

Stephen G. Kochan

Programfejlesztés **C nyelven**

Átfogó bevezetés a C programozási nyelvbe



SAMS



Áttekintés

1. fejezet • Bevezetés 1
2. fejezet • Alapok 5
3. fejezet • Első programunk lefordítása
és futtatása 13
4. fejezet • Változók, adattípusok,
aritmetikai kifejezések 23
5. fejezet • Ciklusszervezés 45
6. fejezet • Döntési szerkezetek 67
7. fejezet • Tömbök 97
8. fejezet • Függvények 123
9. fejezet • Adatszerkezetek 169
10. fejezet • Karakterláncok 199
11. fejezet • Mutatók 239

- 12. fejezet • Bitműveletek 285
 - 13. fejezet • Az előfeldolgozó 307
 - 14. fejezet • Mélyebben az adattípusokról 331
 - 15. fejezet • Nagyobb programok 343
 - 16. fejezet • A C kimeneti és bemeneti műveletei 359
 - 17. fejezet • Speciális lehetőségek a C nyelvben 387
 - 18. fejezet • Programok nyomkövetése 407
 - 19. fejezet • Objektumközpontú programozás 431
-
- A függelék • A C nyelv összefoglalása 447
 - B függelék • A szabványos C programkönyvtár 495
 - C függelék • Programok fordítása gcc-vel 525
 - D függelék • Gyakoribb programozási hibák 529
 - E függelék • Források 535
- Tárgymutató 541

Tartalomjegyzék

Előszó xxiii

A szerzőről	xxiv
Köszönetnyilvánítás	xxiv
Kíváncsiak vagyunk az Ön véleményére!	xxiv

1. fejezet • Bevezetés 1

2. fejezet • Alapok 5

A programozás	5
Magas szintű programnyelvek	6
Operációs rendszerek	6
Programok fordítása	7
Integrált Fejlesztői Környezetek (IDE)	10
Forrásnyelvi értelmezők	10

3. fejezet • Első programunk lefordítása és futtatása 13

A program lefordítása	13
A program futtatása	14
Vizsgáljuk meg első programunkat	15

Változók értékének megjelenítése	17
Megjegyzések	19
Gyakorlatok	21

4. fejezet • Változók, adattípusok, aritmetikai kifejezések 23

Változók	23
Adattípusok, konstansok	25
Az int egész típus	25
A tárolt számok kiterjedése és méréthatárai	26
A float lebegőpontos típus	26
A nagypontosságú double lebegőpontos típus	27
A char karakter típus	28
A logikai adattípus: _Bool	28
A long, long long, short, unsigned és signed típusmódosítók	30
Aritmetikai kifejezések	33
Egész aritmetika és az egyoperandusú mínusz operátor	36
A modulus operátor	38
Egész és lebegőpontos típusátalakítások	39
A típusátalakító (casting) operátor	41
Értékkadás és műveletvégzés egy lépésben	42
Az _Complex és az _Imaginary típus	43
Gyakorlatok	43

5. fejezet • Ciklusszervezés 45

A for utasítás	46
Relációs operátorok	47
A kimenet formába öntése	52
A program bemenete	53
Egymásba ágyazott for ciklusok	54
Változatok for ciklusra	56
Többszörös kifejezések	56
Mezők elhagyása	57
Deklaráció a ciklusfejben	57

A while utasítás	58
A do utasítás	62
A break utasítás	63
A continue utasítás	64
Gyakorlatok	64

6. fejezet • Döntési szerkezetek 67

Az if utasítás	67
Az if-else szerkezet	71
Összetett feltételvizsgálat	73
Egymásba ágyazott if utasítások	76
Az else if szerkezet	78
A switch utasítás	85
Logikai változók	88
A feltételkezelő operátor	92
Gyakorlatok	94

7. fejezet • Tömbök 97

Tömbök megadása	98
Tömbelemek felhasználása számlálóként	102
Fibonacci-számok előállítása	105
Prímszámkeresés tömb használatával	106
Tömbök inicializálása	108
Karaktertömbök	110
Számrendszerök közti váltás tömb használatával	111
A const típusmódosító	113
Többsdimenziós tömbök	115
Változó méretű tömbök1	117
Gyakorlatok	120

8. fejezet • Függvények 123

Függvények megadása	123
Paraméterek és lokális változók	126

A függvényprototípus deklarációja	127
Automatikusan létrejövő lokális változók	128
Visszatérési érték	130
Függvényt hívó függvényt hívó függvény...	135
A visszatérési érték és a paraméter típusának deklarálása	138
Függvények paramétereinek ellenőrzése	140
Programfejlesztés felülről lefelé építkezve	141
Függvények és tömbök	142
Hozzárendelő operátorok	146
Tömbök rendezése	148
Többdimenziós tömbök	151
Többdimenziós, változó méretű tömbök használata függvényekben ..	154
Globális változók	156
Automatikus és statikus változók	160
Rekurzív függvények	163
Gyakorlatok	166

9. fejezet • Adatszerkezetek 169

Dátumot tároló adatszerkezet	170
Adatszerkezetek használata kifejezésekben	172
Függvények és adatszerkezetek	175
Adatszerkezet az idő tárolására	181
Adatszerkezetek inicializálása	184
Összetett betűkonstansok	185
Adatszerkezetekből álló tömbök	186
Adatszerkezeteket tartalmazó adatszerkezetek	190
Tömböket tartalmazó adatszerkezetek	192
Variációk adatszerkezetekre	195
Gyakorlatok	196

10. fejezet • Karakterláncok 199

Karaktertömbök	200
Változó méretű karakterláncok	202
Karakterláncok inicializálása és kiírása	205
Két karakterlánc egyenlőségének vizsgálata	208

Karakterláncok beolvasása	210
Egykarakternyi bemenet	212
Az üres karakterlánc	217
Vezérlőkarakterek	220
Bővebben a konstans karakterláncokról	223
Karakterláncok, adatszerkezetek és tömbök	224
A keresési módszer megjavítása	227
Műveletek karakterekkel	231
Gyakorlatok	235

11. fejezet • Mutatók 239

Mutató megadása	239
Mutatók használata kifejezésekben	244
Mutatók és adatszerkezetek	245
Mutatókat tartalmazó adatszerkezetek	247
Láncolt listák	249
A const kulcsszó és a mutatók	257
Mutatók és függvények	259
Mutatók és tömbök	264
Rövid kitérő a programok optimalizálásáról	268
EZ most tömb vagy mutató?	269
Karakterláncok mutatói	271
Konstans karakterláncok és a mutatók	272
További részletek az inkrementáló és dekrementáló utasításokról	274
Műveletek mutatókkal	278
Függvénymutatók	279
Mutatók és memóriaicímek	280
Gyakorlatok	282

12. fejezet • Bitműveletek 285

Bitenkénti operátorok	286
A bitenkénti ÉS művelet	287
A bitenkénti VAGY művelet	290
A bitenkénti KIZÁRÓ VAGY művelet	291
A bitenkénti negáció művelete	292

Bitenkénti eltolás balra	294
Bitenkénti eltolás jobbra	295
Egy saját eltolási függvény	295
Bitek körbeforgatása	297
Bitmezők	300
Gyakorlatok	305

13. fejezet • Az előfeldolgozó 307

A #define utasítás	307
A program kiterjeszthetősége	311
A program hordozhatósága	312
Bonyolultabb definíciók	314
Paraméterek és makrók	316
Különböző számú paraméter átadása makróknak	319
A # operátor	320
A ## operátor	321
Az #include utasítás	323
A rendszer fejlécállományai	325
Feltételes fordítás	325
Az #ifdef, az #endif, az #else és az #ifndef utasítás	326
Fejlécállományok többszörös beemelésének elkerülése	327
Az #if és az #elif utasítás	328
Az #undef utasítás	329
Gyakorlatok	330

14. fejezet • Mélyebben az adattípusokról 331

A felsorolt (enumerated) típus	331
A typedef utasítás	335
Adattípusok átalakítása	338
Az előjel kiterjesztése	339
A paraméterek átalakítása	340
Gyakorlatok	341

15. fejezet • Nagyobb programok 343

Egy program fájlokra bontása	343
Összetartozó forrásfájlok lefordítása parancssorból	344
A modulok közti kommunikáció	346
Külső változók	346
A statikus, illetve a külső változók és függvények összehasonlítása	349
Fejlécállományok hatékony használata	351
A nagyobb programok írását támogató segédprogramok	353
A make segédprogram	353
A cvs segédprogram	356
Unix segédprogramok: ar, grep, sed és a többiek	356

16. fejezet • A C kimeneti és bemeneti műveletei 359

Karakterszintű bemenet/kimenet: a getchar és a putchar	359
Formázott kimenet/bemenet: printf és scanf	360
A printf függvény	360
A scanf függvény	367
Kimeneti és bemeneti műveletek fájlokkal	372
A kimenet/bemenet fájlba irányítása	372
Állomány vége	375
Fájlkezelő függvények	376
Az fopen függvény	376
A getc és a putc függvény	378
Az fclose függvény	378
Az feof függvény	381
Az fprintf és az fscanf függvény	381
Az fgets és az fputs függvény	382
Az stdin, az stdout és az stderr	382
Az exit függvény	383
Állományok átnevezése és mozgatása	384
Gyakorlatok	385

17. fejezet • Speciális lehetőségek a C nyelvben 387

Néhány újabb utasítás	387
A goto utasítás	387
A null utasítás	388
Az unió (union)	389
A vessző operátor	392
Típusminősítők	393
A register minősítő	393
A volatile minősítő	393
A restrict minősítő	394
Parancssori paraméterek	394
Dinamikus memóriafoglalás	399
A malloc és a calloc függvény	400
A sizeof operátor	400
A free függvény	403

18. fejezet • Programok nyomkövetése 407

Nyomkövetés az előfeldolgozó segítségével	407
Nyomkövetés gdb-vel	414
A változók kezelése	417
A forráskód megjelenítése	419
A program végrehajtásának vezérlése	419
Töréspont beszúrása	419
Soronkénti továbblépés	420
Töréspontok felsorolása és törlése	424
A hívási verem (stack trace) kijelzése	424
Függvényhívások, értékadás tömböknek és adatszerkezeteknek	425
A gdb parancsainak súgója	426
Kiegészítések	428

19. fejezet • Objektumközpontú programozás 431

Végül is mi az az objektum?	431
Példányok és metódusok	432
Törtek kezelése C nyelven	434
Objective-C osztály definiálása törtek kezeléséhez	435
C++ osztály definiálása törtek kezeléséhez	440
C# osztály definiálása törtek kezeléséhez	442

A függelék • A C nyelv összefoglalása 447

1.0 Kettős karakterek és azonosítók	447
1.1 Kettős karakterek	447
1.2 Azonosítók	447
1.2.1 Nemzetközi karakterek	448
1.2.2 Kulcsszavak	448
2.0 Megjegyzések	448
3.0 Konstansok	449
3.1 Egész konstansok	449
3.2 Lebegőpontos konstansok	450
3.3 Karakterkonstansok	450
3.3.1 Vezérlőszorozatok (escape sequences)	450
3.3.2 Széles karakterkonstansok	451
3.4 Karakterlánc-konstansok	451
3.4.1 Karakterláncok összekapcsolása	451
3.4.2 Több bájton tárolt karakterek	452
3.4.3 Széles karakterekből képzett konstansok	452
3.5 Felsorolt konstansok	452
4.0 Adattípusok és deklarációk	452
4.1 Deklarációk	452
4.2 Alapvető adattípusok	452
4.3 Származtatott adattípusok	454

4.3.1 Tömbök	455
4.3.2 Adatszerkezetek	457
4.3.3 Uniók	459
4.3.4 Mutatók	460
4.4 Felsorolt adattípusok	461
4.5 A typedef utasítás	462
4.6 A const, a volatile és a restrict típusmódosítók	463
5.0 Kifejezések	463
5.1 A C operátorok összefoglalása	464
5.2 Konstans kifejezések	467
5.3 Aritmetikai operátorok	468
5.4 Logikai operátorok	469
5.5 Relációs operátorok	469
5.6 Bitenkénti műveletek	469
5.7 Inkrementáló és dekrementáló operátorok	470
5.8 Hozzárendelő operátorok	470
5.9 A feltételkezelő operátorpár	471
5.10 A sizeof operátor	471
5.12 A vessző operátor	472
5.13 Alapvető műveletek tömbökkel	472
5.14 Alapvető műveletek adatszerkezetekkel 1	472
5.15 Alapvető műveletek mutatókkal	473
Tömbmutatók	473
Adatszerkezetekre irányuló mutatók 2	474
5.16 Összetett betűkonstansok (compound literals)	474
5.17 Alapvető adattípusok átalakítása	475
6.0 Tárolási osztályok és a hatókör	476
6.1 Függvények	477
6.2 Változók	477
7.0 Függvények	479
7.1 A függvények definíciója	479
7.2 Függvényhívások	480
7.3 Függvénymutatók	481
8.0 Utasítások	481
8.1 Összetett utasítások	482
8.2 A break utasítás	482
8.3 A continue utasítás	482
8.4 A do utasítás	482

8.5 A for utasítás	483
8.6 A goto utasítás	483
8.7 Az if utasítás	483
8.8 A null utasítás	484
8.9 A return utasítás	484
8.10 A switch utasítás	485
8.11 A while utasítás	486
9.0 Az előfeldolgozó	486
9.1 Hármas karakterek (trigraph sequences)	486
9.2 Az előfeldolgozó utasításai	487
9.2.1 A #define utasítás	487
9.2.2 Az #error utasítás	489
9.2.3 Az #if utasítás	489
9.2.4 Az #ifdef utasítás	490
9.2.5 Az #ifndef utasítás	491
9.2.6 Az #include utasítás	491
9.2.7 A #line utasítás	492
9.2.8 A #pragma utasítás	492
9.2.9 Az #undef utasítás	492
9.2.10 A # utasítás	493
9.3 Előre definiált azonosítók	493

B függelék • A szabványos C programkönyvtár 495

Szabványos fejlécállományok	495
<stddef.h>	495
<limits.h>	496
<stdbool.h>	497
<float.h>	497
<stdint.h>	498
Karakterlánc-kezelő függvények	498
Memóriakezelő függvények	500
Karakterkezelő függvények	502
Kimeneti/bemeneti függvények	503
A memóriában történő formátumátalakítás függvényei	508
Karakterláncok számmá alakítása	509
Dinamikus memóriakezelő függvények	511

Matematikai függvények	512
Komplex aritmetika	520
Általános segédfüggvények	522

C függelék • Programok fordítása gcc-vel 525

A fordítási parancs általános formája	525
Parancssori kapcsolók	526

D függelék • Gyakoribb programozási hibák 529

1. Rossz helyre tett pontosvessző	529
2. Az = operátor használata a == operátor helyett	529
3. A prototípus-deklaráció elhagyása	530
4. Az operátorok végrehajtási sorrendjének eltévesztése	530
5. Karakterkonstans és karakterlánc összetévesztése	530
6. Tömb indexhatárainak megsértése	531
7. A karakterláncot lezáró null karakter helyigényének elfelejtése	531
8. A -> és a . operátorok összetévesztése adattagokra történő hivatkozáskor	531
9. A „címe” operátor (&) elhagyása a scanf nem mutató paraméterei elől ..	531
10. Mutató használata annak inicializálása előtt	532
11. A break elhagyása a case ágak végéről a switch utasításban	532
12. Pontosvessző írása az előfeldolgozónak szánt utasítás végére	532
13. A paraméterek körüli zárójelek elhagyása makródefinícióban	532
14. A makró neve és paraméterlistája közti szóköz a #define-ban	533
15. Mellékhatással rendelkező kifejezés használata makróhívásban	533

E függelék • Források 535

Válaszok a kérdésekre, hibajegyzék stb.	535
A C programozási nyelv	535
Könyvek	535
Weboldalak	536
Hírcsoportok	536
C fordítóprogramok és integrált fejlesztői környezetek	537
gcc	537
MinGW	537
CygWin	537
Visual Studio	537
CodeWarrior	537
Kylix	538
Kiegészítések	538
Objektum-orientált programozás	538
A C++ nyelv	538
A C# nyelv	538
Az Objective-C nyelv	539
Fejlesztői eszközök	539

Tárgymutató 541

Előszó

Nehéz elhinni, de ennek a könyvnek az első kiadását 20 ével ezelőtt írtam. Abban az időben csupán a Kernighan és Ritchie által írt *The C Programming Language (A C Programozási Nyelv)* című könyv volt az egyetlen vetélytársa a piacon. Nagyon változott azóta a világ!

Amikor az 1980-as évek elején elkezdtek beszélni egy készülő ANSI C szabványról, az anyagot két könyvre bontottuk szét. Az egyik a *Programfejlesztés C Nyelven* volt, míg a másik a *Programfejlesztés ANSI C-ben* címet kapta. Ennek az volt az oka, hogy évekbe telt, mire a fordítóprogramok gyártói kiadták a saját ANSI szabvány szerint működő termékeiket, illetve mire azok használata általánosan elterjedtté vált. Mivel pedig úgy éreztem, hogy ha egyetlen könyvben tárgyalom az ANSI és a nem ANSI szerinti fejlesztés fogásait, azzal csak összeavarom az olvasót – ráadásul esetünkben tankönyvről van szó – ezért inkább két könyvet írtam.

Az ANSI C szabvány 1989-es első kiadása óta maga is többször változott. Az utolsó változata a C99. Tulajdonképpen ennek a megjelenése tette szükségessé a kötet újbóli átdolgozását. Ebben a könyvben tehát a régi anyag mellett bemutatom mindeneket a módosításokat is, amelyek a legújabb szabványban bukkantak fel.

A C99 szabvány tárgyalása mellett a kötet két új fejezetet is tartalmaz. Az egyik a C programok nyomkövetésével foglalkozik, a másik az egyre gyakrabban alkalmazott objektumközpontú megközelítést tárgyalja. Ez utóbbi fejezetet az tette szükségessé, hogy a fejlesztők jelenleg számos olyan objektum-orientált nyelvet használnak, amely valamilyen módon a C-ből származik. Ilyen a C++, a C#, a Java és az Objective-C.

Ezen a helyen szeretnék köszönetet mondani azoknak, akik olvasták a könyv korábbi kiadásait, és megírták nekem észrevételeiket. Biztos vagyok benne, hogy a jelenlegi kötet minőségéhez nagyban hozzájárult ez az információ és engem is arra ösztönzött, hogy ne hagyjam abba az írást.

Akik pedig először tartják kezükben ezt az kötetet, azokat szintén arra bátorítom, hogy írjak meg a vele kapcsolatos észrevételeiket és persze őszintén remélem, hogy hasznos és élvezet olvasmány lesz számukra.

Stephen Kochan
2004 június
steve@kochan-wood.com

A szerzőről

Stephen Kochan több mint 20 éve fejleszt programokat C nyelven. Több ezzel a témával foglalkozó igen népszerű könyvnek szerzője vagy társzerzője (*Programming in C*, *Programming in ANSI C*, *Topics in C Programming*). Publikációs tevékenységének másik fő területe a UNIX operációs rendszer (*Exploring the Unix System*, *Unix Shell Programming*, *Unix System Security*). Legújabb könyve a *Programming in Objective-C*, amely a C nyelv egy objektum-orientált kiterjesztését tárgyalja.

Köszönetnyilvánítás

A könyv írása során nyújtott segítségükért köszönetemet szeretném kifejezni a következőknek: Douglas McCormick, Jim Scharf, Henry Tabickman, Dick Fritz, Steve Levy, Tony Iaminno és Ken Brown. Külön köszönet illeti Henry Mullish-t (New York University) azért, hogy olyan sok minden megtanított nekem az írásról és ezzel elindította publikációs tevékenységemet.

A Sams Publishing szerkesztőgárdájából a következőket illeti köszönet: Mark Renflow, Dan Knott, Karen Annett (felelős szerkesztő), Bradley Jones (szakmai szerkesztő). Végezetül szeretnék köszönetet mondani a Sams minden olyan dolgozójának is, aki részesei voltak ennek a projektnek, még ha nem is találkoztam mindegyikükkel személyesen.

Kíváncsiak vagyunk az Ön véleményére!

Mint a könyv olvasója, Ön, kedves olvasó a mi legfőbb kritikusunk. Éppen azért nagyra értékeljük az Ön véleményét. Tudni szeretnénk, ha valamit jól csináltunk, és persze azt is, ha nem. Tudni szeretnénk, milyen más területekről olvasna szívesen könyveket ebben a sorozatban, és természetesen örömmel fogadunk minden egyéb megjegyzést is.

A Sams és a Kiskapu Kiadó örömmel meghallgatja mindenki véleményét. Kérjük írja meg nekünk, mi az, ami tetszett Önnek vagy ami nem tetszett ebben a könyvben, hiszen ez nagyban segítheti további munkánkat, és azt, hogy egyre jobb minőségű könyveket adjunk ki.

Fontosnak tartjuk megjegyezni, hogy a könyv tartalmával kapcsolatos technikai kérdésekben általában nem tudunk segíteni. A Sams Kiadónak ugyanakkor van egy Felhasználói Szolgáltató részlege, amely a könyvvel kapcsolatos kérdéseket megkapja, és esetleg megválaszolja.

Ha ír nekünk, kérjük, adja meg a könyv pontos címét és szerzőjét, a nevét, az e-mail címét és esetleg a telefonszámát. Amennyiben módunkban áll, megvizsgáljuk a felvetést vagy problémát, illetve megosztjuk azt a szerzővel és a szerkesztővel.

A könyv eredeti angol kiadóját a feedback@samps publishing.com címen érheti el. Ha hangsúlyos levélben kívánja a Sams Kiadót megkeresni, levelét a következő címen várjuk:

Email: feedback@samps publishing.com
Mail: Michael Stephens
Associate Publisher
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

A magyar kiadással kapcsolatos megjegyzéseit kérjük a kiado@kiskapu.hu címre küldje.

A Sams egyéb kiadványaival kapcsolatban kérjük látogasson el www.samps publishing.com címre. Ha ezzel a konkrét könyvvel kapcsolatban keres valamit a SAMS webhelyén, a keresőmezőbe kérjük az eredeti angol kiadás ISBN számát (0672326663) gépelje be.

A magyar kiadás weboldalát és letölthető anyagait a Kiskapu Kiadó webhelyén (www.kiskapukiado.hu) találja meg. Mindkét helyen lehetőség van téma illetve kulcsszó szerinti keresésre is.

1

Bevezetés

A C programozási nyelv megalkotója az 1970-es évek elején Dennis Ritchie, az AT&T Bell Laboratories programozója volt. Az 1970-es évek végéig C fordítóprogramok csak a Bell Laboratories falain belül léteztek. Az ezt követő időkben terjedt el aztán ez a programnyelv széles körben és tett szert komoly támogatottságra. Térhódítását nagyban előmozdította a Unix elterjedése, mely szintén az AT&T Bell Laboratories fejlesztése; a Unix leginkább támogatott programozási nyelve ugyanis a C volt. Nem csoda, hisz maga a Unix operációs rendszer is 90%-ban C-ben íródott.

Az IBM PC (és másolatainak) óriási sikere miatt aztán hamarosan az MS-DOS lett a C nyelv legfontosabb környezete. A C programozási nyelv (operációs rendszereken átívelő) népszerűségének növekedtével minden több cég próbált ügyesen helyezkedni, és egyre másra jelentek meg a különféle C fordítók a piacon. A C nyelv definícióját illetően ezek a programok nagyrészt a Brian Kernighan és Dennis Ritchie által kiadott könyv (*A C programozási nyelv – The C Programming Language*) mellékletére támaszkodtak. Ez a melléklet sajnos nem határozta meg egyértelműen a C nyelvet, így bizonyos kérdésekben a fordítóprogramok fejlesztői a saját józan belátásuk szerint döntötték.

Az 1980-as évek elejétől egyre erősebb igény jelentkezett a C nyelv szabványosítására. A kérdésben az Amerikai Nemzeti Szabványügyi Hivatal (*American National Standards Institute, ANSI*) volt az illetékes, így 1983-ban (*X3J11* néven) fel is állított egy *ANSI C* bizottságot, melynek feladataul a C szabványosítását tűzték ki. 1989-ben a bizottság javaslatát jóváhagyták, így 1990-ben megjelenhetett az első szabványos *ANSI C* definíció.

Mivel a C programozási nyelv az egész világon elterjedt, a Nemzetközi Szabványügyi Szervezet (*International Standard Organization, ISO*) is érintetté vált a szabványosítási kérdésekben. A szervezet elfogadta az *ANSI* szabványt, mely ekkor az *ISO/IEC 9899:1990* nevet kapta. Később újabb módosítások kerültek bele a nyelvbe. A legutóbbi szabványt 1999-ben fogadták el. Neve *ANSI C99* vagy *ISO/IEC 9899:1999*. Ez az a verzió, melyen könyvünk is alapul.

A C magas szintű programnyelv, ám ezzel együtt számos lehetőséget nyújt „hardverközeli” műveletek végzésére is, vagyis megengedi, hogy alacsonyabb szinten kommunikálunk a hardveregységekkel. Bár a C általános célú, strukturált programnyelv, eredetileg rendszerközeli alkalmazások programozásra terveztek, így hihetetlen rugalmassággal és hatékonyssággal ajándékozza meg a felhasználót.

Könyvünk célja, hogy megtanítsa az olvasót C-ben programozni. Nem feltételezünk semmilyen programozási előismeretet. Bár elsősorban azoknak szeretnénk sokat nyújtaní, akik most ismerkednek a programozással, könyvünk a tapasztalt programozókhöz is szól. Akik rendelkeznek már programozási tapasztalatokkal, valószínűleg úgy fogják érezni, hogy C-ben gyakran egészen sajátos megoldások adhatók egy-egy problémára, melyek határozottan eltérnek az általuk korábban megszokottaktól.

A C nyelv valamennyi részletét megvizsgáljuk. Valahányszor előkerül egy-egy új programozási elem, egy kicsi, de teljes programmal mutatjuk be a használatát. E könyv filozófiájának alapja abban áll, hogy a legjobban példákon át lehet tanítani. Ahogy egy kép többet mond ezer szónál, ugyanúgy egy-egy jól megválasztott példaprogram is sokat tud adni. Ha az olvasónak lehetősége van C nyelvű programok írására, akkor az lesz a legjobb, ha letölti és lefuttatja a példaprogramokat, majd a kimenetet összehasonlíta a könyvben lévővel. Ezzel nemcsak a nyelv szintaxisát és részleteit illetően tehet szert mélyebb ismeretekre, hanem komoly gyakorlatot szerezhet a programok bevitele, fordítása és futtatása terén is.

A programkód olvashatósága szintén erős hangsúlyt kapott könyvünkben. Én úgy hiszem, hogy a programokat úgy kell megírni, hogy el is lehessen őket olvasni. Akár mi magunk olvassuk a saját kódunkat, akár más, hamar rá fogunk jönni, hogy kellő idő elteltével még a program szerzőjére is az újdonság erejével tud hatni a saját munkája. A tapasztalat és persze a józan ész is azt mutatja, hogy az ebben a könyvben bemutatott kódolási stílust használva a programokat könnyebb megírni, megérteni, nyomkövetni és módosítani is. A programok olvashatósága amúgy természetes következménye annak, ha a programozó végig ragaszkodik a strukturált programozás előírásaihoz.

Mivel tankönyvről van szó, az egyes fejezetekben tárgyalt anyag mindenkorábbiakra épül. A leghasznosabban akkor forgathatjuk tehát ezt a kötetet, ha sorban haladunk az egyes fejezeteken. Nem ajánlatos kihagyni egyetlen részt sem, sőt még a fejezetvégi gyakorlatokat és kérdéseket sem.

A 2. fejezet („Alapok”) bevezetést nyújt a magas szintű programnyelvek általános elméletébe és a programfordítás folyamatába. Ennek megismerése szükséges ahhoz, hogy a további tudnivalókat is megérthessük. A 3. fejezetben („Első programunk lefordítása és futtatása”) rövid bevezetést kapunk a C nyelvű programozás gyakorlati lépéseihe. Ettől kezdve, egészen a 16. fejezetig a nyelv összes lényeges jellemzőjét számba vesszük. A 16. fejezetben aztán elkezdünk mélyebbre ásni, először a kimeneti és be-

meneti műveletek rejtelmeiben. A 17. fejezetben („Speciális lehetőségek a C nyelvben”) szó esik néhány olyan speciális megoldásról, melyekre a C programozási nyelv lehetőséget nyújt.

A 18. fejezet („Programok nyomkövetése”) arról szól, hogyan használható az előfeldolgozó a nyomkövetés elősegítésére. Az olvasó bevezetést kap az interaktív nyomkövetés gyakorlatába is, melyre a népszerű gdb nyomkövetőt használjuk majd.

A programozók világát az elmúlt évtizedben elbűvölte az objektumorientált programozás (OOP). Maga a C nem objektumorientált, ám több olyan leszármazottja van, amely dicsekedhet a fenti jelzővel. A 19. fejezet („Objektumorientált programozás”) rövid áttekintést ad az OOP-ről, annak fogalmairól és „szóhasználatáról”. Három programozási nyelvet használunk itt össze: a C++, a C# és az Objective-C nyelveket.

Az A függelék teljes összefoglalást nyújt a C nyelvről – ez a rész értelemszerűen referenciaként használható.

A B függelék áttekinti a szabványos C programkönyvtár számos függvényét. Ezek minden olyan számítógépen megtalálhatóak, melyek támogatják a C nyelvű programozást.

A C függelék a GNU C fordító (gcc) használatát, lehetőségeit, opciót mutatja be afféle kézikönyvként.

A D függelék bemutatja a leggyakoribb programozási hibákat, tévesztési lehetőségeket mutatja be.

Végül az E függelékben felsoroltam azokat a műveket, amelyekből további hasznos információkat szerezhetünk a C nyelvről, vagy ötleteket további tanuláshoz.

A fejezetvégi kérdések megfejtései megtalálhatók a www.kochan-wood.com weboldalon.

A könyv nem feltételezi semelyik operációs rendszer használatát – bármelyik használható, amelyiken van C fordító. A szövegben mindenkorral végig kizárolag a gcc fordító használatáról lesz szó, minden ezzel fogunk fordítani és futtatni. Szeretnél köszönetet mondanival mindazoknak, akik segítettek a szöveg különböző változatainak gondozásában: Douglas McCormick, Jim Scharf, Henry Tabickman, Dick Fritz, Steve Levy, Tony Ianino és Ken Brown nevét kell megemlítenem. Hasonlóképp hálásan köszönöm a New York Egyetem tanárának, Henry Mullishnak a munkáját, aki megtanított írni és elindított a publikálás útján.

A könyv egy korábbi kiadását Maureen Connellynek, a Hayden Book Company egykori szerkesztőjének ajánlottam. Nekik köszönhettem e könyv első kiadását.

2

Alapok

Ebben a fejezetben tisztázzuk azokat az alapfogalmakat, melyek nélkülözhetetlenek a C nyelvű programozás megértéséhez. Áttekintjük a magas szintű programnyelveket általában, valamint az ilyen nyelven írt programok gépi nyelvre történő lefordítását (*compiling*).

A programozás

A számítógépek alapvetően igen buták. Csak azt képesek elvégezni, amit pontosan megmondunk nekik. A legtöbb számítógépes rendszer egy nagyon alacsony szintű nyelven hajtja végre a műveleteket. Azt a legtöbb számítógép tudja, hogy hogyan kell egy számot eggyel megnövelni, vagy leellenőrizni, hogy nulla-e egy számérték. Az alapvető műveletek köre alig bővebb ennél. A számítógép által (gépi szinten) futtatható alapvető műveletek halmazt gépi utasításkészletnek hívjuk.

Ahhoz, hogy egy feladatot számítógép segítségével megoldjunk, a számítógép utasításkészletéből vett műveletekkel kell azt megfogalmaznunk. A számítógépes program nem más, mint utasítások olyan sorozata, amely alkalmas egy feladat megoldására. Algoritmusnak hívjuk azt a logikai utat, elvi módszert, amit egy feladat megoldása közben használunk. Ha például olyan programot írunk, amely előönti, hogy egy szám páros vagy sem, akkor ehhez egymás után kell illesztenünk néhány alkalmas gépi utasítást. Ezek összessége a program. Az a módszer, amely alapján előöntjük, hogy páros-e egy szám, az algoritmus. Amikor egy feladat megoldására programot fejlesztünk, akkor először általában algoritmikus nyelven fogalmazzuk meg a megoldást, és csak ezután írjuk meg az adott algoritmust végrehajtó programot. A paritást előöntő program algoritmusa valahogy így fest: osszuk el a számot kettővel. Ha az osztási maradék nulla, akkor a szám páros, egyébként pedig páratlan. Az algoritmus ismeretében már megírhatjuk az azt megvalósító programot, azaz megfogalmazhatjuk a szükséges utasításokat egy adott számítógéptípusra. Az utasításokat egy konkrét programozási nyelv parancsaival célszerű kifejezni, például Visual Basic, Java, C++ vagy C nyelven.

Magas szintű programnyelvek

Az első számítógépeket csak gépi nyelven, vagyis nullák és egyesek meghatározott sorozataival lehetett programozni, melyek a számítógépen használható utasítások és memóriacímek bináris megfelelői voltak. A szoftvertechnológia fejlődésének következő lépéseként megjelentek az assembly nyelvek, melyek egy fokkal magasabb absztrakciós szinten tettek lehetővé a munkát a programozó számára. Nem kellett már az összes műveletet bináris számokkal kódolni, hanem szimbolikus rövidítések és memóriacímek használatával lehetett programokat írni. Az assembler az a speciális fordítóprogram, mely az assembly nyelvű programokat gépi nyelvre (vagyis az adott számítógép utasításkészletére) ülteti át.

Mivel minden assembly utasítás kölcsönösen egyértelműen megfeleltethető egy gépi utasításnak, az assembly nyelveket is alacsony szintű nyelvnek tekintjük. Az assembly programozónak a program írásakor ismernie kell az adott számítógépes rendszer utasításkészletét, az eredményül kapott program pedig nem hordozható, vagyis más processzortípuson nem futatható változtatás nélkül. Az egyes processzorok utasításkészlete eltérő, és mivel az assembly nyelvű programok az adott utasításkészlethez készülnek, erősen gépfüggőek.

Az assembly után születtek meg az úgynevezett magas szintű programozási nyelvek. Az előző a Fortran (*FOR*mula *TRAN*slation) volt. A Fortranban dolgozó programozóknak nem kellett már az adott számítógép architektúrájával törődniük. A Fortran programok műveleteit sokkal magasabb szinten lehetett megfogalmazni, mint a processzor utasításkészletét használó programok esetén. A Fortran parancsait több gépi nyelvű lépéssel valósították meg, azaz messzire eltávoladtak az assemblyben megszokott kölcsönösen egyértelmű megfeleltetéstől, mely az assembly utasítások és a gépi kódú lépések között fennállt.

A magas szintű programnyelvek szintaxisának rögzítésével lehetőség nyílt gépfüggetlen programok írása. Az ilyen programok változtatás nélkül (vagy minimális változtatással) futathatóak bármely számítógépen, mely az adott programozási nyelvet támogatja.

Magas szintű programnyelv akkor használható, ha van fordítóprogram, mely a program utasításait olyan alacsony szintű nyelvre alakítja át (*compile*), melyet a számítógép közvetlenül megért. Más szóval a fordítóprogram (*compiler*) az az elem, amely lefordítja a „gép nyelvére” a magas szintű programozási nyelven írt parancsokat.

Operációs rendszerek

Mielőtt továbblépnénk a fordítóprogramok megismerésében, érdemes pár percre elidőzünk egy sajátos programcsoporthoz az úgynevezett operációs rendszereknél.

Az operációs rendszer az a program, amely az egész számítógép működését felügyeli. A számítógépen végrehajtott összes kimeneti/bemeneti (azaz I/O, *input/output*) művelet az operációs rendszeren keresztül valósul meg, de ez felügyeli a számítógép erőforrásait és vezérli az egyes programok futását is.

Az egyik legnépszerűbb operációs rendszer a Bell Laboratories által kifejlesztett és számos számítógéprendszeren kiválóan futtatható Unix. Ennek később különféle változatai is megjelentek, mint például a Linux vagy a Mac OS X. Eredetileg az operációs rendszereket egy konkrét géptípushoz készítették. Mivel azonban a Unix a gép felépítésével kapcsolatban meglehetősen kevés előfeltételezéssel él, ráadásul C nyelven íródott, viszonylag kis erőfeszítéssel sikerült számos, egymástól egészen különböző számítógéprendszerre is átültetni.

A Microsoft Windows XP egy másik példája a népszerű operációs rendszereknek. Ez leginkább Pentium(-kompatibilis) processzorokat tartalmazó számítógépeken fut.

Programok fordítása

A fordítóprogram is egy szoftver, mely alapjában véve alig különbözik a könyünkben található más programoktól, csak jóval bonyolultabb. A fordítóprogram elemzi az adott programozási nyelven írt forráskódot, és ebből kiindulva olyan formátumú programot hoz létre, amely futtatható az adott számítógépes rendszeren.

A 2.1 ábra mutatja a C nyelvű program fordításának főbb lépéseit. A folyamat gyakorlatilag a program beírásából, fordításából és végrehajtásából áll. Az is látható az ábrán, hogy az adott lépésekkel milyen Unix parancssal lehet végrehajtani.

A lefordítandó programot be kell írnunk egy szöveges állományba. A különféle számítógéprendszeren más-más megegyezés szerint történhet az állományok elnevezése, ám végső soron a programozó döntheti el a létrehozandó fájl nevét. A C programozási nyelven írt kódokat általában „.c” végződéssel szoktuk elnevezni. Ez nem követelmény, inkább csak afféle „közmegegyezés”. A prog1.c valószínűleg megfelelő név lesz az olvasó számítógépen is egy C program számára.

A legegyszerűbb szövegszerkesztő (*editor*) is alkalmas C program írására. Az összes Unix rendszeren megtalálható például a vi szövegszerkesztő. A programot tartalmazó szöveges állományt forrásprogramnak vagy forráskódnak hívjuk, ugyanis ez a kiindulópontja minden C nyelvű programnak. A begépelés után nekiláthatunk a program lefordításának.

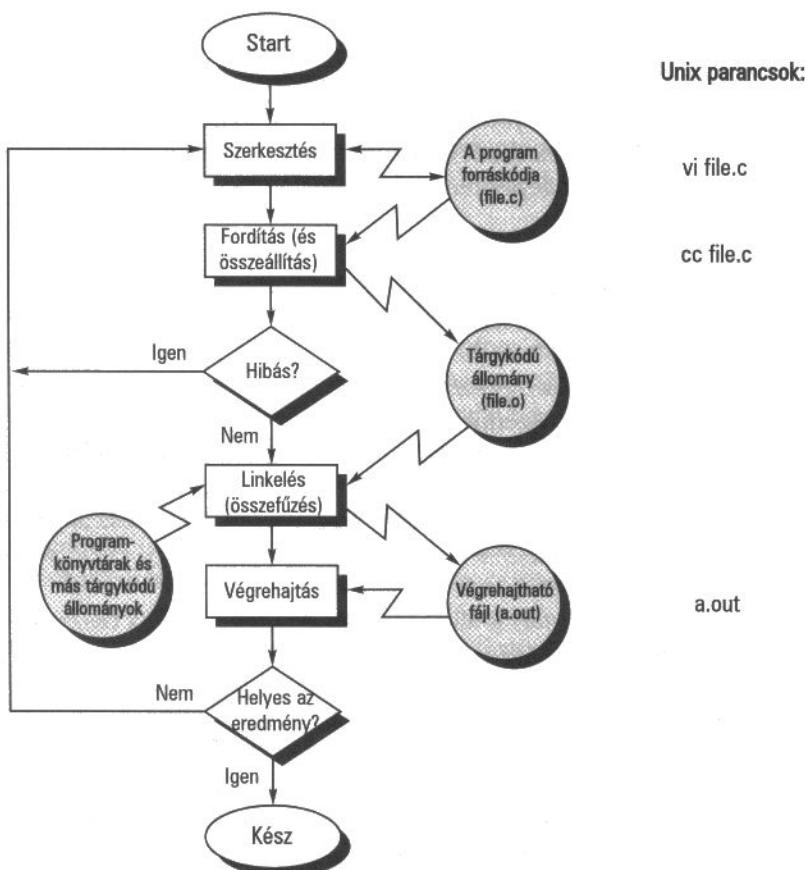
A fordítási folyamatot egy rendszerparancs begépelésével tudjuk elindítani. A parancs neve után megadandó a lefordítandó állomány neve. Unix rendszereken például a cc parancs indítja el a C programok fordítását. Ha a népszerű GNU C fordítót használjuk, akkor a gcc parancssal is elérhetjük ugyanezt. A

```
gcc prog1.c
```

parancs beírásával elkezdődik tehát a prog1.c állományban tárolt C program lefordítása.

A folyamat első lépéseként a fordítóprogram megvizsgálja az állományban található szöveget, hogy nincs-e benne szintaktikai vagy más értelmezési (szemantikai) hiba¹.

Ha a fordító bármilyen hibát talál, akkor ezt azonnal jelzi a felhasználónak, és megszakítja a fordítás folyamatát. Ekkor nyílik lehetőség arra, hogy a forráskódban – szövegszerkesztő használatával – kijavítsuk a hibákat, és újra elindítsuk a fordítást. Ebben a szakaszban tipikusan számítanak az olyasfajta hibák, hogy például elfelejtünk bezárni egy megnyitott zárójelet (szintaktikai hiba), vagy definiáltan változót akarunk használni (értelmezési hiba).



2.1. ábra

Egy C nyelvű program bevitelének, fordításának és futtatásának megvalósítása parancssori eszközökkel.

¹ Technikailag ez annyit jelent, hogy a C fordító végez egy első átfésülést, úgynevezett „előfeldolgozást”, amely során adott utasításokat keres. Az előfeldolgozás lépései részletesen megvizsgáljuk majd a 13. fejezetben.

Miután az összes szintaktikai és értelmezési hibát kijavítottuk, a fordítóprogram feldolgozza a program utasításait, és átülteti egy „alacsonyabb” szintre. A legtöbb rendszeren ez annyit jelent, hogy a fordítóprogram assembly nyelven állítja elő a lefordítandó program parancsainak megfelelő utasításokat.

Az ekvivalens assembly utasítássorozat létrehozása után a fordítási folyamat következő lépése az utasítások átültetése az aktuális gép utasításkészletére. Ezt a lépést egy külön assembler programmal is meg lehet valósítani, de a legtöbb rendszeren ez az elem egybe van építve magával a fordítóprogrammal, tehát nem kell vele külön foglalkoznunk.

Az assembler lefordítja az assembly utasításokat, és az eredményt egy úgynevezett tárgykódú bináris formátumra alakítja, amit el is ment egy állományba. Unix alatt ezt az állományt általában ugyanúgy nevezi el a fordítóprogram, ahogy a forrásprogramot, csak a fájlnév „.c” helyett „.o”-ra (*object, tárgykód*) fog végződni. Windows alatt általában .obj a kiterjesztése a tárgykódú állományoknak.

A tárgykódú állományok készen állnak az összefűzésre, vagyis a linkelésre. Unix alatt (ha a `cc` vagy `gcc` parancssal indítottuk a fordítást) ez automatikusan végbemegy. A linkelés vagy szerkesztés célja olyan formátumú állomány létrehozása, mely már közvetlenül alkalmas az adott számítógépen történő futtatásra. Ha az újonnan írt program használ más programokat, amelyeket már korábban lefordítottak, akkor ezek a linkelési művelet során lesznek befűzve a végrehajtható állományba. A hivatkozott programkönyvtárakban tárolt programrészletek is ekkor kerülnek be a programba.

A fordítás és linkelés lépését egybefoglalva a program felépítésének (*building*) is hívjuk.

A linkelés végeredményeként kapott állomány, amely már futtatható (*executable*) bináris formátumban van, készen áll az operációs rendszer által felügyelt végrehajtásra. Unix alatt (ha nem adunk meg más nevet) ezt a.out néven szokta előállítani a fordítóprogram. Windows alatt a forrásprogram nevének „.c” kiterjesztése „.exe”-re cserélődik a futtatható állomány nevében.

A program elindításához minden össze kell írni a futtatható állomány nevét:

`a.out`

Ennek hatására az a.out állomány betöltődik a számítógép memoriájába és megkezdődik a végrehajtása.

A program végrehajtásakor annak összes utasítása lépésről lépésre végrehajtódik. Ha a program a felhasználótól szeretne bemenő (*input*) adatot kérni, akkor a program futása áll mindenig, amíg az adatbevitel meg nem történik. Olyan is előfordul, hogy a program egy ese-

mény bekövetkezésére vár (például egérkattintásra). A végeredmény, amit kimenetnek (output) is szoktunk hívni, megjelenhet a képernyőn egy ablakban vagy a parancssorban, vagy akár egy állományba is irányítható.

Ha minden rendben megy (ami valószínűleg nem a program első futtatásakor fog bekövetkezni), a program végrehajtja feladatát. Ha nem ez történik, vagyis a program futtatása nem a várt eredményt hozza, akkor át kell gondolni, hogy mit is csináltunk, meg kell vizsgálni a program felépítését. Ezt a műveletet nyomkövetésnek (debugging) hívjuk, amellyel célunk a felderítendő (vagy már ismert) hibák kiküszöbölése. Ennek során minden valószínűség szerint szükség lesz az eredeti forrásprogram módosítására is. Ilyen esetben a teljes fordítási ciklust (fordítás, linkelés és végrehajtás) addig kell ismételgetni, míg elő nem áll a kívánt eredmény.

Integrált Fejlesztői Környezetek (IDE)

Az előző részben megismertük a C nyelvű programok fordítási ciklusát. Szó esett arról is, hogy milyen parancsokkal lehet végrehajtani az egyes lépéseket. A szerkesztésből, fordításból, futtatásból és nyomkövetésből álló műveletsort gyakran egyetlen programmal oldjuk meg, melyet Integrált Fejlesztői Környezetnek hívunk (*Integrated Development Environment; IDE*). Ez általában egy grafikus felhasználói felületen futó program, amely kifejezetten nagy méretű szoftverek forrásprogramjainak kezelését is lehetővé teszi. A különböző ablakokban könnyen megvalósítható programjaink szerkesztése, fordítása, futtása és nyomkövetése.

Mac OS X alatt sok programozó kedveli a Code Warrior, illetve az Xcode nevű integrált fejlesztői környezeteket. Windows alatt talán a Microsoft Visual Studio-t említhetjük meg elsőként a legnépszerűbb fejlesztői környezetek között. Linux alatt többen kedvelik a Kylix fejlesztői környezetet. (Említést érdemel talán még a Geany és az Anjuta is.) Ezek a programok nagymértékben egyszerűsítik a programfejlesztést, így érdemes időt szánni valamelyiknek a megismerésére. A legtöbb IDE a C nyelven kívül más programnyelveket is támogat (például C#-ot vagy C++-t).

A különféle fejlesztői környezetekkel kapcsolatban további információt találhatunk az E függelékben, a Források közt.

Forrásnyelvi értelmezők

Mielőtt befejeznénk a fordítási folyamat bemutatását, érdemes megemlíteni egy másfajta megközelítést is. A magas szintű programnyelveken írt programokat más módon is lehet futtatni: léteznek forrásnyelvi értelmezők (*interpreters*), melyek segítségével a programokat nem kell előre lefordítani, mert futás közben értelmezi őket a rendszer. Az éppen futtatandó utasítást az értelmező közvetlenül a végrehajtása előtt értelmezi. Ily módon egy-

szerűbbé válik a programok nyomkövetése, a hibák kiderítése. Ennek a módszernek ugyanakkor hátránya a jóval lassabb programfutás, hiszen az egyes parancsokat újra meg újra gépi nyelvű utasításokká kell alakítani.

A BASIC és a Java nyelvű programokat a legtöbbször nem fordítják le futtatható állományokká, hanem ilyen forrásnyelvi értelmező segítségével futtatják. A Unix világában is találunk erre a megoldásra példákat: ilyen például maga a rendszerhét (*shell*) vagy a Python nyelv.

Egyes gyártók a C programnyelvhez is készítenek forrásnyelvi értelmezőt.

3

Első programunk lefordítása és futtatása

Ebben a fejezetben az olvasó bevezetést kap a C programozás gyakorlatába. És mi mással is lehetne ezt hatásosabban szemléltetni, mint egy igazi C nyelvű programmal?

Kezdjük tehát egy egyszerű programmal, amely minden össze annyit ír ki a képernyőre, hogy Programming is fun. Ezt a feladatot valósítja meg a 3.1 Listában látható kód.

3.1 Lista • Első C programunk

```
#include <stdio.h>

int main (void)
{
    printf ("Programming is fun.\n");
    return 0;
}
```

A C programozási nyelvben különböző nem számítanak a kis- és nagybetűk. Az utasítások egy soron belül bárhol elkezdődhetnek, amely lehetőséget felhasználhatjuk programjaink olvashatóbbá tételehez. A programozók gyakran tabulátorral oldják meg a sorok tagolását.

A program lefordítása

Visszatérve első C programunk megírásához az első lépés nem más, mint egy szöveges állomány elkészítése. Bármilyen egyszerű szövegszerkesztő alkalmas erre a célra. A Unix felhasználók például gyakran a vi-t vagy az emacs-ot használják.

A legtöbb C fordító onnan ismeri fel a C programokat, hogy „.c”-re végződik a nevük. Tegyük fel tehát, hogy a 3.1 Listának megfelelő programot egy prog1.c nevű szövegfájl tartalmazza. Ezek után már csak a program lefordítása van hátra.

Ha GNU C fordítót használunk, akkor ez az alábbi egyszerű parancs kiadásával történik:

```
$ gcc prog1.c
```

Ha a szabványos Unix C fordítót használjuk, akkor `gcc` helyén `cc` áll. A begépelendő karaktereket félkövéren szedtük. A dollár jelképezi a parancssori promptot (készenléti jelet), amennyiben terminálablakból indítjuk a fordítást. Az olvasó gépen a promptkarakter természetesen ettől eltérő is lehet.

Ha valami hiba csúszott a kódba, a `gcc` parancs kiadását követően a fordító visszajelzést küld. Ilyenkor általában megjelenik a hibás részletet tartalmazó sor száma is. Ha egyszerűen csak visszakapjuk a promptot, az arra utal, hogy hibátlanul lefordult a program.

Amikor a fordító lefordítja és linkeli a programot, akkor tulajdonképpen annak egy (számítógép által) végrehajtható változatát hozza létre. A GNU vagy a szabványos C fordítót használva ennek a neve alapértelmezésként a nem túl beszédes `a.out` lesz. Windows alatt a végrehajtható program gyakran az `a.exe` nevet viseli.

A program futtatása

Ezek után a program futtatása minden össze abból áll, hogy begépeljük a parancssorba a program nevét¹:

```
$ a.out
Programming is fun.
$
```

Természetesen ettől eltérő nevet is adhatunk a végrehajtható állománynak már a fordítás során. A `-o` kapcsoló (azaz a kis o betű) után írhatjuk be a kívánt fájlnevet. Az alábbi utasítás hatására például a `prog1.c` program futtatható változatának neve `prog1` lesz:

```
$ gcc prog1.c -o prog1
```

A program a név beírásával indítható:

```
$ prog1
Programming is fun.
$
```

¹ Ha az `a.out: No such file or directory`, (`a.out: Nincs ilyen állomány vagy könyvtár`) hibajelzést kapjuk, akkor ez valószínűleg azt jelenti, hogy az aktuális könyvtár nem szerepel az elérési utak között, vagyis nincs benne a PATH változóban. Ezen egyszerűen módosíthatunk, vagy ha ezt nem szeretnénk, hivatkozhatunk kifejezetten az aktuális könyvtárra programunk indításakor a következő módon: `./a.out`.

Vizsgáljuk meg első programunkat

Vegyük jobban szemügyre első programunkat. A program első sorában látható utasítást szinte minden C program elején megtaláljuk:

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

Ez tudósítja a fordítóprogramot arról, hogy a program későbbi részében szereplő printf kimeneti parancsot hogyan kell megvalósítani. A 13. fejezetben tárgyaljuk majd ennek a részleteit.

A program következő sora szerint a program belépési pontjául szolgáló függvény neve main, futása végeztével pedig egy egész (*integer*, rövidítve: int) számértéket fog az operációs rendszernek visszaadni.

```
int main (void)
```

Maga a main egy speciális név, azt jelzi, hogy a program végrehajtását a kódon belül *hol* kell elkezdeni. A main után álló zárójelpár arra utal, hogy egy függvényről van szó. A zárójelben álló void (*üres*) kulcsszó pedig jelzi, hogy ez a függvény nem vár paramétert. Ezek a fogalmak a legapróbb részletekig előkerülnek majd a 8. fejezetben.

Miután jelezük a rendszernek, hogy a main függvény következik, ideje megadni, hogy mi is lesz az elvégzendő feladat. Ezt úgy oldjuk meg, hogy a megfelelő programlépéseket kapcsos zárójelek közé foglaljuk. Az összes parancsot, ami a most következő kapcsos zárójelek között szerepel, a main függvény részének tekinti a fordítóprogram. A 3.1 Listában ez minden össze két utasításból áll. Az elsőből az derül ki, hogy egy printf nevű eljárást szeretnénk meghívni. A számára átadandó paraméter egy karakterlánc:

```
"Programming is fun.\n"
```

A printf eljárás a C programkönyvtár része. Feladata egyszerűen annyi, hogy kiírja a képernyőre a neki átadott paramétert (vagy paramétereket, amint azt később látni fogjuk). A két utolsó karakter a „\” (backslash) és az „n” a közmegegyezés szerint az újsor karaktert jelenti. Az újsor karakter pontosan azt jelenti, amit a neve sugall: egy új sor kezdetére viszi a kimeneti megjelenést. Az ezt követő karakterek már ebben az új sorban jelennek meg. Ez igen hasonlít a régmúlt idők írócépeinek „kocsi vissza” műveletéhez. (Emlékszik még egyáltalán valaki erre?)

A C nyelvben minden utasítást pontosvesszővel (;) kell lezárni – emiatt látható a printf utasítás meghívását követően is a pontosvessző.

Az utolsó utasítás azt jelzi, hogy a main végrehajtásának befejeztével 0 lesz a futási állapotra utaló visszatérési érték:

```
return 0;
```

Itt bármilyen más egész számot használhatunk. A nulla a közmegegyezés szerint azt jelenti, hogy a program sikeresen, azaz hiba nélkül lefutott. A különböző számok különféle hibajelenségekre utalhatnak (például arra, hogy a program nem talált meg egy állományt). A futási állapotra utaló visszatérési értéket más programok (például a Unix héj) könnyen meg tudják vizsgálni. Bizonyos helyzetekben fontos lehet tudni, hogy hibátlanul lefutott-e egy-egy program.

Első programunk végiggondolása után vehetjük a bátorságot ahhoz, hogy módosítsunk rajta egy keveset. Írassuk ki vele például azt is, hogy And programming in C is even more fun. (*És C-ben programozni még nagyobb örööm*). Ez egy újabb printf utasítással oldható meg, ahogy az a 3.2 Listában látható. Ne feledkezzünk el arról, hogy minden C utasítást pontossázzal kell lezárnunk.

3.2 Lista

```
#include <stdio.h>

int main (void)
{
    printf ("Programming is fun.\n");
    printf ("And programming in C is even more fun.\n");
    return 0;
}
```

Ha begépeljük a 3.2 Listát, majd lefordítjuk és futtatjuk, akkor az alábbi kimenetet kapjuk a parancssorban (vagy terminálablakban).

3.2 Lista • Kimenet.

Programming is fun.

And programming in C is even more fun.

A következő kódrészlet arra mutat rá, hogy a kimenet különböző sorait nem feltétlenül kell különálló printf utasításokkal kiíratni. Arra kérem az olvasót, hogy vegye szemügyre a 3.3 Listát, és próbálja megjósolni a kimenetét anélkül, hogy megnézne a könyvben.

Nem szabad csalni!

3.3 Lista • Többsoros kimenet kijelzése

```
#include <stdio.h>
int main (void)
{
    printf ("Testing...\n...1\n...2\n....3\n");
    return 0;
}
```

3.3 Lista • Kimenet

```
Testing...
...1
...2
....3
```

Változók értékének megjelenítése

Könyvünkben a `printf` valószínűleg a leggyakoribb utasítás, mivel egyszerű és kényelmes módon lehetővé teszi egy program futási eredményeinek a kijelzését. Nemcsak előre megfogalmazott szövegeket lehet vele kiíratni, hanem változók értékét vagy számítások végeredményét is. Példaként nézzük meg a 3.4 Listát, melyben a `printf` utasítással két szám (50 és 25) összegét íratjuk ki a képernyőre.

3.4 Lista • Változók kijelzése

```
#include <stdio.h>

int main (void)
{
    int sum;

    sum = 50 + 25;
    printf ("The sum of 50 and 25 is %i\n", sum);
    return 0;
}
```

3.4 Lista • Kimenet

```
The sum of 50 and 25 is 75
```

A 3.4 Lista főprogramjában az első utasítás nem más, mint a `sum` változó deklarációja. Itt jelezzük, hogy ez egy egész (`int`) típusú változó lesz. A C nyelvben minden változót deklarálni kell annak használata előtt. A fordítóprogram számára itt derül ki, hogy miként kell bánnia az adott változóval, milyen szerepe lesz a programban. Ez az információ ahhoz szükséges, hogy a lehető leghatékonyabb gépi utasítássor jöhessen létre a változók értékeinek eltárolásához és visszányeréséhez. Egy egészként deklarált változó csak (tizedes-

tört nélküli) egész számokat tárolhat, például ilyeneket: 3, 5, -20, 0. A tizedestörtet is talmazó számokat (mint például 3.14, 2.455, vagy 27.0) lebegőpontos vagy valós számoknak hívjuk.

A sum egész változó tárolja a két egész szám, az 50 és a 25 összegét. Szándékosan maradt ki egy sor a változó deklarációja és első használata között, hogy vizuálisan is nyilvánvaló legyen: más a szerepe a deklaráció(k)nak és a program utasításainak. Ez az elkülönítés ugyanakkor csak stiláris, sokszor olvashatóbbá válik tőle a programkód. Az összeadás szinte minden programnyelven ugyanígy néz ki:

```
sum = 50 + 25;
```

Az 50-et (ahogy az a plusz jelből látszik) hozzáadjuk a 25-höz, az eredményt pedig (az egyenlőségjellel jelölt hozzárendelési operátorral) a sum változóba helyezzük.

A printf utasításnak most két paramétere van, melyek a zárójelben vesszővel elválasztva sorakoznak egymás után. Az első paraméter egy karakterlánc, amit meg szeretnénk jeleníteni. Programjainkban gyakran nem csupán előre meghatározott szövegeket szeretnénk kiíratni, hanem bizonyos változók értékeit is. Esetünkben a sum változó értékét szeretnénk megjeleníteni a

The sum of 50 and 25 is

szövegrész után. Az itt felbukkanó százalék szimbólumnak speciális jelentése van a printf számára. A százalék utáni karakterből derül ki az itt megjelenítendő változó típusa. Programunkban az i karakterrel jelezzük a printf utasításnak, hogy egy egész (integer) számot kell kiírnia.²

Amikor a printf utasítás egy %i szimbólumot talál az első paraméterben, akkor annak helyén (egész számként) megjeleníti a következő paraméter értékét. Példánkban a sum a második paraméter, így ennek értéke fog megjelenni a

The sum of 50 and 25 is

szövegrész után.

Próbáljuk megint megjósolni a 3.5 Lista kimenetét a program alapján.

² Bár a printf utasítás a %d szimbólumot is elfogadja az egész típus jelzésére, a könyvben a %i használata mellett maradunk.

3.5 Lista • Több érték kijelzése

```
#include <stdio.h>

int main (void)
{
    int value1, value2, sum;

    value1 = 50;
    value2 = 25;
    sum = value1 + value2;
    printf ("The sum of %i and %i is %i\n", value1, value2, sum);

    return 0;
}
```

3.5 Lista • Kimenet

The sum of 50 and 25 is 75

A program első utasítása egészkként (`int`) határozza meg a `value1`, `value2`, és a `sum` változókat. Ez akár három különálló deklarációs sorban is megadható:

```
int value1;
int value2;
int sum;
```

A három változó deklarációja után a program hozzárendeli az 50-et a `value1`-hez, illetve a 25-öt a `value2`-höz. Kiszámítjuk ezek összegét, és az eredményt hozzárendeljük a `sum` változóhoz.

A `printf` utasítást most négy paraméterrel hívjuk meg. Az első paraméter (a formátumleíró karakterlánc) megadja, hogy hogyan történjen a hátralevő paraméterek megjelenítése. A `value1` értéke rögtön a „The sum of” szövegrész után jelenik meg. Ehhez hasonlóan a `value2` és a `sum` értékének is ott kell megjelennie, ahol a formátumleíró karakterláncban a következő két `%i` szerepel.

Megjegyzések

A fejezet utolsó programjában (3.6 Lista) kerül elő a megjegyzés fogalma. A megjegyzéseknek az a szerepe, hogy programjainkban dokumentálni tudjuk a fontosabb mozzanatokat, javítva ezzel azok olvashatóságát. Ahogy a következő példán látni fogjuk, a megjegyzésekkel tájékoztathatjuk a program olvasóját arról, hogy mi volt a programozó fejében, amikor egy programot (vagy egy adott kód részletet) megírt. A „program olvasója” lehet maga a szerző, de bárki más is, akinek később karban kell tartania azt.

3.6 Lista • Megjegyzések használata a programban

```
/* A program két egész értéket ad össze,
majd megjeleníti az eredményt */
```

```
#include <stdio.h>

int main (void)
{
    // Változók deklarációja
    int value1, value2, sum;

    // Az értékek változókhöz rendelése és összeadása
    value1 = 50;
    value2 = 25;
    sum = value1 + value2;

    // Az eredmény kiíratása
    printf ("The sum of %i and %i is %i\n", value1, value2, sum);

    return 0;
}
```

3.6 Lista • Kimenet

The sum of 50 and 25 is 75

Kétféleképp is lehet megjegyzésekkel illeszteni a C nyelvű programokba. Lehet perjellel és csillaggal: /* kezdeni egy megjegyzést, mely egészen a csillag és perjel: */ szimbólum-párig fog tartani. Az a szövegrész, mely e két jelpár: /* és */ között áll, mellőzve lesz a program fordításakor. Ez a fajta megjegyzés akkor előnyös, amikor több sort szeretnénk megjegyzésbe tenni. A másik lehetőség rövid megjegyzések írására alkalmas. Írunk két perjelt követenül egymás után: //. Innentől egészen a sor végéig megjegyzésnek tekinti a fordítóprogram a szöveget.

A 3.6 Listában négy megjegyzés is szerepel. Ezektől eltekintve maga a program teljesen megegyezik a 3.5 Listában bemutatottal. Valljuk meg, hogy a négy közül igazából csak a program elején álló megjegyzést nevezhetjük hasznosnak, a többi csak a módszer bemutatását szolgálja. (Azt is láthatjuk, hogy az apró és haszontalan megjegyzések tömkelege inkább rontja egy program olvashatóságát, semmint javítja.)

Nem lehet elégé hangsúlyozni az intelligens megjegyzések fontosságát. Számtalanszor előfordul, hogy egy programozó beleolvas saját, fél éve írt programjába, és szinte kiveri a hideg verejték, mert egyszerűen elképzelése sincs, hogy egy-egy kódrészlettel mi volt a célja az adott helyen. Ezen egy megfelelően elhelyezett, józanul megfogalmazott rövid megjegyzés rengeteget segíthet. Megjegyzések hiányában ugyanakkor hihetetlen mennyiségű időt lehet elpazarolni programjaink, algoritmusaink logikájának újrafelfedezésével.

Jó hozzászokni ahhoz, hogy már a program kódolásakor beírjuk a megfelelő megjegyzéseket. Ennek több oka is van. Egyszerűbb sokkal könnyebb dokumentálni a programot, míg annak logikája még frissen a fejünkben van, mintsem a program megírásának végeztével újra átgondolni minden. Másrészt nagyon hasznos lehet, ha már a program megírásának korai szakaszában megfelelő helyre kerülnek a megjegyzések. Az esetleges logikai hibák feltárásakor, a nyomkövetéskor óriási segítséget jelenthet egy-egy jó eltalált szöveges iránymutatás. A megjegyzések nemcsak abban segítenek, hogy megértsük a program működését, hanem abban is, hogy könnyebben rátalálunk egy-egy hibára. Meg kell persze azt is jegyeznem, hogy még nem találkoztam olyan programozóval, aki örömet lelte volna a megjegyzések írásában, vagy a program dokumentálásában. Amikor egy nehezebb hibakeresés végére ér az ember, nem szívesen áll neki megjegyzéseket írni. Ha azonban a megjegyzéseket már a program fejlesztése közben megírjuk, könnyebben elviselhető a dokumentálás kínos kötelessége is.

Ezzel végére is értünk rövid „technikai bevezetőnknek”. Azzal a megnyugtató érzéssel törhet nyugovóra az olvasó, hogy már írt C programot, és ha úgy adódik, szinte bármilyen más programot is meg fog tudni írni a későbbiekbén. A következő fejezetben újabb finomságokat tanulunk meg erről a gyönyörű, hatékony és rugalmas programozási nyelvről. De előtte végezzünk még néhány ujjgyakorlatot hiszen érdemes konkrét kérdések segítségével tisztázni, hogy valóban megértettük-e a fejezetben tanult fogalmakat.

Gyakorlatok

1. Gépeljük be és futassuk le a fejezetben szereplő hat példaprogramot! A kimeneteit hasonlítsuk össze a könyvben szereplő programkimenetekkel.
2. Írunk programot, mely a következő szöveget írja ki a képernyőre:
 1. A C nyelvben különbözőnek számítanak a kis- és nagybetűk.
 2. A main kulcsszó jelzi a program végrehajtásának kezdetét.
 3. A nyitó és záró zárójelek egy kódblokkba záraják az utasításokat.
 4. A program minden utasítását pontosvesszővel kell lezárni.
3. Milyen kimenetet várunk az alábbi programtól?

```
#include <stdio.h>

int main (void)
{
    printf ("Testing...");
    printf ("....1");
    printf ("...2");
    printf ("..3");
    printf ("\n");

    return 0;
}
```

4. Írunk programot, amely 87-ből kivon 15-öt, és az eredményt (egy szöveges üzenet részeként) a képernyőre írja!
5. Keressük meg az alábbi programban megbúvó szintaktikai hibákat! A hibamentes változatot gépeljük be és próbáljuk ki! Így lehet a legbiztosabban meggyőződni arról, hogy valóban minden hibát megtaláltunk.

```
#include <stdio.h>
int main (Void)
{
    INT sum;
    /* COMPUTE RESULT
    sum = 25 + 37 - 19
    /* DISPLAY RESULTS //
    printf ("The answer is %i\n" sum);
    return 0;
}
```

6. Mi lesz az alábbi program kimenete?

```
#include <stdio.h>

int main (void)
{
    int answer, result;

    answer = 100;
    result = answer - 10;
    printf ("The result is %i\n", result + 5);

    return 0;
}
```

4

Változók, adattípusok, aritmetikai kifejezések

Ez a fejezet további ismereteket nyújt a változókról, konstansokról. Részletesen megvizsgáljuk a C nyelv alapvető adattípusait, valamint az aritmetikai kifejezésekkel kapcsolatos főbb tudnivalókat.

Változók

A korai számítógépek programozónak nagy súly volt a vállán: bináris formában kellett megírniuk a programokat, méghozzá olyan nyelvjárásban, amit az adott számítógép elfogadott. Az utasításokat a programozóknak kellett bináris számokká alakítaniuk, majd ezeket lehetett bevenni a számítógéphez. Ugyancsak a programozónak kellett számszerűleg megadni (íráskor és olvasáskor) minden egyes adattároló memóriacímét.

A mai programnyelvek felhasználóinak már nem kell gépi kódiszavakon vagy memóriacímeken töprengenüük – minden erejükkel a megoldandó problémára koncentrálhatnak. Ebben nagy szerepe van a szimbolikus nevek, az úgynevezett változónevek használatának, melyek különféle számítási eredményeket, értékeket hordozhatnak. A változók nevét a programozó határozhatja meg, célszerűen oly módon, hogy a név tükrözze a tárolt érték szerepét.

A 3. fejezetben már használtunk változókat egész értékek tárolására. A sum változó például a 3.4 Listában két egész szám, 50 és 25 összegét tartalmazta.

A C nyelvben természetesen nemcsak egész típusú változókat használhatunk; tudunk lebegőpontos számokat, karaktereket vagy akár memóriacímre hivatkozó mutatókat is tárolni. Használatukhoz azonban előre deklarálni kell a kívánt típust.

A változónevek létrehozására egyszerű szabályok érvényesek. Betűvel vagy aláhúzás karakterrel (_) kell kezdődniük, majd betűk és számok (illetve aláhúzás) tetszőleges sorozatával folytatódhatnak. A kis- és nagybetűk nem tekintendők azonosnak.

Íme néhány példa, melyek eleget tesznek a fenti kíváncsolomnak:

```
sum
pieceFlag
i
J5x7
Number_of_moves
_sysflag
```

Álljon itt néhány rossz példa is, melyek a C-ben nem használhatók változónévként:

sum\$value	A \$ karakter nem használható.
piece flag	Szóközök nem használhatóak.
3Spencer	A változónevek nem kezdődhetnek számokkal.
int	Az int foglalt kulcsszó.

Az int azért nem használható változónévként, mert speciális jelentése van a C fordítóprogram számára. Az ilyeneket foglalt névnek vagy kulcsszónak hívjuk. Általában is igaz, hogy nem használhatjuk változónévként azokat a karakterláncokat, melyek jelentést hordoznak a fordítóprogram számára. Az A függelék („A C nyelv összefoglalása”) felsorolja az összes ilyen foglalt kulcsszót.

Ne feledjük, hogy a kis- és nagybetűk eltérőnek számítanak a C nyelvben. Ily módon például a sum, a Sum, és a SUM változók nem azonosak. A változónevek tetszőleges hosszúságúak lehetnek, ám a fordítóprogram csak az első 63 karaktert veszi figyelembe az azonosításukkor (sőt, az A függelékben jelzett speciális esetekben csak az első 31 karakter a mérvadó). A gyakorlatban nem célszerű túl hosszú változóneveket használni, mert ez feleslegesen sok gépelést igényel. Bár a következő két értékkedés szerepe azonos, a második változat sokkal rövidebb, miközben lényegében ugyanazt az információt hordozza:

```
theAmountOfMoneyWeMadeThisYear = theAmountOfMoneyLeftAttheEndOfTheYear -
                                theAmountOfMoneyAtTheStartOfTheYear;

moneyMadeThisYear = moneyAtEnd - moneyAtStart;
```

Könnyen belátható, hogy érdemesebb a második, helytakarékos változatot használni.

A változónevek megadásakor tartsuk szem előtt azt a közmanzást, mely szerint a rest két-szer fárad! Vegyük a fáradságot olyan név kitalálására, amely megfelelően tükrözi az adott változó szerepét. A magyarázat nyilvánvaló. Ahogy az ésszerű megjegyzések írásával, úgy a változónevek megfelelő megválasztásával is sokat javíthatunk programunk olvashatóságán, érthetőségén – és így a minőségén. Az erre fordított munka megtérül a hibakeresés-kor és a dokumentáláskor. Némi túlzással fogalmazhatunk akár úgy is, hogy egy magától értetődő programot alig kell dokumentálni.

Adattípusok, konstansok

A legalapvetőbb egész típussal, az int-tel már találkoztunk a korábbiakban. Megtanultuk az előző fejezetben, hogy az int típusuként deklarált változókban csak egész értékeket tárolhatunk (tehát például tizedestörteket nem).

A C programozási nyelv további négy alapvető adattípust különböztet meg: a float, a double, a char, és az _Bool típust. A float típusuként deklarált változók lebegőpontos számokat (tizedestörteket) tárolhatnak. A double típus megegyezik a float-tal, azzal a különbséggel, hogy az így tárolt szám körülbelül kétszer annyi számjegy pontosságú. A char adattípus egy karaktert (mint amilyen például az „a” betű, a „6” számjegy vagy egy pontosvessző) tud tárolni. (Ezt hamarosan részletesen megvizsgáljuk.) Végül, az _Bool típusú változók csak 0 vagy 1 értéket vehetnek fel. Az ilyen típusú változókban valamilyen egyszerű választást: „igaz/hamis”, „igen/nem”, „be/ki” jellegű információt szoktunk tárolni.

A C nyelvben minden szám, karakter vagy karakterlánc konstansként tárolódik. Az 58 számérték például egy konstans egész szám, a "Programozni jó\n" egy konstans karakterlánc. Az olyan kifejezéseket, melyekben csak konstansok szerepelnek, konstans kifejezésnek hívjuk. Ilyen például a következő konstans kifejezés, mivel az összevonás minden tagja konstans:

128 + 7 - 17

Ha azonban az egésznek deklarált i változóval írjuk fel ugyanezt a kifejezést ez már nem konstans kifejezés:

128 + 7 - i

Az int egész típus

A C nyelvben az egész konstansokat számjegyek sorozatával adhatjuk meg. A mínusz előjel értelemszerűen a negatív értékekre utal. A 158, -10 és a 0 tehát érvényes egész konstansok. Sem szóköz, sem a tízezresekkel elválasztó vessző vagy tizedespont nem írható a számjegyek közé (azaz 12 000 vagy 12,000 helyett 12000 írandó).

Lehetőség van viszont a tízestől eltérő számrendszer használatára. Ha az egész szám első számjegye 0, akkor nyolcas (oktális) számrendszerben értelmeződik a szám. Emiatt a kezdő nullát követő számjegyek csak a 0, 1, 2, 3, 4, 5, 6, 7 közül kerülhetnek ki. Például a negyvenet (= 5·8) 050-ként tudjuk felírni a nyolcas számrendszerben. Az oktális 0177 szám tízes számrendszerben 127, hiszen $1 \cdot 64 + 7 \cdot 8 + 7 \cdot 1 = 127$. Egy tízes számrendszerbeli szám oktális értékét úgy is megtudhatjuk, hogy a számot a printf utasítással, a % formátumjelző karakterlánc megadásával íratjuk ki a terminálablakba. Ilyenkor vezető 0 nélkül jelenik meg a számérték. Ha formátumjelző karakterláncként %#o-t adunk meg, akkor a vezető 0 is megjelenik.

Tizenhatos (hexadecimális) számrendszerben úgy adhatunk meg számokat, hogy a kezdő nulla után még egy (kis vagy nagy) x betűt is írunk. Ezt követően hexadecimális „számjegyek” következhetnek, azaz a 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E. A kilences utáni betűk (melyek akár kisbetűk is lehetnek) rendre a 10, 11, 12, 13, 14, 15 számértékeket jelölik, egyetlen karakter formájában. Ha például az `rgbColor` egész változóhoz szeretnénk hozzárendelni a FFEF0D hexadecimális színértéket, azt a következő formában tehetjük meg:

```
rgbColor = 0xFFEF0D;
```

Egy tízes számrendszerbeli szám hexadecimális értékét úgy jeleníthetjük meg, hogy a számot a `printf` utasítással, a %x formátumjelző karakterlánc megadásával íratjuk ki. Ilyenkor vezető 0 nélkül (és kisbetűs a-f hexadecimális számjegyekkel) jelenik meg a számértek. Ha formátumjelző karakterláncként %#x-t adunk meg, akkor a vezető 0 is megjelenik, amit az alábbi kódrészlettel is kipróbálhatunk:

```
printf ("Color is %#x\n", rgbColor);
```

Ha nagybetűs hexadecimális számjegyekkel szeretnénk kiíratni a számot, akkor használunk nagybetűs %X (vagy %#X) formátumjelző karakterláncot a `printf` utasításban.

A tárolt számok kiterjedése és mérethatárai

Minden tárolt (számszerű vagy szöveges) értékhez tartozik egy adott kiterjedés is, mely a memoriában elfoglalt helyet jelenti. Ez nagyban függ az adat típusától, de konkrét méretek nincsenek meghatározva a nyelv definíciójában. Az adatok tényleges kiterjedése függ az adott számítógéptől és a fordítóprogramtól is. Egy egész szám kiterjedése lehet 32 bit vagy akár 64 bit is. Nem érdemes azonban olyan programot írni, amely az adott típus minimálisan használható kiterjedésénél nagyobb számokkal dolgozik. Ezeknek a minimálisan használható kiterjedéseknek a definíciója szerepel a C nyelv leírásában, és könyvünk A függetléniben (az A.4 táblázatban) meg is található. E szerint az egészek legalább 32 bitnyi kiterjedéssel rendelkeznek; sok számítógépen ezt a mennyiséget tekintik „szóhossznak”.

A float lebegőpontos típus

Tizedestörteket is tartalmazó számok tárolására használható a `float` típus. Ha egy lebegőpontos konstanst szeretnénk megadni, azt a tizedesponttal jelezhetjük. A tizedespont előtt vagy után elhagyhatóak a számjegyek, de maga a tizedespont nem maradhat el, ha lebegőpontos típust kívánunk használni. Íme néhány példa:

```
3.  
125.8  
-.0001
```

A lebegőpontos értékek alapértelmezett kiírásához a %f formátumjelző karakterláncot használhatjuk a printf utasításban.

A lebegőpontos számok normálalakban is kiírathatóak. Az 1.7e4 jelenti az $1.7 \cdot 10^4$ -t. Az e karakter előtti részt hívjuk mantissának (*törtrésznek*), az e karakter utáni számérték pedig a karakterisztika (*kitevő*), mely kaphat + vagy – előjelet is. Ez utóbbival adható meg, hogy tiznek milyen hatványával kell szorozni a mantissát (azaz a tizedespontot merre és mennyivel kell elmozdítani). Így például a 2.25e-3 esetében a 2.25-öt (a mantissát) 10^{-3} -nal kell szorozni (vagyis -3 a karakterisztika, így a tizedespontot hárommal kell balra tolni). A fenti konstans tehát 0.00225-tel egyenlő. A mantissát és karakterisztikát elválasztó e betűt írhatjuk nagy E betűvel is.

A lebegőpontos értékek normálalakban történő kiírásához %e formátumjelző karakterláncot használhatunk a printf utasításban. Ha %g formátumjelzőt adunk meg, akkor ezzel a programra bízzuk, hogy alapértelmezett vagy normálalakú formátumban írja ki az átadott lebegőpontos számértéket. Ez a gépi döntés persze nem önkényes, hanem a karakterisztikától függ: -3 és 4 közötti kitevő esetén alapértelmezett (%f szerinti) kiírás történik, egyébként pedig normálalakban (%e formátumban) jelenik meg az érték.

Érdemes tehát a – legesztétikusabb kimenetet eredményező – %g formátumjelző karakterláncot használni a lebegőpontos számok megjelenítéséhez.

Lehetőség van hexadecimális lebegőpontos számok megadására is. A vezető 0x (esetleg 0X) után következhet egy vagy több hexadecimális számjegy (köztük egy „tizenhatodos-pont”), majd a p (vagy P) karakter után a kettes (!) kitevő előjeles értéke. A 0x0.3p10 például $3 \cdot 16^{-1} \cdot 2^{10}$ -t jelent, vagyis 192-t.

A nagypontosságú double lebegőpontos típus

A double alapvetően a float-hoz hasonlít. Ott szokás használni, ahol a float által ábrázolható számok tartománya nem elegendő. A double-ként deklarált változók nagyjából kétszer annyi értékes számjegyet tudnak tárolni, mint a float típusúak. A legtöbb számítógép 64 biten ábrázolja a double típusú változókat.

Hacsak másként nem rendelkezünk, a C fordítóprogramok minden lebegőpontos számot double-ként szoktak tárolni. Ha kifejezetten csak float típusúként szeretnénk használni egy konstanst, akkor illesszünk a szám végére egy f (vagy F) betűt:

12.5f

A double típusú változók kiírásához ugyanúgy használhatóak a %e, %f, és %g formátumjelző karakterláncok, mint a float típusú változók esetében.

A char karakter típus

A char típusú változók egyetlen karaktert tudnak tárolni.¹

Karakterkonstansokat aposztrófok közé zárva adhatunk meg: ilyenek például az 'a' ('a betű), a ';' ('pontosvessző') és a '0' ('a nulla számjegy'). Ez utóbbi nem azonos a nulla számértékkel. A normál aposztrófok (') közé zárt, egyetlen karakterből álló konstansok nem tévesztendőek össze a karakterláncokkal, melyek idézőjelek ("") közt állnak.

Az újsor karakter látszólag nem felel meg a fenti kritériumnak, pedig ez is teljesen helyes karakterkonstans: '\n'. A visszaperjel (backslash) karakternek ugyanis speciális jelentése van a C nyelvben; nem számít önálló karakternek. A C fordító a '\n'-t egyetlen karakter-ként kezeli, annak ellenére, hogy az írásképe két karakterből áll. Vannak más speciális karakterek is, melyekről az A függelékben kaphatunk teljes körű tájékoztatást.

A char típusú változók printf-fel történő kiíratásához %c formátumjelző karakterláncot használhatunk.

A logikai adattípus: _Bool

Az _Bool adattípust úgy hozták létre, hogy csak a 0 és az 1 értéket tudja tárolni. A ténylegesen lefoglalando memóriaméret nincs pontosan meghatározva. _Bool típusú változókat akkor használunk egy programban, ha kétértékű logikai döntéseket kell tárolnunk. Jelölhetjük vele például azt, hogy egy állományból sikerült-e kiolvasnunk az összes adatot.

A közmegegyezés szerint a 0 jelzi a hamis, az 1 az igaz értéket. Ha értéket rendelünk egy _Bool típusú változóhoz, akkor a nulla szám ténylegesen 0-ként, bármely nullától különböző szám pedig 1-ként lesz tárolva.

A <stdbool.h> szabványos fejlécállomány megkönnyíti a munkát a logikai adattípussal azáltal, hogy definiálja a bool, a true és a false értékeket. A 6. fejezet 6.10A Listájában találhatunk erre majd egy példát.

A 4.1 Lista bemutatja a C nyelv alapvető adattípusainak használatát.

4.1 Lista • Az alapvető adattípusok használata

```
#include <stdio.h>

int main (void)
{
    int      integerVar = 100;
    float   floatingVar = 331.79;
```

¹ Az A függeléken bővebb információt találhatunk a kiterjesztett karakterkészletek karaktereinek tárolási lehetőségeiről, valamint a különféle vezérlősorozatokról („escape sequences”), az univerzális és széles karakterekről.

```

double  doubleVar = 8.44e+11;
char    charVar = 'W';

_Bool   boolVar = 0;

printf ("integerVar = %i\n", integerVar);
printf ("floatingVar = %f\n", floatingVar);
printf ("doubleVar = %e\n", doubleVar);
printf ("doubleVar = %g\n", doubleVar);
printf ("charVar = %c\n", charVar);

printf ("boolVar = %i\n", boolVar);

return 0;
}

```

4.1 Lista • Kimenet

```

integerVar = 100
floatingVar = 331.790009
doubleVar = 8.440000e+11
doubleVar = 8.44e+11
charVar = W
boolVar = 0;

```

A 4.1 Lista első utasítása egész változóként deklarálja az `integerVar` változót, és azonnal hozzá is rendeli a 100 kezdőértéket, mintha csak az alábbi két sort használnánk:

```

int integerVar;
integerVar = 100;

```

A program kimenetében láthatjuk, hogy a `floatingVar` változóhoz rendelt 331.79 lebegőpontos érték 331.790009-ként jelenik meg, sőt az éppen használt számítógéptől fügően ez akár más is lehet. A pontatlanság gyökere a számábrázolás módjában rejlik. Bizonyára tapasztalt már az olvasó hasonló pontatlanságot az egyszerűbb zsebszámológépek esetén is. Ha 1-et osztunk 3-mal, akkor a gép 0.33333333-at ír ki, esetleg néhány további hármaszt még őriz a memoriájában. Ezzel a sok hármasossal ábrázolja a zsebszámológép az egyharmadot. Elméletileg végtelen sok hármaszt kellene őriznie, ám egy gép csak véges sok számjegyet tud tárolni. Itt is hasonló pontatlanság lép fel: vannak olyan lebegőpontos számértékek, melyeket (ilyen módszerrel) nem lehet pontosan ábrázolni a számítógép memoriájában.

A `float` és a `double` típusú változók többféleképpen kiírathatóak. A `%f` segítségével valósul meg az alapértelmezett lebegőpontos kiírási formátum. Hacsak másként nem rendelkezünk, a `printf` hat tizedesjegyre kerekítve írja ki a lebegőpontos számokat. A fejezet kézőbbi részében még szó lesz arról, hogyan lehet megadni a kiírandó tizedesjegyek számát.

A float és a double típusú változókat a %e segítségével normálalakban lehet megjeleníteni; itt is hat tizedesjegy jelenik meg alapértelmezésként.

A %g hatására a program dönti el, hogy a %e vagy %f formátumban jelenjenek meg az értékek, mégpedig a szám elején álló nullák nélkül. Ha egész számról van szó, akkor nem jelennek meg tizedesjegyek.

Az utolsó előtti printf utasításban a %c-vel jelenítjük meg a charVar változóban tárolt 'W'-t. Ne feledjük, hogy az idézőjelek között megadott szövegeket (a printf első paraméteréhez hasonlóan) karakterláncnak tekintjük; a karakterkonstansokat ellenben aposztrófok közé kell zárni.

Az utolsó printf utasítás egy _Bool változót ír ki, mégpedig az egészknél használandó %i formátumjelző karakterlánc megadásával.

A long, long long, short, unsigned és signed típusmódosítók

Ha egy egész változó deklarációjában az int kulcsszó előtt long (hosszú) típusmódosító áll, akkor a változó által ábrázolható számtartomány (a legtöbb számítógépen) nagyobb lesz. Érdemes például nagy számokra is felkészíteni a factorial változót a long int típus megadásával:

```
long int factorial;
```

Ennek hatására nagy egész számkként lesz számon tartva a factorial változó. Ahogy a float és a double típusok esetében már említettük, a long változók memóriaoglalása nagyban függ a konkrétan használt számítógéptől. Sok esetben 32 bit áll rendelkezésre mind az int, mind a long int típusú változók számára, azaz a maximálisan ábrázolható szám $2^{31} - 1 = 2147483647$.

A long int típusú konstansok megadásához használhatunk egy (kis vagy nagy) L karaktert, melyet (szóköz nélkül) a szám végéhez kell illeszteni. Így tehát a következő deklaráció hatására a numberOfPoints változó típusa long int lesz, 131071100 kezdőértékkel:

```
long int numberOfPoints = 131071100L;
```

A long int típusú változókat is ki tudjuk íratni a printf segítségével. Az egészek számára fenntartott i, o, vagy x formátumjelző karakter elő (a % jel után) egy kis 1 betű írandó. A %li hatására tehát tízes számrendszerben, %lo hatására nyolcas számrendszerben, %lx hatására pedig hexadecimálisan jelenik meg az adott számérték.

Lehetséges a lehető legnagyobb méretet is kérni, a long long egész adattípussal:

```
long long int maxAllowedStorage;
```

Ennek hatására legalább 64 bitet lefoglal a fordítóprogram a maxAllowedStorage változó számára. Az ilyen változók kiíratásához dupla 1 szükséges a printf formátumjelző karakterláncába, például: %lli.

A long típusmódosító használható a double típus előtt is, a következőképp:

```
long double US_deficit_2004;
```

A long double konstansként megadandó számérték végére 1 vagy L betű írandó:

```
1.234e+7L
```

A long double típusú változók printf-fel történő kiíratásához a nagy L módosító karakter írandó. A %Lf hatására tehát alapértelmezett lebegőpontos megjelenítés jön létre, a %Le hatására a normálalak jelenik meg, %Lg hatására pedig a programra lesz bízva, hogy válasszon a %Lf vagy %Le közül.

Az int előre helyezett short típusmódosító hatására tudomásul veszi a fordítóprogram, hogy egészen kis számokkal kell csak dolgoznia. Ennek akkor lehet értelme, ha egy kifejezetten erőforrás-igényes program esetén takarékoskodni szeretnénk a változók számára lefoglalandó (ám szűkösen rendelkezésre álló) memoriával.

Néhány számítógépen a short int feleannyi memóriát foglal le, mint amennyit a normál int típus, az azonban garantált, hogy legalább 16 bit rendelkezésre fog állni a short int típusú változók számára is.

Arra nincs mód a C nyelvben, hogy egy konstans értékét kifejezetten short int típusú-ként adjuk meg. A megfelelő printf-es kiírás azonban támogatott, és az ehhez használendő módosító karakter a h. Azaz egy egész szám kívánt számrendszerbeli megjelenítéshez használható a %hi, %ho vagy %hx formátumjelző karakterlánc. Sőt, bármilyen egész átalakítás kérhető, ugyanis a printf parancs a short int típusú változókat normál egészekké alakítja a kiírás előtt.

Már csak egyetlen típusmódosítóról kell beszélnünk, melyet akkor érdemes használni, amikor előre tudjuk, hogy az adott változó csak pozitív értéket vehet föl. Ekkor a változót az unsigned típusmódosítóval deklaráljuk. A következő deklarációval a fordítóprogram tudtára adjuk, hogy a counter (számláló) egész változó nem vesz föl negatív értéket:

```
unsigned int counter;
```

Ennek következményeképp szélesebb (pozitív) tartományból kaphat értéket az így megadott változó.

Az `unsigned int` típusú konstansokat úgy adhatunk meg, hogy (kis vagy nagy) u betűvel zárjuk le az így megjelölendő számot:

`0x00ffU`

Ez nem zára ki az 1 (vagy L) méretnövelő módosító használatát, azaz a következő módon is megadhatunk egy számértéket:

`20000UL`

Ez a fordítóprogram számára világossá teszi, hogy egy hosszú, előjel nélküli egészről (`unsigned long`) van szó.

Még ha nem is írunk u/U, l/L jelölőt egy szám után, de az olyan nagy, hogy csak akkor férne el egy változóban, ha azt hosszú, előjel nélküli egészként deklaráljuk, akkor a fordítóprogram automatikusan ilyennek veszi. Hasonlóan ha egy érték olyan kicsi (negatív), hogy nem fér bele az egyszeres pontosságú számok értelmezési tartományába, akkor `long int`-ként kerül be a programba. Ha egy `long int`-nek deklarálandó szám mérete meghaladja az adott típus értelmezési tartományát, ám belefér az `unsigned long int` tartományába, akkor így lesz deklarálva. Ha ebbe sem fér bele, akkor a fordítóprogram megkísérli `long long int`-ként felfogni. Ha pedig még ehhez is túl nagy a szám, akkor `unsigned long long int` típusúnak próbálja meg betudni az adott értéket.

Ha egy változót `long long int`, `long int`, `short int` vagy `unsigned int` típusúnak szeretnénk deklarálni, akkor az `int` kulcsszó (mint alapértelmezett típus) elhagyható. Ennek megfelelően a fent említett előjel nélküli számláló így is deklarálható:

`unsigned counter;`

Némiképp meglepő módon a `char` típust is lehet `unsigned`-ként deklarálni.

A `signed` típusmódosítóval arra hívjuk fel a fordítóprogram figyelmét, hogy a szóban forgó változó előjeles szám lesz. Ennek különösen a `char` típus esetében lesz szerepe, melyet a 14. fejezetben („Mélyebben az adattípusokról”) vizsgálunk meg alaposabban.

Senki ne csüggédjen, ha e pillanatban kissé kaotikusnak tűnik a típusmódosítók rendszere. Könyünk későbbi programjaiban számos példával szemléltetjük ezek használatát. A 14. fejezet pedig majd elmélyíti az adattípusokra és az adatátalakításra vonatkozó ismereteket.

A 14.1 Táblázat összefoglalja az alapvető adattípusokat és típusmódosítókat.

14.1 táblázat • Alapvető adattípusok

Típus	Konstans példák	printf formátumjelzés
char	'a', '\n'	%c
_Bool	0, 1	%i, %u
short int	-	%hi, %hx, %ho
unsigned short int	-	%hu, %hx, %ho
int	12, -97, 0xFFE0, 0177	%i, %x, %o
unsigned int	12u, 100U, 0xFFu	%u, %x, %o
long int	12L, -2001, 0xfffffL	%li, %lx, %lo
unsigned long int	12UL, 100ul, 0xffffeUL	%lu, %lx, %lo
long long int	0xe5e5e5e5LL, 50011	%lli, %llx, %lio
unsigned long long int	12ull, 0xfffeeULL	%llu, %llx, %lio
float	12.34f, 3.1e-5f, 0x1.5p10, 0x1P-1	%f, %e, %g, %a
double	12.34, 3.1e-5, 0x.1p3	%f, %e, %g, %a
long double	12.341, 3.1e-51	%Lf, \$Le, %Lg

Aritmetikai kifejezések

A többi programnyelvhez hasonlóan C-ben is a + jellet lehet két számot összeadni, a - jellet pedig kivonni egymásból. A * a szorzás, a / az osztás jele. Ezeket kétoperandusú műveleti jeleknek (operátoroknak) hívjuk, mivel két értékkel (operandussal) történik a műveletvégzés.

Az előző fejezetben egy egyszerű műveletet végrehajtó C programot láttunk, amely összeadt két számot. A 4.2 Listában az összeadáson kívül kivonásokat, szorzásokat és osztásokat végzünk el. Egy új fogalommal is megismerkedünk: a műveletek kiértékelési sorrendjével (a *precedenciával*). A C nyelvben minden művelethez tartozik egy kiértékelési szint (precedencia). Így a több műveletet is tartalmazó kifejezésekben automatikusan eldönthető a műveletek végrehajtási sorrendje. A magasabb kiértékelési szintű műveleteket kell először kiértékelni. Ha azonos kiértékelési szintű műveletek is vannak, akkor azokat (az operátorról függően) balról jobbra vagy jobbról balra haladva értékeli ki a program. Ezt az irányjellemzőt hívjuk az operátor társíthatósági (asszociativitási) szabályának. Az A függetlenben megtalálható az összes művelet kiértékelési szintje és társíthatósági szabálya.

4.2 Lista • Aritmetikai operátorok használata

```
//Aritmetikai operátorok használata
```

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

```

int main (void)
{
    int a = 100;
    int b = 2;
    int c = 25;
    int d = 4;
    int result;

    result = a - b;           // kivonás
    printf ("a - b = %i\n", result);

    result = b * c;           // szorzás
    printf ("b * c = %i\n", result);

    result = a / c;           // osztás
    printf ("a / c = %i\n", result);

    result = a + b * c;       // kiértékelési sorrend
    printf ("a + b * c = %i\n", result);

    printf ("a * b + c * d = %i\n", a * b + c * d);
    return 0;
}

```

4.2 Lista • Kimenet

```

a - b = 98
b * c = 50
a / c = 4
a + b * c = 150
a * b + c * d = 300

```

Az a, b, c, d és result egész változók deklarációja után a program a result (*eredmény*) változónak értékül adja a és b különbségét, majd meg is jeleníti az eredményt egy megfelelően formázott printf utasítással.

A következő művelet b és c szorzatát tölti a result változóba:

```
result = b * c;
```

Az eredményt ismét a jól ismert printf utasítással jelenítjük meg.

A program következő utasítása az osztást mutatja be, melyet a perjellel végezhetünk el. 100-ban a 25 négyszer van meg, ami meg is jelenik az a / c művelet elvégzése után.

Egyes számítógépeken a nullával való osztás a program futásának rendellenes leállását idézi elő.²

Ha ez nem is következik be, az efféle osztás eredménye értelmetlen lesz.

A 6. fejezetben majd megvizsgáljuk, hogy mi a módja annak, hogy a programot még az osztás előtt felkészítsük erre az eshetőségre. Az osztó ellenőrzésével elkerülhető a helytelen osztás.

A következő kifejezés eredménye nem 2550 (azaz nem 102·25):

`a + b * c`

A `printf` kimenete szerint a kapott érték 150. Más nyelvekhez hasonlóan a C-ben is van a műveleteknek előre meghatározott precedenciája, vagyis végrehajtási sorrendje. A hasonló jellegű műveleteket balról jobbra haladva hajtja végre a program, a szorzás és az osztás azonban magasabb precedenciával rendelkezik, mint az összeadás és a kivonás. Emiatt a fenti műveletet (amúgy az algebra alapvető szabályaival teljes összhangban) a következőképpen értelmezi a C fordítóprogram:

`a + (b * c)`

Zárójelekkel jelezhetjük, ha más sorrendben szeretnénk végrehajtani a műveleteket, azaz a most leírt kifejezést akár használhatnánk is a 4.2 Listában, hiszen ugyanaz lenne az eredménye, mint az eredeti műveletnek:

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

Ellenben a

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

utasítás már tényleg 2550-et rendelne a `result` változóhoz, hiszen a program így a `a` és `b` összegét (102) szorozza meg 25-tel. A zárójelek egymásba ágyazhatók; ilyenkor a legbelőző zárójelpár értékelődik ki először, majd fokozatosan halad kifelé a kifejezés értelmezése. Csak arra kell ügyelnünk, hogy ne tévesszük el a megnyitott és bezárt zárójelek számát!

² Ez a Windows-os gcc-vel fordított programoknál fordul elő. Unix rendszereken a program nem ér véget rendellenesen. 0 értéket eredményez a nullával való egész osztás, és „végtelent” a nullával való lebegőponos osztás.

A 4.2 Lista utolsó utasításából kiderül, hogy a `printf` akkor is helyesen működik, ha közvetlenül a zárójelbeli második paraméter helyén végezzük el a műveletet, és nem tároljuk el az eredményt egy újabb változóban. Az

```
a * b + c * d
```

kifejezés kiértékelése a korábbiakban vázolt szabályok szerint zajlik, azaz a következőképpen:

$$(a * b) + (c * d)$$

Ez numerikusan a következőnek felel meg:

$$(100 * 2) + (25 * 4)$$

Az eredményül kapott 300-at jeleníti meg a `printf`.

Egész aritmetika és az egyoperandusú mínusz operátor

A 4.3 Listában bemutatott programban elmélyítjük az eddig tanultakat, és bevezetjük az olvasót az egész aritmetikába.

4.3 Lista • További példák aritmetikai operátorok használatára

```
//További aritmetikai operátorok
```

```
#include <stdio.h>

int main (void)
{
    int    a = 25;
    int    b = 2;

    float c = 25.0;
    float d = 2.0;

    printf ("6 + a / 5 * b = %i\n", 6 + a / 5 * b);
    printf ("a / b * b = %i\n", a / b * b);
    printf ("c / d * d = %f\n", c / d * d);
    printf ("-a = %i\n", -a);

    return 0;
}
```

4.3 Lista • Kimenet

```
6 + a / 5 * b = 16
a / b * b = 24
c / d * d = 25.000000
-a = -25
```

A program olvashatóságának javítása érdekében szóközökkel egy vonalba igazítottuk az a, b, c és d változók betűjeleit a deklarációban. Bizonyára azt is észlelte az olvasó, hogy a kétoperandusú aritmetikai operátorok előtt és után szóközt írtunk. Mindez nem kötelező, csak esztétikai okokból történt így az eddig bemutatott programokban. Általában is igaz, hogy ahova egy szóköz írható, oda többet is szabad írni. Megéri a fáradtságot, ha javítja az olvashatóságot.

A 4.3 Lista első `printf` utasítása jól rávilágít az operátorok kiértékelési sorrendjének mélyebb rétegeire. A műveletek végrehajtása a következőképpen történik:

- 1 Mivel az osztás magasabb rendű, mint az összeadás, az a értékét (azaz 25-öt) először osztjuk 5-tel. Ez az 5 részeredményt adja.
- 2 Mivel a szorzás is magasabb rendű, mint az összeadás, az 5 részeredményt b-vel (2-vel) szorozzuk. Az eredmény 10.
- 3 Utolsó lépésként hajtódi a 6 és a (most kapott) 10 összeadására, ami végül 16-ot eredményez.

A második `printf` egy újabb meglepetést hoz. Hiszen ki gondolná, hogy ha 25-öt elosztjuk b-vel, majd az eredményt újra megszorzunk b-vel, akkor az nem 25-öt, hanem 24-et ad? Úgy tűnik, mintha a számítógép menet közben elvesztett volna valamit. A jelenség magyarázata az egész aritmetikában rejlik.

Ha megnézzük a program elejét, látszik, hogy a és b egészként lett deklarálva. Ha egy kifejezés egész számokat tartalmaz, akkor a művelet végeredményét is ebből a halmazból választja ki a program. Azaz az osztáskor keletkező törtrészt nem veszi tekintetbe. Emiatt a és b (25 és 2) hányadosa 12-t ad, nem pedig 12.5-et. Ezt a részeredményt pedig 2-vel szorozva tényleg 24-et kapunk. Így „tűnik el” a hiányzó rész. Tartsuk tehát minden szem előtt, hogy egészek osztásakor egész végeredményt kapunk.

A utolsó előtti `printf` utasításból látszik, hogy ugyanezt a műveletet lebegőpontos számokkal is elvégezhetjük, és akkor már a várakozásnak megfelelő végeredményt kapunk.

A célnak megfelelően kell tehát megválasztanunk, hogy egész vagy lebegőpontos típusú változókkal dolgozunk. Ha biztosan soha nem kerülhetnek elő törtszámok, használunk egészeket. Ettől a program hatékonyabb, gyorsabb lesz. Ha azonban a későbbiekben szükség lesz tizedesjegyekre is, akkor a döntés egyértelmű: lebegőpontos értékeket kell

használnunk. Persze még ekkor is megfontolandó, hogy float, double vagy long double típust válasszunk-e. A kérdést az döntheti el, hogy mennyire fontos a számértékek pontossága, illetve mennyire kerülnek elő nagy (abszolút értékű) számok.

Az utolsó `printf` utasításban az egyoperandusú mínusz operátort láthatjuk működés közben: `-a` értéke értelemszerűen a ellentettje lesz. Ennek az operátornak a precedenciája magasabb, mint bármely más aritmetikai műveleté (kivéve az azonos szintű egyoperandusú plusz operátort). Így a következő művelet a várakozásnak megfelelően `-a` és `b` szorzatát adja:

```
c = -a * b;
```

Az A függelékben megtalálható a műveletek és azok kiértékelési sorrendjének teljes listája.

A modulus operátor

A fejezet következő részében bevezetjük a maradékos osztást. Megismerkedünk a maradékok kiszámításával, azaz a modulusképzéssel, melynek műveleti jele a százalék (%). A dolog működését a 4.4 Lista alapján érhetjük meg.

4.4 Lista • A modulus operátor bemutatása

```
// A modulus operátor

#include <stdio.h>

int main (void)
{
    int a = 25, b = 5, c = 10, d = 7;

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

    return 0;
}
```

4.4 Lista • Kimenet

```
a % b = 0
a % c = 5
a % d = 4
a / d * d + a % d = 25
```

A `main` első utasítása egyetlen lépésben deklarálja és inicializálja az `a`, `b`, `c` és `d` változókat.

Amint azt korábbról már tudjuk, a `printf` utasítás a `%` jelet használja a formátum jelzésére: a `%` jelet követő karakterből derül ki, hogy milyen formátumban kell megjelenítenie a második paraméter értékét. Ha azonban a `%` jelet egy újabb `%` jel követi, akkor tényleg egy `%` karaktert ír ki az adott helyen a program.

Mint a kimenetből is látszik, valóban az osztási maradékokat számítja ki a program a `%` operátor hatására. 25-ben az 5 maradék nélkül megvan, ezért az eredmény 0. 25-ben a 10 megvan kétszer, maradék az 5, amit meg is jelenít a `printf` a kimenet második sorában. A 25-öt 7-tel osztva (a hányados 3), a maradék 4, ami a harmadik kimeneti sorban látszik.

A kimenet utolsó sorát érdemes alaposabban szemügyre vennünk. Először is az tűnhet fel, hogy két sorra bontottuk az utasítás kódját. Ez teljesen korrekt. A C nyelvben bárhol beilleszthető sortörés, ahol szóköz is használható lenne. (Kivételt a karakterláncok témajában találunk, de arra még visszatérünk a 10. fejezetben.) Esetenként nemcsak kívánatos, de elkerülhetetlen is lehet, hogy egy utasítást átvigyük a következő sorba. A 4.4 Listában csak a szép megjelenés indokolja a sortörést: így ugyanis jobban látszik az utasítás két paraméterének összhangja.

Nézzük tehát a kiértékelendő kifejezést. Egész értékek közti művelet eredménye a C nyelvben mindenig egész. Ezért az osztás hányadosa is egész lesz, az esetleges maradék nem számít az egész osztáskor. 25-ben a 7 megvan 3-szor. Ezt az értéket az osztóval (7-tel) szorozva 21-et kapunk. Ehhez adva 25-nek 7-tel vett osztási maradékát (a `% d` értékét, vagyis 4-ét) épp az eredeti osztandót, 25-öt kapjuk vissza. Általában is igaz az, hogy a `é`s `d` egészekre az

```
a / d * d + a % d
```

kifejezés minden `a`-t adja vissza. A modulus operátor (`%`) egyébként is csak egészekre működik. Kiértékelési szintje azonos a szorzáséval és osztáséval. Emiatt az alábbi kifejezés:

```
table + value % TABLE_SIZE
```

így értékelődik ki:

```
table + (value % TABLE_SIZE)
```

Egész és lebegőpontos típusátalakítások

A C nyelvű programozáshoz meg kell értenünk az egész és lebegőpontos típusátalakítások szabályait. A 4.5 Lista bemutat néhány példát a számértékek átalakítására. Egyes fordítók figyelmeztetést is adnak, ha efféle implicit átalakításokat végeztetünk a programmal.

4.5 Lista • Egész és lebegőpontos típusok átalakításai

```
// Alapvető típusátalakítások

#include <stdio.h>

int main (void)
{
    float f1 = 123.125, f2;
    int     i1, i2 = -150;
    char   c = 'a';

    i1 = f1;           // lebegőpontos átalakítása egésszé
    printf ("%f assigned to an int produces %i\n", f1, i1);

    f1 = i2;           // egész átalakítása lebegőpontossá
    printf ("%i assigned to a float produces %f\n", i2, f1);

    f1 = i2 / 100;      // egész osztása egéssel
    printf ("%i divided by 100 produces %f\n", i2, f1);

    f2 = i2 / 100.0;      // egész osztása lebegőpontossal
    printf ("%i divided by 100.0 produces %f\n", i2, f2);

    f2 = (float) i2 / 100;    // típusátalakító (casting) operátor
    printf("(float) %i divided by 100 produces %f\n", i2, f2);

    return 0;
}
```

4.5 Lista • Kimenet

```
123.125000 assigned to an int produces 123
-150 assigned to a float produces -150.000000
-150 divided by 100 produces -1.000000
-150 divided by 100.0 produces -1.500000
(float) -150 divided by 100 produces -1.500000
```

Ha egy lebegőpontos számot egésszé alakítunk, akkor egyszerűen elmarad a szám törtrésze. Az f1 átalakítása után az 123.125 helyett csak 123 tárolódik az i1-ben. A kimenet első sora tanúsítja ezt a szabályt.

Ha egy egész számot adunk értékül egy lebegőpontos változónak, akkor nem változik meg a szám értéke, egyszerűen csak lebegőpontos típusuként lesz nyilvántartva ugyanaz a szám. A program kimenetének második sora mutatja, hogy az i2 értéke (-150) helyesen alakult át a lebegőpontos f1 változó értékévé.

A kimenet következő két sora felhívja a figyelmet két olyan mozzanatra, amit minden szem előtt kell tartani az aritmetikai kifejezések írásakor.

Az elsőről már volt szó; az egész aritmetika alapszabálya érvényesül. Egész (azaz short, unsigned, long vagy akár long long) értékekkel felépített számtoni művelet végeredménye csak egész lehet, még akkor is, ha a végeredményt egy lebegőpontos típusú változóhoz rendeljük hozzá. Példaprogramunkban is így történt: az i2 (-150) értékét a 100 konstanssal osztva az eredmény -1, amit eltárolunk a lebegőpontos f1 változóban.

A második műveletben egy egész és egy lebegőpontos változó szerepel. Ha egy számtoni műveletben szerepel akár csak egyetlen lebegőpontos változó vagy konstans, akkor ez a C nyelvben lebegőpontos műveletnek minősül. Így, amikor az i2 (-150) értékét a 100.0 konstanssal osztjuk, lebegőpontos végeredményt kapunk, ami az f1 lebegőpontos változóban tárolt -1.5 lesz.

A típusátalakító (casting) operátor

A 4.5 Lista utolsó osztási műveletében kerül elő a típusátalakító operátor:

```
f2 = (float) i2 / 100; // típusátalakító (cast) operátor
```

Hatására az i2 változó a kiértékelés idejére egyszeres pontosságú lebegőpontos típusúvá (float) alakul. Ez ugyanakkor nem érinti magát az i2 változót. A típusátalakító operátor egyoperandusú, és ennek megfelelően használható. Ahogy az ellentett-képző mínusz előjel sem változtatja meg az érintett változó értékét (- a hatására az a változó értéke nem változik ellentettjére), ugyanúgy a (float) a művelet sem hat vissza magára az a változóra.

A típusátalakító operátor kiértékelési szempontból magasabb rangú, mint bármilyen más aritmetikai művelet, kivéve az egyoperandusú mínusz és plusz műveleteket. Természetesen zárójel minden használható a kívánt műveleti sorrend eléréséhez.

Lássunk egy másik példát a típusátalakításra:

```
(int) 29.55 + (int) 21.99
```

Ez C-ben a következőképpen értékelődik ki, mivel az egésszé alakítás eltünteti az értékek törtrészét:

```
29 + 21
```

A következő két kifejezés ezzel szemben egyaránt másfelet ad eredményül:

```
(float) 6 / (float) 4
(float) 6 / 4
```

Értékadás és műveletvégzés egy lépésben

A C nyelv lehetővé teszi, hogy egy lépében történjen meg egy műveletvégzés és az eredmény értékül adása. Ennek szintaxisa a következő:

`op=`

Itt az op egy operátort jelez, amely a +, -, *, /, vagy % közül bármilyen lehet, sőt akár bitenkénti eltolás vagy maszkolás is, amit később részletezünk. Tekintsük az alábbi utasítást:

`count += 10;`

Ez a „plusz-egyenlő” operátor tehát veszi a tőle balra eső értéket, hozzáadja a jobb oldalán állót, majd az eredményt betölti a bal oldali operandusba. Ugyanaz történik tehát, mintha a következő sort hajtanánk végre:

`count = count + 10;`

A

`counter -= 5`

kifejezés hasonlóképp működik; kivon a counter-ból 5-öt, mintha ez történne:

`counter = counter - 5;`

Egy fokkal bonyolultabb az

`a /= b + c`

kifejezés. Az a értékét az egyenlőség jel jobboldalán álló kifejezéssel osztjuk el, azaz b és c összegével; a végeredmény a-ba kerül. Az összeadást még az osztás előtt elvégzi a program, mivel annak kiértékelési szintje magasabb. A visszön kívül minden más operátor magasabb kiértékelési szinten van, mint a hozzárendelő operátor, vagyis a fenti művelet egyenértékű a következővel:

`a /= (b + c)`

Három okból érdemes ilyesféle hozzárendelő operátort használni. Egyrészt egyszerűbb így megírni a program kódját, mivel a baloldali változót nem kell megismételni jobboldalt. Másrészt egyszerűbb olvasni egy ilyen módon leírt utasítást. Harmadrészt gyorsabb lesz a program futása, mivel a fordítóprogram ki tudja használni az ily módon átadott többlet-információt, és rövidebb utasítássort tud létrehozni.

Az _Complex és az _Imaginary típus

Mielőtt a fejezet végéhez érnénk, érdemes említeni a komplex számok kezelését lehetővé tevő _Complex és _Imaginary típusokról. Ezek támogatottsága fordítóprogramtól függ.³

Részletes leírás ezekről a típusokról az A függelékben található.

Gyakorlatok

1. Gépeljük be és futtassuk a fejezetben szereplő öt példaprogramot! A kimeneteket hasonlítsuk össze a könyvben szereplő programkimenetekkel.
2. Melyek érvényes változónevek az alábbiak közül? Miért?

Int	char	6_05
Calloc	Xx	alpha_beta_routine
floating	_1312	z
ReInitialize	-	A\$

3. Melyek érvényes konstansok az alábbiak közül? Miért?

123.456	0x10.5	0X0G1
0001	0xFFFF	123L
0Xab05	0L	-597.25
123.5e2	.0001	+12
98.6F	98.7U	17777s
0996	-12E-12	07777
1234uL	1.2Fe-7	15,000
1.234L	197u	100U
0XABCDEF	0xabcu	+123

4. Írunk programot, amely átszámítja a 27°-ot Fahrenheitból (F) Celsiusba (C) az alábbi képlet szerint: $C = (F - 32) / 1.8$
5. Mi lesz az alábbi program kimenete?

```
#include <stdio.h>

int main (void)
{
    char c, d;

    c = 'd';
    d = c;
    printf ("d = %c\n", d);

    return 0;
}
```

³ A 3.3-as gcc még nem kezelte minden vonatkozásban helyesen ezeket az adattípusokat, a 3.4-es gcc fordítóprogram ugyanakkor már jól működik.

6. Írunk programot, mely kiszámítja az alábbi polinom helyettesítési értékét $x=2.55$ -re: $3x^3 - 5x^2 + 6$
7. Írunk programot, mely normálalakban jeleníti meg az alábbi kifejezés értékét: $(3.31 \cdot 10^{-8} \cdot 2.01 \cdot 10^{-7}) / (7.16 \cdot 10^{-6} + 2.01 \cdot 10^{-8})$
8. Egy i egész változót a következő képlet alapján lehet egy másik egész (j) következő többszöröséhez kerekíteni:

$$\text{Next_multiple} = i + j - i \% j$$

Ha például 256 napot szeretnénk a következő számhoz kerekíteni, mely héttel osztható, akkor az $i = 256$ és a $j = 7$ értéket kell behelyettesítenünk a képletbe:

$$\begin{aligned}\text{Next_multiple} &= 256 + 7 - 256 \% 7 \\ &= 256 + 7 - 4 \\ &= 259\end{aligned}$$

– ami valóban osztható héttel ($37 \cdot 7 = 259$).

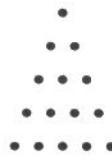
Írunk programot, amely megkeresi az alábbi i értékekhez a következő j-többszöröst:

i	j
365	7
12,258	23
996	4

5

Ciklusszervezés

Ha tizenöt pontot háromszög alakban rendezünk el, akkor az alábbi alakzat jön létre:



Az első sor egy pontot tartalmaz, a második kettőt stb. Az n sorból álló háromszöghöz szükséges pontok száma: az 1-től n -ig terjedő egész számok összege. Az így előálló számokat hívják háromszögszámoknak. A negyedik háromszögszám nem más, mint $1 + 2 + 3 + 4 = 10$.

Írunk programot, amely kiszámítja és kiírja a nyolcadik háromszögszámot! Ezt persze fejben is könnyedén ki lehet számítani, de azért a példa kedvéért írunk most a feladathoz egy programot C-ben. Az 5.1 Lista egy lehetséges megoldást mutat.

A program „működési elve” kisebb számoknál alkalmazható ugyan, de mihez kezdünk akkor, ha teszem azt a kétszázadik háromszögszámot szeretnék megkeresni? Elég nyögenyelős lenne az 5.1 Listát módosítani a megoldás megkereséséhez. Szerencsére van enél egyszerűbb út is.

5.1 Lista • A nyolcadik háromszögszám kiszámítása

```
//A nyolcadik háromszögszám kiszámítását elvégző program
```

```
#include <stdio.h>

int main (void)
{
    int triangularNumber;

    triangularNumber = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;
```

```

    printf ("A nyolcadik hároszögszám a %i\n",
            triangularNumber);

    return 0;
}

```

5.1 Lista • Kimenet

A nyolcadik hároszögszám a 36

A számítógép egyik leghasznosabb tulajdonsága, hogy képes egy-egy utasítássort rengetegszer megismételni anélkül, hogy elfáradna. Efféle ciklusok szervezésével tömör programokat írhatunk, melyek ismétlődő utasítássorokat hajthatnak végre. Ugyanezeket a feladatokat ciklusok nélkül csak utasítások ezreivel vagy millióival lehetne megoldani. A C programnyelvben háromféle ciklus létezik: vannak a `for`, a `while` és a `do` utasítással szerzethető ciklusok. Ezeket részletesen megvizsgáljuk ebben a fejezetben.

A `for` utasítás

Evezzünk ki a mélyre és nézzünk egyből egy `for` ciklust használó programot. Az 5.2 Lista a kétszázadik háromszögszámot számítja ki. Lássuk, megérti-e az olvasó a `for` ciklus működési elvét?

5.2 Lista • A kétszázadik háromszögszám kiszámítása

```
/* A 200. háromszögszám kiszámítását elvégző program.
   A "for" ciklus bevezetése. */
```

```

#include <stdio.h>

int main (void)
{
    int n, triangularNumber;

    triangularNumber = 0;

    for ( n = 1; n <= 200; n = n + 1 )
        triangularNumber = triangularNumber + n;

    printf ("A 200. háromszögszám a %i\n", triangularNumber);

    return 0;
}

```

5.2 Lista • Kimenet

A 200. háromszögszám a 20100

Az 5.2 Lista igényel némi magyarázatot. A kétszázanik háromszögszám kiszámítása is ugyanolyan elven történik, mint ahogy a nyolcadiké (lásd az 5.1 Listában) : összegezni kell az egész számokat 1-től 200-ig. A `for` ciklus révén mentesülünk attól a tehertől, hogy fel kelljen írni a számokat egyesével 1-től 200-ig, hiszen a ciklusszervezéssel „legyárthatjuk” ezeket a számokat.

A `for` ciklus általános alakja a következő:

```
for ( inicializáció; vizsgálat; léptetés )
    ciklusmag_utasítás
```

A zárójelben pontosvesszőkkel elválasztott három kifejezés (*inicializáció – vizsgálat – léptetés*) teremti meg a ciklus „környezetét”. A *ciklusmag_utasítás*, mely a zárójelet követi (és amit természetesen pontosvessző zár le) tetszőleges C utasítást tartalmazhat. Ez képezi a ciklusmagot. Ez az utasítás annyiszor fut le, ahányszor a `for` ciklus „környezete” ezt meghatározza.

A zárójel első utasítása az *inicializáció*. Ebben alapvetően a számláló(k) beállítása szokott szerepelni, még a ciklus megkezdése előtt. Az 5.2 Listában ezen a ponton állítjuk be az `n` értékét 1-re. Látható, hogy az értékadás érvényes kifejezésnek minősül.

A zárójel középső utasítása a *vizsgálat*, melyben a ciklusfeltétel szerepel. Addig folyik a ciklus, amíg ez igaz. Az 5.2 Listában itt az

```
n <= 200
```

relációs kifejezés áll. (Azaz folytasd, amíg `n` kisebb mint 200 vagy egyenlő vele.) A C programozási nyelvben számos hasonló relációs operátor használható, melyekkel feltétel-vizsgálat végezhető. A kiértékelés után „igaz” (azaz „true”) értéket kapunk, ha a feltétel teljesül, és „hamis” (azaz „false”) értéket, ha nem.

Relációs operátorok

Az 5.1 Táblázatban foglaltam össze a C-ben használható relációs operátorokat.

5.1 Táblázat • Relációs operátorok

Operátor	Jelentése	Példa
<code>==</code>	Egyenlő	<code>count == 10</code>
<code>!=</code>	Nem egyenlő	<code>flag != DONE</code>
<code><</code>	Kisebb	<code>a < b</code>
<code><=</code>	Kisebb vagy egyenlő	<code>low <= high</code>
<code>></code>	Nagyobb	<code>pointer > end_of_list</code>
<code>>=</code>	Nagyobb vagy egyenlő	<code>j >= 0</code>

A relációs operátorok alacsonyabb kiértékelési szinthez tartoznak, mint bármelyik aritmetikai operátor. A következő kifejezés tehát zárójelek nélkül is úgy értékelődik ki, ahogy az elvárjuk:

`a < b + c`

`a < (b + c)`

Igaz lesz tehát a kifejezés értéke, ha a kisebb b és c összegénél, egyébként pedig hamis.

Érdemes külön figyelmet szentelni az egyenlőséget vizsgáló `==` operátornak, amely szigorúan megkülönböztetendő az `=` értékadó operátortól. A

`a == 2`

kifejezés azt vizsgálja, hogy egyenlő-e az a értéke 2-vel, míg az

`a = 2`

értékül adja az a változónak a 2-t.

A relációs operátorok közül szándékainknak megfelelően többféleképpen is választhatunk. Az

`n <= 200`

kifejezés helyett használhatjuk az

`n < 201`

kifejezést is, ha biztosak vagyunk abban, hogy az n egész szám.

Visszatérve példaprogramunkra, a `for` ciklusmag-utasítása:

```
triangularNumber = triangularNumber + n;
```

mindaddig ismétlődik, míg a relációs kifejezés vizsgálata „igazat” mutat (vagyis az n kisebb vagy egyenlő 200-nál). Ennek hatására a `triangularNumber` (azaz: háromszögszám) változó értékéhez mindaddig újra és újra hozzáadódik az n aktuális értéke, amíg a ciklusfeltétel teljesül.

Amikor a ciklusfeltétel nem teljesül, akkor a vezérlés a `for` ciklust követő utasítással folytatódik. Programunkban a ciklus után a `printf` utasítás következik.

A `for` ciklusfejben levő zárójel utolsó utasítása a ciklusmag egyes lefutásai *után* hajtódik végre. Az 5.2 Listában ezen a ponton az `n` értéke egygel megnő, azaz a ciklus lépései közben (miután a `triangularNumber` változóhoz való hozzáadása megtörténik) `n` 1-től 201-ig lépdel egyesével.

Érdemes megfigyelni, hogy az `n` által elért legutolsó érték (amikor `n=201`) *nem* adódik hozzá a `triangularNumber` változóhoz. Mihelyt ugyanis a ciklusfeltétel hamisra változik (azaz amikor `n=201`), a program kilép a ciklusból, nem fut le a ciklusmag.

A következőképpen összegezhető tehát a `for` ciklus működése:

- 1 Kiértékelődik az inicializációs kifejezés, amely beállít egy ciklusváltozót általában (de nem föltétlenül) 0 vagy 1 kezdőértékre.
- 2 Lezajlik a ciklusfeltétel kifejezésének vizsgálata. Ha a kifejezés „hamis” (azaz nem teljesül a feltétel), a ciklus futása azonnal befejeződik, és a végrehajtás a ciklust követő utasítással folytatódik
- 3 A ciklusmagot képző utasítás lefut.
- 4 Kiértékelődik a léptetési kifejezés, amely általában egygel növeli (vagy csökkenti) a ciklusváltozó értékét.
- 5 Visszatérünk a 2. ponthoz, vagyis a vizsgálathoz.

Ne feledjük, hogy a ciklusfeltétel a ciklus indulásakor is kiértékelődik, még mielőtt lefutt volna a ciklusmag. Nem szabad pontosvesszőt tenni a ciklusfej zárójele után, mert az egyből befejezi a ciklus futását.

Az 5.2 Lista programja futás közben kiszámítja az első kétszáz háromszögszámot, melyeket jó volna táblázatszerűen megjeleníteni. Kímélendő az értékes helyet dolgozzunk inkább csak az első tíz háromszögszámmal az 5.3 Listában.

5.3 Lista • A háromszögszámok táblázata

// A háromszögszámok táblázatát kiszámító program

```
#include <stdio.h>

int main (void)
{
    int n, triangularNumber;

    printf ("A HÁROMSZÖGSZÁMOK TÁBLÁZATA\n\n");
    printf (" n      Összeg 1-től n-ig\n");
    printf ("---  -----\n");

    triangularNumber = 0;
```

```

for ( n = 1; n <= 10; ++n ) {
    triangularNumber += n;
    printf (" %i           %i\n", n, triangularNumber);
}

return 0;
}

```

5.3 Lista • Kimenet

A HÁROMSZÖGSZÁMOK TÁBLÁZATA

n	Összeg 1-től n-ig
1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36
9	45
10	55

Érdemes néhány `printf` utasítással értelmes kimenetet adni programjainknak. Az 5.3 Lista első három `printf` utasítása pusztán annyit tesz, hogy címet és fejlécet ad a táblázat oszlopainak. Az első `printf` két újsor karaktert is tartalmaz. Ennek hatása a várakozásnak megfelelően az lesz, hogy egy üres sor is megjelenik az első sora melé után.

A csinos fejléc kiírása után a program elkezdi kiszámolni az első tíz háromszögszámot. Az `n` változó tárolja azt az aktuális részletösszeget, amelyet „1-től `n`-ig” összegzünk, a `triangularNumber` változó pedig az `n`-edik háromszögszámot.

A `for` ciklus végrehajtásának kezdetén az `n` értékét 1-re állítjuk. Korábban azt mondhattuk, hogy a ciklusfejet követő első utasítás a ciklusmag. No de mi a teendő akkor, ha nem csak egyetlen utasítást szeretnénk ismételten végrehajtani? Nos, ilyenkor kapcsos zárójelek közé kell írni a ciklusmagnak szánt utasításokat, melyeket ezután a fordítóprogram egyetlen egységeként, utasításblokkként fog kezelni. Ez általában is igaz: a C program bármely utasítása helyett szerepelhet utasításblokk; ügyeljünk azonban arra, hogy minden kapcsos zárójelnek legyen meg a párja.

Az 5.3 Listában tehát mind a `triangularNumber` változó kiszámítása, mind pedig a `printf` utasítás a ciklusmag része, minden cikluslépésben végrehajtódik mindenkitől. Figyeljük meg, miként vannak a program sorai megformázva, beljebb kezdve. Fontos, hogy jól láthatóan elkülönüljenek a ciklusmag utasításai. Ezzel kapcsolatban többféle jól bevált programozói stílus is használható.

Van, aki a következő stílust kedveli:

```
for ( n = 1; n <= 10; ++n )
{
    triangularNumber += n;
    printf (" %i           %i\n", n, triangularNumber);
}
```

Itt a nyitó kapcsos zárójel a for ciklusfejet követő sorban áll. A döntés csupán ízlés dolga, semmi kihatása nincs a program futására.

A ciklus lépései során úgy számítjuk ki a következő háromszögszámot, hogy az előzőhez hozzáadjuk a ciklusváltozó (*n*) értékét. Példaprogramunkban a „plusz egyenlő” operátort használjuk, melyet a 4. fejezetben („Változók, adattípusok, aritmetikai kifejezések”) már megismertünk. Emlékeztetőül – az alábbi két kifejezés egyenértékű:

```
triangularNumber += n;
```

illetve

```
triangularNumber = triangularNumber + n;
```

Amikor először fut le a ciklusmag, akkor az „előző” háromszögszám értéke (a ciklus előtti értékkedásnak megfelelően) nulla. Ilyenkor a *triangularNumber* új értéke egyszerűen *n*, azaz 1 lesz. A táblázat fejlécfeliratainak megfelelően (adott számú szóközzel elválasztva) kiíratjuk az *n* és a *triangularNumber* értékét.

A ciklusmag véget ért – a léptetési utasítás következik. Esetünkben ez kissé furcsán néz ki. Mintha gépelési hiba történt volna:

```
n = n + 1
```

helyett csak egy szűkszavú

```
++n
```

utasítást látunk a programban.

A *++n* teljesen helyes C kifejezés, melyben egy új (és meglehetősen egyedülálló) operátort ismerhetünk meg, a növelő (inkrementáló) operátort. A *++* hatására eggyel megnő az operandus értéke. Szinte minden programban előfordul az „eggyel növelés”, így ehhez a művelethez külön operátort hoztak létre. Így tehát a *++n* egyenértékű az *n = n + 1* utasítással. Eleinte talán szokatlannak tűnik a *++n* alakja (az olvashatóbb *n = n + 1* változathoz képest), de hamarosan meg fogjuk szokni, sőt, idővel örömkünket leljük majd a tömör írásmód szépségében.

Ha egy programnyelvben van növelő operátor, elvárható, hogy legyen egy hasonló jellegű csökkentő (dekrementáló) operátor is, amely az operandus értékét eggyel csökkenti. Tulajdonképpen természetes, hogy létezik ez a művelet is. Szimbóluma a dupla mínusz jel. Ennek megfelelően tehát a

```
bean_counter = bean_counter - 1
```

utasítás

```
--bean_counter
```

formában is írható.

Egyes programozók szeretik a növelő és csökkentő operátort az operandus után írni: `n++`, `bean_counter--`. Ez is elfogadható. A C programnyelvben ízlés dolga, hogy melyiket használjuk.

A kimenet formába öntése

Zavaró, hogy az 5.3 Lista által kiírt háromszögszám-táblázat 10. sorában eggyel eltolódik a második oszlop. A 10-es sorszám ugyanis (ellentétben a felette állókkal) kétjegyű, így a szóközök eggyel jobbra tolják a második oszlopot. Ez a szépséghiba a következő kód részlettel orvosolható:

```
printf ("%2i %i\n", n, triangularNumber);
```

A módosított programnak (hívjuk ezt 5.3A-nak) itt láthatjuk a kimenetét:

5.3A Lista • Kimenet

A HÁROMSZÖGSZÁMOK TÁBLÁZATA

n	Összeg 1-től n-ig
1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36
9	45
10	55

A megváltoztatott sor printf utasításában mezőszélességet is megadunk. A %2i formátumjelző karakterláncjal nemcsak azt hozzuk a printf tudomására, hogy egész számot szeretnénk megjeleníteni, hanem azt is, hogy legalább két karakternyi helyet szeretnénk lefoglalni a számra. Ha a szám csak egy karakternyi helyet foglal el (0 – 9), akkor egy szóközzel egészül ki két karakteresre, méghozzá úgy, hogy az értékes számjegyet a program jobbra igazítja.

Így a %2i-vel biztosíthatjuk azt, hogy az n számára legalább két karakternyi hely lefoglalódik a kiíráskor, emiatt a háromszögszámok oszlopa egyenes lesz, nem fog megtörni.

Ha a kiírandó szám több számjegyet tartalmaz, mint amit a mezőszélességben kértünk, akkor a printf egyszerűen figyelmen kívül hagyja a kérést – annyi számjeggyel írja ki a számot, amennyit az igényel.

A mezőszélességet nemcsak egész számok esetén definiálhatjuk, ahogy azt a következő programokban is látni fogjuk.

A program bemenete

Az 5.2 Lista programja kiszámítja a 200. háromszögszámot – semmi másat nem tesz. Ha épp az ötvenedikre vagy századikra lennének kíváncsiak, ahhoz módosítani kellene a programot, hogy megfelelő számban fusson le a ciklus. A printf utasítást is korrigálni kellene, majd a futtatás előtt újra le kellene fordítani a programot.

Egyszerűbb megoldást jelentene, ha megkérdezné a program, hogy melyik háromszögszámot számítsa ki. A válasz alapján kiszámítható a kérő háromszögszám. A C scanf utasításával ez meg is valósítható. A scanf hasonló elven működik, mint a printf. Ahogy a printf arra szolgál, hogy kiírunk bizonyos értékeket a terminálablakban, a scanf segítségével behihetünk értékeket a programba. Az 5.4 Lista programja megkérdezi a felhasználót, hogy melyik háromszögszámra kíváncsi. Kiszámítja, majd kiírja a keresett értéket.

5.4 Lista • Bemeneti adat kérése a felhasználótól

```
#include <stdio.h>

int main (void)
{
    int n, number, triangularNumber;

    printf ("Hányadik háromszögszám? ");
    scanf ("%i", &number);

    triangularNumber = 0;

    for ( n = 1; n <= number; ++n )
        triangularNumber += n;
```

```

        printf ("A %i. háromszögszám a %i\n", number, triangularNumber);

    return 0;
}

```

Az 5.4 Lista kimenetében vastagon szedtük a felhasználó által begépelet értéket (100), hogy elkölönlöljön a program által kiírt kimenettől.

5.4 Lista • Kimenet

Hányadik háromszögszám? **100**
 A 100. háromszögszám: 5050

A felhasználó a 100-as értéket írta be a program kérdésére. A program tehát kiszámította a 100. háromszögszámot, és kiírta a képernyőre: 5050. Természetesen íihatott volna 10-et vagy 30-at is a felhasználó, az annyiadik háromszögszámot is kiszámította volna a program.

Az 5.4 Lista első `printf` utasítása csak annyit tesz, hogy felszólítja a felhasználót az adat beírására. Mindig jó kiírni, hogy milyen adatot is vár a program. Az üzenet megjelenítése után a `scanf` utasítás fut le. A `scanf` első paramétere hasonlít a `printf` formátumjelző karakterláncához. Itt azonban nem a kiírás formátumát, hanem a beolvasandó adat típusát adhatjuk meg. A `printf`-hez hasonlóan a `%i`-vel egy egész szám beírását jelezzük.

A `scanf` második paraméterével adhatjuk meg, hogy milyen változóban szeretnénk tárolni a beadott értéket. A változó neve előtti & jel nem véletlen, de most ne foglalkozzunk vele. A 11. fejezetben a mutatók kapcsán még találkozunk ezzel az operátorral. A `scanf`-ben minden & operátorról kell tennünk a változó neve elő. Ennek elfejtése kiszámíthatatlan következményekkel járhat, és a program rendellenes befejeződéséhez vezethet.

Ezek után világos, hogy az 5.4 Lista `scanf` utasítása beolvas egy egész számot, amit eltárol a `number` változóban – ez a kiszámítandó háromszögszám sorszáma.

Miután ezt a várt értéket begépelte a felhasználó (és utána Entert ütött), a program folytatódik, és lezajlik a keresett szám kiszámítása. A számolás ugyanúgy történik, mint az 5.2 Listában, azzal a különbséggel, hogy 200 helyett a beírt `number` értékéig fut a ciklus. A keresett háromszögszám kiszámítása után megjelenik a képernyőn az eredmény, majd pedig kilép a program.

Egymásba ágyazott for ciklusok

Az 5.4 Lista segítségével szinte bármelyik háromszögszám kiszámítható. Ha öt különböző háromszögszámat szeretnénk megkapni, ötször kell elindítani a programot, és minden egyes futáskor be kell írni a kívánt sorszámot.

Ugyanezt a feladatot másként is megoldhatjuk, miközben újabb érdekességeket tudhatunk meg a C nyelvről. Megoldásunkban egy újabb ciklust hozunk létre, mely a kívánt számításokat ötször egymás után elvégzi. A **for** ciklus, mint tudjuk, képes ilyen feladat el-látására. Az 5.5 Lista és a hozzá tartozó kimenet ezt mutatja be.

