C Programlama Dili İkinci Tiraj

Dennis M. Ritchie
Brian W. Kernighan
İngilizce Aslından Çeviren: Serhan Ekmekçi

İçindekiler

Önsöz

Bilgi işlem dünyası, 1978 yılında "C Programlama Dili"nin yayınlanmasından itibaren bir devrime uğramıştır. Büyük bilgisayarlar artık daha da büyükler ve kişisel bilgisayarlar artık on yıl öncesinin ana bilgisayarlarına rakip olabilecek kapasitedeler. Bu zaman içerisinde, mütevazi bir şekilde olsa da C de değişime uğradı ve kendi orijini olan sadece UNIX işletim sisteminin dili olmaktan çok uzaklara yayıldı.

C'nin gittikçe artan popülaritesi, yıllar içerisinde dilde gerçekleşen değişimler ve gruplar tarafından derleyicilerin (**compilers**) oluşturulması onun dizaynına bulaştırılmadı; dilin daha açık ve kitabın ilk tirajına göre daha çağdaş bir tarifi (**definition**) için kombine edildi. 1983'te Amerikan Ulusal Standartları Enstitüsü (**American National Standards Institue**) (ANSI) amacı, C dilinin, ruhunu kaybettirmeden kesin ve makineye dayalı olmayan bir tarifini (**definition**) üretmek olan bir komite kurmuştur. Sonuç, C için "ANSI" standardıdır.

Standart, ilk tirajda bahsi geçmiş fakat tam anlamıyla tarif edilmemiş (**define**) yapıları formüllendiriyor. Özellikle yapı ataması ve numaralandırma gibi konularda. Bu; fonksiyon deklarasyonunun çapraz-kontrole (**cross-checking**) izin veren, kullanımla beraber bir tarifini (**definition**) sağlar. Girdi (**input**) ve çıktıları (**output**) sergilemek için olan fonksiyonların geniş bir dizisi, hafıza yönetimi, dize manipülasyonu ve benzer görevlerle birlikte standart bir kütüphane belirtir. Orijinal tarifte (**definition**) ayrıntılı olarak açıklanmamış özelliklerin davranışlarını açık ve kesin bir hale getirir ve aynı zamanda dilin makineye dayalı kalan yönlerini açık bir şekilde belirtir.

C Programlama Dili'nin ikinci tirajı, C'yi ANSI standartlarında olduğu gibi açıklar. Biz, dilin evrildiği yerleri belirtmiş olmamıza rağmen bilerek yeni formda yazmayı tercih ettik. Büyük bir bölüm için bu, çok önemli bir fark yaratmaz. En gözle görülür değişim, fonksiyon deklarasyonunun ve tarifinin (**definition**) yeni formudur. Modern derleyiciler (**compilers**) standardın özelliklerinin çoğunu çoktan desteklemektedir.

Kitabı ilk tirajın kısalığında tutmaya çalıştık. C, büyük bir dil değil ve büyük bir kitap onun çok işine yaramaz. C programlamanın merkezi olan işa-

retleyiciler (**pointers**) gibi kritik özelliklerin anlatımını geliştirdik. Orijinal örnekler işledik ve birkaç bölüme yeni örnekler ekledik. Örneğin; komplike deklarasyonların işlemleri deklarasyonları kelimelere dönüştüren programlar ile karşılıklı olarak arttırıldı. Önceden olduğu gibi bütün örnekler doğrudan, makine tarafından okunulabilir formda olan metinden test edilmiştir.

Ek Bölüm A; referans el kitabı, standart değildir. Fakat bizim girişimimiz standardın gerekliliklerini daha küçük bir alanda iletmek. Bu programcılar tarafından kolay anlaşılmak üzere yapıldı ancak derleyici (**compiler**) yazarları için bir açıklama olarak yapılmadı. - Bu görev tamamen standardın kendisine düşüyor. Ek Bölüm B, standart kütüphanenin özetidir. Ve bu da dahil ediciler emplemantasyoncular için değil programcılar için referans edildi. Ek Bölüm C, orijinal versiyondaki değişimlerin kısa özetidir.

İlk tirajın önsözünde de söylediğimiz gibi C "sizlerden gelişen tecrübeyle hala iyi kalmayı başarıyor." On yılların getirdiği tecrübe ile birlikte biz hala bu şekilde hissediyoruz. Umuyoruz ki bu kitap, C'yi öğrenmenizde ve iyi bir şekilde kullanmanızda yardımcı olur.

Bu ikinci tirajı ortaya çıkarmamızda yardımcı olan arkadaşlarımıza derinden borçlu hissediyoruz. Jon Bentley, Doug Gwyn, Doug Mcllroy, Peter Nelson ve Rob Pike bize neredeyse taslağın her sayfasında sezgilerinden yola çıkarak yorumlarda bulundular. Dikkatli okumaları için Al Aho, Dennis Allison, Joe Campbell, G. R. Emlin, Karen Fortgang, Allen Holub, Andrew Hume, Dave Kristol, John Linderman, Dave Prosser, Gene Spafford, ve Chris Van Wyk'e minnettarız. Ayrıca BHl Cheswick, Mark Kernighan, Andy Koenig, Robin Lake, Tom London, Jim Reeds, Clovis Tondo, ve Peter Weinberger'den yardımcı tavsiyeler aldık. Dave Prosser ANSI standartları hakkında birçok detaylı sorumuzu cevapladı. Programlarımızın yerel (**local**) testleri için yaygın olarak Bjarne Stroustrup'ın C++ çeviricisini kullandık ve Dave Kristol bize final testi için bir ANSI C derleyicisi (**compiler**) sağladı. Rich Drechsler dizgilerde bize son derece yardımcı oldu.

Hepsine içten teşekkürlerimizle.

Brian W. Kernighan
Dennis M. Ritchie

İlk Tiraja Önsöz

C; açıklama (**expression**) idaresi, modern kontrol akışı (**control-flow**), veri yapıları (**structers**) ve zengin bir operatör grubu içeren özellikte, genel amaçlı bir programlama dilidir. C ne bir "çok yüksek seviyeli" ne de büyük bir dil değildir ve herhangi bir uygulamanın belirli bir bölümü için özelleştirilmemiştir. Fakat kısıtlamalarının olmaması ve yaygınlığı onu birçok görev için sözde daha güçlü dillerden daha elverişli ve etkili yapıyor.

C orijinal olarak "UNIX" işletim sisteminde, DEC PDP-11 üzerinde Dennis Ritchie tarafından tasarlanmış ve uygulanmıştır. İşletim sistemi, C derleyicisi (**compiler**) ve aslında tüm "UNIX" uygulama programları (bu kitabı hazırlamak için kullanılan tüm yazılımlar dahil) C dilinde yazılıyor. Üretim derleyicileri (**compilers**) ayrıca IMB Sistem/370, Honeywell 6000 ve Interdata 8/32'yi içeren diğer birkaç makine için de bulunmakta. C, belirli hiçbir donanım ve sisteme bağlı değil ve C destekleyen bir makineden diğerine herhangi bir değişim yapmadan çalışacak programlar yazmak oldukça kolay.

Bu kitap okurun C'de programlamanın nasıl olduğunu öğrenmesine yardım etmek içindir. Yeni kullanıcıların mümkün olan en kısa sürede başlaması için öğretici bir giriş, her majör özellik için ayrı bölüm ve referans el kitabı içerir. İşleyişin çoğu sadece kuralların ifade edilmesinden çok okumaya, yazmaya ve tekrar örneklerine dayalı. Çoğu kısımda örnekler ayrılmış parçalar olmaktan ziyade tamamlanmış gerçek programlar. Bütün örnekler doğrudan makine tarafından okunulabilir (**machine-readable**) formda olan metin üzerinden test edildiler. Biz, dilin etkili kullanımının nasıl olacağını göstermek dışında ayrıca mümkün olduğu yerlerde kullanışlı algoritmaları ve iyi bir tasarımın prensiplerini örneklemeyi denedik.

Bu kitap tanıtıcı bir programlama el kitabı değildir. Okuyucunun değişkenler (**variables**), atama ifadeleri (**statements**), döngüler (**loops**) ve fonksiyonlar gibi temel programlama konseptlerine biraz aşinalığı olduğunu varsayar. Yine de acemi bir programcının, daha bilgili bir iş ortağına erişmesi okuyabilmesi ve dili algılayabiliyor olmasına yardımcı olacaktır.

Bizim deneyimimizde C; kendini çok çeşitli programlar için keyifli, etkileyici ve çok yönlü bir dil olarak kanıtlamıştır. Öğrenmesi kolaydır ve sizlerden gelişen tecrübeyle hala iyi kalmayı başarıyor. Umuyoruz ki bu kitap onu iyi bir şekilde kullanmanızda yardımcı olacaktır.

Birçok arkadaş ve iş ortağımızın düşünceli eleştiri ve önerilerini büyük ölçüde bu kitaba ve onu yazma zevkimize kattık. Özellikle Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy, Bill Roome, Bob Rosin, ve Larry Rosie çoklu versiyonların hepsini dikkatle okudular. Ayrıca çeşitli evrelerdeki yardımcı yorumları için Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill PIauger, Jerry Spivack, Ken Thompson, ve Peter Weinberger'e ve yazım düzenlemesindeki sağlam yardımları için Mike Lesk ve Joe Ossanna'ya minnettarız.

Brian W. Kernighan
Dennis M. Ritchie

Giriş

C, genel amaçlı bir programlama dilidir. C, UNIX sisteminin kendisi olduğu gibi üzerinde çalışan programların da büyük çoğunluğunun C dilinde yazılmış olmasından dolayı, geliştirildiği yer olan UNIX sistemi ile yakından ilişkilidir. Ancak dil herhangi bir işletim sistemi ve makineye bağlı değildir fakat derleyiciler (**compilers**) ve işletim sistemlerini yazmakta kullanışlı olduğu için "sistem programlama dili" olarak adlandırılmış olmasına rağmen aynı ölçüde birçok farklı alanda majör programlar yazmak için de kullanılmıştır.

BCPL dilinden kaynaklanan birçok C fikirlerinin çoğu Martin Richards tarafından geliştirildi. BCPL'in C'nin üzerindeki etkisi dolaylı yoldan Ken Thompson tarafından 1970'te DEC PDP-7'deki ilk "UNIX" sistemi için yazılmış olan B dili aracılığıyla geldi.

BCPL ve B "tipsiz" (**typeless**) dillerdir. Onların aksine C, bir çok çeşit veri tipi (**type**) sağlar. Temel tipler, karakterler, tam sayılar (**integers**), ve bazı boyuttaki ondalıklı (**floating-points**) sayılar. Ek olarak; türetilmiş veri türlerinin (**derived data types**) işaretçiler (**pointers**), diziler (**arrays**), yapılar (**structures**) ve birlikler (**unions**) ile birlikte yarattığı bir hiyerarşi vardır. Açıklamalar (**expressions**), operatörler (**operators**) ve işlenenlerden (**operands**) oluşur. Bir atama (**assignment**) veya bir fonksiyon çağrısı içeren her açıklama (**expression**), bir ifade(**statement**) olabilir. İşaretleyiciler (**pointers**) makineden bağımsız (**machine-independent**) adres aritmetiği için sağlanıyor.

C, iyi yapılandırılmış programlar için gerekli olan temel kontrol akışı (**control-flow**) yapılarını sağlar: ifade (**statement**) gruplaması, karar verme (if-else), mümkün olan durumlar arasından birini seçme (switch), sonucunda döngünün bitirilip bitirilmeyeceğine karar verilen koşul testinin döngünün (**loop**) başında yapıldığı döngüler (**loop**) (while, for) veya bu koşul testinin sonunda yapılmasını sağlayan (do) ve erken döngü (**loop**) bitirmeye yarayan (break).

Fonksiyonlar , temel tiplerin (**type**), yapıların (**structures**), birliklerin (**unions**) veya işaretleyicilerin (**pointers**) değerini döndürebilir.

Her fonksiyon birden fazla kere çağırılabilir. Yerel değişkenler (**variables**) tipik olarak "otomatiktirler", ya da her çağrı ile yeniden yaratılırlar diyebiliriz. Fonksiyonlar iç içe tarif edilemez (**define**) fakat değişkenler (**variables**) blok yapı (**block-structured**) tarzında deklare edilebilir. C programının fonksiyonları ayrı olarak derlenen ayrı kaynak dosyaları içinde bulunabilir. Değişkenler; (**variables**) bir fonksiyon için dahili, harici fakat sadece tek bir kaynak dosyası içinde bilinen ya da tüm program için görülür olabilirler.

Önişleme adımı program metninde; yerine koyma makrosunu, diğer kaynak dosyalarının dahil edilmesini ve koşulsal derleme (**compilation**) işlemlerini gerçekleştirir.

C, nispeten "düşük seviye" bir dil. Bu nitelendirme aşağılayıcı değil, sadece C'nin bilgisayarların çoğunun yaptığı nesnelerle uğraştığı anlamına geliyor, karakterler, sayılar ve adresler. Bunlar gerçek makineler tarafından uygulanan aritmetik ve mantıksal operatörler (**operators**) tarafından birleştirilebilir ve hareket ettirilebilirler. C, karakter öbekleri (**strings**), kümeler (**sets**), listeler (**lists**) ve diziler gibi bileşik objelerle uğraşmak için işlemler sağlamaz. Yapıların (**structures**) bir birim olarak kopyalanabilmesine rağmen karakter öbeklerinin (**string**) ve dizilerin (**arrays**) tamamını değiştirebilecek işlemler yoktur. C, statik tarif etme (**definition**) ve fonksiyonların yerel (**local**) değişkenleri (**variables**) tarafından sağlanan yığın disiplini (**stack discipline**) dışında hiçbir depolama tahsis olanağı tarif etmeyen (**define**) bir dildir. Yığın ağacı (**heap**) veya çöp koleksiyonu (**garbage collection**) yoktur. Son olarak C'nin kendisi girdi/çıktı (**input/output**) olanağı sağlamaz, **READ** veya **WRITE** ifadeleri (**statements**) ve yerleşik dosya erişim metotları yoktur. Bütün bu yüksek seviye mekanizmaların ayrıca çağırılan fonksiyonlar tarafından sağlanması gerekir. Birçok C emplementasyonu, bu fonksiyonların uygun bir standart koleksiyonunu içerir.

