    printf("%d\n", color);

    city = Ankara;         /* geçerli */
    printf("%d\n", city);

    return 0;
}

void Foo()
{
    int color, city;

    color = Green;          /* geçerli */
    printf("%d\n", color);

    city = Ankara;         /* geçersiz */
    printf("%d\n", city);
}
```

Bir enum sabitine '=' atomu ile bir değer verilebilir. Bu durumda sonrakiler onu izler. Aynı değere sahip birden fazla enum sabiti bulunabilir. Örneğin:

```
enum COLORS {Red = 10, Green, Blue = -5, Yellow, Magenta = 9, Purple};
```

Burada Red = 10, Green = 11, Blue = -5, Yellow = -4, Magenta = 9, Purple = 10 değerlerini almaktadır.

C'de enum türleri int türü gibi değerlendirilir. Bu durumda enum sabitleri de sanki int türden sabitlermiş gibi ele alınacaklardır. enum türleri işlem öncesinde yine büyük türe dönüştürülürler. Yani tamamen birer tamsayı türü gibi değerlendirilirler.

enum türünden nesnelere bildirilebilir. Örneğin:

```
enum COLORS color;
```

```
color = Red;    /* geçerli */  
color = 10;     /* geçerli */
```

enum türünden nesnelere sanki int türündenmiş gibi değerlendirilirler. Böylece bir enum türünden nesneye diğer sayısal türleri doğrudan atayabiliriz. Farklı enum türünden değeri de birbirlerine doğrudan atayabiliriz.

Anahtar Notlar: C++'ta, C# ve Java'da enum türleri farklı birer tür belirtmektedir. Yani bir tamsayı türü enum türüne doğrudan atanamaz. Enum türlerine aynı türden enum sabitleri atanabilir. Fakat C'de enum türleri sanki int türü gibi değerlendirilmektedir.

Enum'lar yukarıda da belirtildiği gibi aynı türden bir dizi sembolik sabiti kolay bir biçimde ifade etmek için kullanılmaktadır.

C'de Dosya İşlemleri

İşletim sistemi tarafından organize edilen ikincil belleklerde tanımlanmış bölgelere dosya (file) denilmektedir. Dosyaların isimleri vardır. Özellikleri vardır. Dosyaların erişim hakları vardır. Pek çok sistemde her dosyaya herkes erişemez.

Modern sistemlerde dosyalar dizinlerin (directories) içerisinde bulunur. Aynı dizin içerisinde aynı isimli birden fazla dosya bulunamaz. Fakat farklı dizinlerde aynı isimli dosyalar bulunabilmektedir: Dosya isimlerinin Windows sistemlerinde büyük harf-küçük harf duyarlılığı yoktur. Fakat Unix/Linux sistemlerinde vardır. Yani örneğin "test.txt" dosyası ile "Test.txt" dosyası Windows'ta aynı isimli isimli dosyalardır. Fakat UNIX/Linux sistemlerinde bunlar farklı isimli dosyalardır.

Bir dosyanın yerini belirten yazısal ifadelerle yol ifadeleri (path) denilmektedir. Yol ifadeleri mutlak (absolute) ve görelili (relative) olmak üzere ikiye ayrılır. Yol ifadesinin ilk karakteri Windows'ta '\', UNIX/Linux'ta '/' ise böyle yol ifadelerine mutlak (absolute) yol ifadeleri denilmektedir. Mutlak yol ifadeleri kök dizinden itibaren yer belirtir. Örneğin:

```
"/a/b/c.txt"    /* mutlak */  
"a/b/c.txt"     /* görelili */  
"a.txt"        /* görelili */
```

Windows'ta dizinler sürücüler (drives) içerisinde yer almaktadır. Her sürücünün ayrı bir kökü (root) vardır. Halbuki UNIX/Linux sistemlerinde toplamda tek bir kök vardır. Bu sistemlerde sürücü kavramı yoktur. Başka bir aygıt bu sistemlere takıldığında onun kökü bir dizinin altında gözükür. Bu işleme "mount işlemi" denilmektedir. Windows'ta sürücü bir harf ve ':' karakterinden oluşur. Örneğin C:, D:, E: gibi...

İşletim sistemlerinde çalışmakta olan programlara "process" ya da "task" denilmektedir. Her prosesin bir çalışma dizini (current working directory) vardır. İşte görelili yol ifadeleri prosesin çalışma dizininden itibaren yer belirtir. Örneğin prosesimizin çalışma dizini "/temp" olsun:

"/a/b/c.txt" biçimindeki yol ifadesi mutlaklıdır. Ve her zaman kökten itibaren yer belirtir. Fakat "a/b/c.txt" biçimindeki yol ifadesi görelidir. Bu durumda bu yol ifadesi "/temp/a/b/c.txt" biçiminde ele alınır. Örneğin "a.txt" yol ifadesi görelidir ve prosesin çalışma dizininde aranır.

Peki processin çalışma dizini neresidir? Prosesin çalışma dizini program çalışırken değiştirilebilir. Fakat program çalışmaya başladığında default olarak çalıştırılabilen dosyanın bulunduğu yerdedir. Visual Studio idesinden çalıştırılan programların çalışma dizini proje dizinidir.

Windows sistemlerinde mutlak yol ifadelerinde sürücü belirtilmemişse prosesin çalışma dizini hangi sürücüye

ilişkinse o mutlak yol ifadesinin o sürücüye ilişkin olduğu anlaşılır. Örneğin prosesin çalışma dizini "E:\Study" olsun. Biz de "\\A\B\C.txt" biçiminde bir yol ifadesi belirtelim. Bu yol ifadesi E'nin kökünden itibaren yer belirtecektir. Tabi mutlak yol ifadeleri sürücü içeriyorsa her zaman o sürücü dikkate alınır.

C'de Dosya İşlemleri

C'de dosya işlemleri prototipleri <stdio.h> içerisinde bulunan standart C fonksiyonlarıyla yapılmaktadır. Bütün dosya fonksiyonlarının isimleri f harfi başlar. Dosya işlemleri aslında işletim sisteminin sunduğu sistem fonksiyonlarıyla yapılmaktadır. Standart C fonksiyonları bu işletim sisteminin sistem fonksiyonlarını çağırarak işlemini yapar.

C'de tipik olarak dosya işlemleri 3 aşamda yürütülür:

- 1) Önce dosya açılır
- 2) Sonra dosyadan okuma yazma yapılır
- 3) En sonunda dosya kapatılır

Dosyanın kapatılması zorunlu değildir. Çünkü program bittiğinde exit işlemi sırasında zaten tüm açık dosyalar kapatılmaktadır. Tabi biz dosyayı çeşitli sebeplerden dolayı program bitmeden erken kapatmak isteyebiliriz.

Dosyanın Açılması

Dosyayı açmak için fopen fonksiyonu kullanılır. Fonksiyonun prototipi şöyledir:

```
FILE *fopen(const char *path, const char *format);
```

Fonksiyonun birinci parametresi açılacak dosyanın yol ifadesini belirtir. İkinci parametre açış modunu belirtir. Açış modu aşağıdakilerden biri biçiminde olabilir:

Açış Modu	Anlamı
"r"	Olan dosyayı açmak için kullanılır. Eğer dosya yoksa fonksiyon başarısız olur. Böyle açılmış dosyalardan yalnızca okuma yapabiliriz.
"r+"	Olan dosyayı açmak için kullanılır. Eğer dosya yoksa fonksiyon başarısız olur. Böyle açılmış dosyalardan hem okuma hem de yazma yapabiliriz (+ okuma yazma anlamına gelir.)
"w"	Dosya yoksa yaratılır ve açılır. Dosya varsa sıfırlanır ve açılır. Yalnızca yazma yapılabilir.
"w+"	Dosya yoksa yaratılır ve açılır. Dosya varsa sıfırlanır ve açılır. Hem okuma hem de yazma yapılabilir.
"a"	Dosya yoksa yaratılır ve açılır. Dosya varsa olan dosya açılır. Dosyadan okuma yapamayız. Fakat her yazılan sona ekleme anlamına gelir. (Yani dosyanın ortasına yazma diye bir durum yoktur. Her yazma işlemi ekleme anlamına gelir.)
"a+"	Dosya yoksa yaratılır ve açılır. Dosya varsa olan dosya açılır. Dosyadan okuma yapabiliriz. Fakat her yazılan sona ekleme anlamına gelir. (Yani dosyanın ortasına yazma diye bir durum yoktur. Her yazma işlemi ekleme anlamına gelir.)

fopen fonksiyonu başarı durumunda FILE isimli bir yapı türünden adrese geri döner. fopen açılan dosyanın bilgilerini stdio içerisinde bildirilmiş ve FILE ismiyle typedef edilmiş bir yapı nesnesinin içerisine yerleştirmektedir ve o o yapı nesnesinin adresine geri dönmektedir. Programcı FILE yapısının elemanlarını bilmek zorunda değildir. Ancak FILE isminin bir yapı belirttiğini bilmelidir. Fonksiyon başarısız olabilir. Bu durumda fopen NULL adrese geri döner. Mutlaka başarı kontrolü yapılmalıdır. FILE türünden adrese biz "dosya bilgi gösterici" diyeceğiz. İngilizce genellikle bu adrese "file stream" denilmektedir.

Örneğin:

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

int main(void)
{
    FILE *f;

    if ((f = fopen("test.dat", "w")) == NULL) {
        printf("cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

Dosyanın Kapatılması

Dosyayı kapatmak için `fclose` fonksiyonu kullanılır. Fonksiyonun prototipi şöyledir:

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

Fonksiyon parametre olarak `fopen` fonksiyonundan alınan `FILE` yapı adresini (yani dosya bilgi göstericisini) almaktadır. Fonksiyon başarı durumunda sıfır değerine başarısızlık durumunda `-1` değerine geri döner. Dosyanın kapatılmasının başarısının kontrol edilmesine gerek yoktur. Zaten dosya başarılı bir biçimde açılmışsa kesinlikle başarılı kapatılır.

Örneğin:

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

int main(void)
{
    FILE *f;

    if ((f = fopen("test.dat", "w")) == NULL) {
        printf("cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    fclose(f);

    return 0;
}
```

Dosya Göstericisi (File Pointer) Kavramı

Dosya byte'lardan oluşan bir topluluktur. Dosyanın içerisinde ne olursa olsun dosya işletim sistemi için bir byte yığıdır. Dosyanın her bir byte'ına ilk byte sıfır olmak üzere bir numara verilmiştir. Bu numaraya ilgili byte'ın offset'i denir. Örneğin aşağıdaki çizimde her bir X bir byte'ı gösteriyor olsun. Tüm bu byte'ların bir offset'i vardır.

```
0 1 2 3 4
X X X X X
```

Dosya göstericisi bellekte bir adres belirten daha önce gördüğümüz gibi bir gösterici değildir. Buradaki isim

tamamen kavramsal olarak verilmiştir. Dosya göstericisi bir offset belirtir. Bir imleç görevindedir. Tüm okuma ve yazma işlemleri dosya göstericisinin gösterdiği yerden itibaren yapılır.

Örneğin dosya göstericisi 2'inci offset'i gösteriyor olsun:

```
0 1 2 3 4
X X X X X
    ↑
   D6
```

Şimdi biz dosyadan 2 byte okuyacak olalım. İşte okuma işlemi dosya göstericisinin gösterdiği yerden itibaren yapılır. Yani 2'inci ve 3'üncü offset'teki byte'lar okunur. Benzer biçimde yazma işlemi de her zaman dosya göstericisinin gösterdiği yerden itibaren yapılır. Okuma ve yazma işlemleri dosya göstericisini okunan ya da yazılan byte miktarı kadar ilerletir. Örneğin dosyaya YY ile temsil edilen iki byte'ı yazmış olalım:

```
0 1 2 3 4
X X Y Y X
    ↑ ↑
   D6 D6
```

Dosya ilk kez açıldığında dosya göstericisi sıfırcı offset'i gösteriyor durumdadır.

EOF Durumu

Dosyanın sonunda özel hiçbir karakter ya da byte yoktur. Dosya göstericisinin dosyanın son byte'ından sonraki byte'ı göstermesi durumuna EOF durumu denilmektedir. Örneğin:

```
0 1 2 3 4 5
X X X X X
          ↑
         EOF
```

Dosya göstericisi EOF durumundaysa artık o dosyada olan bir byte'ı göstermiyordur. Bu nedenle EOF durumundan okuma yapılamaz. Dosya göstericisi EOF durumundaysa okuma fonksiyonları başarısız olur. Ancak EOF durumunda yazma yapılabilir. Bu durum dosyaya ekleme yapma anlamına gelir.