5.5 Lista • Egymásba ágyazott ciklusok

```
#include <stdio.h>

int main (void)
{
    int n, number, triangularNumber, counter;

    for ( counter = 1; counter <= 5; ++counter ) {
        printf ("Hányadik háromszögszám? ");
        scanf ("%i", &number);

        triangularNumber = 0;

        for ( n = 1; n <= number; ++n )
            triangularNumber += n;

        printf ("A %i. háromszögszám a %i\n\n", number,
               triangularNumber);
    }

    return 0;
}
```

5.5 Lista • Kimenet

Hányadik háromszögszám? **12**

A 12. háromszögszám a 78

Hányadik háromszögszám? **25**

A 25. háromszögszám a 325

Hányadik háromszögszám? **50**

A 50. háromszögszám a 1275

Hányadik háromszögszám? **75**

A 75. háromszögszám a 2850

Hányadik háromszögszám? **83**

A 83. háromszögszám a 3486

A programban két, egymásba ágyazott `for` ciklus szerepel. A külső `for` ciklus a következő:

```
for ( counter = 1; counter <= 5; ++counter )
```

Ezserint a ciklusmag ötször fut le, hiszen a `counter` változót 1-től kezdve növeljük 5-ig (amely még teljesíti a ciklusfeltételt, de a 6 már nem).

Az előző példákkal ellentétben a `counter` ciklusváltozót nem használjuk semmi másra, mint erre a számlálásra. Ennek ellenére – minthogy ez mégiscsak egy változó – deklarálnunk kell a program elején. A program további része képezi a külső ciklus magját – erre utalnak a kapcsos zárójelek. Könnyebben elképzelhető a program szerkezete az alábbi séma alapján:

```
For 5-ször
{
    A szám bekérése
    A kért háromszögszám kiszámítása
    Az eredmény kiírása
}
```

A ciklusnak az a része, amit „A kért háromszögszám kiszámítása” mondattal jellemzünk, a `triangularNumber` lenullázásából és egy újabb `for` ciklusból áll (ez számítja ki a kért háromszögszámot). Láthatjuk, hogy ez a `for` ciklus a külső `for` cikluson belül szerepel. Ez teljesen korrekt C programnyelvi megoldás – 127 szintig lehet egymásba ágyazni ciklusokat.

A megfelelő formázás (beljebb kezdés, *indentáció*) létkérdés olyankor, amikor több programozási elemet ágyazunk egymásba (például `for` ciklusokat). Az olvasható programban egyből látszik, hogy mely utasítások mely ciklus magjához tartoznak. (Annak bemutatására, hogy milyen olvashatatlan tud lenni egy program, amit nem megfelelően formáztak, érdemes egy pillantást venni a fejezet végén álló 5. gyakorlatra).

Változatok `for` ciklusra

A `for` ciklus szintaktikája tágabb lehetőségeket is hordoz, mint amiről eddig szó volt. Előfordul, hogy több változó inicializációjára is szükség van a ciklus kezdetén, vagy hogy több léptetést is szeretnénk végrehajtani futás közben.

Többszörös kifejezések

A `for` ciklusfej mezőiben több kifejezést is elhelyezhetünk, vesszővel elválasztva őket. Kezdődhet például a következőképpen is egy ciklus:

```
for ( i = 0, j = 0; i < 10; ++i )
    ...
```

Ilyenkor az i és a j értéke is nullázódik a ciklus indulása előtt. Két, vesszővel elválasztott kifejezés áll az inicializáció helyén – mindenkettő végrehajtódik, mert együttesen alkotják az inicializációt. Vagy tekintsük a következő ciklusfejet:

```
for ( i = 0, j = 100; i < 10; ++i, j = j - 10 )  
    ...
```

Itt két változó kap kezdeti értékét (az i nullát, a j százat), és a ciklusmag futásai végén a léptetéskor mindeneket megváltozik: az i inkrementálódik, a j tízzel csökken.

Mezők elhagyása

Ahogy szükség lehet egynél több kifejezés végrehajtására, úgy az is előfordulhat, hogy valamelyik mezőre egyáltalán nincs szükség. Lehetőség van az egyes mezők elhagyására – egyszerűen üresen kell hagyni a pontosvesszők által kijelölt helyet. Ennek leggyakoribb példája az inicializáció elhagyása:

```
for ( ; j != 100; ++j )  
    ...
```

Ilyesfélé ciklusfej használható akkor, ha a j ciklusváltozó értéke már korábban inicializálódott.

Az olyan ciklust, amelyben nincs ciklusfeltétel-vizsgálat, végtelen ciklusnak hívjuk, mert az ilyen ciklus elvileg végtelen sokáig futna. Előfordulhat azonban, hogy mégis elhagyjuk a ciklusfeltétel-vizsgálatot – ilyenkor általában valami más módon szeretnénk kilépni a ciklusból (return, break vagy goto használatával, amiről később még szó lesz).

Deklaráció a ciklusfejben

Arra is lehetőség van, hogy magában a for ciklusfejben deklarálunk egy-egy változót. Ez ugyanúgy történhet, ahogy a korábbiakban is láttuk a változók kezdeti értékadását. A következő módon valósítható meg például egy for ciklus, melynek számláló (*counter*) változóját a ciklus kezdetén deklaráljuk és inicializáljuk:

```
for ( int counter = 1; counter <= 5; ++counter )
```

A counter változó hatóköre ilyenkor csak a for ciklusra terjed ki, vagyis ez egy úgynevezett lokális változó. A cikluson kívül már nem látható. Az alábbi példa két változót is inicializál, amivel az egész fenti példaprogramunk egyetlen ciklus lesz:

```
for ( int n = 1, triangularNumber = 0; n <= 200; ++n )  
    triangularNumber += n;
```

A while utasítás

A while utasítás tovább szélesíti a C programozási nyelvben használható ciklusok palettáját. Ennek a szintén meglehetősen gyakori utasításnak a szintaxisa a következő:

```
while ( kifejezés )
    ciklusmag
```

A zárójelben álló kifejezés kiértékelődik. Amennyiben *igaz* az értéke, lefut a ciklusmag, amely lehet egy utasítás vagy egy (kapcsos zárójelek között álló) utasításblokk. Ezután újra kiértékelődik a kifejezés. Ha továbbra is *igaz* az értéke, újra lefut a ciklusmag. Ez mindenadig így folytatódik, míg a kiértékelés értéke *hamis* eredményt nem ad. Ekkor ugyanis már nem fut le a ciklusmag, hanem az őt követő utasításra kerül a vezérlés.

Nézzünk egy egyszerű példát egy while ciklusra, mely elszámol 1-től 5-ig (5.6 Lista).

5.6 Lista • A while ciklus bevezetése

```
// A "while" ciklus bevezetése

#include <stdio.h>

int main (void)
{
    int count = 1;

    while ( count <= 5 ) {
        printf ("%i\n", count);
        ++count;
    }

    return 0;
}
```

5.6 Lista • Kimenet

```
1
2
3
4
5
```

A program 1-re állítja a számláló (count) változó értékét, majd elkezdődik a while ciklus végrehajtása. Minthogy a count-ra igaz, hogy kisebb vagy egyenlő ötnél, a ciklusmag azonnal lefut. Az utasításblokkban a count változó kiírása és inkrementálása szerepel. A kimenetből látszik, hogy minden ötször zajlik le – vagyis addig, amíg a count változó el nem éri a 6 értéket.

Bizonyára észrevette az olvasó, hogy ugyanezt for ciklussal is meg lehetett volna oldani. Egy for ciklust minden át lehet alakítani while ciklussá, és ez fordítva is igaz. Általánoságban a következő for utasítás:

```
for ( inicializáció; vizsgálat; léptetés )
    ciklusmag_utasítás
```

egyenértékű az alábbi (kiegészített) while utasítással:

```
inicializáció;
while ( vizsgálat ) {
    ciklusmag_utasítás
    léptetés;
}
```

Mihelyt jobban megismerjük a while utasítást, kialakul a tapasztalatunk, hogy mikor érdemes while ciklust használni, és mikor for ciklust. Általában megfogalmazható, hogy ha előre tudható a cikluslépések száma, akkor a for ciklus az alkalmasabb. Akkor is érdemes for ciklust használni, ha az inicializáció, a vizsgálat és a léptetés ugyanazt a változót érinti.

A következő példa is a while ciklust mutatja be. A program kiszámítja két egész szám legnagyobb közös osztóját. A legnagyobb közös osztó (*Inko*) az, amit a neve is mond: két egész szám közös osztói közül a legnagyobb. Például *Inko*(10,15)=5, hiszen 5 a legnagyobb olyan egész szám, aminek 10 is és 15 is többszöröse.

Régóta ismert egy algoritmus arra, hogy hogyan lehet gyorsan kiszámítani két tetszőleges szám legnagyobb közös osztóját. A módszert Euklidesznek tulajdonítják, aki i.e. 300 körül élt és alkotott. Algoritmusának lényege a következő:

- Feladat:** Keressük két nemnegatív egész szám, u és v legnagyobb közös osztóját.
Első lépés: Ha $v=0$, akkor készen vagyunk, és $\text{Inko}(u,v)=u$
Második lépés: Végezzük el a $\text{segédvált}=u \% v$, $u=v$, $v=\text{segédvált}$ hozzárendeléseket, majd folytassuk munkánkat az Első lépéssel.

Ha nem ismerős az algoritmus, akkor most ne törlök az időt ennek megértésével, csak fogadjuk el, hogy működik. Fordítsuk inkább figyelmünket arra, hogy hogyan lehet megírni egy programot, mely a fenti algoritmus alapján kiszámítja két szám legnagyobb közös osztóját!

Ha egy problémát sikerül megoldanunk az algoritmusok nyelvén, akkor már karnyújtásnyira vagyunk a program megírásától. A fenti algoritmusból kiderül, hogy a „Második lépés” mindaddig ismétlődik, míg v értéke 0 nem lesz. Ennek felismerése arra mutat, hogy érdemes while ciklussal megoldani az ismételt lépéseket.

Az 5.7 Lista programja bekér két számot, majd megkeresi a legnagyobb közös osztójukat.

5.7 Lista • A legnagyobb közös osztó kiszámítása

```
/* A program két nemnegatív egész szám
legnagyobb közös osztóját számítja ki */

#include <stdio.h>

int main (void)
{
    int u, v, temp;

    printf ("Adjon meg két nemnegatív egész számot!\n");
    scanf ("%i%i", &u, &v);

    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }

    printf ("Legnagyobb közös osztójuk: %i\n", u);

    return 0;
}
```

5.7 Lista • Kimenet

Adjon meg két nemnegatív egész számot!

150 35

Legnagyobb közös osztójuk: 5

5.7 Lista • Kimenet (ismételt futtatás)

Adjon meg két nemnegatív egész számot!

1026 405

Legnagyobb közös osztójuk: 27

A scanf-ben szereplő két %i mutatja, hogy két begépelt egész számot vár a program. Az első az u változóba, a második a v-be kerül. A számok begépelésekor szóközökkel vagy enterekkel választható el egymástól a két érték.

A számok begépelése (és a két változóban való eltárolása) után a program vezérlése a while ciklusra kerül, melyben kiszámítjuk a legnagyobb közös osztót. A ciklus befejeztével az u értékben van a keresett *Inko*, amit egy alkalmas printf utasítással ki is írunk a képernyőre.

Az 5.8 Lista a while ciklus egy másik alkalmazását mutatja be: segítségével megfordítható egy beírt szám számjegyeinek sorrendje. Ha például 1234-et írunk be, akkor azt várjuk a programtól, hogy jelenítse meg a 4321 számot.

Minden program megírása előtt tisztában kell lennünk az alkalmazandó algoritmussal. A számjegyek megfordításának algoritmusára egyszerűen abban áll, hogy „olvassuk be a számjegyeket egymás után, jobbról balra haladva”. Ezt az egymás utáni számjegybeolvasást természetesen programmal is megvalósíthatjuk, mindenkor azzal, hogy a begépelt szám számjegyei közül sikerüljön leválasztani (és kiírni) a jobboldalit, majd a maradék számjegyekkel is ugyanezt tehessük. Az egymás mellé kiírt számok épp az eredeti szám megfordítását adják.

Egy szám utolsó számjegyét úgy is megkaphatjuk, hogy vesszük a tízzel való osztási maradékát. Például $1234 \% 10$ négyet ad, ami épp az 1234 utolsó (és egyúttal a szám megfordításának első) számjegye. A $\%$ jel a modulus operátor, amely két szám osztási maradékát adja vissza.

A következő számjegyet is kinyerhetjük, ha az eredeti számot (egészket!) osztjuk tízzel. $1234 / 10$ eredménye 123, és $123 \% 10$ visszaadja a 3-as számjegyet, ami a megfordított szám következő számjegye.

Ugyanez a lépéssor folytatható mindaddig, amíg el nem fogy az eredeti szám. Ez onnan is látszik, hogy a tízzel való egész osztás már nullát ad eredményül.

5.8 Lista • Számjegyek megfordítása

```
// Egy szám számjegyeinek fordított sorrendű kiírása

#include <stdio.h>

int main (void)
{
    int number, right_digit;

    printf ("Írja be a megfordítandó számot!\n");
    scanf ("%i", &number);

    while ( number != 0 ) {
        right_digit = number % 10;
        printf ("%i", right_digit);
        number = number / 10;
    }

    printf ("\n");

    return 0;
}
```

5.8 Lista Kimenet

Írja be a megfordítandó számot!

13579

97531

Ahogy a program feldolgozza az eredeti számot, annak számjegyei sorban megjelennek a képernyőn. Nagy szerepe van annak, hogy a `while` ciklusban szereplő `printf` utasítás használata közben nem jelenik meg újsor karakter az egyes számok kiírása után. Így az egyes számjegyek közvetlenül az előző után, azzal egy sorban jelennek meg. A program végén egy külön `printf` utasítást használunk a sortöréshez, hogy a kurzor a következő sorban jelenjen meg a felhasználó számára.

A do utasítás

A két eddig tárgyalt ciklusszervező utasításban *azelőtt* történt meg a feltételvizsgálat, mivelőtt a ciklusmag lefutott volna. Így előfordulhatott, hogy a ciklusmag egyszer sem futott le, ha a feltétel már a ciklus kezdetén sem teljesült. Vannak olyan programozási helyzetek, amikor érdemesebb a feltételvizsgálatot a ciklusmag lefutása *után* megtenni. A C programozási nyelv természetesen lehetőséget ad erre, mégpedig a `do` ciklus révén. Ennek szintaxisa a következő:

```
do
    ciklusmag_utasítás
  while ( vizsgálat );
```

A `do` ciklus végrehajtása azonnal a `ciklusmag_utasítás` lefuttatásával kezdődik. Ezután értékelődik ki a zárójelben szereplő ciklusfeltétel `vizsgálata`. Ha ez igaz, akkor újra végre-hajtódik a `ciklusmag_utasítás`, és ez így ismétlődik mindenkor, amíg a `vizsgálat` igaz eredményt ad. Amikor azonban hamis lesz a kiértékelés eredménye, a program vezérlése továbblép a ciklust követő utasításra.

A `do` ciklus igazából a `while` ciklus átrendezésének tekinthető, melyben a ciklusfeltétel vizsgálata a ciklus eleje helyett a végén történik meg.

Ne felejtsük: (a `for` és a `while` ciklussal ellentétben) a `do` ciklusban a ciklusmag legalább egyszer mindenkor lefut.

Az 5.8 Listában a számjegyek megfordítását `while` ciklussal oldottuk meg. Nézzük meg, mit tenne a program, ha 13579 helyett 0-t adnánk meg bemenő adatként. A `while` ciklus feltétele sohasem teljesülne, így a ciklusmag sohasem futna le. A program csak egy sor-emelést írna ki a képernyőre. Ha azonban `do` ciklust használunk a `while` helyett, akkor legalább egyszer biztosan le fog futni a ciklusmag, azaz legalább egy számjegy biztosan megjelenik a kiíráskor. Az 5.9 Lista ezt az átdolgozott programot mutatja.

5.9 Lista • Számjegyek megfordítása, átdolgozott változat

```
// Egy szám számjegyeinek fordított sorrendű kiírása do-val

#include <stdio.h>

int main ()
{
    int number, right_digit;

    printf ("Írja be a megfordítandó számot!\n");
    scanf ("%i", &number);

    do {
        right_digit = number % 10;
        printf ("%i", right_digit);
        number = number / 10;
    }
    while ( number != 0 );

    printf ("\n");

    return 0;
}
```

5.9 Lista • Kimenet

```
Írja be a megfordítandó számot!
13579
97531
```

5.9 Lista • Kimenet (ismételt futtatás)

```
Írja be a megfordítandó számot!
0
0
```

Ahogy a kimenet is mutatja, ez a változat 0 beírásakor is helyesen fut le.

A break utasítás

Vannak esetek, amikor egy adott feltétel teljesülésekor (például hiba történik, vagy váratlanul elfogynak a feldolgozandó adatok) szeretnénk azonnal kilépni a ciklusból. Ilyenkor használható a **break** (*kitörés, megszakítás*) utasítás, melynek hatására a program azonnal kilép az éppen végrehajtott **for**, **while** vagy **do** ciklusmagból. A ciklusmag **break**-et követő utasításaival mit sem törődve a végrehajtás a ciklust követő utasítással folytatódik.

Ha egymásba ágyazott ciklusok valamelyikében hívjuk meg a **break** utasítást, akkor is igaz a fent vázolt működés: az aktuális ciklusmagból való kilépés ilyenkor annyit jelent, hogy az egygel magasabb szintű ciklusban folytatódik a program futása.

A `break` szintaktikája igen egyszerű; maga a kulcsszó, pontosvesszővel lezárva:

```
break;
```

A `continue` utasítás

A `continue` (*folytasd*) utasítás sok tekintetben hasonlít a `break`-hez, csak ennek hatására nem lépünk ki a ciklusból, hanem az történik, amit a parancs neve is mond. A ciklusmag `continue`-t követő utasításaival mit sem törődve a végrehajtás a következő cikluslépéssel folytatódik (ha van ilyen).

A `continue` utasítást többnyire arra szoktuk használni, hogy (adott feltételek mellett) bizonyos lépéseket kihagyunk, majd a ciklust tovább folytassuk.

A `continue` szintaktikája is egyszerű; maga a kulcsszó, pontosvesszővel lezárva:

```
continue;
```

Amíg komoly jártasságra nem teszünk szert a ciklusok írásában (és a belőlük való kultúrált kilépésben), nem ajánlatos szabadosan teletűzdelni programunkat `break` és `continue` utasításokkal. Segítségükkel ugyanis remekül össze lehet kuszálni és kiismertetlenné lehet tenni egy programot.

A C programozási nyelv alapvető ciklusszervezési utasításainak megismerése után más vezérlési szerkezetek megtanulása sem okozhat majd nehézséget. Következő fejezetünkben a programon belüli döntéshozatalról lesz szó részletesen. Először azonban mélyítsük el a ciklusokról tanultakat néhány gyakorlat elvégzésével.

Gyakorlatok

1. Gépeljük be és futtassuk a fejezetben szereplő kilenc példaprogramot! A kimeneteket hasonlítsuk össze a könyvben szereplő programkimenetekkel.
2. Írunk programot, mely táblázatszerűen kiírja az első tíz számot és azok négyzeteit! Nyomtassunk megfelelő fejlécet is a táblázat két oszlopa fölé.
3. A háromszögszámok nemcsak a korábban említett rekurzív módon, hanem közvetlenül is megadhatóak. Az n. háromszögszám képlete $n(n+1)/2$, ahol n egész szám. Próbáljuk kiszámítani például a (fejezetünkben is kiszámolt) 10. háromszögszámot: ennek értéke a képlet szerint $10 \cdot 11 / 2 = 55$. Írunk programot, mely ezt a képletet használva kiszámít mindenötödik háromszögszámot 5 és 50 között (azaz kiszámítja az 5., 10., 15., ... 50. háromszögszámot).
4. Az első n pozitív szám szorzatát *n faktoriálisnak* hívjuk, jele: *n!* Az 5! például nem más, mint $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Írunk programot, mely kiszámítja az első tíz faktoriális értékét.

5. Az alábbi program tökéletesen működik. Egyetlen hibája, hogy írója vajmi kevés figyelmet fordított az olvashatóságra. Első ránézésre gyakorlatilag követhetetlen. Bár milyen hihetetlen, ebből a programból létre lehet hozni olvasható kódot. A fejezet példaprogramjaira támaszkodva írjuk át ezt a programot úgy, hogy olvashatóbbá váljon, majd gépeljük be és futtassuk le!

```
#include <stdio.h>
int main(void){
    int n,two_to_the_n;
    printf("KETTŐ-HATVÁNYOK TÁBLÁZATA\n\n");
    printf(" n      2 to the n\n");
    printf("---  ----- \n");
    two_to_the_n=1;
    for(n=0;n<=10;++n){
        printf("%2i      %i\n",n,two_to_the_n); two_to_the_n*=2;
    }
    return 0;
}
```

6. A `printf` formátumjelző karakterláncában szereplő mezőszélességi számjegy elő negatív előjelet írva elérhetjük, hogy az adott érték balra igazítva jelenjen meg. Az 5.2 Listában szereplő `printf` utasítást cseréljük ki az alábbira:

```
printf ("% -2i %i\n", n, triangularNumber);
```

Fordítás után futtassuk le a módosított programot, és kimenetét hasonlítsuk össze az eredeti táblázattal.

7. A `printf` formátumjelző karakterláncában szereplő mezőszélességi számjegy elő tizedespontot is írhatunk. Fejtük meg ennek jelentését az alábbi program beírásával, lefordításával és futtatásával. Kísérletezzünk különböző értékek megadásával:

```
#include <stdio.h>

int main (void)
{
    int dollars, cents, count;

    for ( count = 1; count <= 10; ++count ) {
        printf ("Adja meg a dollár mennyiségét: ");
        scanf ("%i", &dollars);
        printf ("Adja meg a cent mennyiségét: ");
        scanf ("%i", &cents);
        printf ("%$.2i.%2i\n\n", dollars, cents);
    }
    return 0;
}
```

8. Az 5.5 Lista csak öt háromszögszámot ír ki. Tegyük lehetővé, hogy a felhasználó megadhassa, hogy hány háromszögszámot szeretne kiszámítatni.
 9. Az 5.2-től 5.5-ig terjedő Listákat írja át úgy, hogy `for` ciklus helyett `while` ciklust használjanak. Futtassuk le valamennyit, és vizsgáljuk meg, hogy valóban egyenértékűek-e a módosított programok az eredetiekkel.

10. Mi történne, ha negatív számot adnánk meg az 5.8 Lista számára? Próbáljuk is ki.
11. Írunk programot, amely kiírja egy szám számjegyeinek összegét. Például a 2155 számjegyeinek összege $2 + 1 + 5 + 5 = 13$. A program fogadjon el tetszőleges egész számot bemenetként.

6

Döntési szerkezetek

Az előző fejezetben láttuk, hogy a számítógép egyik leghasznosabb tulajdonsága, hogy egy-egy utasítássort ismételten végre tud hajtani. Egy másik, kiemelkedően hatékony képesség a döntéshozatal. A ciklusfeltétel vizsgálatakor tulajdonképpen már használtuk is ezt a képességet: a program önállóan eldöntötte, kell-e még folytatni a ciklust, vagy ki kell lépni belőle. Enélkül a döntés nélkül az örökkévalóságig futnának a ciklusok (mint a „végtelen ciklus”).

A C programozási nyelv más döntési szerkezeteket is kínál. Ebben a fejezetben a következőket vesszük szemügyre:

- Az if utasítás
- A switch utasítás
- A feltételkezelő operátor

Az if utasítás

A C programozási nyelvben az általános döntéshozó utasítás neve `if`, szintaxisa a következő:

```
if (kifejezés)
    utasítás
```

Vegyük egy példát az emberi nyelvből. Próbáljuk C nyelven megfogalmazni a következő mondatot: Ha nem esik az eső, elmegyek úszni. C-ben ez valahogy így festene:

```
if (nem esik az eső)
    elmegyek úszni
```

Az if utasítással meghatározhatjuk, feltételhez köthetjük egy másik utasítás(blokk) végre-hajtását. Csak akkor megyek el úszni, ha nem esik az eső. Ehhez hasonlóan az alábbi printf kódrészlet csak akkor hajtódik végre, ha a count értéke nagyobb a COUNT_LIMIT-nél – egyébként nem veszi figyelembe a vezérlés a printf parancsot:

```
if ( count > COUNT_LIMIT )
    printf ("A számláló túllépte a küszöbértéket\n");
```

A következő egyszerű példa segít aprópénzre váltani a fentieket. Írunk egy olyan programot, amely bekér egy számértéket a felhasználótól, majd annak abszolút értékét kiírja a képernyőre. Egy szám abszolút értékét úgy számíthatjuk ki, hogy ellentettjét vesszük, ha a szám negatív. Ebből már adódik is a programba építendő feltétel: „ha a szám kisebb nullánál”. Ezt láthatjuk a 6.1 Listában.

6.1 Lista • Egy egész szám abszolút értékének kiszámítása

```
// Egy egész szám abszolút értékét kiszámító program

int main (void)
{
    int number;

    printf ("Adja meg a számot: ");
    scanf ("%i", &number);

    if ( number < 0 )
        number = -number;

    printf ("Abszolút értéke: %i\n", number);

    return 0;
}
```

6.1 Lista • Kimenet

```
Adja meg a számot: -100
Abszolút értéke: 100
```

6.1 Lista • Kimenet (ismételt futtatás)

```
Adja meg a számot: 2000
Abszolút értéke: 2000
```

Érdemes legalább kétszer lefuttatni a programot, hogy látsszon, valóban másként kezeli a negatív és a pozitív számokat. Lehet persze több próbát is tenni, hogy még meggyőzőbb legyen a program helyessége, de két futtatással már ellenőrizhető a kétféle döntési lehetőség megvalósulása.

Nézzük, mit is tesz a program. Az üzenet kiírása és a felhasználói bemenet bekérése (valamint a number változóban való eltárolása) után a program megvizsgálja a number változó értékét, hogy kisebb-e nullánál. Ha kisebb (azaz a szám negatív), akkor egy mínusz operátor segítségével ellentettjére váltja át a változó értékét. Ha nem teljesült a feltétel (azaz a szám nem kisebb nullánál), akkor ez azt értékadást egyszerűen átugorja a program-

vezérlés. Ez jól is van így, hiszen ha a szám nem negatív, akkor nincs semmi teendő; maga a számérték lesz az abszolút érték. A `printf` utasítás megjeleníti a beadott érték abszolút értékét, és a program kilép.

Nézzük most a 6.2 Listát. Ebben is szerepel az `if` utasítás. Tegyük fel, hogy egy sereg (százalékos) osztályzat átlagát szeretnénk kiszámítani, valamint az elégtelenek számát is szeretnénk tudni. 65 százalék alatt már számítson elégtelennek az eredmény.

Az „elégtelen” fogalma egy döntést takar: eléri-e az adott eredmény a küszöbértékként megadott 65 százalékot. Itt is az `if` utasítás segítségünkre.

6.2 Lista • Osztályzatok átlagának (és az elégtelenek számának) kiszámítása

```
/* Osztályzatok átlagának  
 és az elégtelenek számának kiszámítása */  
  
#include <stdio.h>  
  
int main (void)  
{  
    int      numberofGrades, i, grade;  
    int      gradeTotal = 0;  
    int      failureCount = 0;  
    float    average;  
  
    printf ("Hány százalék-eredményt kíván beírni? ");  
    scanf ("%i", &numberofGrades);  
  
    for ( i = 1; i <= numberofGrades; ++i ) {  
        printf ("Adja meg a lista %i. eredményét: ", i);  
        scanf ("%i", &grade);  
  
        gradeTotal = gradeTotal + grade;  
  
        if ( grade < 65 )  
            ++failureCount;  
    }  
  
    average = (float) gradeTotal / numberofGrades;  
  
    printf ("\nA százalékos átlag: %.2f\n", average);  
    printf ("Az elégtelenek száma: %i\n", failureCount);  
  
    return 0;  
}
```

6.2 Lista • Kimenet

Hány százalék-eredményt kíván beírni? 7

Adja meg a lista 1. eredményét: 93

Adja meg a lista 2. eredményét: 63

Adja meg a lista 3. eredményét: 87

Adja meg a lista 4. eredményét: 65

Adja meg a lista 5. eredményét: 62

Adja meg a lista 6. eredményét: 88

Adja meg a lista 7. eredményét: 76

A százalékos átlag: 76.29

Az elégtelenek száma: 2

A gradeTotal változóban összegezzük a beírt számokat; kezdetben ennek értékét 0-ra állítjuk. Az elégtelen eredmények számát a failureCount változóban gyűjtjük – ezt is nullával inicializáljuk a program elején. Az average (*átlag*) változót float (azaz lebegőpontos) típusúként deklaráljuk, hiszen több szám átlaga általában törtszám.

A program bekéri a feldolgozandó értékek darabszámát, amit eltárol a numberOfGrades változóban. Ezután egy ciklus kezdődik, melynek minden lépésében megjelenik egy felhívás, ami után beírhatunk egy (százalékban kifejezett) egész értéket, mely a grade változóban tárolódik.

A grade változó értéke minden cikluslépésben hozzáadódik a gradeTotal-hoz. Utána következik a döntéshozatal: elégtelenek számít-e az adott eredmény? Ha igen, akkor a failureCount változót inkrementáljuk.

Ugyanezek az utasítások ismétlődnek a ciklus valamennyi lépésében. Amikor az összes eredményt bevittük és összegeztük, kiszámítjuk az átlagot. Bizonyára sokan úgy gondolnák, hogy az $\text{average} = \text{gradeTotal} / \text{numberOfGrades}$ utasítás megfelelő lesz az átlagszámításra. Ennek használatával azonban az eredmény törtrésze elveszne, hiszen *mind* a számláló, *mind* a nevező egész szám, és ilyenkor egész osztás történik, ami a maradékkal nem foglalkozik.

Két megoldás is létezik a problémára. Az egyik az, hogy deklaráljuk lebegőpontosnak a gradeTotal vagy a numberOfGrades változót. Ennek hatására lebegőpontos műveletként menne végbe az osztás. A két említett változóban azonban csak egész számokat tárolunk. Lebegőpontosként való feltüntetésük átláthatatlanabbá tenné a programot, és az ilyen megoldás egyébként sem elegáns.

A másik megoldás az, amit a programban is használunk. A számolás pillanatára lebegőpontossá *alakítjuk* az egyik változót (esetünkben a számlálót) a (float) típusátalakító operátorral. Ennek hatására a gradeTotal változót az osztás pillanatában lebegőpontos-

nak tekinti a program. A művelet ily módon egy lebegőpontos és egy egész szám között zajlik le, amely már lebegőpontos végeredményt ad. Így a teljes hánnyados (törtrészestől) eltárolódik az average változóban.

Kiszámítása után ki is írja a program azt átlag értékét, két tizedesjegy pontossággal.

A `printf` formátumjelző karakterláncában (`f` vagy `e` típus esetén) egy tizedesponttal és számmal jelezhetjük a megjelenítendő tizedesjegyek számát, amely kerekített érték lesz. Ezt pontosságmódosítónak hívjuk. A 6.2 Listában tehát a `.2` pontosságmódosítóval két tizedesjegyre definiáltuk az `average` változó kiírásakor megjelenő tizedesjegyek számát.

A program végül kiírja az elégterek számát, majd befejezi futását.

Az if-else szerkezet

Ha valaki olyan kérdést tesz fel nekünk, hogy páros-e egy adott szám, akkor az az első ötletünk, hogy nézzük meg az utolsó számjegyet. Ha ez `0`, `2`, `4`, `6` vagy `8`, akkor biztosak lehetünk benne, hogy a szám páros, egyébként pedig páratlan.

A számítógép számára egyszerűbben is meg tudjuk ezt fogalmazni, közvetlenül az osztátságóból kiindulva. Ha egy szám maradék nélkül osztható kettővel, akkor páros, egyébként pedig páratlan.

Már találkoztunk a modulus operátorral (%), amellyel egy egész számnak egy másik egésszel vett osztási maradékát számíthatjuk ki. Ez tökéletesen használható a számok paritásának meghatározására is. Ha a számot 2-vel osztva nulla maradékot kapunk, akkor az páros, egyébként pedig páratlan.

Lássuk a 6.3 Listát, amely meghatározza, hogy a begépelt egész szám páros-e vagy páratlan, majd ezt egy megfelelő üzenetben kiírja a képernyőre.

6.3 Lista • Paritás meghatározása

```
// Egy szám paritásának meghatározása

#include <stdio.h>
int main (void)
{
    int number_to_test, remainder;

    printf ("Adja meg a vizsgálandó számot: ");
    scanf ("%i", &number_to_test);

    remainder = number_to_test % 2;
```

```

    if ( remainder == 0 )
        printf ("A szám páros.\n");

    if ( remainder != 0 )
        printf ("A szám páratlan.\n");

    return 0;
}

```

6.3 Lista • Kimenet

Adja meg a vizsgálandó számot: **2455**
A szám páratlan.

6.3 Lista • Kimenet (újfuttatás)

Adja meg a vizsgálandó számot: **1210**
A szám páros.

Miután begépeáltuk a kért számot, a program kiszámítja a kettes maradékot. Az első if utasítás megvizsgálja, hogy a maradék nulla-e. Ha igen, kiírja a megfelelő üzenetet: „*A szám páros*”.

A második if utasítás azt vizsgálja meg, hogy a maradék nullától eltérő-e. Ha igen, megjelenik a képernyőn, hogy a szám páratlan.

A kettővel való oszthatóság kétféle kimenetele alapján biztosan tudjuk, hogy ha az első if-ben megfogalmazott feltétel teljesül, akkor a második nem (és fordítva). Azaz ha a maradék nulla, akkor a szám páros, *egyébként* pedig páratlan. Ez az „*egyébként*” (else) gyakran felmerül a gyakorlatban, így a legtöbb modern programozási nyelv lehetővé teszi ennek a logikának a programban való alkalmazását. A C nyelvben ezt az if-else szerkezetet valósítja meg:

```

if (kifejezés)
    utasítás(blokk)1
else
    utasítás(blokk)2

```

Az if-else szerkezet tulajdonképpen az if utasítás kiegészítése. Ha a *kifejezés* értéke *igaz*, akkor az *utasítás(blokk)1* hajtódiik végre, egyébként pedig az *utasítás(blokk)2*. Tehát vagy az egyik, vagy a másik utasítás(blokk) fut le – együtt sohasem kerül rájuk a vezérlés.

A 6.3 Listát átfogalmazhatjuk: a két if utasítás helyett érdemesebb az if-else szerkezetet elnünk. Ily módon egyszerűbbé, átláthatóbbá és olvashatóbbá válik a program.

6.4 Lista • Paritás meghatározása más módon

```
// Egy szám paritásának meghatározása, 2. változat

#include <stdio.h>
int main (void)
{
    int number_to_test, remainder;

    printf ("Adja meg a vizsgálandó számot: ");
    scanf ("%i", &number_to_test);

    remainder = number_to_test % 2;

    if ( remainder == 0 )
        printf ("A szám páros.\n");
    else
        printf ("A szám páratlan.\n");

    return 0;
}
```

6.4 Lista • Kimenet

Adja meg a vizsgálandó számot: **1234**
 A szám páros.

6.4 Lista • Kimenet (újrafuttatás)

Adja meg a vizsgálandó számot: **6551**
 A szám páratlan.

Ne feledjük, hogy az egyenlőség-vizsgálat operátora a kettős egyenlőségjel (`==`), míg az önmagában álló `=` jel értékadásra való. Sok fejtörést okozhat, ha az `if` feltétel részében értékadást írunk egyenlőség-vizsgálat helyett.

Összetett feltételvizsgálat

Eddigi `if` utasításainkban a feltételvizsgálat minden összege két szám összehasonlításából állt. A 6.1 Listában a `number` értékét hasonlítottuk össze 0-val, a 6.2 Listában a `grade`-et a 65-tel. Előfordul, hogy ennél összetettebb vizsgálatot kell elvégeznünk. Ha a 6.2 Listában az elégtelenek száma helyett arra lettünk volna kíváncsiak, hogy hány eredmény esik a 70 és 79 százalék közé, akkor a `grade`-et két számmal kellett volna összehasonlítani.

A C nyelv lehetőséget ad összetettebb feltételvizsgálatokra is. Az összetett feltételvizsgálat abból áll, hogy az elemi feltételvizsgálatokat logikai operátorokkal kapcsoljuk össze (`és`, `vagy`). Az `és`-nek az `&&`, a `vagy`-nak a `||` karakterpár felel meg. Tekintsük például a következő C utasítást:

```
if ( grade >= 70 && grade <= 79 )
    ++grades_70_to_79;
```

Ez a kódrészlet csak akkor inkrementálja a `grades_70_to_79` értékét, ha a `grade` értéke legalább 70 és ugyanakkor nem nagyobb 79-nél. Hasonlóképpen:

```
if ( index < 0 || index > 99 )
    printf ("Hiba - az index nincs az értelmezési tartományban\n");
```

Itt a `printf` utasítás csak akkor fut le, ha az `index` értéke kisebb 0-nál vagy nagyobb 99-nél.

Az összetett feltételvizsgálatok rendkívül bonyolulttá tudják tenni a C kifejezéseket. A programnyelv megengedi, hogy tetszőleges logikai szerkezeteket valósítsunk meg. Ez a rugalmasság aztán gyakran túlzásokra csábítja a kapkodó programozókat. Az egyszerűbb kifejezéseket jóval könnyebb elolvasnai és nyomon követni.

Az összetett feltételvizsgálatokban szabad zárójeleket használni. Segítségükkel egyértelműbbé válik a kód, és kevésbé esünk abba a hibába, hogy tévesen ítéljük meg a műveletek végrehajtási sorrendjét. Szóközökkel is lehet javítani az olvashatóságot. Egy-egy szóközt írva az `&&` és a `||` karakterpár köré segít azokat vizuálisan elkülöníteni az általuk összekapcsolt kifejezésektől.

Az összetett feltételvizsgálat használatának bemutatására írunk egy olyan programot, amely előönti egy adott évszámról, hogy szökőév-e. minden negyedik év szökőév, kivéve a százzal is osztható évszámokat. A négyszázzal osztható évek azonban szökőévek.

Először is gondoljuk végig, hogyan fogalmazhatnánk meg a fenti meghatározást összetett feltételvizsgálattal. Érdemes kiszámolni a négygyel, százzal és négyszázzal vett maradékokat (*reminder*), és eltárolni őket a `rem_4`, `rem_100` és `rem_400` változókban. Ezeket már könnyebben felhasználhatjuk annak ellenőrzésére, hogy szökőévről van-e szó.

A szökőév fenti meghatározását átfogalmazhatjuk úgy is, hogy egy év pontosan akkor szökőév, ha osztható négygyel és nem osztható százzal, vagy ha osztható négyszázzal. Állunk meg egy pillanatra, és vizsgáljuk meg, hogy valóban egyenértékű-e ez a definíció az eredetivel. Ha ezt sikerült belátnunk, továbbléphetünk a programkód megírása felé. Végül az alábbi eredményre juthatunk:

```
if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )
    printf ("Szökőév!\n");
```

A `rem_4 == 0 && rem_100 != 0` kódrészlet körüli zárójelek elhagyhatóak – alapértelmezésben is a zárójeles változatnak megfelelően történik a kiértékelés. Már csak deklarálni

kell néhány változót, lehetővé kell tenni az adatbevitelt néhány utasítással, és készen is van egy olyan program (6.5 Lista), amellyel a felhasználó meghatározhatja, hogy egy adott év szökőév-e.

6.5 Lista • Szökőév meghatározása

```
// Szökőévet meghatározó program

#include <stdio.h>

int main (void)
{
    int year, rem_4, rem_100, rem_400;

    printf ("Adja meg a vizsgálandó évszámot: ");
    scanf ("%i", &year);

    rem_4 = year % 4;
    rem_100 = year % 100;
    rem_400 = year % 400;

    if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )
        printf ("Ez szökőév!\n");
    else
        printf ("Ez az év nem szökőév.\n");

    return 0;
}
```

6.5 Lista • Kimenet

```
Adja meg a vizsgálandó évszámot: 1955
EZ az év nem szökőév.
```

6.5 Lista • Kimenet (első újrafuttatás)

```
Adja meg a vizsgálandó évszámot: 2000
EZ szökőév!
```

6.5 Lista • Kimenet (második újrafuttatás)

```
Adja meg a vizsgálandó évszámot: 1800
EZ az év nem szökőév.
```

Az első futtatáskor olyan évszámot adtunk meg, amely nem osztható négygyel (1955, nem szökőév). Másodszor egy négyszázzal osztható (azaz szökőév) került megadásra: 2000. Végül olyan évvel futtattuk le a programot, amely négygyel és százzal is osztható (azonban négyszázzal nem), így nem minősül szökőévnek. A vizsgálandó esetek közül nem néztük meg azt, amikor egy év négygyel osztható, de százzal nem – maradjon ez az olvasó feladata.

Ahogy korábban említettük, a C nyelv rendkívül rugalmas lehetőségeket nyújt a megfogalmazható kifejezések terén. Az előző programban nem lett volna szükséges `rem_4`, `rem_100` és `rem_400` változókat használni – a kívánt maradékokat közvetlenül az `if` utasításban is kiszámíthattuk volna:

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

Szóközök okos használatával olvashatóbbá válik a programkód, mert világosan látszik a műveletek végrehajtási sorrendje. Ha elhagynánk a „szükségtelen” zárójeleket és szóközöket, az alábbi olvashatatlan sorhoz jutnánk:

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

Szinte hihetetlen, de ez a kifejezés teljesen egyenértékű az előbbivel, pontosan ugyanazt hajtja végre. Megdöbbentő, milyen sokat jelenthet néhány szóköz az összetett kifejezések megértésének rögös útján.

Egymásba ágyazott if utasítások

Emlékezzünk vissza az `if` utasítás általános formájára. Eszerint, ha a feltétel kiértékelése igaz eredménnyel zárul, akkor a közvetlenül utána álló utasítás fut le. Ez az utasítás lehet akár egy másik `if` utasítás is:

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Te lépsz.\n");
```

Ha a `gameIsOver` változó 0, akkor lefut a következő utasítás, ami szintén egy `if`. Ebben a `playerToMove` és a `YOU` változók összehasonlítása történik meg. Ha értékük egyenlő, akkor megjelenik a „Te lépsz.” üzenet. Azaz a `printf` utasítás akkor fut le, ha a `gameIsOver` változó egyenlő 0-val és a `playerToMove` egyenlő a `YOU`-val. A fenti kódrészlet tehát összetett feltételvizsgálattal is megfogalmazható:

```
if ( gameIsOver == 0 && playerToMove == YOU )
    printf ("Te lépsz.\n");
```

Íme egy praktikusabb példa, amelyben a beágyazott `if` utasításhoz `else` ág is tartozik:

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Te lépsz.\n");
    else
        printf ("Én lépek.\n");
```

Ennek az utasításnak a végrehajtása úgy zajlik, ahogy arról már korábban is szó volt, ki-egészítve azzal, hogy amennyiben a playerToMove és a YOU változók eltérőek, akkor az else ág hajtódiik végre. Ekkor az „Én lépek” üzenet jelenik meg. Ha azonban a gameIsOver változó értéke nem nulla, akkor az egész if utasítást (else ággal együtt) figyelmen kívül hagyja a program.

Figyeljük meg, hogy az else ág ahhoz az if-hez tartozik, amely a playerToMove változót vizsgálja, és nem ahhoz, amelyik a gameIsOver-t. Általános szabály, hogy az else ág minden legutolsó (else-et nem tartalmazó) if-hez tartozik.

Lépjünk tovább azzal, hogy egy else ágat is adunk a külső if-hez. Ez akkor fut le, ha a gameIsOver változó értéke nem nulla.

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Te lépsz.\n");
    else
        printf ("Én lépek.\n");
else
    printf ("Vége a játéknak.\n");
```

A sorok értelemszerű beljebb kezdése (indentálása) sokat segíthet az összetett utasítások megértésében.

A szándékunk szerinti szép formázás azonban nem jelenti azt, hogy a fordítóprogram is a beljebb kezdéseknek megfelelően fogja értelmezni a programkódot. Ha például az előző kódrészletből eltávolítjuk az első else ágat, a következőt kapjuk:

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Te lépsz.\n");
else
    printf ("Vége a játéknak.\n");
```

Ez azonban *nem* jelenti azt, hogy a fordító is így értelmezi majd a programot, hisz az else ág minden formázási erőfeszítésünk ellenére az utolsó if-hez fog tartozni, vagyis a tényleges helyzetet tükröző forma ez lenne:

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Te lépsz.\n");
    else
        printf ("Vége a játéknak.\n");
```

Zárójelek használatával ugyanakkor elérhető ettől eltérő felépítés is. Így már tényleg megoldható, hogy ne a belső if-hez, hanem a külsőhöz tartozzon az else ág:

```
if ( gameIsOver == 0 ) {
    if ( playerToMove == YOU )
        printf ("Te lépsz.\n");
}
else
    printf ("Vége a játéknak.\n");
```

Ezzel a kívánt hatást értük el, vagyis csak akkor jelenik meg a „Vége a játéknak” üzenet, ha a gameIsOver értéke nem 0.

Az else if szerkezet

Láttuk, hogy az else szerkezet milyen szerepet játszik olyan döntéshelyzetekben, melyeknek kétféle kimenetele lehet. Egy egész szám vagy páros, vagy páratlan. Egy év vagy szökőév, vagy sem. A programokban felmerülő vizsgálatok azonban nem minden ilyen egyszerű fekete-fehér helyzetek. Tekintsük például a *signum* (*előjelek*) függvényt, amely -1-et rendel a negatív számokhoz, 0-t a nullához és 1-et a pozitív számokhoz. Mivel ennek a vizsgálatnak háromféle kimenetele lehet, a megvalósításához nem elég egy egyszerű if-else szerkezet. Természetesen használható lenne három független if utasítás, azonban ez a megoldás nem minden működik. Nem minden könnyű egymást kizáró feltételeket megfogalmazni.

Ebben a helyzetben felvehetünk az első else ágba egy másik if-else utasítást. Mivel az else ágban tetszőleges utasítás állhat, semmi akadálya ennek a megoldásnak. Általában is használható a következő szerkezet:

```
if ( kifejezés1 )
    utasítás1
else
    if ( kifejezés2 )
        utasítás2
    else
        utasítás3
```

Ezzel a szokásos kétirányú döntési elágazás háromirányúvá bővül. További if-else szerkezetek is felvehetők az utolsó else ágba, melyekkel tetszőleges számú döntési elágazás hozható létre. Ez olyan gyakran előkerül a programozási gyakorlatban, hogy külön neve is lett: if else szerkezet. A fenti sémának kialakult egy másféle formája is:

```
if ( kifejezés1 )
    utasítás1
else if ( kifejezés2 )
    utasítás2
else
    utasítás3
```

Így jóval olvashatóbb formában követhetjük nyomon a háromirányú döntési elágazást.

A 6.6 Listában a fent vázolt előjelfüggvényt valósítjuk meg.

6.6 Lista • A signum függvény

```
// A signum függvényt megvalósító program

#include <stdio.h>
int main (void)
{
    int number, sign;

    printf ("Adja meg a számot: ");
    scanf ("%i", &number);
    if ( number < 0 )
        sign = -1;
    else if ( number == 0 )
        sign = 0;
    else          // Kötelező pozitív
        sign = 1;

    printf ("Signum: %i\n", sign);

    return 0;
}
```

6.6 Lista • Kimenet

```
Adja meg a számot: 1121
Signum: 1
```

6.6 Lista • Kimenet (első újrafuttatás)

```
Adja meg a számot: -158
Signum: -1
```

6.6 Lista • Kimenet (második újrafuttatás)

```
Adja meg a számot: 0
Signum: 0
```

Ha a beírt szám negatív, a sign változó értéke -1, ha 0-t adunk meg, a sign értéke 0, végül pozitív bemenet esetén a sign változó értéke 1 lesz.

A 6.7 Lista elemzi a megadott karaktert. Eldönti, hogy betű-e (a-z, A-Z), szám-e (0-9) vagy speciális karakter (bármi más). A scanf-ben %c-t kell megadnunk ahhoz, hogy egyetlen karaktert olvasson be a program.

6.7 Lista • Egy begépelet karakter kategorizálása

```
// Egy begépelet karaktert kategorizáló program

#include <stdio.h>

int main (void)
{
    char c;

    printf ("Adjon meg egy karaktert:\n");
    scanf ("%c", &c);

    if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )
        printf ("Ez egy betű.\n");
    else if ( c >= '0' && c <= '9' )
        printf ("Ez egy számjegy.\n");
    else
        printf ("Ez egy speciális karakter.\n");

    return 0;
}
```

6.7 Lista • Kimenet

Adjon meg egy karaktert:
&
 Ez egy speciális karakter.

6.7 Lista • Kimenet (első újrafuttatás)

Adjon meg egy karaktert:
8
 Ez egy számjegy.

6.7 Lista • Kimenet (második újrafuttatás)

Adjon meg egy karaktert:
B
 Ez egy betű.

A begépelés utáni első vizsgálat azt állapítja meg, hogy a c változó betű-e, azaz a kisbetűk vagy a nagybetűk közé tartozik-e. A kisbetűket a

(c >= 'a' && c <= 'z')

a nagybetűket a

(c >= 'A' && c <= 'Z')

kifejezés írja le. Esetükben (a-z, illetve A-Z karakterekre) *igaz* értékkel tér vissza a kiértékelés. Ezek a vizsgálatok minden számítógépes rendszereken működnek, melyek a karaktereket ASCII formátumban tárolják.¹

Ha a c változó értéke betű típusú karakter, akkor az első vizsgálat teljesül, és megjelenik az üzenet: Ez egy betű. Ha nem teljesül a feltétel, akkor az else if ág fut le. Ez az ág vizsgálja meg, hogy számról van-e szó. Érdemes megfigyelni, hogy a vizsgálat során azt nézzük meg, hogy a c értéke a '0'..'9' karakterek közül való-e, nem pedig azt, hogy a 0..9 egész számokkal egyezik-e meg a c. A billentyűzetről ugyanis karakter-beolvasás történik, és a '0'..'9' karakterek nem egyeznek meg a 0..9 számokkal. Olyan számítógépeken, melyeken a karakterek belső ábrázolása ASCII rendszerben történik, a '0' karaktert a 48, az '1'-et a 49 stb. számértékek képviselik.

Ha a c tartalma egy számjegy, akkor az az üzenet jelenik meg, hogy Ez egy számjegy. Ha pedig egyik vizsgálat sem járt sikерrel, azaz a c nem betű, sem nem számjegy, akkor az a felirat jelenik meg, hogy Ez egy speciális karakter. Ezzel véget ér a program futása.

Annak ellenére, hogy a scanf-fel csak egyetlen karaktert olvasunk be, mégis kell egy Entert (vagy Returnt) ütni a bevitel végén. Általában is igaz, hogy a terminálablakból bevitte adatok csak az Enter leütése után kerülnek be a programba.

Következő példánkban egy olyan programot írunk, mely egy egyszerű kifejezés bevitelét várja a felhasználótól,

szám operátor szám

formátumban. A program kiszámítja a kifejezés értékét, és két tizedesjegy pontossággal jeleníti meg. Az operátorok az alapműveletek közül kerülhetnek ki (összeadás, kivonás, szorzás, osztás). A 6.8 Listában egy méretes if utasítást láthatunk, mely számos else if ággal ellenőrzi, melyik operátort írta be a felhasználó.

6.8 Lista • Egy egyszerűbb kifejezés kiértékelése

```
/* Egy szám - operátor - szám formátumú kifejezés
kiértékelését végző program */
```

```
#include <stdio.h>

int main (void)
{
    float value1, value2;
```

¹ A kód részletben látható megoldás helyett érdemesebb a szabványos programkönyvtár islower és isupper függvényeit meghívni. Jobb teljesen elkerülni a belső reprezentáció vizsgálatát. Ehhez be kell illeszteni a #include <cctype.h> sort a program elejébe. Programunk azonban csak illusztrációként szolgál.

```

char operator;
printf ("Adja meg a kifejezést:\n");
scanf ("%f %c %f", &value1, &operator, &value2);

if ( operator == '+' )
    printf ("% .2f\n", value1 + value2);
else if ( operator == '-' )
    printf ("% .2f\n", value1 - value2);
else if ( operator == '*' )
    printf ("% .2f\n", value1 * value2);
else if ( operator == '/' )
    printf ("% .2f\n", value1 / value2);

return 0;
}

```

6.8 Lista • Kimenet

Adja meg a kifejezést:

123.5 + 59.3

182.80

6.8 Lista • Kimenet (első újrafuttatás)

Adja meg a kifejezést:

198.7 / 26

7.64

6.8 Lista • Kimenet (második újrafuttatás)

Adja meg a kifejezést:

89.3 * 2.5

223.25

A scanf formátumjelző karakterláncával meghatározzuk, hogy három értéket olvasson be a program, és tárolja el őket a value1, az operator illetve a value2 változókban. Lebegőpontos értéket a %f formátumjelző karakterláncjal lehetséges beolvasni – mint ahogy kiíratni is ezzel lehet. Kifejezésünk első operandusát (value1) ilyen formátumban olvassuk be.

Ezt követően az operátort olvassuk be. Mivel az operátor nem szám, hanem ('+', '- ', '* ', vagy '/') karakter, ezt egy karakter típusú változóba (operator) olvassuk be. A %c formátumjelző karakterlánc jelzi a rendszernek, hogy karakter beolvasása következik.

A formátumjelző karakterláncban látható szóköz hatására az operandus és az operátor között egy (vagy több) szóköz (vagy újsor) karakter helyezhető el, így jól elkülöníthetőek egymástól a számok és a köztük álló műveleti jel. Ha a %f%c%f formátumjelző karakterláncot használnánk, akkor közvetlenül egymás mellé kellene írni az első számot és az operátort, elválasztó karakter nélkül. A %c ugyanis rögtön az első karaktert beolvassa,

még ha az a szóközt jelenti, akkor is. A %f azonban kissé másként működik: a bevezető szóközötől eltekint a scanf (és ez az egész számok beolvasása esetén is így van). Emiatt a %f %c%f is ugyanúgy működne, mint a programban látható változat.

Miután a második számértéket is beolvasta (és a value2 változóban eltárolta) a program, megvizsgálja az operátort (operator), hogy a megengedett négy változat melyikéről van szó. Ha valamelyik vizsgálat egyezést állapít meg, akkor a megfelelő printf kiíratás lezajlik, és megjelenik a kiszámított eredmény. A program futása véget ér.

Ejtsünk még néhány szót a program alaposságáról. Bár a feladatát elvégzi, mégsem érezzük tökéletesnek, mert nem ad visszajelzést a felhasználónak az esetleges hibákról. Mi történik például akkor, ha a felhasználó az operátor jele helyett egy kérdőjelet ír be? A program egyszerűen „átesik” az if utasításon, és nem jelenik meg semmi – az sem, hogy valamit rosszul csinált a felhasználó.

Egy másik probléma akkor léphet fel, ha nullával szeretne osztani a felhasználó. A programnak ezt ellenőriznie kellene, a C-ben ugyanis tilos nullával osztani.

Egy jó programtól elvárjuk, hogy legyen felkészülve a nemkívánatos eseményekre, és tegye meg ennek érdekében a szükséges elővigyázatossági lépéseket. Elegendő teszteléssel ezek a problémák általában ki szoktak derülni. Egy jó programozó azonban már a programkód írásakor felteszi magának a kérdést, hogy „mi történne, ha...”. Érdemes ezt a faja önfegyelmet kialakítani magunkban, és a kritikus programrészek problematikus lehetőségeit már a kódoláskor kiküszöbölni.

A 6.8A Lista a 6.8 Lista módosított változata, mely kivédi a nullával való osztást és a tiltott operátorok használatát.

6.8A Lista • Egy egyszerűbb kifejezés kiértékelésének újraírt változata

```
/* Egy szám - operátor - szám formátumú kifejezés
kiértékelését végző program második változata */
```

```
#include <stdio.h>

int main (void)
{
    float value1, value2;
    char operator;

    printf ("Adja meg a kifejezést:\n");
    scanf ("%f %c %f", &value1, &operator, &value2);

    if (operator == '+')
        printf ("% .2f\n", value1 + value2);
```

```

        else if ( operator == '-' )
            printf ("% .2f\n", value1 - value2);
        else if ( operator == '*' )
            printf ("% .2f\n", value1 * value2);
        else if ( operator == '/' )
            if ( value2 == 0 )
                printf ("Nullával tilos osztani.\n");
            else
                printf ("% .2f\n", value1 / value2);
        else
            printf ("Ismeretlen műveleti jel.\n");

    return 0;
}

```

6.8A Lista • Kimenet

Adja meg a kifejezést:
123.5 + 59.3
182.80

6.8A Lista • Kimenet (első újrafuttatás)

Adja meg a kifejezést:
198.7 / 0
Nullával tilos osztani.

6.8A Lista • Kimenet (második újrafuttatás)

Adja meg a kifejezést:
125 \$ 28
Ismeretlen műveleti jel.

Ha a begépelt operátor a perjel (/), egy következő if utasításban vizsgáljuk meg, hogy a value2 értéke nulla-e. Ha igen, akkor megjelenik a szükséges hibaüzenet; egyébként pedig a program elvégzi a műveletet, és kiírja az eredményt. Figyeljük meg a beágyazott if utasítást, és a hozzá tartozó else ágat.

A program végén látható else ág megakadályozza, hogy csak úgy „át lehessen esni” az if utasításon. Ha az operátor nem egyezik meg egyik felsorolt lehetőséggel sem, életbe lép az utolsó else ág, és megjelenik az üzenet: „Ismeretlen műveleti jel”.

A switch utasítás

A előző példaprogramban látható if-else láncolat (melyben egy változó értékét többször meg kell vizsgálni) olyan gyakran előkerül a gyakorlatban, hogy a C nyelvben külön utasítás született ennek megvalósítására. Ez a switch (*átkapcsoló*) utasítás, melynek általános alakja a következő:

```
switch ( kifejezés )
{
    case value1:
        utasítás
        utasítás
        ...
        break;
    case value2:
        utasítás
        utasítás
        ...
        break;
    ...
    case valueN:
        utasítás
        utasítás
        ...
        break;
    default:
        utasítás
        utasítás
        ...
        break;
}
```

A zárójelben álló *kifejezés* sorban egymás után összehasonlítódik a value1, value2.. valueN értékekkel, melyek mindegyike egy konstans (vagy konstans kifejezés). Ha valamelyik case ágban egyezés található a zárójeles *kifejezés* és a konstans érték között, akkor az oda tartozó utasítás(ok) lefut(nak). Figyeljük meg, hogy akkor sem kell utasításblokkot használni, ha egynél több utasítást írunk egymás után.

A break kulcsszó jelzi az egyes ágak végét, és egyben a switch utasításból való kiléést is. Ne feledjük break-kel lezárnai az egyes ágakat, ugyanis a break elhagyása esetén a programvezérlés „továbbcsorog” a soron következő case ághoz tartozó utasításokra.

Van egy speciális case ág, melyet egyszerűen a default kulcsszó jelez. Akkor kerül ide a vezérlés, ha egyik case ágban sem volt egyezés. Ennek szerepe ugyanaz, mint az előző program végén található „átesés-gátló” else ágnak.

A switch utasítás igazából egyenértékű a következő if utasítással:

```

if ... ( kifejezés == value1 )
{
    utasítás
    utasítás
    ...
}
else if ( kifejezés == value2 )
{
    utasítás
    utasítás
    ...
}
...
else if ( kifejezés == valueN )
{
    utasítás
    utasítás
    ...
}
else
{
    utasítás
    utasítás
    ...
}

```

Ennek ismeretében könnyen átírható a 6.8A Lista gigantikus if utasítása egy switch utasítássá, ahogy az a 6.9 Listában látható:

6.9 Lista • Egy egyszerűbb kifejezés kiértékelésének második újraírt változata

/ Egy szám - operátor - szám formátumú kifejezés*

*kiértékelését végző program switch-es változata */*

```

#include <stdio.h>

int main (void)
{
    float value1, value2;
    char operator;

    printf ("Adja meg a kifejezést:\n");
    scanf ("%f %c %f", &value1, &operator, &value2);

    switch (operator)
    {

```

```

case '+':
    printf ("% .2f\n", value1 + value2);
    break;
case '-':
    printf ("% .2f\n", value1 - value2);
    break;
case '*':
    printf ("% .2f\n", value1 * value2);
    break;
case '/':
    if ( value2 == 0 )
        printf ("Nullával tilos osztani.\n");
    else
        printf ("% .2f\n", value1 / value2);
    break;
default:
    printf ("Ismeretlen műveleti jel.\n");
    break;
}
return 0;
}

```

6.9 Lista • Kimenet

Adja meg a kifejezést:

178.99 - 326.8

-147.81

A kifejezés beolvasása után az operator értéke sorban összehasonlítódik az egyes alapműveleti jelekkel. Ha valamelyiknél egyezés található, végrehajtódik az oda tartozó művelet. A break kulcsszó kiléptet a switch utasításból, vagyis a program véget ér. Ha nem volt egyezés egyik case ágban sem, akkor az alapértelmezett default ág fut le, és megjelenik a hibajelzés: Ismeretlen műveleti jel.

A default ág végén álló break kulcsszó tulajdonképpen felesleges, hiszen nincs már más ága a switch-nek, amire átcseroghatnánk. Érdemes azonban hozzászokni ahhoz, hogy minden ágat lezárunk egy break-kel – a jó programozási szokások teszik értékessé a programozót.

A switch utasítás írásakor tartsuk szem előtt, hogy nem egyezhet meg semelyik két ág case értéke sem – az azonban előfordulhat, hogy több case érték is tartozzon ugyanazokhoz az utasításokhoz. Ezt úgy lehetjük meg, hogy (a kívánt utasítássor elé) több case sort írunk egymás után, mindegyiket kettősponttal lezárva. Példaként tekintsünk egy olyan esetet, amely az előző listán alapul, de a szorzás szimbólumaként * és x jelet is el fogadunk.

A printf utasítás (value1 és value2 összeszorzása) akkor is lefut, ha a * jellel egyezik meg az operator, és akkor is, ha az x betűvel:

```
switch (operator)
{
    ...
    case '*':
    case 'x':
        printf ("% .2f\n", value1 * value2);
        break;
    ...
}
```

Logikai változók

A kezdő programozók előtt egy idő után ott tornyosul a feladat: írunk programot, mely kiír egy prímszám-táblázatot. Prímszámnak tekintjük azokat az egész számokat, melyek nem bonthatóak fel két kisebb egész szám szorzatára, azaz nincs valódi (1-től és önmaguktól eltérő) osztójuk. Ilyenek például a 2, a 3 (a 4 nem, hiszen az felbomlik 2·2-re), az 5, a 7 stb., végtelen sok van belőlük.

Több módszer van, melyekkel prímszámokat lehet generálni. Ha például az 50 alatti prímszámokat kell előállítanunk, akkor a legegyszerűbb módszer minden (egynél nagyobb) p egész számra megvizsgálni, hogy osztható-e a 2 és p-1 közti számok valamelyikével.

Ha osztható, akkor p nem prím, egyébként pedig az. A 6.10 Lista ezzel a módszerrel dolgozva számítja ki az 50-nél kisebb prímszámokat.

6.10 Lista • Prímszámok táblázatának előállítása

```
// Prímszámok táblázatát előállító program

#include <stdio.h>

int main (void)
{
    int      p, d;
    _Bool   isPrime;

    for ( p = 2; p <= 50;      ++p ) {
        isPrime = 1;

        for ( d = 2; d < p; ++d )
            if ( p % d == 0 )
                isPrime = 0;

        if ( isPrime != 0 )
```

```

        printf ("%i ", p);
    }
    printf ("\n");
    return 0;
}

```

6.10 Lista • Kimenet

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

A 6.10 Listának van néhány megszívlelendő tanulsága. A külső for ciklus elszámol 2-től 50-ig a p ciklusváltozóval, melynek prím voltát szeretnénk megvizsgálni. A ciklus első utasítása 1-et rendel az isPrime változóhoz – ennek szerepe hamarosan világos lesz.

A másik ciklus 2-től p-1-ig számlál, a ciklusváltozóval. minden egyes értékre megvizsgálja a program, hogy p osztható-e a-val (azaz nulla maradékot ad-e). Ha igen, akkor p már nem lehet prímszám, hiszen van valódi osztója. Annak jelzésére, hogy p prím voltát nem szükséges tovább vizsgálni, az isPrime változóhoz 0-t rendelünk.

A belső ciklus végén megvizsgáljuk az isPrime változót. Ha értéke nem nulla, akkor nem találtunk valódi osztót, azaz prímszámról van szó – ezt meg is jelenítjük.

Bizonyára feltűnt, hogy az isPrime értéke csak 0 és 1 lehet. Emiatt választottuk a típusát _Bool-ra. Értéke 1-es, ami azt hivatott jelezni, hogy p prímszám. Mihelyt találunk p-hez egy valódi osztót, az isPrime értékét 0-ra állítjuk át, jelezve, hogy p már nem pályázhat a prímtulajdonság dicsőségére. Az efféle változókat jelzőknek (*flag*, *zászló*) hívjuk, melyek értéke csak kétállapotú lehet. Az ilyen kétállapotú jelzőváltozók értékét a programban általában legalább egyszer megvizsgáljuk, és az eredmény (*igaz/hamis*) függvényében kerül a vezérlés különféle kód részletekre.

A C nyelvben a jelzőváltozók igaz/hamis értékét természetes módon 1 és 0 értékekként kezeljük. Amikor a 6.10 Listában az isPrime értékét 1-re állítjuk, valójában az *igaz* értéket rendeljük hozzá (p *prím*), míg a ciklus közben hozzárendelt 0 értékkel azt jelöljük, hogy *hamis* a prímtulajdonság, p *nem prím*.

Nem teljesen véletlen, hogy az 1 jelöli az *igazat* és 0 a *hamisat*. Ez megfeleltethető egy bit állapotának. Amikor a bit „be van kapcsolva”, akkor az értéke 1, amikor pedig „ki van kapcsolva”, értéke 0. A C nyelvben azonban igen kézenfekvő oka van annak, hogy így jelöljük a logikai *igaz* és *hamis* értékeket.

Emlékezzünk vissza a fejezet elejére, amikor arról volt szó, hogy amennyiben az if utasítás zárójelében álló feltétel „teljesül”, akkor fut le a hozzá tartozó utasítás. De mit is jelent a „teljesülés”? A C-ben ez egyszerűen „nem nullát” jelent, semmi többet.

Így a következő kódrészlet hatására mindenig végrehajtódik a kiírás:

```
if ( 100 )
    printf ("Ez mindenig megjelenik a képernyőn.\n");
```

hiszen a feltétel egy nem nulla (100) értéket hordozó kifejezés, azaz „teljesül”.

Fejezetünk minden programjában előkerül az a fogalom, hogy „a nem nulla azt jelenti, hogy teljesül”, illetve „a nulla azt jelenti, hogy nem teljesül” – egyszerű azért, mert ahol feltételvizsgálat van, ott annak kimenetele csak 0 vagy 1 lehet. 0, ha nem teljesül, és 1, ha teljesül a feltétel. Így az alábbi kódrészlet

```
if ( number < 0 )
    number = -number;
```

a következőképp fut le:

- 1 Kiértékelődik a `number < 0` relációs kifejezés. Ha a feltétel teljesül, azaz a `number` értéke kisebb nullánál, akkor a kifejezés értéke 1, egyébként pedig 0.
- 2 Az `if` utasítás megvizsgálja a kiértékelt kifejezés értékét. Ha az eredmény nem nulla, akkor lefut a következő utasítás, egyébként pedig nem fut le.

Ez az elv érvényes a `for`, `while` és `do` ciklusok feltételvizsgálatára is. Összetett feltételvizsgálat esetén is ugyanígy történik minden:

```
while ( char != 'e' && count != 80 )
```

Ha minden kifejezés igaz, akkor a kiértékelés végeredménye 1, de ha bármelyik meghiúsul, akkor a végeredmény 0. Csak ezután a vizsgálat után ellenőrzi a program az összetett kifejezés teljesülését. Ha ennek eredménye 0, akkor kilép a ciklus, egyébként pedig tovább fut.

A jelző változók fogalmának megismerése után tértünk vissza egy pillanatra a 6.10 Listához: az `if (isPrime != 0)` kódrészlet helyett nyugodtan állhatna `if (isPrime)`.

Annak ellenőrzésére, hogy mikor *hamis* egy jelző változó értéke, használhatjuk a negációs operátort: a felkiáltójelet. Az

```
if ( ! isPrime )
```

kifejezésben azt vizsgáljuk meg, hogy mikor *nem igaz* az `isPrime` értéke. Ezt úgy olvasunk, hogy „ha nem `isPrime`”.

Általában is igaz, hogy a

! kifejezés

a negáltját veszi a *kifejezés*-nek. Ha a *kifejezés* értéke nulla, akkor annak negáltja 1, ha pedig nem nulla, annak negáltja 0 lesz.

A logikai negációval átbillenthető egy jelző értéke az ellentétes állapotba:

```
myMove = ! myMove;
```

A negációs operátornak ugyanolyan a kiértékelési szintje, mint az ellentett-képző művelet, jelnek, azaz előbb végrehajtódnak, mint bármely bináris aritmetikai műveleti jel vagy relációs operátor. Ha negálni szeretnénk egy olyan vizsgálatot, hogy kisebb-e x az y-nál, akkor kénytelenek vagyunk a következőképpen zárójelezni:

```
! ( x < y )
```

Ezzel egyenértékű a következő változat:

```
x >= y
```

A 4. fejezetben már említettük, hogy a C nyelvnek vannak sajátos típusai és konstansai, melyek egyszerűsítik a logikai kifejezések írását. Ilyen a *bool* típus és a *true* és *false* konstans – használatukhoz be kell emelni a programba a *<stdbool.h>* fejlécállományt. A 6.10A Lista a 6.10 Lista újrafogalmazása, melyben már kihasználjuk a *<stdbool.h>* fejlécállomány kínálta lehetőségeket.

6.10A Lista • Prímszámtáblázat előállítása, második változat

```
// Prímszámtáblázatot előállító program - második változat
```

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

int main (void)
{
    int p, d;
    bool isPrime;

    for ( p = 2; p <= 50; ++p ) {
        isPrime = true;

        for ( d = 2; d < p; ++d )
            if ( p % d == 0 )
```

```

        isPrime = false;

        if ( isPrime != false )
            printf ("%i ", p);
    }
    printf ("\n");

    return 0;
}

```

6.10A Lista • Kimenet

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

Amint a kódrészletből láthatjuk, a `<stdbool.h>` használatával már nem kell `_Bool` típust használni, hanem használhatjuk a `bool` típust. Ennek igazából csak esztétikai szerepe van, mert egy ilyen típusnevet könnyebb begépelni és elolvasni, valamint jobban illeszkezik a többi C típusnév (`int`, `char`, `float`) sorába.

A feltételkezelő operátor

A C programozási nyelv legszokatlanabb operátora minden bizonnal a feltételkezelő operátor. A többi (egy- vagy kétoperandusú) operátorral ellentétben ennek az operátornak három operandusa van – tekinthetjük háromoperandusú operátornak. E három operátort a kérdőjellel és a kettősponttal (`? :`) különítjük el egymástól. A feltételkezelő operátor általános alakja:

feltétel ? teljesül_kifejezés : nemteljesül_kifejezés

A *feltétel* egy (általában relációs) kifejezés, amely elsőként értékelődik ki a feltételkezelő operátoron belül. Ha a kiértékelés eredménye *igaz* (azaz nem nulla), akkor akkor *teljesül_kifejezés* értékelődik ki, ha azonban *hamis* (azaz nulla), akkor a *nemteljesül_kifejezés*-re kerül a vezérlés, és ennek eredménye lesz a feltételkezelő operátor visszatérési értéke.

A feltételkezelő operátort legtöbbször értékadáskor használjuk: segítségével eldönthető, hogy két érték közül melyiket kapja meg a változó. Tegyük fel, hogy van két egész változónk, *s* és *x*. A következő feltételkezelő operátorral -1-et rendelhetünk *s*-hez, ha *x* negatív, egyébként pedig x^2 -et:

*s = (x < 0) ? -1 : x * x;*

Az utasítás lefutásakor először az $x < 0$ kifejezés értékelődik ki. Az olvashatóság kedvéért általában zárójelekkel vesszük körül a feltételkezelő operátort. Erre általában nem lenne

szükség, mert igen alacsony a precedenciája – alacsonyabb, mint bármely más operátornak, beleértve a hozzárendelő és a vessző operátort is.

Ha x értéke kisebb nullánál, akkor a vezérlés a ? utáni kifejezésre kerül. Itt egy egyszerű konstans található, így ilyenkor (azaz, ha x negatív) -1 rendelődik az s változóhoz.

Ha x értéke nem kisebb nullánál, akkor a vezérlés a : utáni kifejezésre kerül; ennek visszatérési értékét rendeli a program s-hez. Azaz, ha x értéke nagyobb vagy egyenlő nullánál, akkor x^2 (azaz $x \cdot x$) lesz az s változó értéke.

A feltételkezelő operátor másik alkalmazásaként rendeljük hozzá két változó (a és b) nagyobbikát a maxValue változóhoz:

```
maxValue = ( a > b ) ? a : b;
```

Ha a kettőspont utáni (else ágnak megfelelő) kifejezés egy másik feltételkezelő operátor, akkor ez olyan szerkezetet alakít ki, mint amilyen az „else if” szerkezet. A 6.6 Listában látott előjelfüggvényt két feltételkezelő operátorral tudjuk megvalósítani:

```
sign = ( number < 0 ) ? -1 : (( number == 0 ) ? 0 : 1);
```

Ha a number kisebb nullánál, a sign változó értéke -1 lesz, egyébként pedig: ha a number nulla, akkor 0 lesz a hozzárendelési érték, egyébként 1. Az „else” részt körülvevő zárójel felesleges, ugyanis a feltételkezelő operátor balról jobbra értékelődik ki. A következő két kifejezés tehát teljesen egyenértékű:

```
e1 ? e2 : e3 ? e4 : e5
```

és

```
e1 ? e2 : ( e3 ? e4 : e5 )
```

A feltételkezelő operátor nemcsak „jobbérték” (azaz egy értékkadási egyenlőségjel jobb oldala) lehet, hanem bárhol használható, ahol kifejezést vár a fordítóprogram. A number változó signum-értéke (előjele) úgy is kiíratható, hogy nem használunk közben semmilyen segédváltozót:

```
printf ("Signum = %i\n", ( number < 0 ) ? -1 : ( number == 0 ) ? 0 : 1);
```

A feltételkezelő operátor igen hasznos a C nyelvű előfeldolgozó makrók írásakor. Erről a 13. fejezetben lesz bővebben szó.

Ezzel a döntési szerkezeteiről szóló fejezetünk végére értünk. Következő fejezetünkben tömbökről, összetett adattípusokról lesz szó. A tömb igen hatékony eszköz, melyet a legtöbb C programban használunk is. Mielőtt azonban ezekre rátérnénk, vizsgáljuk meg, hogy megértettük-e a fejezetben tanult fogalmakat: végezzük el a következő gyakorlatokat.

Gyakorlatok

1. Gépeljük be és futtassuk a fejezetben szereplő tíz példaprogramot! A kimeneteket hasonlítsuk össze a könyvben szereplő programkimenetekkel. Kísérletezzünk a megadotttól eltérő bemeneti értékekkel is.
2. Írunk programot, mely bekér a felhasználótól két egész számot, majd pedig megvizsgálja, hogy az első szám osztható-e a másodikkal! Jelenítsük is meg a számítás eredményét.
3. Írunk programot, amely bekér a felhasználótól két egész számot, majd pedig megvizsgálja, hogy az első szám osztható-e a másodikkal! Jelenítsük is meg az oszthatóság eredményét.
4. Írunk programot, amely két egész számot vár a felhasználótól, majd pedig elosztja az első számot a másodikkal! Jelenítsük is meg a számítás eredményét három tizedesjegy pontossággal. Ne feledkezzünk el arról, hogy védekeznünk kell a nullával való osztás ellen.
5. Írunk programot, amely egy egyszerű „nyomtató” számológépként működik. Kérjen be a felhasználótól két adatot, egy számot és egy operátort. Operátorként a következőket fogadj el:

- * / S E

Az S operátor legyen a „mentés”, azaz a begépelt számot tegye el a program a memóriába. Az E jelezze a műveletsor végét. A műveleti jelek pedig jelentsék azt a műveletet, amit a programnak el kell végeznie a memóriában levő számmal és a frissen bevitt számmal. Álljon itt egy próbatípus eredménye – illesfélét várunk a megírandó programtól is:

Kezdődik a számolás

```

10 S A memóriába írunk 10 -et
= 10.000000 A memória tartalma
2 / Osztva kettővel
= 5.000000 A memória tartalma
55 - Kivonunk 55 -öt
-50.000000
100.25 S A memóriába írunk 100.25-öt
= 100.250000
4 * Szorzunk négygyel
= 401.000000
0 E Fejezzük be a számolást
= 401.000000

```

Vége a számolásnak

Figyeljünk a nullával való osztásra és az értelmezhetetlen operátorokra is!

5. Az 5.9 Lista egy olyan program kódja, amely egy szám jegyeit fordított sorrendben írja ki a képernyőre. Ez a program azonban nem jól működik, ha negatív számot adunk meg. Írunk programot, amely figyel erre az esetre, és helyesen kezeli a negatív számokat is. Így a -8645 beírása esetén a kimenet legyen 5468-.
6. Írunk programot, amely egy beírt egész szám számjegyeit kiírja szövegesen (angolul). Azaz, ha például beírjuk a programnak, hogy 932, akkor kimenetként az jeleznik meg, hogy

nine three two

Ne felejtsünk el zero-t írni, ha a felhasználó nullát ad meg! (*Megjegyzés:* ez nem egy könnyű feladat.)

7. A 6.10 Lista programjának vannak olyan jellemzői, amelyeken lehetne javítani. A legfeltűnőbb az, hogy a program a páros számokat is sorban megvizsgálja, pedig azok nyilván nem lehetnek prímszámok. Át kellene ugrani a páros osztandókat (és a páros osztókat is). Az is bosszantó, hogy a belső ciklus minden p értékre megvizsgálja az összes számot (osztóként) 2 és p-1 között – akkor is, ha már kiderült, hogy a vizsgált szám nem prímszám. Illesszünk be egy olyan feltételt a for ciklusba, amely az isPrime-et figyeli. A ciklus így csak addig futna, amíg nem találtunk valódi osztót, és d kisebb p-nél. Módosítsuk a 6.10 Listát a feladatban vázolt két szempont szem előtt tartásával, majd futtassuk a programot, hogy helyesen számol-e. (*Megjegyzés:* a 7. fejezetben hatékonyabb prímkereső módszert is láthatunk majd.)



Tömbök

A C nyelvben lehetőségünk van arra, hogy különféle adatok rendezett sorozataival, vagyis tömbökkel dolgozzunk. Ebben a fejezetben áttekintjük, hogyan lehet tömböket megadni és miként lehet őket használni. A későbbi fejezetekben is többször szó esik majd a tömbökről, mert remekül fel lehet őket használni a különféle függvényekben, struktúrákban. Ráadásul a karakterláncok és a mutatók is kiválóan illeszkednek ehhez az adatszerkezethez.

Tegyük fel, hogy iskolai osztályzatokkal szeretnénk dolgozni. Szeretnénk őket számítógépen tárolni, és különféle műveleteket szeretnénk végezni velük (nagyság szerinti rendezés, medián- és átlagszámítás...). A 6.2 Listában már kiszámítottuk néhány osztályzat átlagát: a beadott adatokat összegeztük, és elosztottuk az eredményt az osztályzatok számával. Ha azonban sorba szeretnénk őket rendezni, akkor ehhez más eszközökre is szükség van. Kissé átgondolva a feladatot hamar rájövünk, hogy nyilvánvalóan csak akkor lehet megoldani a rendezést, ha már minden adat a kezünkben van. A korábban megismert eszközökkel ezt csak úgy lehetne megoldani, hogy minden egyes értéket külön változóban tárolunk:

```
printf ("Adja meg az 1. osztályzatot:\n");
scanf ("%i", &grade1);
printf ("Adja meg a 2. osztályzatot:\n");
scanf ("%i", &grade2);
...
...
```

Az osztályzatok begépelése után kezdődhet a sorba rendezés. Egy sereg `if` utasítás egy-másba ágyazásával megoldható, hogy a program meghatározza a legkisebb osztályzatot, majd a sorrendben rákövetkezőt stb., végül a legnagyobbat. Ha tényleg nekiveselkedünk egy ilyen szerkezetű program megírásának, hamar kiderül, hogy már tíz osztályzat esetén is kezelhetetlenül bonyolult programkódot kapunk. Aggodalomra azonban nincs ok, hiszen itt jönnek képbe a tömbök.

Tömbök megadása

Nevezzük *grades*-nek azt a változót, amely nem egyetlen osztályzatot (*grade*) tartalmaz, hanem egyszerre többet is. Az egyes elemeket a sorszámuk (*indexük*) alapján azonosíthatjuk. A matematikában is vannak indexsel jelölt ismeretlenek, például x_i jelentheti egy sorozat i -edik tagját. A C programozási nyelvben ennek *x[i]* felel meg. Így tehát a

```
grades[5]
```

jelenti a *grades* tömb ötös sorszámú elemét. A tömbök sorszámozása itt nullával kezdődik, azaz a tömb első elemét *grades[0]* tartalmazza. (Érdemesebb tehát inkább „nullás indexű tömbelem” vagy „*grades* nullát” mondani, mintsem a megtévesztő „első elemet”.)

Egy tömbeemet bárhol használhatunk, ahol változónév is állhat. Értékül adhatjuk például egy másik változónak:

```
g = grades[50]
```

Ez az utasítás a *grades[50]*-et rendeli a *g* változóhoz. Az efféle értékkedés általánosabb formában is használható. Legyen például *i* egy egész változó. Ekkor írhatjuk a következőt:

```
g = grades[i]
```

Ilyenkor a *grades* tömb *i* indexű eleme kerül a *g* változóba. Ha ennek a kódrészletnek a végrehajtásakor *i* értéke 7, akkor az utasítás a *grades[7]*-et tölti a *g* változóba.

Természetesen az értékkedő egyenlőségjel bal oldalán is állhat tömbelem. Tekintsük például a következő sort, melyben a *grades* tömb 100-as elemét 95-re állítjuk be:

```
grades[100] = 95;
```

Vagy általánosabb alakban:

```
grades[i] = g
```

Ennek hatására a *grades* tömb *i* indexű eleme megkapja *g* értékét.

A tömbök használata tömör és hatékony programok írását teszi lehetővé, hiszen az összefüggő adatokat ily módon sokkal könnyebben tudjuk tárolni és kezelní, egy egész tömbön pedig végiglépdelhetünk úgy, hogy egy ciklusváltozót a tömb indexeként végigpörgetünk a kívánt értékeken:

```
for ( i = 0; i < 100; ++i )
    sum += grades[i];
0
```

A fenti ciklus végigjárja a `grades` tömb első száz (0-tól 99-ig terjedő) elemét, és minden egyiket hozzáadja a `sum` változóhoz. A ciklus végeztével a `sum` változóban lesz az első száz tömbelem összege (feltéve, hogy előzőleg a `sum` értéke 0 volt).

Ha tömbökkel dolgozunk, minden tartsuk szem előtt, hogy az elemek sorszámozása nullával indul, és a tömb elemszámánál egyelőre kisebb számmal ér véget.

Az egész számok mellett bármilyen egész kifejezés is használható a tömb indexeként. Legyen például `low` és `high` egy-egy egész változó. Ilyenkor a

```
next_value = sorted_data[(low + high) / 2];
```

utasítás a `(low + high) / 2` által meghatározott sorszámú tömbelemet rendeli hozzá a `next_value` változóhoz. Ha mondjuk `low` értéke 1, `high` értéke pedig 9, akkor `sorted_data[5]` kerül a `next_value`-ba. Arra persze vigyázzunk, hogy amikor `low` értéke 1, `high` értéke azonban már 10, akkor is a `sorted_data[5]` kerül a `next_value`-ba, hiszen a $11 / 2$ egész osztás hányadosa 5.

A változókhöz hasonlóan a tömböket is deklarálni kell, mielőtt használni kezdenénk őket. A deklarációban meg kell adni az elemek típusát (például `int`, `float` vagy `char`), valamint az elemek maximális számát. (A C fordítónak erre az információra azért szüksége, hogy a lefordított program a tömbadatok számára le tudja majd foglalni a szükséges memóriaterületet.)

Az

```
int grades[100];
```

deklaráció például egy olyan tömböt határoz meg, melyben száz egész szám fér el. Ebben a tömbben 0-tól 99-ig sorszámozódnak az elemek. Mindig nagy elővigyázatossággal bánunk az indexeléssel, mert a C fordító nem ellenőrzi a hivatkozott tömbelem sorszámának helyességét! Ha a fent megadott deklaráció után a programban `grades[150]`-re hivatkozunk, akkor nem kapunk hibaüzenetet, de megjósolhatatlan lesz a program viselkedése.

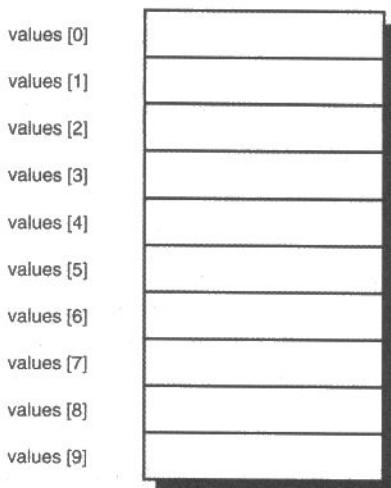
A

```
float averages[200];
```

deklaráció után az `averages` (*átlagok*) tömb kétszáz lebegőpontos értéket tud tárolni. Ugyanígy, az

```
int values[10];
```

annyi helyet foglal le a memóriából, amennyi tíz egész számnak elegendő. Ezt a helyfoglalást jobban el tudjuk képezni, ha szemügyre vesszük a 7.1 ábrát.



7.1 ábra

A *values* tömb számára lefoglalt memóriaterület.

Az egészként, lebegőpontosként vagy char-ként deklarált tömbök elemei ugyanúgy kezelhetők, mint a megfelelő típusú közönséges változók. Lehet nekik értéket adni, ki lehet őket íratni, lehet velük műveleteket végezni (+, -, ...). Az alábbi kód részlet hatására a 7.2 ábra által mutatott értékek kerülnek a *values* tömbbe.

```
int values[10];

values[0] = 197;
values[2] = -100;
values[5] = 350;
values[3] = values[0] + values[5];
values[9] = values[5] / 10;
--values[2];
```

Az első hozzárendelési utasítással 197-et töltünk a *values[0]*-ba. A második és harmadik hozzárendeléssel az elsőhöz hasonló módon -100-at és 350-et tárolunk el a *values[2]*-ben és a *values[5]*-ben. A következő utasítással összeadjuk a *values[0]* és a *values[5]* értékét (ami $197 + 350 = 547$), és betöljük a *values[3]*-ba. Ezt követően a *values[5]* értékét (350-et) tízzel osztjuk, majd az eredményt a *values[9]*-be tesszük. Végül dekrementáljuk a *values[2]*-t, melynek hatására annak értéke -100-ról -101-re csökken.

A 7.1 Listában megtaláljuk a fenti kódrészletet. Az értékadásokat követően egy `for` ciklus-sal végigpásztázzuk a teljes `values` tömböt, és kiírjuk értékeit a képernyőre.

values [0]	197
values [1]	
values [2]	-101
values [3]	547
values [4]	
values [5]	350
values [6]	
values [7]	
values [8]	
values [9]	35

7.2 ábra

A values tömb néhány elemének kezdőértéke

7.1 Lista • Alapműveletek egy tömbben

```
#include <stdio.h>

int main (void)
{
    int  values[10];
    int  index;

    values[0] = 197;
    values[2] = -100;
    values[5] = 350;
    values[3] = values[0] + values[5];
    values[9] =
    values[5] / 10;
    --values[2];

    for ( index = 0; index < 10; ++index )
        printf ("values[%i] = %i\n", index, values[index]);

    return 0;
}
```

7.1 Lista • Kimenet

```
values[0] = 197
values[1] = 0
values[2] = -101
values[3] = 547
values[4] = 0
values[5] = 350
values[6] = 0
values[7] = 0
values[8] = 0
values[9] = 35
```

Az index ciklusváltozó 0-tól 9-ig indexeli a values tömb értékeit, hiszen egy tízelemű tömbnek (a nulladikat is beleszámítva) eddig terjednek a sorszámai. A tömbnek van öt olyan eleme (1., 4., 6., 7., 8.), melyeknek nem adtunk kezdőértéket, így ezek értéke nem meghatározott. Bár a kimenet szerint értékük nulla, az inicializálatlan változók vagy tömbelemek tartalma definiálatlan, így értékükre nem támaszkodhatunk a programban.

Tömbelemek felhasználása számlálóként

Lássunk egy gyakorlatiasabb példát. Tegyük fel, hogy telefonos közvélemény-kutatást végeztünk, melynek során ötezer ember értékelt tízes skálán egy tévéműsort. Van tehát 5000 1 és 10 közé eső számértékünk. Elemezni szeretnénk az eredményeket. Készítsünk egy hisztogramot, vagyis számláljuk össze, hogy hányan szavaztak 1-re, hányan 2-re stb. egészen 10-ig.

Nem teljesen reménytelen feladat, ám meglehetősen izzadságos iparosmunka lenne végignézni az összes szavazatot, és „kézileg” összeszámolni az egyes eseteket. Ha pedig egy általánosabb problémát tekintjük, ahol tíznél több eset is lehetséges (például a válaszadók életkorát is megkérdeztük), akkor még jobban látszik, hogy ésszerűlen lenne ilyen irányból nekikezdeni a megoldásnak.

Olyan programot szeretnénk tehát írni, amely összeszámolja az egyes szavazattípusokat. Az első ötlet tíz számláló létrehozása: szavazat_1, szavazat_2... szavazat_10.

Az egyes szavazatokat végignézve inkrementálnánk az aktuális változót. Ha azonban tíznél sokkal több eset van, már nem tűnik elegánsnak az így megírt program. Tömbök használatával sokkal áttekinthetőbben megoldható az összesítés.

Hozzunk létre egy ratingCounters (*szavazatSzámlálók*) tömböt. Ennek minden az olyan sorszámú elemét fogjuk inkrementálni, amennyi az éppen vizsgált szavazat értéke. A 7.2 Listában csak 20 szavazattal dolgozunk, részben helytakarékkosságból, részben pedig azért, mert a számításigényes problémák megoldásakor minden érdemes egy próba-változatot is készíteni kisebb elemszámra. Az esetleges hibákat így sokkal könnyebb észrevenni, lokalizálni és kijavítani.

7.2 Lista • Számlálótömb létrehozása

```
#include <stdio.h>

int main (void)
{
    int ratingCounters[11], i, response;

    for ( i = 1; i <= 10; ++i )
        ratingCounters[i] = 0;

    printf ("Adja meg a húsz szavazat értékét:\n");

    for ( i = 1; i <= 20; ++i ) {
        scanf ("%i", &response);

        if ( response < 1 || response > 10 )
            printf ("Érvénytelen szavazat: %i\n", response);
        else
            ++ratingCounters[response];
    }

    printf ("\n\nSzavazat      Előfordulási száma\n");
    printf ("-----      ----- \n");

    for ( i = 1; i <= 10; ++i )
        printf ("%4i%14i\n", i, ratingCounters[i]);

    return 0;
}
```

7.2 Lista Kimenet

Adja meg a húsz szavazat értékét:

6
5
8
3
9
6
5
7
15

Érvénytelen szavazat: 15

5
5
1
7
4

```
10
5
5
6
8
9
```

Szavazat	Elofordulási szám a
1	1
2	0
3	1
4	1
5	6
6	3
7	2
8	2
9	2
10	1

A ratingCounters tömb 11-féle értéket tud tárolni. Joggal merül fel a kérdés, hogy miért ez a pazarlás, amikor csak tízféle helyes válasz létezik? A válasz a szavazatok összeszámlálásának módjában rejlik. Az érvényes válaszok 1-től 10-ig terjednek, így a program először meggyőződik a leadott szavazat helyességéről. Ha az valóban 1 és 10 közé esik, akkor (az egyszerűség és átláthatóság kedvéért) a szavazat értékének megfelelő indexű számlálóértéket növeli meg. Ha tehát a szavazat értéke 5, akkor a ratingCounters[5]-ös értéke nő meg egygyel. A számlálás végén a ratingCounters[5] így az összes 5-ös szavazatok számát fogja tartalmazni.

Így már bizonyára érhető, hogy miért van 10 helyett 11 eleme a tömbnek. A legmagasabb szavazat értéke 10, és ahhoz, hogy ezt tárolni tudjuk, ratingCounters[10]-ként kell deklarálni a számlálótömböt, melyben így a nulladikkal együtt 11 elem tárolható. Emlékezzünk vissza arra, hogy minden a használható legnagyobb indexnél egygyel több elem tárolható egy tömbökben. Mivel a 0 nem használható szavazatként, a ratingCounters[0] amolyan használaton kívüli terület. A tömbelemeket inicializáló for ciklus nem is foglalkozik ezzel az elemmel, csak az 1-es sorszámtól kezdődően nullázza a ratingCounters elemeit.

A teljesség kedvéért megemlítjük, hogy természetesen úgy is megírható ez a program, hogy pontosan tíz tömbeemet használjon. Mindössze annyit kell tenni, hogy a szavazatok számánál *egygyel kisebb* indexű számlálót növeljük az egyes lépésekben, azaz a ratingCounters[response - 1]-et. Így a ratingCounters[0] tárolná az 1 értékű szavazatokat, a ratingCounters[1] a 2 értékűeket stb. Ez a lehető legtakarékosabb megközelítés lenne, azonban – lássuk be – sokkal átláthatóbb és ésszerűbb az a program, amely minden változtatás nélkül közvetlenül a szavazat értékét használja fel a számlálótömb megfelelő indexének kiválasztásához.

Fibonacci-számok előállítása

Vegyük szemügyre a 7.3 Listát, amely előállítja az első 15 Fibonacci-számot. Próbáljuk megjósolni, mi lesz a kimenete. Milyen kapcsolatban állnak egymással a táblázat számértékei?

7.3 Lista • Fibonacci-számok előállítása

```
//Az első 15 Fibonacci-számot előállító program

#include <stdio.h>

int main (void)
{
    int Fibonacci[15], i;

    Fibonacci[0] = 0;      // definíció szerint
    Fibonacci[1] = 1;      // ez is

    for ( i = 2; i < 15; ++i )
        Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];
    for ( i = 0; i < 15; ++i )
        printf ("%i\n", Fibonacci[i]);
    return 0;
}
```

7.3 Lista • Kimenet

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

Az első két Fibonacci-szám (F_0 és F_1) definíció szerint 0 és 1. A következők pedig az előző két szám összegeként állnak elő, azaz $F_{i+2} + F_{i+1} = F_i$. Így F_2 az előző két Fibonacci-szám, F_0 és F_1 összegeként számítható ki. A programban ennek megfelelően $Fibonacci[2]$ úgy áll elő, mint $Fibonacci[0] + Fibonacci[1]$. Az összeadás egy for ciklusban zajlik, melyben a ciklusváltozó 2-től 14-ig lépdel; F_2 -től F_{14} -ig (vagyis $Fibonacci[2]$ -től $Fibonacci[14]$ -ig) számítjuk ki a Fibonacci-számokat.

A Fibonacci-számoknak számos matematikai és számítástudományi alkalmazása ismertes. A Fibonacci-féle sorozat eredetileg a nyulak szaporodásának modellezésére született. Tegyük fel ugyanis, hogy egy nyúlpár szaporodni kezd. A második hónaptól kezdve minden hónapban születik egy újabb nyúlpár. A fiatal nyulak két hónap múlva válnak ivaréretté – akkor tudnak újabb nyulakat szülni. Ha feltételezzük, hogy mindegyik nyúl életben marad, akkor hány pár nyúlra szaporodik egy év alatt a kezdeti pár? Vegyük észre, hogy az n . hónap végén F_{n+2} nyúlpárunk van. A 7.3 Lista kimenete szerint a 12. hónap végén 377 nyúlpár él.

Prímszámkeresés tömb használatával

Ideje, hogy visszatérjünk a 6. fejezetben megírt prímszámkereső programunkhoz, tömb használatával ugyanis jóval hatékonyabban is megvalósítható. A 6.10A Listában egy n szám prímtulajdonságát úgy vizsgáltuk meg, hogy 2-től kezdődően egészen $n-1$ -ig az összes számmal megpróbáltuk elosztani n -t. A 6. fejezet 7. gyakorlata rámutatott két kézenfekvő javítási lehetőségre, ám még ezekkel együtt sem mondható igazán hatékonynak ez a fajta a prímkeresés. A hatékonyúság kérdése ötven prímszám esetén még nem okoz gondot, de ha mondjuk az első egymillió prímszámot keressük, akkor már húsa vágóan fontossá válik.

Eljárásunkon a következő észrevétellel javíthatunk: egy szám akkor prímszám, ha nem osztható önmagától különböző prímszámmal. minden összetett szám felbomlik ugyanis önmagánál kisebb prímszámok szorzatára. (a 20 például felbontható, mint $2 \cdot 2 \cdot 5$). Ennek a segédtételnek köszönhetően sokkal hatékonyabb programot tudunk írni: minden számot elegendő elosztani a nála kisebb prímszámokkal (melyeket már korábban megtaláltunk). A „megtalált” prímszámokat szeretnénk megőrizni – itt siet segítségünkre a tömbök használata.

A prímkereső program tovább javítható azzal a meggondolással, hogy ha egy számnak van valódi osztója, akkor az biztosan nem nagyobb a szám négyzetgyökénél. Elegendő tehát megvizsgálni a számok négyzetgyökénél nem nagyobb prímosztókat.

A 7.4 Lista ezeket a meggondolásokat is felhasználja az ötvennél kisebb prímszámok megkereséséhez.

7.4 Lista • A prímszámkeresés javított változata

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

// A prímszámkereső program javított változata

int main (void)
{
    int p, i, primes[50], primeIndex = 2;
```

```

bool isPrime;

primes[0] = 2;
primes[1] = 3;

for ( p = 5; p <= 50; p = p + 2 ) {
    isPrime = true;

    for ( i = 1; isPrime && p / primes[i] >= primes[i]; ++i )
        if ( p % primes[i] == 0 )
            isPrime = false;

    if ( isPrime == true ) {
        primes[primeIndex] = p;
        ++primeIndex;
    }
}

for ( i = 0; i < primeIndex; ++i )
    printf ("%i ", primes[i]);

printf ("\n");

return 0;
}

```

7.4 Lista • Kimenet

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

A

$p / \text{primes}[i] \geq \text{primes}[i]$

kifejezés első látásra legtöbbünknek meglepő. Ebben rejlik azonban az előző bekezdésből származó ötlet, mely szerint elég a p szám négyzetgyököig elmenni a prímosztókkal, azaz a $\text{primes}[i]$ -vel. (Mindaddig, amíg a $\text{primes}[i]$ kellően kicsi, addig a p -t vele osztva kellően nagy szám adódik, és tovább fut a ciklus. Színtiszta matematika.)

A 7.4 Listában a deklarációk után a primes tömb első két eleme felveszi a 2-es és a 3-as (azaz a két legkisebb prímszáml) értéket. A tömböt 50-es méretűre deklártuk, bár ennyi hely nyilvánvalóan nem kell majd a prímszámok tárolásához. A primeIndex változó először a 2 értéken áll – ez jelzi a következő prímszám indexét a primes tömbben. A for ciklus ezután 5-től 50-ig fut, de csak a páratlan számokat vizsgáljuk. Az isPrime logikai változó igazra állítása után rögtön egy belső for ciklus kezdődik. Az eddig megtalált (és a primes tömbben tárolt) prímszámokkal sorban megpróbáljuk elosztani p értékét. Az i

ciklusváltozó 0 helyett 1-ről indul, hisz a paritást (a 2-vel, azaz a `primes[1]`-el való osztathóságot) nem kell vizsgálni, úgyis csak a páratlan számokon lépdelünk végig. A (2 feletti) páros számokról jogosan nem vélelmezzük a prímtulajdonságot. A belső ciklusban `p`-t megpróbáljuk maradék nélkül elosztani `primes[i]`-vel, és ha ez sikerül, akkor az `isPrime` értékét hamisra váljuk. A `for` ciklus addig tart, amíg meg nem győzöttünk a szám összetett voltáról (`isPrime` hamisságáról), vagy el nem értünk `p` négyzetgyökéig a prímosztókkal (`primes[i]`-vel).

A belső `for` ciklusból kilépve megnézzük, hogy igaz maradt-e az `isPrime`, azaz prímsámot találtunk-e. Ha igen, akkor `p` értékét beírjuk a `primes` tömb következő (`primeIndex` változóban tárolt sorszámú) helyére.

Miután végére értünk a külső ciklusnak, azaz minden `p` értéket végignéztünk, a program kiírja a `primes` tömb értékeit, vagyis a megtalált prímszámokat. A kiírási ciklusváltozó 0-tól `primeIndex-1`-ig lépdel; a `primeIndex` ugyanis a következő prímszám `primes`-beli sorszámát tárolja.

Tömbök inicializálása

Ahogy a változóknak kezdőértéket lehet adni a deklarációkor, ugyanúgy a tömböket is el lehet látni kezdőértékkel. Ez a kezdőértékek egyszerű felsorolásából áll, kezdve a legelső értékkel. Az értékek kapcsos zárójelben sorakoznak, vesszővel elválasztva egymástól. Az

```
int counters[5] = { 0, 0, 0, 0, 0 };
```

utasítás egy `counters` nevű, egészekből álló tömböt ad meg, melyek mindegyike nulla kezdőértékkel rendelkezik. Hasonlóképp, az

```
int integers[5] = { 0, 1, 2, 3, 4 };
```

utasítás során az `integers[0]` értéke 0-ra, az `integers[1]` értéke 1-re stb. állítódik be.

Karakterekből álló tömbök is inicializálhatóak hasonló módon, azaz a

```
char letters[5] = { 'a', 'b', 'c', 'd', 'e' };
```

utasítás létrehozza a `letters` tömböt, melynek elemei sorban az a, b, c, d és e karaktereket veszik fel.

Nem kell az összes tömbeemet inicializálni. Ha csak néhány adat van megadva, akkor csak a nekik megfelelő első néhány tömbelem kap kezdőértéket, a többi pedig nulla értéket kap.

Így a

```
float sample_data[500] = { 100.0, 300.0, 500.5 };
```

utasítás hatására a sample_data tömbnek csak az első három eleme kapja meg a felsorolt kezdőértéket, a többi 497 elem nullázódik.

Szögletes zárójelben megadhatjuk a beállítani kívánt elem sorszámát is, így kötetlenül megválaszthatjuk az értékadások sorrendjét. Az alábbi kódrészlet hatása megegyezik az előzőével:

```
float sample_data[500] = { [2] = 500.5, [1] = 300.0, [0] = 100.0 };
```

Az

```
int x = 1233;
int a[10] = { [9] = x + 1, [2] = 3, [1] = 2, [0] = 1 };
```

utasítássor egy tízelemű tömböt ad meg, melynek utolsó elemét $x+1$ -re (azaz 1234-re) inicializáljuk, valamint az első három elemét 1-re, 2-re, illetve 3-ra.

A C programozási nyelv (a nullázáson kívül) sajnos nem tesz lehetővé rövid inicializációt. Ha egy ötszáz elemű tömb minden elemének 1 kezdőértéket szeretnénk adni, akkor kénytelenek vagyunk minden kiírni. Ilyen esetben érdemesebb egy megfelelő `for` ciklust lefuttatni a programban.

A 7.5 Lista két hasznos módszert mutat egy tömb elemeinek inicializálására.

7.5 Lista • Tömbök inicializálása

```
#include <stdio.h>

int main (void)
{
    int array_values[10] = { 0, 1, 4, 9, 16 };
    int i;

    for ( i = 5; i < 10; ++i )
        array_values[i] = i * i;

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

    return 0;
}
```

7.5 Lista • Kimenet

```
array_values[0] = 0
array_values[1] = 1
array_values[2] = 4
array_values[3] = 9
array_values[4] = 16
array_values[5] = 25
array_values[6] = 36
array_values[7] = 49
array_values[8] = 64
array_values[9] = 81
```

Az `array_values` deklarációjában megadtuk az első öt elem értékét (a sorszám négyzeteként, például a hármas indexű elem értéke 9). Az első `for` ciklusban értékadással oldjuk meg az 5-től 9-ig terjedő sorszámú elemek beállítását, szintén a sorszám négyzetét rendelve az elemekhez. A másik `for` ciklus egyszerűen végigjárja a tíz tömbeemet és kiírja az értékeket a képernyőre.

Karaktertömbök

A 7.6 Lista bemutatja, hogy hogyan használhatunk karaktertömböket. A program egy ponton kiemelt figyelmet érdemel. Hol van ez a pont?

7.6 Lista • A karaktertömbök – bevezetés

```
#include <stdio.h>

int main (void)
{
    char word[] = { 'H', 'e', 'l', 'l', 'o', '!' };
    int i;

    for ( i = 0; i < 6; ++i )
        printf ("%c", word[i]);

    printf ("\n");

    return 0;
}
```

7.6 Lista Kimenet

Hello!

A program legérdekesebb része a `word` karaktertömb deklarációja, amelyben nincs szó a tömb méretéről. A C nyelv megengedi, hogy az elemszám megadása nélkül inicializál-

junk egy tömböt. Ilyenkor a megadott kezdőértékek száma dönti el a létrehozandó elemek számát. A 7.6 Listában 6 elemet adtunk meg a word tömb számára, így ennek méretét a C fordító 6-ra állítja be.

Ez csak akkor működik, ha rögtön a deklarációkor megadjuk a tömb néhány elemét. Ha nincsenek kezdőértékek felsorolva, akkor meg kell adni a tömb méretét.

Amennyiben a kezdőértékek felsorolásakor megadunk indexeket is, akkor a legnagyobb index fogja meghatározni a tömb méretét, mint itt is:

```
float sample_data[] = { [0] = 1.0, [49] = 100.0, [99] = 200.0 };
```

A sample_data száz értéket tud majd tárolni, mert a legnagyobb megadott index a 99.

Számrendszerek közti váltás tömb használatával

Következő példánkban tovább folytatjuk az ismerkedést a számokból és karakterekből álló tömbökkel. Olyan programot fogunk írni, amely egy tízes számrendszerbeli számot átvált egy másik (maximum 16-os) számrendszerbe. Bemenetként meg kell adnunk az átváltandó számot és a másik számrendszer alapszámát. A program ez alapján elkészíti a kívánt számrendszerbeli számot, majd kiírja a képernyőre.

A program megírása előtt át kell gondolnunk a számrendszerek közti átváltás módszerét. Próbáljuk meg vázlatosan összefoglalni az átváltási algoritmust. A másik számrendszerbeli szám utolsó számjegyét úgy kaphatjuk meg, hogy vesszük a kiindulási számnak a (számrendszer alapszámával történő) osztási maradékát. Osszuk el ezután a számot a számrendszer alapszámával, és hagyjuk el a maradékot. Ismételjük ezeket a lépéseket addig, amíg az osztás után „el nem fogy” a szám (azaz nullát nem kapunk).

Ez a módszer jobbról kezdődően adja meg az átváltott szám számjegyeit. Lássuk a működését a következő példán. Tegyük föl, hogy tízesből kettes számrendszerbe szeretnénk átváltani egy számot. A 7.1 táblázat mutatja a cél eléréséhez szükséges algoritmus lépéseiit.

7.1 táblázat • Egy tízes számrendszerbeli szám átalakítása kettes számrendszerbe.

Szám	Maradék 2-vel	A szám/2
10	0	5
5	1	2
2	0	1
1	1	0

Az átalakítás után kapott szám az 1010, hiszen (a fentiek értelmében) a számjegyeket a „Maradék 2-vel” oszloból kell elolvasnunk, mégpedig lentről felfelé stb.

Ha ezt az algoritmust szeretnénk megvalósítani egy programmal, meg kell fontolnunk néhány dolgot. Először is, a számjegyek kiszámítása fordított sorrendben történik. Nem tehetjük meg azt, hogy ezeket egyszerűen keletkezésük sorrendjében kiírjuk a képernyőre, hiszen nem várható el a felhasználóktól, hogy a kedvünkért jobbról balra (vagy lentről fölfelé) olvassák el az eredményt. Ezt a problémát könnyen orvosolhatjuk egy tömb segítségével, amiben a közvetlen kiírás helyett először eltároljuk a számjegyeket, aztán a számolás végeztével a szokásos számjegysorrenddel jeleníthetjük meg az eredményt.

A másik megfontolandó dolog az, hogy egészen 16-ig lehetővé kell tenni a számrendszer közti átváltást. A 9-nél nagyobb számjegyeket (10-től egészen 15-ig) az ABC első betűivel szokás jelölni. Itt jön képbe a karaktertömb használata.

Nézzük át a 7.7 Listát, amely minden felvetett kérdést megoldja. Előkerül egy eddig ismeretlen típusmódosító is, a const. Ezt olyan változókra szoktuk használni, melyek értéke konstans, azaz garantáltan nem változik meg a program futása során.

7.7 Lista • Pozitív egész szám átváltása tízes számrendszerből egy másik számrendszerbe.

```
// Pozitív egész átváltása tízes számrendszerből egy másikba

#include <stdio.h>

int main (void)
{
    const char baseDigits[16] = {
        '0', '1', '2', '3', '4', '5', '6', '7',
        '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
    int     convertedNumber[64];
    long int numberToConvert;
    int     nextDigit, base, index = 0;

    // A szám és a számrendszer bekérése

    printf ("Mi az átváltandó szám? ");
    scanf ("%ld", &numberToConvert);
    printf ("Milyen számrendszerbe alakítsuk át? ");
    scanf ("%i", &base);

    // átalakítás a kért számrendszerbe
    do {
        convertedNumber[index] = numberToConvert % base;
        ++index;
    }
```

```

        numberToConvert = numberToConvert / base;
    }
    while ( numberToConvert != 0 ) {

        // fordított sorrendű kiírás

        printf ("Az átalakított szám: ");

        for (--index; index >= 0; --index) {
            nextDigit = convertedNumber[index];
            printf ("%c", baseDigits[nextDigit]);
        }

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

```

7.7 Lista • Kimenet

Mi az átváltandó szám? **10**
 Milyen számrendszerbe alakítsuk át? **2**
 Az átalakított szám: **1010**

7.7 Lista • Kimenet (újrafuttatás)

Mi az átváltandó szám? **128362**
 Milyen számrendszerbe alakítsuk át? **16**
 Az átalakított szám: **1F56A**

A const típusmódosító

A C fordítóprogram lehetővé teszi, hogy a `const` típusmódosítóval adjuk meg azokat a változókat, amelyek értékét változatlanul (azaz *konstansként*) szeretnénk tartani a program futása során. Ha egy ilyen változó értékét megkíséreljük megváltoztatni a programban, vagy akár inkrementálni/dekrementálni próbáljuk, akkor a fordítóprogram még a fordítási időben visszajelzést ad(hat). A konstansok használatának másik következménye, hogy az így megadott változók (a program kódjához hasonlóan) a csak-olvasható memóriaterületen kapnak helyet.

Lássunk néhány példát konstansok használatára.

```
const double pi = 3.141592654;
```

Ez az utasítás a `pi` változót konstansként deklarálja. Ennek hatására a fordítóprogram (hélyesen) nem fogja megengedni a π értékének megváltoztatását. Próbáljuk csak meg beírni egy programba a következő sort:

```
pi = pi / 2;
```

Akkor a `gcc` fordító erre a következő hibaüzenetet adja:

```
foo.c:16: warning: assignment of read-only variable 'pi'
foo.c:16: figyelmeztetés: hozzárendelés a csak-olvasható 'pi' változóhoz
```

Térjünk vissza a 7.7 Listához. A `baseDigits` karaktertömb tárolja az átalakításhoz felhasználható 16 számjegyet, melyet konstans tömbként deklarálunk, hiszen értékük változatlan marad a program futása során. Ez a típusmódosítás a program érhetőségét is javítja.

A `convertedNumber` tömb méretét 64-nek adjuk meg. Ez tárolja az átalakítandó szám számjegyeit. 64 számjegy még a legnagyobb „long int” típusú szám legkisebb (2-es) számrendszerbeli átalakításához is elég szinte minden számítógépen. A `numberToConvert` változó „long int” típusuként szerepel, hogy nagy számokat is át lehessen alakítani. A `base` és az `index` nevű változók `int` típusúak. Az egyik a kért számrendszer alapszámát, a másik a `convertedNumber` tömb aktuális sorszámát őrzi.

A program futása úgy kezdődik, hogy a felhasználó beírhatja az átváltandó számot és a kívánt számrendszer alapszámát. A `scanf` függvény a `%ld` formátumjelző karakterlánckal kéri be a „long int” típusú számot. A konverzió egy do ciklusban történik, ugyanis esetünkben igen előnyös, ha a ciklusmag legalább egyszer mindenkorábban lefut. Akkor is létrejön legalább egy számjegy a `convertedNumber` tömbben, ha a felhasználó nullát ad meg átalakítandó számként.

A ciklusban a soron következő számjegyet a `numberToConvert`-nek a `base`-zel való osztási maradékából számítjuk ki. Az eredményt a `convertedNumber` tömbben tároljuk, az `index` értékét pedig inkrementáljuk. Miután elosztottuk a `numberToConvert`-et a `base`-zel, a do ciklus ellenőrzi a ciklusfeltételt. Ha a `numberToConvert` értéke 0, a ciklus kilép; egyébként pedig tovább folyik a ciklus, hogy kiszámítsa a következő számjegyet.

A do ciklus befejeztével az átváltott szám jegyeinek számát az `index` változó tárolja. Pontosabban eggyel többet mutat a kelleténél, mivel az utolsó ciklusban feleslegesen növeltük meg. Így azzal kezdjük a kiírást, hogy a for ciklus inicializációs részében eggyel csökkentjük az értékét. A számjegyek helyes sorrendű kiírásának érdekében a for ciklus *fordított* sorrendben lépdel végig a `convertedNumber` tömb elemein.

A `convertedNumber` tömb minden egyes számjegyét hozzárendeljük a `nextDigit` változóhoz. Ez lesz a `baseDigits` tömb megfelelő elemére mutató `index`, ugyanis 9-en felül a számok értéke helyett az ABC megfelelő betűjét kell kijeleznünk egyetlen karakter formájában. 0-tól 9-ig pontosan a nekik megfelelő '0', ... '9' karakterek állnak a `baseDigits` tömbben, 10-től 15-ig pedig rendre az 'A', 'B', 'C'... 'F' karakterek. Ha például a `nextDigit` változó értéke 10, akkor a `nextDigit[10]`, azaz az 'A' betű jelenik meg a képernyőn. Ha pedig egy 10 alatti szám, például 8 van a `nextDigit` változóban, akkor a `nextDigit[8]`, konkréten az '8'-as számjegy íródik ki.

Ha az index értéke 0 alá süllyed, a for ciklus véget ér, a program pedig egy újsor karakter megjelenítése után kilép.

Érdemes megjegyezni, hogy a programot a nextDigit változó használata nélkül is meg lehetett volna írni: a printf meghívásakor közvetlenül a convertedNumber [index] értékét is meg lehetett volna adni a baseDigits indexeként. A kiíratáskor a

```
baseDigits[ convertedNumber[index] ]
```

kifejezést használva a 7.7 Listával megegyező hatást érhettük volna el. Ez a kódrészlet azonban jóval áttekinthetetlenebb, mint a nextDigit-et használó változat.

Nem hagyhatjuk említés nélkül a program néhány elnagyolt részletét. Nem ellenőriztük például, hogy valóban 2 és 16 között van-e a számrendszer alapszáma. Ha a felhasználó nullát ad meg alapszámként (base), akkor a do ciklusban nullával való osztás történik. Ezt soha nem szabad megengedni. Ráadásul base=1 megadásakor a program végtelen ciklusba jutna, mert a numberToConvert értéke soha nem csökkenne 0 alá. Ha viszont 16-nál nagyobb alapszámot ad meg a felhasználó, akkor könnyen előfordulhat, hogy általában a baseDigits tömbhatárait a program. Ez egy másik buktató: a tömbhatárokat nem figyeli a fordító, nekünk kell tehát ellenőriznünk.

A 8. fejezetben újraírjuk majd ez a programot, és kijavítjuk az említett hiányosságokat is. Most azonban terjessük ki eddigi tömb-fogalmunkat egy érdekes irányba.

Többdimenziós tömbök

Az eddig használt tömbök mind egydimenziósak (azaz vektorok) voltak. A C programozási nyelv lehetővé teszi a többdimenziós tömbök használatát is. Ebben a részben a kétdimenziós tömbökről lesz szó.

A kétdimenziós tömbök használata természetes módon adódik akkor, amikor mátrixokkal (számtáblázatokkal) kell dolgoznunk. Tekintsük a 7.2 táblázatban látható 4 x 5-ös mátrixot.

7.2 táblázat • Egy 4 x 5-ös mátrix

10	5	-3	17	82
9	0	0	8	-7
32	20	1	0	14
0	0	8	7	6

A mátrixok elemeire legtöbbször két koordinátájukkal (sor, oszlop) hivatkozunk, melyet a mátrix betűjele mellé írt kettős indexssel jelölünk. Az M mátrix $M_{i,j}$ eleme tehát az az szám, ami az i -edik sor j -edik oszlopában van; i 1-től 4-ig, j 1-től 5-ig terjedhet.

Az $M_{3,2}$ tehát a 20-ra utal, mely a mátrix 3. sorának 2. oszlopában található. Az $M_{4,5}$ pedig a 4. sor 5. oszlopában található (azaz 6).

A C-ben hasonlóképp hivatkozhatunk egy kétdimenziós tömb valamely elemére. Azonban a C a 0-tól kezdi a számlálást, így a mátrix első sora valójában a 0-ás sor, és az első oszlop a 0-ás sorszámú oszlop. Nézzük az előző mátrixunkat olyan módon, hogy láthatók legyenek a sorok és oszlopok sorszámai.

7.3 táblázat • Az előző 4 x 5-ös mátrix C nyelvű megfelelője

oszl(j)	0	1	2	3	4
sor(i)					
0	10	5	-3	17	82
1	9	0	0	8	-7
2	32	20	1	0	14
3	0	0	8	7	6

Az $M_{i,j}$ matematikai jelölésnek a C programnyelvben az $M[i][j]$ felel meg. Az első index a sor, a második az oszlop megfelelője. Így a

```
sum = M[0][2] + M[2][4];
```

kifejezés összeadja a 0. sor 2. elemét (azaz -3-at) és a 2. sor 4. elemét (azaz 14-et), az eredményt (11-et) pedig a sum változóban tárolja.

A kétdimenziós tömböket az egydimenziósakhoz hasonlóan deklarálhathatjuk. A következő deklaráció M-et kétdimenziós tömbként határozza meg, melynek 4 sora és 5 oszlopa lehet, azaz összesen húsz (egész értéket tároló) eleme:

```
int M[4][5]
```

A kétdimenziós tömbök inicializációja az egydimenziósakéhoz hasonlóképp történhet. A vesszővel elválasztott értékeket soronként kell kapcsos zárójelbe foglalni, és ezeket az egységeket (vesszővel elválasztva) egy külső kapcsos zárójelben kell megadni. Az M tömböt például az alábbi inicializációval tudjuk a 7.3 táblázatban szereplő értékeknek megfelelően feltölteni:

```
int M[4][5] = {
    { 10, 5, -3, 17, 82 },
    { 9, 0, 0, 8, -7 },
```

```
{ 32, 20, 1, 0, 14 },
{ 0, 0, 8, 7, 6 }
};
```

Figyeljük meg a fenti inicializációt: az értékek sorait képező kapcsos zárójeleket vesszővel választjuk el; az utolsó után azonban nem teszünk vesszőt. A belső zárójelek használata nem kötelező. Ha elhagyjuk a belső zárójeleket, akkor is soronként történik a feltöltés. Így a következő inicializáció megegyezik az előzővel:

```
int M[4][5] = { 10, 5, -3, 17, 82, 9, 0, 0, 8, -7, 32,
                20, 1, 0, 14, 0, 0, 8, 7, 6 };
```

Az egydimenziósak inicializációjához hasonlóan itt sem kötelező az összes elemnek kezdőértéket adni. Így az

```
int M[4][5] = {
    { 10, 5, -3 },
    { 9, 0, 0 },
    { 32, 20, 1 },
    { 0, 0, 8 }
};
```

utasítás csak a sorok első három eleméhez rendeli hozzá az itt meghatározott értékeket, a többi elem 0 értékre állítódik. Ilyen esetben azonban kötelező megadni a belső kapcsos zárójeleket is, mert egyébként sorban haladna az értékadás az első két sor elemein át egészen a harmadik sor első két eleméig. Próbáljuk ki ezt a változatot is!

Indexek is megadhatók az inicializációban, az egydimenziós tömbök esetéhez hasonlóan. Így az

```
int matrix[4][3] = { [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

deklaráció csak a három meghatározott elem értékét állítja be a megadott számértékekre, a többi elem nullázódik.

Változó méretű tömbök¹

A C programozási nyelv támogat egy olyan lehetőséget is, amely szerint nem kötelező előre rögzíteni a használandó tömbök méretét.

Korábban már láttuk, miként lehet egy adott méretű tömböt deklarálni. Lehet azonban változó méretű tömböket is használni. A 7.3 Lista programja például csak az első 15

¹ Nem minden fordítóprogram támogatja a változó méretű tömbök használatát. Érdemes ellenőrizni a dokumentációt, mielőtt erre a lehetőségre építve programot írnánk.

Fibonacci-számot számítja ki. De mi a teendő, ha az első 100 vagy 500 Fibonacci-számra van szükségünk? Megoldható-e az, hogy maga a felhasználó adja meg egy tömb méretét? A 7.8 Listában választ kapunk ezekre a kérdésekre.

7.8 Lista • Fibonacci-számok kiszámítása változó méretű tömbbel

```
// Fibonacci-számok generálása változó méretű tömb használatával

#include <stdio.h>

int main (void)

{
    int i, numFibs;

    printf ("Hány Fibonacci-számot szeretne kiszámítani?
            ↪ (1 és 75 között)? ");
    scanf ("%i", &numFibs);

    if (numFibs < 1 || numFibs > 75) {
        printf ("Sajnos nem megfelelő a megadott szám.\n");
        return 1;
    }

    unsigned long long int    Fibonacci[numFibs];

    Fibonacci[0] = 0;           // definíció szerint
    Fibonacci[1] = 1;           // szintén

    for ( i = 2; i < numFibs; ++i )
        Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];

    for ( i = 0; i < numFibs; ++i )
        printf ("%llu ", Fibonacci[i]);

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

7.8 Lista • Kimenet

```
Hány Fibonacci-számot szeretne kiszámítani? (1 és 75 között)? 50
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229
832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817
39088169 63245986 102334155 165580141 267914296 433494437 701408733
1134903170 1836311903 2971215073 4807526976 7778742049
```

A 7.8 Lista több ponton is figyelmet érdemel. Az i és numFibs változókat egészként deklaráljuk – ez utóbbi tárolja a generálandó Fibonacci-számok számát. Értéke 1 és 75 között lehet. A felhasználói bemenetet ellenőrizzük, ami megszívlelendő programozói gyakorlat. Ha a beadott érték kívül esik a megengedett intervallumon (azaz 1-nél kisebb vagy 75-nél nagyobb), akkor a program egy visszajelző üzenet után (1-es visszatérési érték adásával) kilép. A return utasítás hatására ugyanis a program azonnal befejezi futását, figyelmen kívül hagyva minden további utasítást. Ahogy a 3. fejezetben már említettük, a nullától eltérő visszatérési érték „közmegegyezés szerint” azt jelzi, hogy a program valamilyen hibával ért véget. Ezt a tényt egy másik programmal könnyen ellenőrizhetjük.

A kívánt számérték beadása után programunk az

```
unsigned long long int Fibonacci[numFibs];
```

sorral folytatódik.

A Fibonacci tömb elemszámát a numFibs változó értéke határozza meg. Az így deklarált tömböt változó méretű tömbnek hívjuk, mivel a tömb méretét konstans kifejezés helyett egy változó adja meg. Korábban említettük, hogy a program bármely részén elhelyezhető deklaráció, csak az a fontos, hogy a változókat még használatuk előtt deklaráljuk. Így tehát ez a deklaráció (bár első ránézésre rossz helyen van) teljesen korrekt. Ennek ellenére nem illik a program közepén elhelyezni deklarációt. A közmegegyezés szerint a program elején, egy helyre csoportosítva történik a változók megadása, amiben – lássuk be – van logika. Ha valaki olvasni kezdi a programot, annak jó, ha egyben látja az előkerülő változók típusát és kezdőértékét és nem menet közben érik mindenféle „váratlan meglepetések”.

A Fibonacci-számok igen gyorsan nőnek, így a tömb elemeit a lehető legnagyobb egész típusúként, vagyis „`unsigned long long int`”-ként adtuk meg. Próbáljuk meghatározni, hogy melyik a legnagyobb Fibonacci-szám, amely a számítógépünkön ilyen típusú változóban tárolható!

A program további része magától értetődő. A kért Fibonacci-számokat kiszámítjuk, majd kiírjuk a képernyőre. Végül a program sikeresen kilép.

A programban látható módszert általában „dinamikus memória foglalásnak” hívjuk; ennek használatával futás közben tudunk a tömbök számára memóriaterületet lefoglalni. Közben a háttérben a `malloc` és a `calloc` szabványos programkönyvtári függvényeket használjuk. A témáról részletesen lesz szó a 17. fejezetben.

Láttuk, milyen hatékony eszközt ad kezünkbe a C nyelv a tömbök használatával. Ami a többdimenziós tömbökkel illeti azokról egy részletesebb példa kapcsán még lesz szó a 8. fejezetben. Itt kerül majd elő a C programozási nyelv egyik legfontosabb fogalma, a függvény is. Mielőtt azonban ebbe belevágnánk, végezzünk el néhány gyakorlatot.

Gyakorlatok

1. Gépeljük be és futtassuk a fejezetben szereplő nyolc példaprogramot! A kimeneteket hasonlítsuk össze a könyvben szereplő programkimenetekkel.
2. Módosítsuk a 7.1 Listát. A tömb elemeit állítsuk 0 kezdőértékre egy `for` ciklus segítségével.
3. A 7.3 Lista csak húsz érték begépelését teszi lehetővé. Módosítsuk úgy a programot, hogy tetszőleges számú bemeneti adatot meg lehessen adni. Mentesítsük a felhasználót attól, hogy előre meg kelljen számlálnia az adatokat. Tegyük lehetővé, hogy a 999 megadása jelezze az adatsor végét! (Ötlet: használjuk a `break` utasítást a ciklusból történő kilépéshez.)
4. Írunk programot, amely kiszámítja tíz lebegőpontos érték átlagát!
5. Mi lesz az alábbi program kimenete?

```
#include <stdio.h>

int main (void)
{
    int numbers[10] = { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    int i, j;

    for ( j = 0; j < 10; ++j )
        for ( i = 0; i < j; ++i )
            numbers[j] += numbers[i];

    for ( j = 0; j < 10; ++j )
        printf ("%i ", numbers[j]);

    printf ("\n");

    return 0;
}
```

6. A Fibonacci-számok kiszámításához nem feltétlenül kell tömböt használni. Elegendő három változó is: kettőben tároljuk az előző két Fibonacci-számot, a harmadikban pedig az aktuálisan kiszámítandót. Írjuk át a 7.3 Listát úgy, hogy ne használunk a számításhoz tömböt. Ekkor azonban rögtön a számítás után ki kell jeleznünk a kiszámított értékeket, különben nem készíthető táblázat.
7. Az eddig tárgyaltaktól eltérő módon is kereshetünk prímszámokat. Írunk programot, mely az alábbi algoritmussal, „Eratosztenész szitájával” keresi meg a 150 alatti prímeket! Mit mondhatunk ennek hatékonyságáról, összehasonlítva a korábban tár-gyalt eljárásokkal?

Eratosztenész szitájának algoritmusá

Feladat: keressük az 1 és n közötti prímszámokat.

1. lépés: Hozzunk létre egy P tömböt, melynek 2. és n . közötti értékeit állítsuk 0-ra.

2. lépés: Állítsuk i értékét 2-re.
3. lépés: Ha $i > n$, vége az algoritmusnak
4. lépés: Ha $P_i = 0$, akkor P_i prímszám.
5. lépés: minden olyan pozitív j -re, melyre $i \cdot j = n$, állítsuk $P_{i \cdot j}$ t 1-re.
6. lépés: Növeljük meg i értékét egyelőre, és térjünk vissza a 3. lépéshöz.

8

Függvények

Minden C program mögött ugyanaz az alapvető fogalom húzódik meg: a függvény fogalma. Az eddigi összes programunk függvényeket használt. A printf vagy a scanf is egy-egy függvény, sőt maga a main is az. Mit is jelent minden? A függvények teszik lehetővé a programok könnyű megírását, olvasását, megértését, nyomkövetését, módosítását és karbantartását. Megérdemel tehát egy fejezetet ebben a könyvben egy olyan eszköz, amely mindenzt biztosítja.

Függvények megadása

Először is meg kell értenünk, mik is azok a függvények. Csak ezután lehet őket hatékonyan használni programjainkban. Kanyarodjunk vissza könyünk legelső programjához. A 3.1 Lista programja kiírta a képernyőre, hogy „Programming is fun” (*Programozni élvezet*).

```
#include <stdio.h>

int main (void)
{
    printf ("Programming is fun.\n");
    return 0;
}
```

Nevezzük printMessage-nek (*üzenetiírónak*) azt a függvényt, ami ugyanezt a feladatot látja el:

```
void printMessage (void)
{
    printf ("Programming is fun.\n");
}
```

Két helyen tér el a fenti main és printMessage függvény: az első és az utolsó sornál.

A függvény első sora négy dologról tájékoztatja a fordítóprogramot. Balról jobbra haladva ezek a következők:

1. Ki hívhatja meg. A 15. fejezetben („Nagyobb programok”) bővebben lesz erről szó.
2. A visszaadott érték típusa
3. A neve
4. Az átadandó paraméterek

A `printMessage` függvény első sora arról tudósítja a fordítóprogramot, hogy a függvénynek nem lesz visszatérési értéke (azaz `üres`, `void` lesz). Neve `printMessage`, és az átadandó paraméterek listája is üres (ez a második `void`). Hamarosan többet is megtudunk erről a különös `void` kulcsszóról.

Talán mondanom sem kell, hogy beszédes függvényneveket választani legalább olyan fontos, mint értelmes változóneveket – ezek nagyban befolyásolják egy program érhetőségét.

A 3. fejezetben már volt szó arról, hogy a `main` szó többletjelentést hordoz: jelzi a fordítóprogram számára, hogy ott kezdődik a végrehajtandó program. *Mindig* lennie kell a C programban `main` függvénynek. Az előző kód részletéhez adnunk kell egy `main` függvényt ahhoz, hogy működő programot kapjunk, ahogy az a 8.1 Listában látható.

8.1 Lista • Függvény írása C-ben

```
#include <stdio.h>

void printMessage (void)
{
    printf ("Programming is fun.\n");
}

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

8.1 Lista • Kimenet

`Programming is fun.`

A 8.1 Lista *két* függvényt tartalmaz: a `printMessage`-et és a `main`-t. A program végrehajtása mindenkor a `main` függvénynél kezdődik. Esetünkben ebben található a

```
printMessage ();
```

utasítás, amely jelzi, hogy ezen a ponton kell lefuttatni a printMessage függvényt. A nyitó és a záró zárójel jelzi a fordítóprogram számára, hogy függvényről van szó; másrészt pedig azt, hogy ez a függvény nem vár semmilyen paramétert (így tehát összhangban van a programban korábban definiált függvénnnyel). Amikor egy függvényhívás történik, akkor a vezérlés a megadott függvényre kerül. A printMessage függvényen belül kerül meghívásra a printf függvény, amely a „Programming is fun” szöveget írja ki a képernyőre. Az üzenet megjelenítése után a printMessage függvény befejeződik (ezt a záró kapcsos zárójel is mutatja), és a vezérlés visszatér a főprogramba (main-be). A program az aktuális függvényhívás után folytatódik. Lehetőség lenne arra is, hogy a printMessage függvény végére elhelyezzünk egy return; utasítást. A printMessage függvénynek nincs visszatérési értéke, így nem kell semmit sem megadnunk a return után. A return megadása esetünkben nem kötelező. Az üres return utasítás egyenértékű azzal, mint amikor return nélkül éri el a vezérlés a függvény végét. Más szóval ugyanaz a hatása, ha üres return megadásával vagy anélkül érjük el a printMessage függvény végét.

Mint már említettem, a különféle függvények meghívása ezen a ponton már nem újdonság az olvasó számára. A printf és a scanf is egy-egy függvény. Csak annyiban különböznek a saját függvényeinktől, hogy nem nekünk kell őket megírni, hanem a szabványos C függvénykönyvtár részét képezik. Amikor a printf segítségével írunk ki valamit a képernyőre, akkor a vezérlés átadódik a printf függvénynek, majd feladata végeztével visszatér (return) a főprogramhoz. A program minden esetben a függvényhívást követő utasítással folytatódik.

Próbáljuk kitalálni, mit csinál a 8.2. Listában látható program.

8.2 Lista • Függvények hívása

```
#include <stdio.h>

void printMessage (void)
{
    printf ("Programming is fun.\n");
}

int main (void)
{
    printMessage ();
    printMessage ();

    return 0;
}
```

8.2 Lista • Kimenet

Programming is fun.
Programming is fun.

Programunk végrehajtása itt is a `main` függvényel kezdődik, melyben kétszer hívjuk meg a `printMessage` függvényt. Ennek megfelelően kétszer jelenik meg a képernyőn a "Programming is fun" szöveg, és minden alkalommal visszatér a vezérlés a főprogramba. Végül a program futása véget ér.

Lássunk még egy „alkalmazást”, melyben a `printMessage` függvényt használjuk. Próbáljuk előre kitalálni, mi lesz a 8.3. Listában látható program kimenete.

8.3 Lista • Függvények többszörös hívása

```
#include <stdio.h>

void printMessage (void)
{
    printf ("Programming is fun.\n");
}

int main (void)
{
    int i;

    for ( i = 1; i <= 5; ++i )
        printMessage ();

    return 0;
}
```

8.3 Lista Kimenet

```
Programming is fun.
```

Paraméterek és lokális változók

A `printf` meghívásakor átadunk a függvénynek bizonyos információkat: a formátumjelző karakterláncot, különféle kiírandó értékeket. Ezeket az átadott adatokat paramétereknek hívjuk. A paraméterek teszik igazán használhatóvá és rugalmassá a függvényeket. A `printMessage` függvényrel ellentétben (amely minden ugyanazt írja ki) a `printf` a programozó szándékának megfelelően, azaz a paraméterek szerint hajtja végre a kiírást.

A programozó is írhat olyan függvényeket, melyek elfogadnak paramétereit. Az 5. fejezetben több programot is írtunk a háromszögszámok kiszámítására. Most definiálunk egy olyan függvényt, amely háromszögszámokat hoz létre. Adjunk neki beszédes nevet: legyen ez a `calculateTriangularNumber` függvény. A függvény paramétereként adjuk

meg azt, hogy hányadik háromszögszámot szeretnénk megtudni. A függvény számítsa ki a megfelelő háromszögszámot, majd írja ki az eredményt. A 8.4 Listában láthatjuk ezt a függvényt, valamint egy main eljárást, amely néhány értékre ki is próbálja.