Benzer şekilde C sadece basit, tek iş parçacıklı (**single-thread**) kontrol akışı (**control-flow**) sunar: testler, döngüler (**loops**), gruplama ve alt programlar (**subprograms**). Fakat multi programlama (**multiprogramming**), paralel işlemler, senkronizasyon ya da eşyordamlar (**coroutines**) sunmaz. Bu özelliklerin bazılarının yokluğu ölü gereksinim gibi gözükse de ("İki karakter dizisini karşılaştırmam için bir fonksiyon çağırmam gerektiğini mi söylemek istedin?") dili mütevazi boyutlarda tutmanın gerçek faydaları vardır. C nispeten küçük bir dil olduğundan dolayı, küçük bir alanda açıklanabiliyor ve hızlıca öğrenilebiliyor. Bir programcı mantıklı olarak, tüm dili bilmeyi, anlamayı ve gerçekten muntazam olarak kullanmayı bekleyebilir.

Yıllarca C'nin tarifi, C Programlama Dili'nin ilk tirajındaki referans el kitabı oldu. 1983'te, Amerikan Standartları Enstitüsü (**American National Standards Institute**) (**ANSI**) C'nin modern ve kapsamlı bir tarifini sağlamak için bir komite kurdu. Ortaya çıkan tarif, "ANSI standardı" ya da "ANSI C" 1988'in sonlarına doğru tamamlandı. Standardın birçok özelliği çoktan modern derleyiciler (**compilers**) tarafından desteklenmiştir.

Standart, orijinal referans el kitabını baz almıştır. Dil nispeten biraz değişti; standartın amaçlarından biri hali hazırda varolan programların çoğunun geçerli kalması veya bu mümkün olmazsa derleyicilerin (**compilers**) yeni davranışlar için uyarılar üretmesiydi.

Birçok programcı için en önemli değişim; fonksiyonları deklare etmek veya tarif etmek (**define**) için olan yeni sözdizimi (**syntax**). Bir fonksiyon deklarasyonu (**decleration**) şimdi fonksiyonun argümanlarının açıklamasını içerebilir; tarif etme (**definition**) sözdizimi (**syntax**) eşleşmek üzere değişir. Bu ekstra bilgilendirme derleyicilerin (**compilers**) eşleşmeyen argümanlar yüzünden ortaya çıkan hataları farketmesini daha kolay hale getirir. Bizim deneyimimizde, dile yapılan son derece kullanışlı bir ekleme.

Daha küçük çaplı dil değişimleri var. Geniş ölçüde mevcut olan yapı atamaları (**structure assignment**) ve numaralandırmalar (**enumerations**) şimdi resmi olarak dilin bir parçası. Kayan noktalı sayıların (**floating-point numbers**) hesaplamaları şimdi tek duyarlıklı (**single precision**) formatta yapılabilir. Aritmetiğin özellikleri, özellikle belirlenmemiş tipler (**unsigned types**) için netleştirildi. Önişlemci (**preprocessor**) daha ayrıntılı. Bu değişimlerin çoğu, birçok programcı üzerinde sadece minör etkiye sahip olacaktır.

Standardın ikinci en önemli katkısı, C'ye eşlik etmek üzere bir kütüphanenin tarifidir (**definition**). Bu; işletim sistemine erişmek için fonksiyonlar (örneğin dosyaları okumak ve yazmak için), biçimlendirilmiş girdi (**input**) ve çıktı (**output**), hafıza tahsisi (**memory allocation**), karakter öbeği (**string**) manipülasyonu vb. şeyler belirtir. Standart başlıkların (**header**) koleksiyonu, fonksiyonların ve veri tiplerinin (**data types** deklarasyonlarına tek tip erişim sağlar. Bu kütüphaneyi programların konakçı (**host**) bilgisayar ile etkileşim için kullanması, uyumlu (compatible) davranışlar sergilemelerini güvence altına almıştır. Kütüphanenin büyük bir kısmı "standart I/O kütüphanesinde" (input/output girdi/çıktı) dikkatle modellenmiştir. Bu kütüphane ilk tirajda açıklanıyor ve diğer sistemlerde de geniş ölçüde kullanılmıştır. Tekrar söylüyoruz, birçok programcı çok fazla değişim görmeyecek.

C tarafından sağlanan veri tipleri ve kontrol yapılarının (**structures**) birçok bilgisayar tarafından doğrudan destekleniyor olması yüzünden kendine yeten programları emplement etmek için gerekli olan çalışma kütüphanesi (**run-time library**) oldukça küçük.

Standart kütüphane (**library**) fonksiyonları tekli olarak çağrılıyor. Bu yüzden gerekli olmadıkları durumlarda kaçınılabiliyorlar. Çoğu, C dilinde yazılabilir ve içerdikleri işletim sistemi detayları hariç kendileri taşınabilirler (**portable**).

C, birçok bilgisayarın kabiliyetleri ile uyuşsa da belirli herhangi bir makine mimarisinden bağımsızdır. Biraz özenle, taşınılabilir (**portable**) programlar yazmak kolaydır. Yani değişken donanımlar üzerinde bir değişim olmadan çalışabilecek programlar. Standart, taşınılabilirlik (**portability**) sorunlarını açık hale getirmekte ve programın üzerinde çalıştırıldığı makineyi karakterize eden birtakım sabitler (**constants**) kümesini reçete etmekte.

C, kuvvetle-yazılmış (**strongly-typed**) bir dil değil, fakat geliştikçe, tip kontrolü (**type-checking**) de güçlendirildi . C'nin orijinal tarifi, işaretleyiciler (**pointers**) ve tam sayıların (**integers**) değişimini hoş karşılamadı fakat buna izin verdi. Bu ortadan kaldırılalı uzun zaman oldu ve şimdi standart, iyi derleyiciler (**compilers**) tarafından çoktan zorunlu olan uygun deklarasyonlar ve açık dönüşümler gerektiriyor. Yeni fonksiyon deklarasyonları bu yönde bir başka adım. Derleyiciler (**compilers**) birçok tip (**type**) hatasını bildirir ve uyumsuz veri tiplerinin (**data types**) otomatik dönüşümleri yoktur. Yine de C, programcıların ne yaptığını bildiği basit felsefeyi korur, sadece onların amaçlarını açıkça bildirmelerini gerektirir.

C'nin de başka her dilin olduğu gibi kendi kusurları var. Operatörlerin (**operators**) bazıları yanlış önceliklere sahip. Söz diziminin (**syntax**) bazı kısımları daha iyi olabilir. Yine de C, programlama uygulamalarının geniş çeşitliliği için son derece etkili ve anlamlı bir dil olarak kendini kanıtlamıştır.

Bu kitap şu şekilde organize edilmiştir: Bölüm 1, C'nin merkezi kısmında öğreticidir. Biz; yeni bir dil öğrenmenin o dilde programlar yazmak olduğuna inandığımızdan dolayı amaç, mümkün olduğunca çabuk başlayan okuyucular edinmek. Öğretici kısım; programlamanın temellerinin okuyucuda bulunduğunu farz eder: Ne bilgisayarların, ne derlemenin ne de "n=n+1" gibi bir ifadenin anlamının açıklaması var. Biz mümkün olan yerlerde kullanışlı programlama tekniklerini göstermeyi denedik fakat bu kitap veri yapıları (**structures**) ve algoritmalar üzerinde referans çalışması olarak tasarlanmamıştır. Bir seçim yapmak zorunda kaldığımızda, dil üzerinde yoğunlaştık.

Üzerinde durulan nokta hala yalıtılmış kısımlardan ziyade tamamlanmış programların örnekleri olmasına rağmen Bölüm 2, Bölüm 6 ile beraber C'nin çeşitli yönlerini Bölüm 1'de olduğundan daha detaylı ve daha biçimsel olarak tartışıyor. Bölüm 2, temel veri tipleri (**data types**), operatörler (**operators**) ve ifadeler (**statements**) ile ilgilenir.

Bölüm 3, kontrol akışını (**control-flow**) ele alır: if-else, switch, while, for vb. Bölüm 4, fonksiyonlar ve program yapılarını (**structures**) -harici değişkenler (**variables**), kapsam kuralları, çoklu kaynak dosyaları ve bunun gibikapsar ve ayrıca önişlemciye (**preprocessor**) değinir. Bölüm 5 işaretçileri (**pointers**) ve adres aritmetiğini tartışır. Bölüm 6 yapıları (**structures**) ve birlikleri (**unions**) kapsar.

Bölüm 7, işletim sistemine uygun arayüz sağlayan standart kütüphaneyi açıklar. Bu kütüphane "ANSI" standardı tarafından tarif edilmiştir (**define**) ve C'yi destekleyen tüm makinelerde desteklenmesi için yapıldı. Bu yüzden onu girdi (**input**), çıktı (**output**) ve diğer işletim sistemi erişimi için kullanan programlar, bir sistemden diğerine değiştirilmeden taşınabilir (**prtable**).

Bölüm 8, C programları ve UNIX işletim sistemi, girdi (**input**) ve çıktıya (**output**) odaklanma, dosya sistemi ve depolama tahsisi (**storage allocation**) arasındaki arayüzü açıklar. Bu bölümün bir kısmı özellikle UNIX sistemi için olsa da diğer sistemleri kullanan programcılar da burada standart kütüphanenin bir versiyonunun nasıl uygulandığını kavrama ve taşınılabilirlik önerileri de dahil olmak üzere kullanışlı materyaller bulabilirler.

Ek Bölüm A, bir dil referans el kitabı içerir. Sözdiziminin (**syntax**) ve C'nin anlambiliminin (**semantics**) resmi ifadeleri (**statements**) ANSI standardıdın kendisidir. Ancak bu belge başta derleyici (**compiler**) yazıcılara yöneliktir. Buradaki referans el kitabı dilin tarifini daha kısa ve aynı titiz, katı stil olmadan iletir. Ek Bölüm B, emplamantasyonculardan çok kullanıcılar için standart kütüphanenin bir tarifi (**definition**). Ek Bölüm C, orijinal dildeki değişimlerin kısa bir özeti. Ancak şüpheye düşüldüğünde standartın ve onun kendi derleyicisi (**compiler**), dilin son otoriteleri olarak kaldılar.

Bölüm 1 | Öğretici Kısıma Giriş

C'ye hızlı bir girişle başlayalım. Ana odağımız dilin gerekli elemanlarını detaylara, ayrıntılara, kurallara ve istisnalara girmeden gerçek programlar aracılığıyla göstermek. Bu aşamada tamamlayıcı ve kusursuz olmaya çalışmıyoruz. Sizi, olabilecek en kısa sürede işe yarar programlar yazabilecek hale getirmeye çalışıyoruz. Ve bunu yapabilmek için temellere (değişkenler (**variables**), sabitler (**constants**), aritmetikler, kontrol akışı (**control flow**), fonksiyonlar , girdi (**input**) ve çıktıların (**output**) esasları) odaklanmamız gerekmekte. Bilerek bu bölümde C'nin büyük programlar yazmak için önemli olan özelliklerine değinmiyoruz. Bu işaretleyiciler (**pointers**), yapılar (**structers**), C'nin zengin operatör setinin çoğunu, bazı kontrol akışı (**control-flow**) ifadelerini (**statement**) ve temel C standart kütüphanesini (**library**) içeriyor.

Bu amacın kendine has sakıncaları bulunmakta. En öne çıkanı ise belirli bir dilin baştan sona hikayesinin burada bulunmaması. Aynı zamanda, öğretici kısım özetleyici olduğu gibi yanıltıcı da olabilir. Ve örnekler öğretim amacıyla C'nin tüm gücünden faydalanmadığından, olabilecek en öz ve zarif şekilde değiller. Bu etkileri hafifletebileceğimiz kadar hafiflettik fakat dikkatli olmanız gerekmekte. Bir diğer sakınca ise bundan sonraki bölümlerin bu bölümün bir kısmının tekrarını içeriyor olması. Umuyoruz ki bu tekrarlama size rahatsızlıktan çok yardım sağlar.

Herhangi bir durumda, deneyimli programcılar kendi programcılık ihtiyaçları doğrultusunda verilen materyalden faydalanacaklardır. Yeni başlayanlar ise bunu kendi küçük benzer programlarını yazarak sağlayabilirler. İki grup da bunu çatı (**framework**) olarak Bölüm 2'de başlayan daha detaylı açıklamaların üstüne kullanabilirler.

1.1 Başlamak

Yeni bir programlama dili öğrenmenin tek yolu o dille programlar yazmaktır. Yazılacak ilk program tüm diller için aynıdır.

```
Kelimeleri Bastır
   hello, world
```

Bu büyük bir engel; aşılması için programı metin olarak bir yerde oluşturmak, ardından başarılı bir şekilde derlemek, yüklemek, çalıştırmak, ve çıktının (**output**) nereye gittiğini bilmek gerekiyor. Bu mekaniksel detaylarda ustalaştıktan sonra, geri kalan her şey görece daha kolay.

C'de "hello, world" bastırma programı

```
#include <stdio.h>

main()
{
   printf("hello, world\n");
}
```

Bu programı çalıştırmak kullandığınız sisteme göre değişiklik gösterir. Spesifik bir örnek vermek gerekirse, UNIX işletim sisteminde programı hello.c gibi ".c" uzantılı bir dosyanın içine kaydedip, ardından bu komut ile derlemeniz gerekir.

```
cc hello.c
```

Eğer bir karakter yazmayı unutmak veya bir şeyi yanlış yazmak gibi bir hata yapmadıysanız, derleme sorunsuz ve sessiz bir şekilde gerçekleşmeli, ve a.out isimli bir çalıştırılabilir (**executable**) dosya oluşmalı. Eğer bu komut ile a.out dosyasını çalıştırırsanız

```
a.out
```

bunu bastıracaktır

```
hello, world
```

Diğer sistemlerde kurallar farklı olacaktır, sisteminizle ilgili yerel bir uzmana danışın. Şimdi programın kendisi hakkında biraz açıklama. Bir C programı, boyutu ne olursa olsun, fonksiyonlardan ve değişkenlerden(**variables**) oluşur. Bir fonksiyon , yapılacak bilgisayarsal operasyonları belirten ifadeler (**statements**) ve çalışma sırasında kullanılacak değerleri (**values**) taşıyan değişkenler (**variables**) içerir. C fonksiyonları daha çok altprogramlara (**subroutines**) ya da Fortran dilini bilenlerin aşina olduğu Fortran fonksiyonlarına , Pascal prosedürlerine (**procedures**) ve Pascal fonksiyonlarına

oldukça benzer. Bizim örneğimiz **main** isimli fonksiyon . Normalde fonksiyonlara istediğiniz ismi vermekte özgürsünüz fakat "**main**" için değil. Programınız **main** fonksiyonunu baştan aşağıya doğru çalıştırarak işe başlar. Bu, her programın herhangi bir yerinde bir **main** fonksiyonu olmasını gerektirir. **main** genellikle başka fonksiyonları gerçekleştireceği işe yardım için çağırır, bazen sizin yazdıklarınız, bazen de kütüphanelerde (**libraries**) sizin için sağlanan fonksiyonları. Programınızın ilk satırı,

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

derleyiciye (**compiler**) standart girdi (**input**) çıktı (**output**) hakkında bilgilerin programa dahil edilmesini söyler. Bu satır hemen hemen her C kaynak (**source**) dosyasında bulunur. Standart kütüphane (**library**) Bölüm 7'de ve Ek B'de açıklanmıştır. Fonksiyonlar arasında veri aktarımı için gerekli metot, çağırılan fonksiyonlara argümanlar adı verilen değer veya değerler vermektir. Fonksiyon adından sonraki parantezlerin arasında argüman veya argümanlar bulunur.

```
#include <stdio.h>      standart kütüphane (library) hakkında
                    bilgileri içer
main()                  main adında hiç argümanı
                        olmayan bir fonksiyon deklare et

{
                        main'in ifadeleri (statements) süslü
                        parantezler arasındadır

   printf("hello, world\n"); main bir kütüphane (library)
                         fonksiyonu olan printf
                         fonksiyonunu parantezlerin
                         arasında verilen argümanı ekrana
                         bastırması için çağırıyor; \n yeni
                            bir
}                        satırı temsil eder.
```

İlk C programı.

