Programlamaya Giriş Ders Notları

H. Turgut Uyar

Şubat 2004

Önsöz

Bu notlar, İstanbul Teknik Üniversitesi'nde verilen "Introduction to Scientific and Engineering Computation" dersi için Bilgisayar Mühendisliği Bölümü öğretim görevlisi H. Turgut Uyar tarafından hazırlanmıştır. Yazarının açıkça belirtilmesi koşuluyla eğitim amaçlı kullanılabilir ve dağıtılabilir. Diğer her türlü kullanımı için yazarından izin alınmalıdır.

Sürüm: 0.99.5
Son düzenleme tarihi: 5 Şubat 2004
Web sayfası: http://www.ce.itu.edu.tr/Members/uyar/c

Bu dersin amacı, öğrenciye yalnızca programlamayı ya da C dilini öğretmek değil, aynı zamanda temel sayısal yöntemleri ve sıralama algoritmaları gibi sıkça kullanılan programlama tekniklerini de göstermektir. Bu amaçla uygulamalarda daha çok sayısal yöntemler üzerine örnekler seçilmeye çalışılmıştır. Bu örneklerin çoğu Prof. Dr. Nadir Yücel'in "Sayısal Analiz Algoritmaları" kitabından uyarlanmıştır.

Notların hazırlanmasında yeni standartlara uyumu kolaylaştırması açısından C++ dilinin getirdiği bazı yeniliklerden yararlanılmıştır. Metin içinde yeri geldikçe, C++ dilinde geçerli olup C dilinin izin vermediği özellikler belirtilmiştir.

Programların ekran çıktılarında kullanıcının yazdığı değerler italik olarak işaretlenmiştir.

Örnek programlar ve ders için hazırlanan sunumlar (İngilizce) yukarıda belirtilen web sayfasında bulunabilir.

İçindekiler

Bölüm 1

Giriş

Bir problemi çözmek üzere bir bilgisayar programı yazarken iki temel soruna çözüm getirmek gerekir:

Problemin nasıl temsil edileceği.

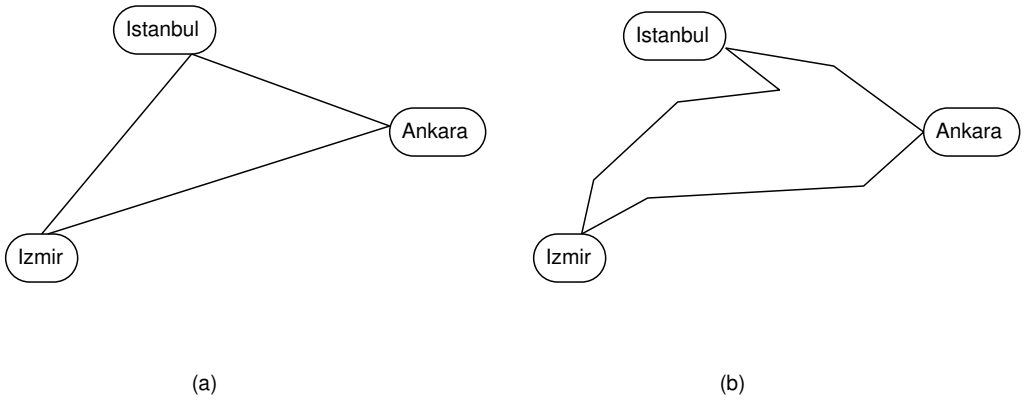
Bilgisayarlar temelde sayılar üzerinde işlem yapmak üzere tasarlanmış makinalardır; dolayısıyla tanıdıkları varlık yelpazesi fazla geniş değildir. Örneğin Türkiye'deki karayollarıyla ilgili bir program yazılacaksa şehirlerin ve aralarındaki yolların bir şekilde temsil edilmesi gerekir çünkü bilgisayarın "şehir", "yol" gibi kavramları yoktur. Bir şehri temsil etmek için şehrin adı, enlemi ve boylamı kullanılabilir. Bir yolun temsili içinse farklı seçenekler düşünülebilir:

- Yolun düz olduğunu varsayarak bir doğruyla göstermek (Şekil 1.1a). Bu durumda yalnızca başlangıç ve bitiş şehirlerini (yani enlem, boylam ve isimlerini) bilmek yeterli olacaktır. Bu yöntem işlemleri çok basitleştirmekle birlikte gerçek durumdan büyük ölçüde sapmaya neden olur.
- Yolu ucuca eklenmiş doğru parçalarıyla göstermek (Şekil 1.1b). Şekilde İstanbul-Ankara yolu iki, İstanbul-İzmir yolu dört, Ankara-İzmir yolu üç doğru parçasından oluşmaktadır. Ancak bu yöntem daha zordur ve yolu ne kadar ayrıntılı temsil etmek isterseniz o kadar fazla doğru parçası kullanmanız gerekir.

Problemi oluşturan varlıkların nasıl temsil edileceklerinin belirlenmesi, problem için bir *model* oluşturulması anlamına gelir. Problemin çözümünde ilk ve en önemli adım doğru ve yeterli ayrıntıda bir model oluşturmaktır. Yukarıdaki örnekten de görülebileceği gibi, çözeceğiniz problemi nasıl modellediğiniz son derece önemlidir, çünkü çözdüğünüz şey problemin kendisi değil, sizin o problemi temsil etmek üzere tasarladığınız modeldir. Modeliniz problemden uzaksa siz ne kadar iyi bir çözüm uygularsanız uygulayın elde edeceğiniz sonuç anlamsız olur.

Diğer yandan, modelin ne kadar ayrıntılı olması gerektiği, çözmek istediğiniz problemin gereksinimleriyle belirlenir. Gerekenden daha ayrıntılı bir model hazırlarsanız belki "daha iyi" bir sonuç elde edersiniz ama o daha iyi sonuca ulaşmak için harcamanız gerekecek ek çaba ya da çözümüzün gerçekleme maliyeti aradaki farka değmeyebilir. Yine yukarıdaki

örnekle sürdürürsek, "İstanbul-Ankara karayolunun 274. kilometresi" gibi bilgilerden söz edecekseniz yolu doğru parçalarıyla modellemeniz gerekecektir çünkü gerçek yaşamda bu nokta büyük olasılıkla o iki şehri birleştiren çizginin üzerinde bir yere düşmeyecektir. Ancak yolların yalnızca uzunluklarıyla ilgileniyorsanız doğrularla modellemeniz yeterlidir; bu durumda da doğru parçalarıyla modellemek yolun uzunluğunun bulunmasında zorluk çıkaracaktır.



Şekil 1.1: Karayollarının gösterilmesi.

Çözümün nasıl ifade edileceği.

Bir problem için düşündüğünüz çözümü başka birine (insan ya da bilgisayar) anlatabilmeniz gerekir. Adım adım hangi işlemlerin yapılacağının açıklanması şeklinde anlatılan bir çözüme *algoritma* adı verilir. Algoritmalara günlük yaşamdan sıkça verilen bir örnek yemek tarifleridir. Aşağıda, bezelyeli Jamaika pilavı yapmak için İnternet'ten alınmış bir tarif algoritma biçiminde yazılmıştır:

1. 1 1/2 kutu bezelyeyi 4-5 fincan suya koy.
2. 1/4 kutu hindistan cevizi sütü, 1 tutam kekik ve ağız tadına göre tuz ve biber ekle.
3. Bezelyeler yumuşayıncaya kadar haşla.
4. Bir soğanın dibini ez ve 2 fincan pirinç, 1/4 kutu hindistan cevizi sütü ve 2 tutam kekik ile birlikte suya ekle.
5. Pirincin üstünden 2 cm kadar su kalacak şekilde fazla suyu al.
6. 5 dakika kaynat.
7. Pirinç yumuşayıncaya kadar pişirmeye devam et.

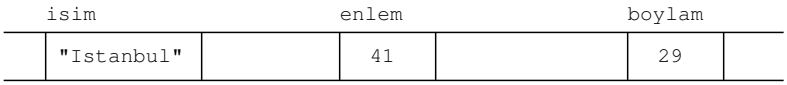
Algoritmanın bir bilgisayar tarafından yürütülebilmesi için iki önemli özellik sağlanmalıdır:

- Her adımda ne yapılacağı açık olarak belli olmalı, hiçbir şekilde yorum gerektirmemelidir. Yukarıdaki örnek bu bakımdan bir algoritma sayılamaz çünkü pek çok adımda ne yapılacağı yoruma bırakılmıştır. Sözgelimi, "4-5 bardak", "ağız tadına göre", "bezelyeler yumuşayıncaya kadar" gibi deyimler yeterince kesin değildir.
- Sonlu sayıda adımda ya çözüm bulunmalı ya da bulunamadığı bildirilmelidir.

1.1 Veriler

Modelinizde kullandığınız büyüklükler programınızın verilerini oluşturur. Her büyüklük programda bir *değişken* ile temsil edilir. Değişken aslında o büyüklüğe verilmiş simgesel bir isimden başka bir şey değildir. Değişkenler, programın işleyişi sırasında *değerler* alırlar. Değişkenlerin bellekte tutuldukları gözönüne alınırsa, değişken bir bellek gözünün adı, değer ise bu gözün içeriğidir.

Örneğin, bir şehri modellemek için üç büyüklük öngörmüştük: isim, enlem ve boylam. Her büyüklüğe bir değişken karşı düşürmemiz gerektiğinden isim bilgisini `isim`, enlem bilgisini `enlem` ve boylam bilgisini `boylam` değişkenleriyle temsil ettiğimizi varsayalım. Bu durumda temsil ettiğimiz şehre göre bu değişkenlere uygun değerler vermemiz gerekir. Sözgelimi, İstanbul şehrini temsil etmek için `isim` değişkenine "İstanbul", `enlem` değişkenine 41, `boylam` değişkenine 29 değerleri verilmelidir (Şekil 1.2).[1]



Şekil 1.2: Şehri temsil eden değişkenler ve örnek değerler.

Bir değişkene bir değer verilmesi işlemine *atama* denir ve sola bakan bir ok işaretiyle gösterilir:[2] `enlem` ← `41`. Atama işaretinin sol tarafına bir değişken adı, sağ tarafına bir *deyim* yazılır. Deyim, bir değer üreten bir hesaplama olarak tanımlanabilir. Bir deyim, tek bir değer ya da bir değişken olabileceği gibi, bunların işlemler ile çeşitli şekillerde bağlanmalarından da oluşabilir:

- `41`: yalnızca bir değer
- `boylam`: yalnızca bir değişken
- `4 * boylam`: bir değer ile bir değişkenin çarpma işlemiyle bağlanması
- `enlem + boylam`: iki değişkenin toplama işlemiyle bağlanması

Atama işlemi bir matematiksel eşitlik değildir. Örneğin `41` ← `enlem` atamasının bir anlamı yoktur. Benzer şekilde, `i` ← `i + 1` ataması, `i` değişkeninin değerinin o anki değerine göre bir artırılması demektir; yani değeri bu işlemden önce 5 ise, işlemden sonra 6 olur. Oysa, bu bir eşitlik olsaydı yanlış olurdu ($0 = 1$).

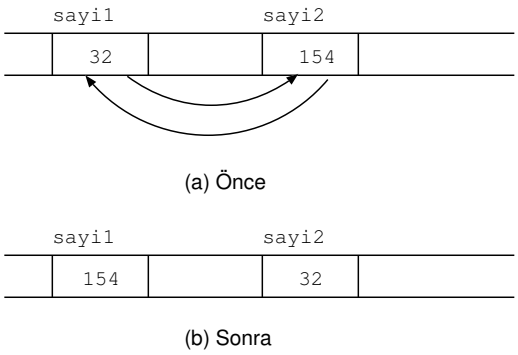
Örnek. Takas

Programlarda sıkça gerekebilen işlemlerden biri, iki değişkenin değerlerini karşılıklı değiştirmektir. Sözgelimi, `sayı1` değişkeninin değeri 32 ve `sayı2` değişkeninin değeri 154 ise bu işlemden sonra `sayı1` değişkeninin 154, `sayı2` değişkeninin de 32 değerini alması istenir (Şekil 1.3).

Takas işlemini yapmak üzere şu atamaların kullanıldığını düşünelim:

[1] Burada kuzey enlemlerinin ve doğu boylamlarının pozitif sayılarla gösterildiği varsayılmıştır. Güney enlemleri ve batı boylamları negatif sayılarla gösterilirse örneğin New York şehrinden söz edildiğinde `isim` değişkeni "New York", `boylam` değişkeni -74, `enlem` değişkeni 40 değerini almalıdır.

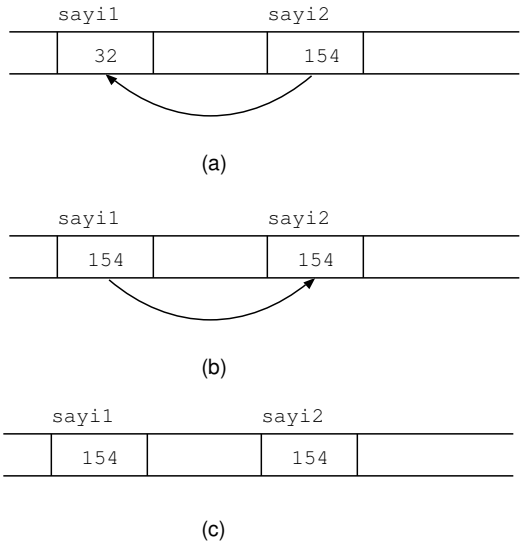
[2] Bu işlem için programlama dilleri genelde = ya da := işaretini kullanırlar.



Şekil 1.3: Takas işlemi.

```
sayı1 ← sayı2
sayı2 ← sayı1
```

Birinci atama işleminden (Şekil 1.4a) sonra `sayı1` değişkeni 154 değerini alır, `sayı2` değişkeninde ise bir değişiklik olmaz. İkinci atama işlemi (Şekil 1.4b) ise `sayı2` değişkenine `sayı1` değişkeninin o anki değerini atadığından `sayı2` değişkenine yine 154 değeri atanır ve sonuçta her iki değişken de 154 değerini almış olur (Şekil 1.4c).

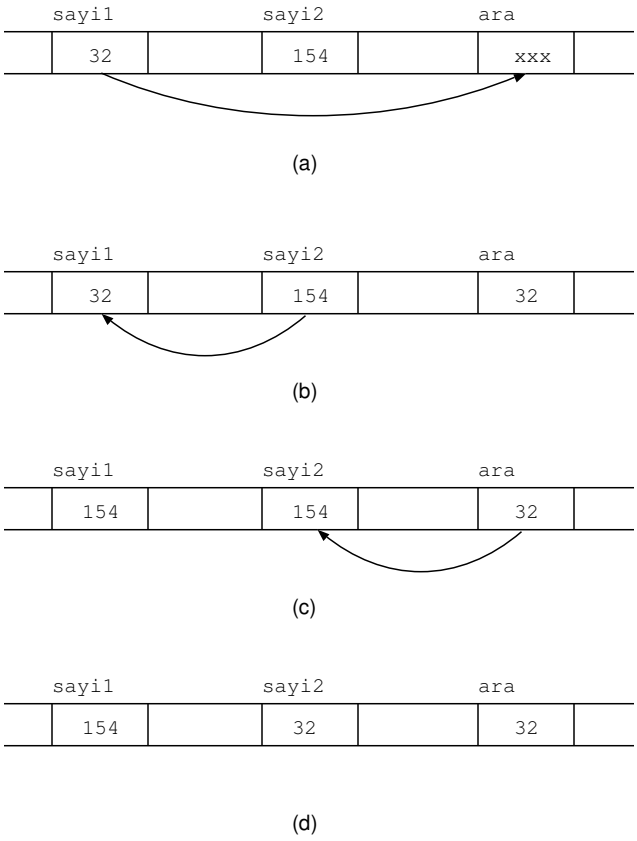


Şekil 1.4: Takas işlemi çözümü (hatalı).

İki atama işleminin yetersiz olduğu görülmektedir. Yapılması gereken, değişkenlerin değerlerini yitirmemek için, bir değişkenin değerini başka bir değişkende yedeklemektir:

```
ara ← sayı1
sayı1 ← sayı2
sayı2 ← ara
```

Birinci atama (Şekil 1.5a) sonucunda `ara` değişkeni 32 değerini alır. İkinci atamada (Şekil 1.5b) `sayı1` değişkenine 154, üçüncü atama (Şekil 1.5c) sonucunda da `sayı2` değişkenine 32 değeri atanır (Şekil 1.5d). Bu atamalardan sonra `ara` değişkeninin değerinin ne olduğunun bir önemi yoktur.



Şekil 1.5: Takas işlemi çözümü (doğru).

Değişkenlerin, temsil ettikleri varlığa göre, bir *tip*leri vardır. Bazı programlama yaklaşımlarında programcının değişkenin hangi veri tipinden olduğunu belirtmesi şart koşulurken, bazılarında değişkenin tipi, içinde kullanıldığı bağlamdan kestirilmeye çalışılır.

1.1.1 Taban Tipler

En sık kullanılan değişken tipi sayılardır. Örnekteki `enlem` ve `boylam` değişkenleri duruma göre birer tamsayı ya da kesirli sayı olarak seçilebilir. Çoğu programda gerekecek temel veri tipleri şunlardır:

tamsayı Bir insanın doğum yılı, soyadındaki harf sayısı, boyunun santimetre cinsinden uzunluğu, bir işlemin kaç kere yapıldığını sayan sayaç gibi bilgiler.

kesirli sayı Bir insanın boyunun metre cinsinden uzunluğu, iki sınavdan alınan notların ortalaması, bir sayının karekökü gibi bilgiler.

mantıksal Bir öğrencinin bir dersten başarılı olup olmadığı, bir insanın onsekiz yaşından büyük olup olmadığı, kullanıcının bastığı tuşun bir rakam tuşu olup olmadığı gibi bilgiler. Bu tipten değişkenler `Doğru` ya da `Yanlış` değerini alabilirler.[3]

simge Birinin adının baş harfi, programın çalışması sırasında "`Devam etmek istiyor musunuz (E/H)?`" sorusuna karşılık hangi tuşa bastığı, bir tarih bilgisinde gün, ay ve yıl arasına hangi işaretin konacağı (nokta, tire, bölü, vs.) gibi bilgiler. Simgeler çoğunlukla tek tırnak işaretleri içinde yazılırlar: 'E', '?', '4' gibi. Burada ayrımı yapılması gereken önemli bir nokta, rakamlar ile rakamları gösteren işaretleri birbirine karıştırmamaktır. Örneğin 5 rakamı ile '5' simgesi farklı büyüklüklerdir (bkz. Ek A).

katar Bir insanın adı, doğduğu şehir, bir kitabın ISBN numarası gibi bilgiler katarlarla temsil edilmeye uygundur. Katarlar çoğunlukla çift tırnak içinde yazılırlar: "Dennis Ritchie", "0-13-110362-8" gibi. Bir büyüklük bütünüyle rakamlardan oluşsa bile bazı durumlarda sayı yerine katarla göstermek daha uygun olabilir. Bir değişkenin ancak üzerinde aritmetik bir işlem yapılacaksa sayı tipinden olması anlamlı olur. Örneğin, İstanbul Teknik Üniversitesi'nde öğrenci numaraları dokuz haneli tamsayılardır ancak bunları tamsayı olarak temsil etmenin bir anlamı yoktur çünkü öğrenci numaraları üzerinde aritmetik işlem yapmak gerekmeyecektir (iki öğrencinin numaralarını toplamak ya da çarpmak gibi).

1.1.2 Kayıtlar

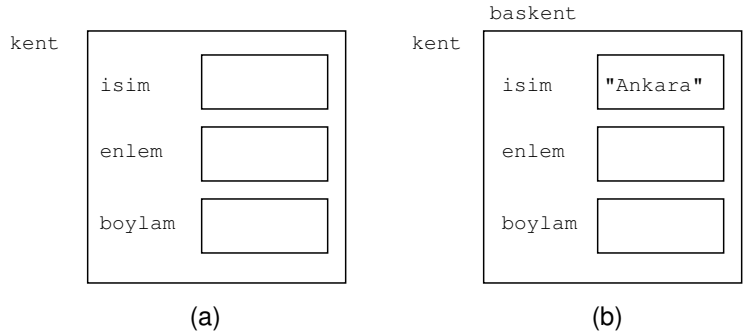
Birden fazla büyüklüğü ortak bir tip altında toplarlar. Burada gruplanan büyüklükler aynı tipten ya da farklı tiplerden olabilir. Örneğin bir şehri temsil etmek üzere kullanılan üç büyüklük (şehrin ismi, enlemi ve boylamı) birleştirilerek şehirleri gösterecek bir `kent` veri tipi oluşturulabilir. Bu veri tipinin biri bir katar (şehir ismi), diğer ikisi de birer kesirli sayı olan (enlem ve boylam) üç alanı olacaktır (Şekil 1.6a). Bu tipten diyelim bir `başkent` değişkeni tanımlandığında, değişkenin alanlarına erişmek için "`başkent` kentinin ismi, `başkent` kentinin enlemi" gibi deyişler kullanılmalıdır. Bu amaçla çoğunlukla noktalı gösterilimden yararlanılır; örneğin `başkent` değişkeninin `isim` alanına "Ankara" değerini atamak için `başkent.isim` ← `''Ankara''` yazılır (Şekil 1.6b).

Kayıtlar alan olarak başka kayıtları da içerebilirler. Örneğin, bir yolu temsil için yolun kodu (katar), uzunluğu (tamsayı) ve başlangıç ve bitiş kentleri bilgileri kullanılabilir (Şekil 1.7). Erişim, tek düzeyli kayıtlara benzer şekilde yapılır (`tem.son_kent.isim` ← `''Ankara''` ve `tem.kod` ← `''E-6''` gibi).

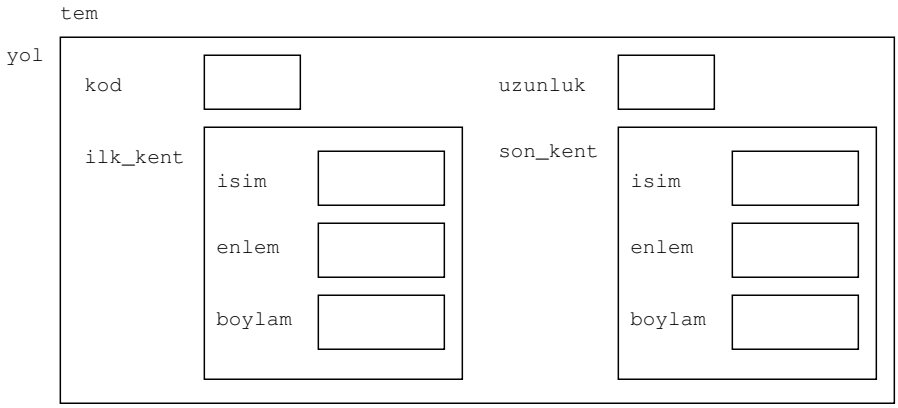
1.1.3 Diziler

Aynı tipten varlıklardan oluşan bir grubu tek bir çatı altına toplamak için kullanılırlar. Bir dizinin bütünü tek bir değişken olarak değerlendirilir. Örneğin 50 öğrencili bir sınıfta bir sınavdan alınan notlar işleneceğinde, her bir öğrencinin notunu göstermek üzere `not1`, `not2`, $\cdots$, `not50` gibi ayrı ayrı 50 tamsayı değişken tanımlanacağına, `notlar` adında 50 elemanlı

[3] Bu değerler örneklerde `D` ve `Y` şeklinde kısaltılacaktır.

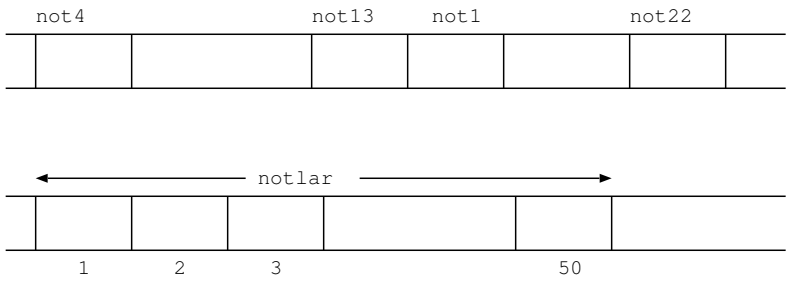


Şekil 1.6: Kayıt tipinden değişken örneği.



Şekil 1.7: İçiçe kayıtlar tipinden değişken örneği.

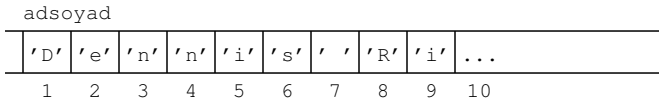
bir tamsayı dizisi tanımlanabilir. Ayrı ayrı değişkenler bellekte dağınık bir yapı oluştururken dizinin elemanları bellekte birbirini izleyen gözlere yerleştirilir (Şekil 1.8).



Şekil 1.8: Dizi tipinden değişken örneği.

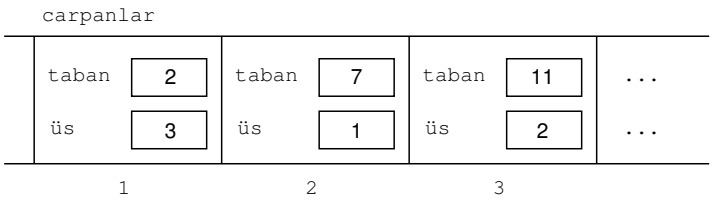
Elemanlar üzerinde işlem yapmak için dizinin kaçıncı elemanından söz ettiğinizi söylemeniz gerekir. Sözgelimi, 22. öğrencinin aldığı not 95 ise yapılacak atama $\texttt{notlar}_{22} \leftarrow \texttt{95}$ şeklinde gösterilir[4].

Katarlar çoğu programlama dilinde simge dizileri olarak görülürler; yani bir katar her bir elemanı bir simge olan bir dizidir. Sözgelimi, birinin adı ve soyadı `adsoyad` isimli bir değişkende tutulacaksa, bu değişken simge dizisi tipinden olacaktır (Şekil 1.9). Örnekte $\texttt{adsoyad}_1$ büyüklüğünün değeri 'D', $\texttt{adsoyad}_7$ büyüklüğünün değeri ' ' (boşluk) simgesidir.



Şekil 1.9: Katar tipinden değişken örneği.

Dizilerle kayıtlar birlikte de kullanılabilir. Sözgelimi, bir sayının asal çarpanlarını temsil etmek üzere bir dizi kullanabiliriz ($6776 = 2^3 * 7^1 * 11^2$). Dizinin her bir elemanı bir asal çarpanı gösterir, her bir asal çarpan da bir taban ve bir üs değerinden oluştuğu için bir kayıtla temsil edilir (Şekil 1.10). Elemanlara erişim daha önce belirtilen kurallarda görüldüğü gibi olacaktır: `ç` $\texttt{arpanlar}_2\texttt{.taban} \leftarrow \texttt{7}$



Şekil 1.10: Kayıt dizisi tipinden değişken örneği.

[4]Bu işlem için programlama dillerinde köşeli ayraçlar kullanılır: `notlar[22]` gibi.

Örneklerden de görülebileceği gibi, bir dizinin bütünüyle temsil edilmesi için dizinin eleman değerlerinin yanısıra kaç elemanı olduğu bilgisinin de bir şekilde tutulması gerekir.

1.2 Algoritmalar

Algoritmaları göstermek için sıkça kullanılan yöntemlerden biri akış çizenekleridir. Akış çizeneklerinde şu simgeler kullanılır:

- Kutu: Bir işlemi gösterir. Kutunun içine işlemi anlatan bir komut yazılır.
- Ok: Akış yönünü belirtir. Algoritmanın bir sonraki adımının hangisi olduğunu gösterir.
- Eşkenar dörtgen: Karar noktalarını gösterir. İçine yazılan sorunun yanıtının doğru ya da yanlış olmasına göre farklı bir yöne gidilmesini sağlar.
- Paralelkenar: Giriş/çıkış işlemlerini gösterir.

Algoritmanın tamamı belirtilmişse akış çizeneği köşeleri yuvarlatılmış kutular içinde bulunan "başla" komutuyla başlar ve "dur" komutuyla biter. Algoritmanın tamamı değil, yalnızca ilgilenilen bir parçası belirtilmek isteniyorsa çizenek boş bir yuvarlak ile başlar ve boş bir yuvarlak ile sona erer. Akış çizeneğinin büyümesi ve topluca görülmesinin zorlaşması durumunda akış çizeneği parçalarının başındaki ve sonundaki yuvarlakların içine etiketler yazarak hangi parçanın hangi parçaya nereden bağlandığı belirtilebilir.

Algoritmaların örnek değerler üzerinde işleyişlerini daha kolay izleyebilmek amacıyla tablolar kullanılabilir. Tablonun her bir satırı algoritmanın bir adımına karşı düşer ve o adımda çeşitli değişkenlerin ya da deyimlerin aldıkları değerlerin görülmesini sağlar; yani değişkenler ve deyimler tablonun sütunlarını oluşturur.

Örnek. En Büyük Eleman Bulma

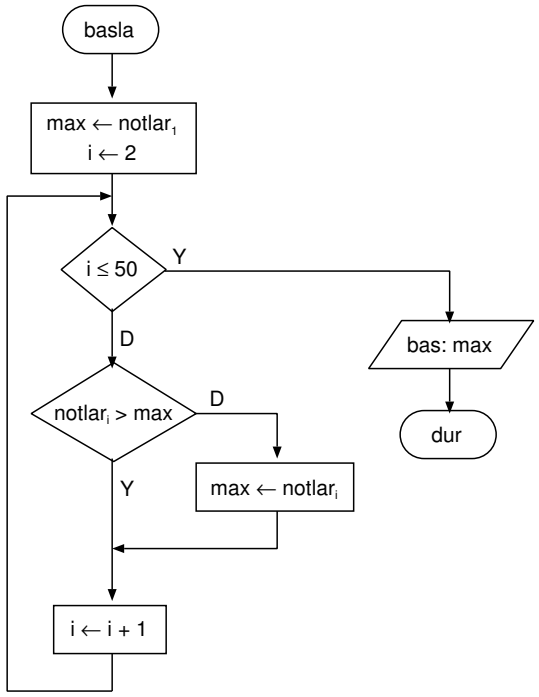
Bir dizinin en büyük elemanını bulma algoritmasını, 50 öğrencili bir sınıfta bir sınavdan alınan en yüksek notun bulunması örneği üzerinde inceleyelim. Bu işi yapacak bir algoritma şöyle yazılabilir:

1. Dizideki ilk notu en yüksek not olarak seç ve sırayı ikinci öğrenciye geçir.
2. Sırada öğrenci varsa 3. adıma, yoksa 5. adıma git.
3. Sıradaki öğrencinin notu şu ana kadarki en yüksek nottan büyükse bu yeni notu en yüksek not olarak seç.
4. Sırayı bir sonraki öğrenciye geçir ve 2. adıma dön.
5. En yüksek notu bildir.

Daha çok gündelik dil kullanılarak yazılmış bu algoritmayı biçimsel olarak ifade edebilmek için öğrencilerin notlarını 50 elemanlı bir tamsayı dizisi (bu değişkene `notlar` adını verelim), o ana kadar bulunmuş en yüksek notu bir tamsayı (`max` değişkeni diyelim) ile ve sıradaki öğrencinin kaçıncı öğrenci olduğunu tutmak için bir sayaç (`i` değişkeni) tanımlarsak:

1. $\texttt{max} \leftarrow \texttt{notlar}_1$, $\texttt{i} \leftarrow \texttt{2}$
2. $\texttt{i} \leq \texttt{50}$ ise 3. adıma, değilse 5. adıma git.
3. $\texttt{notlar}_i > \texttt{max}$ ise $\texttt{max} \leftarrow \texttt{notlar}_i$
4. $\texttt{i} \leftarrow \texttt{i + 1}$ ve 2. adıma dön.
5. en yüksek not: `max`

Yukarıda verilen algoritma örneğinin akış çizeneği Şekil 1.11'de görüldüğü gibidir. 50 eleman yerine 6 elemanlı bir dizide en büyük elemanın bulunması algoritmasının işleyişi Tablo 1.1'de verilmiştir (notların sırasıyla 43, 74, 65, 58, 82, 37 oldukları varsayılmıştır, sürme koşulu $\texttt{i} \leq \texttt{6}$ şeklinde değişmelidir).



Şekil 1.11: En büyük eleman bulma algoritmasının akış çizeneği.

Örnek. Sayı Tahmin Etme

Arkadaşınızın önceden aranızda karşılaştırdığınız iki sınır arasında bir tamsayı tuttuğunu ve sizin bu sayıyı bulmaya çalıştığınızı varsayın. Siz bir sayı söylediğinizde arkadaşınız, tuttuğu

max	i	$i \leq 6$	$notlar_i$ > max
43	2	D (2 < 6)	D (74 > 43)
74	3	D (3 < 6)	Y (65 < 74)
	4	D (4 < 6)	Y (58 < 74)
	5	D (5 < 6)	D (82 > 74)
82	6	D (6 = 6)	Y (37 < 82)
	7	Y (7 > 6)	

Tablo 1.1: En büyük eleman bulma algoritmasının örnek değerlerle işleyişi.

sayı sizin söylediğinizden büyükse "büyük", küçükse "küçük" diyecek, doğru sayıyı söylediğinizde oyun sona erecektir. Bu oyunu oynarken nasıl bir algoritma kullanırsınız?

Kararlaştırdığınız sınır değerlerinden küçük olanını `taban`, büyük olanını `tavan` isimli birer değişken ile gösterelim. Ayrıca, biri arkadaşınızın tuttuğu sayıyı temsil edecek (`tutulan` isimli), diğeriyse sizin söylediğiniz sayıyı temsil edecek (`söylenen` isimli) iki tamsayı değişken daha kullanalım.

Algoritma 1. Alt sınırdan başla, bulana kadar birer artırarak ilerle.

1. `söylenen` ← `taban`
2. `söylenen = tutulan` ise buldun, dur
3. `söylenen` ← `söylenen + 1` ve 2. adıma dön

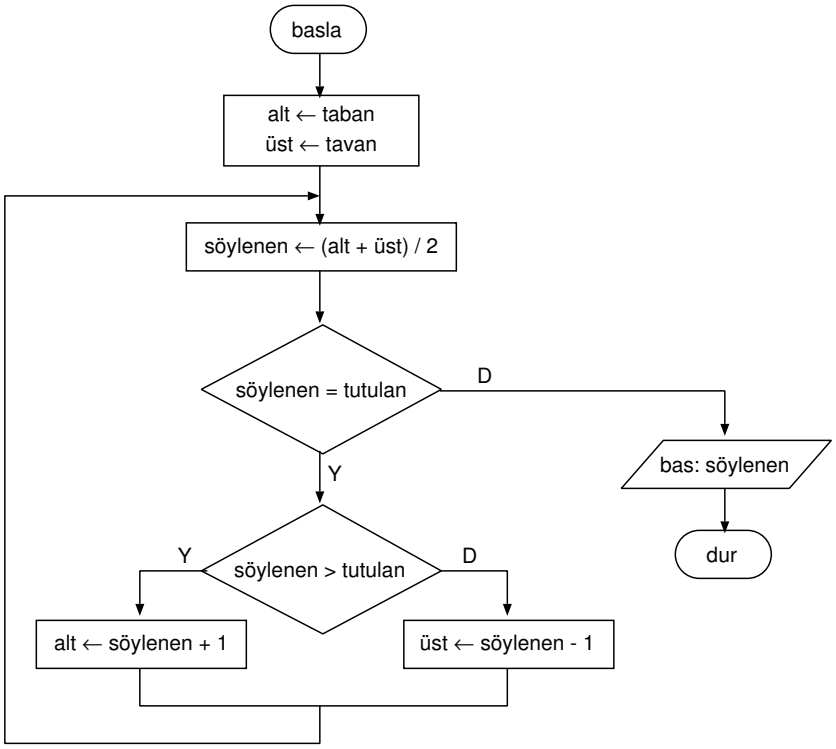
Algoritma 2. Deneme aralığının ortasındaki sayıyı söyle. "Büyük" derse deneme aralığını şu anki aralığın üst kısmına, "küçük" derse alt kısmına daralt. Bu algoritmayı gerçeklemek için o anki deneme aralığının alt ve üst sınırlarını gösterecek iki yeni değişkene (`alt` ve `üst` diyelim) gerek duyulur.

1. `alt` ← `taban, üst` ← `tavan`
2. `söylenen` ← `(alt + üst) / 2`
3. `söylenen = tutulan` ise buldun, dur
4. `söylenen > tutulan` ise `üst` ← `söylenen - 1`, değilse `alt` ← `söylenen + 1` ve 2. adıma dön.

Bu algoritmanın akış çizeneği Şekil 1.12'de verilmiştir. Alt sınırın 1, üst sınırın 63 olduğunu ve arkadaşınızın 19 sayısını tuttuğunu varsayarsak, her turda değişkenlerin aldıkları değerler Tablo 1.2'de görüldüğü gibi olacaktır.

1.2.1 Algoritmaların Karşılaştırılması

Yukarıdaki örneklerde görüldüğü gibi, çoğu zaman bir problemi çözmenin birden fazla yolu vardır. Aynı problemi çözen iki algoritmadan hangisinin daha iyi olduğuna karar vermek üzere algoritmalar iki özelliklerine göre karşılaştırılırlar:



Şekil 1.12: Sayı tahmin etme algoritmasının akış çizeneği.

alt	üst	söylenen	söylenen = tutulan
1	63	32	Y (32 > 19)
	31	16	Y (16 < 19)
17		24	Y (24 > 19)
	23	20	Y (20 > 19)
	19	18	Y (18 < 19)
19		19	D (19 = 19)

Tablo 1.2: Sayı tahmin etme algoritmasının örnek değerlerle işleyişi.

1. Hızları: Hangi algoritma çözümü daha çabuk buluyor? Bu sorunun yanıtı da iki durum için incelenir:

   (a) en kötü durumda

   (b) ortalama durumda

   Sayı bulma için verilen yukarıdaki iki algoritmayı bu bakımdan karşılaştırırsak, birinci algoritmanın örnek değerlerle sayıyı en kötü durumda 63, ortalama durumda 32 denemede bulacağı görülür. Oysa ikinci algoritma sayıyı en kötü durumda 6, ortalama durumda 5.09 denemede bulur.

2. Harcadıkları yer: Hangi algoritma bellekte daha fazla yer kullanıyor?

   Sayı bulma algoritmalarında gördüğümüz gibi, ikinci algoritma birinci algoritmanın kullandıklarına ek olarak iki değişken daha gerektirmektedir.

Günümüzde bilgisayarların kapasiteleri eskiye oranla çok yükselmiş olduğu için harcanılan yer ölçütünün önemi göreli olarak azalmıştır. Yine de projenin boyutuna ve çalışılan ortamın yeteneklerine göre algoritma geliştirirken hızın yanısıra harcanacak yerin de gözönüne alınması gerekebilir.

Algoritmaların hızlarının ve harcadıkları yerlerin incelenmesi "algoritma analizi" dalının konusudur. Bu dalda geliştirilmiş bulunan "karmaşıklık kuramı", benzer problemlerin çözümü için önerilen algoritmaların karşılaştırılmasında başvurulan en önemli araçtır.

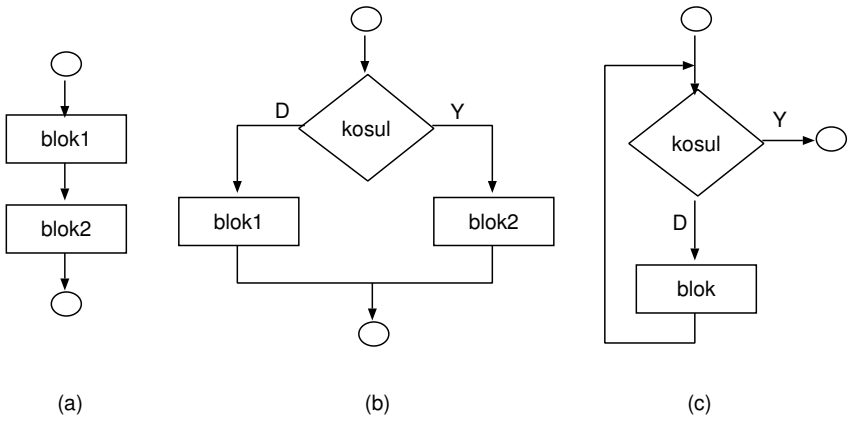
1.3 Blok Yapılı Programlama

Blok yapılı programlamanın temelinde *blok* kavramı yatar. Blok, birbiriyle ilişkili komutların oluşturduğu gruptur. Her algoritma birbirlerine çeşitli şekillerde bağlanmış bloklardan oluşur. Blokları bağlamanın üç yolu vardır:

Sıra Blokların yukarıdan aşağıya doğru yazıldıkları sırayla yürütülürler (Şekil 1.13a). Sıra yapısı, komutların yazılış sıralarının önemli olduğunu vurgular. Bölüm 1.1'de görülen takas örneği için verilen doğru çözümde yapılan üç atama işleminin sıraları değiştirilirse varılacak sonuçlar yanlış olabilir.

Seçim Bir koşulun doğru olup olmamasına göre farklı bir bloğun yürütülmesidir. Yani koşul doğruysa bir blok, yanlışsa başka bir blok yürütülür (Şekil 1.13b). İki bloktan herhangi biri boş olabilir, yani "koşul doğruysa şu bloğu yürüt, yanlışsa hiçbir şey yapma" şeklinde bir yapı kurulabilir.

Yineleme Belirli bir koşul sağlandığı sürece (ya da sağlanana kadar) bir blok yinelenebilir. Akış çizeneğinden (Şekil 1.13c) görülebileceği gibi, bu yapıdan çıkılabilmesi için bloğun koşulu değiştiren bir komut içermesi gerekir, aksi durumda koşul başlangıçta doğruysa hep doğru olacağından yapıdan çıkılamayacaktır. Yine akış çizeneğinden görülebilecek bir diğer özellik de, koşul başlangıçta yanlışsa bloğun hiç yürütülmeyeceğidir. Bazı uygulamalarda blok koşuldan önce de yer alabilir (Şekil 1.12 böyle bir örnektir); böyle durumlarda koşul baştan yanlış olsa bile blok en az bir kere yürütülür.



Şekil 1.13: Temel yapıların akış çizenekleri.

Bu yapıların ortak bir özelliği, hepsinin bir giriş ve bir çıkışlarının olmasıdır. Böylelikle bir bloğun çıkışı öbür bloğun girişine bağlanabilir; başka bir deyişle, bloklar ardarda eklenebilir. Ayrıca, bir bloğun içinde başka bir blok yer alabilir. Şekil 1.12'de `söylenen > tutulan` koşuluyla belirlenen seçim yapısı, `söylenen = tutulan` koşuluyla belirlenen yineleme yapısının bir alt bileşeni durumundadır.

Bir programın okunmasını ve anlaşılmasını en çok zorlaştıran etkenlerden biri programda yer alan dallanma komutlarıdır. Şimdiye kadar yazdığımız bazı algoritmalarda geçen "n. adıma git" tipi komutlar için -programlama dillerinde genellikle `goto` olarak adlandırılan- bir komut bulunması gerektiği düşünülebilir (özellikle yineleme yapıları için). Ancak blok yapılı programlamada yineleme için özel yapılar vardır ve `goto` komutunun kullanılmaması özendirilir.[5]

1.4 Soyutlama

Programlamanın temel düzeneklerinden biri *soyutlama* kavramıdır. Soyutlama, programın yapacağı işin daha küçük ve birbirinden olabildiğince bağımsız alt-işlere bölünmesidir. Alt-işler de, benzer şekilde, yapacaklarını alt-alt-işlere bölebilirler (Şekil 1.14). Bu tip tasarıma *yukarıdan aşağıya tasarım* adı verilir.[6]

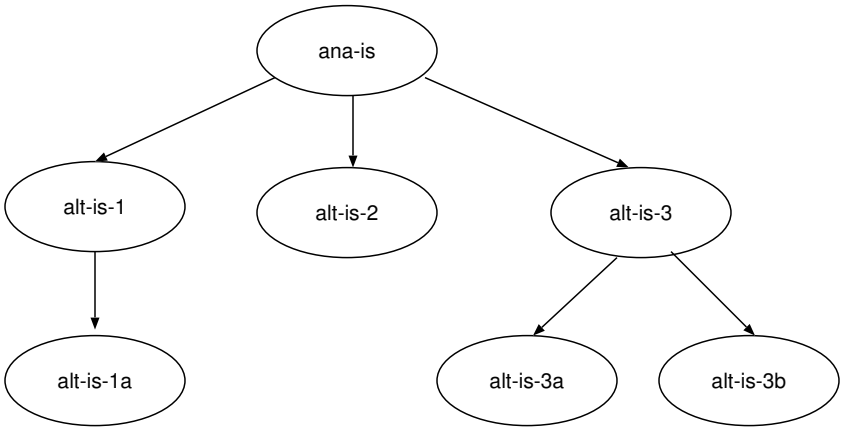
Her iş bir *yordam* (C dilindeki adıyla *fonksiyon*) tarafından gerçeklenir. Ana-yordamın görevi, alt-işleri gerçekleyen yordamları başlatmak ve bunlar arasında eşgüdümü sağlamaktır. Bir üst-yordam, kullandığı alt-yordamların nasıl çalıştıklarıyla değil, yalnızca sonuçlarıyla ilgilenir.

Soyutlamanın kazandırdıkları şöyle özetlenebilir:

- Bir işi gerçekleyen yordam yazılırken, kullandığı alt-yordamların ayrıntılarıyla uğraşılmaz; alt-yordamın doğru çalıştığı varsayılarak yordamın kendi işine yoğunlaşılabilir. Böy-

[5] Bu konuyla ilgili olarak, Edsger W. Dijkstra'nın "Go To Statement Considered Harmful" başlıklı klasik makalesini `http://www.acm.org/classics/oct95/` adresinde bulabilirsiniz.

[6] Bu konuyla ilgili olarak, Niklaus Wirth'ün "Program Development by Stepwise Refinement" başlıklı klasik makalesini `http://www.acm.org/classics/dec95/` adresinde bulabilirsiniz.



Şekil 1.14: Ana işin alt işlere bölünmesi.

lelikle büyük ve çözülmesi zor olan bir sorunla uğraşmak yerine, her biri küçük ve çözülebilir sorunlarla uğraşılır ve bunlar daha sonra biraraya getirilir.

- Programın bakımı kolaylaşır. Alt-yordamların çalışmaları birbirlerinden bağımsız olduğundan bir alt-yordamda bir değişiklik yapıldığında bunu kullanan üst-yordam (üst-yordamla olan etkileşim değişmediği sürece) değişiklikten etkilenmez.