Peki bir dosyayı açıp byte byte okuduğumuzu düşünelim. Ne olur? Biz sırasıyla byte'ları okuruz. Dosya göstericisi en sonunda EOF durumuna gelir. Artık okuma yapamayız.

fgetc Fonksiyonu

fgetc dosyadan bir byte okuyan standart C fonksiyonudur. Prototipi şöyledir:

```
int fgetc(FILE *f);
```

Fonksiyon parametre olarak fopen fonksiyonundan elde edilen dosya bilgi göstericisini alır. Dosya göstericisinin gösterdiği yerdeki byte'ı okur. Okuduğu byte ile geri döner. Geri dönüş değeri int türündedir. Okuma başarılıysa int bilginin yüksek anlamlı byte'larında sıfır bulunur. En düşük anlamlı byte'ında okunan değer bulunur. fgetc IO

hatası nedeniyle ya da EOF nedeniyle okumayı yapamazsa EOF denilen bir değere geri döner: EOF değeri <stdio.h> içerisinde -1 olarak define edilmiştir:

```
#define EOF    -1
```

Pekiye fgetc okuduğu byte'a geri döndüğüne göre neden geri dönüş değeri char değil de int türündendir. Eğer fgetc'in geri dönüş değeri char olsaydı bu durumda dosyadan FF numaralı byte'ı okuduğunda FF = -1 olduğundan fgetc'in başarısız mı olduğu yoksa gerçekten FF numaralı byte'ı mı okuduğu anlaşılamazdı. Halbuki fonksiyonun geri dönüş değeri int olduğu için bu sorun oluşmamaktadır. Yani:

```
00 00 00 FF ---> FF numaralı byte okunmuş  
FF FF FF FF --> başarısız olunmuş
```

Örneğin bir dosyayı sonuna kadar okuyup yazdıran örnek program şöyle yazılabilir:

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main(void)  
{  
    FILE *f;  
    int ch;  
  
    if ((f = fopen("Sample.c", "r")) == NULL) {  
        printf("cannot open file!..\n");  
        exit(EXIT_FAILURE);  
    }  
  
    while ((ch = fgetc(f)) != EOF)  
        putchar(ch);  
  
    fclose(f);  
  
    return 0;  
}
```

feof ve ferror Fonksiyonları

Bir dosya fonksiyonu (örneğin fgetc gibi) başarısız olduğunda bunun nedeni dosya sonuna gelinmiş olması olabilir ya da daha ciddi (genellikle biz artık şey yapamayız) IO hatası olabilir. fgetc fonksiyonu her iki durumda da EOF değeriyle geri dönmektedir. Dosya sonuna gelmiş olmak gayet normal bir durumdur. Halbuki bir disk hatası ciddi bir soruna işaret eder. İşte bizim fgetc gibi bir fonksiyonun neden EOF'a geri döndüğünü tespit edebilmemeiz gerekir. Bunun için feof ve ferror fonksiyonlarından faydalanılır:

```
int feof(FILE *f);  
int ferror(FILE *f);
```

feof fonksiyonu dosya göstericisinin EOF durumunda olup olmadığını tespit eder. Eğer dosya göstericisi EOF durumundaysa fonksiyon sıfır dışı herhangi bir değere, EOF durumunda değilse 0 değerine geri döner. Ancak bu fonksiyon son okuma işlemi başarısız olduğunda kullanılmalıdır. Şöyle ki: Biz dosyadaki son byte'ı fgetc ile okuduğumuzda dosya göstericisi EOF'a çekildiği halde feof sıfıra geri döner. Bundan sonra bir daha fgetc uygularsak fgetc başarısız olur. Arık feof sıfır dışı değere geri döner.

ferror ise son okuma ya da yazma işleminde IO hatası oluşup oluşmadığını belirlemek için kullanılmaktadır. Eğer son dosya işlemi IO hatası nedeniyle başarısız olmuşsa ferror sıfır dışı bir değere, değilse sıfır değerine geri döner.

Özellikle son işlemin başarısı iyi bir teknik bakımından kontrol edilmelidir. Örneğin:

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *f;
    int ch;

    if ((f = fopen("Sample.c", "r")) == NULL) {
        printf("cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    while ((ch = fgetc(f)) != EOF)
        putchar(ch);

    if (ferror(f)) {
        printf("cannot read file!..\n");
        exit(EXIT_FAILURE);
    }

    fclose(f);

    return 0;
}

```

Örneğin aşağıdaki gibi dosya sonuna kadar okuma işlemi hatalıdır:

```

while (!feof(f)) {
    ch = fgetc(f);
    putchar(ch);
}

```

Çünkü burada son byte okunduktan sonra henüz feof sıfır dışı değer vermez. Dolayısıyla program fazladan bir byte okunmuş gibi davranır. Ayrıca burada IO hatası için bir tespit yapılmamıştır. Doğru teknik şöyledir:

```

while ((ch = fgetc(f)) != EOF)
    putchar(ch);

if (ferror(f)) {
    printf("cannot read file!..\n");
    exit(EXIT_FAILURE);
}

```

fputc Fonksiyonu

fputc dosyaya dosya göstericisinin gösterdiği yere bir byte yazan en temel yazma fonksiyonudur. Prototipi şöyledir:

```
int fputc(int ch, FILE *f);
```

Fonksiyon birinci parametresiyle belirtilen int değer in en düşün anlamlı byte'ını yazar. Başarı durumunda yazdığı değer, başarısızlık durumunda EOF değerine geri döner.

Örneğin:

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{

```



```

FILE *f;
char s[] = "this is a test, yes this is a test!";
int i;

if ((f = fopen("test.txt", "w")) == NULL) {
    printf("cannot open file!..\n");
    exit(EXIT_FAILURE);
}

for (i = 0; s[i] != '\0'; ++i)
    if (fputc(s[i], f) == EOF) {
        printf("cannot write file!..\n");
        exit(EXIT_FAILURE);
    }

fclose(f);

return 0;
}

```

Dosya kopyalayan örnek bir program şöyle yazılabilir:

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fs, *fd;
    int ch;

    if ((fs = fopen("sample.c", "r")) == NULL) {
        printf("cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    if ((fd = fopen("xxx.c", "w")) == NULL) {
        printf("cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    while ((ch = fgetc(fs)) != EOF)
        if (fputc(ch, fd) == EOF) {
            printf("cannot write file!..\n");
            exit(EXIT_FAILURE);
        }
    if (ferror(fs)) {
        printf("cannot read file!..\n");
        exit(EXIT_FAILURE);
    }

    printf("success...\n");

    fclose(fs);
    fclose(fd);

    return 0;
}

```

fread ve fwrite Fonksiyonları

fread ve fwrite fonksiyonları getc ve fputc fonksiyonlarının n byte okuyan ve n byte yazan genel biçimleridir. fread dosya göstericisinin gösterdiği yerden itibaren n byte'ı okuyarak bellekte verilen adresten itibaren bir diziye yerleştirir. fwrite ise tam tersini yapmaktadır. Yani bellekte verilen bir adresten başlayarak n byte'ı dosya

göstericisinin gösterdiği yerden itibaren dosyaya yazar. Fonksiyonların prototipleri şöyledir:

```
size_t fread(void *ptr, size_t size, size_t count, FILE *f);
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *f);
```

fread fonksiyonun birinci parametresi bellekteki transfer adresidir. Fonksiyon ikinci ve üçüncü parametrelerin çarpımı kadar byte okur (yani size * count kadar). Son parametre okuma işleminin yapılacağı dosyayı belirtmektedir. Genellikle ikinci parametre okunacak dizinin bir elemanının byte uzunluğu olarak, ikinci parametre ise okunacak eleman uzunluğu olarak girilir. Örneğin dosyada int bilgiler olsun ve biz 10 int değeri okumak isteyelim:

```
int a[10];
```

```
fread(a, sizeof(int), 10, f);
```

fread fonksiyonu başarı durumunda okuyabildiği parça sayısına geri döner. Başarısızlık durumunda sıfır değerine geri dönmektedir. fread fonksiyonu ile dosyada olandan daha fazla byte okunmak istenebilir. Bu durumda fread okuyabildiği kadar byte'ı okur ve okuyabildiği parça sayısına geri döner. Örneğin dosya göstericisinin gösterdiği yerden itibaren dosyada 20 byte kalmış olsun. Biz de aşağıdaki gibi bir okuma yapmış olalım:

```
fread(a, sizeof(int), 10, f);
```

Biz buarad 40 byte okuma talep etmiş olabiliriz. Ancak fread kalan 20 byte'ın hepsini okur ve 5 değerine geri öner. Yani fread toplam okunan byte sayısının ikinci parametrede belirtilen değere bölümüne geri dönmektedir.

fwrite fonksiyonu da tamamen aynı biçimde çalışır. Tek fark fwrite dosyadan belleğe okuma değil bellekten dosyaya yazma yapmaktadır. fwrite da başarı durumunda yazılan parça sayısına başarısızlık durumunda sıfır değerine geri döner.

Örneğin 10 elemanlı bir int dizi dosyaya tek hamlede fwrite fonksiyonuyla şöyle yazdırılabilir:

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

int main(void)
{
    FILE *f;
    int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    if ((f = fopen("Sample.dat", "wb")) == NULL) {
        printf("cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    if (fwrite(a, sizeof(int), 10, f) != 10) {
        printf("cannot write file!..\n");
        exit(EXIT_FAILURE);
    }

    fclose(f);

    return 0;
}
```

Şimdi de yazdığımız sayıları tek hamlede fread ile okuyalım:

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

```

#include <stdlib.h>

int main(void)
{
    FILE *f;
    int a[10];
    int i;

    if ((f = fopen("Sample.dat", "rb")) == NULL) {
        printf("cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    if (fread(a, sizeof(int), 10, f) != 10) {
        printf("cannot read file!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i)
        printf("%d ", a[i]);
    printf("\n");

    fclose(f);

    return 0;
}

```

Şimdi de bir dosyadan belli bir miktar okuyup diğerine yazarak dosya kopyalaması yapmaya çalışalım:

```

#include <stdio.h>
#include <stdlib.h>

#define BUF_SIZE    512

int main(void)
{
    FILE *fs, *fd;
    char buf[BUF_SIZE];
    size_t n;

    if ((fs = fopen("sample.c", "rb")) == NULL) {
        printf("cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    if ((fd = fopen("xxx.c", "wb")) == NULL) {
        printf("cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    while ((n = fread(buf, 1, BUF_SIZE, fs)) > 0)
        if (fwrite(buf, 1, n, fd) != n) {
            printf("cannot write file!..\n");
            exit(EXIT_FAILURE);
        }

    if (ferror(fs)) {
        printf("cannot read file!..\n");
        exit(EXIT_FAILURE);
    }

    printf("success...\n");

    fclose(fs);
    fclose(fd);
}

```

```
    return 0;
}
```

Yapıların Dosyalara Yazılması ve Okunması

Yapı elemanları bellekte ardışıl bir biçimde tutulduğuna göre, biz bir yapıyı tek hamlede fwrite fonksiyonuyla yazıp, fread fonksiyonuyla okuyabiliriz. Örneğin struct PERSON türünden per isimli bir yapı nesnesi olsun:

```
fwrite(&per, sizeof(struct PERSON), 1, f);
```

Klavyeden girilen değerlerden oluşan yapı nesnelere fwrite fonksiyonuyla dosyaya yazdırılmaktadır:

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

typedef struct tagPERSON {
    char name[32];
    int no;
} PERSON;

int main(void)
{
    FILE *f;
    PERSON per;

    if ((f = fopen("person.dat", "wb")) == NULL) {
        printf("cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    for (;;) {
        printf("Adi soyadi:");
        gets(per.name);
        if (!strcmp(per.name, "exit"))
            break;
        printf("No:");
        scanf("%d", &per.no);
        getchar(); /* tamponu boşaltmak için, şimdilik takılmayın */
        if (fwrite(&per, sizeof(per), 1, f) != 1) {
            printf("cannot write file!..\n");
            exit(EXIT_FAILURE);
        }
    }

    fclose(f);

    return 0;
}
```

Aynı dosyadan kayıtları okuyan program da şöyle yazılabilir:

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

typedef struct tagPERSON {
    char name[32];
    int no;
} PERSON;

int main(void)
```

```

{
    FILE *f;
    PERSON per;

    if ((f = fopen("person.dat", "rb")) == NULL) {
        printf("cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    while (fread(&per, sizeof(per), 1, f) > 0)
        printf("%s, %d\n", per.name, per.no);

    if (ferror(f)) {
        printf("cannot read file!..\n");
        exit(EXIT_FAILURE);
    }

    fclose(f);

    return 0;
}

```

fprintf ve fscanf fonksiyonları

fprintf fonksiyonu printf gibi çalışan fakat dosyaya yazdırma amaçlı kullanılan standart C fonksiyonudur. İlk parametresi dışında diğer parametreleri printf fonksiyonu ile aynıdır. fprintf fonksiyonunun prototipi aşağıdaki gibidir.