Bu örnekte **main** fonksiyonu hiçbir argüman beklemeyen bir fonksiyon olarak deklare edildi, bu parantezler arasında hiçbir veri olmamasıyla gösterilir (). Fonksiyonun ifadeleri (**statements**) süslü parantezler içerisinde yer alır **{}**. **main** fonksiyonu sadece bir ifade (**statement**) içeriyor,

```
printf("hello, world\n");
```

C derleyicisi (**compiler**) hata mesajı üretecektir. **printf** asla yeni bir satırı otomatik olarak sağlamayacaktır, yani bu sayede tek bir çıktıyı (**output**) aralıksız bir biçimde printf fonksiyonunu birden fazla kere çağırarak bastırabiliriz ve bu ilk programımız gibi çalışır.

```
#include <stdio.h>

main()
{
   printf("hello, ");
   printf("world");
   printf("\n");
}
```

Bu şekilde aynı çıktıyı (**output**) bastırabiliriz. \n'nin sadece bir karakteri temsil ettiğini farkedin. \n gibi bir kaçış dizesi (**escape sequence**) bize yazılması zor veya görünmez karakteri belirtmek için genel ve genişletilebilir bir mekanik sağlıyor. C'nin diğer sağladığı kaçış dizeleri: \t tab karakteri için, \b geri tuşu (**backspace**) için, \" çift tırnak için, ve \\ ters eğik çizginin (**backslash**) kendisi için. Bunların listesinin tamamını Bölüm 2.3'te bulabilirsiniz.

Egzersiz 1-1. "hello, world" programını sisteminizde çalıştırın. Programın belli başlı parçalarını çıkartın ve alacağınız uyarı mesajlarını ve çıktıları (**outputs**) görün.

Egzersiz 1-2. printf'in argümanı olan karakter öbeği (**string**) burada listelenmemiş olan \c karakterini içerdiğinde ne olacağını görün.

1.2 Değişkenler ve Aritmetik Açıklamalar

Sonraki program aşağıdaki Fahrenhayt dereceleri ve onların Selsiyus (Santigrat) eşitliklerini gösteren tabloyu bastırmak için $^\circ C = (5/9)(^\circ F\text{-}32)$ formülünü kullanıyor.

```
0  -17
20 -6
40 4
60 15
80 26
100 37
120 48
140 60
160 71
180 82
200 93
220 104
240 115
260 126
280 137
300 148
```

Programın kendisi hâlen sadece **main** adında tek bir fonksiyon içeriyor. "hello, world" bastıran programdan biraz uzun ancak korkacak bir şey yok, komplike değil. Bu program bize, yorumlar (**comments**), deklarasyonlar, değişkenler (**variables**), aritmetik açıklamalar, döngüler (**loops**) ve formatlanmış çıktılar (**output**) gibi birkaç yeni fikri gösteriyor.

```
#include <stdio.h>

/* Fahrenhayt-Selsiyus tablosunu bastır

   fahr = 0, 20, ... 300 için */

main()
{
   int fahr, celsius;
   int lower, upper, step;

   lower = 0; /* ısı tablosunun en düşük değeri */
   upper = 300; /* ısı tablosunun en yüksek değeri */
   step = 20; /* adım değeri */

   fahr = lower;

   while (fahr <= upper) {
      celsius = 5 * (fahr-32) / 9;
      printf("%d\t%d\n", fahr, celsius);
      fahr = fahr + step;
   }
}
```

Bu iki satır

```
/* Fahrenhayt-Selsiyus tablosunu bastır

   fahr = 0, 20, ... 300 için */
```

bu durumda programın nasıl çalıştığını, programın kaynak (**source**) kodunu okuyanlara açıklayan *yorum* (**comment**) satırlarıdır. /* ve */ karakterleri arasında olan karakterler derleyici (**compiler**) tarafından görmezden gelinir, bu satırlar programı okurken anlamayı daha kolay hale getirmek için özgürce kullanılabilir. Yorum satırları boşluk, tab veya yeni satır karakterinin koyulabileceği bütün yerlere koyulabilirler.

C'de; bütün değişkenler (**variables**) kullanılmadan önce deklare edildi. Genellikle fonksiyonun başında bütün çalıştırılabilir ifadelerden (**executable statements**) önce deklare edilmeli. Bir deklarasyon değişkenlerin (**variables**) özelliklerini anons eder; bu anons değişkenlerin (**variables**) deklare edilecek olan tipini (**type**) ve değişkenlere (**variable**) verilecek adları içerir, örneğin

```
int fahr, celsius;
int lower, upper, step;
```

tip (**type**) **int** kendisinin ardından listelenen değişkenlerin (**variable**) birer tamsayı (**integer**) olduğunu belirtir. Kayar noktalı yani ondalıklı sayılar (**float**) ile karşılaştırırsak, **int** ve **float**'un da sahip olduğu aralık, kullandığınız makineye dayalıdır; 16-bit **int'ler**, -32768 ve +32767 arasındadır ve 16-bit **int'ler** de 32-bit **int'ler** kadar yaygındır, çoğunlukla **float** sayı 32-bit niceliğindedir, bu en az 6 farklı basamağa ve 10^{-38} ve 10^{+38} arasında bir büyüklüğe tekabül eder.

C **int** ve **float**'un yanında birkaç başka temel veri tipi (**type**) daha sağlıyor:

```
char  karakter - tek byte
short kısa tamsayı (integer)
long  uzun tamsayı (integer)
double 64-bit (double precision) ondalıklı (float)
```

Bu objelerin büyüklüğü de makineye dayalı (**machine-dependent**). Bunların yanında kurs boyunca tanışacağımız dizeler (**arrays**), yapılar (**structers**) ve birlikler (**unions**) gibi basit tip (**types**); onları işaretlemek (**pointing**) için işaretleyiciler (**pointers**) ve onları döndürmek (**return**) için fonksiyonlar vardır.

Isı birimi dönüşümü programında hesaplama atama (**assignment**) ifadeleriyle (**assignment statements**) başlıyor.

```
lower = 0;
upper = 300;
step = 20;
fahr = lower;
```

Bu ifadeler (**statement**) ile değişkenler (**variables**) başlangıç değerlerine ayarlıyor. Tekli (toplu değil, bireysel) ayarlanmak istenen değişkenler (**variables**) noktalı virgül ile bitiriliyor.

Tablonun her satırı aynı şekilde hesaplanıyor, bu yüzden her çıktı (**output**) satırı için bunu tekrarlayabiliriz; işte **while** döngüsünün (**loop**) amacı budur.

```
while (fahr <= upper) {
   ...
}
```

while döngüsü (**loop**) şöyle işliyor: Parantezler arasındaki koşul test edilir, eğer koşul doğru ise (**fahr** küçüktür veya eşittir **upper**'a), döngünün (**loop**) gövdesi (süslü parantezler arasındaki bölüm) çalıştırılır. Ardından koşul tekrardan test edilir ve eğer doğru ise, gövde tekrardan çalıştırılır. Koşul testi sırasında yanlış sonuç ortaya çıkarsa (**fahr** büyüktür **upper**'dan) gövde çalıştırılmaz ve döngü (**loop**) sona erer. Ve programın çalışması döngüden (**loop**) sonrası için olan ifadelerle(**statement**) devam eder. Bu programda döngüden sonra bir ifade (**statement**) yoktu ve bu yüzden program sona erdi.

while'ın gövdesi süslü parantezler arasında bir veya birden çok ifade (**statement**) içerebilir veya süslü parantezler olmadan tek bir ifade (**statement**) içerebilir, örneğin;

```
while (i < j)
   i = 2 * i;
```

Her koşulda, her zaman ifadeleri (**statements**) bir tab (4 boşluk) ile girintileyeceğiz. Bu şekilde sizler göz gezdirirken hangi ifadelerin (**statement**) döngünün (**loop**) içinde olduğunu rahatlıkla görebileceksiniz. Girintileme programın mantıksal yapısını öne çıkartır. Buna karşın C derleyicisi (**compiler**) programın nasıl gözüktüğünü umursamaz, düzgün girintileme ve boşluklama sadece programların insanlar için daha okunabilir olması içindir. Buna karşın insanların girintileme için bazı tutkulu düşünceleri var. Biz popüler tarzlar arasından birini seçtik. Size uygun olanı seçmekten çekinmeyin ve sürekli kullanın.

İşin çoğu döngünün (**loop**) gövde kısmında gerçekleşiyor. Celcius ısı birimi hesaplanıyor ve **celcius** adında bir değişkene (**variable**) ifade (**statement**) tarafından atanıyor.

```
    celsius = 5 * (fahr-32) / 9;
```

Sadece 5/9 ile çarpmak yerine 5 ile çarpılıp ardından 9'a bölünmesinin sebebi C'de ve diğer bir çok dilde, tam sayıların (**integer**) bölümlerinin kesirli (ondalıklı) kısımlarının budanması. 5 ve 9 tamsayı (**integer**) olduğundan, 5/9'un sonucunun kesirli kısmı budanırdı, geriye sadece 0 kalırdı ve tüm Selsiyus ısıları 0 olarak raporlanırdı.

Bu örnek **printf**'in nasıl çalıştığını biraz daha gösteriyor. **printf** Bölüm 7'de detaylı açıklayacağımız genel amaçlı çıktı (**output**) formatlama fonksiyonudur . İlk argümanı, bastırılacak karakter öbeğidir (**string**), her % karakterinin yeri, fonksiyona girilen bir diğer argüman tarafından yeri doldurulur; (ikinci, üçüncü, ...) argümanlar % karakterlerinin yerine geçerler. Örneğin, **%d** bir tamsayı (**integer**) argümanı belirtir. Böylelikle ifade

```
    printf("%d\t%d\n", fahr, celsius);
```

iki tam sayının (**integer**) değerlerin (**values**) yani **fahr** ve **celcius**'un aralarında tab(\t) karakteri ile bastırılmasını sağlar.

Her % karakteri **printf**'in ilk argümanından sonraki uyan ilk argümanıyla eşleşir. Argümanlar ve % karakterleri birbiriyle tip (**type**) olarak ve sıralama olarak eşleşmeli, yoksa yanlış çıktı alırsınız.

Bu arada **printf** C dilinin bir parçası değil: C'nin kendisinde girdi (**input**) ve çıktı (**output**) tarif edilmemiştir (**define**). **printf** sadece standart girdi (**input**) çıktı (**output**) kütüphanesinde (**library**) bulunan yararlı bir fonksiyon . **printf**'in davranışı ANSI standartında belirlenmiştir, böylelikle özellikleri her derleyiciyle (**compiler**) ve kütüphaneyle (**library**) standarta uyar.

C'nin kendisine odaklanmak için, girdi (**input**) ve çıktı (**output**) hakkında Bölüm 7'den önce çok konuşmayacağız. Özellikle, o zamana dek formatlanmış girdiye (**input**) değinmeyeceğiz. Eğer numaraları girdilemeniz (**input**) gerekiyorsa, **scanf** fonksiyonu üzerine olan Bölüm 7.4'ü okuyunuz. **scanf**, **printf** gibi, sadece çıktı yazmak yerine girdi okuyor.

Isı birimi dönüştürme programında birtakım problemler var. Bu problemlerden daha basit olanı çıktının (**output**) pek güzel olmaması, çünkü sayılar tam yerinde değil. Bunu düzeltmesi kolay; her % karakterini **printf** ifadesinde (**statement**) bir genişlik ile yazarsak, sayılar alanlarında olması gereken yerlerinde olacaktır. Örneğin

```
printf("%3d %6d\n", fahr, celsius);
```

ilk sayıyı 3 basamak genişliğinde, ikinci sayıyı da 6 basamak genişliğinde üstteki gibi bastırırsak:

```
  0     -17
 20      -6
 40       4
 60      15
 80      26
100      37
120      48
140      60
160      71
180      82
200      93
220     104
240     115
260     126
280     137
300     148
```

Daha ciddi olan problem ise tam sayı (**integer**) aritmetiği kullandığımızdan dolayı Selsiyus ısıları tam olarak doğru değil; örneğin, 0°F aslında -17.8°C, -17 değil. Daha doğru çıktıları (**output**) almamız için tamsayı (**integer**) yerine ondalıklı (**float**) aritmetik kullanmalıyız. Bu, programda bazı değişiklikleri gerektiriyor. İşte programın ikinci versiyonu:

```
#include <stdio.h>

/* Fahrenhayt-Selsiyus tablosunu bastır

   for fahr = 0, 20, ... 300 */

main()
{
   float fahr, celsius;
   int lower, upper, step;

   lower = 0; /* ısı tablosunun en düşük değeri */
   upper = 300; /* ısı tablosunun en yüksek değeri */
   step = 20; /* adım değeri */

   fahr = lower;

   while (fahr <= upper) {
      celsius = (5.0/9.0) * (fahr-32.0);
      printf("%3.0f\t%6.1f\n", fahr, celsius);
      fahr = fahr + step;
   }
}
```

Bu öncekine çok benziyor, tek farkı **fahr**'ın ve **celsius**'un ondalıklı sayı (**float**) olarak deklare edilmiş olması ve dönüşüm için kullanılan formülün daha doğal şekliyle yazılmış olması. 5/9'u önceki versiyonda kullanamıyorduk çünkü iki tam sayının (**integer**) bölümü ondalıklı kısmını budayarak sonucun 0 çıkmasını sağlıyordu. Bir sabitteki (**constant**) ondalık noktası o sabitin tipinin (**type**) ondalıklı sayı (**float**) olduğunu belirtir, yani 5.0/9.0 budanmadı çünkü işlem iki ondalıklı sayı (**float**) ile yapıldı.