Yordamlar olabildiğince genel amaçlı yazılmalıdır. Sözgelimi bir yordamın işi "BİL105E dersini alan öğrencilerin yılsonu sınav notlarının en büyüğünü bulmak" şeklinde tanımlanabilir. Oysa işi "herhangi bir dizinin en büyüğünü bulmak" olarak tanımlanan bir yordam geliştirmek ve bu yordamı kullanırken hangi dizinin en büyüğünün bulunmasının istendiği belirtmek daha etkin bir çalışma biçimidir. Bu durumda hangi dizi üzerinde işlem yapılacağı bilgisi yordamın *giriş parametresi* olur. Yordam, çalışması sonucu ürettiği değeri *çıkış parametresi* olarak döndürür. Örnekteki yordam kullanılırken giriş parametresi olarak "BİL105E dersini alan öğrencilerin yılsonu sınavı notları" verilirse sınavda alınan en yüksek not, "Los Angeles Lakers basketbol takımının oyuncularının ayakkabı numaraları" belirtilirse takımın en büyük ayaklı oyuncusunun ayakkabı numarası çıkış parametresi olur.

Örnek. En Büyük Ortak Bölen Bulma

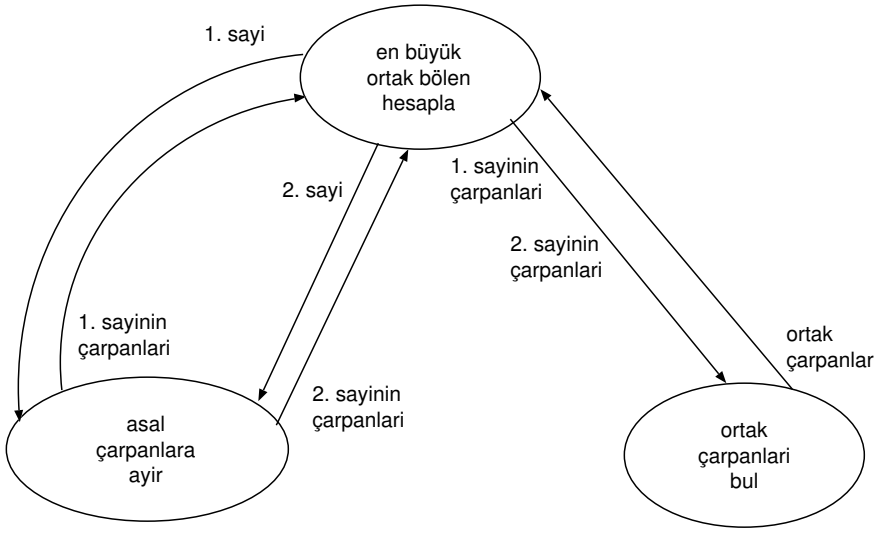
İki sayının en büyük ortak bölenini bulma işi şu şekilde alt işlere bölünebilir:

1. Birinci sayıyı asal çarpanlarına ayır.
2. İkinci sayıyı asal çarpanlarına ayır.
3. Sayıların ortak çarpanlarını belirleyerek en büyük ortak bölenin asal çarpanlarını bul.
4. Bir önceki adımda belirlediğin asal çarpanlardan en büyük ortak böleni hesapla.

Sayıların 9702 ve 945 oldukları varsayılırsa:

1. $9702 = 2 * 3 * 3 * 7 * 7 * 11 = 2^1 * 3^2 * 7^2 * 11^1$
2. $945 = 3 * 3 * 3 * 5 * 7 = 3^3 * 5^1 * 7^1$
3. ortak çarpanlar: $3^2 * 7^1$
4. en büyük ortak bölen: 63

1. ve 2. adımlar için herhangi bir sayıyı asal çarpanlarına ayıran bir yordam yazılabilir ve asal çarpanlarına ayrılacak sayı bu yordama parametre olarak yollanabilir. Benzer şekilde, 3. adımdaki ortak çarpanların bulunması işi de bir alt-yordama verilebilir. 4. adımda ortak çarpanlardan en büyük ortak bölenin hesaplanması işlemleri için alt-yordam kullanmanın fazla bir anlamı yoktur, ana yordama bırakmak daha yerinde olur. Böylece Şekil 1.15'deki yapı ortaya çıkar. Örnek sayılar üzerinde yordamların hangi giriş ve çıkış parametreleriyle çalıştıkları Şekil 1.16'da verilmiştir.

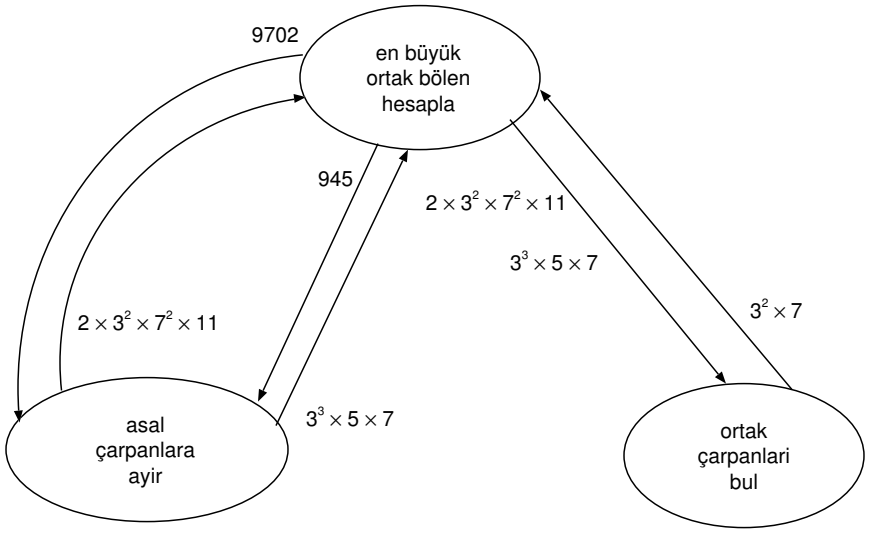


Şekil 1.15: En büyük ortak bölen hesaplama algoritmasının yukarıdan aşağıya tasarımı.

Asal çarpanlara ayırma algoritmasında görülen "bir sonraki asal sayıyı bulma işi" de asal çarpanlarına ayırma işinin bir alt-işi olarak düşünülerek bir başka yordama bırakılabilir. Bu yordam kendisine parametre olarak gönderilen sayıdan bir sonraki asal sayıyı bularak sonucu geri yollayacaktır. Benzer şekilde bu yordam da bir sayının asal olup olmadığını sınamak üzere başka bir yordamdan yararlanmak isteyebilir. Bu örnek için verilen yordamları gerçekleyecek algoritmalar bölümün sonundaki uygulamada verilmiştir.

Örnek. Euclides Algoritması

Yukarıda verilen algoritmanın en önemli sorunu, özellikle büyük sayıları asal çarpanlarına ayırmanın zorluğudur. Bilinen en eski algoritma örneklerinden biri olan Euclides algoritması, iki sayının en büyük ortak böleninin çarpanlara ayırmadan hızlı biçimde hesaplanmasını sağlar. En büyük ortak böleni bulunacak sayılardan büyük olanına `a`, küçük olanına `b` dersek:



Şekil 1.16: Örnek sayılar üzerinde en büyük ortak bölen hesaplama algoritmasının işleyişi.

$$a = q_1 b + r_1$$

şeklinde yazılabilir. Burada $r_1 = 0$ ise iki sayının en büyük ortak böleni b'dir. Değilse a ile b sayılarının en büyük ortak böleni b ile r_1 sayılarının en büyük ortak bölenine eşittir. Dolayısıyla, bölümden kalan 0 olana kadar aşağıdaki eşitlikler yazılabilir:

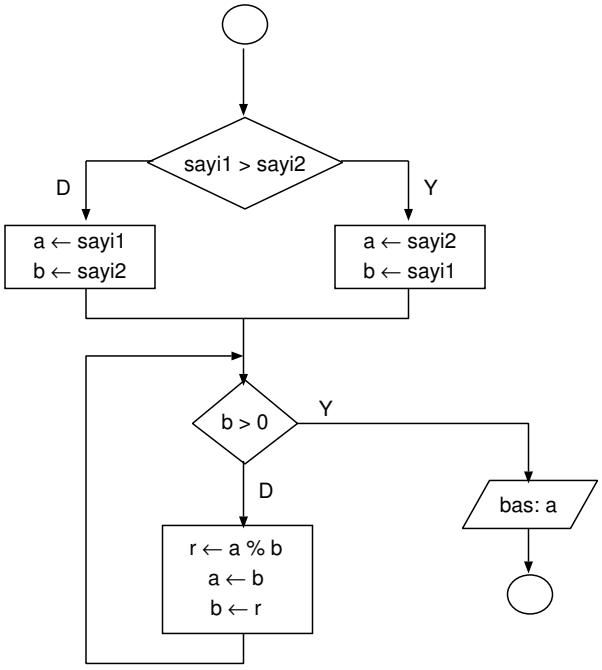
$$\begin{aligned} b &= q_2 r_1 + r_2 \\ r_1 &= q_3 r_2 + r_3 \\ &\ldots \\ r_{n-2} &= q_n r_{n-1} + r_n \\ r_{n-1} &= q_{n+1} r_n + r_{n+1} \quad (r_{n+1} = 0) \end{aligned}$$

Bu durumda a ile b'nin en büyük ortak böleni r_n'dir. Yine $a = 9702$, $b = 945$ örneğini alırsak:

$$\begin{aligned} 9702 &= 10 * 945 + 252 \\ 945 &= 3 * 252 + 189 \\ 252 &= 1 * 189 + 63 \\ 189 &= 3 * 63 + 0 \end{aligned}$$

Her denklem için en büyük ortak böleni alınacak sayılardan büyük olanını `a`, küçük olanını `b` değişkeninde, bu ikisinin bölümünden kalan değeri de `r` değişkeninde tutarsak (kalan işlemi % işaretiyle gösterilsin), bu algoritma bir yineleme yapısıyla gerçeklenebilir. Bir bölme işleminde

her zaman kalan, bölenden küçük olacağı için her yinelemede `a` değişkeni `b`'nin, `b` değişkeni de `r`'nin değerini alarak ilerlenir ve `b` 0 olduğunda en büyük ortak bölen `a` değişkeninde elde edilmiş olur. Bu algoritmanın akış çizeneği Şekil 1.17'de, örnek değerlerle işleyişi Tablo 1.3'de verilmiştir.[7]



Şekil 1.17: Euclides algoritmasının akış çizeneği.

a	b	r
9702	945	252
945	252	189
252	189	63
189	63	0

Tablo 1.3: Euclides algoritmasının örnek değerlerle işleyişi.

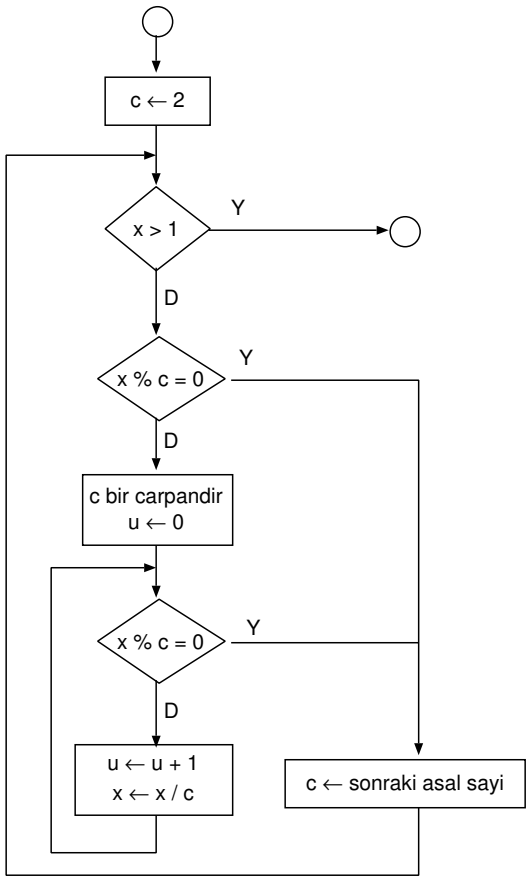
Uygulama: Algoritmalar

Örnek. Asal Çarpanların Bulunması

Herhangi bir sayının asal çarpanlarına ayrılması için kullanılabilecek bir algoritma Şekil 1.18'de verilmiştir. Bu algoritmada `x` asal çarpanlarına ayrılacak sayıyı, `c` çarpan olup olmadığı o anda

[7] Orijinal haliyle algoritmada bölmeden kalan işlemi yerine çıkartma işlemi kullanılıyordu. Bölmeden kalan işlemi algoritmanın önemli ölçüde hızlanmasını sağlayan bir değişiklik olarak sonradan getirilmiş bir yöntemdir.

sınanmakta olan asal sayıyı, `u` da sıradaki çarpanın kuvvetini gösterir. 945 sayısı örnek olarak alınırsa algoritmanın işleyişi Tablo 1.4'de verilmiştir.



Şekil 1.18: Bir sayıyı asal çarpanlarına ayırma algoritması.

Örnek. Ortak Çarpanların Bulunması

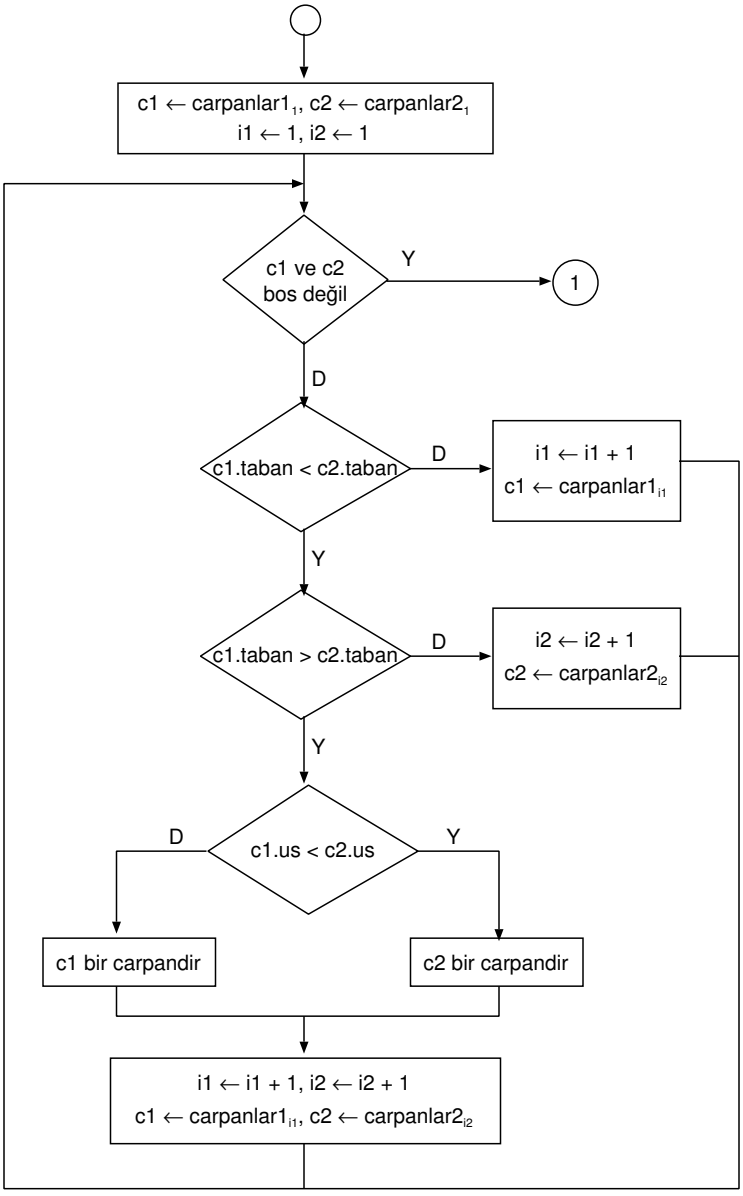
İki sayının çarpanlarından en büyük ortak bölenin çarpanlarını bulma işini yapacak yordamın algoritması geliştirilirken, asal çarpanlara ayırma yordamının çalışma şekli nedeniyle çarpanların küçükten büyüğe doğru sıralı oldukları varsayılabilir. Bu varsayıma gören çalışan algoritma Şekil 1.19'da verilmiştir. Bu algoritmada `carpanlar1` birinci sayının asal çarpanları dizisini, `carpanlar2` ikinci sayının asal sayı çarpanları dizisini, `c1` birinci sayının sıradaki asal çarpanını, `c2` de ikinci sayının sıradaki asal çarpanını gösterir. Her iki dizinin elemanlarına erişmek için de sırasıyla `i1` ve `i2` sayaç değişkenleri kullanılmıştır. Örnek dizilerle algoritmanın işleyişi Tablo 1.5'de verilmiştir.

x	c	x > 1	x % c = 0	carpan	u
945	2	D (945 > 1)	Y (945 mod 2 = 1)		
	3	D (945 > 1)	D (945 mod 3 = 0)	3	0
			D (945 mod 3 = 0)		1
315			D (315 mod 3 = 0)		2
105			D (105 mod 3 = 0)		3
35			Y (35 mod 3 = 2)		
	5	D (35 > 1)	D (35 mod 5 = 0)	5	0
			D (35 mod 5 = 0)		1
7			Y (7 mod 5 = 2)		
	7	D (7 > 1)	D (7 mod 7 = 0)	7	0
			D (7 mod 7 = 0)		1
1			Y (1 mod 7 = 1)		
	11	Y (1 = 1)			

Tablo 1.4: Asal çarpanlarına ayırma algoritmasının işleyişi.

c1	c2	taban	us	carpan
2^1	3^3	$2 < 3$		
3^2		$3 = 3$	$2 < 3$	3^2
7^2	5^1	$7 > 5$		
	7^1	$7 = 7$	$1 < 2$	7^1
11^1	-			

Tablo 1.5: Ortak çarpanları bulma algoritmasının işleyişi.



Şekil 1.19: Ortak çarpanları bulma algoritması.

1.5 Giriş / Çıkış

Bir programlama dilinin kullanılabilir olması için dilde blok yapılı programlamanın kavramlarını desteklemenin dışında bazı gereklilikler de yerine getirilmelidir:

Giriş Bir program her seferinde aynı veri değerleri üzerinde çalışmaz, işleyeceği verileri çalışma sırasında bir şekilde öğrenmesi gerekir. Bunun en sık karşılaşılan yöntemi verileri programı kullanan kişiye sormak olmakla birlikte verileri bir dosyadan okumak ya da ortamdan öğrenmek (sözgelimi odanın sıcaklığını ölçen bir duyargadan almak) gibi seçenekler de bulunabilir. Programlama dillerinin girdi komutları giriş birimlerinden aldıkları değerleri değişkenlere aktarırlar. Aksi belirtilmedikçe standart giriş birimi kullanıcının tuştakımıdır.

Çıkış Program, verilerin işlenmesi sonucu elde ettiği sonuçları bir şekilde değerlendirmelidir. En sık görülen yöntem sonucun kullanıcının ekranına yazılmasıdır ama girişte olduğu gibi sonuçların bir dosyaya yazılması ya da ortama gönderilmesi (oda sıcaklığını denetleyen klimaya bir sinyal yollanması gibi) sözkonusu olabilir. Programlama dillerinin çıktı komutları, değişken değerlerini istenen çıkış birimine yönlendirirler. Aksi belirtilmedikçe standart çıkış birimi kullanıcının ekranıdır.

Programın çalışması sırasında karşılaşılan hata durumlarının bildirilmesi de çıktının bir parçasıdır. Ancak, hataların daha kolay farkedilmelerini sağlamak amacıyla, bu iletilerin normal çıktı iletilerinden ayrılabilmeleri istenir. Bu nedenle, hata iletileri hata birimine yönlendirilir. Aksi belirtilmedikçe bu birim -standart çıkışta olduğu gibi- kullanıcının ekranıdır.

Bu notlarda işlenecek örnek programlarda çalışma hep aynı şekilde olacaktır: verilerin girilmesi, işlenmesi, sonuçların gösterilmesi. Bu tip programlara *konsol* programı, *komut satırı* programı ya da *metin kipi* programı gibi adlar verilir. Grafik arayüzle çalışan programlar ise olaya dayanan bir mantıkla çalışırlar. Program çalışmaya başladığında kullanıcıyla iletişimi için gerekli arayüzü kurar (pencere çizer, menü oluşturur, düğme koyar, v.b.) ve sonra kullanıcının bir edimde bulunmasını bekler. Kullanıcının edimine göre ilgili yordamları çalıştırır.

1.6 Program Geliştirme

Bir programın geliştirilmesi çeşitli aşamalardan oluşur:

Tasarım Bu aşamada yazılacak program "kağıt üzerinde" tasarlanır. Yani programın algoritmasına, hangi programlama dilinin kullanılacağına, kullanıcıyla nasıl bilgi alışverişinde bulunulacağına, kısacası neyin nasıl yapılacağına karar verilir. Dikkatli bir tasarım, programın hem geliştirilmesinde hem de bakımında büyük kolaylıklar sağlar.

Kodlama Bu aşamada program tasarım sırasında verilen kararlara göre bir programlama diliyle yazılır. Bunun sonucunda yazılımın *kaynak kodu* oluşur.[8] Kaynak kodunu bilgisayar ortamında oluşturmak için *editör* adı verilen yazılımlar kullanılır.

[8] Metinde bunda sonra geçen *kod* sözcüğü aksi belirtilmedikçe kaynak koduna karşı düşecektir.

Sınama Bu aşamada program çalıştırılarak çeşitli senaryolar için doğru sonuçlar üretip üretmediğine, beklendiği gibi davranıp davranmadığına bakılır. Bu işlemin etkinliği gözönüne alınan senaryoların gerçekte oluşabilecek durumların ne kadarını örttüğüyle bağlantılıdır.

Hata Ayıklama Bu aşamada sınama aşamasında bulunan hataların nedenleri belirlenerek gerekli düzeltmeler yapılır. Programların ilk yazıldıkları şekliyle doğru olmaları neredeyse olanaksızdır. Hatta üstüne çok çalışılmış, pek çok hatası bulunup düzeltilmiş programlarda bile bütün hataların bulunup düzeltilmiş olması son derece düşük bir olasılıktır. Ayrıca programlara yeni yetenekler eklendikçe yeni hatalar oluşacağından aşağı yukarı hiçbir program hiçbir zaman tam olarak hatasız olmayacaktır. Hata ayıklama işlemine yardımcı olmak üzere *hata ayıklayıcı* adı verilen yazılımlar kullanılır.

Programlarda iki tür hata sözkonusu olabilir:

1. Yazım hataları: Programcının yazdığı bazı komutlar kullandığı programlama dilinin kurallarına uymuyordur.
2. Mantık hataları: Programcının yazdığı kod dilin kuralları açısından doğrudur ama çalıştırılmasında sorun çıkar. Sözgelimi bir tamsayıyı başka bir tamsayı değişkene bölmek geçerli bir işlemdir ama bölen sayı 0 olursa bu işlem gerçekleştirilemez. Bu tür hatalara *çalışma-zamanı hatası* da denir.

1.6.1 Programların Değerlendirilmesi

Bu aşamalar sonucunda ortaya çıkan bir programın iyi bir program olup olmadığının, ya da ne kadar iyi bir program olduğunun değerlendirilmesinde şu ölçütlere başvurulur:

Etkinlik Bu konu algoritmaların karşılaştırılmalarında kullanılan ölçütlerle aynıdır. Program ne kadar hızlı çalışıyor? Düzgün kullanılabilmesi için ne kadar sistem kaynağı gerekiyor (hangi işlemci, ne kadar bellek, ne kadar disk, v.b.?).

Sağlamlık Program, kullanıcının yapacağı kullanım hatalarına karşı dayanıklı mı? Sözgelimi, kendisine bir yıl bilgisi sorulduğunda kullanıcı yanlışlıkla (ya da kötü niyetle) bir isim yazarsa ne oluyor?

Taşınabilirlik Program, üzerinde fazla değişiklik yapılması gerekmeden, başka bir donanım ve başka bir işletim sistemi üzerinde çalışacak hale getirilebiliyor mu?

Anlaşılabilirlik Başka biri programı okuduğunda anlayabilecek mi? Hatta üstünden bir süre geçtikten sonra kendiniz baktığınızda anlayabilecek misiniz?

Bakım kolaylığı Programda hata olduğunda hatanın kaynağının belirlenmesi ve düzeltilmesi kolay mı?

Geliştirilebilirlik Program yeni yeteneklerin eklenmesiyle geliştirilmeye açık mı? Bu tip eklemelerin yapılması kolay mı?

Geliştirme aşamalarında daha az sorunla karşılaşmak ve daha kaliteli yazılımlar üretmek için çeşitli yöntemler geliştirilmiştir. Bu yöntemlerin temel farkı, problemin nasıl modelleneceğine ilişkin yaklaşımlarıdır. Blok yapılı yaklaşım, nesneye dayalı yaklaşım, fonksiyonel yaklaşım gibi yöntemler çeşitli alanlarda yaygın olarak kullanılmaktadır. Seçtiğiniz programlama dili hangi programlama yaklaşımını kullanacağınızı da belirler. Sözgelimi C dili blok yapılı, Java dili nesneye dayalı, Haskell diliyse fonksiyonel dillerdir. C++ dili hem blok yapılı hem de nesneye dayalı özellikler gösteren karma bir dildir.

1.6.2 Programların Çalıştırılması

Bilgisayarların çalıştıracakları programların belli bir biçimi olması zorunludur. Bu biçim, bilgisayardan bilgisayara ve işletim sisteminden işletim sistemine göre farklılıklar gösterir. Sözgelimi, bir kişisel bilgisayar üzerinde Windows işletim sisteminde çalışan bir program, Sun marka bir işistasyonunda Solaris işletim sisteminde çalışmaz. Hatta, yine aynı kişisel bilgisayar üzerinde Linux işletim sisteminde de çalışmaz. Programların bu çalıştırılabilir biçimine *makina kodu* adı verilir. Makina kodu, insanların anlaması ve üzerinde çalışması son derece zor bir biçim olduğundan programcılar programları doğrudan bu kod ile yazmazlar. İnsanın anlaması daha kolay (insan diline daha yakın) "yüksek düzeyli" bir dil ile kaynak kodunu oluşturup yardımcı yazılımlar aracılığıyla makina koduna çevirir ve çalıştırırlar.

Kaynak kodunun makina koduna çevrilmesi ve çalıştırılması işlemi üç farklı yaklaşımla gerçekleştirilebilir:

Yorumlama Programın komutları bir *yorumlayıcı* tarafından *teker teker* okunur, makina koduna çevrilir ve çalıştırılır. Yani yorumlayıcı, önce birinci komutu okur, makina koduna çevirir ve çalıştırır. Sonra ikinci komutu alır ve aynı işlemleri yapar. Programın her çalışmasında çevirme işlemi yeniden yapılır. Herhangi bir komutun çevrilmesinde ya da çalıştırılmasında bir hatayla karşılaştığında bunu kullanıcıya bildirir ve çalışmayı durdurur. Basic, Perl, Tcl gibi diller genellikle yorumlayıcılar aracılığıyla kullanılırlar.

Derleme Program bir *derleyici* tarafından *bir bütün halinde* okunur ve makina koduna çevrilerek bir çalıştırılabilir dosya oluşturulur. Bu dosya daha sonra istendiği zaman çalıştırılır, yani çevirme ile çalıştırma işlemleri birbirinden ayrılır. Böylelikle çevirme işlemi yalnızca bir kere yapılır. Herhangi bir komutta hata varsa çevirme işlemi tamamlanmaz, yani çalıştırılabilir kodun oluşması için hiçbir yazım hatası olmaması gerekir; ancak program daha sonra çalıştırılırken çalışma-zamanı hatalarıyla karşılaşılabilir. Fortran, Pascal, C gibi diller genelde derleyicilerle kullanılırlar.

Karma Hem derleme hem de yorumlama tekniği kullanılır. Bu tip çalışmada kaynak kodu sanal bir bilgisayarın makina koduna (bytecode) çevrilir ve daha sonra bu sanal bilgisayarı gerçekleyen bir program yardımıyla yorumlanarak çalıştırılır. Örneğin Java dilinde yazılmış bir kaynak kodu önce Java derleyicisinden geçirilerek Java sanal makinasının (Java Virtual Machine - JVM) makina koduna dönüştürülür; sonra da bu sanal makinayı gerçekleyen bir Java çalışma ortamı (Java Runtime Environment - JRE) yardımıyla çalıştırılır.

Yorumlama yönteminde kodun okunması ve çevrilmesi programın çalışması sırasında yapıldığından hız düşüktür. Ayrıca yorumlayıcı yalnızca karşılaştığı ilk hatayı rapor edebilir. Bu

hata düzeltildiğinde sonraki çalışmada da program ancak bir sonraki hataya kadar ilerleyebilir. Oysa derleyici kaynak kodundaki bütün hataları bulabilir. Buna karşılık hata ayıklama işlemi yorumlayıcılarla daha kolaydır. Ayrıca derleyicilerle gelen bazı sınırlamaların kalkması nedeniyle daha esnek bir çalışma ortamı sağlanır. Son yıllarda iki yöntemin üstün yanlarını birleştiren karma diller (Java ve Python gibi) öne çıkmaya başlamışlardır.

1.6.3 Kitaplıklar

Bir programcıya gerekebilecek her şeyi programlama dilinin içine almak o dili işleyecek araçların hantallaşmalarına neden olur. C dili bu nedenle oldukça "küçük" bir dil olarak tasarlanmıştır. Örneğin basit aritmetik işlemlerin ötesindeki matematik işlemleri dilin tanımı içinde yer almaz; sözgelimi karekök alma işlemi için bir C komutu yoktur. Bununla birlikte, pek çok programcının bu tip işlemlere gereksinim duyacakları da açıktır. Böyle bir durumda programcı, isterse karekök alma işlemini yapacak kodu kendisi yazabilir. Ancak programcının bu tip işlemler için kendi kodlarını yazmasının önemli sakıncaları vardır:

1. Her programcının aynı işleri yapan kodlar yazması büyük zaman kaybına yol açar.
2. Programcının yazacağı kod hatalı olabilir, yani yanlış sonuç üretebilir. Ya da doğru sonuç üretse bile, yeterince etkin olmayabilir, yani işlemi yapmak için gereğinden fazla zaman ya da sistem kaynağı harcayabilir.

Hem dilin tanımını küçük tutmak hem de bu sakıncaları giderebilmek için, çoğu programcıya gerekebilecek işlemler (yordamlar) *kitaplık* adı verilen arşivlerde toplanmıştır. Bir sayının karekökünü almak isteyen bir programcı matematik kitaplığındaki `sqrt` yordamını kullanabilir. Kitaplığın kullanılması yukarıda sözü geçen sakıncaları giderir, yani programcıya zaman kazandırdığı gibi, doğru ve etkin çalıştığı sınanmış olduğundan dikkatini programın diğer kısımlarına yoğunlaştırma fırsatı verir.

1.6.4 Standartlar

C dilinin geliştirilmesinde gözetilen ana hedeflerden biri taşınabilirlikti. Bunun için değişik ortamlardaki araçlar arasında bir standart olması gerektiği açıktır. C dilinde yapılan standartlaşma çalışmaları sonucunda oluşan ANSI C standardı, hem C dilinin kurallarını belirler, hem de bir C geliştirme ortamında bulunması zorunlu olan standart kitaplıkları ve bu kitaplıklarda yer alacak yordamları tanımlar. Uluslararası Standartlar Organizasyonu'nun (ISO) C++ dili için hazırladığı ISO-C++ standardı da son yıllarda yazılmış bütün C/C++ geliştirme ortamları için en temel kaynaklardan birini oluşturmaktadır.

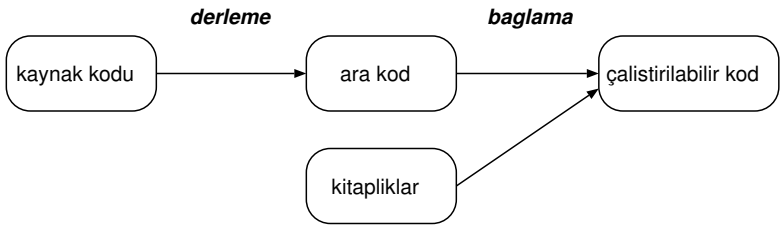
Diğer önemli bir standart olan POSIX standardı ise programlama dili ile işletim sistemi arasındaki bağlantıları belirler. Örneğin dosya silmek, dosya adı değiştirmek, sistemdeki başka bir programla haberleşmek gibi işlemler için gereken yordamlar bu standartta yer alırlar.

Yine de sıkça gereksinim duyulan bütün işlemler için bütün kitaplıklarda bir standart henüz sağlanamamıştır. Örneğin grafik bir arayüze sahip uygulamalardaki grafik işlemlerini yapacak kitaplıklar standartlaştırılmamış olduğundan çeşitli firma ve kurumlar bu işlemleri yapan farklı

kitaplıklar geliştirmişlerdir. Dolayısıyla bu kitaplıklar kullanılarak yazılan programlar tam anlamıyla taşınabilir olmamakta, yalnızca bu kitaplığın desteklediği ortamlara taşınabilmektedir. Son zamanlarda farklı işletim sistemlerinde çalışabilen grafik kitaplıkları yaygınlaştığı için doğru araçlar seçildiğinde bu sorunun da üstesinden gelinebilmektedir.

1.6.5 Derleme Aşamaları

Kaynak koddan üretilecek olan çalıştırılabilir makina kodunda hem programcının kendi yazdığı yordamların, hem de yararlandığı kitaplık yordamlarının makina koduna çevrilmiş hallerinin bulunması gerekir. Bu nedenle, kaynak kodunun makina koduna çevrilmesi iki aşamada gerçekleşir (Şekil 1.20):



Şekil 1.20: Tek kaynak kodlu projelerin derleme aşamaları.

Derleme İlk aşamada kaynak kodu derlenerek bir *ara koda* dönüştürülür. Bu kod, kullanıcının yazdığı yordamların makina kodu karşılıklarını içeren bir dosyadır. Ancak, programcının kullanmış olduğu kitaplık yordamlarını içermediğinden henüz çalıştırılabilir durumda değildir.

Bağlama İkinci aşamada ara kod ile programcının kullandığı kitaplık yordamları arasında bağlantılar kurulur ve çalıştırılabilir dosya üretilir. Sözgelimi, programcı karekök alma için matematik kitaplığındaki bir yordamı kullandıysa, *bağlayıcı* ara kodda karekök yordamının kullanıldığı yerlerde gerekli düzenlemeleri yapar.

Sorular

1. Bölüm 1.1'de verilen takas işlemi için yazılan üç atama komutunun sıralarını değiştirerek ne sonuçlar elde edildiğini belirleyin. Problemin kaç doğru çözümü vardır?

2. Bölüm 1.2.1'de yapılan algoritma hızı karşılaştırmalarında verilen en kötü ve ortalama durum değerlerini kendiniz çıkarın. Alt sınırın 1, üst sınırın 2^n - 1 şeklinde verildiği "genel durumda" en kötü ve ortalama değerleri n cinsinden hesaplayın.

3. İki sayının en büyük ortak bölenini hesaplamak için verilen iki algoritmayı, basitlik ve etkinlik açılarından karşılaştırın. İkiden fazla sayının en büyük ortak bölenlerini hesaplamak için bir algoritma geliştirin.

4. Euclides algoritmasında iki sayıdan büyük olanının hangisi olduğuna bakılmaksızın `a` değişkenine `sayı1`, `b` değişkenine `sayı2` değerleri atanarak yineleme yapısına girildiğini varsayalım. Başlangıçta `sayı1 > sayı2` ya da `sayı1 < sayı2` olması durumlarında algoritma doğru sonuç üretir mi?

5. Çarpanlara ayırarak en küçük ortak kat hesaplamak için gereken "en küçük ortak katın çarpanlarını bulma" yordamının akış çizeneğini çizin ve örnek değerler üzerinde nasıl işleyeceğini gösteren tabloyu oluşturun.

Bölüm 2

C Diline Giriş

Bu bölümde değişkenler, atama, deyimler, giriş-çıkış gibi konuların C dilinde nasıl gerçeklendikleri üzerinde durulacaktır.

Örnek 1. Daire Çevresi ve Alanı

Yarıçapını kullanıcıdan aldığı bir dairenin çevresini ve alanını hesaplayarak ekrana yazan bir program yazılması isteniyor. Programın örnek bir çalışmasının ekran çıktısı Şekil 2.1'de verilmiştir.

```
Yarıçapı yazınız: 4.01
Çevresi: 25.1828
Alanı: 50.4915
```

Şekil 2.1: Örnek 1 ekran çıktısı.

Örnek üzerinde bir C programının bazı temel özelliklerini görelim:

Açıklamalar Anlaşılabilirliği artırmak için kodun içinde gerekli yerlere açıklamalar yazmakta büyük yarar vardır. Özellikle kolayca anlaşılmayacak programlama tekniklerinin kullanıldığı kod parçalarının açıklanması gerekir. Açıklamalar derleyici tarafından gözardı edilir, yani programın işleyişlerine hiçbir etkileri yoktur. Kodun içine açıklama iki şekilde yazılabilir:

- Birinci yöntemde, açıklamanın başına bölü-yıldız, sonuna yıldız-bölü simgeleri konur. Bu şekildeki açıklamalar birden fazla satır sürebilir. Örnekteki birinci açıklama birinci satırdaki "`İlk C programım`" sözcükleriyle başlar ve üçüncü satırdaki "`alanını hesaplar.`" sözcükleriyle biter. Bu tip açıklamalar içiçe yazılamaz, yani bir açıklamanın içinde ikinci bir açıklama olamaz. İçiçe açıklamalar şu şekilde bir yapı oluşturur:

  ```
  ...a... /* ...b... /* ...c... */ ...d... */ ...e...
  ```

Örnek 1 Bir dairenin çevresini ve alanını hesaplayan program.

```
/* İlk C programım.                                                    *
 *                                                                     *
 * Yarıçapı verilen bir dairenin çevresini ve alanını hesaplar. */

#include <iostream>             // cout,cin,endl için
#include <stdlib.h>             // EXIT_SUCCESS için

using namespace std;

#define PI 3.14

int main(void)
{
    float radius;
    float circum, area;

    cout << "Yarıçapı yazınız: ";
    cin >> radius;
    circum = 2 * PI * radius;
    area = PI * radius * radius;
    cout << "Çevresi: " << circum << endl;
    cout << "Alanı: " << area << endl;
    return EXIT_SUCCESS;
}
```

Bu yapıda birinci `/*` ile açıklama başlar. Açıklamanın içinde görülen ikinci `/*` gözardı edilir ve gelen ilk `*/` açıklamayı sona erdirir; yani açıklamayı `b` ve `c` bölgeleri oluşturur. Bundan sonra gelen bölüm (`d` bölgesi) normal C kodu gibi değerlendirileceğinden hataya yol açar.

- İkinci yöntemde, açıklama çift bölü ile başlar ve satırın sonuna kadar sürer.[1] Kısa açıklamalar için bu yöntem daha kullanışlıdır. Örnekteki ikinci açıklama beşinci satırdaki "`cout,cin,endl için`" sözcüklerinden oluşur. Üçüncü açıklama da altıncı satırdaki "`EXIT_SUCCESS için`" sözcüklerini kapsar.

Komutlar C dilinde komutlar noktalı virgül ile sona erer. Peşpeşe birden fazla boşluk derleyici tarafından tek boşluk olarak değerlendirilir; dolayısıyla bir komutun bir satır içinde başlayıp sona ermesi zorunluluğu yoktur. Yani

```
cout << "Alanı: " << area << endl;
```

komutu

```
cout << "Alanı: "
     << area << endl;
```

biçiminde ya da

```
cout << "Alanı: "
     << area
     << endl;
```

biçiminde yazılabilirdi.

Gelenek

C dilinde komutların noktalı virgül ile sona ermesi ve fazla boşluklar ile satır geçişlerinin öneminin olmaması nedeniyle komutlar satırlara istendiği şekilde yerleştirilebilir.Ancak bu konuda bir düzene uyulmazsa kodun okunması ve anlaşılması son derece zorlaşır. Okunabilirliği artırmak amacıyla alt-bloklar bir miktar içeriden başlatılırlar (*girintileme*). Böylece aynı düzeydeki (aynı bloğa ait) komutlar aynı hizadan başlarlar. Örnekte fonksiyon bloğundaki bütün komutlar satır başından dört harf içeriden başlamaktadır. Özellikle içiçe yapıların kullanıldığı durumlarda (seçimin içindeki yinelemenin içindeki seçimin içindeki seçim gibi) blokları hiyerarşik bir şekilde girintilemek büyük önem taşır.

Atama Atama işlemi eşit işaretiyle yapılır. Örnekteki

```
circum = 2 * PI * radius;
```

komutu bir atama komutudur, `2 * PI * radius` deyiminin sonucunu `circum` değişkenine atar.[2]

[1] Bu açıklama yöntemi C++ ile getirilmiş bir yeniliktir, C dilinde geçerli değildir.

[2] Tek atama komutuyla birden çok değişkene aynı değer atanabilir. Örneğin

```
a = b = c = 24;
```

komutu `a`, `b` ve `c` değişkenlerinin üçüne de 24 değerini atar. Bu işlem sağdan sola doğru gerçekleştirilen peşpeşe atamalar şeklinde düşünülebilir:

```
c = 24; b = c; a = b;
```

Fonksiyonlar Blok yapılı programlamadaki yordamlar C dilinde fonksiyonlar ile gerçekleştirilir. Ana işi yapan fonksiyonun adı `main` olarak verilmelidir. Programın yürütülmesine ana işten başlanacağı için her programın bir ve yalnız bir `main` fonksiyonu bulunması zorunludur.

C dilinde bir fonksiyonun içerdiği komutlar bir blok olarak değerlendirilir. Bloklar açsüslü-ayraç ile başlar ve kapa-süslü-ayraç ile sona erer. Bu notlardaki çoğu örnekte `main` fonksiyonu şu şablona uyacaktır:

```
int main(void)
{
   ...
   return EXIT_SUCCESS;
}
```

Başlık dosyaları Kitaplıklarda tanımlanmış olan fonksiyonlar, birimler ya da büyüklükler kullanıldığı zaman derleyiciye bunlarla ilgili bilgileri nerede bulabileceğini söylemek gerekir. Bu bilgilerin yer aldığı başlık dosyaları program başında `#include` komutuyla belirtilir. Örneğin `cout` birimi `iostream` başlık dosyasında bulunduğundan örnek programda

```
#include <iostream>
```

komutu yer almaktadır.[3] Benzer şekilde, `EXIT_SUCCESS` sözcüğünün kullanılabilmesi için `stdlib.h` dosyası içerilmelidir.

Bir kitaplıktan yararlanacağınız zaman gerekli bilgilerin hangi başlık dosyalarında yer aldığını bilmeniz gerekir. Temel fonksiyonlar standartlarda belirlenmiş olduğundan (bkz. Bölüm 1.6.4) bunların başlık dosyalarını çoğu C/C++ kitabında bulabilirsiniz. Standartlara girmemiş fonksiyonlar içinse kullandığınız derleyici ve kitaplıkların yardımcı belgelerine başvurmalısınız.[4]

2.1 İsimler

Programlardaki değişken, fonksiyon gibi varlıklara verilen isimlerin bazı kurallara uymaları gerekir:

- İsimler, İngilizce büyük ve küçük harfler, rakamlar ve altçizgi işaretinden oluşabilir. Bu kurala göre `pi`, `weight`, `weight1` ve `weight_1` geçerli isimlerdir, ancak π, `ağırlık` ve `weight-1` geçerli isimler değildir.
- İsmin ilk simgesi bir rakam olamaz. Bu kurala göre `2weight` geçerli bir isim değildir.

[3] C standardında başlık dosyalarına `.h` uzantısı verilir. C++ standardında ise bazı başlık dosyalarının uzantısı bulunmayabilir. Örnekteki `iostream` başlık dosyası C++ ile tanımlanmış yeni başlık dosyalarına bir örnektir.

[4] Unix işletim sistemlerinde yardımcı belgeler ile ilgili bilgi için bkz. Ek B.1.

- İsmin en az ilk 31 simgesi anlamlıdır. Başka bir deyişle, ilk 31 simgesi aynı olmayan iki ismin farklı olacağı kesindir ama aynılarsa derleyici bu iki ismin farklı olmadığına karar verebilir. Örneğin `population_increase_between_years_2000_and_2001` ile `population_increase_between_years_2001_and_2002` isimli iki değişken tanımlanırsa ilk 31'er simgeleri aynı olduğundan bazı derleyiciler bu ikisinin aynı değişken olduklarına karar verebilir.

- İsimlerde büyük-küçük harf ayrımı vardır. Yani `weight1`, `Weight1`, `WEIGHT1` ve `wEiGHT1` isimlerinin hepsi geçerli olmakla birlikte hepsi birbirinden farklıdır.

- C dilinin sözcükleri isim olarak seçilemez. Yani örneğin `int`, `main`, `void`, `return` geçerli isimler değildir. Kitaplıklardan alınan fonksiyon ya da diğer varlıkların isimleri saklı isimler arasında değildir. Sözgelimi, istenirse `cout` isimli bir değişken kullanılabilir ancak bu durumda çıktı için kullanılan `cout` kitaplık biriminden artık yararlanılamaz.

  Büyük projelerde isim çakışmaları sorun yaratmaya başlar. Bu durumu düzeltmek amacıyla isim uzaylarından yararlanılır. Bu kitaptaki örnekler küçük boyutta oldukları için bu sorunlar gözardı edilecek ve standart isim uzayının kullanıldığını belirtmek üzere programın başında

  ```
  using namespace std;
  ```

  bildirimi yapılacaktır. Bu bildirim `cin`, `cout` ve `endl` değerlerine kolay erişimi sağlar; kullanılmazsa bu değerlere erişim için başlarına `std::` eklemek gerekir:

  ```
  std::cin >> radius;
  std::cout << "Area: " << ... << std::endl;
  ```

İsimlerin seçiminde çoğu programcının uyduğu bazı gelenekler vardır: *Gelenek*

- Değişken ve fonksiyon isimleri küçük harflerle başlar. Başka bir deyişle, büyük harfler ya da altçizgi işaretiyle başlamaz.

- Değişken ve fonksiyonlara gösterdikleri bilgiye ya da yaptıkları işe uygun düşen, anlamlı bir isim verilir. Sözgelimi bir insanın ağırlığı bilgisini tutacak bir değişkene `x4szb` gibi anlamsız ya da `height` gibi yanıltıcı isimler verilmez.