8.4 Lista • Néhány háromszögszám kiszámítása függvény segítségével

//Az n. háromszögszámot kiszámító függvény

```
#include <stdio.h>

void calculateTriangularNumber (int n)
{
    int i, triangularNumber = 0;

    for ( i = 1; i <= n; ++i )
        triangularNumber += i;

    printf ("%i. háromszögszám: %i\n", n, triangularNumber);
}

int main (void)
{
    calculateTriangularNumber (10);
    calculateTriangularNumber (20);
    calculateTriangularNumber (50);

    return 0;
}
```

8.4 Lista • Kimenet

10. háromszögszám: 55
20. háromszögszám: 210
50. háromszögszám: 1275

A függvényprototípus deklarációja

A calculateTriangularNumber függvény igényel némi magyarázatot. A függvény első sora így néz ki:

```
void calculateTriangularNumber (int n)
```

Ez a függvényprototípus deklarációja, amely tudatja a fordítóprogrammal, hogy a calculateTriangularNumber olyan függvény, amelynek nincs (azaz üres, void) a visszatérési értéke, és egyetlen paramétert várt: az int típusú n-et. A paraméterként választott név (azaz a formális paraméternév), valamint maga a függvény neve bármilyen szintaktikailag megfelelő név lehet – a helyesnek tekintett nevekről a 4. fejezetben már szó volt. Nyilvánvalóan érdemes „beszédes”, tartalmas nevet választani.

A formális paraméterlistában megadott neve(ke)t használhatjuk a függvény törzsében, ha a paraméter(ek)re szeretnénk hivatkozni.

A definiálandó függvény törzse a nyitó kapcsos zárójellel kezdődik. Esetünkben az *n*. háromszögszámot szeretnénk kiszámítani, így szükségünk lesz egy int típusú változóra (*triangularNumber*), amely ezt a háromszögszámot (és menet közben a végeredményhez vezető részeredményeket) tárolja. Előrelátó módon egy int típusú ciklusváltozót is deklarálunk, i néven. Ezeket a változókat ugyanúgy deklaráthatjuk és inicializálhatjuk, ahogy azt már megszoktuk korábbi programjaink main eljárásaiban.

Automatikusan létrejövő lokális változók

A függvényen belüli változókat automatikusan létrejövő lokális változóknak hívjuk, mert a függvény meghívásakor automatikusan létrejönnek, és értékük csak a függvényen belül látszik, azaz „lokálisak”. Más függvény nem tudja elérni a bennük tárolt értéket. Ha kezdőértéket adunk egy ilyen változónak, akkor az *minden* egyes alkalommal hozzárendelődik, amikor csak meghívódik a függvény.

A függvényen belüli változók deklarációjákor precízebb lenne az *auto* kulcsszót használni a típusnév előtt, például így:

```
auto int i, triangularNumber = 0;
```

A C fordító azonban tudja, hogy a függvényen belüli változók alapértelmezésként *auto* típusúak, ezért ezt a típusmódosítót nem is szokás használni; könyvünkben sem fog többször szerepelni.

Példaprogramunkhoz visszatérve: a deklaráció után a függvény kiszámítja a háromszögszámot, majd kiírja az eredményt a képernyőre. A záró kapcsos zárójel mutatja a függvény végét.

A *main* eljárásban először 10-et adunk át a *calculateTriangularNumber* függvény paramétereiként. A vezérlés ezután magára a függvényre kerül. Az egyedüli formális paraméter, *n* szerepében 10 áll. A függvény kiszámítja tehát a tizedik háromszögszámot és kiírja ennek értékét.

Ezután a *calculateTriangularNumber* paramétereként 20-at megadva újra meghívjuk a függvényt. Az előzőhöz hasonlóan most 20 kerül a formális paraméter, *n* szerepébe. A függvény kiszámítja a huszadik háromszögszámot, és ki is írja a képernyőre.

Annak bemutatása végett, hogy egynél több paramétert is elfogadhat egy függvény, a legnagyobb közös osztót kiszámító (5.7 Lista) programunkat most újraírjuk. A legnagyobb közös osztó (*gcd*) kiszámításához két szám kell – ezek szerepelnek majd függvényünk paraméterlistájában. Lássuk tehát a 8.5 Listát!

8.5 Lista • A legnagyobb közös osztót kiszámító program első újraírása

```
/* A program két nemnegatív egész szám
legnagyobb közös osztóját számítja ki */

#include <stdio.h>

void gcd (int u, int v)
{
    int temp;

    printf ("%i és %i legnagyobb közös osztója ", u, v);

    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }

    printf ("%i\n", u);
}

int main (void)
{
    gcd (150, 35);
    gcd (1026, 405);
    gcd (83, 240);

    return 0;
}
```

8.5 Lista • Kimenet

150 és 35 legnagyobb közös osztója 5
 1026 és 405 legnagyobb közös osztója 27
 83 és 240 legnagyobb közös osztója 1

A gcd függvény két egész paramétert vár, melyekre a függvény formális nevük (u és v) révén hivatkozik. A függvényben deklarálunk egy egész típusú, temp nevű változót, majd kiíratjuk u és v értékét egy megfelelő üzenetben. Ezek után a függvény kiszámítja és kiírja u és v legnagyobb közös osztóját.

Két printf utasítást is láthatunk a gcd függvényben. u és v értékét érdemes még azelőtt kiírni, mielőtt elkezdődik a while ciklus, ugyanis ott már megváltozik u és v értéke. A ciklus végén már más értékeket tárol u és v, mint amit áadtunk a függvénynek. Úgy is lehetne orvosolni ezt a problémát, hogy még a while ciklus előtt eltárolnánk u és v ere-

deti értékét két ideiglenes változóban, melyeket azután u megváltozott értékével együtt (amely a ciklus után már a legnagyobb közös osztót tárolja) kiíratnánk egyetlen `printf` utasítással.

Visszatérési érték

A 8.4 és 8.5 Lista függvényei konkrét számításokat végeznek, melyek eredményét végül ki is írják a képernyőre. Gyakran előfordul azonban, hogy nincs szükség a függvény által kiszámított érték azonnali megjelenítésére. A C nyelv igen kényelmes megoldást kínál az ilyen esetekre: a függvény visszatérési értékét tud küldeni az őt meghívó eljárásnak. Ez nem jelent újdonságot, mert a `main` függvényben ezt a módszert eddig is alkalmaztuk. A szintaktikája nem bonyolult:

```
return kifejezés;
```

Az utasítás azt jelzi, hogy a függvény a *kifejezést* szeretné visszaadni a hívó eljárásnak. Vannak programozók, akik szeretnek zárójelet tenni a kifejezés köré – ez csak stílusbeli kérdés.

Az alkalmas `return` utasítás azonban nem elég. A függvény deklarációjakor helyesen kell megjelölni a *visszatérési érték típusát*, melynek helye közvetlenül a függvény neve előtt van. Korábbi példáinkban minden ott állt az `int` kulcsszó a `main` előtt, ami azt jelzi, hogy maga a program egész értéket ad vissza. A következő deklaráció pedig egy lebegőpontos értéket visszaadó függvényt ad meg:

```
float kmh_to_mph (float km_speed)
```

A `kmh_to_mph` függvény egy `float` típusú paramétert vár (melyet a függvénytörzsben `km_speed` néven érhetünk el). Hasonlóképp, az

```
int gcd (int u, int v)
```

deklarációban a `gcd` függvény két egész értéket vár (`u`-t és `v`-t), és egy egész értéket ad vissza. Módosítsuk úgy a 8.5 Listát, hogy ne a függvényben történjen a kiírás, hanem a `gcd` adja vissza a főprogramnak a kiszámított értéket, és ott történjen meg az érték kijelzése. Ennek programja látható a 8.6 Listában.

8.6 Lista • A legnagyobb közös osztó visszatérési értékként történő visszaadása

```
/* Függvényünk két egész szám legnagyobb közös osztóját számítja ki, az eredményt a visszatérési érték tartalmazza */
```

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

```

int gcd (int u, int v)
{
    int temp;

    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }

    return u;
}

int main (void)
{
    int result;

    result = gcd (150, 35);
    printf ("150 és 35 legnagyobb közös osztója %i\n", result);

    result = gcd (1026, 405);
    printf ("1026 és 405 legnagyobb közös osztója %i\n", result);

    printf ("83 és 240 legnagyobb közös osztója %i\n", gcd (83,
240));

    return 0;
}

```

8.6 Lista • Kimenet

150 és 35 legnagyobb közös osztója 5
 1026 és 405 legnagyobb közös osztója 27
 83 és 240 legnagyobb közös osztója 1

A gcd függvény a kiszámított legnagyobb közös osztót (amit a ciklus végén az u változót tárol) visszatérési értékül adja a főprogramnak a

return u

utasítással.

Felmerülhet a kérdés, hogy hogyan lehet felhasználni a visszatérési értéket. A main eljárásban látható, hogy az első két esetben egy változónak adjuk értékül a függvény visszatérési értékét. Pontosabban, a

result = gcd (150, 35);

utasítás meghívja a `gcd` függvényt a 150 és 35 paraméterekkel, és a visszatérési értéket a `result` változóban tárolja.

Példaprogramunk végén látható, hogy nem feltétlenül kell változóhoz rendelni a visszatérési értéket – ez közvetlenül is felhasználható egy megfelelő utasításban. Esetünkben a

```
gcd (83, 240)
```

értékét a `printf` utasítás minden további értékkadás nélkül kiírja.

A C függvények csak egyetlen értéket tudnak a fent leírt módon visszaadni. Más programnyelvektől eltérően a C nem tesz különbséget az eljárás és a függvény között. C-ben csak függvény van, melynek vagy van visszatérési értéke, vagy nincs. Ha a deklarációban nem adunk meg típusjelzést a visszatérési értéket illetően, akkor a C fordító azt feltételezi, hogy a visszatérési érték (ha lesz egyáltalán, akkor) egész típusú. Vannak programozók, akik ezt az alapértelmezést kihasználva nem adnak meg típusjelzést a visszatérési értékhez; ez nem jó gyakorlat, lehetőleg kerüljük el az ilyen megoldásokat. minden függvényhez adjunk meg visszatérési érték típust a függvény fejlécében, már csak az olvashatóság kedvéért is. Így utólag is azonnal látni fogjuk a függvény fejlécére tekintve (a függvény nevén és paraméterein kívül azt is), hogy van-e visszatérési értéke, és milyen típusú.

Említettük korábban, hogy a `void` kulcsszó kifejezetten arra való, hogy rámutasson a visszatérési érték hiányára. Ha ennek megadása után a függvényben mégis megpróbálunk visszatérési értéket adni, akkor a fordítóprogram hibajelzéssel leáll. Tekintsük például a 8.4 Listát, melyben a `calculateTriangularNumber` függvényhez `void` visszatérési értéket deklaráltunk. Ha ezek után a főprogramban

```
number = calculateTriangularNumber (20);
```

kódssorral arra próbálnánk rávenni a programot, hogy mégis adjon visszatérési értéket, a fordítóprogram hibára futna.

A `void` típus tehát az adat *hiányát* hivatott jelezni. Így a `void` visszatérési típusú függvényeknek nincs visszatérési értéke, és nem használhatók fel különféle kifejezésekben.

A 6. fejezetben a döntési szerkezetekkel kapcsolatban írtunk egy programot, amely kiszámítja és kiírja egy szám abszolút értékét. Írunk most egy függvényt, amely a paraméterként megadott szám abszolút értékét adja vissza! A 6.1 Listával ellentétben most ne csak egész számot fogadjunk el, hanem lebegőpontost (`float`) kérjünk, és ugyanilyet adjunk vissza. Lássuk tehát a 8.7 Listát!

8.7 Lista • Az abszolút érték kiszámítása

```
// Az abszolút értéket visszaadó függvény

#include <stdio.h>

float absoluteValue (float x)
{
    if ( x < 0 )
        x = -x;

    return x;
}

int main (void)
{
    float f1 = -15.5, f2 = 20.0, f3 = -5.0;
    int i1 = -716;
    float result;

    result = absoluteValue (f1);
    printf ("result = %.2f\n", result);
    printf ("f1 = %.2f\n", f1);

    result = absoluteValue (f2) + absoluteValue (f3);
    printf ("result = %.2f\n", result);

    result = absoluteValue ( (float) i1 );
    printf ("result = %.2f\n", result);

    result = absoluteValue (i1);
    printf ("result = %.2f\n", result);

    printf ("% .2f\n", absoluteValue (-6.0) / 4 );

    return 0;
}
```

8.7 Lista • Kimenet

```
result = 15.50
f1 = -15.50
result = 25.00
result = 716.00
result = 716.00
1.50
```

Az abszolútérték-számító függvény működése világos. Megvizsgáljuk, hogy az `x` formális paraméter negatív-e. Ha igen, akkor negáljuk az értékét. A kapott eredményt (`x` abszolút értékét) visszatérési értékként küldjük el a `return` utasítással.

Érdemes megfigyelni, hogy milyen vizsgálatoknak vetjük ála az `absoluteValue` függvényt a `main` eljárásban. Először a `-15.5` értéket tároló `f1` változóval hívjuk meg a függvényt, ahol ez az érték `x` változónében kerül felhasználásra. Mivel az `if` utasítás vizsgálata igaz eredményt ad, az `x`-et negáló kifejezés fut le, így `x` értéke átállítódik `15.5`-re. A következő (`return`) utasításban visszaadjuk `x` értékét a főprogramnak – itt betöljtük a `result` változóba, majd pedig kiírjuk.

Amikor az `absoluteValue` függvényen belül megváltoztatjuk `x` értékét, az semmi kihatással nincs az `f1` változóra. Az `f1` átadása csak érték szerint történik meg; értéke átmásolódik az `x` változóba. Így hiába változtatjuk meg az `x` változót, az nem hat vissza az `f1`-re. Ezt a második `printf` utasítás is igazolja, amelyben változatlanul jelenik meg `f1` értéke. Véssük jól az eszünkbe, hogy a paraméterként átadott változók eredeti értékét nincs mód megváltoztatni egy függvényben – minden nem változtatás a másolatot érinti.

Az `absoluteValue` függvény következő két meghívása azt mutatja be, hogy miként használhatjuk algebrai kifejezésekben a függvények visszaadott értékét. `f2` abszolút értékét és `f3` abszolút értékét összeadjuk, és az összeget a `result` változóba töltjük.

Az `absoluteValue` függvény negyedik meghívása azt mutatja be, hogy a függvénynek átadott paraméter típusának meg kell egyeznie a függvény deklarációjakor megadott formális paraméter típusával. Mivel az `absoluteValue` függvény lebegőpontos értéket vár, az egész típusú `i1` változót először lebegőpontossá alakítjuk a (`float`) típusátalakító operátorral. Ha elhagyjuk ezt az operátort, akkor a fordítóprogram automatikusan „oda-képzeli” helyettünk, mert tudja, hogy az `absoluteValue` függvény lebegőpontos számot vár. (Ezt az `absoluteValue` függvény ötödik meghívása igazolja.) Sokkal világosabb azonban az a program, amiben jelezzük az elvárt típusátalakítást, és nem a rendszertől várjuk el annak automatikus véghezvitelét.

Az `absoluteValue` függvény utolsó meghívása arra mutat rá, hogy az aritmetikai kifejezések kiértékelése attól is függ, hogy milyen típusú értékeket adnak vissza a benne szereplő függvények. Minthogy az `absoluteValue` függvény lebegőpontos visszatérési értékű, a fordítóprogram lebegőpontos osztásként kezeli a kifejezésben szereplő osztást. Lebegőpontos számot osztunk egéssel, és ha visszaemlékszik az olvasó, ez lebegőpontos műveletnek minősül, hiszen az egyik operandus lebegőpontos. Ennek megfelelően `-6.0` abszolút értékét (`6.0`-ot) `4`-el osztva `1.5`-et kapunk.

Kezünkben van tehát egy olyan függvény, amely kiszámítja bármilyen lebegőpontos szám abszolút értékét. Abszolútérték-számító függvényünket bárhol tudjuk majd használni, ha szükség lesz rá – ezt meg is tesszük a következő programban (8.8 Lista).

Függvényt hívó függvényt hívó függvény...

Manapság nem jelent nehézséget egy szám négyzetgyökének kiszámítása, ha kéznél van egy számológép. De néhány évtizeddel ezelőtt a diákok komoly módszereket tanultak arra, hogy hogyan lehet közelítő számítással megkapni egy szám négyzetgyökét. Az egyik ilyen közelítés, mely igencsak kínálja magát a számítógépes megvalósításra, a Newton-Raphson iteráció. A 8.8 Listában ezzel az algoritmussal határozzuk meg egy x szám négyzetgyökét.

A Newton-Raphson módszer a következőképp működik: Kiindulunk egy „*tipp*-ból”, amely az x négyzetgyökének becsült értéke. Minél közelebb van a megtippelt érték a tényleges négyzetgyökhöz, annál kevesebb lépéssel eljuthatunk egy pontosabb közelítéshez. A példa kedvéért tekintsük úgy, hogy nem vagyunk jó matematikusok, és minden indulunk ki az 1-es *tipp*-ból.

Az x -et elosztjuk a *tipp*-vel, majd a hányadost hozzáadjuk a *tipp*-hez. Ezt a részeredményt megfelezzük: ez lesz az új *tipp* érték a következő iterációs lépéshez. Újra elosztjuk x -et a *tipp*-el, hozzáadjuk a hányadost a *tipp*-hez és vesszük az eredmény felét. Ezzel az új *tipp* értékkel aztán tovább folyhat a közelítés.

Mivel ezt nem szeretnénk a végtelenségig csinálni, ki kell találni valamilyen küszöböt, amikor abba hagyjuk az iterációt. Az egyes lépések során egyre közelebb kerül a *tipp* az x négyzetgyökéhez, ezért megadható egy pontossági határ, amin belül már elfogadjuk az eredményt. A *tipp* négyzete és az x közti különbség egyre csökken, és ha ez egy adott (ϵ , epsilon) hibakorlát alá kerül, akkor abba hagyjuk a közelítést – ekkor már kellő pontossággal kezünkben van x négyzetgyöke.

Ez a műveletsor az algoritmusok nyelvén is megfogalmazható:

A Newton-Raphson módszer: x négyzetgyökének kiszámítása

1. lépés: Legyen a *tipp* értéke: 1.
2. lépés: Ha $|tipp^2 - x| < ?$, folytassuk a **4. lépéssel**.
3. lépés: Legyen a *tipp* értéke: $(x/tipp + tipp) / 2$, és térjünk vissza a **2. lépéshöz**.
4. lépés: Az x szám négyzetgyökének elfogadható közelítésű értéke: *tipp*.

Fontos, hogy a 2. lépésben a különbség *abszolút értékét* állítsuk szembe $?$ -nal, mivel a *tipp* értéke bármely irányból közelítheti x négyzetgyökét (azaz hol kisebb, hol nagyobb nála).

Kezünkben van egy remek algoritmus – innen már csak egy lépés a négyzetgyököt vonó program megírása. A hibakorlát, azaz $?$ értéke legyen 0.00001. Lássuk tehát a 8.8 Listát!

8.8 Lista • Egy szám négyzetgyökének kiszámítása

```
#include <stdio.h>

// Egy szám abszolút értékét (!) kiszámító függvény

#include <stdio.h>

float absoluteValue (float x)
{
    if ( x < 0 )
        x = -x;
    return (x);
}

// Egy szám négyzetgyökét kiszámító függvény

float squareRoot (float x)
{
    const float epsilon = .00001;
    float guess = 1.0;

    while ( absoluteValue (guess * guess - x) >= epsilon )
        guess = ( x / guess + guess ) / 2.0;

    return guess;
}

int main (void)
{
    printf ("Négyzetgyök (2.0) = %f\n", squareRoot (2.0));
    printf ("Négyzetgyök (144.0) = %f\n", squareRoot (144.0));
    printf ("Négyzetgyök (17.5) = %f\n", squareRoot (17.5));

    return 0;
}
```

8.8 Lista • Kimenet

```
Négyzetgyök (2.0) = 1.414216
Négyzetgyök (144.0) = 12.000000
Négyzetgyök (17.5) = 4.183300
```

Előfordulhat, hogy az utolsó számjegyek eltérnek egyes számítógépeken.

A 8.8 Listához fűznünk kell némi magyarázatot. Az `absoluteValue` függvény megegyezik a 8.7 Listában szereplő, azonos nevű függvénnnyel – ennek definiálásával kezdődik a program.

Ezután következik a `squareRoot` függvény, mely egyetlen paramétert vár: `x`-et, és a visszatérési értéke egy lebegőpontos szám. A függvénytörzsben két lokális változót használunk: az `epsilon`-t és a `guess`-t (*tipp*). A hibakorlát szerepét betöltő `epsilon` értékét 0.00001-re inicializáljuk, `guess` értékét pedig (amelyben a négyzetgyök közelítő értéke lesz) 1.0 kezdőértékre állítjuk. A két inicializáció mindenkorral megtörténik, ahány-szor csak meghívják a függvényt.

A lokális változók deklarációja után egy `while` ciklus következik – ez végzi el a tényleges iterációt. A ciklus magját képező utasítás végrehajtása mindenkorral ismétlődik, amíg a `guess2` és az `x` közti különbség `epsilon` alá nem csökken. A

```
guess * guess - x
```

kifejezés eredménye átadódik az `absoluteValue` függvénynek. A különbség abszolút értékét összehasonlítjuk `epsilon`-nal. Ha a különbség nagyobb vagy egyenlő, mint `epsilon`, akkor még nem értük el a kellő pontosságot – ekkor a ciklus továbblén, hogy egy újabb közelítő `guess` értéket számítson ki.

Ha a `guess` értéke kellően közel van a négyzetgyökhöz, akkor a `while` ciklus kilép. Ekkor a `guess` változó értéke visszakerül a függvény hívójához. A `main` függvényben ez a visszatérési érték a `printf` második paramétere szerepel, így a megfelelő üzenettel együtt megjelenik a képernyőn.

Bizonyára feltűnt az olvasónak, hogy mind a `squareRoot`, mind az `absoluteValue` függvény formális paramétere `x`. A C fordítót azonban ez nem zavarja – a két értéket külön kezeli.

Minden függvénynek saját „különbejáratú” formális paraméterlistája van.

Az `absoluteValue` függvényben használt `x` formális paraméter teljesen független a `squareRoot` függvényben használt `x` formális paramétertől.

Ugyanez a helyzet a lokális változókkal. Nyugodtan lehet ugyanazokat a lokális változóneveket használni több függvényen belül. A C fordítóprogram nem veszi ezt rossz néven, mert fel van készítve arra, hogy a lokális változók csak abban a függvényben elérhetőek, amelyikben deklárljuk őket. Szakkifejezéssel elve azt mondhatjuk, hogy a lokális változók hatóköre csak arra a függvényre terjed ki, amelyben a változót létrehoztuk. (A 11. fejezetben („Mutatók”) szó lesz arról, hogy a C nyelv lehetőségeit ad arra, hogy közvetett módon el tudjuk érni egy függvény lokális változóját a függvényen kívülről is.)

Az előzőek alapján már bizonyára érthető, hogy amikor a `guess2 - x` értékét átadjuk az `absoluteValue` függvénynek (ahol ez az érték az `x` formális paraméterbe kerül), akkor ennek az értékadásnak semmi hatása *nincs* a `squareRoot` függvényen belüli `x` értékre.

A visszatérési érték és a paraméter típusának deklarálása

Korábban említettük, hogy a C fordítóprogram alapértelmezésként azt feltételezi, hogy minden függvény visszatérési értéke `int` típusú. Pontosabban fogalmazva: a függvényhíváskor a fordító feltételezi, hogy a visszatérési érték `int` típusú, hacsak nem áll fenn a következő két eset valamelyike:

1. A függvény definíciója már a függvényhívás előtt, a program egy korábbi pontján megtörtént.
2. A függvény visszatérési értékének típusát a programozó még azelőtt *deklarálta*, mielőtt a függvényhívás bekövetkezett volna.

A 8.8 Listában az `absoluteValue` függvény definíciója már azelőtt megtörténik, mielőtt a `squareRoot` függvényben meghívásra kerülne. A fordító ekkor már tisztában van azzal, hogy az `absoluteValue` függvény lebegőpontos értéket ad vissza. Ha ellenben az `absoluteValue` függvény csak a `squareRoot` függvény *után* lenne definiálva, akkor azt feltételezné a fordítóprogram a `squareRoot`-beli `absoluteValue` meghívásakor, hogy ez utóbbi visszatérési értéke egész típusú. A legtöbb C fordító észleli ezt a helyzetet, és ad egy figyelmeztető üzenetet.

Arra is van lehetőség, hogy az `absoluteValue` függvényt a `squareRoot` függvény után definiáljuk (vagy akár más állományban – lásd a 15. fejezetet), de ebben az esetben *deklarálni* kell az `absoluteValue` függvény által visszaadott érték típusát, még mielőtt a függvény meghívása sorra kerülne. Maga a deklaráció történhet a `squareRoot` függvény belséjében, vagy rajta kívül is. Ez utóbbi esetben célszerű a függvény-deklarációkat a program elején megfogalmazni.

Nem pusztán a visszatérési érték típusának előrejelzése miatt kell esetenként függvények deklarációjával foglalkozni, hanem a várt paraméterek száma és típusa is kell a fordítóprogramnak ahhoz, hogy helyes kódot állítson elő.

Tegyük fel, hogy az `absoluteValue` függvényt szeretnénk előre deklarálni. A függvény egyetlen lebegőpontos értéket kér, és ugyancsak lebegőpontos értéket ad vissza. Az alábbi deklaráció használható:

```
float absoluteValue (float);
```

Elég tehát (a név helyett) csak a változó típusát megadni a zárójelben. Ha jobban tetszik a látvány, írható egy „álnév” is a típusnév után:

```
float absoluteValue (float x);
```

Ennek a változónévnek nem kell megegyeznie azzal, ami a függvény formális paraméterlistáján is szerepel – a fordítóprogram úgyis eltekint tőle.

Egy bolondbiztos megoldást jelent az, ha a deklarációt úgy állítjuk elő szövegszerkesztőnkben, hogy a függvénydefiníció első sorát a deklaráció helyére másoljuk. Ne felejtsünk el pontosvesszőt tenni az utasítás után.

Ha a függvény nem vár paramétert, használhatjuk a zárójelbe tett `void` (*türes*) kulcsszót. Ha a függvénynek nincs visszatérési értéke, ezt is deklarálhatjuk. Így elejét vehetjük mindenemű kísérletnek, amely megpróbálna mégis visszatérési értéket kérni a függvénytől:

```
void calculateTriangularNumber (int n);
```

Ha a függvénynek változó számú paramétert szánunk (mint ahogy az a `printf` és `scanf` esetében is történik), arról tudatni kell a fordítóprogramot. Az

```
int printf (char *format, ...);
```

deklaráció közli a fordítóprogrammal, hogy a `printf` egy karakterlánc *mutatót* vár első paraméterként (a mutatókról majd később), amit több más paraméter követhet. A „több más paraméter” jelzésére való a három pont. A `printf` és a `scanf` a `stdio.h` speciális állományban kerül dekláralásra. Emiatt szoktuk elhelyezni programjaink elejére a következőt:

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

Enélkül a fordítóprogram azt feltételezné, hogy a `printf` és a `scanf` rögzített számú paramétert vár, és ennek hatására rossz kódot hozna létre.

A függvények meghívásakor a fordítóprogram automatikusan átalakítja paramétereiket a megfelelő típusúra, de csak akkor, ha a függvény definíciója (vagy a függvénynek és paraméterei típusának a deklarációja) a függvény meghívása előtt áll.

Álljon itt néhány tanács a függvények használatát illetően:

1. Ne feledjük, hogy a fordítóprogram alapértelmezésként azt feltételezi, hogy a függvények visszatérési értéke `int`.
2. Ha egy egész értéket visszaadó függvényt szeretnénk definiálni, akkor definiáljuk ennek megfelelően.
3. Ha olyan függvényt definiálunk, amelynek nincs visszatérési értéke, akkor definiáljuk `void`-ként.
4. A fordítóprogram csak akkor alakítja át paramétereinket olyanná, amilyeneket a függvény vár, ha előzőleg már definiáltuk vagy deklártuk a függvényt.

5. A biztonság kedvéért deklarálunk a program elején minden függvényt, még akkor is, ha meghívásuk előtt már definiálásra kerülnek. (Előfordulhat, hogy később úgy döntünk, hogy máshová tesszük át a függvényünket az állományon belül, vagy akár egy másik állományba.)

Függvények paramétereinek ellenőrzése

A negatív számok négyzetgyöke kivezet a valós számok birodalmába, és átvezet a komplex számok világába. Mi történik tehát akkor, ha negatív számot adunk meg a `squareRoot` függvény paramétereként? Ilyen esetben a Newton-Raphson iteráció nem konvergens, azaz a ciklus lépéseiben adódó részeredmények (*tippek*) nem közelítenek a szám négyzetgyökéhez. Így a hibahatárt vizsgáló feltétel soha nem teljesül: végtelen ciklusba keveredik a vezérlés. A program végrehajtását csak valamilyen megszakító parancsal vagy speciális billentyűkombinációval (*Ctrl+C*) tudjuk megállítani.

Nyilvánvaló a feladat: módosítanunk kell a programot, hogy fel legyen készülve erre az esetre is. Ennek terhét rálőcsölhetjük a hívó eljárásra is, és mondhatjuk azt, hogy legyen szíves tartózkodni a negatív számokból történő gyökvonástól. Ez első látásra elfogadható álláspontnak tűnik, azonban van néhány hátrólütője. Nem nehéz elképzelni, hogy valamelyik gyökvonás előtt egyszerűen elfelejtjük ellenőrizni a paramétert. Ilyenkor előfordulhat, hogy mégis negatív számot adunk át a `squareRoot` függvénynek. Így végtelen ciklusba sodorjuk a programot, amit csak a fent vázolt módon lehet megállítani.

Sokkal bőlcsebb dolog magára a gyökvonó függvényre bízni az ellenőrzés feladatát. Ily módon a függvény „védettséget” élvez, bárki hívja is meg, bármilyen programból. Érdekes tehát az `x` formális paramétert a függvényen belül megvizsgálni, és negatív `x` esetén hibajelzéssel nyugtázni az ellehetetlenült végrehajtást. A függvény ezután rögtön visszaadhatja a vezérlést az őt meghívó eljárásnak anélkül, hogy feleslegesen küzdene a négyzetgyök megkeresésével. Ilyen helyzetben érdemes visszajelzni hívó eljárásnak, hogy nem a szokványos műveletvégzés történt: ezt egy speciális visszatérési értékkel tehetjük meg.¹

A következőkben a fentiek szerint módosított `squareRoot` függvényt láthatjuk. Ebben már megvizsgáljuk a négyzetgyökvonás paraméterét. Érdemes megfigyelni az `absoluteValue` függvényprotótípust, melyet a korábban vázoltak szerint deklarálunk.

```
/* Egy szám négyzetgyökét kiszámító program.
Negatív paraméter esetén hibaüzenet jelenik meg.
Ilyenkor -1.0 lesz a visszatérési érték. */
float squareRoot (float x)
{
```

¹ A szabványos C programkönyvtár négyzetgyökfüggvénye, az `sqrt` értelmezési tartomány hibát („domain error”-t) ad negatív paraméter esetén. A konkrét visszatérési érték implementációfüggő, azaz a különféle fordítóprogramok másként járhatnak el ilyen esetben. Ha kiíratjuk a visszatérési értéket, többnyire a „*nan*”, azaz „not a number” (nem szám) karakterláncot látjuk.

```
const float epsilon = .00001;
float guess = 1.0;
float absoluteValue (float x);

if ( x < 0 )
{
    printf ("Negatív szám a négyzetgyök alatt.\n");
    return -1.0;
}

while ( absoluteValue (guess * guess - x) >= epsilon )
    guess = ( x / guess + guess ) / 2.0;

return guess;
}
```

Negatív paraméter esetén megjelenik a megfelelő figyelmeztető üzenet, és -1 visszatérési értékkel azonnal visszakerül a vezérlés a hívó eljárásba. Ha az érték nem negatív, akkor a korábbiak szerint lefut a függvény, kiszámítva az x négyzetgyökének nagy pontosságú közelítését.

Ahogy a módosított `squareRoot` függvényben láthatjuk (és ahogy ez az előző fejezet utolsó példájából is kiderült), egynél több `return` utasítás is megadható egy függvényen belül. Bárhol is ütközik bele a végrehajtás egy `return` utasításba, azonnal visszakerül a vezérlés a hívó eljárásba. A függvény `return` utáni esetleges utasításait figyelmen kívül hagyja a program. Ezt a lehetőséget olyan függvények esetén is fel lehet használni, melyeknek nincs visszatérési értéke. Ahogy korábban már említettük, a `return` utasítás önmagában is használható; ilyenkor nincs megadva visszatérési érték:

```
return;
```

Ha azonban a függvény úgy van definiálva, hogy visszatérési értéket várunk tőle, akkor ez a megoldás nem használható a függvényből való kilépésre.

Programfejlesztés felülről lefelé építkezve

Az a lehetőség, hogy függvények meghívjanak függvényeket, melyek újabb függvényeket hívnak meg stb., megalapozza a strukturált programozást. A 8.8 Lista main eljárása többször is meghívja a `squareRoot` függvényt. A négyzetgyök kiszámítására vonatkozó összes részlet a `squareRoot` függvényben található, és nem a `main`-ben. Így a program váza (az a „jéghegy csúcsa”) már azelőtt megírható, mielőtt létezne a `squareRoot` függvény, feltevé, hogy már adott a paraméterek és a visszatérési érték típusa.

Amikor aztán sor kerül a `squareRoot` függvény kódjának megírására, ugyanez a „fentről lefelé” történő programozási stratégia használható. Az abszolút értéket kiszámító függvényt már azelőtt meghívhatjuk a négyzetgyökvonásnál, mielőtt annak részletei már készülnek. Elég annyit tudni, hogy azt a függvényt is meg lehet írni.

Ez az elv, ami leegyszerűsíti a programok megírását, egyszerűbbé teszi azok olvasását is. Ha valaki ránéz a 8.8 Listára, egyből látja, hogy a program három szám négyzetgyöökét számítja ki. Ehhez nem kell a négyzetgyökvonás részleteit megvizsgálnia. Ha kíváncsi a részletekre, akkor természetesen megnézhető a `squareRoot` függvény kódja is. Ugyanez vonatkozik a `squareRoot`-on belüli abszolút érték függvényre. Ennek működését nem kell ismerni ahhoz, hogy érthető legyen a négyzetgyök vonásának algoritmusa. Azonban ha valaki kíváncsi arra is, akkor megtalálja a keresett információt az `absoluteValue` függvény definíciójában.

Függvények és tömbök

Függvénynek paramétereként átadható egy tömb valamely eleme, vagy akár egy teljes tömb is. A 7. fejezetben a `printf` függvénnyel kiírtunk egy-egy tömbelemet – ugyanígy más esetekben is átadható egy tömb valamely eleme. Egyszerűen meg kell nevezni a tömb adott sorszámu elemét a függvény paramétereként. Például az `averages[i]` négyzetgyökét úgy tudjuk hozzárendelni a `sq_root_result` változóhoz, hogy kiadjuk a következő utasítást:

```
sq_root_result = squareRoot (averages[i]);
```

A `squareRoot` függvényen belül semmit sem kell tenni ahhoz, hogy egy tömb valamely elemét el tudja fogadni paraméterként. A függvény meghívásakor az átadott tömbelem paraméter értéke (más változókhöz hasonlóan) átmásolódik a formális paraméterbe, és minden meggy tovább a szokásos módon.

Egy egész tömb átadása már más súlycsoportba tartozó feladat. Ilyen esetben a függvény meghívásakor csak a tömb nevét adhatjuk meg paraméterként, `index(ek)` nélkül. Tegyük fel, hogy a `gradeScores` tömb száz elemet tartalmaz. A

```
minimum(gradeScores)
```

utasítás a teljes (száz elemet tartalmazó) `gradeScores` tömböt átadja a `minimum` függvénynek. Természetesen az éremnek két oldala van: a `minimum` függvényt úgy kell megadni, hogy fogadni tudja ezt a száz elemet tartalmazó tömböt. A formális paraméter deklarációjának illeszkednie kell ehhez.

A `minimum` függvény tehát valahogy így nézhet ki:

```
int minimum (int values[100])
{
    ...
    return minValue;
}
```

A deklaráció úgy határozza meg a `minimum` függvényt, hogy az egy egész értéket adjon vissza, és egy olyan tömböt fogadjon el paraméterként, mely 100 egészet tartalmaz. A formális paraméterben megadott `values` tömb adott elemére történő hivatkozás igazából a paraméterként megadott tömb adott elemére mutat. A fent látható függvényhívás és a megfelelő függvénydeklaráció alapján a `values[4]`-re történő hivatkozással igazából a `gradeScores[4]`-re hivatkozunk.

Írunk egy olyan programot, amely egy tömböt elfogadó függvény segítségével megkeresi tíz szám közül a legkisebbet. Ez a `minimum` függvény látható a 8.9 Listában a megfelelő `main` eljárással együtt, amely kezdőértékeket ad a tömb elemeinek, majd meghívja a `minimumszámító` függvényt.

8.9 Lista • Egy tömb elemeinek minimumát megkereső program

```
//Egy tömb elemei közül keressük meg a legkisebbet
```

```
#include <stdio.h>

int minimum (int values[10])
{
    int minValue, i;

    minValue = values[0];

    for ( i = 1; i < 10; ++i )
        if ( values[i] < minValue )
            minValue = values[i];

    return minValue;
}

int main (void)
{
    int scores[10], i, minScore;
    int minimum (int values[10]);

    printf ("Adj meg 10 pontszámot!\n");
}
```

```

for ( i = 0; i < 10; ++i )
    scanf ("%i", &scores[i]);

minScore = minimum (scores);
printf ("\nA legkisebb pontszám: %i\n", minScore);

return 0;
}

```

8.9 Lista • Kimenet

Adjon meg 10 pontszámot!

69
97
65
87
69
86
78
67
92
90

A legkisebb pontszám: 65

Szembeötlő a main elején a `minimum` függvényprototípus deklarációja. Ezzel tudatjuk a fordítóprogrammal, hogy a `minimum` függvény egy egész számot ad vissza, és tíz egészből álló tömböt vár paraméterként. Nem lett volna kötelező megadni itt ezt a deklarációt, mivel a `minimum` függvényt már a meghívása előtt definiáltuk. Azonban jobb a biztonság – deklaráljunk előre minden függvényt, ami szerepet játszik a programban.

A `scores` tömb deklarációja után a program bekéri a tömb elemeinek az értékét. A `scanf` segítségével betöljtük a beírt értékeket a `scores[i]`-be, ahol az `i` 0-tól 9-ig lépdel. Az értékek beírása után meghívjuk a `minimum` függvényt a `scores` tömbbel, mint paraméterrel.

A függvényen belül a `values` formális paraméternévvel hivatkozhatunk az átadott tömb elemeire. Ez egy tízelemű tömbnek van dekláralva. A lokális `minValue` változó tárolja a tömb (pillanatnyilag ismert) legkisebb értékét. Kezdetben ezt egyszerűen `values[0]`-ra állítjuk, a tömb kezdőelemére. Egy `for` ciklus végiglépdel a tömb többi elemén. A tömb elemeit sorban összehasonlítjuk a `minValue` -val, és ha a tömbelem a kisebb, akkor (mivel jobb minimumot találtunk) felülírjuk a `minValue` értékét a tömbelem értékével. Ugyanígy megvizsgáljuk a tömb összes elemét is, így a ciklus végén valóban a legkisebb elemet fogja tartalmazni a visszatérési értékül adandó `minValue`. A `main` eljárásban a `minimum` függvény visszatérési értékét betöljtük a `minScore` változóba, majd egy alkalmas üzenettel körítve kiírjuk a képernyőre.

Kezünkben van tehát egy általánosan használható függvény, melynek segítségével bármilyen tíz elemű tömb minimumát meghatározhatjuk. Ha lenne öt különböző tízelemes tömbünk, könnyen írhatnánk egy programot, amely ötször egymás után meghívja a minimumszámító függvényt. Ehhez hasonlóan lehetne persze maximumot, mediánt, számtani közepet stb. is számítani a tömb elemeiből.

Az efféle apró, de jól definiált feladatot ellátó függvényekkel összetett alkalmazásokat is fel lehet építeni. Definiálhatunk például egy *statisztika* nevű függvényt, amelynek átadhatunk egy tömböt. Függvényünk meghívhat más függvényeket, amelyek kiszámítják például az átlagot, a szórást stb., így különféle statisztikai adatokat gyűjthetünk az adatsorról. Az ilyen típusú programok kiváló iskolapéldái annak a programozói stílusnak, amelyet érdemes megtanulni. Az így fejlesztett programokat könnyű megírni, megérteni, módosítani és karbantartani.

Mindannyian érezzük, hogy ez a program még nem az igazi, hiszen ritkán van szükség éppen tízelemű tömb feldolgozására. Ezen a problémán azonban nem nehéz felülemelkedni. Könnyen kiterjeszthetjük függvényünket oly módon, hogy a tömb elemszámát is paraméterként lehessen megadni. A függvény deklarációjakor a formális paraméternévből el kell hagynunk a tömb elemszámát. A C fordítóprogram egyébként sem szentel különösebb figyelmet ezen a helyen a tömb elemszámának – csak azt szűri le a deklarációból, hogy egy tömböt fog majd paraméterként kapni a függvény; ennek elemszáma érdektelen.

A 8.10 Lista a 8.9 Lista újraírt változata, melyben a minimum függvény egy tetszőleges méretű (egészkeből álló) tömbben keresi meg a legkisebb értéket.

8.10 Lista • Egy tömb elemeinek minimumát megkereső program újraírt változata

```
//Egy tömb elemei közül megkeressük a legkisebb elemet - 2. változat

#include <stdio.h>

int minimum (int values[], int numberOfElements)
{
    int minValue, i;

    minValue = values[0];

    for ( i = 1; i < numberOfElements; ++i )
        if ( values[i] < minValue )
            minValue = values[i];

    return minValue;
}

int main (void)
{
```

```

int array1[5] = { 157, -28, -37, 26, 10 };
int array2[7] = { 12, 45, 1, 10, 5, 3, 22 };
int minimum (int values[], int numberOfElements);

printf ("array1 minimum: %i\n", minimum (array1, 5));
printf ("array2 minimum: %i\n", minimum (array2, 7));

return 0;
}

```

8.10 Lista • Kimenet

```

array1 minimum: -37
array2 minimum: 1

```

Itt a minimumszámító függvénynek már két paramétere van: az első az átadott tömb neve (melyben a legkisebb elemet keressük), a második ennek elemszáma. A tömb megadásakor a nyitó és záró jelek közvetlenül egymás után állnak a `values` név után, így: `[]`. Ebből tudja a fordítóprogram, hogy egészkeből álló *tömbről* van itt szó. Ahogy az előbb is mondtuk: az most itt senkit nem érdekel, hogy ez a tömb mekkora.

A `for` ciklusban álló `numberOfElements` helyettesíti az előző programban használt 10-es konstanst – ez jelzi a ciklusváltozó fölső határát a ciklusfeltételben. Így tehát a `for` ciklus végigvándorol a `values[1]`-től kezdődően egészen a tömb utolsó értékéig, egészen a `values[numberOfElements - 1]`-ig.

A `main` eljárásvan két tömböt használunk, `array1` és `array2` néven szerepel az öt- és a hételemű tömb.

Az első `printf`-en belül meghívjuk a `minimum` függvényt, mégpedig az `array1` és az 5 paraméterrel. Ez utóbbi paraméter az `array1` elemszámára utal. A `minimum` függvény megkeresi és visszaadja a tömb legkisebb elemét (-37-et), ami meg is jelenik a képernyőn. Második alkalommal is meghívjuk a `minimumszámító` függvényt az `array2` és a 7 paraméterrel, és a visszaadott 1 értéket átadjuk a `printf`-nek, hogy írja ki a képernyőre.

Hozzárendelő operátorok

Figyeljük meg a 8.11 Listát és próbáljuk meg kitalálni a program kimenetét, mielőtt még megnéznénk a tényleges eredményt.

8.11 Lista • Egy tömb elemeinek megváltoztatása függvény segítségével

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

```

void multiplyBy2 (float array[], int n)
{
    int i;

```

```

    for ( i = 0; i < n; ++i )
        array[i] *= 2;
}

int main (void)
{
    float  floatVals[4] = { 1.2f, -3.7f, 6.2f, 8.55f };
    int     i;
    void    multiplyBy2 (float  array[], int   n);

    multiplyBy2 (floatVals, 4);

    for ( i = 0; i < 4; ++i )
        printf ("% .2f    ", floatVals[i]);

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

```

8.11 Lista • Kimenet

2.40 -7.40 12.40 17.10

Ha végignézte az olvasó a 8.11 Listát, bizonyára megakadt a tekintete a következő soron:

```
array[i] *= 2;
```

A „-szor egyenlő” ($\ast=$) operátor összeszorozza a tőle balra álló értéket a tőle jobbra állóval, és az eredményt eltárolja a *tőle balra álló* változóban. Mintha csak ezt látnánk a programban:

```
array[i] = array[i] * 2;
```

Visszatérve programunk lényegi mondanivalójához: meglepő módon a `multiplyBy2` függvény *ténylegesen* megváltoztatja a `floatVals` tömböt. Nincs ez ellentmondásban a korábban tanultakkal, miszerint a függvény a paraméter eredeti értékéhez nem tud hozzáférni? Nincs.

Ez a kód részlet rámutat arra a kivételezett helyzetre, amit a paraméter-tömbök nél már említettünk: ha a függvény megváltoztatja a paraméterként kapott tömb valamelyik elemének értékét, akkor valójában az eredeti tömb adott értékét változtatja meg. A változtatás akkor is érvényben marad, amikor a függvény kilép, és a vezérlés visszatér a hívó eljárás hozzá.

Az a tény, hogy a paraméter-változók (így a paraméterként átadott tömb-elemek is) más-ként viselkednek, mint a paraméter-tömbök, kíván némi magyarázatot. Az előbbieket nem lehet függvényből megváltoztatni. Említettük, hogy az argumentumként átadott változók értéke átmásolódik a megadott formális paraméterekekbe. Ez a megfogalmazás teljesen igaz. Amikor azonban tömböt adunk át paraméterként, a tömb nem másolódik át seholról. Ehez azt a *memóriacímet* kapja meg a függvény, ahol az adott tömb elhelyezkedik. Így bár miféle változás, amit a függvény a paraméterként kapott tömbön elkövet, az eredeti tömb elemein hajtódklik végre – ennek hatása pedig a függvény kilépése után is megmarad.

Ne feledjük, hogy a fenti gondolatmenet csak akkor van érvényben, amikor egy teljes tömböt adunk át paraméterként, nem pedig annak egyes elemeit. A tömbelemek paraméterként való átadásakor másolás zajlik, így a formális paramétereken végbevitt változtatások nem érintik az eredeti tömbelemeket. A 11. fejezetben a mutatók kapcsán részletesen visszatérünk erre a kérdésre.

Tömbök rendezése

A paraméterként megadott tömbök függvénnel való módosításának illusztrálására álljon itt még egy példa: hozzunk létre egy olyan függvényt, amely egy egészekből álló tömb elemeit sorba rendezzi. A rendezési algoritmusok mindenkor is a figyelem középpontjában állnak a programozók világában – talán amiatt, mert az adatok rendezése meglehetősen gyakori igény. Sok furfangos adatrendező algoritmus napvilágot látott már annak elérésére, hogy a lehető legrövidebb idő alatt, a lehető legkevesebb memória használatával sikerüljön sorba rendezni egy adathalmazt. Könyünknek nem célja az összes ilyen algoritmus számba vétele, így most csak egy kellemesen áttekinthető tömbrendezést fogunk megvalósítani a sort függvénnnyel. Célunk a növekvő sorrendbe rendezés, azaz az elemek sorrendjének olyan kialakítása, hogy az egymás utáni elemek értékei monoton növekvő sorozatot képezzenek a legkisebbtől a legnagyobbig. A rendezés végeztével a legkisebb elem lesz a tömb kezdőeleme, és a legnagyobb elem lesz az utolsó.

Ha egy n elemből álló tömböt szeretnénk növekvő sorrendbe rendezni, akkor ezt megtehetjük úgy is, hogy sorban összehasonlítsuk az elemeket egymással. Kezdhetjük azzal, hogy az első elemet összehasonlítsuk a másodikkal. Ha a második elem a nagyobb, akkor felcseréljük a két elemet, (azaz a két helyen álló elemek értékeit).

Ezután az első elemet (amelyről már tudjuk, hogy kisebb a másodiknál) a harmadik tömb-elemmel vetjük össze. Ha az első elem nagyobb, akkor ismét csere következik – egyébként minden marad változatlanul. Ekkor biztosak lehetünk abban, hogy az első három elem legkisebbike az első helyen áll.

Ha ugyanezt a műveletsort (összehasonlítás, majd csere, ha az első a nagyobb) végigvisszük a tömb összes elemére vonatkozóan, akkor a tömb legkisebb eleme lesz az első helyen.

Ugyanezt az eljárást hajtsuk végre a második elemmel kezdődően, a tömb harmadik, negyedik stb. elemére vonatkozóan is. Ennek hatására a második helyen a tömb második legkisebb eleme foglal majd helyet.

Gondolom, ezek után világos, hogy hogyan zajlik a rendezés további része. Végig lépünk a harmadik elemtől kezdve, a negyediktől kezdve stb. egészen az utolsó előtti elemig, és (alkalmas cserék végre hajtásával) a sorrendben következő nagyságú elemet tesszük az adott helyre. Végül a tömb összes eleme nagyság szerinti sorrendben áll rendelkezésünkre.

Az alábbi algoritmus még tömörebben megfogalmazza a fent vázolt módszert. Legyen a rendezendő n elemű tömbünk neve a .

Az egyszerű cserélő rendezés algoritmusa

1. lépés: Állítsuk i értékét 0-ra.
2. lépés: Állítsuk j értékét $i + 1$ -re.
3. lépés: Ha $a[i] > a[j]$, akkor cseréljük fel az értékeket.
4. lépés: Növeljük eggyel j értékét. Ha $j < n$, lépjünk vissza a 3. lépéshoz.
5. lépés: Növeljük eggyel i értékét. Ha $i < n - 1$, lépjünk vissza a 2. lépéshoz.
6. lépés: Az a tömb elemei növekvő sorrendben követik egymást.

A 8.12 Lista a fenti algoritmust valósítja meg a sort függvénnnyel. Ez két paramétert vár: a rendezendő tömb nevét és elemszámát.

8.12 Lista • Egészekből álló tömb növekvő sorrendbe történő rendezése

// Egy egészekből álló tömb nagyság szerinti rendezése

```
#include <stdio.h>

void sort (int a[], int n)
{
    int i, j, temp;

    for ( i = 0; i < n - 1; ++i )
        for ( j = i + 1; j < n; ++j )
            if ( a[i] > a[j] ) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
}

int main (void)
{
```

```

int i;
int array[16] = { 34, -5, 6, 0, 12, 100, 56, 22,
                  44, -3, -9, 12, 17, 22, 6, 11 };
void sort (int a[], int n);

printf ("A tömb elemei a rendezés előtt:\n");

for ( i = 0; i < 16; ++i )
    printf ("%i ", array[i]);

sort (array, 16);

printf ("\n\nA tömb elemei a rendezés után:\n");

for ( i = 0; i < 16; ++i )
    printf ("%i ", array[i]);

printf ("\n");

return 0;
}

```

8.12 Lista • Kimenet

A tömb elemei a rendezés előtt:

34 -5 6 0 12 100 56 22 44 -3 -9 12 17 22 6 11

A tömb elemei a rendezés után:

-9 -5 -3 0 6 6 11 12 12 17 22 22 34 44 56 100

A sort függvény egymásba ágyazott `for` ciklusokkal valósítja meg az algoritmust. A különböző ciklus végiglépdel a tömbön az elsőtől az utolsó előtti elemig, `a[n-2]`-ig. Ezekhez az elemekhez kapcsolódóan lefut egy belső ciklus, mely az elemet követő elemtől az utolsóig pásztázza végig a tömb elemeit.

Ha a szemügyre vett két elem nincs jó sorrendben (azaz `a[i]` nagyobb `a[j]`-nél), akkor felcseréljük őket. A `temp` változó ideiglenes adattárolásra szolgál a csere idejére.

Mire minden két `for` ciklus véget ér, az elemek már nagyság szerinti sorrendben állnak a tömbben. A függvény elvégezte feladatát.

A main eljárásban definiáljuk az `array` tömböt, és a 16 egész kezdőértékkkel inicializáljuk. A program kiírja az `array` tömb eredeti értékeit, majd meghívja a `sort` függvényt. Paraméterei: `array` és 16, azaz a rendezendő tömb neve és elemszáma. A függvényből való visszatérés után a program ismét kiírja az `array` tömb tartalmát. Ahogy a kimenetből is látszik, a függvény sikeresen sorba rendezte a tömböt.

A 8.2 Lista sort függvénye meglehetősen egyszerű, ám ennek ára van: a futási idő. Ha igazán nagy (például több ezer elemből álló) tömböt rendezünk, akkor a program tekintélyes ideig fut. Ilyen esetben érdemes más rendezési módszert választani. Segítségünkre lehet ebben Donald E. Knuth és Addison-Wesley „A számítógép-programozás művészete” (*The Art of Computer Programming*) c. könyvének 3. kötete: „Keresés és rendezés”.²

Többdimenziós tömbök

Egy többdimenziós tömb valamely eleme ugyanúgy átadható paraméterként, mint bármi-lyen más változó (vagy például egydimenziós tömb egy eleme). Így a

```
squareRoot (matrix[i][j]);
```

utasítás a `matrix[i][j]` értékét adja át paraméterként a meghívott `squareRoot` függ-vénynek.

Átadhatunk paraméterként egy teljes többdimenziós tömböt is, az egydimenziós tömbnek megfelelő módon. Egyszerűen meg kell adni a tömb nevét. Ha például a `measured_values` egy egészekből álló kétdimenziós tömb, a következő utasítás:

```
scalarMultiply (measured_values, constant);
```

jelenthet egy olyan szorzó függvényt, melynek során a tömb valamennyi elemét megszo-rozzuk egy adott (`constant`) számmal. Ez azt is feltételezi, hogy a `scalarMultiply` függvény meg tudja változtatni a kétdimenziós tömb (`measured_values`) elemeinek az értékét. Az egydimenziós tömbökönél tárgyalta itt is igazak: minden változtatás, melyet a függvény a formális paraméter-tömbön eszközöl, az eredeti tömbön történik meg.

Tanultuk, hogy a függvényen belüli egydimenziós tömb-deklarációt nem kell dimenziószámo(ka)t megadni. Elég csak szögletes zárójellel jelezni a fordítóprogramnak, hogy itt egy tömböt várunk paraméterként, és kész. A többdimenziós tömbök esetében nem teljesen ez a helyzet. Kétdimenziós tömbök esetében a tömb sorainak számát elhagy-hatjuk, de az oszlopok számát nem. Így tehát az

```
int array_values[100][50]
```

és az

```
int array_values[][][50]
```

² Van egy szabványos C programkönyvtári függvény, a `qsort`, amely tetszőleges típusú tömböt rendezni tud. Mielőtt azonban használni kezdenénk, érdemes megismерkedni a 11. fejezetben a függvénymutatókkal.

deklarációk helyesek, ám az

```
int array_values[100][]
```

és az

```
int array_values[][]
```

deklarációk nem. Az első két eset jól használható például egy olyan tömb esetében, melynek 100 sora és 50 oszlopa van. Az utóbbi két eset azonban nem alkalmazható, mert az oszlopok számát mindenkor meg kell adni.

A 8.13 Listában megadunk egy scalarMultiply függvényt, amely egy tömb elemeit összeszorozza egy adott számértékkal. A példában használunk egy 3×5 -ös mátrixot. A main eljárás kétszer hívja meg a scalarMultiply függvényt; mindenkor után meghívódik a displayMatrix függvény is, amely megjeleníti a mátrix tartalmát. Figyeljük meg az egymásba ágyazott for ciklusokat, melyek mindenkor a scalarMultiply, mindenkor a displayMatrix függvényben végigjárják a kétdimenziós tömb elemeit.

8.13 Lista • Többdimenziós tömbök használata paraméterként

```
#include <stdio.h>

int main (void)
{
    void scalarMultiply (int matrix[3][5], int scalar);
    void displayMatrix (int matrix[3][5]);
    int sampleMatrix[3][5] =
    {
        { 7, 16, 55, 13, 12 },
        { 12, 10, 52, 0, 7 },
        { -2, 1, 2, 4, 9 }
    };

    printf ("Az eredeti mátrix:\n");
    displayMatrix (sampleMatrix);

    scalarMultiply (sampleMatrix, 2);

    printf ("\nSzorozva kettővel:\n");
    displayMatrix (sampleMatrix);

    scalarMultiply (sampleMatrix, -1);

    printf ("\nVégül -1-el szorozva:\n");
```

```

    displayMatrix (sampleMatrix);

    return 0;
}

// Egy 3 x 5-ös tömb szorzása egy számmal

void scalarMultiply (int matrix[3][5], int scalar)
{
    int row, column;

    for (row = 0; row < 3; ++row)
        for (column = 0; column < 5; ++column)
            matrix[row][column] *= scalar;
}

void displayMatrix (int matrix[3][5])
{
    int row, column;

    for (row = 0; row < 3; ++row) {
        for (column = 0; column < 5; ++column)
            printf ("%5i", matrix[row][column]);

        printf ("\n");
    }
}

```

8.13 Lista Kimenet

Az eredeti mátrix:

7	16	55	13	12
12	10	52	0	7
-2	1	2	4	9

Szorozva kettővel:

14	32	110	26	24
24	20	104	0	14
-4	2	4	8	18

Végül -1-el szorozva:

-14	-32	-110	-26	-24
-24	-20	-104	0	-14
4	-2	-4	-8	-18

A main eljárásban definiáljuk a sampleMatrix tömböt, majd meghívjuk a displayMatrix függvényt, hogy megtekinthessük a tömb kezdőértékeit. A külső for ciklus a mátrix sorain lépdel végig, így a row (sor) változó 0-tól 2-ig növekszik. minden row-hoz elindul egy belső for ciklus, mely (a column ciklusváltozóval 0-tól 4-ig) az adott sor oszlopain halad végig.

A printf utasítás kiírja a mátrix megadott sorához (row) és oszlopához (column) tartozó elemét, mégpedig a %5i formátumjelző karakterláncjal. Így az értékek egymás alatt jelennek meg, hiszen mindegyikhez legalább 5 karakternyi hely lesz lefoglalva. Amikor a belső ciklus végzett – azaz a sor összes eleme megjelent a képernyőn – akkor egy újsor káraktert is kiíratunk, hogy a mátrix sorainak megfelelően történjen a számok kijelzése (soronként új sorban).

A scalarMultiply első meghívásakor a sampleMatrix tömb elemeit 2-vel szorozzuk. A függvényen belül két egyszerű egymásba ágyazott for ciklus gondoskodik arról, hogy minden elem sorra kerüljön. A *= operátornak megfelelően a matrix[row][column] megszorzódik a scalar változóval. Mihelyt a függvény visszatér a főprogramba, meghívódik a displayMatrix függvény, hogy újra láthassuk a sampleMatrix értékeit. A kimenetből látszik, hogy valóban minden elem értéke megszorzódott a megadott számmal, vagyis 2-vel.

Még egyszer meghívjuk a scalarMultiply függvényt, hogy a módosított sampleMatrix mátrix elemeit megszorozzuk -1-gyel. A módosított tömb a displayMatrix függvény hívása révén még egyszer megtekinthető a képernyőn, majd a program kilép.

Többdimenziós, változó méretű tömbök használata függvényekben

A C nyelv lehetőséget kínál arra, hogy változó méretű többdimenziós tömböt hozzunk létre. A példa kedvéért a 8.13 Listát újraírtuk, így a scalarMultiply és a displayMatrix függvények már bármilyen méretű mátrixot fel tudnak dolgozni; tetszőleges mennyiségű sort-osalapot tartalmazó mátrixok paraméterül adhatóak a 8.13A Lista programjának.

8.13A Lista • Többdimenziós, változó méretű tömbök

```
#include <stdio.h>

int main (void)
{
    void scalarMultiply (int nRows, int nCols,
                         int matrix[nRows][nCols], int scalar);
    void displayMatrix (int nRows, int nCols, int
                        ➔ matrix[nRows][nCols]);
    int sampleMatrix[3][5] =
    {
        { 7, 16, 55, 13, 12 },
        { 12, 10, 52, 0, 7 },
        { -2, 1, 2, 4, 9 }
    };

    printf ("Az eredeti mátrix:\n");
    displayMatrix (3, 5, sampleMatrix);
```

```

scalarMultiply (3, 5, sampleMatrix, 2);
printf ("\nSzorozva kettővel:\n");
displayMatrix (3, 5, sampleMatrix);

scalarMultiply (3, 5, sampleMatrix, -1);
printf ("\nVégül -1-el szorozva:\n");
displayMatrix (3, 5, sampleMatrix);

return 0;
}

// Mátrix szorzása számmal

void scalarMultiply (int nRows, int nCols,
                     int matrix[nRows][nCols], int scalar)
{
    int row, column;

    for (row = 0; row < nRows; ++row)
        for (column = 0; column < nCols; ++column)
            matrix[row][column] *= scalar;
}

void displayMatrix (int nRows, int nCols, int matrix[nRows][nCols])
{
    int row, column;

    for (row = 0; row < nRows; ++row) {
        for (column = 0; column < nCols; ++column)
            printf ("%5i", matrix[row][column]);

        printf ("\n");
    }
}
}

```

8.13A Lista • Kimenet

Az eredeti mátrix:

7	16	55	13	12
12	10	52	0	7
-2	1	2	4	9

Szorozva kettővel:

14	32	110	26	24
24	20	104	0	14
-4	2	4	8	18

Végül -1-el szorozva:

-14	-32	-110	-26	-24
-24	-20	-104	0	-14
4	-2	-4	-8	-18

A scalarMultiply függvény deklarációja így fest:

```
void scalarMultiply (int nRows, int nCols, int matrix[nRows][nCols],
    → int scalar)
```

Az nRows és nCols változókat, azaz a mátrix oszlopait és sorait még a matrix megadása előtt fel kell sorolni, hogy a fordítóprogram a paraméter-tömb megadásakor már tudja a pontos dimenzióértékeket. Ha ehelyett így próbáljuk meg:

```
void scalarMultiply (int matrix[nRows][nCols], int nRows, int nCols,
    → int scalar)
```

akkor a fordítóprogram hibaüzenetet ad, mivel a matrix tömb létrehozásakor még nincs tudomása az nRows és az nCols értékéről.

Amint látható, a kimenet megegyezik a 8.13 Lista kimenetével. Mostantól fogva kezünkben van két függvény, a scalarMultiply és a displayMatrix, melyek segítségével bár milyen méretű mátrixot fel tudunk dolgozni. Ez a változó méretű tömbök használatának egyik előnye.³

Globális változók

Ideje összefoglalni, mi minden tanultunk ebben a fejezetben; emellett néhány új fogalmat is megismérünk. Nézzük a 7.7 Listát, amely átalakít egy számot más számrendszerbe. Írjuk újra a programot függvény alapúvá. Ehhez logikai egységekre kell bontanunk a kódot. Ha ránéz az olvasó a programra, látszik, hogy ez a felbontás igen kézenfekvő (már csak a megjegyzések alapján is). Három jól elkülönülő részre osztható a main eljárás. Magától adódik a három megvalósítandó függvény: az átváltandó szám bekérése, maga az átalakítás, végül az eredmény kiírása.

Definiálhatjuk az ezeket megvalósító függvényeket. Az első neve legyen getNumberAndBase. Ez a függvény szólítja fel a felhasználót, hogy írja be az átalakítandó számot és az új számrendszer alapszámát, majd a megadott értékeket egy-egy változóban tárolja el. Ezen a ponton javítsuk meg az eredeti programot: adjunk hibajelzést, ha 2-nél kisebb vagy 16-nál nagyobb alapszámot próbál megadni a felhasználó. Ilyenkor legyen a számrendszer alapszáma 10, azaz úgy érjen véget a program, hogy újra kiírja a megadott számot. (Az is kézenfekvő lenne, hogy ilyenkor új adatot írhasson be a felhasználó, ám ezt a lehetőséget meghagyjuk az egyik gyakorlatra.)

³ Ahogy korábban említettük, nem minden fordítóprogram támogatja a változó méretű tömböket. Így nem garantált, hogy ezt a lehetőséget az olvasó egy adott számítógépen ki tudja használni.

Legyen a második függvény neve `convertNumber`. Ez kapja meg a felhasználó által beírt számértéket, amit átvált a kívánt számrendszerbe. Az új szám számjegyeinek értékét a `convertedNumber` tömbben tároljuk.

A harmadik függvény legyen a `displayConvertedNumber`. A függvény megkapja a `convertedNumber` tömböt, aminek segítségével (helyes sorrendben) meg tudjuk jeleníti az átváltott számot. A megfelelő formátumú kiíráshoz a `baseDigits` tömböt használjuk, melyben minden egyes számértéknek egyetlen karakter felel meg.

A három függvény globális változók révén tud kapcsolatot tartani egymással. Ahogy korábban tanultuk, a lokális változók csak azon a függvényen belül láthatóak, amelyben deklarálták őket. Bizonyára sejt az olvasó, hogy ennek ellenkezője érvényes a globális változókra: ezek értéke *bármely* függvényből elérhető és megváltoztatható.

Úgy nevezhetünk ki globálissá bizonyos változókat, hogy a függvényeken *kívül* deklaráljuk őket. Az jelzi a változók globális voltát, hogy nem tartoznak egyik függvényhez sem. Ettől kezdve bármely függvényben lehet őket használni.

A 8.14 Listában négy globális változót vezetünk be – mindegyiket legalább két függvény használja.

Mivel a `baseDigits` tömb és a `nextDigit` változó csak a `displayConvertedNumber` függvényben kerül elő, őket *nem* definiáljuk globális változóként. Ehelyett ezeket a `displayConvertedNumber` függvényen belüli lokális változóként adjuk meg.

Először tehát a globális változók deklarációja történik meg, melyeket még a függvények előtt megadunk, így nem tartoznak egyik függvényhez sem: bárhonnan elérhetők lesznek.

8.14 Lista • Pozitív egész szám átváltása tízes számrendszerből egy másik számrendszerbe.

```
// Pozitív egész átváltása tízes számrendszerből egy másikba

#include <stdio.h>

int      convertedNumber[64];
long int numberToConvert;
int      base;
int      digit = 0;

void    getNumberAndBase (void)
{
    printf ("Mi az átváltandó szám? ");
    scanf ("%li", &numberToConvert);
```

```

printf ("Milyen számrendszerbe alakítsuk át? ");
scanf ("%i", &base);

if    ( base < 2 || base > 16 ) {
    printf ("Helytelen alapszám - csak 2 és 16 közötti érték
            ➔ adható meg.\n");
    base = 10;
}
}

void convertNumber (void)
{
    do {

        convertedNumber[digit] = numberToConvert % base;
        ++digit;
        numberToConvert /= base;
    }
    while ( numberToConvert != 0 );
}

void displayConvertedNumber (void)
{
    const char baseDigits[16] =
        { '0', '1', '2', '3', '4', '5', '6', '7',
          '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
    int nextDigit;

    printf ("Az átalakított szám: ");

    for (--digit; digit >= 0; --digit) {
        nextDigit = convertedNumber[digit];
        printf ("%c", baseDigits[nextDigit]);
    }

    printf ("\n");
}

int main (void)
{
    void getNumberAndBase (void), convertNumber (void),
        displayConvertedNumber (void);

    getNumberAndBase ();
    convertNumber ();
    displayConvertedNumber ();

    return 0;
}

```

8.14 Lista • Kimenet

Mi az átváltandó szám? **100**

Milyen számrendszerbe alakítsuk át? **8**

Az átalakított szám: **144**

8.14 Lista • Kimenet (újrafuttatás)

Mi az átváltandó szám? **1983**

Milyen számrendszerbe alakítsuk át? **0**

Helytelen alapszám - csak 2 és 16 közötti érték adható meg.

Az átalakított szám: **1983**

Figyeljük meg, milyen sokat jelentenek a jól megválasztott függvénynevek. Teljesen világossá teszi a 8.14 Listát, szükségtelenné téve az eredeti program eligazító megjegyzéseit. A főprogramban szinte elolvasható a program működése (angolul): `getNumberAndBase, convertNumber, displayConvertedNumber`. Azaz: kérd be a számot és az alapszámot, alakítsd át a számot, írd ki az átalakított számot. A 7.7 Listához képest nagyságrendekkel jobb érhetőséget azzal értük el, hogy a részfeladatokat külön függvényekbe csoportosítottuk, ezáltal feleslegessé váltak a magyarázó megjegyzések is. A függvények nevei magukért beszélnek.

A globális változókat olyan programokban használjuk, melyekben bizonyos változókat több függvény is el szeretné érni. Ahelyett, hogy ezek értékét paraméterként adnánk át, sokszor egyszerűbb lehetővé tenni a közvetlen hivatkozást ezekre a változókra. Ennek azonban vannak hátrólai. Mivel a függvény közvetlenül kiutal valamelyen változóra, a program általánossága csorból. Bármikor kerül sor e függvény használatára, meg kell győződni arról, hogy a szükséges globális változók megvannak-e, és megfelelő-e a nevük.

A 8.14 Lista `convertNumber` függvénye például csak akkor működik helyesen, ha létezik egy `numberToConvert` nevű globális változó, mely az átalakítandó számot tárolja, valamint egy `base` nevű másik globális változó, amelyből a számrendszer alapszáma olvasható. Sőt, a `convertedNumber` tömbnek és a `digit` változónak is elérhetőnek kell lennie a számára. Sokkal általánosabban használható függvényt írhatnánk azáltal, ha mindeneket a változókat paraméterként kapná meg a függvény.

A globális változók tehát csökkentik ugyan a használandó paraméterek számát, azonban nemileg aláássák a program általánosságát és esetenként az érhetősséget is. Az olyan programok, amelyben egy-egy függvény fejlécére nézve nem látszanak azonnal a használt változók, nem minden könnyen érhetőek. Hasznos lehet látni a függvények meghívásakor, hogy milyen bemeneti és milyen kimeneti értékekkel dolgoznak.

Jó programozási gyakorlatnak számít egy kis g betű írni a globális változók nevei elő. A 8.14 Lista deklarációi így néznének ki ezzel a közmegegyezéssel:

```
int      gConvertedNumber[64];
long int gNumberToConvert;
int      gBase;
int      gDigit = 0;
```

Érdemes élni ezzel a gyakorlattal, ugyanis így a programkód olvasásakor jól elkülönülnek a globális változók a lokálisaktól. Egy ilyen utasításból például egyből látszik, hogy a nextMove lokális, míg a gCurrentMove globális változó:

```
nextMove = gCurrentMove + 1;
```

A program olvasója számára világos lesz, hogy mekkora a hatóköre az egyes változóknak, és hogy hol érdemes keresnie a deklarációjukat.

Végül meg kell jegyeznünk a globális változókról azt az érdekes tényt, hogy akkor is van kezdőértékük (0), ha nem inicializáljuk őket. Az alábbi globális deklaráció után a gData tömb minden eleme nullázódik:

```
int gData[100];
```

Véssük jól emlékezetünkbe: a globális változók automatikusan nullázódnak a létrejöttük-kor, míg a lokális változókat a programozónak kell inicializálni.

Automatikus és statikus változók

Amikor egy függvényen belül megadunk egy változót (ahogyan például az alábbi squareRoot függvényben az epsilon-t és a guess-t), akkor azok alapértelmezetten automatikus lokális változók lesznek:

```
float squareRoot (float x)
{
    const float epsilon = .00001;
    float guess      = 1.0;
    ...
}
```

Említettük, hogy az ilyen változók elő odaírhatnánk, hogy auto, de mivel ez az alapértelmezett típusmódosító, nem szoktuk kiírni. Az automatikus változók mindenkorral létrejönnek, amikor a függvény meghívódik. A fenti példában az epsilon és a guess minden-

annyiszor létrejön, ahányszor csak a `squareRoot` függvény meghívódik. A függvényből való kilépés után viszont megszűnik létezni minden lokális változó. Ez a folyamat automatikusan megy vége – innen az `auto` megnevezés.

Az automatikus lokális változóknak kezdőértéket lehet adni, ahogy az ebben a példában is látszik az `epsilon` és a `guess` esetében. Bármilyen érvényes C kifejezés kezdőértékül adható egy egyszerű automatikus változónak. Ennek a kifejezésnek az értékét minden alkalommal kiszámítja a program, amikor csak meghívódik az adott függvény.⁴

Mivel az automatikus változók eltűnnek a függvényből való kilépés után, ilyenkor az értékük sem érhető már el. Más szóval, az automatikus változóknak *garantáltan* nem lesz értéke a legközelebbi függvényhíváskor.

Ha a `static` típusmódosítóval élünk, akkor egészen más lesz a helyzet. Ez a szó itt nem valamiféle statikus elektromos töltésre utal, hanem a mozgás, változás hiányára. A statikus változók lényege ebben áll: értékük nem jön-megy annak függvényében, ahogy a függvény meghívódik vagy kilép. Ebből az is következik, hogy az ilyen változó az aktuális értékét a függvény kilépése után is megőrzi mindaddig, amíg legközelebb használatba nem kerül.

A statikus változók az inicializáció tekintetében is sajátosak. A statikus lokális változók a program futása során csak egyszer inicializálódnak, nem pedig unos-untalan, amikor a függvény meghívásra kerül. Az értékül adott kifejezés pedig nem lehet akármilyen, hanem csak konstans. Amennyiben nem adunk kezdőértéket egy statikus változónak, akkor is rendelkezik a 0 kezdőértékkel (így ebben is eltér az automatikus változóktól).

Tekintsük az alábbi `auto_static` függvényt:

```
void auto_static (void)
{
    static int staticVar = 100;
    .
    .
}
```

Itt a `staticVar` csak egyetlen egyszer inicializálódik 100-ra, a program végrehajtásának kezdetén. Ha minden alkalommal külön inicializálni szeretnénk, azt csak kimondott hozzárendeléssel oldhatjuk meg:

```
void auto_static (void)
{
    static int staticVar;
```

⁴ A `const` változók értéke a csak-olvasható memóriában tárolódik, így nem kell őket minden függvényhívás alkalmával újrainicializálni.

```

staticVar = 100;
.
.
}

}

```

A staticVar ily módon történő újrainicializálása természetesen értelmetlenné teszi az előtte álló sor static kulcsszavát.

A 8.15 Lista rávilágít a statikus és automatikus változók közti eltérésekre, és segít megérteni a háttérben álló fogalmakat.

8.15 Lista • Statikus és automatikus változók bemutatása

```

// Statikus és automatikus változók működésének bemutatása

#include <stdio.h>

void auto_static (void)
{
    int         autoVar = 1;
    static int  staticVar = 1;

    printf ("automatic = %i, static = %i\n", autoVar, staticVar);

    ++autoVar;
    ++staticVar;
}

int main (void)
{
    int i;
    void auto_static (void);

    for ( i = 0; i < 5; ++i )
        auto_static ();

    return 0;
}

```

8.15 Lista • Kimenet

```

automatic = 1, static = 1
automatic = 1, static = 2
automatic = 1, static = 3
automatic = 1, static = 4
automatic = 1, static = 5

```

Az `auto_static` függvényben két helyi változót deklarálunk. Az egyik egy automatikus egész változó, 1 kezdőértékkel, a neve `autoVar`. A másik a `staticVar` nevet viselő statikus egész változó, szintén 1 kezdőértékkel. A függvény a `printf` segítségével kiírja a két változó értékét, majd inkrementálja őket. Végül a függvény kilép.

A `main` eljárás egy ciklusban ötször meghívja az `auto_static` függvényt. A 8.15 Lista kiemelte rámutat a két változótípus közti különbségre. Az automatikus változó értékét nem lehet 1-ből kibillenteni, mert a függvény minden meghívásakor újrainicializálódik. Ugyanakkor a statikus változó értéke egyenletesen változik 1-től egészen 5-ig. A statikus változó ugyanis csak a program indulásakor inicializálódik, majd pedig megőrzi (megváltozott) értékét a függvényből való kilépések idején is.

A megoldandó feladattól függ, hogy statikus vagy automatikus változót érdemesebb-e használni. Ha fontos, hogy a változó őrizze meg az értékét az egyes függvényhívások közt (például olyan esetben, amikor a függvények hívásait számláljuk), akkor érdemes statikus változót választani. Olyankor is jó választás egy változót statikusnak kinevezni, amikor előre tudható, hogy csak egyszer kap értéket a program futása során, majd pedig változatlan marad. Ebben az esetben megspórolható az alkalomról alkalomra történő újrainicializálás. Ez akkor különösen fontos, amikor nagy adatmennyiségekről, tömbökről van szó.

Más irányból közelítve: ha egy lokális változót jó volna még a program indulásakor, minden függvényhívás előtt inicializálni, akkor is hasznosnak bizonyulhatnak a statikus változók.

Rekurzív függvények

A C programozási nyelv lehetővé teszi a rekurzív függvények használatát is. Rekurzív függvényekkel igen tömören és hatékonyan oldhatóak meg egyes feladattípusok. Különösen sikeresen alkalmazható akkor, amikor a problémát vissza lehet vezetni kisebb adathalmazt érintő (azonos típusú) problémá(k)ról. Ilyen például az egymásba ágyazott zárójeleket tartalmazó kifejezések kiértékelése. Egy másik tipikus példa a különféle adatszerkezetekben (*fák, listák*) történő adatrendezés vagy keresés.

A rekurzív függvényeket legtöbbször a permutációk számának kiszámításával szokták bemutatni, azaz az „ n faktoriálissal”, melynek jele: $n!$. Ez nem más, mint a számok szorzata 1-től n -ig. A 0 faktoriálist definíció szerint 1-nek vesszük. Az $5!$ tehát így számítható ki:

$$\begin{aligned} 5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ &= 120 \end{aligned}$$

vagy hasonlóképpen:

$$\begin{aligned} 6! &= 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ &= 720 \end{aligned}$$

Ha összehasonlítjuk az $5!$ és $6!$ értékét, szembetűnik, hogy a $6!$ az $5!$ hatszorosa, azaz $6! = 6 \cdot 5!$. Ez általánosan is igaz, bármely pozitív egész n -re:

$$n! = n \cdot (n-1)!$$

Az $n!$ ilyen módon történő megadása rekurzív definíció, hiszen az adott szám faktoriálisát egy másik (kisebb) szám faktoriálisára vezetjük vissza. Ezzel a módszerrel jól használható függvényt írhatunk $n!$ kiszámítására – ezt meg is tesszük a 8.16 Listában.

8.16 Lista • Faktoriális-számítás rekurzív módon

```
#include <stdio.h>

int main (void)
{
    unsigned int    j;
    unsigned long int factorial (unsigned int n);

    for ( j = 0; j < 11; ++j )
        printf ("%2u! = %lu\n", j, factorial (j));

    return 0;
}

// Rekurzív függvény egy pozitív egész faktoriálisának kiszámítására

unsigned long int factorial (unsigned int n)
{
    unsigned long int result;

    if ( n == 0 )
        result = 1;
    else
        result = n * factorial (n - 1);

    return result;
}
```

8.16 Lista • Kimenet

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
```

$8! = 40320$
 $9! = 362880$
 $10! = 3628800$

A factorial függvény önmagát hívja meg: ilyen a rekurzív függvény. Nézzük például a $3!$ kiszámítását. Ilyenkor a formális n paraméter értéke 3. Mivel ez nem nulla, meghívódik a

```
result = n * factorial (n - 1);
```

utasítás, ami az adott n-hez tartozóan

```
result = 3 * factorial (2);
```

-nek felel meg. A kifejezésben újra szerepel a factorial függvény, de már csak 2 a formális paraméter értéke. A factorial(2) kiszámítását meg kell várni, csak utána jöhet a hárommal való szorzás.

Annak ellenére, hogy ugyanazt a függvénynevet hívjuk meg, tudatosítanunk kell, hogy ez egy másik függvényhívás. A C-ben minden függvényhíváshoz létrejönnek a munkához szükséges lokális változók és formális paramétereik. Így tehát a result és az n változók, melyek a factorial(3) kiszámításakor életben vannak, nincsenek kihatással a factorial(2) kiszámításakor létrejövő result és n változóra.

A factorial(2) kiszámításához is a

```
result = n * factorial (n - 1);
```

utasítás kerül meghívásra, ami ebben az esetben már

```
result = 2 * factorial (1);
```

helyettesítési értékkkel bír. Ennek kiszámításával is várni kell mindaddig, amíg a factorial(1) értéke világossá nem válik. Az $n=1$ értékkel is meghívódik tehát a

```
result = n * factorial (n - 1);
```

kifejezés, ami most a

```
result = 1 * factorial (0);
```

alakot ölti. Amikor a factorial függvényt a 0 paraméterrel hívjuk meg, akkor nem következik újabb rekurzív függvényhívás, hanem a result értéke 1-et kap, és ezzel

a visszatérési értékkel kilép a függvény. Ezzel megkezdődhet a függőben maradt számítások elvégzése.

Amikor tehát a `factorial(0)`, azaz az 1 visszajut a hívó függvényhez (ami történetesen most a `factorial` függvény), akkor 1-el szorzódik, és az eredményt értékül adjuk a `result` változónak. Ennek értéke 1, ez képviseli a `factorial(1)`-et, ami visszatérési értékül adódik a hívó függvénynek (ami most újfent a `factorial` függvény). Itt 2-vel szorzódik, majd a `result` változóba jut az eredmény, ami és visszatérési értékül szolgál a `factorial(2)` kiszámításakor. Végül ez a 2-es érték szorzódik 3-mal. Itt már véget érnek a függőben levő számítások: megkaptuk a `factorial(3)`, azaz $3!$ értékét, a 6-ot. Ez a visszatérési érték íródik ki a `printf` segítségével.

Összegezve a fentieket, az alábbi kiértékelések zajlanak le a `factorial(3)` kiszámításakor:

```
factorial (3) = 3 * factorial (2)
                = 3 * 2 * factorial (1)
                = 3 * 2 * 1 * factorial (0)
                = 3 * 2 * 1 * 1
                = 6
```

Igen hasznos, ha papírral-ceruzával végigkövetjük a faktoriális függvény működését egy adott értékkel, például 4-gyel. Táblázatszerűen írjuk fel az `n` és a `result` értékét minden egyes függvényhíváskor!

A függvényekről és a különféle változókról szóló fejezetünk végéhez értünk. A C programozási nyelv egyik fő erősségeit a függvények képviselik. Nem lehet eléggyé hangsúlyozni a kicsi, jól definiált függvények fontosságát a profi programok megírásának érdekében. A könyv hátralevő részében is erősen fogunk támaszkodni a függvényekre. Érdemes tehát újra átgondolni, amit ebben a fejezetben elsajátítottunk. Az alábbi gyakorlatok gondos elvégzése sokat segíthet a tárgyalt téma megemészítésében.

Gyakorlatok

1. Gépeljük be és futtassuk le a fejezetben szereplő tizenhat példaprogramot! A kimeneteket hasonlítsuk össze a könyvben szereplő programkimenetekkel.
2. Módosítsuk a 8.4 Listát úgy, hogy a `triangularNumber` értéke a `calculateTriangularNumber` függvény visszatérési értékeként álljon elő! Ezután térjünk vissza az 5.5 Listához, és módosítsuk úgy, hogy a kívánt értékeket a frissen létrehozott `calculateTriangularNumber` függvény állítsa elő!
3. Módosítsuk a 8.8 Listát úgy, hogy a függvény `epsilon` értékét paraméterként kapja meg! Kísérletezzünk különféle `epsilon` értékkkel, és figyeljük meg, milyen hatás-sal van a négyzetgyök kiszámított értékére vonatkozóan.

4. Módosítsuk a 8.8 Listát úgy, hogy a végérték becslése (a `guess` változó értéke) jelezzen meg a `while` ciklus minden lépésében! Figyeljük meg, milyen gyorsan konvergál az érték a tényleges négyzetgyökhöz. Vonjuk le a tanulságot az iterációs lépések számát, a paraméterként kapott számértéket és a kezdeti becslést illetően.
5. A 8.8 Lista `squareRoot` függvényének kiléptető hibakorlátja nem működik jól, amikor nagyon vagy nagyon kicsi számból szeretnénk négyzetgyököt vonni. Az `x` és a `guess`² közti *különbség* helyett e két érték *hányadosát* kellene megvizsgálnunk. Minél közelebb van a hányados 1-hez, annál jobb a becslés.
Módosítsuk a 8.8 Listát a vázolt kritériumnak megfelelően.
6. Módosítsuk a 8.8 Listát oly módon, hogy a `squareRoot` függvénynek mind a paramétere, mind a visszatérési értéke dupla pontosságú lebegőpontos szám (`double`) legyen! Ennek megfelelően állítsuk be az `epsilon` változót is, hogy helyesen kövesse a dupla pontosságú számításokat.
7. Írunk olyan függvényt, amely egy egész számot fölemel egy pozitív egész hatványra! Legyen a neve `x_to_the_n`, paraméterei legyenek `x` és `n`. A függvény visszatérési értéke (azaz `x^n`) legyen `long int` típusú.
8. A másodfokú egyenlet általános alakja

$$ax^2 + bx + c = 0$$

ahol a , b és c konstans számértékeket jelentenek. Így például a

$$4x^2 - 17x - 15 = 0$$

egyenlet olyan másodfokú egyenlet, amelyben $a = 4$, $b = -17$ és $c = -15$. Az egyenlet gyökeinek hívjuk azokat az x értékek, amelyek kielégítik az egyenletet. a , b és c függvényében kiszámíthatóak az egyenlet gyökei.

Ha $b^2 - 4ac$ értéke (melynek neve: *diszkrimináns*) negatív, akkor csak képzetes gyökei (x_1 , x_2) vannak az egyenletnek, valós gyöke nincs. Ha a diszkrimináns nemnegatív, akkor a két (esetleg egybeeső) gyököt a $(-b \pm \sqrt{b^2 - 4ac}) / (2a)$ megoldóképlettel lehet kiszámítani.

Írunk programot, amely megold egy másodfokú egyenletet. A felhasználónak meg kell tudni adnia a , b és c értékét. Ha a diszkrimináns negatív, akkor jelenjen meg egy hibaüzenet, hogy nincs valós gyöke az egyenletnek. Egyébként pedig számítsa ki a program a gyököt(ek)t. A fenti képlettben `squareRoot` jelenti a négyzetgyök függvényt; használjuk azt a változatot, amelyet fejezetünkben megírtunk!

9. Két pozitív egész szám (u és v) legkisebb közös többszöröse (*lkkt*) az a legkisebb szám, amely egyaránt többszöröse u -nak és v -nek is. Így például 10 és 15 legkisebb közös többszöröse: $lkkt(10, 15) = 30$, hiszen 30 a legkisebb olyan szám, amellyel a 10 és a 15 is osztható. Írunk egy `lkkt` nevű függvényt, amely két egész számot kap paraméterként, és visszatérési értéke a két szám legkisebb közös többszöröse. Az `lkkt` kiszámításához jól használható a 8.6 Lista, melyben két szám legnagyobb közös osztóját (`lnko`, angolul `gcd`) számítottuk ki. A két függvény közti összefüggés a következő:

$$lkkt(u, v) = uv / lnko(u, v), \text{ ahol } u, v \geq 0$$

10. Írunk egy `prime` nevű függvényt, mely 1-et ad vissza, ha a paraméterként átadott szám prímszám, egyébként pedig 0-t.
11. Írunk egy `arraySum` nevű kétparaméteres függvényt. Paraméterei legyenek: egy tömb neve és elemszáma. Visszatérési értéke legyen a tömb elemeinek összege.
12. Az i sorból és j oszlopból álló M mátrix *transzponálja* a j sorból és i oszlopból álló N mátrix, mely szemléletesen a mátrix 45 fokos irányszögű egyenesre való tükrözésével áll elő. Pontosabban: minden alkalmas a és b értékre fennáll az $N_{a,b} = M_{b,a}$ egyenlőség.
 - a) Írunk egy `transposeMatrix` nevű függvényt, melynek két paramétere legyen egy 4x5-ös és egy 5x4-es mátrix. A függvény transzponálja a 4x5-ös mátrixot, és tárolja el az 5x4-es mátrixban. Írunk a teszteléshez alkalmas `main` eljárást is.
 - b) Változó méretű tömbök használatával írjuk újra az a) feladatrész `transposeMatrix` függvényét úgy, hogy a sorok és oszlopok száma is paraméter legyen. Így hajtsuk végre a megadott méretű mátrix transzponálását!
13. Módosítsuk a 8.12 Lista `sort` függvényét úgy, hogy egy harmadik paraméterként meg lehessen adni, hogy növekvő vagy csökkenő sorrendben szeretnénk-e rendezni a tömböt! Ennek megfelelően korrigáljuk a `sort` függvényt, hogy helyes eredményt adjon.
14. Az utolsó négy gyakorlat függvényeit módosítsuk úgy, hogy használjanak paraméterek helyett globális változókat! Az előző gyakorlat például használjon globális változóként megadott tömböt.
15. Módosítsuk a 8.14 Listát úgy, hogy helytelen alapszám megadása esetén a program kérje be újra a felhasználótól az értéket! Ezt mindaddig ismételje a program, amíg a felhasználó helyes értéket nem ad meg.
16. Módosítsuk a 8.14 Listát úgy, hogy a felhasználó egymás után adhassa meg az átalakítandó számokat (és alapszámot) mindaddig, amíg 0-t nem ad meg! Ez jelentse az átszámítás-sorozat végét.

9

Adatszerkezetek

A 7. fejezetben tanultunk a tömbökről, amelyek lehetővé teszik, hogy azonos típusú adatok egy halmazát egyetlen logikai egységgé kezeljünk. A tömb bármely elemét elérhetjük a megfelelő sorszám (index) megadásával.

A C nyelvben más eszközök is rendelkezésre állnak az adatok csoportosítására. Az adatszerkezetről (**struct**) van szó, mely fejezetünk témája lesz. Az adatszerkezet igen hatékony eszköz, melyet valószínűleg sok programban használni fogunk.

Tegyük fel, hogy egy dátumot szeretnénk eltárolni egy programban, például azt, hogy 2004.09.25. Előfordulhat, hogy ezt a dátumot egy program kimenetének fejlécében szeretnénk használni, vagy akár számítási célból lesz szükségünk rá. Kézenfekvőnek tűnik tehát az a megoldás, hogy az évet, hónapot és napot tegyük be egy-egy egész változóba:

```
int month = 9, day = 25, year = 2004;
```

Ez egy remekül használható megközelítés. De tegyük fel, hogy programunknak egy másik dátumot is tárolnia kéne, például egy adott tárgy beszerzési dátumát. Hasonló módon tovább lehet haladni a `purchaseMonth`, `purchaseDay` és `purchaseYear` deklarálásával. Ebből egyértelműen kiderül a beszerzés dátuma.

Ebből a megközelítésből adódóan három különböző, mégis egymáshoz tartozó változót kell manipulálni a programban minden dátum esetében. Érezzük, hogy jobb lenne ezt a három összetartozó adatot csoportosan kezelni. Pontosan ez történik a C nyelv adatszerkezetének (**struct**) használatakor.

Dátumot tároló adatszerkezet

Definiálunk date néven egy olyan adatszerkezetet, amelyben három összetevő található az év, a hónap és a nap nyilvántartására. Ennek a szintaxisa egyszerű:

```
struct date
{
    int month;
    int day;
    int year;
};
```

Az így definiált date struktúrának három tagja (*member*) van: month, day és year. Bizonyos értelemben egy újfajta adattípust definiálunk ezzel az utasítással. A továbbiakban ilyen date típusuként is megadhatunk változókat a következőképp:

```
struct date today;
```

A fentebb említett purchaseDate is lehet ilyen típusú:

```
struct date purchaseDate;
```

Sőt, ezt a két deklarációt egyetlen sorban is megadhatjuk:

```
struct date today, purchaseDate;
```

Az adatszerkezetek változóinak manipulálásakor más szintaxist kell követnünk, mint például az int, a float vagy a char típusoknál. Az adatszerkezet valamely tagját úgy lehet elérni, hogy az adatszerkezet neve után pontot írva megnevezzük a tagot is. A today adatszerkezet day tagját például a következőképp állíthatjuk be 25-re:

```
today.day = 25;
```

Vigyázzunk, nehogy szóközt írunk a pont valamelyik oldalára. A year tagot így állíthatjuk be 2004-re:

```
today.year = 2004;
```

A következőképpen vizsgálhatjuk meg, hogy a today adatszerkezet month tagja 12-e:

```
if ( today.month == 12 )
    nextMonth = 1;
```

Próbálja meg kitalálni az olvasó az alábbi kódrészlet hatását:

```
if ( today.month == 1 && today.day == 1 )
    printf ("Boldog új évet!!!\n");
```

A 9.1 Lista a fejezetben eddig tanultakat foglalja össze egy C program formájában:

9.1 Lista • Az adatszerkezet bemutatása

```
// Az adatszerkezet használatát bemutató program

#include <stdio.h>

int main (void)
{
    struct date
    {
        int month;
        int day;
        int year;
    };

    struct date today;

    today.month = 9;
    today.day = 25;
    today.year = 2004;

    printf ("A mai dátum: %i/%i/.%i.\n", today.month, today.day,
           today.year % 100);

    return 0;
}
```

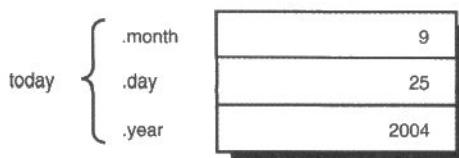
9.1 Lista • Kimenet

A mai dátum: 9/25/04.

A main eljárás első utasítása megadja, hogy a date adatszerkezetnek három egész tagja legyen: month, day és year. A második utasításban a today változót ilyen date szerkezetként deklaráljuk. Az első utasítás csak tudatja a fordítóprogrammal, hogy milyen legyen a date adatszerkezet, de ez még nem jár együtt semmilyen memória foglalással. Ellenben a következő utasítással (ahol egy változó deklarációja zajlik) már lefoglalunk egy adott memóriaterület a today változó számára. Ne haladjunk tovább az olvasással, amíg nem világos a különbség egy adatszerkezet definíciója és egy adott szerkezetű változó deklarációja között!

A `today` deklarációja után a program értéket ad a három adattagnak, ahogy az a 9.1 ábrán is látható.

```
today.month = 9;
today.day = 25;
today.year = 2004;
```



9.1 ábra

Értékadások egy adatszerkezet tagjai számára

Az értékadások után az adatszerkezet tartalmát kiíratjuk egy megfelelő `printf` utasítással. Az évből csak a két utolsó számjegyet kérjük (egy 100-zal történő maradékos osztás maradványnak kiírásával), így csak 04 jelenik meg. Emlékezzünk vissza: a `% .2i` formátumjelző karakterlánc hatására a (két karakternyi helyen kiírt) egész szám üres helyiértékein nullák lesznek. Ennek eredményeként az egyjegyű évszámok is szabványos módon, két helyiértéken ábrázolva jelennek meg.

Adatszerkezetek használata kifejezésekben

A kifejezésekben az adatszerkezetek tagjai ugyanúgy viselkednek, mint bármely normál C nyelvű változó. Ha egy egész adattagot egy másik egész értékkal osztunk, akkor a művelet az egész aritmetika szabályainak megfelelően fog történni:

```
century = today.year / 100 + 1;
```

Írunk egy olyan programot, amely bekéri a mai dátumot, és ez alapján kiírja a holnapit! Első látásra ez egyszerű feladatnak tűnik; mintha az alábbi művelet meg is oldaná a kérdést (`today=ma, tomorrow=holnap`):

```
tomorrow.month = today.month;
tomorrow.day    = today.day + 1;
tomorrow.year  = today.year;
```

Az esetek túlnyomó többségében ez a számítás helyes eredményt ad, ám sajnos vannak kivételek:

1. Amikor a hónap utolsó napját adjuk meg.
2. Amikor az év utolsó napját adjuk meg (azaz december 31-ét).

A hónap utolsó napjának vizsgálatához készítünk egy egészekből álló tömböt, amely nyilvántartja a hónapok napjainak számát, az alábbi módon:

```
int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Ebben a 12 elemű daysPerMonth tömbben könnyen rá tudunk keresni bármely hónap napjainak a számára. Az i. hónaphoz a daysPerMonth[i - 1] jelent a keresett számot. Például április (4. hó) napjainak a számát a daysPerMonth[3] adja meg: ez éppen 30. (Lehetne 13 elemű tömböt is készíteni, amelyben minden korrekció nélkül közvetlenül a daysPerMonth[i] adná meg az i. hó napjainak számát. Így a tömbelem sorszámából nem kellene egyet levonni ahhoz, hogy megtaláljuk a napok számát. Ízlés dolga, hogy a 12 vagy 13 elemű tömb alkalmazása felé indulunk el.)

Ha az derül ki, hogy egy megadott nap a hónap utolsó napja, akkor a másnapi dátum egyszerűen úgy számítható ki, hogy a hónapszámot eggyel megnöveljük, valamint a napok számát 1-re állítjuk.

A fenti második problematikus esetben meg kell határozni, hogy a mai dátum hó végén van-e, és hogy ezzel egy időben a szóban forgó hónap december-e (azaz hónapszám 12). Ilyenkor mind a hónapot, mind a napot 1-re kell állítanunk, és az évek számát eggyel meg kell növelnünk.

A 9.2 Lista bekéri a felhasználótól a mai dátumot, kiszámítja a holnapi dátumot, majd kiírja az eredményt.

9.2 Lista • A holnapi dátum kiszámítása

// A holnapi dátumot kiszámító program

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

```
int main (void)
```

{}

struct date

{

int month

```

printf ("Adja meg a mai dátumot (hh nn éééé): ");
scanf ("%i%i%i", &today.month, &today.day, &today.year);

if ( today.day != daysPerMonth[today.month - 1] ) {
    tomorrow.day = today.day + 1;
    tomorrow.month = today.month;
    tomorrow.year = today.year;

}

else if ( today.month == 12 ) {           // év vége
    tomorrow.day = 1;
    tomorrow.month = 1;
    tomorrow.year = today.year + 1;
}

else {                                     // hó vége
    tomorrow.day = 1;
    tomorrow.month = today.month + 1;
    tomorrow.year = today.year;
}

printf ("A holnapi dátum: %i/%i/.2i.\n", tomorrow.month,
        tomorrow.day, tomorrow.year % 100);
return 0;
}

```

9.2 Lista • Kimenet

Adja meg a mai dátumot (hh nn éééé): **12 17 2004**
 A holnapi dátum: 12/18/04.

9.2 Lista • Kimenet (újfuttatás)

Adja meg a mai dátumot (hh nn éééé): **12 31 2005**
 A holnapi dátum: 1/1/06.

9.2 Lista • Kimenet (2. újfuttatás)

Adja meg a mai dátumot (hh nn éééé): **2 28 2004**
 A holnapi dátum: 3/1/04.

A kimenetre nézve azonnal látszik egy hiba: 2004. február 28. után nem március 1. következik, hanem február 29., hiszen a 2004 szökőév – ám erre nem készítettük föl a programot. A következőkben ezt is megoldjuk, de először nézzük át a program működését.

A date adatszerkezet definiálása után ezzel a típussal deklaráljuk a today és a tomorrow változót. A program bekéri a mai dátumot. A három beadott számértéket a today.month, a today.day, és a today.year változókban tároljuk el. Ezután megvizsgáljuk, hogy a megadott nap egy hónap utolsó napja-e. Összehasonlítjuk a today.day értékét

a daysPerMonth[today.month - 1] értékével. Ha a két érték nem egyezik meg, akkor a holnapi nap kiszámításához egyszerűen hozzáadunk egyet a mai nap értékéhez, illetve a keresett év és hónap adathoz hozzárendeljük a mai nap év/hó adatát.

Ha a hónap utolsó napjáról van szó, akkor még egy vizsgálatot végzünk: év vége van-e. Ha a hónap száma 12, akkor december 31. van, így a másnapi dátum a következő év január elseje. Ha ellenben nem decemberben vagyunk, akkor (az azonos év) következő hónapjának első napja lesz a keresett dátum.

Miután kiszámítottuk a másnapi dátumot, egy alkalmas printf utasítással kiírjuk a képernyőre, majd a program befejezi futását.

Függvények és adatszerkezetek

Térjünk vissza az előbb felfedezett hibához: a programnak kezelnie kellene a szökőéveket. A programban a februári napok számaként 28 van megadva, így az ezt követő napként minden március 1-et ad. Be kell illeszteni egy extra ellenőrzést, hogy szökőév van-e. Ha igen, és az aktuális hónap a február, akkor a napok száma 29, egyébként pedig a daysPerMonth tömbben található érték.

Ennek a különleges esetnek a felismeréséhez hozzunk létre egy numberOfDays függvényt – ez fogja visszaadni az adott hónap napjainak számát. A függvény ellenőrzi, hogy szökőév van-e, valamint elvégzi a szükséges keresést a daysPerMonth tömbben. A main eljárásban csak annyit kell változtatnunk, hogy az if utasításban a today.day értékét a daysPerMonth[today.month - 1] helyett a numberOfDays által visszaadott értékkel hasonlítjuk össze.

Vizsgáljuk meg alaposan a 9.3 Listát. Milyen paramétert adunk át a numberOfDays függvénynek?

9.3 Lista • A holnapi dátum kiszámítását végző program újraírt változata

```
// A holnapi dátumot kiszámító program újraírt változata

#include <stdio.h>
#include <stdbool.h>

struct date
{
    int month;
    int day;
    int year;
};

int main (void)
{
```

```

    struct date today, tomorrow;
    int numberOfDays (struct date d);

    printf ("Adja meg a mai dátumot (hh nn éééé): ");
    scanf ("%i%i%i", &today.month, &today.day, &today.year);

    if ( today.day != numberOfDays (today) ) {
        tomorrow.day = today.day + 1;
        tomorrow.month = today.month;
        tomorrow.year = today.year;
    }
    else if ( today.month == 12 ) {      // év vége
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year + 1;
    }
    else {                                // hónap vége
        tomorrow.day = 1;
        tomorrow.month = today.month + 1;
        tomorrow.year = today.year;
    }

    printf ("A holnapi dátum: %i/%i%.2i.\n", tomorrow.month,
            tomorrow.day, tomorrow.year % 100);

    return 0;
}

// Függvény, amivel az adott hónap napjainak számát kérdezzük le.

int numberOfDays (struct date d)
{
    int days;
    bool isLeapYear (struct date d);
    const int daysPerMonth[12] =
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    if ( isLeapYear (d) == true && d.month == 2 )
        days = 29;
    else
        days = daysPerMonth[d.month - 1];

    return days;
}

// Függvény, mellyel megvizsgáljuk, hogy szökőév van-e

bool isLeapYear (struct date d)
{

```

```

    bool leapYearFlag;

    if ( (d.year % 4 == 0 && d.year % 100 != 0) || 
        d.year % 400 == 0 )
        leapYearFlag = true; // It's a leap year
    else
        leapYearFlag = false; // Not a leap year

    return leapYearFlag;
}

```

9.3 Lista • Kimenet

Adja meg a mai dátumot (hh nn éééé): **2 28 2004**
 A holnapi dátum: 2/29/04.

9.3 Lista • Kimenet (újrafuttatás)

Adja meg a mai dátumot (hh nn éééé): **2 28 2005**
 A holnapi dátum: 3/1/05.

Szembeötlő, hogy a date adatszerkezet definiálása minden függvényen kívül, már rögtön a program legelején megtörténik. Így a program bármely pontján elérhető lesz. Az adatszerkezetek definíciója nagyon hasonló a változókhoz. Ha egy függvényen belül adunk meg egy adatszerkezetet, akkor az csak abban a függvényben lesz elérhető. Ezt hívjuk lokális adatszerkezet-definícióknak. Ha ellenben függvényen kívül definiálunk adatszerkezeteket, akkor azok globálisak lesznek. A globális adatszerkezet-definíciók lehetővé teszik, hogy a program bármelyik (függvényen belüli vagy azon kívüli) változója ilyen adatszerkezetű lehessen.

A main eljárásban az alábbi függvényprototípus-deklarációt találjuk:

```
int numberOfDays (struct date d);
```

Ez tájékoztatja a fordítóprogramot arról, hogy a `numberOfDays` függvény visszatérési értéke egy egész szám, és paraméterként egy `date` adatszerkezetű változót vár.

A `today.day`-t a `daysPerMonth[today.month - 1]` helyett a `numberOfDays` visszatérési értékével vetjük össze:

```
if ( today.day != numberOfDays (today) )
```

A függvényhívásból látszik, hogy a `today` adatszerkezetet adjuk át paraméterként. A `numberOfDays` függvény fejlécén belül is meg kell adnunk, hogy ilyen paramétert várunk.

A paraméterként átadott adatszerkezetek nem annyira a paraméterként használt tömbök-höz, hanem sokkal inkább a paraméter-változókhöz hasonlítanak. A formális paraméter-változó (jelen esetben adatstruktúra) függvényen belüli megváltoztatása nem hat vissza arra az adatstruktúrára, amit paraméterként átadtunk. A függvény meghívásakor másolat készül a paraméterként kapott adatszerkezetről; ezzel dolgozhatunk a függvényben.

A `numberOfDays` függvény először megvizsgálja (az `isLeapYear` függvény meghívásával), hogy szökőév van-e, és azon belül február-e a hónap. Az `isLeapYear` függvényre még visszatérünk. Most elég annyit megfigyelni, hogy az

```
if ( isLeapYear (d) == true && d.month == 2 )
```

utasítás alapján azt sejtjük, hogy a függvény `true` értékkel tér vissza, ha szökőévről van szó, és `false` értékkel akkor, ha nem. Ezzel ahhoz a témahez nyúlunk vissza, amiről a 6. fejezetünkben már szó volt. Emlékeztetőül: a `<stdbool.h>` fejlécállomány beillesztésével használhatóvá válnak a `bool`, `true` és `false` értékek – ez történik a 9.3 Listában is.

Az előző `if` utasításra visszatérve érdemes megfigyelni, mennyire találó az „`isLeapYear`” (szökőév?) függvénynév – hatására kényelmesen olvashatóvá válik az utasítás („ha szökőév...”), és az `is` sejthető, hogy ennek visszatérési értéke valamilyen logikai érték, *igen/nem*.

Visszatérve a programra: ha kiderül, hogy egy szökőév februárjában vagyunk, akkor a havi napok számát (`days`) 29-re állítjuk, egyébként pedig a `daysPerMonth` tömb adott hónapra vonatkozó értékét olvassuk ki, és ezt rendeljük hozzá a `days` változóhoz. Ennek értéke visszakerül a hívó `main` eljárásba, és folytatódik a program futása.

Az `isLeapYear` függvény csöppet sem bonyolult. Egyszerűen megvizsgálja a paraméterként kapott `today` adatszerkezetben található `year` (év) adattagot, és amennyiben szökőévről van szó, `true` értékkel tér vissza, ellenkező esetben `false` értékkel.

Csaklógyakorlatként fogalmazzuk át programunkat úgy, hogy szébb, strukturáltabb legyen: a másnapi dátum meghatározására szolgáló részt tegyük be egy különálló függvénybe. Legyen ennek a neve `dateUpdate`, paramétere pedig legyen a mai dátum. Visszatérési érték kül adjva vissza a kiszámított másnapi dátumot. Mindezt a 9.4 Lista valósítja meg.

9.4 Lista • A holnapi dátum kiszámítását végező program 2. újraírt változata

```
// A holnapi dátumot kiszámító program, 2. újraírt változat
```

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

struct date
{
```

```
int month;
int day;
int year;
};

// A holnapi dátumot kiszámító függvény

struct date dateUpdate (struct date today)
{
    struct date tomorrow;
    int numberOfDays (struct date d);

    if ( today.day != numberOfDays (today) ) {
        tomorrow.day = today.day + 1;
        tomorrow.month = today.month;
        tomorrow.year = today.year;
    }
    else if ( today.month == 12 ) {           // év vége
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year + 1;
    }
    else {                                     // hónap vége
        tomorrow.day = 1;
        tomorrow.month = today.month + 1;
        tomorrow.year = today.year;
    }

    return tomorrow;
}

// Függvény, mellyel az adott hónap napjainak számát kérdezzük le.

int numberOfDays (struct date d)
{
    int days;
    bool isLeapYear (struct date d);
    const int daysPerMonth[12] =
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    if ( isLeapYear (d) && d.month == 2 )
        days = 29;
    else
        days = daysPerMonth[d.month - 1];

    return days;
}
```

```

// Függvény, mellyel megvizsgáljuk, szökőév van-e

bool isLeapYear (struct date d)
{
    bool leapYearFlag;

    if ( (d.year % 4 == 0 && d.year % 100 != 0) || 
        d.year % 400 == 0 )
        leapYearFlag = true; // szökőév
    else
        leapYearFlag = false; // nem szökőév

    return leapYearFlag;
}

int main (void)
{
    struct date dateUpdate (struct date today);
    struct date thisDay, nextDay;

    printf ("Adja meg a mai dátumot (hh nn éééé): ");
    scanf ("%i%i%i", &thisDay.month, &thisDay.day,
           &thisDay.year);

    nextDay = dateUpdate (thisDay);

    printf ("A holnapi dátum: %i/%i/.2i.\n", nextDay.month,
           nextDay.day, nextDay.year % 100);

    return 0;
}

```

9.4 Lista • Kimenet

Adja meg a mai dátumot (hh nn éééé): **2 28 2008**
 A holnapi dátum: **2/29/08.**

9.4 Lista • Kimenet (újrafuttatás)

Adja meg a mai dátumot (hh nn éééé): **2 22 2005**
 A holnapi dátum: **2/23/05.**

A main-beli

```
next_date = dateUpdate (thisDay);
```

utasítás jól mutatja, hogy adatszerkezetet át lehet adni függvény-paraméterként, és egy függvény visszatérési értéke is lehet adatszerkezet. A dateUpdate függvény fejlécéből is látszik, hogy date adatszerkezetű paramétert vár, és ugyanilyen típusú a visszatérési értéke is. A függvényben szinte ugyanaz a kód látható, mint a 9.3 Lista main eljárásában.

A numberOfDays és az isLeapYear függvény teljesen azonos maradt az előző programbeli változattal.

Csak akkor haladjon tovább az olvasó, ha világosan érti az előző program függvényhívásainak rendszerét: a main meghívja a dateUpdate-et, amely aztán a numberOfDays-t, ez pedig futása közben rákérdezi az isLeapYear -re.

Adatszerkezet az idő tárolására

Tegyük fel, hogy időpontokat is szeretnénk tárolni a programban, óra:perc:másodperc bontásban. Minthogy a dátumok kezelésénél remekül bevált az adatszerkezet alkalmazása, kézenfekvő, hogy az időpontokat is kezeljük így. Legyen ennek definíciója a következő:

```
struct time
{
    int    hour;
    int    minutes;
    int    seconds;
};
```

A legtöbb számítógépes alkalmazás 24 órás időábrázolást használ, így nem kell vesződni a délelőtt és délután kérdésével. Az órák száma 0-val kezdődik éjfélkor, és egyesével nő egészen 23-ig, ami éjjel 11 órát jelent. 4:30 jelenti a hajnali fél ötöt, míg 16:30 jelenti a délutáni fél ötöt. 12:00 jelenti a delet, és 00:01 az éjfél utáni első perct.

Elvileg minden számítógépben van egy belső óra, ami minden működik. Többféle célja is van: tájékoztatja a felhasználót a pontos időről, használható bizonyos események kiváltására vagy programok időzített elindítására, vagy bizonyos időpillanatok elmentésére. Vannak programok, melyek ehhez az órához hozzá tudnak férfi. Szokott lenni egy olyan program is, amely másodpercenként lefutva frissít a számítógép memoriájában tárolt időadatot.

Próbálunk megírni egy ilyesfélé programot, amely tehát másodpercenként frissíti az időadatot. Ha egy másodpercre belegondolunk, hamar átlátható, hogy a feladat lényegében ugyanaz, mint az előbbi programban, mely a dátumot mozdította el egyel.

Ahogy a következő nap megkeresésének megvoltak a maga peremfeltételei, úgy ebben a feladatban is így van:

1. Ha a másodpercek értéke eléri a 60-at, nullázni kell, és a percek értékét inkrementálni.
2. Ha a percek értéke eléri a 60-at, nullázni kell, és az órák értékét inkrementálni.
3. Ha az órák értéke eléri a 24-et, akkor minden időadatot nullázni kell.

A 9.5 Listában a timeUpdate függvénynek adjuk át paraméterként az aktuális időpillanatot; visszatérési értéke az egy másodperccel későbbi időpillanat.

9.5 Lista • Az időpont növelése egy másodperccel.

```
// Az időpontot inkrementáló program

#include <stdio.h>

struct time
{
    int hour;
    int minutes;
    int seconds;
};

int main (void)
{
    struct time    timeUpdate (struct time    now);
    struct time    currentTime, nextTime;

    printf ("Adja meg az időpontot (óó:pp:mm): ");
    scanf ("%i:%i:%i", &currentTime.hour,
           &currentTime.minutes, &currentTime.seconds);

    nextTime = timeUpdate (currentTime);

    printf ("A következő másodperc: %.2i:%.2i:%.2i\n", nextTime.hour,
           nextTime.minutes, nextTime.seconds );

    return 0;
}

// Függvény az időpillanat eltolására egy másodperccel

struct time timeUpdate (struct time    now)
{
    ++now.seconds;
    if ( now.seconds == 60 ) {          // következő perc
        now.seconds = 0;
    }
}
```

```

++now.minutes;

if ( now.minutes == 60 ) { // következő óra
    now.minutes = 0;
    ++now.hour;

    if ( now.hour == 24 ) // éjfél
        now.hour = 0;
}
}

return now;
}

```

9.5 Lista • Kimenet

Adja meg az időpontot (óó:pp:mm): **12:23:55**
A következő másodperc: **12:23:56**

9.5 Lista • Kimenet (újrafuttatás)

Adja meg az időpontot (óó:pp:mm): **16:12:59**
A következő másodperc: **16:13:00**

9.5 Lista • Kimenet (2. újrafuttatás)

Adja meg az időpontot (óó:pp:mm): **23:59:59**
A következő másodperc: **00:00:00**

A main eljárásban felszólítjuk a felhasználót az időpont megadására. A scanf meghívásakor a

"%i:%i:%i"

formátumjelző karakterláncot használjuk az adatok beolvasásához. Ahol nem formátumjelző karaktert adunk meg (esetünkben kettőspontot), ott azzal azt jelezzük, hogy konkréten milyen karaktert várunk a bemenetről. Így a 9.5 Lista formátumjelző karakterláncával azt adjuk a fordítóprogram tudtára, hogy három egész számot várunk, köztük kettősponttal. A 16. fejezetben megtanuljuk, hogy a scanf visszatérési értékéhez hogyan használható fel annak megvizsgálására, hogy a várt formátumban adta-e meg az adatokat a felhasználó.

Az időpont bekérése után a program meghívja a timeUpdate függvényt, átadva neki a currentTime (*pillanatnyi idő*) paramétert. A függvény visszatérési értékét a time adatszerkezetű nextTime kapja meg, melyen egy alkalmas printf utasítás megjelenít a képernyőn.

A `timeUpdate` függvény azzal kezdi meg működését, hogy egy másodperccel inkrementálja a now-ban tárolt másodpercek számát, majd megvizsgálja, hogy ezzel elérőük-e a 60-at. Ha igen, akkor nullázza az értéket, ám a now-ban tárolt percek számát eggyel növeli. Ha a percek száma is elérte a 60-as küszöbértéket, akkor nullázza azt, és az órák számát növeli eggyel. Végül megnézi a függvény, hogy elérőük-e az éjfélét (amit a „24” óra helyett programunkban „0” órával jelölünk): ha igen, akkor nullázza az órák számát. A függvény a megnövelte időpontot tároló now értékét adja vissza a hívó eljárásnak.

Adatszerkezetek inicializálása

Az adatszerkezeteket ugyanúgy inicializálhatjuk, mint a tömböket: kapcsos zárójelben fel kell sorolni a kezdőértékeket, vesszővel elválasztva.

A `date` adatszerkezetű `today` változó „2005. július 2”-ára történő inicializálásához az alábbi utasítás használható:

```
struct date today = { 7, 2, 2005 };
```

A

```
struct time this_time = { 3, 29, 55 };
```

utasítással „3 óra 29 perc 55 másodperc”-re állítható be a `time` adatszerkezetű `this_time`. Más változókhöz hasonlóan, ha a `this_time` egy függvény (automatikus) lokális változója, akkor a függvény minden meghívásakor lefut az inicializáció. Ha azonban az adatszerkezet statikus (vagyis ott áll előtte a `static` típusmódsító), akkor csak egyszer inicializálódik, mégpedig a program elindulásakor. Bárminelyik esetről legyen is szó, a kapcsos zárójelben felsorolt értékeknek konstans kifejezésnek kell lenniük.

A tömbök inicializációjához hasonlóan megtehetjük azt is, hogy nem adunk kezdőértéket minden adattagnak. Így a

```
struct time time1 = { 12, 10 };
```

utasítás 12-re állítja be a `time1.hour` értékét és 10-re a `time1.minutes`-t, de nem ad kezdőértéket a `time1.seconds`-nak. Ennek definiáltan marad az értéke.

Az inicializálni kívánt adattagot név szerint is megnevezhetjük. Ilyenkor az általános szintaxis a következő:

```
.member = value
```

Ezzel tetszőleges sorrendben tetszőleges adattagot inicializálhatunk. A

```
struct time time1 = { .hour = 12, .minutes = 10 };
```

utasítással például az előzővel azonos módon inicializálhatjuk a `time1` változót, míg a

```
struct date today = { .year = 2004 };
```

utasítás a `today` változónak csak az év adattagját állítja be 2004-re.

Összetett betűkonstansok

Egyetlen hozzárendeléssel is értéket adhatunk egy teljes adatszerkezetnek az úgynévezett „összetett betűkonstansok” (*compound literals*) segítségével. Ha a `today` egy korábban deklarált `date` adatszerkezetű változó, akkor a 9.1 Listában látható hozzárendelés az alábbi módon is megoldható:

```
today = (struct date) { 9, 25, 2004 };
```

A hozzárendelés a program bármely részén elhelyezhető mivel ez nem deklaráció.

A típusátalakító operátor tudatja a fordítóprogrammal a kifejezés típusát, amely ebben az esetben a `date` adatszerkezet. Ezután kapcsos zárójelben, vesszővel elválasztva sorakoznak az adattagokhoz rendelendő értékek – akárcsak az inicializációjánál.

Itt is használható a `.tag` jelölésmód, amely sorrendfüggelenséget biztosít:

```
today = (struct date) { .month = 9, .day = 25, .year = 2004 };
```

Ha az adattagok nevei nincsenek megadva, akkor a deklarációjánál megadott sorrendben történnek a hozzárendelések.

Írjuk át a `dateUpdate` függvényt az összetett betűkonstansok jelölésmódjával.

```
// A holnapi dátumot kiszámító függvény - az összetett betűkonstansok
// jelölésmódjával

struct date dateUpdate (struct date today)
{
    struct date tomorrow;
    int numberOfDays (struct date d);

    if ( today.day != numberOfDays (today) )
        tomorrow = (struct date) { today.month, today.day + 1,
today.year };
    else if ( today.month == 12 )           // év vége
```

```

        tomorrow = (struct date) { 1, 1, today.year + 1 };
    else                                // hónap vége
        tomorrow = (struct date) { today.month + 1, 1, today.year };

    return tomorrow;
}

```

Az összetett betűkonstansok használata természetesen nem kötelező – a programozó belátására van bízva, él-e ezzel a lehetőséggel. Ebben a példában olvashatóbbá, rövidebbé vált a dateUpdate függvény kódja a tömör jelölésmód révén.

Összetett betűkonstansokat használhatunk bárhol, ahová adatszerkezetből álló kifejezés kívánkozik. Teljesen érvényes, ám feleslegesen túltömörített az alábbi kifejezésmód:

```
nextDay = dateUpdate ((struct date) { 5, 11, 2004 } );
```

A dateUpdate függvény egy date adatszerkezetet vár, ami típusát illetően megegyezik a paraméterként átadott összetett betűkonstans-kifejezéssel.

Adatszerkezetekből álló tömbök

Láttuk, milyen hasznos a logikailag összetartozó adatok adatszerkezetben történő egybefoglalása. A time adatszerkezettel például (három helyett) csak egyetlen változót kell számon tartani a programban. Ha tíz időpont kerül elő egy listában, akkor így harminc helyett csak tíz változót kell karbantartani.

Erre az esetre azonban van egy még hatékonyabb eszköz: a C nyelv két erősséget, az adatszerkezeteket és a tömböket lehet egyszerre is használni. Egy tömbben ugyanis nemcsak alaptípusú elemeket lehet tárolni, hanem tetszőleges adatszerkezettel definiált elemeket is. Az alábbi deklarációval például egy tíz elemű tömböt adatunk meg, melynek neve experiments (*kísérletek*), elemei pedig time adatszerkezetűek:

```
struct time experiments[10];
```

Hasonlóképp, a

```
struct date birthdays[15];
```

utasítás a birthdays (*születésnapok*) tömböt 15 elemüként adja meg, melynek elemei dátumok, azaz date adatszerkezetű változók. Az egyes elemekre értelemszerűen történhet a hivatkozás. A birthdays tömbben a második születésnapot a következő utasítássorozattal állíthatjuk be 1968. augusztus 8-ra:

```
birthdays[1].month = 8;
birthdays[1].day   = 8;
birthdays[1].year  = 1986;
```

Az experiments[4]-et átadhatjuk például egy checkTime függvénynek:

```
checkTime (experiments[4]);
```

A checkTime-nak persze tudnia kell arról, hogy milyen paramétert fog kapni:

```
void checkTime (struct time t0)
{
    .
    .
}
```

Az adatszerkezeteket tartalmazó tömbök inicializálása a többdimenziós tömbökhez hasonlít. A

```
struct time runTime [5] =
{ {12, 0, 0}, {12, 30, 0}, {13, 15, 0} };
```

utasítás a runTime tömb első három elemének ad kezdőértéket, beállítva őket 12:00:00, 12:30:00 és 13:15:00 értékűre. A belső zárójelek elhagyhatóak, azaz a fenti kifejezés egyenértékű az alábbival:

```
struct time runTime[5] =
{ 12, 0, 0, 12, 30, 0, 13, 15, 0 };
```

A

```
struct time runTime[5] =
{ [2] = {12, 0, 0} };
```

kifejezés csak a tömb harmadik elemét inicializálja, míg a

```
static struct time runTime[5] = { [1].hour = 12, [1].minutes = 30 };
```

kifejezés csak a második elem „óra” és „perc” adattagját állítja be 12-re és 30-ra.

A 9.6 Lista létrehoz egy time elemekből álló, testTimes nevű tömböt, majd meghívja a 9.5 Lista timeUpdate függvényét. Helytakarékkosság miatt itt nem idézzük újra ezt a függvényt, csak egy megjegyzéssel jelöljük a helyét.

A testTimes tömbbe öt különböző időpont kerül: 11:59:59, 12:00:00, 1:29:59, 23:59:59 és 19:12:27. A 9.2 ábra segít megérteni, hogy ezek az információk (a testTimes tömb elemei) hogyan helyezkednek el a memóriában. minden egyes elemet elérhetünk a sorszámaival (0-4); adattagjaik (hour, minutes, seconds) pedig úgy érhetők el, hogy egy pont után megadjuk a nevüket.

A program kiírja a testTimes tömb valamennyi kezdőértékét, majd – meghívva a timeUpdate függvényt minden elemre – megtekinthetjük a megnövelt időpontokat.

9.6 Lista • Adatszerkezetekből álló tömbök bemutatása

```
// Adatszerkezetekből álló tömböket bemutató program

#include <stdio.h>

struct time
{
    int hour;
    int minutes;
    int seconds;
};

int main (void)
{
    struct time timeUpdate (struct time now);
    struct time testTimes[5] =
    { { 11, 59, 59 }, { 12, 0, 0 }, { 1, 29, 59 },
      { 23, 59, 59 }, { 19, 12, 27 } };
    int i;

    for ( i = 0; i < 5; ++i ) {
        printf ("Az idő %.2i:%.2i:%.2i", testTimes[i].hour,
               testTimes[i].minutes, testTimes[i].seconds);
        testTimes[i] = timeUpdate (testTimes[i]);
        printf (" ...egy másodperccel később már %.2i:%.2i:%.2i\n",
               testTimes[i].hour, testTimes[i].minutes,
               testTimes[i].seconds);
    }

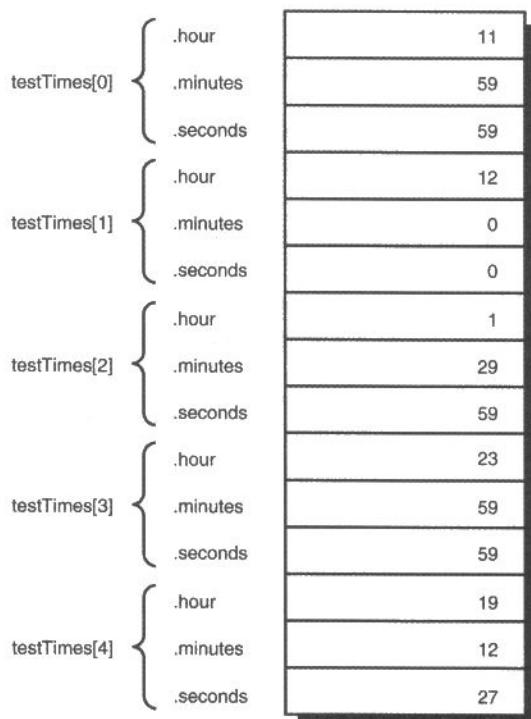
    return 0;
}

// ***** Ide illesztendő be a timeUpdate függvény *****
```

9.6 Lista • Kimenet

Az idő 11:59:59 ... egy másodperccel később már 12:00:00
 Az idő 12:00:00 ... egy másodperccel később már 12:00:01
 Az idő 01:29:59 ... egy másodperccel később már 01:30:00
 Az idő 23:59:59 ... egy másodperccel később már 00:00:00
 Az idő 19:12:27 ... egy másodperccel később már 19:12:28

Az adatszerkezetekből álló tömbök a C nyelv igen erős és fontos eszközét jelentik. Ne hagyjon tovább az olvasással, amíg teljes mélységeben meg nem érti.



9.2 ábra

A testTimes tömb elhelyezkedése a memóriában

Adatszerkezeteket tartalmazó adatszerkezetek

A C nyelv rendkívül rugalmas az adatszerkezetek megadása terén. Megadhatóak olyan adatszerkezetek, melyek adattagjai maguk is lehetnek adatszerkezetek, sőt, tömbök is.

Láttuk, hogy miként szervezhetők egy egységbe a logikailag összetartozó adatok. Például az év-hónap számértékeket összefogtuk egy „dátum” jelentésű (`date`) típusba, az óra-perc-másodperc adatait pedig egy „időpont” (`time`) típusba. Lehetnek olyan alkalmazások, melyben e két adattípust szeretnénk egyszerre használni. Például egy eseménynaplárban szükség van az egyes események dátum és időpont adataira.

A következőkben tehát egy olyan helyzetet tételezünk fel, melyben összekapcsoltan szeretnénk használni a dátum és az időpont információkat. Definiálunk ehhez egy `dateAndTime` adatszerkezetet, melynek legyen két adattagja: egy `date` és egy `time` típusú adatszerkezet.