Eğer aritmetik operatörün tam sayı (**integer**) işlenenleri (**operands**) olursa tam sayılarla (**integers**) gerçekleşen bir işlem yapılırdı. Eğer aritmetik operatörün bir ondalıklı sayı (**float**) ve bir tam sayı (**integer**) işleneni (**operand**) var ise bir şey farketmez ve tam sayı (**integer**) işlemden önce ondalıklı sayıya (**float**) dönüştürülür ve ardından hesaplama yapılırdı. Eğer **fahr-32** yazmış olsaydık, **32** otomatikman ondalıklı sayıya (**float**) dönüştürülürdü. Yine de ondalıklı sayı (**float**) olan sabitleri tam (**integral**) olmalarına rağmen ondalık noktasını kullanarak yazmamız, onların ondalıklı sayı (**float**) doğasını insan okuyucular için vurgular. Tam sayıların (**integer**) ondalıklı sayılara (**float**) dönüşümlerinin detaylarını Bölüm 2'de göreceğiz.

Şuanlık şu atamayı (**assignment**) farkedin

```
    fahr = lower;
```

ve test edin.

```
    while (fahr <= upper)
```

İkisi de çalışıyor, **int** operasyondan önce **float**'a dönüştürülüyor ve ardından hesaplama yapılıyor.

printf dönüşümü belirteçi **%3.0f** ondalıklı sayının (**floating-point number**) (burada fahr) en az 3 karakter genişliğinde, ondalık noktası ve kesir basamağı olmadan bastırılmasını belirtiyor. **%6.1f** diğer bir sayı olan **celcius**'un en az 6 karakter genişliğinde ve ondalık noktasından sonra sadece bir kesir basamağı olacak şekilde bastırılmasını belirtiyor. Çıktı (**output**) şöyle gözüküyor:

```
  0    -17.8
 20     -6.7
 40      4.4
 60     15.6
 80     26.7
100     37.8
120     48.9
140     60.0
160     71.1
180     82.2
200     93.3
220    104.4
240    115.6
260    126.7
280    137.8
300    148.9
```

Genişlik ve nicelik bir belirteçle görmezden gelinebilir: **%6f** sayının en az 6 karakter genişliğinde olmasını belirtiyor; **%.2f** ondalık noktasından sonraki iki karakteri temsil ediyor fakat genişlik zoraki değil ve **%f** sayıyı ondalıklı sayı (**float**) olarak bastırmasını belirtiyor.

```
%d       tamsayı(integer) olarak bastır

%6d      tamsayı(integer) olarak bastır, 6 karakter
   genişliğinde

%f       ondalıklı sayı(float) olarak bastır

%6f      ondalıklı sayı(float) olarak bastır, 6 karakter
         genişliğinde

%.2f     ondalıklı sayı(float) olarak bastır, ondalık
         noktasından sonra sadece iki karakter

%6.2f    ondalıklı sayı(float) olarak bastır, en az 6
   karakter
         genişliğinde ve ondalık noktasından sonra iki
            karakter
```

Diğerlerininin yanında, **printf** ayrıca **%o** belirtecini sekizlikler(**octal**), **%x** onaltılıklar(**hexademical**) için, **%c**'yi karakterler için, **%s**'yi karakter öbekleri(**string**) için, ve **%%**'yi %'ün kendisi için kullanıyor.

Egzersiz 1-3. Isı birimi dönüşüm programını tablonun başındaki yorum satırlarındaki başlığı bastıracak şekilde düzenleyin.

Egzersiz 1-4. Tablodakileri bastıracak benzer bir programı yardım almadan kendiniz yazın.

1.3 For Döngüsü (loop)

Bir programı belirli bir görevi gerçekleştirmesi için çok farklı şekillerde yazabiliriz. Haydi ısı dönüşüm programının bir varyasyonunu oluşturalım.

```
#include <stdio.h>

/* Fahrenhayt-Selsiyus tablosu bastır */
main()
{
   int fahr;

   for (fahr = 0; fahr <= 300; fahr = fahr + 20)
      printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Bu da aynı çıktıyı (**output**) üretecektir, fakat kesinlikle farklı gözüküyor.

En büyük değişiklik birçok değişkenin (**variables**) saf dışı bırakılması; sadece **fahr** kaldı ve onu bir **int** yaptık. **lower** ve **upper** limitleri ve adım büyüklüğü sadece **for** ifadesi (**statement**) içinde sabit (**constant**) olarak karşımızda, bu artık yeni bir inşâ ve Selsiyus sıcaklığını hesaplayan açıklama ayrı bir ifade (**statement**) yerine artık **printf**'nin üçüncü argümanı olarak karşımızda.

Bu son değişiklik bize bir tipin (**type**) değişkeni (**variable**) yerine doğrudan değer (**değer**) kullanmamıza izin verilmesi durumuna bir örnek, bu durumlarda o tipin (**type**) daha komplike açıklamaları kullanılabilir. **printf**'in üçüncü argümanı **%6.1f** ile eşleşecek olan bir ondalıklı sayı (**float**) olması gerektiğinden herhangi bir ondalıklı sayı (**float**) oraya konulabilir.

for ifadesi (**statement**) bir döngü (**loop**), **while**'ın bir genellemesi. Eğer erken (**earlier**) **while** ile karşılaştırırsak, işleyişi açıkça görebiliriz. Parantezler içerisinde noktalı virgüller ile ayrılan üç ayrı bölüm mevcut. İlk bölüm, deklarasyon:

```
fahr = 0
```

Döngü (**loop**) başlamadan önce sadece bir kere yapılır. İkinci bölüm ise test veya döngüyü (**loop**) kontrol edecek olan koşul bölümüdür:

```
fahr <= 300
```

Bu koşul değerlendirilir; eğer doğru ise, döngünün (**loop**) gövde (**body**) kısmı (burada sadece **printf**) çalıştırılır. Ardından arttırma bölümü

```
fahr = fahr + 20
```

çalıştırılır ve koşul tekrardan test edilir. Döngü (**loop**) eğer koşul yanlış (**false**) olursa sonlanır. **while**'da olduğu gibi gövde (**body**) kısmı tek bir ifadeden (**statement**) veya süslü parantezler arasındaki ifade (**statements**) gruplarından oluşabilir. Tanımlama, koşul ve arttırma bölümleri herhangi bir açıklamadan (**expression**) oluşabilir.

while ve **for** arasındaki seçim isteğe bağlı, hangisi gereken durum için daha temiz görünüyorsa onu kullanın. **while**'dan daha kompakt olduğundan ve döngü (**loop**) kontrol ifadelerini (**statements**) beraber tek bir yerde tuttuğundan, **for** genellikle deklarasyon ve arttırmanın tek bir ifade olduğu ve mantıksal olarak ilişkili olduğu döngüler (**loop**) için daha uygundur.

Egzersiz 1-5. Isı dönüşüm programını tabloyu tersine bastıracak şekilde düzenleyin, 300 dereceden 0'a doğru.

1.4 Sembolik Sabitler (constants)

Isı dönüşümünü sonsuza dek bir kenara bırakmadan önce son bir gözlem yapalım. Programa 300 ve 20 gibi "sihirli sayılar" gömmek kötü; bu sayılar bu programı daha sonra okuyacak bir kişinin kolayca görmek ve değiştirmek istediği değerler (**values**) taşıyorlar, bu sayıları bu şekilde gömmek onları sistematik yolla değiştirmeyi zorlaştırdığı gibi hangi sayının neye ait olduğunu da bilmemizi zorlaştırıyor. Bu sihirli sayılarla ilgilenmenin bir yolu onlara anlamlı isimler vermek. #**define** satırı belirli bir karakter öbeğini bir sembolik isim veya sembolik sabit (**constant**) olarak tarif eder (**define**):

```
#define isim(name) karakter öbeği
```

Daha sonra programda herhangi bir yerde bu isim (**name**) geçtiğinde, isim (**name**) ile eşleşen karakter öbeği (**string**) ismin yazıldığı yere geçer. İsmin (**name**) formu değişkenlerin (**variables**) adlarının formuyla aynıdır: bir harf ile başlayan, sayı ve harflerden oluşan dizi. Eşleşen karakter öbeği her türlü karakterden oluşabilir, sadece sayılarla sınırlı değildir.

```
#include <stdio.h>

#define LOWER  0       /* tablonun en düşük limiti */
#define STEP   20      /* adım büyüklüğü */
#define UPPER  300     /* en yüksek limiti */

/* Fahrenhayt-Selsiyus tablosunu bastır */
main ()
{
   int fahr;

   for(fahr = LOWER; fahr <= UPPER; fahr += STEP) {
      printf("%3d\t%6.1f\n", fahr, (5.0/9.0)*(fahr-32));
   }

}
```

LOWER, **UPPER** ve **STEP** nicelikleri birer sembolik sabittir (**constant**), değişken (**variables**) değil; bu yüzden deklarasyonlarla yapılmıyorlar. Sembolik sabitler (**constant**) genellikle büyük harflerle yazılır böylece kolayca küçük harfli değişken (**variable**) adlarından ayırt edilebilirler. #**define** satırının sonunda noktalı virgül olmadığını farkedin.

1.5 Karakter Girdisi (input) ve Çıktısı (output)

Artık karakter verisi işleyecek, birbiriyle alakalı bir program ailesini konuşacağız. Birçok programın, sadece buradaki prototiplerin genişletilmiş versiyonları olduğunu farkedeceksiniz.

Standart girdi (**input**) çıktı (**output**) kütüphanesinin (**library**) desteklediği girdi (**input**) çıktı (**output**) modeli oldukça basit. Metin girdisi (**input**) ve çıktısı (**output**); nereden kaynaklandığı ve nereye gittiği farketmeksizin, karakter akıntıları (**stream of charachters**) ile yapılır. *Metin akıntısı* (**text stream**) satırlara bölünmüş bir karakter dizisidir; her satır sıfır veya daha fazla karakterden ve onları takiben gelen bir yeni satır karakterinden oluşur. Her girdi (**input**) ve çıktı (**output**) akışını bu modele uydurmak kütüphanenin (**library**) sorumluluğunda; C programcısı kütüphaneyi satırların program dışında nasıl temsil edildiği hakkında endişelenmemek için kullanıyor.

Standart kütüphane (**library**) bize bir kerede bir adet karakter yazdırmak için birçok fonksiyon sağlıyor, **getchar** ve **putchar** en basitleri. Her çağrıldığında **getchar** bir sonraki girdi (**input**) karakterini metin akışından okuyor ve onu kendi değeri (**value**) olarak döndürüyor (**return**). Değişken (**variable**) c

```
c = getchar()
```

bir sonraki girdi (**input**) karakterini içerir. Karakterler genelde klavyeden gelir; dosyalardan girdileri (**input**) Bölüm 7'de konuştuk. **putchar** fonksiyonu her çağrıldığında bir karakteri bastırıyor:

```
putchar(c)
```

Tam sayı olan değişken (**variable**) c'yi karakter olarak bastırıyor, genellikle ekrana. **putchar** ve **printf** çağrıları belki iç içe geçmiş olabilir, çıktı (**output**) yapılan çağrıya göre ortaya çıkacaktır.

1.5.1 Dosya Kopyalama

Bilinen **getchar** ve **putchar** ile, girdi (**input**) ve çıktı (**output**) dışında başka bir şey bilmeden şaşırtıcı derecede fazla yararlı kod yazabiliriz. En basit örnek girdiyi (**input**) çıktıya (**output**) bir kerede bir karakter olarak kopyalayan program:

```
karakteri oku
while (karakter dosya-sonu göstergesi değil)
   okuduğun karakteri çıktıla(output)
   karakter oku
```

Bunu C'ye dönüştürürsek:

```
#include <stdio.h>

/* girdiyi (input) çıktıya (output) kopyala; 1. versiyon */
main()
{
   int c;

   c = getchar();
   while (c != EOF) {
      putchar(c);
      c = getchar();
   }
}
```

Mantıksal bir operatör olan **!=** "eşit değil" anlamına gelir.

Tabiki ekrandaki yada klavyedeki karakter, diğer her şey gibi, bit olarak içeride saklanıyor. Tür (**type**) **char** özellikle karakter verisi taşıması için oluşturuldu fakat herhangi bir tam sayı (**integer**) değeri kullanılabilir. Biz ince ve önemli bir sebeple **int** kullandık.

Problem, girdinin (**input**) sonunun geçerli veriden ayırılması. Çözüm ise **getchar**'ın girdi (**input**) kalmadığında herhangi gerçek bir karakterle karıştırılamayacak belirgin bir değer dönmesi (**return**). Bu değer **EOF** olarak adlandırılıyor, "dosya sonu" (**end of file**) anlamına geliyor. Biz c'yi **getchar**'ın döndürebileceği kadar büyük bir değer taşıyan bir tipte deklare etmemiz gerekiyor. c, lazım olduğunda EOF'yi taşıyabileceği kadar büyük olması gerektiğinden **char** kullanamazdık. Bu nedenle **int** kullandık.

EOF <**stdio.h**>'da bir tam sayı (**integer**) olarak tarif edilmiştir (**define**), fakat EOF'nin sayı değeri hakkında **char** değeri ile aynı olmadığından dolayı endişelenmemiz gerekmiyor. Sembolik sabiti (**Symbolic Constant**) kullanarak, programdaki hiçbir şeyin belirli bir sayı değerine dayalı olmadığından emin olduk.

Kopyalama programı tecrübeli C programcıları tarafından daha farklı yazılabilir. C'de herhangi bir atama (**assignment**), örneğin

```
c = getchar()
```

bir açıklamadır (**expression**) ve değeri vardır, eşitliğin sol tarafindekinin değeri atamadan (**assignment**) sonra belirlenir. Böylece bir atama (**assignment**) daha geniş bir açıklamanın bir parçası olabilir. Eğer c'nin bir karaktere atanması **while** döngüsünün test kısmının içerisine konulursa, kopyalama programı bu şekilde yazılabilir:

```
#include <stdio.h>

/* girdiyi (input) çıktıya (output) kopyala; 2. versiyon */

main()
{
   int c;

   while ((c = getchar()) != EOF)
      putchar(c);
}
```

while karakteri alıyor, c'ye atıyor, ve karakterin dosya-sonu (**end-of-file**) sinyali olmadığı koşulunu test ediyor. Eğer değilse, **while**'ın gövde (**gövde**) kısmı çalıştırılıyor ve karakter bastırılıyor. Sonra **while** tekrar ediyor. Ve girdinin sonuna ulaşıldığında, **while** sona eriyor ve ardından tabiki **main** de.

Bu versiyon girdiyi (**input**) merkezleştiriyor - artık sadece tek **getchar**'a tek bir referans var - ve programı kısaltıyor. Son program çok daha kompakt, ve bu kullanımda ustalaştıkça, okumak da daha kolay. Bu tarzı çok sık göreceksiniz. (bunu biraz abartmak ve anlaşılmaz kodlar yazmak mümkün, bu kendimizi tutmamız gereken bir eğilim.)

Atamanın etrafındaki koşulun içindeki parantezler gerekli. !='in *önceliği* (**precedence**)=='den yüksek, bu yüzden parantezler olmasaydı mantıksal test olan != atama (**assignment**) olan=='den önce yapılırdı.

Bu yüzden ifade (**statement**)

```
c = getchar() != EOF
```

eşittir

```
c = (getchar() != EOF)
```

Bunun **getchar**'ın dosya-sonu (**end-of-file**) sinyaline ulaşıp ulaşmadığına göre c'yi ya 0'a yada 1' atamak (**assignment**) gibi bir istenmeyen etkisi var. (Bölüm 2'de bununla ilgili fazlası var.)

Egzersiz 1-6. Açıklama (**expression**) getchar() != EOF'nin 0'a yada 1'e eşit olduğunu doğrulayın.

Egzersiz 1.7. EOF'in değerini bastıran bir program yazın.

1.5.2 Karakter Sayma

Bir sonraki program karakterleri sayıyor; kopyalama programına oldukça benziyor.