```
int fprintf(FILE *f, const char *format, ...);
```

Fonksiyonun birinci parametresi yazılacak dosyayı belirtmektedir. Fonksiyonun ikinci parametresi formatlanması istenen yazıdır. printf fonksiyonu ile aynıdır. fprintf fonksiyonu aşağıdaki gibi kullanılabilir:

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *f;
    char file_name[] = "test.txt";
    int i;

    if ((f = fopen(file_name, "w")) == NULL) {
        printf("can not open file!..%s\n", file_name);
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i)
        fprintf(f, "sayı=%d\n", i + 1);

    fclose(f);

    return 0;
}

```

Burada fprintf fonksiyonu ile formatlı bir şekilde dosyaya yazma yapılmıştır. Yukarıda yazma yapılan dosyanın içeriği okunarak ekrana aşağıdaki gibi bir programla yazdırılabilir:

```

#include <stdio.h>
#include <stdlib.h>

```

```

int main(void)
{
    FILE *f;
    char file_name[] = "test.txt";
    int i, ch;

    if ((f = fopen(file_name, "r")) == NULL) {
        printf("can not open file...%s\n", file_name);
        exit(EXIT_FAILURE);
    }

    while ((ch = fgetc(f)) != EOF)
        putchar(ch);

    if (ferror(f)) {
        printf("can not read file...!");
        exit(EXIT_FAILURE);
    }

    fclose(f);

    return 0;
}

```

fprintf fonksiyonu ile dosyaya formatlı bir bilgi yazılmaktadır. Aşağıdaki programlar fprintf ve fwrite fonksiyonlarının farkını göstermektedir:

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *f;
    char file_name[] = "test.txt";
    int val, ch;

    if ((f = fopen(file_name, "w")) == NULL) {
        printf("can not open file...%s\n", file_name);
        exit(EXIT_FAILURE);
    }

    val = 1234;

    fprintf(f, "%d\n", val);

    fclose(f);

    if ((f = fopen(file_name, "r")) == NULL) {
        printf("can not open file...%s\n", file_name);
        exit(EXIT_FAILURE);
    }

    while ((ch = fgetc(f)) != EOF)
        putchar(ch);

    fclose(f);

    return 0;
}

```

```

#include <stdio.h>
#include <stdlib.h>

```

```

int main(void)
{
    FILE *f;
    char file_name[] = "test.txt";
    int val, ch;

    if ((f = fopen(file_name, "w")) == NULL) {
        printf("can not open file...%s\n", file_name);
        exit(EXIT_FAILURE);
    }

    val = 1234;

    fwrite(&val, sizeof(int), 1, f);

    fclose(f);

    if ((f = fopen(file_name, "r")) == NULL) {
        printf("can not open file...%s\n", file_name);
        exit(EXIT_FAILURE);
    }

    while ((ch = fgetc(f)) != EOF)
        putchar(ch);

    fclose(f);

    return 0;
}

```

Anahtar Notlar: Bilindiği fwrite ve fread fonksiyonları dosyaya ikili olarak bilgi yazıp okumaktadırlar:

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *f;
    char file_name[] = "test.txt";
    int val, i;

    if ((f = fopen(file_name, "wb")) == NULL) {
        printf("can not open file...%s\n", file_name);
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 4; ++i)
        fwrite(&i, sizeof(int), 1, f);

    fclose(f);

    if ((f = fopen(file_name, "rb")) == NULL) {
        printf("can not open file...%s\n", file_name);
        exit(EXIT_FAILURE);
    }

    while (fread(&val, sizeof(int), 1, f) > 0)
        printf("%d\n", val);

    fclose(f);
}

```

```
    return 0;
}
```

fscanf fonksiyonu scanf fonksiyonu gibi ancak dosyadan okuma yapma amaçlı kullanılmaktadır. fscanf fonksiyonu dosyadan formatlı olarak okuma yapar. Fonksiyonun prototipi aşağıdaki gibidir:

```
int fscanf(FILE *f, const char *format, ...);
```

Fonksiyonun birinci parametresi okuma yapılacak dosyayı belirtmektedir. Diğer parametreleri scanf fonksiyonu ile aynıdır.

fscanf fonksiyonu ile dosyadan okuma işlemi aşağıdaki gibi yapılabilir.

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

int main(void)
{
    FILE *f;
    char file_name[] = "test.txt";
    int a, b;

    if ((f = fopen(file_name, "r")) == NULL) {
        printf("can not open file...%s\n", file_name);
        exit(EXIT_FAILURE);
    }

    fscanf(f, "%d %d", &a, &b);

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

    fclose(f);

    return 0;
}
```

fscanf fonksiyonun geri dönüş değeri dosyadan okunarak bellekteki alanlara yazılan değer sayısıdır. Hiç bir yazma yapılmadıysa fonksiyon 0 (sıfır) değerine geri döner. Eğer ilk alana atama yapılmadan dosyanın sonuna gelinmişse, ya da bir hata oluşmuşsa fonksiyon EOF değerine geri döner. Aşağıdaki program her bir satırında 4 adet sayı olan bir dosyadan okuma yapmaktadır ve her bir dördümlü grubun toplamını bulmaktadır:

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

int main(void)
{
    FILE *f;
    char file_name[] = "test.txt";
    int a, b, c, d;

    if ((f = fopen(file_name, "r")) == NULL) {
        printf("can not open file...%s\n", file_name);
        exit(EXIT_FAILURE);
    }

    while (fscanf(f, "%d %d %d %d", &a, &b, &c, &d) != EOF)
        printf("%d\n", a + b + c + d);

    if (ferror(f)) {
```



```

        printf("can not read file....!\n");
        exit(EXIT_FAILURE);
    }

    fclose(f);

    return 0;
}

```

fscanf fonksiyonu ile her okunan bilgi dönüştürülerek bir bellek alanına aktarılmak zorunda değildir. Böyle okumalarda % ile format karakteri arasına * konularak okunan değer devre dışı bırakılması (ama okunması) sağlanabilir. Örneğin:

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *f;
    char file_name[] = "test.txt";
    int a, b;

    if ((f = fopen(file_name, "r")) == NULL) {
        printf("can not open file...%s\n", file_name);
        exit(EXIT_FAILURE);
    }

    fscanf(f, "%d*s%d", &a, &b);

    if (ferror(f)) {
        printf("can not read file....!\n");
        exit(EXIT_FAILURE);
    }

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

    fclose(f);

    return 0;
}

```

Burada dosya içerisinde örneğin

```
10 ----- 20
```

gibi bir bilgiden yalnızca 10 ve 20 sayıları okunmaktadır.

Sınıf Çalışması: Formatı aşağıdaki gibi olan bir dosya içerisinde dosya formatının geçerliliğini kontrol etmeden yaş ortalamasını bulan program

Dosya formatı:

```

isim mert yas 18
isim oguz yas 40
isim serkan yas 35
isim mustafa yas 42

```

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

```

#include <stdlib.h>

int main(void)
{
    FILE *f;
    char file_name[] = "test.txt";
    int age, sum = 0, count = 0;

    if ((f = fopen(file_name, "r")) == NULL) {
        printf("can not open file...%s\n", file_name);
        exit(EXIT_FAILURE);
    }

    while (fscanf(f, "%*s%*s%*s%d", &age) != EOF) {
        sum += age;
        count++;
    }

    if (ferror(f)) {
        printf("can not read file....!\n");
        exit(EXIT_FAILURE);
    }

    printf("Yas Ortalamasi:%lf\n", (double) sum / count);

    fclose(f);

    return 0;
}

```

Sınıf Çalışması: Formatı aşağıdaki gibi olan bir dosya içerisinde dosya formatının geçerliliğini kontrol etmeden değerlerin ortalamasını bulan programı yazınız.

```

Firma1 30 40 56
Firma2 10 20 35
Firma3 10 20 33

```

```

#include <stdio.h>
#include <stdlib.h>

#define MAX_LEN    32

typedef struct tagCOMPANYINFO {
    char name[MAX_LEN];
    int val1, val2, val3;
} COMPANYINFO;

int main(void)
{
    FILE *f;
    char file_name[] = "test.txt";
    int i;
    COMPANYINFO companies[3];

    if ((f = fopen(file_name, "r")) == NULL) {
        printf("can not open file...%s\n", file_name);
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 3; ++i) {

```

```

        fscanf(f, "%s %d %d %d", companies[i].name, &companies[i].val1, &companies[i].val2,
&companies[i].val3);
    }

    if (ferror(f)) {
        printf("can not read file....!\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 3; ++i) {
        double avg = (companies[i].val1 + companies[i].val2 + companies[i].val3) / 3.;

        printf("%s ---> %lf\n", companies[i].name, avg);
    }

    fclose(f);

    return 0;
}

```

Aşağıdaki program rasgele ad, soyad ve not belirleyerek grades.txt isimli dosyaya formatlı bir şekilde yazma yapmaktadır.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void exit_sys(const char *msg);

int main(void)
{
    FILE *f;
    char file_name[] = "grades.txt";
    int i;
    int number_of_students;
    char *names[8] = {"oguz", "mert", "durukan", "boran", "mustafa", "serkan", "onur", "latif"};
    char *snames[8] = {"aksoy", "kaya", "vural", "altan", "karpuz", "sert", "mulaim", "naif"};

    if ((f = fopen(file_name, "w")) == NULL)
        exit_sys("can not open file\n");

    srand((unsigned)time(NULL));

    number_of_students = rand() % 20 + 30;

    for (i = 0; i < number_of_students; ++i)
        fprintf(f, "%s %s %d\n", names[rand() % 8], snames[rand() % 8], rand() % 101);

    fclose(f);

    return 0;
}

void exit_sys(const char *msg)
{
    printf(msg);
    exit(EXIT_FAILURE);
}

```

Sınıf Çalışması: Yukarıda oluşturulan grades.txt dosyasını kullanarak belirli bir geçme değeri belirleyip, geçenleri pass.txt kalanları fail.txt isimli dosyaya formatlı olarak yazdıran programı yazınız.

```

#include <stdio.h>
#include <stdlib.h>

```

```

#include <time.h>

#define PASS_GRADE 50

void exit_sys(const char *msg);

int main(void)
{
    FILE *fgrades, *fpass, *ffail;
    char grades_file_name[] = "grades.txt";
    char pass_file_name[] = "pass.txt";
    char fail_file_name[] = "fail.txt";
    char name[20], sname[20];
    int grade;

    if ((fgrades = fopen(grades_file_name, "r")) == NULL)
        exit_sys("can not open file\n");

    if ((fpass = fopen(pass_file_name, "w")) == NULL)
        exit_sys("can not open file\n");

    if ((ffail = fopen(fail_file_name, "w")) == NULL)
        exit_sys("can not open file\n");

    while (fscanf(fgrades, "%s%s%d", name, sname, &grade) != EOF)
        if (grade < PASS_GRADE)
            fprintf(ffail, "%s %s %d\n", name, sname, grade);
        else
            fprintf(fpass, "%s %s %d\n", name, sname, grade);

    if (ferror(fgrades))
        exit_sys("can not read file\n");

    fclose(fgrades);
    fclose(fpass);
    fclose(ffail);

    return 0;
}

void exit_sys(const char *msg)
{
    printf(msg);
    exit(EXIT_FAILURE);
}

```

Metin ve İkili Dosyalar

C de bir dosya metin (text) ya da ikili (binary) modda açılabilir. Açılış modunda hiç belirtme yapılmassa default olarak dosya text modda açılır. Programcı isterse açılış mod yazısının sonuna "t" ekleyerek de açılışı text modda yaptırabilir. Örneğin:

```

if (f = fopen("test.txt", "w") == NULL)
    /*...*/

```

ya da örneğin

```

if (f = fopen("test.txt", "wt") == NULL)
    /*...*/

```

Bir dosyayı binary modda açmak için mod yazısının "b" getirilmelidir. Örneğin:

```
if (f = fopen("test.txt", "wb") == NULL)
    /*...*/
```

Text ve binary modda açılan dosyalar için Windows ve Unix/Linux sistemlerinde farklılıklar bulunmaktadır. Bir dosya text modda açılmışsa ve çalışılan sistem windows ise yazma yapan herhangi bir fonksiyon Line feed (LF) ('\n') karakterini yazdığı anda aslında dosyaya Carriage Return (CR)('\r') ve LF karakterlerinin ikisi birden yazılır. Benzer şekilde dosyadan okuma yapan fonksiyonlar çalışılan sistem windows ise ve dosya text modda açılmışsa CRLF karakterlerini yanyana gördüğünde yalnızca LF olarak okuma yaparlar. Aşağıdaki program bir text modda açılmış bir dosyaya beş adet LF karakteri yazmaktadır:

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

int main(void)
{
    FILE *f;
    int i;

    if ((f = fopen("test.txt", "w")) == NULL) {
        printf("can not open file");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 5; ++i)
        fputc('\n', f);

    fclose(f);

    return 0;
}
```

Bu program Windows sistemlerinde çalıştırıldığında yazma yapılan dosya 10 byte olacaktır. Çünkü her LF yazması aslında CR ve LF karakterlerinin ikisinin birden yazılması demektir. Unix/Linux sistemlerinde text dosya ve binary dosya arasında hiç bir fark yoktur. Yani yukarıdaki program Unix/Linux sistemlerinde çalıştırıldığında 5 byte olacaktır. Çünkü beş adet LF karakteri dosyaya yazılmıştır.

Aşağıdaki program az önce üretilmiş olan dosya içerisindeki karakterleri okumaktadır:

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

int main(void)
{
    FILE *f;
    int i;
    int ch;

    if ((f = fopen("test.txt", "rb")) == NULL) {
        printf("can not open file");
        exit(EXIT_FAILURE);
    }

    while ((ch = fgetc(f)) != EOF)
        printf("%d\n", ch);

    fclose(f);

    return 0;
}
```

```
}
```

Bu program çalıştırıldığında binary modda olduğundan CRLF çiftlerini okuyacaktır.