- Anlamlı isimler verilmek istendiğinde bazen birden fazla sözcüğe gereksinim duyulabilir. Bu durumda ya ismi oluşturan iki sözcük bir altçizgi işaretiyle birleştirilir (örneğin `birth_month`) ya da ikinci sözcük büyük harfle başlar (örneğin `birthMonth`).

2.2 Değerler

Tamsayıların doğal gösterilimleri onlu düzendedir, ancak istenirse sekizli ya da onaltılı düzende de gösterilebilirler. Örnekte 2 sayısı onlu gösterilimde yazılmış bir tamsayıdır. 0 rakamıyla başlayan sayıların sekizli, 0x ile başlayanların onaltılı düzende oldukları varsayılır. Buna göre, 59 sayısı sekizli düzende 073, onaltılı düzende 0x3b olarak yazılır.

Kesirli sayıların doğal gösterilimi noktalı gösterilimdir, ancak istenirse bilimsel gösterilim (mantis * 10^{us}) de kullanılabilir. Örnekte 3.14 sayısı noktalı gösterilimde yazılmış bir kesirli sayıdır. Sayının yazılışında E simgesi geçiyorsa bu simgenin öncesi mantis, sonrası üs olarak değerlendirilir. Buna göre, 3.14 sayısı bilimsel gösterilimde 314E-2 (314 * 10^{-2}) olarak yazılabilir.

Simgeler tek, katarlar çift tırnak işaretleri arasında yazılırlar. Örnekte "Alanı: " değeri bir katardır; yalnızca A harfinden söz edilmek isteniyorsa 'A' şeklinde yazılır. Katar içinde yazılan her boşluğun önemi vardır, yani sözgelimi peşpeşe beş boşluk bırakıldığında bu bir boşluk olarak değil, beş boşluk olarak değerlendirilir. Örneğin "Alanı : " katarında "Alanı" sözcüğü ile iki nokta üstüste işareti arasında beş boşluk bulunacaktır.

2.3 Değişkenler

C dilinde, bir değişkene bir değer vermeden ya da değerini bir hesapta kullanmadan önce değişkenin tanımlanması gerekir. Tanımlama işlemi, bellekte gerektiği kadar yerin bu değişken için ayrılmasını sağlar; böylece sistem, bellekte ayrılan bu yeri bu değişken için kullanır ve başka işlerde kullanmaz.

Tanımlama, değişkenin tipinin ve adının belirtilmesinden oluşur. Örnekte

```
float radius;
```

tanımı derleyiciye `radius` adında bir değişken olduğunu ve bir kesirli sayı tutacağını belirtmeye yarar.

Aynı tipten olan değişkenler aynı tanımın içinde yer alabilirler. Örnekteki

```
float circum, area;
```

tanımı, her ikisi de birer kesirli sayı olan, `circum` ve `area` adında iki değişken kullanacağımız anlamına gelir. Örnekteki tanımlar istenirse

```
float radius;
float circum;
float area;
```

şeklinde ayrılarak ya da

```
float radius, circum, area;
```

şeklinde birleştirilerek de yazılabilirdi.

Tanım sırasında istenirse değişkene başlangıç değeri de verilebilir:

```
float radius, circum = 0.0, area = 0.0;
```

Bu durumda tanım sırasında `circum` ve `area` değişkenlerinin her ikisine de 0.0 değeri verilir, `radius` değişkenine bir başlangıç değeri verilmez.

Gelenek Değişkenlere başlangıç değeri atanması yararlı bir alışkanlıktır. Başlangıç değeri atanmazsa değişken rasgele bir değer alabilir ve programcının dikkatsiz davranması durumunda bu rasgele değer yanlışlıkla hesaplarda kullanılabilir ve istenmeyen sonuçlara yol açabilir.

Gelenek Genelde bir blok içindeki değişken tanımları ile komutlar birbirlerinden ayrılır, yani kullanılacak bütün değişkenlerin tanımları bittikten sonra komutlar başlar.[5] Tanımların sona erdiği ve komutların başladığı yerin kolayca görülmesini sağlamak amacıyla tanımlar ile komutlar arasında bir satır boşluk bırakılır.

2.4 Veri Tipleri

C dilinde tanımlanan taban veri tipleri şunlardır (bkz. Bölüm 1.1.1):

tamsayı `int` saklı sözcüğüyle belirtilir. Bu veri tipi `short` ve `long` belirteçleriyle geliştirilerek kısa tamsayı (`short int`) ve uzun tamsayı (`long int`) tipleri oluşturulabilir. Aksi belirtilmedikçe tamsayı tiplerinin işaretli oldukları varsayılır, yani hem pozitif hem de negatif değerler alabilirler. Değişken yalnızca pozitif sayılardan (0 da olabilir) değer alacaksa işaretsiz olarak tanımlamak için `unsigned` sözcüğü kullanılabilir. Kısacası, 6 adet tamsayı tipi vardır: `int`, `short int`, `long int`, `unsigned int`, `unsigned short int` ve `unsigned long int`.

kesirli sayı `float` saklı sözcüğüyle belirtilir. Sayı daha yüksek duyarlılıkla gösterilmek istenirse çifte duyarlılıklı (`double`) ya da uzun çifte duyarlılıklı (`long double`) tipleri kullanılabilir.

simge `char` saklı sözcüğüyle belirtilir. Simgeler kullanılan kodlamadaki sıralarına göre değerler alırlar (bkz. Ek A). Örneğin 'A' harfi ASCII kodlamasında 65. sırada olduğundan 'A' simgesinin sayı değeri 65'tir.

mantıksal `bool` saklı sözcüğüyle belirtilir. Bu tipten değişkenler `true` ya da `false` değerini alabilirler.[6]

Bir değişkenin tanımlanacağı veri tipi, o değişkenin alabileceği değer aralığını belirlediğinden son derece önemlidir. Bu nedenle bir değişkenin veri tipine karar verirken o veri tipinin izin verdiği değer aralığına dikkat etmek gerekir. Sözgelimi `short int` veri tipi yalnızca -32768 ile +32767 arasındaki sayıların gösterilebilmesine olanak veriyorsa ve sizin tanımlayacağınız değişkenin 45000 değerini alması sözkonusu olabilecekse değişkeninizi bu tipten tanımlamamalısınız. Bir veri tipinin boyu `sizeof` işlemi yardımıyla belirlenebilir. Her değişken bellekte bu boy kadar yer kaplar (sekizli cinsinden).

Kısa tamsayı veri tipinin boyunun 2 sekizli yani 16 bit olduğunu varsayalım. Bu boy hem işaretli hem de işaretsiz kısa tamsayılar için geçerlidir. Bu durumda işaretsiz kısa tamsayı cinsinden tanımlanan bir değişkenin alabileceği en küçük değer 0, en büyük değer ise $2^{16} - 1$ yani 65535 olacaktır.

[5] C++ dilinde komutlar başladıktan sonra da değişken tanımlanabilir. C dilinde buna izin verilmez.

[6] C dilinde mantıksal verileri temsil edecek özel bir veri tipi yoktur, bu tip C++ dilinde getirilmiştir; yani `bool`, `true` ve `false` sözcükleri geçerli C sözcükleri değildir.

2.5 Değişmezler

Programda kullanılan bazı bilgiler de programın farklı çalışmaları arasında değer değiştirmezler. Örnekteki `PI` sayısı ve dairenin çevresinin hesaplanmasında kullanılan 2 sayısı bu tipten büyüklüklerdir. Değişmezlerin bazılarına isim vermek birtakım kolaylıklar sağlar:

- Anlaşılırlığı artırır. Programın içinde 3.14 yazmak yerine `PI` yazmak programı okuyan birinin bu sayının neye karşı düştüğünü daha kolay anlamasını sağlar.
- Değiştirmek kolay olur. Diyelim programın geliştirilmesinin ileri aşamalarında 3.14 değerinin yetersiz kaldığına ve 3.14159 değerinin daha uygun olduğuna karar verirsek kod içinde tek bir noktada değiştirmek yeterli olur. Öbür türlü, kodun içindeki bütün 3.14 sayılarının yerine 3.14159 yazmamız gerekir. Daha da kötüsü, kodun içindeki başka büyüklükleri gösteren 3.14 değerlerinin de geçmesi olasılığıdır. Bu durumda bütün 3.14 değerlerini tek tek inceleyerek değiştirilip değiştirilmeyeceğine karar vermek gerekir.

Değişmezler iki şekilde tanımlanabilirler:

1. `#define` bildirimiyle tanımlama. Bu yöntem bir değere bir isim vermekte kullanılır. Örnekte yapılan 3.14 sayısına `PI` ismini vermektir. Bu bildirimin sonucu, programcının kod içinde `PI` geçen her yere kendisinin 3.14 değerini yazmış olmasıyla aynıdır. Bu şekilde tanıtılan değişmezlere gelenek olarak tamamı büyük harflerden oluşan isimler verilir.

   Bu bildirim yönteminde değişmezlerin tipleri ayrıca belirtilmez. Tamsayı değişmezler `int` tipine sığmıyorlarsa `long int` varsayılırlar. Değerlerinin sonuna `l` ya da `L` harfleri eklenirse `long`, `u` ya da `U` eklenirse `unsigned` belirteci seçilmiş olur.

   ```
   #define MAXSHORT 0x7FFF
   #define MAXUSHORT 65535U
   ```

   Kesirli değişmezlerin değerlerinin sonuna `f` ya da `F` eklenmemişse `double` tipinden oldukları varsayılır. `l` ya da `L` eklenirse `long double` tipinden olurlar.

   ```
   #define EULER 2.81782F
   #define PERCENT 1E-2
   ```

2. Değişken tanımına benzer şekilde ancak veri tipinin önüne `const` niteleyicisi koyarak tanımlama. Böylece değeri değiştirilemeyen bir değişken tanımlanmış olur. Bu şekilde tanımlanan değişmezlere büyük harflerden oluşan isimler verilmez. Örnekteki `PI` değişmezinin bu yöntemle tanımı şöyle olurdu:

   ```
   const float pi = 3.14;
   ```

Her iki tanımda da değişmez olarak bildirilmiş bir büyüklüğe değer atanmaya çalışırsa derleyici hata verir.[7]

[7] C dili `const` ile tanımlanmış değişmezlerin değerlerinin değiştirilebilmesine izin verir. Bu tipten bir değişkene atama yapılmak istendiğinde derleyici hata değil, yalnızca bir uyarı üretecektir.

2.6 Aritmetik Deyimler

Aritmetik deyimlerde kullanılabilecek işlemler şunlardır:

- Toplama: `+` işleciyle gerçeklenir.
- Çıkartma: `-` işleciyle gerçeklenir.
- Çarpma: `*` işleciyle gerçeklenir.
- Bölme: `/` işleciyle gerçeklenir.
- Kalan: `%` işleciyle gerçeklenir. Yalnızca iki tamsayı arasında yapılabilir, kesirli sayılarda tanımlı değildir.

Bunların yanısıra matematik kitaplığında yer alan fonksiyonlar da aritmetik deyimlerde yer alabilirler (bkz. Bölüm 2.9).

Komutların okunurluğunu artırmak amacıyla C programcılarının uydukları geleneklerden biri de, eşit işaretinin öncesinde ve sonrasında birer boşluk bırakmaktır. Benzer şekilde, deyimlerde yer alan işleçlerin (örnekte `+` simgesi) önce ve sonralarında da birer boşluk bırakılır. *Gelenek*

C'de deyimlerin hesaplanmasında izlenen öncelik sırası, matematikten alışık olunan sıradır. Yüksek öncelikliden alçak öncelikliye doğru öncelik grupları şöyledir:

1. Ayraç içindeki deyimler
2. Sayı işareti belirten `+` ve `-` işlemleri, artırma, azaltma
3. Çarpma, bölme, kalan
4. Toplama, çıkarma

Eşit öncelik grupları kendi içlerinde soldan sağa doğru değerlendirilirler.

Örnek.

$$\frac{a+b+c+d+e}{5}$$

aritmetik deyimi C'de

```
(a + b + c + d + e) / 5
```

şeklinde yazılmalıdır. Ayraçlar kullanılmazsa ortaya çıkan

```
a + b + c + d + e / 5
```

C deyimi

$$a + b + c + d + \frac{e}{5}$$

aritmetik deyimine karşı düşer.

Örnek.

```
p * r % q + w / x - y
```

deyimi şu sırayla hesaplanır:

```
t1: p * r
t2: t1 % q
t3: w / x
t4: t2 + t3
t5: t4 - y
```

2.7 Tip Dönüşümleri

İşleme giren sayıların her ikisi de tamsayı ise sonuç da tamsayı olur. Sayılardan herhangi birinin kesirli sayı olması durumunda sonuç da kesirli sayı olacaktır. Bu durum bölme işleminde dikkat edilmesi gereğini doğurur. Bölme işlemine giren her iki sayı da tamsayı ise sonuç da tamsayı olacaktır, yani sonucun varsa kesir kısmı atılacaktır. Örneğin `14 / 4` deyiminin sonucu 3.5 değil 3 olacaktır. İşleme giren her iki sayı da tamsayı ise ve sonucun kesir kısmının yitirilmemesi isteniyorsa sayılardan en az birinin kesirli sayı olmasını sağlamak gerekir. Yukarıdaki örnek için çözüm şu yazılışlarla sağlanabilir:

```
14.0 / 4
14 / 4.0
14.0 / 4.0
```

Bir bölme işlemine giren iki değişkenin ikisinin de tamsayı tipinden olması durumunda sonuç yine tamsayı olacaktır. Örneğin

```
int num1 = 14, num2 = 4;
float quotient;

quotient = num1 / num2;
```

işlemleri sonucui `quotient` değişkeni 3.0 değerini alır. İşlemin doğru olarak yapılmasını sağlamak için `num1` ve `num2` değişkenlerinden en az birinin kesirli sayı tipine çevrilmesi gerekir. Bu işleme *tip zorlama* adı verilir ve bir deyimin başına ayraç içinde deyimin sonucunun alması istenen tipin adının yazılmasıyla yapılır. Yukarıdaki örneğin doğru çalışması için şu komutlardan herhangi biri kullanılabilir:

```
quotient = (float) num1 / num2;
quotient = num1 / (float) num2;
quotient = (float) num1 / (float) num2;
```

Birinci komutta yalnızca `num1` değişkeni, ikinci komutta yalnızca `num2` değişkeni, üçüncü komutta ise her ikisi birden kesirli sayıya çevrilmektedir. Buna karşılık

```
quotient = (float) (num1 / num2);
```

komutu ise önce bölmeyi sonra tip zorlamasını yapacağından `quotient` değişkenine yine 3.0 değerini atar.

Genel olarak, bir işleme giren sayılar farklı tiptense, bunların ortak bir tipe çevrilmeleri gerekir. Dar tipten geniş tipe geçiş işlemleri derleyici tarafından otomatik olarak yapılır. Örneğin, bir toplama işleminin işlenenlerinden biri tamsayı diğeri kesirli sayı ise tamsayı olanı kesirli sayıya çevrilir ve toplama yapılır. Aşağıdaki örnekte toplama işleminin ikinci işleneni kesirli sayı olduğundan toplamadan önce `num1` değişkeni kesirli sayıya çevrilecektir:

```
int num1 = 14;
float sum;

sum = num1 + 7.32;
```

Geniş tipten dar tipe geçişler ise bilgi yitirilmesine neden olabilirler. Örneğin kesirli sayı tipinden bir değişkenin tamsayı tipinden bir değişkene atanması sırasında sayının kesir kısmı yitirilebilir:

```
int num1;
float num2 = 65.717;

num1 = num2;
```

Derleyiciler bu gibi durumlarda genelde hata değil yalnızca uyarı üretirler.

2.8 Artırma / Azaltma

Bir atamanın sağ tarafındaki deyim, atamanın sol tarafındaki değişkeni içeriyorsa, yani

```
değişken = değişken ∘ deyim;
```

(∘ herhangi bir işlem simgesi olabilir) şeklindeyse bu komut

```
değişken ∘= deyim;
```

şeklinde kısaltılabilir. Örneğin:

```
a += 5;       // a = a + 5;
a /= 2;       // a = a / 2;
a *= c + d;   // a = a * (c + d);
```

Buna göre, bir değişkenin değerini artırmak ya da azaltmak için aşağıdaki komutlar kullanılabilir:

```
a = a + 1; a += 1;
a = a - 1; a -= 1;
```

Ancak artırma ve azaltma işlemleri programlarda sıkça gereksinim duyulan işlemlerden olduklarından C'de bunlar için özel işleçler tanımlanmıştır. `++` işleci, önüne ya da arkasına yazıldığı değişkenin değerini 1 artırır; `--` işleciyse önüne ya da arkasına yazıldığı işlecin değerini 1 azaltır. Her iki işleç de yalnızca tamsayı veri tipleri ile çalışırlar ve basit kullanımda işlecin değişkenin önüne mi arkasına mı yazıldığının bir önemi yoktur:[8]

```
a++;
a--;
++a;
--a;
```

2.9 Matematik Kitaplığı

ANSI standardı, bir C derleme ortamının matematik kitaplığında bulunması zorunlu olan fonksiyonları belirler. Bu fonksiyonların kullanılabilmesi için programların başında `math.h` başlık dosyasının alınması gerekir.

Matematik kitaplığının en sık kullanılan fonksiyonları Tablo 2.1'de verilmiştir. Bütün bu fonksiyonlardaki `x` ve `y` giriş parametreleri `double` tipindendir ve çıkış parametresi de `double` tipinden olacaktır.

[8] Artırma ve azaltma işlemleri başka işlemlerle birlikte kullanılabilir. Sözgelimi bir artırma işlemiyle bir atama işlemi aynı komut içerisinde yapılabilir. Böyle durumlarda artırma/azaltma işlecinin değişken adının önüne ya da arkasına yazılması önem kazanır. Önüne yazıldığında önce artırma, sonra atama yapılacaktır. Arkasına yazıldığındaysa önce atama, sonra artırma yapılır. Yani

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

komutu

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

koduna karşı düşerken

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

komutu

Adı	İşlevi
`sin(x) cos(x) tan(x)`	trigonometrik fonksiyonlar
`asin(x) acos(x) atan(x)`	ters trigonometrik fonksiyonlar
`sinh(x) cosh(x) tanh(x)`	hiperbolik trigonometrik fonksiyonlar
`exp(x)`	e^x
`log(x) log10(x)`	e ve 10 tabanında logaritma fonksiyonları
`pow(x,y)`	x^y
`sqrt(x)`	$\sqrt{x}$
`floor(x) ceil(x)`	alt ve üst sınır fonksiyonları: $\lfloor x \rfloor$ $\lceil x \rceil$
`fabs(x)`	$\lvert x \rvert$

Tablo 2.1: Matematik kitaplığı fonksiyonları.

Trigonometrik fonksiyonlarda giriş parametresi derece değil radyan cinsinden verilmelidir; örneğin `sin(30)` deyimi 0.5 değil -0.988032 değerini üretir. 30 derecenin sinüsünü almak için `sin(30*3.141529/180)` ya da `sin(3.141529/6)` deyimini yazmak gerekir.

Alt ve üst sınır fonksiyonlarının çalışmasında giriş parametresinin pozitif ya da negatif olmasına dikkat edilmelidir. Alt sınır fonksiyonu her zaman o sayıdan daha küçük olan ilk tamsayıyı, üst sınır fonksiyonu ise o sayıdan daha büyük olan ilk tamsayıyı verir.

- `floor(3.1)` → `3`, `floor(-3.1)` → `-4`
- `ceil(3.1)` → `4`, `ceil(-3.1)` → `-3`

Tamsayılar için de tamsayı giriş parametresi alan ve tamsayı çıkış parametresi üreten `abs` isimli bir mutlak değer fonksiyonu vardır.

2.10 Giriş / Çıkış

C dilinde kullanıcıyla iletişimi sağlayan işlemler giriş/çıkış birimleri üzerinden yapılır (bkz. Bölüm 1.5). Girdiler `cin` biriminden alınırken, çıktılar `cout`, hatalar da `cerr` biriminebirimine gönderilir.[9]

Örnekteki

```
cin >> radius;
```

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

koduna karşı düşer.

[9] C dilinde `cin`, `cout` ve `cerr` birimleri yoktur, bunların yerine standart girdi/çıktı kitaplığındaki fonksiyonların kullanılması gerekir (bkz. Ek 8).

komutu sonucunda `radius` değişkeni kullanıcının tuştakımından yazdığı sayının değerini alır. Birden fazla değişken birlikte okunmak isteniyorsa bunlar birbiri ardına $>>$ işaretleriyle belirtilebilir. Sözgelimi, iki tane sayı değeri okunacak ve kullanıcının yazdığı birinci sayının değeri `num1`, ikinci sayının değeriyse `num2` değişkenine aktarılacaksa:

```
cout << "Sayıları yazınız: ";
cin >> num1 >> num2;
```

gibi bir program parçası kullanılabilir. Bu örnekte kullanıcının sayıları yazarken aralarında bir boşluk bırakması gerekir.

Çıkış birimlerine katarlar ya da deyimler gönderilebilir. Katarlar oldukları gibi yazılırken deyimlerin değerleri görüntülenir. Örnekteki

```
cout << "Alanı: " << area << endl;
```

komutu ekrana önce `"Alanı: "` katarını, daha sonra `area` değişkeninin değerini yazar ve yeni satıra geçer. Programda `circum` ve `area` değişkenleri tanımlanmadan ve atama komutları kullanılmadan çıktı işlemleri

```
cout << "Çevresi: " << 2 * PI * radius << endl;
cout << "Alanı: " << PI * radius * radius << endl;
```

komutlarıyla da yapılabilirdi.

Uygulama: Geliştirme Ortamı

Bu uygulamada basit giriş/çıkış işlemlerine ve matematik kitaplığının kullanımına örnek verilmesi istenmektedir. Grafik bir editör (kate) tanıtılacak ve C derleyicisinin nasıl kullanılacağı öğretilecektir. Basit hatalarda derleyicinin hangi mesajları ürettiği gözlenecektir.[10]

Örnek 2. Formül Hesabı

$y = e^{-\pi x}$ formülünde x değerini kullanıcıdan alarak y değerini hesaplayan ve ekrana yazdıran bir program yazılması isteniyor.

- Örnekte verilen programı yazın, derleyin ve çalıştırın. `x` için 0.002 değerini verdiğinizde `y`'nin aldığı değeri gözleyin.
- `PI` değişmezinin tanımını 3.14 yerine 3.14159 olarak verin ve aynı `x` değerinde `y`'nin aldığı değeri gözleyin.
- Değişken tanımlarının yapıldığı

[10] Unix işletim sistemlerinde derleme aşamaları ile ilgili bilgi için bkz. Ek B.2.

Örnek 2 $y = e^{-\pi x}$ formülünü hesaplayan program.

```
#include <iostream>             // cout,cin.endl
#include <stdlib.h>             // EXIT_SUCCESS
#include <math.h>               // exp

using namespace std;

#define PI 3.14

int main(void)
{
    float x, y;

    cout << "x: ";
    cin >> x;
    y = exp(-PI * x);
    cout << "y: " << y << endl;
    return EXIT_SUCCESS;
}
```

```
    float x, y;
```

satırındaki 'x' harfini 'X' ile değiştirin ve derleme sonucunda oluşan hatayı gözleyin.

- Örnekteki

```
    cin >> x;
```

satırının ardına

```
    PI = 3.14159;
```

komutunu ekleyin ve derleme sonucunda oluşan hatayı gözleyin.

- Bir önceki adımda yaptığınız değişikliği silmeden, örnekte `PI` değişmezinin tanımlandığı `#define` satırını silin ve

```
    float x, y;
```

satırının ardına

```
    const float PI = 3.14;
```

değişmez tanımını ekleyin. Derleme sonucunda oluşan hatayı gözleyin.

- Baştaki `#include <math.h>` satırını silin ve derleme sonucunda oluşan hatayı gözleyin.

Sorular

1. Çalıştığınız geliştirme ortamında `int` tabanlı veri tiplerinin değer aralıklarını belirleyin ve bu aralıklar dışında bir değer atandığında nasıl bir sonuç elde edildiğini gözleyin.

2. Aşağıdaki deyimlerin sonuçlarını bulun:

```
8 + 17 % 3 * 5
45 / (4 + 7) - 5.0 / 2
```

3. Aşağıdaki aritmetik deyimleri gerçekleyen C deyimlerini yazın:

$$k \leftarrow \frac{\frac{1}{2ab} - cd\frac{ef-g}{h}}{x(y+z)}$$

4. Aşağıdaki C deyimlerinin gerçeklediği aritmetik deyimleri yazın:

```
a + 3 * b - (c + d) / 2 * y;
```

Bölüm 3

Akış Denetimi

Bu bölümde blok yapılı programlamada gereken seçim ve yineleme yapılarının C dilinde nasıl gerçeklendikleri üzerinde durulacaktır.

Örnek 3. İkinci Derece Denklem Kökleri

İkinci dereceden ($ax^2 + bx + c = 0$ şeklinde) bir denklemin köklerinin yerlerini belirleyen bir program yazılması isteniyor. Diskriminantın $b^2 - 4ac$ şeklinde tanımlandığını ve köklerin

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

formülüne göre hesaplanacağını gözönünde bulundurarak, program denklemin katsayılarını kullanıcıdan aldıktan sonra aşağıdaki işlemleri gerçekleştirecektir:

- Denklemin gerçel kökü yoksa (diskriminant negatifse) bu durum kullanıcıya bildirilir.
- Denklemin kökleri çakışıksa (diskriminant sıfırsa) bu durumu kök değeriyle birlikte kullanıcıya bildirilir.
- Denklemin iki farklı gerçel kökü varsa (diskriminant pozitifse) bunlar hesaplanarak ekrana çıkartılır.

Programın örnek bir çalışmasının ekran çıktısı Şekil 3.1'de verilmiştir

```
a, b ve c katsayılarını yazınız: -2 1 9.2
-1.90928 ve 2.40928 noktalarında iki gerçel kökü var.
```

Şekil 3.1: Örnek 3 ekran çıktısı.

Örnek 3 İkinci dereceden bir denklemin köklerini bulan program.

```
#include <iostream>             // cout,cin,endl
#include <stdlib.h>             // EXIT_SUCCESS
#include <math.h>               // sqrt

using namespace std;

int main(void)
{
    float a, b, c;
    float disc;
    float x1, x2;

    cout << "a, b ve c katsayılarını yazınız: ";
    cin >> a >> b >> c;
    disc = b * b - 4 * a * c;
    if (disc < 0)
      cout << "Gerçel kök yok." << endl;
    else {
      if (disc == 0) {
        x1 = -b / (2 * a);
        cout << x1 << " noktasında çakışan iki kök var. " << endl;
      } else {
        x1 = (-b + sqrt(disc)) / (2 * a);
        x2 = (-b - sqrt(disc)) / (2 * a);
        cout << x1 << " ve " << x2
             << " noktalarında iki gerçel kök var." << endl;
      }
    }
    return EXIT_SUCCESS;
}
```

3.1 Koşul Deyimleri

Blok yapılı programlamadaki seçim ve yineleme yapılarında bir koşula göre bir karar verilmesinin sağlanması gerektiği görülmüştü (bkz. Bölüm 1.3). Koşullar, koşul deyimleriyle gösterilirler. C dilinde bir koşul deyimi, iki sayısal büyüklüğün (aritmetik deyimin) karşılaştırılması ile oluşturulur ve mantıksal bir değer (doğru ya da yanlış) üretir. Diğer bütün veri tipleri gibi mantıksal değerler de sayılarla temsil edilirler; yanlış değeri 0, doğru değeriyse 1 sayısıyla gösterilir.

3.1.1 Karşılaştırma İşlemleri

İki sayı değeri arasında şu karşılaştırma işlemleri yapılabilir:

- Eşitlik: `==` işleciyle gerçeklenir. İşlecin solundaki değerle sağındaki değer aynıysa doğru, farklıysa yanlış sonucunu üretir.
- Farklılık: `!=` işleciyle gerçeklenir. Eşitlik karşılaştırmasının tersidir. İşlecin solundaki değerle sağındaki değer farklıysa doğru, aynıysa yanlış sonucunu üretir.
- Küçüklük: `<` işleciyle gerçeklenir. İşlecin solundaki değer sağındaki değerden küçükse doğru, küçük değilse yanlış sonucunu üretir.
- Büyüklük: `>` işleciyle gerçeklenir. İşlecin solundaki değer sağındaki değerden büyükse doğru, büyük değilse yanlış sonucunu üretir.
- Küçüklük veya eşitlik: `<=` işleciyle gerçeklenir. Büyüklük karşılaştırmasının tersidir. İşlecin solundaki değer sağındaki değerden küçük veya eşitse doğru, büyükse yanlış sonucunu üretir.
- Büyüklük veya eşitlik: `>=` işleciyle gerçeklenir. Küçüklük karşılaştırmasının tersidir. İşlecin solundaki değer sağındaki değerden büyük veya eşitse doğru, küçükse yanlış sonucunu üretir.

Örnekler

- Yıl 4'e kalansız bölünebiliyor mu?

```
(year % 4) == 0
```

- Yaş 18'den büyük ya da eşit mi?

```
age >= 18
```

Karşılaştırma işlemleri tam ya da kesirli sayıların karşılaştırılmasında kullanılabileceği gibi, harflerin abecedeki sıralarına göre karşılaştırılmasında da kullanılabilir. Sözgelimi

```
'm' < 'u'
```

koşul deyimi doğru değerini üretir. Ancak bu işlem yalnızca İngilizce harfler arasında doğru sonuç üretecektir. Ayrıca, büyük-küçük harf karşılaştırmalarında da abece sırasından farklı sonuçlar çıkabilir. Örneğin şu koşul deyimi doğru değerini üretir:

```
'Z' < 'a'
```

Bu nedenlerle, simgelerin karşılaştırılması yalnızca İngilizce harfler için ve ikisi de büyük ya da ikisi de küçük harflerin karşılaştırılmasında kullanılmalıdır.[1] Ayrıca, karşılaştırma işleçleriyle katarlar abece sırasına göre karşılaştırılamaz, yani aşağıdaki komut geçerli değildir (doğru ya da yanlış sonucunu üretmez, derleyicinin hata vermesine neden olur):

```
"dennis" < "ken"
```

Kesirli sayıların eşitlik açısından karşılaştırılmasında da dikkatli olunması gerekir. Kesirli sayılar bilgisayarlarda tam olarak temsil edilemeyebileceklerinden bunlar arasında eşitlik karşılaştırması yapmak hatalı sonuçlar doğurabilir. Genellikle farklarının mutlak değerinin yeterince küçük bir sayı olup olmadığına bakmak daha emin bir yöntemdir. Örneğin

```
f1 == f2
```

yerine aşağıdaki gibi bir koşul yazılabilir:

```
fabs(f1 - f2) < 1E-6
```

3.1.2 Mantıksal İşlemler

Bazı durumlarda tek bir karşılaştırma işlemi bütün koşul deyimini oluşturmak için yeterli olmaz. Örneğin bir insanın yaşının 18 ile 65 arasında olup olmadığını sınamak istiyorsanız, bunu tek bir karşılaştırma işlemiyle yapamazsınız (`18 <= x < 65` gibi bir deyim yazılamaz). Böyle durumlarda, karşılaştırma işlemleri mantıksal işlemlerle bağlanarak karmaşık koşul deyimleri üretilebilir. Üç tane mantıksal işlem vardır:

- DEĞİL işlemi: `!` işleciyle gerçeklenir. Önüne yazıldığı koşul deyimini değiller, yani doğruysa yanlış, yanlışsa doğru değerini üretir (bkz. Tablo 3.1).

`koşul`	`!(koşul)`
doğru	yanlış
yanlış	doğru

Tablo 3.1: Değil işlemi doğruluk tablosu.

Örnek Yaş 18'den küçük değil:

```
!(age < 18)
```

koşul1	koşul2	(koşul1) && (koşul2)
doğru	doğru	doğru
doğru	yanlış	yanlış
yanlış	doğru	yanlış
yanlış	yanlış	yanlış

Tablo 3.2: VE işleminin doğruluk tablosu.

- VE işlemi: `&&` işleciyle gerçeklenir. Bağladığı koşulların hepsi doğruysa doğru, en az biri yanlışsa yanlış değerini üretir (bkz. Tablo 3.2).

  Örnek Yaş 18'den büyük veya eşit ve 65'den küçük:

  ```
  (age >= 18) && (age < 65)
  ```

- VEYA işlemi: `||` işleciyle gerçeklenir. Bağladığı koşulların hepsi yanlışsa yanlış, en az biri doğruysa doğru değerini üretir (bkz. Tablo 3.3).

koşul1	koşul2	(koşul1) || (koşul2)
doğru	doğru	doğru
doğru	yanlış	doğru
yanlış	doğru	doğru
yanlış	yanlış	yanlış

Tablo 3.3: VEYA işleminin doğruluk tablosu.

  Örnek Yaş 18'den küçük veya 65'den büyük veya eşit:

  ```
  (age < 18) || (age >= 65)
  ```

Örnek Bir yılın artık yıl olup olmadığını belirleyen koşul deyimi. Sonu 00 ile biten (100'e kalansız bölünen) yıllar dışındaki yıllar 4 sayısına kalansız bölünüyorlarsa artık yıl olurlar. Sonu 00 ile bitenler ise 400 sayısına kalansız bölünüyorlarsa artık yıldırlar. Bunların dışında kalan yıllar artık yıl değildir. Sözgelimi, 1996, 2000, 2004 ve 2400 yılları artık yıldır ama 1997, 2001, 1900 ve 2100 yılları artık yıl değildir.

```
((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)
```

Ayraçlar kullanılmadığında mantıksal işleçlerin öncelikleri yüksek öncelikliden başlayarak DEĞİL, VE, VEYA sırasıyladır. Buna göre yukarıdaki örnek

```
(year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0)
```

ya da

[1] Aslında burada karşılaştırılan bu iki simgenen ASCII kodlamasındaki sıralarıdır.

```
(year % 400 == 0) || (year % 4 == 0) && (year % 100 != 0)
```

biçiminde de yazılabilirdi.[2]

3.2 Seçim

C dilinde seçim yapıları, `if/else` bloklarıyla gerçeklenir. Verilen koşul deyimi doğru değerini alıyorsa `if` ile başlayan blok (`blok1`), yanlış değerini alıyorsa `else` ile başlayan blok (`blok2`) yürütülür:

```
if (koşul) {
  blok1;
} else {
  blok2;
}
```

Örnekte içiçe iki seçim yapısı vardır. İçteki seçim yapısı şu şekildedir:

```
if (disc == 0) {
  x1 = -b / (2 * a);
  cout << x1  << " noktasında çakışan iki kök var." << endl;
} else {
  x1 = (-b + sqrt(disc)) / (2 * a);
  x2 = (-b - sqrt(disc)) / (2 * a);
  cout << x1 << " ve " << x2
       <<  " noktalarında iki gerçel kök var." << endl;
}
```

Burada `disc` değişkeninin değeri 0 ise çakışık köklerle ilgili işlemler yapılır (iki komuttan oluşan bir blok); değilse farklı iki gerçel kök bulunması durumundaki işlemler yapılır (üç komuttan oluşan bir blok).

Seçim yapılarında bir `else` bloğu bulunması zorunlu değildir, yani yapı

```
if (koşul) {
  blok;
}
```

[2]Birden fazla koşuldan oluşan mantıksal deyimler değerlendirilirken sonuç belli olduğu zaman deyimin geriye kalanı hesaplanmaz. Örneğin

```
(year % 4 == 0) && (year % 100 != 0)
```

deyiminde `(year % 4 == 0)` koşulu yanlış değerini verirse deyimin geri kalanına bakılmaya gerek kalmayacağından `(year % 100 != 0)` koşulu sınanmaz. Benzer şekilde

```
(year % 4 == 0) || (year % 100 != 0)
```

deyiminde `(year % 4 == 0)` koşulu doğru değerini verirse yine ikinci koşul sınanmaz.

şeklinde olabilir. Bu durumda koşul doğruysa blok yürütülür, değilse hiçbir şey yapılmaz.

Bir blok tek bir komuttan oluşuyorsa bloğun süslü ayraçlar ile sınırlanması zorunlu değildir. Örnekteki dış seçim yapısında dikkat edilirse koşulun doğru olması durumunda yürütülecek blokta süslü ayraçlar kullanılmamıştır:

```
if (disc < 0)
  cout << "Gerçel kök yok." << endl;
else {
  ...
  ...
  ...
}
```

Burada istenseydi birinci blok da süslü ayraçlarla belirtilebilirdi:

```
if (disc < 0) {
  cout << "Gerçel kök yok." << endl;
} else {
  ...
  ...
  ...
}
```

Gelenek olarak tek komutlu bloklarda süslü ayraçlar kullanılmaz, yani örnek programda kullanılan yazılış şekline uyulur. Ancak bu durumda süslü ayraç kullanılmayan bloğun tek komut içermesine çok dikkat etmek gerekir. Diğer bir gelenek de, koşulun doğru ya da yanlış olmasına göre yürütülecek bloklardan biri diğerine göre çok kısaysa, kısa olan blok üste gelecek (koşulun doğru olması durumunda yürütülecek) şekilde düzenlenmesidir. Örnekte bu geleneğe uyulmamış olsaydı dış seçim yapısı şu şekilde yazılabilirdi: *Gelenek*

```
if (disc >= 0) {
  ...
  ...
  ...
} else
  cout << "Gerçel kök yok." << endl;
```

Seçim ve ileride görülecek yineleme yapıları bir bütün olarak tek bir komut olarak değerlendirilirler. Dolayısıyla bir blokta yer alan tek yapı başka bir seçim ya da yineleme bloğuysa süslü ayraçların kullanılması zorunlu değildir. Yani şu blok

```
if (koşul1) {
    if (koşul2)
        blok;
}
```

şu şekilde de yazılabilir:

```
if (koşul1)
    if (koşul2)
        blok;
```

Süslü ayraçlar kullanılmadığında içiçe `if / else` yapılarında belirsizlikler oluşabilir. Girintilenmeden yazılmış şu örneğe bakalım:

```
if (koşul1)
if (koşul2)
blok1;
else
blok2;
```

Burada `blok2` hangi koşul sağlanmazsa yürütülecektir? C dilinin bu konudaki kuralı `else`'in kendinden önce gelen son `if` koşuluna bağlandığıdır, yani örnekte *blok2*, `koşul1` doğru ve `koşul2` yanlışsa yürütülür. Doğru bir girintilemeyle yazılırsa

```
if (koşul1)
    if (koşul2)
        blok1;
    else
        blok2;
```

Yine de karışıklığa neden olmaması için böyle durumlarda süslü ayraçlar kullanmak daha doğrudur.

DİKKAT Herhangi bir aritmetik deyim koşul değeri olarak değerlendirilebilir; deyimin sonucu 0 ise yanlış, 0'dan farklı ise doğru olduğu varsayılır. Buna göre, sözgelimi 8 - 2 deyimi doğru, 8 - 2 * 4 deyimiyse yanlış bir koşul olarak değerlendirilir. Bu davranış özellikle eşitlik karşılaştırmalarında hataya yol açabilir. En sık yapılan C hatalarından biri bir eşitlik karşılaştırmasında `==` yerine `=` işareti kullanılmasıdır.

```
age = 12;
...
...
if (age = 18)
  blok1;
else
  blok2;
```

Yukarıdaki programda koşulun sınanması sırasında eşitlik işlemi değil atama işlemi belirtildiğinden `age` değişkenine 18 atanır, deyimin değeri 18 olarak bulunur ve 0'dan farklı olduğu için doğru sayılarak `blok1` yürütülür. Oysa `==` simgesi kullanılsaydı `age` değişkeninin değeri değişmez ve `blok2` yürütülürdü.

3.2.1 Koşullu İşleç

Koşullu işleç, bir koşulun gerçekleşip gerçekleşmemesine göre iki deyimden birini seçer.

```
deyim1 ? deyim2 : deyim3
```

Burada öncelikle ***deyim1*** değerlendirilir. Sonuç doğruysa ***deyim2***, yanlışsa ***deyim3*** seçilir.

Basit bir örnekle, iki sayıdan küçüğünü seçmek için şöyle bir kod kullanılabilir:

```
if (x < y)
  z = x;
else
  z = y;
```

Aynı işlem koşullu işleç kullanılarak şöyle de yazılabilirdi:

```
z = x < y ? x : y;
```

Adından da anlaşılabileceği gibi, koşullu işleç bir işleçtir, yani birtakım büyüklükler üzerinde bir işlem yapar ve belli bir tipten bir sonuç üretir. Bir seçim yapısı değildir, yani programın akışının nasıl devam edeceği üzerinde bir etki yaratmaz; akış bir sonraki komutla devam eder.

Örnek 4. İşlem Seçimi

İşlemi ve işlenenleri kullanıcının belirttiği hesaplamayı yaparak sonucu ekrana yazan bir program yazılması isteniyor. Programın örnek bir çalışmasının ekran çıktısı Şekil 3.2'de verilmiştir.

```
İşlemi yazınız: 18 / 5
18/5 işleminin sonucu: 3
```

Şekil 3.2: Örnek 4 ekran çıktısı.

3.3 Çoklu Seçim

Bir deyimin çok sayıda değer içinden hangisini almış olduğunu sınamak istiyorsak yazacağımız `if` kodu uzun ve çirkin bir görünüm alır. Örneğimizde yapılacak işlemin hangisi olduğunu anlamak için yazılacak `if` kodu şu tip bir şey olurdu:

```
if (op == '+') {
  ...
} else {
```

Örnek 4 Kullanıcının belirttiği işlemi yapan program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

int main(void)
{
    int num1, num2, result;
    char op;

    cout << "İşlemi yazınız: ";
    cin >> num1 >> op >> num2;
    switch (op) {
        case '+': result = num1 + num2;
                  break;
        case '-': result = num1 - num2;
                  break;
        case '*': result = num1 * num2;
                  break;
        case '/': result = num1 / num2;
                  break;
        case '%': result = num1 % num2;
                  break;
        default:  cout << "Böyle bir işlem yok." << endl;
                  return EXIT_FAILURE;
    }
    cout << num1 << op << num2 << " işleminin sonucu: " << result << endl;
    return EXIT_SUCCESS;
}
```

```
  if (op == '-') {
    ...
  } else {
    if (op == '*') {
      ...
```

Seçim bloklarının tek bir komut olarak değerlendirilmesi çoklu karşılaştırmalarda da daha kolay bir yazım olanağı sağlar. Buna göre, yukarıdaki çoklu karşılaştırma yapısı şu şekilde de yazılabilir:

```
if (op == '+')
  ...
else if (op == '-')
  ...
else if (op == '*')
  ...
```

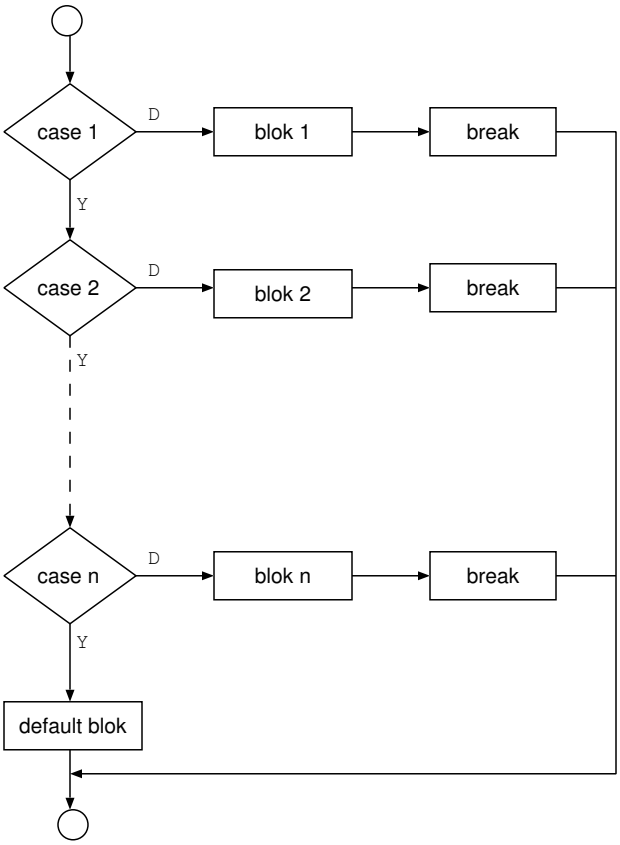
`switch` komutu bu tip karşılaştırmalar için daha okunaklı bir yapı sunar:

```
switch (deyim) {
  case değer_1: blok_1;
                break;
  case değer_2: blok_2;
                break;
  ...
  case değer_n: blok_n;
                break;
  default:      blok;
                break;
}
```

Bu komutun akış çizeneği Şekil 3.3'de görüldüğü gibidir. Deyim, önce birinci değerle karşılaştırılır ve eşitse buna ilişkin blok yürütülür ve `break` komutuyla `switch` yapısının dışına çıkılır. Birinci değere eşit değilse ikinci değerle karşılaştırılır. Karşılaştırmaların hiçbiri doğru değilse `default` bloğu yürütülür ve `switch` sona erer. Her `switch` yapısında bir `default` bloğu bulunması zorunlu değildir.

Burada dikkat edilmesi gereken bir nokta, deyimin diğer bir deyimle değil, bir değerle karşılaştırıldığıdır. Yani, karşılaştırılacak değerlerin yazıldığı deyimlerde değişkenler yer alamaz. Örneğin aşağıdaki yazımda deyim yazılışı geçerlidir ancak değerin yazılışı geçerli değildir:

```
switch (5 * x) {
  ...
  case y: ...
  ...
}
```



Şekil 3.3: `switch` komutunun akış çizeneği.

Karşılaştırılan deyim ve değerler yalnızca tamsayı ve simge tipinden olabilir (örnekte simge tipinden değerlerle karşılaştırma yapılmıştır). Örneğin aşağıdaki yazımlar geçerli değildir:

```
switch (name) {
  ...
  case "Dennis": ...
  ...
}

switch (x) {
  ...
  case 8.2: ...
  ...
}
```

Yine dikkat edilmesi gereken bir nokta, karşılaştırmanın yalnızca eşitlik üzerinden yapıldığıdır. Bu yazımda, deyimin değerden küçük ya da büyük olup olmadığı ya da diğer herhangi bir mantıksal deyimin doğruluğu belirtilemez. Ayrıca, yazımdan görülebileceği gibi, `switch` yapılarında bloklar süslü ayraçlar içine alınmaz.

C programlarında sıkça yapılan hatalardan biri bloklardaki `break` komutlarının unutulmasıdır. Örneğin birinci bloğun sonuna konması gereken `break` komutu unutulursa `switch` komutu Şekil 3.4'de görüldüğü gibi çalışır. Birinci karşılaştırma işlemi başarısız olursa bir sorun çıkmaz ama başarılı olursa *`blok 1`* yürütüldükten sonra *`blok 2`* de yürütülür ve `break` ile `switch` sonlanır. Yani, başarılı olan ilk karşılaştırmadan başlanarak `break` komutu görülene kadar gelen bütün bloklar yürütülür.[3]

Örnek programdaki hiçbir `break` komutu konmamış olsa işlemin toplama olduğu durumda önce toplama, sonra çıkartma, çarpma, bölme ve son olarak da kalan işlemi yapılır (yani yalnızca kalan işlemi geçerli olur), ayrıca da **"Böyle bir işlem yok."** denilerek herhangi bir sonuç görüntülenmeden programdan çıkılır. İşlem çıkartma olduğunda toplama kısmı atlanır, gerisi deminki gibi devam eder.

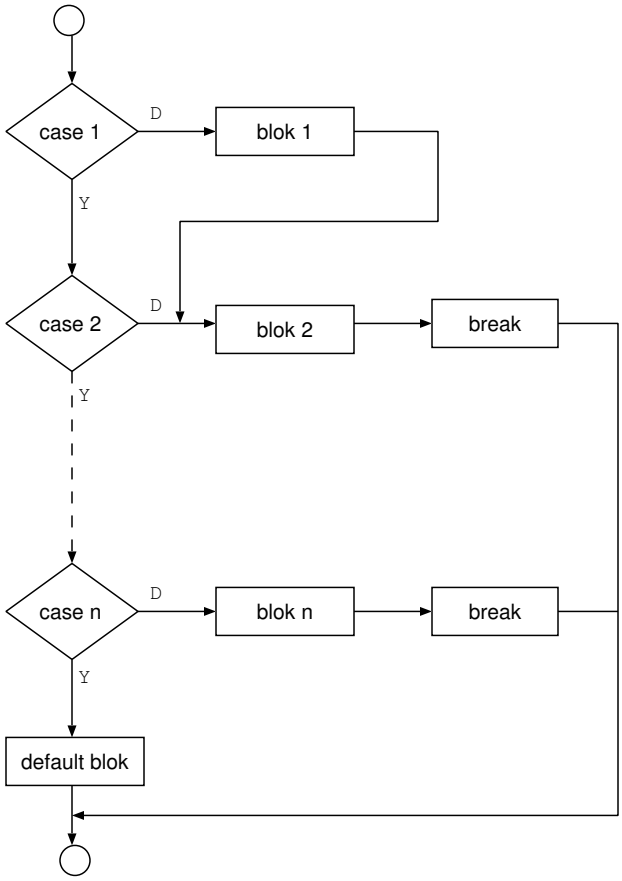
Uygulama: Seçim Yapıları

Bu uygulamada Windows ortamında bir tümleşik geliştirme ortamı (Visual C++ ya da Dev-C++) tanıtılacaktır. Hata ayıklayıcıda adım adım ilerleme ve değişken değerlerinin gözlenmesi gösterilecektir.

Örnek 5. Birim Dönüşümü

Bu örnekte İngiliz uzunluk ve sıcaklık ölçü birimlerinin metrik birimlere dönüşümünü yapan bir program yazılacaktır. inch biriminden verilen bir uzunluğu santimetre birimine çevirmek için

[3] Bu davranış, bazı durumlarda yararlı yönde de kullanılabilir (bkz. Örnek 10).



Şekil 3.4: `switch` komutunda `break` kullanılmazsa oluşan akış çizeneği.

$$1\,inch = 2.54\,cm$$

eşitliği ve Fahrenheit biriminde verilen bir sıcıklığı Celcius birimine çevirmek için de

$$C = \frac{5}{9}(F - 32)$$

formülü kullanılacaktır.

- Verilen örnek programı yazın, çalıştırın, birkaç değer için deneyin
- case 1 bloğunun sonundaki break komutunu silin ve programı çalıştırarak her iki dönüşüme de birer örnek deneyin
- cm->inch ve c->f dönüşümlerini yapmak üzere kodunuzu geliştirin
- $1ft = 12inch$ eşitliğini kullanarak ft cinsinden verilmiş bir uzunluğu mt cinsine çevirecek eklemeyi yapın
- mt->ft dönüşümünü yapacak eklemeyi yapın

Örnek 6. Euclides Algoritması

İki sayının en büyük ortak bölenini Euclides algoritmasını kullanarak bulan bir program yazılması isteniyor. Bu algoritmaya ilişkin akış çizeneği Şekil 1.17'de, programın örnek bir çalışmasının ekran çıktısı Şekil 3.5'de verilmiştir.