```
#include <stdio.h>
/* girdideki (input) karakterleri say; 1. versiyon */
main()
{
   long nc;

   nc = 0;
   while (getchar() != EOF)
      ++nc;
   printf("%ld\n", nc);
}
```

İfade (**statements**)

```
++nc;
```

yeni bir operatörü bize tanıtıyor, ++, *bir ile arttır* anlamına geliyor. Bunun yerine nc= nc+1 yazabilirdiniz fakat ++nc çok daha kısa ve çoğunlukla daha verimli. Bu operatörün eşi – operatörü ise bir ile azalt anlamına geliyor. ++ ve – operatörleri önek (**prefix**) (++nc) veya arkak (**postfix**) (nc++) olabilirler; bu ki formun Bölüm 2'de göstereceğimiz gibi açıklamalarda (**expressions**) farklı değerleri olur, fakat hem nc++ hem ++nc de nc'yi birer arttıracaktır. Şuanlık sadece önek (**prefix**) formunu kullanacağız..

Karakter sayma programı sayımlarını **int** yerine **long** bir değişken (**variable**) içinde biriktirir. **long** tam sayılar (**integer**) en az 32 bit olsa da bazı makinelerde **long** ve **int** aynı boyuttadır, diğerlerinde ise **int** maksimum değeri 32767'dir ve 16 bit yer kaplar. Bu sebeple **int** bir sayacı geçmek oldukça az

girdi (**input**) gerektirecektir. Dönüşüm özelliği (**conversion spesification**) **%ld printf**'e eşleşecek olan argümanın bir **long** tam sayı (**integer**) olacağını söyler.

Bundan bile daha büyük sayılarla bile başa çıkmak ise **double** (double precision float) ile mümkündür. Ayrıca döngü (**loop**) yazmanın bir diğer yolunu göstermek için **while** ifadesi (**statement**) yerine de **for** ifadesi kullanacağız.

```
#include <stdio.h>

/* girdideki (input) karakterleri say; 2. versiyon */
main()
{
   double nc;

   for (nc = 0; getchar() != EOF; ++nc)
      ;
   printf("%.0f\n", nc);
}
```

printf %f'i hem **float** hem de **double** için kullanıyor; **%0.f** ise ondalık noktasını ve bu programda her zaman 0 olan kesirli kısmın bastırılmamasını sağlıyor.

Bu **for** döngüsünün (**loop**) gövde (**body**) kısmı boş, çünkü bütün iş koşul ve arttırma bölümleri tarafından yapılıyor. Fakat C'nin gramatik kuralları gereği **for** ifadesinin (**statement**) bir gövdesi olması gerek. Yalnız duran noktalı virgül, *boş ifade* (**null statement**) olarak adlandırılır, bu C'nin bu gerekliliğini sağlamak için kullanılır. Bunu daha görünür yapmak için ayrı bir satıra koyduk.

Karakter sayma programını bir kenara bırakmadan önce, eğer girdi (**input**) hiç karakter içermiyorsa, **while** veya **for** daha **getchar**'ı ilk çağırışta sona eriyor, ve program doğru sonuç olan 0 sonucunu üretiyor. Bu önemli. **while** ve **for** ile ilgili güzel şeylerden biri ise, daha döngünün (**loop**) gövde (**body**) kısmına başlamadan önce koşul bölümünü test ediyor olmaları. Eğer yapılacak bir şey yoksa, hiçbir şey yapılmıyor, bu döngünün (**loop**) gövde (**body**) kısmının asla çalıştırılmayacak olduğu anlamına gelse bile. Programlar onlara sıfır-uzunlukta girdi (**input**) verildiğinde zekice haraket etmelidirler. **while** ve **for** ifadeleri (**statements**) en ekstrem (sınır, limit) koşullarda bile programların mantıklı şeyler yapmasından emin oluyor.

1.5.3 Satır Sayma

Bir sonraki program girdideki (**input**) satırları sayıyor. Daha önce belirttiğimiz gibi, standart kütüphane, girdi (**input**) metin akışının satır dizele-

riyle ortaya çıkmasını sağlıyor, her satır bir yeni satır karakteri ile bitiyor. Bu yüzden satırları saymak ile yeni satır karakterlerini saymak aynı şey:

```
#include <stdio.h>

/* girdiden (input) satır say */
main()
{
   int c, n1;

   n1 = 0;
   while ((c = getchar()) != EOF)
      if (c == '\n')
         ++n1;

   printf("%d\n", n1);

}
```

while'ın gövde kısmı şuan **if**'ten oluşuyor, burada **if**, ++n1 artışını kontrol ediyor. **if** ifadesi kendisini takiben gelen (**statement**) parantezler içerisindeki koşulu test ediyor, ve koşul doğru (**true**) ise, ifadeyi (**statement**) (veya süslü parantezler içerisindeki bir grup ifadeyi çalıştırıyor. Neyin ne tarafından kontrol edildiğini göstermek için yine girintiledik.

Çift eşittir== gösterimi "eşit mi" anlamına gelen bir C notasyonudur (Pascal'ın tek=i veya Fortran'ın .EQ.'sı gibi). Bu sembol eşitlik testini C'de atama (**assignment**) için kullanılan tek=den ayırmak için kullanıldı. Biraz uyarı: C'ye yeni başlayanlar ara sıra ==i kast etmelerine rağmen = yazıyorlar. Bölüm 2'de göreceğimiz üzere bunun sonucu geçerli bir açıklama (**expression**) oluyor ve bir uyarı almıyorsunuz.

Tek tırnak arasına yazılan karakter, makinenin karakter setindeki, karakterin numerik değerine eşit olan bir tamsayıyı (**integer**) temsil ediyor. Bu *karakter sabiti* (**character constant**) olarak adlandırılıyor, buna rağmen bu sadece küçük bir tamsayı (**integer**) yazmanın bir diğer yolu. Örneğin, **'A'** bir karakter sabiti (**character constant**); ASCII karakter setindeki değeri 65, bu, karakter **A**'nın içerideki temsili. Tabiki **'A'** 65'in üzerinde bir değer olarak tercih edilebilir: bunun anlamı açık, ve belirli bir karakter setinden bağımsız.

Karakter öbeği sabitlerinde (**string constants**) kullanılan kaçış dizelerinin (**escape sequences**) kullanımı da geçerli, yani **'\n'** yeni satır karakterinin değerini temsil ediyor, ASCII'de bu değer 10. **n**'in tek bir karakter olduğunu farketmeniz gerekiyor, ve bu açıklama (**expression**) sadece bir tamsayı (**integer**); bir diğer deyişle, **"\n"** sadece tek bir karakter içeren bir karakter öbeği sabiti (**string constant**). Karakter öbeklerinin (**string**) ve

karakterlerin farklarına Bölüm 2'nin sonlarında bahsedeceğiz.

Egzersiz 1-8. Boşlukları, tableri, ve yenı satırları sayan bir program yazınız.

Egzersiz 1-9. Girdisini (**input**) çıktısına (**output**) kopyalayan her karakter öbeğindeki (**string**) birden fazla olan boşlukları tek boşlukla değiştirecek bir program yazınız.

Egzersiz 1-10. Girdisini (**input**) çıktısına (**output**) kopyalayan bir program yazınız, her tabi \t ile, her geri tuşunu \b ile ve her ters eğik çizgiyi \\ile değiştirecek. Bu tableri, ters eğik çizgiyi ve geri tuşu karakterlerini daha açık bir yolla gösterecek.

1.5.4 Kelime Sayma

Yararlı programlar serimizdeki dördüncü program, gevşek bir tarif ile (**definition**) boşluk, tab veya yeni satır içermeyen her karakter sekansını kelime olarak saymakla birlikte satırları ve karakterleri sayıyor.

Bu bir UNIX programı olan **wc**'nin oldukça temel bir hali.

```
#include <stdio.h>

#define IN 1 /* kelimenin içinde */
#define OUT 0 /* kelimenin dışında */

/* girdideki (input) satırları, kelimeleri ve karakterleri
   say */
main ()
{
   int c, n1, nw, nc, state;

   state = OUT;
   n1 = nw = nc = 0;
   while((c = getchar()) != EOF) {
      ++nc;
      if (c == '\n')
         ++n1;
      if (c == ' ' || c == '\n' || c == '\t')
         state = OUT;
      else if (state == OUT) {
         state = IN;
         ++nw;
      }
   }
   printf("%d %d %d\n", n1, nw, nc);
}
```

Program her bir kelimenin ilk harfi ile karşılaştığında bir kelime daha sayıyor. Değişken (**variable**) **state** programın anlık olarak bir kelimenin içinde mi yoksa dışında mı olduğunu kaydediyor; başlangıçta "kelimenin içinde değil" yani OUT'un değerine atanıyor. Biz, gerçek sayılar olan 0 ve 1 yerine IN ve OUT olan sembolik sabitleri (**symbolic constants**) kullanmayı tercih ediyoruz çünkü bu programı daha okunabilir bir hale getiriyor. Bu gibi küçük programlara oldukça az fark yaratıyor, fakat daha büyük programlarda, programın berraklığındaki artış programın en başından beri sadece mütevazi bir eforla bu şekilde yazmaya değer. Ayrıca programda geniş değişiklikler yapmanın önemli sayıların sadece sembolik sabitler (**symbolic constants**) olarak ortaya çıkmasıyla daha kolay olduğunu göreceksiniz.

Bu satır

```
n1 = nw = nc = 0;
```

üç değişkeni (**variable**) de 0'a atıyor. Bu özel bir durum değil, fakat bu durumun sonucu oluşan, bir değerle beraber bir açıklama (**expression**) olan bir atama (**assignment**) ve atamalar (**assignments**) sağdan sola doğru bağlanır. Bu şekilde yazmışız gibi

```
n1 = (nw = (nc = 0));
```

Operatör (**operator**) || yada (**OR**) anlamına gelir, yani bu satır

```
if (c == ' ' || c == '\n' || c == '\t')
```

şunu söylüyor "c bir boşluk veya c bir yeni satır veya c bir tab ise". (Kaçış dizesi (**escape sequence**) olan \t'nin tab karakterinin görünür temsili olduğunu belirtelim) VE'nin (**AND**) yerine geçen bir operatör var &&; ve önceliği ||'den büyük. && veya || ile bağlanan açıklamalar (**expressions**) soldan sağa doğru değerlendirilir, ve değerledirmenin yanlış (**false**) olma sonucuna ulaştığı bilindiğinde duracağı garanti edilmiştir. Eğer c boşluk ise, c'nin yeni satır veya tab olup olmadığını test etmeye gerek yoktur, bu yüzden bu testler yapılmaz. Bunun burada bir önemi yok, fakat yakında göreceğimiz daha komplike durumlarda önem arz ediyor.

Örnek, eğer **if**'in koşulu yanlış (**false**) çıkarsa alınacak aksiyonları belirten **else**'i gösteriyor. Genel form şu şekilde

```
if (açıklama (expression))
    ifade1 (statement1)
else
    ifade2 (statement2)
```

İki ifadenin (**statement**) tek ilişkisi **if-else**'in gerçekleşmesidir. Eğer açıklama (**expression**) doğru (**true**) ise, $ifade_1$ ($statement_1$) çalıştırılır, eğer doğru değil ise, $ifade_2$ ($statement_2$) çalıştırılır. Her ifade (**statement**) tek veya süslü parantezler içerisinde birden fazla olabilir. Kelime sayma programında, **else**'den sonraki if süslü parantez içerisinde iki ifadeyi (**statement**) kontrol ediyor.

Egzersiz 1-11. Kelime sayma programını nasıl test ederdin ? Eğer programda hatalar varsa bunları ortaya çıkaracak girdi (**input**) türleri nelerdir ?

Egzersiz 1-12. Satır başı tek kelime bastıracak bir program yazınız.

1.6 Diziler (arrays)

Her rakamın kaç kere yazıldığını, boşluk karakterleri (boşluk, tab, yeni satır) ve diğer tüm karakterleri sayan bir program yazalım. Bu yapmacık, fakat bize C'nin bir çok yönünü tek bir programda göstermemizi sağlayacak.

Programımızda girdinin (**input**) 12 farklı kategorisi var, bu yüzden her rakamın kaç kere yazıldığını bir dizide (**array**), 10 farklı değişken (**variable**) kullanmaktan çok daha uygun. İşte programın bir versiyonu:

```
#include <stdio.h>

/* rakamları, boşlukları ve diğerlerini say */

main()
{
   int c, i, nwhite, nother;
   int ndigit[10];

   nwhite = nother = 0;
   for (i = 0; i < 10; ++i)
      ndigit[i] = 0;

   while ((c = getchar()) != EOF)
      if (c >= '0' && c <= '9')
         ++ndigit[c-'0'];
      else if (c == ' ' || c == '\n' || c == '\t')
         ++nwhite;
      else
         ++nother;

   printf("digits =");
   for (i = 0; i < 10; ++i)
      printf(" %d", ndigit[i]);
   printf(", white space = %d, other = %d\n",
         nwhite, nother);
}
```

Programa girdi (**input**) olarak kendi kaynak kodu verildiğinde çıktısı (**output**)