Metin dosyaları için Windows sistemlerinde diğer sistemlere bir fark daha vardır. Windows sistemlerinde text modunda okuma yapılırken 26 numaralı Ctrl+Z karakteri sonlandırıcı karakter olarak kabul edilir. Örneğin aşağıdaki program bir exe dosyanın tüm karakterini text modda okumaktadır:

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

int main(void)
{
    FILE *f;
    int i;
    int ch;
    long count = 0;

    if ((f = fopen("Sample.exe", "r")) == NULL) {
        printf("can not open file");
        exit(EXIT_FAILURE);
    }

    while ((ch = fgetc(f)) != EOF)
        count++;

    printf("Count=%ld\n", count);

    fclose(f);

    return 0;
}
```

Bu program windows sistemlerinde çalıştırıldığında dosya içerisinde ctrlz karakteri görüldüğünde okuma sonlanacaktır. Ancak aynı program Unix/Linux sistemlerinde çalıştırıldığında herhangi sonlanma olmayacaktır.

Dosya Göstericisinin Konumlandırılması ve fseek Fonksiyonu

Anımsanacağı gibi dosya açıldığında dosya göstericisi 0'ıncı offset'tedir. Okuma yazma işlemleriyle dosya göstericisi otomatik ilerletilmektedir. Ancak biz istersek dosya göstericisini fseek fonksiyonuyla istediğimiz offset'e konumlandırabiliriz. fseek fonksiyonunun prototipi şöyledir:

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);
```

Fonksiyonun birinci parametres, konumlandırılacak dosyaya ilişkin dosya bilgi göstericisini alır. İkinci parametre dosya offset'ini belirtmektedir. Üçüncü parametre konumlandırma orijini belirtir. Üçüncü parametre üç değerden birisi olabilir: 0, 1 veya 2. Bu değerler aynı zamanda aşağıdaki gibi define da edilmiştir:

```
#define SEEK_SET      0
#define SEEK_CUR      1
#define SEEK_END      2
```

Eğer üçüncü parametre sıfır girilirse ikinci parametre dosyanın başından itibaren offset belirtir. Yani ikinci parametre ≥ 0 olmak zorundadır. Örneğin:

```
fseek(f, 100, SEEK_SET);
```

Burada dosya göstericisi 100'üncü offset'e konumlandırılmıştır. Eğer üçüncü parametre 1 girilirse konumlandırma dosya göstericisinin gösterdiği yerden itibaren yapılır. Tabi bu durumda ikinci parametre sıfır, pozitif ya da negatif olabilir. Örneğin:

```
fseek(f, 1, SEEK_CUR);
```

Bu işlemle dosya göstericisi neredeyse onun 1 ilerisine konumlandırılır. Örneğin:

```
fseek(f, -10, SEEK_CUR);
```

Burada dosya göstericisi 10 geriye konumlandırılmaktadır. Eğer son parametre 2 girilirse bu durumda konumlandırma EOF'dan itibaren yapılır. Tabi bu durumda ikinci parametre ≤ 0 girilmek zorundadır. Örneğin:

```
fseek(f, 0, SEEK_END);
```

Bu durumda dosya göstericisi EOF'a konumlandırılır. Yani artık yazma yapmak istediğimizde dosyaya ekleme yapmış oluruz. Fonksiyon konumlandırma başarılıysa 0 değerine, başarısızsa EOF (-1) değerine geri döner. Örneğin:

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

int main(void)
{
    FILE *f;

    if ((f = fopen("Test.txt", "r+")) == NULL) {
        printf("can not open file");
        exit(EXIT_FAILURE);
    }

    fseek(f, 0, SEEK_END);

    fprintf(f, "Yes, this is a test");

    fclose(f);

    return 0;
}
```

Burada dosya göstericisi dosyanın sonuna çekilip yazma yapılmıştır. Örneğin:

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

int main(void)
{
    FILE *f;
    char s[10];

    if ((f = fopen("Test.txt", "r+")) == NULL) {
        printf("can not open file");
        exit(EXIT_FAILURE);
    }
}
```

```

fseek(f, 10, SEEK_SET);

fgets(s, 5, f);
puts(s);

fclose(f);

return 0;
}

```

Burada dosya göstericisi dosyanın 10'uncu offset'ine çekilip okuma yapılmıştır.

C'de okumadan yazmaya yazmadan okumaya geçişte mutlaka fseek ya da fflush işlemi yapılmalıdır.

stdin, stdout ve stderr Dosyaları

Bir C programında stdin, stdout ve stderr FILE * türünden dosya bilgi göstericisi belirtmektedir. Bu isimler <stdio.h> dosyasında bildirilmiştir. (Dolayısıyla bu isimler anahtar sözcük değildir. Bunları kullanabilmemiz için <stdio.h> dosyasını include etmeliyiz.

stdin dosyasına "standard input" dosyası, stdout dosyasına "standart output" dosyası, stderr dosyasına ise "standard error dosyası" denilmektedir.

C standartlarında "klavye" ve "ekran" lafı edilmemiştir. Standartlara göre stdout ekrana ya da istenirse başka aygıtlara yönlendirilebilir. Örneğin printf stdout nereye yönlendirilmişse oraya yazar. stdout eğer yazıcıya yönlendirilmişse printf de yazıcıya yazdıracaktır. Bugün kullandığımız masaüstü ve dizüstü bilgisayarlarda stdout default olarak ekrana yönlendirilmiş durumdadır. Fakat biz onu değiştirebiliriz. Örneğin standartlara göre getch, gets fonksiyonlar stdin dosyasından okuma yapar. stdin de bilgisayarlarımızda default olarak klavyeye yönlendirilmiş durumdadır. Fakat biz onu başka aygıtlara yönlendirebiliriz. Örneğin:

```
printf(.....);
```

çağrısının aslında,

```
fprintf(stdout, ....);
```

çağrısından hiçbir farkı yoktur. Örneğin:

```

#include <stdio.h>

int main(void)
{
    fprintf(stdout, "This is a test\n");

    return 0;
}

```

Benzer biçimde örneğin getch aslında fgetc(stdin) ile aynı işlemi yapar.

stderr dosyası hata mesajlarının yazdırılacağı dosyayı temsil eder. Default durumda stderr dosyası da ekrana yönlendirilmiştir. Ancak biz onu dosyaya ya da başka aygıtı yönlendirebiliriz. Programın hata mesajlarının stdout yerine fprintf fonksiyonuyla stderr dosyasına yazdırılması iyi bir tekniktir. Örneğin:

```

if ((p = malloc(size)) == NULL) {
    fprintf(stderr, "cannot allocate memory!..\n");
}

```



```
    exit(EXIT_FAILURE);  
}
```

Windows'ta ve UNIX/Linux sistemlerinde stdout dosyasını yönlendirmek için komut satırında > işareti kullanılır. Örneğin:

```
sample > x.txt
```

Benzer biçimde stderr dosyası 2> sembolüyle yönlendirilir. Örneğin:

```
sample 2> y.txt
```

Burada stdout dosyasına yazılanlar yine ekrana çıkar fakat stderr dosyasına yazılanlar y.txt dosyasına yazılır. İki yönlendirme beraber de kullanılabilir:

```
sample > x.txt 2> y.txt
```

stdin dosyasını yönlendirmek için < sembolü kullanılmaktadır. Örneğin:

```
sample < x.txt
```

Bu arada artık stdin dosyasından okuma yapıldığında sanki x.txt dosyasındaki klavyeden girilmiş gibi işlem gerçekleşir.

Bit Operatörleri (Bitwise Operators)

Bit operatörleri sayıların karşılıklı bitleri üzerinde işlem yapan operatörlerdir. Bit operatörleri şunlardır:

~	Bit Not (Bitwise Not)
<<	Sola Öteleme (left shift)
>>	Sağa Öteleme (right shift)
&	Bit And (Bitwise And)
	Bit Or (Bitwise Or)
^	Bit Exor (Bit Exclusive Or)

Bit And ve Bit Or Operatörleri

Tek ampersand Bit And, Tek çubuk Bit Or operatörü anlamına gelir. (Anımsanacağı gibi bunlardan iki tane yan yana getirilirse mantıksal And ve Mantıksal Or operatörleri anlaşılır.) Bu operatörler sayının karşılıklı bitlerini And ve Or işlemine sokarlar. Örneğin:

```
#include <stdio.h>  
  
int main(void)  
{  
    unsigned char a = 0x5F;  
    unsigned char b = 0xC4;  
    unsigned c;  
  
    c = a & b;  
    printf("%02X\n", c);        /* 0x44 */  
  
    c = a | b;
```

```
printf("%02X\n", c);      /* 0xDF */
return 0;
}
```

Soru: Bir sayının n'inci bitinin durumunu belirleyiniz.

Yanıt: Sayı tüm bitleri sıfır n'inci 1 olan bir sayıyla Bit And işlemine sokulur. Sonuç sıfırsa n'inci sıfır, sonuç sıfırdan farklıysa n'inci bir 1'dir.

Soru: Sayının diğer bitlerine dokunmadan n'inci bitini 0 yapınız.

Yanıt: O sayıyı bütün bitleri 1 olan n'inci biti 0 olan bir sayıyla Bit And işlemine sokarız.

Soru: Sayının diğer bitlerine dokunmadan n'inci bitini 1 yapınız.

Yanıt: O sayıyı bütün bitleri 0 olan n'inci biti 1 olan bir sayıyla Bit Or işlemine sokarız.

Örneğin:

```
#include <stdio.h>

int main(void)
{
    unsigned char a = 0x7F;

    a &= 0xDF;

    printf("%02X\n", a);      /* 0x5F */

    return 0;
}
```

Bit Exor Operatörü

Eğer bitler aynıysa 0 değerini veren bitler farklıysa 1 değerini veren işleme EXOR (exclusive Or) işlemi denilmektedir. EXOR işleminin doğruluk tablosu şöyledir:

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Örneğin:

```
#include <stdio.h>

int main(void)
{
    unsigned char a = 0x9A;
    unsigned char b = 0x7C;
    unsigned char c;
```

```

c = a ^ b;

printf("%02X\n", c);      /* 0xE6 */

return 0;
}

```

Exor geri dönüşümlü bir operatördür. Bu nedenle şifreleme işlemlerinde çok kullanılır. Yani $a \oplus b = c$ ise, $c \oplus b = a$ ve $c \oplus a = b$ 'dir. Böylece bir dosyanın byte'ları birtakım değerlerle Exor çekilerek bozulmuşsa yine aynı değerlerle exor çekilerek düzeltilebilir. (Yani değeri bozan anahtarla açan anahtar aynı olabilmektedir.) Örneğin Exor ile şifreleme için şöyle bir örnek verilebilir:

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *f;
    char passwd[128];
    unsigned seed;
    int i;
    int ch;

    printf("Enter password:");
    gets(passwd);
    seed = 0;
    for (i = 0; passwd[i] != '\0'; ++i)
        seed = seed * 31 + passwd[i];

    if ((f = fopen("test.txt", "r+")) == NULL) {
        fprintf(stderr, "cannot open file!..\n");
        exit(EXIT_FAILURE);
    }
    srand(seed);

    while ((ch = fgetc(f)) != EOF) {
        ch ^= rand() % 256;
        fseek(f, -1, SEEK_CUR);
        fputc(ch, f);
        fseek(f, 0, SEEK_CUR);
    }

    fclose(f);

    return 0;
}

```

Exor işleminde 0 etkisiz elemandır. 1 ise tersleme işlemi yapar.

Soru: Bir sayının diğer bitlerine dokunmadan n'inci bitini onun tersiyle yer değiştiriniz.

Yanıt: Sayı tüm bitleri 0 olan n'inci biti 1 olan bir sayıyla Exor çekilir.

Bit Not Operatörü

C'de ~ sembolü tek operandlı önek bir bit operatörüdür. Sayının 0 olan bitlerini 1, 1 olan bitlerini sıfır yapar.

Örneğin :