```
Sayıları yazınız: 9702 945
9702 ve 945 sayılarının en büyük ortak böleni: 63
```

Şekil 3.5: Örnek 6 ekran çıktısı.

3.4 Koşul Denetiminde Yineleme

C dilinde temel yineleme yapıları `while` sözcüğüyle kurulur:

```
while (koşul) {
  blok;
}
```

Örnek 5 İngiliz-metrik birim dönüşümü yapan program.

```
#include <iostream>             // cout,cin,endl
#include <stdlib.h>             // EXIT_SUCCESS,EXIT_FAILURE

using namespace std;

int main(void)
{
    int choice;
    float inch, cm, f, c;

    cout << "1. inch-cm" << endl;
    cout << "2. f-c" << endl;
    cout << "3. Çıkış" << endl;
    cout << "Seçiminiz: ";
    cin >> choice;

    switch (choice) {
        case 1:
            cout << "inch cinsinden uzunluk: ";
            cin >> inch;
            cm = 2.54 * inch;
            cout << cm << " cm" << endl;
            break;
        case 2:
            cout << "f cinsinden sıcaklık: ";
            cin >> f;
            c = (f - 32) * 5.0 / 9.0;
            cout << c << " C" << endl;
            break;
        case 3:
            return EXIT_SUCCESS;
        default:
            cout << "Böyle bir işlem yok." << endl;
            return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Örnek 6 İki sayının en büyük ortak bölenini bulan program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

int main(void)
{
    int num1, num2;
    int a, b, r = 1;

    cout << "Sayıları yazınız: ";
    cin >> num1 >> num2;

    a = num1 > num2 ? num1 : num2;
    b = num1 > num2 ? num2 : num1;

    while (b > 0) {
        r = a % b;
        a = b;
        b = r;
    }
    cout << num1 << " ve " << num2
         << " sayılarının en büyük ortak böleni: " << a << endl;
    return EXIT_SUCCESS;
}
```

Örnekteki `while` bloğu `b` değişkeninin değeri 0'dan büyük olduğu sürece yinelenecek, 0 olduğunda sona erecektir. Bu örnekte algoritmanın doğası gereği bu değişken eninde sonunda 0 değerini alacaktır; ancak yineleme yapıları kurarken döngünün her durumda mutlaka sonlanmasının sağlanmasına özellikle dikkat edilmelidir.

Benzer bir diğer yineleme yapısı ise `do-while` sözcükleriyle kurulabilir:

```
do {
  blok;
} while (koşul);
```

Bu yapının `while` yapısından en önemli farkı, koşulun doğru olup olmamasına bakılmaksızın bloğun en az bir kere yürütülmesidir. Örnekteki yineleme bölümü şu şekilde de yazılabilirdi:

```
do {
  r = a % b;
  a = b;
  b = r;
} while (b > 0);
```

Gelenek

Başlangıçta `num1` ve `num2` değişkenlerinden birinin 0 olması durumunda `b` değişkeni 0 değerini alırdı. İlk örnekte koşul sağlanmayacağı için en büyük ortak bölen 1 olarak bildirilirdi; ikinci örnekte ise sıfıra bölme yapmaya kalkılacağından bir çalışma zamanı hatası oluşurdu. Her ne kadar `while` ile kurulan yapılar ve `do-while` ile kurulan yapılar birbirlerinin eşi olacak şekilde düzenlenebilseler de, yineleme yapısı kurmak için genelde `while` kullanılması önerilir.

3.5 Döngü Denetimi

Yineleme yapılarında kullanılan diğer bir yöntemse, döngüyü bir sonsuz döngü olarak kurup döngüden çıkma durumu oluştuğunda `break` komutunu kullanmaktır:[4]

```
while (true) {
  ...
  if (<çıkma koşulu sağlanıyor>)
    break;
  ...       // çıkma koşulu sağlanmadığı zaman yapılacaklar
}
```

Çoklu seçim (`switch`) yapısında olduğu gibi burada da `break` komutu içinde bulunulan yapıdan çıkışı sağlar. Ancak örnekte görüldüğü gibi içinden çıkılan yapı `if` değil `while` yapısıdır. Yani, `break` komutu `switch`, `while`, `do-while` ve birazdan göreceğimiz `for` yapılarından çıkartır, `if` yapısından çıkartmaz.

İçiçe döngüler kullanıldığında, beklenebileceği gibi, `break` komutu yalnızca en içteki döngüden çıkartır ve bir üstteki döngünün sıradaki komutundan devam edilmesini sağlar:

[4] Bu tür bir yapı Örnek 10'da görülebilir.

```
while (true) {
  ...
  while (true) {
    ...
    if (<çıkma koşulu sağlanıyor>)
      break;
    ...
  }
  ... // üstteki break komutu sonucunda buraya gelinir
}
```

Döngülerde akış denetimine yönelik diğer bir komut ise `continue` komutudur. Bu komut döngünün geri kalan kısmının atlanarak döngü başına gidilmesini sağlar:

```
i = 0;
while (i < 50) {
  i++;
  ...
  if (<koşul>) {
    ...
    continue;
  }
  ...
}
```

Bu örnekte `if` ile belirtilen koşul sağlanıyorsa birtakım işlemlerin gerçekleştirilmesinden sonra döngünün başına dönülür. Yani `if` bloğundan sonra gelen komutlar atlanarak `i` değişkeninin bir sonraki değerinden işlemler sürdürülür.

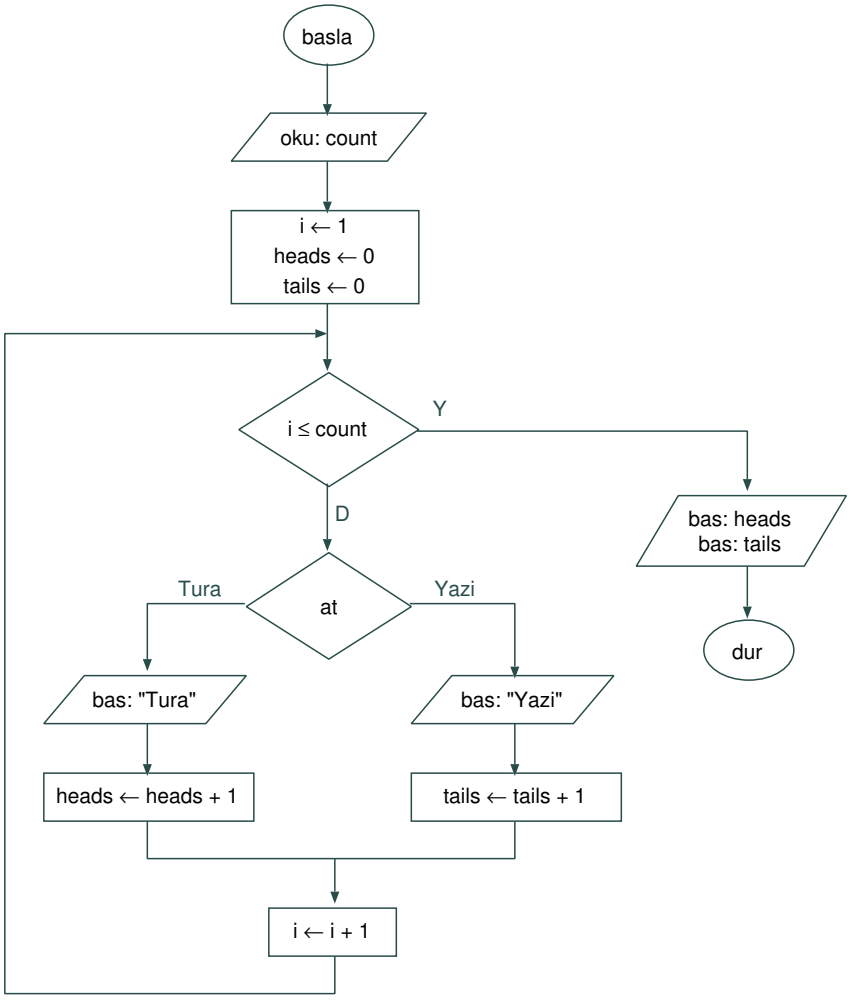
Örnek 7. Yinelemeli Yazı-Tura Atışı

Kullanıcının verdiği sayı kadar yazı-tura atarak yazı ve turaların kaçar kere geldiğini sayan ve sonuçları ekrana çıkartan bir program yazılması isteniyor. Bu programa ilişkin akış çizeneği Şekil 3.6'da, programın örnek bir çalışmasının ekran çıktısı Şekil 3.7'de verilmiştir.

3.6 Sayaç Denetiminde Yineleme

Örnekte yazı-tura simülasyonunun kullanıcının belirteceği sayıda yinelenmesi isteniyor. Bir bloğun belli sayıda yinelenmesi istendiğinde kullanılabilecek en uygun yapı `for` yapısıdır. Bir sayacın denetiminde yineleme yapmak için şunların belirtilmesi gerekir:

- Sayacın başlangıç değeri: Örnekte bu değer 1'dir.
- Kaça kadar sayılacağı: Örnekte bu değer kullanıcıdan alınan sayının tutulduğu `count` değişkeniyle belirlenir.



Şekil 3.6: Yinelemeli yazı-tura atışının simülasyonu.

```
Kaç kere atılacak? 7
Yazı
Tura
Tura
Yazı
Yazı
Yazı
Yazı
Tura sayısı: 2, Yüzdesi: %28.5714
Yazı sayısı: 5, Yüzdesi: %71.4286
```

Şekil 3.7: Örnek 7 ekran çıktısı.

Örnek 7 Yinelemeli yazı-tura atışı simülasyonu yapan program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS,srand,rand,RAND_MAX
#include <time.h>               // time

using namespace std;

int main(void)
{
    int count, i;
    float number;
    int heads = 0, tails = 0;

    cout << "Kaç kere atılacak? ";
    cin >> count;
    srand(time(NULL));
    for (i = 1; i <= count; i++) {
        number = (float) rand() / RAND_MAX;
        if (number < 0.5) {
            cout << "Tura" << endl;
            heads++;
        } else {
            cout << "Yazı" << endl;
            tails++;
        }
    }
    cout << "  Tura sayısı: " << heads
         << ", Yüzdesi: %" << 100.0 * heads / count << endl;
    cout << "  Yazı sayısı: " << tails
         << ", Yüzdesi: %" << 100.0 * tails / count << endl;
    return EXIT_SUCCESS;
}
```

- Kaçar kaçar sayılacağı: Örnekte sayacın birer birer artırılacağı belirtilmiştir.

`for` yapısı bu üç belirtimin aynı anda yapılmasına olanak sağlar. Ay içinde önce başlangıç değeri ataması, ikinci olarak hangi koşul sağlandığı sürece devam edileceği ve son olarak da sayacın nasıl artırılacağı belirtilir.

```
for (i = 1; i <= count; i++)
```

komutu şöyle okunabilir:

> `i` değişkenine 1 değerini ver ve bu değişkenin değeri `count` değişkeninin değerinden küçük veya eşit olduğu sürece bloğu her yürütüşünden sonra `i` değişkeninin değerini 1 artır.

`for` döngüleri `while` döngüleri şeklinde de yazılabilirler:

```
for (başlangıç_ataması; sürme_koşulu; artırma_komutu) {
  blok;
}
```

döngüsü

```
başlangıç_ataması;
while (sürme_koşulu) {
  blok;
  artırma_komutu;
}
```

döngüsüne eşdeğerlidir. Yine de sayaç denetiminde yinelemeler için `for`, koşul denetiminde yinelemeler için `while` kullanmak anlaşılırlık açısından daha doğrudur.

Bu yapıyla ilgili olarak bazı noktalara dikkat etmek gerekir:

- Döngü, belirtilen koşul sağlanmadığı zaman sonlanır ve programın yürütülmesi döngünün arkasından gelen komutla sürdürülür. Örnekte `i` değişkeninin değeri `count` değişkeninin değerinden büyük olduğu zaman döngü sona erer.
- Artırma işlemi döngü gövdesi yürütüldükten sonra yapılır. Örnekte döngünün gövdesi `i` değişkeninin 1, 2, ..., `count` değerleri için yinelenir, 1 değeri atlanmaz.
- Verilen başlangıç değeri döngü koşulunu sağlamıyorsa döngünün gövdesi hiç yürütülmez. Örneğin

  ```
  for (i = 1; i == 10; i++)
  ```

döngüsünde başlangıç değeri sürme koşulunu sağlamadığından (1== 10 doğru olmadığından) döngüye hiç girilmez.

- Artım miktarı için artırma işleci kullanılması zorunlu değildir, herhangi bir C deyimi kullanılabilir.

```
for (i = 1; i < count; i = i + 3)
```

döngüsü 1, 4, 7, 10, 13, ... şeklinde sayarken

```
for (i = 1; i < count; i = i * 2)
```

döngüsü 1, 2, 4, 8, 16, ... şeklinde sayar.

Rasgele Sayılar

Yazı-tura atışını temsil etmek için en kolay yol 0 ile 1 arasında bir rasgele sayı üretmek ve bu sayının 0.5'den küçük olması durumunu bir sonuca (diyelim tura), eşit ya da büyük olmasını da diğer sonuca (bu durumda yazı) atamaktır.

C standart kitaplığındaki `rand` fonksiyonu 0 ile `RAND_MAX` arasında bir rasgele tamsayı üretir. `RAND_MAX` standart kitaplıkta tanımlanmış bir değerdir ve sistemden sisteme farklılık gösterebilir. `rand` fonksiyonundan gelen sayı `RAND_MAX` değerine bölünürse 0 ile 1 arasında bir rasgele kesirli sayı elde edilir. Rasgele sayıların kullanımında daha çok 1 ile bir üst sınır arasında değer alacak bir rasgele tamsayıya gereksinim duyulur. Bu üst sınır `max` ile gösterilirse

```
1 + rand() % max
```

deyimi istenen türden bir sayı üretir (Neden?).

Rasgele sayılar, bir seri olarak üretilirler; yani her rasgele sayı seride bir önceki rasgele sayıdan üretilir. Bu serinin başlayabilmesi için ilk sayıyı üretmekte kullanılacak bir başlangıç değeri (*tohum*) verilmesi gerekir. `srand` fonksiyonu bu tohumun belirtilmesini sağlar. Aynı tohumla başlanırsa aynı seri üretilir. Her serinin programın çalışmasındaki bir senaryoya karşılık düştüğü düşünülürse istendiği zaman aynı senaryoyu üretebilmek programın hatalarının ayıklanması açısından yararlıdır.

Her defasında farklı bir tohumla başlamak için tohumun da her defasında farklı verilmesi gerekir. Standart kitaplıktaki `time` fonksiyonu, 1 Ocak 1970 tarihinden o ana kadar geçen saniyelerin sayısını verdiğinden her çağrılışında farklı bir tohum üretebilir. Bu fonksiyona `NULL` giriş parametresini göndermek yeterlidir.

Uygulama: Döngüler

Bu uygulamada Unix bir hata ayıklayıcı (ddd) tanıtılacaktır. Hata ayıklayıcıda adım adım ilerleme ve değişken değerlerinin gözlenmesi gösterilecektir.

Örnek 8. Seri Toplamı

Bu uygulamada görülecek programlarda, e^x fonksiyonunun hesaplanması için aşağıdaki seri toplamından yararlanılacaktır:

$$f(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

Birinci programda (Örnek 8) içiçe iki döngü vardır. İçerideki döngü o anda işlenmekte olan terimde gerekecek faktöryel değerinin hesaplanmasını sağlar. Dış döngüyse her yinelenişinde serinin o anki terimi hesaplayarak toplama ekler. Hesaplanan terim kullanıcının belirttiği hatadan küçük olduğunda değerin istenen duyarlılıkta hesaplandığına karar verilerek sonuç ekrana çıkartılır.

Örnek 8 e^x deyimini genel terimden giderek hesaplayan program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS
#include <math.h>               // pow

using namespace std;

int main(void)
{
    float x, error, term, result = 1.0;
    int i = 1, f;
    float fact;

    cout << "x: ";
    cin >> x;
    cout << "Hata: ";
    cin >> error;

    term = error + 1;
    while (term > error) {
        fact = 1.0;
        for (f = 2; f <= i; f++)
            fact = fact * f;
        term = pow(x, i) / fact;
        result = result + term;
        i++;
    }
    cout << "Sonuç: " << result << endl;

    return EXIT_SUCCESS;
}
```

İkinci programdaysa (Örnek 9) aynı işlemin nasıl daha etkin (daha az hesap yüküyle) yapılabileceği görülecektir. Serinin genel teriminin $a_i = \frac{x^i}{i!}$ olduğu gözönüne alındığında dizide bir elemanın kendisinden önceki elemana bölümü $\frac{a_i}{a_{i-1}} = \frac{x}{i}$ olarak hesaplanabilir. Yani her terim, kendisinden önceki terimin x ile çarpılıp i'ye bölünmesiyle bulunabilir. Böylece her adımda üs alma ve faktöryel hesaplama işlemlerine gerek kalmaz ve seri toplamı çok daha hızlı elde edilir.

Örnek 9 e^x deyimini bir önceki terimden giderek hesaplayan program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

int main(void)
{
    float x, error, term, result = 1.0;
    int i = 1;

    cout << "x: ";
    cin >> x;
    cout << "Hata: ";
    cin >> error;

    term = 1;
    while (term > error) {
        term = term * x / i;
        result = result + term;
        i++;
    }
    cout << "Hata: " << result << endl;

    return EXIT_SUCCESS;
}
```

- Her iki programı da yazarak çalıştırın.
- Taşma durumunun gözlenmesi

Sorular

1. `(x % 3 == 2) && (x > 4) || !(x % 3 == 2) && (x < 6)` mantıksal deyimini
   (a) doğru yapacak bir `x` değeri verin.
   (b) yanlış yapacak bir x değeri verin.

(c) `x` işaretsiz bir tamsayı ise, `x`'in hangi değerleri için bu deyim doğru değerini alır?

2. Aşağıda verilen program parçası için x değişkeninin başlangıç değerinin 115 olduğu varsayımıyla değişkenlerin alacakları değerleri izlemek üzere bir çizelge oluşturun. Akış çizeneğini çizin.

```
i = -1;
while (x >= 1) {
  i++;
  y[i] = x % 8;
  x = x / 8;
}
for (j = i; j >= 0; j--)
  cout << y[j];
```

3. x ve m sayıları aralarında asalsa x sayısının m sayısına göre evriği r sayısı, $xr = 1 \, mod \, m$ eşitliğini sağlayan sayıdır ($r < m$). Örneğin, $x = 5$ ve $m = 7$ ise $r = 3$ olur. Buna göre, kullanıcıdan aldığı x ve m sayılarına göre r değerini hesaplayan ve ekrana çıkartan bir program yazın. $x = 8$, $m = 11$ değerleri için programınızdaki değişkenlerin alacakları değerleri bir çizelge halinde gösterin.

4. Kusursuz bir sayı, kendisinden küçük bütün çarpanlarının toplamına eşit olan sayıdır. Örneğin, $28 = 1 + 2 + 4 + 7 + 14$. Buna göre, ilk 10000 sayı içindeki kusursuz sayıları ekrana döken bir program yazın.

5. İnsan bedeninin alanını hesaplamakta kullanılabilecek iki formül aşağıda verilmiştir.

   DuBois formülü: $bsa = h^{0.725} * w^{0.425} * 71.84 * 10^{-4}$

   Boyde formülü: $bsa = h^{0.3} * w^{(0.7285 - 0.0188 \, log \, w)} * 3.207 * 10^{-4}$

   Her iki formülde de h değişkeni cm cinsinden boyu ve bsa değişkeni de m^2 cinsinden beden alanını gösterirken, w değişkeni kütleyi DuBois formülünde kg, Boyde formülündeyse g cinsinden gösterir. DuBois formülü yetişkinlerde iyi sonuç verirken, çocuklarda (bu formülle elde edilen alan değeri 0.6'dan küçükse) çok hatalı olabilmektedir. Buna göre, boy ve kütlesini kullanıcıdan aldığı bir insanın (yetişkin ya da çocuk) beden alanını yukarıda anlatılan ölçütlere göre hesaplayarak ekrana çıkaran bir program yazın.

6. Aşağıda verilen Pascal üçgeni için i. satır j. sütundaki değer (binom katsayısı) yanda verilen formülle hesaplanır:

	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

$$binom_{i,j} = \begin{cases} 1 & j = 0 \vee j = i \\ binom_{i-1,j-1} + binom_{i-1,j} & aksi \, durumda \end{cases}$$

   Buna göre, üçgenin ilk 30 satırını hesaplayıp ekrana çıkartacak bir program yazın.

7. Aşağıdaki seri toplamı hesaplanmak isteniyor:

$$\sum_{i=0}^{n}(-1)^i\frac{x^i}{(2i)!} = \frac{x^0}{0!} - \frac{x^1}{2!} + \frac{x^2}{4!} - \frac{x^3}{6!} + \frac{x^4}{8!} - \frac{x^5}{10!} + \dots$$

(a) Bu serinin i. elemanı a_i ise, a_{i+1}/a_i oranı nedir?

(b) Bu bilgiyi kullanarak, serinin toplamını hesaplayan bir program yazın (x ve n değerleri kullanıcıdan alınacaktır).

8. Rasgele sayı üretirken kalan işlecini (`%`) kullanarak sayıyı belli bir aralığa indirgemek sayıların üretilme olasılıklarını bozar mı? Sözgelimi, `RAND_MAX`'ın değerinin 32767 olduğunu varsayarak 1 ile 6 arasında üreteceğiniz rasgele sayıda bu altı sayının gelme olasılıkları eşit midir? Değilse, daha iyi bir yöntem önerebilir misiniz?

Bölüm 4

Türetilmiş Veri Tipleri

Bu bölümde programcının var olan veri tiplerinden kendi veri tiplerini (sözgelimi kayıtlar, bkz. Bölüm 1.1.2) nasıl türetebileceği üzerinde durulacaktır.

C dilinin programcıya kendi veri tiplerini tanımlayabilmesi için sunduğu temel olanak, var olan veri tiplerine yeni isimler verilebilmesidir. Bunun için

```
typedef veri_tipi yeni_isim;
```

komutu kullanılır. Bunun sonucunda *veri_tipi* isimli tipe *yeni_isim* adında bir isim daha verilmiş olur.

Örneğin bir öğrencinin sınav notları işlenmek isteniyor olsun. Notların 0 ile 100 arasında birer tamsayı olacağı varsayımıyla, öğrenci notu tipinden bilgileri temsil etmek üzere bir veri tipi tanımlanabilir:

```
typedef int score_t;
```

Böylece `int` veri tipine `score_t` diye yeni bir isim verilmiş olur. Daha sonra bu tipten değişken tanımlamak için

```
score_t midterm1, midterm2, final;
```

gibi bir komut yazılabilir. İstenirse asıl veri tipi isminin kullanılmasında da bir sakınca yoktur, yani

```
int midterm1, midterm2, final;
```

tanımı da geçerliliğini korur ve bu iki tanım birbirine eşdeğerlidir.

Bir veri tipine yeni bir isim vermenin yararları şöyle açıklanabilir:

- Anlaşılırlık artar: Programın kodunu okuyan kişi bu veri tipinin temsil ettiği bilgiyle ilgili daha iyi bir fikir edinebilir.

- Değiştirmek kolay olur: Programın geliştirilmesinin ileri aşamalarında ya da sonraki sürümlerde öğrenci notlarının kesirli olabileceği durumu ortaya çıkarsa yalnızca veri tipine isim verme komutunun

  ```
  typedef float score_t;
  ```

  biçiminde değiştirilmesi yeterli olur. Veri tipi tanımı olmasa bütün kodun taranarak öğrenci notuna karşılık gelen `int` veri tiplerini bulup değiştirmek gerekir. Bu da bazı `int` sözcüklerinin değişmesi, bazılarının değişmemesi anlamına gelir ve programın boyutlarına göre büyük zorluklar çıkarabilir.

Örnek 10. Barbut

Barbut oyununun kuralları şöyledir:[1]

- Oyuncu bir çift zar atar.
  - Attığı zarların toplamı 7 ya da 11 ise oyuncu kazanır.
  - Attığı zarların toplamı 2, 3 ya da 12 ise oyuncu kaybeder.
  - Diğer durumlarda attığı zarların toplamı oyuncunun sayısı olur.
- Oyuncu aynı toplamı veren zarları bir daha atana kadar ya da attığı zarların toplamı 7 olana kadar zar atmaya devam eder.
  - Aynı toplamı bir daha atarsa oyuncu kazanır.
  - Attığı zarların toplamı 7 olursa oyuncu kaybeder.

Verilen örnek, bu oyunu simüle eden bir programdır. Bu örneğin ilginç bir yönü `switch` yapısının bazı durumlarında kasıtlı olarak `break` kullanılmamasıdır. Böylelikle durumlar gruplanarak her bir grup için yapılacak işlemlerin bir kere belirtilmesi sağlanmıştır. Programın örnek bir çalışmasının ekran çıktısı Şekil 4.1'de verilmiştir.

```
Zarlar: 1 + 4 = 5
Sayı: 5
Zarlar: 5 + 1 = 6
Zarlar: 5 + 5 = 10
Zarlar: 6 + 5 = 11
Zarlar: 4 + 1 = 5
Oyuncu kazanır.
```

Şekil 4.1: Örnek 10 ekran çıktısı.

[1] Bu örnek, H.M. Deitel ve P.J. Deitel'in yazdıkları "C: How to Program" kitabından uyarlanmıştır.

Örnek 10 Barbut oyununu simüle eden program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS,srand,rand
#include <time.h>               // time

using namespace std;

enum status_e { GAME_CONTINUES, PLAYER_WINS, PLAYER_LOSES };
typedef enum status_e status_t;

int main(void)
{
    int die1, die2, sum, point;
    status_t game_status = GAME_CONTINUES;

    srand(time(NULL));
    die1 = 1 + rand() % 6;
    die2 = 1 + rand() % 6;
    sum = die1 + die2;
    cout << "Zarlar: " << die1 << " + " << die2 << " = " << sum << endl;
    switch (sum) {
        case  7:
        case 11: game_status = PLAYER_WINS; break;
        case  2:
        case  3:
        case 12: game_status = PLAYER_LOSES; break;
        default: game_status = GAME_CONTINUES;
                 point = sum;
                 cout << "Sayı: " << point << endl;
                 break;
    }

    while (game_status == GAME_CONTINUES) {
        die1 = 1 + rand() % 6;
        die2 = 1 + rand() % 6;
        sum = die1 + die2;
        cout << "Zarlar: " << die1 << " + " << die2 << " = " << sum << endl;
        if (sum == point)
            game_status = PLAYER_WINS;
        else {
            if (sum == 7)
                game_status = PLAYER_LOSES;
        }
    }

    if (game_status == PLAYER_WINS)
      cout << "Oyuncu kazanır." << endl;
    else
      cout << "Oyuncu kaybeder." << endl;

    return EXIT_SUCCESS;
}
```

4.1 Numaralandırma

Örnekte oyunun içinde bulunabileceği çeşitli durumlara birer sayı değeri atanarak oyunun o an hangi durumda olduğu izlenebilir. Örneğin kodumuzda oyunun durumunu gösteren `game_status` değişkeninin değerinin 0 olması oyunun sürüyor olmasına, 1 olması oyuncunun kazanmasına ve 2 olması da oyuncunun kaybetmesine karşılık düşürülebilir. Burada 0, 1 ve 2 değerlerinin özel bir anlamları yoktur, herhangi üç farklı değer de aynı işlevi görür.

Okunulurluğu artırmak amacıyla bu sayılara birer isim vermek yararlı olur. Öyleyse

```
#define GAME_CONTINUES 0
#define PLAYER_WINS 1
#define PLAYER_LOSES 2
```

şeklinde değişmez tanımları yapılarak kodun içinde sayılar yerine isimler yazılabilir.

Bu tip, birbiriyle ilintili birden fazla değişmezi birlikte tanımlamak için numaralandırma yöntemi kullanılabilir.

```
enum { GAME_CONTINUES, PLAYER_WINS, PLAYER_LOSES };
```

komutu yukarıda yazılmış olan üç `#define` komutuyla aynı etkiyi yaratır. Aksi belirtilmedikçe, süslü ayraçlar içinde yazılan değişmezlerden ilkine 0, sonrakine 1, sonrakine 2 vs. şeklinde değer verilir. Programcı isterse başka değerler belirtebilir ancak bizim örneğimizde -demin de söylendiği gibi- değerlerin, farklı oldukları sürece, ne olduklarının bir önemi yoktur:

```
enum { GAME_CONTINUES = 58, PLAYER_WINS = 17, PLAYER_LOSES = 154 };
```

Herhangi bir değişmeze değer verilirse onu izleyen değişmezlerin değerleri birer artarak devam eder:

```
enum { JANUARY = 1, FEBRUARY, MARCH, APRIL, MAY, JUNE,
       JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER };
```

Numaralandırma ile oluşturulmuş bir değişmezler kümesine de topluca bir isim verilerek yeni bir veri tipi oluşturulabilir:

```
enum künye { değişmez_tanımları };
```

Bu komutun sonucunda `enum` *`künye`* isimli yeni bir veri tipi oluşur ve bu tipten değişkenler tanımlanabilir. Örnekte önce bu şekilde `enum status_e` isimli bir tip oluşturulmuş, daha sonra da bu tipe `typedef` komutuyla yeni bir isim verilmiştir. Bu tipten değişken tanımlarken her iki yeni isim de kullanılabilir, yani örnekteki

```
status_t game_status = GAME_CONTINUES;
```

komutu

```
enum status_e game_status = GAME_CONTINUES;
```

komutuyla eşdeğerlidir.[2]

Düzenli bir çalışma için numaralandırma tipinden tanımlanan değişkenlerin yalnızca o numaralandırmayla belirlenen değişmezler kümesinden değer alması gerekir. Sözgelimi örneğimizde tanımlanan `game_status` değişkeni `GAME_CONTINUES`, `PLAYER_WINS` ve `PLAYER_LOSES` dışında herhangi bir değer alamamalıdır. Ancak C dili böyle bir kısıtlama getirmez, yani

```
game_status = 25;
```

gibi anlamsız olacak bir atama C dilinin kurallarına göre yasak değildir.

Örnek 11. Paralel Direnç Eşdeğeri

Dirençlerin paralel eşdeğeri aşağıdaki formülle hesaplanabilir:

$$\frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \cdots + \frac{1}{R_n}}$$

Buna göre, kullanıcıdan aldığı dirençlerin paralel eşdeğerini hesaplayarak sonucu ekrana çıkartan bir program yazılması isteniyor. Kullanıcının kaç tane direnç değeri gireceği baştan belli olmadığı için bu döngü `for` yapısıyla değil `while` yapısıyla kurulmaya daha uygundur. Sonsuz döngüden çıkabilmek için kullanıcının bir şekilde bunu belirtmesine olanak verilmelidir. Bu programda uygulanan yöntem, özel bir değeri (örneğin 0) bitirme işareti olarak seçmektir. Kullanıcı bu değeri girerek başka direnç değeri yazmayacağını belirtebilir. Sonlandırma değerinin uygun seçilmesi gerekir; geçerli direnç değerleri sonlandırma değeri olmaya uygun değildir. Programın örnek bir çalışmasının ekran çıktısı Şekil 4.2'de verilmiştir.

Bu programın çalışmasında dikkat çekici bir nokta, kullanıcının yazdığı direnç değerlerinin uygun şekilde toplama eklenmelerinin ardından "unutulmaları"dır. Yani ileride kullanıcının girmiş olduğu değerlerle bir işlem yapılmak istense bunlar bir şekilde ulaşılabilir olmayacaklardır. Bu programın amacı açısından bu durum bir sakınca doğurmaz ancak başka problemlerde farklı yapılar kullanmak gerekebilir.

[2] C dilinde mantıksal bir veri tipi ve `true`, `false` değerleri bulunmadığı daha önce söylenmişti. Ancak, doğru ve yanlış büyüklükleri programlarda sıkça gereksinim duyulan değerler olduklarından C programlarında genellikle bunlar genellikle programın başında değişmez olarak tanımlanırlar.

```
#define TRUE 1
#define FALSE 0
```

Daha sık kullanılan bir yöntemse bunları bir numaralandırma içinde tanımlayarak oluşacak veri tipine yeni bir isim vermektir.

```
enum bool_e { FALSE, TRUE };
typedef enum bool_e bool_t;
```

Böylelikle C++ dilinde olduğu gibi mantıksal bir veri tipi tanımlanmış olur ve bu tipten değişkenler kullanılabilir.

Örnek 11 Paralel direnç eşdeğeri hesaplayan program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

struct rational_s {
    int nom;
    int denom;
};
typedef struct rational_s rational_t;

int main(void)
{
    int res_value;
    rational_t equiv = { 0, 1 };
    int a, b, r;
    int i = 0;

    while (true) {
        i++;
        cout << "R" << i << " (bittiyse 0): ";
        cin >> res_value;
        if (res_value == 0)
            break;
        equiv.nom = equiv.nom * res_value + equiv.denom;
        equiv.denom = equiv.denom * res_value;

        // kesiri basitleştir
        a = equiv.nom;
        b = equiv.denom;
        while (b > 0) {
            r = a % b;
            a = b;
            b = r;
        }
        equiv.nom = equiv.nom / a;
        equiv.denom = equiv.denom / a;
    }
    if (equiv.nom != 0)
        cout << "Eşdeğer direnç: " << equiv.denom
             << " / " << equiv.nom
             << " = " << (float) equiv.denom / equiv.nom << endl;
    else
        cout << "Hatalı işlem." << endl;
    return EXIT_SUCCESS;
}
```

```
R1 (bittiyse 0): 5
R2 (bittiyse 0): 8
R3 (bittiyse 0): 11
R4 (bittiyse 0): 4
R5 (bittiyse 0): 0
Eşdeğer direnç: 440/293 = 1.50171
```

Şekil 4.2: Örnek 11 ekran çıktısı.

4.2 Yapılar

Kayıt tiplerinin özelliklerinden Bölüm 1.1.2'de söz edilmişti. C dilinde kayıtlara *yapı* adı verilir ve şöyle tanımlanırlar:

```
struct künye {
  alan_tanımları ;
};
```

Yapı tanımının sonucunda `struct künye` isimli yeni bir veri tipi oluşur ve bu tipten değişkenler tanımlanabilir. Alan tanımları değişken tanımlarına benzer şekilde yapılır ancak değişken tanımı *değildirler*. Değişken tanımlarının bellekte ilgili değişken için yer ayrılmasına neden olduğu görülmüştü, oysa tip tanımı bellekte yer ayrılmasına neden olmaz. Yer ayrılması ancak bu yapı tipinden bir değişken tanımlandığında gerçekleşir.

Örnekte rasyonel sayıları göstermek üzere iki alanı olan bir yapı kullanılmıştır: sayının pay kısmını gösteren `nom` ve payda kısmını gösteren `denom`. Rasyonel sayıların tanımı gereği her iki alan da tamsayı tipindendir. Tip tanımı sonucunda artık

```
struct rational_s
```

adında bir veri tipi oluşmuştur. Kullanım kolaylığı açısından bu veri tipine `typedef` komutuyla yeni bir isim verilmiştir.[3]

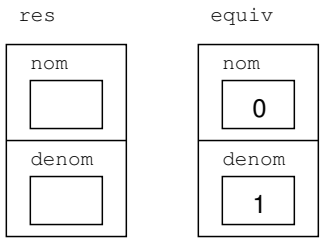
Yapı tipinden değişken tanımlanırken istenirse künyeli isim, istenirse `typedef` ile verilen yeni isim kullanılabilir ve değişkenin alanlarına süslü ayraçlar içinde başlangıç değerleri verilebilir. Aşağıdaki iki komut aynı işi yaparlar, yani `res` ve `equiv` isimlerinde, her biri birer rasyonel sayı yapısında olan iki değişken tanımlarlar ve `equiv` değişkeninin `nom` alanına 0, `denom` alanına 1 başlangıç değerini verirler (Şekil 4.3):

[3] Tipin tanımlanması ve yeni isim verilmesi işlemleri istenirse tek komutta birleştirilebilir:

```
typedef struct rational_s {
  int nom, denom;
} rational_t;
```

Birleştirilmiş tanımlamada istenirse tipin künyesi de belirtilmeyebilir, yani yukarıdaki örnekte `rational_s` sözcüğü yazılmasa da olurdu. Yine de çoğunlukla önerilen yöntem, belirtilmeleri zorunlu olmasa da künyeleri yazmaktır. Bu yazılış numaralandırmalar için de geçerlidir.

```
rational_t res, equiv = { 0, 1 };
struct rational_s res, equiv = { 0, 1 };
```



Şekil 4.3: Yapı tipinden değişken tanımlama.

Bu değişkenlerin her biri, yapıda belirtilen alanları barındırır. Alanlar üzerinde işlem yapmak için, daha önce görüldüğü gibi, noktalı gösterilim kullanılır, yani değişkenin adından sonra nokta işaretiyle alanın adı belirtilir. Bu durumda, eşdeğer direnç değerini tutan değişkenin payda kısmıyla bir işlem yapılacaksa `equiv.denom` yazılır.

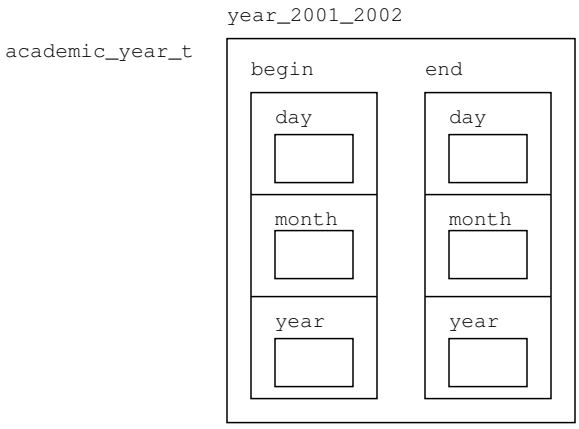
Yapıların alanları skalar tiplerden tanımlanabileceği gibi, başka yapı tiplerinden ya da tipli numaralandırma tipinden de tanımlanabilir:

```
enum month_e { JANUARY = 1, FEBRUARY, ..., DECEMBER };
typedef enum month_e month_t;

struct date_s {
  int day;
  month_t month;
  int year;
};
typedef struct date_s date_t;

struct academic_year_s {
  date_t begin, end;
};
typedef struct academic_year_s academic_year_t;
```

tanımları Şekil 4.4'de görülen yapıyı oluşturur. Alanlara erişim yine noktalı gösterilimle sağlanır:

```
academic_year_t year_2001_2002;
year_2001_2002.end.day = 31;
year_2001_2002.end.month = MAY;
year_2001_2002.end.year = 2002;
```

Aynı yapı tipinden değişkenler arasında atama işlemi yapılabilir. Atama sonucunda kaynak değişkenin bütün alan değerleri varış değişkenin aynı isimli alanlarına atanır. Bu gösterilim bütün alt alanların tek tek atanması zorunluluğunu giderir, yani



Şekil 4.4: Yapı içinde yapı kullanma.

```
equiv.nom = res.nom;
equiv.denom = res.denom;
```

şeklinde atama yapmak yerine yalnızca

```
equiv = res;
```

yazılabilir.

Uygulama: Yapılar

Örnek 12. Noktanın Daireye Göre Konumu

Koordinatlarını kullanıcıdan aldığı bir noktanın, merkez noktasının koordinatları ile yarıçapını yine kullanıcıdan aldığı bir dairenin içinde mi, dışında mı, üzerinde mi olduğunu belirleyerek sonucu ekrana çıkaracak bir program yazılması isteniyor. Noktanın koordinatları x ve y, daire merkezinin koordinatları x_c ve y_c, dairenin yarıçapı r ile gösterilirse:

$\sqrt{(x-x_c)^2+(y-y_c)^2} < r$ ise nokta dairenin içinde

$\sqrt{(x-x_c)^2+(y-y_c)^2} > r$ ise nokta dairenin dışında

$\sqrt{(x-x_c)^2+(y-y_c)^2} = r$ ise nokta dairenin üzerindedir.

- Programı yazarak çalıştırın.
- Merkez noktasının koordinatlarını (2.1,5.664), yarıçapını 3.2, aranan noktanın koordinatlarını (5.3,5.664) olarak verin. Bu nokta dairenin neresindedir? Program ne çıktı veriyor? Hatalıysa neden hatalıdır ve nasıl düzeltilebilir?

Örnek 12 Noktanın daireye göre konumunu bulan program.

```
#include <iostream>             // cout,cin,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

typedef struct point_s {
    float x, y;
} point_t;

typedef struct circle_s {
    point_t center;
    float radius;
} circle_t;

int main(void)
{
    circle_t circle1;
    point_t point1;
    float p, deltaX, deltaY;

    cout << "Daire merkezinin koordinatlarını yazın (x y): ";
    cin >> circle1.center.x >> circle1.center.y;
    cout << "Dairenin yarıçapını yazın: ";
    cin >> circle1.radius;
    cout << "Noktanın koordinatlarını yazın (x y): ";
    cin >> point1.x >> point1.y;
    deltaX = point1.x - circle1.center.x;
    deltaY = point1.y - circle1.center.y;
    p = deltaX * deltaX + deltaY * deltaY;
    if (p < circle1.radius * circle1.radius)
        cout << "Nokta dairenin içinde." << endl;
    else {
        if (p > circle1.radius * circle1.radius)
            cout << "Nokta dairenin dışında." << endl;
        else
            cout << "Nokta dairenen üzerinde." << endl;
    }
    return EXIT_SUCCESS;
}
```

- Bir noktanın konumunun belirtilmesinden sonra programdan çıkmadan kullanıcının aynı daireye göre başka noktaların da konumlarını sorabilmesi için programda gerekli değişiklikleri yapın.
- Merkez koordinatları ve yarıçapları kullanıcıdan alınan iki dairenin kesişip kesişmediklerini belirleyen bir program yazın.

Sorular

1. Örnek 11'da `equiv` değişkenine farklı bir başlangıç değeri verilebilir miydi? Örneğin değişken tanımlama komutu

   ```
   rational_t res, equiv = { 0, 0 };
   ```

   şeklinde olsaydı program doğru çalışır mıydı?