```
digits = 9 3 0 0 0 0 0 0 0 1, white space = 126, other =
   358
```

Tanımlama (**decleration**)

```
int ndigit[10];
```

ndigit'i 10 tamsayıdan (**integer**) oluşan biri dizi (**array**) olarak deklare ediyor.

Dizilerin (**arrays**) alt elemanları (**subscripts**) C'de her zaman 0 ile sayılmaya başlanır, Yani elemanlar bu şekildedir **ndigit[0], ndigit[1], ..., ndigit[9]**. Bu, dizileri (**arrays**) başlatan ve onları bastıran **for** döngülerinde (**loop**) gösterilmiştir.

Eleman (**subscript**) herhangi bir tamsayı (**integer**) açıklama (**expression**) olabilir, i gibi tamsayı (**integer**) değişkenleri (**variables**) ve tamsayı (**integer**) sabitleri (**constants**) gibi.

Bu program rakamların karakter temsillerinin özelliklerine dayanıyor. Örneğin, bu test

```
if (c >= '0' && c <= '9')
```

c'nin içindeki karakterin bir rakam olup olmadığını belirliyor. Eğer öyle ise, o rakamın numerik değeri

```
c - '0'
```

Bu sadece '0', '1', ..., '9' gibi ardışık değerlerde çalışıyor. Neyse ki bu bütün karakter setleri için geçerli.

Tanımları gereği **char**'lar sadece küçük tamsayılar (**integer**), yani **char** değişkenleri (**variables**) ve sabitleri (**constants**) aritmetik açıklamalarda (**expressions**) **int** olarak tanınırlar. Bu gayet doğal ve kullanışlı; örneğin **c-'0'**, 0 ve 9 arasında c içinde saklanan '0' ile '9' arasındaki bir karakter ile eşleşen bir tamsayı (**integer**) açıklaması (**expression**), bu nedenle **c-'0' ndigit** dizisi (**array**) için geçerli bir eleman (**subscript**).

Karakterin bir rakam mı, yoksa bir boşluk karakteri mi, veya başka bir şey mi olduğu bu sekans ile yapılıyor

```
if (c >= '0' && c <= '9')
   ++ndigit[c-'0'];
else if (c == ' ' || '\n' || c == '\t')
   ++nwhite;
else
   ++nother;
```

Bu desen

```
if ( koul1 (condition))
    ifade1 (statement)
else if ( koul2 (condition))
    ifade2 (statement)
...
   ...
else
   ifaden (statement)
```

sıklıkla karşımıza çok yönlü kararların açıklandığı programlarda çıkıyor. Koşullar (**conditions**) yukarıdan aşağıya bir koşul (**condition**) doğru olana kadar değerlendiriliyor; bu noktada eşleşen *ifade* (**statement**) kısmı çalıştırılır, ve bütün yapı (**construction**) biter. (Her *ifade* (**statement**) süslü parantezler içerisinde birden fazla olabilir.) Eğer hiçbir koşul (**condition**) sağlanmazsa, eğer varsa **else**'den sonraki *ifade* (**statement**) çalıştırılır. Eğer son **else** ve *ifade* (**statement**) atlanırsa, kelime sayma programında yapıldığı gibi, herhangi bir aksiyon alınmaz. **if** ve son **else** arasında

```
else if ( koul (condition) )
    ifade (statement)
```

sayısız grup olabilir.

Tarz gereği, yapıyı (**construction**) gösterdiğimiz formatta yazmanız önerilir; eğer her **if else**'den önce girdilenirse kararlardan uzun bir sekans sayfanın sağ tarafına doğru kayacaktır.

switch ifadesi (**statement**), Bölüm 3'de anlatılacaktır, **switch**, koşul (**condition**), bir sabitler (**constants**) setiyle eşleşen bir tamsayı (**integer**) veya karakter açıklaması (**expresion**) olduğunda kullanımı özellikle uygun olan bir diğer çok yönlü dal (**branch**) yazmamıza olanak sağlıyor. Karşılaştırmak için bu programın **switch** versiyonunu Bölüm 3.4'de göstereceğiz.

Egzersiz 1-13. Girdisindeki (**input**) kelimelerin uzunluğunun bir histogramını basan bir program yazınız. Yatay barlarla bir histogram çizmek oldukça kolay; dikey dizilim çok daha zorlu olacaktır.

Egzersiz 1-14. Farklı karakterlerin sıklığının bir histogramını bastıran bir program yazınız.

1.7 Fonksiyonlar (functions)

C'de, fonksiyonlar Fortran'daki altprogramların (**subroutines**) veya fonksiyonların veyahut Pascal'daki prosedürlerin (**procedure**) veya fonksiyonların denkidir. Bir fonksiyon , bir hesaplamayı daha sonra kendi emplementasyonunda hakkında endişelenilmeden kullanmak üzere barındırmak için kullanışlı bir yol sağlar. Uygun şekilde düzenlenen fonksiyonlarla , işin *nasıl* gerçekleştirildiğini görmezden gelmek mümkün; sadece *ne* yapıldığını bilmek yeterlidir. C, fonksiyonların kullanımını kolay, kullanışlı ve verimli hale getiriyor; sıklıkla kısa bir fonksiyonun tarif edildiğini (**define**) ve sadece küçük bir parça kodu içerdiğinden sadece bir kere çağırıldığını (**call**) göreceksiniz.

Bu zamana kadar sadece **printf**, **getchar**, ve **putchar** gibi bize sağlanan fonksiyonları kullandık; artık birkaçını kendimizin yazma vakti geldi. C'nin Fortran'ın **'ı gibi bir üsalma operatörü olmadığından, fonksiyon tarifinin (**definition**) mekaniklerini göstermek için **power(m,n)** fonksiyonunu, m tamsayısının (**integer**) n kuvvetini hesaplamak ve bunu göstermek için yazalım. Bu, **power(2,5)**'in değerini 32 yapar. Bu fonksiyon , sadece küçük tamsayıların (**integer**) pozitif değerleriyle hesap yapabildiğinden pratik bir bir üsalma rutini değil, fakat bu gösterimi yapmamız için yeterli. (Standart kütüphane (**library**) **pow(x,y)** adında x^y'yi hesaplayan bir fonksiyon içeriyor.)

İşte, fonksiyon **power** ve onu çalıştırması için main programı, bu sayede bütün yapıyı (**structure**) bir arada görebilirsiniz.

```
#include <stdio.h>

int power(int m, int n);

/* power fonksiyonunu test et */
main()
{
   int i;

   for (i = 0; i < 10; ++i)
      printf("%d %d %d\n", i, power(2,i), power(-3,i));
   return 0;
}

/* power: base sayısını n'inci kuvvetine yükselt; n >= 0 */
int power(int base, int n)
{
   int i ,p;

   p = 1;
   for (i = 1; i <= n; ++i)
      p = p * base;
   return p;
}
```

Fonksiyon tarif etmek (**definition**) böyle bir forma sahip:

```
dönüş-tipi fonksiyon-adı(parametre deklarasyonları, eğer
   varsa)
{
   deklarasyonlar
   ifadeler  (statements)
}
```

dönüş-tipi (**return-type**) fonksiyon

Fonksiyon tarifleri (**definition**) herhangi bir şekilde bulunabilir, bir kaynak (**source**) dosyasında veya birden fazlasında, buna karşın hiçbir fonksiyon dosyalar arasında bölünemez. Eğer kaynak (**source**) program birden fazla dosya halinde bulunuyorsa, belki tek dosya halinde bulunandan daha fazla derleyip (**compile**) yükleyecek (**load**) olduğunuzu söyleyebilirsiniz fakat bu bir işletim sistemi konusu, dilin bir niteliği değil. Şu anlık, iki fonksiyonun da aynı dosya içinde bulunduğunu farz ediyoruz, bu yüzden şuana kadar C programlarının çalışması hakkında öğrendiğiniz her şey hala geçerli.

Fonksiyon **power**, **main** tarafından iki kere çağırılıyor, bu satırda

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

her çağırma, her defasında formatlanıp bastırılacak olan bir tam sayı (**integer**) döndüren **power**'a iki argüman geçiriyor. Açıklamada **power(2,i)** aynı i ve 2 gibi birere tamsayı (**integer**). (Her fonksiyon bir tamsayı (**integer**) değeri üretmez; bunu Bölüm 4'de konuşacağız.)

power'ın kendisinin ilk satırı:

```
int power(int base, int n)
```

parametre tiplerini (**type**), adlarını ve fonksiyonun döndüreceği sonucun tipini (**type**) deklare ediyor. **power** tarafından parametreleri için kullanılan isimler **power**'a yereldir (**local**), ve bir değer fonksiyon tarafından görünebilir değildir: diğer rutinler aynı ismi bir çakışma yaşanmadan kullanabilirler. Bu aynen i ve p değişkenleri (**variable**) için de geçerli; **power** içindeki i ile **main** içindeki i tamami ile ilişkisiz.

Biz genellikle *parametre*'yi fonksiyon tarifinde (**definition**) parantezler içerisindeki listede adlandırılmış değişkenler (**variable**) için, argümanı ise fonksyon çağırımı (**call**) sırasında kullanılan değerler (**value**) için kullanacağız. *biçimsel* (**formal**) argümanın ve *asıl* (**actual**) argümanın koşulları bazen aynı ayrımı yapmak için kullanılır.

power'ın hesapladığı değer, **return** ifade (**statement**) tarafından **main**'e döndürüldü. **return**'ü takip eden herhangi bir açıklama:

```
return açıklama (expression)
```

Bir değer döndürmesine gerek olmayan fonksiyonda: bir açıklaması (**expression**) olmayan **return** ifadesi (**statement**) kontrolü sağlar, fakat çağırıcı (**caller**) tarafından kullanılacak kullanışlı bir değeri (**value**) değil, fonksiyonu sonlandıran sağ süslü paranteze ulaşarak fonksiyonun "sonuna düşmek" ile olduğu gibi. Ve çağırılan fonksiyon , fonksiyon tarafından döndürülen (**return**) değeri (**value**) görmezden gelebilir.

main'in sonunda bir **return** ifadesi (**statement**) olduğunu farketmişsinizdir. **main** diğerleri gibi bir fonksiyon olduğundan dolayı, o da çağıra-

nına (**caller**) bir değer gönderebilir, bu da programın çalıştırıldığı çevreyi (**environment**) etkileyebilir. içeriği sıfır olan bir **return** değeri (**value**) normal bitişi (**termination**); sıfır-olmayan değerler (**values**) ise sinyal alışılmadık (**unusual**) veya hata (**erroneous**) bitirme koşullarıdır (**conditions**). Bu noktadan önce basitlik uğruna, **main** fonksiyonlarımızdaki **return** ifadesini (**statement**) atladık, fakat bundan böyle artık onları da dahil edeceğiz, bir hatırlatma olarak, programların durumlarını çevrelerine (**environment**) döndürmelerinin (**return**) iyi olacağını belirtelim.

main'den hemen önceki bu deklarasyon

```
int power(int m, int n);
```

power'ın iki **int** argüman bekleyen ve int döndüren (**return**) bir fonksiyon olduğunu söylüyor. *fonksiyon prototipi* olarak adlandırılan bu deklarasyon, **power**'ın tarifi (**definition**) ve kullanımı konusuna açıklık getirmek zorunda. Bu fonksiyonun tarifi (**definition**) veya herhangi bir kullanımı ile prototipi arasında bir uyuşmazlık var ise bu hataya sebep olur.

Parametre isimlerinin uyuşmasına gerek yok. Aslında parametre isimleri fonksiyon prototipi için opsiyonel, yani prototip için bunu da yazabilirdik

```
int power(int, int);
```

Güzel çizilen isimler güzel dökümantasyon sağlar, her nasılsa, onları sıklıkla kullanacağız.

Tarih notu: ANSI C ve daha önceki versiyonlar arasındaki en önemli fark, fonksiyonların nasıl tarif (**definition**) edildiği ve deklare edildiğidir. C'nin orijinal tarifinde (**definition**), **power** fonksiyonu bu şekilde yazılmalıydı:

```
/* power: base sayısını n'inci kuvvetine yükselt; n >= 0 */
/*      (eski-stil versiyon) */
power(base, n)
int base, n;
{
   int i, p;

   p = 1;
   for (i = 1; i <= n; ++i)
      p = p * base;
   return p;
}
```

Parametreler parantezler arasında isimlendiriliyordu, ve onların tipleri (**type**) açılan sol süslü parantezden önce deklare ediliyor; deklare edilmemiş (**undeclared**) parametreler **int** olarak alınıyordu. (Fonksiyonun gövdesi önceki ile aynı)

Eski stilde **power**'ın deklarasyonu böyle gözükmeliydi:

```
int power();
```

Parametre listesine izin verilmiyordu, bu yüzden derleyici (**compiler**) **power**'ın doğru çağırılıp çağırılmadığını kolayca kontrol edemiyordu. Aslında, varsayılan olarak **power**'ın **int** döndüreceği (**return**) farzedildiğinden, tüm deklarasyon işlemi atlanabilirdi.

Fonksiyon prototiplerinin yeni sözdizimi (**syntax**) derleyicinin (**compiler**) hataları ve argümanların sayılarını veya onların tiplerini (**types**) tespit etmesini oldukça kolaylaştırıyor. Eski stil deklarasyon ve tarif etme (**definition**) ANSI C'de halen çalışıyor, en azından bir geçiş periyodu için, fakat eğer destekleyen bir derleyiciniz (**compiler**) varsa yeni formu kullanmanızı şiddetle tavsiye ediyoruz.

Egzersiz 1-15. Bölüm 1.2'nin ısı dönüşüm programını dönüşüm için bir fonksiyon kullanmak üzere yeniden yazınız.

1.8 Argümanlar - Değer ile Çağırma

C fonksiyonlarının bir yönü başka programlama dillerini, özellikle Fortran'ı kullanmış olan programcılara yabancı gelebilir. C'de, bütün fonksiyon argümanları "değerleriyle" (**by value**) geçirilir. Bu, çağırılan (**call**) fonksiyona, fonksiyonun argümanlarına verilmiş değerlerin (**value**), orijinallerinden ziyade, geçici değişkenler (**variables**) olarak verilmesidir. Bu, bizi "referans ile çağır" (**call by reference**) kullanan Fortran gibi veya **var** parametreleri içeren Pascal gibi dillerden farklı özellikler getiriyor, bu gibi dillerde çağırılan (**call**) rutinin yerel (**local**) bir kopyadan ziyade, orijinal argümana erişimi var.

Ana değişiklik, C'de çağırılan (**call**) fonksiyonun direkt olarak çağırılan fonksiyondaki değişkende (**variable**) bir değişiklik yapamamasıdır; sadece kendi özel, geçici kopyasını değiştirebilir.

Değer ile çağır (**call by value**) bir varlık (asset), ancak, bir yükümlülük değil. Bu, daha az fazladan değişken (**variable**) ile daha kompakt programları sağlıyor, çünkü çağırılan (**call**) rutinde parametrelere, sıradan şekilde ilklendirilmiş (**initialize**) yerel (**local**) değişkenler gibi davranılabiliyor. Örneğin, **power**'ın bu özelliği kullanan bir versiyonu.