```
struct dateAndTime
{
    struct date     sdate;
    struct time     stime;
};
```

Ennek az adatszerkezetnek az első tagja a `date` adatszerkezetű `sdate`, második tagja pedig a `time` adatszerkezetű `stime`. Ehhez a definícióhoz elengedhetetlen, hogy már korábban definiáljuk a `date` és a `time` adatszerkezetet.

Ettől kezdve `dateAndTime` típusuként is megadhatunk változókat, ahogy az a következő sorban is történik:

```
struct dateAndTime event;
```

Ha az `event` (*esemény*) változó `date` adatszerkezetű tagjára szeretnénk hivatkozni, a jelölésmód az eddig megszokott módon történhet:

```
event.sdate
```

Ezzel meghívható a `dateUpdate` függvény; sőt a visszatérési vissza is írható ugyanebbe a változóba:

```
event.sdate = dateUpdate (event.sdate);
```

Hasonlóképp oldható meg a `dateAndTime` adatszerkezet `stime` adattagjának előremozdítása is:

```
event.stime = timeUpdate (event.stime);
```

Ha ezeken az adattagokon *belüli* adattagokra van szükségünk, akkor egy újabb ponttal (és a szükséges név megadásával) tudunk hivatkozni rájuk:

```
event.sdate.month = 10;
```

Ez az utasítás az event adatszerkezet sdate adattagjának month (*hónap*) adattagját októberre állítja be. Hasonlóképp, a

```
++event.stime.seconds;
```

utasítás az event adatszerkezet stime adattagjának seconds (*másodperc*) adattagját inkrementálja.

Az event változó a várakozásnak megfelelően inicializálható:

```
struct dateAndTime event =
    { { 2, 1, 2004 }, { 3, 30, 0 } };
```

Ez az event változó dátum tagját 2004. február elsejére, az időpont tagját pedig hajnali 3:30:00-ra állítja be.

A tagok nevét természetesen az inicializációkor is használhatjuk:

```
struct dateAndTime event =
    { { .month = 2, .day = 1, .year = 2004 },
        { .hour = 3, .minutes = 30, .seconds = 0 }
    };
```

Létrehozhatunk dateAndTime elemekből álló tömböt is, például a következő deklarációval:

```
struct dateAndTime events[100];
```

Az events tömb száz dateAndTime adatszerkezetű elemet fog tartalmazni. Ezek közül a negyedik dateAndTime elemre az events[3]-mal hivatkozhatunk. Az *i*. dátumot pedig a következő módon növelhetjük meg egy nappal:

```
events[i].sdate = dateUpdate (events[i].sdate);
```

A tömb kezdőelemét így állíthatjuk be a délré:

```
events[0].stime.hour      = 12;
events[0].stime.minutes  = 0;
events[0].stime.seconds  = 0;
```

Tömbököt tartalmazó adatszerkezetek

Amint a címből sejthető, megadhatunk olyan adatszerkezeteket is, melyek tömböket tartalmaznak. Leggyakrabban karaktertömbökkel fordul elő ez a változat a különféle alkalmazásokban. Létrehozhatunk például egy month (*hónap*) adatszerkezetet, melyben az adott hónap napjai számát tartjuk nyilván az egyik adattagban, a másikban pedig a hónap nevét. A következő definíció ilyen struktúrát hoz létre:

```
struct month
{
    int    numberOfDays;
    char   name[3];
};
```

Ebben az adatszerkezetben a `numberOfDays` egész adattag tárolja a napok számát, valamint van benne egy `name` nevű hárombetűs karakterlánc. A megszokott módon hoztunk létre egy `month` adatszerkezetű változót:

```
struct month aMonth;
```

A következő utasítássorozattal inicializálhatjuk frissen létrehozott `aMonth` változónkat:

```
aMonth.numberOfDays = 31;
aMonth.name[0] = 'J';
aMonth.name[1] = 'a';
aMonth.name[2] = 'n';
```

Mindez megoldható rövidebben is:

```
struct month aMonth = { 31, { 'J', 'a', 'n' } };
```

Olyan adatszerkezetet is definiálhatunk, melyben minden a 12 hónap helyet kap:

```
struct month months[12];
```

A 9.7 Lista a `months` tömböt mutatja be. A célja minden össze annyi, hogy kezdőértéket adjon a tömb elemeinek, majd ezeket meg is jeleníti a képernyőn.

A 9.3 ábra segíthet megérteni a programban használt jelölésmódot. A `months` tömb egyes elemeire való hivatkozás nem teljesen magától értetődő.

9.7 Lista • Adatszerkezetek és tömbök bemutatása

```
// Adatszerkezeteket és tömböket bemutató program

#include <stdio.h>

int main (void)
{
    int i;
```

```

struct month
{
    int    numberOfDays;
    char   name[3];
};

const struct month months[12] =
{ { 31, {'J', 'a', 'n'} }, { 28, {'F', 'e', 'b'} },
{ 31, {'M', 'a', 'r'} }, { 30, {'A', 'p', 'r'} },
{ 31, {'M', 'a', 'y'} }, { 30, {'J', 'u', 'n'} },
{ 31, {'J', 'u', 'l'} }, { 31, {'A', 'u', 'g'} },
{ 30, {'S', 'e', 'p'} }, { 31, {'O', 'c', 't'} },
{ 30, {'N', 'o', 'v'} }, { 31, {'D', 'e', 'c'} } };

printf ("Month      Number of Days\n");
printf ("-----\n");

for ( i = 0; i < 12; ++i )
    printf (" %c%c%c      %i\n",
           months[i].name[0], months[i].name[1],
           months[i].name[2], months[i].numberOfDays);

return 0;
}

```

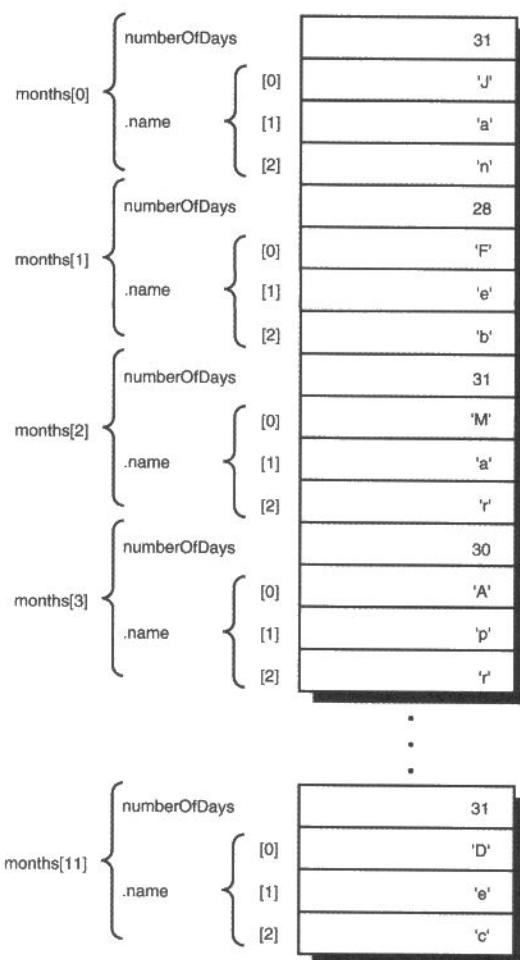
9.7 Lista • Kimenet

Month	Number of Days
Jan	31
Feb	28
Mar	31
Apr	30
May	31
Jun	30
Jul	31
Aug	31
Sep	30
Oct	31
Nov	30
Dec	31

Ahogy a 9.3 ábrán is látható, a

months[0]

jelölésmóddal hivatkozhatunk a (month tömb kezdőelemeként tárolt) *teljes* hónap adatszerkezetre. Ennek az elemnek a típusa: struct month. Így annak a függvénynek, melynek ezt paraméterül adjuk, month adatszerkezetű változót kell várnia formális paraméterként.



9.3 ábra

A *months* tömb

Egyetlen továbblépve:

`months[0].name`

jelöli a `month[0]` hónaphoz nyilvántartott `numberOfDays` month-adattagot. Ennek típusa int. A

`months[0].name`

kifejezés pedig egy name nevű háromelemű karaktertömbre utal, amely szintén az előző tömbelem egyik month adattagja.

Ha ezt adjuk paraméterül egy függvénynek, akkor a vonatkozó formális paramétert char tömbként kell megadni.

Végül, a

```
months[0].name[0]
```

kifejezés a months[0]-ban tárolt month típusú elem name tömbjének kezdőelemét jelenti, azaz a 'J' karaktert.

Variációk adatszerkezetekre

Az adatszerkezetek megadásakor többféleképpen járhatunk el. Lehetőség van például az adatszerkezet definíciójához kapcsoltan azonnal, helyben megadni néhány változó deklarációját is. Ehhez egyszerűen fel kell sorolni a szükséges változóneveket az adatszerkezetet lezáró pontosvessző előtt. Az alábbi kifejezés például definiálja a date adatszerkezetet, és egyben létre is hoz két ilyen típusú változót todaysDate és purchaseDate néven:

```
struct date
{
    int month;
    int day;
    int year;
} todaysDate, purchaseDate;
```

Ugynézzel a lendülettel kezdőértéket is adhatunk a változóknak:

```
struct date
{
    int month;
    int day;
    int year;
} todaysDate = { 1, 11, 2005 };
```

Ez a kód részlet a date definícióján túl (2005. január 11-re) inicializálja is a todaysDate változót.

Ha egy adatszerkezet definíciójakor minden tagváltozó típusa adott, akkor az azonnali deklaráció esetében el is hagyható az adatszerkezetnek történő névadás. Így a

```
struct
{
    int month;
    int day;
    int year;
} dates[100];
```

utasítás létrehoz egy 100 elemet tartalmazó dates nevű tömböt, melynek minden eleme három egész adattagot rejti: month, day és year néven. Mivel itt nem adtunk nevet az adatszerkezetnek, ugyanígy meg kell adni minden adattagot, ha a programban szeretnénk még egy ugyanilyen változót deklarálni.

Láthattuk, miként lehet egyetlen címke alá betömöríteni a logikailag összetartozó adatokat az adatszerkezetek segítségével. Azt is megtanultuk e fejezetben, hogy milyen egyszerűen lehet adatszerkezetekből álló tömböket létrehozni, és ezeket függvényekkel együttes használni. Következő fejezetünkben a karaktertömbök (vagyis a karakterláncok) világában mélyedünk el. Előtte azonban végezzünk el néhány gyakorlatot.

Gyakorlatok

1. Gépeljük be és futtassuk a fejezetben szereplő hét példaprogramot! A kimeneteket hasonlítsuk össze a könyvben szereplő programkimenetekkel.
2. Különféle alkalmazásokban, különösen is a pénzügyi programokban gyakori feladat annak meghatározása, hogy két megadott dátum között hány nap van. Például 2005. július 2. és 2005. július 16. között nyilvánvalóan 14 nap van. Nehezebb azonban kiszámítani például a 2004. augusztus 8. és a 2005. február 22. közötti időszak napjai számát. Szerencsére adott egy képlet a két adott dátum közti napok számának meghatározására. Ehhez egy segédváltozót használunk (N), melyet külön-külön kiszámítunk a két dátumhoz, és a két N különbsége adja meg a két dátum között eltelt napok számát.

N -et a következő módon határozhatjuk meg:

$$N = 1461 \times f(\text{év}, \text{hó}) / 4 + 153 \times g(\text{hó}) / 5 + \text{nap}$$

ahol a két felhasznált függvény:

$$f(\text{év}, \text{hó}) = \begin{cases} \text{év} - 1, & \text{ha a hó} \leq 2 \\ \text{év} & \text{egyébként} \end{cases}$$

$$g(\text{hó}) = \begin{cases} \text{hó} + 13, & \text{ha a hó} \leq 2 \\ \text{hó} + 1 & \text{egyébként} \end{cases}$$

Illusztrációként számítsuk ki a 2004. augusztus 8. és a 2005. február 22. közötti napok számát. Ehhez határozzuk meg a dátumokhoz tartozó N_1 és N_2 értékét a megadott képlettel:

$$\begin{aligned} N_1 &= 1461 \times f(2004, 8) / 4 + 153 \times g(8) / 5 + 3 \\ &= (1461 \times 2004) / 4 + (153 \times 9) / 5 + 3 \\ &= 2927844 / 4 + 1377 / 5 + 3 \\ &= 731961 + 275 + 3 \\ &= 732239 \end{aligned}$$

$$\begin{aligned} N_2 &= 1461 \times f(2005, 2) / 4 + 153 \times g(2) / 5 + \\ &= (1461 \times 2004) / 4 + (153 \times 15) / 5 + 21 \\ &= 2927844 / 4 + 2295 / 5 + 21 \\ &= 731961 + 459 + 21 \\ &= 732441 \end{aligned}$$

$$\begin{aligned}
 \text{Az eltelt napok száma} &= N_2 - N_1 \\
 &= 732441 - 732239 \\
 &= 202
 \end{aligned}$$

Így tehát a két megadott dátum között 202 nap telt el. A megadott képlet 1900. március 1. utáni dátumokra használható. (1800. március 1. és 1900. február 28. között 1 adandó az N értékhez, 1700. március 1. és 1800. február 28. között pedig 2 adandó N -hez.)

Írunk programot, amely lehetővé teszi a felhasználónak, hogy megadjon két dátumot, majd számítsuk ki a két dátum között eltelt napok számát a fenti algoritmus-sal! Próbáljuk logikai egységenként elkülönülő függvényekre bontani a programot. Írunk egy olyan függvényt, amely paraméterként megkap egy `date` adatszerkezetű változót, és visszatérési értékként a fenti képlet eredményét (N) adjja. Ezt a függvényt kétszer kell meghívni a két dátumnak megfelelően, és a visszatérési értékek különbségét kell kiírni – ez jelenti a dátumok közti napok számát.

3. Írunk egy `elapsed_time(eltelt idő)` nevű függvényt, amely két `time` adatszerkezetű változót kap meg paraméterként, és visszatérési értékként a két időpillanat között eltelt időt adja meg, szintén `time` adatszerkezetű változóként.

Ha például `time1 = 3:45:15` és `time2 = 9:44:03`, akkor az `elapsed_time(time1, time2)` függvényhívásnak egy olyan adatszerkezet legyen a visszatérési értéke, melyben az órák száma 5, a percek száma 58 és a másodpercek száma 48. Vigyázzunk az éjfélén átívelő időintervallumokkal!

4. Tekintsük a 2. feladatban vázolt képlettel kiszámolt N értéket. Ha kivonunk belőle 621049-et és az eredmény 7-tel vett osztási maradékát nézzük, akkor megkapjuk, hogy a héten melyik napjára esik az adott dátum. (0 felel meg a vasárnapnak, 1 a hétfőnek ... stb., 6 pedig szombatnak). 2004. augusztus 8-nak például $N=732239$ felel meg, ebből kivonva 621049-et 111190 adódik. Ennek 7-tel vett osztási maradéka: $111190 \% 7 = 2$, azaz keddre esett az adott nap.

A 2. feladatban megírt függvények felhasználásával írunk programot, amely kiszámítja, milyen napra esik egy megadott dátum. A program írja ki szövegesen a hétnapjának a nevét!

5. Írunk egy `clockKeeper(óraFrissenTartó)` nevű függvényt, amelynek paramétere legyen egy (a fejezetben megismert) `dateAndTime` adatszerkezetű változó! A függvény hívja meg a `timeUpdate` függvényt. Amikor az idő eléri az éjfélét, akkor hívja meg a `dateUpdate`-et is, hogy a dátum értéke változzon át a másnapira!

A `clockKeeper` függvény visszatérési értéke is legyen `dateAndTime` típusú.

6. A 9.4 Lista `dateUpdate` függvényét cseréljük le egy olyanra, amely (a fejezetben megismert) összetett betűkonstans-kifejezést használ! Futtassuk is a programot, hogy lássuk, helyesen működik-e.

10

Karakterláncok

Vágunk bele a karakterláncok mélyebb megismerésébe! Volt már szó karakterláncokról a 3. fejezetben is. Első programunk legfontosabb utasítása ez volt:

```
printf ("Programming in C is fun.\n");
```

A printf-nek a következő karakterláncot adtuk át paraméterként:

```
"Programming in C is fun.\n"
```

A dupla idézőjelek mutatják a karakterlánc határát; ezen belül bármilyen betű, szám, (az idézőjeltől eltérő) speciális karakter szerepelhet. Sőt, mint hamarosan látni fogjuk, az idézőjel karakterláncba illesztése is lehetséges.

A char adattípusról azt tanultuk, hogy *egyetlen* karaktert tárolhatunk az ilyen változókban. A megfelelő karakter hozzárendelésekor aposztrófok között kell megadni a hozzárendelt értéket; így a

```
plusSign = '+';
```

értékkadás a + jelet adja át a plusSign változónak (feltéve, hogy ez megfelelően lett deklarálva korábban). Arról is volt már szó, hogy az aposztróf és az idézőjel különböző szerepet tölt be. Ha a plusSign változó char típusúként lett deklarálva, akkor hibás a következő utasítás:

```
plusSign = "+";
```

A C nyelvben az aposztróffal és az idézőjellel eltérő típusú konstanst lehet megadni.

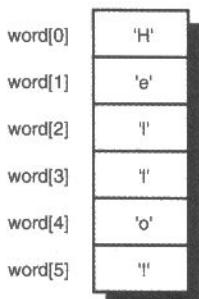
Karaktertömbök

A karaktertömbök használata akkor válik szükségesre, amikor olyan változókkal szeretnénk dolgozni, melyek egynél több karakter tárolására¹ is alkalmasak.

A 7.6 Listában a következő módon definiáltuk a word karaktertömböt:

```
char word [] = { 'H', 'e', 'l', 'l', 'o', '!' };
```

Ha egy tömbnek nem adjuk meg a méretét, akkor a fordítóprogram a kezdőértékek száma alapján állítja be azt. A fenti utasítás hat karakter számára foglal le helyet a memóriában, ahogy azt a 10.1 ábrán is láthatjuk.



10.1. ábra

A karaktertömb helyfoglalása a memoriában.

A word tömböt úgy írtattuk ki, hogy végiglépünk az elemein, és a %c formátumjelző karakterláncjalccal megjelenítettük az egyes karaktereket.

Ezzel az egyszerű módszerrel felvértezve elkezdhetjük összegyűjteni azokat a függvényeket, melyek megkönyítik a karakterláncok kezelését. A leggyakoribb műveletek: két karakterlánc összefűzése (*konkatenációja*), egy karakterlánc bemásolása egy másikba, egy karakterlánc valamely részének kiemelése, valamint annak eldöntése, hogy két karakterlánc megegyezik-e. Nézzük például az elsőt, a karakterlánc-összefűzést. Legyen a függvény neve concat, és definiáljuk az alábbi módon:

```
concat (result, str1, n1, str2, n2);
```

¹ Emlékeztetőül: a wchar_t olyan típus, amely „széles karaktereket” tárol, azaz a nemzetközi karakterkészlet karaktereit. Itt nem erről van szó, hanem karakterek hosszabb sorozatról.

Itt str1 és str2 jelenti a két összeillesztendő karakterláncot, n1 és n2 pedig az egyes karakterláncok hosszát (azaz a tömbök méretét). Ezzel a megközelítéssel kellően rugalmas függvényt kapunk, melynek segítségével két tetszőleges hosszúságú karakterláncot össze tudunk fűzni. A result tömb jelenti az str1 és str2 összefűzéseként kapott karakterláncot. Lássuk mindezt a 10.1 Listában.

10.1 Lista • Karaktertömbök összefűzése

```
// Két karaktertömb összefűzésére szolgáló függvény

#include <stdio.h>

void concat (char result[], const char str1[], int n1,
              const char str2[], int n2)
{
    int i, j;

    // az str1 átvitele a result -ba

    for ( i = 0; i < n1; ++i )
        result[i] = str1[i];

    // az str2 átvitele a result -ba

    for ( j = 0; j < n2; ++j )
        result[n1 + j] = str2[j];
}

int main (void)
{
    void concat (char result[], const char str1[], int n1,
                  const char str2[], int n2);
    const char s1[5] = { 'T', 'e', 's', 't', ' ' };
    const char s2[6] = { 'w', 'o', 'r', 'k', 's', '.' };
    char s3[11];
    int i;

    concat (s3, s1, 5, s2, 6);

    for ( i = 0; i < 11; ++i )
        printf ("%c", s3[i]);

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

10.1 Lista • Kimenet

Test works.

A concat függvény első for ciklusa az str1 tömb karaktereit másolja be a result tömbbe. A ciklus n1-szer fut le – ez az str1 tömb mérete.

A második for ciklus az str2 tömb karaktereit teszi be a result tömbbe. A másolás már nem a tömb legelejéről, hanem az n1. pozíóról kezdődik: a result[n1]-től, ami az str1-ből átvett utolsó karakter utáni hely a result-ban (hiszen az str1 hossza n1). A második for ciklus végeztével a result tömbben az str1+str2 összefűzött karakterlánc áll (összesen n1+n2 karakternyi helyen).

A main eljárásban van két const character tömb, s1 és s2. Az első kezdőértéke a következő elemekből áll: 'T', 'e', 's', 't', '!'. A legutolsó egy szóköz, ami ugyanúgy érvényes karakterkonstans, mint bármely más betű. A második karakterlánc kezdőértéke: 'w', 'o', 'r', 'k', 's', '!'. Egy harmadik karaktertömböt is definiálunk, s3-at, mégpedig úgy, hogy beleférjen s1 és s2 összefűzése, 11 karakter. Ez már nem konstans tömb, hiszen meg fog változni a program futása közben. A

```
concat (s3, s1, 5, s2, 6);
```

függvényhívással összefűzzük s1 és s2 karakterláncait az s3 karakterláncban. Az 5-ös és 6-os értékű paraméter fejezi ki s1 és s2 elemszámát.

A concat függvény lefutása után a vezérlés visszatér a main eljárásba, ahol egy for ciklus kiírja s3 karaktereit. Mind a 11 karakter megjelenik a képernyőn – a kimenetből is látható módon teljesen helyesen. Programpéldánkban feltételeztük, hogy az összefűzés eredményeként adódó tömbben (a concat függvény első paraméterében) elegendő hely áll rendelkezésre a két karakterlánc számára. Ha ezt nem biztosítjuk, a programunk előreláthatatlan következményekhez vezethet.

Változó méretű karakterláncok

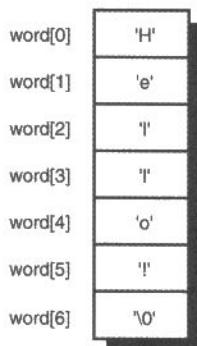
A concat függvényhez hasonlóan más karakterlánc-kezelő függvényeket is létrehozhatunk. Az eljárásoknak megadhatjuk paraméterként a használandó karakterláncok nevét és hosszát. Egy idő után azonban fárasztó állandóan azzal foglalkozni, hogy hány karakterből is áll az átadott paraméter – különösen akkor, ha változó méretű tömbökben tároljuk karakterláncainkat. Arra lenne szükségünk, hogy úgy tudjunk dolgozni a karaktertömbökkel, hogy ne kelljen törődni a benne levő karakterek számával.

Ennek az igénynek úgy tudunk megfelelni, hogy egy speciális karakterrel zárajuk le karakterláncainkat. Így a függvény érzékelni tudja, ha elérte ezt a speciális karaktert – ekkor van vége az adott karakterláncnak. Ha így írjuk meg karakterlánc-kezelő függvényeinket, akkor nem kell bajlódni a karaktertömbök méretének megadásával.

A C nyelvben a *null* karaktert, azaz a '\0' karaktert szokták speciális zárókarakterként használni. Így a

```
const char word [] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
```

utasítás *hét* karakterrel definiálja a *word* nevű tömböt; a karakterlánc végén látható a *null* karakter. (Emlékeztetőül: backslash, vagyis a „\” karakter speciális jelentéssel bír a C nyelvben, így a '\0' csak egyetlen speciális karaktert ad meg.) A *word* tömb memóriabeli elhelyezkedését mutatja a 10.2 ábra.



10.2 ábra

A word tömb a null zárókarakterrel

Annak bemutatására, hogy hogyan is működnek a zárókaraktert figyelő függvények, írunk egy *stringLength* nevű függvényt, amely megszámlálja egy karakterlánc karaktereinek számát (10.2 Lista). Paraméterként egy null karakterrel lezárt karakterláncot vár. A függvény meghatározza a tömb karaktereinek számát, és visszatérési értékül adja a hívó eljárásnak. Tekintsük a tömb karaktereinek számát úgy, hogy nem számít bele a záró null karakter. Ennek megfelelően a

```
char characterString[] = { 'c', 'a', 't', '\0' };
```

karakterláncra meghívott

```
stringLength (characterString)
```

függvény visszatérési értéke 3.

10.2 Lista • Egy karakterlánc hosszának meghatározása

```
// Egy karakterlánc karaktereinek számát meghatározó függvény

#include <stdio.h>

int stringLength (const char string[])
{
    int count = 0;

    while (string[count] != '\0')
        ++count;

    return count;
}

int main (void)
{
    int stringLength (const char string[]);
    const char word1[] = { 'a', 's', 't', 'e', 'r', '\0' };
    const char word2[] = { 'a', 't', '\0' };
    const char word3[] = { 'a', 'w', 'e', '\0' };

    printf ("%i %i %i\n", stringLength (word1),
            stringLength (word2), stringLength (word3));

    return 0;
}
```

10.2 Lista • Kimenet

5 2 3

A `stringLength` függvény konstans tömbként kéri a paraméterét, mivel ez nem fog meg változni a függvény futása során (hiszen csak megszámoljuk az elemszámát). A függvény először 0 kezdőértékre inicializálja a `count` változót, majd elkezd egy `while` ciklust, amely végiglépdel a tömb elemein, egészen a záró karakterig. Amikor ezt elérte, akkor már nem teljesül a ciklusfeltétel; a vezérlés kilép a ciklusból, és visszatérési értékként elküldi a `count` változó értékét. Ebben a változóban van a tömb elemszáma (nem számítva a záró karaktert).

Szeretnénk kipróbálni, hogy jól számol-e a `stringLength` függvény, így a `main` eljárásban létrehozunk három karaktertömböt, `word1`, `word2` és `word3` néven. Mindhárom karakterláncra a lefuttatjuk a `stringLength` függvényt, és az eredményt egy alkalmas `printf` utasítással íratjuk ki.

Karakterláncok inicializálása és kiírása

Ideje visszatérnünk az 10.1 Lista karakterlánc-összefűző programjához. Írjuk át úgy, hogy változó méretű karakterláncokat is feldolgozhassunk vele. A függvényhez biztosan hozzá kell nyúlnunk, hiszen szeretnénk elhagyni a méretre vonatkozó paramétereket. A függvénynek három paramétert szeretnénk csak átadni: kettőt az összefűzendő karakterláncok megnevezésére és a harmadikat az eredményeként adódó karakterlánc számára.

Mielőtt belevetnénk magunkat a programozásba, ismerkedjünk meg a C nyelv két hasznos sajátsgával, melyek révén nagyban leegyszerűsödik a karakterláncok kezelése. Lehetőség van konstans karakterláncnal inicializálni egy karaktertömböt, és nem kell egyesével idézni a karaktereket. Így a

```
char word[] = { "Hello!" };
```

utasítással pontosan azt a hatást érjük el, mintha a 'H', 'e', 'l', 'l', 'o', '!' és '\0' karaktereket adtuk volna kezdetértékül a word tömbnek. Sőt, a kapcsos zárójelek is elhagyhatóak:

```
char word[] = "Hello!";
```

Mindkét utasítás egyenértékű a következővel:

```
char word[] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
```

Ha kimondottan megadjuk a tömb méretét, akkor ügyeljünk rá, hogy legyen hely a zárókarakterként megadandó nullnak is. A

```
char word[7] = { "Hello!" };
```

utasítás hatására a fordítóprogram talál elég helyet a null számára is, ám a

```
char word[6] = { "Hello!" };
```

megadásával a záró null már nem fog beleférni a tömbbe (és nem is fog emiatt visszajelzni a fordítóprogram).

Általában is igaz, hogy a karakterlánc-konstansok automatikusan a null karakterrel fejeződnek be. A

```
printf ("Programming in C is fun.\n");
```

utasításban az újsor karakter után kéretlenül is odakerül a null karakter. A printf függvény ebből tudja meg, hogy meddig terjed a kiírandó karakterek tömbje.

A másik tudnivaló a karakterláncok kiírásával kapcsolatos. A printf-ben használható %s formátumjelző karakterlánc arra utal, hogy a kiírandó karaktertömb null karakterrel zárul. Ha tehát a word egy nullal lezárt karakterlánc, akkor a következő utasítással is kiíratható a képernyőre:

```
printf ("%s\n", word);
```

A %s hatására a printf arra számít, hogy az ehhez tartozó paraméter egy nullal lezárt karakterlánc lesz.

Az itt említett két megoldást megfigyelhetik az (újraírt concat összefűző függvényt megvalósító) 10.3 Lista main eljárásában. Mivel már nem szeretnénk a karakterláncok hosszának megadásával bajlódni, a függvénynek a null karakter felismerésével meg kell határoznia, hogy mikor éri el az egyes karakterláncok végét. Amikor az str1-et bemásoljuk a result tömbbe, figyelnünk kell rá, hogy a zárókaraktert már *ne* tekintsük átmásolandónak. Ettől ugyanis már itt lezártnak minősülne a result karakterlánc. Az str2 átírása után azonban gondoskodnunk kell a zárókarakterről – ezzel tudjuk jelezni az újonnan gyártott karakterlánc végét.

10.3 Lista • Karakterláncok összefűzése

```
#include <stdio.h>

int main (void)
{
    void concat (char result[], const char str1[], const char str2[]);
    const char s1[] = { "Test " };
    const char s2[] = { "works." };
    char s3[20];

    concat (s3, s1, s2);

    printf ("%s\n", s3);

    return 0;
}

// Két karakterlánc összefűzését elvégző függvény

void concat (char result[], const char str1[], const char str2[])
{
    int i, j;

    // str1 átmásolása a result-ba
    for ( i = 0; str1[i] != '\0'; ++i )
        result[i] = str1[i];
```

```

// str2 átmásolása a result-ba
for ( j = 0; str2[j] != '\0'; ++j )
    result[i + j] = str2[j];

// Az összefűzött karakterlánc lezárása a null karakterrel

result[i + j] = '\0';
}

```

10.3 Lista • Kimenet

Test works.

A concat függvény első for ciklusában az str1 karakterlánc átmásolódik a result-ba. A ciklusfeltétel úgy van megfogalmazva, hogy a csak a null karakter eléréséig igaz, ezért maga a null karakter már nem másolódik át a result tömbbe.

A második for ciklusban az str2 karakterlánc átmásolódik a result tömbbe, mégpedig az str1-ből átkerült utolsó karakter utáni pozíciótól kezdve. Ez a ciklus felhasználja az előző ciklusból adódó i értéket, amely az első karakterlánc elemeinek a (nullkaraktermentes) száma. Az alábbi hozzárendelés:

```
result[i + j] = str2[j];
```

oldja meg az str2 átírását a result megfelelő részére.

A második ciklus után a concat függvény egy null karaktert ír az eredményül kapott karakterlánc végére. Nézzük meg alaposan a programot, hogy i és j szerepével teljesen tisztában legyünk. A karakterlánc-kezelő programok hibáinak nagy része abból származik, hogy valamilyen irányban eggyel eltér az indexváltozó a helyes értéktől.

Ne feledjük, hogy a tömbök kezdőelemét a 0. sorszámmal jelöljük. Ha például egy string nevű karakterlánc (a nullkarakter nélkül) n karaktert tartalmaz, akkor a string[n-1] jelenti az utolsó nemnull karaktert, míg a string[n] a null karakter. A string karakterláncot úgy kell definiálni, hogy legalább n+1 karaktert tartalmazzon, hisz a lezáró null karakternek is kell egy hely.

Visszatérve a programra: a main eljárás két char tömböt ad meg, s1-et és s2-t. Kezdőértéket a korábban leírtak szerint állítjuk be. Az s3 tömb húsz karakter tárolására van méretezve. Így biztosan lesz elegendő hely az összefűzendő karakterlánc számára, és nem kell időt vesztegeznünk a pontos számlálgatással.

A concat függvényt e három paraméterrel hívjuk meg: s1, s2 és s3. Bár az eredményül adódó s3 karakterlánc húsz karakternyi helyre lett lefoglalva, a printf függvény csak annyit ír ki, amennyi a null karakter előtt található.

Két karakterlánc egyenlőségének vizsgálata

Két karakterlánc egyenlőségét nem tudjuk ilyen egyszerűen megvizsgálni:

```
if ( string1 == string2 )
    ...
```

Az egyenlőségi operátor ugyanis csak alapvető változótípusoknál használható, mint a float, az int vagy a char. Ám összetett adattípusoknál (mint amilyenek a tömbök vagy az adatszerkezetek) már nem.

Karakterláncok összehasonlításakor egyesével meg kell vizsgálni a karaktereket, hogy megegyeznek-e. Ha egyszerre érjük el a vizsgálandó karakterek végét, és közben is minden karakter azonos volt, akkor a két karakterlánc megegyezik; ellenkező esetben nem azonosak.

Jól jöhet egy olyan függvény, amely összehasonlít két karakterláncot, ahogy azt a 10.4 Lístában is láthatjuk. Az equalStrings függvény két paramétere a két összehasonlítandó karakterlánc. Most csak annyi a célunk, hogy a karakterláncok azonosságát megvizsgáljuk. Így a visszatérési érték lehet egy logikai érték: true (azaz nem nulla), amennyiben igaz az egyenlőség, és false, amennyiben nem. Ennek az is előnye, hogy közvetlenül felhasználható feltételvizsgálatként:

```
if ( equalStrings (string1, string2) )
    ...
```

10.4 Lista • Két karakterlánc azonosságának vizsgálata

```
// Két karakterlánc egyenlőségét vizsgáló függvény
```

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

bool equalStrings (const char s1[], const char s2[])
{
    int i = 0;
    bool areEqual;

    while ( s1[i] == s2[i] &&
            s1[i] != '\0' && s2[i] != '\0' )
        ++i;

    if ( s1[i] == '\0' && s2[i] == '\0' )
        areEqual = true;
    else
        areEqual = false;
```

```
        return areEqual;
    }

int main (void)
{
    bool equalStrings (const char s1[], const char s2[]);
    const char stra[] = "string compare test";
    const char strb[] = "string";

    printf ("%i\n", equalStrings (stra, strb));
    printf ("%i\n", equalStrings (stra, stra));
    printf ("%i\n", equalStrings (strb, "string"));

    return 0;
}
```

10.4 Lista • Kimenet

```
0
1
1
```

Az equalStrings függvény egy while ciklust használ az s1 és s2 karakterláncok végigjárására. A ciklus addig tart, amíg a megfelelő sorszámu karakterek megegyeznek ($s1[i] == s2[i]$), valamint amíg el nem érjük valamelyik karakterlánc végét ($s1[i] != '\0' \&& s2[i] != '\0'$). Az i ciklusváltozót minden cikluslépésben inkrementáljuk.

A while ciklust követő if utasítás azt vizsgálja, hogy egyszerre értük-e el az s1 és az s2 karakterlánc végét. A programbeli utasítás helyett használhatnánk az

```
if ( s1[i] == s2[i] )
    ...
```

utasítást is, annak is ugyanez lenne a hatása. Ha valóban minden karakterlánc végéhez értük, akkor a két karakterlánc megegyezik. Ekkor az areEqual értéke true állapotba kerül, és ez lesz a függvény visszatérési értéke.

A main-ben két különböző karakterláncot inicializálunk, az stra-t és az strb-t.

Az equalStrings első meghívása ezt a két karakterláncot hasonlítja össze. Mivel a karakterláncok eltérnek, helyesen false, azaz 0 választ kapunk a vizsgálatra.

Az equalStrings második meghívása az stra karakterláncot hasonlítja össze önmagával. A program kimenete arról tanúskodik, hogy a vizsgálat true értéket ad, azaz a függvény egyenlőnek találja a „két” karakterláncot.

Az `equalStrings` harmadik meghívása érdekes. Amint a programból látszik, konstans karakterlánc is átadható a karaktertömböt váró függvényeknek. A 11. fejezetben a mutatók kapcsán majd részletesen megvizsgáljuk, hogy ez hogyan is működik.

Az `equalStrings` függvény összehasonlítja az `strb` karakterláncot a „`string`” karakterláncnal, és azt a választ adja, hogy a két karakterlánc megegyezik.

Karakterláncok beolvasása

Az már világos, hogy a karakterláncok kiírásakor mire használható a `%s` formátumjelző szimbólum. De mi a helyzet a felhasználótól bekért karakterláncok beolvasásával? Több könyvtári függvény is rendelkezésre a felhasználói bemenet feldolgozására. A `scanf` függvény a `%s` formátumjelző szimbólummal felhasználható arra, hogy beolvasson egy karakterláncot. A beolvasás az első szóközig, tabulátorig vagy újsor karakterig tart – attól függően, hogy melyik adódik először. Az alábbi kódrészlet beolvassa a felhasználó által begépelt karakterláncot, és eltárolja a `string` tömbben:

```
char string[81];
scanf ("%s", string);
```

Vegyük észre, hogy a korábbi `scanf` utasításuktól eltérően itt nem írtunk & szimbólumot a változó neve előre. A 11. fejezetben magyarázatot kapunk erre is.

Ha a fenti kódrészlet futtatásakor azt gépel ki a felhasználó, hogy

Shawshank

akkor a `scanf` függvény által beolvasott „Shawshank” karakterlánc eltárolódik a `string` tömbben. Ha azonban azt írná ki, hogy

iTunes playlist

akkor csak az első szó, az `iTunes` kerülne be a `string`-be, hiszen az utána álló szóköz lezárná a karakterláncot. Ha még egyszer meghívunk a `scanf` függvényt, akkor a `playlist` kerülne be a `string`-be. A `scanf` függvény ugyanis minden ollen folytatja a karakterláncok beolvasását, ahol abba hagyta.

A `scanf` függvény a beolvasott karakterláncot automatikusan lezárja a null karakterrel. Ennek beolvasása után:

abcdefghijklmnoprstuvwxyz

a teljes kisbetűs (latin) ábécé eltárolódik a `string` karakterlánc első 26 pozícióján, és a `string[26]` null értéket kap.

Legyen s1, s2 és s3 alkalmasan definiált karaktertömb. A

```
scanf ("%s%s%s", s1, s2, s3);
```

végrehajtása a

```
micro computer system
```

bemenettel azt eredményezi, hogy a három tömbbe bekerül (sorban) a három szó. s1 tartalma lesz a „micro”, s2-é a „computer”, s3-é pedig a „system”. Ha pedig csak annyit gépel be a felhasználó, hogy:

```
system expansion
```

akkor az s1-be kerül a „system”, s2-be pedig az „expansion”. Mivel a harmadik %s számára még nem érkezett meg a bemenet, a scanf tovább várakozik erre.

A 10.5 Listában a scanf három karakterláncot vár.

10.5 Lista • Karakterláncok beolvasása scanf-fel.

```
//A scanf %s formátumjelző karakterláncát bemutató program
```

```
#include <stdio.h>

int main (void)
{
    char s1[81], s2[81], s3[81];

    printf ("Adja meg a szöveget:\n");
    scanf ("%s%s%s", s1, s2, s3);

    printf ("\ns1 = %s\ns2 = %s\ns3 = %s\n", s1, s2, s3);

    return 0;
}
```

10.5 Lista • Kimenet

Adja meg a szöveget:

```
system expansion
bus
```

```
s1 = system
s2 = expansion
s3 = bus
```

Programunkban a `scanf` három karakterláncot olvas be: `s1`-et, `s2`-t és `s3`-at. A bevitt szöveg első sora csak két szót tartalmaz. Mivel a `scanf` arra számít, hogy három szót kap (melyek szóközzel, tabulátorral vagy újsorral vannak határolva), vár még egy szó bevitelére. Ezután egy `printf` hívással ellenőrizzük, hogy valóban jól került-e be a három karakterlánc az `s1`, `s2` és `s3` változóba.

Ha 80-nál több karaktert ütünk be az előző program bemeneteként (szóköz, tabulátor és Enter nélkül), akkor a `scanf` túlcordulást okoz az adott karakterláncban. Ez azt okozhatja, hogy a program szabálytalanul ér véget, sőt, ez előre nem látható következményekhez is vezethet. Arra sajnos nincs lehetőség, hogy a `scanf` felismerje egy tömb méretét. Ha a `%s` formátumjelző karakterláncjal accal van megadva, akkor addig fogadja a beírt karaktereket a `scanf`, míg csak egy zárókaraktert nem visz be a felhasználó.

Ha azonban a `%` jel után egy számot is beírunk a formátumjelző karakterláncba, akkor ezzel kifejezhetjük a maximálisan bevhető karakterek számát. Az alábbi módon 80-ra korlátosztatjuk a beírandó karakterek számát:

```
scanf ("%80s%80s%80s", s1, s2, s3);
```

Ha ezzel cseréljük le a 10.5 Lista megfelelő kifejezését, akkor a `scanf` nem fog 80-nál több karaktert beolvasni egyik karakterláncból sem. (Mivel még kell egy karakternyi hely a null számára is, ezért nem `%81s`-et használunk, hanem csak `%80s`-et).

Egykarakternyi bemenet

A szabványos C programkönyvtár tartalmaz megoldásokat a karakterek és karakterláncok terminálból történő írására/olvasására is. A `getchar` függvényel beolvashatunk egy karaktert a terminálból. Ha ezt a függvényt ismételten meghívjuk, akkor több egymást követő karaktert is be tudunk olvasni. Amikor lezártuk a sort egy Enterrel, akkor az újsor karaktert (`'\n'`) küldi el a függvény. Ha például azt gépeljük be, hogy `'abc'`, majd Entert ütünk, akkor a `getchar` függvény első meghívása `'a'`-t ad vissza, a következő `'b'`-t, majd a `'c'` következik, végül egy `'\n'` (újsor) karakter. Ha ötödször is meghívjuk, akkor bemenetre fog várni a program.

Talán meglepi az olvasót, hogy miért hozakodom elő ezzel a függvényel, amikor korábban már volt szó arról, hogy a `scanf` is be tud olvasni egy karaktert a `%c` formátumjelző karakterlánc megadásával. A `scanf` is remekül megfelel a célnak, ám a `getchar` egy jóval közvetlenebb megközelítést jelent: e függvénynek egyetlen célja van (egy karakter beolvasása), és nem kér semmiféle paramétert. A függvény visszatérési értéke a beolvasott karakter, melyet betölthetünk egy változóba, vagy tetszőleges módon felhasználhatjuk a programban. Több szöveges alkalmazásban egy egész sor beolvasása egyszerre történik meg. Ez a sor egy tárolóhelyen („pufferben”) tartózkodik addig, amíg más módon fel nem dolgozzuk. A `scanf`-et nem tudjuk ilyen üzemmódban használni, hiszen a `%s` hatására már az első szóköznél megszakad a bemeneti szöveg beolvasása.

A függvénykönyvtárban található egy `gets` nevű függvény, melynek egyetlen célja egy sornyi szöveg beolvasása. Programozás gyakorlatként vizsgálják meg a 10.6 Listában található `readLine` nevű függvényt, melynek ugyanez a célja. Függvényünk erősen támaszkodik a `getchar` függvényre. Egyetlen paramétere van: az a karaktertömb, melyben a be-menő sort el kell tárolnunk. A függvény az első újsor karakterig olvassa be (és tárolja el) a karaktereket – magát az újsor karaktert már nem.

10.6 Lista • Egy sornyi szöveg beolvasása

```
#include <stdio.h>

int main (void)
{
    int    i;
    char   line[81];
    void   readLine (char  buffer[]);

    for ( i = 0; i < 3; ++i )
    {
        readLine (line);
        printf ("%s\n\n", line);
    }

    return 0;
}

// Terminálablakra írt szöveges sort beolvasó függvény

void   readLine (char  buffer[])
{
    char   character;
    int    i = 0;

    do
    {
        character = getchar ();
        buffer[i] = character;
        ++i;
    }
    while ( character != '\n' );

    buffer[i - 1] = '\0';
}
```

10.6 Lista • Kimenet

```
This is a sample line of text.  
This is a sample line of text.
```

```
abcdefghijklmnopqrstuvwxyz  
abcdefghijklmnopqrstuvwxyz
```

```
runtime library routines  
runtime library routines
```

A `readLine` függvény do ciklusa építi fel a bemenő karakterekből a buffer karaktertömbbeli karakterláncot. A `getchar` függvény által visszaadott karaktereket a tömb soron következő pozícióján helyezzük el. Az újsor karakter elérése jelzi a bemenet végét – ekkor a ciklus kilép. Ekkor (a függvényből való kilépés előtt, a beolvasott újsor karaktert lecserélve) egy null karaktert helyezünk el a karakterlánc végén. Ehhez az indexváltozó értékét eggyel le kell csökkentenünk (`i-1`), hogy a ciklus utolsó lépésében megnövelte i érték helyett az újsor karakter pozíciójába tehessük a karakterlánc-záró null karaktert.

A `main` eljárásban definiálunk egy `line` nevű karaktertömböt, 81 karakter számára. Így „egy sornyi” (azaz a klasszikus terminálablakba beírható 80 db) karakter fér el, és még egy null karakter a karakterlánc végére. Azonban még a soronként 80 vagy annál kevesebb karaktert tartalmazó terminálablakban is megvan a veszélye annak, hogy túlcordulást okozzunk a tömbben azzal, hogy 80-nál több karaktert gépelünk be Enter nélkül. Érdemes kiegészíteni a `readLine` függvényt egy második paraméterrel, mely a puffer méretét tartalmazza. Ezzel a függvény garantálni tudja, hogy a puffer semmiképpen sem fog túlcordulni.

A program egy háromlépéses `for` ciklusban ismételten meghívja a `readLine` függvényt, mindegyikkel beolvasva egy-egy sort a terminálablakból. A beolvasott karakterláncokat ki is íratjuk, ellenőrizve a függvény helyes működését. A harmadik sor kiírása után a program kilép.

Következő kód részletünk (10.7 Lista) azt mutatja be, hogy miként számlálhatjuk össze egy szöveg szavait. Legyen `countWords` annak a függvénynek a neve, amely paraméterként egy karaktertömböt kap, visszatérési értéke pedig az ebben található szavak száma.

Az egyszerűség kedvéért tekintsük szónak azokat a karakterlánc-részeket, melyek pusztán az ábécé betűit tartalmazzák. A függvény be tudja járni a kapott karakterláncot, közben pedig meg tudja keresni az első alfabetikus karaktert, majd el tud lépni a következő nem-alfabetikus karakterig. A közben érintett karaktereket tekintjük egy szónak. Ezután továbblépni a karaktereken, egészen a következő alfabetikus karakterig, ami a következő szó kezdetét jelenti stb.

10.7 Lista • Szavak összeszámítása (soronként)

```
// Függvény, amely megállapítja, hogy egy karakter alfabetikus-e  
  
#include <stdio.h>  
#include <stdbool.h>  
  
bool alphabetic (const char c)  
{  
    if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )  
        return true;  
    else  
        return false;  
}  
  
/* Egy karakterlánc szavait összeszámító függvény */  
  
int countWords (const char string[])  
{  
    int i, wordCount = 0;  
    bool lookingForWord = true, alphabetic (const char c);  
  
    for ( i = 0; string[i] != '\0'; ++i )  
        if ( alphabetic(string[i]) )  
        {  
            if ( lookingForWord )  
            {  
                ++wordCount;  
                lookingForWord = false;  
            }  
        }  
        else  
            lookingForWord = true;  
  
    return wordCount;  
}  
  
int main (void)  
{  
    const char text1[] = "Nos, így haladunk.";  
    const char text2[] = "Majd aztan megint csak... így.;"  
    int countWords (const char string[]);  
  
    printf ("%s - szavak száma = %i\n", text1, countWords (text1));  
    printf ("%s - szavak száma = %i\n", text2, countWords (text2));  
  
    return 0;  
}
```

10.7 Lista • Kimenet

Nos, így haladunk. - szavak száma = 3

Majd aztan megint csak... így. - szavak száma = 5

Az `alphabet ic` függvény igen egyszerű: megvizsgálja a paraméterként kapott karaktert, hogy az kisbetű vagy nagybetű-e. Ha ez teljesül, akkor igaz lesz a visszatérési értéke, egyébként pedig hamis.

A `countWords` függvény működése már korántsem ennyire nyilvánvaló. A karakterlánc karaktereit végigjáró ciklusváltozó neve `i`. A `lookingForWord` logikai értékű jelzőváltozó mutatja meg, hogy épp abban a fázisban van-e a függvény, amikor keresi a következő szó kezdetét. A függvény futásának kezdetén mindenkiéppen ebben a fázisban vagyunk, ezért `true` értékre inicializáljuk a változót. A `wordCount` lokális változó nyilvánvaló célja a karakterláncban található szavak számlálása.

A karakterlánc minden karakterére meghívjuk az `alphabet ic` függvényt, amely eldönti, hogy a karakter az ábécé betűi közül való-e. Ha igen, akkor ellenőrzi a program, hogy `lookingForWord`, azaz szókezdet-keresési állapotban vagyunk-e. Ha igen, akkor e pillanatban új szót találtunk, így ennek dokumentálására megnöveljük eggyel a `wordCount` változót, valamint hamisra állítjuk a `lookingForWord`-öt (jelezve, hogy már nem keresünk szókezdetet).

Ha a vizsgált karakter az ábécé betűi közül való, de a `lookingForWord` értéke hamis (azaz egy szó belsejében vagyunk), akkor egyszerűen engedjük továbblépni a `for` ciklust.

Ha a vizsgált karakter nem alfabetikus, azaz nincs benne az ábécében, akkor ez vagy azt jelenti, hogy vége lett az előző szónak, vagy azt, hogy még nem találtunk új szókezdetet – minden esetben igazra állítjuk a `lookingForWord` jelzőváltozót. (Még akkor is, ha az már előtte is igaz volt.)

Amikor az átadott karakterlánc összes betűjét számba vettük, akkor a függvény visszaadja a hívó eljárásnak a `wordCount` értékét. Ebben van tárolva az adott pillanatig talált szavak száma.

Segítséget jelent a megértésben, ha táblázatos formában végigkövetjük a `countWords` függvény egyes változóinak értékét. A 10.1 táblázat egy ilyen „nyomkövetést” mutat, attól a pillanattól kezdve, amikor először meghívásra kerül a 10.7 Lista `countWords` függvénye. A táblázat első sora mutatja a `wordCount` és `lookingForWord` változók kezdeti értékeit, még a `for` ciklus elkezdése előtti állapotban. A következő sorokban láthatóak a változók értékei a `for` ciklus lépései közben. Így a második sorban a `wordCount` értéke 1-re változik, valamint a `lookingForWord` jelzőváltozó értéke hamisra (0), miután az 'N' betűt fel dolgoztuk. A táblázat utolsó sora mutatja a változók végső értékeit, amikor elértek a ka-

rakterlánc végét. Érdemes időt szánni a táblázat áttanulmányozására, és a countWords függvény logikáját szem előtt tartva megfigyelni a változók értékeit. Ha ezzel megbirkózott az olvasó, akkor már jóval könnyebb lesz megérteni és megjegyezni a szöösszeszámítató algoritmus működési elvét.

10.1 táblázat • A wordCount függvény véghajtása

I	string[i]	wordCount	lookingForWord
		0	true
0	'W'	1	false
1	'e'	1	false
2	'l'	1	false
3	'l'	1	false
4	', '	1	true
5	' '	1	true
6	'h'	2	false
7	'e'	2	false
8	'r'	2	false
9	'e'	2	false
10	' '	2	true
11	'g'	3	false
12	'o'	3	false
13	'e'	3	false
14	's'	3	false
15	'.'	3	true
16	'\0'	3	true

Az üres karakterlánc

Nézzünk most a wordCount függvényhez egy gyakorlati alkalmazást. A readLine függvény segítségével tegyük lehetővé a felhasználónak, hogy több sort is beírhasson. A program számlálja össze a szöveg szavait, majd jelenítse meg az összeget.

Hogy rugalmasabb legyen a program, ne határozzuk meg a beírható sorok számát. Tegyük lehetővé a felhasználó számára, hogy valahogyan jelezni tudja, ha végzett a teljes szöveg bevitelével. Ennek egy kézenfekvő módja az, hogy a szöveg végén két Enter üt. Amikor a readLine függvény szembesül az (önmagában álló) újsor karakterrel, akkor eltárolja az előtte levő „üres karakterláncot” a pufferbe. A program ezt figyeli, és rögtön tudni fogja, hogy nem kell több sornyi szöveget beolvasnia – következhet a szavak kijelzése.

A C nyelvben külön neve van annak a karakterláncnak, amely nem tartalmaz karaktereket – ez az üres karakterlánc. Ha belegondolunk, az üres karakterlánc teljesen jól használható az összes függvényel, melyet fejezetünkben megírtunk. A `stringLength` függvény 0-t ad vissza az üres karakterlánc hosszaként; a `concat` függvény helyesen „`semmit`” illeszt a másik karakterlánchoz. Sőt, a `equalStrings` is helyesen dolgozik az üres karakterlánc-`cal`: csak akkor tekinti egyenlőnek egy másik karakterlánckal, ha az is üres.

Ne felejtük, hogy az üres karakterlánc is tartalmaz egy karaktert: konkrétan a null karaktert.

Esetenként érdemes üresre állítani egy-egy karakterláncot. Ezt két, egymást követő idézőjellel tehetjük meg, A

```
char buffer[100] = " ";
```

deklaráció egy `buffer` nevű tömböt hoz létre, és az üres karakterláncot adja értékül neki. Ne tévesszük össze az `" "` szimbólumot ezzel: `" ",` mert ez utóbbi a szóközt jelenti. (Aki nem hiszi, járjon utána: vizsgáltassa meg őket az `equalStrings` függvényel.)

A 10.8 Lista az előző programban megírt `readLine`, `alphabetic`, és `countWords` függvényeket használja. Helytakarékkosság miatt csak jelezzük, hová kell beírni őket.

10.8 Lista • Szövegrész szavainak összeszámítása

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

***** Ide illesztendő be az alphabetic függvény *****/
***** Ide illesztendő be a readLine függvény *****/
***** Ide illesztendő be a countWords függvény *****/

int main (void)
{
    char text[81];
    int totalWords = 0;
    int countWords (const char string[]);
    void readLine (char buffer[]);
    bool endOfText = false;

    printf ("Gépelje be a szöveget.\n");
    printf ("Amikor kész, üssön 'ENTER'-t. \n\n");

    while ( ! endOfText )
    {
        readLine (text);
        if (alphabetic (text))
            totalWords++;
        if (text[0] == '\n')
            endOfText = true;
    }
    printf ("%d szó volt a szövegben.\n", totalWords);
}
```

```

        if ( text[0] == '\0' )
            endOfText = true;
        else
            totalWords += countWords (text);
    }
    printf ("\nA szövegen %i szó van.\n", totalWords);

    return 0;
}

```

10.8 Lista • Kimenet

Gépelje be a szöveget.

Amikor kész, üssön 'ENTER'-t.

Wendy glanced up at the ceiling where the mound of lasagna loomed like a mottled mountain range. Within seconds, she was crowned with ricotta ringlets and a tomato sauce tiara. Bits of beef formed meaty moles on her forehead. After the second thud, her culinary coronation was complete.

Enter

A szövegen 48 szó van.

Az Enter-rel jelölt sorban történt az Enter (vagy Return) leütése.

Az endOfText jelzőváltozó mutatja, hogy elérők-e a szöveg végét. A while ciklus addig fut, amíg ennek a változónak hamis az értéke. A ciklusmag minden lépésében meghívjuk a readLine függvényt, hogy olvassa be a következő sornyi szöveget. Egy if utasítással megvizsgáljuk, hogy a null karakter van-e a beolvashatt sor első pozíójában (azaz Entert ütött-e a felhasználó). Ha igen, akkor az endOfText jelzőváltozót igazra állítjuk, jelezve, hogy vége a szövegnak.

Ha a pufferben van feldolgozandó szöveg, akkor meghívjuk a countWords függvényt a text paramétertömbbel. A visszatérési értéket hozzáadjuk a szöveg eddigi szavait összeszámító totalWords változóhoz.

A while ciklusból való kilépés után a program – megfelelő szöveges üzenettel – kiírja a totalWords változó értékét.

Első ránézésre úgy tűnik, hogy a program nem sokat segít a felhasználón, hiszen szegénynek be kell gépelnie a teljes szöveget – ez pedig nagyobb munka, mint a szavak összeszámítása. A 16. fejezetben azonban kiderül, hogy ugyanezt a programot felhasználhatjuk egy szöveges állomány szavainak az összeszámítására is.

Azaz egy ilyesfajta programot tényleg haszonnal alkalmazhat például egy kézirat szerkesztője, akinek tudnia kell, hogy egy szöveg hány szóból (esetleg karakterből) áll.

Ez a példa azt feltételezi, hogy a szöveg nem valamiféle bináris formátumban (például Microsoft Word) van elmentve, hanem (UNICODE karakterekről mentes) normál szöveges állományként.

Vezérlőkarakterek

Korábban volt már szó arról, hogy a backslash karakterek (\) sajátos jelentése van, mert ezzel adható meg például az újsor és a null karakter. Ahogy a backslash és az n együtt egy újsor karaktert képez, ugyanígy előállíthatóak más speciális karakterek is. A backslash karakterrel kezdődő jeleket más néven vezérlőkaraktereknek is hívjuk. A 10.2 táblázatban foglaltuk össze a vezérlőkaraktereket.

10.2 táblázat • Vezérlőkarakterek

Vezérlőkarakter	Hatása
\a	Hangjelzés
\b	Visszatörölés (backspace)
\f	Laptovábbítás (form feed)
\n	Újsor
\r	Kocsi vissza (carriage return)
\t	Vízsintes tabulátor
\v	Függőleges tabulátor
\\\	Maga a backslash karakter
\"	Idézőjel
\'	Aposztróf
\?	Kérdőjel
\nnn	nnn értékű oktális (8-as számrendszerbeli) karakter
\unnnnn	16 bites UNICODE karakter
\Unnnnnnnnn	32 bites UNICODE karakter
\xnn	nn értékű hexadecimális karakter

A táblázat első hét sorában szereplő vezérlőkarakterek szinte minden számítógépes rendszer kimeneti eszközén működnek. A \a-val előívhatalt egy hangjelzés, „csengetés”. A következő utasítás hatására egy hallható csengetés kíséretében jelenik meg egy vészjósító üzenet a rendszerleállásra vonatkozóan:

```
printf ("\aA RENDSZER 5 PERC MÚLVA LEÁLL!!\n");
```

A `\b` (visszatörles) hatására a kurzor egy karakternyit visszalép azon a képernyőn, amely ezt a műveletet támogatja. A `\t` hatására adott távolsággal (a tabulátorral) eltolva jelennek meg a kiírandó karakterláncok, például:

```
printf ("%i\t%i\t%i\n", a, b, c);
```

Itt az a kiírása után a kurzor a következő tabulátor-pozícióra ugrik (ami általában nyolc karakterenként szokott lenni), majd a `b` következik, végül (újabb tabulátor után) a `c`. A vízszintes tabulátor igen jól használható különféle adatok táblázatszerű oszlopokba való rendezésére.

Magának a backslash karakternek a kiírásához önmagát kell „levédenie” a védőkarakternek, vagyis a

```
printf ("A \\t a vízszintes tabulátor vezérlőkaraktere.\n");
```

utasítás azt írja ki, hogy

```
A \t a vízszintes tabulátor vezérlőkaraktere
```

Figyeljük meg, hogy a korábban álló `\` értékelődik ki a fenti karakterláncban, nem pedig a `\t`, így nem a tabulátor jelenik meg, hanem a backslash karakter.

Az idézőjel idézéséhez le kell védenünk a (saját jelentéssel rendelkező) idézőjel karaktert:

```
printf ("\\"Szia!\" mondta András.\n");
```

Ennek hatására megjelenik az üzenet:

```
"Szia!" mondta András.
```

Ha egy aposztrófot szeretnénk hozzárendelni egy karakteres változóhoz, akkor azt is le kell védeni az értelmező elől. Ha például `c` egy char típusú változó, akkor a

```
c = '\';
```

utasítással tudunk egy aposztrófot hozzárendelni `c`-hez.

A backslash karaktert követő `? jelenti magát a kérdőjel karaktert. A ? bizonos (ASCII-n kívüli) háromkarakteres jelsorozatok esetén hordoz sajátos jelentést – a részleteket illetően lapozzuk fel az A függeléket („A C nyelv összefoglalása”).`

A 10.2 táblázat utolsó négy sora olyan lehetőségeket mutat be, melyek révén *bármilyen* karakter beilleszthető egy karakterláncba. Az '\nnn' vezérlőkarakterben az nnn helyén egy maximum háromjegyű oktális (nyolcas számrendszerbeli) szám állhat. Az '\xnn' vezérlőkarakterben az nn egy hexadecimális szám. Ezekkel a számokkal megadható a megjelenítendő karakter belső ábrázolási kódja. Ily módon olyan karaktereket is előhívhatunk, amelyekhez nincs billentyű rendelve. A 33 oktális kódú ASCII vezérlőkaraktert például a \33 vagy a \x1b jelsorozatokkal lehet beilleszteni egy karakterláncba.

Az előző bekezdésekben emlegetett '\0'-nak még a vezérlőkarakterek közt is speciális helyzete van. Ennek a karaktereknek a belső ábrázolási kódja 0. Mivel tehát a null karakterek a számszerű értéke nulla, ezt a tényt a programozók gyakran kihasználják különféle feltételvizsgálatokban, különösen a változó méretű karakterláncok esetében. Egy karakterlánc hosszának a kiszámításához például (a 10.2 Lista `stringLength` függvényében található `string[count] != '\0'` feltétel helyett) elegendő a következő kód részlet:

```
while ( string[count] )
    ++count;
```

A `string[count]` értéke ugyanis mindenkor nem nulla, amíg el nem érünk a karakterlánc végéig – ekkor pedig ki kell lépnünk a `while` ciklusból.

Még egyszer szeretném hangsúlyozni, hogy ezek a vezérlőkarakterek (írásmódjuk ellenére) *egyetlen* karaktert jelentenek. A

```
\033\"Hello\"\\n"
```

karakterlánc tehát csak kilenc karakterből áll (nem számítva a záró null karaktert). Az első a '\033', a második az idézőjel, öt karakter a Hello szó, majd még egy idézőjel következik (eddig ez nyolc), végül az újsor karakter. Ez ki is próbálható, ha átadjuk a fenti karakterláncot a `stringLength` függvénynek.

UNICODE karakterek is előállíthatóak a \u vezérlőkaraktert követő négy hexadecimális számjeggyel, vagy a \U vezérlőkaraktert követő nyolc hexadecimális számjeggyel. Ezek a karakterek már egy kiterjesztett karakterkészlet elemei; nem minden karakterükhet nyolc biten ábrázolni. A UNICODE karakterek vezérlőkaraktereivel ilyen kiterjesztett karakterkészlethez tartozó karaktereket tudunk megadni, melyek némelyike csak 16 (vagy 32) biten ábrázolható. További információk az A függelékben („A C nyelv összefoglalása”).

Bővebben a konstans karakterláncokról

Ha egy backslash segítségével levédjük egy (újsorral lezárt) sor végét, akkor a fordítóprogram ezt úgy fogja értelmezni, mintha nem lenne ott sorlezárás. Ez a sorhosszabbító megoldás főleg karakterlánc-konstansok esetén jön jól, amikor a hosszú karakterláncokat a következő sorban tudjuk folytatni. Az előfeldolgozóról szóló 13. fejezetben is szó lesz még erről, amikor egy makró definíciót több soron keresztül szeretnénk írni.

A sorhosszabbító karakter nélkül az alábbi deklaráció fordítási hibát ad:

```
char letters[] =
{ "abcdefghijklmnopqrstuvwxyz"
ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

A megfelelő helyre beírt sorhosszabbító karakterrel azonban több soron keresztül is folytatható egy karakterlánc-konstans:

```
char letters[] =
{ "abcdefghijklmnopqrstuvwxyz\
ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

Fontos, hogy a folytatott sort a legelső karakternél kezdjük, mert egyébként a sorkezdő szóközök (vagy tabulátorok) is bekerülnek a karakterláncba. A fenti utasítás tehát a következő karakterláncra inicializálja a letters tömböt:

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

A hosszú sorok bevitelének egy másik módja a „szomszédos karakterláncok” alkalmazása. A szóköz(ökk)el vagy tabulátor(okk)al, esetleg újsor karakter(ekk)el elválasztott karakterlánc-konstansokat hívjuk szomszédos karakterláncoknak. Ezeket a fordítóprogram automatikusan egymáshoz illeszti. Az alábbi két utasítás tehát szintaktikailag egyenértékű:

```
"one"      "two"      "three"
"onetwothree"
```

A letters tömböt így is feltölthetjük az ábécé betűivel:

```
char letters[] =
{ "abcdefghijklmnopqrstuvwxyz"
"ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

Sőt, ez a módszer magában a `printf` utasításban is használható:

```
printf ("Programming in C is fun\n");
printf ("Programming" " in C is fun\n");
printf ("Programming" " in C" " is fun\n");
```

Mindhárom utasítás *egyetlen* paramétert ad át a `printf`-nek, mivel a függvény második és harmadik meghívásakor a fordítóprogram összekapcsolja a karakterlánc-részleteket.

Karakterláncok, adatszerkezetek és tömbök

A C programozási nyelv alapvető elemeit igen hatékony programozási szerkezetekkel lehet összeilleszteni. A 9. fejezetben („Adatszerkezetek”) már láttuk, hogy minden egyszerű egy adatszerkezetkből álló tömböt létrehozni. A 10.9 Listában tovább mélyítjük az adatszerkezet-tömb fogalmát, miközben a változó méretű karakterláncokat is munkába állítjuk.

Szeretnénk írni egy értelmező szótári programot, amelytől azt várjuk, hogy megtudhassuk belőle egy-egy szó jelentését. A program tegye lehetővé, hogy begépeljük a keresett szót, majd keresse elő és jelenítse meg a körülírását.

Ahogy gondolkodni kezdünk a programon, rögtön látszik annak az igénye, hogy ábrázolni kellene a szavakat és a meghatározásukat. Mivel a szó és annak jelentése logikailag erősen összetartozik, kézenfekvőnek látszik egy alkalmas adatszerkezet használata. Lehet definiálni például egy `entry` (*bejegyzés*) nevű adatszerkezetet, mely a szót (`word`) és a meghatározását (`definition`) tartalmazza:

```
struct entry
{
    char word[15];
    char definition[50];
};
```

Ez az adatszerkezet 14 karakternyi szavakat és 49 karakternyi meghatározásokat tud majd tárolni (ne feledjük az ábrázolandó null karakterek helyigényét). Tekintsük a következő példát, mely a *blob* szót és meghatározását tartalmazza egy `word1` nevű `entry` adatszerkezetben:

```
struct entry word1 = { "blob", "an amorphous mass" };
```

Mivel számos szó meghatározását tárolni szeretnénk, célszerű létrehozni egy tömböt, melynek elemei `entry` adatszerkezetek:

```
struct entry dictionary[100];
```

A példánkban szereplő tömb 100 elemű. Ez távol áll a használható szótárak méretétől, melyben legalább 100 000 bejegyzés szokott lenni. Valós program esetében azonban nyilván más megoldást keresnénk, és a memória helyett a számítógép merevlemezén tárolnánk a szótár anyagát.

Miután megadtuk a szótár szerkezetét, elgondolkozhatunk azon, hogy hogyan is szervezzük meg a szavak tárolását. A legtöbb szótár ábécé-sorrendben tárolja a szavakat; tegyük mi is így – ehhez szoktunk hozzá a kéziszótárok használatakor. Egy programban ennél több okunk is van a szótár rendezett kialakítására, de erre a kérdésre még később visszatérünk.

Milyen is legyen a szavakat kereső program? Célszerű írni egy függvényt, amely rákeres egy adott szóra a szótárban. Ha megtalálja a keresett szót, adjon vissza annak sorszámát a szótári tömbben, ha pedig nem, adjon vissza -1-et. Így nézhet ki a `lookup` függvény meghívása:

```
entry = lookup (dictionary, word, entries);
```

A függvény három paramétere a (szavakat és meghatározásukat tartalmazó) szótárra, a kerest szóra és a szótárban található bejegyzések számára utal. A függvény próbálja megtalálni a keresett `word` szót a `dictionary` szótárban. Ha megtalálja, akkor a bejegyzés sorszámát adja vissza; ha azonban sikertelen a keresés, akkor -1 lesz a visszatérési érték.

A 10.9 Lista `lookup` függvénye a 10.4 Listában megírt `equalStrings` függvényt használja fel annak eldöntésére, hogy a megadott szó egyezik-e az éppen vizsgált címszóval.

10.9 Lista • Keresés egy értelmező szótárban

```
// Szótári keresőprogram - lineáris kereséssel
```

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

struct entry
{
    char    word[15];
    char    definition[50];
};

***** Ide illesztendő be az equalStrings függvény *****

// Egy szótári szót megkereső függvény

int  lookup (const struct entry dictionary[], const char search[],
             const int   entries)
{
    int  i;
```

```

        bool equalStrings (const char s1[], const char s2[]);

        for ( i = 0; i < entries; ++i )
            if ( equalStrings (search, dictionary[i].word) )
                return i;

        return -1;
    }

int main (void)
{
    const struct entry dictionary[100] =
    { { "aardvark", "a burrowing African mammal" },
    { "abyss", "a bottomless pit" },
    { "acumen", "mentally sharp; keen" },
    { "addle", "to become confused" },
    { "aerie", "a high nest" },
    { "affix", "to append; attach" },
    { "agar", "a jelly made from seaweed" },
    { "ahoy", "a nautical call of greeting" },
    { "aigrette", "an ornamental cluster of feathers" },
    { "ajar", "partially opened" } };

    char word[10];
    int entries = 10;
    int entry;
    int lookup (const struct entry dictionary[], const char
search[],
                           const int entries);

    printf ("Adja meg a keresett szót: ");
    scanf ("%14s", word);
    entry = lookup (dictionary, word, entries);

    if ( entry != -1 )
        printf ("%s\n", dictionary[entry].definition);
    else
        printf ("Sajnos nem található a szótárban ilyen szó: %s\n",
word);

    return 0;
}

```

10.9 Lista • Kimenet

Adja meg a keresett szót: **agar**
a jelly made from seaweed

10.9 Lista • Kimenet (újrafuttatás)

Adja meg a keresett szót: **accede**

Sajnos nem található a szótárban ilyen szó: accede

A lookup függvény végignézi a szótár összes bejegyzését. Az equalStrings függvény segítségével minden szótári szót összehasonlít a keresendő szóval. Ha a lookup függvény talál egyezést, akkor a bejegyzés szótábeli sorszámát (*i-t*) visszatérési értékül adja a hívó eljárásnak. A return utasítással a függvény azonnal kilép, annak ellenére, hogy a for ciklus még nem ért véget.

Ha a lookup függvény végignézte az összes bejegyzést, és sehol nem talált egyezést, akkor a ciklusból való kilépés után return -1 utasítás jelzi a hívó eljárásnak a keresés sikertelenségét.

A keresési módszer megjavítása

A lookup függvény keresési módszere igen egyszerű: sorban végignézi a szótári bejegyzéseket, és ha az adott bejegyzés címszava egyezik a keresett szóval (vagy előre a szótár végét), akkor kilép. Kisebb szótárak esetén, mint amilyen a példában is szerepelt, ez bőven elég. Nagyobb szótárak esetén (amelyek több ezer vagy több tízezer bejegyzést is tartalmazhatnak) ez már nem megfelelő, mert igen időigényessé válik a (lineáris) keresés. Az időigényesség nem biztos, hogy órákat jelent, de sokszor a másodperc tört része is gondot okozhat, ha sokszor kell kivájni. A keresőprogramok megitélésének egyik legalapvetőbb szempontja a keresési sebesség. Mivel a keresés igen gyakori feladat a programozói gyakorlatban, ügyes és igen hatékony algoritmusokat dolgoztak ki a megoldására (akkárcsak a rendezések esetében).

A szótár ábécé-sorrendbeli rendezettségét igen jól kihasználhatjuk egy hatékonyabb algoritmus alkalmazásához. Az első nyilvánvaló javítási lehetőség abban az esetben használható ki, amikor a keresett szó nincs a szótárban. A lookup függvényt felkészíthetjük arra, hogy a kereséssel mikor lépi át a szóba jövő szavak határát. Ha például az „active” szót keressük a 10.9 Listában megadott szójegyzékben (előlről hátrafelé haladva), akkor az „acumen” elérésekor már tudhatjuk, hogy a keresett szó nem fog előfordulni a szótár hátralévő szavai között. Az „acumen”-nek ugyanis a névsorban később kell állnia, mint az „active”-nak.

Ez a javítás azonban csak akkor érzékelhető a futási sebességen, ha a keresett szó *nem* található meg a szótárban. Olyan módszert keresünk, ami nem csupán a keresések egy kis hányadát javítja meg, hanem minden keresést. Ennek a módszernek a neve: logaritmikus keresés.

A logaritmikus keresés elve nem bonyolult. Bemutatására képzeljünk el egy játékot, amelyben egy számot kell kitalálni a lehető legkevesebb lépésekben 1 és 99 között. minden kérdés után ki fog derülni, hogy a kitalálandó szám kisebb-e vagy nagyobb a tippelt ér-

téknél (vagy éppen eltáltá a játékos). Néhány próbálkozás után mindenki rájön, hogy az intervallum-felezéses úton juthatunk el leggyorsabban a keresett számhoz. Ha rögtön az első alkalommal 50-et tippelünk, akkor a válasz megérkezése után („kisebb” vagy „nagyobb”) már száz helyett csak 49 lehetőség közül kell találgnunk. Ha a kitalálandó szám „kisebb” a tippeltnél, akkor 1 és 49 között, ha „nagyobb”, akkor 51 és 99 között lehet a helyes megoldás.

Ezt a felezéses módszert használhatjuk tovább a maradék 49 szám esetén is. Ha a kitalálandó szám „nagyobb” a tippeltnél, akkor a következő próbálkozásnál az 51 és 99 közti intervallumot érdemes megfelezni, azaz érdemes a 75-re tippelni. Ez a módszer követendő egészen addig, amíg a lehetséges számok halmaza egyetlen számra fog leszűkülni. A módszer hatékonysága (általános esetekben) jóval meghaladja bármely más keresési módszerét.

Világos tehát az algoritmus elve. Fogalmazzuk most ezt meg szabatosabban is. Egy n elemből (nagyság szerint növekvő sorrendbe rendezett) M tömbben keressük az x értéket.
(A változónevek jelentése: low : alsó, $high$: felső, mid : középső)

A logaritmikus keresés algoritmusa

1. lépés Állítsuk be low értékét 0-ra, $high$ értékét $n-1$ -re.
2. lépés Ha $low > high$, akkor x nem található meg M -ben: kilépés
3. lépés Legyen mid értéke $(low + high) / 2$.
4. lépés Ha $M[mid] < x$, legyen low értéke $mid + 1$, és ugorjunk a 2. lépéshöz
5. lépés Ha $M[mid] > x$, legyen $high$ értéke $mid - 1$, és ugorjunk a 2. lépéshöz
6. lépés $M[mid]$ egyenlő x -szel, megtaláltuk a keresett értéket: kilépés

A 3. lépében leírt (egész aritmetikájú) művelet egy maradékos osztás, azaz ha low értéke 0 és a $high$ értéke 49, akkor a mid 24 lesz.

Kezünkben van tehát a logaritmikus keresés pontos algoritmusa – nincs más hátra, mint hogy ennek megfelelően írjuk át a lookup függvényt. A logaritmikus keresésnek tudnia kell bármely két (szótári) érték egymáshoz való viszonyát, így az equalStrings függvényt érdemes lecserélni egy másikra, amelyik már nem pusztán az egyenlőséget tudja eldönteni, hanem azt is, hogy milyen értelemben „nem egyenlő” két érték. Legyen ez a függvény a compareStrings. Visszatérési értéke legyen -1, ha az első paraméterként megadott érték van lexikografikusan (ábécé-sorrendben) előbb; legyen 0, ha a két érték azonos, és legyen 1 akkor, ha az első paraméter lexikografikusan nagyobb a másodiknál. Ennek megfelelően a

```
compareStrings ("alpha", "altered")
```

függvényhívás -1 visszatérési értékkel ér véget, hiszen az „*alpha*” előbb van a névsorban (például egy szótárban), mint az „*altered*”. A

```
compareStrings ("zioty", "yucca");
```

visszatérési értéke viszont 1 lesz, mert a „*yucca*” lexikografikusan megelőzi a „*zioty*”-t.

A 10.10 Listában láthatjuk a compareStrings függvényt. A lookup már ezt használja föl a logaritmikus kereséshez. A main eljárás ugyanaz, mint az előző példában, hiszen a minden dő ugyanaz.

10.10 Lista • Logaritmikus keresés egy értelmező szótárban

```
// Szótári keresőprogram – logaritmikus kereséssel
```

```
#include <stdio.h>

struct entry
{
    char word[15];
    char definition[50];
};

// Két karakterlánc összehasonlítását elvégző függvény

int compareStrings (const char s1[], const char s2[])
{
    int i = 0, answer;

    while ( s1[i] == s2[i] && s1[i] != '\0' && s2[i] != '\0' )
        ++i;

    if ( s1[i] < s2[i] )
        answer = -1; /* s1 < s2 */
    else if ( s1[i] == s2[i] )
        answer = 0; /* s1 == s2 */
    else
        answer = 1; /* s1 > s2 */

    return answer;
}

// Egy szótári szót megkereső függvény

int lookup (const struct entry dictionary[], const char search[],
            const int entries)
{
    int low = 0;
    int high = entries - 1;
```

```

int mid, result;
int compareStrings (const char s1[], const char s2[]);

while ( low <= high )
{
    mid = (low + high) / 2;
    result = compareStrings (dictionary[mid].word, search);

    if ( result == -1 )
        low = mid + 1;
    else if ( result == 1 )
        high = mid - 1;
    else
        return mid; /* megvan */
}

return -1; /* nincs meg */
}

int main (void)
{
    const struct entry dictionary[100] =
    { { "aardvark", "a burrowing African mammal" },
      { "abyss", "a bottomless pit" },
      { "acumen", "mentally sharp; keen" },
      { "addle", "to become confused" },
      { "aerie", "a high nest" },
      { "affix", "to append; attach" },
      { "agar", "a jelly made from seaweed" },
      { "ahoy", "a nautical call of greeting" },
      { "aigrette", "an ornamental cluster of feathers" },
      { "ajar", "partially opened" } };

    int entries = 10;
    char word[15];
    int entry;
    int lookup (const struct entry dictionary[], const char
search[],
                const int entries);

    printf ("Adja meg a keresett szót: ");
    scanf ("%14s", word);
    entry = lookup (dictionary, word, entries);

    if ( entry != -1 )
        printf ("%s\n", dictionary[entry].definition);
    else
        printf ("Sajnos nem található a szótárban ilyen szó: %s\n",
               word);
    return 0;
}

```

10.10 Lista • Kimenet

Adja meg a keresett szót: **aigrette**
an ornamental cluster of feathers

10.10 Lista Kimenet (Újrafuttatás)

Adja meg a keresett szót: **acerb**
Sajnos nem található a szótárban ilyen szó: acerb

A `compareStrings` függvény a `while` ciklus végéig ugyanazt teszi, mint az `equalStrings` függvény. A `while` ciklus végén a függvény megvizsgálja azt a karaktert, amely a ciklus lezárulását okozta. Ha `s1[i]` kisebb `s2[i]`-nél, akkor `s1` lexikografikusan kisebb `s2`-nél; ilyenkor -1 a visszatérési érték. Ha `s1[i]` megegyezik `s2[i]`-vel, akkor a két karakterlánc egyenlő; a visszatérési érték 0. Ha ezek egyike sem teljesül, akkor `s1` lexikografikusan nagyobb `s2`-nél, így 1 visszatérési értékkel lép ki a függvény.

A `lookup` függvény egészként definiálja a `low` és a `high` változókat, és a fent vázolt algoritmusnak megfelelő kezdőértéket ad nekik. A `while` ciklus addig fut, amíg `low` nagyobb nem lesz `high`-nál. A ciklusmagban kiszámítjuk `mid` értékét, ami `low` és `high` (egész aritmetikával nyert) számtani közepe. Ezzel a `mid` indexsel mutatunk rá a szótár kiszemelt bejegyzésének szavára a `compareStrings` függvény meghívásakor: `compareStrings(dictionary[mid].word, search)`. A visszatérési érték a `result` változóba kerül.

Ha ez a visszatérési érték -1, akkor a `dictionary[mid].word` lexikografikus értéke kisebb, mint a keresett szóé (`search`). Ilyenkor `low` értékét `mid + 1`-re állítjuk.

Ha a `compareStrings` visszatérési értéke 1, akkor a `dictionary[mid].word` lexikografikusan nagyobb a keresett szónál. Ilyenkor `high` új értéke `mid - 1` lesz.

Ha a fenti két eset egyike sem teljesül, akkor éppen rátaláltunk a keresett szóra, mert a két karakterlánc egyenlő. A `lookup` függvény visszaadja `mid` értékét, ami a keresett szóhoz tartozó bejegyzés indexe.

Ha `low` nagyobbra nő `high`-nál, akkor a keresett szó nincs a szótárban. Ebben az esetben a `lookup` függvény -1 jelzéssel tér vissza, mutatva a keresés sikertelenségét.

Műveletek karakterekkel

Karakteres változók és konstansok gyakran kerülnek elő relációs és aritmetikai kifejezésekben. Alaposan meg kell értenünk, hogy a fordítóprogram hogyan kezeli a karaktereket, mert egyébként nem fogjuk tudni megfelelően kihasználni a karakterek nyújtotta lehetőségeket.

Amikor egy karakteres változó vagy konstans előkerül egy C kifejezésben, azonnal egész számmá alakul, és a későbbiekben is az marad.

A 6. fejezetben már láthattuk, hogy a

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

kifejezéssel el tudtuk dönteni, hogy az adott c karakter kisbetű-e. Azt is említettük, hogy az efféle vizsgálatot csak olyan számítógépes rendszeren lehet használni, ahol ASCII alapú karakterábrázolás van érvényben, vagyis ahol a kisbetűk egymás után, megszakítás nélkül sorakoznak. A fenti kifejezés első részében a c változót 'a'-val hasonlítjuk össze: valójában itt számok összehasonlítása történik. A c változót az 'a'-nak megfelelő ASCII szám-kóddal vetjük össze. Az ASCII-ban az 'a' értéke 97, a 'b' értéke 98 stb.

Így a `c >= 'a'` kifejezés minden kisbetűs c esetén true (azaz nem nulla, igaz) értéket ad. Mivel vannak más karakterek is, melyek ASCII kódja nagyobb 97-nél, kénytelenek vagyunk egy másik feltétellel is közrefogni a kisbetűsnek szánt karakter kódját. Emiatt c értékét a 'z' karakter ASCII kódjával (122-val) is összevetjük.

A karakterek összehasonlítása tehát számértékek összehasonlítását jelenti. A fenti relációs kifejezés teljesen egyenértékű a következővel:

```
c >= 97 && c <= 122
```

Ez is éppúgy csak a kisbetűs karaktert tartalmazó c-re teljesül, mint a fenti kifejezés. Érde-mesebb azonban inkább az előző változatot használni, mert egyrészt annak megértéséhez nem kell tudni a karakterek belső ASCII kódját, másrészt pedig abból sokkal világosabb a programozó szándéka, mit az utóbbi kifejezésből. A

```
printf ("%i\n", c);
```

függvényhívással kiírhatató a c változóban tárolt karakter ASCII kódja. Ha ASCII rendszerű karakterkészlettel dolgozunk, akkor a

```
printf ("%i\n", 'a');
```

utasítás 97-et ad eredményül.

Próbáljuk megjósolni, mi lesz a következő kódrészlet hatása:

```
c = 'a' + 1;
printf ("%c\n", c);
```

Mivel az 'a' értéke (ASCII rendszerben) 97, az első sor 98-at rendel a c változóhoz. Mivel ez a 'b' ASCII kódjának felel meg, a második sor printf utasítása egy 'b' betűt ír ki.

Első látásra nem tűnik túl gyakorlatias megoldásnak, hogy egy karakteres változóhoz hozzáadjunk egy számot. Az alábbi módszer azonban egyszerű módon lehetővé teszi, hogy a '0'...'9' karakterekhez hozzárendeljük a nekik megfelelő 0...9 számértékeket. A '0'-hoz tartozó ASCII kód ugyanis nem 0, az '1'-é nem az 1 stb. Ehelyett a '0' numerikus kódja 48, amiről egy alkalmas printf utasítással meg is győződhetünk:

```
printf ("%i\n", '0');
```

Tegyük fel, hogy a c változóban a '0'...'9' karakterek valamelyike van, és ki szeretnénk számítani a számjegynek megfelelő számértéket. Mivel minden karakterkészletben egy-mást követő pozíciókban vannak a számjegyek, könnyen kiszámítható a számjegynek megfelelő számérték azáltal, hogy az adott számjegy kódjából kivonjuk a '0' karakterkonstans kódját. Ha i egy egész típusú változó, akkor az

```
i = c - '0';
```

kifejezés a c-ben tárolt számjegy-karakternek megfelelő számértéket tölti be i-be. Tegyük fel, hogy c tartalma az '5' karakter, melynek ASCII kódja 53. Ha ebből kivonjuk a '0' karakterkonstans kódját (48-at), 5-öt kapunk. A várakozásnak megfelelően ezt rendeljük hozzá i-hez. Ha ugyanezt a műveletet egy másik kódkészletet használó gépen futtatjuk le (amelyiken a '0' és az '5' másként van elködölve), nagy valószínűséggel akkor is helyes eredményt kapunk.

A módszer kiterjeszhető többjegyű számok karakterláncból számértékké történő átalakítására is. A 10.11 Listában az strToInt függvény a paraméterként (karakterláncban) megkapott „számot” alakítja át tényleges számértékké. A függvény az első nem-numerikus karakterig olvassa be a bemenetet, majd visszatérési értékként a kiszámított számértéket adja. Feltesszük, hogy a visszatérési értéket fogadó int változó kellően nagy méretű a kapott szám tárolásához.

10.11 Lista • Numerikus karakterlánc átszámítása számértékké

```
// Karakterláncot számmá alakító függvény

#include <stdio.h>

int strToInt (const char string[])
{
    int i, intValue, result = 0;

    for (i = 0; string[i] >= '0' && string[i] <= '9'; ++i)
```

```

        intValue = string[i] - '0';
        result = result * 10 + intValue;
    }

    return result;
}

int main (void)
{
    int strToInt (const char string[]);

    printf ("%i\n", strToInt("245"));
    printf ("%i\n", strToInt("100") + 25);
    printf ("%i\n", strToInt("13x5"));

    return 0;
}

```

10.11 Lista • Kimenet

```

245
125
13

```

A `for` ciklus addig fut, amíg a `string[i]` értéke számjegy-karakter. A ciklusmag lépései során a `string[i]` értéke számmá alakul, majd hozzáadódik az előző `result` érték tízszereséhez (azaz a szám végére kerül új számjegyként). Ez lesz az új `result` érték. Kövessük nyomon a függvény működését abban az esetben, amikor „245”-öt adunk át paraméterként.

A ciklus első lefutásakor az `intValue` értéke `string[0] - '0'` lesz. Mivel a `string[0]` értéke '2', a 2 számérték rendelődik hozzá az `intValue` változóhoz. A ciklus első lefutásakor a `result` értéke 0, ennek tízszerese is 0. Ehhez adódik hozzá az `intValue`. Az eredmény visszaíródik a `result`-ba. A ciklusmag első lefutása után tehát a `result` értéke 2.

A második cikluslépésben az `intValue`-ba '4'-'0', azaz 4 kerül. A `result` értékenek tízszerese 20 lesz, ehhez adjuk hozzá az `intValue`-t, ami így 24-et ad. Az eredményt újra beírjuk a `result`-ba.

A harmadik cikluslépés során az `intValue`-ba '5'-'0' = 5 kerül, ami a `result` (24) tízszereséhez (240) adódik hozzá. Így a `result` értéke 245 lesz, amikor harmadszor ér véget a ciklusmag lefutása.

A null karakterbe ütközve a `for` ciklus kilép, a `result` értéke (245) pedig átadódik a hívó eljárásnak.

Az `strToInt` függvényt kétféle módon lehetne még javítani. Egyrészt a negatív számok kezelésére is rá lehetne bírni, másrészt pedig visszajelzést kellene adni arról, hogy talált-e egyáltalán érvényes számjegyet a kapott karakterláncban. Az `strToInt ("xxx")` ugyanis ugyanúgy 0-t ad eredményül, mintha a "0"-t adtuk volna át paraméterként. Ezek a javítási lehetőségek azonban megmaradnak gyakorlási lehetőségeként.

Ezzel véget ért karakterláncokról szóló fejezetünk. Ahogy láthattuk, a C nyelv számtalan lehetőséget biztosít arra, hogy a karakterláncokat hatékonyan és könnyen lehessen kezelni. A szabványos könyvtárban is számos karakterlánc-kezelő függvény található. Kiszámíthatjuk egy karakterlánc hosszát az `strlen` függvénytel, összehasonlíthatunk két karakterláncot az `strcmp`-vel, összefűzhetünk két karakterláncot az `strcat`-tel, átmásolhatunk egy karakterláncot egy másikba az `strcpy` függvénytel. Egy számjegyeket tartalmazó karakterláncot számmá alakíthatunk az `atoi`-vel, valamint rendelkezésünkre áll az `isupper`, az `islower`, az `isalpha` és az `isdigit` függvény, amivel megvizsgálhatjuk, hogy egy karakter nagybetűs-e, kisbetűs-e, alfabetikus vagy szám-e. Jó gyakorlási lehetőséget jelent fejezetünk példaprogramjainak újraírása ezen függvények használatával. A B függeléken („A szabványos C programkönyvtár”) számos hasznos karakterlánc-kezelő könyvtári függvényt megtalálunk.

Gyakorlatok

1. Gépeljük be és futtassuk a fejezetben szereplő tizenegy példaprogramot! A kimeneteket hasonlítsuk össze a könyvben szereplő programkimenetekkel.
2. A 10.4 Listában szereplő `while` ciklusfeltételt lecserélhetjük az alábbi utasításra:
`while (s1[i] == s2[i] && s1[i] != '\0')`
3. A 10.7 és a 10.8 Lista `countWords` függvénye az aposztrófot tartalmazó szavakat két szónak számolja. Módosítsuk a programot, hogy ez ne így történjen. Arra is készítsük fel a függvényt, hogy a pozitív és negatív számokat is szavaknak tekintse (akkor is, ha tizedesvesszőt vagy pontot is tartalmaznak).
4. Írjuk meg a `substring` függvényt, amely kiemeli egy karakterlánc meghatározott részét! A következő módon működjön a függvény:

```
substring (forras, kezdet, hossz, eredmeny);
```

A `forras` paraméter jelentse azt a karakterláncot, amelyből kiemeljük a keresett részt. A `kezdet` jelentse a `forras` azon pozícióját (vagyis a tömb azon indexét), ahonnan a kimásolás kezdődhet. A `hossz` legyen a kiválasztott karakterlánc-rész hossza (karakterekben), és az `eredmeny` legyen az eredményül kapott karakterlánc tömbje. A

```
substring ("karakter", 4, 3, eredmeny);
```

utasítás például a 'kte' karakterlánc-részt tegye be az `eredmeny`-be. Azt a 3-karakteres karakterláncot, amely a 4. pozíciótól (vagyis az 5. betűtől) kezdődik.

A függvény írja be a null karaktert az eredményül kapott karakterlánc végére. El- lenőrizze a függvény, hogy a hivatkozott sorszámú karakterek tényleg léteznek a megadott karakterláncban. Ha nincs is meg minden szükséges karakter, legalább annyit emeljen ki a függvény a forrásból, amennyit csak tud (a karakterlánc végéig). Azaz a

```
substring ("hat alma", 4, 20, eredmeny);
```

hívás csak az „alma” szót tegye be az eredmeny-be (annak ellenére, hogy 20 karaktert kér volna a függvény meghívója).

- Írjuk meg a `findString` függvényt, amely eldönti, hogy egy adott karakterlánc része-e egy másik karakterláncnak! A függvény első paramétere legyen a párosztázan-dó karakterlánc („szénakazal”), a második paraméter pedig a keresendő szövegrész („tű”). Ha a függvény megtalálja a keresett részletet, legyen a visszatérési érték a keresett rész pozíciója (a „szénakazalban”). Ha nem találja a „tűt”, legyen a vissza-térési értéke -1. Az

```
index = findString ("hatalmas", "alma");
```

utasítás az „alma” szót keresi a „hatalmas” szóban. Ennek a keresésnek sikерrel kell zárulnia, visszatérési értékül pedig 3-at kell adnia. A 3. tömb pozícióján (azaz a 4. betűnél) kezdődik ugyanis az „alma” részlet a „hatalmas” szóban.

- Írjuk meg a `removeString` függvényt, amely egy adott részt eltávolít egy karakter-láncból! Legyen három paramétere: az első legyen a forrás karakterlánc, amelyből kivágjuk a kívánt részt. A második legyen a forrás azon pozíciója (vagyis a tömb azon indexe), ahonnan a kivágás kezdődhet. A harmadik pedig legyen a kivága-n-dó karakterlánc-rész hossza (karakterekben). Ha például a `text` tartalma „pehely-paplan”, akkor a

```
removeString (text, 4, 7);
```

hatására ki kell nyesni a karakterláncból a 4. pozíciótól kezdve 7 betűt („elypapl”), így eredményül csak a „pehan” maradjon a `text` változóban.

- Írjuk meg az `insertString` függvényt, amely egy adott karakterláncot beilleszt egy másik karakterláncba! A függvény első paramétere legyen a bővíteni karak-terlánc, a második legyen a beszúrandó szövegrész, a harmadik pedig a beszúrás helyének pozíciója. Ha a `text` tartalma „mutat”, akkor a

```
insertString (text, "lat", 2);
```

utasítás hatására a `text` tartalma már „mulattat” legyen, azaz a 2. pozíciótól (3. betűtől) kezdve illesszük be a „lat” szót a „mutat” szóba úgy, hogy a „tat” részt toljuk hátrább.

- Az előző gyakorlatokban megírt `findString`, `removeString` és `insertString` függvények segítségével írunk egy `replaceString` nevű függvényt, amely kicse-réli egy karakterlánc adott részét egy másikra. Legyen három paramétere:

```
replaceString (forras, s1, s2);
```

A `forras` paraméter legyen a megváltoztatandó karakterlánc, és `s1` illetve `s2` a ke-resejt és a kicsérélendő karakterlánc-rész. A függvény a `findString` segítségével keresse meg az `s1` karakterlánc helyét a `forras`-ban, majd a `removeString`-gel tö-

rölje ki ezt, végül az `insertString` segítségével szűrja be az `s2-t` a megfelelő helyre. Így a

```
    replaceString (text, "1", "egy");
```

függvényhívás cserélje ki a `text`-beli szövegben levő „1” első előfordulását (ha van) arra, hogy „egy”. Ugyanígy, a

```
    replaceString (text, "*", "");
```

utasításnak ki kell nyesnie a `text`-ből a csillag karaktert (azaz a semmivel, a null karakterláncnal kell helyettesítenie).

9. Tovább javíthatjuk az előző gyakorlat `replaceString` függvényét azáltal, hogy visszatérési értékül adjuk a csere sikerességét mutató logikai értéket. Adjon vissza a függvény `true` értéket akkor, ha sikerült megtalálni a kicsérélendő karakterláncot, és `false` értéket, ha nem. Ily módon a `text` összes szóközét ki tudjuk vágni a következő kód részlettel:

```
do
    stillFound = replaceString (text, " ", "");
    while ( stillFound = true );
```

Változtassuk meg a `replaceString` függvényt a fentieknek megfelelően, és próbáljuk is ki különféle karakterláncokkal, hogy meggyőződjünk munkánk helyességéről!

10. Írunk egy `dictionarySort` nevű függvényt, amely a (10.9 és 10.10 Listában használt adatszerkezetű) szótár szavait lexikografikus sorrendbe rendezi!

11. A 10.11 Lista `strToInt` függvényét egészítsük ki úgy, hogy negatív számokkal is tudjon bánni! Ha az átadott karakterlánc első karaktere „-” jel, akkor a visszaadott érték legyen a megfelelő negatív szám.

12. Írunk egy `strToFloat` nevű függvényt, amely lebegőpontos számmá alakít egy megfelelően formázott karakterláncot! A függvény fogadjon el negatív előjelet is. Például az

```
strToFloat ("−867.6921");
```

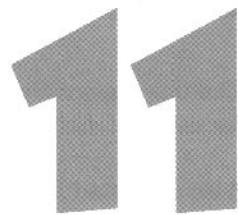
függvényhívás visszatérési értéke legyen `-867.6921`.

13. Ha `c` egy kisbetűs karaktert tartalmazó változó, akkor a

```
c = 'a' + 'A'
```

kifejezés `c` nagybetűs megfelelőjét adja (ASCII típusú karakterkészletet feltételezve). Írjuk meg az uppercase függvényt, amely egy karakterlánc kisbetűs karaktereit a megfelelő nagybetűvé alakítja!

14. Írjuk meg az `intToStr` függvényt, amely a paraméterként kapott egész számot karakterláncá alakítja! A függvénynek a negatív számokat is helyesen kell kezelnie.



Mutatók

Ebben a fejezetben a C programozási nyelv egyik legkifinomultabb eszközével, a mutatókkal ismerkedhet meg az olvasó. A mutatók használatából eredő hatékonyság és rugalmasság emeli ki a C nyelvet sok más programozási nyelv közül. A mutatók teszik lehetővé az összetett adatszerkezetek megvalósítását, a paraméterként átadott változók függvényből történő megváltoztatását és a dinamikus memóriakezelést (részletek a 17. fejezetben: „Speciális lehetőségek a C nyelvben”). A mutatóknak köszönhető a tömbök tömör és hatékony kezelése.

A mutatók működésének megértéséhez ismernünk kell a közvetett elérés fogalmát (*indirection*). A hétköznapi életben számos példát találunk erre. Tegyük fel, hogy munkahelyünkön kifogy a nyomtatóból a tinta. Az efféle vásárlásokat a Beruházási osztály intézi. Fel kell tehát hívni Bélát a Beruházási osztályon, hogy szerezzen be megfelelő nyomtatópatront. Béla fel is hívja a helyi informatikai üzletközpontot, hogy rendeljen az igényelt nyomtatópatronból. Esetünkben közvetett módon tudunk hozzájutni a nyomtatópatronhoz: ahelyett, hogy magunk vásárolnánk meg az üzletből, valaki mással vásároltatjuk meg.

A C nyelv mutatói hasonló elven működnek: közvetett módon teszik lehetővé a hozzáérést egy-egy adatalemhez. Jó oka van annak, hogy miért csak a Beruházási osztályon keresztül juthat a dolgozó nyomtatópatronhoz (például mert nem biztos, hogy tudja, hogy honnan érdemes/lehet megfelelő nyomtatópatront kapni). Hasonlóképp megvan annak is az oka, hogy időnként miért érdemes mutatókkal dolgozni a C programokban.

Mutató megadása

Elég a sok beszédből – lássuk, hogy hogyan is működnek a mutatók. Tegyük fel, hogy van egy `count` nevű egész változónk:

```
int count = 10;
```

Definiálhatunk egy másik változót is `int_pointer` néven, mellyel indirekt módon érhető majd el a `count`:

```
int *int_pointer;
```

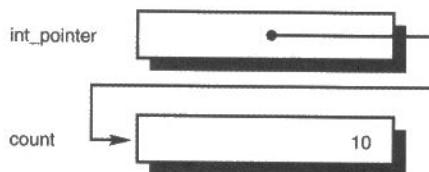
A csillag jellel tudatjuk a fordítóprogrammal, hogy az `int_pointer` egy `int` típusú változóra mutat. Ez annyit jelent, hogy az `int_pointer`-rel tudunk közvetett módon elérni egy (vagy több) egész számértéket.

Az eddigi programjainkban is sokszor használt `scanf` függvényben többnyire & jelet használtunk a változónév előtt. Ezzel a „címe” („*address of*”) egyoperandusú operátorral hozhatunk létre mutatót valamely C objektumra. Ha `x` egy adott típusú változó, akkor `&x` egy mutató erre a változóra. A `&x` értékét átadhatjuk minden olyan mutatónak, ami ugyanolyan típusú változó mutatójaként lett definiálva, mint amilyen az `x`.

A fent megadott `count` és `int_pointer` változókkal tehát érvényes az alábbi művelet:

```
int_pointer = &count;
```

Közvetett hivatkozást létesítünk az `int_pointer` és a `count` között. A „címe” operátor nem a `count` változó értékét, hanem a memóriabeli *címét* rendeli hozzá az `int_pointer`-hez, ami ezzel egy érvényes *mutató* szerepbe kerül. Az `int_pointer` és a `count` közti kapcsolatot a 11.1 ábra mutatja. A nyíl szimbolizálja azt, hogy az `int_pointer` nem a `count` változó értékét tartalmazza, hanem csak rámutat a `count` változó memóriacímére.



11.1 ábra

Egy egész változó mutatója

Ha az `int_pointer` változón keresztül szeretnénk elérni a `count` tartalmát, akkor a közvetett elérés (*indirection*) operátort, vagyis a `*` jelet kell használnunk. Ha tehát `x` egy `int` típusú változó, akkor az

```
x = *int_pointer;
```

utasítás azt az értéket rendeli `x`-hez, amelyre (közvetett módon) az `int_pointer` mutat.

Mivel korábban az `int_pointer`-t úgy állítottuk be, hogy mutasson a `count`-ra, ennek az utasításnak a hatására a `count` változó értéke (vagyis 10) töltődik be `x`-be.

A fenti utasítások a 11.1 Listában is megtalálhatóak. A két alapvető mutató-operátor, a „cím” (&) és a „közvetett elérés” (*) operátor is megfigyelhető a programban.

11.1 Lista • A mutatók illusztrálása

```
// A mutatókat illusztráló program

#include <stdio.h>

int main (void)
{
    int    count = 10, x;
    int    *int_pointer;

    int_pointer = &count;
    x = *int_pointer;

    printf ("count = %i, x = %i\n", count, x);

    return 0;
}
```

11.1 Lista • Kimenet

```
count = 10, x = 10
```

A `count` és az `x` változók normál egész típusúként lettek létrehozva. A következő sorbeli `int_pointer` típusa azonban már „egészre mutató”. A két sort egybe is olvaszthatjuk, a következő módon:

```
int count = 10, x, *int_pointer;
```

Majd a „cím” operátort alkalmazzuk a `count` változóra – ennek hatására létrejön egy mutató érték, mely erre a változóra hivatkozik. Ezt rendeljük hozzá az `int_pointer`-hez.

A program az

```
x = *int_pointer;
```

sorral folytatódik. A közvetett elérés operátorával (*) jelezzük a fordítóprogram számára, hogy ezt a változót most mutatóként szeretnénk használni (olyan típusú adatra vonatkozóan, amilyet a mutató deklarációjakor megadtunk). A fenti hozzárendelő utasítással ki-nyerjük a memóriából az `int_pointer` által mutatott számadatot, és átadjuk az `x` változó-

nak. Mivel a program elején azt közöltük a fordítóprogrammal, hogy az `int_pointer` egész számra fog mutatni, világos, hogy az `*int_pointer` által hivatkozott adat egy egész szám. Az előző sorban a `count`-ra mutatóan állítottuk be az `int_pointer-t`, így ezzel a közvetett elérési művelettel a `count` értékét érjük el.

A 11.1 Lista egy meglehetősen mesterkélt programocska, és nem sok köze van a mutatók gyakorlati használatához. Meg kell még ismerkednünk néhány mutató-megadási és -kezelési módszerrel, hogy megnézhessünk egy valósághoz közelebb eső példát.

A 11.2 Lista rávilágít a mutatók érdekes tulajdonságaira. Itt egy karakteres változóra hivatkozó mutató kerül elő.

11.2 Lista • A mutatók alapjai

```
// További illusztrációk a mutatókkal kapcsolatban

#include <stdio.h>

int main (void)
{
    char c = 'Q';
    char *char_pointer = &c;

    printf ("%c %c\n", c, *char_pointer);

    c = '/';
    printf ("%c %c\n", c, *char_pointer);

    *char_pointer = '(';
    printf ("%c %c\n", c, *char_pointer);

    return 0;
}
```

11.2 Lista • Kimenet

```
QQ
//(
()
```

A `c` karakteres változó kezdőértékeként a '`Q`' karaktert adjuk meg. A program következő sorában a `char_pointer`-t karakteres változó mutatójaként deklaráljuk – ezzel azt jelezzük, hogy a változó tartalmát (bármi is az) úgy fogjuk tekinteni, mint közvetett hivatkozást egy karakterre. A deklarációkor (mint bármilyen más változó esetében) megadhatunk kezdőértéket is, és itt is ez történik. A „címe” operátor segítségével a `c` változóra utaló

mutató értéket rendelünk a `char_pointer`-hez. (Fontos, hogy a `c` változó már azelőtt létezzen, mielőtt ez az értékadás megtörténik, mert egyébként „nincs mire mutatni”, és a fordítóprogram hibával leáll.)

A `char_pointer` deklarációját és a kezdőérték beállítását két sorra bontva is megtehetjük, a következő nem éppen triviális módon:

```
char *char_pointer;
char_pointer = &c;
```

Vigyázat, a következő két sor helytelen, pedig látszólag ez lenne az eredeti inicializáció „felbontása”:

```
char *char_pointer;
*char_pointer = &c;
```

Ne feledjük, hogy egy mutató értéke nem értelmezhető addig, amíg be nem állítottuk valamelyen „célontra”.

Mindhárom `printf` utasítás ugyanazt teszi: kiírja `c` értékét, valamint azt az értéket, amire a `char_pointer` mutat. Minthogy a `char_pointer`-t úgy állítottuk be, hogy `c`-re mutasson, az első `printf` nyilván ugyanazt a két értéket fogja megjeleníteni.

A következő utasításban a '/' karaktert töltjük be a `c` változóba. Mivel a `char_pointer` továbbra is a `c`-re mutat, a `*char_pointer` kiírásával is a '/' karaktert (`c` új tartalmát) jelenítjük meg. Ez egy igen fontos elv: annak ellenére, hogy a `char_pointer` értéke mit sem változott, a `*char_pointer` mindenkor c aktuális értékét éri el. Ha tehát `c` értéke megváltozik, akkor a `*char_pointer` is megváltozik.

Így már érthetővé válik a program utolsó szakaszának a működése is. Mivel a `char_pointer` értéke továbbra sem változik meg, a `*char_pointer` ezen túl is `c`-re mutat. Így a

```
*char_pointer = '(';
```

értékadással a nyitó zárójel értékét igazából a `c` változónak adjuk át. Pontosabban fogalmazva: a '(' karaktert annak a változónak adjuk, amire a `*char_pointer` mutat. A program korábbi részéből kiderül, hogy a `char_pointer`-t a `c` változó címére mutatóan állítottuk be – tehát a `c` kapja meg a megadott karaktert.

Az itt vázolt elvek képezik a mutatók használatának alapjait. Ne haladjunk tovább az olvasással mindaddig, amíg van olyan részlet, ami nem világos.

Mutatók használata kifejezésekben

A 11.3 Listában két egész mutatót adunk meg, p1-et és p2-t. Figyeljük meg, hogy a mutatók által hivatkozott értékeket hogyan használhatjuk fel aritmetikai kifejezésekben. Ha p1 egész mutatóként lett definiálva, akkor mit mondhatunk a *p1-ről, amit a kifejezésekben használunk?

11.3 Lista • Mutatók használata kifejezésekben

```
// További tudnivalók a mutatókról

#include <stdio.h>

int main (void)
{
    int i1, i2;
    int *p1, *p2;

    i1 = 5;
    p1 = &i1;
    i2 = *p1 / 2 + 10;
    p2 = p1;

    printf ("i1 = %i, i2 = %i, *p1 = %i, *p2 = %i\n", i1, i2, *p1, *p2);

    return 0;
}
```

11.3 Lista • Kimenet

```
i1 = 5, i2 = 12, *p1 = 5, *p2 = 5
```

Az i1 és i2 egész változók, valamint a p1 és p2 egész mutatók deklarálása után a program 5-öt rendel az i1 változóhoz, majd az i1 címét tárolja el p1-ben. Ezután egy értékadás során az alábbi kifejezést értékeli ki a program:

```
i2 = *p1 / 2 + 10;
```

A 11.2 Lista után kifejtett elvek szerint: ha egy x-re hivatkozó px mutató x típusának megfelelően lett definiálva, akkor a különféle kifejezésekben a *px ugyanolyan szerepet tölt be, mint az x.

Mivel a 11.3 Listában a p1 egész mutatóként jött létre, a fenti kifejezés az egész aritmetikának megfelelően értékelődik ki. *p1 értéke 5 (hiszen p1 i1-re mutat), így a kifejezés értéke 12 – ezt i2-ben tároljuk el. (A közvetett elérés operátora, a * szimbólum magasabb kiértékelési szinten van a C nyelvben, mint az osztás aritmetikai művelete. Sőt, *bármely* kétoperandusú operátornál magasabb szinten van, csak úgy, mint a „címe” operátor, a & szimbólum.)

A program következő sorában p2 értékéül p1-et adjuk át. Ez teljesen szabályos művelet – hatására p2 is ugyanarra az adatra fog mutatni, ahol várunk. Mivel p1 i1-re mutat, a hozzárendelés után p2 is i1-re fog mutatni. Akárhány mutatót beállíthatunk ugyanarra az értékre.

A printf által mutatott kimenet igazolja, hogy az i1, a *p1 és a *p2 mind ugyanazt az értéket adja (5-öt) és hogy az i2 értéke valóban 12-re lett beállítva.

Mutatók és adatszerkezetek

Láttuk, hogy miként állíthatjuk be a mutatókat egy egész szám vagy egy karakteres változó címére. Ám a mutatóknak ennél bonyolultabb adatszerkezetek címét is átadhatjuk. A 9. fejezetben a date adatszerkezetet a következő módon definiáltuk:

```
struct date
{
    int month;
    int day;
    int year;
};
```

Ahogy megadhattunk date adatszerkezetű változót:

```
struct date    todaysDate;
```

Hasonlóképpen megadhatunk date adatszerkezetet címző mutatót is:

```
struct date    *datePtr;
```

Az így definiált datePtr változó várakozásainak megfelelően működik. Beállítható például a todaysDate címére a következő hozzárendeléssel:

```
datePtr = &todaysDate;
```

Egy ilyen értékadás után a datePtr által hivatkozott date adatszerkezet bármely adattagja használható a megfelelő közvetett elérés operátorral:

```
(*datePtr).day = 21;
```

Ezzel a datePtr által hivatkozott date adatszerkezet day (nap) adattagját 21-re állítjuk be. A zárójel nem hagyható el, mert az adattag operátor (.) magasabb precedenciájú, mint a közvetett elérés operátor (*).

A datePtr által hivatkozott date adatszerkezet hónap adattagját a következő utasítással vizsgálhatjuk meg:

```
if      ( (*datePtr).month == 12 )  
    ...
```

Az adatszerkezetekre hivatkozó mutatók olyan gyakran kerülnek elő a C programozási nyelvben, hogy saját operátora is született. A „mutat” -> operátor (kötőjel + „nagyobb” jel) lehetővé teszi, hogy ne kelljen zárójelet és adattag operátort használnunk az adatszerkezetekre hivatkozó mutatókban, hanem a

```
(*x).y
```

kifejezés helyett használhassuk a következőt:

```
x->y
```

Így a fenti (kissé kényelmetlen) feltételvizsgálat így is megfogalmazható:

```
if ( datePtr->month == 12 )  
    ...
```

A 9. fejezet első kód részletét, amely bemutatta az adatszerkezetek használatát, most újrafogalmazzuk adatszerkezetekre hivatkozó mutatókkal a 11.4 Listában.

11.4 Lista • Adatszerkezetekre hivatkozó mutatók használata

```
// Adatszerkezetekre hivatkozó mutatók
```

```
#include <stdio.h>  
  
int main (void)  
{  
    struct date  
    {  
        int month;  
        int day;  
        int year;  
    };  
  
    struct date today, *datePtr;  
  
    datePtr = &today;  
  
    datePtr->month = 9;  
    datePtr->day = 25;  
    datePtr->year = 2004;
```

```

printf ("A mai dátum: %i/%i/%.2i.\n",
       datePtr->month, datePtr->day, datePtr->year % 100);

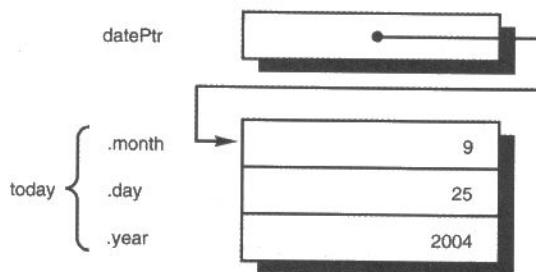
return 0;
}

```

11.4 Lista • Kimenet

A mai dátum: 9/25/04.

A 11.2 ábra a today és a datePtr memóriabeli helyzetét vázolja, miután lefutottak a fenti program értékkedásai.



11.2 ábra

Adatszerkezetre hivatkozó mutató

Be kell vallanunk, hogy ebben a programban sem volt semmi különösebb okunk a mutatók használatára, hiszen mutatók nélkül is boldogan működött a program (már a 9. fejezet eszközeivel is). Hamarosan találunk majd motivációt a mutatók használatára.

Mutatókat tartalmazó adatszerkezetek

Mutatókat adatszerkezetek tagjaként is megadhatunk. Tekintsük például az alábbi definíciót:

```

struct intPtrs
{
    int *p1;
    int *p2;
};

```

Az adatszerkezet két egész mutatót tartalmaz, p1-et és p2-t. intPtrs adatszerkezetű változót a szokásos módon lehet deklárníni:

```
struct intPtrs pointers;
```

A pointers változót a megszokott szintaxisossal lehet használni; szem előtt tartva, hogy ő maga nem mutató, csak van két mutató adattagja.

A 11.5 Lista mutatja be az intPtrs adatszerkezet felhasználási lehetőségeit.

11.5 Lista • Mutatókat tartalmazó adatszerkezetek használata

```
// Mutatókat tartalmazó adatszerkezetek

#include <stdio.h>

int main (void)
{
    struct intPtrs
    {
        int *p1;
        int *p2;
    };

    struct intPtrs pointers;
    int i1 = 100, i2;

    pointers.p1 = &i1;
    pointers.p2 = &i2;
    *pointers.p2 = -97;

    printf ("i1 = %i, *pointers.p1 = %i\n", i1, *pointers.p1);
    printf ("i2 = %i, *pointers.p2 = %i\n", i2, *pointers.p2);
    return 0;
}
```

11.5 Lista • Kimenet

```
i1 = 100, *pointers.p1 = 100
i2 = -97, *pointers.p2 = -97
```

A változók definíciója után a

```
pointers.p1 = &i1;
```

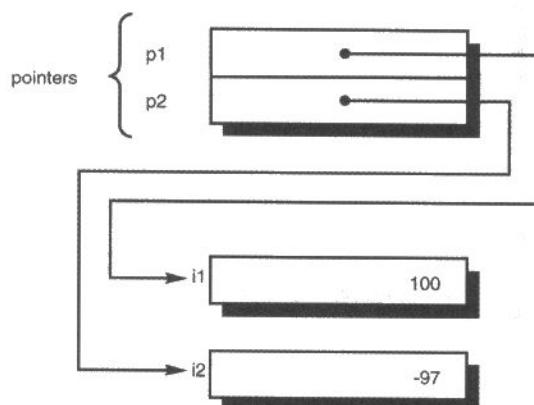
hozzárendelés a pointers adatszerkezet p1 adattagját az i1 nevű egész változó címére mutatva állítja be, míg a

```
pointers.p2 = &i2;
```

hozzárendelés a p2-vel mutat i2-re. Ezután -97-et töltünk be abba a változóba, amelyikre pointers.p2 mutat (esetünkben i2-re). A -97-et tehát áttételesen az i2-höz rendeljük hozzá. Nem kell zárójelet használnunk, mert – ahogy az imént szó volt róla – az adattag operátor (.) magasabb precedenciájú, mint a közvetett elérés operátor (*). Így az adattagra történő hivatkozás *hamarabb* megtörténik (szándékainknak megfelelően), mint a közvetett elérés kiértékelése. A biztonság kedvéért természetesen használhatunk zárójeleket, mivel az ember hamar elfelejtí, hogy mi is az operátorok alapértelmezett végrehajtási sorrendje.

A két printf utasítás igazolja, hogy a hozzárendelések várakozásainknak megfelelően történtek.

A 11.5 Listában szereplő i1, i2 és pointers változók szerepét és kapcsolatrendszerét vizágítjuk meg egy ábrával. A 11.3 ábrán látható, hogy a p1 az i1-re mutat (melynek értéke 100), míg a p2 az i2-re (ahol a -97-et tároljuk).



11.3 ábra

Mutatókat tartalmazó adatszerkezet

Láncolt listák

Az adatszerkezetek címét hordozó mutatók és a mutatókat tartalmazó adatszerkezetek igen hatékony eszközöket adnak a programozók kezébe. Segítségükkel olyan összetett struktúrák valósíthatóak meg, mint például az *egyszeresen láncolt listák*, a *kétszeresen láncolt listák* vagy a *fák*.

Tegyük fel, hogy a következő adatszerkezetet definiáljuk:

```
struct entry
{
    int          value;
    struct entry *next;
};
```

Az entry nevű adatszerkezetnek két tagja van, egy egész szám (value) és egy entry adatszerkezetet címző mutató (next). Gondolkozzunk el ezen egy keveset. Egy entry adatszerkezeten belül találunk egy mutatót egy entry adatszerkezetre. Ennek semmi akadálya nincs a C nyelv logikája szerint.

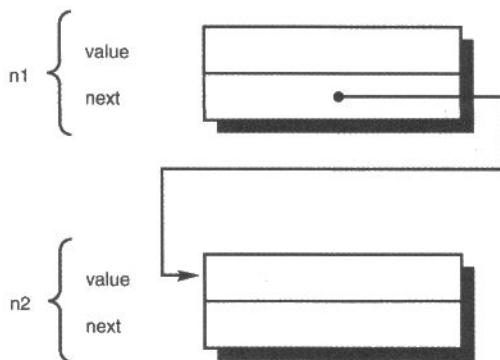
Legyen két változónk, melyeket entry adatszerkezetűnek definiálunk:

```
struct entry n1, n2;
```

Az

```
n1.next = &n2;
```

utasítással az n1 next tagmutatóját az n2 címére tudjuk beállítani. Kapcsolatot létesítünk tehát az n1 és az n2 között (11.4 ábra).



11.4 ábra

Láncolt adatszerkezetek

Ha az n3-at szintén entry adatszerkezetű változóként hozzuk létre, akkor ezt is hozzákapcsolhatjuk az n2-höz a következő utasítással:

```
n2.next = &n3;
```

Az eredményül kapott elemkapcsolati láncot (hivatalos nevén láncolt listát) a 11.5 ábra mutatja grafikusan, és a 11.6 Lista valósítja meg C nyelven.

11.6 Lista • Láncolt lista használata

```
// Láncolt lista

#include <stdio.h>

int main (void)
{
    struct entry
    {
        int           value;
        struct entry *next;
    };

    struct entry n1, n2, n3;
    int          i;

    n1.value = 100;
    n2.value = 200;
    n3.value = 300;
```

```

n1.next = &n2;
n2.next = &n3;

i = n1.next->value;
printf ("%i ", i);

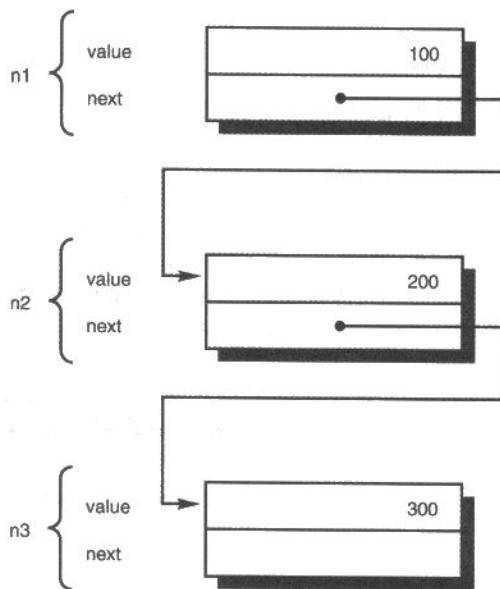
printf ("%i\n", n2.next->value);

return 0;
}

```

11.6 Lista • Kimenet

200 300



11.5 ábra

Láncolt lista

Az n1, n2 és n3 változókat entry adatszerkezetként hozzuk létre, azaz mindegyiknek van egy value nevű egész adattagja és egy entry adatszerkezetet címző (next nevű) mutatója tagja. Az n1, n2 és n3 változók value tagjához sorban hozzárendeljük a 100, 200 és a 300 értéket. A program lelkét jelentő

```

n1.next = &n2;
n2.next = &n3;

```

utasítások összekapcsolják a megfelelő láncszemeket, azaz az n1 next tagját n2-re mutatón, az n2 next tagját n3-ra mutatóán definiáljuk.

Itt következik az

```
i = n1.next->value;
```

utasítás, ami betölti i-be az n1 adatszerkezet next tagja által mutatott adatszerkezet value tagját (ami egy egész szám). Mivel n1.next-et az n2-re mutatóan adtuk meg, a fenti értékkadás az n2 adatszerkezet value tagjára hivatkozik; ennek az értéke 200. Ezt a program kimenete is mutatja.

Első látásra meglepőnek tűnhet, hogy az

```
n1.next.value
```

helyett

```
n1.next->value
```

szintaxisossal hivatkoztunk a másik adatszerkezet kívánt adattagjára. Az n1.next nem adatszerkezet, így nem használhatjuk rá az „adattag” operátort. Ellenben az n1.next (adatszerkezetet címző) mutató; a „mutat” operátor pedig épp ennek a feloldására való. Fontos ez a megkülönböztetés – súlyos programozási hibákhoz vezet, ha ennek megértése nélkül szeretnénk továbbjutni a C nyelv megismerésében.

Az adattag operátor (.) és a „mutat” (->) operátor azonos kiértékelési szinten van. A fenti kifejezésben mindenkor szerepel, ilyenkor balról jobbra történik a műveletek elvégzése. A fenti kifejezés tehát ezzel egyenértékű:

```
i = (n1.next)->value;
```

Zárójelek nélkül is szándékainknak megfelelően zajlik a műveletvégzés.

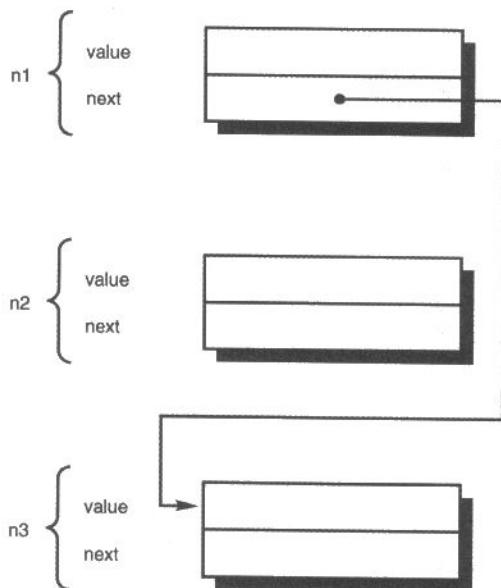
A 11.6 Lista második printf utasításával az n2.next által hivatkozott value értéket írjuk ki. Mivel az n2.next az n3-ra mutat, így az n3.value-t látjuk megjelenni a képernyőn.

A láncolt lista igen jól használható lehetőség. Ha adott egy óriási (de változó) adathalmaz, melynek elemeit szeretnénk meghatározott elrendezésben felsorolni (listázni), akkor a láncolt listával könnyedén megoldható egy-egy elem törlése vagy beszúrása.

Tekintsük a fenti példát, melyben n2 a lista középső eleme. Ha ezt szeretnénk törölni a listából, akkor elég annyit tennünk, hogy az n1.next mutatóját arra állítsuk, amire n2.next mutatója mutatott:

```
n1.next = n2.next;
```

Az utasítás igazából magát a(z n2.next-ben lévő) mutatót másolja át az n1.next-be. Mivel n2.next az n3-ra mutatott, így az n3 változó címe kerül az n1.next-be. Ezek után n1.next nem az n2-re mutat, így elmondható, hogy n2-t töröltük a listából. A 11.6 ábra mutatja a törlés utáni helyzetet.



11.6 ábra

Elem törlése láncolt listából

Természetesen közvetlenül is átadhattuk volna n3 címét n1.next-nek:

```
n1.next = &n3;
```

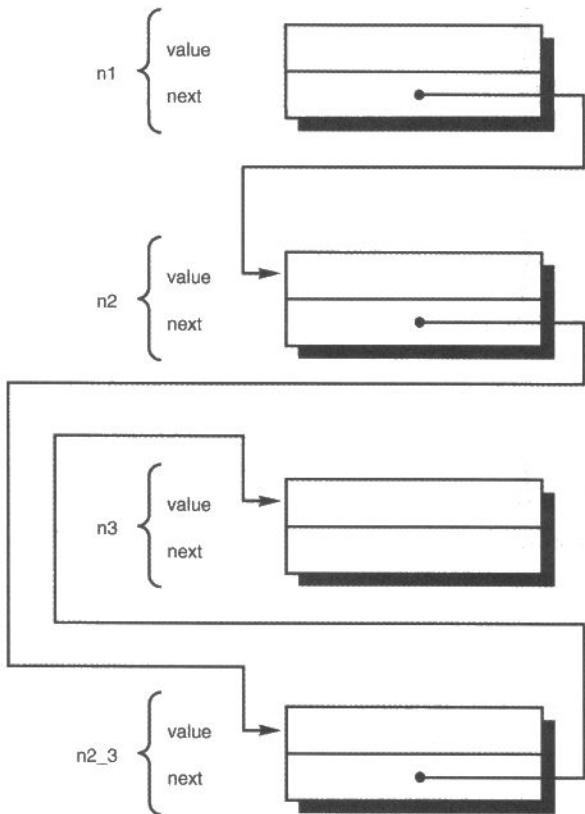
Ám ez nem annyira elegáns megoldás, mert tudnunk kell hozzá, hogy hová mutatott n2.next. Az előző értékadás általánosan oldja meg a törlést.

Egy új elem beszúrása hasonlóan egyszerű. Ha egy entry adatszerkezetű n2_3 változót szeretnénk n2 után felvenni a listába, akkor nincs más teendő, mint beállítani n2_3.next-et arra, amire n2.next mutatott, majd pedig n2.next-et n2_3 címére. Vagyis az

```
n2_3.next = n2.next;
n2.next = &n2_3;
```

utasítássor n2 után illeszti be a listába n2_3-at. Az utasítások sorrendje lényeges, mert a második utasítás felülírja az n2.next-ben tárolt mutatót. Ha ezt hajtanánk végre először, nem lenne mit átadni az n2_3.next számára. Az n2_3 elem beszúrása utáni helyzetet

mutatja a 11.7 ábra. Különös módon maga az n2_3 elem nem az n1 és n3 közé kerül a memóriában. Fontos tudatosítani: egy láncolt lista új eleme nem attól kerül be a listába, hogy a memóriában a szomszédai (n2 és n3) „között” kap helyet. Éppen ez a láncolt listák használatának egyik fő indoka: a listaelémeket nem kell folytonosan elhelyezni a memóriában (mint a tömbök esetében), így gyorsan módosítható egy hatalmas lista is.



11.7 ábra

Elem beszűrása láncolt listába

Mielőtt megírnánk néhány hasznos függvényt a láncolt listák kezelésére, két témát meg kell még említenünk.

A láncolt listákhoz kapcsoltan általában használunk még egy mutatót. Ez többnyire a lista kezdőelemére mutat. Háromelemű listánkhoz (n1, n2 és n3) tehát még vegyünk fel egy `list_pointer` nevű változót, és irányítsuk a lista első elemére a következő értékkadással:

```
struct entry *list_pointer = &n1;
```

A hozzárendelés működéséhez az n1-nek ekkor már definiálnak kell lennie. A lista mutatója arra kell, hogy végig tudjunk lépdelni a listán – ezt hamarosan látni fogjuk.

A másik téma arra vonatkozik, hogy valahogy dokumentálnunk kell, hogy hol is van a lista vége. Ez is arra kell, hogy megfelelően végiglépdelhessünk a lista elemein, és tudjuk, ha ennek a műveletnek a végére értünk. Közmegegyezés szerint ezt úgy szokás megoldani, hogy egy konstans 0-t írunk a lista utolsó elemének mutatójába. Az ilyen célt szolgáló mutatót nullmutatónak (*null pointer*) hívjuk. Ezzel jelezhetjük a lista végét.¹

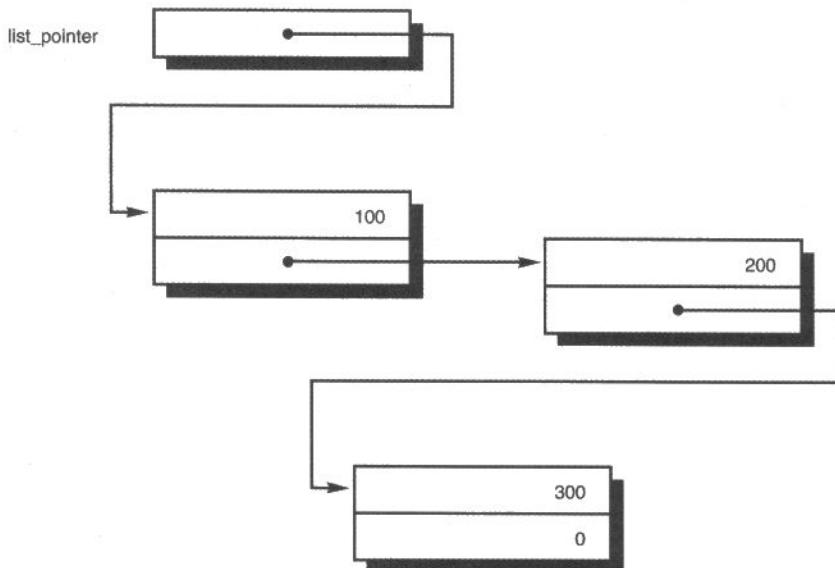
Háromelemű listánkban nullmutató beállításával jelezhetjük, hogy az utolsó elem az n3:

```
n3.next = (struct entry *) 0;
```

A 13. fejezetben látni fogjuk, hogyan tehetjük olvashatóbbá ezt az értékkedést.

Típusátalakító operátorral adjuk a fordítóprogram tudtára, hogy azt a 0-t minden típusúként („entry adatszerkezetre mutatóként”) szeretnénk használni. Ezt nem lenne kötelező így kiírni, de javítja a program érthetőségét, ha megtezzük.

A 11.8 ábra már úgy mutatja be láncolt listánkat, hogy van hozzá (kezdőelemet megadó) list_pointer listamutató, és a nullmutatóból kiderül, melyik a lista utolsó eleme.



11.8 ábra

Láncolt lista listamutatóval és záró nullmutatóval.

¹ A nullmutatóban belsőleg tárolt szimbólum nem feltétlenül a konstans 0. A fordítóprogramnak azonban tudnia kell, hogy ha 0-t adunk értékül egy mutatónak, akkor azt nullmutatónak szánjuk. Ez igaz az összehasonlításra is: akkor kell igaznak értékelnie egy mutató 0-val való összevetését, ha az nullmutató.

A 11.7 Lista az itt leírt elvek szerint hoz létre egy láncolt listát. Egy while ciklussal lépünk végig a lista elemein, és megjelenítjük a lista valamennyi elemének value tagját.

11.7 Lista • Láncolt lista végigjárása

```
// Láncolt lista elemein végigjáró program

#include <stdio.h>

int main (void)
{
    struct entry
    {
        int             value;
        struct entry   *next;
    };

    struct entry    n1, n2, n3;
    struct entry   *list_pointer = &n1;

    n1.value = 100;
    n1.next  = &n2;

    n2.value = 200;
    n2.next  = &n3;

    n3.value = 300;
    n3.next = (struct entry *) 0; // nullmutató jelzi a lista végét

    while ( list_pointer != (struct entry *) 0 ) {
        printf ("%i\n", list_pointer->value);
        list_pointer = list_pointer->next;
    }

    return 0;
}
```

11.7 Lista • Kimenet

```
100
200
300
```

A program elején definiáljuk az n1, n2 és n3 listaelemeket, valamint a list_pointer mutatót, amely a lista első elemére, n1-re mutat. A program következő utasításaival összekapcsoljuk a lista elemeit, az utolsó elem next mutatóját pedig nullmutatóra állítjuk.

Egy while ciklussal végigjárjuk a lista elemeit. Az aktuális listaelemre mindenkor a list_pointer mutat. A ciklus addig tart, amíg a list_pointer nem nullmutató. A ciklusmagból egy printf-fel kiírjuk az aktuális listaelem value tagját.

A ciklusmag `printf` utasítását követően fut le minden cikluslépésben a

```
list_pointer = list_pointer->next;
```

értékkadás, ami a lista következő („`next`”) elemének címét tölti be a `list_pointer`-be. Amikor először fut le a ciklusmag, akkor értékkadás az `n1.next` mutatót adja át a `list_pointer`-nek (hiszen kezdetben a `list_pointer` az `n1`-re mutat). Mivel ez nem nullmutató (hanem az `entry` adatszerkezetű `n2`-re irányuló mutató), a `while` ciklus fut tovább.

A második cikluslépésben a ciklusmag az `n2.value` értékét jeleníti meg (azaz 200-at). Az `n2.next` adattagja másolódik be a `list_pointer`-be, és (minthogy ezt `n3`-ra irányítottuk), a `list_pointer` a ciklus második lépésének a végén már `n3`-ra mutat.

A harmadik cikluslépésben a `printf` az `n3.value`-ban tárolt 300-at írja ki. Ezen a ponton a `list_pointer->next` (ami ekkor konkrétan az `n3.next`) bemásolódik a `list_pointer`-be. Ezt a program elején nullmutatónak definiáltuk, ezért ezután a három cikluslépés után a `while` ciklus befejezi futását.

Érdemes a `while` ciklus futását végigkövetni, és táblázatszerűen, papírral és ceruzával dokumentálni a változók értékeit. Nem mondhatja magáról senki, hogy érti a mutatók filozófiáját, ha nem érti minden részletében ennek a ciklusnak a működését. Programunk komoly erőssége, hogy a lista elemszámától függetlenül használható ugyanez a kód részletek (feltéve, hogy az utolsó elemet nullmutató jelzi).

A valódi programokban használt valódi láncolt listák esetében nem úgy történik az elemek egymáshoz kapcsolása, mint a fenti példában. Nem „kézileg” hozzuk létre és kapcsoljuk össze az elemeket – ez csak a példa megvilágítása miatt történt így. A valódi életből vett programokban azt kérjük a rendszertől, hogy minden új listaelem létrehozásakor bocsássa rendelkezésünkre a szükséges memóriaterületet, és a program futása közben illesztjük be az egyes listaelemeket. Ez a dinamikus memóriakiosztás (*dynamic memory allocation*) mechanizmusa, melyet a 17. fejezetben („Speciális lehetőségek a C nyelvben”) vizsgálunk meg alaposabban.

A `const` kulcsszó és a mutatók

Azt már látuk, hogy miért érdemes bizonyos változókat vagy tömböket konstansként (`const`) definiálni. Ez információt hordoz mind a fordítóprogram, mind a programot olvasó ember számára – az adott változót nincs szándékunkban a program futása során megváltoztatni. A mutatókkal kapcsolatban kétféle változás jöhet szóba: vagy a hivatkozott memóriacím változhat meg, vagy a mutató által hivatkozott változó értéke. Álljunk meg itt egy pillanatra. Figyeljük meg a következő deklarációt:

```
char c = 'X';
char *charPtr = &c;
```

A `charPtr` mutatót ráírányítottuk a `c` változóra. Ha ez a mutató örökké csak `c`-re fog mutatni, akkor érdemes konstansként megadni:

```
char * const charPtr = &c;
```

(Ezt úgy olvassuk, hogy a `charPtr` egy „konstans mutató egy karakterre”.) Ezek után nem adható ki olyasféle utasítás, mint ez:

```
charPtr = &d; // érvénytelen
```

Ilyenkor a GNU fordítóprogram² a következő hibaüzenetet adja:

```
foo.c:10: figyelmeztetés: hozzárendelési kísérlet a csak-olvasható
      ↬ 'charPtr' változóhoz.
(foo.c:10: warning: assignment of read-only variable 'charPtr')
```

Ha viszont az szeretnénk, hogy a hivatkozott változó értékét ne lehessen megváltoztatni a *mutatón keresztül*, akkor ezt így adhatjuk a fordítóprogram tudtára:

```
const char *charPtr = &c;
```

(Úgy olvasandó, hogy a `charPtr` egy „egy konstans karakter mutatója”.) Ez természetesen nem jelenti azt, hogy a (`charPtr` által hivatkozott) `c` változó értékét ne tudnánk megváltoztatni. A fenti deklaráció csak annyit jelent, hogy a következő utasítás hibajelzést fog kiváltani:

```
*charPtr = 'Y'; // érvénytelen
```

A GNU C fordító esetén ilyen üzenetet kapunk:

```
foo.c:11: figyelmeztetés: hozzárendelési kísérlet egy csak-olvasható
      ↬ memória helyhez
(foo.c:11: warning: assignment of read-only location )
```

Ha a mutató által tárolt memóriacím és a hivatkozott változó értéke is állandó a programban, akkor a sor elejére, valamint a típusmegjelölés és a változónév közé is érdemes odaírni a `const` kulcsszót (és még egy `*-ot`):

```
const char * const *charPtr = &c;
```

² Előfordulhat, hogy az olvasó által használt fordítóprogram más hibajelzést (vagy épp semmilyent sem) ad.

Az első `const` arra utal, hogy a mutató által hivatkozott változó tartalma konstans, a második pedig azt deklarálja, hogy maga a mutató sem változhat meg. Ez a deklaráció (főként a két csillag) kétségekívül zavarba ejtő, de ebben a pillanatban nem érdemes ezzel többet küszködni; fogadjuk el, hogy így definiálták a C nyelvet.³

Mutatók és függvények

A mutatók és a függvények igen jól kiegészítik egymást. A mutatók átadhatóak függvényparaméterként, és a függvények visszatérési értéke is lehet mutató.

Az első állítás (mutatók átadása paraméterként) teljesen nyilvánvaló: egyszerűen be kell írni a paraméterek közé a kívánt mutató nevét. Az előző példában szereplő `list_pointer`-t például így adhatjuk át egy `print_list` nevű függvénynek:

```
print_list (list_pointer);
```

A `print_list` függvényben helyes típusmegjelöléssel kell megadni a megfelelő formális paramétert:

```
void print_list (struct entry *pointer)
{
    ...
}
```

Ettől kezdve a `pointer` formális paraméter ugyanúgy használható a függvényben, mint bármely más mutató. Egyet azonban érdemes megjegyezni a paraméterként átadott mutatókkal kapcsolatban: a függvény meghívásakor a mutatók értéke másolódik át a megfelelő formális paraméterbe. Így az eredeti mutatóra nem hat semmilyen változtatás, melyet a függvényen belül a (formális paraméterként látható) mutatón végrehajtunk. De van egy nagyszerű lehetőség: magát a(z eredeti) mutatót ugyan nem tudja megváltoztatni a függvény, ám a mutató által hivatkozott adatok változása maradandó! A 11.8 Lista világossá teszi az elhangzottakat.

11.8 Lista • Mutatók és függvények

```
// A mutatók és függvények barátságát bemutató program

#include <stdio.h>

void test (int *int_pointer)
{
```

³ A `const` kulcsszót nem használtuk minden példaprogramban, ahol lehetett volna, csak a kiválasztott példákban. Amíg egy programozó meg nem barátkozik az olyasfélé kifejezésekkel, mint amit az imént láttunk, addig ezeknek a használata inkább rontja a megértést, mintsem javítaná.

```

        *int_pointer = 100;
    }

int main (void)
{
    void test (int *int_pointer);
    int i = 50, *p = &i;

    printf ("A teszt meghívása előtt: i = %i\n", i);
    test (p);
    printf ("A teszt meghívása után: i = %i\n", i);

    return 0;
}

```

11.8 Lista • Kimenet

A teszt meghívása előtt: i = 50
A teszt meghívása után: i = 100

A test függvényt úgy definiáltuk, hogy egyetlen paramétere egy egész mutató. A függvényen belül egyetlen utasítás fut le: az int_pointer által hivatkozott egész számhoz 100-at rendelünk hozzá.

A main eljárás elején az i egész változót 50-re inicializáljuk, és ráállítunk egy p nevű mutatót. Kiíratjuk az i értékét, majd meghívjuk a test függvényt, átadva neki a p paramétert. Ahogy a kimenet második sorából is látszik, a test függvény valóban megváltoztatta az i értékét (100-ra).

Tekintsük meg most a 11.9 Listát!

11.9 Lista • Értékek megváltoztatása mutatók segítségével

// További tudnivalók a mutatók és függvények világából

```

#include <stdio.h>

void exchange (int * const pint1, int * const pint2)
{
    int temp;

    temp = *pint1;
    *pint1 = *pint2;
    *pint2 = temp;
}

int main (void)
{

```

```

void exchange (int * const pint1, int * const pint2);
int i1 = -5, i2 = 66, *p1 = &i1, *p2 = &i2;

printf ("i1 = %i, i2 = %i\n", i1, i2);

exchange (p1, p2);
printf ("i1 = %i, i2 = %i\n", i1, i2);

exchange (&i1, &i2);
printf ("i1 = %i, i2 = %i\n", i1, i2);

return 0;
}

```

11.9 Lista • Kimenet

```

i1 = -5, i2 = 66
i1 = 66, i2 = -5
i1 = -5, i2 = 66

```

Az `exchange` függvény megcseréli a paraméterként kapott két értéket. A függvény fejléce szerint minden paraméter konstans mutató egy egész számra – azaz maguk a mutatók nem változhatsanak meg a függvényben:

```
void exchange (int * const pint1, int * const pint2)
```

A függvény egyetlen lokális változója a `temp`, amely az értékcsere közben ideiglenes tároló szerepet tölt be. Értékül kapja a `pint1` által mutatott számot. A `pint2` által mutatott számot bemásoljuk a `pint1` által mutatott egész szám memóriaterületére, majd a `temp` értékét töltjük be a `pint2` által mutatott memóriaterületre. Ezzel a csere megtörtént.