2. Aynı örnekte kullanıcı ilk direnç değeri olarak 0 verirse programın davranışı nasıl olur? Gerekiyorsa programda dazeltmeler yapın.

3. Örnek 10'da uygulaması yapılan barbut oyunu adil bir oyun mudur? Yani oyuncunun kazanma olasılığı ile kaybetme olasılığı aynı mıdır?

Bölüm 5

Diziler

Bu bölümde programlamada sıkça gereken dizi tiplerinin (bkz. Bölüm 1.1.3) C dilinde nasıl kullanılacağı üzerinde durulacaktır.

Örnek 13. İstatistik Hesapları

Kullanıcıdan aldığı sınav notlarının ortalamasını, varyansını, standart sapmasını ve mutlak sapmasını hesaplayan bir program yazılması isteniyor. Programın örnek bir çalışması Şekil 5.1'de verilmiştir.

Öğrenci sayısı n, i. öğrencinin notu s_i, ortalama m, varyans v, standart sapma sd, mutlak sapma ad ile gösterilirse:

$$\begin{aligned} m &= \frac{\sum_{i=1}^{n} s_i}{n} \\ v &= \frac{\sum_{i=1}^{n} (s_i - m)^2}{n-1} \\ sd &= \sqrt{v} \\ ad &= \frac{\sum_{i=1}^{n} |s_i - m|}{n} \end{aligned}$$

Bu problemde dört döngü işlemi görülebilir:

1. Kullanıcının girdiği notları okumak için bir döngü.
2. Ortalama için gereken, notların toplamını hesaplamakta kullanılan döngü (birinci eşitlikteki $\sum$ işaretine karşı düşer).
3. Varyans ve standart sapma için gereken, her bir notun ortalamayla farklarının karelerinin toplamını hesaplamakta kullanılan döngü (ikinci eşitlikteki $\sum$ işaretine karşı düşer).
4. Mutlak sapma için gereken, her bir notun ortalamayla farklarının mutlak değerlerinin toplamını hesaplamakta kullanılan döngü (dördüncü eşitlikteki $\sum$ işaretine karşı düşer).

Örnek 13 Çeşitli istatistik hesapları yapan program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS
#include <math.h>               // fabs,sqrt

using namespace std;

#define MAXSTUDENTS 100

int main(void)
{
    int score[MAXSTUDENTS];
    int no_students = 0;
    float mean, variance, std_dev, abs_dev;
    float total = 0.0, sqr_total = 0.0, abs_total = 0.0;
    int i = 0;

    cout << "Kaç öğrenci var? ";
    cin >> no_students;
    for (i = 0; i < no_students; i++) {
        cout << i + 1 << ". öğrencinin notu: ";
        cin >> score[i];
        total = total + score[i];
    }
    mean = total / no_students;
    for (i = 0; i < no_students; i++) {
        sqr_total = sqr_total + (score[i] - mean) * (score[i] - mean);
        abs_total = abs_total + fabs(score[i] - mean);
    }
    variance = sqr_total / (no_students - 1);
    std_dev = sqrt(variance);
    abs_dev = abs_total / no_students;
    cout << "Ortalama: " << mean << endl;
    cout << "Varyans: " << variance << endl;
    cout << "Standart sapma: " << std_dev << endl;
    cout << "Mutlak sapma: " << abs_dev << endl;
    return EXIT_SUCCESS;
}
```

```
Kaç öğrenci var? 5
1. öğrencinin notu: 65
2. öğrencinin notu: 82
3. öğrencinin notu: 45
4. öğrencinin notu: 93
5. öğrencinin notu: 71
Ortalama: 71.2
Varyans: 329.2
Standart sapma: 18.1439
Mutlak sapma: 13.04
```

Şekil 5.1: Örnek 13 ekran çıktısı

Burada birinci ve ikinci döngüler tek bir döngüde birleştirilebilir, kullanıcı notları girdikçe bunlar toplama eklenebilir. Ancak üçüncü ve dördüncü döngüler bu döngüye eklenemez, çünkü o döngülerde her bir notun ortalamayla farkına gereksinim vardır ve bu ortalama ancak birinci döngü sona erdiğinde elde edilmektedir. Döngü sayacı aynı sınır değerleri arasında değiştiğinden ve her yinelemede aynı öğrenci notu üzerinde işlem yapıldığından üçüncü ve dördüncü döngüler de kendi aralarında tek bir döngüde birleştirilebilirler.

Örnek 11'da kullanıcının girdiği direnç değerleri toplama eklendikten hemen sonra yitiriliyorlardı. Oysa bu örnekte öğrenci notlarının birinci döngünün çıkışında unutulmaması gerekir, çünkü ikinci döngüde de bunlar gerekecektir. Yani bu notların bir yerde tutulması gerekir. Aynı tipten çok sayıda elemanı bulunan değişkenler için en uygun yapının diziler olduğu Bölüm 1.1.3'de görülmüştü. Örneğimizde de öğrenci notları bir dizi olarak temsil edilmektedir.

5.1 Tek Boyutlu Diziler

Dizi tanımının yazımı

veri_tipi dizi_adı[*dizi_boyu*];

şeklindedir. Burada *dizi_adı* dizi tipinden tanımlanan değişkenin adını, *dizi_boyu* bu dizide kaç eleman bulunduğunu, *veri_tipi* ise her bir elemanın hangi tipten olduğunu belirtir. Örnekte

```
int score[MAXSTUDENTS];
```

komutu, her biri tamsayı tipinden `MAXSTUDENTS` elemanlı, `score` adında bir dizi tanımlar. Bu dizi için bellekte birbirini izleyecek şekilde `MAXSTUDENTS * sizeof(int)` sekizli yer ayrılır.

Diğer değişkenlerde olduğu gibi, dizilere de tanım sırasında başlangıç değeri verilebilir. Örneğin:

```
int score[4] = { 85, 73, 91, 66 };
```

tanımı 4 elemanlı bir dizi tanımlar ve elemanlara sırasıyla 85, 73, 91 ve 66 değerlerini atar. Özel bir durum olarak, bütün diziye 0 başlangıç değeri verilmek isteniyorsa şu komut kullanılabilir:

```
int score[50] = { 0 };
```

Derleyicinin dizi için bellekte ne kadar yer ayrılacağını bilmesi gerekir, yani derleme aşamasında dizinin kaç elemanının olacağı belli olmalıdır. Bunun için iki yöntem kullanılabilir:

açık belirtim Dizinin kaç elemanı olduğunun tanım sırasında açık olarak belirtildiği, yukarıda da örnekleri görülen yöntemdir. Boy olarak değişmez bir değer vermek zorunludur. Örnekte sınıftaki öğrenci sayısı `no_students` değişkeniyle gösterilmektedir, yani dizinin `no_students` elemanı olması gerekli ve yeterlidir. Ancak `no_students` değişkeninin değerinin ne olacağı programın çalışması sırasında belirlendiğinden derleyici bu değeri dizi boyu olarak kullanamaz, yani

```
int score[no_students];
```

şeklinde bir dizi tanımı yapılamaz. Başka bir deyişle, dizi boyu için yazılacak deyimde yalnızca sayılar ve değişmezler yer alabilir, değişkenler yer alamaz. Bu durumda, dizinin kaç elemanı olacağı baştan bilinmiyorsa gerekebilecek en büyük eleman sayısı boy olarak belirtilmelidir. Örnekteki `MAXSTUDENTS` değişmezi bu işlevi görmektedir. Burada belirtilen sayı, bir yandan gereksiz bellek kullanımına yol açabilir, diğer yandan da programın bir kısıtlaması haline gelir. Örnek programın yaptığı işin açıklamasını şu şekilde düzeltmek gerekir:

> Bu program, *en fazla 100 öğrencili bir sınıfta*, öğrenci notlarının ortalamasını, varyansını ve standart ile mutlak sapmalarını hesaplar.

örtülü belirtim Diziye başlangıç değeri verilirse eleman sayısını belirtmek zorunluluğu yoktur, yani

```
int score[] = { 85, 73, 91, 66 };
```

tanımı da dört elemanlı bir tamsayı dizisi tanımlar ve söylenen başlangıç değerlerini atar. Bu durumda derleyici başlangıç değeri olarak verilen dizideki eleman sayısını sayarak boyu kendisi belirler. Dizideki eleman sayısı ileride artmayacaksa bu yazımı kullanmanın bir sakıncası yoktur, ancak artması olasılığı varsa yine gerekebilecek en büyük eleman sayısının dizi tanımında açık olarak belirtilmesi gerekir çünkü aksi durumda sonradan gelen elemanlar için yer ayrılmamış olur.

Dizinin bir elemanı üzerinde işlem yapmak için o elemanın kaçıncı eleman olduğunu belirtmek gerekir. Bu belirtim de dizi adının yanında elemanın sıra numarasının köşeli ayraçlar içine yazılmasıyla yapılır, yani `score[i]`, `score` dizisinin `i`. elemanı anlamına gelir. Bu yazım

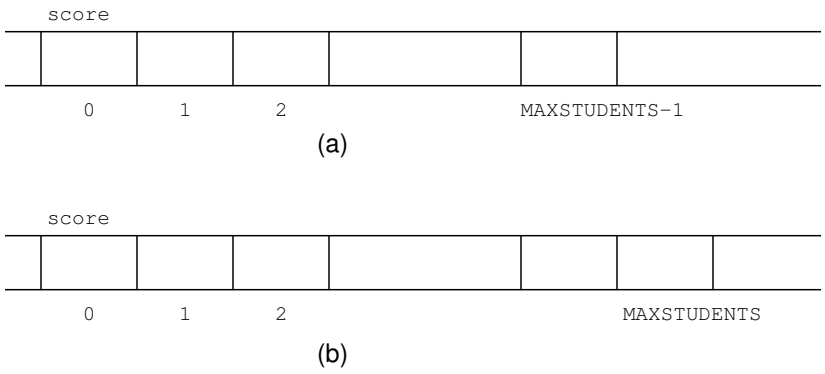
dizi tanımındaki yazımla aynı olmakla birlikte anlam olarak tamamıyla farklıdır: tanımda köşeli ayraçlar içine yazılan sayı dizinin tutabileceği eleman sayısını gösterirken burada kaçıncı eleman üzerinde işlem yapıldığını gösterir.

C dilinde dizilerin ilk elemanının sıra numarası her zaman 0'dır; yani ilk elemanın sıra numarası 0, ikinci elemanın sıra numarası 1 olacak şekilde ilerler (Şekil 5.2a). Bu durumda n elemanlı bir dizinin son elemanının sıra numarası n - 1 olur. Bu özellik nedeniyle `n` elemanlı bir dizi üzerinde işlem yapacak tipik bir C döngüsü

```
for (i = 0; i < n; i++)
```

şeklinde başlar. İlk elemanın sıra numarası 0 olduğundan `for` bloğuna ilk girişte dizinin ilk elemanı ile işlem yapılır. Son işlem de n - 1 sıra numaralı elemanla yapılır, döngü sayacı n değerini aldığında döngüden çıkılır. Burada dizinin tanımda belirtilen boyuyla gerçekten kullanılan eleman sayısı arasındaki ayrıma dikkat edilmelidir. Örnekte tanımda belirtilen boy `MAXSTUDENTS` olsa da döngü 0. elemandan `MAXSTUDENTS - 1`. elemana kadar gitmez çünkü son geçerli elemanın sıra numarası `no_students - 1` olacaktır:

```
for (i = 0; i < no_students; i++)
```



Şekil 5.2: Dizi elemanlarının sıra numaraları.

Derleyici dizilerin elemanlarına erişimde dizi sınırlarının denetimini yapmaz; yani n elemanlı bir dizinin n. ya da daha sonraki elemanlarına erişilmeye kalkılırsa hata vermez. Dizi sınırlarından taşmamak programcının sorumluluğundadır. Dizi sınırından taşacak bir sıra numarası verilirse bellekte ilk elemandan başlanarak istenen sayıda eleman kadar ilerlenip orada ne bulunursa o değerle işlem yapılmaya çalışılır (Şekil 5.2b). Böyle bir erişim çalışma anında iki tür soruna yol açabilir:

1. Gelinen bellek gözü, başka bir değişken için ayrılmış bir bölgeye düşebilir. Bu durumda, başka bir değişkenin değeri istenmeden değiştirilmiş olur ve programın çalışması üzerinde beklenmedik etkiler yaratabilir.

2. Erişilmek istenen bellek bölgesi kullanıcının izni olan bölgeler dışında bir yere düşebilir. Bu durumda bir *bellek hatası* oluşur.

Örnek 14. Tümce Tersine Çevirme

Kullanıcının yazdığı tümceyi tersine çevirerek ekrana çıkartan bir program yazılması isteniyor. Programın örnek bir çalışması Şekil 5.3'de verilmiştir.

```
Tümce: Deneme bir ...
Tersi: ... rib emeneD
```

Şekil 5.3: Örnek 14 ekran çıktısı.

Örnek 14 Tümceyi tersine çeviren program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS
#include <stdio.h>              // gets
#include <string.h>             // strlen

using namespace std;

#define MAXLENGTH 80

int main(void)
{
    char sentence[MAXLENGTH];
    int len, i;
    char tmp;

    cout << "Tümce: ";
    gets(sentence);
    len = strlen(sentence);
    for (i = 0; i < len / 2; i++) {
        tmp = sentence[i];
        sentence[i] = sentence[len - 1 - i];
        sentence[len - 1 - i] = tmp;
    }
    cout << "Tersi: " << sentence << endl;
    return EXIT_SUCCESS;
}
```

Örnek program, baştan ve sondan aynı sırada olan simgelerin yerlerinin takas edilmesi algoritmasını kullanır. Yani ilk simgeyle son simge, ikinci simgeyle sondan bir önceki simge v.b. takas edilir. Bu algoritmanın doğru çalışması için takasın kataın ortasına kadar sürmesi ve katar ortasını geçmemesi gerekir. (Neden?)

5.2 Katarlar

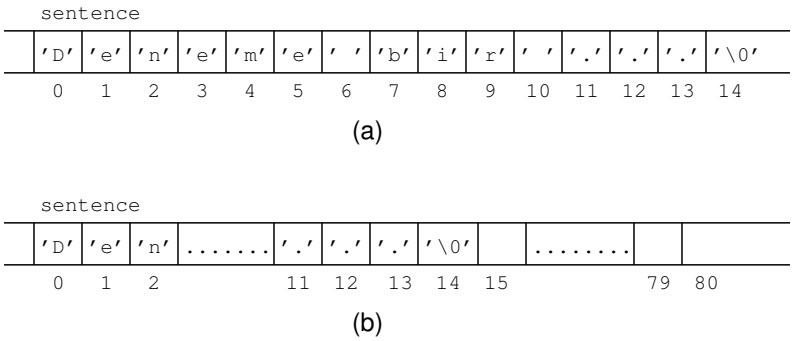
C dilinde katarlar birer simge dizisi olarak değerlendirilir ve sonlarına konan '\0' simgesiyle sonlandırılırlar. Bu nedenle katar için bellekte ayrılması gereken yer, katarın içerdiği simge sayısının bir fazlasıdır. Örnekteki

```
char sentence[MAXLENGTH];
```

tanımı, her biri simge tipinden, 80 elemanlı, `sentence` adında bir katar oluşturur. Bu elemanlardan biri '\0' simgesi için kullanılacağından kullanıcının yazacağı tümce en fazla 79 simge uzunluğunda olabilir.

Katar tipinden bir değişkenin tanımlanması sırasında çift tırnaklar içinde başlangıç değeri belirtilebilir. Tanımlanan boyun katar için gereken yeri ayırmasına dikkat edilmelidir. İstenirse boyut belirtilmeyebilir; bu durumda derleyici gerekli yeri kendisi ayırır. Aşağıdaki iki tanım aynı işi görürler (Şekil 5.4a):

```
char sentence[15] = "Deneme bir ...";
char sentence[] = "Deneme bir ...";
```



Şekil 5.4: Katarlara başlangıç değeri atanması.

Tam gerektiği kadar yer ayırmak riskli bir davranıştır. Programın içinde katarın uzaması sözkonusu olabilecekse gerekebilecek maksimum alan gözönüne alınmalı ve tanım sırasında bu boy belirtilmelidir.

```
char sentence[MAXLENGTH] = "Deneme bir ...";
```

komutu `MAXLENGTH` simgelik yer ayırır ancak yalnızca ilk 15 simgeyi doldurur (Şekil 5.4b).

Katarların elemanlarına dizilerde olduğu gibi teker teker erişilebilir. Dizilerde olduğu gibi, ilk elemanın, yani katarın ilk simgesinin sıra numarası 0 olacaktır. Son elemanın sıra numarası da ayrılan yerin bir eksiğidir. Yukarıdaki örnekte `sentence[0]` elemanı 'D', `sentence[6]` elemanı ' ', `sentence[13]` elemanı '.', `sentence[14]` elemanı '\0' değerlerini alırlar.

Katar tipinden değişkenlerin `cout` birimi yardımıyla çıktıya gönderilmelerinde herhangi bir sorun olmazken, `cin` ile girdi aşamasında sorun çıkabilir. Kullanıcının yazdığı katarda boşluk simgesi varsa, `cin` birimi birden fazla değişkenin girildiğini düşünerek yalnızca ilk boşluğa kadar olan bölümü değişkene aktaracaktır. Örnekte bu amaçla katarın girdi işlemi `gets` kitaplık fonksiyonuyla gerçekleştirilmiştir.[1] Bu komutun yerine

```
cin >> sentence;
```

komutu kullanılmış olsaydı programın çalışması Şekil 5.5'de görüldüğü gibi olurdu.

```
Tümce: Deneme bir ...
Tersi: emeneD
```

Şekil 5.5: Örnek 14 hatalı ekran çıktısı.

5.3 Katar Kitaplığı

Katarlar birer dizi olduklarından katarlar arasında atama yapmak ya da katarları karşılaştırmak için katar kitaplığındaki fonksiyonların kullanılması gerekir. Örnek programda, katarın uzunluğunu belirlemek üzere bu kitaplıktaki `strlen` fonksiyonundan yararlanılmıştır.

Katar tipinden iki değişkenin birbirine atanması ya da karşılaştırılması işlemleri standart işleçler kullanılırsa beklenmedik sonuçlar doğurabilir. Başka bir deyişle, atama için

```
str1 = str2;
```

ya da karşılaştırma için

```
if (str1 == str2)
```

gibi yapılar kullanılmaz.[2]

Katarlar üzerinde sıkça kullanılan temel işlemler için bir katar kitaplığı tanımlanmıştır. Bu fonksiyonların kullanılabilmesi için programların başında `string.h` başlık dosyasının alınması gerekir.

Katar kitaplığının en sık kullanılan fonksiyonları Tablo 5.1'de verilmiştir.

Katar uzunluğu fonksiyonu sondaki `'\0'` simgesini gözönüne almaz.

- `strlen("computer")` → `8`

[1] Güvenlik nedenleriyle `gets` kitaplık fonksiyonunun kullanılması tehlikelidir. Bu komutun yerine `fgets` fonksiyonunun kullanılması önerilir (bkz. Bölüm 8).

[2] Daha ayrıntılı açıklama için bkz. Bölüm .7.3.

Adı	İşlevi
`strlen(s)`	katar uzunluğu
`strcopy(dest,src) strncpy(dest,src,n)`	katar kopyalama
`strcmp(s1,s2) strncmp(s1,s2,n)`	katar karşılaştırma
`strcat(dest,src) strncat(dest,src,n)`	katar bitiştirme

Tablo 5.1: Katar kitaplığı fonksiyonları.

Katar kopyalama fonksiyonları ikinci giriş parametresi olan katarı, birinci giriş parametresi olan katarın üstüne yazarlar. Bitiştirme fonksiyonlarıysa ikinci giriş parametresi olan katarı, birinci giriş parametresi olan katarın ardına eklerler. Her iki fonksiyon grubu da sonuç katarının sonuna konması gereken '\0' simgesini kendileri koyarlar.

Katar işlemi yapan fonksiyonların bazılarında uzunluk denetimi yapılır, bazılarında yapılmaz. Sözgelimi `strcpy` fonksiyonu `src` katarının değerini `dest` katarının üstüne yazarken, `strncpy` fonksiyonu da aynı işlevi görür, ancak en fazla `n` simgeyi kopyalar. Güvenlik açısından uzunluk denetimi yapan fonksiyonların kullanılması önerilir.[3]

Katar karşılaştırma fonksiyonları giriş parametresi olan katarlar arasında İngilizce dilinin kurallarına göre sözlük sırası karşılaştırması yaparlar. İki katar eşitse 0, birinci katar ikinciden önce geliyorsa -1, sonra geliyorsa 1 değerini döndürürler.

- `strcmp("abc", "ad")` → -1
- `strcmp("abcd", "abc")` → 1
- `strncmp("abcd", "abcde", 4)` → 0

Kopyalama ve bitiştirme fonksiyonlarında dikkat edilmesi gereken bir nokta, hedef katarın yeterli uzunlukta olmasının sağlanmasıdır. Örneğin, aşağıdaki program parçası hataya yol açabilir:

```
char author1[] = "Brian", author2[] = "Dennis";
...
strcpy(author1, author2);
```

Derleyici `author1` katarına 6 simgelik, `author2` katarınaysa 7 simgelik yer ayıracaktır. Daha sonra `author1` katarının üstüne `author2` katarı yazılmaya kalkıldığında `author1` katarında yeterli yer olmadığı için bu değişkenin sınırlarından taşılır. Benzer şekilde, bitiştirme işlemlerinde de hedef katar için iki asıl katarın toplam uzunluklarını tutmaya yetecek kadar yer ayrılmış olmalıdır.

[3] Uzunluk denetimi yapmayan katar fonksiyonları "tampon taşırma" adı verilen saldırılara yol açan kaynaklar arasında en büyük yeri tutar.

Uygulama: Tek Boyutlu Diziler

Örnek 15. Polinom Hesabı

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

şeklinde yazılan n. dereceden bir polinomda verilen bir x değeri için $p(x)$'in hesaplanması $\frac{n(n+1)}{2}$ adet çarpma ve n adet toplama işlemi gerektirir. İçiçe çarpma ve toplama yöntemiyle bu sayı azaltılabilir.

Polinomu şu şekilde yeniden yazalım:

$$p(x) = (((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0$$

O halde bir b değeri, başlangıçta $b = a_n$ ve i. adımda $b = bx + a_{n-i}$ olacak şekilde hesaplanarek gidilirse a_0 katsayısının da toplanmasıyla elde edilecek b değeri hesaplanmak istenen $p(x)$ değeri olur. Yapılması gereken işlem sayısı da n çarpma ve n toplamaya iner.

- Çıktısı verilen programı kullanarak

  $$p(x) = 13x^7 - 9x^6 + 12x^4 - 4x^3 + 3x + 5$$

  polinomunun $p(4)$, $p(2.5)$ ve $p(19)$ değerlerini hesaplatın.

Örnek 16. Matris Çarpımı

Bu bölümde üzerinde çalışılacak program, boyutları ve elemanları kullanıcıdan alınan iki matrisi çarparak sonucu ekrana çıkartır. Programın örnek bir çalışmasının ekran çıktısı Şekil 5.6'da verilmiştir.

5.4 Çok Boyutlu Diziler

Çok boyutlu bir dizi tanımlanmasında her boyut ayrı bir köşeli ayraç çifti içinde belirtilir. Örnekteki

```
int left[MAXSIZE][MAXSIZE];
```

komutu, 30 satırlı ve 30 sütunlu bir matris tanımlar. Bu matrisin, her biri birer tamsayı olan, 30*30=900 elemanı olacaktır, yani bellekte 900 tamsayı tutacak kadar yer ayrılmasına neden olur. Daha çok boyutu olan bir dizi tanımlanmak istenseydi boyutlar yanyana sürdürülebilirdi:

```
int m[30][20][7][12];
```

Örnek 15 Polinom değeri hesaplayan fonksiyon.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

#define MAXDEGREE 50

int main(void)
{
    float a[MAXDEGREE];
    float x, b;
    int n, i;
    char response = 'E';

    cout << "Polinomun derecesi: ";
    cin >> n;

    cout << "Katsayılar: " << endl;
    for (i = n; i >= 0; i--) {
        cout << "a" << i << ": ";
        cin >> a[i];
    }

    while (true) {
        cout << "x: ";
        cin >> x;
        b = a[n];
        for (i = n - 1; i >= 0; i--)
            b = b * x + a[i];
        cout << "p(x): " << b << endl;
        cout << "Devam etmek istiyor musunuz (E/H)? ";
        cin >> response;
        if ((response == 'H') || (response == 'h'))
            break;
    }

    return EXIT_SUCCESS;
}
```

```
Sol matrisin satır sayısı: 5
Sol matrisin sütun sayısı: 3
Sağ matrisin sütun sayısı: 4
Sol matris:
[1,1]: 2
[1,2]: 7
[1,3]: -1
[2,1]: 3
[2,2]: 5
[2,3]: 2
[3,1]: 0
[3,2]: 4
[3,3]: 1
[4,1]: 9
[4,2]: 0
[4,3]: -3
[5,1]: 4
[5,2]: 7
[5,3]: 2
Sağ matris:
[1,1]: 0
[1,2]: 5
[1,3]: 2
[1,4]: -4
[2,1]: 1
[2,2]: 7
[2,3]: 3
[2,4]: 2
[3,1]: 5
[3,2]: 2
[3,3]: 0
[3,4]: -1
Çarpım sonucu:
2 57 25 7
15 54 21 -4
9 30 12 7
-15 39 18 -33
17 73 29 -4
```

Şekil 5.6: Örnek 16 ekran çıktısı.

Örnek 16 İki matrisi çarpan program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

#define MAXSIZE 30

int main(void)
{
    int left[MAXSIZE][MAXSIZE], right[MAXSIZE][MAXSIZE];
    int product[MAXSIZE][MAXSIZE] = { 0 };
    int rl, cl, cr;
    int &rr = cl;
    int i, j, k;

    cout << "Sol matrisin satır sayısı: "; cin >> rl;
    cout << "Sol matrisin sütun sayısı: "; cin >> cl;
    cout << "Sağ matrisin sütun sayısı: "; cin >> cr;

    cout << "Sol matris: " << endl;
    for (i = 0; i < rl; i++) {
        for (j = 0; j < cl; j++) {
            cout << "  [" << i + 1 << "," << j + 1<< "]: ";
            cin >> left[i][j];
        }
    }

    cout << "Sağ matris: " << endl;
    for (j = 0; j < rr; j++) {
        for (k = 0; k < cr; k++) {
            cout << "  [" << j + 1 << "," << k + 1 << "]: ";
            cin >> right[j][k];
        }
    }

    for (i = 0; i < rl; i++) {
        for (j = 0; j < cr; j++) {
            for (k = 0; k < cl; k++)
                product[i][j] = product[i][j] + left[i][k] * right[k][j];
        }
    }

    cout << "Çarpım sonucu:" << endl;
    for (i = 0; i < rl; i++) {
        for (k = 0; k < cr; k++)
            cout << "\t" << product[i][k];
        cout << endl;
    }

    return EXIT_SUCCESS;
```

Bu işlem sonucunda da bellekte 30*20*7*12=50400 tamsayı tutacak kadar yer ayrılır.

Tek boyutlu dizilerde olduğu gibi, çok boyutlu dizilerde de başlangıç değeri verilebilir. Bunun için her bir boyutun kendi içinde süslü ayraçlar içine alınması gerekir:

```
int m[2][3] = { { 1, 2, 1 }, { 3, 5, 1 } };
```

tanımı $m = \begin{bmatrix} 1 & 2 & 1 \\ 3 & 5 & 1 \end{bmatrix}$ matrisini oluşturur. Tek boyutlu dizilerdekine benzer şekilde `{ 0 }` başlangıç değeri bütün elemanlara 0 başlangıç değerini atar. Çok boyutlu dizilerde ilki dışında bütün boyutların belirtilmesi zorunludur. Sözgelimi bir isim dizisi oluşturulmak isteniyorsa

```
char names[][] = { "Dennis", "Brian", "Ken" };
```

tanımlaması derleyicinin hata vermesine neden olur. Birinci boyut dışındakiler belirtilerek yazılırsa

```
char names[][10] = { "Dennis", "Brian", "Ken" };
```

her bir elemanı en fazla 10 simge uzunluğunda olabilen 3 elemanlı bir katar dizisi tanımlanmış olur. Yani derleyici isim dizisinin eleman sayısını sayar ancak belirtilen başlangıç katarlarının uzunluklarını hesaplamaya çalışmaz. Yine de belirtilen uzunluktan daha uzun bir başlangıç katarı yazılırsa (diyelim 15 uzunluklu) derleyici bir hata üretebilir.

5.5 Başvurular

Başvurular, aynı değişkene ikinci bir isim verilmesini sağlarlar.[4] Örnekte sol matrisin satır sayısı ile sağ matrisin sütun sayılarının aynı olması zorunluluğu nedeniyle bu bilgiyi tek bir değişkenle temsil etmek yeterli görülmüş ve `cl` değişkeni tanımlanmıştır. Ancak döngülerde anlaşılırlığı artırmak amacıyla `rr` değişkeninin de tanımlanmasının yararlı olacağı düşünülerek `rr` değişkeninin `cl` değişkenine başvurması sağlanmıştır. Bu iki değişken bellekte aynı gözde bulunurlar ve dolayısıyla birinde yapılan değişiklik doğrudan doğruya diğerini etkiler.

Örnekte `rr` değişkeni başvuru olmak yerine ikinci bir değişken olarak da tanımlanabilirdi. Bu durumda `cl` değişkeni kullanıcıdan okunduktan sonra `rr = cl` şeklinde bir atamayla işlem sürdürülebilirdi. Bu yaklaşımda, iki değişken birbirinden ayrılmış olacağı için birindeki değişiklik diğerini etkilemezdi, ancak programda boyut değişkenleri üzerinde bir değişiklik olmadığı için bir sorun çıkmazdı. Tek fark, gereksiz yere ikinci bir değişken tanımlanmış olması olurdu.

[4] Başvurular, C++ ile gelmiş bir yeniliktir, C dilinde yoktur.

Uygulama: Matrisler

Örnek 17. Gauss Eliminasyon Yöntemi

$ax = b$ şeklinde yazılan n bilinmeyenli bir lineer denklem takımında a katsayı matrisini, b değişmezler vektörünü ve x çözüm vektörünü gösterir. Bu denklem takımının çözümü iki aşamalıdır:

1. a matrisi bazı dönüşümlerle bir üçgen matris haline getirilir. Bunun için önce sistemin $i = 2, 3, \ldots, n.$ denklemlerinden x_1 yok edilir (birinci denklem $\frac{a_{i1}}{a_{11}}$ ile çarpılıp $i.$ denklemden çıkarılır). Benzer şekilde, $i = 3, 4, \ldots, n.$ denklemlerden x_2 yok edilerek devam edilir. Birinci denklemin birinci elemanı olan a_{11}'e pivot adı verilir ve x_1 yok edilip ikinci denkleme geçildiğinde a_{22} elemanı pivot olur. Yöntemin çalışabilmesi için bütün pivotların sıfırdan farklı olması gerektiği açıktır.

2. x_n değeri $n.$ denklemden doğrudan hesaplanır. Bu değer $n-1.$ denklemde yerine konursa x_{n-1} bulunur ve böylece geriye doğru yerine koyma işlemleriyle bütün bilinmeyen değerler hesaplanabilir.

Denklem takımı şu şekilde verilmiş olsun:

$$\begin{aligned} x_1 + x_2 &= 3.5 \\ x_1 - x_2 + 5x_3 &= 11 \\ 2x_1 - x_2 + x_3 &= 3.3 \end{aligned}$$

a matrisi ve b vektörünün değerlerini adım adım izlersek:

başlangıçta: $a = \begin{bmatrix} 1 & 1 & 0 \\ 1 & -1 & 5 \\ 2 & -1 & 1 \end{bmatrix}$ $b = \begin{bmatrix} 3.5 \\ 11 \\ 3.3 \end{bmatrix}$

x_1'in yok edilmesinden sonra: $a = \begin{bmatrix} 1 & 1 & 0 \\ 0 & -2 & 5 \\ 0 & -3 & 1 \end{bmatrix}$ $b = \begin{bmatrix} 3.5 \\ 7.5 \\ -3.7 \end{bmatrix}$

x_2'nin yok edilmesinden sonra: $a = \begin{bmatrix} 1 & 1 & 0 \\ 0 & -2 & 5 \\ 0 & 0 & -6.5 \end{bmatrix}$ $b = \begin{bmatrix} 3.5 \\ 7.5 \\ -14.95 \end{bmatrix}$

Son denklemden $x_3 = 2.3$ elde edilir. Bu değer ikinci denklemde yerine konursa:$-2x_2 + 5x_3 = 7.5$ denkleminden $x_2 = 2$ ve bu değer birinci denklemde yerine konursa $x_1 + x_2 = 3.5$ denkleminden $x_1 = 1.5$ bulunur.

Gauss eliminasyon yöntemini kullanarak n bilinmeyenli bir denklem takımını çözen program Örnek 17'da verilmiştir.

Örnek 17 Gauss eliminasyon yöntemiyle denklem takımı çözen program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

#define MAXEQUATIONS 5

int main(void)
{
    float a[MAXEQUATIONS][MAXEQUATIONS], b[MAXEQUATIONS];
    float x[MAXEQUATIONS];
    float pivot, f;
    int n;
    int i, j, k;

    cout << "Denklem sayısı: ";
    cin >> n;
    for (i = 0; i < n; i++) {
        cout << "Denklem " << i + 1 << ": ";
        for (j = 0; j < n; j++) {
            cin >> a[i][j];
        }
        cin >> b[i];
    }

    for (j = 0; j < n - 1; j++) {
        pivot = a[j][j];
        for (i = j + 1; i < n; i++) {
            f = a[i][j] / pivot;
            for (k = j + 1; k < n; k++)
                a[i][k] = a[i][k] - f * a[j][k];
            b[i] = b[i] - f * b[j];
        }
    }

    x[n-1] = b[n-1] / a[n-1][n-1];
    for (i = n - 2; i >= 0; i--) {
        f = b[i];
        for (k = i + 1; k < n; k++)
            f = f - a[i][k] * x[k];
        x[i] = f / a[i][i];
    }

    for (i = 0; i < n; i++)
        cout << "x" << i + 1 << ": " << x[i] << endl;

    return EXIT_SUCCESS;
}
```

Sorular

1. Kullanıcının girdiği bir tümcedeki sözcük sayısını sayan bir program yazın. Sözcüklerin önünde ve arkasında boşluk olduğu varsayılacaktır. Ancak sözcük arasında istendiği sayıda boşluk olabileceği gibi tümcenin başında ve sonunda da istendiği kadar boşluk olabilir. Örnek olarak kullanıcı `“  the world  is not  enough  ”` tümcesini girerse programın çıktısı 5 olacaktır.

2. Eratosthenes kalburu yöntemini kullanarak ilk 10000 sayı içindeki asal sayıları ekrana çıkartan bir fonksiyon yazın.

3. Kullanıcıdan alınan bir sayıyı yazı olarak ekrana çıkartan bir program yazın. Örneğin kullanıcı 21355 sayısını girerse program ekrana “Yirmibirbinüçyüzellibeş” yazacaktır.

4. Kullanıcının verdiği sayıyı Roma rakamlarına çevirerek ekrana çıkartan bir program yazın. Örneğin kullanıcı 523 sayısını girerse ekrana “DXXIII” katarı çıkarılmalıdır.

5. Verilen bir tarihin haftanın hangi gününe geldiğini bulan bir program yazın. Örneğin kullanıcı “24-09-2002” tarihini girerse ekrana “Salı” katarı çıkarılmalıdır.

6. Kullanıcıdan alınan iki tarih arasında geçen gün sayısını hesaplayan bir program yazın.

7. Kullanıcıdan aldığı bir tarihin haftanın hangi gününe geldiğini bulan bir program yazın.

Bölüm 6

Fonksiyonlar

Şu ana kadar yapılan örneklerde bütün işlemler tek bir fonksiyonda (`main` fonksiyonu) gerçekleştiriliyordu. Gereken yerlerde kitaplık fonksiyonlarından yararlanılmakla birlikte, kendi yazdığımız kod tek bir fonksiyonla sınırlıydı. Oysa blok yapılı programlamada soyutlama kavramının öneminden Bölüm 1.4'de söz edilmiş, işleri alt-işlere, alt-işleri alt-alt-işlere bölerek yazılan programların hem geliştirme hem de bakım aşamalarında sağlayacağı kolaylıklar görülmüştü. Bu bölümde de programcının kendi fonksiyonlarını nasıl yazacağını inceleyeceğiz.

C dilinde her iş ya da alt-iş bir fonksiyon tarafından gerçeklenir. Fonksiyonlar yapacakları işi belirleyen giriş parametreleri alırlar ve işlemin sonucuna göre *bir* çıkış parametresi üretirler. Yani giriş parametrelerinin sayısı birden fazla olabilir ancak çıkış parametresi en fazla bir tanedir. Şimdiye kadar kullanılan kitaplık fonksiyonlarının bazılarına parametreleri açısından bakarsak:

- `sqrt`: Kesirli sayı tipinden bir giriş parametresi alır, kesirli sayı tipinden bir çıkış parametresi üretir.
- `pow`: İkisi de kesirli sayı tipinden iki giriş parametresi alır, kesirli sayı tipinden bir çıkış parametresi üretir.
- `strlen`: Katar tipinden bir giriş parametresi alır, tamsayı tipinden bir çıkış parametresi üretir.
- `rand`: Giriş parametresi almaz, tamsayı tipinden bir çıkış parametresi üretir.
- `srand`: İşaretsiz tamsayı tipinden bir giriş parametresi alır, çıkış parametresi üretmez.

Bir fonksiyonun bir alt-işini yaptırmak üzere başka bir fonksiyonu kullanmasına fonksiyon *çağrısı* adı verilir. Örnek 7'da, `main` fonksiyonu, `for` bloğunun ilk komutunda `rand` fonksiyonunu çağırmaktadır. Fonksiyon çağrısında üst fonksiyona *çağıran* fonksiyon, alt fonksiyona da *çağrılan* fonksiyon denir, yani örnekte `main` fonksiyonu çağıran, `rand` fonksiyonuysa çağrılan fonksiyondur. Fonksiyon çağrısının sona ermesinden sonra çağıran fonksiyonda akış bir sonraki komutla devam eder.

Çağıran fonksiyon çağrılan fonksiyona giriş parametrelerini gönderir, çağrılan fonksiyonsa ürettiği sonucu çağıran fonksiyona döndürür. Döndürülen değer çağıran fonksiyonda bir değişkene

atanabileceği ya da bir deyimde kullanılabileceği gibi, doğrudan başka bir fonksiyona giriş parametresi olarak da gönderilebilir.[1] Örnekte `time` fonksiyonunun döndürdüğü değer, `srand` fonksiyonuna parametre olarak gönderilmiştir:

```
number = rand();              // doğrudan atama
number = rand() % 6;          // deyimde kullanma
srand(time(NULL));            // başka fonksiyona gönderme
```

Çıkış parametrelerinin kullanımında tip uyumuna dikkat edilmelidir. Sözgelimi, `rand` fonksiyonu tamsayı tipinden bir değer döndürdüğüne göre `number` değişkeni de tamsayı tipinden olmalıdır, katar tipinden olamaz.[2] Benzer şekilde, bir fonksiyondan gelen bir değer kalan işlemine sokulacaksa bu değerin tamsayı tipinden olmasına dikkat etmek gerekir.

Giriş parametrelerinin aktarımı için çağıran fonksiyondaki parametreleri yollama şekliyle çağrılan fonksiyonun parametreleri bekleme şekli uyumlu olmalıdır. Bu uyumluluk şu şekilde tanımlanabilir:

1. Çağrılan fonksiyon kaç parametre bekliyorsa çağıran fonksiyon o sayıda parametre yollamalıdır.

2. Çağrılan fonksiyon her bir sıradaki giriş parametresinin tipinin ne olmasını bekliyorsa çağıran fonksiyon bu tipe uyumlu bir değer göndermelidir.

Örneğin `pow` fonksiyonu iki tane kesirli sayı bekliyorsa bu fonksiyona iki tane kesirli sayı (ya da tamsayı) değeri yollanmalıdır. Bir, üç ya da daha fazla değer yollanması, hiç değer yollanmaması, yollanan iki değerin sayı dışında bir tipten (sözgelimi katar) olması hataya yol açar.

Örnek 18. Asal Çarpanlara Ayırma

Kullanıcının girdiği bir sayının asal çarpanlarını ekrana döken bir program yazılması isteniyor. Gerekli algoritmanın akış çizeneği Şekil 1.18'de, örnek bir çalışmasının ekran çıktısı Şekil 6.1'de verilmiştir. Programda bir sayının asal olup olmadığının sınanmasına ve asal sayıların bulunmasına gerek duyulacaktır. Soyutlama ilkesine göre bu işlemler fonksiyonlarla gerçekleştirilecek, ana fonksiyon, asal sayıları bulan fonksiyondan sırayla aldığı asal sayıların kullanıcının verdiği sayıyı bölüp bölmediklerini sınayacaktır. Örnekte `is_prime` fonksiyonu kendisine gönderilen bir sayının asal olup olmadığını belirleyerek geriye asalsa doğru, değilse yanlış mantıksal değerini yollar. `next_prime` fonksiyonu ise kendisine gönderilen sayıdan daha büyük olan ilk asal sayıyı bularak kendisini çağıran fonksiyona döndürür.

[1] Aslında bu durumların hepsi deyimde kullanma olarak değerlendirilebilir.

[2] Eski C derleyicileri bu tip denetimlerde fazlasıyla gevşek davranabilmektedir. Bu durum programcının hata yapmasına zemin hazırlar.

Örnek 18 Bir sayıyı asal çarpanlarına ayıran program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

int next_prime(int prime);

int main(void)
{
    int number, factor = 2;

    cout << "Sayıyı yazınız: ";
    cin >> number;
    while (number > 1) {
        while (number % factor == 0) {
            cout << factor << " ";
            number = number / factor;
        }
        factor = next_prime(factor);
    }
    cout << endl;
    return EXIT_SUCCESS;
}

bool is_prime(int cand)
{
    int count;

    if (cand == 2)
        return true;
    if (cand % 2 == 0)
        return false;
    for (count = 3; count * count <= cand; count += 2) {
        if (cand % count == 0)
            return false;
    }
    return true;
}

int next_prime(int prime)
{
    int cand = (prime == 2) ? 3 : prime + 2;

    while (!is_prime(cand))
        cand += 2;
    return cand;
}
```

```
Sayıyı yazınız: 6468
2 2 3 7 7 11
```

Şekil 6.1: Örnek 18 ekran çıktısı.

6.1 Fonksiyon Bildirimi ve Tanımı

Bir fonksiyon iki bileşenden oluşur:

- Fonksiyonun *başlığı*, fonksiyonun adını ve giriş/çıkış parametrelerini belirtmeye yarar. Soyutlamadaki karşılığıyla fonksiyonun NE yaptığını anlatır. Fonksiyon başlığının belirtilmesi işlemine fonksiyonun *bildirimi* denir ve başlığın sonuna bir noktalı virgül konarak yapılır. Bildirim komutunun yazımı şu şekildedir:

  çıkış_parametresi_tipi fonksiyon_adı(*giriş_parametresi_listesi*);

  Örnekte `next_prime` fonksiyonu için yapılan

  ```
  int next_prime(int prime);
  ```

  bildirimine göre, `next_prime` fonksiyonu tamsayı tipinden bir giriş parametresi alır ve yine tamsayı tipinden bir çıkış parametresi döndürür.

  Çıkış parametresi herhangi bir skalar ya da bileşke tipten olabilir ancak dizi olamaz. Giriş parametresi listesiyse birbirinden virgülle ayrılmış *veri_tipi değişken_adı* çiftleri şeklindedir. Çıkış parametresi döndürmeyen fonksiyonların çıkış parametresi tipi `void` saklı sözcüğüyle belirtilir. Benzer şekilde, giriş parametresi almayan fonksiyonların giriş listesi parametresine de `void` yazılır. Bu bilgileri gözönünde bulundurarsak, kullandığımız bazı kitaplık fonksiyonlarının sistemde şöyle bildirilmiş olmaları gerektiği görülebilir:

  ```
  double sqrt(double x);
  double pow(double x, double y);
  int rand(void);
  void srand(unsigned int seed);
  ```

  Giriş parametresi listesindeki değişkenlerin yazımı değişken tanımı yazımına benzer ancak aynı tipten değişkenler gruplanamaz. Sözgelimi aşağıdaki bildirim hatalıdır:

  ```
  double pow(double x, y);
  ```

  Fonksiyon bildirimi, derleyiciye giriş parametresi listesi tipinden değişkenler alan ve çıkış parametresi tipinden değer üreten belirtilen isimde bir fonksiyon olduğununun bildirilmesini sağlar. Böylelikle derleyici bu fonksiyon çağrısını gördüğü yerde parametrelerin uyumlu olup olmadıklarını denetleyebilir. Çalışma anında fonksiyon çağrısına gelindiğinde uygun yere gidilmesini sağlayacak bağlantıları kurmak derleyicinin değil bağlayıcının görevidir.

- Fonksiyonun *gövdesi*, fonksiyonun yapacağı işleri belirten bloktan oluşur, yani fonksiyonun işini NASIL yaptığını anlatır. Bir fonksiyonun bütününün, yani giriş/çıkış parametrelerinin yanısıra gövdesinin de belirtildiği yere fonksiyonun *tanımı* denir. Örnekte `is_prime` fonksiyonu, `main` fonksiyonunun bitiminden sonra, `next_prime` fonksiyonu da `is_prime` fonksiyonunun bitiminden sonra tanımlanmıştır.

  Çağrılan fonksiyon ürettiği değeri çağıran fonksiyona `return` komutu yardımıyla döndürür. `return` saklı sözcüğünden sonra yazılan deyim fonksiyonun başlığında belirtilen veri tipine uygun tipten bir değer üretmelidir. Örnekte `is_prime` fonksiyonu mantıksal tipten tanımlanmıştır ve fonksiyonun çeşitli noktalarında `false` ya da `true` değerlerini döndürmektedir. `next_prime` fonksiyonuysa tamsayı tipinden tanımlanmıştır ve döndürdüğü değeri tutan değişken de (`cand`) tamsayı tipindendir.