```
/* power: base sayısını n'inci kuvvetine yükselt; n >= 0;
   versiyon 2 */
int power(int base, int n)
{
   int p;

   for (p = 1; n > 0; --n)
      p = p * base;
   return p;
}
```

Parametre n, geçici bir değişken (**variable**) olarak kullanılıyor, ve geriye doğru sayılıyor (**for** döngüsü (**loop**) geriye doğru çalışıyor, sıfır olana kadar; artık değişken (**variable**) i'ye ihtiyaç yok. **power**'ın içinde n'ye ne yapıldıysa **power**'ın çağırıldığı argüman üstünde hiçbir etkisi yok.

Gerektiğinde, çağırılan (**call**) rutindeki bir değişkeni (**variable**) değiştirmek üzere bir fonksiyon yapılabilir. Çağıran, değişkenin (**variable**) yazılacak bir *adresini* sağlamalı (teknik olarak değişkeni (**variable**) işaret eden bir *işaretleyici* (**pointer**)), ve çağırılan (**call**) fonksiyon parametrenin bir işaretleyici (**pointer**) olduğunu deklare etmeli ve değişkene (**variable**) bunun üzerinden direkt olarak erişmeli. İşaretleyicileri (**pointer**) Bölüm 5'de konuşacağız.

Diziler (**array**) için hikaye farklı. Bir dizinin (**array**) adı argüman olarak kullanıldığında, fonksiyona geçen değer, dizinin (**array**) başlangıcının konumu veya adresi - dizi (**array**) elemanların bir kopyalanma seçeneği yok. Bu değerin (**value**) yardımıyla, fonksiyon bir dizinin (**array**) her elemanına erişip değiştirebilir. Bu bir sonraki bölümün konusu.

1.9 Karakter Dizileri (arrays)

C'deki en yaygın dizi (**array**) tipi, karakter dizileridir (**array**). Karakter dizilerinin (**array**) ve onları manipüle etmek için kullanılan fonksiyonları göstermek üzere, bir grup metin satırı okuyarak aralarından en uzun olanı bastıran bir program yazalım. Programın dış hatları bu kadar basit:

```
while (başka bir satır var)
   if (bu satır bir önceki en uzun satırdan daha uzun)
      satırı kaydet
      satırın uzunluğunu kaydet
print en büyük satır
```

Bu dış hatlar, programın doğal olarak ayrılacağı parçaları açığa kavuşturuyor. Bir parça yeni bir satır alıyor, bir diğeri test ediyor, bir diğeri kaydediyor, ve kalanı işlemi kontrol ediyor.

Bunlar güzelce parçalara ayırıldığından, bunları aynı şekilde yazmak da güzel olacak. Buna uygun olarak, bir sonraki girdi (**input**) satırını okuyacak **getline** isimli ayrı bir fonksiyon yazalım. Fonksiyonu farklı şartlar altında da yararlı olacak şekilde yazmaya çalışacağız. En basiti, **getline** olası bir dosya sonu (**end of file**) durumunda bir sinyal döndürmeli (**return**); daha kullanışlı bir tasarım satırın uzunluğunu, veya dosya sonu (**end of file**) ile karşılaşıldığında zero döndürürdü Sıfır kabul edilebilir bir **end-of-file** döndürülüşü çünkü sıfır geçerli bir satır uzunluğu değil. Her metin satırı en azından bir karaktere sahip; yeni satır karakteri içeren bir satırın bile uzunluğu 1.

Bir önceki en uzun satırdan daha uzun bir satır bulduğumuzda, bu satır bir yere kaydedilmeli. Bu bize bir diğer yazmamız gerekn fınksiyonu öneriyor, yeni satırı başka bir yere kopyalamak için **copy** fonksiyonu.

Sonunda, **getline** ve **copy** fonksiyonlarını kontrol etmesi için bir main programına ihtiyacımız var. Sonuç burada. [1]

[1] Çevirmenden not:
getline fonksiyonu, kitabın yazıldığı süreçte olmasa da artık bir GNU/POSIX eklentisi olduğundan standart kütüphaneye (**library**) dahil edildi. Bu yüzden kitabın orjinalinde olduğu gibi **getline** fonksiyonu aynı isimle tarif edilirse (**define**) derleme (**compile**) sırasında çakışma (**conflict**) hatası verecektir. gcc'yi -ansi gönderiyle (**flag**) (gcc -ansi test.c şeklinde) çalıştırmak sorunu çoğu sistemde çözse de MacOS ve BSD sistemlerde bilinmez bir nedenle işe yaramadığından fonksiyonun adını kaynak kodunda **getline2** olarak değiştirmeyi uygun görsek de metin içerisinde metnin orijinalliği esasında **getline** olarak kullanmaya devam etmeyi daha uygun gördük.

```
#include <stdio.h>
#define MAXLINE 1000 /* maksimum girdi (input) satır
   büyüklüğü */

int getline2(char line[], int maxline);
void copy(char to[], char from[]);

/* girdideki en uzun satırı bastır */
main ()
{
   int len;       /* tedavüldeki satırın uzunluğu */
   int max;       /* görülen en uzun satırın uzunluğu */
   char line[MAXLINE];  /* tedavüldeki satır */
   char longest[MAXLINE]; /* en uzun satır burada
      kaydediliyor */

   max = 0;
   while ((len = getline2(line, MAXLINE)) > 0)
      if (len > max) {
         max = len;
         copy(longest, line);
      }
   if (max > 0) /* satır vardı */
      printf("%s", longest);
   return 0;
}

/* getline2: s'deki satırı oku, uzunluğunu döndür (return)
   */
int getline2(char s[], int lim)
{
   int c, i;

   for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
      s[i] = c;
   if (c == '\n') {
      s[i] = c;
      ++i;
   }
   s[i] = '\0';
   return i;
}

/* copy: from'dan to'ya kopyala; yeterince büyük olduğunu
   farz et */
void copy(char to[], char from[])
{
   int i;

   i = 0;
   while ((to[i] = from[i]) != '\0')
```

```
            ++i;
    }
```

getline ve **copy** fonksiyonları programın başında deklare edildiler, tek bir dosya içinde bulunduklarını farz ediyoruz.

main ve **getline** bir argüman çifti ve döndürülen (**return**) bir değer (**value**) aracılığı ile iletişim kuruyorlar. **getline**'da argümanlar bu satır ile deklare ediliyor

```
int getline(char s[], int lim)
```

bu satır, ilk argümanın, s'in, bir karakter dizisi olduğunu (**array**), ve ikincisi ise, **lim**'in bir tamsayı (**integer**) olduğunu belirtiyor. Bir dizi (**array**) deklarasyonu yaparken dizinin (**array**) boyutunu belirlememizin sebebi, o miktarda depolama yerinin bir kenara ayrılmasını sağlamaktır. **s** dizisinin (**array**) boyutunu zaten hali hazırda **main** içerisinde belirlendiğinden **getline** içerisinde belirtmek gereksizdir. **getline** çağırana (**caller**) değer göndermek için **return**'ü kullanıyor, aynen fonksiyon **power**'ın yaptığı gibi. Bu satır aynı zamanda **getline**'ın **int** döndüreceğini (**return**) deklare ediyor; **int** varsayılan döndürme tipi (**return type**) olduğundan bu atlanabilirdi.

Bazı fonksiyonlar yararlı değerler (**value**) gönderir; copy gibi diğerleri ise sadece yaratacağı etkiler için kullanılır ve değer döndürmez. **copy**'nin dönüş tipi (**return type**) **void**, bu açık bir şekilde bir değer (**value**) döndürülmediğini gösteriyor.

getline '\0' karakterini (*null karakteri*, değeri (**value**) sıfır) yarattığı dizinin (**array**) sonuna, karakter öbeğinin (**string**) sonu olduğunu işaretlemek için koyuyor. Bu düzen aynı şekilde C dili tarafından da kullanıyor: bu gibi karakter öbeği (**string**) sabitleri (**constant**)

```
"hello\n"
```

bir C programında, karakterlerin bir dizisi (**array**) olarak saklanır ve bittiği en sonunda '\0' karakteri ile bittiği işaretlenen bir karakter öbeği (**string**) içerir.

h	e	l	l	o	\n	\0

printf'teki '**%s** format belirteci bu formda temsil edilecek olan eşleşen bir karakter öbeği (**string**) bekliyor. Aynı zamanda **copy** de girdi (**input**) argümanının '\0' ile bitmesi gerçeğine dayanıyor, ve bu karakteri çıktı (**output**) argümanına kopyalıyor. (Bütün bunlar '\0', karakterinin bir normal metnin bir parçası olmadığı anlamına geliyor.)

Bu kadar küçük bir programın dahi bazı tatsız dizayn problemleri göstermesi bahsetmeye değer. Örneğin, **main**'in limitinden daha büyük bir satır ile karşılaşması durumunda ne yapması gerekir ? **getline** güvenli bir şekilde çalışıyor, **getline** array dolduğunda toplamayı bırakıyor, yeni satır karakteri

görülmemişse bile. Uzunluğun ve döndürülen (**return**) son karakterin testi ile, **main** satırın uzunluğunun çok uzun olduğunu belirleyebilir ve ona göre istediği gibi bununla başa çıkabilir. Özlüğün esası için, bu sorunu görmezden geliyoruz.

getline'ın kullanıcısının girdinin (**input**) ne kadar olabileceğini bilmesinin hiçbir yolu yok, bu yüzden **getline** bir taşmayı (**overflow**) kontrol ediyor. Bir diğer yandan, **copy**'nin kullanıcısı karakter öbeklerinin (**string**) ne kadar büyük olduklarını zaten biliyor (veya öğrenebilir), bu yüzden bu kısımlara hata denetlemesi koymadık.

Egzersiz 1-16. En uzun satır programının main rutinini ne kadar uzun satır girilirse girilsin boyutunu bastırabilecek, ve metnin mümkün olduğu kadarını bastırabilecek şekilde tekrar gözden geçirin.

Egzersiz 1-17. 80 karakterden büyük bütün girdi (**input**) satırlarını bastıracak bir program yazınız.

Egzersiz 1-18. Sondaki boşlukları ve tab karakterlerini girdideki (**input**) satırlardan silen bir program yazınız, ayrıca tamami ile boş olan satırları da silen.

Egzersiz 1-19. reverse(s) isimli s karakter öbeğinin (**string**) karakterlerini ters çevirecek bir fonksiyon yazınız. Bunu, girdisini (**input**) satır satır ters çeviren bir program yazmak için kullanınız.

1.10 Harici Değişkenler (variables) ve Kapsam (scope)

line, **longest** vb. **main**'deki değişkenler (**variables**), **main**'e özel (**private**) veya yereldir (**local**). Çünkü onlar **main**'in içinde deklare edilmiştir, bu yüzden başka hiçbir fonksiyon onlara direkt olarak erişemez. Aynısı diğer fonksiyonlardaki tüm değişkenler (**variables**) için de geçerli; örneğin, **getline** fonksiyonundaki **i** ile **copy** fonksiyonundaki **i** tamami ile ilişkisiz. Bir fonksiyondaki her yerel (**local**) değişken (**variable**) ancak fonksiyon çağırıldığında (**call**) varoluyor, ve fonksiyon çalışmayı bitirdiğinde kayboluyor. Bu yüzden böyle değişkenler (**variables**), diğer dillerde de bunu takip eden bir terminoloji ile, *otomatik* değişkenler (**variables**) olarak biliniyor. Bundan böyle bu yerel (**local**) değişkenleri (**variables**) belirtmek için otomatik terimini kulla-

nacağız. (Bölüm 4'de çağırımlar (**call**) arası değerlerini (**value**) kaybetmeyen **static** depolama sınıfını (**class**) açıklayacağız.)

Çünkü otomatik değişkenler (**variables**) fonksiyon ilklendirmesi (**initialize**) gelip gidiyor, bir çağırıdan (**call**) diğerine değerlerini (**value**) saklamıyorlar, ve açıkça her girişte belirlenmeleri gerek. Eğer belirlenmezlerse çöp (**garbage**) içereceklerdir.

Otomatik değişkenlere (**variables**) bir alternatif olarak, bütün fonksiyonlara *harici* (**external**) olan bir değişken (**variable**) tarif edilebilir (**define**) bu, tüm fonksiyonlar tarafınca ismi ile erişilebilen bir değişken (**variable**) tarif eder (**define**). (Bu mekanizma Fortan'ın veya Pascal'ın en dıştaki bloğunda deklare edilen değişkenleri (**variables**) gibidir.) Çünkü harici (**external**) değişkenler (**variables**) global olarak erişilebilir, argüman listeleri yerine fonksiyonlar arası veri iletişimi için kullanılabilir. Ayrıca harici (**external**) değişkenler (**variables**) her fonksiyon çağırımında (**call**) ortaya çıkıp kaybolmaktan ziyade sonsuza kadar varoluşlarını koruduklarından dolayı, onları belirleyen fonksiyonlar döndükten (**return**) sonra bile değerlerini (**value**) koruyorlar.

Harici (**external**) bir değişken (**variable**), tam olarak bir kere, herhangi bir fonksiyonun dışında *tarif edilmeli* (**define**); bu onun için bir depolama ayıracak. Aynı zamanda değişken (**variable**) ona erişmek isteyen her fonksiyonun içinde *deklare* edilmeli; bu değişkenin (**variable**) tipini belirliyor.