```

#include <stdio.h>

```

```

int main(void)
{
    unsigned a = 0, b;

    b = ~a;
    printf("%u\n", b);    /* 4294967295 */

    return 0;
}

```

&, |, ^ ve ~ Operatörlerinin Öncelik Tablosundaki Yerleşimi

Bit operatörlerinden ~ operatörü tek operandlı olduğu için öncelik tablosunun ikinci düzeyinde sağdan-sola grupta bulunur. & ve | operatörleri mantıksal operatörlerden daha önceliklidir.

() [] -> .	Soldan Sağa
+ - ++ -- ! (tür), sizeof * & ~	Sağdan Sola
* / %	Soldan Sağa
+ -	Soldan Sağa
< > <= >=	Soldan Sağa
== !=	Soldan Sağa
&	Soldan Sağa
^	Soldan Sağa
	Soldan Sağa
&&	Soldan Sağa
	Soldan Sağa
? :	Sağdan Sola
= += -= *= /= %= = &= ^=	Sağdan Sola
,	Soldan Sağa

& ve | operatörleri karşılaştırma operatörlerinden daha düşük önceliklidir. Maalesef programcılar aşağıdaki gibi hataları sık yapmaktadır:

```

if (a & 0x80 != 0) {
    ...
}

```

Burada sayının en yüksek anlamlı bitinin 1 olup olmadığına bakılmak istenmiştir. Ancak != operatörü & operatöründen daha öncelikli olduğu için kullanım hatalıdır. İşlemin şöyle yapılması gerekirdi:

```

if ((a & 0x80) != 0) {
    ...
}

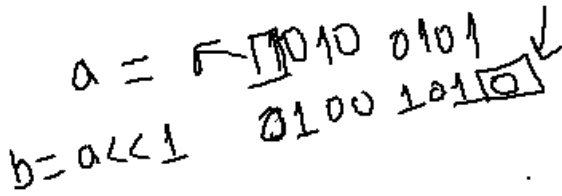
```

Bazı derleyiciler bu tür durumlarda uyarı ile programcının dikkatini çekmektedir.

Sola ve Sağa Öteleme Operatörleri

C'de << operatörüne sola öteleme (left shift), >> operatörüne ise sağa öteleme (right shift) operatörü denilmektedir. Bu operatörler iki operandlı aralık operatörlerdir.

Öteleme operatörlerinin sol tarafındaki operand'lar ötelenecek değeri, sağ tarafındaki operand'lar öteleme miktarını belirtir. Sola ötelemede her bir bir sola kaydırılır. En soldaki bit kaybedilir, en sağdan sıfırla besleme yapılır. Örneğin:



Sayıyı 1 kez sola ötelemek onu ikiyle çarpmak anlamına gelir. Örneğin:

```
a = 0x02; /* 0000 0010 */  
b = a << 1; /* 0000 0100 */
```

Örneğin:

```
#include <stdio.h>  
  
int main(void)  
{  
    unsigned int a = 2, b;  
  
    b = a << 1;  
    printf("%d\n", b);  
  
    return 0;  
}
```

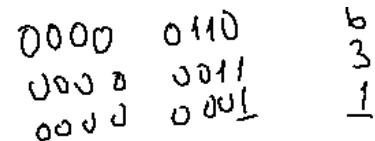
İşaretili sayılarda sola öteleme işlemi sırasında taşma olabilir. Eğer sola ötelemede taşma olursa tanımsız davranış (undefined behavior) oluşur. Örneğin:

```
int a = -2147483647, b;  
b = a << 1; /* Bu sayının iki katı int sınırlarının dışında! Tanımsız davranış! */
```

Örneğin:

```
int a = -1, b;  
  
b = a << 1; /* sorun yok b = -2 */
```

Sağa ötelemede ise sayının bütün bitleri bir sağa kaydırılır. en sağdaki bit kaybedilir. En soldan sıfır ile besleme yapılır. Sayıyı bir kez sağa ötelemek 2'ye tam bölmek anlamına gelir.



Örneğin:

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

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

    b = a >> 2;
    printf("%d\n", b);    /* 25 */

    return 0;
}
```

Eğer sayı işaretliyse ve negatifse en soldan besleme 0 ile yapılırsa sayı pozitif hale gelir. Sayının negatifliğinin korunması için en soldan 1 ile beslenmesi gerekir. İşte standartlara göre işaretli negatif sayıların sağa ötelenmesinde en soldan 0'la mı besleme yapılacağı yoksa 1 ile mi besleme yapılacağı (başka bir deyişle işaret bitinin korunup korunmayacağı) derleyicileri yazanların isteğine bırakılmıştır. Bu nedenle işaretli sayıların sağa ötelenmesine dikkat edilmelidir. Örneğin Microsoft derleyicileri ve gcc derleyicileri işaret bitini korumaktadır:

```
#include <stdio.h>

int main(void)
{
    int a = -2, b;

    b = a >> 2;
    printf("%d\n", b);    /* -1 */

    return 0;
}
```

Biz sola ya da sağa istediğimiz kadar öteleme yapabiliriz. Kuralları özetlemek gerekirse:

- İşaretsiz sayıların sola ve sağa ötelenmesinde tanımsız davranış ya da derleyiciye bağlı davranış gözükmez. Sola ötelemede taşma olsa bile yüksek anlamlı bit atılır.
- İşaretli sayıların sola ötelenmesinde taşma olursa tanımsız davranış oluşur.
- İşaretli negatif sayıların sağa ötelenmesinde işaret bitinin korunup korunmayacağı derleyicileri yazanların isteğine bırakılmıştır.

Soru: Sayının n'inci, l'leyiniz . Fakat n'in değerini bilmiyoruz. (Örneğin n klavyeden giriliyor.)

Yanıt: Mademki n değeri bilinmiyor işlem şöyle yapılabilir (a ilgili sayı olsun, n de bit numarasını gösterebilir):

```
b = a | 1 << n;
```

Soru: Sayının n'inci, 0 layınız. Fakat n'in değerini bilmiyoruz. (Örneğin n klavyeden giriliyor.)

Yanıt: İşlem şöyle yapılabilir:

```
b = a & ~(1 << n )
```

Soru: Klavyeden bir n değeri isteyiniz. Öyle bir sayı oluşturunuz ki, son n biti 1 olsun diğer bitleri 0 olsun. Örneğin n = 4 için şöyle bir sayı oluşturulmalıdır:

```
0000 1111
```

n = 2 için şöyle bir sayı oluşturulmalıdır:

0000 0011

Yanıt: İşlem şöyle yapılabilir:

$a = \sim(\sim 0U \ll n);$

Soru: Klavyeden bir n değeri isteyiniz. Öyle bir sayı oluşturunuz ki ilk n biti 1 olsun diğer bitleri 0 olsun. Örneğin $n = 4$ için şöyle bir sayı oluşturulmalıdır:

1111 0000

Yanıt:

$a = \sim(\sim 0U \gg n)$

Soru: İşaretsiz bir sayı var. 1 olan bitlerinin kaç tane olduğunu bulunuz.

Yanıt: Eğer döngüsz yapılacaksa bir look-up table oluşturulur. 1 byte'lık sayılar için bu mümkün olsa da 2 byte'lık ve daha uzun sayılar için sorunludur. Örneğin:

```
#include <stdio.h>

int main(void)
{
    unsigned n, count;

    printf("Sayı giriniz:");
    scanf("%d", &n);

    count = 0;
    do {
        if (n & 1)
            ++count;
        n >>= 1;
    } while (n);

    printf("%d\n", count);

    return 0;
}
```

Öteleme operatörleri öncelik tablosunda aritmetik operatörlerle karşılaştırma operatörleri arasında bulunur. Öncelik tablonun son şekli şöyledir:

() [] -> .	Soldan Sağa
+ - ++ -- ! (tür), sizeof * & ~	Sağdan Sola
* / %	Soldan Sağa
+ -	Soldan Sağa
<< >>	Soldan Sağa
< > <= >=	Soldan Sağa
== !=	Soldan-Sağa
&	Soldan Sağa
^	Soldan Sağa

	Soldan Sağa
&&	Soldan Sağa
	Soldan Sağa
? :	Sağdan Sola
= += -= *= /= %= = &= ^=	Sağdan Sola
,	Soldan Sağa

Programların Komut Satırı Argümanları

Bir programı komut satırından çalıştırırken programdan isminden sonra girilen yazılara komut satırı argümanları (command line arguments) denilmektedir. Standartlara göre C'de main fonksiyonunun geri dönüş değeri int olmak zorundadır. main fonksiyonunun parametresi ya void olmak zorundadır ya da main iki parametreye sahip olabilir. Parametrelerden biri int türden diğeri char ** türündendir. Bu parametre char *[] biçiminde de ifade edilemektedir. Yani main fonksiyonu aşağıdaki iki biçimden biri olarak tanımlanmak zorundadır:

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

```
int main(int argc, char *argv[])
{
    ...
}
```

İkinci biçimdeki parametre değişken isimleri istenildiği gibi alınabilir. Ancak argc (argument counter) ve argv (argument variable list) isimleri gelenekseldir.

main fonksiyonuna bu argümanları işletim sistemi ve derleyicilerin giriş kodları geçirmektedir. Birinci parametre komut satırına girilen boşlukla ayrılmış yazıların sayısını belirtir (programın ismi de dahil). Örneğin:

```
ls -l -i
```

Burada argc 3 olarak geçirilir. Yalnızca programın ismini yazarak programı çalıştırmak isteseydik argc değeri 1 olurdu. argv ise komut satırı argümanlarına ilişkin yazıların başlangıç adreslerinin bulunduğu diziyi belirtir. Yani argv'nin her elemanı sırasıyla programın isminden başlayarak bir argümanı bize verir. Dizinin sonunda NULL adresin bulunacağı garanti edilmektedir. Örneğin:

```
ls -l -i
```

```
argv[0] → ls | \0
argv[1] → -l | \0
argv[2] → -i | \0
argv[3] → NULL
```

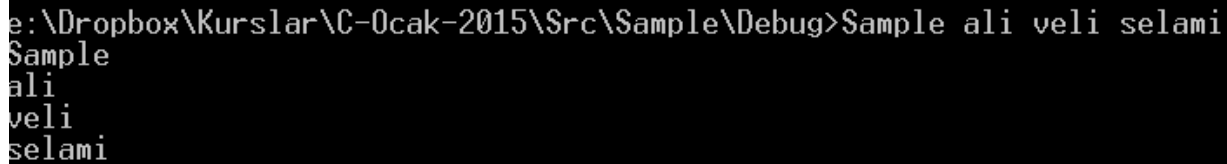

Örneğin:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    for (i = 0; i < argc; ++i)
        printf("%s\n", argv[i]);

    return 0;
}
```



```
e:\Dropbox\Kurslar\C-0cak-2015\Src\Sample\Debug>Sample ali veli selami
Sample
ali
veli
selami
```

Aynı döngü şöyle de kurulabilirdi:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    for (i = 0; argv[i] != NULL; ++i)
        printf("%s\n", argv[i]);

    return 0;
}
```

Visual Studio IDE'sinde komut satırı argümanları proje ayarlarında "Debugging / Command Arguments" kısmından girilebilir.

Örneğin type (ya da cat) işlemini yapan bir program şöyle yazılabilir:

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

int main(int argc, char *argv[])
{
    FILE *f;
    int ch;

    if (argc == 1) {
        printf("usage: <mytype> <path>\n");
        exit(EXIT_FAILURE);
    }

    if (argc > 2) {
        fprintf(stderr, "too many arguments!..\n");
        exit(EXIT_FAILURE);
    }

    if ((f = fopen(argv[1], "r")) == NULL) {
        fprintf(stderr, "cannot open file: %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }
}
```

```

}

while ((ch = fgetc(f)) != EOF)
    putchar(ch);

if (ferror(f)) {
    fprintf(stderr, "IO error!\n");
    exit(EXIT_FAILURE);
}

return 0;
}

```

Programın başında komut satırı argümanlarının kontrol edilmesiyle sık karşılaşılmaktadır. Yukarıda da gördüğünüz gibi önce program uygun komut satırı argümanlarıyla çalıştırılmış mı diye bakılmıştır.

Komut satırı argümanlarının birer yazı biçiminde bize verildiğine dikkat ediniz. Örneğin biz n tane sayının toplamını yazdıran add isimli bir program yazmak isteyelim:

```
./add 10 20 30 40
```

Önce bu tazıları sayıya dönüştürmemiz gerekir. Bunun için ise atoi, atol ya da atof fonksiyonları kullanılmaktadır.

atoi, atol ve atof Fonksiyonları

Prototipi <stdlib.h> dosyasında bulunan bu standart C fonksiyonları bizden sayı belirten bir yazıyı parametre olarak alır ve onu gerçekten sayı olarak bize verir:

```

int atoi(const char *str);
long atol(const char *str);
double atof(const char *str);

```

Örneğin:

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char s[] = "123.45";
    double d;

    d = atof(s);
    printf("%f\n", d);

    return 0;
}

```

Bu fonksiyonlar yazının başındaki boşluk karakterlerini (white space) atarlar. İlk uygun olmayan karakter gördüklerinde işlemini sonlandırırlar. Hiçbir uygun karakter bulunamazsa bu fonksiyonlar 0 ile geri dönmektedir. atoi fonksiyonu şöyle yazılabilir.

```

#include <stdio.h>
#include <ctype.h>

int myatoi(const char *str)
{
    int val = 0;
    int sign = 1;