Bir fonksiyon çağrısının yapılabilmesi için çağrılan fonksiyonun ya bildirimi ya da tanımı çağıran fonksiyondan önce yapılmalıdır. Örnekte `main` fonksiyonu `next_prime` fonksiyonunu, `next_prime` fonksiyonu da `is_prime` fonksiyonunu çağırır. Birinci çağrı, `next_prime` fonksiyonunun bildirimi `main` fonksiyonunun tanımı başlamadan yapıldığından başarılı olur. İkinci çağrıysa `is_prime` fonksiyonu `next_prime` fonksiyonundan önce tanımlandığından başarılı olur. Ancak örneğin `main` fonksiyonu `is_prime` fonksiyonunu çağıramaz çünkü öncesinde `is_prime` fonksiyonunun ne tanımı ne de bildirimi vardır.

Kendi yazdığınız fonksiyonlarda olduğu gibi, kitaplık fonksiyonlarını da çağırabilmeniz için bu fonksiyonların bildirimlerinin daha önceden yapılmış olması gerekir. Bu bildirimler kullandığınız derleyiciyle gelen başlık dosyalarında yer alır; bir kitaplık fonksiyonunu kullandığınızda başlık dosyasını belirtmeniz zorunluluğu da buradan doğar.

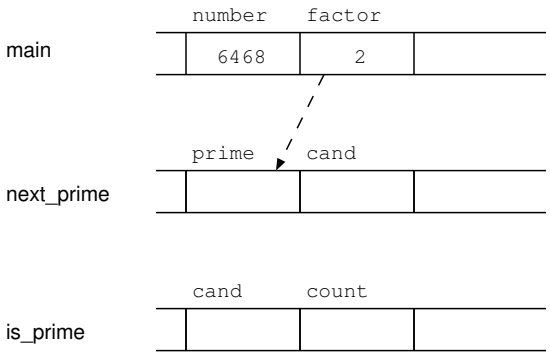
6.2 Parametre Aktarımı

Giriş parametrelerinin aktarımında fonksiyon çağrısında belirtilen parametre listesiyle fonksiyonun başlığında belirtilen liste uyumlu olmalıdır. Çağrıda yazılan her değer, çağrılan fonksiyonun başlığında aynı sırada yazılmış değişkene atanır. Bu parametre aktarım yöntemine *değer aktarımı* adı verilir. Örnekteki birinci fonksiyon çağrısında:

```
factor = next_prime(factor);
```

`main` fonksiyonundaki `factor` değişkeninin değeri `next_prime` fonksiyonunun giriş parametresi olan `prime` değişkenine atanır. Dönüşte de `next_prime` fonksiyonunun `return` komutuyla geri yolladığı `cand` değişkeninin değeri `main` fonksiyonundaki `factor` değişkenine atanır. Fonksiyonun bir döngü içinde çağrıldığı gözönüne alınarak, parametre olarak ilk çağrıda 2 değerinin gönderileceği ve 3 değerinin döneceği, ikinci çağrıda 3 değerinin gönderileceği ve 5 değerinin döneceği, ileriki çağrılarda 5, 7, 11, ... değerlerinin gönderileceği görülebilir. Benzer şekilde, ikinci fonksiyon çağrısında `next_prime` fonksiyonundaki `cand` değişkeninin değeri `is_prime` fonksiyonunun giriş parametresi olan `cand` değişkenine atanır.

Tipi uygun olduğu sürece parametre olarak herhangi bir deyim belirtilebilir. Sözgelimi aşağıdaki çağrılar geçerlidir:



Şekil 6.2: Parametre aktarımı örneği - 1. adım.

```
is_prime(13)
is_prime(cand + 1)
```

Benzer şekilde, geri döndürülecek değer için uyumlu tipten değer üreten bir deyim yazılabilir:

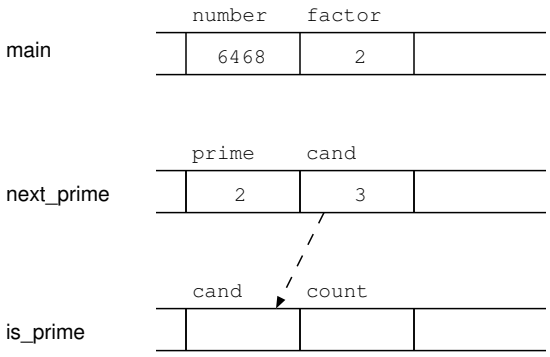
```
return cand + 2;
```

6.3 Yerel Değişkenler

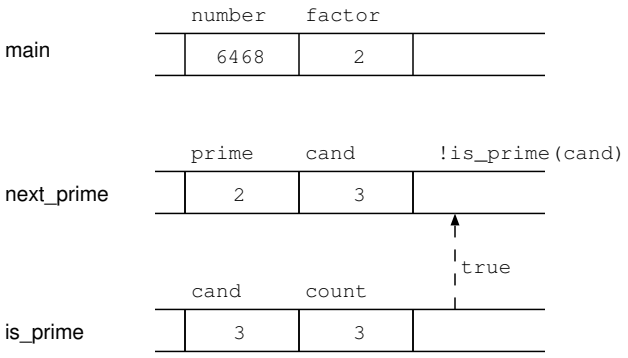
Parametre aktarımı anlatılırken "`main` fonksiyonundaki `factor` değişkeninin değeri `next_prime` fonksiyonunun giriş parametresi olan `prime` değişkenine" , "`next_prime` fonksiyonundaki `cand` değişkeninin değeri `is_prime` fonksiyonunun giriş parametresi olan `cand` değişkenine" gibi sözler kullanıldı. Buradan da görülebileceği gibi, değişkenler içinde tanımlandıkları fonksiyon ile birlikte değerlendirilirler ve yalnızca bu fonksiyon içinde geçerlidirler. `main` fonksiyonundaki `factor` değişkeni ile `next_prime` fonksiyonundaki `prime` değişkeni farklı değişkenlerdir, bellekte farklı yerlerde bulunurlar; dolayısıyla, birinde yapılan değişiklik diğerini etkilemez. Değişken adlarının aynı olmasının da bir önemi yoktur: `next_prime` fonksiyonundaki `cand` değişkeniyle `is_prime` fonksiyonundaki `cand` değişkeni de farklı değişkenlerdir.

Örnekte ilk fonksiyon çağrıları şöyle gerçekleşir. `main` fonksiyonunda `factor` değişkenine başlangıçta atanmış olan 2 değeri `next_prime` fonksiyonunun `prime` adlı giriş parametre değişkenine aktarılır (Şekil 6.2). `next_prime` fonksiyonunda `cand` değişkeni 3 değerini alır ve bu değer asal olup olunmadığının belirlenmesi için `is_prime` fonksiyonunun `cand` adlı giriş parametre değişkenine aktarılır (Şekil 6.3). `is_prime` fonksiyonunda `count` değişkeni 3 değerini alır ancak döngü koşulu baştan sağlanmadığından döngüden çıkılarak `next_prime` fonksiyonuna `true` değeri döndürülür (Şekil 6.4). Bu değer `!is_prime(cand)` koşul deyiminin yanlış sonucunu üretmesine neden olduğundan döngüden çıkılır ve `main` fonksiyonuna `cand` değişkeninin o anki değeri, yani 3 döndürülür. Bu değer de `main` fonksiyonundaki `factor` değişkenine atanır (Şekil 6.5)

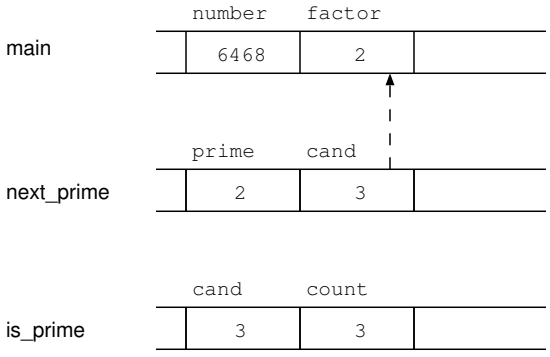
Fonksiyonun parametre olarak aldığı ya da fonksiyon gövdesinde tanımlanan değişkenlere o fonksiyonun *yerel* değişkenleri adı verilir ve bunların *tanım bölgesi* tanımlandıkları fonksiyonla



Şekil 6.3: Parametre aktarımı örneği - 2. adım.



Şekil 6.4: Parametre aktarımı örneği - 3. adım.



Şekil 6.5: Parametre aktarımı örneği - 4. adım.

sınırlanır. Örnekteki bütün fonksiyonların ikişer (`main`: `number` ve `factor`, `is_prime`: `cand` ve `count`, `next_prime`: `prime` ve `cand`) yerel değişkeni vardır. Fonksiyonun sona ermesiyle tanımladığı değişkenler de geçersiz hale gelir. Başka bir deyişle, değişkenlere tanım bölgeleri dışında erişilemez. Yani sözgelimi `main` fonksiyonu `number` ve `factor` dışında kalan değişkenleri kullanamaz.

6.4 Genel Değişkenler

Bazı durumlarda birden fazla fonksiyonun aynı değişkeni kullanabilmeleri istenir. Bu tip değişkenlere *genel* değişken adı verilir ve tanım bölgeleri bütün fonksiyonlar olarak belirlenir. Genel değişkenler bütün fonksiyonların tanımlarından önce tanımlanırlar.

DİKKAT Genel değişkenlere örnek vermek için programda bazı değişiklikler yapılabilir. BU DEĞİŞİKLİKLERİN SONUCUNDA OLUŞACAK PROGRAM İYİ BİR PROGRAM OLMAYACAKTIR. BU ÖRNEK YALNIZCA GENEL DEĞİŞKEN KULLANIMINI AÇIKLAMAK AMACIYLA VERİLMİŞTİR. `next_prime` ve `is_prime` fonksiyonları aslında `cand` değişkenini parametre olarak aktarmak yerine ortak bir bellek gözünde paylaşabilirler. Bu durumda `is_prime` fonksiyonunun işlevi bu değişkenin değerinin asal olup olmadığını belirlemek, `next_prime` fonksiyonunun işleviyse bu değişkenin değerini kendisinden büyük ilk asal sayıya ilerletmek olarak düşünülebilir. Benzer şekilde, ana fonksiyon da çarpan adayı olan sayıyı parametre aktarımıyla almak yerine aynı genel değişkende erişerek öğrenebilir. Değişiklikler sonucu oluşan program Örnek 19'de verilmiştir.

Bir fonksiyon bir genel değişkenle aynı isimde bir yerel değişken tanımlarsa o fonksiyonun çalışması boyunca yerel değişken genel değişkeni ezer. Diyelim `is_prime` fonksiyonu

```
int count, cand;
```

şeklinde bir değişken tanımı yapmış olsaydı, genel olan `cand` değişkeninin yanında bir de yerel olan `cand` değişkeni oluşurdu ve `is_prime` fonksiyonunun içindeyken genel olan `cand` değişkenine ulaşılamazdı.

Genel değişkenler parametre aktarımı yerine kullanılabilecek bir yöntem olmakla birlikte programın anlaşılırlığını azalttıkları ve fonksiyonları birbirlerine bağımlı kıldıkları için gerekmedikçe kullanılmamaları önerilir. Soyutlamanın kazandırdıklarından birinin ileride bir fonksiyonda değişiklik yapıldığı zaman bunu çağıran fonksiyonların değişiklikten etkilenmemesi olduğu söylenmişti; oysa fonksiyonların değişken paylaşırlarsa bir fonksiyonda yapılan değişiklikler diğer fonksiyonları etkileyebilir.

6.5 Başvuru Aktarımı

Değer aktarımı yöntemiyle parametre aktarılması bazı işlemlerde yeterli olmaz. Sözgelimi, kendisine giriş parametresi olarak gönderilen iki tamsayı değişkenin değerlerini takas eden bir fonksiyonun Örnek 20'daki gibi yazıldığını düşünelim.

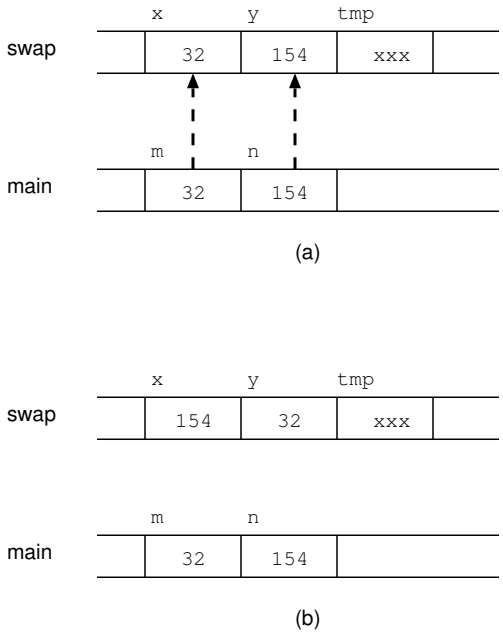
Bu program çalıştırıldığında ekran çıktısı şu şekilde olur:

Örnek 19 Genel değişken kullanarak bir sayıyı asal çarpanlarına ayıran program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

int cand = 2;

void next_prime(void);

int main(void)
{
    int number;

    cout << "Sayıyı yazınız: ";
    cin >> number;
    while (number > 1) {
        while (number % cand == 0) {
            cout << cand << " ";
            number = number / cand;
        }
        next_prime();
    }
    cout << endl;
    return EXIT_SUCCESS;
}

bool is_prime(void)
{
    int count;

    if (cand == 2)
        return true;
    if (cand % 2 == 0)
        return false;
    for (count = 3; count * count <= cand; count += 2) {
        if (cand % count == 0)
            return false;
    }
    return true;
}

void next_prime(void)
{
    cand = (cand == 2) ? 3 : cand + 2;
    while (!is_prime())
        cand += 2;
}
```

Örnek 20 İki sayıyı takas eden fonksiyon ve kullanımı (hatalı).

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

void swap(int x, int y)
{
    int tmp;

    tmp = x;
    x = y;
    y = tmp;
}

int main(void)
{
    int m = 32, n = 154;

    cout << m << " " << n << endl;
    swap(m, n);
    cout << m << " " << n << endl;
    return EXIT_SUCCESS;
}
```



Şekil 6.6: Takas fonksiyonunda parametre aktarımı.

```
32 154
32 154
```

Programın neden istendiği gibi çalışmadığını inceleyelim: `main` fonksiyonundaki `m` değişkeninin değeri `swap` fonksiyonundaki `x` değişkenine, `n` değişkeninin değeri de `y` değişkenine aktarılır (Şekil 6.6a). `swap` fonksiyonunda `x` ve `y` değişkenlerinin değerleri takas edilir ancak çağıran fonksiyondaki `m` ve `n` değişkenleri bu takastan etkilenmezler ve fonksiyon çağrısından dönüldüğünde eski değerlerini koruyor olurlar (Şekil 6.6b).

Bu sorunun çözümü için yapılması gereken, çağrılan fonksiyonda girdi parametrelerini başvuru tipinden tanımlamaktır (bkz. Bölüm 5.5). Böylelikle çağrılan fonksiyonun girdi parametresi, çağıran fonksiyonun gönderdiği parametreye verilmiş ikinci bir isim haline gelir ve üstünde yapılan değişiklikler çağıran fonksiyondaki değişkeni doğrudan etkiler: Bu parametre aktarım yöntemine *başvuru aktarımı* adı verilir. Bir parametrenin başvuru olarak aktarıldığını göstermek üzere yapılması gereken tek şey çağrılan fonksiyondaki girdi parametre değişkeninin adının başına `&` simgesi koymaktır. Yani örnekteki tek değişiklik `swap` fonksiyonunun başlık satırının şu şekle getirilmesi olacaktır:

```
void swap(int &x, int &y)
```

Uygulama: Fonksiyonlar

Örnek 21. Kombinasyon Hesabı

Bu örnekte $C_n^m = \frac{m!}{n!(m-n)!}$ değerinin hesaplanması istenmektedir.

Örnek 21 Kombinasyon hesabı (yavaş).

```
#include <iostream.h>           // cout,cin
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

int combin(int a, int b);
int fact(int x);

int main(void)
{
    int n, r;

    cout << "n ve r değerlerini yazınız: ";
    cin >> n >> r;
    cout << combin(n, r) << endl;
    return EXIT_SUCCESS;
}

int combin(int a, int b)
{
    int f1, f2, f3;

    f1 = fact(a);
    f2 = fact(b);
    f3 = fact(a - b);
    return f1 / (f2 * f3);
}

int fact(int x)
{
    int fx = 1;
    int i;

    for (i = 2; i <= x; i++)
        fx = fx * i;
    return fx;
}
```

Örnek 22 Kombinasyon hesabı (hızlı).

```
#include <iostream.h>           // cout,cin
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

int combin(int a, int b);

int main(void)
{
    int n, r;

    cout << "n ve r de?erlerini yaz?n?z: ";
    cin >> n >> r;
    cout << combin(n, r) << endl;
    return EXIT_SUCCESS;
}

int combin(int a, int b)
{
    int f1 = 1, f2, f3;
    int first = b, second = a - b;
    int i;

    if (b > a - b) {
        first = a - b;
        second = b;
    }

    for (i = 2; i <= first; i++)
        f1 = f1 * i;
    f2 = f1;
    for (i = first + 1; i <= second; i++)
        f2 = f2 * i;
    f3 = f2;
    for (i = second + 1; i <= a; i++)
        f3 = f3 * i;
    return f1 / (f2 * f3);
}
```

Örnek 23. En Büyük Ortak Bölen Bulma

Bölüm 1.4'de anlatılan algoritmaları kullanarak iki sayının en büyük ortak bölenini hesaplayan bir program yazılması isteniyor. Bunun için Örnek 18'de bir sayıyı asal çarpanlarına ayıran programın benzeri bir fonksiyon (`factorize`) olarak yazılacak ve bu fonksiyon kendisine parametre olarak gönderilen sayının asal çarpanlarını bir dizide oluşturacaktır. Fonksiyon yine aynı örnekte yazılmış olan `next_prime` ve `is_prime` fonksiyonlarını hiçbir değişiklik olmadan kullanacaktır. Asal çarpan dizilerinden en büyük ortak bölenin asal çarpanlarını bulmak üzere `gcd_factors` fonksiyonu kullanılacak, bu fonksiyonun oluşturduğu çarpanlar dizisinden en büyük ortak böleni ana fonksiyon kendisi hesaplayacaktır.

Örnek 23 İki sayının en büyük ortak bölenini bulan program (ana fonksiyon).

```
int main(void)
{
    int number1, number2;
    factor_t factors1[MAXFACTOR], factors2[MAXFACTOR], factors3[MAXFACTOR];
    int n1, n2, n3;
    long int gcd = 1L;
    int i;

    cout << "Sayıları yazınız: ";
    cin >> number1 >> number2;
    factorize(number1, factors1, n1);
    factorize(number2, factors2, n2);
    gcd_factors(factors1, n1, factors2, n2, factors3, n3);
    for (i = 0; i < n3; i++)
        gcd = gcd * (long int) pow((double) factors3[i].base,
                                   (double) factors3[i].power);
    cout << "En büyük ortak bölen: " << gcd << endl;
    return EXIT_SUCCESS;
}
```

6.6 Giriş Parametreleri Üzerinden Değer Döndürme

Asal çarpanlara ayırma işini yapan `factorize` fonksiyonunun giriş parametresinin çarpanlarına ayrılacak sayı, çıkış parametresinin de çarpanlar dizisi olması gerektiği görülmüştü. Burada iki sorunla karşılaşılır:

1. C dilinde çıkış parametresi olarak geriye dizi döndürülemez.

2. Dizinin elemanlarını döndürmek yeterli değildir, dizide kaç tane geçerli eleman olduğunu da döndürmek gerekir. Oysa C fonksiyonları çağıran fonksiyona birden fazla değer döndüremez.

Birden fazla değerin döndürülmesi gerektiği durumlarda bu değerler giriş parametrelerinde değişiklik yapma yöntemiyle ana fonksiyona aktarılabilir. Buna göre, `factorize` fonksiyonunun işlevi şöyle açıklanabilir:

> Birinci giriş parametresi olarak gönderilen sayının çarpanlarını ikinci giriş parametresi olan dizide oluşturur ve bu dizinin kaç elemanı olduğunu üçüncü giriş parametresine yazar. Geriye bir değer döndürmez.

Buna göre koddaki

```
factorize(number1, factors1, n1);
```

fonksiyon çağrısının işlevi:

> `number1` sayısını asal çarpanlarına ayır, sonuçları `factors1` dizisinde oluştur ve bu dizideki geçerli eleman sayısını `n1` değişkenine yaz.

6.7 Dizilerin Fonksiyonlara Aktarılması

Bir giriş parametresinin dizi tipinden olduğunun belirtilmesi için fonksiyon başlığında değişken adının ardına boyut belirtmeden köşeli ayraçlar konur. Dizi değişkenlerinin giriş parametresi olarak aktarımlarında en önemli özellik, aksi belirtilmedikçe dizi elemanlarının değiştirilebilir olmalarıdır. Yani dizi elemanları değiştirilmek isteniyorsa dizi değişkeninin adı olduğu gibi yazılır, başvuru aktarımı kullanılmaz (değişken adının başına `&` işareti konmaz).[3] Aksine, dizi elemanlarının değer *değiştirmemesi* isteniyorsa önlem almak gerekir. Bu amaçla giriş parametresi tanımı `const` saklı sözcüğüyle başlatılır.

Örnekte sonuçların oluştuğu `factors1` dizisi ve `n1` değişkeninin ana fonksiyonun yerel değişkenleri olduğuna dikkat edilmelidir. Fonksiyon çağrısından önce bu değişkenlerde herhangi anlamlı bir değer yer almazken çağrıdan dönüldükten sonra anlamlı değerlerle doldurulmuş olmaları beklenmektedir. Bu nedenle, yukarıda belirtilen yazım kurallarına göre, `factorize` fonksiyonunun başlığı şöyle olmalıdır:

```
void factorize(int number, factor_t factors[], int &n)
```

biçimindedir. Bu fonksiyonun tam çıktısı Örnek 24'de verilmiştir.

Ortak çarpanları bulan fonksiyon ise iki tane asal çarpan dizisi alacak ve ortak çarpanlardan bir asal çarpan dizisi oluşturacaktır. Dizilerin eleman sayıları da parametre olarak gönderilmesi gerektiği için her çarpan dizisi, bir eleman dizisi ve bir eleman sayısı ile gösterilecektir. Buna göre `gcd_factors` fonksiyonunun işlevi şu şekilde yazılabilir:

[3] Bunun nedeni Bölüm 7'de açıklanacaktır.

Örnek 24 İki sayının en büyük ortak bölenini bulan program (asal çarpanlara ayırma fonksiyonu).

```
void factorize(int x, factor_t factors[], int &n)
{
    int factor = 2;

    n = 0;
    while (x > 1) {
        if (x % factor == 0) {
            factors[n].base = factor;
            factors[n].power = 0;
            while (x % factor == 0) {
                factors[n].power++;
                x = x / factor;
            }
            n++;
        }
        factor = next_prime(factor);
    }
}
```

> Birinci parametrede verilen çarpanları içeren ve ikinci parametrede verilen sayıda elemanı olan çarpan dizisiyle, üçüncü parametrede verilen çarpanları içeren ve dördüncü parametrede verilen sayıda elemanı olan çarpan dizilerinden ortak çarpanları bulur. Bu dizinin elemanlarını beşinci parametrede verilen dizide oluşturur ve bu dizinin eleman sayısını altıncı parametreye yazar. Geriye bir değer döndürmez.

Bu fonksiyonda ilk dört parametre gerçek anlamda giriş parametresiyken son iki parametre aslında çıktı parametresi olup kısıtlamalar nedeniyle giriş parametresi olarak gönderilen değerlerdir Yani ilk dört parametre değişmeyecek, son iki parametre değişecektir. Buna göre `gcd_factors` fonksiyonunun başlığı şöyle olmalıdır:

```
void gcd_factors(const factor_t factors1[], int n1,
                 const factor_t factors2[], int n2,
                 factor_t factors[], int &n);
```

Burada `const` sözcüklerinin kullanılmaması programın doğru çalışmasına etki etmez, yalnızca programcının aslında değişmemesi gereken bir değeri yanlışlıkla değiştirmesine engel olmak için konmuştur. Fonksiyonun tanımı Örnek 25'de verilmiştir.

Uygulama: Dizilerin Fonksiyonlara Aktarılması

Bu uygulamada profiler kullanımı gösterilecektir (gprof).

Örnek 25 İki sayının en büyük ortak bölenini bulan program (ortak çarpanları bulma fonksiyonu).

```
void gcd_factors(const factor_t factors1[], int n1,
                 const factor_t factors2[], int n2,
                 factor_t factors[], int &n)
{
    int i1 = 0, i2 = 0;

    n = 0;
    while ((i1 < n1) && (i2 < n2)) {    // iki dizi de bitmedi
        if (factors1[i1].base < factors2[i2].base)
            i1++;
        else if (factors1[i1].base > factors2[i2].base)
            i2++;
        else {
            factors[n].base = factors1[i1].base;
            if (factors1[i1].power < factors2[i2].power)
                factors[n].power = factors1[i1].power;
            else
                factors[n].power = factors2[i2].power;
            i1++;
            i2++;
            n++;
        }
    }
}
```

Örnek 26. Newton-Raphson Yöntemiyle Polinom Kökü Bulunması

Bir $f(x)$ fonksiyonunun kökü, $f(x) = 0$ koşulunu sağlayan x değeridir. Bu değere $\overline{x}$ dersek $f(x)$ fonksiyonunun $\overline{x}$ civarında Taylor açılımının ilk iki terimi şöyle yazılabilir

$$f(\overline{x}) = f(x_i) + (\overline{x} - x_i)f'(x_i)$$

Bu değer 0 olması gerektiğinden denklem

$$f(x_i) + (\overline{x} - x_i)f'(x_i) = 0$$

şekline getirilebilir ve buradan da

$$\overline{x} = x_i - \frac{f(x_i)}{f'(x_i)}$$

yazılabilir. Bu formülde $\overline{x}$ yerine x_{i+1} konursa, fonksiyonun kökü ardışıl yerine koyma yöntemiyle hesaplanabilir. Yani her adımda o adımdaki x değeri formüldeki x_i değerinin yerine konarak bir sonraki adımda kullanılacak x değeri hesaplanır.

Bu yöntemin $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ şeklinde n. dereceden bir polinoma uygulandığını varsayalım. O halde yineleme formülü

$$x_{i+1} = x_i - \frac{p(x_i)}{p'(x_i)}$$

olacaktır. Bir polinomun hesaplanmasının içiçe çarpma ve toplama yöntemiyle nasıl hızlandırılacağı Örnek 15'de görülmüştü. O örnekteki $b$değerleri bir dizide tutularak hesaplanırsa:

$$\begin{aligned} b_n &= a_n \\ b_{n-1} &= b_n x_i + a_{n-1} \\ b_0 &= b_1 x_i + a_0 \end{aligned}$$

Bu durumda, $p'(x)$ polinomu şu şekilde yazılabilir (Neden?):

$$p'(x) = b_n x^{n-1} + b_{n-1}x^{n-2} + \cdots + b_2 x + b_1$$

Bu da bir polinom olduğundan, hesabında yine içiçe çarpımlar yöntemi kullanılabilir:

$$\begin{aligned} c_n &= b_n \\ c_{n-1} &= c_n x_i + b_{n-1} \\ c_1 &= c_2 x_i + b_1 \end{aligned}$$

Bu yöntemi kullanarak katsayılarını kullanıcının girdiği bir polinomun köklerini hesaplayan program Örnek 26'de verilmiştir.

Örnek 26 Polinom kökü hesaplayan fonksiyon.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS
#include <math.h>               // fabs

using namespace std;

#define MAXDEGREE 50

float newton_raphson(float x, const float a[], int n);

int main(void)
{
    float a[MAXDEGREE];
    int n, i;
    float xi, xj, error;

    cout << "Polinomun derecesi: "; cin >> n;
    for (i = n; i >= 0; i--) {
        cout << "a" << i << ": ";
        cin >> a[i];
    }

    cout << "Hata: "; cin >> error;
    cout << "x0: ";   cin >> xi;
    while (true) {
        xj = newton_raphson(xi, a, n);
        if (fabs(xj - xi) < error)
            break;
        xi = xj;
    }
    cout << "Kök: " << xj << endl;
    return EXIT_SUCCESS;
}

float newton_raphson(float x, const float a[], int n)
{
    float b[MAXDEGREE], c[MAXDEGREE];
    float xn;
    int i;

    b[n] = a[n];
    c[n] = b[n];
    for (i = n - 1; i > 0; i--) {
        b[i] = b[i+1] * x + a[i];
        c[i] = c[i+1] * x + b[i];
    }
    b[0] = b[1] * x + a[0];
    xn = x - b[0] / c[1];
    return xn;
}
```

- Bu programı kullanarak

$$p(x) = 13x^7 - 9x^6 + 12x^4 - 4x^3 + 3x + 5$$

polinomunun kökünü $x_0 = 1$ değerinden başlayarak, 0.001 hatadan daha küçük olacak şekilde bulun.

6.8 Eş İsimli Fonksiyonlar

C++ dilinde giriş parametre listeleri farklı olduğu sürece birden fazla fonksiyonun isimlerinin aynı olması bir sorun yaratmaz.[4] Örneğin, iki giriş parametresini takas eden fonksiyonlarda isimlerin aynı olmasına izin verilmese iki tamsayıyı takas edecek fonksiyona `swap_int`, iki kesirli sayıyı takas edecek fonksiyona `swap_float`, iki katarı takas edecek fonksiyona `swap_str` gibi isimler vermek gerekir. Oysa giriş parametreleri farklı tiplerden olduğu için üç fonksiyona da `swap` adı verilebilir:

```
void swap(int &x, int &y)
{
  int tmp;

  tmp = x;
  x = y;
  y = tmp;
}

void swap(float &x, float &y)
{
  float tmp;

  tmp = x;
  x = y;
  y = tmp;
}

void swap(char s1[], char s2[])
{
  char tmp[MAXSIZE];

  strcpy(tmp, s1);
  strcpy(s1, s2);
  strcpy(s2, tmp);
}
```

[4] C dilinde aynı isimde birden fazla fonksiyon olamaz.

6.9 Varsayılan Parametreler

Fonksiyonlar tanımlanırken istenirse bazı giriş parametrelerine varsayılan değerler verilebilir. Varsayılan değer giriş parametresi listesinde ilgili değişkenden sonra atama komutunda olduğu gibi yazılır. Bu durumda, çağıran fonksiyon o parametre için bir değer göndermezse varsayılan değer kullanılır.

Örneğin bir katar içinde bir simgenin kaçıncı sırada olduğunu belirten bir fonksiyon yazalım. Normal durumda simge katarın başından başlanarak aranır ve simgeye ilk raslanılan konum belirlenir. Sözgelimi "Dennis Ritchie" katarında 'e' simgesinin sıra numarası 1'dir. Ancak bazen de bir noktadan ileriye doğru arama yapmak istenebilir. Aynı örnek üzerinde aynı simge 4. konumdan başlanarak aranırsa sonuç 13 olacaktır. Bu durumda, yazılacak fonksiyonun başlığı şöyle olur:

```
int find_char(char s[], char c, int start);
```

Üçüncü parametrenin çoğu zaman kullanılmayacağından programcıları gerek duymadıkları bir parametreyi göndermek zorunda bırakmak yerine varsayılan değer kullanmak daha esnek bir çözümdür:

```
int find_char(char s[], char c, int start = 0)
{
    int index = -1, i;

    for (i = start; s[i] != '\0'; i++) {
        if (s[i] == c) {
            index = i;
            break;
        }
    }
    return index;
}
```

Örnekte `start` parametresine 0 varsayılan değeri verilir. Böylelikle bu değişken, fonksiyon çağrılırken belirtilmezse 0 değerini, belirtilirse verilen değeri alır. İlk iki parametrenin çağrıda belirtilmesi zorunludur; yani fonksiyon iki ya da üç parametreyle çağrılabilir:

```
find_char("Dennis Ritchie", 'e')      // 1
find_char("Dennis Ritchie", 'e', 4)   // 13
```

Sorular

1. Kendisine parametre olarak gönderilen bir katarda, yine kendisine parametre olarak gönderilen bir simgenin ilk ve son pozisyonları arasında kaç simge olduğunu bularak sonucu döndüren bir fonksiyon yazın. Sözgelimi, giriş katarı

```
"the world is not enough"
```

ve giriş simgesi `'o'` ise, fonksiyon 13 değerini döndürmelidir (`"rld is not en"`).

2. Bir katardaki bir simgenin yerine -o simgenin bulunduğu her noktada- başka bir simge geçirilmek isteniyor. Örneğin "2002-04-10" katarında '-' simgesi yerine '/' simgesi konacaksa "2002/04/10" katarı elde edilecektir.

   (a) Bu işlemi gerçekleştiren bir fonksiyon yazın.

   (b) Verilen örnek tarih üzerinde bu işlemi gerçekleştirmek üzere (a) şıkkında yazdığınız fonksiyonu kullanan bir ana fonksiyon yazın.

3. İki tamsayı dizisindeki ortak elemanların sayısı bulunmak isteniyor. Örneğin birinci dizi "21 10 9 13 15", ikinci dizi "10 7 1 13 15 8" ise ortak elemanların sayısı 3'tür (10, 13, 15). Örnekten de görülebileceği gibi, dizilerin aynı sayıda elemanları bulunması zorunlu değildir. Bunun için:

   (a) Bir sayının bir dizide bulunup bulunmadığını sınayan bir fonksiyon yazın. Fonksiyonun giriş parametreleri dizi, dizinin boyu ve aranan sayı olmalıdır. Geriye sayı dizide varsa 1, yoksa 0 değeri döndürülmelidir.

   (b) Yukarıda yazdığınız fonksiyonu kullanarak, iki dizideki ortak elemanların sayısını belirleyen bir fonksiyon yazın. Fonksiyonun giriş parametreleri her iki dizinin kendileri ve boyları olmalıdır. Fonksiyon geriye ortak elemanların sayısını döndürmelidir.

   (c) Yukarıda yazdığınız fonksiyonları kullanarak, boyunu ve elemanlarını kullanıcıdan aldığı iki dizinin ortak eleman sayısını bularak ekrana çıkartan bir ana fonksiyon (main) yazın.

4. Bir dizinin kipi, dizide en çok yinelenen elemandır. Sözgelimi

   ```
   75 32 45 43 75 66 43 88 66 92 66 27
   ```

   dizisinin kipi 66'dır. Buna göre, bir sınavdaki öğrenci notlarının kipi bulunmak isteniyor.

   (a) Bir dizinin en büyük elemanının dizideki sırasını döndüren bir fonksiyon yazın.

   (b) Yukarıda yazdığınız fonksiyonu kullanarak bir dizinin kipini döndüren bir fonksiyon yazın. (Yol gösterme: Elemanları ilgili notun kaç kere geçtiğini gösteren 101 elemanlı bir tamsayı dizisi kullanın. Örneğin `counts[55]`, kaç öğrencinin 55 aldığını göstersin.)

   (c) Yukarıda yazdığınız fonksiyonları kullanarak, öğrenci sayısını ve notlarını kullanıcıdan alarak notların kipini bulan ve ekrana çıkartan bir ana fonksiyon (main) yazın.

5. Polar sistemde düzlemde bir nokta kutuptan olan uzaklığını belirten r ve kutup eksenine yaptığı θ değerleriyle belirtilir. Aynı noktanın kartezyen sistemdeki koordinatları x ve y ise $x = r \cdot cos\theta$ ve $y = r \cdot sin\theta$ eşitlikleri geçerlidir.

   (a) Noktanın polar ve kartezyen koordinatlarını temsil etmek üzere birer kayıt tanımı yazın.

(b) Yukarıda yazdığınız kayıt tanımlarını kullanarak, parametre olarak bir noktanın polar koordinatlarını alan ve geriye kartezyen koordinatlarını döndüren bir fonksiyon yazın.

(c) Kayıt yapıları kullanmadan polar koordinatları kartezyen koordinatlara çevirecek bir fonksiyon yazın.

(d) (b) ve (c) şıklarında yazdığınız fonksiyonların kullanımlarına birer örnek verin.

Bölüm 7

İşaretçiler

Şu ana kadar yapılan örneklerde skalar, bileşke ya da vektörel tipten olsun, bütün değişkenlerin bellekte kaplayacakları alan baştan belliydi. Sözgelimi, 100 elemanlı bir tamsayı dizisi tanımlanırsa bu diziye bellekte 100 adet tamsayıyı tutacak kadar yer ayrılacağı derleme aşamasında biliniyordu. Bu tip değişkenlere *statik değişken* adı verilir. Statik bir değişkenin saklanacağı bellek alanı program çalışmaya başladığında ayrılır ve programın sonuna kadar bırakılmaz. Bu durumun bazı sakıncaları vardır:

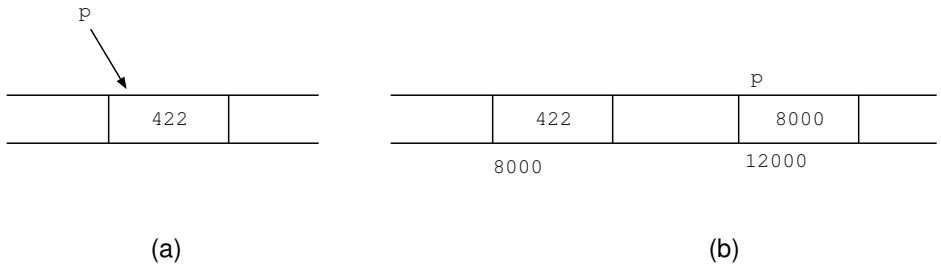
- Statik dizilerde görüldüğü gibi (bkz. Bölüm 5), dizinin eleman sayısı derleme aşamasında belli değilse gerekebilecek en büyük miktarda yer ayrılmak zorunda kalınır. Örneğin, değişik sınıflardaki öğrencilerin notlarını tutmak üzere bir tamsayı dizisi tanımlanacak olsun. Bu durumda bir sınıftaki maksimum öğrenci sayısı konusunda bir varsayım yapıp (diyelim 100) dizi bu boyutta açılmalıdır. Verilen bu boyut hem programın bir sınırlaması olacak, hem de öğrenci sayısı bunun altında kaldığı zamanlarda gereksiz bellek harcanmasına yol açacaktır.
- Programınızda kullanmak istediğiniz değişkenlerin kaplayacağı toplam bellek alanı çalıştığınız bilgisayarda bulunmayabilir. Diğer yandan, bu değişkenlerin hepsine birden aynı anda gereksinim duymuyor olabilirsiniz, yani bir değişken için ayrılan yerin programın bütün işleyişi boyunca tutulması gerekli olmayabilir. Sözgelimi bir dizi belli bir noktada kullanılmaya başlıyor ve bir noktadan sonra da kullanılmıyor olabilir. Böyle bir durumda diziye gerekli olduğu zaman yer ayırmak, işi bittikten sonra da ayrılan yeri geri vermek programın toplam bellek gereksinimlerini azaltır.

İşaretçiler, bellekte kaplanacak yerin derleme sırasında değil çalışma sırasında belirlenmesini sağlarlar. Böylelikle gerektiği zaman gerektiği kadar yer almak ve gerek kalmadığı zaman da geri vermek olanaklı hale gelir. Bir şekilde kullanılan değişkenlere *dinamik değişken* adı verilir. Dinamik değişkenlerin zorluğu, bellek alanlarının yönetimi programcıya bırakıldığından programların en sık hata yapılan bölümleri olmalarıdır.

İşaretçi, bir bellek gözüne "işaret eden" bir değişkendir. Bunun anlamı, işaretçi değişkeninin o bellek gözünün adresini tutmasıdır. Başka bir deyişle, işaretçi değişkeninin değeri, bellek gözünün adresidir. Dolayısıyla, bir işaretçi için iki değerden söz edilebilir:

- kendi değeri: işaret edilen bellek gözünün adresi
- işaret ettiği değer: işaret edilen bellek gözünün içeriği

Şekil 7.1a'da işaretçi değişkeniyle işaret edilen bellek gözü arasındaki ilişki simgesel olarak gösterilmiştir. Burada `p` değişkeni, içinde `422` yazan bir bellek gözüne işaret etmektedir. Şekil 7.1b, aynı durumun örnek adres değerleriyle nasıl sağlandığını gösterir. `422` değerinin yazılı olduğu bellek gözünün adresinin `8000` olduğu varsayımıyla, `p` değişkeninin `8000` değerini taşıdığı görülür. Adres değerlerinin (bu örnekteki `8000` sayısı) ne oldukları programcıyı ilgilendirmez, bunları işletim sistemi belirler. Programcı adres değerlerinin ne olacağı konusunda bir varsayımda bulunamaz.



Şekil 7.1: İşaretçi tipinden değişkenler.

İşaretçinin işaret ettiği bellek gözünün içeriğinin okunması için `*` işleci kullanılır. Şekildeki örnekte `p` deyiminin değeri `8000`, `*p` deyiminin değeriyse `422`'dir. İşaretçilerle işlem yaparken, işaretçinin kendisiyle mi, yoksa işaret ettiği alanla mı işlem yapıldığına dikkat edilmelidir. Örneğin `p++` komutu `p` işaretçisinin bir sonraki (diyelim `8001`) bellek gözüne işaret etmesine neden olur. Oysa `422` değerinin bir artırılması isteniyorsa `(*p)++` komutu kullanılmalıdır.

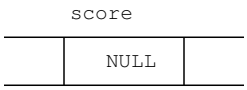
İşaretçilerle ilgili işlemlerde kullanılan diğer bir işleç de `&` işlecidir. *Adres işleci* adı verilen bu işleç, bir değişkenin adresinin öğrenilmesini sağlar. Örnekte `p` değişkeninin adresi 12000 olduğundan `&p` deyiminin değeri 12000'dir. Örnekteki değerleri toparlarsak:

- `&p`: 12000
- `p`: 8000
- `*p`: 422

İşaretçi değişkenleri için `NULL` adında özel bir değer tanımlanır. Bu değerin anlamı, o işaretçinin geçerli bir bellek gözüne işaret etmiyor olduğudur. Dolayısıyla, değeri `NULL` olan bir işaretçi değişkeninin `*` işleciyle içeriğinin okunmaya çalışılması derleyicinin farkedemeyeceği ama çalışma anında bellek hatasına yol açacak bir programlama hatasıdır.

Örnek 27. Dinamik Diziler

Bu bölümdeki program, Örnek 13'de yazılan istatistik programının aynısıdır. Dolayısıyla ekran çıktısı Şekil 5.1'de verilenle aynıdır. Programın işleyişindeki tek fark, statik diziler yerine di-



Şekil 7.2: İşaretçi tipinden değişken tanımlama.

namik diziler kullanılmasıdır. Böylece sınıftaki öğrenci sayısı kullanıcıdan öğrenildikten sonra, tam gerektiği kadar eleman tutacak bir dizi tanımlanabilmiştir.

7.1 İşaretçi Tipinden Değişkenler

İşaretçiler, değerleri birer adres (bir tür tamsayı) olan değişkenlerdir, diğer değişken tiplerinden bir farkları yoktur. Dolayısıyla, diğer değişkenlerde olduğu gibi, işaretçi tipinden bir değişken tanımlandığında bellekte bir adres tutmaya yetecek kadar yer ayrılır. İşaretçinin kendi değeri her zaman bir adrestir ama işaret ettiği bellek gözünün nasıl yorumlayacağının belirtilmesi gerekir. Bu nedenle, işaretçi tanımı şu şekilde yazılır:

veri_tipi * *değişken_adı* ;

Bu tanımın anlamı, adı verilen değişkenin bir işaretçi olduğu ve gösterdiği bellek gözünde belirtilen tipten bir değer bulunacağıdır. Buradaki * işareti, işaretçinin gösterdiği bellek gözünün içeriği anlamına gelen * işleciyle karıştırılmamalıdır. Örnekteki

```
int *score = NULL;
```

tanımı, `score` değişkeninin tamsayı barındıran bir bellek gözüne işaretçi olduğunu belirtir (Şekil 7.2). İşaretçilere başlangıç değeri olarak genellikle `NULL` atanır.

7.2 Bellek Yönetimi

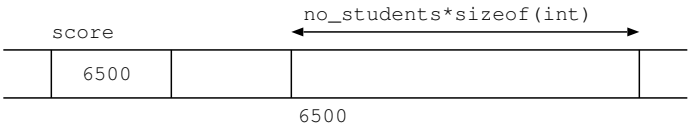
Programda kullanılacak her türlü bellek bölgesinin kullanılacağı iş için ayrılması gerektiği görülmüştü. Yani işaretçinin işaret edeceği bellek alanının da ayrılması gerekir. Ayrılacak bu alan tek bir eleman boyunda olabileceği gibi, birden fazla elemandan oluşan bir dizi olarak da kullanılabilir. Dinamik dizi kullanırken işaretçi tanımında belirtilen veri tipi dizinin her bir elemanının tipi olarak düşünülebilir.

Yer ayırma işlemini gerçekleştiren `new` işleci istenen büyüklükte, birbirini izleyen gözlerden oluşan bir bellek alanını ayırarak başlangıç adresini verir. Yer alma girişimi başarısızlıkla sonuçlanırsa, örneğin bellekte yer kalmadıysa, geriye NULL değerini döndürür. Yazımı şu şekildedir:

`new` *veri_tipi* [*eleman_sayısı*]

Örnek 27 Örnek 13'in dinamik dizilerle gerçeklenmesi.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS
#include <math.h>               // fabs,sqrt

using namespace std;

int main(void)
{
    int *score = NULL;
    int no_students = 0;
    float mean, variance, std_dev, abs_dev;
    float total = 0.0, sqr_total = 0.0, abs_total = 0.0;
    int i = 0;

    cout << "Kaç öğrenci var? ";
    cin >> no_students;
    score = new int[no_students];
    for (i = 0; i < no_students; i++) {
        cout << i + 1 << ". öğrencinin notu: ";
        cin >> score[i];
        total = total + score[i];
    }
    mean = total / no_students;
    for (i = 0; i < no_students; i++) {
        sqr_total = sqr_total + (score[i] - mean) * (score[i] - mean);
        abs_total = abs_total + fabs(score[i] - mean);
    }
    variance = sqr_total / (no_students - 1);
    std_dev = sqrt(variance);
    abs_dev = abs_total / no_students;
    cout << "Ortalama: " << mean << endl;
    cout << "Varyans: " << variance << endl;
    cout << "Standart sapma: " << std_dev << endl;
    cout << "Mutlak sapma: " << abs_dev << endl;
    delete score;
    return EXIT_SUCCESS;
}
```



Şekil 7.3: Yer ayrılmasından sonraki durum.

Örnekte öğrenci notlarını tutacak dizi için

```
score = new int[no_students];
```

komutuyla yer alınmıştır. Bu komut sonucunda bellekte `no_students * sizeof(int)` kadarlık bir alan ayrılır ve bu alanın başlangıç adresi `score` işaretçisine atanır (Şekil 7.3).