```
#include <stdio.h>

#define MAXLINE 1000 /* maksimum girdi (input) satırı
   boyutu */

int max;           /* görülen maksimum uzunluk */
char line[MAXLINE]; /* tedavüldeki girdi (input) satırı */
char longest[MAXLINE]; /* en uzun satır burada
   kaydediliyor */

int getline2(void);
void copy(void);

/* en uzun girdi (input) satırı bastır; özelleştirilmiş
   versiyon */
main()
{
   int len;
   extern int max;
   extern char longest[];

   max = 0;
   while ((len = getline2()) > 0)
```

```
        if (len > max) {
           max = len;
           copy();
        }
     if (max > 0) /* satır vardı */
        printf("%s",longest);
     return 0;
}

/* getline: özelleştirilmiş versiyon */
int getline2(void)
{
     int c, i;
     extern char line[];

     for (i = 0; i < MAXLINE -1
        && (c = getchar()) != EOF && c != '\n'; ++i)
        line[i] = c;
     if (c == '\n') {
        line[i] = c;
        ++i;
     }
     line[i] = '\0';
     return i;
}

/* copy: özelleştirilmiş versiyon */
void copy(void)
{
     int i;
     extern char line[], longest[];

     i = 0;
     while ((longest[i] = line[i]) != '\0')
        ++i;
}
```

main'deki harici (**external**) değişkenler (**variables**) olan, **getline** ve **copy** yukarıdaki örneğin ilk satırlarında tarif edildi (**define**), bu onların tipini ve onlara ayrılacak olan depolamayı belirledi. Sözdizimsel (**syntactically**) olarak, harici (**external**) tarifler (**definition**) yerel (**local**) değişkenlerin (**variable**) tarifi (**definition**) ile aynı, fakat tüm fonksiyonların dışında olduklarından dolayı, değişkenler (**variables**) harici (**external**). Fonksiyon bir harici (**external**) değişkeni (**variable**) kullanmadan önce, değişkenin (**variable**) adı fonksiyonca bilinmelidir. Bunu yapmanın bir yolu bir fonksiyonun içinde bir **extern** deklarasyonu yazmaktır; eklenilen **extern** anahtar kelimesi dışında deklarasyon önceki yaptığımız deklarasyonların aynısıdır.

Belirli koşullar altında, **extern** deklarasyonu atlanabilir. Eğer harici (**external**) değişkenin (**variable**) tarifi (**definition**), herhangi bir fonksiyonda kullanılmadan önce, aynı kaynak dosyası içinde bulunuyorsa, fonksiyonun içinde **extern** deklarasyonu yapmaya gerek yoktur. **main'**deki **extern** deklarasyonları olan **getline** ve **copy** bu yüzden gereksiz. Aslında, çoğunlukla yapılan, bütün harici (**external**) değişkenlerin (**variable**) tariflerini (**definition**) kaynak dosyasının başına koymak ve bütün **extern** deklarasyonlarını atlamaktır.

Eğer program birden fazla kaynak dosyasında ise, ve değişken *dosya1*'de tarif edilip (**define**) *dosya2*'de ve *dosya3*'de kullanıldıysa, **extern** deklarasyonları *dosya2*'de ve *dosya3*'de değişkenin (**variable**) kullanışlarını bağlamak için gereklidir. Genellikle yapılandeğişkenlerin (**variables**) ve fonksiyonların bütün **extern** deklarasyonlarını ayrı bir dosyada toplamaktır. Tarihsel olarak *header* olarak bilinir bu dosya, ve her kaynak dosyasının başında #**include** ile dahil edilir. Uzantı **.h** header adları için gelenekseldir. Standart kütüphanenin (**library**) fonksiyonları, örneğin, <**stdio.h**> içinde deklare edilir. Bu konu Bölüm 4'de konuşuldu, ve kütüphanenin (**library**) kendisi Bölüm 7'de ve Ek B'de konuşuldu.

getline ve **copy**'nin özelleştirilmiş versiyonlarının hiç argümanı olmadığından, mantıken dosyanın başındaki fonksiyon prototipinin **getline()** ve **copy()** olması gerekiyor. Fakat eski C programları ile uyumluluk için standard boş listeyi eskistil deklarasyon olarak alıyor, ve bütün argüman listesi kontrolünü devre dışı bırakıyor; **void** açık bir ekilde boş liste için kullanılıyor. Bunu Bölüm 4'ün sonlarında konuşacağız.

tarif etme (**definition**) ve deklarasyon kelimelerini bu bölümde harici (**external**) değişkenleri (**variables**) anlatırken ne kadar dikkatli kullandığımızı farketmişsinizdir. "Tarif" (**definition**), değişkenin (**variable**) oluşturulduğu ve ona bir depolama atandığı yere karşılık gelirken, "deklarasyon" değişkenin (**variable**) doğasının belirlendiği ancak bir bir depolama ayrılmadığı yere karşılık gelir.

Bu arada, her şeyi **extern** değişkeni (**variable**) şeklinde yapmak gibi bir eğilim var, çünkü bu iletişimi oldukça basitleştiriyor-argüman listeleri oldukça kısa ve değişkenler (**variables**) onları ne zaman istersen oradalar. Fakat harici (**external**) değişkenler (**variable**) onları istemediğin zaman bile oradalar. Harici (**external**) değişkenlere çok güvenmek veri bağlantıları gözle görülür, açık olmayan programlara yol açtığından tehlikeli bir şekilde endişe verici-değişkenler (**variables**) beklenmedik hatta istenmedik şekillerde değişebilirler, ve program düzenlenmesi oldukça zor olacaktır. en-uzun-satır programının ikinci versiyonu ilk versiyonuna kıyasla bir kısmı belirttiğimiz nedenlerden dolayı, bir kısmı ise iki yararlı fonksiyonu manipüle ettikleri değişkenlerin (**variables**) isimlerine bağlayarak genelliğini yok ettiği için daha ast durumda.

Bu noktada, geleneksel olarak C'nin temeli olarak bahsedilen şeyi kavramış bulunmaktayız. Bu bir avuç blok inşâsı ile, belli bir boyuttaki programları yazmak mümkün, ve eğer bunu yapmak için yeterince ara verdiyseniz bu muhtemelen bu iyi bir fikirdi. Bu egzersizler bu bölümdeki programların daha kompleks hallerini istiyor.

Egzersiz 1-20. detab isimli girdideki (**input**) tab karakterlerini uygun sayıda boşluk karakterleriyle bir sonraki tab durağına kadar değiştiren bir program yazınız. Sabit bir tab durağı farz edin, her **n** bir basamak olsun. **n** bir değişken (**variable**) mi ? Yoksa bir sembolik parametre mi olmalı ?

Egzersiz 1-21. entab isimli boşluk karakterlerinin öbeklerini (**string**) aynı boşluk aralığını sağlamak için minimum sayıda tablerle ve boşluklarla değiştiren bir program yazınız. **detab**'deki aynı tab durağını kullanın. Bir tab veya tek bir boşluk tab durağına erişmede yeterli olduğunda hangisi tercih edilmeli ?

Egzersiz 1-22. Uzun girdi (**input**) satırlarını en sonuncu boş-olmayan (**non-blank**) karakterden önce, girdinin (**input**) **n'inci** basamağından önce ortaya çıkan karakterden önce iki yada daha fazla daha kısa satıra "katlayacak" bir program yazınız. Programınızın, çok uzun satırlarla karşılaştığında veya belirlenen basamaktan önce herhangi bir boşluk yada tab olmadığında zekice bir şey yaptığına emin olun.

Egzersiz 1-23. Bir C programındaki bütün yorumları silen bir program yazınız. Alıntılanmış karakte öbeklerini (**string**) ve karakter sabitlerini (**constants**) düzgünce hesaba katmayı unutmayın. C yorumları iç içe geçmez.

Egzersiz 1-24. Bir C programının kapatılmamış parantezler ve süslü parantezler gibi ilkel sözdizimi (**syntax**) hatalarını kontrol eden bir program yazınız. Tırnak işaretlerini , tekli ve çift olanları, kaçış sekanslarını (**sequence**), ve yorumları unutmayın. (Bu program bütün çoğunluğu ile yapılırsa zorlayıcı.)

Bölüm 2 | Tipler, Operatörler ve Açıklamalar (expression)

Değişkenler (**variables**) ve sabitler (**constants**) bir programda manipüle edilen basit veri objeleridir. Deklarasyonlar kullanılacak değişkenleri (**variables**) listeler ve onların hangi türde olduklarını ve istendiği zaman belki de başlangıç değerlerini belirtir. Operatörler onlara neler olacağını belirler. Açıklamalar (**expressions**) değişkenleri (**variables**) ve sabitleri (**constants**) yeni değerler (**values**) üretmek için kombine ederler. Objenin tipi objenin ne tür değerler alabileceğini ve üzerinde ne gibi operasyonlar gerçekleşebileceğini belirler. Bu inşa (**building**) blokları bu bölümün konularıdır.

ANSI standardı basit tip ve açıklamalara bir çok küçük eklemede bulundu. Artık **signed** ve **unsigned** bütün tamsayılardan (**integer**), ve **unsigned** sabitlerin (**constants**) ve hexademical karakter sabitlerinin **constants** için notasyonlarından oluşabiliyor. Kayan nokta (**floating-point**) operasyonları tek nicelikli (**single precision**) yapılabiliyor; genişletilmiş nicelik (**extended precision**) için ise **long double** tipi bulunmaktadır. Karakter öbeği (**string**) sabitleri (**constants**) derleme (**compile**) zamanında birleştirilebiliyor. Uzun süredir bir özellik olarak numaralandırmalar (**enumurations**) artık dilin bir parçası oldular. Objeler **const** olarak deklare edilebilirler bu onları daha sonra değiştirilmesini önler. Otomatik zorlamalar için kurallar arasından aritmetik tipler daha zengin tipleri ile başa çıkmak için arttırıldı.

2.1 Değişken (variable) Adları

Bölüm 1'de değişken (**variable**) ve sabitlerin (**constants**) isimlerinde bazı kısıtlamalar olduğunu belirtmedik. İsimler harflerden ve rakamlardan

oluşuyor; ilk karakter mutlaka bir harf olmalıdır. Altçizgi "_" bir harf olarak sayılır; bazen uzun değişken (**variable**) adlarını okumada yardımcı olabilir. Yine de çoğunlukla kütüphane (**library**) rutinleri sıklıkla böyle isimler kullandığından altçizgi ile başlayan değişken (**variable**) adları kullanmayın. Büyük ve küçük harfler birbirinden ayrılır, yani x ve X iki farklı isimdir. Geleneksel C pratiği değişken adları için küçük sembolik sabitler (**constants**) için büyük harflerin kullanılmasıdır.

En azından dahili bir değişkenin (**variable**) ilk 31 karakteri önemlidir. Fonksiyon adları ve harici değişkenler (**variables**) için, sayı 31'den düşük olabilir çünkü harici adları dilin kontrolünde olmayan birleştiriciler (**assembler**) veya yükleyiciler (**loaders**) tarafından kullanılıyor olabilir. Harici adlar için, standard ilk 6 karakterin ve tek bir durumun (harflerin küçük olması) benzersizliğini garanti ediyor. **if**, **else**, **int**, **float**, etc. gibi anahtar kelimeler rezervedir ve değişken adları olarak kullanılamaz. Küçük harfli olmaları gerekmektedir.

Değişkenin (**variable**) adını değişken (**variable**) ile ilintili seçmek mantıklı olacaktır, böylece tipografik olarak karışmaları daha zor olacaktır. Yerel değişkenler (**local**) için, özellikle döngü göstergeleri (**loop indices**) için kısa isimler, ve harici değişkenler (**external variables**) için daha uzun isimler kullanma eğilimindeyizdir.

2.2 Veri Tipleri ve Büyüklükleri

C'de sadece birkaç basit veri tipi bulunmakta:

```
char    tek bayt, sadece yerel karakter setinden
        tek bir karakter tutabilir.

int     tamsayı, genellikle konak makinedeki
        tam sayıların boyutunu yansıtır.

float   tek duyarlıklı kayan noktalı sayı.

double  çift duyarlıklı kayan noktalı sayı.
```

Ek olarak, bu basit tiplere ekleyebileceğiniz bazı niteleyiciler (**qualifiers**) mevcut. **short** ve **long** tamsayılara (**integers**):

```
short int sh;
long int counter;
```

int kelimesi böyle deklarasyonlarda atlanabilir, ve genellikle atlanır.

short ve **long**'un amacı, pratik olan yerlerde farklı büyüklüklerde tam sayılar sağlıyor olması; **int** normalde makinedeki belirlenmiş boyut kadar olacaktır. **short** genelde 16 bit, **long** genelde 32, ve **int** ya 16 ya 32 bit olabilir. Her derleyeci (**compiler**) kullandığı donanıma (**hardware**) göre boyut seçmekte özgürdür, tek kısıtlama **short**'ların ve **int**'lerin en az 16 bit, **long**'ların 32 bit olması ve **short**'ların **long**'lardan küçük olan **int**'lerden küçük olmasıdır.

signed ve **unsigned** niceleyicileri bazen **char**'lara ve her **int**'e uygulanabilir. **unsigned** sayılar her zaman pozitif veya 0'dır, Modulo 2 Arithmetic kurallarına uyarlar, **n** tipteki (**type**) bit sayısıdır. Örneğin, eğer **char**'lar 8 bit ise, **unsigned char** değişkenleri (**variables**) 0 ve 255 arası değerleri alabilir oysa ki **signed char**'lar -128 ile 127 arasında herhangi bir değer alabilirler. (bir ikinin tümleyeni makinesinde **two's complement**). Düz (**char**)'ların **signed** veya **unsigned** olmaları makineye-dayalıdır, fakat bastırılabilen (**print**) bütün **char**'lar pozitiftir.

long double tipi (**type**) genişletilmiş duyarlıklı (**extended-precision**) kayan noktalıyı (**float**) belirler. Tam sayılarda (**integers**) olduğu gibi, kayan noktalı (**float**) objelerin boyutlara emplemantasyonlar tarafından belirlenir; **float double** ve **long double** birbirinden farklı bir, iki veya üç boyutu temsil ediyor olabilir.

Standard başlıklar (**header**) <**limits.h**> ve <**float.h**> makinenin ve derleyicinin (**compiler**) özelliklerinin yanında bu üç boyut için sembolik sabitler (**constant**) içerir. Bunu Ek B'de konuştuk.

Egzersiz 2-1. char, **short**, **int** ve **long** gibi değişkenlerin (**variables**) hem **signed** hem **unsigned** için, genişliğini standard başlıklardan (**header**) uygun değerleri (**values**) bastıran direkt hesaplayan bir program yazınız. Eğer yazacağınız program hesaplayacak ise daha da zor olacaktır: bir çok kayar noktalı tipin (**type**) genişliğini belirleyiniz.

2.3 Sabitler (constants)