A `main` eljárás -5 és 66 értékkel inicializálja `i1`-et és `i2`-t, majd e két változó címére állítja be `p1` és `p2` mutatókat. Megtekinthetjük `i1` és `i2` értékét, majd `p1` és `p2` paramétereit meghívásra kerül a felcserélő függvény (`exchange`). Itt a `p1` által mutatott egész szám és a `p2` által mutatott egész szám helyet cserél. Mivel a korábbiak szerint e két mutató `i1`-re és `i2`-re irányul, az `i1` és `i2` értéke cserélődik fel. A kimenetből látható is a helycsere.

Itt azonban nincs még vége a programnak, sőt, a folytatás az igazán érdekes, ugyanis a felcserélő függvényt most más módon hívjuk meg. Az `exchange` paramétereit (ami két mutató) ezúttal „röptében” gyártjuk le, azaz közvetlenül az `i1` és az `i2` címét adjuk át a megfelelő helyen. Mivel az `&i1` típusa egy egész szám mutatója, átadható egy olyan függvénynek, amely ilyen paramétert vár. Ugyanez a helyzet a második paraméterrel is. A kimenetből látható, hogy a program a várakozásainknak megfelelően működik, azaz újra felcserélődik a két számérték.

Érdemes tudatosítani, hogy mutatók nélkül nem lennének képesek olyan függvényt írni, amely felcseréli két változó értékét. A függvényeknek ugyanis csak egy visszatérési értékük van, és a paraméterként kapott értékek megváltozása csak a függvényen belül marad érvényben. Tanulmányozzuk át még egyszer alaposan a 11.9 Listát. Sok olyan lényeges mozzanatot tartalmaz, amely nélkülözhetetlen a C nyelv mutatóinak a megértéséhez.

A 11.10 Lista azt illusztrálja, hogy miként lehet egy függvény visszatérési értéke egy mutató. A programban definiáljuk a findEntry függvényt, amely egy láncolt listában keres egy adott értéket. Amikor megtalálta, visszatérési értékül adja az adott elemre irányuló mutatót. Ha a keresett érték nincs a listában, akkor nullmutatót ad vissza a függvény.

11.10 Lista • Mutatót visszaadó függvény

```
#include <stdio.h>

struct entry
{
    int value;
    struct entry *next;
};

struct entry *findEntry (struct entry *listPtr, int match)
{
    while ( listPtr != (struct entry *) 0 )
        if ( listPtr->value == match )
            return (listPtr);
        else
            listPtr = listPtr->next;

    return (struct entry *) 0;
}

int main (void)
{
    struct entry *findEntry (struct entry *listPtr, int match);
    struct entry n1, n2, n3;
    struct entry *listPtr, *listStart = &n1;

    int search;

    n1.value = 100;
    n1.next = &n2;

    n2.value = 200;
    n2.next = &n3;

    n3.value = 300;
    n3.next = 0;
```

```

printf ("Adja meg a keresendő értéket: ");
scanf ("%i", &search);

listPtr = findEntry (listStart, search);

if ( listPtr != (struct entry *) 0 )
    printf ("Megvan a %i.\n", listPtr->value);
else
    printf ("A keresett érték nem található.\n");
return 0;
}

```

11.10 Lista • Kimenet

Adja meg a keresendő értéket: 200
 Megvan a 200.

11.10 Lista • Kimenet (újrafuttatás)

Adja meg a keresendő értéket: 400
 A keresett érték nem található.

11.10 Lista • Kimenet (újrafuttatás)

Adja meg a keresendő értéket: 300
 Megvan a 300.

A `struct entry *findEntry (struct entry *listPtr, int match)` függvényfejléc szerint a `findEntry` visszatérési értéke egy `entry` adatszerkezetre irányuló mutató. Két paramétert vár: egy ugyanilyen mutatót, valamint egy egész számot (a keresett értéket). A függvény egy while ciklussal lépdel végig a láncolt lista elemein. A ciklus addig tart, amíg meg nem találjuk a keresett értéket az elemek között (ilyenkor az aktuális `listPtr` értéke azonnal visszaadódik a hívó eljárásnak), vagy amíg a lista végére nem érünk (azaz bele nem botlunk egy nullmutatóba, ami kilépteti a ciklust; ekkor nullmutató lesz a visszatérési érték).

A `main` eljárásban az előző programokhoz hasonlóan állítjuk elő a láncolt listát, majd bekérjük a felhasználótól a keresendő értéket. A `findEntry` függvényt a lista első elemére irányuló `listStart` mutatóval és a keresendő értékkel (`search`) hívjuk meg.

A `findEntry` által visszaadott mutatóértéket egy `entry` adatszerkezetre irányuló mutató, a `listPtr` kapja meg. Ha a `listPtr` nem nullmutató, akkor a `listPtr` által hivatkozott listaelem `value` adattagját megjelenítjük. Ennek meg kell egyeznie a felhasználó által beírt értékkel. Ha azonban a `listPtr` nullmutató, akkor a keresés sikertelenségét visszajelző üzenet jelenik meg.

A kimenetből látható, hogy a program megtalálta a 200 és a 300 értéket, ám a 400-at – helyesen – nem.

Úgy tűnik, mintha semmi haszna nem lenne a `findEntry` által visszaadott mutatónak. Ám vannak olyan programozási helyzetek, amikor a listaelem más adattagjait is szeretnénk elérni a keresés után – ilyenkor nélkülözhetetlen a visszakapott mutató. Ha a 10. fejezetben használt szótárat egy láncolt listaként valósítanánk meg, akkor a `findEntry` (vagy a 10. fejezet szóhasználatával `lookup`) meghívásával megkereshetnénk egy szót a láncolt-listaszótárban. A `lookup` által visszaadott mutatót felhasználhatnánk a szóhoz tartozó meghatározás kinyerésére.

Lennének bizonyos előnyei a szótár láncolt listaként való tárolásának. Könnyen be lehetne illeszteni és könnyen el is lehetne távolítani egy-egy bejegyzést. (A beillesztés csak annyiból állna, hogy megkeresnénk, hogy hová kell beszúrni a szót, majd átállítanánk két mutatót; ezt a fejezet korábbi részében már láttuk.) A 17. fejezetben még előkerül a dinamikus memóriaoglalás – ilyen szempontból is előnyös egy láncolt lista használata, amikor bővítjük a szótárat.

A szótár láncolt listaként való megvalósításának azonban van egy komoly hátránya. Ilyen listára nem alkalmazható a logaritmikus gyorskeresés. Az algoritmus ugyanis csak közvetlenül indexelt adatokkal használható. Meg kell elégednünk tehát a lineáris kereséssel, mert az egyes elemeket csak az előző elem segítségével találhatjuk meg.

Van mód arra is, hogy egyesítsük az elemek gyors beszúrási és törlési lehetőségét a gyorskeresés lehetőségével. Ehhez egy más struktúrát: *fát* kell használnunk a szótár tárolásához. Vannak más eljárások is, például a hasítótáblák (*hash tables*). Az olvasó találhat érdekes algoritmusokat a szakirodalomban, például Donald E. Knuth és Addison-Wesley „A számítógép-programozás művészete” (*The Art of Computer Programming*) című könyvének 3. kötetében: „Keresés és rendezés”. Az itt előkerülő struktúrák és algoritmusok könnyen megvalósíthatóak a C programozási nyelv (általunk is tárgyalt) eszközeivel.

Mutatók és tömbök

A C nyelv mutatóit legtöbbször tömbökre irányulóan használjuk. Így ugyanis kényelmes jelölési lehetőségek kínálkoznak, másrészt a program igen gyors lesz, és viszonylag kevés memóriát használ (azaz hatékony lesz). Ennek okai hamarosan világossá válnak az olvasó számára.

Legyen egy tömbünk, amely száz egész számot tud tárolni. Létrehozható egy `valuesPtr` mutató, amellyel a tömb elemei elérhetőek. Az

```
int *valuesPtr;
```

utasítás megadásakor nem jelöljük, hogy tömbként szeretnénk használni a hivatkozott memóriacímet, csak azt, hogy milyen típusú adatot(ka)t szeretnénk elérhetővé tenni a változóval.

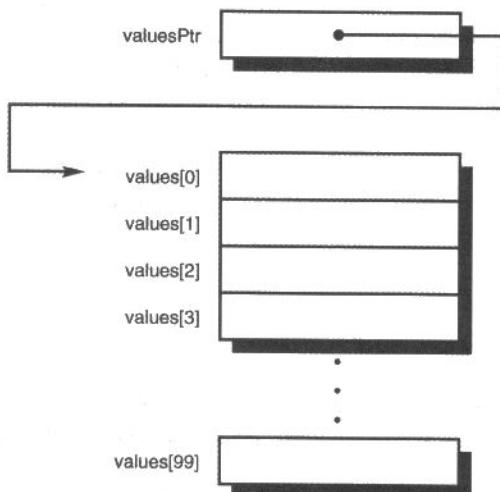
Egy text nevű karaktertömbhöz az előzőhöz hasonlóan létrehozhatunk egy mutatót:

```
char *textPtr;
```

A valuesPtr-rel úgy mutathatunk rá a tömb első elemére, hogy egyszerűen értékül adjuk neki a tömb nevét:

```
valuesPtr = values;
```

Nem használjuk a „címe” operátort, mivel a C fordítóprogram a tömbnév index nélküli megjelenését mutatóként kezeli. Így a név index nélküli megadása az első elemre irányuló mutatót adja (11.9 ábra).



11.9 ábra

Tömb adott elemére irányuló mutató

Az első elemre irányuló mutató létrehozható a „címe” operátorral is. A fenti értékkedással egyenértékű a következő utasítás:

```
valuesPtr = &values[0];
```

Azaz a valuesPtr mutatót ráirányíthatjuk a tömb első elemének a címére.

A következő, egymással egyenértékű utasításokkal mutathatunk rá a text karakterlánc első elemére (a textPtr-nek történő értékkedáskor):

```
textPtr = text;
textPtr = &text[0];
```

A választás pusztán ízlés dolga.

A mutatók használatának előnye akkor tapasztalható meg igazán, amikor végig kell lépdelnünk egy tömb elemein. Ha a `valuesPtr` egy (már korábban definiált, a `values` tömb első elemére irányuló) mutató, akkor a

```
*valuesPtr
```

használható a `values` tömb első elemének (azaz a `values[0]`-nak) az eléréséhez.

Ha a `values[3]`-at szeretnénk elérni a `valuesPtr`-en keresztül, akkor hármat hozzá kell adni a `valuesPtr` értékéhez(!), és erre kell meghívni a közvetett elérés operátort:

```
*(valuesPtr + 3)
```

Általában is igaz, hogy a tömb `i`-ik elemét, a `values[i]`-t így érhetjük el:

```
*(valuesPtr + i)
```

Ha mondjuk a `values[10]`-be szeretnénk 27-et betölteni, akkor ezt a következő két utasítás bármelyikével megtehetjük:

```
values[10] = 27;
*(valuesPtr + 10) = 27;
```

Ha a `valuesPtr` értékét nem a tömb kezdőelemére, hanem a második elemére szeretnénk irányítani, akkor a `values[1]`-re kell alkalmazni a „címe” operátort, és ezt kell értékel adni a `valuesPtr`-nek:

```
valuesPtr = &values[1];
```

Ha a `valuesPtr` `values[0]`-ra mutat, akkor nem nehéz elérni, hogy inkább a `values[1]`-re mutasson. Mindössze meg kell növelni 1-el a `valuesPtr` értékét:

```
valuesPtr += 1;
```

Ez egy teljesen helyes C kifejezés, és *bármilyen* adattípusra vonatkozó mutató esetén használható.

Általánosságban ha a egy olyan tömb, melynek elemei `x` típusúak, és `px` egy `x` típust címző mutató, valamint `i` és `n` egész konstans (vagy változó), akkor a

```
px = a;
```

utasítás úgy állítja be `px`-et, hogy ez az a tömb első elemére mutasson.

A

```
* (px + i)
```

kifejezés következésképp az `a[i]`-re mutat. A

```
px += n;
```

utasítás pedig a tömbön belül `n` pozícióval arrébb állítja a `px` mutatót, *függetlenül attól, hogy a tömb milyen típusú elemekből áll.*

Az inkrementáló és dekrementáló operátorok (`++` és `--`) különösen is hasznosak a mutatók kezelésekor. Az inkrementáló utasítás ugyanúgy hat a mutatókra, mintha egyet hozzáadnánk, a dekrementálás pedig olyan, mintha egyet kivonnánk. Ha például a `textPtr` egy karakter típusra irányuló mutató, és a `text` karaktertömb kezdetére mutat, akkor a

```
++textPtr;
```

utasítás a szöveg következő karakterére, azaz `text[1]`-re irányítja át a `textPtr`-t. Hasonlóképp a

```
--textPtr;
```

utasítás a `textPtr`-t a `text` karakterlánc aktuális karakterét megelőző karakterére állítja át (feltéve, hogy van ilyen; azaz a `textPtr` nem a `text` legelejére mutatott).

A C nyelvben két mutató értéke összehasonlítható. Ez akkor nagyon hasznos, amikor ugyanahhoz a tömbhöz két mutatót is használunk. Megvizsgálhatjuk például, hogy a `valuesPtr` mutató túlhaladt-e már a száz elemet tartalmazó tömb utolsó elemén. Ehhez elég az utolsó elemmel összehasonlítani a mutatót. Ha a

```
valuesPtr > &values[99]
```

kifejezés igaz értéket ad (azaz nem nullát), akkor a `valuesPtr` már túlhaladt a tömb utolsó elemén; egyébként pedig hamis (nulla) értéket ad a vizsgálat. Az imént elhangzottaknak megfelelően a fenti vizsgálat egyenértékű a következővel:

```
valuesPtr > values + 99
```

Az index nélkül használt tömbnév ugyanis tulajdonképpen a tömb kezdőelemére irányuló mutató. (Ez ugyanolyan, mintha `&values[0]`-t írnánk.)

A 11.11 Lista a tömbökre irányuló mutatókat illusztrálja. Az `arraySum` függvény egy egészkből álló tömb elemeinek az összegét számítja ki.

11.11 Lista • Tömbökre irányuló mutatók

```
// Egészkeből álló tömb elemeinek az összegét kiszámító függvény

#include <stdio.h>

int arraySum (int array[], const int n)
{
    int sum = 0, *ptr;
    int * const arrayEnd = array + n;

    for ( ptr = array; ptr < arrayEnd; ++ptr )
        sum += *ptr;

    return sum;
}

int main (void)
{
    int arraySum (int array[], const int n);
    int values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

    printf ("Az összeg: %i\n", arraySum (values, 10));

    return 0;
}
```

11.11 Lista • Kimenet

Az összeg: 21

Az arraySum függvényben megadunk egy egész számra irányuló konstans mutatót, az arrayEnd-et, és beállítjuk az array tömb utolsó elemére. Ezután egy for ciklussal bejárjuk az array elemeit. A ptr-t az array kezdetére irányítjuk a ciklus inicializációs részében. minden cikluslépésben megnöveljük a sum értékét a ptr által mutatott tömbelem értékével. A ciklus léptetési utasítása a ptr inkrementálásából áll; ennek hatására a ptr a következő elemre mutat. Amikor a ptr túllépi a tömb végső határát, a for ciklus kilép, és a sum értékét visszaadjuk a hívó eljárásnak.

Rövid kitérő a programok optimalizálásáról

Ha alaposabban megnézzük az előző programot, láthatjuk, hogy az arrayEnd változó lényegében felesleges. A ciklusfeltételben közvetlenül is elvégezhető lenne a tömb végső határának a kiszámítása és összehasonlítása a pointer-rel:

```
for ( ...; pointer <= array + n; ... )
```

Az `arrayEnd` változót kizárálag a program optimalizálása érdekében vezettük be. minden cikluslépésben lezajlik a feltételvizsgálat. Az `array + n` értéke nem változik meg a ciklus futása során; értéke konstansnak tekinthető. Érdemes tehát még a ciklus elindulása előtt kiszámítani ezt a küszöbértéket, hogy ne kelljen minden cikluslépésben futási időt pazarolni a számításra. Példaprogramunkban túl sok időt nem nyerünk a tízelemű tömb összadásakor (különösen így, hogy csak egyszer kerül meghívásra az `arraySum` függvény), nagyobb tömbök feldolgozása esetén azonban jelentős időt nyerhetünk az efféle optimalizációval.

A hatékonyság másik forrása maga a mutatóhasználat. A fentebb látható `arraySum` függvény for ciklusában a `*ptr` kifejezést használjuk a tömb elemeinek az eléréséhez. Amíg nem ismertük a mutatókat, ugyanezt az összegzést valószínűleg egy `i` ciklusváltozóval és az `array[i]` lekérdezésével oldottuk volna meg. Egy tömb indexelési folyamata általában tovább tart, mint egy mutató által hivatkozott érték elérése. Igazából ez a legfőbb oka annak, hogy a programozók szeretnek mutatót használni egy tömb elemeinek az eléréséhez: általában gyorsabb, hatékonyabb kódot lehet így kapni. Természetesen ez csak akkor jelent időnyereséget, ha sorban dolgozzuk fel a tömb elemeit. Önmagában a `*(pointer + j)` kiszámítása ugyanannyi ideig tart, mint az `array[j]` elöhívása.

Ez most tömb vagy mutató?

Amikor egy tömböt szeretnénk paraméterül adni egy függvénynek, akkor egyszerűen meg kell adni a tömb nevét (ahogy azt az `arraySum` függvényben is tettük). Másrészt akkor is elég volt megadnunk a tömb nevét, amikor egy tömbre irányuló mutatót szerettünk volna létrehozni. E két tényből az következik, hogy az `arraySum` meghívásakor igazából a `values` tömbre irányuló mutatót adjuk át a függvénynek. Pontosan ez a helyzet, és ez ad magyarázatot arra is, hogy miért lehet egy tömb elemeit megváltoztatni a függvénykből.

Felmerülhet a kérdés, hogy (ha igazából tömbre irányuló mutatót adunk át a függvénynek) akkor ennek megfelelően a formális paraméterlistában miért nem mutató típust adunk meg? Konkrétan az `arraySum`-beli `array` miért nem az

```
int *array;
```

utasítással kerül megadásra? Nem kellene-e a tömböket mind mutatókként definiálni a függvényekben?

Ezen kérdések megválaszolásához térjünk vissza az előző meggondolásokhoz a mutatókkal és tömbökkel kapcsolatban. Ha a `valuesPtr` ugyanolyan típusú elemre mutat, mint amilyeneket a `values` tömb tartalmaz, akkor a `*(valuesPtr + i)` kifejezés végül is teljesen egyenértékű a `values[i]` kifejezéssel, feltéve, hogy a `valuesPtr` a `values` tömb kezdőelemére mutat. Ebből az következik, hogy a `*(values + i)` kifejezéssel is hivatkozhatunk a `values` tömb `i`-ik elemére. Általában is igaz a C nyelvben, hogy ha `x` egy (bármi-lyen típusú) tömb, akkor az `x[i]` minden egyenértékű a `*(x + i)` kifejezéssel.

Látható, hogy a tömbök és mutatók bensőséges kapcsolatban állnak egymással a C nyelvben. Ez okozza azt, hogy az `arraySum` függvényben az `array`-t egészekből álló *tömbként* és egészre irányuló *mutatóként* is megadhatjuk. Mindkét deklaráció jól működő programhoz vezet – érdemes kipróbalni.

Ha egy függvényben indexelve használunk egy tömböt, akkor a célszerű őt a formális paraméterlistában tömbként definiálni; ha pedig közvetett eléréssel (*) használjuk, akkor érdemesebb mutatóként megadni. A cél az, hogy a deklaráció tükrözze a változó felhasználási módját.

Mivel az `array`-t fenti programunkban (egy értékkadás után) végül is mutatóként használjuk, kifejezőbb ennek megfelelően is deklarálni. Ezzel kiküszöbölik a `ptr` segédváltozó használatát. Ezt mutatja be a következő kódrészlet.

11.12 Lista • Egy tömb elemeinek az összegzése, második verzió

```
// Egészekből álló tömb elemeinek az összegét kiszámító függvény,
// második verzió
```

```
#include <stdio.h>

int arraySum (int *array, const int n)
{
    int sum = 0;
    int * const arrayEnd = array + n;

    for ( ; array < arrayEnd; ++array )
        sum += *array;

    return sum;
}

int main (void)
{
    int arraySum (int *array, const int n);
    int values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

    printf ("Az összeg: %i\n", arraySum (values, 10));

    return 0;
}
```

11.12 Lista • Kimenet

Az összeg: 21

Az előzőek alapján a program magától értetődő. A `for` ciklus inicializációs részét elhagy-tuk, mert nem kellett semmit sem inicializálni a ciklus kezdetekor. Van egy figyelemre méltó pontja a programnak: az `arraySum` függvény meghívásakor az `array` (tömbre irányuló) mutatót adjuk át paraméterként; az ennek megfelelő formális paramétert a függvényen belül is ugyanígy nevezzük. Magának az `array` mutatónak a függvényen belüli megváltozása azonban semmilyen módon nem hat vissza az `array` tömb tartalmára.

Az `array` változó inkrementálásával csak az módosul, hogy mire mutat a mutató, de a hivatkozott értékek megmaradnak eredeti állapotukban. (Természetesen a mutató által hivatkozott értékek, azaz a tömb elemei megváltoztathatóak lennének a függvényből is a mutató révén, de ez itt nem történik meg, mert csak összegünk.)

Karakterláncok mutatói

Tömbre irányuló mutatókat leggyakrabban karaktertömbök esetén használunk. Ennek okai a kényelmes jelölési lehetőségből és a hatékonyságból adódnak. Annak bemutatására, hogy milyen jól használhatóak a karakterlánc-mutatók, írunk meg egy `copyString` nevű függvényt, mely átmásol egy karakterláncot. Normál tömbindexelési módszerrel ez így nézhet ki:

```
void copyString (char to[], char from[])
{
    int i;

    for ( i = 0; from[i] != '\0'; ++i )
        to[i] = from[i];

    to[i] = '\0';
}
```

A `for` ciklus kilép, mielőtt a null karakter átmásolódna a másik tömbbe, emiatt látható a függvényben az utolsó utasítás.

Ha ugyanezt a függvényt mutatókkal fogalmazzuk meg, akkor nincs szükség az `i` változónak. Ezt a változatot mutatja be a 11.13 Lista.

11.13 Lista • Karakterlánc másolása mutató-léptetéssel

```
#include <stdio.h>

void copyString (char *to, char *from)
{
    for ( ; *from != '\0'; ++from, ++to )
        *to = *from;

    *to = '\0';
}
```

```

int main (void)
{
    void copyString (char *to, char *from);
    char string1[] = "Egy átmásolandó karakterlánc.";
    char string2[50];

    copyString (string2, string1);
    printf ("%s\n", string2);

    copyString (string2, "És még egy.");
    printf ("%s\n", string2);

    return 0;
}

```

11.13 Lista • Kimenet

Egy átmásolandó karakterlánc.
És még egy.

A `copyString` függvénynek két formális paramétere van, a `to` és a `from`. Ezeket karakter-mutatóként és nem karaktertömbként adjuk meg (szemben a korábbi `copyString`-verzióval). Ezzel fejezzük ki a változók felhasználási módját a legmegfelelőbben.

Ekkor kezdődik el egy (inicializáció nélküli) `for` ciklus, amely átmásolja a `from` által mutatott karakterláncot a `to` által mutatott karakterláncba. Cikluslépésenként inkrementáljuk a `from` és a `to` értékét a ciklusfej léptetési részében. Ez eggyel továbblépteti a hivatkozott karakterpozíciót mind a másolandó, mind a végeredményként előálló karakterláncban, azaz a következő lépésekben a forrás- és a célkarakter is a következő helyen levő karakter lesz.

Ha a `from` egy null karakterre mutat, a ciklus kilép. Ekkor még egy null karaktert illesztünk a végeredményként kapott karakterlánchoz.

A `main` eljárásban a `copyString`-et kétszer hívjuk meg; először a `string1`-et másoljuk át a `string2`-be, majd pedig egy konstans karakterláncot („És még egy.”) másolunk át a `string2`-be.

Konstans karakterláncok és a mutatók

A következő utasítás meglepően jól működik, pedig egy karakterlánc-mutatót váró függvénynek konstans karakterláncot adtunk át benne:

```
copyString (string2, "És még egy.")
```

Ennél több is igaz: egy C programban bárhol is használunk konstans karakterláncot, ott igazából egy mutató jön létre, mely a megadott karakterláncra irányul. Legyen a `textPtr` egy karakter-mutató:

```
char *textPtr;
```

Ilyenkor a

```
textPtr = "Konstans karakterlánc";
```

utasítás hatására a `textPtr` mutató a „Konstans karakterlánc” első elemére fog mutatni. Figyeljünk arra, hogy a karakter-mutatók és a karaktertömbök nem teljesen azonosak. A fenti értékkedés ugyanis megdöbbentő módon *nem* használható karaktertömbök esetén. Legyen a `text` egy karaktertömb:

```
char text[80];
```

Ebben az esetben az alábbi utasítás nem működik:

```
text = "Így sajnos nem lehet";
```

Kivéve akkor, amikor egy karaktertömböt a létrehozásakor inicializálunk:

```
char text[80] = "Így már helyes";
```

Ha ily módon inicializáljuk a karaktertömböt, akkor nem az történik, hogy létrejön egy mutató az „így már helyes” karakterláncra, hanem az, hogy a frissen létrejött karaktertömb memóriaterülete feltöltődik a megadott karakterekkel.

Ha ellenben a `text` egy karakter-mutató, akkor a

```
char *text = "Így már helyes";
```

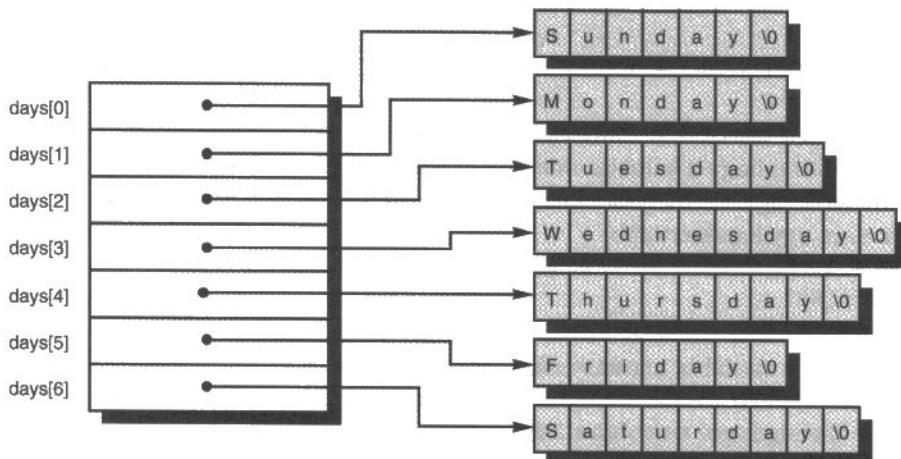
inicializáció ráállítja a `text` mutatót az „így már helyes” (programkódbeli) szövegrészre.

Nézzünk egy példát arra, hogy milyen eltérések mutatkoznak a (tömbként megadott) karakterláncok és a karakter-mutatók között. Az alábbi utasítás egy `days` (*napok*) nevű tömböt hoz létre, amely a hétfő napjaira (mint karakterláncokra) irányuló mutatókat tartalmaz.

```
char *days[] = { "Sunday", "Monday", "Tuesday", "Wednesday",
    ↪ "Thursday", "Friday", "Saturday" };
```

A days tömb hét mutatót tartalmaz, melyek mindenike egy-egy karakterláncra mutat. A days[0] tehát a „Sunday” karakterláncra, a days[1] a „Monday” karakterláncra stb. mutat (11.10 ábra). A hét harmadik napját például így írathatjuk ki:

```
printf ("%s\n", days[3]);
```



11.10 ábra

Mutatók tömbje

További részletek az inkrementáló és dekrementáló utasításokról

Egészen eddig csak úgy használtuk az inkrementáló és a dekrementáló utasításokat, hogy semmilyen más operátor nem szerepelt a kifejezésben. Amikor azt írtuk, hogy $++x$, akkor biztosak voltunk abban, hogy ez az utasítás eggyel megnöveli x értékét. Azt is látuk, hogy ez az operátor a mutatókkal is jól együttműködik. Ha x egy tömb valamely elemére mutat, akkor az inkrementálás után már a következő elemre mutat.

Inkrementáló és dekrementáló utasítások olyan szövegkörnyezetben is használhatóak, ahol más operátorok is megjelennek. Ilyenkor fontossá válik, hogy mikor is léptetnek ezek az operátorok.

Az eddigiekben mindig a változó neve elő írtuk az operátort. Például i megnöveléséhez a

```
++i;
```

kifejezést használtuk. Valójában az operátor a változó neve után is elhelyezhető:

```
i++;
```

Mindkét változat helyes, és mindenkető megnöveli a értékét eggyel. Az első változatot, melyben a változó neve előtt szerepel az operátor, szaknyelven *prefix inkrementálásnak*, a másodikat *postfix inkrementálásnak* hívjuk (a „prefix” előtagot, a „postfix” pedig utótagot jelent). Ugyanez igaz a dekrementáló operátorra is; azaz a

`--i;`

prefix dekrementálást, az

`i--;`

pedig *postfix dekrementálást* hajt végre. Mindkettő kivon egyet i-ből. Ennek a *prefix* vagy *postfix* tulajdonságának akkor lesz jelentősége, amikor más operátorok is megjelennek a szövegkörnyezetben.

Legyen két egész változónk: i és j; az i értékét inicializáljuk 0-ra. Ekkor a

`j = ++i;`

utasítás j-hez nem 0-t, hanem 1-et rendel hozzá. Prefix inkrementálás esetén az érték megváltoztatása a kifejezés kiértékelése előtt történik meg. A fenti utasításban tehát első lépésben az i értéke 0-ról megnő 1-re, és csak ezután történik meg a hozzárendelés. Mintha a következő két utasítás futna le:

`++i;
j = i;`

Ha pedig postfix inkrementálást hajtunk végre, akkor a

`j = i++;`

utasításban az inkrementálás a kifejezés kiértékelése után történik meg. Ha a fenti utasítás előtt az i értéke 0, akkor 0-t rendelünk hozzá j-hez, és csak ezután inkrementáljuk az i-t, mintha ez állna a programban:

`j = i;
++i;`

Nézzünk egy másik példát. Ha i értéke 1, akkor az

`x = a[--i];`

a[0]-t rendeli hozzá x-hez, mivel a csökkentés még a kifejezés kiértékelése (azaz a megfelelő index megkeresése) előtt megtörténik. Ezzel szemben az

`x = a[i--];`

utasítás az a[1]-et tölti be x-be, mivel a csökkentés csak a kifejezés kiértékelése (azaz a megfelelő index használata) után történik meg.

A prefix és postfix inkrementálás és dekrementálás harmadik példájaként tekintsük a következő két függvényhívást. A

```
printf ("%i\n", ++i);
```

utasítás először növeli meg i-t, és utána hívja meg a printf-et, míg a

```
printf ("%i\n", i++);
```

először kiírja i-t, és csak utána növeli meg az értékét. Ha például i tartalma előzőleg 100 volt, akkor az első utasítás 101-et ír ki, míg a második 100-at. Mindkét esetben 101 lesz i értéke a printf utasítás után.

Befejező példaként tekintsük meg a 11.14 Listát. Az ebben szereplő

```
* (++textPtr)
```

kifejezés először megnöveli a textPtr értékét, és csak utána olvassa be a hivatkozott karaktert, míg a

```
* (textPtr++)
```

először beolvassa a textPtr által mutatott értéket, és csak utána lépteti odébb a mutatót. A zárójel minden esetben elhagyható, mert a * és a ++ operátorok azonos precedenciájúak; jobbról balra értékelődnek ki.

Térjünk még vissza a 11.13 Lista copyString függvényéhez. Írjuk át oly módon, hogy az inkrementálás művelete az értékkopíáskor történjen meg.

Mivel a to és a from mutatók minden értékkopíával alkalmával továbbléptetődnek a ciklusmagban, postfix változatban kellene magába az értékkopíásba beilleszteni az inkrementálást. A 11.13 Lista újraírt változatában tehát teljesen elmarad a ciklusfej léptetési kifejezése:

```
for ( ; *from != '\0'; )
    *to++ = *from++;
```

A hozzárendelési művelet az alábbi módon zajlik le: a program kiolvassa a from által mutatott értéket, majd továbblépteti a from mutatót a forrás karakterlánc következő karakterére. Ezután az előbb kiolvasott karakterérték bekerül a to által mutatott memóriacímre, majd a to mutató is megnő egygyel, és az utasítást követően már a cél-karakterlánc következő pozíciójára mutat.

Vizsgáljuk meg alaposan a fenti ciklust, mielőtt továbbhaladunk az olvasással. Az efféle megoldások gyakran előfordulnak a C programokban, így nem érdemes halogatni a megértését.

Korábban szó volt arról, hogy mikor érdemes `for` ciklust használni, és mikor `while` ciklust. Ebben a lerövidült kódészletben a `for` ciklusfej már nem tartalmaz sem inicializációt, sem léptetést – a feltételvizsgálat pedig egy while ciklusban is megoldható. Érdemesebb tehát átérni egy `while` ciklusra. A 11.14 Listában egy újabb verzióját készítjük el a `copyString` függvénynek – ebben már `while` ciklust használunk. A ciklusfeltételben azt is kihasználjuk, hogy a null karakter valójában 0; ezzel a lehetőséggel gyakran élnek a C programozók.

11.14 Lista • Karakterlánc másolása mutató léptetéssel, átírt változat

```
//Karakterlánc másolása mutató léptetéssel, while ciklussal

#include <stdio.h>

void copyString (char *to, char *from)
{
    while ( *from )
        *to++ = *from++;

    *to = '\0';
}

int main (void)
{
    void copyString (char *to, char *from);
    char string1[] = "Egy átmásolandó karakterlánc.";
    char string2[50];

    copyString (string2, string1);
    printf ("%s\n", string2);

    copyString (string2, "És még egy.");
    printf ("%s\n", string2);

    return 0;
}
```

11.14 Lista • Kimenet

Egy átmásolandó karakterlánc.
És még egy.

Műveletek mutatókkal

Azt már láttuk fejezetünk korábbi részében, hogy egész számokat hozzáadhatunk a mutatókhöz (vagy kivonhatunk belőlük). Arra is láttunk példát, hogy két mutató összehasonlító egymással; egyenlők-e, vagy valamelyik nagyobb-e. Ezeken kívül csak egyfajta művelet megengedett a mutatókkal: az azonos típusú mutatókat ki lehet vonni egymásból. A különbség arra mutat rá, hogy hány elem van a két mutatott elem között. Ha a egy adott típusú tömb valamelyik eleme, és b ugyanennek a tömbnek egy másik elemére mutat, akkor a b – a különbség azt adja meg, hogy hány elemet fog közre a két mutató. Legyen p egy mutató, mely az x tömb valamelyik elemére mutat. Ekkor az

```
n = p - x
```

hozzárendelés hatására az n (egész) változó értéke annak az elemnek az indexe, amire p mutat.⁴

Ha például p a századik x-beli elemre mutat a következő utasítás folytán:

```
p = &x[99];
```

akkor n értéke a fenti kivonás után 99 lesz.

Ennek a frissen megszerzett tudásnak a birtokában már megérthető a 11.15 Lista, mely a 10. fejezetben olvasható `stringLength` függvény újraírt változatát tartalmazza.

A programban a `cptr` mutatót használjuk fel arra, hogy végiglépdeljünk a `string` karakterláncon, amíg el nem érjük a végét jelző null karaktert. E fontos pillanatban pedig ki-vonjuk a `cptr` értékéből a `string` értékét. Ezzel megkapjuk a `string` karakterlánc hosszát. A kimenet igazolja a program helyes működését.

11.15 Lista • Karakterlánc hosszának meghatározása mutatóval

```
// Egy karakterlánc karaktereinek összeszámítása (mutatóval)

#include <stdio.h>

int stringLength (const char *string)
{
    const char *cptr = string;

    while (*cptr)
        ++cptr;
```

⁴ A kivonással keletkező előjeles egész szám konkrét típusa (`int`, `long int` vagy `long long int`) a `ptrdiff_t`-ben tárolódik, mely a `<stddef.h>` szabványos fejlécállományban van megadva.

```

        return cptr - string;
    }

int main (void)
{
    int stringLength (const char *string);

    printf ("%i ", stringLength ("stringLength test"));
    printf ("%i ", stringLength (""));
    printf ("%i\n", stringLength ("complete"));

    return 0;
}

```

11.15 Lista • Kimenet

17 0 8

Függvénymutatók

Elég nehezen emészthető témát jelentenek a függvénymutatók, de a teljesség kedvéért beszélünk kell róluk. A fordítóprogramnak nemcsak azt kell tudni egy függvénymutató esetében, hogy az adott mutató egy függvényre mutat, hanem azt is, hogy milyen a függvény visszatérési értéke, és hogy milyen típusú paraméterei vannak. Legyen egy olyan függvényünk, amely nem kér semmilyen paramétert, és a visszatérési értéke int típusú. Az erre irányuló mutatót az alábbi módon adhatjuk meg:

```
int (*fnPtr) (void);
```

A `*fnPtr` köré kénytelenek vagyunk zárójelet írni, mert különben a fordítóprogram egy `int`-re irányuló mutatót visszaadó függvényként értelmezné a deklarációt. A függvényhívást jelző `()` operátor ugyanis magasabb kiértékelési szinten van, mint a „címe” `(*)` operátor.

Egy függvénymutatót úgy irányíthatunk rá egy függvényre, hogy egyszerűen értékül adjuk a nevét. Ha például a `lookup` egy paraméter nélküli, de egy `int` típusú számot visszaadó függvény, akkor az

```
fnPtr = lookup;
```

utasítás úgy állítja be az `fnPtr` mutatót, hogy az a `lookup`-ra mutasson. A zárójel nélkül megadott függvénynév hasonlít a szögletes zárójel (azaz `index`) nélkül megadott tömbnévhez. Ilyenkor a fordítóprogram automatikusan az adott függvényre (vagy tömbre) irányuló mutatót adja vissza. Lehet & jelet írni a függvény neve elő, de nem kötelező.

Ha előzőleg nem definiáltuk a `lookup` függvényt, akkor a fenti értékkadás előtt deklarálnunk kell, azaz egy

```
int lookup (void);
```

utasítással jeleznünk kell a függvény paraméter-szignatúráját, mielőtt átadhatnánk az `fnPtr`-nek.

A mutató által hivatkozott függvény meghívható a mutató nevére alkalmazott „függvény-hívó zárójel” operátorral (benne felsorolva a paramétereket). Például az

```
entry = fnPtr ();
```

utasítás meghívja az `fnPtr` által mutatott függvényt, melynek visszatérési értéke az `entry` változóba kerül.

A függvénymutatók egyik gyakori alkalmazása abban áll, hogy paraméterül adhatóak más függvényeknek. A szabványos C programkönyvtár is használ ilyen függvényt: a `qsort` gyorsrendező (*quicksort*) függvény például tömbelemeket tud rendezni. Ennek egyik paramétere egy mutató egy függvényre, amely minden alkalommal meghívódik, amikor két tömbeemet összehasonlít a `qsort`. Ily módon a `qsort`-tal bármilyen típusú tömb rendezhető. A sorrendet kialakító összehasonlítások ugyanis (a `qsort` helyett) a felhasználó által is megadható függvényvel történnek. A B függelék („A szabványos C programkönyvtár”) részleteiben tárgyalja a `qsort` függvényt – ott találunk egy példát is a felhasználására.

A függvénymutatók egy másik alkalmazását a szétküldő táblázatok (*dispatch tables*) jelentik. Függvényeket nem tárolhatunk tömbben, ám függvénymutatókat igen. Ily módon létrehozhatunk egy olyan táblázatot, amely a különböző esetekben meghívandó függvények mutatóit tárolja. Elkészíthető például egy olyan táblázat, amely a felhasználó által megadott parancsok alapján különböző függvényeket hív meg. Az egyes bejegyzések tárolhatják a megengedett parancsokat és a hozzájuk tartozó függvények mutatóját. A felhasználói parancs megadása után meg kell keresnünk a megfelelő bejegyzést, és meg kell hívni az oda tartozó függvényt.

Mutatók és memóriacímek

Mielőtt befejeznénk a C nyelv mutatóinak taglalását, beszélünk kell arról is, hogy miként történik a mutatók megvalósítása. A számítógép memóriája felfogható egymás után írt tárolócellák sorozataként. minden cellának van egy száma, azaz egy „*címe*” – ezek számozása általában a 0-tól kezdődik. Egy cellát többnyire egy bájt képvisel.

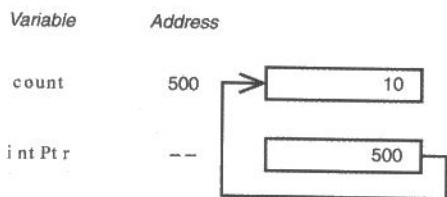
A számítógép memóriájában van eltárolva maga a futtatott program, és a változók értékei is. Amikor int típusúnak deklarálunk egy count nevű változót, akkor a rendszer lefoglalja a szükséges memóriacellát(ka)t, ahol a program futása során a count változó értéke tárolódni fog. Legyen ennek a memóriacellának a címe mondjuk 500.

A C programozási nyelv szerencsére kellően magas szintű ahhoz, hogy ne kelljen a programozónak azzal foglalkoznia, hogy melyik változóhoz milyen memóriacím tartozik. Ezeket automatikusan osztja ki a rendszer. Mégis, segít a mutatók megértésében, ha tudjuk, hogy minden változóhoz tartozik egy cím.

Amikor a „címe” operátort alkalmazzuk egy C változóra, akkor azt a memóriacímet kapjuk, amelyen az adott változó értéke őrződik a memóriában. (Innen van a „címe” operátor elnevezése is.) Így tehát, folytatva fenti példánkat, az

```
intPtr = &count;
```

utasítás azt a memóriacímet tölti be az intPtr-be, amin a count változó értéke őrződik. Ha a count változó az 500-as memóriacellában van tárolva (és értéke 10), akkor a fenti utasítás 500-at tölteni kell az intPtr-be. Ezt láthatjuk a 11.11 ábrán.



11.11 ábra

Változók és memóriacímek

Az intPtr címe az ábrán -- jellettel van jelölve, ennek értéke ugyanis érdektelen a példa szempontjából.

Maga az intPtr tehát egy memóriacímet tartalmaz. Hogyan tudjuk ezt felhasználni? A közvetett elérés operátort alkalmazva egy mutatóra (például a *intPtr kifejezéssel) megkapjuk annak a memóriacellának a tartalmát, amire az intPtr mutat. Ezt olyan típusú változóként értelmezi a rendszer, amilyen típusúként a mutatót meghatároztuk. Ha az intPtr int típusú mutató, akkor a *intPtr által kiolvasott értéket egész számként értelmezzük. Konkrét példánkban az 500-as memóriacellában tárolt értéket olvassuk ki, és egész számként értelmezzük. Ennek eredményeként (int típusú) 10-et kapunk.

Hasonlóképp tölhetünk be értékeket a mutatókkal hivatkozott memóriacímre, például a

```
*intPtr = 20;
```

utasítással. Az `intPtr`-ben tárolt értéket memóriacímként tekinti a rendszer. A megadott egész számot ezen a címen tároljuk el. A fenti utasítás hatására tehát az 500-as memória-cellá tartalma 20 lesz.

Vannak esetek, amikor a programozóknak konkrét memóriacellákat kell elérniük. Ilyenkor különösen hasznos, ha ismerjük a mutatók működését.

Fejezetünkben bizonyára kiderült, hogy a mutatók a C nyelv egyik leghasznosabb eszköz-készletét képviselik. A mutatók definiálási lehetőségei még a fejezetünkben vázoltakat is felülmúlják. Megadhatunk mutatóakra irányuló mutatókat, sőt mutatókra irányuló mutatókra irányuló mutatókat is stb. Ezek már túlmutatnak könyünk keretein, bár igazából azt a logikát viszik tovább, amiről itt is szó volt.

Minden bizonnal a mutatók jelentik a legnagyobb kihívást a C nyelvvel ismerkedők számára. Érdemes újraolvasni minden olyan részt, ami bizonytalanságot hagyott maga után. A gyakorlatok megoldása ugyancsak sokat segíthet az olvasottak elmélyítésében.

Gyakorlatok

1. Gépeljük be és futtassuk a fejezetben szereplő tizenöt példaprogramot! A kimeneteket hasonlítsuk össze a könyvben szereplő programkimenetekkel.
2. Írjuk meg az `insertEntry` függvényt, amely egy új elemet illeszt be egy láncolt listába! Legyen a függvénynek két paramétere. Az egyik legyen a beszúrandó elemre irányuló mutató, melynek típusa a fejezetben vázolt „entry adatszerkezetű” legyen. A másik pedig annak a listaelemnek a mutatója legyen, amely *után* szeretnék beilleszteni az új elemet.
3. Az előző gyakorlat beszúró függvénye csak olyan elemek beszúrását teszi lehetővé, amely előtt már van listaelem (vagyis új kezdőelemet nem tudunk megadni). Hogyan tudnánk a függvényt úgy megjavítani, hogy ne kelljen ezzel a megszorítással élnünk? (*Ötlet:* van egy olyan adatszerkezet, amely a lista elejére mutat.)
4. Írjuk meg a `removeEntry` függvényt, amely eltávolítja egy láncolt lista adott elemét! Egyetlen paramétere van csak szükség: egy listamutatóra. A függvény távolítsa el a mutatott listaelementet *követő* elemet. (Miért nem lehet magát a mutatott elemet eltávolítani?) Az előző gyakorlatban is említett sajátos adatszerkezet használatára lesz szükség ahoz, hogy a lista kezdőelemét is el lehessen távolítani.
5. Kétszeresen láncolt listának hívjuk azt a struktúrát, melyben a listaelemek nemcsak a következő elemre irányuló mutatót tartalmazzák, hanem az őket megelőző elemre is rátámasztanak. Adjunk meg olyan adatszerkezet-definíciót, mely egy kétszeresen láncolt lista elemének típusát hozza létre. Írunk programot, amely előállít egy kissébőkétszeresen láncolt listát, és kiírja az elemeit.

6. Írjuk meg az `insertEntry` és a `removeEntry` függvényeket, amelyek a 2-3-4. gyakorlatban szereplő feladatokat látják el egy kétszeresen láncolt lista esetében!
A `removeEntry` függvény azonban ebben az esetben már el tudja távolítani azt a listaelemet is, amire a paraméter mutat. Mi ennek az oka?
7. A 8. fejezetben („Függvények”) találkoztunk egy `sort` nevű függvénnel. Írjuk meg ennek egy olyan változatát, amely mutatókat használ! Küszöböljük ki teljesen a tömbök használatát, és mindenhol (ciklusokban, függvényekben) használjunk kizárolag mutatókat!
8. Írjuk meg a `sort3` függvényt, amely három egész számot nagyság szerinti sorrendbe rendez! (Oldjuk meg a feladatot tömb használata nélkül.)
9. Írjuk újra a 10. fejezet `readLine` függvényét oly módon, hogy karaktertömb helyett használjunk karakter-mutatót!
10. Írjuk újra a 10. fejezet `compareStrings` függvényét oly módon, hogy karaktertömbök helyett használjunk karakter-mutatókat!
11. A fejezetben található `date` adatszerkezet definíciójának felhasználásával írunk `dateUpdate` néven olyan egy függvényt, amely egy `date` adatszerkezetre irányuló mutatót vár paraméterként, és ami a dátumot a másnapra dátumra állítja át (lásd a 9.4 Listát).
12. Az alábbi definíciók alapján:

```
char *message = "Programming in C is fun\n";
char message2[] = "You said it\n";
char *format = "x = %i\n";
int x = 100;
```

döntsük el, hogy az alábbi `printf`-sorozatok működnek-e, és csoporthonként ugyanazt a kimenetet adják-e:

```
/** 1. csoport */
printf ("Programming in C is fun\n");
printf ("%s", "Programming in C is fun\n");
printf ("%s", message);
printf (message);

/** 2. csoport */
printf ("You said it\n");
printf ("%s", message2);
printf (message2);
printf ("%s", &message2[0]);

/** 3. csoport */
printf ("said it\n");
printf (message2 + 4);
printf ("%s", message2 + 4);
printf ("%s", &message2[4]);

/** 4. csoport */
printf ("x = %i\n", x);
printf (format, x);
```

12

Bitműveletek

A könyv elején szó volt arról, hogy a C programozási nyelvet alapvetően rendszerprogramok megírásához fejlesztették ki. Ezt bizonyítja a mutatók kifinomult felhasználási lehetősége, melynek révén a programozók kedvükre tevékenykedhetnek a számítógép memóriájában. A rendszerprogramozóknak ugyanilyen okokból gyakran kell „biteket buherálniuk” – a C nyelvben ehhez is komoly támogatást találunk különféle operátorok formájában.

Gondoljunk vissza arra, ami az előző fejezet végén szóba került: a memória felosztására, a bájtok szerepére. A legtöbb rendszeren egy bájt nyolc bitből (elemi egységből) áll. Egy bitnek két értéke lehet: 0 vagy 1. Az 1000-es memóriacímen eltárolható például a következő nyolcbites szám:

01100100

Célszerű ezt a karakterláncot kettes számrendszerbeli (bináris) egész számként felfogni. A jobboldali a legkisebb helyiértékű bit (*Least Significant Bit, LSB*), míg a baloldali a legnagyobb helyiértékű bit (*Most Significant Bit, MSB*). Azaz a jobboldali bit megfelel 2^0 -nak, vagyis 1-nek; a tőle balra álló a 2^1 , vagyis 2, az ezt követő a $2^2 = 4$ stb. A fenti bináris szám esetén adjuk össze az 1-essel jelzett helyiértékeket. Az eredmény: $2^2 + 2^5 + 2^6 = 4 + 32 + 64 = 100$, tízes számrendszerben kifejezve.

A negatív számok tárolása egy kissé körmö;font módon történik. A legtöbb számítógépen ezeket a számokat „kettes komplementben” ábrázolják. Ez abból áll, hogy a baloldali bitet kinevezzük előjelbitnek. Ha az előjel-bit 0, akkor a szám pozitív (mint a fenti példán is látuk), ha pedig 1, akkor a szám negatív. A negatív számok azonban nem a többi bit közvetlen helyiérték-összegéből állnak össze. Kettes komplementben a -1 úgy áll elő, ahogy a bitek a 00000000-ból 1-et kivonva kialakulnak: csupa 1-es áll minden helyiértéken:

11111111

Egy kettes komplementként ábrázolandó negatív szám értéke úgy számítható ki a tízes számrendszerbeli alakból, hogy 1-et hozzáadtunk a számhoz, kifejezzük a szám abszolút

értékét binárisan, majd minden számjegyet invertálunk: 0 helyett 1-et, 1 helyett 0-t írunk az egyes helyértékekre. Például vegyük a -5-öt. 1-et hozzáadva -4 adódik, ennek abszolút értéke 4, ami binárisan 00000100. Ennek invertált alakja 11111011.

Ugyanez a módszer fordítva is működik: egy kettes komplementensbeli bináris negatív számot úgy számíthatunk át tízes számrendszerbe, hogy először invertáljuk a számjegyeket, kiszámítjuk a tízes számrendszerbeli alakot, ellenetettjét vesszük, majd kivonunk egyet az értékből.

Kettes komplementens ábrázolásmóddal az n biten ábrázolható legnagyobb pozitív szám a $2^{n-1} - 1$, azaz nyolc biten $2^7 - 1 = 127$. A legkisebb negatív szám pedig a -2^{n-1} , azaz nyolc biten $-2^7 = -128$ (Vajon miért nem ugyanannyi a legkisebb és legnagyobb ábrázolható szám abszolút értéke?)

A mai processzorok az egész számokat négy bájton (32 biten) ábrázolják, és ennyi helyet foglalnak le számukra a memóriában is. A kifejezhető legnagyobb egész szám tehát a $2^{32-1} - 1 = 2^{31} - 1 = 2147483647$, míg a legkisebb ábrázolható egész szám a $-2^{32} = -2147483648$.

A 4. fejezetben szó volt az unsigned típusmódosítóról. Tanultuk, hogy felhasználható arra is, hogy az ábrázolandó számtartományt átrendezzük, azaz (annak rovására, hogy nem ábrázolunk negatív számokat) kétszer annyi pozitív szám férhet bele a rendelkezésre álló memóriatartományba. Ezt úgy érjük el, hogy a szükségtelenné vált előjel-biten is értékes számjegyet tárolunk. Pontosabban fogalmazva, előjel nélküli egész számok esetén n biten $2^n - 1$ -félé számot tudunk tárolni. Az olyan számítógépeken, amelyek 32 biten ábrázolják az egész számokat, 0-tól 4294967296-ig terjed a tárolható egész számok tartománya.

Bitenkénti operátorok

A bevezető gondolatok után vizsgáljuk meg, milyen bitenkénti operátorok vannak a C nyelvben:

12.1 Táblázat • Bitenkénti operátorok

Szimbólum	Művelet
&	Bitenkénti ÉS
	Bitenkénti VAGY
^	Bitenkénti KIZÁRÓ VAGY
~	Bitenkénti negáció
<<	Eltolás balra
>>	Eltolás jobbra

A negáció kivételével a 12.1 Táblázat valamennyi operátora kétoperandusú. Bitenkénti műveletek valamennyi karakteres és egész típusú változóval végezhetőek, függetlenül a típusmódosítótól (short, long, long long, signed vagy unsigned); lebegőpontos számokkal azonban nem dolgozhatunk.

A bitenkénti ÉS művelet

Amikor két értéket bitenkénti ÉS-sel (`&`) kapcsolunk össze, akkor az értékek bináris formáját hasonlítjuk össze bitenként. Az összehasonlítás eredménye csak akkor 1-es, ha minden összehasonlított bit 1-es értékű; minden más esetben 0. Legyen b_1 és b_2 a két összehasonlított bit. A bitenkénti ÉS művelet lehetséges kimeneteleit igazságáblázatban foglalhatjuk össze:

b_1	b_2	$b_1 \& b_2$
0	0	0
0	1	0
1	0	0
1	1	1

Ha például w_1 és w_2 short int típusú egész számok, w_1 értéke 25, w_2 pedig 77, akkor a $w_3 = w_1 \& w_2;$

utasítás eredményeként 9 kerül a w_3 változóba. A műveletet akkor értjük meg, ha binárisan (kettes számrendszerben) nézzük meg az egyes értékeket. Tegyük fel, hogy a short int 16 bites számokat takar.

w_1	00000000000011001	25
w_2	000000001001101	$\&$ 77
w_3	00000000000010001	9

Gondoljunk vissza arra, hogy miként működik a logikai „és” (`&&`) művelet. A két kifejezésből előálló harmadik csak akkor igaz, ha minden összehasonlított bit 1-es értékű. Hasonlóképp működik a bitenkénti ÉS is. Vigyázzunk, hogy ne keverjük össze a kétféle operátort! A logikai „és” operandusai logikai értékek, és eredménye is egy igaz/hamis érték – ez esetben nincs szó bitenkénti műveletekről.

A bitenkénti ÉS legtöbbször maszkolási feladatot lát el, azaz lenullázhatunk vele néhány bitet. A

$w_3 = w_1 \& 3;$

utasítás w3-nak értékül adja w1 és 3 bitenkénti ÉS-sel vett összehasonlítását. Ezzel w1 két jobboldali bitjén kívül minden más bitet lenullázzunk, ám w1 két jobboldali bitjét megőrizzük eredeti állapotában.

A többi kétoperandusú művelethez hasonlóan a bitműveleteket is egybevonhatjuk egy értékkadással. A

```
word &= 15;
```

utasítás ugyanazt a műveletet végzi el, mint a

```
word = word & 15;
```

értékkadás, nevezetesen a négy legalacsonyabb biten kívül lenullázza a word bitjeit.

Ha konstansokat használunk a bitenkénti műveletekhez, akkor érdemes a számokat oktálist vagy hexadecimális formában megadni. A választás attól is függ, hogy mekkora a szám. Ha 32 bites számítógépen dolgozunk, akkor igen kényelmes a hexadecimális formátum, ugyanis $32 = 4 \cdot 8$, így nyolc hexadecimális „számjeggyel” megadható egy egész számérték. (Egy hexadecimális „számjegy” ugyanis négy bitet fed le.)

A 12.1 Lista a bitenkénti ÉS-t mutatja be. Mivel a programban csak pozitív számok szerepelnek, az egész számok unsigned int-ként vannak deklarálva.

12.1 Lista • A bitenkénti ÉS művelet

```
// A bitenkénti ÉS-t bemutató program
#include <stdio.h>

int main (void)
{
    unsigned int word1 = 077u, word2 = 0150u, word3 = 0210u;

    printf ("%o ", word1 & word2);
    printf ("%o ", word1 & word1);
    printf ("%o ", word1 & word2 & word3);
    printf ("%o\n", word1 & 1);

    return 0;
}
```

12.1 Lista • Kimenet

50 77 10 1

Emlékeztetőül: ha egy egész konstans 0-val kezdődik, akkor a szám oktális (nyolcas számrendszerbeli) jelölésmód szerint értelmeződik a C nyelvben. Így a három unsigned int típusú szám (word1, word2 és word3) oktálisan inicializálódik: 077, 0150 és 0210 kezdőértékekkel. A 4. fejezetben volt szó arról is, hogy ha egy egész konstans után u-t vagy U-t írunk, akkor azt előjel nélküliként (unsigned) értelmezi a fordítóprogram.

A kimenet első számértéke az oktális 50 („öt-nulla”), ami a word1 és a word2 bitenkénti ÉS-sel való összehasonlításának eredménye. A program a következőképpen számolt:

word1	...	000	111	111	077
word2	...	001	101	000	& 0150

		... 000	101	000	050

Csak a 9 legalacsonyabb helyiértéket nézzük, mert a többi helyen 0 áll. Hármas csoporthoz állítottuk a biteket, mert így könnyebb átváltani egymásba a nyolcas és kettes számrendszerbeli számokat.

A második kiírt szám az oktális 77, ami a word1 önmagával vett bitenkénti ÉS-műveletének az eredménye. Ez a művelet minden szám esetén magát a számot adja eredményül (hisz a 0-k és az 1-esek is megmaradnak).

A harmadik kiírt számot a word1, word2 és word3 ÉS-sel való összekapcsolásával kaptuk. A bitenkénti ÉS művelet asszociatív, azaz ugyanazt az eredményt adja a & b & c esetén, mint (a & b) & c esetén, illetve, mint az a & (b & c) zárójelezéssel. Ha nem adunk meg zárójelet, balról jobbra történik a kiértékelés. Legyen az olvasó feladata ellenőrizni, hogy a három szám ÉS műveettel történő összekapcsolása valóban a kimenetként kapott oktális 10 értéket adja-e.

Az utolsó printf hívás a word1 legalacsonyabb helyiértékű bitjét írja ki. Ezzel a paritásvizsgálatnak egy újabb módját ismertük meg. Páratlan számok esetén ugyanis ez a bit 1-es, páros számok esetén pedig 0. Így az

```
if      ( word1 & 1 )
...

```

feltételvizsgálat páratlan word1 esetén igazként értékelődik ki, páros word1 esetén pedig hamisként. (Vigyázzunk arra, hogy a kettes komplement használó számábrázolás esetén a *negatív számok paritását nem vizsgálhatjuk így* – ott éppen a páros számokra igaz a kifejezés.)

A bitenkénti VAGY művelet

Amikor a C nyelvben két értéket bitenkénti VAGY-gyal (`|`) kapcsolunk össze, akkor az értékek bináris reprezentációját hasonlítuk össze bitenként. Az összehasonlítás eredménye csak akkor 0, ha minden összehasonlított bit 0 értékű; minden más esetben 1-es. Tekintük meg a bitenkénti VAGY művelet igazságátblázatát:

b1	b2	b1 b2
0	0	0
0	1	1
1	0	1
1	1	1

Legyen w1 és w2 két short int típusú egész szám; w1 értéke legyen oktálisan 0431, w2-é pedig 0152. A

w3 = w1 | w2;

bitenkénti VAGY művelet utáni értékadás eredményeként 0573 kerül a w3 változóba, a következő számítás alapján:

w1 ... 100 011 001	0431
w2 ... 001 101 010	0152

w3 ... 101 111 011	0573

Ahogy arról a bitenkénti ÉS műveletnél is szó volt, nem tévesztendő össze a bitenkénti VAGY (`|`) a logikai *vagy* művelettel (`||`). Ez utóbbi két logikai érték összehasonlítását végzi el (és akkor ad igaz értéket, ha valamelyik igaz a kettő közül).

A bitenkénti VAGY-ot (vagy tükörfordítással: a „bitenkénti beleértő VAGY”-ot) többnyire arra használjuk, hogy egy érték bizonyos bitjeit egyesre állítsuk. A

w1 = w1 | 07;

utasítás w1 három legalacsonyabb helyiértékű bitjét állítja 1-re (függetlenül azok korábbi értékétől). Természetesen itt is egyesíthető az operátor az értékadó operátorral, azaz a fenti utasítás

w1 |= 07;

formában is megadható.

A bitenkénti VAGY műveletet illusztráló programot a fejezet egy későbbi szakaszán vizsgáljuk majd meg, más műveletekkel összehasonlítva.

A bitenkénti KIZÁRÓ VAGY művelet

A bitenkénti KIZÁRÓ VAGY (\wedge) művelet akkor ad 1-es értéket két bit összehasonlításakor, ha a két bit közül csak az egyik 1-es. Tekintsük meg a bitenkénti KIZÁRÓ VAGY művelet igazságítáblázatát:

b1	b2	b1	\wedge	b2
0	0	0		
0	1	1		
1	0	1		
1	1	0		

Állítsuk be w1 és w2 (short int) számok értékét 0536-ra és 0266-ra. A

w3 = w1 \wedge w2;

bitenkénti KIZÁRÓ VAGY művelet után 0750 kerül a w3-ba, a következőképpen:

w1	...	101 011 110	0536
w2	...	010 110 110	\wedge 0266
w3	...	111 101 000	0750

Érdekes tulajdonsága a KIZÁRÓ VAGY-nak, hogy egy számot önmagával összekapcsolva 0-t kapunk (hisz két 0 és két 1-es is 0-t ad). Régebbi assembly programokban találkoztunk olyan kódrészletekkel, melyben ezzel a módszerrel nulláznak egy értéket, vagy így vizsgálják meg két szám egyenlőségét. Nem ajánlatos azonban ily módon homályba borítani egy C programot, mert semmi időt sem spórolunk meg vele.

Egy másik érdekessége a KIZÁRÓ VAGY-nak, hogy segítségével két egész számértéket úgy tudunk felcserélni, hogy ahhoz nem kell harmadik változót (és hozzá memóriaterületet) lefoglalnunk. Bizonyára emlékszik az olvasó, hogy i1 és i2 értékét eddig programjainkban egy segédváltozó használatával tudtuk felcserélni:

```
temp = i1;
i1 = i2;
i2 = temp;
```

Ezzel szemben (egész számok felcserélésekor) a KIZÁRÓ VAGY alkalmazásával az alábbi három értékkadás is elegendő, segédváltozó használata nélkül:

```
i1 ^= i2;
i2 ^= i1;
i1 ^= i2;
```

Az olvasó feladata ellenőrizni, hogy ez a kódrészlet helyesen működik-e.

A bitenkénti negáció művelete

A bitenkénti negáció (~) egyoperandusú operátor; egyszerűen átállítja az operandus bitjeit. Az összes 1-es 0 lesz, és az összes 0 1-es lesz a művelet végrehajtása után. A teljesség kedvéért ennek a műveletnek az igazságátablázatát is közöljük:

b1	$\sim b1$
0	1
1	0

Ha b1 egy 16 bites short int típusú szám, melynek oktalis értéke 0122457, akkor ennek negáltja az oktalis 0055320 értéket adja.

```
w1 1 010 010 100 101 111 0122457
~w1 0 101 101 011 010 000 0055320
```

A bitenkénti negáció operátora (~) nem tévesztendő össze az ellentett (-) operátorral, sem pedig a logikai tagadással (!). Ha az (int típusú) w1 értéke 0, akkor $\sim w1$ szintén 0. Ha a bitenkénti negációt alkalmazzuk w1-re, akkor minden biten 1-es lesz, ami kettes komplementben -1. Ha pedig a logikai tagadást alkalmazzuk w1-re, akkor *igaz* értéket kapunk, azaz 1-et (hiszen a 0 a *hamisnak* felel meg).

A bitenkénti negációt felhasználhatjuk a bitek számának felderítésére. Ez sokat segíthet programunk hordozhatóvá téTELÉBEN (azaz nem csak a konkrétan használt számítógép-architektúrán fog futni, hanem könnyebben lefordítható más típusú gépekre is).

Tegyük fel, hogy szeretnénk 0-ra állítani egy szám legalacsonyabb helyiértékű bitjét. Ehhez bitenkénti ÉS kapcsolatba kell hozni magát a számot egy olyan bitsorozattal, melynek minden eleme 1-es, csak az utolsó 0. Ez 32 bites gépeken nagyon egyszerűen megoldható az alábbi utasítással:

```
w1 &= 0xFFFFFFFF;
```

Célszerűbb azonban így megfogalmaznunk a programkódot:

```
w1 &= ~1;
```

Így w1 legalacsonyabb helyiértékű bitje minden géptípuson nullázódik. Az 1-re elvégzett bitenkénti negáció hatására ugyanis olyan bitsorozat jön létre, mely csupa 1-esekből áll, kivéve az utolsó 0-t. (például 32 bit esetén 31 db 1-es és egy 0 áll elő.)

A 12.2 Lista összefoglalja az eddig tanult bitenkénti műveleteket. Mielőtt megnéznénk magát a programot, érdemes néhány percert szánnunk a kiértékelési sorrendre. Az ÉS, VAGY és KIZÁRÓ VAGY műveletek mind alacsonyabb precedenciájúak, mint az aritmetikai műveletek vagy a relációs operátorok, de magasabbak a logikai „és”, „vagy” műveleteknél. A bitenkénti ÉS magasabb precedenciájú a KIZÁRÓ VAGY-nál, ez pedig erősebb a VAGY operátornál. A bitenkénti műveletek közül a legerősebb a negáció – ennek precedenciája magasabb minden kétoperandusú operátornál. Az operátorok kiértékelési sorrendjével kapcsolatban az A függelékben érdemes további részletekről tájékozódni („A C nyelv összefoglalása”).

12.2 Lista • Bitenkénti operátorok

```
/* Bitenkénti műveleteket bemutató programrészlet */

#include <stdio.h>

int main (void)
{
    unsigned int w1 = 0525u, w2 = 0707u, w3 = 0122u;

    printf ("%o %o\n", w1 & w2, w1 | w2, w1 ^ w2);
    printf ("%o %o\n", ~w1, ~w2, ~w3);
    printf ("%o %o\n", w1 ^ w1, w1 & ~w2, w1 | w2 | w3);
    printf ("%o %o\n", w1 | w2 & w3, w1 | w2 & ~w3);
    printf ("%o %o\n", ~(~w1 & ~w2), ~(~w1 | ~w2));

    w1 ^= w2;
    w2 ^= w1;
    w1 ^= w2;
    printf ("w1 = %o, w2 = %o\n", w1, w2);

    return 0;
}
```

12.2 Lista • Kimenet

```
505 727 222
37777777252 37777777070 37777777655
0 20 727
527 725
727 505
w1 = 707, w2 = 525
```

A fenti kódrészlet minden műveletét érdemes papírral és ceruzával ellenőrizni, hogy meggyőződjünk arról, megértettük-e a különféle operátorok működését. A program olyan számítógépen futott, amely az int típusú számokat 32 biten ábrázolja.

A negyedik `printf` utasítás értelmezésénél tartsuk szem előtt, hogy a bitenkénti ÉS korábban kiértékelődik, mint a bitenkénti VAGY. Ez nagyban befolyásolja a művelet eredményét.

Az ötödik `printf` hívás a De Morgan azonosságot szemlélteti: $\sim(\sim a \& \sim b)$ ugyanazt adja, mint az „ $a \mid b$ ”, valamint $\sim(\sim a \mid \sim b)$ egyenlő „ $a \& b$ ”-vel. A program befejező része bemutatja, hogy a bitenkénti KIZÁRÓ VAGY használatával segédváltozó nélkül is megcserélhető két egész érték.

Bitenkénti eltolás balra

Amikor a „bitenkénti eltolás balra” (`<<`) műveletet hajtjuk végre, szó szerint az történik, amit az operátor neve mond: a bitek eggyel balra tolódnak, jobbról (a legalacsonyabb helyiértékről) pedig csupa 0 tölti fel a megüresedett helyeket. A baloldalon túlcorduló bitek elvesznek.

Nézzünk egy konkrét példát. Legyen `w1` értéke 3. A

```
w1 = w1 << 1;
```

művelet, amely így is írható:

```
w1 <= 1;
```

eggyel balra tolja a biteket, ami 6-ot eredményez:

<code>w1</code>	...	000	011	03
<code>w1 << 1</code>	...	000	110	06

A `<<` operátor baloldalán áll az a változó, amelyen az eltolást végrehajtjuk, jobboldalt pedig az eltolás pozícióinak száma (itt ez 1).

Ha az előbb kapott értéken még egyszer végrehajtunk egy bitenkénti balra tolást, akkor a `w1` változóba az oktális 014 kerül:

<code>w1</code>	...	000	110	06
<code>w1 << 1</code>	...	001	100	014

A balra tolás művelete igazából kettővel való szorzásnak felel meg. Sok fordítóprogram a kettővel szorzást is bitenkénti balra tolással valósítja meg, mert ez a legtöbb számítógépen sokkal gyorsabb művelet, mint a szorzás.

A bitenkénti eltolást bemutató példaprogramot a jobbra való eltolás bemutatása után tekinthetik meg.

Bitenkénti eltolás jobbra

Amint a nevéről sejthető, a „bitenkénti eltolás jobbra” (>>) művelet a biteket egygyel jobbra tolja. A jobboldalt túlcorduló bitek elvesznek. Előjel nélküli (`unsigned`) egészek esetén balról (a legmagasabb helyiértékről) csupa 0 tölti fel a megüresedett helyeket. Előjeles egészek esetén a balról beérkező bitek függnek az eredeti szám előjelétől. Ha az előjel-bit (azaz a legmagasabb helyiértékű bit) 0, akkor 0 érkezik balról. Ha azonban 1 (azaz negatív a kiindulási szám), akkor gépfüggő, hogy 0 vagy 1 érkezik-e a megüresedő helyre. Ha 1 érkezik, akkor ezt a fajta eltolást „aritmetikai eltolásnak” hívjuk, ha pedig 0 érkezik, akkor „logikai eltolásnak” nevezzük.

Lehetőleg ne tételezzük föl a rendszerről, hogy aritmetikai vagy logikai eltolást használ. Az előjeles számokat jobbra toló program ettől az előfeltevééstől függően jól működhet az egyik gépen, ám meghiúsulhat egy másikon.

Legyen `w1` egy 32 biten ábrázolt `unsigned int` típusú szám, melynek értéke hexadecimálisan `F777EE22`. Toljuk el a szám bitjeit jobbra a következő utasítással:

```
w1 >>= 1;
```

Ennek hatására `w1` értéke `7BBBF711` lesz.

```
w1      1111 0111 0111 0111 1110 1110 0010 0010 F777EE22
w1 >> 1 0111 1011 1011 1111 0111 0001 0001 7BBBF711
```

Ha viszont `w1` előjeles egész szám, akkor néhány számítógépen ugyanezt az értéket kapjuk, másokon viszont (aritmetikai eltolásként) `FBBBBF711` lesz a művelet eredménye.

Jó tudni, hogy a C nyelv nem definiálja az olyan eltolási műveletek végeredményét, melyben a lépések száma nagyobb, mint ahány bitből az operandus áll. Például 32 biten ábrázolt egész számok esetén 32 vagy több bitnyi eltolás után (akár jobbra, akár balra történik) nem várhatunk jól definiált végeredményt. Ugyanez a helyzet akkor, ha negatív számot adunk meg eltolásként.

Egy saját eltolási függvény

Ideje munkába állítanunk az eltolási operátorokat – ezt tesszük meg a 12.3 Listában. Van-nak olyan számítógépek, melynek gépi utasításai között szerepel egy olyan eltolás, amely balra léptet, ha a megadott lépésszám pozitív, és jobbra, ha negatív. Írunk egy olyan függvényt, amely elvégzi ezt a feladatot! Két paramétere legyen: az eltolandó érték és a léptetések (előjeles) száma. Ha ez utóbbi pozitív, akkor történjen bitenkénti eltolás balra (a megadott számmal), ha pedig negatív, akkor a megadott szám abszolút értékével toljuk el a másik paraméter értékét jobbra.

12.3 Lista • Saját eltolási függvény megvalósítása

```
// Eltolási függvény, amely balra léptet, ha a megadott
// lépésszám pozitív, és jobbra, ha negatív.

#include <stdio.h>

unsigned int shift (unsigned int value, int n)
{
    if ( n > 0 )          // left shift
        value <<= n;
    else                  // right shift
        value >>= -n;

    return value;
}

int main (void)
{
    unsigned int w1 = 0177777u, w2 = 0444u;
    unsigned int shift (unsigned int value, int n);

    printf ("%o\t%o\n", shift (w1, 5), w1 << 5);
    printf ("%o\t%o\n", shift (w1, -6), w1 >> 6);
    printf ("%o\t%o\n", shift (w2, 0), w2 >> 0);
    printf ("%o\n", shift (shift (w1, -3), 3));

    return 0;
}
```

12.3 Lista • Kimenet

```
7777740 7777740
1777    1777
444     444
177770
```

A 12.3 Listában megvalósított shift függvény a value paramétert előjel nélküli egészketént deklarálja, így a jobbra léptetés mindenkorban nullákkal való feltöltést eredményez (vagyis logikai eltolás történik). Ha n értéke (az eltolás-számláló) pozitív, akkor a value értékét balra toljuk el n bittel. Ha n negatív (vagy nulla), akkor -n bittel (ami n abszolút értéke) jobbra toljuk a value-t.

A shift függvény első meghívásában w1 értékét öt bittel toljuk el balra. Az eredményt ki-jelző printf hívásban azt is láthatjuk, hogy a natív „öt bittel balra tolás” mit eredményez, így ellenőrizhető függvényünk helyes működése.

A shift függvény második meghívásakor w1 értékét (-6 bittel balra, azaz) hat bittel toljuk el jobbra. A kapott eredmény a kimenet szerint megegyezik azzal, amit a közvetlen „hat bittel jobbra tolás” eredményez.

A harmadik shift híváskor nulla eltolást kérünk, aminek ugyanúgy nincs léptető hatása (a kimenet szerint), mint egy nulla eltolásnak jobbra.

Az utolsó printf-ben egymásba ágyazottan hívjuk meg kétszer a shift függvényt. Először a belső shift fut le, ami w1 értékét jobbra tolja három bittel. Az eredményként kapott 0017777 újra átadódik a shift függvénynek, ami ekkor balra tolja az értéket három bittel. A kimenetből látható, hogy ennek az lesz a hatása, hogy a három legalacsonyabb helyiértékű bitbe 0 kerül. (Ez a művelet egyszerűbben is megvalósítható egy ~7-tel történő bitenkénti ÉS művelettes.)

Bitek körbeforgatása

Következő példaprogramunkban az eddig tanult bitműveletek segítségével megírunk egy olyan függvényt, amely egy értéket bitenként „körbeforgat” (*rotate*) jobbra vagy balra. Ez a művelet hasonlít a bitenkénti eltolásra, azzal a különbséggel, hogy az „eltűnő” bitek bejönnek a szám másik oldalán. Ha 32 bites egészekkel dolgozunk, akkor a 80000000 hexadecimális szám balra forgatásának eredménye 00000001 lesz, mivel a legnagyobb helyiértékű (előjel-)bit, ami itt 1-es (és ami elveszne egy balra toláskor), átkerül a szám túloldalára, a legkisebb helyiértékre.

A függvénynek legyen két paramétere: az egyik a bitenként körbeforgatandó érték, a másik pedig a forgatási lépések száma. Ha ez utóbbi pozitív, akkor történjen balra forgatás, egyébként pedig forgassunk jobbra.

A műveletet érdemes három részre bontanunk. Egy x int érték n bites balra forgatása (ahol n nemnegatív, de kisebb, mint x bitjeinek száma) azt jelenti, hogy eltároljuk x baloldali n bitjét, végzünk n bitnyi léptetést balra, majd az eltárolt biteket beillesztjük x jobboldali bitjeire. Ugyanilyen elven történhet a jobbra forgatás is.

A 12.4 Lista ennek megfelelően valósítja meg a rotate függvényt. Feltesszük, hogy számítógépünk 32 biten tárolja az int értékeit. A fejezet végén található egyik gyakorlat megoldásával kiküszöbölnétek ezt a zavaró előfeltevést.

12.4 Lista • Bitek körbeforgatását megvalósító függvény

// Egy egész szám bitjeinek körbeforgatása

```
#include <stdio.h>

int main (void)
{
```

```

        unsigned int w1 = 0xabcdef00u, w2 = 0xfffff1122u;
        unsigned int rotate (unsigned int value, int n);

        printf ("%x\n", rotate (w1, 8));
        printf ("%x\n", rotate (w1, -16));
        printf ("%x\n", rotate (w2, 4));
        printf ("%x\n", rotate (w2, -2));
        printf ("%x\n", rotate (w1, 0));
        printf ("%x\n", rotate (w1, 44));

        return 0;
    }

// Előjel nélküli egész szám elforgatását végző függvény

unsigned int rotate (unsigned int      value, int n)
{
    unsigned int result, bits;

    // A túl nagy eltolási érték lefaragása a szükséges mértékre

    if ( n > 0 )
        n = n % 32;
    else
        n = -(-n % 32);

    if ( n == 0 )
        result = value;
    else if ( n > 0 ) {      // balra forgatás
        bits = value >> (32 - n);
        result = value << n | bits;
    }
    else {                  // jobbra forgatás
        n = -n;
        bits = value << (32 - n);
        result = value >> n | bits;
    }

    return result;
}

```

12.4 Lista • Kimenet

cdef00ab
ef00abcd
fffff1122f
bffffc448
abcdef00
def00abc

A függvény először megnézi, érvényes lépésszámot (n -et) adtunk-e meg. Az

```
if (n>0)
    n = n % 32;
else
    n = -(-n % 32);
```

kódrészletben megvizsgáljuk, hogy n pozitív-e. Ha igen, akkor az n helyett csak azt az osztási maradékot tároljuk el n -ben, ami a (32-nek feltételezett) int típus bitméretével osztva keletkezik. (32 többszöröseivel való elforgatás a helybenhagyással egyenértékű, ezért azoktól eltekinthetünk.) Ily módon 0 és 31 közé szorul a léptetések száma.

Ha n negatív vagy nulla, akkor a maradék kiszámítása előtt az n ellentettjét vesszük (ezzel pozitív vagy nulla lesz). Negatív számokra ugyanis nincs definiálva a $\%$ operátor – gépfügő, hogy milyen előjelű lenne úgy az eredmény. A modulus helyes előjelét utólag korrigáljuk, így az -31 és 0 közé fog esni.

Ha az eltolás értéke 0, akkor azonnal megvan az eredmény: a függvény magát a value értéket tölti a visszaadandó result változóba. Egyébként pedig folytatódik a függvény futása.

A balra történő n -bites körbeforgatás három részműveletre bontható. Először a value baloldali n bitjét szeretnénk megőrizni, mégpedig egy egész típus méretének jobboldali n bitjén. Ehhez (egy bits nevű változóba töltve) jobbra toljuk a value bitjeit 32-n bittel; 32 képviseli az aktuális int-méretet. Így a bits összes többi bitje 0 lesz, csak a jobboldali n biten lesz a value baloldali n bitje. Ezután magát a value-t toljuk el n bittel balra. Utolsó lépésben pedig egy bitenkénti VAGY műveettel összefűsüljük a bits és a(z) eltolt) value bitjeit. Ugyanez a lépéssor (az irányok ellentétes értelmezésével) megvalósítható a jobbra forgatáskor is.

Figyeljük meg a main eljárásban a hexadecimális jelölésmódot. A rotate függvény első meghívásakor nyolc bittel forgatjuk el w1 értékét balra. A kimenetből látható, hogy a függvény a hexadecimális cdef00ab értéket adja vissza, ami valóban az abcdef00 nyolc bittel balra történő elforgatottja.

A rotate második meghívásakor w1 értékét 16 bittel forgatjuk el jobbra.

A következő két elforgatás hasonló műveleteket hajt végre w2-vel, melyek eredménye magáért beszél. A rotate utolsó előtti meghívása 0 bittel „forgatja el” a megadott értéket. A kimenetből látható, hogy a függvény – helyesen – az eredeti értéket adja vissza.

A rotate utolsó meghívása 44 bittel balra forgatja el w1 értékét. Ennek hatására 12 bittel balra forgatjuk el w1-et, hiszen $44 \% 32 = 12$.

Bitmezők

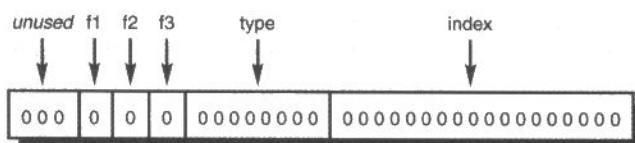
Az eddig tárgyalt operátorok segítségével számtalan kifinomult bitművelet megvalósítható. Bitenkénti műveleteket többnyire olyan adattípusokon végzünk, melyek tömörítve tárolnak valamiféle információt. Ahogy a `short int` is megspórol némi memóriát, hasonlóképp előfordul, hogy egy-egy bájt vagy szó csak egy bitnyi információt hordoz, így ki-sebb tárhelyre is be tudjuk szorítani a tényleges adatot. A logikai jelzőváltozók által tárolt *igaz* vagy *hamis* érték például elfér egyetlen biten. Ha deklarálunk egy `char` típusú jelzőváltozót, akkor ez (a `_Bool` típusú változókhöz hasonlóan) nyolc biten tárol egyetlen bitnyi információt. Ha nagy mennyiséggű jelzőváltozóval dolgozunk, akkor jelentős memória veszhet kárba ily módon.

A C nyelvben kétféle módszer is van az adatok összecsomagolására, a memóriahasználat visszasorítására. Az egyik lehetőség az `int` típusú számok használata, majd pedig a kívánt bitek (frissen tanult operátorokkal történő) közvetlen elérése. A hatékony adattömörítés másik lehetőségét a bitmezőnek nevezett C nyelvi konstrukció képviseli.

Az első lehetőség bemutatásához tegyük fel, hogy öt adatértéket szeretnénk egy szóba összetölteni. Ennek oka lehet az, hogy nagyon sok ilyen adattal dolgozik a program, és szeretnénk visszafogni a memóriahasználatot. Az öt adatérték közül legyen három jelölőbit (`f1`, `f2`, `f3`), egy `type` nevű egész szám, amely 1-től 255-ig tárol számokat, valamint egy `index` nevű egész szám, amely 18 biten őrzi értékeit 0 és százezer között. Így az öt adatérték (`f1`, `f2`, `f3`, `type`, `index`) számára $1+1+1+8+18=29$ bit szükséges. Kinevezhetünk egy egész számot ezeknek a tárolására:

```
unsigned int packed_data;
```

majd tetszőlegesen csoportosítva a biteket, hozzárendelhetjük az egyes részeket az öt adatértékhez. Ennek egy változatát láthatjuk a 12.1 ábrán, amelyen a `packed_data` méretét 32 bitnek tételezzük fel.



12.1 ábra

Bitmező kiosztása a `packed_data` változóban

Látható, hogy a `packed_data` változónak van három fel nem használt bitje. Az egész értékű változó valamennyi adatcsoportja beállítható vagy kiolvasható a megfelelő bitenkénti műveletek alkalmazásával. A `type`-nak elnevezett (és kezdetben csupa 0-t tartalmazó) bit-

csoport például úgy állítható be a 7-es értékre, hogy a megfelelő lépésszámban eltoljuk a 7-et balra, majd az eredményt bitenkénti VAGY kapcsolattal másoljuk át a packed_data kívánt helyeire:

```
packed_data |= 7 << 18;
```

Ehhez hasonlóan egy tetszőleges (0 és 255 közti) n értéket is használhatunk:

```
packed_data |= n << 18;
```

Az n leszorítása 255-nél kisebb (vagy egyenlő) értékre úgy is megoldható, hogy bitenkénti ÉS kapcsolatba hozzuk az n-t 0xff-fel.

Az előző utasítás természetesen csak akkor működik, ha előre tudjuk, hogy a type bitcsoportja nulla; ellenkező esetben le kell nullázni egy megfelelően kialakított ÉS művelettel. Az ehhez használt konstans (maszk) a type bitcsoportnak megfelelő nyolc helyiértéken nullát kell, hogy tartalmazzon (a többi helyen pedig egyest):

```
packed_data &= 0xfc03ffff;
```

Megkímélhetjük magunkat a fenti maszk kiszámításától (és egyúttal gépfüggetlenebb is lesz a kódunk) egy ügyes utasítás révén:

```
packed_data &= ~(0xff << 18);
```

Ezt a kódsort egy korábban tárgyalt utasítással egyesítve megtehetjük, hogy a packed_data type bitcsoportja (a korábban benne tárolt értéktől függetlenül) n legalacsonyabb nyolc bitjének értékét vegye fel:

```
packed_data = (packed_data & ~(0xff << 18)) | ((n & 0xff) << 18);
```

Itt néhány zárójel elhagyható lenne, de az olvashatóság érdekében mégis kiírtuk őket.

Látható, hogy egy ilyen (viszonylag egyszerű) művelet is (mint a type bitcsoport adott értékre való beállítása) milyen összetett kódot igényel. Egy bitcsoport kiolvasása már nem ilyen vészes: az eredeti szám eltolható úgy, hogy a legalacsonyabb helyiértékekben legyen minden szükséges bit, majd az eredmény ÉS kapcsolatba hozható egy megfelelő szélességű maszkkal. A type bitcsoport kiolvasása és n-be való betöltése tehát így történhet:

```
n = (packed_data >> 18) & 0xff;
```

A C nyelvben van ennél kényelmesebb módja is a bitek közvetlen kezelésének. Megadható egy sajátos szintaktikájú adatszerkezet, mellyel különféle bitcsoportok definiálhatók és nevezhetők el. Amikor a C nyelv szövegkörnyezetében „bitmezőkről” beszélünk, akkor ezt az adatszerkezetet értjük rajta.

A fent vázolt adatcsoportokat a következő módon tudjuk bitmezőként egybeszerkeszteni:

```
struct packed_struct
{
    unsigned int :3;
    unsigned int f1:1;
    unsigned int f2:1;
    unsigned int f3:1;
    unsigned int type:8;
    unsigned int index:18;
};
```

A packed_struct adatszerkezetnek hat tagja van. Az elsőnek nincs neve; a :3 három névtelen bitet takar. A második tag neve f1, ez is előjel nélküli egész típus. A tag neve után álló :1 jelzi, hogy mindenre egy bitnyi információt fog tárolni. Az f2-t és az f3-at szintén egy bites adatként definiáljuk. A type tag típusa nyolc bites, míg az index 18 bit hosszúságúra nyúlik.

A C fordítóprogram egymás mellé tömörítve tárolja a definiált bitcsoportokat. Az a nagyszerű az egészben, hogy ezek után ugyanúgy hivatkozhatunk egy packed_struct típusú szerkezet mezőire, mint egy adatszerkezet tagjaira. Deklaráljuk a következő módon a packed_data változót:

```
struct packed_struct packed_data;
```

A packed_data type mezőjét ezután könnyen be tudjuk állítani például 7-re:

```
packed_data.type = 7;
```

Sőt, tetszőleges (!) egész értéket is áthatunk neki:

```
packed_data.type = n;
```

Ebben az esetben még azzal sem kell bajlódnunk, hogy kellően kicsiny értéket tartsunk az n-ben, ugyanis csak a type-nak megfelelő számú (esetünkben 8) legalacsonyabb bit kerül átadásra.

Egy bitmező valamely értékének kiolvasása is hasonlóan egyszerű:

```
n = packed_data.type;
```

Ennek hatására a packed_data type mezője (automatikusan a legalacsonyabb helyértékekre léptetve) kerül be n-be.

A bitmezők normál módon használhatóak a kifejezésekben – itt egészüként értelmeződnek. Így tehát az

```
i = packed_data.index / 5 + 1;
```

kifejezés teljesen helyes, akárca az

```
if ( packed_data.f2 )  
    ...
```

feltételvizsgálat, amely az f2 jelölőbit igaz/hamis voltát vizsgálja.

Egy dolgot azonban jó szem előtt tartanunk: arra nincs garancia, hogy maguk az egyes bitcsoportok balról jobbra, vagy jobbról balra rendezve helyezkednek el a bitmezők memóriaterületén. Ez csak akkor jelenthet gondot, ha más számítógéppel állítottuk elő az adatokat, mint amelyiken az ezeket feldolgozó program fut. Ilyen esetben tudni kell, hogy milyen irányban sorakoznak egymás után a bitcsoportok – ezt a definícióval „utánozhatjuk”. Ha a packed_struct definícióját a fenti sorrend helyett így adjuk meg:

```
struct packed_struct  
{  
    unsigned int index:18;  
    unsigned int type:8;  
    unsigned int f3:1;  
    unsigned int f2:1;  
    unsigned int f1:1;  
    unsigned int :3;  
};
```

akkor a másik irányú elrendezést használó géppel előállított bitmező-adatokat is használhatjuk (12.1 ábra). Lehetőleg ne tételezzünk föl semmit arra vonatkozóan, hogy egy adatszerkezet tagjai (akár tartalmaznak bitmezőket, akár nem) hogyan vannak elhelyezve a memóriában.

Bitmezőket tartalmazó adatszerkezetekben normál típusú adattagok is tárolhatók. Ha egy olyan adatszerkezetet szeretnénk létrehozni, melyben van egy int, egy char, és két jelölőbit típusú adat, akkor ezt a következőképpen definiálhatjuk:

```
struct table_entry
{
    int          count;
    char         c;
    unsigned int f1:1;
    unsigned int f2:1;
};
```

Vannak azért megszorítások a bitmezőkkel együtt használható normál adattípusokra. Ezek csak egész vagy _Bool típusúak lehetnek. Ha csak annyit adunk meg, hogy int, akkor implementációfüggő, hogy előjeles vagy előjel nélküli lesz a típus. A biztonság kedvéért mindig érdemes kiírni, hogy signed int vagy unsigned int típust szeretnénk használni.

Bitmezőknek nem lehet indexe, azaz nem használhatunk tömböket vagy flag:1[5]-szerű bitcsoportokat. Nem lehet az egyes bitcsoportok „címét” kinyerni, nincs tehát „bitcsoport-mutató” sem.

A bitmezők a szerkezeti definíciójának megfelelően *egységekbe* vannak szervezve, de egy-egy egység mérete implementációfüggő. Általában ez a szóhossz többszöröse szokott lenni.

A C fordítóprogram *nem* rendezi át a bitmezők definíciós sorrendjét annak érdekében, hogy kevesebb memóriát használjon.

Még valamit érdemes tudni a bitmezőkkel kapcsolatban: a név nélküli, 0 hosszúságú bitcsoportnak sajátos jelentése van. Ennek hatására a definíció következő bitcsoportja a következő egység kezdetére kerül a memóriában.

Ezzel véget ért a C bitműveleteiről szóló fejezetünk. Láthattuk, milyen hatékony és rugalmas eszközök állnak rendelkezésünkre a C nyelvben a bitek manipulálására. Számos bitenkénti műveletet használhatunk (ÉS, VAGY, KIZÁRÓ VAGY, negáció, eltolás). Ha több különböző bitcsoporttal kell dolgoznunk, akkor használhatunk bitmező-definíciókat, melynek révén kényelmesen (maszkolás vagy eltolás nélkül) írhatjuk-olvashatjuk adataink bitcsoportjait.

Érdemes majd elmélyedni a 14. fejezetben („Mélyebben az adattípusokról”), ahol szó lesz arról, hogy mi történik akkor, amikor eltérő egész típusok között hajtunk végre bitenkénti műveleteket, például egy unsigned long int és egy short int között.

A következő fejezet elkezdése előtt oldjuk meg az alábbi gyakorlatokat, hogy elmélyítsük a C nyelv bitműveleteire vonatkozó ismereteinket.

Gyakorlatok

1. Gépeljük be és futtassuk a fejezetben szereplő négy példaprogramot! A kimeneteket hasonlítsuk össze a könyvben szereplő programkimenetekkel.
2. Írunk programot, amely felderíti, hogy az adott számítógép aritmetikai vagy logikai létpetést használ!
3. A ~0 kifejezés csupa egyesekből álló egész számot ad. Ennek felhasználásával írjuk meg az int_size függvényt, amely visszatérési értékül adja az adott számítógép int típusához tartozó bitek számát!
4. A 3. feladat felhasználásával módosítsuk a 12.4 Lista rotate függvényét, hogy ne legyen belehuzalozva az int típus méretére vonatkozó információ!
5. Írjuk meg a bit_test függvényt, amely megmondja, hogy egy szám adott bitje egyes-e! Két paramétere legyen: egy unsigned int típusú szám és egy bit-sorszám: n. Adjon vissza a függvény 1-est akkor, ha a megadott szám n. bitje 1-es, és 0-t akkor, ha 0. Sorszámozzuk a biteket a legmagasabb helyiérték felől, 0-val kezdve (tehát a baloldali bit a nulladik). Írjuk meg az fentivel azonos paraméterekkel meghívható bit_set függvényt is, amely egyesre állítja a megadott szám n. bitjét (és ezt adjá visszatérési értékül)!
6. Írjuk meg a bitpat_search függvényt, amely egy megadott bitmintát keres az első paraméterként megadott (előjel nélküli egész) számban! Három paramétere legyen a függvénynek:

```
bitpat_search (forras, minta, n)
```

A forras paraméterben keressük a minta paraméter n legalacsonyabb helyiértékű bitjét. Ha megtalálható a bitminta, akkor legyen a függvény visszatérési értéke annak a bitnek a sorszáma, ahol a bitminta kezdődik (a baloldali bit sorszáma legyen 0). Ha nem található meg a minta, akkor adjon vissza a függvény -1-et.

Így például az

```
index = bitpat_search (0xe1f4, 0x5, 3);
```

meghívásával a bitpat_search függvény a 0xe1f4 számértékben (ami binárisan 1110 0001 1111 0100) keresi a három bites 0x5 mintát (ami binárisan 101).

A függvény 11-et adjon visszatérési értékül, ugyanis a 11. biten kezdődik a megadott bitminta.

A függvény ne tételezzen fel semmit az int típus méretére vonatkozóan (lásd a 3. feladatot)!

7. Írjuk meg a bitpat_get függvényt, amely az első paraméterként megadott (előjel nélküli egész) számból kiolvas egy bitmintát! Három paramétere legyen a függvénynek: az első legyen egy unsigned int, a második legyen egy egész szám, amely a kívánt bitminta kezdőbitjét tartalmazza, a harmadik pedig adjon meg a kiemelendő bitek számát. A baloldali bit sorszáma legyen a nulladik. Olvassuk ki az első paraméter kívánt bitjeit! Például a

```
bitpat_get (x, 0, 3)
```

utasítással olvassuk ki az x három baloldali bitjét, valamint a

```
bitpat_get (x, 3, 5)
```

hívásával olvassunk ki az x szám (balról) negyedik bitjétől kezdve öt bitet.

8. Írjuk meg a bitpat_set függvényt, amellyel beállíthatunk néhány bitet egy meghatározott minta szerint! A függvény négy paramétert fogadjon el: egy unsigned int-re irányuló mutatót, amelynek a biteit átszeretnénk állítani; egy unsigned int-et, amely a kívánt bitmintát tartalmazza a jobboldali (alacsonyabb) bitjein; egy sorszámot, ahol elkezdődhet a bitek beállítása (balról nullával kezdve); valamint egy int típusú számot, amely a beállítandó bitek számát adja meg. Így tehát a

```
bitpat_set (&x, 0, 2, 5);
```

utasítás hatására az x szám harmadiktól kezdődő öt bitjét nullázzuk le. Hasonlóan a

```
bitpat_set (&x, 0x55u, 0, 8);
```

függvényhívás után az x szám nyolc baloldali bitje a hexadecimális 55 értéket vegye fel.

Ne tételezzünk fel semmit az int típus méretére vonatkozóan (lásd a 3. feladatot)!

13

Az előfeldolgozó

Ebben a fejezetben a C programozási nyelvnek egy olyan jellemzőjét vesszük szemügyre, amely sok más magas szintű nyelvből hiányzik. A C előfeldolgozó olyan eszközöket ad a kezünkbe, amelyek segítségével könnyebben fejleszthetők a programok; könnyebb olvasni és módosítani őket, sőt könnyebb a programokat átalakítani az eredetitől eltérő számítógép-architektúrán való működéshez. Az előfeldolgozó arra is használható, hogy a szó szoros értelmében testreszabjuk a C nyelvet, hogy alkalmasabb legyen egy adott alkalmazás kifejlesztéséhez. Saját programozási stílusunk kialakításához is jó eszközöket biztosít az előfeldolgozó.

Az előfeldolgozás (preprocessing) a programkód lefordításának első fázisa, melyben meghatározott kifejezéseket keres a fordítóprogram. Ahogy a nevéből is látszik, az előfeldolgozás még *azelőtt* lezajlik, mielőtt magát a programot értelmezni kezdené a fordítóprogram. Az előfeldolgozónak szánt utasítások sorai # jellet kezdődnek (Pontosabban az első nem térköz karakterek kell #-nek lennie). Látni fogjuk, hogy az előfeldolgozó parancsainak szintaxisa eltér a C utasításokétől. Kezdjük fejezetünket a `#define` vizsgálatával.

A `#define` utasítás

A `#define` használatának elsődleges célja az, hogy jól olvasható szimbolikus neveket rendeljünk a konstansokhoz. A

```
#define YES 1
```

előfeldolgozó-utasítás definiálja a YES helyettesítő nevet; használata egyenértékű lesz az 1-es konstanssal. Azaz a YES-t bárhol használhatjuk, ahol a programban 1-est használunk. Ahol ez a név megjelenik a forráskódban, ott az előfeldolgozó 1-est helyez el a fordítási folyamat előtt. Tekintsük például az alábbi – jól olvasható – utasítást, amely a `gameOver` értékéhez YES-t rendel:

```
gameOver = YES;
```

Nem is kell azzal törődnünk, hogy a YES milyen értéket takar. A vájtfülű olvasó azonban tudja, hogy korábban ezt 1-esként definiáltuk, így a fenti értékadás olyan, mintha 1-et adnánk át a gameOver-nek. A

```
#define NO 0
```

előfeldolgozó-utasítás a NO helyettesítő nevet definiálja, és ennek későbbi előfordulásait egyenértékűvé teszi a 0 használatával. Így a

```
gameOver = NO;
```

hozzárendelés NO értékét tölti be a gameOver-be, illetve az

```
if ( gameOver == NO )
    ...
```

feltételvizsgálat a gameOver értékét a NO értékével veti össze. A definiált helyettesítő nevek egy helyzetben nem használhatóak karakterláncokban. Azaz a

```
char *charPtr = "YES";
```

utasítás a charPtr mutatót a „YES” karakterláncra irányulóan fogja inicializálni, és nem az „1”-esre.

A definiált helyettesítő nevek *nem* változók, így nem lehet nekik értéket adni a programban – haocsak a maga helyettesített objektum nem egy változónév. Ahol csak (karakterlánc-konstansokon kívül) megjelenik a programban egy helyettesítő név, ott az előfeldolgozó a #define ezt kicseréli a definiált értékre. Mintha csak „keresés és csere” műveletet hajtanánk végre egy szövegszerkesztőben az összes definiált helyettesítő névre.

Figyeljük meg a #define szintaxisát: nincs benne értékadó egyenlőségjel, sem pontosvessző nincs az utasítás végén. Hamarosan visszatérünk ennek okára. De először lássunk egy programot, amely a fent megadott YES és NO értékeket használja!

A 13.1 Lista isEven függvénye YES-t ad vissza, ha a paraméterként megadott egész szám páros, és NO-t, ha páratlan.

13.1 Lista • A #define utasítás használata

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

```
#define YES      1
#define NO       0
```

```
// Egy egész szám paritását vizsgáló függvény

int isEven (int number)
{
    int answer;

    if ( number % 2 == 0 )
        answer = YES;
    else
        answer = NO;

    return answer;
}

int main (void)
{
    int isEven (int number);

    if ( isEven (17) == YES )
        printf ("yes ");
    else
        printf ("no ");

    if ( isEven (20) == YES )
        printf ("yes\n");
    else
        printf ("no\n");

    return 0;
}
```

13.1 Lista • Kimenet

no yes

A program első utasítása a `#define`. Ez azonban nem előírás: a névhelyettesítő definíció bárhol lehet a programban, mielőtt az adott helyettesítő név először előfordul. A helyettesítő név abban is eltér a normál változóktól, hogy nem lehet „lokális”. Attól kezdve, hogy megjelent a `#define`, a helyettesítő név bárhol megjelenhet, függvényekben vagy azokon kívül. A programozók általában a program elején adják meg a névhelyettesítéseket, vagy egy külön *include* fájlban. Ez utóbbi megoldás előnye, hogy a definíciók így több forrásvállományban is könnyen érvényesíthetők.¹

A `NONE` nevet gyakran használják a programozók a nullmutatók helyettesítésére.²

¹ A fejezet későbbi részéből kiderül, hogy miként lehet a névhelyettesítő definíciókat külön fájlokban elhelyezni.

² Az `<stddef.h>` fejlécállományban megtalálható a `NONE` definíciója. A fejlécállományok kérdésére hamarosan visszatérünk.

A programban elhelyezett

```
#define NULL 0
```

névhelyettesítés révén olvashatóbbá válik a programunk. Az alábbi ciklus addig fut, amíg a listPtr nullmutató nem lesz:

```
while ( listPtr != NULL )
...
```

A névhelyettesítés következő példájaként képzeljük el, hogy három függvényt kell megírnunk. Az egyiknek egy adott sugarú kör kerületét, a másodiknak a kör területét, a harmadiknak pedig egy adott sugarú gömb térfogatát kell kiszámítania. Mindhárom függvényhez használnunk kell a π értékét, amely nem könnyen megjegyezhető konstans. Célszerű tehát egy helyettesítő nevet bevezetni a program elején, hogy minden függvényben látható legyen a π konstans.³

A 13.2 Listában bemutatjuk, hogy miként lehet egy konstans helyettesítő nevét megadni és használni.

13.2 Lista További tudnivalók a `#define` utasításról

```
/* Egy adott sugárral megadott kör kerületét, területét és az ugyanilyen sugarú gömb térfogatát kiszámító függvény */
```

```
#include <stdio.h>

#define PI           3.141592654

double area (double r)
{
    return PI * r * r;
}

double circumference (double r)
{
    return 2.0 * PI * r;
}

double volume (double r)
{
    return 4.0 / 3.0 * PI * r * r * r;
```

³ Az M_PI megtalálható a `<math.h>` fejlécállományban – ennek beemelésével a π elérhetővé válik a programban.

```

int main (void)
{
    double area (double r), circumference (double r),
           volume (double r);

    printf ("sugár = 1: %.4f %.4f %.4f\n",
            area(1.0), circumference(1.0), volume(1.0));

    printf ("sugár = 4.98: %.4f %.4f %.4f\n",
            area(4.98), circumference(4.98), volume(4.98));
    return 0;
}

```

13.2 Lista • Kimenet

```

sugár = 1: 3.1416 6.2832 4.1888
sugár = 4.98: 77.9128 31.2903 517.3403

```

A PI helyettesítő nevet 3.141592654-ként definiáljuk a program elején. A kerület, terület és térfogat kiszámítását végző függvényekben ezt a nevet használjuk. A program fordításakor mindenhol, ahol PI szerepel, behelyettesítődik a fent megadott konstans.

A konstansokat azért is érdemes névhelyettesítéssel definiálni, mert így nem kell fejben tartanunk őket, amikor a program különböző pontjain előkerülnek. Ha netán tévesen adjuk meg az értéket, akkor elegendő egy helyen (a #define utasításban) korrigálni. Ha nem használhatnánk névhelyettesítést, akkor tévedés esetén az összes előfordulási helyen bele kellene javítanunk a programba.

Bizonyára feltűnt, hogy az összes helyettesített nevet (YES, NO, NULL, PI) nagybetűsen használtuk. Célunk, hogy a névhelyettesítéseket vizuálisan is jól elkülönítsük a változónevektől. Ezt a közmeggyezést sok programozó elfogadja. Mások úgy döntötték, hogy inkább egy *k* kezdőbetűvel jelölik a helyettesített neveket, és nem szedik nagybetűsen őket (például *kMaximumValues*, *kSignificantDigits*).

A program kiterjeszthetősége

A definiált konstansokat arra is felhasználhatjuk, hogy elősegítsük programunk kiterjeszthetőségét, aktualizálását. Amikor egy tömböt definiálunk, akkor (kifejezett számmal vagy kezdőérték-listával) meg kell adnunk a méretét. Jó néhány utasításban szükség lesz erre az információra. Tegyük fel, hogy a *dataValues* tömböt a következőképp definiáljuk:

```
float dataValues[1000];
```

Ezek után valószínűleg többször is támaszkodnunk kell erre az 1000-es küszöbértékre, például a tömb elemeit bejáró *for* ciklusban:

```

for ( i = 0; i < 1000; ++i )
    ...

```

vagy a tömb határának átlépését vizsgáló feltételben:

```
if ( index > 999 )
    ...
```

Ha egy ilyen programban (például az adatok növekvő száma miatt) meg kell változtatni a tömb méretét, akkor minden olyan helyen bele kell nyúlni a forráskódba, ahol a konstansok támaszkodnak a `dataValues` tömb méretére.

A tömbhatárok rugalmasabb kezelését megoldhatjuk helyettesítő név használatával. Adjuk meg a `MAXIMUM_DATAVALUES`-t egy alkalmas `#define` utasítással:

```
#define MAXIMUM_DATAVALUES 1000
```

Ezután a `dataValues` tömböt már a `MAXIMUM_DATAVALUES` értékével tudjuk definiálni:

```
float dataValues[MAXIMUM_DATAVALUES];
```

A helyettesítő név szerepelhet a tömb elemeit bejáró `for` ciklusban:

```
for ( i = 0; i < MAXIMUM_DATAVALUES; ++i )
    ...
```

A tömb határának átlépését így is megvizsgálhatjuk:

```
if ( index > MAXIMUM_DATAVALUES - 1 )
    ...
```

Az egészben az a legjobb, hogy egy ilyen programban elegendő egy helyen (a fenti `#define` utasításban) átírnunk a tömb elemszámát:

```
#define MAXIMUM_DATAVALUES 2000
```

Ha a tömb méretét felhasználó összes kódrészletben a `MAXIMUM_DATAVALUES` értékére támaszkodunk, akkor sehol másol nem kell módosítanunk a programot.

A program hordozhatósága

A névhelyettesítéseket arra is felhasználhatjuk, hogy megkönnyítsük programjaink átalakítását más számítógép-architektúrákon való futtatáshoz. Időnként szükségünk lehet egy-egy olyan konstansra, amely csak az adott számítógéptípuson érvényes. Ez lehet egy konkrét memóriacím, egy fájlnév, vagy az adott gépen használt szóhossz. A 12.4 Lista `rotate` függvényében például támaszkodtunk arra, hogy az `int` típus 32 bites.

Ha ezt a programot olyan gépen szeretnénk futtatni, amely 64 biten tárolja az int értéket, akkor a fenti `rotate` függvény nem fog helyesen működni.⁴

Figyeljük meg a következő kód részletet! Ha kénytelenek vagyunk gépfüggő értékeket használni, akkor ezeket érdemes elkülöníteni a program többi részétől, amennyire csak lehetséges. Segítségünkre lehet ebben az elkülönítésben a `#define` utasítás. A `rotate` függvény következő verzióját könnyű lenne más géptípusokra átvinni (bár ez egyébként sem különösebben nehéz kérdés). Íme a függvény:

```
#include <stdio.h>

#define kIntSize 32 // *** Gépfüggő !!! ***

// Előjel nélküli egész szám elforgatását végző függvény

unsigned int rotate (unsigned int value, int n)
{
    unsigned int result, bits;

    // A túl nagy eltolási érték lefaragása a szükséges mértékre

    if (n>0)
        n = n % kIntSize;
    else
        n = -(-n % kIntSize);
    if (n == 0)
        result = value;
    else if (n > 0)           // balra forgatás
    {
        bits = value >> (kIntSize - n);
        result = value << n | bits;
    }
    else                      // jobbra forgatás
    {
        n = -n;
        bits = value << (kIntSize - n) ;
        result = value >> n | bits;
    }

    return result;
}
```

⁴ Természetesen a `rotate` függvény átírható oly módon, hogy futás közben meghatározza az éppen aktuális int típus biteinek számát (és így gépfüggetlen legyen). Ezt kéri a 12. fejezet 3. és 4. gyakorlata.

Bonyolultabb definíciók

A névhelyettesítéskor nem pusztán egyszerű konstansok adhatóak meg. Kifejezések is (sőt, szinte bármilyen) értékül adható a definiált neveknek!

A következő definícióban TWO_PI néven adjuk meg 2.0 és 3.141592654 szorzatát:

```
#define TWO_PI 2.0 * 3.141592654
```

Ettől kezdve a fenti nevet bárhol használhatjuk, ahol $2.0 \cdot 3.141592654$ -re lenne szükségünk. Az előző program circumference (kerületszámító) függvényének például ez lehetne a visszatérési értéke:

```
return TWO_PI * r;
```

Amikor egy névhelyettesítési definíció megjelenik, akkor a program további részében (a karakterlánc-konstansok kivételével) a megadott név betű szerint kicserélődik mindenre, ami a definícióban tőle jobbra szerepel. A TWO_PI esetében például ennek összes előfordulását $2.0 \cdot 3.141592654$ -re cseréli ki az előfeldolgozó.

Ez a betű szerinti csere magyarázza azt is, hogy miért nem kell pontosvesszőt tenni a #define utasítás végére. Ha pontosvesszőt írunk, akkor az is a cserélendő szöveghez fog tartozni. Ha így adtuk volna meg PI értékét:

```
#define PI 3.141592654;
```

majd pedig ez következne a kódban:

```
return 2.0 * PI * r;
```

Akkor a csere eredménye így festene:

```
return 2.0 * 3.141592654; * r;
```

ez pedig szintaktikai hibához vezetne.

Az előfeldolgozó által lecserélendő kifejezések nem feltétlenül kell „megengedett C kifejezések” lennie; csak a csere után előálló kifejezéseknek kell működniük.

Lehet például ilyen definíciót is adni:

```
#define LEFT_SHIFT_8 << 8
```

pedig a LEFT_SHIFT_8 után megjelenő karakterlánc nem érvényes kifejezés. A LEFT_SHIFT_8-at felhasználhatjuk programainkban:

```
x=y LEFT_SHIFT_8;
```

Ennek hatására az y bitjei nyolccal balra tolódnak, és az eredmény bekerül az x változóba. Vagy tekintsük az ennél jóval gyakorlatiasabb definíciókat:

```
#define AND      &&
#define OR       ||
```

melyek alapján fogalmazhatunk így is:

```
if ( x > 0 AND x < 10 )
    ...
```

vagy akár így:

```
if ( y == 0 OR y == value )
    ...
```

Még az egyenlőségvizsgálatot is átnevezhetjük:

```
#define EQUALS ==
```

majd így vethetjük be fegyvertárunkat:

```
if ( y EQUALS 0 OR y EQUALS value )
    ...
```

Ezzel a kódrészlettel kiküszöbölhetünk egy könnyen elkövethető hibát (szimpla = jel írása a == helyett), valamint (angolul) olvashatóbbá is válik a kódunk.

Bár a fenti példák bemutatják a #define erejét, mégis érdemes tudni, hogy a programozók nem tekintik jó szokásnak a nyelv szintaxisának túlságosan mélyreható árajzolását. Főként azért nem, mert így mások számára nehezebbé válhat a kód megértése.

Érdekességgéként megemlíjtük, hogy egy névhelyettesítés maga is alapjául szolgálhat egy másik névhelyettesítésnek. Nézzük a következő (teljesen működőképes) kódrészletet:

```
#define PI      3.141592654
#define TWO_PI   2.0 * PI
```

Egy új név helyettesítésének megadásakor *bármilyen* használhatunk, ami addig a pontig definiált a programban.

A jól eltalált névhelyettesítések még a megjegyzések használatát is feleslegessé tehetik. Nézzük a következő feltételvizsgálatot:

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

Ez az if utasítás azt ellenőrzi, hogy szökőév-e a megadott év. Nézzük a következő névhelyettesítést:

```
#define IS_LEAP_YEAR      year % 4 == 0 && year % 100 != 0 \  
                      || year % 400 == 0  
...  
if ( IS_LEAP_YEAR )  
    ...
```

Az előfeldolgozó alapvetően arra számít, hogy minden #define egysoros. Ha netán nem fér ki valami egy sorban, akkor backslash karakterrel jelezhetjük, hogy folytatódik a sor. Magát a backslash természetesen figyelmen kívül hagyja az előfeldolgozó. A további sorok is ugyanígy illeszthetők az előzőekhez egy-egy újabb backslash segítségével.

Az utolsó if utasítást sokkal könnyebb elolvasni, mint a fentit. Kommentár nélkül is teljesen érthető. Az IS_LEAP_YEAR szerepe hasonlít egy függvényéhez. Használhattunk volna egy is_leap_year nevű függvényt is, ami hasonlóan olvasható kódot eredményezett volna. Egyéni ízlés dolga, hogy melyiket választjuk. Az is_leap_year függvény természetesen rugalmasabb megoldást jelent, hiszen egy függvény paramétereket is elfogadhat. Egy függvénnyel bármilyen változó szökőév-volta vizsgálható, nem pedig csak konkrétan a year változóé (mint a #define IS_LEAP_YEAR esetében). A különös az, hogy a #define-ok is elfogadnak paramétert, amint arról a következő bekezdésben szó lesz.

Paraméterek és makrók

Az IS_LEAP_YEAR definíciója elfogadhat paramétert is (mely esetünkben y néven szerepel):

```
#define IS_LEAP_YEAR(y)      y % 4 == 0 &&      y % 100 != 0      \  
                           ||      y % 400 == 0
```

A függvényekkel ellentétben az y típusát nem kell megadnunk, ugyanis pusztán betűszintű helyettesítés történik, nem pedig függvényhívás. Szóköz nem kerülhet a #define-ban szereplő név és a paraméterlista zárójele közé.

Ez utóbbi `#define` segítségével az alábbi két kód részlet is használható:

```
if ( IS_LEAP_YEAR (year) )
    ...
```

illetve:

```
if ( IS_LEAP_YEAR (next_year) )
    ...
```

Az első `if` a `year`, a második a `next_year` változót ellenőrzi, hogy szökőév-e. Az utóbbi utasításban az `IS_LEAP_YEAR` által definiált szövegrészlet kerül be az `if` feltételvizsgálatába, de előtte a `next_year` lecseréli a definíció minden `y`-t tartalmazó részletét. A fordítóprogram végül ezt fogja látni:

```
if ( next_year % 4 == 0 && next_year % 100 != 0
    || next_year % 400 == 0 )
    ...
```

A C nyelv efféle definícióit gyakran makróknak hívják; különösen is azokat, amelyek egy vagy több paramétert is elfogadnak. A makróknak megvan az az előnye a függvényekkel szemben, hogy lényegtelen a paraméter típusa. Létrehozhatunk például egy `SQUARE` nevű makrót, amely négyzetre emeli a paraméterként kapott értéket. A

```
#define SQUARE(x) x*x
```

utasítás az alábbi kifejezés használatát teszi lehetővé:

```
y = SQUARE (v);
```

Ennek hatására `y` felveszi `v2` értékét. Fontos látni, hogy a fenti makró ugyanúgy lefut akkor is, ha `v` `int` típusú, vagy ha `long` vagy `float` típusú. Ha a `SQUARE`-t függvényként adnánk meg, akkor el kellene döntenünk, hogy milyen típust fogadjon el. Ha `int`-re fogalmaznánk meg a függvényt, akkor nem adhatnánk át neki `double` értéket.

A makrók használatának vannak mellékhatásai, amikről tudnunk kell. Mivel az előfeldolgozó szövegként illeszti be a makrókat a programba, nagyobb memóriamennyiség szükséges a lefordításukhoz, mint a függvények esetében. Azonban kevesebb futási időbe telelik egy makró által behelyettesített kód részlet végrehajtása, mint egy függvényhívás.

Bár a `SQUARE` fenti definíciója nem egy bonyolult művelet, mégis rejt egy apró csapdát.

Ha az

```
y = SQUARE (v);
```

utasítás helyett (amely v^2 értékét tölti be y-ba) az

```
y = SQUARE (v + 1);
```

utasítást használjuk, akkor első nekifutásra azt várnánk, hogy ezzel $(v+1)^2$ értéke kerül y-ba. Azonban, ha jobban belegondolunk a szövegcserébe, láthatjuk, hogy a névhelyettesítés feloldása után az

```
y = v + 1 * v + 1;
```

kódrészletet kapjuk, ami nem a várt eredményt számítja ki. Ennek kivédésére zárójeleket szokás alkalmazni a makródefinícióban:

```
#define SQUARE(x) ( (x) * (x) )
```

Bár ez így furcsán fest, minden szem előtt kell tartanunk, hogy itt betű szerinti cseréről lesz szó, és az x összes előfordulási helyére behelyettesítődik az átadott paraméter. Így ebben az esetben az

```
y = SQUARE (v + 1);
```

a várakozásnak megfelelően helyettesítődik be:

```
y = ( (v + 1) * (v + 1) );
```

A feltételkezelő operátort igen jól lehet használni a makródefiníciókban. Az alábbi módon hozhatunk létre egy maximumkereső makrót:

```
#define MAX(a,b) (( (a) > (b) ) ? (a) : (b) )
```

Ezzel ilyen értékadásokat hajthatunk végre:

```
limit = MAX (x + y, minValue);
```

Hatására x + y és minValue maximuma kerül a limit változóba.

A definícióban szereplő külső zárójelpár teszi lehetővé, hogy a

```
MAX (x, y) * 100
```

kifejezés is helyesen működjön. A paraméterek körüli zárójelek hatására pedig a

```
MAX (x & y, z)
```

esetben is helyesen kapjuk meg a keresett maximumt. A bitenkénti ÉS műveletnek (<&>) ugyanis alacsonyabb a precedenciája, mint a makróban szereplő > operátornak. Zárójel nélkül korábban kiértékelődne a „nagyobb” jel, mint a bitenkénti ÉS, így nem a kívánt eredményt kapnánk.

A következő makró azt vizsgálja, hogy egy karakter kisbetűs-e.

```
#define IS_LOWER_CASE(x) ((x) >= 'a') && ((x) <= 'z')
```

Ezzel igen szemléletesen lekérdezhetjük egy karakter kisbetűsségeit:

```
if (IS_LOWER_CASE(c))  
    ...
```

Ezt akár egy másik makródefinícióban is használhatjuk, amely az ASCII kisbetűket nagybetűkké alakítja (változatlanul hagyva a kisbetűkön kívüli karaktereket):

```
#define TO_UPPER(x) (IS_LOWER_CASE(x) ? (x) - 'a' + 'A' : (x))
```

A következő ciklus végigjárja a mutató által megadott karakterláncot, nagybetűvé alakítva a kisbetűket:⁵

```
while (*string != '\0')  
{  
    *string = TO_UPPER(*string);  
    ++string;  
}
```

Különböző számú paraméter átadása makróknak

A makrók úgy is definiálhatók, hogy nem határozzuk meg egyértelműen a paraméterek számát. Ezt három ponttal jelezhetjük az előfeldolgozónak. A meg nem adott paraméterek listája a __VA_ARGS__ nevű azonosítóval érhető el. Nézzük a következő példát, amely a debugPrintf makró számára változó számú paraméter használatát teszi lehetővé:

```
#define debugPrintf(...) printf("DEBUG: " __VA_ARGS__);
```

⁵ Számos könyvtári függvény is lehetővé tesz karakter-átalakításokat és -vizsgálatokat. Az islower és toupper függvény ugyanazt a feladatot látja el, mint a fenti IS_LOWER_CASE és TO_UPPER makrók. A részleteket lásd a B függelékben.

Ilyen definíció után teljesen jól működnek a következő kódsorok:

```
debugPrintf ("Hello world!\n");
debugPrintf ("i = %i, j = %i\n", i, j);
```

Az első esetben

```
DEBUG: Hello world!
```

lesz a kimenet, míg a második esetben (i=100 és j=200 értékek esetén)

```
DEBUG: i = 100, j = 200
```

olvasható a képernyőn.

Az első eset printf függvénye így bomlik ki:

```
printf ("DEBUG: " "Hello world\n");
```

Az előfeldolgozó azonban egyesíti (!) a szomszédos karakterlánc-konstansokat. Végül tehát ez a sor látható a fordítóprogram számára:

```
printf ("DEBUG: Hello world\n");
```

A # operátor

Ha a makródefinícióban egy # jelet helyezünk el a paraméter neve elő, akkor az előfeldolgozó konstans karakterláncot hoz létre a paraméterből. Legyen str a következő makró:

```
#define str(x) #x
```

Ennek hatására az

```
str (testing)
```

meghívása

```
"testing"
```

helyettesítő értékre vezet. Az alábbi két printf hívás tehát egyenértékű:

```
printf (str (Programming in C is fun.\n));
printf ("Programming in C is fun.\n");
```

Az előfeldolgozó két idézőjellel zára közre a kettőskereszttel megadott paramétert. Ha a paraméterként megadott karakterlánc idézőjelet vagy backslash karaktert tartalmaz, akkor azt levédi az előfeldolgozó. Az

```
str ("hello")
```

a következő helyettesítésre vezet:

```
"\hello\"
```

Nézzük a # operátor használatának egy gyakorlatiasabb példáját:

```
#define printint(var) printf (# var " = %i\n", var)
```

Ezzel a makróval egy egész típusú változó értékét jeleníthetjük meg. Legyen count egy egész változó, melynek értéke 100. Ekkor a

```
printint (count);
```

utasítás a következő helyettesítésre vezet:

```
printf ("count" " = %i\n", count);
```

amely a karakterláncok egybevonása után így fest:

```
printf ("count = %i\n", count);
```

Vagyis a # operátor lehetővé teszi, hogy karakterláncot állítsunk elő a makró paramétere-ből. A # és a paraméter közé tehető szóköz, de ez nem kötelező.

A ## operátor

Két jelet összekapcsolhatunk a ## operátorral. Előtte vagy utána egy paraméternévnek kell állnia. Az előfeldolgozó behelyettesíti az átadott paramétert, majd „összeragasztja” a ## operátor túloldalán levő karakterláncot.

Legyen x1, x2 ... x100 száz különböző változó. Létrehozhatunk egy printx nevű makrót, amelynek egyetlen paramétere egy 1 és 100 közötti szám, és meghívásakor kiírja a megfelelő x változót:

```
#define printx(n) printf ("%i\n", x ## n)
```

A definíció ide vonatkozó része így néz ki:

```
x ## n
```

Ebből az derül ki az előfeldolgozó számára, hogy a ## előtti és utáni jelet (vagyis az x szimbólumot és az n paramétert) egyetlen jellé kell egybevonni. A

```
printx (20);
```

utasítás így bomlik ki a makró lefutása után:

```
printf ("%i\n", x20);
```

A printx makróban felhasználhatjuk az imént definiált printint makrót, amellyel egy változó neve és értéke jeleníthető meg:

```
#define printx(n) printint(x ## n)
```

A

```
printx (10);
```

makróhívás

```
printint (x10);
```

utasításként értelmeződik, ez pedig

```
printf ("x10" " = %i\n", x10);
```

formában bomlik ki, végül a szomszédos karakterláncok egyesítése után a

```
printf ("x10 = %i\n", x10);
```

utasítást fogja látni a fordítóprogram.

Az #include utasítás

Ha már hosszabb idő óta programozik az ember C-ben, egy idő után hasznos makrók gyűlnek össze, melyeket egyre több programban szeretnénk működtetni. Ezeket nem is kell mindenhol bemásolni. Az előfeldolgozó lehetővé teszi, hogy külön fájlokba gyűjt-sük a gyakrabban használt makrókat, majd pedig ezeket az állományokat beemeljük (*include*) a programok elejére. Ehhez az `#include` utasítást használhatjuk. Ezeknek a fej-lécállományoknak a neve általában .h-ra végződik.

Tegyük fel, hogy több programban szeretnénk használni mértékegység-átváltást. A képletekben szereplő konstansokat `#define`-okban célszerű megadni:

```
#define INCHES_PER_CENTIMETER 0.394
#define CENTIMETERS_PER_INCH 1 / INCHES_PER_CENTIMETER

#define QUARTS_PER_LITER 1.057
#define LITERS_PER_QUART 1 / QUARTS_PER_LITER

#define OUNCES_PER_GRAM 0.035
#define GRAMS_PER_OUNCE 1 / OUNCES_PER_GRAM
...
```

Ha ezeket a kódsorokat egy külön állományba (például `metric.h`) mentjük el, akkor ezekre bármely programban hivatkozhatunk. Mindössze annyi a teendő, hogy azoknak a programoknak az elején, melyekben használni szeretnénk a konstansokat, meg kell adni a beemelési parancsot:

```
#include "metric.h"
```

Ennek az utasításnak még azelőtt szerepelnie kell, mielőtt az állományban található definiciókra támaszkodnánk a programban. Az előfeldolgozó az adott ponton megkeresi a hivatkozott állományt, és ténylegesen bemásolja a tartalmát a programba oda, ahol az `#include` utasítás szerepelt. Ettől kezdve ugyanúgy látja a forráskódot a fordítóprogram, mintha begépeltük volna a hivatkozott fájl tartalmát a kívánt helyre.

A beemelendő állomány neve körüli idézőjel arra utal, hogy a megadott fájlt egy (vagy több) könyvtárban keresse. (Általában ott, ahol maga a forráskód is van – az pedig rendszerfüggő, hogy ezen kívül még hol.) Ha a fájl itt nem található, akkor az előfeldolgozó automatikusan a rendszerkönyvtárban fogja keresni, amiről mindenki szó lesz.

Ha a fájlnevet < és > jelek közé zárjuk, akkor az előfeldolgozó egy megadott rendszerkönyvtárban (vagy könyvtárakban) fogja keresni az állományt:

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

Ennek helye rendszerfüggő. Unix rendszereket (beleértve a Mac OS X-et is) a rendszerkönyvtár a `/usr/include`, így az `stdio.h` szabványos fejlécállomány itt található: `/usr/include/stdio.h`.

Annak bemutatására, hogy hogyan is használhatók a beemelendő fájlok egy programban, gépeljük be a fenti mértékegység-átváltásokat egy `metric.h` nevű állományba, majd futtassuk le a 13.3 Lista programját!

13.3 Lista • Az `#include` utasítás használata

```
/* Az #include használatát bemutató program.
A mértékegység-átváltási definíciók
feltételezett helye a metric.h. */

#include <stdio.h>
#include "metric.h"

int main (void)
{
    float liters, gallons;

    printf ("*** Literból gallonba ***\n\n");
    printf ("Liter: ");
    scanf ("%f", &liters);

    gallons = liters * QUARTS_PER_LITER / 4.0;
    printf ("%g liter = %g gallon\n", liters, gallons);

    return 0;
}
```

13.3 Lista • Kimenet

```
*** Literból gallonba ***
Liter: 55.75
55.75 liter = 14.73 gallon.
```

Példánk meglehetősen egyszerű, ugyanis csak egyetlen konstanst (`QUARTS_PER_LITER`) használ a `metric.h` fejlécállományból. Ennek ellenére jól látható, hogy a megfelelő `#include` utasítás után használhatóak a `metric.h` definíciói.

Az állományok beemelésének egyik legfőbb előnye, hogy ily módon „központosítva” tartózkodnak a definíciók – minden programban ugyanazokat az értékeket tudjuk használni. Hiba esetén elég egy helyen javítani a téves adatot, nem kell több programba beleírni. A javított értékre hivatkozó programokat elég módosítás nélkül újrafordítani.

A beemelt állományokban bármi szerepelhet, nemcsak #define utasítások. Jó programozási gyakorlatnak számít a gyakran használt előfeldolgozó-utasítások, adatszerkezet-definíciók, függvény-deklarációk és globális változók központosított állományokba való összegyűjtése.

Még egy tudnivaló a beemelt állományokról: a fájlok beemelése egymásba ágyazható. Egy behívott fájl magába emelhet más fájlokat, azok is továbbiakat stb.

A rendszer fejlécállományai

Említettük, hogy a `<stddef.h>` fejlécállomány tartalmazza a NULL definícióját, amit jól felhasználhatunk például a nullmutatók ellenőrzésére. Arról is szó volt, hogy a `<math.h>` fejlécállomány tartalmazza az `M_PI` konstanst, amely π -nek egy jó közelítése.

A `<stdio.h>` fejlécállomány a szabványos kimeneti/bemeneti könyvtár eljárásaihoz tartalmaz információkat. Erről az állományról még szó lesz a 16. fejezetben („A C kimeneti és bemeneti műveletei”). Ha kimeneti/bemeneti műveleteket használunk egy programban (gyakorlatilag minden), akkor be kell emelnünk ezt a fájlt.

Két további hasznos rendszer-fejlécállomány a `<limits.h>` és a `<float.h>`. A `<limits.h>` rendszerfüggő konstansokat tartalmaz a különféle (karakteres és egész szám jellegű) típusok méretéről. Az int számok maximumát például a `MAX_INT` adja meg, az unsigned long int típus legnagyobb értékét pedig az `ULONG_MAX` stb.

A `<float.h>` fejlécállomány a lebegőpontos típusok adatait definiálja. Az `FLT_MAX` adja meg a float típus maximumát, a `FLT_DIG` pedig a float típus tizedesjegyeinek számát.

Más rendszer-fejlécállományok különféle rendszerkönyvtárbeli függvények deklarációt tartalmazzák. A `<string.h>` fejlécállomány például a karakterkezelő könyvtári függvények prototípus-deklarációt adja meg (másolás, összehasonlítás, összefűzés stb.).

További részleteket a B függelékben találunk.

Feltételes fordítás

A C előfeldolgozó lehetővé teszi a feltételes fordítást. Ennek segítségével megoldható, hogy a különféle számítógép-architektúrákon különbözőképpen folyjon a program fordítása. Arra is felhasználható, hogy egy-egy utasításcsoportot (például a változók értékeit megjelenítő vagy a program futási folyamatát visszajelző nyomkövetési utasításokat) ki- vagy bekapsolunk.

Az #ifdef, az #endif, az #else és az #ifndef utasítás

Szó volt arról, hogyan lehet a 12. fejezet `rotate` függvényét hordozhatóbbá tenni. Láttuk, hogy komoly segítséget jelent egy-egy jól eltalált `#define` utasítás:

```
#define kIntSize 32
```

Ezzel elkülöníthető az `unsigned int` típus bitjeinek számára vonatkozó információ a program többi részétől. Azt is megállapítottuk, hogy ez a függőség teljesen megszüntethető, hiszen a függvény maga is meg tudja határozni az `unsigned int` típus bitjeinek a számát.

Vannak olyan programok, amelyeknek mindenkorban támaszkodniuk kell rendszerfüggő adatokra (például állományok nevére). Ezek operációs rendszertől függően eltérőek lehetnek.

Tegyük fel, hogy van egy nagyméretű programunk, amely több ponton függ az aktuális hardvertől és operációs rendszer paramétereitől. Amennyire lehet, megpróbáljuk ezeket a függőségeket minimalizálni, ám mégis számos `#define` utasítás kerül a programba. Ezek nagy részét módosítani kell, amikor más architektúrájú számítógépre is le szeretnénk fordítani a programot.

A változtatások problémájának egy részét meg tudjuk oldani az előfeldolgozó feltételes fordítási utasításaival. Megadható, hogy adott géptípusokra mely utasításokat kell használnia a fordítóprogramnak, és melyeket nem. Ennek formája a következő lehet:

```
#ifdef UNIX
#  define DATADIR "/uxn1/data"
#else
#  define DATADIR "\usr\data"
#endif
```

A `UNIX` konstans létezésekor a `DATADIR` értéke „`/uxn1/data`” lesz, egyébként pedig „`\usr\data`”. Ahogy a kódrészletből is látszik, a sorkezdő `#` után (az előfeldolgozó-utasítás elő) tehetünk szóközöt.

Az `#ifdef`, `#else` és `#endif` utasítások a várakozásnak megfelelően működnek. Ha az `#ifdef` által megadott konstans értékét már (egy korábbi `#define`-nal, vagy a programfordításkor a parancssorban) definiáltuk, akkor az ezt követő `#else`, `#elif` vagy `#endif` sorig terjedő utasítások hajtódnak végre. (Ha definiálattan a keresett konstans, akkor figyelmen kívül hagyja az előfeldolgozó az említett kódrészletet.)

A UNIX konstans definiálásához elég egy

```
#define UNIX 1
```

utasítás, sőt ennyi is elég:

```
#define UNIX
```

A legtöbb fordítóprogram azt is lehetővé teszi, hogy a fordítás elindításakor parancssori paramétereket adjunk meg. A

```
gcc -D UNIX program.c
```

parancssori utasítás a fordítás megkezdésekor egyben definiálja a UNIX állandót is. Ennek hatására az `#ifdef UNIX` feltételvizsgálat igazként értékelődik ki. (Fontos, hogy a `-D UNIX` paramétert még az állomány neve előtt megadjuk.) Ily módon a forráskódba való szerkesztés *nélkül* adhatunk meg konstansokat.

Értéket is adhatunk frissen definiált konstansainknak a parancssorból:

```
gcc -D GNUDIR=/c/gnustep program.c
```

A fenti utasítás a GNUDIR konstansnak adja a `/c/gnustep` értékét, mielőtt meghívásra kerülne a `gcc` fordítóprogram.