Tek bir elemanlık bölge ayrılacaksa eleman sayısının belirtilmesine gerek yoktur. Örneğin, Şekil 7.1'de çizilen durum şu komutlarla yaratılabilir:

```
int *p = NULL;

p = new int;     // 8000 olduğu varsayılıyor
*p = 422;
```

Ayrılan yerin geri verilmesi `delete` işleciyle gerçeklenir. Bu işleç, bir işaretçi için alınan bütün bölgeyi geri verir; bölgenin bir kısmını geri vermek gibi bir seçenek yoktur. Bu nedenle, işlece yalnızca işaretçinin adını vermek yeterlidir, geri verilecek eleman sayısı yeniden belirtilmez. Yukarıda yazılan her iki (birden fazla eleman ya da bir eleman) yer ayırma işleminin de geri vermesi benzer şekildedir:

```
delete score;
delete p;
```

DİKKAT

İşaretçi tipinden değişkenler için her zaman yer ayrılması zorunluluğu yoktur. Zorunlu olan nokta, işaretçilerin her zaman geçerli bellek gözlerine işaret etmeleridir. Bu bellek gözleri örneklerde olduğu gibi dinamik olarak ayrılmış olabilecekleri gibi, başka bir işaretçi değişkeni tarafından ayrılmış ve hatta statik olarak tanımlanmış bile olabilirler. Örneğin Şekil 7.1'de çizilen durum şu komutlarla da yaratılabilirdi:

```
int x = 422;
int *p = &x;
```

Bu durumda 422 değerini tutan bellek gözüne programın başında statik olarak yer ayrılır; `p` işaretçisi ise bu bellek gözünün adresini taşır. Önemli olan `x` ile `*p` değişkenlerinin aynı bellek gözünde bulunduklarının ve birinin değişmesiyle öbürünün de değişeceğinin gözönünde bulundurulmasıdır. Örnekte `p` değişkeni için yer ayrılmadığından buranın `delete` ile geri verilmesi de sözkonusu değildir; böyle bir deneme hataya neden olacaktır.

Bellek yönetimi için `new` ve `delete` işleçleri C++ dilinde getirilmiş yeniliklerdir. C dilinde aynı işlemleri yapmak için `malloc` ve `free` fonksiyonlarını kullanmak gerekir. Buna göre örnekteki programın ilgili satırları şu şekilde değiştirilebilir

```
score = (int *) malloc(no_students * sizeof(int));
...
free(score);
```

`malloc` fonksiyonu `new` işleciyle aynı şekilde yer ayırır. Aralarındaki yazım farkları şu şekilde açıklanabilir:

1. `new` işlecinde eleman tipi ve sayısı belirtilir, `malloc` fonksiyonunda ise ayrılacak alanın boyunu sekizli cinsinden vermek gerekir.
2. `malloc` fonksiyonu geriye `void *` tipinden bir değer döndürür (ham işaretçi). Bu değer değişkene atanırken uygun şekilde tip dönüşümü yapılması gerekir.
3. `malloc` ve `free` birer fonksiyon olduklarından parametrelerinin ayraçlar içinde yazılması gerekir. Aynı nedenle, kullanılabilmeleri için bir başlık dosyasının (`stdlib.h`) alınması gerekir.

7.3 İşaretçi - Dizi İlişkisi

Statik diziler ile dinamik diziler arasındaki tek fark oluşturulmalarındadır, elemanlara erişim her ikisinde de aynıdır. Yani şu iki tanım arasında, bellekte oluşan durum açısından bir fark yoktur:

```
int p[10];                    int *p;
...                           p = new int[10];
for (i = 0; i < 10; i++)      ...
  ...p[i]...                  for (i = 0; i < 10; i++)
...                             ...p[i]...
                              ...
                              delete p;
```

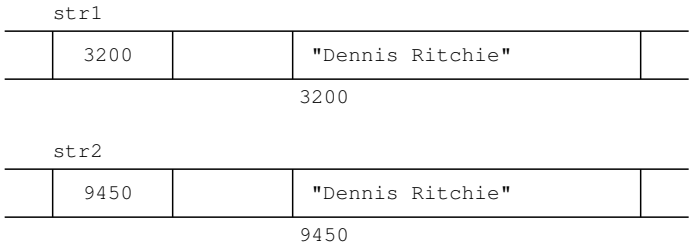
Statik dizilerle dinamik dizilerin aynı şekilde kullanılabilmeleri iki özelliğe dayanır:

1. Statik bir dizinin adı, dizinin ilk elemanına bir işaretçidir.
2. İşaretçinin değeri bir sayıyla toplanırsa, işaretçinin gösterdiği adresten işaret edilen tipten o sayı kadar ilerlenerek gelinen bellek gözünün adresi elde edilir. Benzer şekilde, çıkartma işleminde bu miktar kadar geriye gidilir.

Bu özellikler nedeniyle, işaretçinin tipi ne olursa olsun, aşağıdaki deyimler eşdeğerlidir:

```
p[0]                *p
p[1]                *(p + 1)
p[n]                *(p + n)
```

Katarlar da birer dizi olduklarından katarlar arasındaki atama ve karşılaştırma gibi işlemlerin neden beklendiği gibi çalışmayacaklarına tekrar dönelim. Şekil 7.4'deki yapıda `str1` ile `str2` değişkenleri karşılaştırılırsa (`str1 == str2`), karşılaştırma sonucu yanlış değeri üretilir (3200≠9450).



Şekil 7.4: Katarların karşılaştırılması.

Benzer şekilde, aynı örnekte `str1 = str2` ataması `str1` işaretçisinin `str2` işaretçisiyle aynı değeri alması sonucunu doğurur (Şekil 7.5). Bu durum aynı katarın iki farklı kopyasının oluşmasını sağlamadığı gibi `str1` katarının önceki işaret ettiği bellek bölgesinin ("Dennis Ritchie" değerinin bulunduğu bölge) de yitirilmesine yol açar.

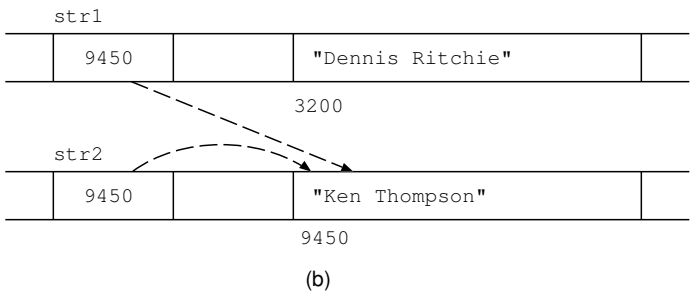
Örnek 28. Morse Kodlaması

Kullanıcıdan aldığı bir sözcüğü Morse abecesinde kodlayan bir program yazılması isteniyor. Programın örnek bir çalışmasının ekran çıktısı Şekil 7.6'da verilmiştir.

7.4 İşaretçi Tipinden Parametreler

Dizilerin fonksiyonlara parametre olarak aktarılmalarında statik ya da dinamik gösterilimler arasında bir fark yoktur. Sözgelimi, Örnek 24'de yazılan asal çarpanlarına ayırma fonksiyonunun bildirimi için şu ikisi eşdeğerlidir:

```
void factorize(int number, factor_t factors[], int &n);
void factorize(int number, factor_t *factors, int &n);
```



Şekil 7.5: Katarların atanması.

Örnek 28 Morse kodlaması yapan program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS
#include <stdio.h>              // gets
#include <string.h>             // strcat

using namespace std;

#define MAXLENGTH 80

char *encode(const char *s);

int main(void)
{
    char word[MAXLENGTH];
    char *morse = NULL;

    cout << "Sözcüğü yazınız: ";
    cin >> word;
    morse = encode(word);
    cout << "Morse karşılığı: " << morse << endl;
    delete morse;
    return EXIT_SUCCESS;
}

char *encode(const char *s)
{
    static char encoding[][5] =
        { ".-",   "-...", "-.-.", "-..",  ".",    "..-.", "--.",
          "....", "..",   ".---", "-.-",  ".-..", "--",   "-.",
          "---",  ".--.", "--.-", ".-.",  "...",  "-",    "..-",
          "...-", ".--",  "-..-", "-.--", "--.." };
    char *morse = new char[MAXLENGTH];
    int i;

    morse[0] = '\0';
    for (i = 0; s[i] != '\0'; i++) {
        strcat(morse, encoding[s[i] - 'a']);
        strcat(morse, " ");
    }
    return morse;
}
```

```
Sözcüğü yazınız: istanbul
Morse karşılığı: .. ... - .- -. -... ..- .-..
```

Şekil 7.6: Örnek 28 ekran çıktısı.

Fonksiyonun başlık satırında bu iki gösterilimden herhangi biri kullanılabilir, fonksiyonun gövdesinde bir değişiklik yapmak gerekmez.

İşaretçiler, dizilerin çıktı parametresi olarak döndürülememesi kısıtlamasını da çözerler. Statik bir dizi bir bütün halinde (bütün elemanlarının kopyası oluşturularak) çıktı parametresi olarak döndürülemez ancak dizinin başına bir işaretçi döndürülebilir. Örnekte bu özellik kullanılarak `encode` fonksiyonu

```
char *encode(const char *s);
```

şeklinde bildirilmiştir. Bunun anlamı, bu fonksiyonun değiştirilmeyecek bir katar aldığı ve ürettiği katarı geri döndürdüğüdür. Fonksiyonun gövdesinde tanımlanan `morse` değişkeni, kodlanmış sözcüğü tutar ve fonksiyon sonunda `return` ile geri döndürülür.

Burada önemli olan bir nokta, `morse` katarı için yer ayırma işini `encode` fonksiyonunun, geri verme işiniyse `main` fonksiyonunun yapmasıdır. Geri verme işlemi yine `encode` fonksiyonunca yapılamaz çünkü katarın işi henüz sona ermemiştir. Benzer şekilde `morse` değişkeni `encode` fonksiyonunda statik olarak da (`char morse[MAXLENGTH]` şeklinde) tanımlanamaz çünkü böyle tanımlandığında bu bir yerel değişken olacağından fonksiyonun sona ermesiyle onun için ayrılmış olan bellek geri verilir ve sonuç ana fonksiyona aktarılamaz.

7.5 Statik Değişkenler

Örnekte `encode` fonksiyonunda tanımlanan `encoding` dizisi bu fonksiyonun bir yerel değişkenidir, yani fonksiyonun her yaratılışında bu dizi yeniden yaratılır, elemanlarına değerler verilir ve fonksiyonun sona ermesiyle yok edilir. Bu işlemin her defasında tekrar tekrar yapılması istenmiyorsa, `encoding` değişkeni genel bir değişken olarak tanımlanabilir:

```
char encoding[][5] = { ... };

int main(void)
{
    ...
}

char *encode(char *word)
{
    ...
}
```

Genel değişken tanımlamak sözü edilen sakıncayı giderir ama `encoding` değişkeninin gereksiz yere `main` fonksiyonundan da erişilebilir hale gelmesine yol açar. Daha düzgün bir çözüm, `encoding` değişkenini `encode` fonksiyonunun içinde statik olarak tanımlanamaktır:

```
static char encoding[][5] = { ... };
```

Böyle yapıldığında `encoding` dizisi genel bir değişken gibi sürekli yaşar ama `encode` fonksiyonu dışında kullanılamaz.

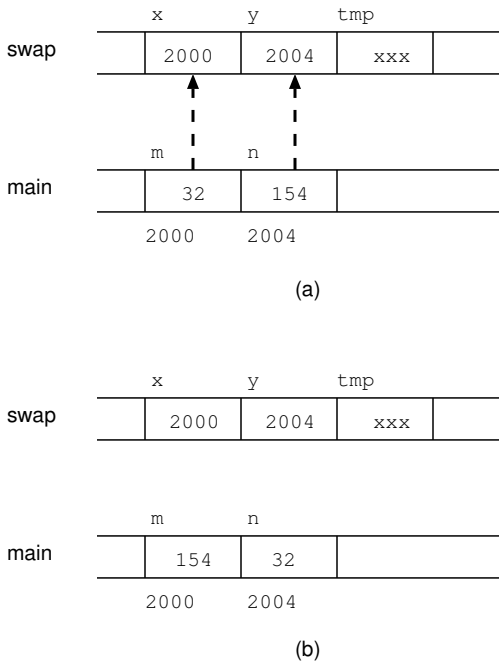
7.6 Adres Aktarımı

Başvuru aktarımı yöntemi C++ dilinde gelmiş olduğundan C dilinde bunun yerine adres aktarımı yöntemi kullanılır, yani çağırılan fonksiyona değişkenin adresi yollanır. Çağrılan fonksiyon bu adresi işaretçi tipinden bir değişkene alır ve bu işaretçinin gösterdiği yerde değişikliği yapar. Böylece değişiklik çağıran fonksiyondaki değişkeni doğrudan etkiler. Buna göre, Örnek 20'da anlatılan ve düzeltilen `swap` fonksiyonu yöntemi Örnek 29'de olduğu gibi de gerçeklenebilirdi:

Örnek 29 İki sayıyı adres aktarımıyla takas eden fonksiyon ve kullanımı.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

void swap(int *x, int *y)
{
    int tmp;

    tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void)
{
    int m = 32, n = 154;

    cout << m << " " << n << endl;
    swap(&m, &n);
    cout << m << " " << n << endl;
    return EXIT_SUCCESS;
}
```

Bu örnekte `x` değişkeni tamsayıya işaretçi (yani adres) tipinden bir değişken olurdu ve değeri `m` değişkeninin adresi olurdu. Yani `x` işaretçisinin gösterdiği yere yazılan değer `m` değişkenine yazılmış olurdu (Şekil 7.7).



Şekil 7.7: Takas fonksiyonunda parametre aktarımı.

Günümüzde neredeyse bütün C geliştirme ortamlarında C++ yetenekleri bulunduğundan, giriş parametrelerinde değişiklik yapmak istendiğinde adres aktarımı yöntemini kullanmanın artık bir gereği yoktur. Adres aktarımı programcı hatalarına daha elverişli olduğundan ancak derleyiciniz C++ desteklemiyorsa ya da probleminiz açısından daha uygunsa kullanmanız önerilir.

Uygulama: İşaretçiler

Örnek 30. Seçerek Sıralama

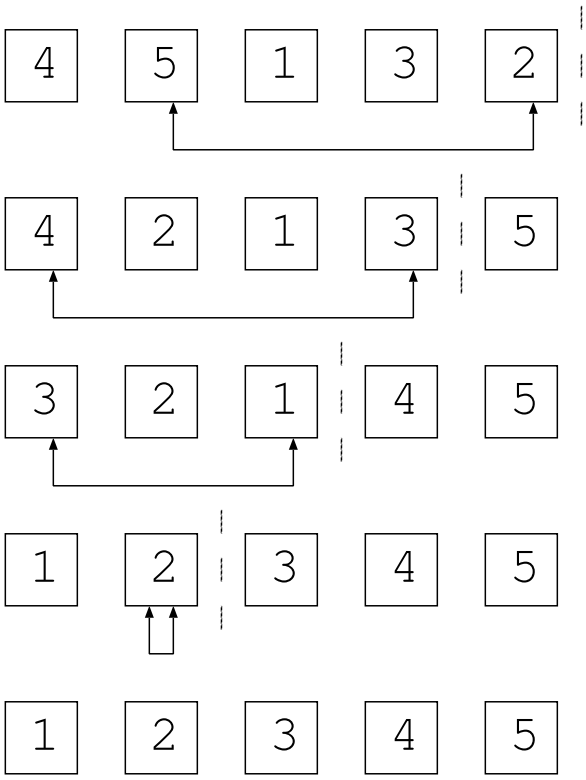
Kullanıcıdan aldığı sayı kadar öğrencisi olan bir sınıfta kullanıcının girdiği öğrenci notlarının ortadeğerini bulan bir program yazılması isteniyor. Bir dizinin ortadeğeri, dizi sıralandığında dizinin ortasında yer alan değerdir. Çift sayıda elemanı olan dizilerde dizinin ortasında bir eleman olmadığından ortadaki iki elemanın aritmetik ortalaması ortadeğer kabul edilir.

Dizinin ortadeğerini bulmak için öncelikle diziyi sıralamak gerekir. Örnekte kullanılan "seçerek sıralama" yöntemi, en basit sıralama algoritmalarından biridir. Bu yöntemde, küçükten büyüğe doğru sıralama yapılacağı varsayımıyla, bu algoritmanın her adımında dizinin en büyük elemanı bulunur ve sondaki elemanla yeri karşılıklı değiştirilir. Böylece en büyük eleman en sona alınır ve dizinin boyu bir azaltılarak işleme devam edilir. n elemanlı bir dizide sözü geçen işlem $n-1$ kere yinelenecektir. Örnek bir dizi üzerinde seçerek sıralama algoritmasının çalışması Şekil 7.8'de verilmiştir.

- Sıralama işlemini kabarcık sıralama algoritması kullanacak şekilde düzenleyin.

Örnek 30 Öğrenci notlarının ortadeğerini bulan program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

void selsort(int *numbers, int count);

int main(void)
{
    int *score = NULL;
    float median;
    int no_students, i;

    cout << "Öğrenci sayısı: ";
    cin >> no_students;
    score = new int[no_students];

    for (i = 0; i < no_students; i++) {
        cout << i + 1 << ". öğrencinin notu: ";
        cin >> score[i];
    }

    selsort(score, no_students);

    median = (no_students % 2 == 1) ?  score[no_students/2] :
                (score[no_students/2] + score[no_students/2-1]) / 2.0;
    cout << "Orta değer: " << median << endl;

    delete score;
    return EXIT_SUCCESS;
}

void selsort(int *numbers, int count)
{
    int round, max, i;
    int tmp;

    for (round = 0; round < count - 1; round++) {
        max = 0;
        for (i = 1; i < count - round; i++) {
            if (numbers[max] < numbers[i])
                max = i;
        }
        tmp = numbers[max];
        numbers[max] = numbers[count - 1 - round];
        numbers[count - 1 - round] = tmp;
    }
}
```



Şekil 7.8: Seçerek sıralama örneği.

Sorular

1. Sezar şifrelemesi yönteminde şifrelenecek sözcükteki her harfin yerine (İngilizce) abecede kendisinden 3 sonra gelen harf konur (A yerine D, B yerine E, ..., V yerine Y, W yerine Z, X yerine A, Y yerine B, Z yerine C). Buna göre "HUNGRY" sözcüğünün karşılığı "KXQJUB" olur.

   (a) Bildirimi aşağıda verildiği şekliyle bir sözcük alan ve bunun şifrelenmesi sonucu oluşan yeni katarı döndüren bir fonksiyon yazın:

   ```
   char *caesar(const char s[]);
   ```

   (b) Kullanıcıdan aldığı bir sözcüğü yukarıdaki fonksiyon yardımıyla şifreleyen ve sonucu ekrana yazan bir ana fonksiyon yazın.

   (c) (a) şıkkında yazdığınız fonksiyonu öteleme miktarı da bir parametre olacak şekilde genelleştirin ve (b) şıkkında yazdığınız fonksiyonu da uygun biçimde düzenleyin.

2. İngilizce'de 'q' harfinden sonra çoğu zaman 'u' harfi gelir. Buna göre kendisine parametre olarak gönderilen katarda 'q' harfinden sonra 'u' harfi geliyorsa 'u' harfini silen bir fonksiyon yazın. Sözgelimi, fonksiyona "you must be quick" katarı parametre olarak gelirse bu fonksiyon katarı "you must be qick" diye değiştirmelidir (yeni bir katar üretmiyor, geriye bir değer döndürmüyor). Bu fonksiyonu denemek üzere bir ana fonksiyon yazın.

Bölüm 8

Giriş-Çıkış

Bu bölümde öncelikle giriş-çıkış kitaplığındaki fonksiyonları kullanarak giriş-çıkış işlemlerinin nasıl yapıldığı gösterilecektir. Daha sonra dosyalar üzerinde okuma-yazma işlemlerinin nasıl yapıldığı anlatılacaktır.

8.1 Çıkış

C dilinde `cout` birimi yoktur. Bunun yerine `printf` fonksiyonu kullanılır. Bu fonksiyonun yapısı şu şekildedir:

```
printf(biçim katarı, deyim1, deyim2, deyim3, ...);
```

Biçim katarının temel işlevi yazdırılması istenen iletileri belirtmektir. Sözgelimi:

```
cout << "Merhaba dünya!" << endl;
```

komutunun C'deki karşılığı şöyledir:

```
printf("Merhaba dünya!\n");
```

Bu örnekte ekrana herhangi bir değişken ya da deyim değeri yazdırılmamakta, yalnızca bir ileti görüntülenmektedir. Katarın sonundaki `'\n'` simgesi katar sona erdiğinde alt satıra geçilmesini sağlar (C++'daki `endl` karşılığı).

Bir deyim değerinin ekranda gösterilmesi isteniyorsa bu değerin tipi de biçim katarında belirtilmelidir. Her veri tipinin kendine özgü bir belirteci vardır. (bkz. Tablo 8.1).

Buna göre Örnek 1'de geçen

```
cout << "Alanı: " << area << endl;
```

komutunun karşılığı şu şekilde olur:

Veri Tipi	Belirteç
Onlu düzende tamsayı	`%d`
Onlu düzende uzun tamsayı	`%ld`
Onaltılı düzende tamsayı	`%x`
Noktalı gösterilimde kesirli sayı	`%f`
Bilimsel gösterilimde kesirli sayı	`%e`
Simge	`%c`
Katar	`%s`

Tablo 8.1: Biçim belirteçleri.

```
printf("Alanı: %f\n", area);
```

Çıktının nasıl oluşturulacağını biçim katarı belirler. Belirteçler dışında kalan bölümler olduğu gibi çıkışa aktarılır; bir belirteç ile karşılaşıldığında deyim listesinde sıradaki deyim hesaplanarak elde edilen değer çıkışa aktarılır. Dolayısıyla, biçim katarında geçen belirteçler ile deyim listesindeki deyimlerin sayı ve tiplerinin birbirini tutması gerekir.

Örnek. `radius` değişkeninin kesirli sayı tipinden olduğu ve kullanıcının giriş sırasında `2.4` değerini yazdığı varsayımıyla

```
printf("Yarıçapı %f olan dairenin alanı: %f\n",
       radius, 3.14 * radius * radius);
```

fonksiyonunun işleyişi şu şekilde olur:

- Biçim katarında ilk `%` işaretine kadar görülen her simge ekrana çıkartılır (belirteçten önceki boşluk dahil): `"Yarıçapı "`
- Biçim katarından sonraki ilk deyimin değeri kesirli sayı biçiminde ekrana çıkartılır (önce ve sonraki boşluklar hariç): `"2.4"`
- Bir sonraki belirtece kadar görülen her simge ekrana çıkartılır: `" olan dairenin alanı: "`
- Biçim katarından sonraki ikinci deyimin değeri kesirli sayı biçiminde ekrana çıkartılır: `"18.07"`
- `\n` simgesi nedeniyle sonraki satıra geçilir.

Yüzde ve ters bölü işaretleri biçim katarında özel anlam taşıdıklarından bunların çıkışa gönderilmesi istendiğinde özel bir yazım gerekir. Yüzde işaretini çıkarmak için '`%%`', ters bölü işaretini çıkarmak içinse '\\' simgeleri kullanılmalıdır.

Biçim katarı değişken değerlerinin çıkışa gönderilmesinde ayrıntılı denetim olanağı da sağlar. Örneğin sayı değerlerinin belli bir uzunlukta olması sağlanabilir. "%5d" şeklinde belirtilen bir tamsayı değeri beş haneliyse boşluksuz, dört haneliyse bir boşluk ve sayı, üç haneliyse iki boşluk ve sayı v.b. şeklinde değerlendirilerek çıkışa gönderilir. Bu yöntem düzgün şekilde altalta gelmiş çıktılar oluşturmak için yararlıdır. Kesirli sayılarda da noktadan önce ve sonra

kaç hane bulunduğu belirtilebilir. Sözgelimi "%20.12f" belirteci sayının toplam 20 hane (nokta dahil) yer tutacağını ve bunun 12 hanesinin noktadan sonra olacağını gösterir.

DÜZELT: daha fazla ayrıntı

8.2 Giriş

Çıkış biriminde olduğu gibi, C dilinde giriş için kullanılabilecek `cin` birimi de yoktur. Bunun yerine `scanf` fonksiyonu kullanılır. Bu fonksiyonun yapısı şu şekildedir:

```
scanf(biçim katarı, &değişken1, &değişken2, &değişken3, ...);
```

Biçim katarı, `printf` fonksiyonundakine benzer bir işlev görür ve okunacak değerlerin tipinin ne olacağını belirler. Kullanılan veri tipi belirteçleri de aynıdır.

Giriş yapılırken kullanıcının yazdığı değerin alınacağı değişken `scanf` fonksiyonunda değer değiştireceğinden fonksiyona bu değişkenin adresi gönderilir (bkz. Bölüm 7.6). Bu nedenle, `scanf` fonksiyonuna gönderilen değişkenlerin adlarının başına adres işleci olan `&` simgesi konur. Örneğin

```
cin >> radius;
```

komutunun C dilindeki karşılığı şu şekildedir:

```
scanf("%d", &radius);
```

Katar tipinden olan değişkenlerde katarlar bir dizi olduklarından ve adları zaten dizinin ilk elemanına bir işaretçi olduğundan `&` simgesi kullanılmaz. Sözgelimi, kullanıcının yazdığı sözcüğü katar tipinden bir `word` değişkenine almak için aşağıdaki komut kullanılır:

```
scanf("%s", word);
```

Örnek 31. İstatistik Hesapları

Bu örnekte, Örnek 13'de yapılan öğrenci notları üzerindeki istatistik hesapları dosyalar yardımıyla gerçeklenecektir. Öğrenci notları bir dosyadan okunacak, işlem sonuçları da yine bir dosyaya yazılacaktır. Notların hangi dosyadan okunacağı ve sonuçların hangi dosyaya yazılacağı program çalıştırılırken komut satırından belirtilecek, böylelikle program çalışması sırasında kullanıcıya hiçbir şey sormayacak, ürettiği hiçbir sonucu da ekranda göstermeyecektir. Bu programın yazılı olduğu dosya `stat3.cpp` ve derleme ile bağlama sonucu oluşan çalıştırılabilir dosyanın adı `stat3` olursa program

```
stat3 notlar.txt sonuclar.txt
```

gibi bir komutla çağırılmalıdır. Burada ilk belirtilen isim (örnekte `notlar.txt`) okunacak dosyayı, ikinci isim (örnekte `sonuclar.txt`) sonuçların yazılacağı dosyayı gösterir ve herhangi birinin eksik olması durumunda program nasıl çalıştırılması gerektiğine ilişkin bir kullanım iletisi görüntüler.

Örnek 31 Dosyalar ile giriş/çıkış işlemleri yaparak öğrenci notları üzerinde istatistik hesaplar yapan program (okuma bölümü).

```
#include <iostream>             // cin,cout,cerr,endl
#include <stdio.h>              // fopen,fclose,fprintf,fscanf,feof
#include <stdlib.h>             // exit,EXIT_SUCCESS,EXIT_FAILURE
#include <math.h>               // fabs,sqrt

using namespace std;

#define MAXSTUDENTS 100

int main(int argc, char *argv[])
{
    int score[MAXSTUDENTS];
    int no_students = 0;
    float mean, variance, std_dev, abs_dev;
    float total = 0.0, sqr_total = 0.0, abs_total = 0.0;
    int i = 0;
    FILE *infile, *outfile;

    if (argc != 3) {
        cout << "Kullanım: " << argv[0]
             << " giriş_dosyası çıkış_dosyası" << endl;
        return EXIT_FAILURE;
    }

    infile = fopen(argv[1], "r");
    if (infile == NULL) {
        cerr << "Giriş dosyası açılamadı." << endl;
        exit(EXIT_FAILURE);
    }

    no_students = 0;
    while (true) {
        fscanf(infile, "%d", &score[no_students]);
        if (feof(infile))
            break;
        total = total + score[no_students];
        no_students++;
    }
    fclose(infile);

    ...

    return EXIT_SUCCESS;
}
```

Örnek 32 Dosyalar ile giriş/çıkış işlemleri yaparak öğrenci notları üzerinde istatistik hesaplar yapan program (yazma bölümü).

```
#include <iostream>             // cin,cout,cerr,endl
#include <stdio.h>              // fopen,fclose,fprintf,fscanf,feof
#include <stdlib.h>             // exit,EXIT_SUCCESS,EXIT_FAILURE
#include <math.h>               // fabs,sqrt

using namespace std;

#define MAXSTUDENTS 100

int main(int argc, char *argv[])
{
    int score[MAXSTUDENTS];
    int no_students = 0;
    float mean, variance, std_dev, abs_dev;
    float total = 0.0, sqr_total = 0.0, abs_total = 0.0;
    int i = 0;
    FILE *infile, *outfile;

    ...

    mean = total / no_students;
    for (i = 0; i < no_students; i++) {
        sqr_total = sqr_total + (score[i] - mean) * (score[i] - mean);
        abs_total = abs_total + fabs(score[i] - mean);
    }
    variance = sqr_total / (no_students - 1);
    std_dev = sqrt(variance);
    abs_dev = abs_total / no_students;

    outfile = fopen(argv[2], "w");
    if (outfile == NULL) {
        cerr << "Çıkış dosyası açılamadı." << endl;
        exit(EXIT_FAILURE);
    }

    fprintf(outfile, "Öğrenci sayısı: %d\n", no_students);
    fprintf(outfile, "Ortalama: %f\n", mean);
    fprintf(outfile, "Varyans: %f\n", variance);
    fprintf(outfile, "Standart sapma: %f\n", std_dev);
    fprintf(outfile, "Mutlak sapma: %f\n", abs_dev);

    fclose(outfile);

    return EXIT_SUCCESS;
}
```

8.3 Ana Fonksiyona Parametre Aktarma

Ana fonksiyon da diğer fonksiyonlar gibi bir fonksiyon olmakla birlikte giriş ve çıkış parametrelerinin aktarımı bakımından farklılık gösterir. Bir fonksiyonun giriş parametreleri alması ve çıkış parametresi döndürmesi, o fonksiyonun çağırılabilmesi anlamına gelir. Oysa ana fonksiyon çalışmanın başladığı fonksiyon olduğundan diğer fonksiyonlarca çağırılmaz. Ana fonksiyonu çağıran işletim sistemidir, yani ana fonksiyonun çağırılması programın işletim sistemince yürütülmeye başlanmasına karşı düşer. Bu durumda ana fonksiyon giriş parametrelerini işletim sisteminden alır, çıkış parametresini de işletim sistemine döndürür.

Ana fonksiyonun çıkış parametresinin nasıl belirtildiği şu ana kadarki bütün örneklerde görülmüştü. Bu parametre programın çalışması sonucu oluşan durumun işletim sistemine bildirilmesi anlamını taşır ve başarı durumunda

```
return EXIT_SUCCESS;
```

başarısızlık durumunda

```
return EXIT_FAILURE;
```

komutlarıyla belirtilir.

Ana fonksiyonun giriş parametreleriyse kullanıcının programı çalıştırırken belirttiği parametrelerdir. Giriş parametrelerinin okunabilmesi için ana fonksiyonun giriş parametresi listesi

```
int argc, char *argv[]
```

şeklinde verilir. Burada `argc` parametre sayısını, `argv` ise parametre dizisini gösterir. Parametre dizisinin her bir elemanı, başlıktan da görülebileceği gibi, bir katardır.

Programın adı da parametreler arasında sayıldığından parametre sayısı en az 1 olabilir. Yani `argc` değişkeni program yalnızca `stat3` komutuyla çağırılırsa 1, yukarıda verilen şekilde çağrılırsa 3 değerini alır. Parametre değerleri de `argv` dizisinin elemanlarını oluştururlar. Yine örnek üzerinden gidersek:

```
argv[0] = "stat3"
argv[1] = "notlar.txt"
argv[2] = "sonuclar.txt"
```

Örnekteki

```
if (argc != 3)
```

komutu programın doğru sayıda parametreyle çalıştırılıp çalıştırılmadığını sınamak için konmuştur. Parametre sayısının hatalı olduğu durumda programın ekrana bir kullanım iletisi basıp sonlanmasını sağlar.

Bütün giriş parametrelerinin birer katar olduğuna dikkat edilmelidir. Komut satırından verilen değerlerin sayı olarak kullanılabilmesi için uygun kitaplık fonksiyonlarıyla (tamsayılar için `atoi`, kesirli sayılar için `atof`) sayıya çevrilmeleri gerekir.

8.4 Dosyalar

Dosyalar üzerinde işlem yapmak için öncelikle dosyayı programda temsil edecek bir değişken tanımlanmalıdır. C dilinde bu değişken "dosya işaretçisi" olarak adlandırılır ve `FILE *` tipinden tanımlanır. Örnekte biri giriş dosyasını (`infile`) diğeri de çıkış dosyasını (`outfile`) temsil etmek üzere iki dosya işaretçisi tanımlanmıştır. Dosya işaretçisi sıradaki okuma ya da yazma işleminin dosya üzerinde hangi noktada yapılacağını belirler ve yapılan her işlemle ileri ya da geri doğru hareket eder.

8.4.1 Dosya Açma - Kapama

Bir dosya üzerinde işlem yapmadan önce ilk yapılması gereken dosyanın açılmasıdır. Açma işlemi bildirimi aşağıda verilmiş olan `fopen` fonksiyonu yardımıyla yapılır:

```
FILE *fopen(const char *path, const char *mode);
```

Fonksiyon başlığında görülen `path` parametresi, açılacak dosyanın sistemdeki tam adının belirtilmesini sağlar. İkinci parametre olan `mode` ise dosya üzerinde ne işlem yapılması istendiğini belirtmeye yarar. Bu parametre için verilebilecek örnek değerler şöyledir:

- "`r`": dosya yalnızca okunacak (dosya varsa sıfırlanmaz, yoksa yaratılmaz)
- "`w`": dosyaya yalnızca yazılacak (dosya varsa sıfırlanır, yoksa yaratılır)
- "`r+`": dosyada hem okuma hem yazma yapılacak (dosya varsa sıfırlanmaz, yoksa yaratılmaz)
- "`w+`": dosyada hem okuma hem yazma yapılacak (dosya sıfırlanır, yoksa yaratılır)
- "`a`": dosyanın sonuna ekleme yapılacak (dosya varsa sıfırlanmaz, yoksa yaratılır)

Fonksiyon başlığından da görülebileceği gibi bu fonksiyon geriye açtığı dosya için bir işaretçi döndürür, dosya üzerinde sonraki işlemlerde bu işaretçi kullanılacaktır.

Ekleme kipinde açma dışındaki kiplerde dosya açma işlemi dosya işaretçisini dosyanın başına konumlandırır; yani ilk okuma ya da yazma dosyanın başından yapılır.

Dosya üzerindeki işlemler bittikten sonra da dosyanın kapatılması gerekir. Bu amaçla bildirimi aşağıda verilmiş olan `fclose` fonksiyonu kullanılır:

```
int fclose(FILE *stream);
```

Bu fonksiyon parametre olarak verilen dosya işaretçisinin gösterdiği dosyayı kapatır. Başarılı olursa 0, başarısız olursa `EOF` değerini döndürür.

8.4.2 Dosyada Okuma-Yazma

Her okuma ya da yazma işlemi işaretçiyi okunulan ya da yazılan miktar kadar ilerletir; böylelikle peşpeşe okuma işlemleri dosyanın sırayla okunmasını sağlar (yazma için de benzer şekilde). Okuma-yazma işlemleri için `fscanf` ve `fprintf` fonksiyonları kullanılabilir. Bu fonksiyonların kullanımları `scanf` ve `printf` fonksiyonları ile aynıdır; tek farkları ek olarak en başa bir dosya işaretçisi parametresi almalarıdır.

Dosyadan başka birimlerde okuma yapmak da istenebilir. Örneğin bir satırın bütün halinde okunması amacıyla `gets` fonksiyonuna benzer `fgets` fonksiyonu kullanılabilir. Bu fonksiyonun bildirimi şu şekildedir:

```
char *fgets(char *s, int size, FILE *stream);
```

Bu fonksiyon `stream` parametresi ile belirtilen dosyadan en fazla `size - 1` simge okur ve okuduklarını `s` parametresi ile belirtilen katara yazar. Satır sonu ya da dosya sonuna raslarsa daha fazla okumaz. Başarısız olursa `NULL`, başarılı olursa `s` değerini döndürür. Güvenlik açısından `gets` fonksiyonu yerine bu fonksiyonun kullanılması önerilir.

Dosyadan tek bir simge okumak için `fgetc` fonksiyonu kullanılabilir. Bu fonksiyonun bildirimi şu şekildedir:

```
int fgetc(FILE *stream);
```

Bu fonksiyon `stream` parametresi ile belirlenen dosyadan `okuduğu` sıradaki simgeyi bir tamsayı olarak geri döndürür.

Dosya sonuna gelinip gelinmediğini öğrenmek amacıyla `feof` fonksiyonundan yararlanılır. Bu fonksiyon kendisine parametre olarak gönderilen dosya işaretçisinin ilgili dosyanın sonu olup olmadığını sınar ve sona gelindiyse doğru değerini döndürür.

8.5 Standart Giriş / Çıkış Birimleri

Standart giriş, çıkış ve hata birimleri de birer dosya gibi davranırlar. Standart giriş birimi `stdin` adında önceden tanımlanmış özel bir değişkende tutulur. Benzer şekilde standart çıkış için `stdout`, standart hata için de `stderr` değişkenleri tanımlanmıştır. Basit bir örnek verecek olursak

```
printf(biçim katarı, deyimler);
```

komutu

```
fprintf(stdout, biçim katarı, deyimler);
```

komutuyla aynı anlama gelir. Benzer şekilde aşağıdaki ikisi de giriş işlemleri için eşdeğerlidir:

```
scanf(biçim katarı, değişkenler);
fscanf(stdin, biçim katarı, değişkenler);
```

8.6 Hata İletileri

Hata iletilerinin standart çıkış iletilerinden ayrılmasının yararlı bir alışkanlık olduğu Bölüm 1.5'de söylenmişti. Bir C++ programında bu işlem iletinin `cout` değil, `cerr` birimine yönlendirilmesiyle sağlanabilir. Örnekte dosyaların açılamaması durumunda görüntülenen iletilerde bu işlem görülebilir:

```
cerr << "Giriş dosyası açılamadı." << endl;
```

C dilinde ise `cerr` birimi olmadığından aynı işlem aşağıdaki komutla gerçekleştirilmelidir:

```
fprintf(stderr, ."Giriş dosyası açılamadı.\n");
```

Hata oluştuğunda programın ne yapacağı durumdan duruma değişebilir. Düzeltilemeyecek bir hata oluştuysa programlar `exit` fonksiyonuyla sonlandırılırlar. Bu fonksiyona `EXIT_FAILURE` değeri gönderilirse programın başarısız sonlandığı işletim sistemine bildirilmiş olur. Bu işlemin `return` ile dönmeden farkı hangi fonksiyondan çağırılırsa çağırılsın programın derhal sonlanmasıdır; return ise yalnızca çağıran fonksiyona dönüşü sağlar.

Hata iletilerini standartlaştırmak amacıyla `perror` fonksiyonu tanımlanmıştır. Bu fonksiyon son oluşan hataya göre uygun bir mesajı standart hata birimine gönderir. Kullanımında gelenek olarak hatanın hangi fonksiyonda ortaya çıktığı belirtilir. Örnekte giriş dosyası açılamadığında `cerr` birimine yönlendirme yerine

```
perror("main: giriş dosyası açılamadı");
```

komutu kullanılsaydı ve programın çalıştırılması sırasında belirtilen giriş dosyası sistemde bulunamasaydı çalışma anında şöyle bir ileti görünürdü:

```
main: giriş dosyası açılamadı: No such file or directory
```

8.7 Katarlar ile Giriş-Çıkış

Standart giriş-çıkış kitaplığındaki `sprintf` ve `sscanf` fonksiyonları aynı işlemlerin katarlar ile yapılmasını sağlar. Bu fonksiyonların `printf` ve `scanf` fonksiyonlarından farkları, ilk parametrelerinin okuma ya da yazma yapılacak katarları belirtmeleridir. Örneğin dosyadan bir satırı bütün olarak okuyup, üzerinde belki bazı denetlemeler yaptıktan sonra değerlerin değişkenlere aktarılması isteniyorsa `fscanf` ile doğrudan değişkenlere aktarmak yerine aşağıdaki teknik kullanılabilir:

```
// fp dosyasından bir satırı line katarına oku
// ilk değeri x tamsayı değişkenine,
// ikinci değeri y kesirli değişkenine aktar
fgets(line, fp);
...
sscanf(line, "%d %f", &x, &y);
```

`sprintf` fonksiyonu da çıktının ekrana basılmadan bir katarda oluşturulması işleminde yararlı olur. Sözgelimi, `x` sayısını `strx` katarına çevirmek için aşağıdaki basit komut kullanılabilir:

```
sprintf(strx, "%d", x);
```

8.8 İkili Dosyalar

'b' bayrağı

fread, fwrite, fseek

Uygulama: Dosyalar

Örnek 33. Grafların Enlemesine Taranması

DÜZELT: YAZILACAK

graf tip tanımı:

```
struct graph_s {
    int nodes;
    int adjacency[MAXNODES][MAXNODES];
};
typedef struct graph_s graph_t;
```

okuma fonksiyonu bildirimi:

```
void read_matrix(FILE *fp, graph_t &g);
```

Sorular

1. Bitişiklik matrisini komut satırında belirtilen bir dosyadan okuduğu grafın bağlantı matrisini Warshall algoritması yardımıyla hesaplayan bir program yazın.

Örnek 33 Bir grafı enlemesine tarayan program (ana fonksiyon).

```
int main(int argc, char *argv[])
{
    FILE *fp;
    graph_t graph;
    int vertices[MAXNODES];
    bool visited[MAXNODES];
    int start_vertex, next_vertex;
    int count, index, i;

    if (argc != 3) {
        cerr << "Kullanım: " << argv[0]
             << " matris_dosyası başlangıç_düğümü" << endl;
        return EXIT_FAILURE;
    }

    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        cerr << "Matris dosyası açılamadı." << endl;
        exit(EXIT_FAILURE);
    }
    sscanf(argv[2], "%d", &start_vertex);

    read_matrix(fp, graph);

    for (i = 0; i < graph.nodes; i++)
        visited[i] = false;

    vertices[0] = start_vertex;
    visited[start_vertex] = true;
    count = 1;
    index = 0;
    while ((index < graph.nodes) && (count < graph.nodes)) {
        next_vertex = vertices[index];
        for (i = 0; i < graph.nodes; i++) {
            if ((graph.adjacency[next_vertex][i] == 1)
                 && (!visited[i])) {
                vertices[count] = i;
                visited[i] = true;
                count++;
            }
        }
        index++;
    }

    for (i = 0; i < graph.nodes; i++)
        cout << vertices[i] << endl;

    fclose(fp);
    return EXIT_SUCCESS;
}
```

Örnek 34 Bir grafı enlemesine tarayan program (matris okuma fonksiyonu).

```
void read_matrix(FILE *fp, graph_t &g)
{
    int c;
    int i = 0, j = 0;

    fscanf(fp, "%d\n", &g.nodes);
    while (true) {
        c = fgetc(fp);
        if (c == EOF)
            break;
        if (c == '\n') {
            i++;
            j = 0;
            continue;
        }
        g.adjacency[i][j] = c - '0';
        j++;
    }
}
```

Bölüm 9

Önişlemci

Bir C kaynak dosyasından çalıştırılabilir dosya oluşturulması için geçilen aşamalar derleme ve bağlama olarak belirtilmişti. Aslında kaynak dosyası, derleyiciye verilmeden önce bir de önişlemciden geçirilir. Önişlemcinin yaptıkları şöyle özetlenebilir:

- Açıklamaları ayıklar: `/*` ile `*/` arasında kalan ya da `//` işaretinden satır sonuna kadar olan bölümleri koddan siler; yani bu bölümler derleyiciye hiç gitmez.
- Önişlemci komutlarını işler: `#` simgesiyle başlayan komutlar önişlemci komutlarıdır ve önişlemci tarafından işlenirler.

En sık kullanılan önişlemci komutlarını yeniden görelim:

#**define** Değişmez ya da makro tanımlamakta kullanılır. Kodun içinde değişmezin adının geçtiği her yere değerini yazar. Sözgelimi `#define PI 3.14` önişlemci komutu, kodda `PI` yazan her yere `3.14` yazarak derleyiciye o haliyle gönderir; yani derleyici `PI` sözcüğünü görmez.

#**include** Belirtilen dosyayı o noktada kodun içine ekler. Sözgelimi `#include <stdlib.h>` önişlemci komutu, `stdlib.h` isimli dosyayı bularak kaynak kodun içine yerleştirir. Kod derleyiciye geldiğinde bu dosyanın içeriğini de barındırır.

9.1 Makrolar

Programın içinde sıkça yinelenmesi gerekebilecek, ancak bir fonksiyon haline getirmeye de değmeyecek küçük kod parçaları makrolar yardımıyla gerçeklenir. Makrolar da değişmez tanımlarına benzer şekilde `#define` sözcüğüyle yapılırlar. İşleyişleri de yine değişmez tanımlarına benzer şekilde olur, yani makronun adının geçtiği yere açılımı konur.

Örnek 13'de geçen

```
sqr_total = sqr_total + (score[i] - mean) * (score[i] - mean);
```

komutunu basitleştirmek üzere bir deyimin karesini alan

```
#define sqr(x) (x) * (x)
```

makrosunu kullanarak komutu şu şekle getirebiliriz:

```
sqr_total = sqr_total + sqr(score[i] - mean);
```

Bunun sonucunda makro tanımındaki `x` simgesinin yerine makronun kullandıldığı yerdeki `score[i] - mean` deyimi konur (programcı kendisi bu şekilde yazmış gibi).

Bu işlem bir sözcük ya da sözcük grubunun yerine başka bir sözcük ya da sözcük grubunun yerleştirilmesi şeklinde yürüdüğünden kullanımına dikkat etmek gerekir. Örnekteki makro

```
#define sqr(x) x * x
```

şeklinde tanımlansaydı makro açılımıyla oluşacak (hatalı) kod şu şekilde olurdu:

```
sqr_total = sqr_total + score[i] - mean * score[i] - mean;
```

Örnek 35. Projeler

Bir sayının asal çarpanlarının ekrana dökülmesini ve iki sayının en büyük ortak bölen ve en küçük ortak katlarının hesaplanması işlemlerini yapan bir program yazılması isteniyor. Programın örnek bir çalışması Şekil 9.1'de verilmiştir. Bu örnekte kaynak kodu birden fazla dosyaya bölünecek, kullanıcıyla etkileşim kısmını yürüten fonksiyon (aynı zamanda `main` fonksiyonu) `project.cpp` dosyasına (Örnek 35), işlemleri yapan fonksiyonlar `ops.cpp` dosyasına (Örnek 36) konacaklardır.[1]

9.2 Projeler

Yazılan programın kapsamı büyüdükçe bütün kaynak kodunun tek bir dosyada toplanması zorlaşmaya başlar. Binlerce satırlık bir kaynak kodunun tek bir dosyada tutularak program geliştirilmesi son derece zordur. Böyle projelerde kaynak kodu farklı dosyalara bölünür. Birden fazla kaynak dosyasına bölünmüş bir proje derlenirken önce her kaynak dosyası ayrı ayrı derlenerek ara kodlar oluşturulur, sonra bağlayıcı bu ara kodlar ve varsa kullanılan kitaplıklar arasındaki bağlantıları kurarak çalıştırılabilir kodu üretir (Şekil 9.3).

Derleme süresi bağlama süresinden çok daha uzun olduğundan kaynak kodun bu şekilde bölünmesi çalıştırılabilir dosyanın üretilmesi için gereken zamanı da azaltır. Tek bir büyük dosyadan oluşan projelerde herhangi bir yordamdaki herhangi bir değişiklikte bütün yordamların yeniden derlenmeleri ve bağlanmaları gerekir. Oysa kaynak dosyaları bölünürse yalnızca değiştirilen

[1] Bu örneğin nasıl derleneceğini Ek B.2'de görebilirsiniz.