```

```

while (isspace(*str))
    ++str;

if (*str == '-') {
    sign = -1;
    ++str;
}

while (isdigit(*str)) {
    val = val * 10 + *str - '0';
    ++str;
}

return val * sign;
}

int main(void)
{
    char s[] = " -1205";
    int i;

    i = myatoi(s);
    printf("%d\n", i);    /* 12 */

    return 0;
}

```

Şimdi yukarıda belirtilen add programını yazalım:

```

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    double total = 0;
    int i;

    for (i = 1; i < argc; ++i)
        total += atof(argv[i]);

    printf("%f\n", total);

    return 0;
}

```

itoa ve ltoa Fonksiyonları

itoa fonksiyonu atoi fonksiyonun, ltoa fonksiyonu da atol fonksiyonun ters işlemini yapan fonksiyonlardır. Ancak bu fonksiyonlar standart C fonksiyonları değildir. Dolayısıyla bazı C derleyicilerinde bulunmayabilirler. Örneğin bu fonksiyonlar Microsoft ve Borland derleyicilerinde varken, gcc derleyicilerinde bulunmamaktadır. Bunun yerine standart sprintf fonksiyonu kullanılabilir.

sprintf Fonksiyonu

sprintf fonksiyonu printf fonksiyonunun char türden bir diziye yazan versiyonudur. Nasıl fprintf fonksiyonu printf'in dosyaya yazan versiyonuysa sprintf de diziye yazan versiyonudur. Örneğin:

```

#include <stdio.h>

int main(void)
{

```

```

char s[100];
int a = 10, b = 20;

sprintf(s, "a = %d, b = %d", a, b);
puts(s);

return 0;
}

```

sprintf fonksiyonunun birinci parametresi char türden bir adrestir. Diğer parametreleri printf ile aynıdır. Genel olarak bir sayıyı yazıya dönüştürmek için de sprintf kullanılabilir. Örneğin:

```

#include <stdio.h>

int main(void)
{
    char s[100];
    double d = 12.345;

    sprintf(s, "%f", d);
    puts(s);

    return 0;
}

```

sscanf Fonksiyonu

Bu fonksiyon scanf'in char türden diziden okuyan versiyonudur. Yani fscanf nasıl dosyadan okuyorsa, sscanf de diziden girilenleri sanki klavyeden (stdin dosyasından) girilmiş gibi ele alır. Örneğin:

```

#include <stdio.h>

int main(void)
{
    char s[100] = "    123 456 ";
    int a, b;

    sscanf(s, "%d%d", &a, &b);

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

    return 0;
}

```

sscanf fonksiyonunun birinci parametresi char türden dizinin adresini alır. Diğer parametreleri scanf fonksiyonunda olduğu gibidir. Dolayısıyla bu fonksiyon da yazıyı sayıya dönüştüren atoi, atol ve atof fonksiyonlarının daha genel bir biçim olarak kullanılabilir.

Gösteriyi Gösteren Göstericiler (Pointers to Pointer)

Göstericiler de birer nesne olduğuna ve bellekte yer kapladığına göre onların da adresleri alınabilir. Bir göstericinin adresi alınırsa nasıl bir göstericiye yerleştirilmelidir. İşte T türünden bir göstericinin adresi T* türünden bir göstericiye yerleştirilebilir. Böyle göstericiler iki tane * atomu ile bildirilirler. Örneğin:

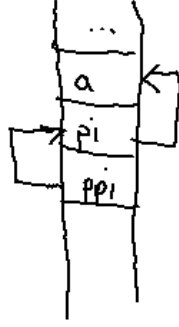
```
int **ppi;
```

Burada ppi bir gösteriyi gösteren göstericidir. Yani gösterici türünden göstericidir. Biz ppi'yi * operatörüyle kullanırsa (*ppi'yi kapatıp sola bakın) elde ettiğimiz nesne int * türündendir. Yani bir göstericidir. Yani:

```
ppi, int ** türündendir.
```

*ppi, int * türündendir.
**ppi, int türündendir.

```
int a = 10;  
int *pi = &a;  
int **ppi = &pi;
```



Örneğin:

```
#include <stdio.h>  
  
int main(void)  
{  
    int a = 10;  
    int *pi;  
    int **ppi;  
  
    pi = &a;  
    ppi = &pi;  
  
    printf("%d\n", **ppi);  
  
    return 0;  
}
```

Anımsanacağı gibi bir dizinin ismini bir ifadede kullandığımızda aslında o dizinin adresini kullanmış oluruz. Başka bir deyişle bir dizinin ismi o dizinin ilk elemanın adresi gibidir. O halde bir gösterici dizisinin ismi de o gösterici dizisinin ilk elemanın adresi gibi olacağına göre iki yıldızlı bir göstericiye atanabilir. Örneğin:

```
char *s[10];
```

Burada s ifadesi char ** türündendir. Yani char türden bir göstericiyi gösteren göstericiye atanabilir:

```
char **ppc;
```

```
ppc = s;
```

Örneğin biz bir gösterici dizini bir fonksiyona geçirmek istesek fonksiyonun parametre değişkeni göstericiyi gösteren gösterici olmalıdır. Örneğin:

```
#include <stdio.h>  
  
void disp_names(char **ppnames);  
  
int main(void)  
{  
    char *names[] = { "ali", "veli", "selami", "ayse", "fatma", NULL };  
  
    disp_names(names);  
  
    return 0;  
}
```

```

void disp_names(char **ppnames)
{
    int i;

    for (i = 0; ppnames[i] != NULL; ++i)
        printf("%s\n", ppnames[i]);
}

```

Fonksiyonun göstericiyi gösteren gösterici parametresi yine dizi formunda belirtilebilir. Yani örneğin:

```

void disp_names(char **ppnames)
{
    ...
}

```

ile,

```

void disp_names(char *ppnames[])
{
    ...
}

```

aynı anlamdadır. Ya da örneğin:

```

int main(int argc, char *argv[])
{
    ...
}

```

ile,

```

int main(int argc, char **argv)
{
    ...
}

```

aynı anlamdadır.

Bazen fonksiyona bir gösterici bir göstericinin adresi gönderilir. Fonksiyon da o göstericinin içerisine birşeyler yazabilir. Örneğin:

```

#include <stdio.h>
#include <stdlib.h>

void mallocstr(char **ptr, size_t size)
{
    *ptr = (char *)malloc(size);

    if (*ptr == NULL) {
        fprintf(stderr, "cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
}

int main(void)
{

```

```

char *str;

mallocstr(&str, 32);
gets(str);
puts(str);

return 0;
}

```

Bir göstericiyi gösteren göstericiyi 1 artırdığımızda içindeki adresin sayısal değeri kaç artar? Yanıt bir gösterici kadar artar. Örneğin `argv` `char **` türünden olsun. `++argv` yaptığımızda `argv`'nin içerisindeki adres bir gösterici (yani 32 bit sistemlerde 4 byte, 64 bit sistemlerde 8 byte) artacaktır. Örneğin:

```

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    while (*argv != NULL)
        puts(*argv++);

    return 0;
}

```

Bilindiği gibi C'de `void *` türünden bir göstericiye biz her türden adresi atayabiliriz. Ancak `void **` türünden göstericiye her adresi atayamayız. Yalnızca `void *` türünden göstericinin adresini atayabiliriz. Örneğin:

```

int a;
int *pi;
void *pv;
void **ppv;

pi = &a;          /* geçerli */
pv = pi;         /* geçerli */
ppv = &pv;       /* geçerli */
ppv = &pi;       /* geçersiz! */

```

`int *` türünün `void *` türüne atanması normal ve geçerlidir. Ancak bu `int **` türünün `void **` türüne atanabileceği anlamına gelmez. `void **` türüne biz yalnızca `void *` türünden bir adresi atayabiliriz. Biz aslında `int **` türünü de istersek `void *` türünden bir göstericiye atayabiliriz. Örneğin:

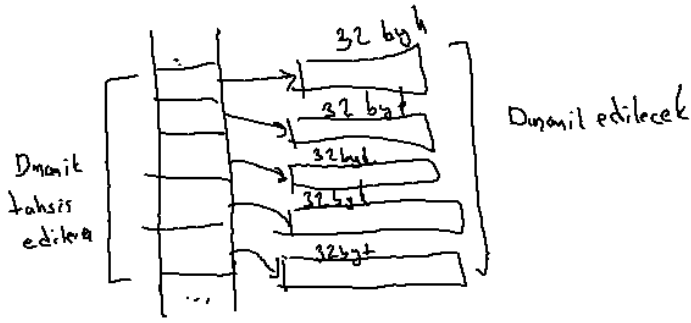
```

int a;
int *pi;
int **ppi;
void *pv;
int **ppi2;

pi = &a;          /* geçerli */
ppi = &pi;       /* geçerli */
pv = ppi;        /* geçerli */
ppi2 = (int **)pv; /* tür dönüştürmesi zorunlu değildir */

```

Gösterici dizileri için dinamik tahsisatlar yapabiliriz. Örneğin önce 5 elemanlı `char` türden bir gösterici dizisini dinamik olarak tahsis edelim. Sonra da dizinin her elemanının 32 byte'lık `char` türden dinamik tahsis edilmiş bir diziyi göstermesini sağlayalım:



```

char **ppc;
int i;

if ((ppc = (char **)malloc(5 * sizeof(char *))) == NULL) {
    fprintf(stderr, "cannot allocate memory!..\n");
    exit(EXIT_FAILURE);
}

for (i = 0; i < 5; ++i)
    if ((ppc[i] = (char *)malloc(32)) == NULL) {
        fprintf(stderr, "cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }

...
for (i = 0; i < 5; +i)
    free(ppc[i]);
free(ppc);

```

Soru: Yukarıdaki tahsisat sistemini tek malloc ile oluşturunuz. Böylece tek free ile sistem serbest bırakılsın.

Yanıt: Sistem için gereken tüm bellek $5 * \text{sizeof(char *)} + 5 * 32$ byte kadardır. Önce bu tahsis edilir. Sonra da gösterici dizisinin elemanlarının kendi alanlarını göstermesi sağlanır:

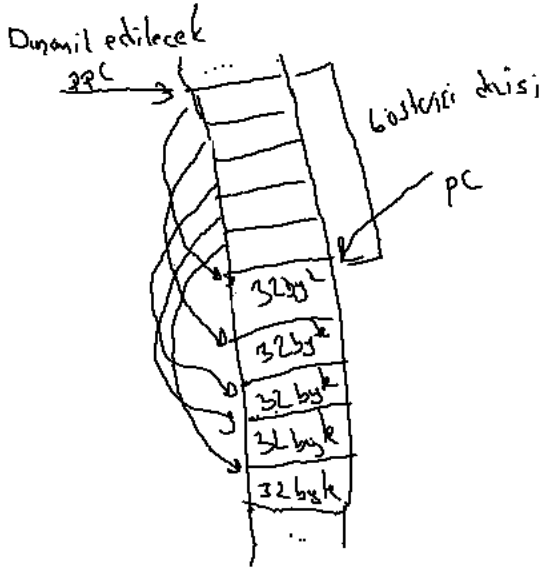
```

char **ppc;
char *pc;
int i;

if ((ppc = (char **)malloc(5 * (sizeof(char *) + 32))) == NULL) {
    fprintf(stderr, "cannot allocate memory!..\n");
    exit(EXIT_FAILURE);
}
pc = (char *)(ppc + 5);

for (i = 0; i < 5; ++i) {
    ppc[i] = pc;
    pc += 32;
}
...
free(ppc);

```

Tabi böyle bir sistemde 32 byte'lık diziler daha sonra realloc ile büyütülü küçültülemezler.

C'de aslında göstericiyi gösteren göstericiyi gösteren göstericiler ve daha çok kademeli göstericiler de bildirilebilir. Ancak üç yıldızlı göstericilerin bile pek bir kullanım alanı yoktur. Biz bir göstericiyi gösteren göstericinin adresini üç yıldızlı bir göstericiye yerleştirebiliriz. Örneğin:

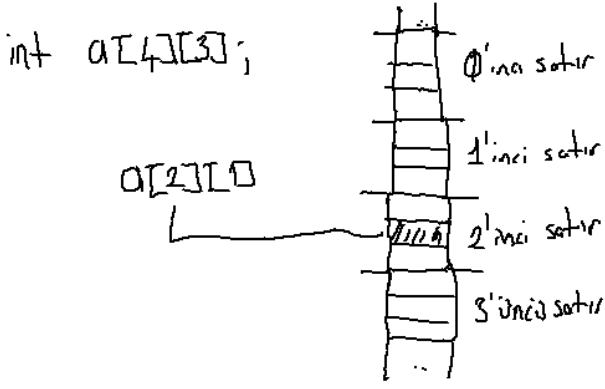
```
int a = 10;
int *pi;
int **ppi;
int ***pppi;
int ****ppppi;
```

```
pi = &a;
ppi = &pi;
pppi = &ppi;
ppppi = &pppi;
```

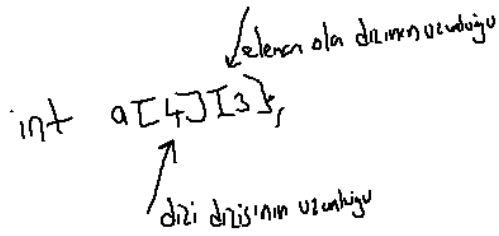
Çok Boyutlu Diziler

Çok boyutlu diziler bazı olayları temsil ederken okunabilirliği artırmak için kullanılabilir. Örneğin bir satranç tahtasını temsil etmek için iki boyutlu bir dizi kullanabiliriz. Bir matematiksel bir matrisi temsil etmek için yine iki boyutlu bir diziden faydalanabiliriz. Dizilerin boyut sayıları fazla olabilsede pratikte en çok kullanılan çok boyutlu diziler iki boyutlu dizilerdir. İki boyutlu dizilere matris de denilmektedir.