Fejlécállományok többszörös beemelésének elkerülése

Az `#ifndef` utasítás igen hasonlít az `#ifdef`-hez; mindkettőt ugyanolyan szintaktikával használhatjuk. A feltételvizsgálat viszont akkor értékelődik ki igazként, ha *nincs* definiálva a keresett konstans. Az `#ifndef`-et általában azért használjuk, hogy elkerüljük a fejlécállományok többszörös behívását. Ha egy fejlécállomány programozójaként biztosak vagyunk lenni abban, hogy őt magát eddig még nem emelték be, akkor egy olyan egyedi azonosítóra célszerű rákérdezni, amit a későbbiekben ugyanott definiálunk. Tekintsük például a következő kódrészletet:

```
#ifndef _MYSTUDIO_H
#define _MYSTUDIO_H
...
#endif /* _MYSTUDIO_H */
```

Tegyük fel, hogy ezt a `mystdio.h` fájlban tároljuk. A fejlécállomány beemelése a szokásos módon történhet:

```
#include "mystdio.h"
```

A fájlon belüli `#ifndef` megvizsgálja, hogy megadtuk-e már a `_MYSTDIO_H` konstansot. Ha most emeljük be először a fejlécállományt, akkor ez nyilván nincs definiálva (honnán máshonnan is lehetne), így az `#ifndef` és a hozzá tartozó `#endif` közti sorokat végrehajtja az előfeldolgozó. Itt feltételezhetően az összes olyan definíció és más utasítás „elhangzik”, amire a programban támaszkodni szeretnénk. Figyeljük meg, hogy rögtön az elején megadjuk a `_MYSTDIO_H` konstanst. Ha netán egy későbbi ponton újra megkísérelné a program beemelni a `mystdio.h`-t, akkor a `_MYSTDIO_H` már létezik, így a megfelelő `#endif`-ig (ami feltételezhetően az egész fájt jelenti) figyelmen kívül maradnak az utasítások. Ezzel jól kivédhető a többszörös csere.

Ugyanezen a módon járnak el a rendszer-fejlécállományok is a többszörös meghívás elkerülésére. Nézzünk meg néhányat!

Az `#if` és az `#elif` utasítás

Az `#if` előfeldolgozó-utasítás az előzőeknél is általánosabb fordítás-szabályozást tesz lehetővé. Megvizsgálhatjuk vele, hogy egy konstans kifejezés nullától eltérő értéket ad-e. Ha nem nulla az érték, akkor a következő `#else`, `#elif` vagy `#endif` utasításig feldolgozásra kerül a forráskód; egyébként figyelmen kívül marad. Példánkban tekintsünk egy olyan helyzetet, amikor az OS konstansban megadjuk a használt operációs rendszer kódszámát. Legyen ez 1-es, ha Macintosh OS-t használunk, 2 a Windows esetén, 3 pedig Linux használatakor stb. Az OS konstansra építve a következő feltételes fordítási kifejezéseket fogalmazhatjuk meg:

```
#if      OS == 1  /* Mac OS */
...
#elif   OS == 2  /* Windows */
...
#elif   OS == 3  /* Linux  */
...
#else
...
#endif
```

A legtöbb fordítóprogram megengedi, hogy a parancssorból (például a `-D` kapcsolóval) értéket rendeljünk hozzá egy-egy konstanshoz (esetünkben az `OS`-hez). A

```
gcc -D OS=2 program.c
```

utasítás 2-es OS-sel fordítja le a `program.c`-t. Ennek hatására a Windows számára megadott adatokkal fordul le a program.

A `defined` (név) feltétel is használható az `#if` utasításokban. A következő két kódrészlet egyenértékű:

```
#if defined (DEBUG)
...
#endif
```

illetve

```
#ifdef DEBUG
...
#endif
```

Az alábbi kódrészlet a `BOOT_DRIVE`-ot „C:/”-ként definiálja, ha a `WINDOWS` vagy `WINDOWSNT` definiált, egyébként pedig „D:/” lesz a konstans értéke:

```
#if defined (WINDOWS) || defined (WINDOWSNT)
# define BOOT_DRIVE "C:/"
#else
# define BOOT_DRIVE "D:/"
#endif
```

Az `#undef` utasítás

Előfordulhat, hogy egy korábban definiált konstanst meg szeretnénk szüntetni. Ez az `#undef` utasítással tehető meg egy adott név-re vonatkozóan:

```
#undef név
```

Így az

```
#undef WINDOWS_NT
```

visszavonja a `WINDOWS_NT` konstanst. Az ezt követő `#ifdef WINDOWS_NT` vagy `#if defined (WINDOWS_NT)` utasítások hamisként értékelődnek ki.

Ezzel az előfeldolgozó lehetőségeit taglaló fejezetünk végére értünk. Láthattuk, hogy miként hozhatunk létre az előfeldolgozási utasítások segítségével könnyebben olvasható, megírható és módosítható programokat. Arról is szó volt, hogy különálló beemelhető fájlokban tárolhatjuk (és több programban is könnyen felhasználhatjuk) a szükséges konstansokat, definíciókat, deklarációkat. Néhány előfeldolgozó-utasításról nem esett szó – ezeket megtaláljuk az A függelékben.

A következő fejezetben mélyebben megvizsgáljuk az adattípusokat és a típusátalakítások lehetőségeit. Előtte azonban oldjuk meg a hátralevő gyakorlatokat.

Gyakorlatok

1. Gépeljük be és futtassuk a fejezetben szereplő három példaprogramot – ne felejtjük el a 13.3 Listához tartozó beemelendő állományról sem! A kimeneteket hasonlítsuk össze a könyvben szereplő programkimenetekkel.
2. Keressük meg rendszerünkön a `<stdio.h>`, a `<limits.h>`, és a `<float.h>` fejléc-állományokat. (Unix rendszereken ezek többnyire a `/usr/include` könyvtárban találhatóak.) Figyeljük meg, mi minden található bennük!
3. Írjuk meg a `MIN` nevű makrót, amely kiszámítja két megadott érték minimumát! Írunk programot is, amely felhasználja a makródefiníciót.
4. Írunk `MAX3` néven egy makrót, amely kiszámítja három megadott érték maximumát! Írunk programot, amely felhasználja ezt a makrót.
5. Írunk `SHIFT` néven egy makrót, amely ugyanazt a feladatot látja el, mint a 12.3 Lista `shift` függvénye!
6. Írunk `IS_UPPER_CASE` néven egy makrót, amely a nagybetűs karakterekre nemnulla értéket ad vissza!
7. Írunk `IS_ALPHABETIC` néven egy makrót, amely a betű típusú karakterekre nemnulla értéket ad vissza! Használjuk a fejezetben olvasható `IS_LOWER_CASE` makródefiníciót, valamint a 6. gyakorlatban megírt `IS_UPPER_CASE` makrót!
8. Írunk `IS_DIGIT` néven egy makrót, amely a szám típusú ('0', '1', ...'9') karakterekre nemnulla értéket ad vissza! Ennek felhasználásával írunk még egy makrót `IS_SPECIAL` néven, amely akkor ad vissza nullától eltérő értéket, ha „speciális karaktert” kap, azaz nem betű és nem számot. Ehhez a 7. gyakorlat `IS_ALPHABETIC` makróját is célszerű felhasználni.
9. Írunk `ABSOLUTE_VALUE` néven egy makrót, amely a paraméter abszolút értékét adja vissza! Vigyázzunk, hogy az `ABSOLUTE_VALUE` (`x + delta`) típusú utasítások is helyesen működjenek.
10. Tekintsük a fejezetben már megfogalmazott `printint` makrót:

```
#define printint(n) printf ("%i\n", x ## n)
```

Használható-e a következő kódrészlet az `x1..x100` változók kiíratására? Miért?

```
for (i = 1; i < 100; ++i)
    printx (i);
```
11. Vizsgáljuk meg azokat a könyvtári függvényeket, melyek egyenértékűek a 6., 7. és 8. gyakorlatban megírt makrókkal. Ezek a következők: `isupper`, `isalpha` és `isdigit`. Ezeket a függvényeket a `<cctype.h>` rendszer-fejlécállomány beemelésével tudjuk felhasználni programjainkban.

Mélyebben az adattípusokról

Ebben a fejezetben szó lesz egy eddig még nem használt adattípusról, a *felsorolt* típusról. Megtanuljuk azt is, miként lehet saját (vagy származtatott) adattípusokat megadni a `typedef` kulcsszóval. Végül részletesen megvizsgáljuk, hogy a fordítóprogram milyen módon végzi el a típusáltalakításokat a különféle kifejezésekben.

A *felsorolt* (enumerated) típus

Milyen jó is lenne, ha előre megadhatnánk, hogy egy változó minden értéket vehessen fel! Legyen például egy `myColor` nevű változónk, melyben csak az alapszínek tárolását szeretnénk megengedni, és más színét nem. Ezt a „testreszabást” a felsorolt típus segítségével valósíthatjuk meg.

A felsorolt típust az `enum` kulcsszóval adhatjuk meg. Rögtön a kulcsszó után következik a változó neve, majd pedig a megengedett értékek felsorolása kapcsos zárójelben (vesszővel elválasztva). A következő utasítással hozhatjuk létre a példaként említett változóhoz szükséges `primaryColor` felsorolt típust:

```
enum primaryColor { red, yellow, blue };
```

Az ilyen típusú változók értéke csak `red`, `yellow` és `blue` lehet (azaz az alapszínek), és semmi más. Legalábbis elméletben. A legtöbb fordítóprogram hibát jelez, ha más értéket próbálunk meg értékül adni egy ilyen változónak – azonban vannak fordítóprogramok, amelyek eltűrik más értékek hozzárendelését is.

Egy (vagy több) változó `enum primaryColor` típusuként való definiálásához ismét használnunk kell az `enum` kulcsszót, majd a típus neve következik (`primaryColor`), végül a változónevek listája:

```
enum primaryColor myColor, gregsColor;
```

Ez az utasítás a `myColor` és a `gregsColor` változókat `primaryColor` típusúként hozza létre. A megadott változókhöz csak a `red`, `yellow` és `blue` értékeket lehet hozzárendelni, például így:

```
myColor = red;
```

Természetesen a feltételvizsgálat is csak ezekre vonatkozhat:

```
if ( gregsColor == yellow )
    ...

```

Egy másik példa lehet a hónapok felsorolása. Hozzunk létre egy `enum month` típust, amellyel az év hónapjait tartalmazó változókat definiálhatunk:

```
enum month { january, february, march, april, may, june,
             july, august, september, october, november, december };
```

A fordítóprogram a felsorolt változókat (egymást követő) egész értékekkel tekinti. Az első névhez 0-t, a másodikhoz 1-et rendel stb. Ha a programunkban szerepel a következő két sor:

```
enum month thisMonth;
...
thisMonth = february;
```

akkor a `thisMonth` változó értéke 1 lesz (nem pedig „february”), mivel ez az azonosító a második az adott felsorolt típusban.

Arra is van lehetőség, hogy eltérjünk ettől a folytonos számozástól. A típus megadásakor számértéket is rendelhetünk az egyes azonosítókhöz – az ezt követő többi azonosító pedig minden egyet nagyobb értéket kapnak (hacsak mást nem adunk meg). Tekintsük például a következő definíciót:

```
enum direction { up, down, left = 10, right };
```

Itt a `direction` (irány) adattípust az `up`, `down`, `left` és `right` értékek felsorolásaként adjuk meg. A fordítóprogram 0-t rendel az első azonosítóhoz, 1-et a másodikhoz, 10-et a `left`-hez (hiszen megadtuk, hogy annyi legyen), végül az ezt követő `right`-hoz 11-et (ugyanis itt nincs megadva érték, így az előzőt növeli egyel).

A 14.1 Lista bemutatja a felsorolt típus alkalmazását. A `month` típusban a január 1-es értéket kap, így a hónapok értékei 1-től 12-ig terjednek. A program ezután beolvashat egy egész számot, amelyet a hónapoknak megfelelő `switch` utasítással vizsgál meg. Mivel a felsorolt

változók konstansként tárolódnak, lehet őket case értékekkel használni. A days értéke felveszi a hónapban levő napok számát, amelyet ki is ír a program a switch utasítás után. Ha a hónap február, akkor még egy kiegészítő megjegyzés is megjelenik az esetleges szökövévre vonatkozóan.

14.1 Lista • A felsorolt típus használata

```
// Egy hónap napjai számának megjelenítése

#include <stdio.h>

int main (void)
{
    enum month { january = 1, february, march, april, may, june,
                 july, august, september, october, november,
                 december };
    enum month aMonth;
    int         days;

    printf ("Adja meg a hónap számát: ");
    scanf ("%i", &aMonth);

    switch (aMonth) {
        case january:
        case march:
        case may:
        case july:
        case august:
        case october:
        case december:
            days = 31;
            break;
        case april:
        case june:
        case september:
        case november:
            days = 30;
            break;
        case february:
            days = 28;
            break;
        default:
            printf ("Hibás érték\n");
            days = 0;
            break;
    }

    if ( days != 0 )
        printf ("A napok száma %i\n", days);
}
```

```

    if ( aMonth == february )
        printf (...vagy 29, ha szökőév van\n);

    return 0;
}

```

14.1 Lista • Kimenet

Adja meg a hónap számát: **5**
A napok száma 31

14.1 Lista • Kimenet (újrafuttatás)

Adja meg a hónap számát: **2**
A napok száma 28
...vagy 29, ha szökőév van

A felsorolt azonosítók értéke meg is egyezhet. Használhatjuk például a következő definíciót:

```
enum switch { no=0, off=0, yes=1, on=1 };
```

Ezek után a no és az off értéke is 0-t takar, valamint a yes és az on 1-est.

Ha egy egész számot szeretnénk hozzárendelni egy felsorolt típusú változóhoz, akkor típusátalakító operátort kell használnunk. Az alábbi utasítás például 5-öt rendel a thisMonth-hoz:

```
thisMonth = (enum month) (monthValue - 1);
```

A felsorolt változók használatakor lehetőleg arra törekedjünk, hogy a program ne támaszkodjon erősen arra a tényre, hogy ezek igazából csak egész számok. Tekintsük inkább őket külön adattípusnak. A felsorolt típus révén szimbolikus nevekkel hivatkozhatunk bizonyos számokra. Ha ezt a számértéket mi szeretnénk megadni, azt csak a definíciókor tehetjük meg. Amikor különböző feltevésekkel élünk a felsorolt típus azonosítóit illetően, akkor ezzel lemondunk a felsorolt típus egyszerűségéből eredő szépségről.

A felsorolt típus megadásakor hasonló lehetőségeink vannak, mint az adatszerkezetek esetében: elhagyható a típus neve, és a deklarálandó változónév közvetlenül a felsorolás után is megadható. Ezt a két lehetőséget egyszerre használja ki a következő utasítás:

```
enum { east, west, south, north } direction;
```

Itt egy névtelen felsorolt típust adunk meg, amelyet a direction változó fog képviselni. Értékei lehetnek: east, west, south és north.

A felsorolt típus definíciója a hatókör szempontjából hasonlít a változók és az adatszerkezetek definíciójához. Egy utasításblokkon belül történő definíció csak az adott blokkon belül lesz érvényes. Ha azonban minden függvényen kívül, a program elején adjuk meg a definíciós utasítást, akkor az adott típus globális lesz – mindenütt használhatjuk.

A felsorolt típus megadásakor ügyeljünk arra, hogy ne használjunk olyan azonosítónevet, amelyet más változó (vagy felsorolt azonosító) is használ az adott hatókörön belül.

A `typedef` utasítás

A C nyelv lehetővé teszi, hogy a `typedef` kulcsszó segítségével az adattípusoknak saját nevet adjunk. A következő utasítás a `Counter` (*Számláló*) nevet egyenértékűvé teszi a C `int` típusával:

```
typedef int Counter;
```

Ettől kezdve `Counter` típusú változókat is megadhatunk:

```
Counter j, n;
```

A fordítóprogram persze igazából egésznek tekinti a `j` és az `n` változót. Az új típusnév előnye az olvashatóság és érthetőség, hiszen ebből nyilvánvalóan látszik, milyen szerepe lesz a fenti változóknak. Ha egyszerűen csak `int`-ként deklaráltuk volna őket, abból nem derülne ki a program olvasója számára, hogy milyen célból definiáljuk e változókat. Természetesen a változónevek beszédes megválasztása is hasonlóan célravezető.

A `typedef` használatát sok esetben egyenértékű módon megoldhatjuk egy alkalmas `#define` utasítással. A fenti `typedef` utasítás helyett a következő kód részlet is állhatna:

```
#define Counter int
```

Mivel a `typedef` utasítás (az előfeldolgozó helyett) a C fordítóprogram hatáskörébe tartozik, összetettebb típusdefiníciós megoldásokat tesz lehetővé, mint a `#define`. A következő `typedef` utasítás például a `Linebuf` karaktertömböt 81 karakter hosszúságúra definiálja:

```
typedef char Linebuf [81];
```

Ezután már `Linebuf` típusú változók is létrehozhatóak:

```
Linebuf text, inputLine;
```

A `text` és az `inputLine` változó is 81 karakterre méretezett tömb lesz. Jó tudni, hogy ugyanezt az eredményt *nem* érhettük volna el `#define` használatával.

A következő `typedef` utasítás a `StringPtr` típust karakter-mutatóként definiálja:

```
typedef char *StringPtr;
```

Ezt követően minden `StringPtr`-ként megadott változót karakter-mutatóként értelmez a fordítóprogram, például az alábbi `buffer`-t is:

```
StringPtr buffer;
```

A következő módon definiálhatunk egy típust a `typedef` segítségével:

1. Írjuk meg a definiáló utasítást úgy, mintha már létezne a kívánt típus.
2. A létrehozandó változó nevének a helyére írjuk be az új típusnevet.
3. Mindez előre írjuk oda a `typedef` kulcsszót.

Lássunk egy példát erre. Hozzunk létre egy `Date` nevű típust: egy olyan adatszerkezet, amely három egész adattagot (hónapot, napot és évet) tartalmaz. A változó neve helyére (a pontosvessző előtt) írjuk oda a `Date` szót. Mindezek előtt pedig írjuk oda a `typedef` kulcsszót.

```
typedef struct
{
    int    month;
    int    day;
    int    year;
} Date;
```

Ezzel a típusdefinícióval már létre is hozhatóak különféle változók, tömbök. Száz `Date` adatszerkezetű elemből álló tömbként így adhatjuk meg a `birthdays`-t:

```
Date birthdays[100];
```

Amikor olyan programokkal dolgozunk, melyek forráskódját egynél több fájlban tároljuk (bővebben erről lásd a 15. fejezetet), akkor célszerű a gyakran használt típusdefiníciókat egy gyűjtőfájlban elhelyezni, amit be lehet hívni egy alkalmas `#include` utasítással.

Másik példaként tekintsünk egy olyan grafikai programot, amely például körök és egyenesek rajzolását teszi lehetővé. A program erőteljesen támaszkodik a koordinátarendszer használatára.

Az alábbi típusdefinícióval létrehozható egy Point típus, amely egy x és egy y lebegő pontos adattagot tartalmaz:

```
typedef struct
{
    float x;
    float y;
} Point;
```

Ezzel a típusdefinícióval neki lehet kezdeni egy grafikai programkönyvtár kialakításának, és illesfele deklarációkat lehet létrehozni:

```
Point origo = { 0.0, 0.0 }, currentPoint;
```

Az origo és a currentPoint (*aktuális pont*) Point típusú változó lesz; az origo koordinátáit be is állítjuk az origóba, a (0;0) pontba.

A distance függvény kiszámítja két pont távolságát:

```
#include <math.h>

double distance (Point p1, Point p2)
{
    double diffx, diffy;

    diffx = p1.x - p2.x;
    diffy = p1.y - p2.y;

    return sqrt (diffx * diffx + diffy * diffy);
}
```

Korábban említettük, hogy a négyzetgyök (*square root*) vonását elvégző könyvtári függvény a sqrt. Ez a math.h fejlcállományban található, amit a kód részlet elején be is emelünk.

Ne feledjük, hogy a typedef utasítással nem hozunk létre új típust, csak új nevet adunk egy már létező típusnak. A szakasz elején említett j és n Counter változók tehát egyszerű egész számok lesznek a fordítóprogram számára.

Adattípusok átalakítása

A 4. fejezetben az adattípusok kapcsán röviden volt már szó arról, hogy a rendszer automatikus típusátalakításokat is végez a különféle kifejezések kiértékelésekor. A lebegőpontos és az egész típusok közti átalakításokat vizsgáltuk. Láttuk, hogy ha egy lebegőpontos és egy egész érték közt végzünk műveletet, akkor a művelet lebegőpontosként hajtódik végre; azaz kiértékelés közben az egész adat lebegőpontossá alakul.

Azt is megtanultuk, hogy a típusátalakító operátorral is megadhatunk egy-egy átalakítást. Így például az

```
atlag = (float) osszeg / n;
```

utasítás hatására például az osszeg lebegőpontossá alakul a művelet elvégzése előtt, és így lebegőpontos osztás eredménye kerülhet az atlagba.

A C fordítóprogram szigorú szabályok szerint végzi el a kifejezésekben található típusok átalakítását.

1. Ha bármelyik operandus long double, akkor a többi operandus és az eredmény is long double lesz.
2. Ha bármelyik operandus double, akkor a többi operandus és az eredmény is double lesz.
3. Ha bármelyik operandus float, akkor a többi operandus és az eredmény is float lesz.
4. Ha bármelyik operandus _Bool, char, short int, bitmező vagy felsorolt típus, akkor int típusúvá alakul.
5. Ha bármelyik operandus long long int, akkor a többi operandus és az eredmény is long long int lesz.
6. Ha bármelyik operandus long int, akkor a többi operandus és az eredmény is long int lesz.
7. Ha elérünk eddig a lépésig, akkor az operandusok már csak int típusúak lehetnek; ilyenkor az eredmény is int lesz.

Ez igazából egy egyszerűsített változata a ténylegesen lezajló kifejezés-átalakító műveletsornak. A szabályok akkor bonyolódnak el, amikor előjel nélküli operandusok lépnek föl. A teljes szabálygyűjtemény az A függelékben található („A C nyelv összefoglalása”).

Amikor a fenti lépéssorban elérünk egy olyan tagmondatot, hogy „...az eredmény ilyen típusú lesz”, akkor véget ér a típusátalakítás folyamata.

Nézzünk egy példát. Legyen f egy float, i egy int, l egy long int és s egy short int típusú („beszédes nevű”) változó. Értelmezzük a következő kifejezést:

$f * i + l / s$

Először f-et szorozzuk i-vel, ami egy float és egy int közti művelet. A fenti 3. lépés szerint f float típusa miatt i-t (és a szorzás eredményét) is float-nak tekinti a fordítóprogram. Ezután következik l osztása s-sel, ami egy long int és short int közti művelet. A 4-es lépés szerint a short int átalakul int-té. Továbbhaladva azonban az átalakítási szabályok sorában az derül ki a 6. lépésekben, hogy – mivel az l operandus long int – a másik (s) is ilyenné alakul, sőt az eredmény is long int lesz. Vagyis az osztás egy long int eredményt ad, így csak a hánnyados egész része őrződik meg.

Végül a 3. lépés jut szerephez: azaz, ha valamelyik operandus float (mint ahogy f * i is az), akkor a másik operandus és az eredmény is az lesz. Vagyis a kiszámított 1/s hánnyados float típusúá alakul, és így adódik f és i szorzatához. A kifejezés végeredménye tehát float típusú lesz.

Jó tudni, hogy a típusáltalakító operátor minden felhasználható a kifejezésekben – segítségevel szándékunknak megfelelően vezérelhetjük a típusok átalakítását.

Ha tehát el szeretnénk kerülni, hogy az egészosztással elvesszen a hánnyados törtrésze, akkor kényszerítenünk kell a lebegőpontos osztást:

$f * i + (\text{float}) l / s$

Ebben a kifejezésben az l értéke lebegőpontossá alakul az osztás végrehajtása előtt. A típusáltalakító operátor ugyanis magasabb precedenciájú, mint az osztás. Ha pedig a művelet egyik operandusa lebegőpontos, akkor a másik operandus és az eredmény is az lesz.

Az előjel kiterjesztése

Amikor egy előjeles signed int vagy signed short int nagyobb méretű egésszé alakul, akkor az előjelet átvisszük a bal oldali bitre. Ezzel oldjuk meg azt, hogy például egy short int típusú változó -5 értéke megmaradjon long int típusúként is. Az előjel nélküli egészek átalakításakor nem történik előjel-kiterjesztés.

Néhány számítógép-rendszeren (például a Mac G4/G5 vagy a Pentium processzorok esetében) a karakteres változók előjeles mennyiségeként vannak eltárolva. Ez azt jelenti, hogy a karakteres változók egésszé alakításakor előjel-kiterjesztés történik. Amíg megmaradunk a szabványos ASCII karakterkészletnél, ez nem okoz gondot. Ha azonban túllépünk ezen

a karakterkészleten, akkor meglepetéseket okozhat az előjel-kiterjesztés. Például egy '\377' értékű karakteres konstans (Mac gépeken) -1-é alakul, hiszen 8 bites előjeles számkként tekintve ennyi az értéke.

A C nyelv lehetővé teszi az előjel nélküli karakteres konstansok használatát. Ezzel elkerülhető a fenti probléma. Egy `unsigned char` változó soha nem szenved előjel-kiterjesztést – akkor is nemnegatív marad, ha számmá alakítjuk. Az előjeles karakterek tipikusan -128-tól 127-ig vehetnek fel értékeket, míg az előjel nélküliek 0-tól 255-ig.

Ha mégis szeretnénk egy karakteres változó előjelét kiterjeszteni típusátalakításkor, akkor `signed char`-ként hozzuk létre. Így olyan számítógépeken is megtörténik az előjel-kiterjesztés, amelyeken alapértelmezetten ez nem valósulna meg.

A paraméterek átalakítása

A könyvben eddig valamennyi példafüggvényhez megadtunk egy prototípus-deklarációt. A 8. fejezetben megtanultuk, hogy a függvény definíciója igen szabadon kezelhető: a függvényhívás előtt és után is elhelyezhető, sőt egy másik állományban is tartható. Arról is szó volt, hogy a fordítóprogram automatikusan átalakítja a paramétereket a megfelelő típusúakká – feltéve, hogy birtokában van annak az információnak, hogy mit is vár az adott függvény. Ez abban az esetben áll fenn, ha a függvény definíciója vagy a prototípus-deklaráció már korábban szerepelt.

Tanultuk, hogy amennyiben a fordítóprogram egy adott függvény meghívásakor még nem találkozott sem a definícióval, sem a prototípus-deklarációval, akkor a függvény visszatérési értékét `int`-nek tekinti.

A fordítóprogram a függvény paramétereinek típusát illetően is feltételezésekkel él, amennyiben erre vonatkozóan nincs pontos információja. A `_Bool`, `char` vagy `short` típusú paramétereket `int`-té, a `float` paramétereket `double` típusúvá alakítja.

Tegyük fel, hogy a fordítóprogram a következő kódrészlettel találkozik:

```
float x;
...
y = absoluteValue (x);
```

Ha korábban nem látta az `absoluteValue` függvénynek sem a definícióját, sem pedig a prototípus-deklarációját, akkor olyan kódot generál, amely az `x` értékét `float`-ból `double` típusúvá alakítja, és ezt adja át a függvénynek. Azt is feltételezi a fordítóprogram, hogy a függvény visszatérési értéke `int` típusú.

Az `absoluteValue` függvény definíciója egy másik állományban található, és így néz ki:

```
float  absoluteValue (float  x)
{
    if ( x < 0.0 )
        x = -x;

    return x;
}
```

Ezzel elég komoly problémába ütközünk. Mi lebegőpontos visszatérési értéket szeretnénk, a fordítóprogram viszont egészet visszaadó kódot generál. A függvényt a definícióban `float` típusú paraméter használatára készítettük fel, a fordítóprogram viszont `double`-t fog neki átadni.

Az ilyen típusú hibalehetőségek miatt szeretnénk kiemelni annak fontosságát, hogy mindenkor először elhelyezni a prototípus-deklarációt a függvény meghívása előtt. Ezzel megelőzhetjük azt, hogy a fordítóprogram téves feltevésekkel éljen a paraméter(ek)et vagy a visszatérési értéket illetően.

Ebben a fejezetben sok érdekes részletet megtanultunk az adattípusokról – ideje megnézni, hogyan tudunk olyan programokkal dolgozni, melyek több állományt használnak. Erről fog szólni következő fejezetünk. Mielőtt azonban ebbe belevágnánk, érdemes néhány gyakorlat elvégzésével tisztázni, hogy valóban megértettük-e a fejezetben tanult fogalmakat.

Gyakorlatok

1. A `typedef` segítségével definiáljuk a `FunctionPtr` függvénymutató típust, amely nem vár egyetlen paramétert sem, és visszatérési értéke egy egész szám. A 11. fejezetből kiderül, hogy hogyan lehet függvénymutatókat megadni.
2. Írunk függvényt `monthName` néven, amely paraméterként egy `enum month` felsorolt típusú változót vár (a hónap számát), visszatérési értéke pedig egy karakterlánc-mutató, mely a hónap nevét tartalmazó karakterláncra mutat. Így az alábbi módon jeleníthető meg egy `enum month` változóban tárolt hónap neve:

```
printf ("%s\n", monthName (aMonth));
```

3. Tekintsük az alábbi változókat:

```
float      f = 1.00;
short int  i = 100;
long int   l = 500L;
double     d = 15.00;
```

A fejezetben található hétlépéses típusátalakítási szabálygyűjtemény alapján állapít-suk meg az alábbi kifejezések típusát:

```
f + i  
l / d  
i / l+f  
l * i  
f / 2  
i / (d + f)  
l / (i * 2.0)  
l + i / (double) l
```

Nagyobb programok

A könyvben eddig szereplő programok kicsik és egyszerűek voltak. A való élet problémáinak megoldására szolgáló programok azonban rendszerint nagyok és bonyolultak. Az ilyen programok írásának munkamódszere lesz ennek a fejezetnek a témája. A C nyelv természetesen biztosítja mindeneket a lehetőségeket, amik egy nagyméretű program hatékony fejlesztéséhez szükségesek. Különféle segédprogramokat is használhatunk (ezekről röviden említést is teszünk), melyek megkönnyítik a nagyobb projektekkel történő munkát.

Egy program fájlokra bontása

Az eddigi programoknál többnyire azt feltételeztük, hogy az egész forráskód egyetlen állományba van begépelve valamilyen szerkesztőprogrammal (emacs-szel, vim-mel, geanyvel vagy valamelyik Windows alatt futó szerkesztővel). Ezután már csak le kellett fordítani, és következhetett a futtatás. A program által használt valamennyi függvény (a scanf, printf... könyvtári függvények kivételével) ebben a fájlban volt. Néhány szabványos fejlecállományt (<stdio.h>, <stdbool.h>) kellett csak beemelnünk bizonyos definíciók és függvény-deklarációk eléréséhez. Ez a megközelítés kisebb programok esetében remekül működik – százegyhány utasításig. A nagyobb programoknál azonban akadályokba ütközünk. Az utasítások számának növekedtével a program szerkesztéséhez szükséges idő (és a fordítás ideje) is növekszik. Ráadásul a komolyabb alkalmazások gyakran több programozó munkáját igénylik. Nehezen lenne megszervezhető, hogy mindenki ugyanazon a forrásfájlon dolgozzon (és a másolatok készítése sem egyszerű megoldás).

A C nyelv támogatja a moduláris programozást. Eszerint nem kell egy program összes utasítását egyetlen fájlból tárolni: az egyes részek, modulok kódsorait különböző állományokban is elhelyezhetjük. A *modul* szó alatt egy vagy több függvény értendő, amelyeket logikailag összetartozónak gondolunk.

Ha valamilyen grafikus projektkezelő eszközzel dolgozunk, mint például a Metrowerks *CodeWarrior*, a Microsoft *Visual Studio*-ja vagy az Apple *Xcode*-ja, akkor leegyszerűsödik a forrásfájlok kezelése. Csak meg kell adni a projekthez tartozó állományokat – az In-

tegrált Fejlesztői Környezet (*Integrated Development Environment: IDE*) elvégzi a munka többi részét. A következő részben arról lesz szó, hogy hogyan lehet ilyen eszköz *nélkül* több forrásfájllal dolgozni. Vagyis azt vizsgáljuk meg, hogy parancssori eszközökkel (gcc, cc) hogyan lehet nagyobb programokat lefordítani.

Összetartozó forrásfájlok lefordítása parancssorból

Tegyük fel, hogy egy programot három modulra bontottunk. Az elsőhöz tartozó utasításokat a mod1.c-be, a másodikhoz tartozókat a mod2.c-be, magát a főprogramot pedig a main.c-be írtuk. Úgy adhatjuk tudtára a rendszernek hogy ezek az állományok összetartoznak, hogy felsoroljuk őket egymás után a fordítás megkezdésekor:

```
$ gcc mod1.c mod2.c main.c -o dbtest
```

A fenti utasítás hatására a három forrásfájl (mod1.c, mod2.c, main.c) tartalmát külön-külön értelmezi a fordítóprogram. Ha valamelyikben hibát észlel, azt külön is jelzi. A következőképp nézhet ki például a fordítás kimenete:

```
mod2.c:10: mod2.c: In function 'foo':  
mod2.c:10: error: 'i' undeclared (first use in this function)  
mod2.c:10: error: (Each undeclared identifier is reported only once  
mod2.c:10: error: for each function it appears in.)
```

Eszerint a mod2.c 10. sorában levő foo függvényben hiba van. Mivel a mod1.c-re és a main.c-re vonatkozóan semmilyen visszajelzést nem látunk, biztosak lehetünk abban, hogy azok rendben lefordultak.

Általában annak a modulnak a kijavításával érdemes kezdeni a munkát, amelyikre vonatkozóan hibajelzés érkezett.¹

Esetünkben csak a mod2.c-ben volt hiba, így ezt az fájt kell megvizsgálnunk és a téves forráskód-részletet kijavitnunk. A korrekció után újra megkísérelhetjük a fordítást:

```
$ gcc mod1.c mod2.c main.c -o dbtest  
$
```

Nem érkezett hibaüzenet – a lefordított program a dbtest állományba került.

¹ Előfordulhat, hogy az adott modulba beemelt fejlécállományban van a hiba. Ilyenkor természetesen nem a modult kell átszerkeszteni, hanem a hibás fejlécállományt.

A fordítóprogram minden forrásfájlhoz elkészít egy (köztes szerepű) tárgykódú állományt, melynek a neve hasonló a forrásfájléhoz: a mod2.c esetében mod2.o. (A Windows alatt futó fordítóprogramok is hasonlóan működnek, csak a tárgykódú állományok neve .o helyett .obj-ra végződik.) A fordítási folyamat végén ezek a köztes fájlok általában törlődnek. Van néhány C fordító (például a szabványos Unix C fordító is ilyen), melyek történeti okokból nem törlik ezeket a tárgykódú állományokat, ha egynél több forrásfájlt fordítunk egyszerre. Ez jól jön olyankor, amikor csak néhány modulon módosítunk egy keveset. Előző példánkban a mod1.c-ben és a main.c-ben nem volt hiba, így hasznos, ha a keletkezett (mod1.o és main.o) tárgykódú állományok még megvannak a következő fordításkor is. Ha a .c fájlnév-végződést .o-ra cseréljük a fordítási parancsban, akkor a legutóbb keletkezett .o állományt fogja használni a fordítóprogram. A következő parancs olyan fordítóprogrammal (például cc) használható, amely nem törli a köztes állományokat:

```
$ cc mod1.o mod2.c main.o -o dbtest
```

Hibátlan fordítás esetén tehát nemcsak, hogy nem kell belenyűlni a mod1.c-be és a main.c-be, hanem az újrafordításuk is elkerülhető.

Ha olyan fordítóprogramot használunk, amely automatikusan törli a köztes tárgykódú állományokat, akkor is használhatjuk a fokozatos fordítás előnyeit. Ilyenkor egyesével kell lefordítani az egyes forrásfájlokat, mégpedig a -c kapcsolóval. Ennek hatására nem történik meg a linkelés (összefűzés), azaz a végrehajtható program létrehozása, és megmaradnak a köztes állományok. Azaz a

```
$ gcc -c mod2.c
```

utasítás lefordítja a mod2.c-t, és létrehozza a mod2.o tárgykódú állományt.

Általában az alábbihoz hasonló utasítás-sorozattal lehet elérni a három fájlból álló dbtest program fokozatos fordítását:

```
$ gcc -c mod1.c      mod1.c fordítása => mod1.o
$ gcc -c mod2.c      mod2.c fordítása => mod2.o
$ gcc -c main.c      main.c fordítása => main.o
$ gcc mod1.o mod2.o mod3.o -o dbtest
    A végrehajtható állomány létrehozása
```

A három modul külön-külön fordul le. Az utasítások lefutása után nem kaptunk hibajelzést a fordítóprogramról. Ha kaptunk volna, akkor csak a hibás forrásfájlt kellett volna kijavítanunk és újrafordítanunk. Az utolsó sorban csak tárgykódú állományok vannak megadva a fordításhoz, forrásfájl nincs. Ilyen esetben csak linkelés történik, fordítás már nem. A folyamat végén előáll a futtatható dbtest állomány.

Érdemes a fenti példa szerint eljárni nagyobb, sok modulus programjainknál is. Az elkülönített fordítási folyamat hatékony munkát tesz lehetővé a hatalmas programok esetében is. A következő parancs például egy hatmodulos program lefordítását indítja el, melyek közül csak a `legal.c`-t kell újrafordítani:

```
$ gcc -c legal.c Lefordítja a legal.c-t, kimenetét legal.o-ba téve
$ gcc legal.o makemove.o exec.o enumerator.o evaluator.o display.o -o
superchess
```

A fejezet utolsó részében szó lesz arról, hogy a fokozatos fordítást automatizálhatjuk is a `make` program segítségével. A fejezetben említett integrált fejlesztői környezetek mindenike ismeri ezt a technikát, és csak a szükséges állományokat fordítják újra a javításkor.

A modulok közti kommunikáció

A különböző fájlokban tárolt modulok közti hatékony kommunikáció többféle módon is megoldható. Ha az egyik fájlban tárolt függvény meg szeretne hívni egy másik fájlban tárolt függvényt, akkor ennek semmi akadálya: a szokásos módon meghívható a függvény, és a visszatérési érték is normál módon kapható meg. Természetesen a függvényt meghívó állományban *mindig el kell helyezni a megfelelő prototípus-deklarációt, hogy a fordítóprogram ismerje a szükséges függvények paramétereinek és visszatérési értékének a típusát*. Ahogy a 14. fejezetben már volt szó róla, a szükséges információk hiányában a fordítóprogram azt feltételezi, hogy int a függvények visszatérési értéke. Ha nincs erre vonatkozóan több adat, akkor a short és char típusú paramétereket int-té, a float típusúakat double-é alakítja a függvényhíváskor.

Érdemes tudatosítani, hogy a fordítóprogram egyszerre minden csak egyetlen forrásfájllal foglalkozik, és *egymástól függetlenül fordítja le* őket. Akkor is, amikor több lefordítandó állományt adunk meg. Semmiféle információ nem jut át az egyik modulból a másikba a fordítás közben, ami az adatszerkezet-definíciókat, visszatérési értékeket vagy paramétereket jelenti. Teljesen a programozóra van bízva, hogy megfelelő adatokat adjon a fordítóprogram számára, és így az elvárásoknak megfelelően fordulhasson le a program összes modulja.

Külső változók

A különböző állományokban levő függvények *külső változókon* keresztül tudnak egymással kommunikálni. A „külső (external) változók” fogalma tulajdonképpen a 8. fejezetben a függvények kapcsán tárgyalt „globális változók” fogalmának a kiterjesztése.

A külső változókra az jellemző, hogy értékük más modulokból is elérhető és megváltoztatható. Ha egy modulban el szeretnénk érni egy külső változót, akkor a deklarációja előtt kell írnunk az `extern` kulcsszót. Ezzel jelezhetjük a rendszernek, hogy egy másik állomány globális változóját szeretnénk elérni.

Tegyük fel, hogy létre szeretnénk hozni egy `moveNumber` nevű, `int` típusú változót, melynek értékét egy másik állomány függvényéből is el szeretnénk érni és valószínűleg szeretnénk módosítani is. A 8. fejezetben tanultuk, hogy ha az

```
int moveNumber = 0;
```

utasítást minden függvény előtt, a program elején helyezzük el, akkor értéke az egész programban elérhető lesz: így egy globális változót definiáltunk.

Ez a definíció a `moveNumber`-t más állományok számára is elérhetővé teszi: vagyis nem csak globálissá, hanem külső (*external*) globális változóvá avatjuk a fenti változót.

Ha egy másik modulból el szeretnénk érni ennek értékét, akkor a deklarációja elé oda kell írnunk az `extern` kulcsszót:

```
extern int moveNumber;
```

A `moveNumber` értéke most már abból a modulból is elérhető és megváltoztatható, amelyikben ez a deklaráció szerepel. Akár több modulban is használhatjuk ezt a deklarációt – minden hasonlóképpen elérheti majd a változót.

A külső változók használatakor nem szabad elfelejtkezünk egy fontos szabályról: valamelyik forrásállományban *definiálnunk* is kell a külsőnek szánt változót. Ezt két módon is megtehetjük. Elhelyezhetjük a definíciót minden függvényen kívül, az `extern` kulcsszó nélkül:

```
int moveNumber;
```

Itt szabad kezdőértéket megadni, ahogy azt korábban láttuk, de ez nem kötelező.

A másik lehetőség az, hogy `extern` típusmódosítóval definiáljuk a változót (szintén minden függvényen kívül), ilyenkor azonban meg *kell* adni egy kezdőértéket:

```
extern int moveNumber = 0;
```

E két megoldás egyenértékű.

Összefoglalva: amikor egy külső változót használunk, akkor az `extern` kulcsszó a forrásfájljaink egyetlen definíciós helyén hagyható csak el. Ha egyszer sem hagyjuk el, akkor viszont kezdőértéket kell adni a változónak.

Nézzünk egy példát a külső változók használatára. Tegyük fel, hogy a `main.c` program a következőképp néz ki:

```
#include <stdio.h>

int i = 5;

int main (void)
{
    printf ("%i ", i);

    foo ();

    printf ("%i\n", i);

    return 0;
}
```

Az `i` változót úgy definiáltuk, hogy az minden olyan modulban használható lesz, amely megfelelően használja ehhez az `extern` kulcsszót. Legyen a `foo.c` tartalma a következő:

```
extern int i;

void foo (void)
{
    i = 100;
}
```

Ekkor le is fordíthatjuk a két állományt:

```
$ gcc main.c foo.c
```

A program végrehajtása után ez jelenik meg a képernyőn:

```
5 100
```

Vagyis a `foo` függvény valóban el tudja érni, meg tudja változtatni az `i` külső változó értékét.

Mivel az `i` külső változóra a `foo` függvényen *belül* történik hivatkozás, magát a változó-deklarációt is elhelyezhetjük a függvényen belül:

```
void foo (void)
{
    extern int i;

    i = 100;
}
```

Ha a `foo.c` állományon belül több függvény is szeretné használni az `i` értékét, akkor egyszerűbb a deklarációt az állomány legelején megadni. Ha viszont csak egy (vagy kevés) függvénynek van szüksége az `i`-re, akkor érdemesebb egyesével megadni a függvények számára a külső változóra utaló deklarációt. Ez ugyanis a program jobb szervezettségéhez vezet, valamint ennek a különleges változónak a használatát azokra a függvényekre korlátozza, amelyeknek valóban szükségük van rá.

Ha külső változóként tömböt deklarálunk, akkor nem kell megadni a méretét. Az

```
extern char text[];
```

utasítás elérhetővé teszi számunkra a (valahol másutt, külső változóként definiált) `text` tömböt. A formális paraméterként használt többdimenziós tömbökhöz hasonlóan itt is az a szabály, hogy az első dimenziót nem kell megadni, de a többöt igen. Így az

```
extern int matrix[][][50];
```

deklaráció megfelelő ahhoz, hogy ezután hivatkozni tudjunk a kétdimenziós, 50 oszlopot tartalmazó (külső) `matrix` tömbre.

A statikus, illetve a külső változók és függvények összehasonlítása

Megtanultuk, hogy a függvényeken kívül definiált változók nemcsak globálisak, hanem külsők is. Számos olyan helyzet van azonban, amikor egy változót pusztán globálisan szeretnénk használni, nem pedig külső változóként. Vagyis olyan globális változót szeretnénk megadni, amely az adott modulra (forrásfájlra) nézve lokális. Felmerül tehát az igény a változók definiálásának egy olyan módjára is, amely csak az adott modulban levő függvényekből teszi érhetővé a kérdéses változót. A C nyelvben ezt úgy érhetjük el, hogy a változót statikusan (static) definiáljuk.

Ha a

```
static int moveNumber = 0;
```

utasítást minden függvényen kívül adjuk ki, akkor a `moveNumber` változó ettől a definíciótól kezdve mindenhol elérhető az *adott fájlon belül*.

Ha olyan globális változót használunk, amelyet nem kell más állományokból elérni, akkor definiáljuk a változót statikusan. Ez tiszta programozási stílust eredményez. A statikus deklaráció pontosabban tükrözi az adott változó szerepét. Használatával csökkenthető a különböző modulokban definiált külső változók neveinek összekeveredése, ütközése.

Korábban már említettük, hogy egy másik modulban levő függvény meghívása semmi gondot nem okoz. A változókkal ellentében semmilyen külön trükköt nem kell alkalmaz-

ni a távoli függvények eléréséhez. Egy másik fájlban található függvény meghívásához tehát nem kell a függvényt `extern` kulcsszóval deklarálni.

Ha már definiáltunk egy függvényt, akkor *lehet* külsőként vagy statikusként deklarálni – az előbbi az alapértelmezett. Statikus függvényt csak ugyanaból az állományból lehet meghívni, amelyben maga is található. Ha például adott egy `squareRoot` nevű függvényünk, akkor megtehetjük, hogy a függvény deklarációja elé odaírjuk a `static` kulcsszót. Ennek hatására a függvény csak a saját fájljában lesz elérhető:

```
static double squareRoot (double x)
{
    ...
}
```

A `squareRoot` függvény lokálisnak tekinthető arra a fájdra nézve, amelyikben definiáltuk. A fájlon kívül már nem érhető el.

A statikus függvények használata ugyanolyan esetekben javasolt, mint amit a statikus változók esetében megemlíttünk.

A 15.1 ábra vázolja a modulok közti kommunikációt. Két modult ábrázolunk, a `mod1.c-t` és a `mod2.c-t`. A `mod1.c` két függvényt definiál: a `doSquare`-t és a `main`-t. Példánkban a `main` meghívja a `doSquare`-t, ez pedig a `square`-t. Ez utóbbi függvényt a `mod2.c` modulban definiáltuk. Mivel a `doSquare`-t statikusként deklaráljuk, ez csak a `mod1.c`-ből hívható meg, más modulból nem.

```
double x;
static double result;

static void doSquare (void)
{
    double square (void);

    x = 2.0;
    result = square ();
}

int main (void)
{
    doSquare ();
    printf ("%g\n", result);

    return 0;
}
```

mod1.c

```
extern double x;

double square(void)
{
    return x * x;
}
```

mod2.c

15.1 ábra

Modulok közti kommunikáció

A mod1.c-ben két globális változót használunk, az x-et és a result-ot – mindenki tő double típusú. Az x-et bármelyik (mod1.c-vel összelinkelt) modulból elérhetjük. A result előtt álló static kulcsszó következtében ez a változó csak a mod1.c-beli függvényekből (azaz a doSquare-ből és a main-ből) érhető el.

A futtatás kezdetén a main eljárás meghívja a doSquare-t. Ez a függvény 2.0-t rendel az x globális változóhoz, majd meghívja a square függvényt. A square egy másik modulban található (mod2.c-ben), ráadásul nem is int a visszatérési értékének a típusa. Így kényetlennek vagyunk a doSquare elején egy alkalmas deklarációval jelezni, hogy hogyan szeretnékn használni a square függvényt. A square függvény az x (globális változó) négyzetével tér vissza. Mivel a square olyan változót szeretne elérni, amely nem a saját moduljában lett megadva, szándékunkat jelezni kell egy megfelelő extern deklarációval. (Esetünkben lényegtelen, hogy ezt a square függvényen belül vagy azon kívül adjuk meg.)

A doSquare függvényben a result globális változó kapja meg a square visszatérési értékét, amit aztán a függvény visszaad a main eljárásnak. Itt egy alkalmas printf utasítás kiírja az eredményt (result). Példaprogramunk kimenete 4.0-t ad, ami helyes, hiszen ez a 2.0 négyzete.

Tanulmányozzuk a programot, amíg teljesen meg nem értjük. Ez a kicsiny (gyakorlati szempontból teljesen haszontalan) példa jól bemutatja a modulok közti kommunikáció finomságait. Ennek alapos megértése nélkül nem vághatunk bele nagyobb programok megírásába.

Fejlécállományok hatékony használata

A 13. fejezetben az előfeldolgozó kapcsán tanultunk a fejlécállományok használatáról. Megállapítottuk, hogy a gyakran előforduló definíciókat érdemes összefogni egy ilyen állományba. Ezt be tudjuk emelni bármely programba, amelyben használni szeretnénk ezeket a definíciókat. Nincs hasznosabb felhasználási lehetősége az #include utasításnak, mint olyan programokban, amelyet több modulra bontva fejlesztünk.

Ha több programozó dolgozik egy programon, akkor a fejlécállományok egyfajta szabványsírást tesznek lehetővé. minden programozó ugyanazokat a definíciókat, ugyanazokat a kezdőértékeket használhatja. Ráadásul a programozók mentesülnek attól a tehertől, hogy nekik kelljen (egy-egy hibát is vétele) begépelni az egyes definíciókat azokba a fájlokba, melyekben szükség van rájuk. Az itt vázolt előnyök különösen is érvényesek akkor, ha közös adatszerkezet-definíciókat, külső változókhöz tartozó deklarációkat, típusdefiníciókat és függvényprototípus-deklarációkat is tárolunk a fejlécállományokban. A nagyobb programrendszer moduljai általában közös adatszerkezetekkel dolgoznak. Ezeket közös fejlécállomány(ok)ba összefogva kizárhatsuk azt a hibalehetőséget, hogy két modul különböző definíciót használjon ugyanarra az adatszerkezetre. Ha pedig változtatni kell valamelyik adatszerkezet definíóján, akkor azt elég egyetlen helyen megtenni: a fejlécállományban.

Emlékezzünk vissza a 9. fejezetben használt date adatszerkezetre, és nézzük át az alábbi fejlécállományt. Ilyesfélét használhatnánk egy olyan programban, amely sok dátumot használ a különféle modulokban. A példa azért is tanulságos, mert rámutat arra, hogy hogyan érdemes összekapcsolni a témára vonatkozó ismereteinket.

```
// Fejlécállomány a dátumokhoz

#include <stdbool.h>

// Felsorolt típusok

enum kMonth { January=1, February, March, April, May, June,
               July, August, September, October, November, December };

enum kDay { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday };

struct date
{
    enum kMonth month;
    enum kDay     day;
    int          year;
};

// Date típus definíciója
typedef struct date Date;

// A dátumokat kezelő függvények
Date dateUpdate (Date today);
int   numberOfDays (Date d);
bool  isLeapYear (Date d);

// Makró: dátum beállítása egy adatszerkezetben
#define setDate(s,mm,dd,yy) s = (Date) {mm, dd, yy}

// Hivatkozhatunk a todaysDate kúlső változóra
extern Date todaysDate;
```

A fejlécállomány két felsorolt típust ad meg, a kMonth-t és a kDay-t, valamint a date adatszerkezetet. Figyeljük meg a felsorolt típusú változóneveket! A typedef segítségével létrehozunk egy Date típust, valamint néhány alkalmas függvényt, amelyek ezt a típust használják. Egy makrót is láthatunk, amely értékeket rendel egy dátum típusú adatszerkezethez (ehhez összetett betűkonstanst használ – lásd a 9. fejezetben). Elérhetővé teszünk egy kúlső változót todaysDate néven, amely valószínűleg a mai dátumot (*todays date*) fogja tárolni. Ezt valamelyik forrásállományban még definiálni kell ahoz, hogy jól működjön.

Ennek a fejlécállománynak a felhasználásával újrafogalmaztuk a 9. fejezet dateUpdate függvényét:

```
#include "date.h"

// A holnapi dátumot kiszámító függvény

Date dateUpdate (Date today)
{
    Date tomorrow;

    if ( today.day != numberOfDays (today) )
        setDate (tomorrow, today.month, today.day + 1, today.year);
    else if ( today.month == December )          // end of year
        setDate (tomorrow, January, 1, today.year + 1);
    else                                         // end of month
        setDate (tomorrow, today.month + 1, 1, today.year);

    return tomorrow;
}
```

A nagyobb programok írását támogató segédprogramok

Korábban volt már szó az integrált fejlesztői környezetekről, amely hatékony eszköz a nagyméretű programok fejlesztéséhez. Ha valaki azonban szeretne parancssori eszközökkel elbálni méretes programjaival, akkor ahhoz is vannak kitűnő eszközök. Ezek nem részei a C programozási nyelvnek, azonban sok időt takaríthatunk meg velük – ez pedig igen fontos szempont.

Az alábbiakban felsorolunk néhány eszközt, amelyek használatát érdemes megfontolni, ha nagyobb programokat fejlesztünk. Unix alatt számtalan jobbnál jobb eszköz áll rendelkezésünkre, melyek megkönnyítik munkánkat a fejlesztés során; itt csak a jéghegy csúcsát tudjuk bemutatni. A különféle héjprogramozási nyelvek (mint például a Unix héj) szintén igen hasznosak lehetnek akkor, amikor sok állománnyal kell dolgoznunk.

A make segédprogram

Ez a nagy tudású program (vagy annak GNU változata, a gnumake) lehetővé teszi, hogy függőségekkel együtt megadjunk fájlokat a fordításhoz. Ezeket az információkat a *Makefile*-ba kell beírnunk. A make program csak azokat a modulokat fordítja újra, melyek módosultak a legutóbbi fordítás óta. A módszer a fájlrendszerben nyilvántartott módosítási időpontokon alapul. Ha a make úgy találja, hogy egy .c forrásállomány későbbi, mint a hasonló nevű .o tárgykódú állomány, akkor automatikusan kiadja a fordítási parancsot. Ennek hatására létrejön egy friss tárgykódú állomány. Megadhatjuk azt is, hogy az egyes forrásfájlok milyen fejlécállományuktól függjenek. A datefuncs.o modulra vo-

natkozóan például kiköthetjük, hogy a `datefuncs.c`-n kívül még a `date.h` fejlécállománytól is függön. Ha tehát bármi módosul a `date.h`-ban, a `make` automatikusan újrafordítja a `datefuncs.c`-t. Ez egyszerűen azon múlik, hogy az utolsó módosítás időbélyege frissebb a fejlécállományon, mint a forrásfájlon.

Az alábbi Makefile a fejezetben használt hárommodulos példára épül. A fájt ugyanabba a könyvtárba kell helyezni, ahol a forrásfájlok is vannak.

```
$ cat Makefile
SRC = mod1.c mod2.c main.c
OBJ = mod1.o mod2.o main.o
PROG = dbtest

$(PROG) : $(OBJ)
    gcc $(OBJ) -o $(PROG)

$(OBJ) : $(SRC)
```

Nézzük meg részletesen, hogyan is működik ez a Makefile. Megadunk néhány forrásfájlt (`SRC`), a hozzájuk tartozó tárgykódú állományokat (`OBJ`), a futtatható állomány nevét (`PROG`), valamint a függőségeket. Az első függőség így fest:

```
$(PROG) : $(OBJ)
```

Ez azt jelenti, hogy a futtatható fájl a tárgykódú állományuktól függ. Ha tehát valamelyik tárgykódú állomány megváltozik, újra kell építeni a végrehajtható fájlt. Ennek módja (azaz a szükséges `gcc` utasítás) is meg van szabva a Makefile következő sorában, melyet egy tabulátorral kell kezdeni:

```
    gcc $(OBJ) -o $(PROG)
```

A Makefile utolsó sora a következő:

```
$(OBJ) : $(SRC)
```

Ezserint minden tárgykódú állomány függ a megfelelő forrásfájltól. Ha valamelyik forrásfájl megváltozik, elő kell állítani a megfelelő tárgykódú állományt. A `make`-nek beépített szabályrendszere van arra, hogy ezt hogyan kell megoldani.

A make első futtatásakor a következő történik:

```
$ make
gcc -c -o mod1.o mod1.c
gcc -c -o mod2.o mod2.c
gcc -c -o main.o main.c
gcc mod1.o mod2.o main.o -o dbtest
$
```

Micsoda szívmelengető látvány! A make lefordította az összes forrásfájlt, és a keletkezett tárgykódú állományokat összelinkelte egy futtatható állománnyá.

Ha hiba lépett volna fel például a mod2.c fájlban, akkor valahogyan így festene a kimenet:

```
$ make
gcc -c -o mod1.o mod1.c
gcc -c -o mod2.o mod2.c
mod2.c: In function 'foo2':
mod2.c:3: error: 'i' undeclared (first use in this function)
mod2.c:3: error: (Each undeclared identifier is reported only once
mod2.c:3: error: for each function it appears in.)
make: *** [mod2.o] Error 1
$
```

A make hibát talált a mod2.c-ben. Ilyenkor az alapértelmezett művelet a fordítási folyamat megszakítása.

A mod2.c kijavítása után újra elindíthatjuk a make-et:

```
$ make
gcc -c -o mod2.o mod2.c
gcc -c -o main.o main.c
gcc mod1.o mod2.o main.o -o dbtest
$
```

Láthatjuk, hogy a make nem indította el a mod1.c újrafordítását, hiszen tudta, hogy nincs arra semmi szükség. Az ilyen döntésekben rejlik a make igazi ereje és eleganciája.

Ezt az egyszerű példát alapul vehetjük, átszerkeszthetjük későbbi programjainkhoz, ha a make-et szeretnénk használni. Az E függelékben („Források”) találhatunk részletes információt arra vonatkozóan, hogy mi mindenre képes még ez a hathatós segédprogram.

A cvs segédprogram

Ez az egyik legelterjedtebb verziókövető rendszer. Lehetővé teszi a forráskód különböző verzióinak nyomon követését, számon tartja a változtatásokat. Lehetőséget ad arra, hogy átérjünk egy másik (általában korábbi) verzióra, ha a jelenlegi nem (jól) működik. Az is előfordulhat, hogy egy ügyfelünk egy másik változattal dolgozik – az ő kedvénél is szükséges lehet adott pillanatban a verzióváltás. A cvs-szel (*Concurrent Versions Systems*) másolatot kérhetünk a szükséges fájlokról („*check out*”), céljainknak megfelelően megváltoztathatjuk őket, majd pedig visszaküldhetjük („*commit*”) a fájlok frissített változatát a rendszerbe. Ezzel a módszerrel csökkenhető annak a veszélye, hogy több programozó egymással ütköző változtatásokat hajtson végre azonos fájlokban. A cvs segítségével a programozók különféle helyekről dolgozhatnak ugyanazon a forráskód-készleten, mert minden művelet megvalósítható a hálózaton keresztül.

Unix segédprogramok: ar, grep, sed és a többiek

Unix alatt számos remek eszköz áll rendelkezésre, melyek megkönnyítik és hatékonyabbá teszik a fejlesztő munkáját. Az ar segítségével például programkönyvtárakat hozhatunk létre. Ez akkor lehet hasznos, ha van számtalan segédfüggvényünk, melyeket gyakran használunk, és esetleg másokkal is meg szeretnénk osztani. Ha a szabványos matematikai programkönyvtárat szeretnénk használni, azt a -1m linkelési kapcsolóval tudjuk megtenni. Ehhez hasonlóan saját programkönyvtárat is meghatározhatunk a -1lib kapcsolóval. A linkelési fázisban a rendszer átpásztázza a megadott könyvtára(ka)t, és ha van olyan függvény, amit keresünk, akkor azt beépíti a programba.

A grep és a sed akkor segíthet, amikor egy kifejezést vagy egy karakterláncot keresünk a forrásfájlokban, vagy szeretnénk tömeges változtatásokat eszközölni fájljainkon. Egy minimális héjprogramozási tudással könnyen megoldható, hogy a sed-del egy adott karakterláncot lecseréljünk valami másra, mégpedig az általunk meghatározott fájlokban. A grep csak megkeresi a megadott szövegrészt tartalmazó állományokat (és bennük magát a szövegrészt is). Ezzel könnyen megkereshetünk egy változót vagy egy függvényt a forrásfájljaink között, vagy egy makrót a fejlécállományok között. Például a

```
$ grep todaysDate main.c
```

parancsal rákereshetünk main.c-ben a todaysDate karakterláncra. A

```
$ grep -n todaysDate *.c *.h
```

parancs pedig végigpásztázza az összes forrásfájlt és fejlécállományt, keresve a todaysDate karakterláncot. Amelyekben megtalálja, ott (a -n kapcsoló miatt) fájlnévvel és sorszámmal együtt megjeleníti a találatot.

Láttuk, hogy a C nyelv komoly támogatást nyújt ahhoz, hogy programunkat modulokra bontsuk, melyeket külön-külön, egymástól függetlenül le is fordíthatunk. A fejlécállományok biztosítják a „ragasztóanyagot” a különböző modulok között. Segítségükkel megoszthatóak a prototípus-deklaráció, makrók, adatszerkezet-definíciók, felsorolt típusok stb.

Ha integrált fejlesztői környezetet használunk, akkor elég egyszerűen megoldható a modulokra bontott fejlesztés. Maga a fejlesztői környezet követi nyomon, hogy mely fájlokat kell újrafordítani a változtatások következtében. Ha ehelyett parancssori fordítót használunk (például `gcc -t`), akkor többféle döntési lehetőségünk van. Teljesen kézben tarthatjuk a fokozatos fordítás folyamatát, de rá is bízhatjuk ezt egy automatizálási eszközre, például a `make` segédprogramra. A parancssorból végzett hatékony munkát más eszközök is támogatják, például a különféle keresőprogramok, a tömeges változtatást lehetővé tevő programok, valamint a programkönyvtárak létrehozását és karbantartását segítő alkalmazások.

16

A C kimeneti és bemeneti műveletei

A könyv eddigi fejezeteiben az adatok beolvasása és kiírása a terminálon keresztül történt.¹

Ha eddigi programjainkban be akartunk kérni valamilyen információt, akkor a `scanf` vagy a `getchar` függvényt használtuk, az eredményeket pedig a `printf`-fel jelenítettük meg.

A C nyelvben nincs beépített kimeneti/bemeneti (I/O) utasítás, ám ezeket a műveleteket szabványos programkönyvtári függvényekkel meg lehet valósítani.

A `printf`-et használó programjaink elején mindig beemeltünk egy fájlt:

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

Ez a fejlécállomány tartalmazza a szabványos programkönyvtár kimeneti/bemeneti függvényeit és makródefinícióit. Így ezek használatakor be kell emelnünk a `<stdio.h>`-t a programba.

Ebben a fejezetben elmélyítjük ismereteinket a szabványos programkönyvtár több kimeneti/bemeneti függvényéről. Könyvünk kereteibe természetesen nem fér bele valamennyi rendelkezésre álló függvény megtárgyalása. A B függelék („A szabványos C programkönyvtár”) ugyanakkor bőséges anyagot szolgáltat az érdeklődőknek.

Karakterszintű bemenet/kimenet: a `getchar` és a `putchar`

Egyetlen karakter beolvasására a legkényelmesebb megoldást a `getchar` függvény jelenti. Ezt használtuk a `readLine` függvénynél is, amelyet azért írtunk, hogy beolvassunk vele egy terminálablakban begépelt sort. A függvény mindenkor újra és újra meghívta a `getchar` függvényt, amíg egy újsor karakterrel nem találkozott.

¹ A „terminál” szót itt tág értelemben használjuk. Alapvetően azt a felületet (ablakot) jelenti, amelyben a programot futtatjuk, vagy ahol a program kimenete megjelenik. Egyes rendszereken ezt konzolnak hívják.

Van egy ehhez hasonló függvény, amely egyetlen karaktert ír ki: a `neve putchar`.

A `putchar` meghívása igen egyszerű. Egy paramétere van, a kijelzendő karakter. A

```
putchar (c);
```

mehívásakor megjelenik a képernyőn a (`char` típusként definiált) `c` karakter. Hasonlóképp, a

```
putchar ('\n');
```

az újsor karaktert jeleníti meg, amely – mint tudjuk – a következő sor elejére viszi a kurzort.

Formázott kimenet/bemenet: printf és scanf

A könyvben számtalanszor használtuk már a `printf` és a `scanf` függvényeket. Ebben a szakaszban még alaposabban végignézzük azokat a formázási lehetőségeket, melyeket ezekkel a függvényekkel megvalósíthatunk.

Mind a `printf`, mind a `scanf` függvény első paramétere egy karaktermutató, amely egy formátumjelző karakterláncra mutat. Ez hordozza azt az információt, hogy a további paraméterek miként kell megjeleníteni (a `printf` esetében), illetve, hogy a beolvasott adatokat hogyan kell értelmezni (a `scanf` esetében).

A printf függvény

Korábbi példaprogramjainkban megtörtént, hogy a `%` jel és a formátumkarakter közé egyéb karaktereket is elhelyeztünk, így még pontosabban meg tudtuk adni a kimenet formátumát. Az 5.3A Listában például egy egész szám megjelenítésekor megadtuk a *mező-szélességet* is. A `%2i` formátumjelző karakterlánc hatására legalább két karakternyi helyet kap minden kiírt egész szám, mégpedig jobbra igazítva. Az 5. fejezet 6. gyakorlatában említettük, hogy egy minusz jelrel megoldható a balra igazítás.

A `printf` formátumjelző karakterláncának általános alakja a következő:

```
%[kapcsolók][szélesség][.pontosság][hLL]típus
```

A szögletes zárójelben felsorolt információk elhagyhatóak, de ha megadjuk őket, akkor a fenti sorrendben kell szerepelniük.

A 16.1, 16.2 és 16.3 táblázat bemutatja, hogy milyen karakterek adhatóak meg a `%` jel és a típus betűjele között a formátumjelző karakterláncban.

16.1 Táblázat • A printf kapcsolói

Kapcsoló	Jelentése
-	Balra igazított érték
+	A számérték + vagy - jellel kezdődjön
(space)	A pozitív számok szóközzel kezdődjenek
0	Az üres helyeket töltse fel nullákkal
#	Az oktális számokat kezdje 0-val, a hexadecimálisakat 0x-szel; használjon tizedespontot a lebegőpontos számok esetében; hagyjon kezdő nullát a g vagy a G formátumjelzésnél.

16.2 Táblázat • A printf-ben használható mezőszélességi és pontossági adatok

Jelölés	Jelentése
szám	a mezőszélesség legalább ennyi legyen
*	a printf következő paraméterét értelmezze mezőszélességgént
.szám	egészek esetén a számjegyek száma; e és f formátumjelzésnél a tizedesjegyek száma; g formátumjelzésnél az értékes számjegyek száma; s formátumjelzésnél a kiírandó karakterek maximális száma
.*	A printf következő paraméterét értelmezze pontosságént (és alkalmazza az előző sornak megfelelően)

16.3 Táblázat • A printf típusmódosítói

Típus	Jelentése
hh	az egész paramétert jelenítse meg karakterként
h*	jelenítse meg az értéket short integer-ként.
l*	jelenítse meg az értéket long integer-ként.
ll*	jelenítse meg az értéket long long integer-ként.
L	jelenítse meg az értéket long double-ként.
j*	jelenítsen meg intmax_t vagy uintmax_t értéket
t*	jelenítsen meg ptrdiff_t értéket
z*	jelenítsen meg size_t értéket

*Megjegyzés: a csillaggal jelzett típusmódosítók az n formátumjelzés előtt is szerepelhetnek. Ezzel adhatjuk meg, hogy az aktuális mutató milyen típusú adatra mutat.

A 16.4 Táblázatban láthatjuk a formátumjelző karaktereket.

16.4 Táblázat • A printf formátumjelző karakterei

Karakter	Szerepe
i vagy d	(tízes számrendszerbeli) egész szám
u	előjel nélküli egész
o	oktális (nyolcas számrendszerbeli) egész
x	hexadecimális egész, a-f karakterekkel
X	hexadecimális egész, A-F karakterekkel
f vagy F	lebegőpontos szám, alapértelmezetten hat tizedesjeggyel e vagy E normálalakban (azaz „exponenciális formában”) jelenjen meg a lebegőpontos szám (az e hatására kisbetűs, az E hatására nagybetűs exponens-jelzéssel)
g	lebegőpontos szám f vagy e formátumban
G	lebegőpontos szám F vagy E formátumban
a vagy A	lebegőpontos szám 0xd.dddःp±d hexadecimális formátumban
c	egyetlen karakter
s	null-végződésű karakterlánc
p	mutató
n	nem nyomtat ki semmit, hanem az adott függvényhívásban eddig kiírt karakterek számát tárolja el abban az egész számban, amit a vonatkozó paraméter mutat (lásd a 16.3 Táblázat megjegyzését)
%	százalék jel

Az utóbbi négy táblázatban talán nyomasztóan sok az információ. Az minden esetre lát-szik, hogy igen finoman szabályozható a megjelenítendő adatok formátuma. A legegyszerűbb tanulási módszer talán a kísérletezés, vagyis ha számos lehetőséget és kombinációt kipróblálunk. Arra figyelni kell, hogy a formátumjelző karakterlánc % jeleinek a száma megegyezzen a printf többi paramétereinek a számával (természetesen a %% párok kivételével). Abban az esetben, ha a * jelet is használjuk mezőszélesség vagy pontosság beolvasására, akkor a csillagokhoz is vár egy-egy paramétert a printf.

A 16.1 Lista bemutat néhány lehetőséget a printf használatára.

16.1 Lista • A printf formátum-beállítási lehetőségei

```
// Különféle printf megoldások bemutatása
```

```
#include <stdio.h>

int main (void)
{
    char          c = 'X';
```

```
char          s[] = "abcdefghijklmnopqrstuvwxyz";
int           i = 425;
short int     j = 17;
unsigned int   u = 0xf179U;
long int      l = 75000L;
long long int L = 0x1234567812345678LL;
float         f = 12.978F;
double        d = -97.4583;
char          *cp = &c;
int           *ip = &i;
int           c1, c2;

printf ("Integers:\n");
printf ("%i %o %x %u\n", i, i, i, i);
printf ("%x %X %#x %#X\n", i, i, i, i);
printf ("%+i % i %07i %.7i\n", i, i, i, i);
printf ("%i %o %x %u\n", j, j, j, j);
printf ("%i %o %x %u\n", u, u, u, u);
printf ("%ld %lo %lx %lu\n", l, l, l, l);
printf ("%lli %lio %llx %llu\n", L, L, L, L);

printf ("\nFloats and Doubles:\n");
printf ("%f %e %g\n", f, f, f);
printf ("% .2f %.2e\n", f, f);
printf ("% .0f %.0e\n", f, f);
printf ("%7.2f %7.2e\n", f, f);
printf ("%f %e %g\n", d, d, d);
printf ("%.*f\n", 3, d);
printf ("%.*.*f\n", 8, 2, d);

printf ("\nCharacters:\n");
printf ("%c\n", c);
printf ("%3c%3c\n", c, c);
printf ("%x\n", c);

printf ("\nStrings:\n");
printf ("%s\n", s);
printf ("% .5s\n", s);
printf ("%30s\n", s);
printf ("%20.5s\n", s);
printf ("% -20.5s\n", s);

printf ("\nPointers:\n");
printf ("%p %p\n\n", ip, cp);

printf ("Ilyen nincs.\n", &c1, &c2);
printf ("c1 = %i, c2 = %i\n", c1, c2);

return 0;
}
```

16.1 Lista • Kimenet

Integers:

```
425 651 1a9 425
1a9 1A9 0x1a9 0X1A9
+425 425 0000425 0000425
17 21 11 17
61817 170571 f179 61817
75000 222370 124f8 75000
1311768465173141112 110642547402215053170 1234567812345678
1311768465173141112
```

FLOATS and Doubles:

```
12.978000 1.297800e+01 12.978
12.98 1.30e+01
13 1e+01
12.98 1.30e+01
-97.458300 -9.745830e+01 -97.4583
-97.458
-97.46
```

Characters:

```
X
X X
58
```

Strings:

```
abcdefghijklmnopqrstuvwxyz
abcde
abcdefghijklmnopqrstuvwxyz
abcde
abcde
```

Pointers:

```
0xbffffc20 0xbffffbf0
```

Ilyen nincs.

```
c1 = 4, c2 = 12
```

Érdemes elidőznünk ennél a programnál, hogy megfigyeljük, milyen sokoldalúan formázható meg a kimenet. Az első szakaszban egészeket jelenítünk meg: short, long, unsigned és „normál” int típusú számokat. Az első sor az i értékét decimálisan (%i), oktálisan (%o), hexadecimálisan (%x) és előjel nélküli, „unsigned” formátumban (%u) jelemezti meg. Figyeljük meg, hogy az oktális számérték nem 0-val kezdődik.

A következő sor is az i értékét használja fel több különböző változatban. Először a %x hexadecimális jelölésmód szerepel, majd pedig nagy x-et használunk. A kisbetűs x esetén

kis a-f karakterek, nagybetűs x-nél nagy A-F karakterek szerepelnek a hexadecimális számban. A # hatására a hexadecimális számok 0-val és x-szel kezdődnek, a #x esetében kisbetűsen (0x), a #X esetében nagybetűsen (0X).

A negyedik printf a + jel hatására megjeleníti az előjelet (még pozitív számok esetében is, amikor ez nem lenne szükséges). Ezután egy szóköz megadásával arra utasítjuk a printf-et, hogy a pozitív érték előre tegyen egy szóközt. (Időnként ez hasznos lehet, ha pozitív és negatív számokat is kiíratunk, és szeretnénk őket egy vonalban látni. A pozitívak egy szóközzel kezdődnak, a negatívok pedig mínuszjellel.) Ezután a %07 segítségével az i értékét jobbra igazítva jelenítjük meg, hét karakternyi helyen. A 0 miatt nullákkal feltöltve jelennek meg a számok, azaz a 425 előre négy nulla kerül. Ennek a printf-nek az utolsó paraméterét a %.7i-vel jelenítjük meg, azaz i legalább hét számjeggyel fog látszani. Ennek pontosan ugyanaz a hatása, mint a %07i-nek: négy nulla után áll a háromjegyű 425.

Az ötödik printf a j (short int) változót jeleníti meg többféle formában. Bármilyen egész formátum megadható egy short int típusú szám kijelzésekor.

A következő printf azt mutatja be, hogy miként jeleníthetünk meg egy előjel nélküli egész számot. A %i hatására előjeles egészként kellene kiírnia az u értékét. Ez azonban egyes rendszerekben nagyobb a maximálisan használható (signed int) pozitív egész számnál, így negatív számként értelmeződik.

Az utolsó előtti printf hívás az 1 módosítókaraktert szemlélteti egy hosszú egész szám kiírása kapcsán, a legutolsó pedig egy long long int megjelenítését mutatja be.

A formázási lehetőségek sokféleségét bemutató program következő részében a float és a double típusú változók megjelenítését követhetjük nyomon. A kimenet második bekezdésének első sorában egy float típusú változót jelenítünk meg %f, %e és %g formátumban. A %f és %e hatására alapértelmezetten hat tizedesjegy pontossággal jelennek meg a lebegőpontos számok. Ha %g-t adunk meg, akkor a printf dönt arról, hogy (a szám nagyságához és a kívánt pontossághoz jobban illő módon) %e vagy %f formátumban történjen-e a megjelenítés. Ha a kitevő kisebb -4-nél vagy nagyobb a megadott pontosságnál (ez alapértelmezetten 6), akkor a %e formátum valósul meg, egyébként pedig a %f. A szám előtt álló nullák nem jelennek meg, és a tizedespont is csak akkor, ha a számnak van törrész. Általában elmondható, hogy a %g nyújtja a legkellemesebb látványt egy lebegőpontos szám megjelenítésekor.

A kimenet következő sorában a .2-vel két tizedesnyire szabjuk meg a kijelzendő szám pontosságát. Látszik, hogy a printf kellően intelligens ahhoz, hogy az f értékét automatikusan a megfelelő pontosságra kerekítse (ne csak levágja ott a számot).

A következő sorban láthatjuk, mit tesz a .0 pontossági definíció: ilyenkor a %f esetén egészre kerekítve jelennek meg a számok.

A 7.2 formátum-definíció hatására legalább hét karakternyi helyen jelenik meg a megfelelő változó, két tizedesjegy pontossággal. Amennyiben hétnél kevesebb helyen is kifér a szám, akkor balról a megfelelő számú szóközzel feltöltve, jobbra igazítva jelenik meg.

A következő három sorban a `double` típusú `d` változót jelenítjük meg többféle módon. A formátum definíciójában ugyanazok a karakterek adhatóak meg, mint egy `float` típusú számnál, hiszen a `float` értékek is `double` típusúvá alakulnak, amikor paraméterként adjuk őket át egy függvénynek. A

```
printf ("%.*f\n", 3, d);
```

meghívásakor három tizedesjegy pontossággal jelenik meg a `d` értéke. A tizedespont utáni csillag azt jelzi a `printf`-nek, hogy a pontossági értéket paraméterként kell beolvasnia. Esetünkben ez egy konstans érték (3), ám ezt változóban is tárolhatjuk. A

```
printf ("%.*f\n", pontossag, d);
```

utasításhoz hasonló megoldás olyankor lehet hasznos, amikor futási időben szeretnénk meghatározni a kijelzendő szám pontosságát.

A lebegőpontos számkijelzés szakaszának utolsó variációjaként a `d` változót a `%.%` formátumjelző karakterlánc használatával íratjuk ki. Ilyenkor (a két csillagnak megfelelően) minden mezőszélesség, minden pontosság paraméterként szerepel. Mivel a következő paraméter a 8, ez lesz a mezőszélesség, és az ezt követő paraméterként megadott 2 lesz a pontosság. A `d` változó értéke tehát 8 karakternyi helyen, 2 tizedesjegy pontossággal kerül kijelzésre. A mínusz előjel és a tizedespont is beleszámít a mezőszélességebe (ez minden szélességi adatnál így van).

Programunk következő részében a `c` karakterváltozót jelenítjük meg többféle módon. A változó az 'x' értékét tárolja. Először az ismerős `%c` formátumjelző karakterláncjalccal valósítjuk meg a kiírást. Ezt követően egy utasítással kétszer írjuk ki a változót (mind a kétszer 3 karakternyi helyen, jobbra igazítva, szóközzel kitöltve).

A karakterváltozók bármilyen egész formátumban is megjeleníthetők. A kimenet következő sorában a `c` értékét hexadecimálisan írjuk ki. Az derül ki, hogy az adott számítógép az 'x'-et belsőleg az 58 hexadecimális számmal ábrázolja.

A kimenet utolsó részében az `s` karakterláncot jelenítjük meg néhány változatban. Először csak egy artikulálatlan `%s` formátumjelző karakterláncjalccal dolgozik a `printf`, majd megadunk egy pontossági adatot (5), melynek hatására csak az első öt karakter (azaz a latin ábécé első öt betűje) jelenik meg.

Ezután 30-as mezőszélességet adunk meg – ez több, mint a karakterlánc hossza, így jobbra igazítva láthatjuk a latin ábécét.

A szakasz utolsó két sorában 5 karakternyi pontosság és 20-as mezőszélesség van megadva. Az egyik esetben nincs megadva kizárási irány (azaz jobbra zárt lesz a kimenet), a másik esetben a mínusz előjel miatt balra zárt szöveget kapunk. Ha egy függőleges vonal (|) karaktert is kinyomtatunk a karakterlánc után, akkor látszik, hogy az öt betűt 15 szóköz követi.

A mutatók értékét a %p formátumjelző karakterláncjal lehets megjeleníteni. Programunkban az egészre irányuló ip mutatót és a karakterre irányuló cp mutatót íratjuk ki. Ha valaki lefuttatja a fenti programot, akkor biztos, hogy a fentiekől eltérő értékeket fog a képernyőn látni, hiszen minden számítógép máshol foglalja le a változókhhoz tartozó memória-területet.

A %p által előidézett kimenet pontos formája implementációfüggő. Példánkban hexadecimálisan jelennek meg a mutatók értékei. A kimenet tanúsága szerint az ip a bfffffc20 hexadecimális címet tartalmazza, a cp mutató pedig a bfffffbf0 címet.

A kimenet utolsó szakaszában a %n formátum-definíciót figyelhetjük meg. Ebben az esetben a megfelelő printf-paraméterek egy int mutatónak kell lennie, ha csak nem adunk meg hh, h, l, ll, j, z vagy t típusmódosítót. A printf eltárolja az adott pontig megjelennitett karakterek számát, mégpedig abban a változóban, amire a paraméterként megadott mutató hivatkozik. Így a %n első előfordulásakor 4 lesz a c1 értéke (mivel addig négy karaktert jelenítettünk meg: „Ilye”), majd a %n második felbukkanásakor 12 kerül be c2-be (ugyanis 12 karaktert jelenítettünk meg addig összesen). A %n-nek semmi hatása nincs a megjelenésre vonatkozóan.

A scanf függvény

A printf-hez hasonlóan a scanf-et is sokszor használtuk már, mégis érdemes elmélyülni a megismérésében. Számos olyan lehetőséget rejt, amiről eddig nem esett szó.

A 16.5 táblázat bemutatja, hogy milyen módosítások adhatóak meg a scanf számára a % jel és a típust meghatározó formátumkarakter között. Maguk a formátumkarakterek a 16.6 táblázatban láthatóak.

A felhasználói bemenet beolvasásakor a scanf eltekint a bevezető térköz karakterektől. Térköz karakter a szóköz, a vízszintes tabulátor ('\t'), a függőleges tabulátor ('\v'), a „kocsí vissza” ('\r'), az újsor ('\n') és a lapdobás ('\f') karakter. Kivételt képez a %c formátum-definíció, melynek használatakor a scanf mindenkiépp beolvassa a következő karaktert. Hasonlóképpen kivételt jelent az az eset, amikor szöglletes zárójelben kifejezetten soroljuk a beolvasandó (vagy éppen be nem olvasandó) karaktereket.

16.5 Táblázat • A scanf típusmódosítói

Típus	Jelentése
*	a vonatkozó mezőt figyelmen kívül kell hagyni, nem kell beolvasni
méret	a mezőszélesség legalább ennyi legyen
hh	a beolvasott értéket tároljuk előjeles vagy előjel nélküli char típusúként
h	a beolvasott értéket tároljuk short int-ként.
l	a beolvasott értéket tároljuk long int-ként, double-ként vagy wchar_t-ként.
j, z vagy t	a beolvasott értéket tároljuk size_t-ként (%j), ptrdiff_t-ként (%z), intmax_t-ként vagy uintmax_t-ként (%t).
ll	a beolvasott értéket tároljuk long long int-ként.
L	a beolvasott értéket tároljuk long double-ként.
típus	az átalakítási formátumtípust jelző karakter

16.6 Táblázat • A scanf formátumjelző karakterei

Karakter	Szerepe
d	A beolvasandó érték tízes számrendszerben van megadva. Az ide tartozó mutató csak egészre irányulhat – ez lehet short, long vagy long long int.
i	Mint a %d, ám a szám lehet (0 kezdettel) oktális vagy (0x vagy 0X kezdettel) hexadecimális is.
u	A beolvasandó érték egész szám, az ide tartozó mutató pedig egy előjel nélküli egészre mutat.
o	A beolvasandó egész érték oktálisan (nyolcas számrendszerben) értelmezendő. Kezdődhet 0-val, de ez nem kötelező. A megfelelő mutató int-re irányul, kivéve, ha (h, l vagy ll megadásával) short, long vagy long long int a választott típus.
x	A beolvasandó egész érték hexadecimálisan értelmezendő. Kezdődhet 0x-szel (vagy 0X-szel), de ez nem kötelező. A megfelelő mutató előjel nélküli int-re irányul, de ezt (h, l vagy ll választással) módosíthatjuk short-ra, long-ra vagy long long int-re is.
a, e, f vagy g	A beolvasandó érték lebegőpontos szám, melynek lehet előjele. Használható exponenciális alak is (például 3.45 e-3). A megfelelő mutató float-ra irányul, de ezt (1 vagy L választásával) módosíthatjuk double-ra vagy long double-ra.

Karakter	Szerepe
c	A beolvasandó érték egy karakter. A következő leütés tartalma mindenkiépp bekerül a megfelelő (char-ra irányuló) mutatóba, még akkor is, ha az térköz karakter (azaz szóköz, tabulátor, újsor vagy lapdobás). Megadható egy szám is a c előtt: ez a beolvasandó karakterek számaként értelmeződik.
s	A beolvasandó érték egy karakterlánc. Az első nem-térköz karakterrel kezdődik és az utolsó nem-térköz karakterrel zárol a beolvasás. A megfelelő paraméternek egy karakterláncra kell mutatnia, melyben van elég hely a szükséges számú karakter (és az automatikusan odakerülő lezáró null karakter) tárolására. Ha egy szám is áll az s előtt, akkor csak az ennek megfelelő számú karaktert olvassa be a scanf (hacsak nem találkozik hamarabb egy térköz karakterrel, ami lezárja a beolvasást).
[...]	A zárójelben álló karakterek közül kerülhet ki a (%s-hez hasonlóan értelmezett) beolvasandó karakterlánc. A beolvasást lezárja bár-mely olyan karakter, amely nem szerepel a felsorolásban. Megadhatjuk azokat a karaktereket is, amelyeket <i>nem</i> szeretnénk beolvasztatni (azaz a beolvasást lezáró karakterek listáját): ehhez a zárójelen belüli első helyre ^ jelet kell írnunk. Ilyenkor az első „tiltott” karakterig fog terjedni a beolvasott karakterlánc.
n	Nem olvas be semmit a scanf, hanem az adott függvényhívásban addig beolvasott karakterek számát tárolja el abban az egész számban, amit a vonatkozó paraméter mutat.
p	A beolvasandó értéket mutatónak tekinti a scanf, mégpedig ugyanolyan módon, ahogy a %p-t a printf értelmezi. A megfelelő mutató egy void-ra hivatkozó mutatóra irányuló mutató.
%	A következő nem-térköz karakternek százalék jelnek kell lennie a bemeneten.

A scanf addig olvas be egy adott értéket, amíg el nem éri a kívánt mezőszélességet (ha van ilyen), vagy amíg érvénytelen karakterrel nem találkozik. Egész számok esetén az érvényes karakterek: az előjel, a számrendszerből függő számjegyek (tízes számrendszerben 0-9, nyolcas számrendszerben 0-7, hexadecimálisban 0-9 és a-f/A-F). Lebegőpontos számok esetén megengedett az előjel, majd számjegyek tetszőleges sorozata, esetlegesen tizedespont és újabb számjegysorozat. Mindezeket követheti még az exponens (kitevő) megadása egy e/E betűvel és egy (akár előjeles) számmal. A %a formátumjelzés esetén hexadecimális lebegőpontos szám adható meg. Ennek formátuma egy bevezető 0x, majd hexadecimális számjegyek sorozata, esetlegesen „tizenhatodospont” és újabb számjegysorozat. Mindezeket követheti még a kitevő megadása (p/P betűvel kezdve).

A %s-sel kért beolvasásnál csak a térköz karakterek számítanak érvénytelennek, a %c esetén pedig nincs érvénytelen karakter. Ha szögletes zárójelben soroljuk fel a karaktereket, egyértelműek az érvényes karakterek (vagy [...] esetén az érvénytelen karakterek).

A 9. fejezet („Adatszerkezetek”) néhány programjában a felhasználónak be kellett gépelnie egy időpontot a terminálablakban. Ezekben a formátumjelző karakterlánc úgy volt megadva, hogy a formátumjelzésen (például %i) kívüli konkrét karaktereket (például ':') az adott helyen be kellett írnia a felhasználónak. Például a

```
scanf ("%i:%i:%i", &hour, &minutes, &seconds);
```

meghíváskor a felhasználónak három egész számot kellett beírnia (órát, percet és másodpercet), kettősponttal elválasztva.

Ha százalék jelet szeretnénk bekérni, azt dupla % jelleggel adhatjuk meg a formátumjelző karakterláncban:

```
scanf ("%i%%", &percentage);
```

A formátumjelzésben szereplő térköz karakterek helyén tetszőleges számú ilyen karakter beírható. A

```
scanf ("%i%c", &i, &c);
```

utasítás futtatásakor adjuk meg a következő bemenetet:

```
29      w
```

Ekkor 29 kerül az i változóba és szóköz (!) a c-be, ugyanis a szám utáni első karakter a szóköz. Ha azonban ugyanezt a bemenetet a

```
scanf ("%i %c", &i, &c);
```

futtatásakor adjuk meg, akkor 29 töltődik az i-be és 'w' karakter a c-be. A formátumjelző karakterláncban szereplő szóköz hatására ugyanis a scanf tetszőleges számú térközt elvisel a két adat között.

A 16.5 Táblázatban szerepelt a csillag karakter, melynek hatására a scanf kihagy egy mezőt. Futtassuk le a

```
scanf ("%i %5c %*f %s", &i1, text, string);
```

utasítást, és írjuk be a következőket:

```
144abcde      736.55      (bor és sajt)
```

Ennek hatására 144 kerül az i1-be, az abcde karakterlánc pedig a text karaktertömbbe. A 736.55 lebegőpontos értéket felismeri ugyan a scanf, de figyelmen kívül hagyja. A string változtóba bekerül a '(bor' karakterlánc, nullal lezárva. A scanf következő meghívásakor az adatok beolvasása *onnan folytatódik, ahol az előző híváskor véget ért.* Ha az előző kódrészlet után a

```
scanf ("%s %s %i", string2, string3, &i2);
```

következik, akkor a string2 változóba bekerül az 'és', a string3-ba pedig a 'sajt)', és a függvény tovább várakozik egy egész szám beírására.

Ne felejtsük, hogy a scanf mutatókat vár paraméterként, melyeket felhasználhat arra, hogy a bemenetet eltárolja a megfelelő változókhoz tartozó memóriaterületen. A 11. fejezetben volt szó arról, hogy miért van erre szükség. Más módon a scanf nem lenne képes maradandó változást előidézni, nem tudná eltárolni a beadott értékeket. Azt is tanultuk, hogy egy tömbre irányuló mutató esetében elég a tömb nevét megadni. Ha a text változót (megfelelő méretű) karaktertömbként definiáltuk, akkor a

```
scanf ("%80c", text);
```

meghívásakor a felhasználói bemenetből 80 karakter kerül a text tömbbe. A

```
scanf ("%[^/]", text);
```

hatására a perjelen kívül minden karaktert elfogad a scanf. Ha ennek meghívásakor a bemenet:

```
(bor és sajt)/
```

akkor a teljes '(bor és sajt)' karakterlánc bekerül a text-be. Ugyanis a bemenet első „érvénytelen” karaktere a '/' – az előtte álló összes adat hozzárendelődik a megadott változóhoz. Maga a perjel pedig a következő scanf híváskor kerül beolvasásra.

Megoldható, hogy a felhasználói bemenet egy sora hozzárendelődjön egy változóhoz. Az alábbi utasítás hatására a buf változóba kerül az első újsor karakterig minden:

```
scanf ("%[^\\n]\\n", buf);
```

Maga az újsor karakter a szögletes zárójelen kívül is szerepel, így a következő `scanf` híváskor már nem kerül beolvasásra. (A `scanf` ugyanis mindenkoronnan folytatja a beolvasást, ahol az előző híváskor abba hagyta.)

Ha olyan karaktert adunk meg felhasználói bemenetként, amely „tiltottnak” minősül (például ha egy egész szám beírását váró `scanf`-nek '`x`'-et adunk meg), akkor a `scanf` nem vár további bemenetre, hanem kilép. Mivel a függvény visszatérési értéke a sikeresen beolvasott (és változóhoz hozzárendelt) adatok száma, ezt könnyen ellenőrizni lehet; hiba esetén belátásunknak megfelelően járhatunk el a programban. A következő hívás például azt ellenőri, hogy sikeresen megtörtént-e a három adat beolvasása és hozzárendelése:

```
if ( scanf ("%i %f %i", &i, &f, &l) != 3 )
    printf ("Hiba a bemenetben\n");
```

Ha hiba történt, akkor ezt egy megfelelő üzenettel nyugtázzuk.

Tartsuk szem előtt, hogy a `scanf` visszatérési értéke a beolvasott és változóba töltött adatok számát adja meg, tehát a helyesen lefutó

```
scanf ("%i %*d %i", &i1, &i3)
```

utasítás visszatérési értéke nem 3, hanem 2. Itt ugyanis csak két egész számot olvasunk be, a középső adatot kihagyjuk. A `%n` használata (az addig beolvasott karakterek száma) sem számít bele a visszatérési értékben szereplő összegbe.

Érdemes sokat kísérletezni a `scanf` által nyújtott beolvasási formátumok lehetőségeivel. A `printf`-hez hasonlóan úgy mélyíthetjük el leginkább tudásunkat, hogy a különféle formátum-definíciókat programokban próbáljuk ki.

Kimeneti és bemeneti műveletek fájlokkel

A `scanf` eddigi meghívásaikor a felhasználói bemenet mindenkoronnan a parancssorból érkezett, és a `printf` hívások kimenetei is a képernyőn jelentek meg. Ebben a szakaszban arról lesz szó, hogy hogyan írhatunk és olvashatunk fájlokba/fájlokba.

A kimenet/bemenet fájlba irányítása

Az írási és olvasási fájlműveletek számos operációs rendszerben egyszerűen megvalósíthatóak. Ahhoz, hogy a felhasználói bemenet helyett fájlból történjen a beolvasás, Unixban és Windowsban nem kell módosítani a programot, csak megfelelő módon kell elindítani.

Ha a program kimenetét (a képernyő helyett) a `data` állományba szeretnénk átirányítani, ahhoz az alábbi parancsot kell kiadni a terminálablakban:

```
prog > data
```

Az átirányító parancs arra utasítja az operációs rendszert, hogy a program végrehajtásakor a kimenetet ne a képernyőn jelenítse meg, hanem írja a `data` fájlba. Így a `printf` által történő kiírások minden esetben a fájlba kerülnek.

Ennek kipróbálásához vegyük elő első programunkat, a 3.1 Listát. Fordítsuk le a szokásos módon, és futtassuk (legyen `prog1` a neve).

```
prog1
```

Ha minden rendben, akkor megjelenik a képernyőn a felirat:

Programming is fun.

Indítsuk most így a programot:

```
prog1 > data
```

Ekkor nem látunk semmit a képernyőn, mert a kimenet a `data` állományba íródott. A `data` fájl tartalma most ebből a sorból áll:

Programming is fun.

Vagyis a program kimenetét sikerült átirányítanunk a képernyőről a `data` fájlba. Ennél boholyultabb (több sort megjelenítő) program futtatásával is kipróbálható ez a megoldás.

Hasonlóképpen irányítható át egy program bemenete is a parancssori bevitel helyett egy fájlból történő beolvasásra. A parancssorból bemenő adatot váró függvények (például a `scanf` és a `getchar`) meghívásakor megoldható, hogy a bemenetet egy fájlból olvassa ki a program. Az 5.8 Lista egy szám számjegyeit írta ki fordított sorrendben. A program a `scanf` segítségével olvasta be a megfordítandó számot a parancssorból. Szeretnénk, ha a kézi beírás helyett a program a `number` állományból olvasná ki a keresett számot. Ezt a bemenet átirányításával tudjuk megtenni. Legyen programunk neve `reverse` – ekkor az alábbi utasítással irányíthatjuk át a bemenetét:

```
reverse < number
```

Ha beírjuk a `number` állományba a 2001 értékét, akkor a fenti parancs futtatása után ezt láthatjuk:

```
Írja be a megfordítandó számot!
1002
```

A program (elindulása után) nem vár felhasználói bemenetre, hiszen az át van irányítva oly módon, hogy a `number` fájlból történjen az adatolvasás. A kimenet nincs átirányítva, azt továbbra is láthatjuk a képernyőn. Végső soron a `scanf` a parancssor helyett a `number` fájlból olvas. Ugyanazt az információt kell a fájlba beírni, mint amit a parancssorban is megadnánk. A `scanf` függvény igazából nem is tudja eldöntenи (erre nincs is szükség), hogy honnan érkezett a bemenet: terminálablakból-e vagy egy állományból – csak az érdeklő, hogy megfelelően van-e minden megformázva.

Természetesen az is megoldható, hogy mind a kimenet, mind a bemenet át legyen irányítva. A

```
reverse < number > data
```

utasítás hatására a `reverse` program a `number` fájlból fogja olvasni a bemenetet, és kimenetet a `data` fájlba fogja írni. Próbáljuk ki ezt az 5.8 Lista programjával!

A kimenet vagy/és a bemenet átirányítása gyakran hasznos. A 10.8 Lista programja megszámolja a bemeneten megjelenő szavakat. Tegyük fel, hogy írunk egy újságcikket, melyet az `ujsgcikk` fájlban tárolunk. Ennek szavait is megszámolhatjuk az említett programmal, a bemenet átirányításával:²

```
wordcount < ujsgcikk
```

Fontos, hogy az `ujsgcikk` szöveg végét két enterrel zárjuk le (és máshol ne tegyük így), mert a program arra számít, hogy az adatsor végét egy magányos újsor karakter jelzi.

Az itt vázolt kimeneti/bemeneti átirányítás nem része az ANSI C nyelvnek – elképzelhető tehát olyan operációs rendszer, amely nem támogatja ezt a lehetőséget. Szerencsére a legtöbb rendszerben mindez megoldható.

² A Unix rendszerekben megtalálható a `wc` („*word count*”) parancs, amellyel szintén meg tudjuk számolni a szavakat. A 10.8 Listához hasonlóan ez is arra készült, hogy egyszerű szövegfájlokkal dolgozzon, nem pedig bináris dokumentumformátummal, mint amilyen például a MS Word `.doc` formátuma.

Állomány vége

Egy pillanatra térjünk még vissza az előző szakasz végén említett kérdésre az adatsor végevel kapcsolatban. Állományok esetén ezt a feltételt (!) „fájlvégének” (*End Of File, EOF*) hívjuk – ez akkor teljesül, amikor az állomány utolsó adatát is beolvastuk. Ha az állomány végének elérése után kísérelünk meg adatot beolvasni, az a program hibás véget éréséhez vagy végtelen ciklushoz vezethet, ha ezt a feltételt nem figyeli a program. A szabványos kimeneti/bemeneti könyvtár legtöbb függvénye szerencsére jelzi az állomány végének elérését egy speciális érték visszaadásával. Ez nem más, mint az EOF konstans értéke, ami a `<stdio.h>` szabványos fejlécállományban van definiálva.

A `getchar` függvény használatát és ezzel összekapcsoltan egy EOF-vizsgálatot mutat be a 16.2 Lista. A program karaktereket olvas be a bemenetről, és kiírja őket a kimenetre. A művelet addig tart, amíg el nem érjük az állomány végét. Figyeljük meg a `while` ciklusban található kifejezést! A hozzárendelés és a vizsgálat egy lépésben történik meg.

16.2 Lista • Karakterek másolása a bemenetről a kimenetre

//Karakterek ismétlése a fájl végéig.

```
#include <stdio.h>

int main (void)
{
    int   c;

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

    return 0;
}
```

Nevezzük ezt a programot `copyprog`-nak. Lefordítása után átirányíthatjuk a bemenetét a következő parancssal:

```
copyprog < infile
```

A program kiírja az `infile` tartalmát a képernyőre. Próbáljuk is ki!

Ez a kis példaprogram a Unix rendszereken használatos `cat` parancshoz hasonló alapvető feladatot lát el. Megjeleníthetjük vele egy szöveges állomány tartalmát.

A 16.2 Lista programjának `while` ciklusában a `getchar` függvény által visszaadott értéket felveszi a `c` változó. Ennek értékét hasonlíta össze a program az EOF-fal. Ha e kettő meggyezik, akkor a ciklus elért az állomány végéhez, az utolsó karakterhez. Fontos, hogy

a getchar által visszaadott érték int típusú legyen, nem pedig char. Az EOF értékének ugyanis egyedinek kell lennie, amely nem állhat elő más (getchar által visszaadott) normál karakter értékeként. Emiatt rendeljük hozzá programunkban a getchar visszatérési értékét char helyett int típusú változóhoz. Ez a megoldás működik, mert a C nyelv megengedi, hogy a karakterek értékét int változóban tároljuk – bár általanosságban ez nem a legszerencsesebb programozási gyakorlat.

Ha a getchar visszatérési értékét char változóban tároljuk, akkor megjósolhatatlan a programfutás kimenetele. Olyan rendszereken, ahol előjel-kiterjesztés történik, rendben futhat a program. Ha nincs előjel-kiterjesztés, akkor viszont végtelen ciklusba juthatunk.

Összességében azt kell mondanunk, hogy érdemesebb a getchar visszatérési értékét int változóban tartani, hogy biztosan meg tudjuk állapítani az állomány végének az elérését.

A ciklusfeltételben látható hozzárendeléses feltételvizsgálat mutatja a C nyelv hihetetlen rugalmasságát a kifejezések használata terén. A zárójel nem hagyható el az értékadásnál, mert az értékadási operátor alacsonyabb precedenciájú a „nem egyenlő” operátornál.

Fájlkezelő függvények

Fejlesztendő programjaink kimeneti/bemeneti műveleteinek nagy része megoldható a getchar, putchar, scanf és a printf függvények (és az átirányítás) segítségével. Adódhatnak azonban olyan helyzetek is, amikor összetettebb fájlműveletekre van szükség. Elképzelhető például, hogy több programból kell adatokat olvasni, vagy több állományba kell írni a kimenetet. Az ilyen feladatok megoldásához speciális fájlkezelő függvények állnak rendelkezésünkre. Ezek közül a legfontosabbakat megvizsgáljuk a következő bekezdésekben.

Az fopen függvény

Mielőtt bármilyen kimeneti/bemeneti műveletet végrehajthatnánk egy állományon, meg kell azt *nyitni*. Ehhez megadandó a fájl neve. A rendszer megvizsgálja, hogy a megadott fájl létezik-e, és ha nem, akkor létrehozza (bizonyos esetekben). Az állomány megnyitásakor azt is meg kell adnunk, hogy milyen kimeneti/bemeneti műveletet szeretnénk vele végrehajtani. Ha csak olvasni szeretnénk belőle, akkor *olvasási módban* (*read*) érdemes megnyitni. Ha írni is akarunk egy állományba, akkor *írási módra* (*write*) lesz szükségünk. Ha egy meglévő fájl végéhez szeretnénk újabb adatokat illeszteni, akkor *huzzáfűzési módban* (*append*) kell megnyitni. Ha a megnyitandó állomány nem létezik, akkor az utóbbi két esetben (írási és huzzáfűzési módban) létrehozza a fájlt a rendszer, ha azonban egy olvasási módban megnyitandó állomány hiányzik, akkor hibaüzenetet kapunk.

Mivel egyszerre több fájl is nyitva lehet a program futásakor, valahogyan azonosítanunk kell őket a fájlműveletek elvégzéséhez. Ezt *fájlmutatók* révén tudjuk megenni.

A szabványos programkönyvtár fopen függvénye a megadott állomány megnyitásakor visszaadja a fájlra irányuló mutatót, amellyel a továbbiakban kezelní tudjuk a megnyitott állományt. Az fopen két paramétert vár: a megnyitandó állomány nevét és a megnyitás módjára utaló karakterláncot. A visszatérési értékként kapott fájlmutatót a többi könyvtári függvény fel tudja használni a keresett állomány azonosításához.

Ha a fájlt valamilyen okból nem sikerül megnyitni, akkor a függvény (a `<stdio.h>` fejléc-állományban definiált) NULL értékkel tér vissza.³

Ugyanebben a fejlécállományban van definiálva a FILE típus is. Az fopen visszatérési értékének érkező mutató tárolásához „FILE-ra irányuló mutató” típusú változót kell létrehozni.

A fenti megjegyzések szem előtt tartásával már megérthető a következő kód részlet:

```
#include <stdio.h>

FILE *inputFile;

inputFile = fopen ("data", "r");
```

A data fájlt olvasási módban nyitjuk meg. (Az írási mód jele 'w', a hozzáfűzési módé 'a'.) Az fopen hívás visszaadja a megnyitott állomány azonosítóját, amelyet hozzárendelünk az inputFile-hoz (ami egy FILE-mutató). Ennek értékét összevetjük a NULL-lal, hogy látassuk, sikeres volt-e a művelet:

```
if ( inputFile == NULL )
    printf ("*** Sikertelen megnyitás.\n");
else
    // kezdődhet az adatok olvasása
```

Az fopen a fent említetteken túl többféle megnyitási módot is támogat. Háromféle „frissítési” mód létezik: "r+", "w+", és "a+". Mindhárom mód lehetővé teszi az írást és az olvasást is a megnyitott állományból. Az „olvasva frissítés” ("r+") létező fájlt tud megnyitni írásra és olvasásra. Az „írva frissítés” ("w+") hasonlít az írási módhoz, azaz a meglévő fájl tartalma törlődik, a nemlétező fájl pedig létrejön; de ebben az esetben az írás is és az olvasás is megengedett. A „hozzáfűző frissítés” ("a+") megnyitja a létező fájlt, vagy létrehozza a nemlétezőt. Az olvasási művelet a fájl bármely pontján megengedett, azonban írni csak az állomány legvégre lehet.

³ A NULL „hivatalosan” a `<stddef.h>` fejlécállományban van megadva, de a legtöbb esetben a `<stdio.h>`-ban is megtalálható.

Egyes operációs rendszerek (például a Windows) megkülönböztetik a szöveges állományokat a binárisaktól. Az ilyen környezetben futó programokban a megnyitási mód karakterláncának a végére írnunk kell egy 'b' karaktert abban az esetben, ha bináris fájl szeretnénk olvasni vagy írni. Ha ezt nem tesszük meg, a programunk akkor is fog futni, de furcsa eredményeket fog produkálni. Ennek oka a sajátos sorzáró „kocsi vissza + soremelés” karakterpár, amelyek újsorrá alakulnak, amikor a rendszer kiolvassa (vagy visszairja) az adatokat egy szöveges fájlból (vagy fájlból).

Ha egy bemeneti bináris állomány CTRL+Z karaktert tartalmaz, akkor ez kiváltja az EOF feltételt, hacsak nem bináris módban nyitottuk meg a fájlt. Érdemes tehát az alábbi módon megnyitni a bináris fájlokat (esetünkben a `data` fájlt) olvasásra:

```
inputFile = fopen ("data", "rb");
```

A `getc` és a `putc` függvény

A `getc` függvény lehetővé teszi egy karakter beolvasását egy állományból. Ez a függvény ugyanúgy működik, mint a korábban vázolt `getchar` függvény, azzal a különbséggel, hogy paraméterként megadandó az a fájlmutató, amely alapján dolgoznia kell. A fenti `fopen` parancs után használhatjuk tehát a következő utasítást:

```
c = getc (inputFile);
```

A függvényhívás és az értékadás hatására a `data` fájlból egy karakter bekerül a `c` változóba. Ugyanennek az utasításnak az ismételt meghívásával a többi karaktert is beolvastathatjuk.

A `getc` is EOF-ot ad vissza a fájl végének elérésekor, így (a `getchar`-hoz hasonlóan) itt is int típussal érdemes elfogni a visszatérési értéket.

Az elhangzottak alapján bizonyára kitalálta már az olvasó, hogy a `putc` a `putchar`-hoz hasonlóan működik, csak a két paramétert vár (egy helyett). Az első paraméter a fájlba írandó karaktert tartalmazza, míg a második egy FILE-mutató. Így a

```
putc ('\n', outputFile);
```

utasítás egy újsor karaktert ír ki az `outputFile` által hivatkozott állományba. Ennek működéséhez azonban előzőleg meg kell nyitnunk az `outputFile`-et írási vagy hozzáfűzési módban (vagy valamelyik frissítési módban).

Az `fclose` függvény

A fájlokkal kapcsolatos másik fontos művelet a fájlok *bezárása*. Az `fclose` függvény bizonyos értelemben az `fopen` ellentettje: tájékoztatja a rendszert arról, hogy a továbbiakban nincs szükségünk a megnevezett állományra. A fájl lezárásakor a rendszer elvégez bi-

zonyos munkálatokat (például a memória gyorstárában lévő adatokat beírja az állományba), valamint felbontja a kapcsolatot a fájl és a hozzá tartozó azonosító között. A fájl bezárása után már nem lehet belőle olvasni vagy írni (hacsak újra meg nem nyitjuk).

Érdemes rászokni arra, hogy a fájlműveletek után lezárjuk a fájlt. Ha egy program szabályosan fejezi be futását, akkor automatikusan lezár minden megnyitott állományt – jobb azonban ezt azonnal megtenni, mihelyt nincs szükségünk rájuk. Ennek az előnye akkor jelentkezik leginkább, ha sok fájllal dolgozik a program. Van egy gyakorlati kúszóbérték, amelynél több állományt nem célszerű egyszerre nyitva tartani. Ennek értéke gépenként változik. A probléma csak akkor jelentkezik, ha több állománnyal egyszerre dolgozik a program.

Az `fclose` függvény paramétere egy `FILE` típusú mutató, amely a bezárandó állományra irányul. Így az

```
fclose (inputFile);
```

utasítás lezárja az `inputFile` által mutatott állományt.

Az `fopen`, `putc`, `getc` és `fclose` függvényekkel már tudunk írni egy olyan programot, amely egy állomány tartalmát átmásolja egy másikba. A 16.3 Lista lehetővé teszi a felhasználó számára, hogy megadja a másolandó és a célfájl nevét. A program a 16.2 Listán alapul. Érdemes a kettőt összehasonlítani.

Tegyük fel, hogy a `masolj` fájl tartalma a következő:

```
Ez egy próbaszöveg a fájlmásoló program számára,  
amit az fopen, fclose, getc, és putc függvények  
felhasználásával készítettünk.
```

16.3 Lista • Fájlmásolás

```
// Egy fájlt egy másikba átmásoló program

#include <stdio.h>

int main (void)
{
    char inName[64], outName[64];
    FILE *in, *out;
    int    c;

    // kérjük be a fájlok neveit

    printf ("Adja meg a másolandó állományt: ");
```

```

scanf ("%63s", inName);
printf ("Adja meg a célfájl nevét: ");
scanf ("%63s", outName);

// a bemeneti és kimeneti fájl megnyitása

if ( (in = fopen (inName, "r")) == NULL ) {
    printf ("%s nem nyitható meg olvasásra.\n", inName);
    return 1;
}
if ( (out = fopen (outName, "w")) == NULL ) {
    printf ("%s nem nyitható meg írásra.\n", outName);
    return 2;
}

// a bemenet átmásolása a kimenetbe

while ( (c = getc (in)) != EOF )
    putc (c, out);

// A megnyitott állományok lezárása

fclose (in);
fclose (out);
printf ("Az állomány másolása sikeresen véget ért.\n");

return 0;
}

```

16.3 Lista • Kimenet

Adja meg a másolandó állományt: **masolj**

Adja meg a célfájl nevét: **ide**

Az állomány másolása sikeresen véget ért.

Ellenőrizzük az ide fájl tartalmát: ennek meg kell egyeznie a masolj állomány három sorával.

A scanf függvény hívásakor a megadható karakterlánc hosszát 63-ra korlátozzuk, hogy az állománynevek biztosan elférjenek az inName és outName változóban. A program az egyiket olvasásra, a másikat írásra nyitja meg. Ha a kimeneti állomány már létezik (és írási módban nyitjuk meg), akkor a legtöbb rendszeren felülíródik eredeti tartalma.

Ha a két fopen hívás bármelyike meghiúsul, akkor a program egy megfelelő üzenet kiírása után kilép, mégpedig nullától eltérő (1 vagy 2) visszatérési értékkel. Ha azonban mindkét fájlmegnyitás sikerül, akkor a bemeneti állomány (a getc és a putc meghívásával) ka-

rakterenként átmásolódik a kimeneti állományba, amíg csak el nem éri a program a bemeneti fájl végét. Ekkor lezárjuk a két állományt, és nulla visszatérési értékkel nyugtázzuk a sikeres futást.

Az feof függvény

Egy fájl végének elérését a feof függvénnyel is megvizsgálhatjuk. Ennek egyetlen paramétere egy FILE-mutató. A függvény egy nullától eltérő egész számot ad vissza, ha a fájl vége után kísérelünk meg beolvasást, egyébként pedig nullát. Tekintsük a következő kódrészletet:

```
if ( feof (inFile) ) {  
    printf ("Vége az adatsornak.\n");  
    return 1;  
}
```

Ha az inFile által hivatkozott állományra vonatkozóan teljesül a „fájl vége” feltétel, akkor megjelenik a „Vége az adatsornak” felirat. Ne felejtjük el, hogy a feof arról tájékoztat, hogy a fájl vége után kísérelünk meg adatot beolvasni – ez nem ugyanaz, mint amikor a fájl legutolsó adatát olvassuk ki. A legutolsó érvényes olvasás után még kell egyet olvasnunk ahhoz, hogy a feof nullától eltérő visszatérési értéket adjon.

Az fprintf és az fscanf függvény

Az fprintf és az fscanf függvény értelemszerűen úgy működik, mint a printf és a scanf, csak még egy paramétert várnak, amely megadja, hogy mely fájlba (fájlból) törtenjen a kiírás (beolvasás). A kívánt paraméter típusa FILE-mutató kell, hogy legyen. Ha a "Programming in C is fun.\n" feliratot szeretnénk kiírni az outFile-ra, akkor ezt a következő utasítással tehetjük meg:

```
fprintf (outFile, "Programming in C is fun.\n");
```

Ha a következő lebegőpontos számot be szeretnénk olvasni az inFile-ból a fv változóba, akkor ehhez az

```
fscanf (inFile, "%f", &fv);
```

utasítást használhatjuk. A scanf-hez hasonlóan az fscanf is a sikeresen beolvasott és értékű adott paraméterek számát adja vissza, vagy az EOF értékét, ha a fájl véget még azelőtt eléri, hogy a megadott bemenet-átalakítások bármelyikét is meg tudta valósítani.

Az fgets és az fputs függvény

Teljes sorok beolvasására és kiírására használható az fgets és az fputs függvény. Az fgets függvény a következőképp hívható meg:

```
fgets (buffer, n, filePtr);
```

A *buffer* egy karaktertömb-mutató, ahol a beolvasandó sort szeretnénk tárolni. A *filePtr* adja meg azt az állományt, amelyben a kiolvasandó sor található.

Az fgets függvény kiolvassa a megadott fájl soron következő karaktereit mindaddig, amíg újsor karakterrel nem találkozik, vagy a beolvasott karakterek száma el nem éri az *n-1*-et. A karakterlánc végére null karaktert helyez el a függvény. Ha a beolvasás sikeres volt, akkor az első paraméter (*buffer*) értéke lesz a visszatérési érték, ha pedig hiba következik be a beolvasáskor (vagy a fájl vége után kíséreljük meg a beolvasást), akkor NULL értéket ad vissza a függvény.

A fgets jól használható a sscanf-fel együtt (lásd a B függeléket), ha a sor alapú beolvasást a scanf kínálta lehetőségeknél finomabban szeretnénk megoldani. Az fputs függvény egy sort ír ki a megadott állományba; a következőképp hívható meg:

```
fputs (buffer, filePtr);
```

A *buffer*-ben tárolt karakterek (egészen a null karakterig) kiíródnak a *filePtr* által mutatott fájlba. A karakterláncot lezáró null karakter már nem kerül kiírásra.

Vannak még hasonló függvények (például a gets és a puts), melyek jól használhatóak egy-egy sor terminálablakból (vagy onnan) történő beolvasására (kiírására); ezeket a B függeléken találjuk.

Az stdin, az stdout és az stderr

Amikor egy C programot elindítunk, három állományt automatikusan megnyit a rendszer a program számára. Ezeket a *konstans* stdin, stdout és stderr FILE-mutatók azonosítják, melyek a <stdio.h>-ban vannak definiálva. Az stdin FILE-mutató a program szabványos bemenetére mutat, ami alapértelmezetten a terminálablak. Mindazok a bemeneti függvények, amelyek nem kapnak paraméterként FILE-mutatót, az stdin-ről veszik a bemenetet. A scanf függvény például innen dolgozik – meghívása egyenértékű azzal, mint-ha az fscanf-et az stdin paraméterrel hívánk meg. Így az

```
fscanf (stdin, "%i", &i);
```

utasítás a következő egész számot a szabványos bemenetről olvassa be, amely alapértelmezetten a terminálablak. Ha a szabványos bemenetet átirányítottuk valamelyik másik fájlba, akkor az előbbi utasítás az átirányított fájlból fogja beolvasni a következő egész számot.

Ezek után már sejthető, hogy az `stdout` a szabványos kimenetre utal, amely alapértelmezzen szintén a terminálablak. Így a

```
printf ("Üdvözlet!\n");
```

utasítás egyenértékű egy megfelelően paraméterezett `fprintf`-fel:

```
fprintf (stdout, "Üdvözlet!\n");
```

Az `stderr` FILE-mutató adja meg a szabványos hibacsatornát. A rendszer által küldött hibaüzenetek ide érkeznek – alapértelmezetten ez is a képernyő. Sokszor hasznos lehet, ha a hibaüzenetek nem ugyanott gyűlnek, ahol a program normál kimenete. Különösen akkor, ha a program kimenetét fájlba irányítjuk. Ilyenkor a normál kimenetet a megadott állományba írja a program, de a hibaüzeneteket láthatjuk a képernyőn. Megtörténhet, hogy saját hibaüzeneteket is szeretnénk használni, amelyek a többi hibához hasonlóan az `stderr`-ben jelennek meg. Tekintsük a következő `fprintf` hívást:

```
if ( (inFile = fopen ("data", "r")) == NULL )
{
    fprintf (stderr, "Nem sikerült az adatok megnyitása.\n");
    ...
}
```

Ha a `data` nem nyitható meg, akkor a megfelelő üzenet a `stderr` fájlba íródik. Ha a szabványos kimenetet át is irányítja a felhasználó, ez a hibaüzenet akkor is a képernyőn jelenik meg.

Az `exit` függvény

Előfordulhat, hogy szükség lehet arra, hogy egy adott pillanatban azonnal befejezzük a program futását – például akkor, ha egy hibába ütközünk. Normál módon akkor ér véget egy program futása, amikor a vezérlés eléri a `main` eljárás utolsó utasítását, vagy lefut egy `main`-beli `return` utasítás. Ha valami miatt azonnal ki szeretnénk lépni a programból, akkor használhatjuk az `exit` függvényt:

```
exit (n);
```

Ez azonnal felfüggeszti a program futását. A még nyitva lévő fájlokat bezárja, és a program visszatérési értékeként `n`-et ad vissza (mintha csak egy `main`-beli "return `n`";"-ről lenne szó). Ezt „kilépési állapotnak” is hívjuk.

A `<stdlib.h>` szabványos fejlécállomány definiálja az `EXIT_FAILURE` és az `EXIT_SUCCESS` konstanst. Az előzőt a hibás programfutás jelzésére, az utóbbit pedig a sikeres kilépés jelzésére használhatjuk.

Ha a program a `main` utolsó utasításával ér véget, akkor a kilépési állapot nincs definiálva. Ha egy másik programnak szüksége van erre az értékre, akkor ezt nem engedhetjük meg. Ilyen esetben oldjuk meg, hogy csak jól definiált kilépési állapottal lépjen ki a program (például egy megfelelő `return` vagy `exit` használatával).

Az `exit` függvény használatának egy példájaként nézzük a következő kód részletét. A függvény `EXIT_FAILURE` állapottal lépteti ki a programot, ha a paraméterként megadott állományt nem lehet megnyitni olvasásra. Természetesen egy valódi programban egy alkalmas üzenettel is érdemes ezt jelezni; nem stílusos csak úgy drasztikusan felfüggeszteni a program futását.

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

FILE *openFile (const char *file)
{
    FILE *inFile;

    if ( (inFile = fopen (file, "r")) == NULL ) {
        fprintf (stderr, "Can't open %s for reading.\n", file);
        exit (EXIT_FAILURE);
    }

    return inFile;
}
```

Tartsuk szem előtt, hogy igazából nincs különbség egy `exit` vagy egy `main`-beli `return` révén történő kilépés között. Mindkettő befejezi a program futását, és visszatérési értékül adja a program kilépési állapotát. A fő különbség az, hogy az `exit` a main függvényen kívül is állhat. Az `exit` azonnal befejezi a program futását, míg a `return` csak visszaadja a vezérlést a hívó eljárásnak.

Állományok átnevezése és mozgatása

A `rename` könyvtári függvény használható egy fájl nevének megváltoztatására. Két paramétere van: a régi és az új fájlnév. Ha az átnevezés meghiúsul (például azért, mert a megadott fájl nem létezik, vagy a rendszer nem engedi meg a névváltoztatást), akkor a `rename` egy nullától eltérő értéket ad vissza. Tekintsük a következő kód részletet:

```
if ( rename ("tempfile", "database") ) {
    fprintf (stderr, "A tempfile nem nevezhető át\n");
    exit (EXIT_FAILURE);
}
```

A `tempfile`-t próbáljuk meg átnevezni database-re. Ha ez nem sikerül, akkor hibaüzenettel kilépünk.

A `remove` függvény törli a paraméterként megadott állományt. Ha nem sikerül a törlés, akkor nullától eltérő visszatérési értéket ad. Egy példa:

```
if ( remove ("tempfile") )  
{  
    fprintf (stderr, "A tempfile nem törölhető\n");  
    exit (EXIT_FAILURE);  
}
```

Megkíséreljük törölni a `tempfile`-t. Ha nem sikerült, egy megfelelő üzenettel jelezzük ezt a hibacsatornán és kilépünk.

Lehetőség van arra is, hogy a szabványos könyvtári függvények hibáit is jelentsük. Erre szolgál a `perror` függvény, melynek részleteit a B függelékben találjuk.

Ezzel véget ért a C nyelv kimeneti/bemeneti műveleteit részletező fejezetünk. Mint már jelezük, hely hiányában nem beszélünk minden lehetőségről. A szabványos C könyvtár függvények széles skáláját biztosítja, melyekkel karakterláncokat kezelhetünk, véletlenszerű kimeneti/bemeneti műveleteket hajthatunk végre, matematikai számításokat végezhetünk vagy dinamikus memóriakezelést használhatunk. A B függelék számos ilyen függvényt bemutat.

Gyakorlatok

1. Gépeljük be és futtassuk a fejezetben szereplő három példaprogramot. A kimeneteiket hasonlítsuk össze a könyvben szereplő programkimenetekkel.
2. Kísérletezzünk a könyv korábbi fejezeteiben található programokkal, irányítsuk át azok bemeneteit, kimeneteit.
3. Írunk programot, amely egy fájlt úgy másol át egy másikba, hogy a kisbetűket nagybetűkké változtatja.
4. Írunk programot, amely két állomány sorait (felváltva véve egyet-egyet a két állományból) összefésüli, és kiírja az eredményt `stdout`-ra. Ha az egyik fájlból elfognak a sorok, akkor a másik fájl többi sorát egyszerűen másolja ki a `stdout`-ra.
5. Írunk programot, amely egy állomány minden sorának *m*. karakterétől kezdődően az *n*. karakterig bezárólag kiírja a karaktereket a `stdout`-ra. Az *m* és *n* értékét a terminálablakból kérje be a program.
6. Írunk programot, amely egy állomány tartalmát húsz soronként kiírja a képernyőre. A kiírás után várakozzon a program egy billentyűleütésre. Ha q billentyűt üt le a felhasználó, akkor lépjen ki a program; bármely más billentyű esetén pedig jelezzen ki újabb húsz sort a program.

Speciális lehetőségek a C nyelvben

Ebben a fejezetben a C nyelv olyan érdekes lehetőségeit fogjuk számba venni, amelyek eddig nem kerültek szóba. Megemlíttünk néhány nehezebb területet is. Ilyen például a parancssori paraméterek vagy a dinamikus memória foglalás kérdése.

Néhány újabb utasítás

Ebben a szakaszban két olyan nyelvi eszköz kerül elő, amelyeket eddig nem említettünk: a `goto` és a `null` utasítás.

A `goto` utasítás

Aki hallott már a strukturált programozásról, bizonyára ismeri a `goto` utasításra vonatkozó elmarasztaló álláspontot. Elvileg minden programozási nyelvben létezik ilyen lehetőség.

A `goto` utasítás végrehajtásakor átkerül a vezérlés a program egy adott pontjára. Ez a fél-tétel nélküli vezérlésátadás azonnal megtörténik. Az ugrás célpontjának azonosításához címkét kell használnunk. A címke egy karakterlánc, melyet ugyanolyan szabályok szerint lehet létrehozni, mint a változóneveket. Kettősponttal lezárvva az elő az utasítás elő kell beírni, amelynek át akarjuk adni a vezérlést. A `goto` ugrással az aktuális függvényen belül kell maradni. A

```
goto out_of_data;
```

utasítás például annak az utasításnak adja át a vezérlést, amely az `out_of_data:` címke után áll. A címke a függvényen belül akárhol elhelyezhető, a `goto` utasítás előtt vagy után:

```
out_of_data: printf ("Az adatok váratlanul elfogytak.\n");
...
```

Lusta programozók gyakran használnak `goto` utasítást. Ez megtöri a program normál „soros” menetét, és kifejezetten megnehezíti a programozó logikájának követését. Egy határon túl ez már olyan mértéket ölhet, hogy gyakorlatilag lehetetlen „visszafejteni” a programkódot. Ez az, amiért a `goto` utasítás használatát a legtöbbben nem tartják jó programozási gyakorlatnak.

A null utasítás

A C nyelv megengedi, hogy egy önmagában álló pontosvesszőt tegyünk oda, ahol egyébként egy utasítás állhatna. Ezt hívjuk `null` utasításnak. Várakozásunknak megfelelően ez nem csinál semmit, látszólag tehát haszontalan. A C programozók azonban gyakran használják, különösen `while`, `for` és `do` ciklusokban. Az alábbi utasítással például beolvashatjuk a szabványos bemenetet az első újsor karakterig, és a karakterláncot eltárolhatjuk a `text` karaktertömbben:

```
while ( (*text++ = getchar ()) != '\n' )
;
```

Minden művelet a `while` ciklusfeltételeiben zajlik, így a ciklusmag üres marad. A `null` utasításra azért van szükség, mert a fordítóprogram a `while` ciklusfej utáni első utasítást tekinti ciklusmagnak. A `null` utasítás nélkül a program következő utasítása kerülne (szándékunk ellenére) a ciklusmag szerepébe.

A következő kódrészlet karaktereket másol a szabványos bemenetről a szabványos kimenetre, amíg csak a fájl véget nem ér:

```
for ( ; (c = getchar ()) != EOF; putchar (c) )
;
```

A következő `for` utasítás a szabványos bemeneten megjelenő karaktereket számolja össze:

```
for ( count = 0; getchar () != EOF; ++count )
;
```

A `null` utasítás utolsó példájaként nézzünk egy olyan `while` ciklust, amely a `from` mutató által megadott karakterláncot átmásolja a `to` által mutatott helyre:

```
while ( (*to++ = *from++) != '\0' )
;
```

Egyes programozók sportot ūznek abból, hogy a lehető legtöbb információt bezsúfolják a ciklusfeltételbe (vagy a `for` ciklus léptetési részébe). Az olvasó lehetőleg ne gyarapítsa az ilyen programozók táborát. Csak azok a kifejezések kerüljenek a ciklusfeltételbe, ame-

lyek tényleg a ciklusváltozót ellenőrzik. A többi utasítást írjuk a ciklusmagba. A hatékony-ság az egyetlen olyan ok, amire hivatkozva kivételt tehetünk. Ilyenkor szabad bonyolultabb kifejezéseket is megadni a ciklusfejben. Mindazonáltal hacsak nem nagyon fontos a végrehajtás sebessége, kerüljük el az efféle megoldásokat.

Gondoljunk csak bele, mennyivel egyszerűbb megérteni az iménti helyett az alábbi while ciklust:

```
while ( *from != '\0' )
    *to++ = *from++;
*to = '\0';
```

Az unió (union)

A C programozási nyelv egyik legszokatlanabb megoldása az unió (union). Ez a szerkezet többnyire olyan alkalmazásokban kerül elő, amelyekben többféle adattípust kell tárolnunk ugyanazon a tárterületen. Tegyük fel, hogy az x változót úgy szeretnénk létrehozni, hogy lehessen benne karaktert, lebegőpontos számot és egész számot is tárolni. Legyen ennek a típusnak a neve mixed.