```
Sayı 1: 0
Sayı 2: 0

1. Sayı 1'i değiştir
2. Sayı 2'yi değiştir
3. Sayı 1'in çarpanlarını göster
4. En büyük ortak bölen bul
5. En küçük ortak kat bul
6. Çık

Seçiminiz: 1
Sayıyı yazınız: 9702

Sayı 1: 9702
Sayı 2: 0

1. Sayı 1'i değiştir
2. Sayı 2'yi değiştir
3. Sayı 1'in çarpanlarını göster
4. En büyük ortak bölen bul
5. En küçük ortak kat bul
6. Çık

Seçiminiz: 2
Sayıyı yazınız: 945
```

Şekil 9.1: Proje örneği ekran çıktısı.

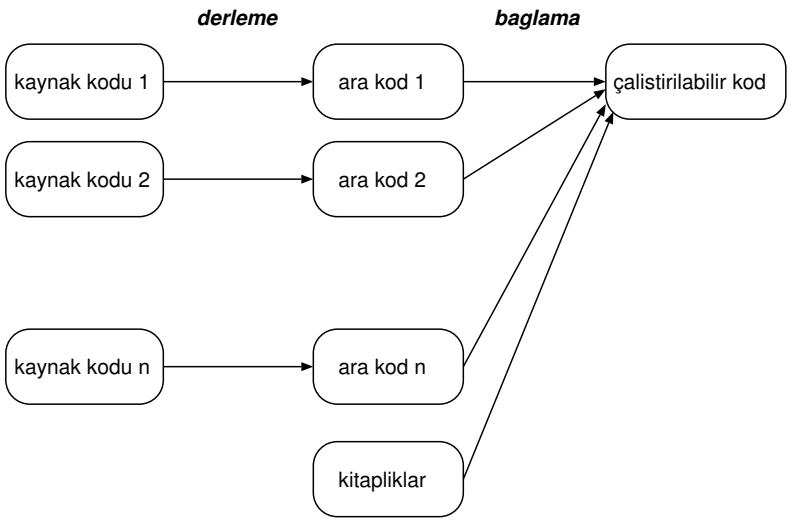
```
Sayı 1: 9702
Sayı 2: 945

1. Sayı 1'i değiştir
2. Sayı 2'yi değiştir
3. Sayı 1'in çarpanlarını göster
4. En büyük ortak bölen bul
5. En küçük ortak kat bul
6. Çık

Seçiminiz: 3
2^1 3^2 7^2 11^1

Sayı 1: 9702
Sayı 2: 945

1. Sayı 1'i değiştir
2. Sayı 2'yi değiştir
3. Sayı 1'in çarpanlarını göster
4. En büyük ortak bölen bul
5. En küçük ortak kat bul
6. Çık

Seçiminiz: 4
En büyük ortak bölen: 1323

Sayı 1: 9702
Sayı 2: 945

1. Sayı 1'i değiştir
2. Sayı 2'yi değiştir
3. Sayı 1'in çarpanlarını göster
4. En büyük ortak bölen bul
5. En küçük ortak kat bul
6. Çık

Seçiminiz: 5
En küçük ortak kat: 145530

Sayı 1: 9702
Sayı 2: 945

1. Sayı 1'i değiştir
2. Sayı 2'yi değiştir
3. Sayı 1'in çarpanlarını göster
4. En büyük ortak bölen bul
5. En küçük ortak kat bul
6. Çık

Seçiminiz: 6
```

Örnek 35 Giriş/çıkış fonksiyonlarını içeren kaynak dosyası.

```
#include <iostream>             // std::xxx
#include <stdlib.h>             // EXIT_SUCCESS
#include "ops.h"                // gcd,lcm,...

int main(void)
{
    int num1 = 0, num2 = 0;
    factor_t factors[MAXFACTOR];
    int n, i;
    int choice;

    while (true) {
        std::cout << "Sayı 1: " << num1 << std::endl;
        std::cout << "Sayı 2: " << num2 << std::endl << std::endl;
        std::cout << "1. Sayı 1'i değiştir" << std::endl;
        std::cout << "2. Sayı 2'yi değiştir" << std::endl;
        std::cout << "3. Sayı 1'in çarpanlarını göster" << std::endl;
        std::cout << "4. En büyük ortak bölen bul" << std::endl;
        std::cout << "5. En küçük ortak kat bul" << std::endl;
        std::cout << "6. Çık" << std::endl << std::endl;
        std::cout << "Seçiminiz: ";
        std::cin >> choice;
        if (choice == 6)
            exit(EXIT_SUCCESS);
        switch (choice) {
            case 1:
            case 2:
                std::cout << "Sayıyı yazınız: ";
                if (choice == 1)
                    std::cin >> num1;
                else
                    std::cin >> num2;
                break;
            case 3:
                factorize(num1, factors, n);
                for (i = 0; i < n; i++)
                    std::cout << factors[i].base << "^" << factors[i].power << " ";
                std::cout << std::endl;
                break;
            case 4:
                std::cout << "En büyük ortak bölen: " << gcd(num1, num2) << std::endl;
                break;
            case 5:
                std::cout << "En küçük ortak kat: " << lcm(num1, num2) << std::endl;
                break;
        }
        std::cout << std::endl;
    }
    return EXIT_SUCCESS;
}
```

Örnek 36 Hesap fonksiyonlarını içeren kaynak dosyası.

```
#include <math.h>                       // sqrt,pow
#include "ops.h"                        // struct factor_s

#define max(x, y) (x) > (y) ? (x) : (y)
#define min(x, y) (x) > (y) ? (x) : (y)

void gcd_factors(const factor_t factors1[], int n1,
                 const factor_t factors2[], int n2,
                 factor_t factors[], int &n);
void lcm_factors(const factor_t factors1[], int n1,
                 const factor_t factors2[], int n2,
                 factor_t factors[], int &n);

int gcd(int number1, int number2)
{
    ...
}

int lcm(int number1, int number2)
{
    ...
}

bool is_prime(int cand)
{
    ...
}

int next_prime(int prime)
{
    ...
}

void factorize(int x, factor_t factors[], int &n)
{
    ...
}

void gcd_factors(const factor_t factors1[], int n1,
                 const factor_t factors2[], int n2,
                 factor_t factors[], int &n)
{
    ...
}

void lcm_factors(const factor_t factors1[], int n1,
                 const factor_t factors2[], int n2,
                 factor_t factors[], int &n)
{
```



Şekil 9.3: Birden fazla kaynak kodlu projelerin derleme aşamaları.

yordamın bulunduğu kaynak dosyası yeniden derlenir ve bağlama işlemi yapılır; değişmeyen kaynak kodlarının yeniden derlenmelerine gerek kalmaz.

Böyle bir çalışmada, dosyalardan yalnızca birinde `main` fonksiyonu bulunabileceği açıktır; aksi durumda bağlama işlemi belirsizlik nedeniyle başarısız olur. Ayrıca farklı dosyalardaki fonksiyonların birbirlerini çağırabilmeleri, dosyalar arasında değişken paylaşabilmeleri gibi konular için bazı düzenlemeler yapmak gerekir.

Dağıtmanın yararlı olabilmesi için fonksiyonlar, amaçlarına göre gruplanarak dosyalara bölünmelidir. Örnekte olduğu gibi, kullanıcıyla etkileşimi sağlayan (giriş/çıkış işlemlerini yapan) fonksiyonların hesaplamaları yapan fonksiyonlarla ayrı dosyalara toplanması sık kullanılan bir bölümleme tekniğidir.

Hesap işlemlerini yapan `factorize`, `gcd`, `lcm` fonksiyonları `project.cpp` dosyasında bulunmadıklarından bu dosyanın derlenmesi sırasında sorun çıkar. Derlenebilmesi için bu üç fonksiyonun bildirimleri dosya başına eklenmelidir. Bildirimler elle yazılabilir ancak daha doğru olan yöntem, `ops.cpp` dosyasını bir kitaplık gibi düşünüp tanımladığı fonksiyonların bildirimlerini içeren bir başlık dosyası hazırlamak ve bu başlık dosyasını diğer dosyanın içine almaktır. Böylelikle `ops.cpp` ve `ops.h` dosyaları başka projelerde de kullanılabilirler. Örnek projede `ops.cpp` dosyası için `ops.h` başlık dosyası hazırlanmış (Örnek 37) ve bu dosya

```
#include “ops.h”
```

önişlemci komutuyla her iki kaynak dosyasının da içine alınmıştır. Burada <> simgeleri yerine “” simgeleri kullanılması, önişlemcinin başlık dosyasını sistem klasörlerinden önce bulunulan klasörde aramasını sağlar.

Başlık dosyası, ilgili kitaplığın arayüzüdür; başka dosyalardaki fonksiyonların gerek duyabilecekleri bilgileri barındırır. Bir fonksiyon yalnızca o dosya içinde kullanılıyorsa ve dışarıdan

Örnek 37 Hesap fonksiyonları için başlık dosyası.

```
#ifndef OPS_H
#define OPS_H

#define MAXFACTOR 50

struct factor_s {
    int base, power;
};
typedef struct factor_s factor_t;

void factorize(int x, factor_t factors[], int &n);
int gcd(int number1, int number2);
int lcm(int number1, int number2);

#endif
```

fonksiyonlar tarafından çağrılmayacaksa (örnekteki `next_prime` ve `is_prime` fonksiyonları gibi) bildirimi başlık dosyasına yazılmaz.

Başlık dosyaları, fonksiyon bildirimlerinin dışında, tip tanımları da içerebilirler. Örnekte çarpanları göstermek için kullanılan `factor_t` tipi `project.cpp` dosyasında da gerek duyulan bir tip olduğundan başlık dosyasına alınmıştır. Program bir sayının asal çarpanlarını listeleme işini yapmayacak olsaydı, bu veri tipi ve `factorize` fonksiyonunun bildirimine `project.cpp` dosyasında gerek kalmayacağı için başlık dosyasına yazılmayabilirlerdi. Bu veri tipi `ops.cpp` dosyasında da kullanıldığından bu dosyanın da aynı başlık dosyasını içermesi gerekir.

Benzer şekilde, değişmez ve makro tanımları da başlık dosyalarında yer alabilir. Örnekte ana fonksiyon bir sayının asal çarpanlarını temsil etmek üzere statik bir dizi tanımlamaktadır. Bu dizinin eleman sayısına kendisi karar verebileceği gibi, diğer fonksiyonlarla uyum açısından bu bilgiyi başlık dosyasından alması daha uygundur.

Başlık dosyalarında yapılmaması gereken iki önemli işlem vardır:

- Fonksiyon tanımlamak: Başlık dosyalarına fonksiyonların yalnızca bildirimleri yazılır, gövdeleri yazılmaz. Bir başlık dosyası birden fazla C dosyası tarafından alınabileceğinden aynı fonksiyonun birden fazla kere tanımlanması bağlama aşamasında hataya neden olur.

- Değişken tanımlamak: Benzer şekilde, başlık dosyasında tanımlanan değişkenler birden fazla dosya tarafından alınma durumunda aynı isimli genel değişkenler olarak değerlendirilir ve bağlama hatasına neden olurlar.

  Farklı dosyalar arasında paylaşılacak genel değişken tanımlanmak isteniyorsa bu değişken başlık dosyasında `extern` saklı sözcüğüyle bildirilmeli ve C kaynak dosyalarından birinde normal biçimde tanımlanmalıdır. Örneğin, başlık dosyasında `extern int counter;` şeklinde bildirilebilir ve dosyalardan biri `int counter;` komutuyla tanımlayabilir.

Sorular

1. $x \leftarrow |y - z|$ işlemini gerçekleştirmek üzere

   (a) bir `if` komutu yazın.

   (b) bir makro tanımı yazın.

2. Üç sayının harmonik ortalaması şöyle tanımlanır:

$$\frac{3}{\frac{1}{x_1} + \frac{1}{x_2} + \frac{1}{x_3}}$$

   (a) Üç sayının harmonik ortalamasını hesaplamak üzere kullanılabilecek harmonic(x, y, z) makrosunun tanımını yazınız.

   (b) Yazdığınız makro

```
b = harmonic(m - 1, n + 3, p);
```

   şeklinde çağrılırsa bu makronun açılımı nasıl olur?

Bölüm 10

Bağlantılı Listeler

Örnek 38. En Büyük Ortak Bölen Bulma

Örnek 23'de gerçeklendiği şekliyle, en büyük ortak bölen bulma programında bir sayının çarpanları statik bir diziyle temsil ediliyordu. Bu örnekte statik dizi yerine dinamik dizi kullanmak bellek kullanımını etkinleştirmek adına bir işe yaramaz; çünkü dinamik dizilerde yapılan, eleman sayısı belli olduktan sonra gerektiği kadar yer ayırmaktır, oysa bu örnekte eleman sayısı ancak çarpanlarına ayırma algoritmasının sona ermesiyle sonra belli olur.

Bağlantılı listeler, bu tip algoritmalarda dizilere göre daha uygun bir veri yapısıdır. Bir bağlantılı liste, birbirinin eşi düğümlerden oluşur; her düğüm tutulmak istenen bilgileri taşımasının yanısıra listede kendisinden sonra gelen düğüme de bir işaretçi içerir. Böylece listenin başı biliniyorsa sonraki düğümü gösteren alanlar üzerinden ilerlenerek bütün düğümlere erişilebilir. Listenin sonunu belirtmek üzere son düğümün sonraki alanına özel bir değer (`NULL`) yazılır. Bu yapıda, dizilerin aksine, eleman sayısını tutmanın gereği yoktur.

Örnek 38 aynı algoritmaları statik diziler yerine bağlantılı listeler üzerinde gerçekleyen bir programdır.

Örnekteki bağlantılı liste yapısının oluşturulması için kullanılan yapı tanımı şöyledir:

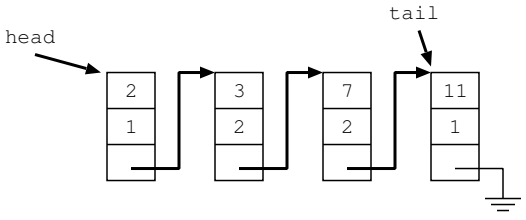
```
struct factor_s {
    int base, power;
    struct factor_s *next;
};
typedef struct factor_s factor_t;
```

Görüldüğü gibi, bir düğümde asal çarpanın değeri ve üssüne ek olarak düğümün kendi tipine işaret eden bir işaretçi alanı vardır. Böyle bir yapı üzerinde işlem yapmak için tek gerekli değişken dizinin ilk elemanına işaret eden bir değişken tutmaktır, örnekteki `head` değişkenleri bu amaçla tanımlanmıştır. Listeye ekleme algoritmalarında kolaylık sağlaması için bir de listenin son düğümünü (`next` alanında `NULL` yazan düğüm) gösteren bir de `tail` değişkeni kullanılmıştır.

Bu tanıma göre oluşturulan bağlantılı listelerde 9702 sayısının çarpanlarının nasıl gösterildiği Şekil 10.1'de verilmiştir.

Örnek 38 Bağlantılı listeler kullanarak en büyük ortak bölen hesaplayan program (ana fonksiyon).

```
int main(void)
{
    int number1, number2;
    factor_t *factors1 = NULL, *factors2 = NULL, *factors3 = NULL;
    long int gcd = 1L;
    factor_t *f = NULL;

    cout << "Sayıları yazınız: ";
    cin >> number1 >> number2;
    factors1 = factorize(number1);
    factors2 = factorize(number2);
    factors3 = gcd_factors(factors1, factors2);
    for (f = factors3; f != NULL; f = f->next)
        gcd = gcd * (long int) pow((double) f->base,
                                   (double) f->power);
    delete_factors(factors1);
    delete_factors(factors2);
    delete_factors(factors3);
    cout << "En büyük ortak bölen: " << gcd << endl;
    return EXIT_SUCCESS;
}
```



Şekil 10.1: Bağlantılı liste örneği.

Bu veri yapısı kullanıldığında bir sayıyı asal çarpanlarına ayıran fonksiyonun giriş parametresi olarak yalnızca asal çarpanlarına ayrılacak sayıyı alması ve çıkış parametresi olarak oluşturduğu çarpanlar listesinin başlangıç elemanını döndürmesi yeterlidir:

```
factor_t *factorize(int x);
```

Benzer şekilde, ortak çarpanları bulma algoritması da iki çarpan listesini alarak ortak çarpanlar listesini döndürür:

```
factor_t *gcd_factors(factor_t *factors1, factor_t *factors2);
```

Bağlantılı listeler dinamik olarak oluşturulduklarından işleri bittiğinde sisteme geri verilmeleri gerekir. Bu amaçla tanımlanan `delete_factors` fonksiyonu ilk düğümüne işaretçi aldığı bir listenin bütün düğümlerini sisteme geri verir. Burada `head` işaretçisi o anda geri verilecek düğümün adresini tutarken p işaretçisi bir sonraki düğümün unutulmamasını sağlar. Bu fonksiyon şöyle yazılabilir:

```
void delete_factors(factor_t *head)
{
    factor_t *p = NULL;

    while (head != NULL) {
        p = head->next;
        delete head;
        head = p;
    }
}
```

Örnek programda kullanılan diğer fonksiyonlar olan `factorize` ve `gcd_factors` fonksiyonları sırasıyla Örnek 39 ve Örnek 40'de verilmiştir.

10.1 Yapılara İşaretçiler

Bir yapıya işaret eden bir değişken tanımlandığında bu yapının alanlarına erişmek için `->` işleci kullanılır. Örnekte ortak çarpanlardan en büyük ortak bölenin hesaplandığı döngüdeki `f->base` yazımı buna bir örnektir. Bunun yerine önce `*` ile işaretçinin başvurduğu yere erişilip daha sonra noktalı gösterilimle istenen alanın değeri de alınabilir: `(*f).base` gibi. Ancak bu ikinci gösterilim pek yeğlenmez.

Uygulama: Bağlantılı Listeler

Bu örnekte, Örnek 30'de yapılan ortadeğer bulma programı seçerek sıralama algoritması yerine araya sokarak sıralama algoritmasıyla gerçeklenecektir.

Örnek 39 Bağlantılı listeler kullanarak en büyük ortak bölen hesaplayan program (asal çarpanlara ayırma fonksiyonu).

```
factor_t *factorize(int x)
{
    factor_t *head = NULL, *tail = NULL, *f = NULL;
    int factor = 2;

    while (x > 1) {
        if (x % factor == 0) {
            f = new factor_t;
            f->base = factor;
            f->power = 0;
            while (x % factor == 0) {
                f->power++;
                x = x / factor;
            }
            f->next = NULL;
            if (head == NULL)
                head = f;
            if (tail != NULL)
                tail->next = f;
            tail = f;
        }
        factor = next_prime(factor);
    }
    return head;
}
```

Örnek 40 Bağlantılı listeler kullanarak en büyük ortak bölen hesaplayan program (ortak çarpanları bulma fonksiyonu).

```
factor_t *gcd_factors(factor_t *factors1, factor_t *factors2)
{
    factor_t *factors = NULL, *head = NULL, *tail = NULL, *p = NULL;

    while ((factors1 != NULL) && (factors2 != NULL)) {
        if (factors1->base < factors2->base)
            factors1 = factors1->next;
        else if (factors1->base > factors2->base)
            factors2 = factors2->next;
        else {
            factors = new factor_t;
            factors->base = factors1->base;
            factors->power = min(factors1->power, factors2->power);
            factors->next = NULL;
            if (head == NULL)
                head = factors;
            if (tail != NULL)
                tail->next = factors;
            tail = factors;
            factors1 = factors1->next;
            factors2 = factors2->next;
        }
    }
    return head;
}
```

Örnek 41. Araya Sokarak Sıralama

Araya sokarak sıralama algoritması, temel sıralama yöntemlerinden biridir. Bu yöntemde, her yeni gelen eleman o ana kadar gelen elemanlara göre sıralı olacak şekilde yerine yerleştirilir. Örneğin 45, 22, 91, 18, 62 sayıları sıralanacaksa şu şekilde ilerlenir:

```
45
22 45
22 45 91
18 22 45 91
18 22 45 62 91
```

Bu yöntem statik diziler üzerinde gerçeklenmeye uygun değildir çünkü her yeni gelen eleman eklendiğinde ondan büyük olan bütün elemanların bir konum sağa kaydırılmaları gerekir. Örnekte 18 sayısı dizinin en başına ekleneceğinden üç elemanın birden sağa kaydırılmasına neden olur. Oysa bu algoritma bağlantılı listeler üzerinde gerçeklenmeye gayet uygundur. Örnek 41'de bir diziyi sıralı bir bağlantılı listeye çeviren program verilmiştir.

- Ekleme işlemini yapan fonksiyonu şu durumlar için inceleyin:
  - liste boş
  - eleman en başa ekleniyor
  - eleman en sona ekleniyor
  - eleman arada bir yere ekleniyor
- assert
- core dump

Örnek 41 Bağlantılı liste üzerinde araya sokarak sıralama programı (ana fonksiyon).

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

struct node_s {
    int value;
    struct node_s *next;
};
typedef struct node_s node_t;

node_t *insertsort(int *numbers, int count);
void delete_nodes(node_t *head);

int main(void)
{
    int *score = NULL;
    node_t *head = NULL, *f = NULL;
    float median;
    int no_students, i;

    cout << "Öğrenci sayısı: ";
    cin >> no_students;
    score = new int[no_students];

    for (i = 0; i < no_students; i++) {
        cout << i + 1 << ". öğrencinin notu: ";
        cin >> score[i];
    }

    head = insertsort(score, no_students);

    f = head;
    for (i = 0; i < no_students / 2 - 1; i++)
        f = f->next;
    median = (no_students % 2 == 1) ? f->next->value :
                (f->value + f->next->value) / 2.0;
    cout << "Orta değer: " << median << endl;

    delete_nodes(head);

    delete score;
    return EXIT_SUCCESS;
}
```

Örnek 42 Bağlantılı liste üzerinde araya sokarak sıralama programı (liste fonksiyonları).

```
node_t *insert(node_t *head, int v)
{
    node_t *p = head, *newnode = NULL, *last = NULL;

    newnode = new node_t;
    newnode->value = v;
    while ((p != NULL) && (p->value < v)) {
        last = p;
        p = p->next;
    }
    newnode->next = p;
    if (last == NULL)
        return newnode;
    last->next = newnode;
    return head;
}

node_t *insertsort(int *numbers, int count)
{
    node_t *head = NULL;
    int i;

    for (i = 0; i < count; i++)
        head = insert(head, numbers[i]);
    return head;
}

void delete_nodes(node_t *head)
{
    node_t *p = NULL;

    while (head != NULL) {
        p = head->next;
        delete head;
        head = p;
    }
}
```

Bölüm 11

Rekürsiyon

İki sayının en büyük ortak bölenini bulmak üzere kullandığımız Euclides algoritması, "`a` ile `b` sayılarının en büyük ortak böleni `b` ile `a % b` sayılarının en büyük ortak bölenine eşittir" ilkesine dayanıyordu. Problemin çözümünün, bu örnekte olduğu gibi, kendisi cinsinden ifade edilmesine *rekürsif* tanım adı verilir. Çözülmesi istenen problem, kendisi cinsinden daha küçük bir probleme indirgenir. Sürekli indirgemeler yoluyla çözümü bilinen bir duruma (taban durum) ulaşılmaya çalışılır. Euclides algoritmasında taban durum küçük olan sayının 0'a gelmesi durumuydu; bu durumda diğer sayı en büyük ortak bölen oluyordu. Aksi halde daha küçük sayılar üzerinde en büyük ortak bölen aranmaya devam ediliyordu. Bu ilkeyi gerçekleyen bir fonksiyon şu şekilde yazılabilir:

```
int gcd(int a, int b)
{
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}
```

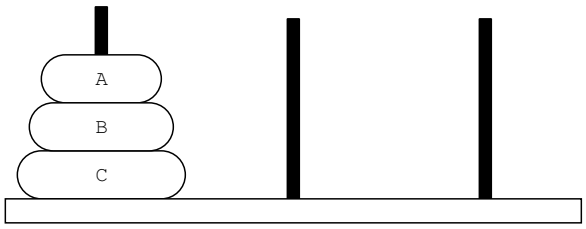
Rekürsiyona en çok verilen örneklerden biri de faktöryel hesaplanmasıdır. Rekürsif olarak faktöryel hesaplayan bir fonksiyon şöyle yazılabilir:

```
int factorial(int x)
{
    if (x == 0)
        return 1;
    else
        return x * factorial(x - 1);
}
```

Bu iki örnek, C gibi blok yapılı dillerde rekürsif gerçeklenmeye uygun örnekler değildir; çünkü yinelemeli olarak gerçeklendiklerinde daha etkin çalışırlar. Rekürsif yazım bazen daha güzel görünse ve matematikteki tanıma daha yakın olsa da, başarım açısından dezavantajlı olabilir.

Örnek 43. Hanoi Kuleleri

Hanoi kuleleri probleminde 3 adet direk ve 64 adet ortası delik disk vardır. Diskler başlangıçta birinci direğe geçirilmiş durumdadır. Çapı en geniş olan disk en altta, en dar olan en üstte yer alır ve her disk kendisinden daha geniş çaplı bir diskin üzerinde durur. Amaç, diskleri teker teker direkler arasında taşıyarak aynı düzeni üçüncü direkte oluşturmaktır. Burada kural, hiçbir diskin hiçbir aşamada kendisinden dar bir diskin üzerine konamamasıdır. Şekil 11.1'de üç direk ve üç diskli örnek verilmiştir. Bu örneğin çözümünü yapan programın çıktısı Şekil 11.2'de verilmiştir.



Şekil 11.1: Hanoi kuleleri problemi başlangıç durumu.

Hanoi kuleleri probleminin genel çözümünü oluşturmaya en geniş çaplı diskin nasıl taşınacağını düşünmekle başlayalım. Bu disk başka hiçbir diskin üstüne konamayacağı için üçüncü diske başka bir direk üzerinden aktarılarak geçemez, bir kerede götürülmelidir. Bunun yapılabilmesi için birinci diskte kendisinden başka disk bulunmamalı, üçüncü disk de boş olmalıdır (Şekil 11.3).

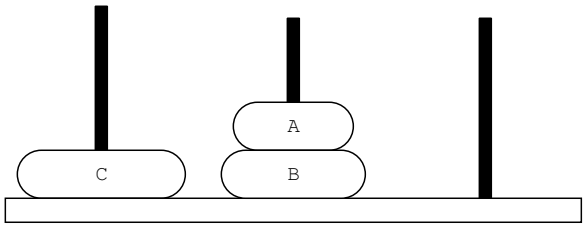
Bu durumda çözüm üç aşamalı olarak görülebilir:

1. Dar 63 diski birinci diskten ikinci diske taşı.
2. En geniş diski birinci diskten üçüncü diske taşı.
3. Dar 63 diski ikinci diskten üçüncü diske taşı.

Birinci ve üçüncü adımlardaki taşıma işlemi aslında 63 disk için aynı problemin çözülmesinden başka bir şey değildir. O halde problemimizi `n` diskin `a` direğinden `b` direğine `c` direği üzerinden taşınması şeklinde ifade ederek çözümü şu şekilde yazabiliriz:

```
Bir diski 1 direğinden 3 direğine taşı.
Bir diski 1 direğinden 2 direğine taşı.
Bir diski 3 direğinden 2 direğine taşı.
Bir diski 1 direğinden 3 direğine taşı.
Bir diski 2 direğinden 1 direğine taşı.
Bir diski 2 direğinden 3 direğine taşı.
Bir diski 1 direğinden 3 direğine taşı.
```

Şekil 11.2: Hanoi kuleleri problemini çözen programın ekran çıktısı.



Şekil 11.3: Hanoi kuleleri problemi (en geniş diskin taşınması).

1. `n - 1` diski `a` direğinden `c` direğine `b` direği üzerinden taşı.
2. `a` direğinde kalan diski `b` direğine taşı.
3. `n - 1` diski `c` direğinden `b` direğine `a` direği üzerinden taşı.

Her seferinde direk sayısı azaldığından bu algoritma sonlanma koşulunu sağlar. Taban durum, taşınacak disk sayısının 0 olduğu durumdur, bu durumda hiçbir şey yapılmayacaktır. O halde bu problemi çözen program Örnek 43'de görüldüğü gibi yazılabilir.

Bu algoritmanın en güzel yanlarından biri, deneme-yanılma yöntemiyle değil, herhangi bir diski gereksiz yere bir direkten bir direğe aktartmadan bir kerede çözümü bulmasıdır.

Uygulama: Rekürsiyon

Örnek 44. Çabuk Sıralama

DÜZELT: YAZILACAK

Sorular

Örnek 43 Hanoi kuleleri problemini çözen program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

#define DISKS 3

void move(int n, int a, int b, int c);

int main(void)
{
    move(DISKS, 1, 3, 2);
    return EXIT_SUCCESS;
}

void move(int n, int a, int b, int c)
{
    if (n > 0) {
        move(n - 1, a, c, b);
        cout << "Bir diski " << a << " direğinden "
             << b << " direğine taşı." << endl;
        move(n - 1, c, b, a);
    }
}
```

Örnek 44 Çabuk sıralama algoritmasını gerçekleyen program.

```
#include <iostream>             // cin,cout,endl
#include <stdlib.h>             // EXIT_SUCCESS

using namespace std;

void quicksort(int arr[], int first, int last);

int main(void)
{
    int numbers[] = { 26, 33, 35, 29, 19, 12, 22 };
    int i;

    quicksort(numbers, 0, 6);
    for (i = 0; i < 7; i++)
        cout << numbers[i] << endl;
    return EXIT_SUCCESS;
}

void swap(int &x, int &y)
{
    int tmp;

    tmp = x; x = y; y = tmp;
}

int partition(int arr[], int first, int last)
{
    int pivotloc = first;
    int pivot = arr[first];
    int i;

    for (i = first + 1; i <= last; i++) {
        if (arr[i] < pivot) {
            pivotloc++;
            swap(arr[pivotloc], arr[i]);
        }
    }
    swap(arr[first], arr[pivotloc]);
    return pivotloc;
}

void quicksort(int arr[], int first, int last)
{
    int pivotloc;

    if (first < last) {
        pivotloc = partition(arr, first, last);
        quicksort(arr, first, pivotloc - 1);
        quicksort(arr, pivotloc + 1, last);
    }
```

Ek A

Simgelerin Kodlanması

Bilgisayarlarda diğer her şey gibi simgeler de sayılarla gösterilirler. Bunun için hangi simgenin hangi sayıyla gösterileceği (ya da tersine, hangi sayının hangi simgeye karşı düşeceği) konusunda bir uzlaşma olması gerekir. Bu amaçla tanımlanan *kodlamalar*, simgelere birer numara verirler.

Yaygın kullanılan ilk kodlamalardan biri ASCII kodlamasıydı (Şekil A.1). ASCII, 7 bitlik bir kodlama olduğundan 128 farklı simgenin kodlanmasına olanak verir. İlk kodun numarası 0, son kodun numarası 127'dir. Bunlardan ilk 32 simge (0-31) ve son simge (127) "basılamaz" simgelerdir (satır sonu, bip sesi v.b.). Aradaki 95 simgeyse (32-126) İngilizce'nin bütün küçük ve büyük harfleri, rakamlar, noktalama işaretleri ve tuştakımı üzerinde gördüğünüz her türlü özel simgeyi içerir. ASCII kodlaması günümüzde kullanılan bütün kodlamaların temelini oluşturur.

	+0	+1	+2	+3	+4	+5	+6	+7
32		!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7
56	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G
72	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W
88	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g
104	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	

Tablo A.1: ASCII kodlama çizelgesi.

ASCII kodlaması, İngilizce dışında kalan abecelerin harflerini içermediğinden, her dil için o dilde bulunan farklı simgeleri de içeren 8 bitlik kodlamalar oluşturulmuştur; böylelikle 256 farklı simgenin kodlanmasına olanak sağlanmıştır. Bütün bu kodlamalarda ilk 128 simge

ASCII çizelgesinde olanın aynısıdır, düzenlemeler ikinci 128 simge üzerinde yapılır. Bu kodlamaların en bilineni ISO8859 standart ailesinde tanımlananlardır. ISO8859 kodlamalarında da 128-159 arası sayılar kullanılmaz.

- ISO8859-1: Batı Avrupa dilleri (latin1 kodlaması adıyla da bilinir)
- ISO8859-2: Doğu Avrupa dilleri
- ISO8859-9: Türkçe (latin5 kodlaması adıyla da bilinir). ISO8859-1 kodlamasıyla neredeyse aynıdır, yalnızca ISO8859-1'deki İzlandaca harflerin yerine Türkçe harflerin gelmesiyle oluşturulmuştur. İki kodlama arasında değişen 6 simge şunlardır: Ğ ğ İ ı Ş ş.

Son yıllarda, farklı diller konuşan, farklı ülkelerde yaşayan insanların ortak kullandıkları uygulamaların sayısı çok büyük bir hızla arttığından, bütün dillerin bütün simgelerini içeren bir kodlamaya geçmek bir zorunluluk halini almıştır. Bu amaçla geliştirilen Unicode kodlaması (resmi adıyla ISO10646-1), 16 ve 32 bitlik sürümleri olan bir kodlamadır. Yeryüzünde bilinen bütün dillerin simgelerini içermenin yanısıra daha pek çok ek simgenin tanımlanabilmesine de olanak verir. Ancak bütünüyle bu kodlamaya geçmek için yaygın kullanılan bütün programlarda büyük miktarlarda değişiklik gerekecektir. Bu nedenle, yeni yazılan uygulamalarda bu kodlamanın yalın hali olan UTF-16 ya da UTF-32 kodlamalarına uyulması öngörülürken, kullanımı süren eski uygulamalarla uyum sorununu azaltmak için bir geçiş kodlaması olarak UTF-8 geliştirilmiştir.

Bir değerin bilgisayar belleğinde bir göze ham bir veri olarak yazıldığı düşünülebilir. Sözgelimi, bir bellek gözünde 240 sayısı yer alıyor olsun. Program bu gözü bir tamsayı olarak değerlendirirse 240 değerini elde eder ve diyelim bu sayıyı başka bir sayı ile toplayabilir. ISO8859-9 kodlamasında bir simge olarak değerlendiriyorsa 'ğ' harfini elde eder ve Türkçe kurallarına göre sıralama yapmada kullanabilir. ISO8859-1 kodlamasında bir sayı olarak değerlendiriyorsa " harfini elde eder ve İzlandaca kurallarına göre sıralamada kullanabilir. Kısacası, değer bellekte ham haliyle bulunur; nasıl anlam verileceği, ne amaçla kullanılacağı programın belirleyeceği konulardır.

Ek B

Unix'de Program Geliştirme

B.1 Yardımcı Belgeler

Unix işletim sistemlerinde çoğu zaman `man` *`fonksiyon`* komutuyla o fonksiyonla ilgili bilgi alabilirsiniz. Bu komut fonksiyonun ne iş yaptığını ve hangi başlık dosyasında yer aldığını söyleyecektir. Örnek: `man sqrt`.

info

KDE: Konqueror ya da Alt-F2 #sqrt

B.2 Derleme

Unix ailesi işletim sistemlerinde C/C++ dilinde geliştirme yapmak için en çok GNU C Compiler (gcc) derleyicisi kullanılır. Temel kullanımı şöyledir:

```
gcc kaynak_dosyası -o çalıştırılabilir_dosya
```

Kaynak dosyanız bir C++ koduysa

```
g++ kaynak_dosyası -o çalıştırılabilir_dosya
```

komutuyla çalıştırmanız gerekir. Bu komutun sonucunda kaynak dosya derlenir, bağlanır ve ismi verilen çalıştırılabilir dosya oluşturulur.

Derleyici başlık dosyalarını `/usr/include`, kitaplık dosyalarınıysa `/usr/lib` kataloglarında arar. Derleyicinin çalışması bazı bayraklar yardımıyla denetlenebilir. Aşağıdaki örneklerde kaynak dosyasının adının `main.cpp`, hedef dosya adının `main.o`, çalıştırılabilir dosya adının da `main` olduğu varsayılırsa:

- Kaynak dosyayı yalnız derlemek istiyorsanız, yani bağlamak istemiyorsanız `-c` bayrağını kullanabilirsiniz:

```
g++ main.cpp -c -o main.o
```

- Derleyicinin bütün uyarılarılarını görmek istiyorsanız `-Wall` bayrağını kullanabilirsiniz:

  ```
  g++ main.cpp -Wall -o main
  ```

- Başka kataloglarda da başlık dosyası aramasını istiyorsanız `-I` bayrağıyla bunu belirtebilirsiniz:

  ```
  g++ main.cpp -I/usr/X11R6/include -o main
  ```

- Başka kataloglarda da kitaplık aramasını istiyorsanız `-L` bayrağıyla bunu belirtebilirsiniz:

  ```
  g++ main.cpp -I/usr/X11R6/include -L/usr/X11R6/lib -o main
  ```

- Optimizasyon: -O2

Birden fazla kaynak dosyasından oluşan bir projenin derlenmesi birkaç aşamalı olur. Örnek 35 üzerinde bir projenin nasıl derleneceğini görelim:

- `project.cpp` dosyasını derle (bağlamadan):

  ```
  g++ -c project.cpp -o project.o
  ```

- `ops.cpp` dosyasını derle (bağlamadan):

  ```
  g++ -c ops.cpp -o ops.o
  ```

- bağla

  ```
  g++ project.o ops.o -o project
  ```

Kaynak dosyalarının herhangi birinde bir değişiklik yapılırsa yalnızca o dosyanın yeniden derlenmesi ve bağlama yeterlidir.

Her seferinde elle derleme komutlarını yazmak zahmetli bir iş olduğundan derleme işlemlerini kolaylaştırmak için make aracı kullanılır. Bu araç, programcının yazdığı, derleme aşamalarını belirten `Makefile` isimli bir dosya denetiminde çalışır. `Makefile` dosyası hedeflerden oluşur. Her hedef için hedefin nelere bağlı olduğu ve nasıl oluşturulacağı belirtilir. Örnek bir hedef şöyle yazılabilir:

```
project: project.o ops.o
	g++ project.o ops.o -o project
```

Bu yazımın anlamı, `project` hedefinin `project.o` ve `ops.o` dosyalarına bağlı olduğu ve oluşturulması için ikinci satırda yazılan komutun kullanılacağıdır.[1] Bir hedefin bağlı olduğu dosyalarda değişme olursa (dosyanın tarih ve saati hedefinkinden yeniyse) hedefin yeniden oluşturulması gerektiğine karar verilir. Örnek projedeki diğer hedefler de şöyle yazılabilir:

[1] Hedefi oluşturmakta kullanılacak komutlar satır başından bir sekme içeriden başlamalıdır. Bir hedefi oluşturmakta birden fazla komut kullanılabilir.

```
project.o: project.cpp
	g++ -c project.cpp -o project.o
ops.o: ops.cpp
	g++ -c ops.cpp -o ops.o
```

`make` komutu çalıştırılırken hangi hedefin oluşturulmasının istendiği belirtilir. Hedef belirtilmezse dosyada bulunan ilk hedef oluşturulmaya çalışılır. Başlangıçta `project.o`, `ops.o` ve `project` dosyalarının hiçbirinin olmadığını varsayarsak, `make project` komutunun çalışması şöyle olur:

1. `project` hedefi oluşturulmaya çalışılır. Bu hedef `project.o` ve `ops.o` hedeflerine bağlı olduğu için ve bunlar henüz oluşturulmamış olduğu için sırayla bunlar oluşturulmaya çalışılır.

2. `project.o` hedefi oluşturulmaya çalışılır. Bu hedef `project.cpp` dosyasına bağlı olduğu için oluşturulmasına geçilebilir ve

   ```
   g++ -c project.cpp -o project.o
   ```

   komutu yürütülür.

3. `ops.o` hedefi oluşturulmaya çalışılır. Bu hedef `ops.cpp` dosyasına bağlı olduğu için oluşturulmasına geçilebilir ve

   ```
   g++ -c ops.cpp -o ops.o
   ```

   komutu yürütülür.

4. Artık `project` hedefinin bağlı olduğu iki dosya da oluşmuş olduğundan bu hedefin oluşturulmasına geçilebilir ve

   ```
   g++ project.o ops.o -o project
   ```

   komutu yürütülür.

İleride `ops.cpp` dosyasında bir değişiklik yapıldığını ve `make project` komutunun yürütüldüğünü düşünelim:

1. `project` hedefi oluşturulmaya çalışılır. Bu hedef `project.o` ve `ops.o` hedeflerine bağlı olduğu için önce onlara bakılır.

2. `project.o` hedefi oluşturulmaya çalışılır. Bu hedef `project.cpp` dosyasına bağlıdır ancak `project.cpp` dosyasının saati `project.o` dosyasından eski olduğu için bir işlem yapılmaz.

3. `ops.o` hedefi oluşturulmaya çalışılır. Bu hedef `ops.cpp` dosyasına bağlıdır ve `ops.cpp` dosyasının saati `ops.o` dosyasından yeni olduğu için hedefin yeniden oluşturulması gerektiğine karar verilir ve

```
g++ -c ops.cpp -o ops.o
```

komutu yürütülür.

4. `project` hedefine dönüldüğünde `ops.o` dosyasının saati artık `project` dosyasının saatinden yeni olduğu için bu hedefin de yeniden oluşturulması gerektiğine karar verilir ve

```
g++ project.o ops.o -o project
```

komutu yürütülür.

`Makefile` dosyalarında ayrıca bazı değişmezler tanımlanarak okunulurluk ve esneklik artırılabilir. Örnek proje için yazılmış bir `Makefile` Örnek 45'de verilmiştir. Bu dosya sıkça kullanılan, kaynak dosyalar dışında kalan her şeyi silen `clean` hedefini de içermektedir; `make clean` komutu sizin yazdığınız dışında kalan bütün dosyaların silinmesi için kullanılır.

Örnek 45 Örnek bir `Makefile` dosyası.

```
CXX=g++
CXXFLAGS=-O2 -Wall -g -pg
LDFLAGS=-pg

proje: proje.o ops.o
	$(CXX) $(LDFLAGS) proje.o ops.o -o proje

proje.o: proje.cpp
	$(CXX) $(CXXFLAGS) -c proje.cpp -o proje.o

ops.o: ops.cpp
	$(CXX) $(CXXFLAGS) -c ops.cpp -o ops.o

clean:
	rm -f *.o proje
```

Önişlemci: cpp

B.3 Editörler

nedit kate mcedit: yazım renklendirme, ayraç eşleştirme, makrolar, tab emülasyonu, kendiliğinden girintileme

B.4 Hata Ayıklayıcılar

Çalıştırılabilir dosya üzerinde hata ayıklama yapacaksanız `-g` bayrağını kullanmalısınız:

```
g++ main.cpp -g -o main
```

Hata ayıklama bilgilerini silmek için -s

strip

gdb ddd: adım adım çalıştırma, değişken değerlerini izleme

İşaretçilerle çalışırken yapılan hatalar sıklıkla programın çökmesine ve o anki bellek görüntüsünün bir dosyaya yazılmasına neden olurlar (core dump). Bu bellek görüntüsü programınızın neden çöktüğüne ilişkin önemli bilgiler içerir ve hatanın nedenini bulmanıza yardımcı olabilir. Örnek 39'de

```
if (x % factor == 0) {
```

satırından sonraki

```
f = new factor_t;
```

komutunu silerek programı derleyin ve çalıştırın. Program bellek hatası vererek çökecektir. ddd hata ayıklayıcısında önce "Open Program" komutuyla çalıştırılabilir dosyayı, sonra da "Open Core Dump..." komutuyla yaratılan `core` dosyasını açın. Programın çöktüğü anda

```
f->base = factor;
```

satırında kaldığını ve `f` işaretçisinin değerinin `NULL` olduğunu göreceksiniz.

B.5 Başarım İnceleme

Programınızdaki fonksiyonların kaçar kere çağrıldıklarını ve hangi fonksiyonda ne kadar zaman harcandığını ölçmek için gprof aracını kullanabilirsiniz. gprof'un kullanılabilmesi için gcc/g++ derleyicisine hem derleme hem bağlama aşamasında `-pg` bayrağı verilmelidir. Örnek 23'de hangi fonksiyonun kaç kere çağrıldığını görmek için:

```
g++ -pg lcm.cpp -o lcm
```

komutuyla program derlendikten sonra çalıştırılır ve en küçük ortak katı hesaplanmak istenen değerler girilir. Bu çalışma sonucunda `gmon.out` isimli bir dosya oluşur ve

```
gprof lcm gmon.out
```

komutu programın çalışmasıyla ilgili bir rapor üretir. 32424 ve 563324 değerlerinin girilmesiyle oluşan örnek bir raporun bazı bölümleri aşağıda verilmiştir:

```
  %   cumulative    self  self   total
time    seconds seconds calls us/call   us/call name
95.24      0.20    0.20 70516    2.84      2.84 is_prime
4.76       0.21    0.01     2 5000.00 105000.00 factorize
0.00       0.21    0.00 13127    0.00     15.24 next_prime
0.00       0.21    0.00     1    0.00      0.00 merge_factors
```