C'de çok boyutlu diziler aslında bellekte tek boyutlu diziymiş gibi tutulmaktadır. Zaten bellek çok boyutlu değildir. Tek boyutludur. Çok boyutlu diziler aslında "dizi dizileri" gibi düşünülmelidir. Örneğin 4x3'lik bir matris aslında her elemanı 3 elemanlık dizi olan 4'lik dizi gibidir. C'de çok boyutlu diziler birden fazla köşeli parantezle bildirilirler. Örneğin:



Bildirimdeki ilk köşeli parantez dizi dizisinin uzunluğunu, ikinci köşeli parantez eleman olan dizilerin uzunluğunu belirtir. Örneğin:



Çok boyutlu bir diziye birden fazla küme paranteziyle ilkdeğer verilebilir. Örneğin:

```
int a[4][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}};
```

Örneğin:

```
#include <stdio.h>

int main(void)
{
    int a[4][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } };
    int i, k;

    for (i = 0; i < 4; ++i) {
        for (k = 0; k < 3; ++k)
            printf("%d ", a[i][k]);
        printf("\n");
    }

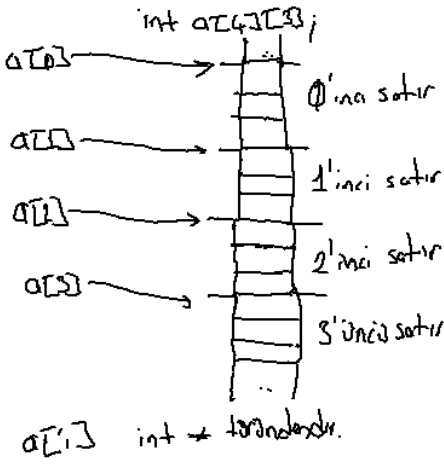
    return 0;
}
```

Aslında iç küme parantezleri ayrıca kullanılmak zorunda değildir. Örneğin:

```
int a[4][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

Fakat bir elemanın yazılması unutulduğunda diğer elemanlar birer kayabilir. Bu nedenle iç küme parantezlerinin belirtilmesi tavsiye edilir.

Biz matriste erişim için yalnızca ilk köşeli parantezi kullanırsak (örneğin a[2]) bu ifade bir nesne belirtmez. Dizi dizisinin o indise ilişkin dizisinin başlangıç adresini belirtir. Örneğin:



Örneğin:

```
#include <stdio.h>

int main(void)
{
    int a[4][3];
    int *pi;

    pi = a[2];    /* geçerli */
    pi[0] = 10;   /* a[2][0] = 10 */
    pi[1] = 20;   /* a[2][1] = 20 */
    pi[2] = 30;   /* a[2][2] = 30 */

    printf("%d, %d, %d\n", a[2][0], a[2][1], a[2][2]);

    return 0;
}
```

Bir matrisin yani dizisinin ismi yine o dizi dizisinin başlangıç adresini belirtir. Bir dizi dizisinin başlangıç adresi dizi göstericisine atanabilir. Gösterdiği yer bir dizi olan göstericiye dizi göstericisi (pointer to array) denir. Dizi göstericileri şöyle bildirilir:

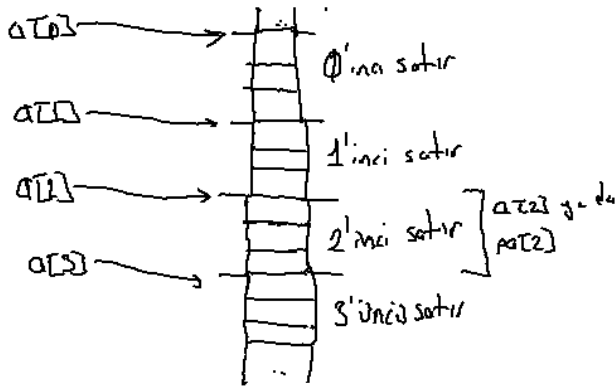
<tür> (*<gösterici_ismi>)[<sütun uzunluğu>];

Örneğin:

```
int (*pa)[4];
```

Burada pa'nın gösterdiği yerde (yani *pa) int[4] türünden bir nesne başka bir deyişle 4 elemanlı bir dizi vardır. *pa bir nesne değildir, sanki bir dizinin ismi gibidir. İşte int[N][M] türünden bir matrisin ismi böyle bir göstericiye atanabilir. Örneğin:

```
int a[4][3];
int (*pa)[3];
pa = a;
a[2][2] ile
pa[2][1] eşleşir
```



Aşağıdaki gibi bir matris bildirimi yapılmış olsun:

```
int a[4][3];
```

Burada a'yı ve a[i]'yi atayacağımız göstericiler nasıl olmalıdır?

```
mat a[4][3];
mat *pi;
int (*pa)[3];
pa = a;
pi = a[2];
```

Aşağıdaki gibi bir dizi göstericisi bildirilmiş olsun:

```
int (*pa)[3];
```

Burada pa'yı bir artırdığımızda pa'nın içerisindeki adres $3 * \text{sizeof}(\text{int})$ kadar artar.

Bir matrisin ismini atayacağımız dizi göstericisinin sütun uzunluğuyla matrisin sütun uzunluğu aynı olmak zorundadır. Örneğin:

```
int a[4][3];
int (*pa)[5];
pa = a; // geçersiz * /
```

İkiden fazla boyut söz konusu olduğunda aynı prensip söz konusudur. Örneğin:

```
int a[3][4][5];
```

Burada a "dizi dizisi dizisidir". a adresini atayacağımız dizi göstericisi de iki köşeli parantez içermelidir:

```
int (*pa)[4][5];
```

```
pa = a;
```

Yani birinci boyut dışındaki bütün boyutların uzunlukları belirtilmek zorundadır. Burada *pa nesne değildir. pa[i][k]'da nesne değildir. Ancak pa[i][k][j] bir nesnedir. Peki a[i]'yi atayacağımız gösterici nasıl olmalıdır? Yanıt:

```
int (*pa)[5];
```

biçiminde olmalıdır. Çünkü a[i] bir matrisin adresi gibidir. O da yukarıdaki gibi bir göstericiye atanabilir.

Peki bir matrisi fonksiyona parametre olarak nasıl geçirebiliriz:

```
#include <stdio.h>

void disp(int(*pa)[3], int size)
{
    int i, k;

    for (i = 0; i < size; ++i) {
        for (k = 0; k < 3; ++k)
            printf("%d ", pa[i][k]);
        printf("\n");
    }
}

int main(void)
{
    int a[4][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } };

    disp(a, 4);

    return 0;
}
```

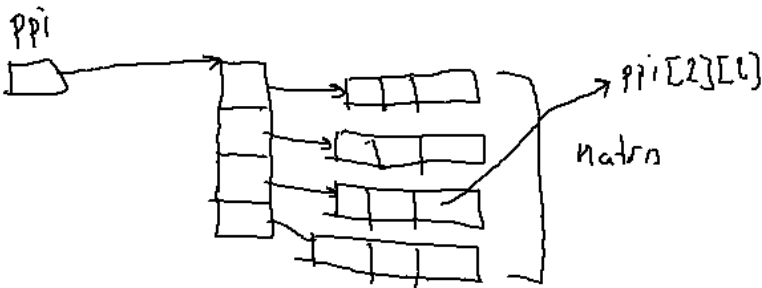
Göstericiyi Gösteren Göstericiler ve Matrisler

Matrisel bir sistem oluşturabilmek için iki seçenek söz konusu olabilir:

1) Önce göstericiyi gösteren gösterici için sonra da onun elemanı olan göstericiler için tahsisat yapmak. Örneğin:

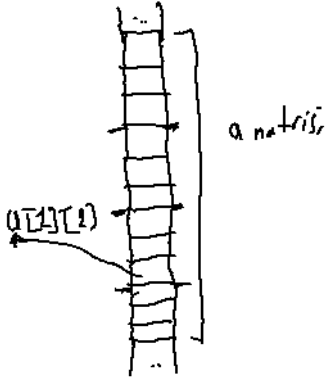
```
int **ppi;
int i;

ppi = (int **)malloc(4 * sizeof(int *));
for (i = 0; i < 4; ++i)
    ppi[i] = (int *) malloc(3 * sizeof(int));
```



2) Doğrudan çok boyutlu bir dizi kullanmak.

```
int ppi[4][3];
```



Şüphesiz ikinci biçim bellekte daha az yer kaplamaktadır. Ancak birinci biçimde matrisin her satırı aynı uzunlukta olmak zorunda değildir.

Bir matris için dinamik tahisat yapılabilir. Örneğin:

```
int(*pa)[3];  
pa = (int(*)[3])malloc(4 * 3 * sizeof(int));
```

Tabii typedef bildirimini ile türler daha sade gösterilebilir. Örneğin:

```
typedef int(*PA3)[3];  
  
PA3 pa;  
pa = (PA3)malloc(4 * 3 * sizeof(int));
```

Makrolar

#define önişlemci komutunun parametrik kullanımına makro denilmektedir. Makro yazılırken #define komutunda STR1 yazısı parantez içerir. Parantez içerisindekiler makro parametrelerini belirtir. Makro parametrelise biz onu kullanırken sanki bir fonksiyon gibi argüman gireriz. Örneğin:

```
#define square(a) a * a  
...  
result = square(10);  
      ↗ 10 * 10
```

Burada square(10) ifadesini önişlemci 10 * 10 biçiminde açmaktadır. a makro parametresidir.

```
#include <stdio.h>  
  
#define square(a) a * a  
  
int main(void)  
{  
    int result;  
  
    result = square(10);
```

```

printf("%d\n", result);

return 0;
}

```

Örneğin:

```

#include <stdio.h>

#define square(a) a * a

int main(void)
{
    int result;
    int x = 10;

    result = square(x); /* result = x * x */
    printf("%d\n", result);

    return 0;
}

```

Bir makro fonksiyon gibi işlev görmelidir. Kodu inceleyen kişi onun bir fonksiyon mu yoksa makro mu olduğunu anlamak zorunda kalmamalıdır. Oysa yukarıdaki square tam olarak bir fonksiyon gibi kullanılamaz. Örneğin:

*Handwritten diagram illustrating the macro expansion of `result = square(5-2);`. The expression `square(5-2)` is shown. An arrow points from the `5-2` inside the macro to the `5-2` in the expanded expression `5-2 * 5-2`. Below this, the result of the calculation is shown as `-7`.*

Burada önişlemci `5 - 2` ifadesini `a` kabul edip bunu `5 - 2 * 5 - 2` biçiminde açar. Buradaki problemi ortadan kaldırmak için makro parametreleri paranteze alınmalı ve makro en dıştan ayrıca paranteze alınmalıdır. Örneğin:

```

#define square(a) ((a) * (a))

```

Şimdi artık square tam olarak bir fonksiyon etkisi yaratır. En dıştan paranteze alınmasının nedeni yüksek öncelikli operatörlerden tüm ifadenin etkilenmesini sağlamaktır. Örneğin:

```

#include <stdio.h>

#define max(a, b) ((a) > (b) ? (a) : (b))

int main(void)
{
    int a = 3, b = 4, c;

    c = max(a, b); /* c = ((a) > (b) ? (a) : (b)) */
    printf("%d\n", c);

    return 0;
}

```

Örneğin:

```

#include <stdio.h>

#define beep() putchar('\7');

int main(void)
{

```

```

beep();
beep();

return 0;
}

```

Makro daha karmaşık olabilir. Bu durumda birden fazla satıra yazmak okunabilirliği artırır. Anımsanacağı gibi \ karakterini hemen ENTER (LF) izlerse derleyici onun aşağısındaki satırı aynı satır gibi kabul eder. Örneğin:

```

#include <stdio.h>
#include <stdlib.h>

#define check(a)  if ((a) == 0) { \
                  fprintf(stderr, "Error\n"); \
                  exit(EXIT_FAILURE); \
                }

int main(void)
{
    int a = 0;

    check(a);

    return 0;
}

```

Ancak çok satırlı ve deyim içeren makrolar yazılırken dikkat edilmelidir. Yukarıdaki check(a); işlemi aslında önişlemci tarafından şöyle açılır:

```

if ((a) == 0) {
    fprintf(stderr, "Error\n");
    exit(EXIT_FAILURE);
};

```

Buradaki son noktalı virgül check çağrısının sonundaki noltalı virgüldür. Gerçi bu noktalı virgül derleme modülü tarafından boş deyim kabul edileceğinden çoğu kez soruna yol açmaz. Fakat aşağıdaki gibi bir kullanımda soruna yol açacaktır:

```

if (x != -1)
    check(x);
else
    foo();

```

→

```

if (x != -1)
    if ((x) == 0) {
        fprintf(stderr, "error\n");
        exit(EXIT_FAILURE);
    };
else
    foo();

```

Bu nedenle çok satırlı makroların sanki do-while deyimiymiş gibi yazılması gerekmektedir. Örneğin:

```

#define check(a)  do { \
                  if ((a) == 0) \
                    fprintf(stderr, "error\n"); \
                    exit(EXIT_FAILURE); \
                } while(0)

```


Artık if içerisinde çağrılan makro şöyle açılacaktır:

```
if (x != -1)
do {
    if (a == 0) {
        fprintf(stderr, "error\n");
        exit(EXIT_FAILURE);
    } while (p);
} else
    fool;
```

Böylece check fonksiyonunun sonundaki noktalı virgül do-while'ın noktalı virgüllü olur, boş deyim olmaz.

Bazı makroları çağırırken ++ ve -- operatörlerinin argümanlarda kullanılmaması gerekir. Örneğin:

```
#define square(a) ((a) * (a))
```

Biz böyle bir makroyu aşağıdaki gibi çağırılmış olalım:

```
int x = 3, y;

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

Burada square bir fonksiyon olsaydı hiçbir sorun ortaya çıkmazdı. Ancak makro olarak yazıldığında aynı ifadede birden fazla ++x gözüküyor olur. Böyle bir kod derleyici için tanımsız davranışa yol açacaktır.

Makro mu Fonksiyon mu?

Bir fonksiyonun çağırılması sırasında küçük bir maliyet söz konusudur. Bu maliyeti oluşturan etkenler şunlardır:

- Fonksiyon çağırılırken kullanılan CALL ve RET makina komutları
- Fonksiyona girişte ve çıkışta gereken bazı makina komutları (örneğin stack frame düzenlenmesi için gereken komutlar).
- Parametrelerin aktarımı için gereken makina komutları

Yukarıdaki makina komutları ortalama 10 civarındadır. Oysa bazen birer satırlık fonksiyonlar yazmak isteyebiliriz. Örneğin:

```
double square(double a)
{
    return a * a;
}
```

İşte bir iki satırlık küçük fonksiyonların fonksiyon olarak yazılması onların çağırılması sırasında görelili (çoğu zaman bunun hiçbir önemi yoktur) zaman kaybına yol açabilmektedir. Bu tür fonksiyonların fonksiyon olarak değil de makro olarak organize edilmesi uygun olur. Büyük kodların makro olarak tanımlanması kodu ciddi biçimde büyütebilmektedir. O halde çok küçük fonksiyonlar makro olarak diğerleri normal fonksiyon olarak yazılabilirler.

Şüphesiz makrolar başlık dosyalarına yerleştirilmelidir. Onları çağırarak kişi onların fonksiyon mu makro mu olduğunu bilmek zorunda değildir. Örneğin getchar fonksiyonu bazı derleyicilerde makro olarak yazılmıştır.

```
#define getchar() fgetc(stdin)
```

Diğer Önışlemci Komutları

Biz şimdiye kadar #include ve #define komutlarını gördük. Bu bölümde diğer önemli bazı önışlemci komutları görülecektir.

#if, #else, #elif ve #endif Komutu

#if önışlemci komutunun yanında tamsayı türlerine ilişkin bir sabit ifadesi bulunmak zorundadır. Örneğin:

```
#if MAX > 10
....
#else
...
#endif
```

Önışlemci #if komutunun yanındaki ifadenin sayısal değerini hesaplar. Bu değer sıfır dışı bir değerse #else'e kadar kısım derleme modülüne verilir, sıfır ise #else ile #endif arasındaki kısım derleme modülüne verilir.

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

#define SIZE 10000

int main(void)
{
    #if SIZE < 1000
        int a[SIZE];
    #else
        int *a;
        if ((a = (int *)malloc(SIZE * sizeof(int))) == NULL) {
            fprintf(stderr, "cannot allocate memory!..\n");
            exit(EXIT_FAILURE);
        }
    #endif
    ...

    return 0;
}
```

Burada SIZE değeri 1000'den küçükse dizi normal olarak büyükse dinamik olarak tahsis edilmiştir. Komutta #else kısmı olmak zorunda değildir.

#if komutu bir önışlemci komutu olduğuna göre işleme sokulması derleme modülünden önce yapılır. #elif komutu #else #if etkisi yaratır ancak bu durumda tek bir #endif yetmektedir. Örneğin:

```
#define SYSTEM 2

#if SYSTEM == 1
/* ... */
#elif SYSTEM == 2
/* ... */
#elif SYSTEM == 3
/* ... */
```

```
#endif
```

#ifdef, #else, #endif Komutu

#ifdef komutunu bir değişken ismi izlemek zorundadır. Bu değişken ismi bir sembolik sabit ya da makro ismidir. Önışlemci böyle bir sembolik sabitin ya da makronun daha önce #define komutu ile define edilip edilmediğine bakar. (Ancak onun kaç olarak define edildiğine bakmaz). Eğer o sembolik sabit ya da makro daha yukarıda define edilmişse önışlemci #else'e kadar olan kısmı derleme modülüne verir, define edilmemişse #else ile #endif arasındaki kısmı derleme modülüne verir. Örneğin:

```
#include <stdio.h>

#define TEST

int main(void)
{
#ifdef TEST
    printf("TEST define edilmiş!\n");
#else
    printf("TEST define edilmemiş!\n");
#endif

    return 0;
}
```

C derleyicileri bir sembolik sabiti derleme sırasında define edebilme olanacağını vermiştir. Örneğin gcc ve cl derleyicilerinde -D seçeneği ile bu yapılabilmektedir:

```
gcc -o sample -D TEST sample.c
```

Visual Studio IDE'sinde aynı etki proje seçeneklerinde "C-C++/Preprocessor/Preprocessor Definitions" menüsüyle yapılabilmektedir.

#ifdef çeşitli platformlar için farklı kodların derleme dahil edilmesi amacıyla çok sık kullanılır. Örneğin:

```
#ifdef UNIX
....
#else
...
#endif
```

Komutta #else kısmı olmak zorunda değildir.

#ifndef, #else, #endif Komutu

Bu komut #ifdef komutunun tam tersi işlem görmektedir. Yani yanındaki sembolik sabit ya da makro define edilmemişse #else'e kadar olan kısım #define edilmişse #else ile #endif arasındaki kısım derleme modülüne verilir.

#ifndef include korumalarında (include guards) çok sık kullanılmaktadır. Aynı başlık dosyasının ikinci include edilmesi pek çok derleme hatasının oluşmasına yol açabilir. (Örneğin aynı yapının iki kez bildirilmesi, aynı enum'un ikinci kez bildirilmesi, aynı typedef bildiriminin ikinci kez yapılması geçersizdir.) Bazen kontrolümüz dışında aynı dosyanın birden fazla kez include edilme durumu oluşabilmektedir. Örneğin a.h dosyası ve b.h dosyaları kendi içerisinde x.h dosyasını include etmiş olsun. Biz de bir uygulamada mecburen a.h ve b.h dosyalarını include edersek x.h dosyası iki kez include edilmiş olur. Büyük projelerde bu olasılık çok fazladır. include koruması şöyle oluşturulur:

```
/* x.h dosyası */
```

```
#ifndef X_H_
#define X_H_

...

...

...

#endif
```

Burada önışlemci x.h dosyasını ilk kez açtığıında X_H_ sembolik sabiti henüz define edilmediği için #endif'e kadar kısmı yani bütün dosya içeriğini derleme modülüne verir. İkinci kez aynı dosyayı açtığıında artık X_H_ sembolik sabiti define edilmiş olacağı için artık dosya içeriğini derleme modülüne vermez. Bu örnekteki X_H_ makro ismi dosya isminden hareketle uydurulmuş herhangi bir isimdir. Tüm include dosyaları böyle bir koruma ile oluşturulmalıdır (çünkü onların her zaman dolaylı olarak birden fazla kez include edilmesi mümkün olabilmektedir.)

#error Komutu

#error komutu önışlemci tarafından derlem işlemini fatal error ile sonlandırır. Komutun yanında bir yazı bulunur. Bu yazı ekrana hata mesajı olarak yazdırılır (Yazının iki tırnak içerisinde olması gerekmez.) Örneğin:

```
#include <stdio.h>

#ifdef LINUX
#error this program only compiles in Linux
#endif

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

#pragma Komutu

#pragma komutu standart bir önışlemci komutudur. #pragma anahtar sözcüğünü başka bir komut sözcüğü izler. Bu komut sözcüğü standart değildir. Yani #pragma komutu standarttır ancak onun yanındaki komut sözcüğü derleyiciden derleyiciye değişebilmektedir. Her derleyicinin pragma komutları diğerinden farklı olabilir. Pragma komutları belli bir derleyiciye özgü işlemlerin yaptırılması için kullanılır. Örneğin:

```
#include <stdio.h>

#pragma pack(1)

struct SAMPLE {
    char a;
    int b;
    char c;
    int d;
};

#pragma pack()

int main(void)
{
    struct SAMPLE s;
```

```

printf("%u\n", sizeof(s));
return 0;
}

```

Burada pack pragma komutu her derleyicide olmak zorunda değildir. Microsoft derleyicilerinde ve gcc derleyicilerinde bu komut hizalama sağlamak için kullanılabilir.

Derleyicilerin pragma komutlarının listesi onların referans dokümanlarından öğrenilebilir.

(Token Pasting) Önışlemci Operatörü

Bu önışlemci komutu iki yazıyı birleştirmek için kullanılır. Örneğin:

```

#define Name(x, y)  x##y

struct Name(CSD, Sample) {
    ...
};

```

Önışlemci bu yapı bildirimini şöyle açar:

```

struct CSDSample {
    ...
};

```

Örneğin:

```

#include <stdio.h>

#define Name(x) CSD##x

int main(void)
{
    int Name(x), Name(y);

    CSDx = 10;
    CSDy = 20;

    printf("%d, %d\n", CSDx, CSDy);

    return 0;
}

```

C'nin Önceden Tanımlanmış Sembolik Sabitleri ve Makroları

C önışlemcisi bazı sembolik sabit ve makro isimlerini hiç define edilmediği halde define edilmiş gibi kabul etmektedir. Bunlara önceden tanımlanmış (predefined) sembolik sabitler ve makrolar denir. Bunların hepsinin başında ve sonun iki alt tire vardır.

__FILE__: Önışlemci bu sembolik sabiti gördüğünde bunun yerine iki tırnak içerisinde bunun bulunduğu kaynak dosyanın ismini yerleştirir. Örneğin:

```

#include <stdio.h>
#include <stdlib.h>

int foo(void)
{

```

```

    return 0;
}

int main(void)
{
    if (!foo()) {
        fprintf(stderr, "Error in file: %s\n", __FILE__);
        exit(EXIT_FAILURE);
    }

    return 0;
}

```

__LINE__: Önışlemci bu sembolik sabiti gördüğünde bu sembolik sabit hangi satıra yerleřtirilmiřse onun satır numarasını yazar (iki tırnak içerisinde deęil). Örneęin:

```

#include <stdio.h>
#include <stdlib.h>

int foo(void)
{
    return 0;
}

int main(void)
{
    if (!foo()) {
        fprintf(stderr, "Error in file: %s\n in line %d\n", __FILE__, __LINE__);
        exit(EXIT_FAILURE);
    }

    return 0;
}

```

__DATE__ ve **__TIME__**: Önışlemci bu makrolar yerine iki tırnak içerisinde derleme işleminin yapıldığı tarihi ve zamanı yerleřtirir.

__STDC__: Bu makro eęer standart C derleyicisinde çalışılıyorsa define edilmiş kabul edilir, deęilse define edilmemiş kabul edilir. Örneęin:

```

#ifdef __STDC__
error this program cannot compile with non standard C compiler
#endif

```

Yukarıdaki önceden tanımlanmış sembolik sabitlerin dışında derleyicilerin kendilerine özgü önceden tanımlanmış başka sembolik sabitleri ve makroları da vardır. Örneęin **_WIN32** ve **_WIN64** sembolik sabitleri Microsoft C derleyicilerine özgüdür:

```

#ifdef _WIN64
#error this program can only compile in Windows 64 platform
#endif

```