```
union mixed
{
    char    c;
    float   f;
    int     i;
};
```

Az unió deklarációja ugyanúgy történik, mint egy adatszerkezeté, csak a struct kulcsszó helyett union szerepel. Az adatszerkezetek és az uniók közti fő különbség a memóriafoglalásban áll. Ha egy változót így definiálunk:

```
union mixed x;
```

akkor ez az x *nen* tartalmazhat egyszerre háromfélé (c, f és i) adattagot, hanem csak *egyet* ezek közül. Az x változó vagy char, vagy float, vagy int típusú lehet, de egyszerre csak egy ezek közül. Az alábbi módon tárolhatunk el egy karaktert x-ben:

```
x.c = 'K';
```

Hasonlóképpen tudjuk kinyerni az x-ben tárolt értéket. A képernyőre való kiíratáskor például így járhatunk el:

```
printf ("A karakter: %c\n", x.c);
```

Egy lebegőpontos számot az `x.f` jelölésmóddal tárolhatunk el `x`-ben:

```
x.f = 786.3869;
```

Végül egy egész szám (count) felét így tölthetjük be `x`-be:

```
x.i = count / 2;
```

Mivel a lebegőpontos, a karakter és az egész típusú érték a memória ugyanazon területén tárolódik, egyszerre csak egy értéket tárolhatunk `x`-ben. Az aztán már a programozó felelőssége, hogy az utoljára eltárolt típust olvassa ki a változóból.

Az uniók adattagjaira ugyanazok az aritmetikai szabályok érvényesek, mint bármely más kifejezésre. Azaz a következő kifejezésben az egész aritmetika szerint történik az osztás, hiszen minden operandus egész:

```
x.i / 2
```

Egy uniónak bármennyi adattagja lehet. A C fordító gondoskodik arról, hogy a legnagyobb méretű adattag is beférjen a lefoglalt memóriaterületre. Adatszerkezeteknek vagy tömböknek is lehetnek elemei uniók. Egy unió megadásakor nem kell neki (mint típusnak) nevet adni – lehet egyből változónévhez is társítani a definíciót (akkár az adatszerkezetknél). Megadhatóak olyan mutatók, melyek uniókra irányulnak – ennek szintaxisa megegyezik az adatszerkezetre vonatkozó mutatók megadásával.

Az unió egyik adattagja inicializálható. Ha ilyenkor nem nevezünk meg adattagot, akkor az unió első adattagjához rendelődik a megadott érték:

```
union mixed x = { '#' };
```

Ebben az utasításban `x` első adattagját (`c`) állítjuk be a `#` karakter értékére.

Az adattag nevének megadásával az unió bármely adattagja inicializálható, mégpedig a következőképpen:

```
union mixed x = { .f = 123.456; };
```

Így az `x` változó `f` adattagját 123.456-ra állítjuk be.

Egy unió változó alapján inicializálható egy másik (ugyanolyan típusú) változó:

```
void foo (union mixed x)
{
```

```

union mixed y = x;
...
}

```

Itt a `foo` függvényben az `y` automatikus unió változót a függvény paramétere, `x` alapján inicializáljuk.

UniÓk felhasználásával megadhatunk olyan tömböket, melyek különböző típusú adatokat tárolnak. A következő utasítás a `table` tömböt hozza létre (`kTableEntries` elemszámmal):

```

struct
{
    char             *name;
    enum symbolType type;
    union
    {
        int      i;
        float   f;
        char    c;
    } data;
} table [kTableEntries];

```

A tömb minden eleme egy adatszerkezet, melyben szerepel egy `name` nevű karakter-mutató, egy `type` nevű felsorolt típusú változó, valamint egy `data` nevű uniÓ. A tömb valamennyi `data` adattagja tartalmazhat `int`, `float` vagy `char` típusú adatot. A `type` adattag használható a `data` adattagban tárolt típus dokumentálására. Ennek értéke lehet `INTEGER`, ha `int`-et tartalmaz a `data`; `FLOATING`, ha `float` a típusa és lehet `CHARACTER`, ha `char`-t tartalmaz. Ez alapján minden tudható, hogy miként kell hivatkozni a szükséges `data` adat-elemre.

Tegyük fel, hogy a '#' karaktert szeretnénk eltárolni a `table[5]`-ben, és a megfelelő `type` mezőt oly módon szeretnénk beállítani, hogy jelezze: karaktert tárolunk el. Ehhez a következő két értékadást használhatjuk:

```

table[5].data.c = '#';
table[5].type = CHARACTER;

```

Amikor a `table` elemeit végigjárjuk, minden tudható, hogy az adott helyen milyen típusú adat áll. Ehhez egy alkalmás lekérdezést kell megfogalmaznunk. Az alábbi ciklus kiírja az egyes tömbelemek neveit és a hozzájuk tartozó értéket:

```
enum symbolType { INTEGER, FLOATING, CHARACTER };
```

```

...
```

```

for ( j = 0; j < kTableEntries; ++j ) {
    printf ("%s ", table[j].name);

    switch ( table[j].type ) {
        case INTEGER:
            printf ("%i\n", table[j].data.i);
            break;
        case FLOATING:
            printf ("%f\n", table[j].data.f);
            break;
        case CHARACTER:
            printf ("%c\n", table[j].data.c);
            break;
        default:
            printf ("Ismeretlen típus (%i), %i. elem\n",
table[j].type, j );
            break;
    }
}

```

Az itt vázolt megoldás használható például egy szimbólum-táblázatban, amely a szimbólumok nevét, típusát és értékét tartja számon (és esetleg más adatait is).

A vessző operátor

Lehet, hogy az olvasó eddig nem tudatosította, hogy maga a vessző is egy operátor, amelyet különféle kifejezésekben használhatunk. A vessző operátor áll a precedencia-hierarchia legalján. Az 5. fejezetben tanultuk, hogy egynél több kifejezést is használhatunk a **for** ciklusfej-mezőkben, és hogy ezeket vesszővel kell egymástól elválasztani. Az alábbi **for** utasítás a ciklusmag első lefutása előtt i értékét 0-ra inicializálja, j értékét pedig 100-ra; a léptetéskor pedig i-t inkrementálja, j-t pedig tízesével csökkenti:

```

for ( i = 0, j = 100; i != 10; ++i, j -= 10 )
    ...

```

A vessző operátor bármely érvényes C kifejezés helyén használható két kifejezés elválasztására. A kifejezések báról jobbra értékelődnek ki. Így az alábbi ciklusmagban először a **data[i]** értéke hozzáadódik a **sum**-hoz, és csak ezután inkrementálódik i értéke:

```

while ( i < 100 )
    sum += data[i], ++i;

```

Zárójelre nincs szükségünk, hiszen a **while** ciklusfeltételt csak egyetlen utasítás követi (melyet a vessző operátor két utasítássá különít el).

A C nyelv operátorai minden visszaadnak valamilyen értéket. A vessző operátor esetén ez nem más, mint a jobboldali kifejezés.

Megjegyzendő, hogy a függvényhívásokban használt paraméter-elválasztó vessző vagy a változók felsorolásakor használt vessző önálló szintaktikai elem – nem tévesztendő össze a vessző operátorral.

Típusminősítők

Az alábbi típusminősítők változónevek előtt állhatnak. A fordítóprogram számára többlet-információt nyújtanak a változó célját illetően, így segítik a kód optimalizációját.

A register minősítő

Ha egy függvény gyakran használ egy adott változót, akkor kérhetjük a fordítóprogramtól, hogy a lehető leggyorsabb elérést biztosítsa számára. Általában ezt úgy oldják meg, hogy a változót valamelyik regiszterben tárolják a függvény végrehajtása közben. Ennek igényléséhez a változó deklarációja előre kell írnunk a register kulcsszót:

```
register int index;  
register char *textPtr;
```

Lokális változók és formális paraméterek is deklarálhatóak a register módosítóval. Az már erősen gépfüggő, hogy milyen típusú változók kerülhetnek be a regiszterekbe. Az alapvető adattípusok általában tárolhatóak regiszterben, valamint többnyire a mutatók és más adattípusok is.

A legtöbb fordítóprogram lehetővé teszi a register módosító használatát, ám ez nem jelenti azt, hogy ezt minden tekintetben veszi a kód generálásakor. A változók elhelyezésének kérdéséről a fordítóprogram dönt.

Fontos tudni, hogy register változóra nem használható a „címe” operátor. Ettől eltekintve azonban ugyanúgy használhatóak a register változók, mint bármely más szokványos automatikus változó.

A volatile minősítő

Ez igazából a const ellentétpárja. Arról tájékoztatja a fordítóprogramot, hogy ez a változó bizony *meg fog változni*. Ez arra lehet jó, hogy visszatartsa a fordítóprogramot attól, hogy kiküszöbölie például a látszólag felesleges hozzárendeléseket, vagy azt, hogy a látszólagos változatlansága ellenére újra és újra megvizsgáljuk egy változó értékét. Jó példa erre a kimeneti/bemeneti kapu kezelése. Tegyük fel, hogy van egy kimeneti kapunk, melyet

egy `outPort` nevű mutató jelöl. Ha ki akarunk írni két karaktert a portra, például egy O és egy N betűt, akkor ezt az alábbi kódrészlettel tudjuk megtenni:

```
*outPort = 'O';
*outPort = 'N';
```

Egy okos fordítóprogram észreveheti, hogy a két egymást követő hozzárendelés ugyanarra a változóra vonatkozik, és – mivel az `outPort` nem módosul közben – egyszerűen elhagyhatja az első értékadást. Ennek megelőzésére érdemes az `outPort`-ot `volatile` mutatóként deklarálni:

```
volatile char *outPort;
```

A restrict minősítő

A `register` minősítőhöz hasonlóan a `restrict` is a fordítás optimalizációját célozza. Mint ilyent, a fordítóprogram figyelmen kívül hagyhatja. Arra használhatjuk, hogy tudassuk a fordítóprogrammal, hogy az adott mutató az egyetlen (közvetlen vagy közvetett) hivatkozás az adott értékre az aktuális hatókörön belül. Azaz nincs több mutató vagy változó, amely az adott értéket tárolná (vagy arra mutatna) a hatókörön belül. Az alábbi sorokkal tájékoztathatjuk a fordítóprogramot arról, hogy az `intPtrA` és `intPtrB` hatókörében a két mutató soha nem fog azonos memóriaterületre mutatni:

```
int * restrict intPtrA;
int * restrict intPtrB;
```

Ha például egy tömb elemeire mutatunk velük, akkor ki van zárva, hogy azonos elemre mutassanak.

Parancssori paraméterek

A legtöbb program azt igényli, hogy a felhasználó adjon meg néhány adatot. Ez lehet egy szám (például a kiszámítandó háromszögszám sorszáma) vagy egy szótárban megkerendő szó stb.

A program ezeket az adatokat futás közben is bekérheti a felhasználótól, de sokszor hasznos lehet, ha már a program indításakor meg tudjuk adni őket. Ezt a lehetőséget a parancssori paraméterek használatával aknázhatsuk ki.

Korábban említettük, hogy a `main` eljárás egyetlen megkülönböztető jegye a neve. Innen tudja a fordítóprogram, hogy hol kezdődik a program végrehajtása. Valójában a program elindításakor a `main` függvényt hívja meg a C rendszer (pontosabban a futtatási rendszer) – ugyanúgy, ahogy bármilyen más függvényt is meg tudunk hívni egy C programban.

Amikor a `main` végrehajtása véget ér, a vezérlés visszakerül a futtatási rendszerbe, ami tudni fogja, hogy a program lefutott.

Amikor a `main`-t meghívja a futtatási rendszer, akkor két paramétert mindig átad az eljárásnak. Az elsőt megállapodásszerűen `argc`-nek hívják (*argument count, paraméterszám*): ez egy egész szám, amely megadja a parancssorban kapott paraméterek számát. A `main` második paramétere egy olyan tömb, amely karakterláncok mutatóit tartalmazza. Ezt `argv`-nek hívjuk (*argument vector, paraméter-vektor*). Ebben az egydimenziós tömbben (azaz vektorban) `argc + 1` mutató található; az `argc` minimális értéke 0. A tömb első eleme az éppen végrehajtott program saját nevére mutat (vagy null karakterláncra, ha a program neve nem érhető el a rendszer számára). Az ezt követő tömbelemek azokra a paraméterekre (mint karakterláncokra) mutatnak, melyeket a program elindításakor a felhasználó megadott a parancssorban. Az `argv` utolsó mutatója `argv[argc]`, amely definíciószerűen null értéket tartalmaz.

A parancssori paraméterek eléréséhez úgy kell megadni a `main` eljárást, hogy jelen legyenek a szükséges formális paraméterek. Ezeket általában így szokták deklarálni:

```
int main (int argc, char *argv[])
{
    ...
}
```

Jegyezzük meg, hogy az `argv` deklarációjákor egy olyan tömböt adunk meg, melynek elemei karakter-mutatók.

A parancssori paraméterek gyakorlati felhasználásának bemutatásához keressük elő a 10.10 Listát, amely megkeres egy szót a megadott szótárban, és kiírja annak jelentését. Adjuk meg parancssori paraméterként a keresendő szót:

```
lookup aerie
```

Ez szükségtelenné teszi, hogy a programnak fel kelljen szólítania a felhasználót a szó beírására (hiszen az már megérkezett parancssori paraméterként).

A fenti parancs kiadásakor a rendszer az `argv[1]`-ben átad a `main` eljárásnak egy karakterláncot, melynek tartalma "aerie". Az `argv[1]`-ben a program neve van, mely ebben az esetben lehet például "lookup".

A main eljárást módosítsuk a következőképpen:

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

int main (int argc, char *argv[])
{
    const struct entry dictionary[100] =
    { { "aardvark", "a burrowing African mammal" },
      { "abyss", "a bottomless pit" },
      { "acumen", "mentally sharp; keen" },
      { "addle", "to become confused" },
      { "aerie", "a high nest" },
      { "affix", "to append; attach" },
      { "agar", "a jelly made from seaweed" },
      { "ahoy", "a nautical call of greeting" },
      { "aigrette", "an ornamental cluster of feathers" },
      { "ajar", "partially opened" } };

    int entries = 10;
    int entryNumber;
    int lookup (const struct entry dictionary [], const char
search[],
                const int entries);

    if ( argc != 2 )
    {
        fprintf (stderr, "Nem adott meg keresendő szót
                   ↪ a parancssorban.\n");
        return EXIT_FAILURE;
    }

    entryNumber = lookup (dictionary, argv[1], entries);

    if ( entryNumber != -1 )
        printf ("%s\n", dictionary[entryNumber].definition);
    else
        printf ("Sajnos nem található a szótárban ilyen szó: %s\n",
               ↪ argv[1]);

    return EXIT_SUCCESS;
}
```

A main eljárás ellenőrzi, hogy a felhasználó megadta-e keresendő szót a program indításakor. Ha nem adta meg (vagy egynél többet adott meg), akkor az argc értéke nem 2; ezt egy feltételvizsgállal ellenőrzi a program, és hiba esetén egy visszajelző üzenet kíséretében kilép. A kilépési állapot ilyenkor az EXIT_FAILURE értéke lesz.

Ha viszont az argc értéke 2, akkor meghívódik a lookup függvény, hogy megkeresse az argv[1]-ben tárolt szót a szótárban. Ha megtalálta, megjeleníti a szó értelmezését.

A parancssori paraméterek használatának másik példájaként nézzük a 16.3 Listát, amely egy fájlmásoló program. A most következő 17.1 Lista erre épül: két fájlnevet kér be parancssori paraméterként, így futás közben már nem kell felhasználói bevitelre várnia.

17.1 Lista • Fájlmásolás parancssori paraméterekkel

```
// Egy fájlt egy másikba átmásoló program - 2. változat

#include <stdio.h>

int main (int argc, char    *argv[])
{
    FILE *in, *out;
    int    c;

    if ( argc != 3 ) {
        fprintf (stderr, "Két fájlnevet kell megadni.\n");
        return 1;
    }

    if ( (in = fopen (argv[1], "r")) == NULL ) {
        fprintf (stderr, "%s nem nyitható meg olvasásra.\n", argv[1]);
        return 2;
    }

    if ( (out = fopen (argv[2], "w")) == NULL ) {
        fprintf (stderr, "%s nem nyitható meg írásra.\n", argv[2]);
        return 3;
    }

    while ( (c = getc (in)) != EOF )
        putc (c, out);

    printf ("Az állomány másolása sikeresen véget ért.\n");

    fclose (in);
    fclose (out);

    return 0;
}
```

A program először ellenőrzi, hogy a program neve után minden paramétert megadta-e a felhasználó. Ha igen, akkor argv[1] a bemeneti állomány nevére mutat, argv[2] pe-

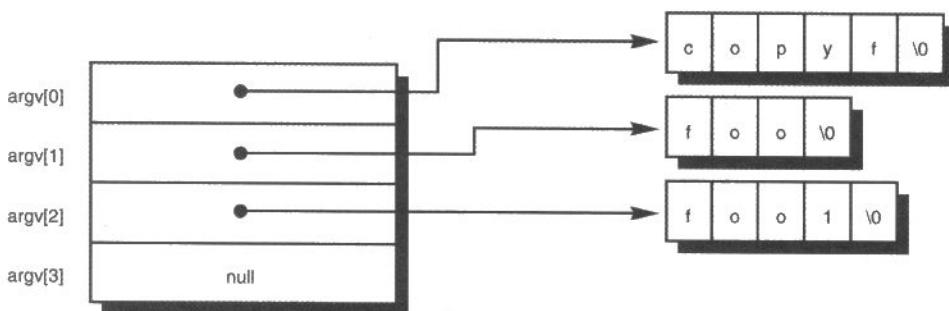
dig a célfájl nevére. Az előbbit olvasásra, az utóbbit írásra nyitja meg a program, és ha ez sikerült, akkor karakterről karakterre átmásolódik az egyik fájl a másikba (ahogy az az eredeti programban is történt).

Programunk futása négyféleképp érhet véget. Lehetséges, hogy helytelen számú parancssori paramétert ad meg a felhasználó. Lehet, hogy a másolandó fájt nem lehet megnyitni olvasásra. Az is lehet, hogy a célfájt nem sikerül megnyitni írásra. Végül az is elképzelhető, hogy minden jól megy, és a program normál módon lefut. Ha valamelyik másik program szeretné használni programunk kilépési állapotának számértékét, akkor ne felejtsük el, hogy minden futási esetre adjunk meg egy megfelelő értéket. Ha programunk úgy ér véget, hogy egyszerűen rácsorog a main eljárás végére, akkor definiáltalan marad a kilépési állapot.

Nevezzük a 17.1 Lista programját `copyf`-nek. Egy lehetséges futtatási módja a következő:

```
copyf foo foo1
```

A main eljárás kezdetén az argv tömböt a 17.1 ábrának megfelelően képzelhetjük el.



17. ábra

Az argv tömb a copyf futtatásának kezdetén

Ne felejtsük el, hogy a parancssori paraméterek *mindig* karakterláncban tárolódnak.

Futtassuk le a power (*hatvány*) nevű programot a következő két parancssori paraméterrel:

```
power 2 16
```

Ezzel keletkezik egy argv[1] karakterlánc, mely a "2"-t tartalmazza, valamint egy argv[2] karakterlánc, melyben a "16"-ot tartja a futtató rendszer. A program várhatólag számokat szeretne kapni (ha már hatványozni akarunk vele), így ezeknek az értékeknek

a típusát meg kell változtatni a programban. Erre a célra rendelkezésünkre állnak különféle eljárások a programkönyvtárban, például a `sscanf`, az `atof`, az `atoi`, az `strtod` és az `strtol`. Ezek leírását a B függelékben találjuk („A szabványos C programkönyvtár”).

Dinamikus memóriafoglalás

Amikor definiálunk egy C változót – legyen az egy alaptípus, vagy akár egy tömb vagy egy adatszerkezet – akkor akkor voltaképpen helyet foglalunk le számára a memóriában. A C fordító tudja, az adott típus számára mekkora hely kell.

Gyakran kívánatos, sőt esetenként megkerülhetetlen, hogy a program futása közben, dinamikusan foglaljunk le tárterületet. Képzeljünk el egy olyan programot, amely beolvas egy adatsort egy fájlból, majd betölti egy tömbbe. Könnyen elképzelhető, hogy nem tudjuk előre az adatok mennyiségét. Három választásunk van:

- A tömböt megpróbáljuk az elképzelhető legnagyobb elemszámúra beállítani már fordítási időben.
- Változó méretű tömböt használunk, amelynek méretét futási időben állítjuk be.
- Dinamikusan foglaljuk le a szükséges memóriát egy alkalmas memóriafoglaló eljárással.

Az első módszert követve a tömböt megpróbáljuk jó nagy méretűre definiálni. Sejtésünk az, hogy ezernél több adat nem kerül elő:

```
#define kMaxElements 1000  
  
struct dataEntry dataArray [kMaxElements];
```

Amíg az adatállományban ezernél kevesebb adat van, addig sínén vagyunk. Ha azonban túllépjük ezt a küszöbértéket, akkor meg kell változtatnunk a programban a `kMaxElements` értékét, majd újra le kell fordítanunk. Bármilyen nagy számot választunk is, megvan az esélye annak, hogy alulbecsüljük a szükséges tárterületet, és később is problémába ütközünk.

A második megközelítésnél meg kell tudnunk az adatok számát az adatsor beolvasása előtt (például a fájl méretéből). Ezután már létrehozhatunk egy megfelelő méretű tömböt a következők szerint:

```
struct dataEntry dataArray [dataItems];
```

Az utasítás futtatásakor azt feltételezzük, hogy a `dataItems` változó tárolja az adatok előbb említett mennyiségét.

A dinamikus memóriafoglaló függvényekkel pontosan annyi tárhelyet tudunk lefoglalni, amennyi szükséges, mégpedig a program futása közben. Ennek használatához azonban meg kell ismernünk három függvényt és egy új operátort.

A `calloc` és a `malloc` függvény

A szabványos C programkönyvtár tartalmaz két függvényt, `calloc` és `malloc` néven, melyekkel futási időben lehet lefoglalni memóriaterületet. A `calloc` függvény két paramétert vár. Az egyik tartalmazza a lefoglalandó egységek számát, a másik pedig az egyes egységek méretét bájtokban. A függvény egy mutatót ad vissza, amely a lefoglalt tárterület elejérre mutat. A tárterület automatikusan nullázásra kerül.

A `calloc` által visszaadott mutató `void` típusú, ami a C általános mutatótípusa. Mielőtt eltároljuk a programban ennek a mutatónak az értékét, érdemes típusátalakításnak alávetni, hogy a szükséges típusú adatmutatóként lehessen majd használni.

A `malloc` függvény is hasonlóan működik, de az csak egy paramétert vár – a lefoglalandó tárterület mennyiségett bájtokban. Ez a függvény is nullázza a tárterületet.

A dinamikus memóriafoglaló függvények definíciója a `<stdlib.h>` szabványos fejlécállományban található. Ezt be kell emelnünk azokba a programokba, melyek használni szereznék a fenti függvényeket.

A `sizeof` operátor

A `calloc` vagy `malloc` használatához tudnunk kell az egyes adattípusok méretét. Ezt gépfüggetlenül tudjuk meghatározni a C nyelv `sizeof` operátorával. A `sizeof` operátor a megadott adattípus (bájtokban megadott) méretével tér vissza. A `sizeof` operátor paramétere lehet egy változó, egy tömb neve, egy (alapvető vagy származtatott) adattípus neve vagy akár egy kifejezés. Például a

```
sizeof (int)
```

kifejezés visszaadja az `int` típushoz tartozó adatterület méretét bájtokban. Egy Pentium 4-es gépen ez az érték 4, mivel ez az architektúra 32 bites egész számokat használ. Ha `x` egy százelemű tömb, akkor a

```
sizeof (x)
```

kifejezés értéke megadja azt a tárterület-mennyiséget, amelyen száz egész számot lehet tárolni (Pentium 4-es gépeken ez 400). A

```
sizeof (struct dataEntry)
```

kifejezés egy `dataEntry` adatszerkezet számára szükséges memóriaterületet adja meg. Ha a `data` egy `dataEntry` adatszerkezetekből álló tömb, akkor a következő kifejezés adja meg az elemei számát:

```
sizeof (data) / sizeof (struct dataEntry)
```

A `data` változót a tömb előtt kell definiálni, és nem lehet formális paraméter vagy külsőleg hivatkozott tömb. Ugyanezt az eredményt adja a következő kifejezés:

```
sizeof (data) / sizeof (data[0])
```

Ezt egy makróban is megfogalmazhatjuk, amivel nagyban növelhetjük programunk olvashatóságát:

```
#define ELEMENTS(x) (sizeof(x) / sizeof(x[0]))
```

Ennek a makrónak a révén a következő kód részletek fogalmazhatóak meg:

```
if ( i >= ELEMENTS (data) )  
...
```

és

```
for ( i = 0; i < ELEMENTS (data); ++i )  
...
```

Ne felejtsük, hogy a `sizeof` egy operátor és nem függvény, bár úgy néz ki, mint egy függvény. Ez az operátor fordítási időben (!) értékelődik ki, nem pedig futási időben, ha csak nem változó méretű tömb a paramétere. Egyéb esetekben maga a fordítóprogram számítja ki a `sizeof` kifejezés értékét, ami innentől kezdve konstansnak tekinthető.

Ahol csak lehet, használjuk ki a `sizeof` operátor által nyújtotta általánosítási lehetőségeket. Ezzel sok „adat-beledrótozást” elkerülhetünk programjainkban.

Térjünk vissza a dinamikus memória foglaláshoz. Ezer egész számnak a következőképpen tudunk tárhelyet lefoglalni a `calloc` utasítással:

```
#include <stdlib.h>  
...  
int *intPtr;  
...  
intPtr = (int *) calloc (sizeof (int), 1000);
```

Hasonló eredményt érhetünk el a `malloc` használatával:

```
intPtr = (int *) malloc (1000 * sizeof (int));
```

Ne felejtsük, hogy a `malloc` és az `alloc` is `void` mutatót ad vissza, amit át kell alakítanunk, hogy a megfelelő adattípusra irányuló mutatóként használhassuk. Előző példáinkban a mutatót egész mutatóvá alakítottuk, és úgy adtuk értékül az `intPtr`-nek.

Ha nem áll rendelkezésre elegendő memória a kívánt mennyiséggű adat számára, akkor a `calloc` (és a `malloc`) nullmutatót ad vissza. Mindkét függvény használatakor győződjünk meg a memóriafoglalás sikereségéről, mielőtt továbbbenedjük a vezérlést.

A következő kódrészlet ezer egész számára foglal le tárterületet, és ellenőrzi is a visszaadott mutató helyességét. Ha a memóriafoglalás meghiúsul, a program hibaüzenetet ír ki, majd kilép.

```
#include <stdlib.h>
#include <stdio.h>
...
int *intPtr;
...
intptr = (int *) calloc (sizeof (int), 1000);

if ( intPtr == NULL )
{
    fprintf (stderr, "A calloc nem járt sikerrel.\n");
    exit (EXIT_FAILURE);
}
```

Ha sikeres a memóriafoglalás, akkor az `intPtr` egész mutató felhasználható úgy, mint egy egészeket tartalmazó, 1000 elemszámú tömb mutatója. Ha például minden elemet -1-re szeretnénk beállítani, akkor azt így tehetjük meg:

```
for ( p = intPtr; p < intPtr + 1000; ++p )
    *p = -1;
```

Ehhez `p`-nek egy egészre irányuló mutatónak kell lennie.

Ha egy `n` elemű, `dataEntry` adatszerkezeteket tartalmazó tömb számára szeretnénk tárterületet lefoglalni, akkor ehhez először létre kell hozni a megfelelő mutatótípust:

```
struct dataEntry *dataPtr;
```

Ezután már meghívható a `calloc` függvény a megfelelő méretű memóriaterület lefoglalására.

```
dataPtr = (struct dataEntry *) calloc (n, sizeof (struct dataEntry));
```

A hozzárendelés lépései a következők:

1. A `calloc` függvényt két paraméterrel hívjuk meg, az első adja meg a lefoglalandó adategységek számát (ami `n`), a második pedig az egyes elemek méretét.
2. A `calloc` függvény visszatérési értéke egy mutató, amely a memória azon pontjára mutat, ahol a lefoglalt terület kezdődik. Ha nem volt sikeres a memóriafoglalás, akkor a visszatérési érték egy nullmutató.
3. A mutatót „egy `dataEntry` adatszerkezetre irányuló mutató” típusúvá alakítjuk, majd értékül adjuk a `dataPtr`-nek.

Ismét hangsúlyozzuk, hogy a `dataPtr`-t ellenőrizni kell annak érdekében, hogy biztosak lehessünk a memóriafoglalás sikerességében. Ekkor ugyanis ez nem lehet nullmutató.

Ha meggyőződtünk a mutató helyességeről, akkor ugyanúgy használhatjuk, mint bármely más mutatót, amely egy `n` elemű, `dataEntry` elemeket tartalmazó tömbre irányul.

Ha a `dataEntry` típus egyik adattagját `index`-nek definiáltuk, akkor a következő utasítással rendelhetünk a `dataPtr` által mutatott elemhez 100-as értéket:

```
dataPtr->index = 100;
```

A free függvény

Ha befejeztük a munkát a `calloc` vagy a `malloc` függvénnel lefoglalt memóriaterületen, érdemes azt felszabadítani, vagyis visszaadni a rendszernek. Ezt a `free` függvény segítségével tehetjük meg. A függvény egyetlen paramétere egy mutató, amely a lefoglalt memóriaterület elejére mutat (amit annak idején a `calloc`-tól vagy a `malloc`-tól visszakaptunk). Így, ha a `dataPtr` a lefoglalt memóriaterület *elejére* mutat, akkor a

```
free (dataPtr);
```

utasítás visszaadja a rendszernek a `calloc` által (egy fenti kód részletben) lefoglalt tárterületet.

A `free` függvénynek nincs visszatérési értéke.

A `free` által felszabadított memóriaterület később újra felhasználható a `calloc` vagy a `malloc` meghívásával. Ez nagyon hasznos lehet olyan programok esetén, amelyeknek nagyobb memóriaigényük van, mint amennyit egyszerre le lehetne foglalni. Figyeljünk arra, hogy a `free` függvénynek olyan érvényes mutatót adjunk át, amely egy előzőleg lefoglalt memóriaterület elejére mutat.

A dinamikus memóriafoglalás felbecsülhetetlen értékű a láncolt adatszerkezetek esetében, például a láncol listáknál. Ha egy új bejegyzést szeretnénk létrehozni a listában, akkor egyszerűen lefoglaljuk a szükséges memóriaterületet az adat számára, és a (`malloc`-tól vagy `calloc`-tól) visszakapott mutató segítségével a bejegyzést becsatoljuk a lista megfelelő helyére. Tegyük fel, hogy a `listEnd` egy olyan egyszeresen láncolt lista végpontjára mutat, melynek elemei `entry` adatszerkezetűek:

```
struct entry
{
    int           value;
    struct entry *next;
};
```

Álljon itt egy `addEntry` nevű függvény, amely paraméterként egy láncolt lista kezdetére irányuló mutatót kér, és amely be tud fűzni egy új bejegyzést a lista végére.

```
#include <stdlib.h>
#include <stddef.h>

// új bejegyzés hozzáadása a láncolt listához

struct entry *addEntry (struct entry *listPtr)
{
    // a lista végének megkeresése

    while ( listPtr->next != NULL )
        listPtr = listPtr->next;

    // az új bejegyzés számára memóriát foglalunk le

    listPtr->next = (struct entry *) malloc (sizeof (struct entry));

    // a lista új végét nullal jelezzük

    if ( listPtr->next != NULL )
        (listPtr->next)->next = (struct entry *) NULL;

    return listPtr->next;
}
```

Sikerességek memóriafoglalás esetén nullmutató kerül a frissen befűzött bejegyzés `next` (*következő*) adattagjába (amelyre a `listPtr->next` mutatott).

A függvény visszatérési értéke nullmutató, ha nem sikerült a memóriafoglalás, ellenkező esetben pedig a frissen létrehozott bejegyzésre irányuló mutató. Érdemes ezt ellenőrizni,

hogy valóban ez történik-e. Ehhez egy rajzot is készíthetünk egy láncolt listáról, melyen lépésről lépésre végigkövethetjük az `addEntry` futását. Ez sokat segíthet a fenti program megértésében.

Van még egy függvény, a `realloc`, amely szintén a dinamikus memóriafoglalással kapcsolatos. Ezt használhatjuk fel arra, hogy egy korábban lefoglalt memóriaterületet bővítsünk vagy szűkítsünk. Részleteket a B függelékben találhat az olvasó.

Ezzel a végére is értünk ennek a fejezetnek. Immár a C nyelv valamennyi részterületéről szó esett. A következő fejezetben arról lesz szó, hogy miként követhetjük nyomon programjaink futását. Ehhez az egyik út az előfeldolgozón át vezet, egy másik lehetőség pedig egy speciális eszköz, az interaktív nyomkövető (*debugger*) használata.

18

Programok nyomkövetése

Ez a fejezet két módszert mutat be a programok nyomkövetésére. Az egyik lehetőség az előfeldolgozó használata, amely lehetővé teszi, hogy a nyomkövetést célzó utasítások csak bizonyos feltételek mellett forduljanak le. A másik módszer egy interaktív nyomkövető használata. Én a népszerű gdb nyomkövetőt fogom ismertetni. Ha más nyomkövetőt használunk is (például a dbx-et vagy valamilyen integrált fejlesztői környezet saját nyomkövetőjét), az is bizonyára a gdb-hez hasonló elvek szerint működik.

Nyomkövetés az előfeldolgozó segítségével

A 13. fejezetben az előfeldolgozó kapcsán már volt szó a feltételes fordításról, és arról is, hogy ez a lehetőség felhasználható a programok nyomkövetésére. A C előfeldolgozó felhasználható arra, hogy nyomkövetési utasításokat tegyünk a forráskódba. Alkalmas `#ifdef` utasításokkal a nyomkövetési parancsok ki-/bekapcsolhatóak döntésünk szerint. A 18.1 Listában bemutatott (bevallottan mesterkélt) program beolvás három egész számot, majd kinyomtatja az összegüket. Amikor az előfeldolgozó definiált állapotban találja a DEBUG (nyomkövetés) azonosítót, akkor a nyomkövetési kódrészletek (amelyek az stderr-be nyomtatnak) belefordulnak a programba, ám amikor nincs definiálva a DEBUG, akkor kimaradnak a programból.

18.1 Lista • Nyomkövetési utasítások beillesztése az előfeldolgozóval

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

int process (int i, int j, int k)
{
    return i + j + k;
}

int main (void)
{
    int i, j, k, nread;

    nread = scanf ("%d %d %d", &i, &j, &k);
```

```
#ifdef DEBUG
    fprintf (stderr, "A beolvasott egészek száma: %i\n", nread);
    fprintf (stderr, "i = %i, j = %i, k = %i\n", i, j, k);
#endif

    printf ("%i\n", process (i, j, k));
    return 0;
}
```

18.1 Lista • Kimenet

```
1 2 3
A beolvasott egészek száma: 3
i = 1, j = 2, k = 3
6
```

18.1 Lista • Kimenet (újrafuttatás)

```
1 2 e
A beolvasott egészek száma: 2
i = 1, j = 2, k = -1208553484
-1208553481
```

Megjegyzendő, hogy a második futtatáskor k értéke bármi lehet, hiszen ezt nem olvassa be a scanf, így nem inicializálódik. Az alábbi kódrészletet az előfeldolgozó értelmezi:

```
#ifdef DEBUG
    fprintf (stderr, "A beolvasott egészek száma: %i\n", nread);
    fprintf (stderr, "i = %d, j = %d, k = %d\n", i, j, k);
#endif
```

Ha a DEBUG azonosító defined (azaz definiált) állapotú, akkor az előfeldolgozó engedélyezi az #ifdef DEBUG és a következő #endif közti sorok lefordítását a fordítóprogram számára. Ha a DEBUG nem definiált, akkor a két fprintf-et nem is látja a fordítóprogram, mert az előfeldolgozó „kivágja” ezeket a sorokat. A kimenetből látható, hogy a program a számok beolvasása után írja ki az üzeneteket. A második futtatáskor érvénytelen karaktert adtunk meg (e). A nyomkövetési kimenet tájékoztat a hibáról. A nyomkövetés kikapcsolásához minden sort kell törölni (vagy megjegyzésbe tenni):

```
#define DEBUG
```

Ekkor az fprintf utasítások nem fordulnak le a program többi részével együtt. Ez a kis program kellően rövid ahhoz, hogy feleslegesnek érezze az ember az efféle szórszálhasogatást. De egy több száz soros programnál már komoly haszonnal járhat ez a lehetőség, hogy egyetlen sor törlésével ki- és be tudjuk kapcsolni a nyomkövetési információk kijelzését.

Akár a parancssori indításkor is beállítható a DEBUG azonosító definiált állapota. Ha gcc fordítót használunk, akkor a következő parancssal kérhető a nyomkövetési információkat is kijelző kód fordítása:

```
gcc -D DEBUG debug.c
```

Ez teljesen egyenértékű azzal, mintha beírnánk a debug.c forráskódjába a

```
#define DEBUG
```

sort.

Nézzünk meg egy kissé hosszabb programot. A 18.2 Lista két parancssori paramétert vár. Ezeket egészekké alakítja a program, majd betölti az arg1 és arg2 változókba. Egy parancssori paraméter egésszé alakításához az atoi szabványos könyvtári függvény használható, amely egy karakterláncot vár paraméterként, és a vele megegyező alakú egész számot adja vissza. Az atoi függvény a <stdlib.h> fejlécállományban található – a 18.2 Lista elején ezt is beemeljük.

A paraméterek feldolgozása után a meghívjuk a process függvényt, átadva neki a kapott két parancssori paraméter egésszé alakított értékét. A process függvény egyszerűen összeszorozza a két kapott számot. Amint láthatják, a DEBUG azonosító definiált állapotában különféle nyomkövetési információk is megjelennek; ha azonban nincs definiálva ez az azonosító, akkor csak a végeredményt jelzi ki a program.

18.2 Lista • Nyomkövetési kód részlet programba fordítása

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

int process (int i1, int i2)
{
    int val;

#ifndef DEBUG
    fprintf (stderr, "process (%i, %i)\n", i1, i2);
#endif
    val = i1 * i2;
#ifndef DEBUG
    fprintf (stderr, "return %i\n", val);
#endif
    return val;
}
```

```

int main (int argc, char *argv[])
{
    int arg1 = 0, arg2 = 0;

    if (argc > 1)
        arg1 = atoi (argv[1]);
    if (argc == 3)
        arg2 = atoi (argv[2]);
#ifndef DEBUG
    fprintf (stderr, "%i paraméter feldolgozva\n", argc - 1);
    fprintf (stderr, "arg1 = %i, arg2 = %i\n", arg1, arg2);
#endif
    printf ("%i\n", process (arg1, arg2));

    return 0;
}

```

18.2 Lista • Kimenet

```

$ gcc -D DEBUG p18-2.c      DEBUG beállításával fordítva
$ a.out 5 10
2 paraméter feldolgozva
arg1 = 5, arg2 = 10
process (5, 10)
return 50
50

```

18.2 Lista • Kimenet (újralfuttatás)

```

$ gcc p18-2.c                DEBUG beállítása nélkül fordítva
$ a.out 2 5
10

```

Amikor a program elkészült, és átadjuk a végfelhasználóknak, akkor sem kell kivenni a nyomkövetési utasításokat a forráskódóból – elég, ha a DEBUG azonosító nincs bekapsolva. Ha később kiderül egy hiba, akkor könnyen újra lehet fordítani a programot a (hiba okát feltáró) nyomkövetési utasítások kijelzésével.

Ez a módszer nem az igazi, mert rontja a program olvashatóságát. A probléma azonban áthidalható, csak megfelelő módon kell használnunk az előfeldolgozót.

Hozzunk létre egy makrót, amely változó számú paramétert tud feldolgozni:

```
#define DEBUG(fmt, ...) fprintf (stderr, fmt, __VA_ARGS__)
```

Ha az `fprintf` helyett ezt az utasítást használjuk:

```
DEBUG ("process (%i, %i)\n", i1, i2);
```

akkor a fenti makró így fogja helyettesíteni a `DEBUG`-ot:

```
fprintf (stderr, "process (%i, %i)\n", i1, i2);
```

Innenől kezdve a `DEBUG` függvényt széltében-hosszában használhatjuk; teljesen világosan látszik a programozó szándéka, amint azt a 18.3 Lista is mutatja.

18.3 Lista • A DEBUG makró használata

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

#define DEBUG(fmt, ...) fprintf (stderr, fmt, __VA_ARGS__)

int process (int i1, int i2)
{
    int val;

    DEBUG ("process (%i, %i)\n", i1, i2);
    val = i1 * i2;
    DEBUG ("return %i\n", val);

    return val;
}

int main (int argc, char *argv[])
{
    int arg1 = 0, arg2 = 0;

    if (argc > 1)
        arg1 = atoi (argv[1]);
    if (argc == 3)
        arg2 = atoi (argv[2]);

    DEBUG ("A beolvasott egészek száma: %i\n", argc - 1);
    DEBUG ("arg1 = %i, arg2 = %i\n", arg1, arg2);
    printf ("%d\n", process (arg1, arg2));

    return 0;
}
```

18.3 Lista • Kimenet

```
$ gcc pre3.c
$ a.out 8 12
A beolvasott egészek száma: 2
arg1 = 8, arg2 = 12
process (8, 12)
return 96
96
```

A program olvashatosága látványosan javult ezzel a megoldással. Amikor már nincs szükség a nyomkövetési kimenetre, egyszerűen „semmiként” kell definiálni a DEBUG makrót:

```
#define DEBUG(fmt, ...)
```

Így az előfeldolgozó a DEBUG makró előfordulásait null utasításként fogja átadni a fordítóprogramnak.

A DEBUG makrót még kifinomultabban is meg lehet fogalmazni. Megoldható, hogy ne csak fordítási időben, hanem futási időben is vezérelhessük a nyomkövetést. Hozzunk létre egy Debug nevű globális változót a nyomkövetési szint (*debug level*) jelzéséhez. Mindazok a DEBUG utasítások, amelyek nem nagyobbak a megadott nyomkövetési szintnél, végrehajtódnak (azaz megjelenítik a rájuk bízott információt). A DEBUG makró ilyenkor legalább két paramétert vár, melyből az első a nyomkövetési szint:

```
DEBUG (1, "Adatfeldolgozás\n");
DEBUG (3, "Az elemek száma: %i\n", nelems)
```

Ha a nyomkövetési szint 1 vagy 2, akkor csak az első DEBUG utasításnak lesz hatása a kimenetre. Ha azonban a nyomkövetési szint 3 vagy még magasabb, akkor minden DEBUG utasítás működésbe lép. A nyomkövetési szint egy parancssori kapcsolóval is állítható (futási időben, újrafordítás nélkül):

a.out -d1	A nyomkövetési szint 1-re állítva
a.out -d3	A nyomkövetési szint 3-re állítva

A DEBUG makró definíciója nem bonyolult:

```
#define DEBUG(level, fmt, ...) \
    if (Debug >= level) \
        fprintf (stderr, fmt, __VA_ARGS__)
```

Így a

```
DEBUG (3, "Az elemek száma: %i\n", nelems)
```

a következőképpen alakul át:

```
if (Debug >= 3)
    fprintf (stderr, "Az elemek száma: %i\n", nelems);
```

Itt is igaz az, hogy ha a DEBUG-ot „semmiként” definiáljuk, akkor az összes DEBUG hívás null utasítássá válik.

Az alábbi definíció minden eddig említett összetevőt tartalmaz, még a DEBUG fordítási időben történő definíciójának vezérlését is:

```
#ifdef DEBON
# define DEBUG(level, fmt, ...) \
    if (Debug >= level) \
        fprintf (stderr, fmt, __VA_ARGS__)
#else
# define DEBUG(level, fmt, ...)
#endif
```

Célszerű ezt a kódrészletet egy alkalmas fejlécállományban elhelyezni, és onnan beemelni a programba. Amikor egy olyan programot fordítunk, amely ezt tartalmazza, akkor a fordítás pillanatában eldönthetjük, hogy be legyen-e kapcsolva a DEBON azonosító, vagy sem. Ha így indítjuk a fordítást:

```
$ gcc prog.c
```

akkor a DEBUG makróhívások null utasításként fognak szerepelni, hiszen az #else ágban ez áll. Ha azonban bekapsoljuk a DEBON-t:

```
$ gcc -D DEBON prog.c
```

akkor fprintf-ként kerül beillesztésre a DEBUG makró, amely ráadásul a nyomkövetési szinteket is fogja tudni értelmezni.

Amennyiben bekapcsolt nyomkövetéssel fordítottuk a programot, futtatáskor parancssorból választható meg a kívánt nyomkövetési szint, például 3-as értékre:

```
$ a.out -d3
```

Ezt a parancssori paramétert feltehetően tudja értelmezni a programunk. A nyomkövetési szintet tárolhatjuk például egy `Debug` nevű globális változóban. A fenti utasítás hatására csak a hármas vagy annál magasabb szintű nyomkövetési kimenetek jelennek majd meg.

Jó tudni, hogy az `a.out -d0` hatására a nyomkövetési szint nullára áll, és semmilyen nyomkövetési kiírás nem jelenik meg, bár a kódban be van kapcsolva a nyomkövetés lehetősége.

Összegezve: a nyomkövetésről két szinten dönthetünk. A nyomkövetési kód részletek lefordítását ki- és bekapcsolhatjuk. Ha befordítjuk a megfelelő parancsokat programunkba, a különféle nyomkövetési szintek megadásával még szabályozhatjuk a program pillanatnyi állapotáról történő visszajelzés mértékét.

Nyomkövetés gdb-vel

A `gdb` egy hatékony, interaktív nyomkövető, amit gyakran használnak a GNU `gcc`-vel fordított programok elemzéséhez. Lehetővé teszi, hogy a program futtatásakor megállíjjunk egy adott ponton, megnézzünk vagy/és átállítsunk bizonyos változókat, majd továbbengedjük a program futását. Lépésről lépésre is futathatjuk a programot, figyelve közben a változásokat. A `gdb` akkor is használható, ha egy program lefagyásakor „core dump”, állapotjelző összesítés képződik. Ez olyankor szokott bekövetkezni, amikor valami nem várt esemény történik, például nullával való osztás vagy egy tömb határán túl megkísérelt olvasás. Ekkor a rendszer létrehoz egy „core”-nak (*mag*) nevezett állományt, amely a folyamathoz tartozó memória állapotát egy „pillanatképben” összefoglalja.¹

Ha a `gdb` nyomkövetőt szeretnénk használni, akkor programunkat a `-g` kapcsolóval kell lefordítani. Ennek hatására a C fordító egyéb információkat is belefordít a programba, mint például a változók, típusok, forrástárral törökített neveit, valamint a forráskódbeli C utasítások és a megfelelő gépi kódú utasítások egymáshoz rendelési információit.

A 18.4 Lista egy olyan programot mutat, amely egy tömb vége után kísérel meg egy elemet kiolvasni.

18.4 Lista • Egy egyszerű program a `gdb` használatának bemutatásához

```
#include <stdio.h>

int main (void)
{
```

¹ Az operációs rendszer beállítható úgy is, hogy ne keletkezzen `core` állomány – okozhat ugyanis gondot ezeknek a nagyméretű állapotjelző összesítéseknek a létrejötte. Az `ulimit` parancssal megadható egy küszöbérték a létrehozható fájlok maximális méretére vonatkozóan – ezzel kézben tarthatjuk a `core` állományok kritikus mértékű felgyülemlését.

```
const int data[5] = {1, 2, 3, 4, 5};  
int i, sum;  
  
for (i = 0; i >= 0; ++i)  
    sum += data[i];  
  
printf ("sum = %i\n", sum);  
  
return 0;  
}
```

Ha a programot lefuttatjuk, akkor (Mac OS X vagy Linux rendszeren) a következő hibaüzenetet kapjuk:

```
$ a.out  
Szegmentálási hiba
```

Nyilván operációs rendszerenként más és más lehet a konkrét hibaüzenet. Használjuk a gdb-t a hiba kinyomozásához. Ez egy nyilvánvalóan mesterkélt példa, ám jól bemutatható rajta az interaktív nyomkövetés módszere.

Először is fordítsuk le a programot a -g kapcsolóval, majd a gdb irányításával indítsuk el a futtatható állományt. Ennek hatására az alábbi(hoz hasonló) üzenetek jelennek meg a képernyőn:

```
$ gcc -g p18.4.c          A program újrafordítása a gdb számára  
$ gdb a.out               szükséges információkkal  
                           A futtatható kód gdb alatti elindítása  
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec 4 15:41:30 GMT  
2003)  
Copyright 2003 Free Software Foundation, Inc.  
GDB is free software, covered by the GNU General Public License, and  
you are  
welcome to change it and/or distribute copies of it under certain  
conditions.  
Type "show copying" to see the conditions.  
There is absolutely no warranty for GDB. Type "show warranty" for  
details.  
This GDB was configured as "powerpc-apple-darwin".  
Reading symbols for shared libraries .. done
```

Amikor a gdb felkészült a parancsok fogadására, kijelzi a (gdb) készenléti jelet (promptot). Ebben az egyszerű példában elég annyit tennünk, hogy elindítjuk a program futását a run parancssal.

Ennek hatására a gdb elindítja a programot, amely a hiba bekövetkeztéig fut:

```
(gdb) run
Starting program: /Users/stevekochan/MySrc/c/a.out
Reading symbols for shared libraries . done

Program received signal EXC_BAD_ACCESS, Could not access memory.
0x000001d7c in main () at p18-4.c:9
9          sum += data[i];
(gdb)
```

Programunk tehát hibára futott (mint korábban), de a vezérlés a gdb-nél maradt. Ez igen kedvező a számunkra, mert most megvizsgálható, hogy mi volt a helyzet a hiba bekövetkezének pillanatában. Megvizsgálhatóak például a változók értékei.

A fenti kijelzésből látható, hogy a program a 9. sorban megkísérelt egy szabálytalan memóriaelérést. A forráskód érintett sora megjelent a hibajelzésben. Ha kíváncsiak vagyunk a szövegkörnyezetre, akkor egy list parancs kiadásával a környező tíz sort is megnézhetjük (5 sort előtte és 4 sort utána).

```
(gdb) list 9
4          {
5              const int data[5] = {1, 2, 3, 4, 5};
6              int i, sum;
7
8              for (i = 0; i >= 0; ++i)
9                  sum += data[i];
10
11             printf ("sum = %i\n", sum);
12
13             return 0;
(gdb)
```

A változók tartalmát a print utasítással nézhetjük meg. Mi volt vajon a sum értéke a program leállásának pillanatában?

```
(gdb) print sum
$1 = -1089203864
```

Ez egy nyilvánvalóan eszement érték (és egy másik számítógépen nyilván más érték láttható). A gdb \$n-nel jelöli a kiírt értékeket, hogy a későbbiekben könnyebben hivatkozhassunk rájuk.

Nézzük, mi áll az `i` ciklusváltozóban:

```
(gdb) print i
$2 = 232
```

Ejnye! Ez így nem helyes. A tömbnek csak öt eleme van, mi pedig a hiba pillanatában a 233. elemet akartuk kiolvasni. Ha az olvasó kipróbálja a programot, elképzelhető, hogy más érték adódik – de a hibajelzésnek minden gépen meg kell érkeznie.

Mielőtt kilépünk a `gdb`-ből, vessünk egy pillantást a `data` tömbre. Örööm nézni, milyen kifejezően bánik a nyomkövető ezekkel az adatokkal:

<pre>(gdb) print data \$3 = {1, 2, 3, 4, 5}</pre>	<i>Megnézzük a data tömböt</i>
<pre>(gdb) print data[0]</pre>	<i>Megnézzük a tömb első elemét</i>
<pre>\$4 = 1</pre>	

A későbbiekben egy adatszerkezetet is megvizsgálunk. Első példánk befejeztével lépjünk ki a nyomkövetőből. Ezt a `quit` (*kilépés*) parancsal tehetjük meg:

```
(gdb) quit
The program is running. Exit anyway? (y or n) y
$
```

Annak ellenére, hogy a program hibára futott, még aktív volt a nyomkövetőben. A hiba nem okozta a program leállását, csak felfügesztését. Emiatt kért megerősítést kilépési szándékunkról a nyomkövető.

A változók kezelése

A `gdb`-ben két fő parancs áll rendelkezésünkre a változók kezeléséhez. Az egyiket már láttuk: ez a `print`. A másik parancs lehetővé teszi egyes változók beállítását: ez pedig a `set var` utasítás. A `set` igazából több opciót is elfogad – számunkra most a `var` az érdekes, mert azzal tudunk betölteni egy változóba egy adott értéket.

<pre>(gdb) set var i=5 (gdb) print i \$1 = 5</pre>	<i>Bármilyen érvényes kifejezést írhatunk</i>
<pre>(gdb) set var i=i*2 (gdb) print i \$2 = 10</pre>	<i>Használhatunk ún. "kényelmi változókat"</i>
<pre>(gdb) set var i=\$1+20 (gdb) print i \$3 = 25</pre>	

Csak olyan változót használhatunk, amely elérhető az adott függvényből. A folyamatnak „aktívnak” kell lennie, azaz futnia kell. A gdb számon tartja az „aktuális sort” (akár egy szövegszerkesztő), az „aktuális állományt” (amely a megfelelő forráskód) és az „aktuális függvényt”. Amikor a gdb core fájl nélkül indul el, akkor az aktuális függvény a main, az aktuális állomány a main-t tartalmazó állomány, és az aktuális sor a main első utasítása. Ha rendelkezésünkre áll a core állomány, akkor az aktuális sor, fájl és függvény az, ahol a hiba történt.

Ha nem létezik a megadott nevű lokális változó, akkor a gdb ugyanilyen nevű külső változót kezd el keresni. Előző példánkban a main-ben történt a hiba, és az i lokális változó a main-ben.

Ha egy adott függvény adott változójára vagyunk kíváncsiak, azt is megtudhatjuk. Ilyenkor a függvény::változó szintaxist kell használnunk, például így:

(gdb) print main::i	<i>Kiírjuk a main-beli i értékét</i>
\$4 = 25	
(gdb) set var main::i=0	<i>Beállítjuk a main-beli i értékét 0-ra</i>

Ha egy olyan változót szeretnénk beállítani, amely egy inaktív függvényhez tartozik (azaz éppen nem fut a függvény, vagy várakozik más függvény visszatérésére), akkor az alábbi hibajelzést kapjuk:

No symbol "var" in current context.

Globális változókra a 'file'::var megnevezéssel hivatkozhatunk. Ez a file-beli külső változó elérésére utasítja a gdb-t, és ha az adott függvényben van ugyanilyen nevű helyi változó, akkor attól eltekint.

Az adatszerkezetek és uniók adattagjai a szokásos C szintaxissal érhetőek el.

Ha a datePtr egy date adatszerkezetre mutat, akkor a print datePtr->year kiírja a datePtr által mutatott adatszerkezet év adattagját.

Ha egy adatszerkezetre vagy unióra úgy hivatkozunk, hogy nem adunk meg adattagot, akkor a teljes adathalmazt megmutatja a gdb.

Az is megadható a gdb-nek, hogy milyen formátumban jelenítse meg az egyes változókat. A kiírási parancs után egy perjellel védett karakterrel tudunk formázási utasítást megfogalmazni. Sok gdb utasítás egyetlen betűvel is rövidíthető. Az alábbi példában a print utasítást p-vel rövidítjük:

(gdb) set var i=35	<i>Az i-t 35-re állítjuk be</i>
(gdb) p /x i	<i>Hexadecimálisan íratjuk ki az i értékét</i>
\$1 = 0x23	

A forráskód megjelenítése

A gdb-ben több parancs is rendelkezésünkre áll a forráskód megjelenítéséhez. Ez nagy előny, mert ily módon a nyomkövetéshez nem kell nyitva tartani egy szövegszerkesztőben az összes forráskód-állományt.

Említettük, hogy a gdb számon tartja az aktuális sort és állományt. Láttuk, hogy a `list` utasítással (melyet 1-nek is rövidíthetünk) kiíratható egy adott sor környezete. Az ismételt `list` utasítás (vagy akár csak az Enter leütése) után egymás után megjelenik a folytatónak következő tíz-tíz sor. Maga ez a 10-es szám csak egy alapértelmezett érték, és átalítható a `listsize` parancssal.

Ha egy adott tartomány szeretnénk kiíratni, ahoz vesszővel elválasztva meg kell adnunk a kezdő és a záró sor számát:

(gdb) `list 10,15`

A 10-től 15-ig terjedő sorok kiírása

A `list` parancssal név szerint (!) is megadhatjuk, mely függvényre vagyunk kíváncsiak:

(gdb) `list függveny`

A függveny sorainak kiírása

Az sem okoz gondot, ha a függvény nem az aktuális állományban van – a gdb akkor is megkeresi. Az aktuális állományt egyébként az `info source` parancssal tudjuk megkérdezni a gdb-től.

A `list` utasítás után megadott + jelrel a következő tíz sort írathatjuk ki (ugyanúgy, mint egy ismételt `list` parancssal), – jelrel pedig az előző tíz sort. A + és – jelhez megadhatunk egy számértéket is, amely az aktuális sorhoz viszonyítottan jelzi ki a kívánt sorokat.

A program végrehajtásának vezérlése

Egy forráskód-állomány valamely sorának a megjelenítése még nem jelenti a program futásának a megváltoztatását; ehhez más parancsokra van szükség. Két parancsot már látunk: az egyik a `run` (*futtatás*) – ez a program elejtől indítja a végrehajtást, a másik pedig a `quit` (*kilépés*), ami befejezi a program futását.

A `run` parancssal használhatunk parancssori paramétereket és/vagy átírányítást (< vagy >). Ha újra kiadjuk a `run` parancsot, akkor az előzőleg kiadott összes paraméter (és átírás) továbbra is érvényben marad. Az aktuális paramétereket a "show args" utasítással tekinthetjük meg.

Töréspont beszúrása

Töréspontot (*breakpoint*) a `break` parancssal tudunk beszúrni. Ennek hatására a program futása a kívánt ponton „megtörök”, azaz felfüggesztődik. Ilyenkor meg lehet nézni a különböző változók tartalmát; meg lehet vizsgálni, mi is a helyzet az adott futási pillanatban.

Töréspontot bárhol elhelyezhetünk a programban: ehhez csak meg kell adni a kívánt sor számát. Ha a sorszámot fájlnév nélkül adjuk meg, akkor az aktuális fájl adott sorát fogja bejelölni a gdb. Ha megadunk egy függvénynevet, akkor az adott függvény első (végre-hajtható) során fog megállni a program.

```
(gdb) break 12          Töréspont a 12 . sorra
Breakpoint 1 at 0x1da4: file mod1.c, line 12.
(gdb) break main        Töréspont a main kezdetére
Breakpoint 2 at 0x1d6c: file mod1.c, line 3.
(gdb) break mod2.c:foo   Töréspont a mod2.c fájl foo függvényére
Breakpoint 3 at 0x1dd8: file mod2.c, line 4.
```

Amikor a program vezérlése egy töréspontba ütközik, a gdb felfüggeszti a program futását, kijelzi a töréspont helyét, és lehetőséget ad a felhasználónak a különféle műveletekre. Megjeleníthetőek változók, töréspontok adhatóak meg (vagy vonhatóak vissza), vagy a continue (rövidítve c) utasítással folytattható a program futása.

Soronkénti továbblépés

A program vérehajtását a step (*továbblépés*; rövidítve s) utasítással is kézbe vehetjük. Ennek hatására a programkód következő sora fut le. Ha számot is megadunk paraméterként, akkor annyi sor hajtódik végre. Elképzelhető, hogy egy sorban több utasítás is van – azonban a gdb a sorokat tekinti, nem az utasításokat. Ha a sor utolsó utasítása áthajlik a következő sor(ok)ba, akkor a step hatására az(ok) is lefut(nak). A step meghívható bárhol, ahol a continue utasítás használható (hiba vagy töréspont után).

Ha az aktuális utasítás egy függvényhívás, akkor a step hatására a gdb belép az adott függvénybe. Ha step helyett a next utasítást használjuk, akkor a gdb csak vérehajtja a függvényt, de nem lép bele.

Próbáljuk ki a tanultakat a 18.5 Listán. A programnak nincs más hasznos célja.

18.5 Lista • A gdb használata

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

struct date {
    int month;
    int day;
    int year;
};

struct date foo (struct date x)
{
    ++x.day;
```

```
    return x;
}

int main (void)
{
    struct date today = {10, 11, 2004};
    int         array[5] = {1, 2, 3, 4, 5};
    struct date *newdate, foo ();
    char         *string = "test string";
    int          i = 3;

    newdate = (struct date *) malloc (sizeof (struct date));
    newdate->month = 11;
    newdate->day = 15;
    newdate->year = 2004;

    today = foo (today);

    free (newdate);

    return 0;
}
```

A 18.5 Lista programjának gdb-beli futtatása az olvasó számítógépen másként is festhet, a rendszertől és a gdb verziószámától függően.

18.5 Lista Egy gdb munkamenet

```
$ gcc -g p18-5.c
$ gdb a.out
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec 4 15:41:30 GMT
2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "powerpc-apple-darwin".
Reading symbols for shared libraries .. done
(gdb) list main
14
15     return x;
16 }
17
18 int main (void)
19 {
```



```
$10 = (struct date *) 0x100140
(gdb) info locals          Valamennyi helyi változó értékének megjelenítés
today = {
    month = 10,
    day = 11,
    year = 2004
}
array = {1, 2, 3, 4, 5}
newdate = (struct date *) 0x100140
string = 0x1fd4 "test string"
i=3
(gdb) break foo            Töréspont a foo elejére
Breakpoint 2 at 0x1c98: file p18-5.c, line 13.
(gdb) continue             Folytassuk a futtatást
Continuing.
Breakpoint 2, foo (x={month = 10, day = 11, year = 2004})
at p18-5.c:13
13          ++x.day; 0x8e in foo:25: {
(gdb) print today          A today (ma) kijelzése
No symbol "today" in current context
(gdb) print main::today     A main-beli today kijelzése
$11 = {
    month = 10,
    day = 11,
    year = 2004
}
(gdb) step
15          return x;
(gdb) print x.day
$12 = 12
(gdb) continue
Continuing.
Program exited normally.
(gdb)
```

Figyeljük meg a gdb sajátos viselkedését: az egyesével való lépegetés esetén, vagy amikor a vezérlés elér egy töréspontot, akkor a gdb nem az utoljára végrehajtott sort, hanem a következő lépésben végrehajtandó sort jeleníti meg. Az array tömb még nem volt megadva az első kiíratásakor, pedig az előző step után már megjelent az inicializáló sor. A következő lépés után kapott csak kezdőértéket a tömb. Azt is jó tudni, hogy azok a definíciók, amelyek kezdőértéket is adnak a változóknak, végrehajtható utasításnak minősülnek (helyesen, ugyanis a fordítóprogram ténylegesen létrehoz ilyenkor végrehajtandó utasításokat).

Töréspontok felsorolása és törlése

Ha beállítottunk egy töréspontot, akkor az érvényben marad, amíg csak ki nem lépünk az adott gdb munkamenetből, vagy nem törljük szándékosan. Az érvényben lévő töréspontokat az info break utasítással tudjuk felsorolni:

```
(gdb) info break
Num Type Disp Enb Address What
1 breakpoint keep y 0x00001c9c in main at p18-5.c:20
2 breakpoint keep y 0x00001c4c in foo at p18-5.c:13
```

Egy adott sorhoz tartozó töréspontot a clear parancs után megadott sorszámmal tudunk törlni. A törléskor lehetőség van függvénynév megadására is – ekkor a függvény első végrehajtható sorához tartozó töréspont törlődik.

```
(gdb) clear 20 A 20 . sorhoz tartozó töréspont törlése
Deleted breakpoint 1
(gdb) info break
Num Type Disp Enb Address What
2 breakpoint keep y 0x00001c4c in foo at p18-5.c:13
(gdb) clear foo A foo belépési pontjára elhelyezett töréspont törlése
Deleted breakpoint 2
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

A hívási verem (stack trace) kijelzése

Szükség lehet rá, hogy megtudjuk, hogy a vezérlés egy adott pillanatban a függvények végrehajtási hierarchiájának mely pontján áll. Ez a core fájlok vizsgálatakor különösen hasznos. A hívási vermet a backtrace (bt) parancssal tudjuk lekérdezni. Az alábbiakban a 18.5 Listát fogjuk nyomon követni:

```
(gdb) break foo
Breakpoint 1 at 0x1c4c: file p18-5.c, line 13.
(gdb) run
Starting program: /Users/stevekochan/MySrc/c/a.out
Reading symbols for shared libraries . done

Breakpoint 1, foo (x={month = 10, day = 11, year = 2004})
at p18-5.c:13
13          ++x.day;
(gdb) bt           A hívási verem kijelzése
#0 foo (x={month = 10, day = 11, year = 2004}) at p18-5.c:13
#1 0x00001d48 in main () at p18-5.c:31
(gdb)
```

Amikor a vezérlés eléri a `foo` belépési pontjára elhelyezett töréspontot, meghívjuk a `bt` utasítást. A kimenetből látszanak az egymást meghívó függvények: a `main` hívta meg a `foo`-t. A függvények paraméterei is leolvashatóak. Vannak hasznos parancsok, melyeket most nem részletezünk (`up`, `down`, `frame` és `info args`) – ezekkel keresztül-kasul bejárhatjuk a hívási vermet, és részletesen megvizsgálhatjuk az egyes paraméter-átadásokat és lokális változókat.

Függvényhívások, értékadás tömböknek és adatszerkezeteknek

A `gdb` kifejezésekben használhatunk függvényhívásokat is:

```
(gdb) print foo(*newdate)
mutatott paraméterrel
$13 = {
    month = 11,
    day = 16,
    year = 2004
}
(gdb)
```

A `foo` meghívása a `newdate` által

Itt a 18.5 Lista `foo` függvényét hívtuk meg.

Tömböknek és adatszerkezeteknek úgy adhatunk értékeket, hogy felsoroljuk őket kapcsos zárójelben, vesszővel elválasztva:

```
(gdb) print array
$14 = {1, 2, 3, 4, 5}
(gdb) set var array = {100, 200}
(gdb) print array
$15 = {100, 200, 0, 0}          A definiálatlan értékek nullázódtak
(gdb) print today
$16 = {
    month = 10,
    day = 11,
    year = 2004
}
(gdb) set var today={8, 8, 2004}
(gdb) print today
$17 = {
    month = 8,
    day = 8,
    year = 2004
}
(gdb)
```

A gdb parancsainak súgója

A különféle parancsokról (vagy parancstípusokról, „osztályokról”) való (többnyire angol nyelvű) információszerzéshez használhatjuk a gdb beépített súgóját, a help utasítást.

Paraméter nélkül a help kiírja a parancsok típusait:

```
(gdb) help
```

List of classes of commands:

```
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the
program
user-defined -- User-defined commands
```

Type "help" followed by a class name for a list of commands in that class.

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.

Ha részleteket szeretnénk megtudni egy-egy parancstípusról, akkor a keresett osztály nevét meg kell adni a help paramétereként:

```
(gdb) help breakpoints
```

Making program stop at certain points.

List of commands:

```
awatch -- Set a watchpoint for an expression
break -- Set breakpoint at specified line or function
catch -- Set catchpoints to catch events
clear -- Clear breakpoint at specified line or function
commands -- Set commands to be executed when a breakpoint is hit
condition -- Specify breakpoint number N to break only if COND is
true
delete -- Delete some breakpoints or auto-display expressions
disable -- Disable some breakpoints
enable -- Enable some breakpoints
future-break -- Set breakpoint at expression
```

```
hbreak -- Set a hardware assisted breakpoint
ignore -- Set ignore-count of breakpoint number N to COUNT
rbreak -- Set a breakpoint for all functions matching REGEXP
rwatch -- Set a read watchpoint for an expression
save-breakpoints -- Save current breakpoint definitions as a script
set exception-catch-type-regexp -
    Set a regexp to match against the exception type of a
caughtobject
set exception-throw-type-regexp -
    Set a regexp to match against the exception type of a
thrownobject
show exception-catch-type-regexp -
    Show a regexp to match against the exception type of a
caughtobject
show exception-throw-type-regexp -
    Show a regexp to match against the exception type of a
thrownobject
tbreak -- Set a temporary breakpoint
tcatch -- Set temporary catchpoints to catch events
thbreak -- Set a temporary hardware assisted breakpoint
watch -- Set a watchpoint for an expression
```

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)

Az itt felsorolt parancsokról is kérhetünk eligazítást:

```
(gdb) help break
Set breakpoint at specified line or function.
Argument may be line number, function name, or "*" and an address.
If line number is specified, break at start of code for that line.
If function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
With no arg, uses current execution address of selected stack frame.
This is useful for breaking on return to a stack frame.
```

Multiple breakpoints at one place are permitted, and useful if
conditional.

break ... if <cond> sets condition <cond> on the breakpoint as it is
created.

Do "help breakpoints" for info on other commands dealing with
breakpoints.

(gdb)

Amint láthatjuk, bőséges információ áll rendelkezésünkre a gdb parancsait illetően. Érdekes rágérdezni egy-egy témaéra, ha nem vagyunk tisztában a lehetőségekkel.

Kiegészítések

A gdb számos egyéb lehetőséggel is bír, amelyekről hely hiányában nem beszélünk. Fel-sorolásként had említsük meg néhányat.

- Be lehet állítani ideiglenes töréspontokat, melyek törlődnek, mihelyt az első leállást előidézték
- A töréspontok ki/bekapcsolhatóak anélkül, hogy törlni kellene őket.
- A memória adott része a kívánt formátumban menthető (*dump*)
- Beállítható egy figyelő (*watchpoint*), amely lehetővé teszi, hogy a program akkor álljon meg, amikor egy bizonyos kifejezés (például egy változó értéke) megváltozik
- Megadható, hogy milyen változók értékei jelenjenek meg a program megállásakor
- Név szerint is létrehozhatunk saját „kényelmi változókat”.

A 18.1 Táblázat bemutatja a fejezetünkben tárgyalta gdb parancsokat. Félkövéren szedve látszanak a parancsok rövidítési lehetőségei.

18.1. Táblázat • A leggyakoribb gdb parancsok

A forráskóddal kapcsolatos információk

Parancs	Jelentése
list [n] ²	Kijelzi az <i>n</i> . sor (vagy <i>n</i> hiányában az aktuális sor) környezetében lévő 10 sort
list <i>m,n</i>	Kijelzi a sorokat <i>m</i> -től <i>n</i> -ig.
list +[n]	Kijelzi az <i>n</i> . sorhoz (vagy <i>n</i> hiányában az aktuális sorhoz) képest tíz sorral később lévő 10 sornyi környezetet.
list -[n]	Kijelzi az <i>n</i> . sorhoz (vagy <i>n</i> hiányában az aktuális sorhoz) képest tíz sorral korábban lévő 10 sornyi környezetet.
list fuggv	Kiírja a <i>fuggv</i> függvényt
listsize <i>n</i>	A kijelzendő sorok számát <i>n</i> -re állítja át.
info source	Kiírja az aktuális forráskód fájlnevét

² minden parancs, amely elfogad számot vagy függvénynevet paraméterként, használható állománynév megadásával együtt is. Ilyenkor az fájlnév után tett kettősponttal jelezhetjük a kívánt sor(tartomány) vagy függvény helyét, például: list main.c:1,10 vagy break main.c:12

Változók és kifejezések

Parancs	Jelentése
<code>print /fmt expr</code>	Az <code>fmt</code> formátumnak megfelelően írja ki az <code>expr</code> kifejezést. Az <code>fmt</code> lehet <code>d</code> (decimális), <code>u</code> (előjel nélküli), <code>o</code> (oktális), <code>x</code> (hexadecimális), <code>c</code> (karakter), <code>f</code> (lebegőpontos), <code>t</code> (bináris) vagy <code>a</code> (cím)
<code>info locals</code> két írja ki	Az aktuális függvény lokális változóinak értékeit írja ki
<code>set var var=expr</code>	A <code>var</code> változónak átadja <code>expr</code> értékét.

Töréspontok

Parancs	Jelentése
<code>break n</code>	Töréspontot állít be az <code>n</code> . sorra
<code>break fuggv</code> pontjára	Töréspontot állít be a <code>fuggv</code> függvény belépési pontjára
<code>info break</code>	Megmutatja a töréspontokat
<code>clear [n]</code>	Törli a töréspontot az <code>n</code> . sorból (vagy a következő sorból, ha <code>n</code> nincs megadva)
<code>clear fuggv</code>	Törli a töréspontot a <code>fuggv</code> függvény belépési pontjáról

A futtatás vezérlése

Parancs	Jelentése
<code>run [par] [<file> [>file]]</code>	Elindítja előlről a program végrehajtását
<code>continue</code>	Folytatja a program végrehajtását
<code>step [n]</code> <code>ző n sort)</code>	Végrehajtja a következő sort (vagy a következő <code>n</code> sort)
<code>next [n]</code>	Végrehajtja a következő sort (vagy a következő <code>n</code> sort), de nem lép be a meghívott függvényekbe
<code>quit</code>	Kilép a <code>gdb</code> -ből

Súgó

Parancs	Jelentése
<code>help [parancs]</code>	Kiírja a kért parancs súgóját vagy paraméter nélkül felsorolja az elérhető osztályokat
<code>help [osztaly]</code>	Kiírja a kért osztály parancsait

19

Objektumközpontú programozás

Az objektumközpontú vagy objektum-orientált programozás (OOP) igen népszerű. Mivel számos olyan OOP nyelv van, amit a C-re épül (a C++, a C#, a Java és az Objective-C), érdemes röviden átgondolni, miről is szólnak az objektumközpontú nyelvek, mennyiben tér el alapkoncepciójuk a C nyelv filozófiájától. A fejezet az OOP elvének áttekintésével kezdődik, majd egy kisebb programot írunk meg a fent említett nyelvek közül három nyelven (amelyekben szerepel a „C” betű). Nem célunk megtanítani az olvasót az objektumközpontú programozásra, sőt még csak összefoglalást sem szeretnénk adni, hanem csak egy rövid ízelítőt nyújtunk e nyelvek stílusából.

Végül is mi az az objektum?

Az objektum egy *dolog*. Úgy kell elképzelni az objektum-orientált programozást, hogy vannak a világban dolgok, és mi szeretnénk valamit csinálni ezekkel a dolgokkal. Ez a megközelítés ellentétes a C nyelv (pontosabban a procedurális nyelvek) világával, ahol először gondoljuk ki azt, hogy mit kell tennünk (és ehhez esetleg függvényeket is írunk), majd ezután hozzuk létre az „objektumokat”.

Nézzünk egy példát a minden nap életből: tegyük fel, hogy van egy autónk. Ez nyilvánvalóan egy objektum – a saját autónk. Nem pusztán egy általános autónk van, hanem egy konkrét autó valamelyik gyáról – talán Detroitból vagy Japánból vagy bárhonnan más-honnan. Az autónknak van rendszáma is, amely egyértelműen azonosítja.

Objektum-orientált kifejezésmódjal élve a *mi autónk egy autópéldány*. A szóhasználatot továbbfűzve: az *autó* a neve annak az *osztálynak*, amelyből ez a példány származik. minden alkalommal, amikor a gyáról kigördül egy autó, az autók osztálya példányosul. Az autópéldányokat *objektumként* nevezhetjük meg.

Az autónk lehet ezüst, fekete belsőjű, lehet lehúzható vagy merev teteje stb. Ráadásul bizonyos dolgokat meg is lehet tenni az autónkkal. Például lehet vezetni, meg lehet tankolni, sőt (remélhetőleg) időnként le is szoktuk mosni, el szoktuk vinni szervizbe stb. Mind-ez a 19.1 Táblázatban van összefoglalva.

19.1 Táblázat • Műveletek objektumokkal

Objektum	Mit lehet vele tenni
Az autónk	Vezetni
	Megtankolni
	Lemosni
	Szervizbe vinni

A 19.1 Táblázat műveleteit nemcsak a saját autónkkal lehet megtenni, hanem más autójával is. Nővérünk is vezetheti a saját autóját, lemoshatja, megtankolhatja stb.

Példányok és metódusok

Egy osztály egyedi megvalósulását hívjuk példánynak, a végrehajtható műveleteket pedig metódusoknak (tagfüggvényeknek). A műveletek vonatkozhatnak példányokra vagy osztályokra is. Az autómosás (és a 19.1 Táblázat valamennyi művelete) egy autópéldányra vonatkozhat, még annak megállapítása, hogy a gyártó hány autót gyártott, ez már osztály szintű művelet.

C++-ban a következő jelölésmóddal lehet meghívni egy példányhoz tartozó metódust:

```
Instance.method();
```

A C# metódusa is hasonló jelölésmóddal történik:

```
Instance.method();
```

Egy Objective-C üzenethívásnak (*message call*) a következő a formátuma:

```
[Instance method]
```

Térjünk vissza az előző táblázathoz, és fogalmazzunk meg üzenethívásokat az itt vázolt szintaxisossal. Tegyük fel, hogy a yourCar a Car osztály egy objektuma. A 19.2 Táblázat bemutatja, hogy hogyan néznének ki az üzenethívások a fenti három OOP nyelven.

19.2 Táblázat • Üzenethívások különféle OOP nyelveken

C++	C#	Objective-C	Művelet
yourCar.drive()	yourCar.drive()	[yourCar drive]	Vezetni az autónkat
yourCar.getGas()	yourCar.getGas()	[yourCar getGas]	Megtankolni az autónkat
yourCar.wash()	yourCar.wash()	[yourCar wash]	Lemosni az autónkat
yourCar.service()	yourCar.service()	[yourCar service]	Szervízre vinni az autónkat

Ha nővérünknek is van egy autója (legyen a neve `suesCar`), akkor az ő autójával is hasonló műveletek végezhetőek:

C++	C#	Objective-C
<code>suesCar.drive()</code>	<code>suesCar.drive()</code>	[<code>suesCar drive</code>]

Ez az objektumközpontú programozás egyik kulcsfogalma: ugyanazok a metódusok különféle objektumokra alkalmazhatóak.

Egy másik alapfogalom a többalakúság (polimorfizmus). Eszerint ugyanazt az üzenetet lehet elküldeni különböző osztályok példányainak. Ha van egy `Boat` (*hajó*) osztályunk is, és annak egy példánya a birtokomban van (`myBoat`), akkor a többalakúság jegyében érvényesek a következő C++ kifejezések:

```
myBoat.service()
myBoat.wash()
```

Az a lényeg, hogy a `Boat` osztály számára is megírhatjuk azt a metódust, amely ismeri, hogy hogyan kell egy hajót szervizbe vinni. Ez valószínűleg gyökeresen eltér az autóra vonatkozó azonos nevű művelettől, de ez nem jelent gondot – ilyen a többalakúság.

Fontos különbség az OOP nyelvek és a C nyelv között, hogy az objektumközpontú nyelvekben objektumokkal dolgozunk, mint például az autók és a hajók, míg C-ben leginkább függvényekkel (eljárásokkal). A (C-hez hasonló) ún. procedurális nyelvekben először a `service` függvényt írnánk meg, és ezen belül különböző kódrészletek kezelnék a különféle járműveket: kocsikat, hajókat vagy kerékpárokat. Ha később egy új járműtípus szeretnénk felvenni, akkor mindeneket a függvényeket módosítani kellene, melyek járműveket kezelnek. Az OOP nyelvekben csak egy definiálni kell egy új osztályt az új jármű-

mű számára, és megírni a megfelelő metódusokat ezen osztály számára. Nem kell törödni a többi járműosztállyal – azok függetlenek a frissen létrehozottól, így nem kell (sőt, esetleg megfelelő jogosultság hiány nem is lehet) módosítani őket.

Az OOP programokban használt osztályaink valószínűleg nem autókat vagy hajókat fognak megvalósítani. Inkább ablakokat, téglalapot, vágólapokat stb. Ilyesfélé üzeneteket szoktak küldeni a C#-hoz hasonló nyelvekben:

<code>myWindow.erase()</code>	Töröl egy ablakot
<code>myRect.getArea()</code>	Kiszámolja egy téglalap területét
<code>userText.spellCheck()</code>	Helyesírásilag ellenőrzi a szöveget
<code>deskCalculator.setAccumulator(0.0)</code>	Törli a számológép memóriáját
<code>favoritePlaylist.showSongs()</code>	Lejátssza a kedvenc dalokat

Törtek kezelése C nyelven

Tegyük fel, hogy szükségünk van egy olyan programra, amely törteket kezel. Szeretnénk őket összeadni, kivonni és esetleg szorozni. Ehhez létrehozhatunk egy törteket reprezentáló adatszerkezetet, majd megírhatjuk a szükséges függvényeket.

A 19.1 Lista egy egyszerű példaprogram, amelyben csak annyi történik, hogy definiáljuk a törtek tárolási módját. Létrehozunk egy törtet számlálójával és nevezőjével, majd megjelenítjük.

19.1 Lista • Törtek kezelése C-ben

```
// Egy egyszerű program a törtek kezeléséhez
#include <stdio.h>

typedef struct {
    int numerator;
    int denominator;
} Fraction;

int main (void)
{
    Fraction myFract;

    myFract.numerator = 1;
    myFract.denominator = 3;

    printf ("A tört: %i/%i\n", myFract.numerator,
myFract.denominator);

    return 0;
}
```

19.1 Lista • Kimenet

A tört: 1/3

A következő három szakaszban bemutatjuk, hogy milyen módon lehet törteket kezelő programot írni Objective-C, C++ és C# nyelven. Az OOP nyelvre vonatkozó bevezetés a 19.2 Lista után található, így az adott sorrendben érdemes elolvasni az egyes bekezdéseket.

Objective-C osztály definiálása törtek kezeléséhez

Az Objective-C nyelvet Brad Cox találta föl az 1980-as évek elején. A NeXT Software által 1988-ban szabadalmaztatott nyelv a SmallTalk-80-ra épült. Amikor az Apple 1988-ban megvásárolta a NeXT-et, a Mac OS X operációs rendszer alapjaként a NEXTSTEP-et használta. A Max OS X-en ma futó legtöbb alkalmazás Objective-C nyelven íródott.

19.2 Lista • Törtek kezelése Objective-C nyelven

```
// Program a törtek kezeléséhez - Objective-C változat
```

```
#import <stdio.h>
#import <objc/Object.h>

//----- @interface section -----

@interface Fraction: Object
{
    int numerator;
    int denominator;
}
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(void) print;

@end

//----- @implementation section -----

@implementation Fraction;

// hozzáférő (getter) metódusok

-(int) numerator
{
    return numerator;
}

-(int) denominator
{
```

```

        return denominator;
    }

    // beállító (setter) metódusok

    -(void) setNumerator: (int) num
    {
        numerator = num;
    }

    -(void) setDenominator: (int) denom
    {
        denominator = denom;
    }

    // Egyéb

    -(void) print
    {
        printf ("A tört értéke %i/%i\n", numerator, denominator);
    }
@end

//----- Maga a program -----

int main (void)
{
    Fraction      *myFract;
    myFract = [Fraction new];
    [myFract setNumerator: 1];
    [myFract setDenominator: 3];
    printf ("A számláló %i, a nevező pedig %i\n",
            [myFract numerator], [myFract denominator]);
    [myFract print]; // a megfelelő metódussal is kiírjuk a törtet
    [myFract free];
    return 0;
}

```

19.2 Lista • Kimenet

A számláló 1, a nevező pedig 3

A tört értéke 1/3

Ahogy a 19.2 Lista megjegyzéseiből is látszik, a program három részre oszlik: az @interface (*felhasználói felület*), az @implementation (*megvalósítás*) és a program szakaszra. Ezek általában külön állományokban helyezkednek el. Az @interface szakasz legtöbbször egy fejlécállományba kerül, amit minden olyan program beemelhet, amelyik az adott osztálytalálkozás szeretne dolgozni. Ez tudatja a fordítóprogrammal, hogy milyen változók és metodusok vannak az adott osztályban.

Az `@implementation` szakasz tartalmazza a metódusok megvalósításának a kódját. Végül pedig a program szakasz tartalmazza az adott feladatot megoldó program kódját.

Az osztályok a szülőosztálytól örökölnek változókat és metódusokat; a létrehozandó új `Fraction` (`tört`) osztály az `Object` osztály leszármazottja.

Az `@interface` szakaszban látható két deklaráció, amely egy-egy egész számot (számlálót és nevezőt) társít a `Fraction` objektumhoz:

```
int numerator;           // számláló
int denominator;        // nevező
```

Az ebben a szakaszban deklarált adattagokat példányváltozóknak (*instance variables*) hívjuk. Amikor létrehoznunk egy új objektumot, létrejön egy új és egyedi példányváltozókészlet. Ha tehát két törtünk vagy, `fracA` és `fracB`, akkor mindegyiknek lesz saját számlálója és nevezője.

Meg kell adnunk a törteket használó metódusokat is. Be kell tudnunk állítani egy tört értékét. Mivel nincs közvetlen hozzáférésünk a tört belső ábrázolásához (azaz a példányváltozókhöz), meg kell írnunk a megfelelő metódusokat a számláló és a nevező beállításához. Ezeket beállító (*setter*) metódusoknak hívjuk. A törtek értékének (a példányváltozóknak) a kinyeréséhez is külön hozzáférő (*getter*) metódusok kellenek.¹

Az objektum példányváltozói tehát rejte maradnak az objektum felhasználói előtt. Ezt hívják beágyazásnak (*data encapsulation*); ez is fontos vonása az OOP-nek. Ez teszi lehetővé, hogy kiterjesszünk vagy módosítsunk egy-egy osztályt, melynek minden adat- (azaz példányváltozó-)elérési kódja metódusokban áll fenn. A beágyazás révén kellemes szigetelőréteg kerül a programozó és az osztály-fejlesztő közé.

Így néz ki egy beállító (*setter*) metódus:

```
- (int) numerator;
```

A bevezető mínusz jel (-) mutatja, hogy példánymetódusról van szó. Ehelyett állhatna plusz jel (+) is, amely az osztálymetódusok jele. Az osztálymetódus magán az osztályon végez el valamilyen műveletet, például létrehoz egy új példányt az osztályból. Mintha egy új autó jönne létre a gyárban – ebben a hasonlatban az autó járassa az osztály szerepét; ebből szeretnénk egy új példányt. Ennek megvalósítása osztálymetódusnak tekinthető.

¹ Megoldható, hogy közvetlenül hozzáférjünk a példányváltozókhöz, de ez általában elég gyenge programozási megoldásnak számít.

A példánymetódus az osztály valamely példányán végez el műveleteket; beállítja vagy ki-nyeri (esetleg kijelzi) az értékét stb. Az autós példánál maradva: az autót (létrehozása után) érdemes megtankolni. Az autó megtankolása már egy konkrét példányra vonatkozik, így ez a művelet már példánymetódus.

Az új metódusok deklarálása hasonlít a függvénydeklarációhoz. Ilyenkor arról tájékoztat-juk az Objective-C fordítót, hogy a metódusnak van-e a visszatérési értéket, és ha igen, akkor milyen? A visszatérési értéket zárójelbe tesszük a bevezető $+/-$ jel után. Az alábbi deklaráció szerint a numerator (*számláló*) egy egész értéket ad vissza:

```
- (int) numerator;
```

Az alábbi sor is ehhez hasonló, de a `setNumerator` metódus nem ad vissza semmilyen értéket, viszont vár egy egész paramétert; ezzel lehet beállítani a számláló értékét:

```
- (void) setNumerator: (int) num;
```

Ha a metódus elfogad paraméter(*eke*), akkor egy kettőspontot kell a metódus neve után ír-ni, amikor meghívjuk. A számlálót és nevezőt beállító két egyparaméteres metódust tehát `setNumerator:` és `setDenominator:` néven kell meghívni. A `numerator` és `denominator` metódus kettőspont nélküli megnevezése jelzi, hogy paraméter nélkül működnek.

A `setNumerator:` metódus átveszi a `num` nevű egész paramétert, és eltárolja a `numera-tor` példányváltozóban. Hasonlóképp tesz a `setDenominator:` metódus, csak ez a `denom` paramétert tárolja el a `denominator` példányváltozóban. A metódusoknak köz-vetlen hozzáférésük van a példányváltozókhöz.

Objective-C programunk utolsó metódusa a tört értékét kiíró `print`. Ahogy látható, nem kér paramétert és nem is ad vissza semmit. Egyszerűen csak meghívja a `printf` függ-vényt, melynek segítségével egy törtjellel elválasztva kiírja a számlálót és a nevezőt.

A `main`-ben definiáljuk a `myFract` változót:

```
Fraction *myFract;
```

Ebből a sorból az derül ki, hogy a `myFract` egy `Fraction` típusú objektum lesz, azaz a `myFract` a `Fraction` osztály egy új példányának adatait fogja tárolni. A csillag (*) arra utal, hogy a `myFract` igazából egy mutató egy olyan adatszerkezetre, amelyet `Fraction` osztály definiál.

Elkészült tehát egy olyan objektum, ami képes lesz tárolni egy törtet – azonban azt még létre kell hoznunk. A gyárral meg kell még csinálnunk az új autót. Ez a következő sor révén történik meg:

```
myFract = [Fraction new];
```

Memóriaterületet szeretnénk lefoglalni az új törtnek. A

```
[Fraction new]
```

kifejezés üzenetet küld az újonnan létrehozott Fraction osztálynak. Arra kérjük a Fraction osztályt, hogy használja a new (új) metódust. De hiszen ilyen metódust még nem adtunk meg! Honnan létezik ez tehát? A válasz: a szülőobjektumtól öröklődött.

Készen állunk arra, hogy beállítsuk törtünk értékét; ezt a következő sorok oldják meg:

```
[myFract setNumerator: 1];
[myFract setDenominator: 3];
```

Az első utasítás elküldi a myFract számára a setNumerator: üzenetet. Az átadott paraméter értéke 1. Ezután a setNumerator-hoz kerül át a vezérlés, amelyet a Fraction osztályhoz definiáltunk. Az Objective-C futtató rendszer tudja, hogy a myFract egy Fraction osztályból származó objektum, így elérhetők a metódusai.

A setNumerator: metódusban egyetlen programsor található, amely az átvett paramétert eltárolja a numerator példányváltozóban. Vagyis sikerült a myFract számlálóját 1-re beállítani.

A setDenominator: metódus meghívása hasonlóképpen történik.

A tört beállítása után a 19.2 Lista programja meghívja a két hozzáférő metódust, a numerator-t és a denominator-t, hogy kinyerje a myFract példányváltozóit. Az eredményt egy szöveges üzenet formájában jelzi ki a printf utasítás .

A program ezután meghívja a print metódust, amely kiírja annak a törtnek az értékét, amely őt tagfüggvényként tartalmazza. Láthattuk a programban, hogy a nevező és számláló hozzáférő függvényekkel is elérhető, azonban szemléltetési céllal ezt a print metódust is beillesztettük a Fraction osztály definíciójába.

A program utolsó utasítása felszabadítja a Fraction objektum által lefoglalt memóriát:

```
[myFract free];
```

C++ osztály definiálása törtek kezeléséhez

A 19.3 Lista bemutat egy programot, melyben a Fraction osztályt C++ nyelven valósítjuk meg. A C++ nyelv, mely a Bell Laboratories egyik fejlesztőjének, Bjarne Stroustrupnak a nevéhez fűződik, rendkívül népszerű programozási nyelvvé vált. Tudomásom szerint ez volt az első C-re épülő objektum-orientált nyelv.

19.3 Lista • Törtek kezelése C++ nyelven

```
int main (void)
{
    Fraction myFract;

    myFract.setNumerator (1);
    myFract.setDenominator (3);

    myFract.print ();

    return 0;
}
```

19.3 Lista • Kimenet

A tört értéke 1/3

A numerator és denominator C++ adattagok (példányváltozók) private címkével kerültek bejegyzésre, ami a beágyazást erőteljes szintjére utal: semmiképpen nem érhetőek el más osztályokból.

A setNumerator metódus deklarációja a következő:

```
void Fraction::setNumerator (int num)
```

A metódus neve előtt ott áll a Fraction:: előtag, azonosítva azt az osztályt, amelyhez tartozik. A Fraction új példányát úgy hozhatjuk létre, mint egy normál C változót, a main eljárásban:

```
Fraction myFract;
```

A tört nevezőjét és számlálóját 1-re és 3-ra állítjuk be az alábbi utasításokkal:

```
myFract.setNumerator (1);
myFract.setDenominator (3);
```

A tört értékét ezután tört saját tagfüggvénye, a print jelzi ki.

A 19.3 Lista legszokatlanabbul kinéző utasítása minden bizonnal a következő:

```
std::cout << "A tört értéke " << numerator << '/'
                << denominator << '\n';
```

A cout jelzi a szabványos kimenetet – ez felel meg a C-beli std::cout-nak. A << a „kimenet átirányító” operátor, amely egyszerű módon lehetővé teszi a kimenet létrehozását. Talán emlékszik az olvasó, hogy ugyanezt az operátort a C nyelvben a bitenkénti eltolás operá-

torának hívtuk. Tanúi lehetünk a C++ nyelv egy igen jellemző vonásának, amit *operátor túlterhelésnek* (*operator overloading*) hívunk: egy-egy osztályhoz is társíthatunk operátorokat. Itt a bitenkénti eltolás operátorának „túlterhelését” láthatjuk. Amikor ilyen szöveg-környezetben használjuk (azaz egy adatfolyam a bal oldali operandusa), akkor az operátor meghívja a formázott kiírás függvényét, melynek segítségével a kimenetre kerül a megadott karakterlánc. Ilyenkor természetesen nem történik bitenkénti eltolás.

Az operátor túlterhelés másik példájaként gondoljunk arra, milyen jó lenne, ha a + jelet törtek összeadására is használhatnánk! Igen látványos lenne, ha így adhatnánk össze a Fraction osztály két példányát:

```
myFract + myFract2
```

Ehhez meg kell írnunk egy megfelelő Fraction-tagfüggvényt, amely a + operátor hatására meghívódna.

A << operátort követő összes kifejezés kiértékelődik és a szabványos kimenetre kerül. Esetünkben az első karakterlánc "A tört értéke", majd a számláló következik, egy törtonal, végül a nevező és egy újsor karakter.

A C++ nyelv igen gazdag. Érdemes megnézni az E függeléket („Források”), melyben különféle tanfolyam-javaslatokat talál az olvasó.

Megjegyzés: fenti C++ példaprogramunkban létrehoztuk ugyan a Fraction osztály Numerator () és Denominator () tagfüggvényét, de nem használtuk.

C# osztály definiálása törtek kezeléséhez

Fejezetünk utolsó példájaként nézzünk egy olyan törtekkezelő programot, melyet a Microsoft, Inc. által kifejlesztett C# nyelven írnunk meg. A C# az egyik legújabb OOP nyelv, és igen nagy népszerűségnek örvend. Ez lett a .NET keretrendszerrel fejlesztett alkalmazások programozási nyelve.

19.4 Lista • Törtek kezelése C# nyelven

```
// Program a törtek kezeléséhez - C# változat
using System;

class Fraction
{
    private int numerator;
    private int denominator;

    public int Numerator
    {
```

```
get
{
    return numerator;
}

set
{
    numerator = value;
}
}

public int Denominator
{
    get
    {
        return denominator;
    }

    set
    {
        denominator = value;
    }
}

public void print ()
{
    Console.WriteLine("A tört értéke {0}/{1}",
                      numerator, denominator);
}

class example
{
    public static void Main()
    {
        Fraction myFract = new Fraction();

        myFract.Numerator = 1;
        myFract.Denominator = 3;

        myFract.print ();
    }
}
```

19.4 Lista • Kimenet

A tört értéke 1/3

Mint látható, a C# nyelv egy kissé másként fest, mint a másik két OOP nyelv, de az olvasó valószínűleg kitalálja, hogy itt mi történik. A Fraction osztály definíciója úgy kezdődik, hogy a két példányváltozt, a numerator-t és a denominator-t private-ként deklaráljuk. A Numerator és Denominator metódusok tartalmazzák a saját hozzáférő és beállító metódusaikat, melyeket a metódusok *tulajdonságának* (*properties*) hívunk. Vegyük még egyszer szemügyre a Numerator-t:

```
public int Numerator
{
    get
    {
        return numerator;
    }

    set
    {
        numerator = value;
    }
}
```

A get (*hozzáférő*) kódrészlet akkor hajtódiik végre, amikor egy kifejezésben előkerül a számláló neve:

```
num = myFract.Numerator;
```

A set (*beállító*) kódrészlet pedig akkor, amikor a metódushoz érték-hozzárendelés történik:

```
myFract.Numerator = 1;
```

A metódus meghívásakor a hozzárendelendő érték eltárolódik a value változóban. Figyeljük meg, hogy a hozzáférő és beállító függvények meghívásakor nincs szükség zárójelek használatára.

Természetesen úgy is lehet definiálni metódusokat, hogy paraméterként kerüljenek átadásra az értékek, vagy akár több paramétert is kaphatnak a beállító függvények. Elképzelhető olyan C# metódushívás is, melyben egy lépésben állítjuk be a tört értékét 2/5-re:

```
myFract.setNumAndDen (2, 5)
```

Térjünk vissza a 19.4 Listához.

A

```
Fraction myFract = new Fraction();
```

utasítás hozza létre a Fraction osztály egy új példányát, és egyből át is adja a Fraction típusú myFract változónak. Ezután a myFract megkapja az 1/3 értéket a megfelelő beállító függvények révén.

Ezután meghívjuk a myFract print tagfüggvényét, hogy jelezze ki a tört értékét.

A print metódusban láthatjuk a Console osztály WriteLine tagfüggvényét – ez végzi el a tényleges megjelenítést. A printf % jelölésmódjához hasonlóan a {0} jelzi az első behelyettesítendő paraméter-értéket, {1} a másodikat stb. A printf-fel ellentétben azonban itt nem kell foglalkoznunk a helyes típus megadásával.

A C++ példához hasonlóan a Fraction osztály hozzáférő metódusait itt sem használtuk; pusztán szemléltetésként kerültek be a programba.

Itt ér véget rövid kitérőnk, melyet az objektum-orientált programozás világában tettünk. Fejezetünkben remélhetőleg képet formálhatott az olvasó az OOP nyelvekről, illetve arról, hogy mennyiben jelentenek más utat, mint amit például a C nyelv filozófiája sugall. Látunk egy-egy programot a három kiszemelt OOP nyelven; mindegyikben törtszámokat ábrázoló objektumokkal dolgoztunk. Ha komolyan szeretne az olvasó törtekkel bánni, akkor érdemes kiegészíteni a fent vázolt osztálydefiníciókat az összeadás, kivonás, szorzás, osztás, reciprok-képzés és egyszerűsítés műveleteivel. Ez nem jelenthet már komoly gondot az olvasó számára.

További tanulnivalókat (különösen is az OOP nyelvekről) az E függelékben találhatunk.



A C nyelv összefoglalása

Ez a függelék kézikönyv jelleggel foglalja össze a C nyelvet. Nem célunk a nyelv teljes definícióját megfogalmazni – inkább kötetlen, leíró jelleggel szeretnénk vázolni a nyelv jellemzőit. A könyv fejezeteinek megtanulása után érdemes alaposan elmélyedni az itt közzétartott anyagban. Ez egyrészt elmelýíti a tanultakat, másrészt hozzásegíti az olvasót egy átfontolható kép kialakításához.

Az összefoglalás az ANSI C99 szabványon alapul (ISO/IEC 9899:1999).

1.0 Kettős karakterek és azonosítók

1.1 Kettős karakterek

Az A.1 Táblázat speciális kettős karaktereket (*digraphs*) sorol fel, melyek egyenértékűek a megadott karakterekkel.

A.1 Táblázat • Kettős karakterek

Kettős karakter	Jelentése
<:	[
:>]
<%	{
%>	}
%:	#
%:%:	##

1.2 Azonosítók

A C nyelv azonosítóiban lehetnek kis- és nagybetűk, nemzetközi karakterek (lásd az 1.2.1 szakaszt), számok és aláhúzás karakter, de az első karakter csak (latin vagy nemzetközi) betű vagy aláhúzás lehet. Egy külső változó azonosítójának első 31 karakterét garantáltan megkülönbözteti a rendszer, a belső változók és makrók azonosítójának pedig 63 szignifikáns karaktere van.

1.2.1 Nemzetközi karakterek

Egy nemzetközi karakter a következőképp hozható létre: \u után négy hexadecimális „számjegy” (azaz szám vagy a-f, A-F betű) állhat, a \U után pedig nyolc hexadecimális karakternek kell sorakoznia. Ha egy azonosító nemzetközi karakterrel kezdődik, akkor ez nem lehet számjegy karakter. Az azonosítókban használt nemzetközi karakterek számértéke nem lehet kisebb A0-nál (csak esetleg 24, 40 vagy 60 – ezek hexadecimálisan értenődők), és nem tartalmazhatnak D800 és DFFF közti értéket.

A nemzetközi karakterek használhatóak azonosítókban, karakterkonstansokban és karakterláncokban.

1.2.2 Kulcsszavak

A C fordító számára jelentéssel bíró kulcsszavak az A.2 Táblázatban láthatóak.

A.2 Táblázat • Kulcsszavak

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

2.0 Megjegyzések

Kétféleképp lehet egy programba megjegyzéseket beszúrni. Egy megjegyzés kezdődhet két perjellel: //

Az ezt követő karaktereket (az adott sorban) a fordítóprogram figyelmen kívül hagyja.

Lehet /* karakterekkel is kezdeni egy megjegyzést, és */ karakterekkel befejezni – ez a fajta megjegyzés több soron keresztül is húzódhat. A megjegyzés tetszőleges karaktereket tartalmazhat. Bárhol elhelyezhető megjegyzés a programban, ahol térköz karakter is használható. A megjegyzéseket azonban nem lehet egymásba ágyazni, azaz a /* utáni előző */ lezárja a megjegyzést, függetlenül attól, hogy esetleg még más /* jelsorozat is felbukkan a megjegyzésben.

3.0 Konstansok

3.1 Egész konstansok

Az egész konstansok számjegyek sorozatából állnak, esetleg plusz vagy mínusz előjel állhat előttük. Ha a szám első számjegye 0, akkor a fordítóprogram oktális (nyolcas számrendszerbeli) konstansként fogja tekinteni a jelsorozatot, így 0-tól 7-ig terjedhetnek a használt számjegyek. Ha az első számjegy 0, és rögtön utána egy x (vagy X) áll, akkor a jelsorozatot hexadecimális konstansként tekinti a fordítóprogram – ennek számjegyei 0-tól 9-ig és a-tól F-ig (vagy A-tól F-ig) terjedhetnek.

A tízes számrendszerbeli számok végére írt 1 vagy L hatására a konstans long int típusú-vá válik. Ha a szám nem fér bele a long int tartományba, akkor long long int-ként értelmeződik. Ha egy oktális vagy hexadecimális szám végére írunk 1-t vagy L-t, akkor long int típusuként tekinti a fordítóprogram; ha nem fér bele ebbe a tartományba, akkor long long int-ként értelmeződik. Ha még ebben sem fér el, akkor unsigned long long int-ként próbálja meg betudni a fordítóprogram az adott értéket.

Az ll vagy LL betűkkel lezárt tízes számrendszerbeli számok long long int típusúak. Ha a szám oktális vagy hexadecimális, akkor long long int-ként értelmeződik, vagy ha ebben a típusban nem fér el, akkor unsigned long long int-ként tekinti a fordítóprogram.

Az u vagy U betűkkel lezárt egész számok unsigned (előjel nélküli) típusúak. Ha a szám nem fér bele az unsigned int tartományba, akkor unsigned long int-ként értelmeződik. Ha még ebben sem fér el az érték, akkor unsigned long long int-ként próbálja meg betudni a fordítóprogram.

Az unsigned és a long utótaggal alakítható egy egész konstans unsigned long int típusúvá. Ha a szám túl nagy ehhez a típushoz, akkor unsigned long long int-ként értelmeződik.

Az unsigned és a long long utótaggal az egész konstansok unsigned long long int típusúvá tehetőek.

Ha egy utótag nélküli tízes számrendszerbeli szám nem fér el a signed int típusban, akkor long int-ként tekinti a rendszer; ha még ehhez is túl nagy, akkor long long int-ként értelmeződik.

Ha egy utótag nélküli oktális vagy hexadecimális szám nem fér el a signed int típusban, akkor unsigned int-ként tekinti a fordítóprogram; ha ehhez túl nagy, akkor long int-ként értelmeződik. Ha még ebben sem fér el az érték, akkor unsigned long int-ként próbálja meg betudni a rendszer. Ha így sem fér el, akkor long long int-ként értelmezi, vagy ha ez is szűk a konstans számára, akkor unsigned long long int lesz a típusa.

3.2 Lebegőpontos konstansok

Egy lebegőpontos konstans a következőképp épül fel: számjegyek sorozata után tizedespont áll, majd számjegyek újabb sorozata. A szám előtt állhat mínusz előjel, ez értelemszerűen negatív értéket jelöl. A tizedespont előtti vagy utáni számsorozat elhagyható (de nem mindenkor). A tizedespont előtt álló nulla elhagyható.

Ha a lebegőpontos szám után közvetlenül egy e (vagy E) betű áll, majd egy (esetleg előjel) egész, akkor a konstans *normálalakban* értelmeződik. Ez utóbbi egész szám (az *exponens, kitevő*) mutatja azt a tíz-hatványt, amivel az e előtti számot (*mantissát, törtrészt*) meg kell szorozni. Az $1.5e-2$ például $1.5 \cdot 10^{-2}$ -t jelent, azaz 0,015-öt.

A hexadecimális lebegőpontos konstansok formátuma: a bevezető 0x vagy 0X után következik a hexadecimális számjegyek sorozata, esetlegesen „tizenhatodos pont” és újabb számjegysorozat. Mindezeket követheti még a kettes-kitevő (esetleg előjeles) megadása, p/P betűvel kezdve. A 0x3p10 például $3 \cdot 2^{10}$ -t jelent.

A lebegőpontos konstansokat double típusúként értelmezi a fordítóprogram. Az f vagy F utótaggal kényszeríthetjük ki, hogy double helyett inkább csak float-ként tartsa számon az adott értéket a program. Az l vagy L utótaggal long double konstanst adhatunk meg.

3.3 Karakterkonstansom

Az aposztrófok közé zárt karaktert hívjuk karakterkonstansnak. Ha egynél több karaktert fogunk aposztrófok közé, akkor ennek a jelsorozatnak az értelmezése implementációfüggő. Nemzetközi karaktereket is megadhatunk karakterkonstansként az 1.2.1 szakasz jelölésmódjával, ha a szükséges karakter nincs benne az alapértelmezett karakterkészletben.

3.3.1 Vezérlősorozatok (escape sequences)

A backslash karakterrel (\) kezdődő karakterláncoknak speciális jelentése van. Ezeket találjuk meg az A.3 Táblázatban.

A.3 Táblázat • Speciális vezérlősorozatok

Vezérlőkarakter	Hatása
\a	Hangjelzés
\b	Visszatörlés
\f	Laptovábbítás
\n	Újsor karakter
\r	Kocsi vissza
\t	Vízszintes tabulátor
\v	Függőleges tabulátor
\\\	Backslash

Vezérlőkarakter	Hatása
\"	Idézőjel
\'	Aposztróf
\?	Kérdőjel
\nnn	nnn értékű oktális (8-as számrendszerbeli) karakter
\unnnn	16 bites UNICODE karakter
\Unnnnnnnn	32 bites UNICODE karakter
\xnn	nn értékű hexadecimális karakter

Az oktális karaktereknél egytől háromig terjedhet a megadott számjegyek száma. Az utolsó három sorban hexadecimális karakterek használhatóak.

3.3.2 Széles karakterkonstansok

A széles karakterkonstans megadási módja *L 'x'*. Az ilyen konstans típusa `wchar_t`, mely a `<stddef.h>` szabványos fejlécállományban van definiálva. A széles karakterkonstans lehetővé teszi, hogy olyan karakterkészlet karaktereit is ábrázoljuk, amelyek nem férnek el a `char` típusban.

3.4 Karakterlánc-konstansok

Az idézőjelek közé foglalt karakterláncot karakterlánc-konstansnak hívjuk. Az üres karakterlánc is egy konstans. Bármilyen érvényes karakter használható a karakterláncban, beleértve az előző táblázatban felsorolt vezérlőkaraktereket. A fordítóprogram automatikusan lezárja a karakterlánc végét egy null ('\'0') karakterrel.

A fordítóprogram általában előállít egy mutatót, amely a karakterlánc első karakterére mutat (tehát egy `char` típusra irányuló mutató keletkezik). Amikor azonban a `sizeof` vagy `&` operátorral együtt arra használunk egy karakterlánc-konstanst, hogy inicializálunk egy karaktertömböt, akkor a karakterlánc-konstans típusa karaktertömb lesz.

A karakterlánc-konstans nem módosítható a program futása közben.

3.4.1 Karakterláncok összekapcsolása

Az előfeldolgozó automatikusan összekapcsolja a szomszédos karakterlánc-konstansokat. A karakterláncok közt nulla vagy több térköz karakter állhat. Így az összekapcsolás után "a világ érdekes"-ként értelmeződik a következő kódrészlet:

```
"a" " világ "
  "érdekes"
```

3.4.2 Több bájton tárolt karakterek

A karakterláncok megadásakor implementációfüggő módon lehet a különféle vezérlőkarakterekkel bekapsolni (és vissza) a több bájton tárolt karakterek jelzését.

3.4.3 Széles karakterekből képzett konstansok

Kiterjesztett karakterkészletből álló karakterekből az L"..." formátumjelzéssel lehet karakterláncot létrehozni. Az ilyen konstansok típusa „wchar_t-re irányuló mutató”.

A wchar_t típus a <stddef.h> fejlécállományban van definiálva.

3.5 Felsorolt konstansok

A felsorolt típusuként deklarált azonosítók konstansnak minősülnek; a fordítóprogram int-ként kezeli őket.

4.0 Adattípusok és deklarációk

Ebben a szakaszban összefoglaljuk az alapvető adattípusokat, a származtatott adattípusokat, a felsorolt adattípusokat és a typedef kulcsszót. Ugyancsak ebben a szakaszban van összegezve a változók deklarációjának formátuma.

4.1 Deklarációk

Amikor egy adatszerkezetet, uniót, felsorolt típust vagy typedef-es típust definiálunk, a fordítóprogram nem foglalja le automatikusan a szükséges tárterületet. A definíció csak arról tájékoztatja a fordítóprogramot, hogy milyen típusról lesz szó (és esetleg mi lesz a neve). A definíciókat függvényen belül vagy azon kívül is meg lehet tenni. Ha függvényen belül jelenik meg a definíció, akkor csak az adott függvény fog tudni a létezéséről, a másik esetben viszont az adott állomány további kód részleteiben látszani fog a függvény.

Miután megtörténik egy típusdefiníció, a változók már az adott típusuként dekláralhatók. A deklarációkor már lefoglalódik a szükséges memóriaterület a változók számára, ha csak nem extern (külső) deklarációról van szó. Ebben az esetben vagy történik tárterület-foglalás, vagy nem (lásd a 6.0 szakaszt).

Az is lehetséges, hogy a definícióval egy lépésben (akár egyszerre több) adatszerkezet, unió vagy felsorolt típusú változónak foglalunk le helyet. Ehhez egyszerűen fel kell sorolni a változók neveit a definíciót lezáró pontosvessző előtt.

4.2 Alapvető adattípusok

A C nyelv alapvető adattípusait az A.4 Táblázatban láthatjuk. Egy változót a következőképp dekláralhatunk valamely alapvető adattípusúnak:

típus név = kezdőérték;

Nem kötelező a kezdőérték megadása – ennek szabályait a 6.2 szakaszban olvashatjuk. Egyszerre akár több változó is deklarálható az alábbi módon:

`típus név = kezdőérték, név = kezdőérték, ... ;`

A típusdeklaráció előtt megadható a tárolás módja is – ezekről is a 6.2 szakasz ad bővebb információt. Ha megadunk tárolási osztályt, és a változó int típusú, akkor az int elhagyható. Az alábbi utasítás például static int-nek deklarálja a counter változót:

`static counter;`

A.4 Táblázat • Az alapvető adattípusok összefoglalása

Típus	Jelentése
int	Egész számérték (azaz nincs benne tizedespont). Legalább 16 bites tárolás garantált.
short int	Csökkentett pontosságú egész számérték. Néhány géptípuson feleannyi memóriát használ, mint az int típus. Legalább 16 bites tárolás garantált.
long int	Megnövelt pontosságú egész számérték. Legalább 32 bites tárolás garantált.
long long int	Erősen megnövelt pontosságú egész számérték. Legalább 64 bites tárolás garantált.
unsigned int	Pozitív egész számérték; kétszer akkora számokat is tud tárolni, mint az int típus. Legalább 16 bites tárolás garantált.
float	Lebegőpontos számérték; azaz tartalmaznia kell tizedespontot. Legalább hat tizedesjegy pontosság garantált.
double	Megnövelt pontosságú lebegőpontos számérték. Legalább tíz tizedesjegy pontosság garantált.
long double	Erősen megnövelt pontosságú lebegőpontos számérték. Legalább tíz tizedesjegy pontosság garantált.
char	Egyetlen karakter. Néhány rendszeren előjel-kiterjesztés léphet fel a kifejezésekben.
unsigned char	Egyetlen karakter, akárcsak a char típus, azzal az eltéréssel, hogy egész átalakításkor sem lép fel előjel-kiterjesztés a kifejezésekben.
signed char	Mint a char, de ennél a típusnál garantált az egész átalakításkor fellépő előjel-kiterjesztés.
_Bool	Logikai típus, amely kellő méretű ahhoz, hogy a 0-t és az 1-est tárolni tudja.
float _Complex	Komplex szám.

Típus	Jelentése
<code>double _Complex</code>	Megnövelt pontosságú komplex szám.
<code>long double _Complex</code>	Erősen megnövelt pontosságú komplex szám.
<code>void</code>	Üres típus. Annak jelzésére használatos, hogy egy függvénynek nincs visszatérési értéke; azaz kifejezetten „eldobandó”, amit visszaad. Általános mutatótípusként is használják (<code>void *</code>).

Megjegyzés: a `signed` módosító használható a `short int`, `int`, `long int` és `long long int` típusoknál is; azonban ezek a típusok már alapértelmezetten előjelek, ezért rájuk nézve nincs hatása a `signed` módosítónak.

A `_Complex` és az `_Imaginary` adattípusok lehetővé teszik a komplex és a képzetes számok deklarációját és használatát. A programkönyvtárakban megvannak a kezelésükhez szükséges függvények. Ezek használatához általában elegendő beemelni a `<complex.h>` fejlécállományt, amelyben megtalálhatóak a komplex (és képzetes) számok használatához szükséges makrók és deklarációk. Egy `c1` nevű `double _Complex` változót például a következőképpen lehet deklarálni és inicializálni $5 + 10i$ értékre:

```
double _Complex c1 = 5 + 10.5 * I;
```

A `creal` és a `cimag` könyvtári függvények használhatóak `c1` valós és képzetes részének kinyeréséhez.

Nem minden implementáció támogatja a `_Complex` és az `_Imaginary` típust; az is előfordul, hogy a kettő közül csak az egyiket.

A `<stdbool.h>` fejlécállomány beemelésével könnyebben kezelhetőek a logikai változók: ekkor definiálásra kerül a `bool`, a `true` és a `false` makró, amelynek segítségével ilyen utasítások fogalmazhatóak meg:

```
bool endOfData = false;
```

4.3 Származtatott adattípusok

Azokat az adattípusokat hívjuk származtatott adattípusoknak, melyek egy vagy több alapvető típusból vannak összeállítva. Ilyenek a tömbök, struktúrák, uniók és mutatók. Az a függvény is származtatott adattípusnak számít, amely a felsorolt típusok valamelyikét adjá vissza. Ezekről az adattípusokról lesz szó a következőkben. A függvényeket külön tárgyaljuk a 7.0 szakaszban.

4.3.1 Tömbök

Egydimenziós tömbök

A tömbök tetszőleges alapvető vagy származtatott adattípushoz tartalmazhatnak elemként. Függvények nem képezhetnek tömböket (de függvénymutatók igen).

Egy tömb deklarációja a következőképp történhet:

```
típus név[n] = { inicializálóKif, inicializálóKif, ... };
```

Itt az *n* határozza meg a név nevű tömb elemei számát. Ez azonban elhagyható akkor, amikor inicializáló kifejezéseket adunk meg. Ilyenkor a tömb elemszámát a megadott kifejezések száma (vagy a legnagyobb megadott indexű inicializáló kifejezés) határozza meg.

Globális tömbök esetén minden inicializáló kifejezésnek konstansnak kell lennie. Nem kell a tömb minden elemét inicializálni – az elemszámnál többet viszont nem lehetséges. Ha a tömb elemszámánál kevesebb elemet inicializálunk, akkor a többi elem nullázódik.

A tömb-inicializáció speciális esete a karaktertömböké, amely karakterláncgal is inicializálható. Például a

```
char today[] = "Szerda";
```

utasítás hatására a *today* karaktertömbként deklarálódik. Elemei a következők lesznek: 'S', 'z', 'e', 'r', 'd', 'a' és '\0'.

Ha úgy adjuk meg a tömb elemszámát, hogy csak az adott betűk férjenek el benne, akkor a fordítóprogram nem helyez el null karaktert a tömb végére. A

```
char today[6] = "Szerda";
```

hatására a *today* tömb 6 elemű lesz, melyek a következők: 'S', 'z', 'e', 'r', 'd' és 'a'.

Ha az elem sorszámát szögletes zárójelben megadjuk, akkor tetszőleges sorrendben is inicializálhatóak az elemek. Például az

```
int x = 1233;
int a[] = { [9] = x + 1, [3] = 3, [2] = 2, [1] = 1 };
```

utasítások hatására létrejön egy a névű tízelemű tömb (hiszen a maximális elem-sorszám a 9). Az utolsó elem értéke *x* lesz (azaz 1234), az első három elem pedig 1, 2 és 3.

4.3.1.1 Változó méretű tömbök

Egy függvényen vagy utasításblokkon belül úgy is megadhatjuk egy tömb méretét, hogy ehhez egy (változót is tartalmazó) kifejezést használunk. Így a tömb mérete futási időben számítódik ki. Az

```
int makeVals (int n)
{
    int valArray[n];
    ...
}
```

függvény például létrehoz egy automatikus változót, az n elemű valArray függvényt. Az n értéke futási időben értékelődik ki, és az egyes függvényhívások közben akár meg is változhat. A változó méretű tömbök nem inicializálhatóak a deklarációkor.

4.3.1.2 Többdimenziós tömbök

Egy többdimenziós tömb deklarációjának általános alakra:

```
típus név[d1][d2]...[dn] = inicializálóLista;
```

A név nevű tömb $d_1 \times d_2 \times \dots \times d_n$ darab, típus típusú elemet fog tartalmazni. A következő utasítás például egy egészekből álló (kétszáz elemű) háromdimenziós tömböt hoz létre:

```
int three_d [5][2][20];
```

Egy többdimenziós tömb adott elemére (szögletes zárójelbe tett) megfelelő indexekkel hivatkozhatunk. Az alábbi utasítással például értékül adjuk a three_d egyik elemének a 100-at:

```
three_d [4][0][15] = 100;
```

A többdimenziós tömbök az egydimenziósokhoz hasonló módon inicializálhatóak. Egy másba ágyazott kapcsos zárójelpárokkal vezérelhetjük az értékek hozzárendelését az egyes tömbelemekhez. A következő kód részlet a matrix tömböt egy négy sorból és három oszloból álló tömbként deklarálja:

```
int matrix[4][3] =
{ { 1, 2, 3 },
{ 4, 5, 6 },
{ 7, 8, 9 } };
```

A matrix első sorának értékei 1, 2 és 3; a második soré 4, 5 és 6; végül a harmadik soré 7, 8 és 9. A negyedik sor értéke 0-kra áll be, mivel nincsenek konkrét értékek megadva. Ugyanezt a hatást éri el a

```
static int matrix[4][3] =
{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

deklaráció. A többdimenziós tömbök inicializációja ugyanis „dimenzió-sorrendben” történik, azaz a baloldali dimenziótól haladunk a jobboldali felé. Az

```
int matrix[4][3] =
{ { 1 },
{ 4 },
{ 7 } };
```

deklaráció matrix első sorának első elemét 1-re, második sorának első elemét 4-re, harmadik sorának első elemét pedig 7-re állítja be; a többi érték nullázódik.

Végül tekintsük meg a következő deklarációt; ebben csak néhány elemet adunk meg a megfelelő indexek jelzésével:

```
int matrix[4][3] = { [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

4.3.2 Adatszerkezetek

Egy adatszerkezet megadásának általános formája a következő:

```
struct név
{
    tagDeklaráció
    tagDeklaráció
    ...
} változóLista;
```

A név nevű adatszerkezet a tagDeklarációknak megfelelően definiálódik. minden ilyen tagDeklarációhoz meg kell adni típusjelzést és a megfelelő adattag(ok) nevét.

A változók az adatszerkezet definíálásakor is deklarálhatóak (nevüket a pontosvessző előtt felsorolva), vagy utólag, a következő formában:

```
struct név változóLista;
```

Ez a deklaráció csak akkor használható, ha nem hagytuk el a *név* nevet az adatszerkezet definíójakor. A *név* ugyanis elhagyható a fenti definíciós formából, de akkor ott helyben kell megadni az összes változónevet, amit a definiált adatszerkezetként szeretnénk létrehozni.

Egy adatszerkezet-változó inicializációja a tömbökhez hasonló. Az egyes adattagok kapcsos zárójelben felsorolt értékekkel inicializálhatóak:

```
struct point
{
    float x;
    float y;
} start = {100.0, 200.0};
```

Itt létrejön a *point* adatszerkezet, és ezzel egy időben a *start* nevű, *point* adatszerkezetű változó a megadott kezdőértékekkel. Az egyes adattagok közvetlenül is inicializálhatóak (sorrendtől függetlenül) a

```
.adattag = érték
```

kifejezéssel, például így:

```
struct point end = { .y = 500, .x = 200 };
```

A

```
struct entry
{
    char *word;
    char *def;
} dictionary[1000] = {
    { "a",           "first letter of the alphabet" },
    { "aardvark",    "a burrowing African mammal" },
    { "aback",       "to startle" }
};
```

utasítással a *dictionary* (*szótár*) adatszerkezetet ezer *entry* bejegyzés tárolására deklárljuk, melyek közül az első hármat inicializáljuk is. Ugyanezt a hatást érhetjük el a következő módon, konkrétan hivatkozva egyes adattagokra:

```
struct entry
{
    char *word;
    char *def;
} dictionary[1000] = {
    [0].word = "a",           [0].def = "first letter of the alphabet",
```

```
[1].word = "aardvark", [1].def = "a burrowing African mammal",
[2].word = "aback",     [2].def = "to startle"
};
```

Sőt a következő módon is megoldhatjuk ugyanezt:

```
struct entry
{
    char *word;
    char *def;
} dictionary[1000] = {
{ {.word = "a",           .def = "first letter of the alphabet" },
{ {.word = "aardvark",    .def = "a burrowing African mammal" },
{ {.word = "aback",       .def = "to startle" }
};
```

Egy automatikus adatszerkezet-változó inicializálható egy másik (ugyanolyan típusú) adatszerkezet alapján is:

```
struct date tomorrow = today;
```

Ezzel (a korábban deklarált) date adatszerkezetű today változó alapján adunk kezdőértéket a tomorrow változónak, amely szintén date adatszerkezetű.

A tagDeklaráció formátuma a következő is lehet:

```
típus mezőNév : n
```

Az n értéke egy egész szám: ez adja meg a mezoNév nevű bitmező szélességét. Egyes gépeket balról jobbra, más gépeken jobbról balra kerülnek elhelyezésre a bitmezők.

Ha nem adunk meg mezőnevet, akkor is lefoglalódik a megadott számú bit, csak nem tudunk rá hivatkozni. Ha nincs megadva mezőnév és az n értéke 0, akkor a következő mező a tárolási egység szélére kerül. A tárolási egység az implementációtól függően definiált. A mezők típusa lehet _Bool, int, signed int vagy unsigned int. Az int-et néhány implementáció előjelesként, mások előjel nélkülüként értelmezik. Bitmezőkre nem használható a „címe” operátor (&), és tömbök sem hozhatók létre bitmezőkből.

4.3.3 Uniók

Egy unió megadásának általános formája a következő:

```
union név
{
    tagDeklaráció
    tagDeklaráció
    ...
} változóLista;
```

A név nevű unió a *tagDeklaráció*-knak megfelelően definiálódik. Az unió tagjai közös tárterületen osztoznak. A fordítóprogram csak arra ügyel, hogy a legnagyobb helyigényű tag is beférjen az unió memóriaterületére.

Az unió definiálásakor megadhatjuk a változók neveit is:

```
union név változóLista;
```

Ez a deklaráció csak akkor használható, ha megadtunk *név* nevet az unió definíciójakor.

Az a programozó felelőssége, hogy az unióban utoljára eltárolt típust olvassa ki a változóból. Az unió egyik adattagja inicializálható. Ha ilyenkor nem nevezünk meg adattagot, akkor az unió első adattagjához rendelődik a megadott érték. Amennyiben globális változóról van szó, akkor (kapcsos zárójelbe írva) konstans kifejezést kell megadnunk:

```
union shared
{
    long long int l;
    long int      w[2];
} swap = { 0xffffffff };
```

Ebben az utasításban a *swap* nevű unió 1 adattagját állítjuk be a hexadecimális ffffffff értékre. Az adattag nevének megadásával az unió bármely adattagja inicializálható, például így:

```
union shared swap2 = {.w[0] = 0x0, .w[1] = 0xffffffff; }
```

Egy automatikus unió változó alapján inicializálható egy másik (ugyanolyan típusú) változó:

```
union shared swap2 = swap;
```

4.3.4 Mutatók

Egy mutató deklarációjának alapvető formája a következő:

típus **név*

A név azonosítót „*típus*-ra irányuló mutatóként” deklaráljuk. A *típus* lehet alapvető vagy származtatott adattípus. Az alábbi utasítás ip-t egy int-re irányuló mutatóként deklarálja:

```
int *ip
```

A következő deklarációval pedig `ep`-vel egy `entry` adatszerkezetre mutatunk:

```
struct entry *ep;
```

Azokat a mutatókat, amelyek egy tömb valamely elemére mutatnak, úgy kell deklárnai, mintha a tömb elemtípusára irányulnának. Az előző példa `ip` mutatójával tehát rámutathattunk egy egészkből álló tömb valamely elemére is.

A mutatókat összetettebb módon is dekláralhatjuk. Az alábbi utasítás `tp`-t egy száz elemű karaktertömbre irányítja:

```
char *tp[100];
```

A következő sor pedig az `fnPtr`-t olyan függvénymutatóként dekláralja, amely egy `entry` adatszerkezetet ad vissza, és egy `int` paramétert vár:

```
struct entry (*fnPtr) (int);
```

A mutatókat 0-val összehasonlítva megvizsgálhatjuk, hogy nullmutatóról van-e szó. Az már implementációfüggő, hogy a nullmutatót belsőleg 0 vagy más érték tárolásával reprezentálja-e a rendszer. Azonban a rendszernek igazként kell kiértékelnie egy mutató 0-val való összevetését, ha az nullmutató.

A mutatók egészkeké alakítása (és a másik irányú átalakítás is) gépfüggő, minthogy akkorra egész számot kell használni, amiben elfér a mutató értéke.

Létezik egy általános mutatótípus, a „`void`-ra irányuló mutató”. A nyelvi definíció garantálja, hogy bármilyen mutató értékül adható `void` mutatónak (és fordítva) anélkül, hogy a mutató értéke megváltozna.

Ettől az esettől eltérő értékadás egyik fajta mutatóból a másik fajtába nem megengedett; általában ad is rá egy figyelmeztető üzenetet a fordítóprogram.

4.4 Felsorolt adattípusok

A felsorolt adattípusok deklarációjának általános formája a következő:

```
enum név { enum_1, enum_2, ... } változóLista;
```

A `név` nevű felsorolt típust az `enum_1`, `enum_2`, ... definiálja – ezek mindegyike egy önmagában álló azonosítónév vagy egy azonosítónév, mely után egy egyenlőségjel és egy konstans kifejezés áll. A `változóLista` nem kötelező. Az itt megadott változók a megadott `név` nevű felsorolt típushoz fognak tartozni.

A fordítóprogram (0-val indítva) egymást követő egész számokat fog hozzárendelni a felsorolt azonosítókhoz. Ha valamelyik azonosítóhoz = jel és konstans kifejezés tartozik, akkor ennek a kifejezésnek az értéke rendelődik hozzá az azonosítóhoz. Az ezt követő azonosító (hacsak nincs más érték megadva a számára) az eggyel nagyobb egész számot kapja meg. A fordítóprogram konstans egészekként kezeli a felsorolt azonosítókat.

Ha egy előzőleg definiált (és elnevezett) felsorolt típusú változót szeretnénk deklarálni, azt megtehetjük a következő módon:

```
enum név változóLista;
```

Egy adott felsorolt típusként deklarált változóhoz csak az adott típusnak megfelelő értéket lehet hozzárendelni. Léteznek azonban fordítóprogramok, amelyek más esetben sem adnak hibajelzést.

4.5 A **typedef** utasítás

A **typedef** kulcsszó segítségével az (alapvető vagy származtatott) adattípusoknak saját nevet adhatunk. Ilyenkor nem definiálunk új típust, csak egy meglévő típusnak adunk új nevet. Így az újonnan elnevezett típusukként deklarált változókat pontosan úgy tekinti a fordítóprogram, mintha a megadott névhez társított típusukként lett volna deklarálva.

A **typedef** definíció létrehozásakor úgy kell eljárni, mintha egy normál változót deklarálnánk. A létrehozandó változó nevének a helyére írjuk be az új típusnevet. Végül pedig mindenek előre írjuk oda a **typedef** kulcsszót. A következő típusdefinícióval a **Point** nevet egy olyan adatszerkezethez társítjuk, amely két lebegőpontos számot, az **x**-et és az **y**-t tartalmazza:

```
typedef struct
{
    float x;
    float y;
} Point;
```

Ezek után már tudunk **Point** típusú változókat deklarálni; az origót például így adhatjuk meg:

```
Point origo = { 0.0, 0.0 };
```

4.6 A const, a volatile és a restrict típusmódosítók

A `const` kulcsszót olyan esetben érdemes egy típusdeklaráció elé odaírni, amikor azt szeretnénk közölni a fordítóprogrammal, hogy az adott érték nem változhat meg. Így a

```
const int x5 = 100;
```

deklaráció `x5`-öt konstans egészkként adja meg (azaz kimondjuk róla, hogy nincs szándékban megváltoztatni a program futása során). A fordítóprogramnak *nem* dolga vissza-jelezni, ha valaki megkísérli egy konstans változó megváltoztatását.

A `volatile` módosító kifejezetten arra buzdítja a fordítóprogramot, hogy számítson arra, hogy az adott érték lépten-nyomon változni fog (többnyire dinamikusan). Ha egy kifejezés `volatile` változót használ, akkor annak értékét minden előforduláskor újra lekérdezi a rendszer.

A `port17` változót „`volatile` mutató `char`-ra” típusukként deklarálhatjuk a következő utasítással:

```
volatile char *port17;
```

A `restrict` kulcsszó mutatókkal együtt használatos. A `register` minősítőhöz hasonlóan a fordítás optimalizációját célzza. A `restrict` kulcsszó azt tudatja a fordítóprogrammal, hogy az adott mutató az egyetlen hivatkozás egy objektumra; azaz nincs több mutató vagy változó, amely ugyanoda mutatna az aktuális hatókörön belül. Az alábbi sorokkal tájékoztathatjuk a fordítóprogramot arról, hogy az `intPtrA` és `intPtrB` hatókörében a két mutató soha nem fog azonos memóriaterületre mutatni:

```
int * restrict intPtrA;
int * restrict intPtrB;
```

Ha egy egészeket tartalmazó tömb elemeire mutatunk velük, akkor a `restrict` kulcsszóval kizártuk azt, hogy azonos elemre mutassanak.

5.0 Kifejezések

A változók, függvények és tömbök nevei, a konstansok, a függvényhívások, a tömbelem-hivatkozások, a struktúrák és uniók hivatkozásai mind-mind kifejezések. A különféle kifejezésekre alkalmazhatunk egyoperandusú operátorokat, illetve összeköthetjük őket két- vagy háromoperandusú operátorral: ezek újabb kifejezéseket hoznak létre. Egy zárójelbe zárt kifejezés is önálló kifejezésnek tekinthető.

Balértéknek (*value*) hívunk minden `void`-tól eltérő típusú kifejezést, amely egy adatobjektumot azonosít. Ha értéket is rendelhetünk hozzá, akkor ez egy *módosítható balérték*.

Bizonyos helyzetekben módosítható balértékből álló kifejezésekre van szükség. Például egy hozzárendelő operátor baloldalán csak módosítható balértéket lehet megadni. Az inkrementáló és dekrementáló operátorok is csak módosítható balértékekre alkalmazhatóak, akárcsak az egyoperandusú & „címe” operátor (hacsak nem függvényről van szó).

5.1 A C operátorok összefoglalása

Az A.5 Táblázat foglalja össze a C nyelv különféle operátorait végrehajtási sorrendben (azaz a kisebb precedenciájúak fent, a nagyobbak lent szerepelnek). Az azonos végrehajtási sorrendű operátorokat vízszintes vonalakkal csoportosítottuk.

A.5 Táblázat • A C operátorok összefoglalása

Operátor	Leírás	Asszociativitási sorrend
()	Függvényhívás	Balról jobbra
[]	Hivatkozás egy tömb valamely elemére	
->	Adatszerkezet tagjára hivatkozó mutató	
.	Hivatkozás egy adatszerkezet valamely tagjára	
-	Egyoperandusú mínusz	Jobbról balra
+	Egyoperandusú plusz	
++	Inkrementálás	
--	Dekrementálás	
!	Logikai tagadás	
~	Bitenkénti negáció	
*	Közvetett hivatkozás (indirekció)	
&	Címe	
<code>sizeof</code>	Egy objektum memóriabeli mérete	
(típus)	Típuskonverzió	
*	Szorzás	Balról jobbra
/	Osztás	
%	Osztási maradék (modulus)	
+	Összeadás	Balról jobbra
-	Kivonás	
<<	Bitenkénti eltolás balra	Balról jobbra
>>	Bitenkénti eltolás jobbra	

Operátor	Leírás	Asszociativitási sorrend
<	Kisebb	Balról jobbra
<=	Kisebb vagy egyenlő	
>	Nagyobb	
=>	Nagyobb vagy egyenlő	
==	Egyenlő	
!=	Nem egyenlő	
&	Bitenkénti ÉS	Balról jobbra
^	Bitenkénti KIZÁRÓ VAGY	Balról jobbra
	Bitenkénti VAGY	Balról jobbra
&&	Logikai „és”	Balról jobbra
	Logikai „vagy”	Balról jobbra
? :	Feltételkezelő operátor	Jobbról balra
=	Hozzárendelési operátorok	Jobbról balra
* = / = % =		
+ = - = & =		
^ = =		
<<= >>=		
,	Vessző operátor	Jobbról balra

Az A.5 Táblázat használatának megértéséhez tekintsük például a következő kifejezést:

b | c & d * e

A szorzás operátor precedenciája magasabb, mint a bitenkénti VAGY és a bitenkénti ÉS operátoré, mivel lentebb található e két utóbbinál. Az ÉS operátor végrehajtási sorrendje megelőzi a VAGY-ot, mivel alatta látható. Ennél fogva a következő zárójelezés mutatja az alapértelmezett végrehajtási sorrendet:

b | (c & (d * e))

Most tekintsük a következő egyszerű kifejezést:

b % c * d

Mivel a maradékképző operátor és a szorzási operátor az A.5 Táblázat azonos csoportjában van, azonos kiértékelési szint tartozik hozzájuk. A megadott asszociativitási sorrendet kell tehát figyelembe vennünk a kiértékelési sorrend megállapításához: eszerint „balról jobbra” haladhatunk:

(b % c) * d

Nézzük a következő példát:

`++a->b`

Ez a következő módon értékelődik ki, mivel a `->` operátor precedenciája magasabb a `++` operátorénál:

`++(a->b)`

A hozzárendelési operátorok azonos kiértékelési szinten találhatóak, ezért az

`a = b = 0;`

kifejezés

`a = (b = 0);`

módon értelmeződik, azaz a és b értéke is nullázódik.

Az

`x[i] + ++i`

kifejezés esetében nincs pontosan értelmezve, hogy a fordítóprogramnak először a `+` jel baloldalán vagy jobboldalán álló kifejezést kell kiértékelnie, pedig az eredményt ez erősen befolyásolja. Nem mindegy ugyanis, hogy az `i` értéke az `x[i]` olvasása előtt vagy után inkrementálódik.

A következő utasításban sincs pontosan definiálva a kiértékelési sorrend:

`x[i] = ++i`

Itt sincs megadva, hogy az `i` értéke az indexként való használata előtt, vagy csak utána inkrementálódik.

A függvényparaméterek kiértékelési sorrendje sincs definiálva nyelvi szinten. Az

```
f (i, ++i);
```

kifejezésben nem lehet tudni, hogy az első *i* kiértékelése előtt inkrementálódik-e (azaz megegyezik-e a két paraméter értéke a függvényhíváskor).

A C nyelv garantálja, hogy az `&&` és a `||` operátorok balról jobbra értékelődnek ki. Az erőforrásokkal való takarékos bánásmód jegyében csak addig folyik e két operátor kiértékelése, míg biztos nem lesz az eredmény. Azaz az `&&` operátornak *csak* a baloldalán álló kifejezés értékelődik ki, amennyiben az 0, valamint a `||` operátornak is *csak* a baloldala értékelődik ki, ha az nem nulla. Ezt a tényt érdemes szem előtt tartani a következő típusú kifejezések használatakor:

```
if ( dataFlag || checkData (myData) )  
    ...
```

Itt a `checkData` csak akkor hívódik meg, ha a `dataFlag` értéke 0. Egy másik példa egy tömb adott méretre történő feltételes definiálása:

```
if (index >= 0 && index < n && a[index] == 0))  
    ...
```

Az a tömb csak akkor definiálódik, ha az `index` mérete megfelelő.

5.2 Konstans kifejezések

Azok a konstans kifejezések, melyeknek minden részkifejezése konstans érték. Vannak esetek, amikor konstans kifejezést *kell* használnunk:

1. A `switch` utasításban az egyes ágak megadásánál
2. Egy globálisan deklarált vagy inicializálás alatt álló tömb méretének megadásánál
3. Felsorolt változó azonosítójához való hozzárendeléskor
4. Egy adatszerkezet bitmező-méretének meghatározásakor
5. Statikus változók kezdőértékének megadásakor
6. Globális változók kezdőértékének megadásakor
7. Az előfeldolgozónak szánt `#if` utasítás után

Az első négy esetben a konstans kifejezést a következők alkothatják: egész konstansok, karakteres konstansok, felsorolt konstansok és `sizeof` kifejezések. A használható operátorok: aritmetikai és bitenkénti műveleti jelek, relációs jelek, feltételkezelő operátor és a típusátalakító operátor. A `sizeof` operátor változó méretű tömböt tartalmazó kifejezésekre nem használható, mert itt az eredmény futási időben értékelődik csak ki, azaz a kifejezés nem konstans.

Az ötödik és hatodik esetben az imént felsorolt lehetőségeken túl a „címe” operátor is használható (*implicit*, azaz beleértett módon, vagy *explicit*, azaz kifejezett módon is), azonban csak globális vagy statikus változókra/függvényekre. A következő egy érvényes konstans kifejezés, amennyiben x egy globális vagy egy statikus változó:

```
&x + 10
```

Hasonlóképp, a következő is egy érvényes konstans kifejezés, ha a egy globális vagy statikus tömb:

```
&a[10] - 5
```

Végül, minthogy &a[0] egyenértékű az a kifejezással, ez is egy érvényes konstans kifejezés:

```
a + sizeof (char) * 100
```

A hetedik felsorolt esetben, amikor követelmény a konstans kifejezés (#if után), ugyanazok a szabályok érvényesek, mint az első négy esetben, de ott nem használható a sizeof operátor, sem a felsorolt konstansok, sem pedig a típusátalakító operátor. Azonban az előfeldolgozó saját defined operátora használható (lásd a 9.2.3 szakaszt).

5.3 Aritmetikai operátorok

Legyen

a, b	két (void-tól különböző) alapvető adattípus,
i, j	pedig két egész adattípus.

Ekkor a

-a	kifejezés az a ellentettjét képezi,
+a	kifejezés magát az a-t adja,
a + b	kifejezés összeadja a-t és b-t,
a - b	kifejezés kivonja a-ból b-t,
a * b	kifejezés összeszorozza a-t és b-t,
a / b	kifejezés a-t elosztja b-vel,
i % j	kifejezés i-nek j-vel való osztási maradékát adja.

A felsorolt kifejezések mindegyikében megtörténnek az operandusra vonatkozó szokásos aritmetikai konverziók (lásd az 5.17 szakaszt). Ha a egy előjel nélküli érték, akkor -a először „egész átalakításon” megy át, azaz a rendszer veszi az új egész típus legnagyobb értékét, kivonja belőle a-t, majd az eredményhez hozzáad 1-et.

Amikor két egész értéket osztunk el egymással, az eredmény törtrésze elvész. Ha az egyik operandus negatív, akkor ez a „törtrész” fogalom kétféleképpen is érthető. Gépfüggő,

hogy „lefelé” vagy a „nulla felé” kerekítődik-e a szám. Azaz a $-3 / 2$ értéke néhány gépen -1 , más gépeken -2 lesz. Pozitív törtétekben a kerekítés egyértelmű: a $3 / 2$ minden 1 -et ad. A mutatókkal történő aritmetikai műveleteket az 5.15 szakaszban részletezzük.

5.4 Logikai operátorok

Legyen

<code>a, b</code>	két (void-tól különböző) alapvető adattípus, vagy legyenek mutatók.
Ekkor az	
<code>a && b</code>	kifejezés 1 -et ad, ha <code>a</code> és <code>b</code> nem nulla, egyébként pedig 0 -t (<code>b</code> csak akkor értékelődik ki, ha <code>a</code> nem nulla),
<code>a b</code>	kifejezés 1 -et ad, ha <code>a</code> vagy <code>b</code> nem nulla, egyébként pedig 0 -t (<code>b</code> csak akkor értékelődik ki, ha <code>a</code> nulla),
<code>! a</code>	kifejezés 1 -et ad, ha <code>a</code> nulla, egyébként pedig 0 -t.

A szokásos aritmetikai átalakítások végbemennek `a`-n és `b`-n (lásd az 5.17 szakaszt). Az eredmény minden esetben `int` típusú.

5.5 Relációs operátorok

Legyen

<code>a, b</code>	két (void-tól különböző) alapvető adattípus, vagy legyenek mutatók.
Ekkor az	
<code>a < b</code>	kifejezés 1 -et ad, ha <code>a</code> kisebb, mint <code>b</code> , egyébként pedig 0 -t,
<code>a <= b</code>	kifejezés 1 -et ad, ha <code>a</code> kisebb vagy egyenlő, mint <code>b</code> , egyébként pedig 0 -t,
<code>a > b</code>	kifejezés 1 -et ad, ha <code>a</code> nagyobb, mint <code>b</code> , egyébként pedig 0 -t,
<code>a >= b</code>	kifejezés 1 -et ad, ha <code>a</code> nagyobb vagy egyenlő, mint <code>b</code> , egyébként pedig 0 -t,
<code>a == b</code>	kifejezés 1 -et ad, ha <code>a</code> egyenlő <code>b</code> -vel egyébként pedig 0 -t,
<code>a != b</code>	kifejezés 1 -et ad, ha <code>a</code> nem egyenlő <code>b</code> -vel egyébként pedig 0 -t.

A szokásos aritmetikai átalakítások végbemennek `a`-n és `b`-n (lásd az 5.17 szakaszt). Az előző négy relációs kifejezés csak akkor értelmezhető mutatókra, ha minden két változó ugyanarra a tömbre (vagy ugyanazon adatszerkezet, illetve unió elemeire) mutat. Az eredmény minden esetben `int` típusú.

5.6 Bitenkénti műveletek

Legyen

<code>i, j, n</code>	három egész adattípus.
----------------------	------------------------

Ekkor az

<code>i & j</code>	kifejezés <code>i</code> és <code>j</code> között egy bitenkénti ÉS műveletet hajt végre,
<code>i j</code>	kifejezés <code>i</code> és <code>j</code> között egy bitenkénti VAGY műveletet hajt végre,
<code>i ^ j</code>	kifejezés <code>i</code> és <code>j</code> között egy bitenkénti KIZÁRÓ VAGY műveletet hajt végre,
<code>~i</code>	kifejezés <code>i</code> bitenkénti negáltját adja,
<code>i << n</code>	kifejezés <code>i</code> bitjeit <code>n</code> bittel balra tolja,
<code>i >> n</code>	kifejezés <code>i</code> bitjeit <code>n</code> bittel jobbra tolja.

A szokásos aritmetikai átalakítások végbemennek az operandusokon, kivéve a << és a >> esetét, amikor csak egész átalakítás történik rajtuk (lásd az 5.17 szakaszt). Ha az eltolás mértéke negatív vagy legalább akkora, mint az eltolandó objektum bitjeinek száma, akkor az eredmény definiálatlan. A jobbra tolás néhány gépen aritmetikai (azaz az előjeltől függ a baloldalon megjelenő bit értéke), más gépeken pedig logikai (azaz nulla érkezik a baloldali biten). A bitenkénti eltolás eredményének típusa a baloldali operandus egész átalakítás utáni típusával egyezik meg.

5.7 Inkrementáló és dekrementáló operátorok

Legyen

`lv` egy módosítható, nem konstans balérték.

Ekkor a

`++lv` először megnöveli eggyel `lv` értékét, és ezt adja vissza az adott kifejezésben,

`lv++` először visszaadja `lv` értékét a kifejezésben, majd megnöveli `lv`-t eggyel,

`--lv` először lecsökkenti eggyel `lv` értékét, és ezt adja vissza az adott kifejezésben,

`lv--` először visszaadja `lv` értékét a kifejezésben, majd lecsökkenti `lv`-t eggyel.

A mutatókkal történő inkrementáló és dekrementáló műveleteket az 5.15 szakaszban részletezzük.

5.8 Hozzárendelő operátorok

Legyen

`lv` egy módosítható, nem konstans balérték,

`op` olyan operátor, amit használhatunk a hozzárendelő operátorba építve (lásd az A.5 Táblázatot),

`a` pedig egy kifejezés.

Ekkor az

`lv = a` kifejezés eltárolja a értékét `lv`-ben,

`lv op= a` kifejezés végrehajtja az `lv op` a műveletet, majd az eredményt eltárolja `lv`-ben.

Ha az első kifejezésben az a egy (`void`-tól különböző) alapvető adattípus, akkor átalakul úgy, hogy illeszkedjen `lv` típusához. Ha `lv` egy mutató, akkor `a`-nak ugyanilyen típusú (vagy `void`) mutatónak kell lennie, vagy nullmutatónak.

Ha `lv` `void` mutató, akkor a bármilyen típusú mutató lehet. A második kifejezés egyenér tékű egy `lv = lv op (a)` kifejezéssel;azzal a megszorítással, hogy `lv` csak egyszer értékelődik ki (gondoljunk például erre: `x[i++] += 10`).

5.9 A feltételkezelő operátorpár

Legyen

a, b, c három kifejezés.

Ekkor az

a ? b : c kifejezés *b*-t adja vissza, ha a nem nulla, egyébként pedig *c*-t; ennek megfelelően csak *b* vagy *c* értékelődik ki, de nem mindkettő.

b és *c* típusának meg kell egyeznie. Ha nem egyeznek meg, de mindenki aritmetikai (azaz számszerű) adattípus, akkor a szokásos aritmetikai átalakítások végben mennek rajtuk, és végül azonos típusúknak tekinti őket a rendszer. Ha egyikük mutató és a másik nulla, akkor az utóbbi egy (a másikkal megegyező típusú) nullmutatónak tekinti a rendszer. Ha az egyik *void* mutató és a másik valamely más típusú adatra mutat, akkor az utóbbi is *void* mutatóvá alakul (így a végeredmény típusa is az lesz).

5.10 A típusátalakító operátor

Legyen

<i>típus</i>	egy alapvető adattípus, egy (enum kulcsszóval ellátott) felsorolt adattípus, egy <i>typedef</i> -fel megadott típus vagy egy származtatott adattípus neve,
<i>a</i>	pedig egy kifejezés.

Ekkor a

(*típus*) *a* kifejezés *típus* típusúvá alakítja *a*-t.

5.11 A sizeof operátor

Legyen

<i>típus</i>	egy (fent jellemzett) adattípus neve,
<i>a</i>	pedig egy kifejezés.

Ekkor a

<i>sizeof</i> (<i>típus</i>)	kifejezés visszaadja a <i>típus</i> tárolásához szükséges bájtok számát,
<i>sizeof</i> <i>a</i>	kifejezés visszaadja az a kiértékelének tárolásához szükséges bájtok számát.

Ha a *típus* *char*, akkor az eredmény definíciószerűen 1. Ha a egy olyan tömb, amely már megkapta a méretét (kifejezetten vagy beleértett módon, inicializáció révén), és nem formális paraméter vagy dimenzionálatlan külső (*extern*) tömb, akkor a *sizeof* visszaadja az a-beli elemek tárolásához szükséges bájtok számát.

A *sizeof* által visszaadott egész szám típusa *size_t*, amely a *<stddef.h>* fejlécállományban van definiálva.

Ha a egy változó méretű tömb, akkor a *sizeof* csak futási időben értékelődik ki; minden más esetben fordítási időben történik meg a kiértékelése, így konstans kifejezésként használható (lásd az 5.2 szakaszt).

5.12 A vessző operátor

Legyen

a, b két kifejezés.

Ekkor az

a, b kifejezés hatására először a értékelődik ki, majd ezt követően b . A kifejezés b típusát és értékét adja vissza.

5.13 Alapvető műveletek tömbökkel

Legyen

a egy n eleműre deklarált tömb,
 i egy egész típusú kifejezés,
 v pedig egy tetszőleges kifejezés.

Ekkor az

$a[0]$ kifejezés az a tömb első elemét adja,
 $a[n - 1]$ kifejezés az a tömb utolsó elemét adja,
 $a[i]$ kifejezés az a tömb i indexű elemét adja,
 $a[i] = v$ kifejezés v értékét eltárolja az a tömb i indexű elemében.

A eredmény típusa minden esetben az a -ban tárolt elemek típusával egyezik meg.

A mutatókkal és tömbökkel történő műveletek összefoglalása az 5.15 szakaszban található.

5.14 Alapvető műveletek adatszerkezetekkel¹

Legyen

x egy s adatszerkezetű, módosítható balérték kifejezés,
 y egy s adatszerkezetű kifejezés,
 m egy s adatszerkezetű kifejezés valamely adattagja,
 v pedig egy tetszőleges kifejezés.

Ekkor az

x kifejezés a teljes adatszerkezetre utal – típusa s ,
 $y.m$ kifejezés az y adatszerkezet m adattagjára utal – típusa is ennek típusával egyezik meg,
 $x.m = v$ kifejezés v értékét eltárolja az x adatszerkezet m adattagjában – típusa is az adattag típusának felel meg,
 $x = y$ kifejezés a teljes y adatszerkezetet értékül adja x -nek – típusa s ,
 $f(y)$ kifejezés az y adatszerkezet-paraméterrel hívja meg az f függvényt; az f deklarációjákor a megfelelő formális paramétert s adatszerkezetükkel kell megadni,
 $return y;$ utasítás a teljes y adatszerkezetet adja vissza; ehhez az szükséges, hogy a függvény visszatérési értékének típusa s adatszerkezetű legyen.

¹ Mindezek az uniókra is vonatkoznak.

5.15 Alapvető műveletek mutatókkal

Legyen

- x egy t típusú balérték kifejezés,
- pt egy t típusú adatra mutató módosítható balérték kifejezés,
- v pedig egy tetszőleges kifejezés.

Ekkor az

- $\&x$ kifejezés egy x -re irányuló mutatót ad vissza, típusa pedig „egy t típusú adatra irányuló mutató”,
- $pt = \&x$ kifejezés x -re irányítja pt -t, típusa pedig „egy t típusú adatra irányuló mutató”,
- $pt = 0$ kifejezés a nullmutatót rendeli hozzá pt -hez,
- $pt == 0$ kifejezés leellenőrzi, hogy a pt tartalma a nullmutató-e,
- $*pt$ kifejezés visszaadja azt az értéket, amire pt mutat, típusa pedig t ,
- $*pt = v$ kifejezés v értékét betölti a pt által mutatott tárhelyre, típusa pedig t .

Tömbmutatók

Legyen

- a egy t típusú elemeket tartalmazó tömb,
- $pa1$ egy t típusú adatra mutató módosítható balérték kifejezés, mely az a tömb valamelyik elemére mutat,
- $pa2$ egy t típusú adatra mutató balérték kifejezés, mely az a tömb valamelyik elemére (vagy akár az utolsó elemen túlra) mutat,
- v egy tetszőleges kifejezés.
- n pedig egy egész kifejezés.

Ekkor az

- $a, \&a, \&a[0]$ kifejezések egyenértékűek: mindegyik a tömb első elemére irányuló mutatót adja vissza,
- $\&a[n]$ kifejezés a tömb n indexű elemére irányuló mutatót adja vissza; típusa „ t -re irányuló mutató”,
- $*pa1$ kifejezés a -nak azon elemére hivatkozik, amire $pa1$ mutat; típusa t ,
- $*pa1 = v$ kifejezés v értékét betölti a $pa1$ által mutatott tömbelembe; típusa t ,
- $++pa1$ kifejezés a $pa1$ mutatót az a tömb következő elemére állítja át (függetlenül az a elemeinek típusától); típusa „ t -re irányuló mutató”,
- $--pa1$ kifejezés a $pa1$ mutatót az a tömb előző elemére állítja át (függetlenül az a elemeinek típusától); típusa „ t -re irányuló mutató”,
- $++pa1$ kifejezés az inkrementált $pa1$ mutató által hivatkozott a -beli tömb-elemet adja vissza; típusa t ,
- $*pa1++$ kifejezés (az inkrementálás előtti) $pa1$ mutató által hivatkozott a -beli tömbelementet adja vissza; típusa t ,
- $pa1 + n$ kifejezés a $pa1$ mutatót az a tömb n -nel későbbi elemére állítja át; típusa „ t -re irányuló mutató”,
- $pa1 - n$ kifejezés a $pa1$ mutatót az a tömb n -nel korábbi elemére állítja át; típusa „ t -re irányuló mutató”,

$*(\text{pa1} + n) = v$	kifejezés v értékét betölti a $\text{pa1} + n$ által mutatott tömbelembe; típusa t,
$\text{pa1} < \text{pa2}$	kifejezés megvizsgálja, hogy a pa1 az a tömbnek egy korábbi elemére mutat-e, mint a pa2 ; típusa int (bármely más relációs operátor is használható mutatók vizsgálatára),
$\text{pa1} - \text{pa2}$	kifejezés azt a számot adja vissza, ahány tömbelem található a pa1 és a pa2 által mutatott tömbelemelek között (feltéve, hogy pa2 a tömb egy későbbi elemére mutat, mint pa1); típusa int,
$a + n$	kifejezés nem más, mint az a tömb n indexű elemére irányuló mutató, típusa „t-re irányuló mutató”; minden tekintetben egyenértékű az $\&a[n]$ kifejezéssel,
$*(a + n)$	kifejezés az a tömb n indexű elemére hivatkozik, típusa t; minden tekintetben egyenértékű az $a[n]$ kifejezéssel.

A mutatók különbségeként előálló egész szám tényleges típusát a `<stddef.h>` fejlécállományban definiált `ptrdiff_t` adja meg.

Adatszerkezetekre irányuló mutatók²

Legyen

x	egy s adatszerkezetű balérték kifejezés,
ps	egy „s adatszerkezetre irányuló mutató” típusú módosítható balérték kifejezés,
m	egy s adatszerkezetű kifejezés valamely adattagja, melynek típusa t,
v	pedig egy tetszőleges kifejezés.

Ekkor az

$\&x$	kifejezés a teljes x adatszerkezetre rámutat – típusa „s adatszerkezetre irányuló mutató”,
$ps = \&x$	kifejezés a ps mutatót ráállítja x-re – típusa „s adatszerkezetre irányuló mutató”,
$ps->m$	kifejezés a ps által mutatott adatszerkezet m adattagjára utal, típusa t,
$*(ps) . m$	kifejezés szintén az előző sorban megadott adattagra utal – ez a je-lölésmód minden tekintetben egyenértékű a $ps->m$ formával.
$ps->m = v$	kifejezés v értékét eltárolja a ps által mutatott adatszerkezet m adattag-jában; típusa t.

5.16 Összetett betűkonstansok (compound literals)

Az összetett betűkonstans jelölésmód egy zárójelben megadott típusnevet jelent, melyet egy inicializációs lista követ. Ez egy megadott típusú, de névtelen értéket hoz létre, amelynek hatóköre a létrehozás utasításblokkjára korlátozódik. Ha pedig a kifejezés minden utasításblokkon kívül áll, akkor az eredmény hatóköre globális lesz – ez esetben az inicializációs kifejezéseknek konstansoknak kell lenniük.

² Mindezek az uniókra is vonatkoznak.

Tekintsük például a

```
(struct point) { .x = 0, .y = 0 }
```

kifejezést, amely egy olyan point adatszerkezetet ad vissza, amelyben a megadott értékek (0;0) találhatóak. Ez értékül adható egy másik (ugyanilyen típusú) adatszerkezetnek:

```
origin = (struct point) { .x = 0, .y = 0 };
```

Hasonlóan paraméterül adható egy point adatszerkezetet váró függvénynek is:

```
moveToPoint ((struct point) { .x = 0, .y = 0 });
```

Nemcsak adatszerkezetek definiálhatóak így, hanem akár mutatók is. Legyen az intPtr egy egészre irányuló mutató. Ilyenkor a program bármely részén elhelyezhető

```
intPtr = (int [100]) {[0] = 1, [50] = 50, [99] = 99 };
```

utasítás egy (száz egészből álló) tömbre állítja be az intPtr-t, és ezzel egy időben a megadott értékekre inicializálja is a tömb három elemét.

Ha nem adunk meg tömbméretet, akkor ezt az inicializációs értékek alapján határozza meg a fordítóprogram.

5.17 Alapvető adattípusok átalakítása

A C nyelv adott szabályok szerint alakítja át az aritmetikai kifejezésekben szereplő operandusokat. Ezt *szokásos aritmetikai átalakításnak* hívjuk.

1. Ha bármelyik operandus long double, akkor a többi operandus és az eredmény is long double lesz.
2. Ha bármelyik operandus double, akkor a többi operandus és az eredmény is double lesz.
3. Ha bármelyik operandus float, akkor a többi operandus és az eredmény is float lesz.
4. Ha bármelyik operandus _Bool, char, short int, bitmező vagy felsorolt típus, akkor – amennyiben egy int típus maradéktalanul lefedi az igényelt értéktartományt – az operandus int-té alakul, ellenkező esetben pedig unsigned int típusúvá.
Ha az operandusok azonos típusúak, akkor ilyen típusú lesz az eredmény is.
5. Ha minden operandus előjeles vagy mindkettő előjel nélküli, akkor a kisebb egész típus átalakul a nagyobb egész típusára, és ez lesz az eredmény típusa is.
6. Ha az előjel nélküli operandus méretét tekintve nagyobb vagy egyenlő az előjeles operandusnál, akkor az előjeles operandus alakul át az előjel nélküli operandus típusává.

7. Ha az előjeles operandus ábrázolni tudja az előjel nélküli operandus minden elközelhető értékét, akkor az utóbbi alakul át az előbbivé (előjel nélküli az előjelessé), és előjeles típusú lesz az eredmény is.
8. Ha a vizsgálat ehhez a ponthoz ér, akkor minden operandus (az előjeles típusnak megfelelő) előjel nélküli típussá alakul.

A 4. lépést hívtuk korábban *egész átalakításnak*.

Az operandusok átalakítása a legtöbb programozási helyzetben megfelelő, azonban a következőket jó szem előtt tartani:

1. A karakteres változók egésszé alakításakor bizonyos számítógépeken előjel-kiterjesztés történik, ha csak nem definiáljuk a karakteres változót előjel nélkülivé.
2. Egy előjeles egész hosszabb egésszé alakításakor az előjel kitolódik a baloldali bitre (és ennek megfelelően alakul többi „új” baloldali bit is); egy előjel nélküli egész hosszabb egésszé alakításakor viszont a baloldali bitek nullákkal telnek föl.
3. Bármilyen érték `_Bool` típusúvá alakítása 0-t ad, ha az érték nulla, egyébként pedig 1-et.
4. Egy hosszabb egész típus rövidebbé alakítása az egész szám baloldali részének csonkításával történik.
5. Egy lebegőpontos érték egésszé alakítása a törtrész elhagyásával jár együtt.
Ha az egész nem elég nagy a megfelelő átalakított lebegőpontos érték tárolásához, akkor nem definiált az eredmény. Akkor sem, amikor egy negatív lebegőpontos számot előjel nélküli egésszé próbálunk meg átalakítani.
6. Egy hosszabb lebegőpontos érték rövidebbé alakításakor nem garantált a kerekítés, ha csonkításra van szükség.

6.0 Tárolási osztályok és a hatókör

A *tárolási osztály* kifejezés arra utal, hogy a fordítóprogram mekkora memóriát foglal le az egyes változótípusok deklarációjakor, és hogy egy adott függvény definíciója mekkora hatókörrel rendelkezik. A tárolási osztályok a következők: `auto`, `static`, `extern` és `register`. A tárolási osztályt nem kötelező megadni a deklarációkor; ilyenkor az alapértelmezett tárolási osztály kerül felhasználásra, melyet hamarosan részletezünk.

A *hatókör* kifejezéssel jellemizzük egy adott azonosító jelentésének érvényességét egy programban. Ha egy azonosítót minden függvényen és utasításblokkon kívül definiálunk (a továbbiakban ezeket együttesen BLOKKnak rövidítjük), akkor ezt az azonosítót az adott állományon belül bárhol használhatjuk. Egy BLOKKban definiált azonosító lokális a BLOKKra nézve, és felülírja az esetlegesen létező „kívül” definiált azonos nevű azonosító értékét. A címkék és formális paraméterek egy BLOKKon belül mindenütt látszanak. A címkéknek, adatszerkezeteknek és ezek adattagjainak, az unióknak és ezek adattagjainak, valamint a felsorolt típusok neveinek nem kell különböznie egymástól vagy a válto-

zók és függvények neveitől. A felsorolt típusok azonosítóinak azonban az adott hatókörön belül különbözniük kell a többi változónévtől és más felsorolt változók azonosítóneveitől.

6.1 Függvények

Ha egy függvény definiálásakor tárolási osztályt is megadunk, akkor az csak static vagy extern lehet. A statikusként deklarált függvények csak az adott állományon belül érhetők el. Az extern-ként (*kiülsőként*) vagy tárolási osztály nélkül megadott függvények más fájlok ból is meghívhatóak.

6.2 Változók

Az A.6 Táblázat összefoglalja a változók deklarációjakor megadható tárolási osztályokat, valamint a hozzájuk tartozó hatókört és az inicializáció módját.

A.6 Táblázat • Változók: tárolási osztályok, hatókörök és az inicializáció összefoglalása

Ha a tárolási osztály...	és a változó deklarációja...	akkor a változó látható...	és inicializálható...	Megjegyzések
static	minden BLOKKon kívül BLOKKon belül	az állományon belül bárhol az adott BLOKKon belül	csak konstans kifejezéssel	A változók csak egyszer kapnak kezdőértéket, mégpedig a program futásának kezdetén; az értékek csak az adott BLOKKon belül maradnak meg; alapértelmezett értékük 0.
extern	minden BLOKKon kívül BLOKKon belül	az állományon belül bárhol az adott BLOKKon belül	csak konstans kifejezéssel	Az extern kulcsszót a forrásfájljaink egyik definíciós helyén el kell hagynunk (így az deklarációként jelenik meg). Ha egyszer sem hagyjuk el, akkor viszont kezdőértéket kell adni a változónak.

Ha a tárolási osztály...	és a változó deklarációja...	akkor a változó látható...	és inicializálható...	Megjegyzések
auto	BLOKKon belül	az adott BLOKKon belül	bármilyen érvényes kifejezéssel	A változó minden annyiszor inicializálódik, amikor a vezér- lés belép a BLOKKba; nincs alapértel- mezett értéke.
register	BLOKKon belül	az adott BLOKKon belül	bármilyen érvényes kifejezéssel	Nincs garancia arra, hogy tény- legesen egy re- giszterbe kerül a változó tartal- ma; az ilyen deklarációra használható vál- tozók típusai gépfüggőek; nem lehet rá „címe” operá- tort használni; mindannyiszor inicializálódik, amikor a vezér- lés belép a BLOKKba; nincs alapértel- mezett értéke.
nem definiált	minden BLOKKon kívül	az állományon belül bárhol, vagy más fáj- lokból is, ha azok tartal- mazzák a megfelelő deklarációkat	csak konstans kifejezéssel	Ez a deklaráció csak egy helyen jelenhet meg; a változó a program végrehajtásának kezdetén inicia- lizálódik; alap- értelmezett értéke 0.
	BLOKKon belül	(mint az auto-nál)	(mint az auto-nál)	Az auto-val egyezik meg az alapértelmezett értéke

7.0 Függvények

Ebben a szakaszban a függvények szintaxisáról és kezeléséről lesz szó.

7.1 A függvények definíciója

A függvénydeklaráció általános formája a következő:

```
visszatérésiTípus név ( típus1 param1, típus2 param2, ... )
{
    változóDeklarációk
    programUtasítás
    programUtasítás
    ...
    return kifejezés;
}
```

Így definiálhatjuk a *név* nevű függvényt, amely *visszatérésiTípus* típusú értéket ad vissza, és amelynek *param1*, *param2*, ... a formális paraméterei, melyek típusai rendre *típus1*, *típus2* ... stb.

A helyi (*lokális*) változókat általában a függvény elején szoktuk deklární, de ez nem követelmény. Bárhol dekláráthatóak, de csak a deklaráció után lehet őket használni az adott függvényben.

Ha egy függvénynek nincs visszatérési értéke, akkor a *visszatérésiTípus*-t *void*-ként kell megadni.

Ha a zárójelben csak a *void* kulcsszót adjuk meg, akkor a függvény nem fogad el paramétert. Ha utolsó (vagy egyetlen) paraméterként ... (konkrétan három pont) áll, akkor a függvény változó számú paramétert tud fogadni:

```
int printf (char *format, ...)
{
    ...
}
```

A formális paraméterlistában megadott egydimenziós tömböknél nem kell megadni a tömb elemszámát. A többdimenziós tömbök esetében viszont az első dimenzió kivételével meg kell adni a méreteket.

A *return* utasítással kapcsolatban a 8.9 szakasz nyújt bővebb felvilágosítást.

A fordítóprogram optimalizációs törekvéseit segíthetjük (indokolt esetben) a függvény elő írt inline kulcsszóval. Vannak fordítóprogramok, melyek ilyenkor a függvény teljes kódját beillesztik az adott helyre (függvényhívás helyett), így nagyobb, de gyorsabban futó program jön létre. Példaként álljon itt egy minimum-számító függvény:

```
inline int min (int a, int b)
{
    return ( a < b ? a : b);
}
```

7.2 Függvényhívások

Egy függvényt általában így hívhatunk meg:

```
név (param1, param2, ...)
```

Ekkor a *param1*, *param2*... paraméterek átadása mellett a vezérlés a *név* nevű függvényre kerül. Előfordulhat az is, hogy nincsenek paraméterek megadva: ezt azzal jelezzük, hogy csak egy üres zárójelpár áll a függvény neve után: `initialize ()`.

Ha olyan függvényt hívunk meg, melynek definíciójára csak később (vagy akár egy másik állományban) kerül sor, akkor érdemes betenni a programba egy függvényprototípus-deklarációt, mégpedig a következő formában:

```
visszatérésiTípus név (típus1 param1, típus2 param2, ...);
```

Ezzel tájékoztathatjuk a fordítóprogramot a függvény által várt paraméterekről és a visszatérési érték típusáról. Példaképpen tekintsük a következő deklarációt, amely a power függvényt úgy adja meg, mint ami két paramétert vár: egy double és egy int típusú, és egy long double értéket ad vissza (egy szám hatványát):

```
long double power (double x, int n);
```

A zárójelbeli paraméterek neveinek nincs jelentősége, akár el is hagyhatóak; azaz így is megfogalmazható az előbbi deklaráció:

```
long double power (double, int);
```

Ha a fordítóprogram már korábban találkozott a függvény definíciójával vagy a prototípus-deklarációval, akkor a függvényhíváskor minden (erre alkalmas) paraméter átalakul olyan típussá, melyet a függvény feltehetőleg vár.

Ha sem a definíció, sem a prototípus-deklaráció nem áll a fordítóprogram rendelkezésére az adott pillanatban, akkor azt tételezi fel, hogy a függvény int típusú értéket ad vissza. Emellett minden float típusú paramétert double típusúvá alakít, és egész átalakítást hajt végre valamennyi egész paraméteren (lásd az 5.17 szakaszt). Az egyéb típusú paramétek átalakítás nélkül adódnak át a függvénynek.

A változó számú paramétert váró függvényeket ilyenként kell deklárnai. A fordítóprogramnak ugyanis szabadságában áll azt feltételezni, hogy a függvény annyi paramétert vár, amennyi a hívásban ténylegesen szerepel.

A void visszatérési értékkel deklarált függvényeknél a fordítóprogram hibát jelez minden azoknál a függvényhívásoknál, amelyek megkísérelnek visszatérési értéket kinyerni a függvényből.

A függvények paraméterei minden érték szerint kerülnek átadásra, így az értékük megváltozása csak a függvény hatókörében marad meg, utána elvész. Ha mutatót kap meg egy függvény, akkor az általa hivatkozott érték természetesen maradandóan megváltoztatható; ekkor sem marad meg azonban magának a mutatónak az (esetleges) megváltozása a függvényhívás után.

7.3 Függvénymutatók

Ha egy függvény neve után nem adunk meg zárójelpárt, akkor a fordítóprogram ezt mutatóként kezeli, amely az adott függvényre irányul. A „címe” operátor függvénynévre is alkalmazható – ez a megfelelő mutatót adja vissza.

Legyen fp egy függvénymutató. Ilyenkor a megfelelő függvény kétféleképpen is meghívható:

fp ()

vagy

(*fp) ()

Ha a függvénynek vannak paraméterei, akkor azok a zárójelben megadhatóak.

8.0 Utasítások

Utasításnak tekintünk minden érvényes kifejezést (melyek általában értékkedások vagy függvényhívások), amelyeket pontosvessző zár le, vagy azokat, amelyek a következő szakaszban kerülnek tárgyalásra. *Címke* előzhet meg minden utasítást: ez egy azonosítónévből és egy közvetlenül utána írt kettőspontból áll (lásd a 8.6 szakaszt).

8.1 Összetett utasítások

A kapcsos zárójelben szereplő utasításokat *összetett utasításnak* vagy *utasításblokknak* nevezzük. Bárhol állhat ilyen, ahol egyszerű utasítás is szerepelhet. Az utasításblokkoknak lehetnek saját változóik (melyek az adott utasításblokkban lettek deklarálva) – ezek felülírják az egyébként esetlegesen létező, utasításblokkon kívül definiált azonos nevű változók értékét. Hatókörük csak a deklarációt tartalmazó utasításblokkra terjed ki.

8.2 A break utasítás

A break utasítás általános alakja a következő:

```
break;
```

Ha egy for, while, do vagy switch utasításon belül kerül sor egy break utasításra, akkor az adott utasítás végrehajtása befejeződik; a vezérlés azonnal a ciklust vagy switch-et követő utasításra kerül át.

8.3 A continue utasítás

A continue utasítás általános alakja a következő:

```
continue;
```

Ha egy cikluson belül kerül sor egy continue utasításra, akkor az adott ciklusmagon belül a continue-t követő többi utasítás végrehajtása elmarad, de a ciklus futása normál módon folytatódik.

8.4 A do utasítás

A do utasítás általános alakja a következő:

```
continue;
```

Ha egy cikluson belül kerül sor egy continue utasításra, akkor az adott

```
do
    ciklusmag_utasítás
    while ( vizsgálat );
```

A ciklusmag_utasítás mindaddig ismételten lefut, amíg a vizsgálat nullától eltérő eredményt ad. Figyeljünk arra, hogy a vizsgálat csak a ciklusmag_utasítás lefutása után történik meg, így ez legalább egyszer mindenkorban végrehajtódik.

8.5 A for utasítás

A for utasítás általános alakja a következő:

```
for ( inicializáció; vizsgálat; léptetés )  
    ciklusmag_utasítás
```

A zárójelben három kifejezés áll. Az *inicializáció* kifejezése csak egyszer fut le, a ciklus kezdetekor. Ezután értékelődik ki a *vizsgálat* kifejezés. Ha ennek eredménye nem nulla, akkor a *ciklusmag_utasítás* hajtódik végre, majd kiértékelődik a *léptetés* kifejezése. A *ciklusmag_utasítás*, majd a *léptetés* végrehajtása mindaddig megismétlődik, amíg a *vizsgálat* eredménye nullától különböző. Ne feledjük, hogy a *vizsgálat* mindenkor a *ciklusmag_utasítás* előtt zajlik le, így előfordulhat, hogy a *ciklusmag_utasítás* egyszer sem hajtódik végre (amennyiben a *vizsgálat* rögtön a ciklus kezdetén 0 értéket ad).

A for ciklus helyi változói az *inicializáció*-ban deklarálhatóak. Ezen változók hatóköre maga a for ciklus. Például a

```
for ( int i = 0; i < 100; ++i )  
    ...
```

ciklusban deklaráljuk az *i* változót, és 0 kezdőértéket adunk neki a ciklus kezdetén. A változó értéke elérhető a ciklusmag bármely utasításából, de a ciklus lefutása után már nem létezik.

8.6 A goto utasítás

A goto utasítás általános alakja a következő:

```
goto azonosító;
```

A goto utasítás hatására a vezérlés azonnal az *azonosító*-val jellemzett címkére kerül. A címkével ellátott utasításnak ugyanabban a függvényben kell állnia, ahol a rá hivatkozó goto utasítás is áll.

8.7 Az if utasítás

Az if utasítás egyik általános alakja a következő:

```
if (kifejezés)  
    utasítás
```

Ha a *kifejezés* kiértékelése nullától eltérő eredményt ad, akkor lefut az *utasítás*; egyébként pedig nem fut le.

Az `if` utasítás egy másik általános alakja a következő:

```
if ( kifejezés )
    utasítás1
else
    utasítás2
```

Ha a `kifejezés` nem nulla, akkor az `utasítás1` fut le, egyébként pedig az `utasítás2`. Ha az `utasítás2` egy újabb `if` utasítás, akkor egy `if - else if` lánc keletkezik:

```
if ( kifejezés1 )
    utasítás1
else if ( kifejezés2 )
    utasítás2
    ...
else
    utasítás_n
```

Az `else` ág mindenkorában a legutolsó (`else`-et nem tartalmazó) `if` utasításhoz tartozónan értelmeződik. Zárójelekkel megváltoztatható ez az alapértelmezett összekapcsolás.

8.8 A null utasítás

Az `null` utasítás általános alakja a következő:

;

A `null` utasítás véghajtásának nincs semmilyen hatása. Célja pusztán annyi, hogy eleget tehessünk a C nyelv szabályainak például egy `for`, `do` vagy `while` ciklus megírásakor. A következő utasítás például átmásolja a `from` által mutatott karakterláncot a `to` által mutatott memóriaterületre:

```
while ( *to++ = *from++ )
    ;
```

Itt a `null` utasítás csak arra kell, hogy legyen egy (akár semmit sem végző) ciklusmagja a `while` ciklusnak – ezt ugyanis megköveteli a fordítóprogram. (A hasznos folyamat ugyanis már a ciklusfejben lezajlik.)

8.9 A return utasítás

A `return` utasítás egyik általános alakja a következő:

`return;`

A `return` utasítás hatására a vezérlés azonnal visszakerül a hívó függvényhez. Ez a forma azonban csak olyan függvényeknél használható, amelyeknek nincs visszatérési értékük.

Ha egy függvény úgy ér el saját utasításblokkjának a végéhez, hogy nem kerül elő `return` utasítás, akkor ezt úgy tekinti a fordítóprogram, mintha egy `return;` utasítás lenne a függvénytörzs utolsó utasítása. Azaz ilyenkor nem lesz visszatérési értéke a függvénynek.

A `return` utasítás egy másik általános alakja a következő:

```
return kifejezés;
```

Ilyenkor a *kifejezés* értéke adódik vissza a hívó függvényhez. Ha ennek típusa nem egyezik meg azzal, amit a függvény deklarációjakor megadtunk, akkor előzőleg automatikusan a kívánt típusúvá alakul.

8.10 A `switch` utasítás

A `switch` utasítás általános alakja a következő:

```
switch ( kifejezés )
{
    case konstans_1:
        utasítás
        utasítás
        ...
        break;
    case konstans_2:
        utasítás
        utasítás
        ...
        break;
    ...
    case konstans_N:
        utasítás
        utasítás
        ...
        break;
    default:
        utasítás
        utasítás
        ...
        break;
}
```

A zárójelben álló *kifejezés* sorban egymás után összehasonlítódik a `konstans_1`, `konstans_2`... `konstans_N` értékekkel, melyek mindenike egy `konstans` kifejezés.

Ha valamelyik case ágban egyezés található a zárójeles *ki fejezés* és a konstans érték között, akkor az oda tartozó utasítások lefutnak. Ha egyik case ágban sem volt egyezés, akkor a default ág utasításai futnak le (ha van ilyen ág). Ha nincs ilyen ág, akkor nem fut le egyetlen utasítás sem a switch végrehajtásakor.

A *kifejezés* eredményének egész típusúnak kell lennie, és nem lehet két azonos feltételezett megfogalmazó case ág. Az egyes ágak végéről elhagyható a break utasítás, de ennek hatására a vezérlés a következő ágon folytatódik.

8.11 A while utasítás

A while utasítás általános alakja a következő:

```
while ( kifejezés )
       ciklusmag
```

A *ciklusmag* mindaddig lefut, amíg a *kifejezés* kiértékelése „igazat”, azaz nullától eltérő értéket ad. Figyeljünk arra, hogy a *kifejezés* kiértékelése a *ciklusmag* végrehajtása előtt történik meg, előfordulhat, hogy a *ciklusmag* egyszer sem fut le.

9.0 Az előfeldolgozó

Az előfeldolgozó még azelőtt elemzi (és alakítja) a forráskódot, mielőtt a fordítóprogram ténylegesen elkezdené a fordítást. A következőt végzi el:

1. A hármas karaktereket (*trigraph sequences*) helyettesíti a velük egyenértékű egyszerű karakterekkel (lásd a 9.1 szakaszt)
2. A backslash karakterrel (\) végződő sorokat összefűzi egyetlen sorrá
3. A programokat tokenek (értelmezhető jelek) sorozatává alakítja
4. Eltávolítja a megjegyzésekét, szóközökkel helyettesítve őket
5. Végrehajtja az előfeldolgozói utasításokat (lásd a 9.2 szakaszt) és kibontja a makróhelyettesítéseket

9.1 Hármas karakterek (trigraph sequences)

Léteznek olyan karakterkészletek, melyekben nem minden ASCII karakter szerepel. Emiatt hoztak létre speciális hármas karaktereket, melyek egy-egy adott szimbólumot képviselnek a programkódban. Az előfeldolgozó ezeket az A.7 Táblázatnak megfelelően átalakítja.

A.7 Táblázat • Hármas karakterek

Hármas karakter	Jelentése
??=	#
??([
??)]
??<	{
??>	}
??/	\
??^	^
??!	
??-	~

9.2 Az előfeldolgozó utasításai

Az előfeldolgozónak szánt utasításokat, „direktívákat” a sor első nem térköz karaktereként megadott # karakterrel jelezhetjük. A # jelet követhetik szóközök vagy tabulátorok.

9.2.1 A #define utasítás

A #define utasítás általános formája a következő:

```
#define név szöveg
```

A #define utasítás a név azonosítót összekapcsolja azzal a szöveggel, ami a név utáni térköz után áll, egészen a sor végéig. Ahol csak (karakterlánc-konstansokon kívül) megjelenik a programban a név, ott az előfeldolgozó ezt kicseréli a definiált értékre.

A #define utasítás a következőképpen is használható makró definiálására:

```
#define név(param_1, param_2, ..., param_n) szöveg
```

A név nevű makró a param_1, param_2, ..., param_n azonosítókkal megadott paramétereket várja. Ahol csak (karakterlánc-konstansokon kívül) megjelenik a programban a név és a megfelelő paraméterlista, ott az előfeldolgozó ezeket kicseréli a definiált szövegre, behelyettesítve a paramétereket a megfelelő helyeken a szövegen belül.

Ha a makrót változó számú paraméter elfogadására szeretnénk felkészíteni, akkor ezt (a paraméterlistában megadott) három ponttal jelezhetjük az előfeldolgozó számára. A három pont helyén megjelenő paraméterekre __VA_ARGS__-ként hivatkozhatunk. Tekintsük például a következő myPrintf makrót, amely egy bevezető karakterlánc után kiírja a(z előre nem definiált számú) paraméter-értékeket:

```
#define myPrintf(...) printf ("DEBUG: " __VA_ARGS__ );
```

Ezután a myPrintf makrót meghívhatjuk így:

```
myPrintf ("Hello világ!\n");
```

vagy akár így is:

```
myPrintf ("i = %i, j = %i\n", i, j);
```

Ha egy definíció csak több sorban fér el, akkor a folytatandó sorokat backslash karakterrel (\) kell lezárni. Egy név definíciója után az bárhol használható az állományon belül.

A paramétert is használó #define utasításokon belül alkalmazható a # operátor. Ha a makródefinícióban egy # jelet helyezünk el a paraméter neve elő, akkor az előfeldolgozó idézőjellel zára közre a paraméter értékét, azaz konstans karakterláncot hoz létre a paraméterből. Legyen például a printint definíciója a következő:

```
#define printint(x) printf (# x " = %d\n", x)
```

Ha ezt a

```
printint (count);
```

utasítással hívjuk meg, akkor ezt az előfeldolgozó így bontja ki:

```
printf ("count" " = %i\n", count);
```

Ez aztán a szomszédos karakterláncok egybefűzése miatt így fog kinézni:

```
printf ("count = %i\n", count);
```

Eközben az előfeldolgozó \ karakterrel védi le a karakterláncban esetlegesen szereplő idézőjeleket (") és backslash karaktereket (\). Így tehát a

```
#define str(x) #x
```

után az

```
str (A "\t" karakterlánc tabuláltort tartalmaz)
```

makróhívás így bomlik ki az előfeldolgozás után:

```
"A \"\t\" karakterlánc tabuláltort tartalmaz"
```

A paramétert elfogadó `#define` utasításokban két jel összekapcsolására használható a `##` operátor. Előtte vagy utána egy paraméternévnek kell állnia. Az előfeldolgozó behelyettesíti a kapott paramétert, majd „összeragasztja” a `##` operátor túloldalán levő karakterláncot, egyetlen jelet (tokent) hozva létre a két szimbólumból. Ha így definiáljuk a `printx` makrót:

```
#define printx(n) printf ("%i\n", x ## n );
```

akkor a

```
printx(5)
```

a következőképpen bomlik ki az előfeldolgozás után:

```
printf("%i\n", x5);
```

Ha pedig így szól a `printx` makró definíciója:

```
#define printx(n) printf ("x" # n " = %i\n", x ## n );
```

akkor a

```
printx(10)
```

így helyettesítődik be az előfeldolgozás során végrehajtásra kerülő összefűzés után:

```
printf("x10 = %i\n", x10);
```

Nem szükséges szóközöt írni a `#` és a `##` operátor köré (bár úgy olvashatóbb a kód).

9.2.2 Az `#error` utasítás

Az `#error` utasítás általános formája a következő:

```
#error szöveg
```

```
...
```

A megadott *szöveget* hibaüzenetként jeleníti meg az előfeldolgozó.

9.2.3 Az `#if` utasítás

Az `#if` utasítás egyik általános formája a következő:

```
#if konstans_kifejezés
...
#endif
```

Ha a *konstans_kifejezés* kiértékelésekor nem nulla az eredmény, akkor az #endif-ig terjedő sorok végrehajtódnak; egyébként pedig kimaradnak, és sem az előfeldolgozó, sem a fordítóprogram nem vesz róluk tudomást.

Az #if utasítás egy másik általános formája:

```
#if konstans_kifejezés_1
  ...
# elif konstans_kifejezés_2
  ...
# elif konstans_kifejezés_n
  ...
# else
  ...
#endif
```

Ha a *konstans_kifejezés_1* nem nulla, akkor a következő #elif-ig terjedő programsorok feldolgozásra kerülnek, ám az #endif-ig terjedőeket átugorja az előfeldolgozó. Egyébként pedig, ha a *konstans_kifejezés_2* nem nulla, akkor az ezt követő #elif-ig dolgozza fel az előfeldolgozó a programsorokat, majd az #endif-ig terjedőeket kihagyja. Ha minden konstans kifejezés nulla, akkor az (esetlegesen létező) #else utáni sorokat dolgozza fel az előfeldolgozó.

A konstans kifejezésekben használható a defined operátor. Az

```
#if defined (DEBUG)
  ...
#endif
```

kódrészlet hatására csak akkor kerül feldolgozásra az #if és #endif közti rész, ha a DEBUG azonosító korábban definiálásra került (lásd a 9.2.4 szakaszt). A zárójel elhagyható az azonosító mellől, azaz így is írható a feltételes kifejezés:

```
#if defined DEBUG
```

9.2.4 Az #ifdef utasítás

Az #ifdef utasítás általános formája a következő:

```
#ifdef azonosító
  ...
#endif
```

9.2.7 A #line utasítás

A #line utasítás általános formája a következő:

```
#line sor "állománynév"
```

Ez az utasítás arra utasítja az előfeldolgozót, hogy a további sorokat tekintse úgy, mintha az *állománynév*-beli *sor* sorszámtól folytatódna a program. Ha nincs megadva *állománynév*, akkor a legutolsó #line utasítás alapján határozza meg az előfeldolgozó a „fejben tartandó” fájlnévet, és ha ez sincs, akkor az aktuálisan feldolgozott forrásállomány nevét használja.

A #line utasítás leginkább a fordítóprogram által adott hibajelzésekhez tartozó fájlnév- és sorszám-információ manipulálására szolgál.

9.2.8 A #pragma utasítás

A #pragma utasítás megadásának általános formája a következő:

```
#pragma szöveg
```

Ez az utasítás arra utasítja az előfeldolgozót, hogy (implementációfüggő módon) végrehajtson bizonyos műveleteket. Például a

```
#pragma loop_opt (on)
```

egyes fordítóprogramoknál speciális ciklus-optimalizációt hoz létre. Ha az aktuális fordítóprogram nem ismeri fel a *loop_opt* direktívát, akkor az adott #pragma utasítást figyelmen kívül hagyja.

A #pragma után írt STDC kulcsszó speciális jelentést hordoz. Jelenleg a következő „kapcsolók” használhatóak a #pragma STDC után: FP_CONTRACT, FENV_ACCESS és CX_LIMITED_RANGE.

9.2.9 Az #undef utasítás

A #undef utasítás általános formája a következő:

```
#undef azonosító
```

A megadott *azonosító*-t innentől kezdve definiálatlannak tekinti az előfeldolgozó. Az ezt követő #ifdef és #ifndef utasítások úgy fognak viselkedni, mintha a megadott *azonosító* soha nem lett volna definiálva.

9.2.10 A # utasítás

A # az üres előfeldolgozói utasítás – nincs hatása.

9.3 Előre definiált azonosítók

Az előfeldolgozó által használt azonosítókat az A.8 Táblázatban láthatják.

A.8 Táblázat • Előre definiált azonosítók

Azonosító	Jelentése
<code>_LINE_</code>	Az éppen fordítás alatt álló sor sorszáma
<code>_FILE_</code>	Az éppen fordítás alatt álló forrásállomány neve
<code>_DATE_</code>	A fordítás dátuma, "hh nn éééé" formátumban
<code>_TIME_</code>	A fordítás időpontja, "hh:mm:ss" formátumban
<code>_STDC_</code>	1 az értéke, ha a fordítóprogram megfelel az ANSI szabványnak, és 0, ha nem.
<code>_STDC_HOSTED_</code>	1 az értéke, ha a fordítóprogram implementációja elfogadott („hosted”), és 0, ha nem.
<code>_STDC_VERSION_</code>	199901L-ként van definiálva (<i>STandard C VERZIÓ</i>)

B

A szabványos C programkönyvtár

A szabványos C programkönyvtár függvények széles választékát kínálja, melyek jól használhatóak a különféle programokban. Ez a rész nem sorolja fel az összes függvényt, hanem csak a leggyakrabban használtakat. Az elérhető függvények teljes felsorolása is elérhető a fordítóprogramhoz mellékelve, illetve az E függelékben („Források”) megjelölt helyeken.

A függelékben nem beszélünk a dátum- és időkezelő függvényekkel (mint a `time`, `cftime` és `localtime`), a távoli ugrások meghosszabbításával (`setjmp` és `longjmp`), diagnosztikai információk előállításával (`assert`), változó számú paraméter kezelésével (`va_list`, `va_start`, `va_arg` és `va_end`), jelzések (`szignálok`) kezelésével (`signal` és `raise`), a nemzeti nyelv használatát érintő kérdésekkel (ennek elemei a `<locale.h>` fejlécállományban vannak definiálva), sem pedig a széles karaktereket használó karakterláncokkal.

Szabványos fejlécállományok

Ebben a szakaszban a következő öt szabványos fejlécállomány tartalmáról lesz szó: `<stddef.h>`, `<stdbool.h>`, `<limits.h>`, `<float.h>` és `<stdinit.h>`.

`<stddef.h>`

Ebben a fejlécállományban kerül definiálásra néhány szabványos változó, például a következők:

Definíció	Jelentése
<code>NULL</code>	Nullmutató konstans
<code>offsetof</code> (<code>adatszerkezet</code> , <code>adattag</code>)	Az <code>adattag</code> távolsága az <code>adatszerkezet</code> kezdetétől bájtokban; az eredmény típusa <code>size_t</code>
<code>ptrdiff_t</code>	Két mutató különbségeként előálló egész szám típusa

Definíció	Jelentése
<code>size_t</code>	A <code>sizeof</code> operátor által visszaadott egész szám típusa
<code>wchar_t</code>	Annak az egész számnak a típusa, amely egy széles karaktert tárolni tud (lásd az A függeléket, „A C nyelv összefoglalása”).

<limits.h>

Ebben a fejlécállományban kerül definiálásra néhány (implementációsfüggő) küszöbérték, mely a karakteres és egész típusú adatokra vonatkozik. Ezen értékek minimumát garantálja az ANSI C szabvány, melyek zárójelben fel is vannak tüntetve a sorok végén.

Definíció	Jelentése
<code>CHAR_BIT</code>	A <code>char</code> típus bitjeinek száma (8)
<code>CHAR_MAX</code>	A <code>char</code> típusban tárolható objektum maximális értéke (előjel-kiterjesztés esetén 127, egyébként 255)
<code>CHAR_MIN</code>	A <code>char</code> típusban tárolható objektum minimális értéke (előjel-kiterjesztés esetén -127, egyébként 0)
<code>SCHAR_MAX</code>	A <code>signed char</code> típusban tárolható objektum maximális értéke (127)
<code>SCHAR_MIN</code>	A <code>signed char</code> típusban tárolható objektum minimális értéke (-127)
<code>UCHAR_MAX</code>	Az <code>unsigned char</code> típusban tárolható objektum maximális értéke (255)
<code>SHRT_MAX</code>	Az <code>short int</code> típusban tárolható objektum maximális értéke (32767)
<code>SHRT_MIN</code>	Az <code>short int</code> típusban tárolható objektum minimális értéke (-32767)
<code>USHRT_MAX</code>	Az <code>unsigned short int</code> típusban tárolható objektum maximális értéke (65535)
<code>INT_MAX</code>	Az <code>int</code> típusban tárolható objektum maximális értéke (32767)
<code>INT_MIN</code>	Az <code>int</code> típusban tárolható objektum minimális értéke (-32767)
<code>UINT_MAX</code>	Az <code>unsigned int</code> típusban tárolható objektum maximális értéke (65535)
<code>LONG_MAX</code>	Az <code>long int</code> típusban tárolható objektum maximális értéke (2147483647)
<code>LONG_MIN</code>	Az <code>long int</code> típusban tárolható objektum minimális értéke (-2147483647)

Definíció	Jelentése
ULONG_MAX	Az unsigned long int típusban tárolható objektum maximális értéke (4294967295)
LLONG_MAX	A long long int típusban tárolható objektum maximális értéke (9223372036854775807)
LLONG_MIN	A long long int típusban tárolható objektum minimális értéke (-9223372036854775807)
ULLONG_MAX	Az unsigned long long int típusban tárolható objektum maximális értéke (18446744073709551615)

<stdbool.h>

Ebben a fejlécállományban a Boole-algebra alapelemei, vagyis a (`_Bool` típusból eredő) logikai változók kerülnek definiálásra.

Definíció	Jelentése
<code>bool</code>	helyettesítő név a <code>_Bool</code> alapvető adattípushoz
<code>true</code>	1-ként definiált
<code>false</code>	0-ként definiált

<float.h>

Ebben a fejlécállományban a lebegőpontos aritmetika küszöbértékei kerülnek definiálásra. A garantált minimális méret zárójelben van feltüntetve. Nem minden definiált értéket sorolunk fel.

Definíció	Jelentése
<code>FLOAT_DIG</code>	A <code>float</code> típus pontossági számjegyeinek száma (6)
<code>FLOAT_EPSILON</code>	Az a legkisebb érték, amelyet 1.0-hoz adva az eredmény már nem egyenlő 1.0-val ($1e-5$)
<code>FLOAT_MAX</code>	A <code>float</code> típus legnagyobb használható értéke ($1e+37$)
<code>FLOAT_MAX_EXP</code>	A <code>float</code> típus legnagyobb használható értéke ($1e+37$)
<code>FLOAT_MIN</code>	A normálalakra hozott <code>float</code> típus legkisebb használható értéke ($1e-37$)

Hasonló definíciók állnak rendelkezésre a `double` és a `long double` típusok számára is; a `FLOAT` helyett `DBL`-t kell írni a megfelelő azonosítónévbe a `double` típus esetén, és `LDBL`-t a `long double` típusnál. Például a `DBL_DIG` megadja a `double` típus pontossági számjegyeinek számát, és ugyanígy a `LDBL_DIG` a `long double`-ét.

Jó tudni, hogy a `<fenv.h>` fejlécállomány használható arra, hogy információt és további eszközöket kapunk a lebegőpontos környezet kézben tartásához. Van például egy `fesetround` nevű függvény, amely lehetővé teszi a kerekítés módjának megadását. A `<fenv.h>`-ban van definiálva a `FE_TONEAREST` (*legközelebbihez*) a `FE_UPWARD` (*fel felé*), a `FE_DOWNWARD` (*lefelé*) és a `FE_TOWARDZERO` (*nulla felé*) értéke. Lehetőségünk van törölni, létrehozni vagy kipróbálni lebegőpontos kivételeket a `feclearexcept`, a `feraiseexcept` és a `fetextexcept` függvények segítségével.

<stdint.h>

Ebben a fejlécállományban különféle definíciók és konstansok kerülnek definiálásra, melyeket az egészekkel történő munka közben tudunk használni programunk gépfüggőségeinek csökkentésére. A `typedef int32_t` például felhasználható arra, hogy 32 bites egész típust definiálunk anélkül, hogy tudnánk, hogy az adott gépen (ahol programunkat fordítjuk) éppen melyik egész típus 32 bites. Hasonlóképpen az `int_least32_t` típus használható egy olyan egész típus megnevezésére, amelyik legalább 32 bites. Más típus-definíciók is használhatóak; például kiválaszthatjuk, hogy melyik típushoz tartozik a leggyorsabb egész ábrázolás. További információhoz olvassák el a fordítóprogramhoz tartozó dokumentációt, vagy nézzék meg a szóban forgó `stdint.h` fejlécállományt.

Még néhány fontos információ ehhez a fejlécállományhoz:

Definíció	Jelentése
<code>intptr_t</code>	Egész típus, amiben elfér bármilyen mutató értéke
<code>uintptr_t</code>	Előjel nélküli egész típus, amiben elfér bármilyen mutató értéke
<code>intmax_t</code>	A legnagyobb előjeles egész típus
<code>uintmax_t</code>	A legnagyobb előjel nélküli egész típus

Karakterlánc-kezelő függvények

A következő függvények karaktertömbököt kezelnek. Az eljárások leírásában az `s`, `s1` és `s2` nullvégződésű karaktertömbököt jelentenek, `c` egy `int` típusú szám, `n` pedig egy (`stddef.h`-ban megadott) `size_t` típusú egész szám. Az `strnxxx` eljárásban az `s1` és `s2` karaktertömböknek nem kell nullvégződésűnek lenniük.

Ezeknek a függvényeknek a használatához a program elején be kell emelni a `<string.h>` fejlécállományt:

```
#include <string.h>
```

```
char *strcat (s1, s2)
```

Az *s2* karakterláncot hozzáfűzi az *s1* karakterlánc végéhez, az eredmény végére pedig odilleszt egy null karaktert. A függvény visszatérési értéke *s1*.

```
char *strchr (s, c)
```

Az *s* karakterláncban megkeresi *c* első előfordulását. Ha megvan, akkor a karakterre hivatkozó mutatót adja vissza a függvény, egyébként pedig a nullmutatót.

```
int strcmp (s1, s2)
```

Összehasonlítja az *s1* és *s2* karakterláncokat. A visszatérési érték negatív, ha *s1* a kisebb, nulla, ha *s1* egyenlő *s2*-vel és pozitív, ha *s2* a nagyobb.

```
int strcoll (s1, s2)
```

Olyan, mint az `strcmp`, csak itt az *s1* és az *s2* az adott nyelvi környezetbe tartozó (*locale*) karakterláncokra mutat.

```
char *strcpy (s1, s2)
```

Az *s2* (!) karakterláncot átmásolja az *s1*-be; visszatérési értéke *s1*.

```
char *strerror (n)
```

Megjeleníti az *n*-hez tartozó hibaüzenetet.

```
size_t strcspn (s1, s2)
```

Az *s1* elején lévő olyan maximális rész-karakterlánc méretét adja, amely egyetlen karaktert sem tartalmaz az *s2*-ben megadott karakterekből.

```
size_t strlen (s)
```

Visszaadja az *s* karakterlánc hosszát, nem számítva a lezáró null karaktert.

```
char *strncat (s1, s2, n)
```

Az *s1* végéhez fűzi az *s2* legfeljebb első *n* karakterét (azaz ha *n*-nél kevesebb karakterből áll *s2*, akkor csak magát *s2*-t), és visszatér az egyesített *s1* címével.

```
int strncmp (s1, s2, n)
```

Hasonlít az `strcmp`-hez, de legfeljebb az első *n* karaktert hasonlítja össze a két karakterláncból.

```
char *strncpy (s1, s2, n)
```

Az *s2* karakterlánc legfeljebb *n* karakterét átmásolja az *s1*-be; visszatérési értéke *s1*.

```
char *strrchr (s, c)
```

Az *s* karakterláncban megkeresi *c* utolsó előfordulását. Ha megvan, akkor a karakterre hivatkozó mutatót adja vissza a függvény, egyébként pedig a nullmutatót.

```
char *strupr (s1, s2)
```

Az *s2*-beli karakterek valamelyikét keresi *s1*-ben. Az első előfordulás címét adja vissza; illetve a nullmutatót, ha nincs közös karakter.

```
size_t strspn (s1, s2)
```

Az *s1* elején lévő olyan maximális rész-karakterlánc méretét adja, amely csak az *s2*-ben megadott karaktereket tartalmazza.

```
char *strstr (s1, s2)
```

Az *s2* első *s1*-beli előfordulásának címét adja; illetve nullmutatót, ha *s2* nem található meg *s1*-ben.

```
char *strtok (s1, s2)
```

A függvény *s1*-et olyan szimbólumok (*tokenek*) sorozatának tekinti, amelyeket az *s2*-ben található karakterek határolnak. Az első *strtok* hívás átlépi a bevezető elválasztójeleket, visszatér az *s1*-beli első szimbólum címével, ám ezelőtt a szimbólumot null karakterrel zárja. Az *s1* maradék része további szimbólumokra bontható újabb *strtok* hívásokkal (ekkor *s1* helyén nullmutatót kell átadnunk paraméterként). Ha nem marad több szimbólum, akkor nullmutatót ad vissza a függvény.

```
size_t strxfrm (s1, s2, n)
```

Az *s2* első *n* karakterét alakítja át, és az eredményt betölti *s1*-be. Az átalakítás célja, hogy két (adott nyelvi karakterekből álló) karakterláncot az átalakítás után össze lehessen hasonlítani az *strcmp* függénnyel.

Memóriakezelő függvények

A következő függvények is karaktertömbököt kezelnek. Arra terveztek őket, hogy hatékonyan keressenek a memóriában, és gyorsan át tudjanak másolni adatokat egy memóriarendszertről egy másikba. Használatukhoz a program elején be kell emelni a `<string.h>` fejlécállományt:

```
#include <string.h>
```

Az alábbi függvények leírásában *m1* és *m2* típusa `void *`, *c* egy `int` típusú szám, amelyet némely függvény `unsigned int`-té alakít, *n* pedig egy `size_t` típusú egész szám.

```
void *memchr (m1, c, n)
```

Megkeresi *m1* első *n* karakterében *c* első előfordulását. Ha megtalálja, akkor visszaadja a találat címét, egyébként pedig a nullmutatót adja vissza.

```
void *memcmp (m1, m2, n)
```

Összehasonlítja az *m1* és *m2* első *n* karakterét. Ha a két tömb első *n* eleme megegyezik, akkor nulla a visszatérési érték. Ha nem egyeznek meg, akkor az első eltérést okozó karakterek különbsége lesz a visszatérési érték. Ha az eltérést okozó karakter *m1*-ben kisebb, mint *m2*-ben, akkor negatív értéket, ellenkező esetben pedig pozitív értéket ad vissza a függvény.

```
void *memcpy (m1, m2, n)
```

Az *m2* első *n* karakterét átmásolja az *m1*-be; visszatérési értéke *m1*.

```
void *memmove (m1, m2, n)
```

Olyan, mint a `memcpy`, de akkor is garantáltan működik, ha *m1* és *m2* átfedésben van a memóriában.

```
void *memset (m1, c, n)
```

m1 első *n* karakterét beállítja *c* értékére. Visszatérési értéke *m1*.

A most felsorolt függvények nem érzékenyek a tömbökben esetlegesen előforduló null karakterekre. Nemcsak karaktertömbök kezelésére lehet őket használni, de az fontos, hogy idejében megtörténjen a `void *`-ra irányuló mutató-átalakítás. Ha például *data1* és *data2* száz `int` értéket tároló tömb, akkor a

```
memcpy ((void *) data2, (void *) data1, sizeof (data1));
```

utasítás a *data1* első száz egész értékét átmásolja a *data2*-be.

Karakterkezelő függvények

A következő függvények egyedülálló karaktereket kezelnek. Használatukhoz a program elején be kell emelni a <ctype.h> fejlécállományt:

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

Az alábbi függvények mindegyike egy int (*c*) paramétert vár, és TRUE (azaz nem nulla) értéket ad vissza, ha a feltétel teljesül, egyébként pedig FALSE (azaz nulla) értéket.

Név	Feltétel
isalnum	A paraméter alfanumerikus (azaz betű vagy szám) karakter?
isalpha	A paraméter alfabetikus (azaz betű) karakter?
isblank	A paraméter helyköz (blank) karakter (szóköz vagy tabulátor)?
iscntrl	A paraméter vezérlőkarakter?
isdigit	A paraméter számjegy karakter?
isgraph	A paraméter grafikai karakter (a szóköz kivételével valamelyik nyomtatható karakter)?
islower	A paraméter kisbetű?
isprint	A paraméter nyomtatható karakter (akár szóköz is)?
ispunct	A paraméter központozási karakter (szóköz és alfanumerikus karakterek kivételével valamely karakter)?
isspace	A paraméter térköz karakter (szóköz, újsor, kocsi vissza, függőleges vagy vízszintes tabulátor vagy lapdobás)?
isupper	A paraméter nagybetű?
isxdigit	A paraméter hexadecimális számjegy karakter?

Van két függvény, amely elvégzi a kisbetűs és nagybetűs karakterek közötti átalakítást:

```
int tolower(c)
```

Visszaadja *c* kisbetűs megfelelőjét. Ha *c* nem nagybetű, akkor maga *c* lesz a visszatérési érték.

```
int toupper(c)
```

Visszaadja *c* nagybetűs megfelelőjét. Ha *c* nem kisbetű, akkor maga *c* lesz a visszatérési érték.

Kimeneti/bemeneti függvények

A továbbiakban néhány gyakran használt kimeneti/bemeneti függvénnyről lesz szó, melyek a C programkönyvtárakban találhatóak. Használatukhoz a program elején be kell emelni a `<stdio.h>` fejlécállományt a következő utasítással:

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

Ebben a fejlécállományban találhatóak a kimeneti/bemeneti függvények deklarációi, valamint az EOF, NULL, stdin, stdout, stderr (mind konstans érték) és a FILE azonosítónevek definíciói.

A továbbiakban az `állományNév`, `állományNév1`, `állományNév2`, `elérésiMód` és a `formátum` nevek nullvégződésű karakterláncokra irányuló mutatókat jelentenek, a `buffer` egy karaktertömb mutatója, a `filePtr` egy FILE-ra irányuló mutató, `n` és `méret` pozitív egész `size_t` típusú értékek, `i` és `c` pedig int típusú paraméterek.

```
void clearerr (filePtr)
```

Törli a `filePtr`-rel azonosított állomány „fájl vége” jelzését és hibajelzéseit.

```
int fclose (filePtr)
```

Lezárja a `filePtr`-rel azonosított állományt. Nullát ad vissza, ha a művelet sikeres, és EOF-ot, ha hiba következik be.

```
int feof (filePtr)
```

Ha a `filePtr`-rel azonosított állománynak a végére értünk, akkor nem nulla értéket ad vissza, egyébként pedig nullát.

```
int perror (filePtr)
```

Megvizsgálja a megadott fájl hibaállapotát. Ha történt hiba, akkor nullát ad vissza, egyébként pedig nem nullát.

```
int fflush (filePtr)
```

Kiírja a belső tárolóban levő adatokat a megadott fájlba. Nullát ad vissza, ha sikeres volt a művelet, és EOF értéket, ha hiba következik be.

```
int fgetc (filePtr)
```

Visszaadja a `filePtr`-rel azonosított állomány következő karakterét, vagy EOF-ot, ha teljesül a „fájl vége” feltétel. Vigyázat, a függvény int értéket ad vissza.

```
int fgetpos (filePtr, fpos)
```

Kinyeri a *filePtr*-rel azonosított állomány aktuális írási/olvasási pozícióját, és eltárolja a (<stdio.h>-ban definiált) *fpos_t* típusú, *fpos* által mutatott változóban. Az *fgetpos* nullát ad vissza, ha a művelet sikeres, ellenkező esetben pedig nem nullát. Érdemes megismerni a függvény „beállító pájját”, az *fsetpos* függvényt is.

```
char *fgets (buffer, i, filePtr)
```

A megadott állományból olvas ki karaktereket, mégpedig legfeljebb *i*-1-et (ha ennél rövidebb az aktuális sor hossza, akkor kevesebbet). A kiolvasott karaktereket a *buffer* karaktertömbben tárolja el. Az esetlegesen kiolvasott újsor karakter is eltárolódik a tömbben. Ha a függvény eléri az állomány végét, vagy hiba történik, akkor NULL értéket ad vissza; egyébként pedig a *buffer*-t.

```
FILE *fopen (állományNév, elérésiMód)
```

Megnyitja az *állományNév* nevű fájlt a megadott elérési módban, melyek a következők lehetnek: "r" – olvasásra, "w" – írásra, "a" – hozzáfűzésre meglévő fájlhoz, "r+" – írás/olvasás elkezdésére meglévő fájl kezdetéről, "w+" – írásra/olvasásra (ha már létezik ilyen fájl, akkor a tartalma törlődik), "a+" – meglévő fájl olvasására vagy a végére történő írásra. Ha a megnyitandó fájl nem létezik, akkor ("w", "w+", "a" vagy "a+" elérési mód esetén) létrejön; az "a" vagy "a+" hozzáfűző elérési módban megnyitott állományok tartalma nem íródik felül.

Vannak rendszerek, amelyek megkülönböztetik a bináris állományokat a szöveges állományuktól – ilyenkor b karakterrel kell ellátni az elérési mód jelét, ha meg akarunk nyitni egy bináris fájlt (például "rb").

Ha az *fopen* hívás sikeres, akkor egy FILE-mutatót ad vissza a függvény, amely azonosítja az állományt az elkövetkező kimeneti/bemeneti műveletekhez; egyébként pedig a nullmutatót adja vissza.

```
int fprintf (filePtr, formátum, param1, param2, ..., paramn)
```

A megadott paramétereket a *formátum* karakterláncnak megfelelően kiírja a *filePtr*-rel azonosított állományba. A *formátum* jelző karakterláncban ugyanazokat az értékeket használhatjuk, mint a *printf* függvényben (16. fejezet, „A C kimeneti és bemeneti műveletei”). A kiírt karakterek számát adja vissza a függvény; ha ez az érték negatív, akkor hiba történt a kiíráskor.

```
int fputc (c, filePtr)
```

Az előjel nélküli char típusúvá alakított *c* értékét írja ki a *filePtr*-rel azonosított állományba. Ha az írási művelet sikeres, akkor *c* értékét adja vissza, egyébként pedig az EOF értékét.

```
int fputs (buffer, filePtr)
```

A *buffer* által mutatott karaktertömb értékét írja ki a megadott állományba egészen a *buffer*-beli első null karakterig (azt már nem írja ki). A függvény nem ír automatikusan újsor karaktert a fájlba. Hiba esetén az EOF értékét adja vissza.

```
size_t fread (buffer, méret, n, filePtr)
```

A megadott állományból *n* egységnyi adatot ír ki a *buffer*-be. Az adatok méretét (bájtban) a *méret* paraméter adja meg. Például a

```
numread = fread (text, sizeof (char), 80, in_file);
```

utasítás 80 karaktert olvas ki az *in_file*-ból, és mindezt eltárolja a *text* tömbben. A függvény a sikeresen kiolvasott adategységek számát adja vissza.

```
FILE *freopen (állományNév, elérésiMód, filePtr)
```

Lezárja a *filePtr*-rel azonosított állományt, és megnyitja az *állományNév* nevű állományt a megadott *elérésiMód* szerint (lásd az *fopen* függvényt). A megnyitott fájl a továbbiakban a *filePtr*-rel érhető el. Ha sikeresen lefutott a függvény, akkor a *filePtr*-t adja vissza, egyébként pedig a nullmutatót. Az *freopen* függvényt gyakran használják az *stdin*, *stdout* és az *stderr* programból történő átirányítására. Például az

```
if ( freopen ("inputData", "r", stdin) == NULL ) {  
    ... }
```

hatására az *stdin* átirányítódik az *inputData* állományra, amely olvasási módban nyílik meg. Azokat a kimeneti/bemeneti műveleteket, melyek az *stdin*-re vonatkoznak, ettől kezdve az *inputData* állománnyal hajtja végre a program; ugyanúgy, mintha a program végrehajtásakor átirányítottuk volna az *stdin*-t erre a fájlra.

```
int fscanf (filePtr, formátum, param1, param2, ..., paramn)
```

A *formátum* karakterláncnak megfelelően adatokat olvas be a *filePtr*-rel azonosított állományból, és eltárolja a *formátum* után sorakozó paraméterekben, melyek mindegyike egy-egy mutató. A *formátum*jelző karakterláncban ugyanazokat az értékeket használhat-

jur, mint a `scanf` függvényben (16. fejezet). A sikeresen beolvasott és hozzárendelt adatok számát adja vissza a függvény (nem számítva a `\n`-nel jelzett hozzárendeléseket). Ha az állomány végét még az előtt eléri, mielőtt az első beolvasott adatot sikeresen fel tudná dolgozni, akkor az EOF értékét adja vissza.

```
int fseek (filePtr, pozíció, mód)
```

A megadott állomány írási/olvasási mutatóját beállítja a megadott (`long int`) *pozíció*-ra. Az (egészként megadott) *mód*-tól függően ezt a távolságot (bájtban) a fájl aktuális pozíciójától, a fájl elejétől vagy a fájl végétől méri. Ha a *mód* értéke `SEEK_SET`, akkor a pozíciót a fájl elejétől; ha `SEEK_CUR`, akkor a fájl aktuális pozíciójától; ha pedig `SEEK_END`, akkor a fájl végétől méri a függvény. A `SEEK_SET`, `SEEK_CUR` és `SEEK_END` értéke a `<stdio.h>`ban van definiálva.

Azokon az operációs rendszereken, amelyek megkülönböztetik a bináris fájlokat a szöveg fájloktól, a `SEEK_END`-et nem minden tudjuk használni bináris fájlok esetében. A szöveges állományokra a *pozíció*-t nullára kell állítani, vagy egy olyan értékre, amit egy korábbi `ftell` hívásból kaptunk. Ez utóbbi esetben a *mód*-ot `SEEK_SET`-re kell beállítani.

Ha az `fseek` hívás sikertelen, a visszatérési érték nullától eltérő lesz.

```
int fsetpos (filePtr, fpos)
```

A *filePtr*-rel megadott állomány írási/olvasási mutatóját a (`<stdio.h>`-ban definiált) *fpos_t* típusú *fpos*-ra állítja be. Hibátlan lefutás esetén nullát, egyébként nullától eltérő értéket ad vissza. Érdemes tanulmányozni a függvény „kiolvasó pájrát”, az `fgetpos` függvényt is.

```
long ftell (filePtr)
```

A *filePtr*-rel megadott állomány aktuális írási/olvasási mutatójának pozícióját adja vissza, vagy hiba esetén `-1L`.

```
size_t fwrite (buffer, méret, n, filePtr)
```

A *buffer*-ből *n* egységnnyi adatot ír ki a megadott állományba. Az adatok méretét (bájtban) a *méret* paraméter adja meg. A függvény a sikeresen kiírt adategységek számát adja vissza.

```
int getc (filePtr)
```

A megadott állomány következő karakterét olvassa ki és adja vissza. Ha a függvény eléri a fájl végét (vagy hiba lép fel), akkor az EOF lesz a visszatérési érték.

```
int getchar (void)
```

Az `stdin` következő karakterét olvassa ki és adja vissza. Fájl vége (vagy hiba) esetén az `EOF` lesz a visszatérési érték.

```
char *gets (buffer)
```

Beolvassa az `stdin` karaktereit az első újsor karakterig, és mindenzt kiírja a `buffer`-be (az újsor karaktert már nem írja ki). A függvény automatikusan null karakterrel zárja le a karakterláncot. Olvasási hiba esetén – vagy ha nem volt beolvasandó karakter – az `EOF` érték adja vissza; egyébként pedig a `buffer` mutatóját.

```
void perror (üzenet)
```

A legutóbbi `stderr` hibához fűz magyarázatot, megjelenítve az `üzenet`-et. Az alábbi kód részlet például akkor jelenít meg hibaüzenetet, ha az `fopen` hívás meghiúsul, pontosabb eligazítást adva a felhasználónak a hiba okairól:

```
if ( (in = fopen ("data", "r")) == NULL ) {  
    perror ("data file read");  
    exit (EXIT_FAILURE);  
}
```

```
int printf (formátum, param1, param2, ..., paramn)
```

A megadott `formátum` szerint (lásd a 16. fejezetet) kiírja a megadott paramétereket az `stdout`-ra. A kiírt karakterek számát adja vissza.

```
int putc (c, filePtr)
```

`c` értékét előjel nélküli char-ként írja ki a megadott állományba. Siker esetén maga `c` lesz a visszatérési érték, egyébként pedig az `EOF`.

```
int putchar(c)
```

`c` értékét előjel nélküli char-ként írja ki az `stdout`-ra. Siker esetén maga `c` lesz a visszatérési érték, egyébként pedig az `EOF`.

```
int puts (buffer)
```

A `buffer` által mutatott karaktertömb értékét írja ki az `stdout`-ra egészen az első null karakterig (azt már nem írja ki). Ellentétben az `fputs` függvénnyel automatikusan újsor karaktert ír az utolsó karakter után a fájlba. Hiba esetén az `EOF` értékét adja vissza.

```
int remove (állományNév)
```

Törli a megadott állományt. Hiba esetén nem nullát ad vissza.

```
int rename (állományNév1, állományNév2)
```

Az első paraméterként megadott állományt átnevezi állományNév2-re. Hiba esetén nem nullát ad vissza.

```
void rewind (filePtr)
```

A megadott fájl írási/olvasási mutatóját visszaállítja a fájl elejére

```
int scanf (formátum, param1, param2, ..., paramn)
```

A *formátum* karakterláncnak megfelelően (lásd a 16. fejezetet) adatokat olvas be az *stdin*-ról, és eltárolja a *formátum* után sorakozó paraméterekben, melyek mindegyike egy-egy mutató. A sikeresen beolvasott és hozzárendelt adatok számát adja vissza a függvény (nem számítva a %n-nel jelzett hozzárendeléseket). Ha az állomány végét még előtt eléri, mielőtt az első beolvasott adatot sikeresen fel tudná dolgozni, akkor az EOF értékét adja vissza.

```
FILE *tmpfile (void)
```

Létrehoz és megnyit egy ideiglenes bináris állományt frissítési módban ("r+b"). NULL értéket ad vissza hiba esetén. A program kilépésekor automatikusan törlődik az ideiglenes állomány. (Létezik egy *tmpnam* nevű függvény is egyedi nevű ideiglenes állományok létrehozására.)

```
int ungetc (c, filePtr)
```

A függvény „visszateszi” a *c* karaktert a megadott állományba. Valójában nem íródik vissza a karakter, csak bekerül egy olyan tárolóba, amely az adott fájlhoz van társítva. A következő *getc* függvény azonban ezt a karaktert fogja kiolvasni. Az *ungetc* függvény egy állományba csak egyetlen karaktert tud egyszerre „visszatenni”; azaz egy olvasási műveletet végre kell hajtani a fájlon, mielőtt egy újabb *ungetc* hívást végeznénk. A függvény a *c* értékét adja vissza sikeres lefutás esetén, egyébként pedig az EOF-ot.

A memóriában történő formátumátalakítás függvényei

Az *sprintf* és az *sscanf* a memóriában hajtják végre a kívánt adatátalakításokat. Hasonlitanak az *fprintf* és az *fscanf* függvényekhez, de az első paraméterük nem egy karakterlánc, hanem egy *FILE*-mutató. Használatukhoz be kell emelni programunkba az *<stdio.h>* fejlcállományt.

```
int sprintf (buffer, formátum, param1, param2, ..., paramn)
```

A megadott paramétereket a *formátum* karakterláncnak megfelelően (lásd a 16. fejezetet) kiírja a *buffer*-rel azonosított karaktertömbbe, és a legvégre automatikusan egy null karaktert ír. A *buffer*-be tett karakterek számát adja vissza a függvény (amelybe nem számítja bele a záró nullt). Álljon itt egy példa:

```
char buffer[] = "Július 16, 2008", month[10];
int day, year;
...
sscanf (buffer, "%s %d, %d", month, &day, &year);
```

A „Július”-t a *month* változóba tölti, a 16-os egész számot a *day*-be, a 2008-as egészet pedig a *year*-be. A következő kód részlet az (*argv[1]*-ben tárolt) első parancssori paramétert lebegőpontos számmá alakítva írja ki, majd ellenőrzi a *sscanf* által visszaadott értéket, hogy sikeres volt-e az *argv[1]*-ből való olvasás:

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

if ( sscanf (argv[1], "%f", &fval) != 1 ) {
    fprintf (stderr, "Rossz szám: %s\n", argv[1]);
    exit (EXIT_FAILURE);
}
```

A következő szakaszban leírt eljárások között talál az olvasó olyanokat, amivel más módon lehet a karakterláncokat számmá alakítani.

Karakterláncok számmá alakítása

A most tárgyalásra kerülő eljárások a megadott karakterláncot számmá alakítják. Használatukhoz a program elején be kell emelni a *<stdlib.h>* fejlécállományt a következő utasítással:

```
#include <stdlib.h>
```

Az alábbi függvények leírásában az *s* egy null karakterrel lezárt karakterlánc mutatója, a vége egy karakter-mutató mutatója, az *alap* pedig int típusú egész szám.

Ezek a függvények kihagyják a feldolgozandó karakterlánc bevezető helyköz karaktereit, és megállnak a beolvasással, amikor (az átalakítandó érték típusát tekintve) érvénytelen karakterrel találkoznak.

```
double atof (s)
```

Az *s* által mutatott karakterláncot lebegőpontos számmá alakítja, és ezt adja vissza.

```
int atoi (s)
```

Az *s* által mutatott karakterláncot int típusú számmá alakítja, és ezt adja vissza.

```
long atol (s)
```

Az *s* által mutatott karakterláncot long int típusú számmá alakítja, és ezt adja vissza.

```
long long atoll (s)
```

Az *s* által mutatott karakterláncot long long int típusú számmá alakítja, és ezt adja vissza.

```
double strtod (s, vége)
```

Az *s* által mutatott karakterláncot double típusú lebegőpontos számmá alakítja, és ezt adja vissza. Amennyiben a *vége* nem nullmutató, akkor benne tárolja el az utolsó beolvasott karaktert követő *s*-beli karaktert a függvény.

Tekintsük például az alábbi kód részletet:

```
#include <stdlib.h>
...
char    buffer[] = " 123.456xyz", *end;
double value;
...
value = strtod (buffer, &end);
```

Az utolsó értékkadást követően *value* értéke 123.456 lesz. Az *end*-et az *strtod* 'x'-re állítja be, mert a *buffer*-ben ez a karakter áll az utolsó beolvasott 6-os után.

```
double strtod (s, vége)
```

Olyan, mint az *strtod*, csak float típusú lebegőpontos számmá alakítja *s*-et.

```
long int strtol (s, vége, alap)
```

Az *s* által mutatott karakterláncot long int típusú számmá alakítja, és ezt adja vissza.

Az *alap* egy 2 és 36 közötti számrendszer-alapszámot jelent – ennek megfelelően értelmezi a függvény a paraméterként kapott karakterláncot. Ha az *alap* értéke nulla, akkor

az *s* lehet tízes, (vezető 0 esetén) nyolcas vagy (vezető 0x vagy 0X esetén) tizenhatos számrendszerbeli számot ábrázoló karakterlánc. Ha a vége nem nullmutató, akkor benne tárolja el az utolsó beolvasott karaktert követő *s*-beli karaktert a függvény.

```
long double strtold (s, vége)
```

Olyan, mint az strtod, de *s*-et long double típusúvá alakítja.

```
long long int strtoll (s, vége, alap)
```

Olyan, mint az strtol, de *s*-et long long int típusúvá alakítja.

```
unsigned long int strtoul (s, vége, alap)
```

Az *s* által mutatott karakterláncot unsigned long int típusú számmá alakítja, és ezt adja vissza. A vége és az alap úgy értelmeződik, ahogy az strtol esetében.

```
unsigned long long int strtoull (s, vége, alap)
```

Az *s* által mutatott karakterláncot unsigned long long int típusú számmá alakítja, és ezt adja vissza. A többi paraméter úgy értelmeződik, ahogy az strtol esetében.

Dinamikus memóriakezelő függvények

Az alábbi függvények álnak rendelkezésre a dinamikus memória lefoglalására és felszabadítására. A függvények leírásában az *n* és a *méret* egy size_t típusú egész szám, a *mutató* pedig egy void mutató.

A függvényeknek a használatához a program elején be kell emelni a <stdlib.h> fejlécállományt:

```
#include <stdlib.h>
void *calloc (n, méret)
```

Egy adategységet *méret* bájt hosszúságúnak definiálva *n* adategység számára foglal le memóriaterületet, melyet azonnal fel is tölt nullákkal. Siker esetén a lefoglalt terület kezdetének memóriacímét adja vissza, ellenkező esetben pedig a nullmutatót.

```
void free (mutató)
```

Felszabadítja a *mutató* által hivatkozott (calloc, malloc vagy realloc hívása által lefoglalt) memóriaterületet.

```
void *malloc (méret)
```

méret bájt számára foglal le memóriaterületet. Siker esetén a lefoglalt terület kezdetének memóriacímét adja vissza, ellenkező esetben pedig a nullmutatót.

```
void *realloc (mutató, méret)
```

Egy előzőleg már lefoglalt memóriaterület méretét *méret* méretűvé változtatja. Sikeres memória foglalás esetén az új egybefüggő terület kezdetének memóriacímét adja vissza (amely el is térhet az eredeti kezdőcímétől), hiba esetén pedig a nullmutatót.

Matematikai függvények

Az alábbiakban matematikai függvényeket fogunk felsorolni. A függvényeknek a használatához a program elején be kell emelni a `<math.h>` fejlécállományt:

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

A `<tgmath.h>` szabványos fejlécállomány típusfüggetlen makrókat tartalmaz. Ezeknek a segítségével a `math` és a `complex` matematikai könyvtár függvényeit a paraméterek típusától függetlenül tudjuk használni. A négyzetre emeléshez például (a normál függvények használatakor) hat különböző esetet kellene megkülönböztetnünk a paraméter és a visszatérési érték alapján:

- `double sqrt (double x)`
- `float sqrtf (float x)`
- `long double sqrtl (long double x)`
- `double complex csqrt (double complex x)`
- `float complex csqrft (float complex f)`
- `long double complex csqrtl (long double complex)`

Ahelyett, hogy azon törjük a fejünket, hogy melyiket is kell az adott helyzetben meghívni, a `<math.h>` és a `<complex.h>` fejlécállományok beemelése helyett használhatjuk a `<tgmath.h>`-t, amelyben csak egy „általános” négyzetre emelés szerepel `sqrt` néven. A `<tgmath.h>`-ban szereplő makró biztosítja, hogy a megfelelő függvény kerül meghívásra.

De tértünk vissza a `<math.h>`-hoz. Az alábbi makrókkal ellenőrizhető, hogy egy-egy lebegőpontos paraméternek milyen jellemzői vannak:

```
int fpclassify (x)
```

Besorolja *x*-et az alábbiak valamelyikébe: „nem szám”, (*Not A Number*, azaz `FP_NAN`), plusz vagy mínusz végtelen (`FP_INFINITE`), túl kicsi ahhoz, hogy ábrázolni lehessen

(FP_SUBNORMAL), nulla (FP_ZERO); ha egyik sem teljesül, akkor egy normál lebegőpontos számmal van dolgunk (FP_NORMAL) vagy valamilyen implementációfüggő kategóriával.

Minden FP... érték a math.h-ban van definiálva; maga az FP... a lebegőpontosságra (*floating-point*) utal.

int isfin (x)

x véges?

int isinf (x)

x végtelen?

int isgreater (x, y)

Igaz-e, hogy $x > y$?

int isgreaterequal (x, y)

Igaz-e, hogy $x = y$?

int islessequal (x, y)

Igaz-e, hogy $x = y$?

int islessgreater (x, y)

Igaz-e, hogy $x < y$ vagy $x > y$?

int isnan (x)

Igaz-e, hogy x nem szám? (*Not A Number*)

int isnormal (x)

Igaz-e, hogy x normál lebegőpontos érték?

int isunordered (x, y)

Igaz-e, hogy x és y „rendetlen”? (Azaz például valamelyikük nem szám)?

```
int signbit (x)
```

Igaz-e, hogy x a megfelelő előjelbit alapján negatív?

Az alábbi függvények leírásában az x , y és z típusa double, r egy radiánban megadott szög (típusa double), n pedig egy int típusú egész szám.

A függvényekhez tartozó hibajelzések a dokumentációban találhatóak.

```
double acos (x)1
```

Visszaadja x arkusz koszinuszát egy $[0, \pi]$ tartományban levő, radiánban megadott szööként. x -nek a $[-1, 1]$ intervallumban kell lennie.

```
double acosh (x)
```

Visszaadja x arkusz koszinusz hiperbolikusát, $x = 1$.

```
double asin (x)
```

Visszaadja x arkusz szinuszát egy $[-\pi/2, \pi/2]$ tartományban levő, radiánban megadott szööként. x -nek a $[-1, 1]$ intervallumban kell lennie.

```
double asinh (x)
```

Visszaadja x arkusz szinusz hiperbolikusát.

```
double atan (x)
```

Visszaadja x arkusz tangensét egy $[-\pi/2, \pi/2]$ tartományban levő, radiánban megadott szööként.

```
double atanh (x)
```

Visszaadja x arkusz tangens hiperbolikusát, $|x| = 1$.

```
double atan2 (y, x)
```

Visszaadja a y/x arcusz tangensét egy $[-\pi, \pi]$ tartományban levő, radiánban megadott szööként.

¹ A matematikai programkönyvtárban minden függvényhez létezik float, double és long double változat, amely ilyen típusú értéket vár vagy ad vissza. Függelékünkben csak a double változatot mutatjuk be. A float változat ugyanilyen, csak f betűvel végződik (például acosf). A long double változatnak 1 betű szerepel a végén (például acosl).

```
double ceil (x)
```

Visszaadja a legkisebb egész számot, amely nagyobb vagy egyenlő x -nél. Figyelem, a visszatérési érték típusa double. (A szó jelentése képszerű: „plafon”.)

```
double copysign (x, y)
```

Visszaadja egy olyan értéket, melynek abszolút értéke x , de előjele az y -éval egyezik meg.

```
double cos (r)
```

Visszaadja r koszinuszát.

```
double cosh (x)
```

Visszaadja x koszinusz hiperbolikuszát.

```
double erf (x)
```

Kiszámítja és visszaadja az x -hez tartozó hibafüggvényt.

```
double erfc (x)
```

Kiszámítja és visszaadja az x -hez tartozó kiegészítő hibafüggvényt.

```
double exp (x)
```

Visszaadja e^x értékét.

```
double expm1 (x)
```

Visszaadja $e^x - 1$ értékét. (Ennek akkor van értelme, amikor x 0-hoz közel, és a normál módon számított érték igen pontatlan lenne.)

```
double fabs (x)
```

Visszaadja x abszolút értékét.

```
double fdim (x, y)
```

Visszaadja $x - y$ értékét, ha $x > y$; egyébként 0-t.

```
double floor (x)
```

Visszaadja a legnagyobb egész számot, amely kisebb vagy egyenlő x -nél. Figyelem, a visszatérési érték típusa double. (A szó jelentése képszerű: „padló”.)

```
double fma (x, y, z)
```

Visszaadja $(x \cdot y) + z$ értékét.

```
double fmax (x, y)
```

Visszaadja x és y közül a nagyobbikat.

```
double fmin (x, y)
```

Visszaadja x és y közül a kisebbiket.

```
double fmod (x, y)
```

Visszaadja x és y lebegőpontos osztási maradékát. A visszatérési érték előjele x előjelével egyezik meg.

```
double frexp (x, exp)
```

x értékét normálalakra hozza. A 2-hatvánnyal értelmezett kitevőt eltárolja az exp által hi-vatkozott egész változóban. Az $[1/2, 1]$ intervallumbeli mantissa lesz a függvény visszatérési értéke. Ha x nulla, akkor minden a visszaadott mantissa, minden az eltárolt kitevő (karakterisztika) nulla lesz.

```
int hypot (x, y)
```

Visszaadja x és y négyzetösszegének négyzetgyökét.

```
int ilogb (x)
```

x -et normálalakra hozza, majd előjeles egészként (innen az i betű a névben – int) visszaadja a kitevő előjelét.

```
double ldexp (x, n)
```

Visszaadja $x \cdot 2^n$ értékét.

```
double lgamma (x)
```

Visszaadja x Gamma-függvényértéke abszolút értékének természetes alapú logaritmusát.

```
double log (x)
```

Visszaadja x természetes (e) alapú logaritmusát, $x = 0$.

```
double logb (x)
```

x -et normálalakra hozza, és visszaadja a kitevő előjelét (lebegőpontosként).

```
double log1p (x)
```

Visszaadja $(x + 1)$ természetes alapú logaritmusát, $x = -1$.

```
double log2 (x)
```

Visszaadja x kettes alapú logaritmusát, $x = 0$.

```
double log10 (x)
```

Visszaadja x tízes alapú logaritmusát, $x = 0$.

```
long int lrint (x)
```

Visszaadja x (legközelebbi long int egészhez) kerekített értékét.

```
long long int llrint (x)
```

Visszaadja x (legközelebbi long long int egészhez) kerekített értékét.

```
long long int llround (x)
```

Visszaadja x (legközelebbi long long int egészhez) kerekített értékét. A „félúton” levő értékek nullától távolabbrá kerekítődnek (azaz a 0.5 kerekítve 1 lesz).

```
long int lround (x)
```

Visszaadja x (legközelebbi long int egészhez) kerekített értékét. A „félúton” levő értékek nullától távolabbrá kerekítődnek (azaz a 0.5 kerekítve 1 lesz).

```
double modf (x, egészrész)
```

x egészrészét és törtrészét szolgáltatja. A törtrész a függvény visszatérési értéke, az egészrész pedig betölti az *egészrész* által mutatott double változóba.

```
double nan (s)
```

Visszaadja a „nem szám”, (*NaN*, „*Not a Number*”) belső reprezentációját (ha lehetséges) az *s* által hivatkozott karakterlánc tartalma alapján.

```
double nearbyint (x)
```

Lebegőpontos formában adja vissza az *x*-hez legközelebbi egész számot.

```
double nextafter (x, y)
```

Visszaadja *x*-nek *y* irányába eső legközelebbi ábrázolható szomszédját.

```
double nexttoward (x, ly)
```

Visszaadja *x*-nek *ly* irányába eső legközelebbi ábrázolható szomszédját, mint a *nextafter*, csak itt a második paraméter long double típusú.

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

Visszaadja x^y -t. Ha *x* negatív, akkor *y*-nak egésznek kell lennie. Ha *x* nulla, akkor *y* csak pozitív lehet.

```
double remainder (x, y)
```

Visszaadja *x* és *y* osztási maradékát.

```
double remquo (x, y, hányados)
```

Visszaadja *x* és *y* osztási maradékát; hányadosukat pedig a *hányados* nevű mutató által hivatkozott egész változóban tárolja el.

```
double rint (x)
```

Lebegőpontos formában adja vissza az *x*-hez legközelebbi egész számot. Felhasználható arra, hogy lebegőpontos kivételt hozunk létre olyan esetben, ha a visszatérési érték nem egyezik meg a paraméter értékével.

```
double round (x)
```

Lebegőpontos formában adja vissza x legközelebbi egészhez kerekített értékét. A „fél-úton” levő értékek nullától távolabbról kerekítődnek (azaz a 0.5 kerekítve 1 lesz).

```
double scalbln (x, n)
```

Visszaadja $x \cdot \text{FLT_RADIX}^n$ értékét (n long int típusú).

```
double scalbn (x, n)
```

Visszaadja $x \cdot \text{FLT_RADIX}^n$ értékét.

```
double sin (r)
```

Visszaadja r szinuszának értékét.

```
double sinh (x)
```

Visszaadja r szinusz hiperbolikuszának értékét.

```
double sqrt (x)
```

Visszaadja x négyzetgyökének értékét, $x = 0$.

```
double tan (r)
```

Visszaadja r tangenségek értékét.

```
double tanh (x)
```

Visszaadja r tangens hiperbolikuszának értékét.

```
double tgamma (x)
```

Visszaadja a Gamma függvény x helyen vett függvényértékét.

```
double trunc (x)
```

x -et egésszé csonkolja, és az eredményt double típusúként adja vissza.

Komplex aritmetika

A `<complex.h>` fejlécállományban találhatóak a komplex számok használatához szükséges függvények, makrók és deklarációk. Az alábbiakban felsorolunk néhány makrót és függvényt, melyek a komplex aritmetika támogatására születtek.

Definíció	Jelentése
<code>complex</code>	A <code>_Complex</code> típus helyettesítője
<code>_Complex_I</code>	Egy komplex szám képzetesz részét kialakító makró. Például a <code>4 + 6.2 · _Complex_I</code> hozza létre a <code>4 + 6.2i</code> -t.
<code>imaginary</code>	Az <code>_Imaginary</code> típus helyettesítője. Csak akkor van definiálva, ha az implementáció támogatja a képzetesz típusokat.
<code>_Imaginary_I</code>	Egy képzetesz szám képzetesz részét kialakító makró.

Az alábbi függvények leírásában az `y` és `z` típusa `double complex`, `x` `double` típusú, `n` pedig egy `int` típusú egész szám.

`double complex cabs (z)2`

Visszaadja `z` komplex abszolút értékét.

`double complex cacos (z)`

Visszaadja `z` komplex arkusz koszinuszát.

`double complex cacosh (z)`

Visszaadja `z` komplex arkusz koszinusz hiperbolikuszát.

`double carg (z)`

Visszaadja `z` fázisszögét.

`double complex casin (z)`

Visszaadja `z` komplex arkusz szinuszát.

² A komplex matematikai programkönyvtár minden függvényhez létezik `float complex`, `double complex` és `long double complex` változat, amely ilyen típusú értéket vár vagy ad vissza. Itt csak a `double complex` változatot mutatjuk be. A `float complex` változat ugyanilyen, csak f betűvel végződik (például `cacosf`). A `long double` változatnak l betű szerepel a végén (például `cacosl`).

```
double complex casinh (z)
```

Visszaadja z komplex arkusz szinusz hiperbolikuszát.

```
double complex catan (z)
```

Visszaadja z komplex arkusz tangensét.

```
double complex catanh (z)
```

Visszaadja z komplex arkusz tangens hiperbolikuszát.

```
double complex ccos (z)
```

Visszaadja z komplex koszinuszát.

```
double complex ccosh (z)
```

Visszaadja z komplex komplex koszinusz hiperbolikuszát.

```
double complex cexp (z)
```

Visszaadja a természetes alapú hatványfüggvény komplex z helyen vett függvényértékét.

```
double cimag (z)
```

Visszaadja z képzetesz részét.

```
double complex clog (z)
```

Visszaadja z természetes alapú komplex logaritmusát.

```
double complex conj (z)
```

Visszaadja z komplex konjugáltját (azaz csak a képzetesz résznek veszi az ellentéjtét).

```
double complex cpow (y, z)
```

Visszaadja y^z komplex hatványértékét.

```
double complex cproj (z)
```

Visszaadja z -nek a Riemann-gömbre eső vetületét.

```
double complex creal (z)
```

Visszaadja z valós részét.

```
double complex csin (z)
```

Visszaadja z komplex szinuszt.

```
double complex csinh (z)
```

Visszaadja z komplex szinusz hiperbolikuszát.

```
double complex csqrt (z)
```

Visszaadja z komplex négyzetgyökét.

```
double complex ctan (z)
```

Visszaadja z komplex tangensét.

```
double complex ctanh (z)
```

Visszaadja z komplex tangens hiperbolikuszát.

Általános segédfüggvények

Van néhány függvény, amelyik egyik fenti kategóriába sem fér bele. Ezek használatához a `<stdlib.h>` fejlécállományt kell beemelni a programba:

```
#include <stdlib.h>
```

```
int abs (n)
```

Az int típusú n abszolút értékét adja vissza.

```
void exit (n)
```

Befejezi a program végrehajtását, lezárja az esetlegesen megnyitva maradt állományokat és az int típusú n paraméter értékét adja vissza kilépési állapotként. A `<stdlib.h>` fejlécállományban van definiálva az `EXIT_SUCCESS` és az `EXIT_FAILURE` – ezek segítségével jelezhetjük a sikeres (*success*) és a kudarcba fulladt (*failure*) végkifejletet. A függvénykönyvtárban találhatunk még hasonló célú függvényeket `abort` és `atexit` néven.

```
char *getenv (s)
```

Az *s* által hivatkozott környezeti változó értékére irányuló mutatót adja vissza (vagy a nullmutatót, ha nincs ilyen változó). A függvény nyilván erősen függ a használt operációs rendszertől. Unix alatt például a

```
char *homedir;
...
homedir = getenv ("HOME");
```

kódrészlettel olvashatjuk be a *homedir* mutató által hivatkozott változóba a felhasználó *HOME* könyvtárának értékét.

```
long int labs (l)
```

A long int típusú *l* abszolút értékét adja vissza.

```
long long int labs (ll)
```

A long long int típusú *ll* abszolút értékét adja vissza.

```
void qsort (tömbmutató, n, méret, comp_fn)
```

A void típusú *tömbmutató* által hivatkozott tömb adatait rendezzi. A tömbben *n* elem van, minden *yik* *méret* méretű. Az *n* és a *méret* típusa size_t. A negyedik paraméter típusa „olyan függvény mutatója, amely két void mutatót vár, és egy int értéket ad vissza”. A *qsort* akkor hívja meg ezt a függvényt, amikor két tömbelem összehasonlításához átkell adnia neki az elemekre irányuló mutatót. A (felhasználó által megadott) függvénytől azt várja a *qsort*, hogy összehasonlítsa a két megadott elemet, és döntsön, melyik a kisebb. Adjon vissza negatív számot, ha az első a kisebb, nullát, ha egyenlők, és pozitív számot, ha a második a kisebb.

A data nevű, ezer egészet tartalmazó tömb a következőképp rendezhető a *qsort*-tal:

```
#include <stdlib.h>
...
int main (void)
{
    int data[1000], comp_ints (void *, void *);
    ...
    qsort (data, 1000, sizeof(int), comp_ints);
    ...
}
```

```
int comp_ints (void *p1, void *p2)
{
    int i1 = * (int *) p1;
    int i2 = * (int *) p2;
    return i1 - i2;
}
```

Az itt nem részletezett bsearch függvény a qsort-hoz hasonló paramétereket vár. Egy rendezett tömbben keresi meg logaritmikus kereséssel a kérő adatot.

```
int rand (void)
```

Egy [0, RAND_MAX] intervallumba eső véletlenszámot ad vissza. A RAND_MAX értéke a <stdlib.h>-ban van definiálva; értéke legalább 32767. Érdemes még tanulmányozni az srand függvényt is.

```
void srand (mag)
```

Újrainicializálja a véletlenszám-generátort az unsigned int mag paraméterrel.

```
int system (s)
```

Az s tömbben levő karakterláncot átadja az operációs rendszernek végrehajtásra, és a rendszer által megadott értéket adja vissza. Ha s a nullmutató, akkor (amennyiben elérhető az operációs rendszer parancsértelmezője) a system függvény nullától eltérő értéket fog visszaadni.

Példaként tekintsük az alábbi függvényhívást Unix alatt:

```
system ("mkdir /usr/tmp/data");
```

Ennek hatására a rendszer létrehozza a /usr/tmp/data könyvtárat, ha van megfelelő jogosultsága a programot futtató felhasználónak.



Programok fordítása gcc-vel

Ez a függelék GNU C fordító használati lehetőségeit, leggyakoribb opcióit mutatja be. Ha kívánCSI az olvasó az összes használható parancssori kapcsolóra, akkor írja be a terminálablakra (Unix alatt), hogy „man gcc”.

Érdemes meglátogatni a <http://gcc.gnu.org/onlinedocs> oldalt a teljes dokumentációért.

Függelékünk csak a gcc 3.3 verziójának parancssori lehetőségeit tárgyalja, és egyéb (például az Apple Computer, Inc. által hozzáadott) kiterjesztésekéről nem lesz szó.

A fordítási parancs általános formája

A gcc parancs általános formája a következő:

```
gcc [opción] állomány [állomány ...]
```

A szöglletes zárójelben megadott összetevők nem kötelezőek.

A gcc fordítóprogram a felsorolt összes állományt lefordítja. Általában ez előfeldolgozást, fordítást, összeállítást és linkelést jelent; parancssori kapcsolókkal azonban ez megváltozható.

A bemeneti fájlok végződése dönti el, hogy az adott fájlokat hogyan értelmezi a fordítóprogram. Ez felülbírálható a -x kapcsolóval (lásd a gcc dokumentációt). A C.1 Táblázat összefoglalja a leggyakoribb fájlvégződéseket.

C.1 Táblázat • A leggyakoribb fájlvégződések

Végződés	Jelentése
.c	C nyelvű forrásállomány
.cc, .cpp	C++ nyelvű forrásállomány
.h	Fejlécállomány
.m	Objective-C forrásállomány
.pl	Perl forrásállomány
.o	Tárgykódú (előfordított) állomány

Parancssori kapcsolók

A C.2 Táblázat összefoglalja a C programok fordításához használt leggyakoribb kapcsolókat

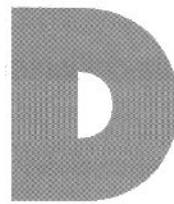
C.2 Táblázat • A gcc leggyakoribb kapcsolói

Kapcsoló	Jelentése	Példa
--help	Kírja a leggyakoribb parancssori kapcsolók leírását.	gcc --help
-c	Nem linkeli össze a tárgykódú objektumokat, csak elmenti őket (az eredeti fájlnév tövéhez kapcsolt) .o végződéssel.	gcc -c enumerator.c
-dumpversion	Kírja az aktuális gcc verziószámát	gcc -dumpversion
-g	Nyomkövetési információkat is fordítson a futtatható állományba; általában gdb-vel használjuk. Ha többféle nyomkövető is támogatott, akkor használjuk a -ggdb kapcsolót.	gcc -g testprog.c -o testprog
-D id -D id=érték	Az első esetben az előfeldolgozó által kiértékelendő id azonosítót 1-re definiálja, a második esetben pedig érték-re.	gcc -D DEBUG=3 test.c
-E	Csak az előfeldolgozási műveletet hajtja végre, és az eredményt a szabványos kimenetre küldi. Tanulságos megvizsgálni, mit eredményez az előfeldolgozás.	gcc -E enumerator.c

Kapcsoló	Jelentése	Példa
-I dir	A fejlécállományok keresésekor felhasznált könyvtárakhoz hozzáveszi a dir könyvtárat is; sőt, ezt a könyvtárat a szabványos könyvtárak előtt fogja átnézni.	gcc -I /users/steve/include x.c
-llibrary	Feloldja a library által megadott könyvtárra vonatkozó hivatkozást. Ezt azután az állomány után kell megadni, amelyik az adott könyvtárból igényel bizonyos függvényeket. A linker szabványos helyeken keresi a liblibrary.a állományt (lásd a -L kapcsolót).	gcc mathfuncs.c -lm
-L dir	A könyvtári állományok keresési könyvtáraihoz hozzáveszi a dir könyvtárat; sőt, ezt a könyvtárat a szabványos könyvtárak előtt fogja átnézni.	gcc -L /users/steve/lib x.c
-o execfile	A végrehajtható állományt execfile néven hozza létre.	gcc dbtest.c -o dbtest
-Oszint	A kód végrehajtását a szint által megadott sebességi szintnek megfelelően optimalizálja. A szint lehet 1, 2 vagy 3. Ha nincs megadva szint (az egyenértékű a -O 1-gyel. A magasabb számok komolyabb optimalizációt jelölnek, és így várhatólag hosszabb fordítási idővel jár a használatuk. Ezzel együtt a nyomkövető (pl. gdb) használati lehetősége is csökken.	gcc -O3 m1.c m2.c -o mathfuncs
-std=standard	Megad egy szabványszintet ¹ a C állományok számára. A GNU kiterjesztés nélküli ANSI C használatához írunk c99-et.	gcc -std=c99 mod1.c mod2.c

¹ Jelenleg az alapértelmezett érték a gnu89, ami az ANSI C90-et és a GNU kiterjesztések foglalja magában. Ez meg fog változni gnu99-re (ANSI C99-et és a GNU kiterjesztések), amikor az összes C99 újdonságot implementálták a gcc-be.

Kapcsoló	Jelentése	Példa
-Wwarning	A warning által megadott figyelmeztetési szintre kapcsol. Hasznos az a11 beállítása, amely minden gyanús eseményt figyelmeztetéssel nyugtáz, és igen sokat segíthet a programfejlesztésben. Beállítható az error is, amely minden figyelmeztetést hibaként értelmez, így nem hagyja, hogy kijavítatlan maradjon.	gcc -Werror mod1.c mod2.c



Gyakoribb programozási hibák

Az alábbi felsorolás összefoglalja a leggyakoribb hibákat, amiket a C programozók el szoktak követni. Nincsenek sorrendbe szedve. A hibalehetőségek ismerete remélhetőleg segíti majd az olvasót abban, hogy elkerülje őket programjai fejlesztésekor.

1. Rossz helyre tett pontosvessző

Példa:

```
if ( j == 100 );
    j = 0;
```

Ebben a kódrészletben j értéke mindenkor 0 lesz, mert a pontosvessző tévesen közvetlenül a zárójel után áll. Emlékezzünk vissza arra, hogy ez szintaktikailag teljesen helyes (hiszen egy null utasítás képviseli a ciklusmagot), így nem kapunk hibajelzést a fordítóprogramról. Ugyanilyen típusú hibát lehet elkövetni a `for` és a `while` ciklusok esetében is.

2. Az = operátor használata a == operátor helyett

Ezt a hibát általában `if`, `while` vagy `do` utasításokban szokták elkövetni.

Példa:

```
if ( a = 2 )
    printf ("Te jóssz.\n");
```

Ez az utasítás teljesen helyes. Hatására a 2 értékű adódik az a változónak, majd végrehajtóik a `printf` hívás. A naiv programozó várakozásával szemben ez a `printf` mindenkor lefut, hiszen az `if` utasítás feltétel része mindenkor 2 lesz, azaz nullától eltérő módon (igaz-ként) értékelődik ki.

3. A prototípus-deklaráció elhagyása

Példa:

```
result = squareRoot (2);
```

Ha a squareRoot függvényt csak a program egy későbbi pontján (vagy egy másik állományban) definiáljuk, és nincs kimondottan dekláralva, akkor a fordítóprogram azt feltételezi, hogy int értéket ad vissza. Ezen kívül a fordítóprogram a float értéket double típusúvá, a _Bool, char és short paramétereket int típusúvá alakítja. Más paraméter-átalakítás nem történik. Fogadjuk meg azt a tanácsot, hogy *minden* meghívott függvény előtt megadjuk a megfelelő prototípus-deklarációt, még akkor is, ha a meghívásuk helyezkedik el a definíójuk. A deklarációt konkrét utasításként vagy egy megfelelő fejlécállomány beemelésével is megtehetjük.

4. Az operátorok véghajtási sorrendjének eltévesztése

Példák:

```
while ( c = getchar () != EOF )
...
if ( x & 0xF == y )
...
```

Az első példában a getchar által visszaadott értéket először az EOF-fal hasonlíta össze a programvezérlés, mivel az egyenlőtlenség vizsgálata magasabb precedenciájú, mint a hozzárendelési operátor. A c-hez tehát már csak egy igaz/hamis vizsgálat eredménye, 0 vagy 1 rendelődik hozzá. (1 akkor, ha a getchar nem EOF-ot ad vissza, egyébként pedig 0.)

A második példában az történik, hogy először a 0xF egész konstans összehasonlítódik az y-nal, mivel az egyenlőségvizsgálat magasabb precedenciájú, mint bármilyen bitenkenti operátor. A vizsgálat eredménye (0 vagy 1) lesz azután bitenkénti ÉS kapcsolatba hozva x értékével.

5. Karakterkonstans és karakterlánc összetévesztése

A

```
text = 'a';
```

utasításban egyetlen karakter kerül be a text változóba, míg a

```
text = "a";
```

utasítással egy "a" karakterláncra hivatkozó mutató kerül a text-be. Az első esetben a text-nek char típusúnak kell lennie, míg a második esetben „char-ra irányuló mutatónak”.

6. Tömb indexhatárainak megsértése

Példa:

```
int a[100], i, sum = 0;  
...  
for ( i = 1; i <= 100; ++i )  
    sum += a[i];
```

Egy tömb indexe 0-tól az elemszám-1-ig terjedhet. A fenti ciklus tehát helytelen, mert 99 helyett 100-ig lépdel a tömbelemek indexein. A program szerzője vélhetőleg a tömb első elemétől akarta kezdeni a ciklust, így i értékét 1 helyett 0-ra lenne érdemes inicializálni a ciklusfejben.

7. A karakterláncot lezáró null karakter helyigényének elfelejtése

Ne felejtsük el, hogy a karaktertömböknek elegendő méretűnek kell lenniük ahhoz, hogy a karakterláncot lezáró null karakter is elférjen bennük. A "hello" karakterláncnak tehát hat karakternyi helyre van szüksége a tömbben, hogy a lezáró null karaktert is ott tudjuk tárolni.

8. A -> és a . operátorok összetévesztése adattagokra történő hivatkozáskor

A . operátor az adatszerkezet-változók adattagjait adja meg, míg a -> operátor az adatszerkezetre irányuló *mutatók* esetén használható. Ha tehát x egy adatszerkezet-változó, akkor az x.m jelölésmód x-nek az m adattagjára utal. Ha viszont x egy adatszerkezetre irányuló mutató, akkor az x->m jelölésmód adja meg helyesen az x-hez tartozó m adattagot.

9. A „címe” operátor (&) elhagyása a scanf nem mutató paraméterei elől

Példa:

```
int number;  
...  
scanf ("%i", number);
```

A scanf-ben a formátumjelző karakterlánc után megjelenő minden paraméternek mutatónak kell lennie.

10. Mutató használata annak inicializálása előtt

Példa:

```
char *char_pointer;
*char_pointer = 'X';
```

A közvetett elérés (indirekció) operátort csak *azután* lehet használni, ha már valahová beállítottuk a mutatót. Példánkban a `char_pointer` nem lett beállítva semmire, így az értékadás értelmetlen.

11. A break elhagyása a case ágak végéről a switch utasításban

Emlékezzünk vissza, hogy a case ágak végét break-kel kell lezárnunk, ha nem akarjuk, hogy a következő ágra is rákerüljön a vezérlés.

12. Pontosvessző írása az előfeldolgozónak szánt utasítás végére

Előbb-utóbb szokásunkká válik, hogy az utasításokat lezárjuk egy pontosvesszővel. Mégsem téveszthetjük szem elől, hogy az előfeldolgozónak szánt #define utasításban az adott sorban a név után álló teljes szövegrész értékül adódik az azonosítónévnek, és a program további részében erre cserélődik le. Így ha az alábbi definíció:

```
#define END_OF_DATA 999;
```

után egy ilyen sor következik:

```
if ( value == END_OF_DATA )
    ...
```

ez szintaktikailag hibás kódot eredményez, mert az előfeldolgozó így cseréli ki az `END_OF_DATA` azonosítót:

```
if ( value == 999; )
    ...
```

13. A paraméterek körüli zárójelek elhagyása makródefinícióban

Példa:

```
#define reciprocal(x) 1/x
...
w = reciprocal (a + b);
```

Ezek után – helytelenül – így fog kinézni az előfeldolgozó által beillesztett makró:

```
w = 1 / a + b;
```

14. A makró neve és paraméterlistája közti szóköz a #define-ban

Példa:

```
#define MIN (a,b) ( ( (a) < (b) ) ? (a) : (b) )
```

Ez a definíció nem jó, mert az előfeldolgozó a név utáni első szóköz után álló teljes szövegrészt a makródefiníció részeként tekinti. Esetünkben a

```
minVal = MIN (val1, val2);
```

kódrészlet – szándékunk ellenére – így bomlik ki az előfeldolgozás után:

```
minVal = (a,b) ( ( (a) < (b) ) ? (a) : (b) )(val1, val2);
```

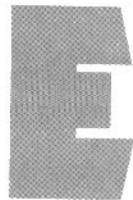
15. Mellékhatással rendelkező kifejezés használata makróhívásban

Példa:

```
#define SQUARE(x) (x) * (x)
...
w = SQUARE (++v);
```

A SQUARE meghívásakor v értéke *kétszer* fog megnőni, mivel az utóbbi sor így helyettesítődik:

```
w = (++v) * (++v);
```



Források

Ez a függelék válogatást nyújt azon forrásokból, amelyekből további hasznos információkat szerezhetünk. A források egy része az interneten található, más része könyv formában kapható. Ha az olvasó nem találja a keresett anyagot, írjon egy e-mailt (angolul) a steve@kochan-wood.com címre – megpróbálok segítségére lenni.

Válaszok a kérdésekre, hibajegyzék stb.

Érdemes meglátogatni a www.kochan-wood.com weboldalt, ahol a kérdésekre adott válaszok és a hibajegyzék is megtalálható. A weboldalon további online források is találhatóak.

A C programozási nyelv

A C programozási nyelv immár több, mint 30 éves múltra tekint vissza, így bőséges információ gyűlt össze róla. Az alábbi gyűjtemény csak a jéghagy csúcsa.

Könyvek

Kernighan, Brian W. és Dennis M. Ritchie. *The C Programming Language*, 2. kiadás. Englewood-Cliffs, NJ: Prentice Hall, Inc., 1988.
(*A C programozási nyelv* – Műszaki Könyvkiadó, 2006. ISBN: 9789631605525)

Ezt nevezhetjük a C bibliájának, amely minden mérföldkő és hivatkozási pont marad. Ez volt az első könyv a C nyelvről, melynek egyik szerzője maga a C nyelv létrehozója, Dennis Ritchie.

Harbison, Samuel P. III és Guy L. Steele Jr. C: *A Reference Manual*, 5. kiadás.
Englewood-Cliffs, NJ: Prentice Hall, Inc., 2002.

Egy másik remek könyv a C programozók számára.

Plauger, P. J. *The Standard C Library*. Englewood-Cliffs, NJ: Prentice Hall, Inc., 1992.

Ez a könyv a szabványos C programkönyvárakat veszi végig. A megjelenés dátumából lát-
szik, hogy nem tartalmazza az ANSI C99 kiterjesztéseit (mint amilyen például a komplex
matematikai programkönyvtár).

Weboldalak

www.kochan-wood.com

Ezen a weboldalon található a *Topics in C Programming* című könyv új, online kiadása,
melyet Patrick Wooddal írtam az eredeti *Programming in C* könyv folytatásaként.

www.ansi.org

Ez az ANSI Weboldala, itt található a hivatalos ANSI C specifikáció. A keresőrubrikába gé-
pelve a 9899:1999 kódot megtalálhatjuk az ANSI C99 specifikációt.

www.opengroup.org/onlinepubs/007904975/idx/index.html

A könyvtári függvények kiváló lelőhelye ez a weboldal, ahol nemcsak az ANSI C függvé-
nyek találhatók meg.

Hírcsoportok

comp.lang.c

Ez a C nyelv számára létrehozott hírcsoport. A feltett kérdésekre előbb-utóbb megérkezik
a válasz. Némi tapasztalat birtokában már válaszolni is szabad a kezdők kérdéseire.

Az egyes levelezések nyomon követése is sokszor tanulságos.

A <http://groups.google.com> oldalon keresztül lehet csatlakozni a hírcsoporthoz.

C fordítóprogramok és integrált fejlesztői környezetek

Az alábbiakban olyan weboldalakat sorolunk fel, melyekről C fordítóprogramok és fejlesztői környezetek töltethetők le (vagy vásárolhatók meg), illetve online dokumentáció is található.

gcc

gcc.gnu.org

A Szabad Szoftver Alapítvány (*Free Software Foundation, FSF*) által kifejlesztett C fordítóprogramot hívjuk gcc-nek. Az Apple is ezt használja a Max OS X operációs rendszeren. A program szabadon letölthető.

MinGW

www.mingw.org

Ha Windows környezetben szeretne az olvasó C programot fordítani, akkor a GNU gcc fordítót erről a weboldalról is letöltheti. A MSYS is egy alternatíva a könnyen használható parancssori munkakörnyezethez.

CygWin

www.cygwin.com

A CygWin egy Linux-szerű környezetet nyújt Windows alatt. Ez a munkakörnyezet is ingyenesen letölthető.

Visual Studio

<http://msdn.microsoft.com/vstudio>

A Visual Studio a Microsoft integrált fejlesztői környezete, amely különféle programozási nyelveken való fejlesztést tesz lehetővé.

CodeWarrior

www.metrowerks.com/mw/products/default.htm

A Metrowerks professzionális fejlesztői környezete többféle operációs rendszeren is fut: Linuxon, Mac OS X-en, Solaris-on és Windows-on.

Kylix

www.borland.com/kylix/

A Kylix a Borland fejlesztői környezete, mely megvásárolható a Linux alatti programfejlesztéshez.

Kiegészítések

A továbbiakban objektum-orientált programozást támogató könyveket és fejlesztői eszközöket sorolunk fel.

Objektum-orientált programozás

Budd, Timothy. *The Introduction to Object-Oriented Programming*, 3. kiadás. Boston: Addison-Wesley Publishing Company, 2001.

Az objektum-orientált programozás klasszikus bevezetője.

A C++ nyelv

Prata, Stephen. *C++ Primer Plus*, 4. kiadás. Indianapolis: Sams Publishing, 2001.

Stephen tankönyvei igen sikeresek – ez a könyve a C++ programozási nyelvet tanítja meg.

Stroustrup, Bjarne. *The C++ Programming Language*, 3. kiadás. Boston: Addison-Wesley Publishing Company, 2000.

(A C++ PROGRAMOZÁSI NYELV I-II., Kiskapu Kiadó 2001. ISBN: 9789639301191)

A C++ nyelv megalkotójának klasszikus műve.

A C# nyelv

Petzold, Charles. *Programming in the Key of C#*. Redmond, WA: Microsoft Press, 2003.

Ez a könyv kiváló fogadtatást kapott, mint a kezdők C# tankönyve.

Liberty, Jesse. *Programming C#, 3. kiadás*. Cambridge, MA: O'Reilly & Associates, 2003.

Jó bevezető a C# nyelvbe a tapasztalatból programozók számára.

Az Objective-C nyelv

Kochan, Stephen. *Programming in Objective-C*. Indianapolis: Sams Publishing, 2004.

Szerénységem által írt könyv, amely a C vagy bármilyen objektumorientált nyelv ismerete nélkül vezet be az Objective-C nyelv rejtelméibe.

Apple Computer, Inc. *The Objective-C Programming Language*. Cupertino, CA: Apple Computer, Inc., 2004.

Remek Objective-C kézikönyv programozók számára. Ingyenesen elérhető PDF formátumban:

<http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf>.

Fejlesztői eszközök

www.gnu.org/manual/manual.html

Itt számos internetes kézikönyv megtalálható, köztük a cvs, gdb, make és sok más Unix parancssori eszköz leírása.

Tárgymutató

! 292
493
operátor 320
operátor 321
#define 307, 325, 326, 487
#elif 326
#else 326
#endif 326, 408
#error 489
#if 489
#ifdef 326, 408, 490
#ifndef 491
#include 491
#line 492
#pragma 492
#undef 492
%c 366, 370
%e 365
%f 365
%g 365
%i 365, 370
%n 367, 372
%p 367
%s 370
& 287, 451
&& 73, 287
.NET 442
^ 291
_VA_ARGS_ 319
_Bool 28, 453
_Complex 43, 454

_Imaginary 43, 454
| 290
|| 73, 290
~ 292
<< 441
<float.h> 325
<limits.h> 325
<math.h> 325
<stddef.h> 325
<stdio.h> 325
<string.h> 325
-> operátor 246

A, Á

adatszerkezet 457
adatszerkezetek 169
adatszerkezetek tagjai 172
adatszerkezetekre irányuló mutatók
 474
adattípus 25
adattípusok 331, 452
adattömörítés 300
Algoritmus 5
Állomány vége 375
aposztrófok 199
append 376
Apple 435
ar 356
argc 395
argv 395
aritmetikai operátorok 468

assembler 6, 9
 assembly nyelv 6
 átadása csak érték szerint 134
 átirányító parancs 373
 atoi 235
 auto 128, 160, 476
 automatikus lokális változók 160
 automatikus típusátalakítás 338
 automatikusan létrejövő lokális
 változók 128
 azonosítók 447

B

backslash 203
 backtrace 424
 bájt 285
 balérték 464
 beágazás 437
 beállító metódus 437
 Bell Laboratories 1, 7, 440
 bináris 285
 bit 285
 bitek körbeforgatása 297
 bitenkénti ÉS 287
 bitenkénti KIZÁRÓ VAGY 291
 bitenkénti műveletek 287, 469
 bitenkénti negáció 292
 bitenkénti operátorok 286
 bitenkénti VAGY 290
 bitmező 300
 Bjarne Stroustrup 440
 Brad Cox 435
 break 63, 85, 419, 482
 building 9

C, Cs

C# 431, 442
 C++ 431, 440
 calloc 400
 case 85
 casting 41
 char 28, 451

ciklusfeltétel 67
 ciklusmag 50
 ciklusok 46
 „címe” operátor 281
 címke 387
 CodeWarrior 343
 compiler 6
 compound literals 185, 474
 Concurrent Versions Systems 356
 const 113, 257, 393, 463
 continue 64, 420, 482
 core dump 414
 counter 57
 cvs 356
 csökkentő (derementáló) operátor 52

D

data encapsulation 437
 dátum 172
 De Morgan azonosság 294
 debugging 10
 decimális 364
 default 85
 deklaráció a ciklusfejben 57
 deklarálás 23
 dekrementálás 267
 dekrementáló 274
 Dennis Ritchie 1
 digraphs 447
 dinamikus memória foglalás 119, 399
 dinamikus memória kezelő
 függvények 511
 do 62, 482
 do ciklus 62
 double 27, 365, 450
 döntéshozatal 67
 dupla idézőjelek 199

E, É

editor 7
 egész aritmetika 36
 egész átalakítás 476

egész konstansok 25, 449
 egész típusú változó 17
 egész típusú változók 23
 egyoperandusú mínusz 36
 eljárás 132
 előfeldolgozás 307
 előfeldolgozó 407, 486
 előjel 78, 339
 előjel nélküli 364
 előjel nélküli karakteres konstansok 340
 előjelbit 285
 else 484
 eltolási függvény 295
 enum 331
 értelmezési hiba 8
 és 73
 escape sequences 28, 450
 esemény 9
 executable 9
 exit 383
 EXIT_FAILURE 383
 EXIT_SUCCESS 383
 exponens 450
 extern 346, 476

F

fájlba irányítás 372
 fájlkezelő függvények 376
 fájlmutatók 376
 fclose 378
 fejlécállomány 325
 fejlécállományok 351
 fejlécállományok többszörös beemelése 327
 felépítés 9
 felsorolt adattípusok 452, 461
 felsorolt típus 331, 452
 felsorolt típus megadása 334
 feltétel nélküli vezérlésátadás 387
 feltételes fordítás 325
 feltételkezelő operátor 92

feltételkezelő operátorpár 471
 feof 381
 fgets 382
 Fibonacci-szám 105
 FILE típus 377
 float 26, 365, 450
 fopen 376, 377
 for 56, 483
 fordítási parancs 525
 fordítóprogram 6
 formális paraméternév 127
 formátumkarakter 360
 formátumleíró karakterlánc 19
 formázás 56
 formázott kimenet/bemenet 360
 forráskód 7
 forráskód megjelenítése 419
 forrásprogram 7
 Fortran 6
 Fortran parancsai 6
 sprintf 381
 fputs 382
 free 403
 fscanf 381
 futtatható 9
 függvény 15, 175
 függvény definiálása 477
 függvény fogalma 123
 függvény törzse 128
 függvény visszatérési értéke 130
 függvénydeklaráció 479
 függvények 479
 függvények meghívása 125
 függvények paraméterei 140
 függvényhívások 480
 függvénymutató 481
 függvényprototípus deklarációja 127

G, Gy

gcc 525
 gdb 414
 gépfüggetlen 313

gépi utasításkészlet 5
 getc 378
 getchar 359, 375, 378
 getter 437
 globális adatszerkezet-definíció 177
 globális változók 157
 gnumake 353
 goto 387, 483
 grep 356

H

hármas karakterek 486
 háromszögszámok 45
 hash tables 264
 hasítótáblák 264
 hatókör 476
 help 426
 hexadecimális 364
 hívási verem 424
 hordozható 6
 hozzáférő metódus 437
 hozzáfűzési mód 376
 hozzárendelési operátor 18
 hozzárendelő operátorok 146, 470

I

if 67, 483
 if-else szerkezet 71
 igazságtablázat 287
 include 324
 indentáció 56
 indentálás 77
 indirection 239, 240
 inicializáció 47
 inkrementáló 267, 274
 inkrementáló és dekrementáló
 operátorok 470
 int 25
 interaktív nyomkövetés 415
 írási és olvasási fájlműveletek 372
 írási mód 376

isalpha 235
 isdigit 235
 islower 81, 235
 isupper 81, 235

J

Java 431
 jobbérték 93
 jobbra igazítás 53

K

karakteres változók 231
 karakteres változók egésszé alakítása
 339
 karakterisztika 27
 karakterkezelő függvények 502
 karakterkonstans 450
 karakterlánc bemásolása 200
 karakterlánc-konstans 451
 karakterlánc-konstansok 223
 karakterlánc-mutatók 271
 karakterláncok 199
 karakterláncok beolvasása 210
 karakterláncok inicializálása 205
 karakterszintű bemenet/kimenet 359
 karaktertömb 110
 karaktertömbök 200
 két karakterlánc egyenlősége 208
 két karakterlánc összefűzése 200
 kétdimenziós tömbök inicializációja
 116
 kettes komplexens 285
 kettes számrendszer 285
 kettős karakterek 447
 kiértékelési szint 33
 kifejezések 463
 kilépési állapot 383
 kimenet 10
 kimeneti/bemeneti függvények 503
 kiterjedés 26
 kiterjesztett karakterkészlet 452

kitevő 27, 450
 Komplex aritmetika 520
 konkatenáció 200
 konstans 25
 konstans kifejezés 25
 konstans kifejezések 467
 konstansok 311
 közvetett elérés 239, 240
 közvetett elérés operátor 241
 kulcsszavak 448
 külső változók 346

L

láncolt adatszerkezetek 404
 láncolt listák 249
 lebegőpontos konstans 450
 lebegőpontos szám 18, 23
 lebegőpontos típus 26
 legkisebb helyiértékű bit 285
 legnagyobb helyiértékű bit 285
 legnagyobb közös osztó 60
 léptetés 47
 linkelés 9
 Linux 7
 list 416, 419
 listszie 419
 'llományok átnevezése 384
 logaritmikus gyorskeresés 264
 logaritmikus keresés 227
 logikai „és” 287
 logikai adattípus 28
 logikai operátorok 469
 logikai tagadás 292
 logikai vagy 290
 logikai változók 88
 lokális adatszerkezet-definíció 177
 lokális változó 57
 lokális változók 126, 157
 lokális változók hatóköre 137
 long 30, 449
 long double 31, 450

long int 30
 long long 31, 449
 long long int 365
 LSB 285
 lusta programozók 388

M

Mac OS X 7, 435
 main függvény 124
 make 353
 Makefile 353
 makró 410
 makrók 316, 317
 malloc 400
 mantissa 27, 450
 maradékos osztás 38
 matematikai függvények 512
 mátrixok 115
 megjegyzés 19
 megjegyzések 448
 memóriakezelő függvények 500
 metódusok 432
 mezőszélesség 53
 Microsoft 442
 modul 343
 moduláris programozás 343
 modulok közti kommunikáció 346
 modulus operátor 38
 MSB 285
 mutató 460
 mutatók 23, 239
 mutatók használata 244
 mutatót visszaadó függvény 262
 műveletek fájlokkal 372
 műveletek karakterekkel 231
 műveletek kiértékelési sorrendje 33

N, Ny

nemzetközi karakter 448
 Newton-Raphson iterációs módszer
 135

next 420, 435
 NEXTSTEP 435
 normálalak 27, 450
 növelő (inkrementáló) operátor 51
 NULL 377, 484
 null karakter 203, 272
 null pointer 255
 null utasítás 388
 nullmutató 255, 461
 nyomkövetés 10, 407

0, Ö

Objective-C 431
 objektum 431
 objektum-orientált programozás 431
 oktális 25, 364
 olvasási mód 376
 OOP 431
 operációs rendszerek 6
 operator overloading 442
 operátor túlterhelés 442
 operátorok 464
 osztály 432
 output 10
 összetett betűkonstansok 185, 474
 összetett feltételvizsgálat 73
 összetett utasítás 482

P

paraméter 15
 paraméterek 126, 316
 paraméterszám 395
 paraméter-vektor 395
 parancssor 10
 parancssori paraméterek 394
 paritásvizsgálat 289
 példány 432
 polimorfizmus 433
 pontosságmódosító 71
 postfix 275
 postfix dekrementálás 275
 postfix inkrementálás 275

precedencia 33
 prefix 275
 prefix dekrementálás 275
 prefix inkrementálás 275
 preprocessing 307
 prímszám 88, 106
 print 416
 printf 360, 373
 private 441
 program optimalizálása 269
 programkönyvtárak 9, 356, 454
 programok fordítása 525
 programozási hibák 529
 projektkezelő eszköz 343
 prototípus-deklaráció 341
 putc 378
 putchar 360, 378

Q

quit 419

R

read 376
 realloc 405
 register 393, 463, 476
 rekurzív függvények 163
 relációs kifejezés 47
 relációs operátorok 47, 469
 remove 385
 rename 384
 rendszerhéj 11
 restrict 394, 463
 return 383, 479, 484
 run 419

S, Sz

scanf 53, 360, 367
 sed 356
 segédfüggvények 522
 set 417
 set var 417
 setter 437

shell 11
 short 31, 453
 short int 31, 365
 show args" 419
 signed 32, 449
 signed int 365
 signum 78
 sizeof 451
 sizeof operátor 400, 471
 SmallTalk-80 435
 stack trace 424
 static 161, 349, 476
 statikus függvények 350
 statikus változók 161, 350
 stderr 382
 stdin 382
 stdio.h 382
 stdlib.h 400
 stdout 382, 441
 step 420
 strcat 235
 strcmp 235
 strcpy 235
 strlen 235
 strToInt 235
 struct 169
 struktúra 170
 struktúra tagja 170
 switch 85, 485
 szabványos C programkönyvtár 495
 szabványos fejlécállományok 495
 számítógépes program 5
 számláló 57
 számrendszer 111
 származtatott adattípusok 452, 454
 széles karakterkonstans 451
 szemantikai hiba 8
 szintaktikai hiba 8
 szomszédos karakterláncok" 223
 szökőév 174
 szöveges állomány 7

T, Ty

tagfüggvények 432
 tárgykódú állomány 345
 tárgykódú bináris 9
 tárolási osztály 476
 társíthatósági (asszociativitási) szabály 33
 típusátalakítások 39
 típusátalakító operátor 41, 338, 471
 típusdefiníció 337, 452
 típusdeklaráció 453
 típusminősítők 393
 tizenhatodos pont 450
 többalakúság 433
 többdimenziós tömb 151, 456
 többdimenziós tömbök 115
 többszörös kifejezések 56
 tömb 142
 tömb deklarációja 455
 tömb-inicializáció 455
 tömbmutatók 473
 tömbök 97, 455
 tömbök megadása 98
 tömbök rendezése 148
 töréspont 419
 törtrész 27, 450
 trigraph sequences 486
 typedef 335, 452, 462

U, Ü

újsor karakter 15, 205
 ulimit 414
 UNICODE karakterek 222
 unió 389, 459
 unió deklarációja 389
 uniók adattagjai 390
 union 389, 459
 Unix 7
 unsigned 31, 364, 449
 unsigned int 32
 utasításblokk 50, 482

utasítások 481
üres előfeldolgozói utasítás 493
üres karakterlánc 218

V

vagy 73
valós szám 18
változó deklarációja 17
változó méretű karakterláncok 202
változó méretű tömbök 117, 154, 456
változók 17, 23
változók deklarációja 477
változónevek 23
végtelen ciklus 57
vessző operátor 392, 472
vezérlőkarakterek 220

vezérlősorozatok 28, 450
Visual Studio 343
visszatérési érték 130, 138
vizsgálat 47
void 15, 454, 479
void mutató 402, 461
volatile 394, 463

W

while 58, 486
while ciklus 58
write 376

X

Xcode 343
XOR 291