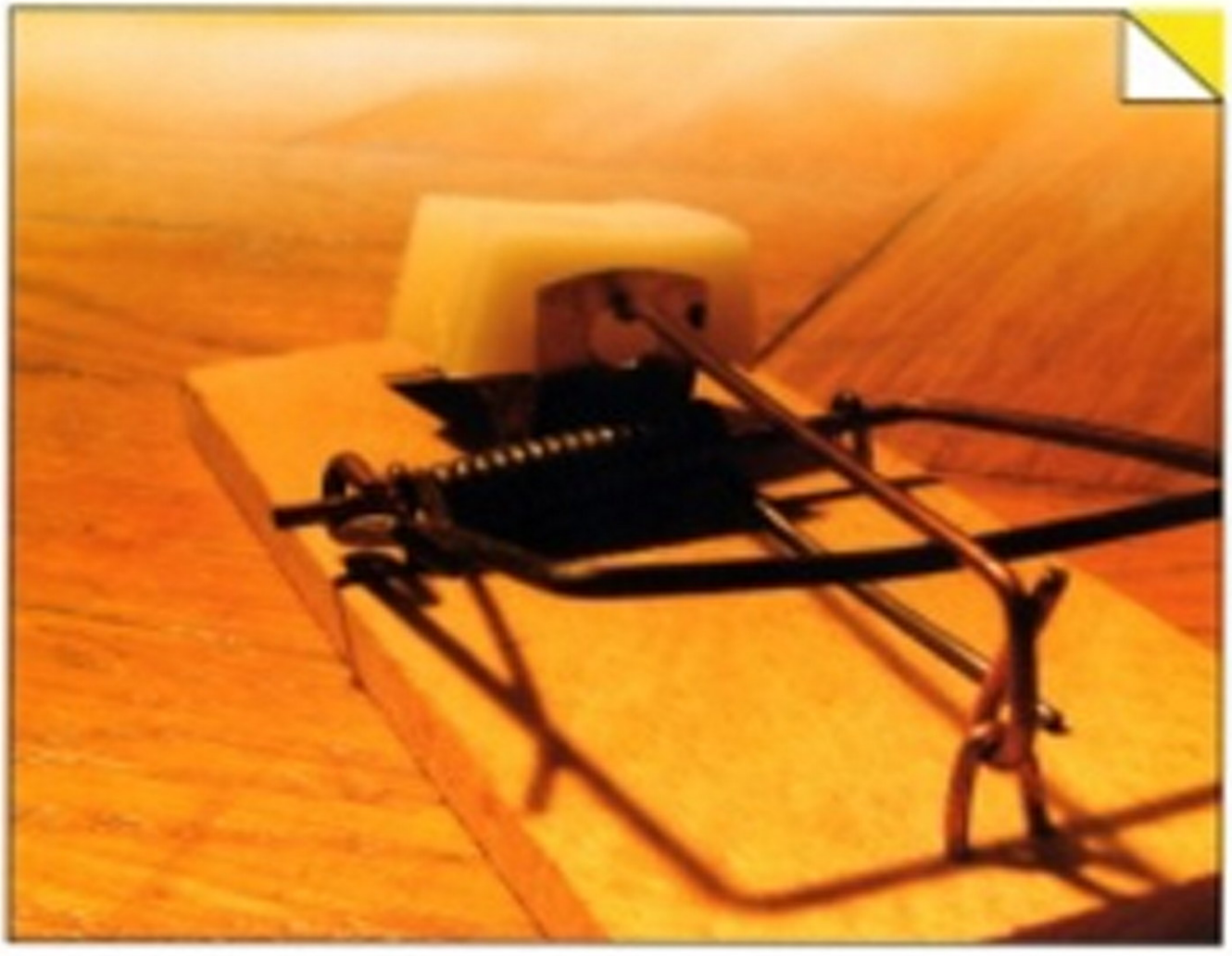
Andrew Koenig

C csapdák és buktatók



KISKAPU

Addison-Wesley

Előszó

Azoknak az eszközöknek a használatát, amelyeket gyakorlás után kényelmesnek találunk, általában nehezebb megtanulni, mint amelyekre azonnal ráérzünk. A tanuló pilóták első útja többnyire hullámvasút, mert időbe telik, amíg rájönnek, milyen finom mozdulatokat igényel a repülés. Segédkerekekkel ellátott biciklin könnyebb megtanulni kerékpározni, de a segédkerekek egy idő után már csak útban vannak.

Így van ez a programozási nyelvekkel is. Minden programozási nyelvnek vannak olyan vonásai, amelyek valószínűleg gondot okoznak azoknak, akik nincsenek minden részlettel tisztában. Ezek a vonások minden nyelvben más-más területen jelentkeznek, mégis minden programozó többnyire ugyanazokba a problémákba botlik. Innen jött az ötlet, hogy ezeket a buktatókat egy kötetbe gyűjtsem.

Első ilyen irányú próbálkozásom 1977-ben történt, amikor *PL/I csapdák és buktatók* címmel előadást tartottam Washingtonban, a SHARE (az IBM nagygépes felhasználói csoportja) találkozóján. Nem sokkal korábban kerültem a Columbia Egyetemről – ahol a PL/I volt a divat – az AT&T Bell Laboratorieshoz, ahol a C nyelvre támaszkodtak. Az ezt követő évtizedben számtalanszor tapasztalhattam, hogy a C programozók (köztük én is) hogyan keverhetik magukat bajba, ha nem tudják biztosan, mit is csinálnak.

A C-ben felmerülő problémákat 1985-ben kezdtem összegyűjteni, és még az év végén belső terjesztésű anyagként közzé is tettem az eredményt. A hatás megdöbbentett: kétezernél is többen kértek belőle példányt a Bell Labs könyvtárában. Ez győzött meg, hogy irományomat könyvvé kell bővítenem.

Mi a könyv célja?

A *C csapdák és buktatók* defenzív programozásra igyekszik bátorítani, azáltal, hogy megmutatja, hogy mások – még a tapasztalt öreg rókák – is bajba kerülhetnek. A bemutatott hibák nagy része könnyen elkerülhető, ha egyszer megértettük a lényegüket, ezért a hangsúlyt a konkrét példákra, és nem az általánosságokra helyeztem.

Ha a C nyelvvel komolyan foglalkozunk, ennek a könyvnek ott a helye a polcunkon, még akkor is, ha profik vagyunk. Több kiváló C programozó, aki látta az első vázlatokat, meglepetéssel fedezett fel bennük olyan hibákat, amelyekbe éppen az azt megelőző héten botlottak. Ha a C nyelvet oktatjuk, az ajánlott olvasmányok között első helyen kell szerepelnie ennek a kötetnek.

Mi nem a könyv célja?

Ez a könyv nem kritizálja a C nyelvet. Programozási hibát bármilyen nyelven véthetünk. A C nyelvvel szerzett évtizedes tapasztalataimat igyekeztem egy tömör kötetbe gyűjteni, abban a reményben, hogy az Olvasó tanul majd mások hibáiból.

A könyv nem kínál kész recepteket. A hibák nem kerülhetők el pusztán recept alapján. Ha így lenne, elég lenne kirakni néhány „Vezessen óvatosan!" plakátot az utak mentén, és nem lenne több autóbaleset. Az ember azonban leghatékonyabban saját, vagy mások kárán okul. Ha megértjük, hogyan fordulhat elő egy adott hiba, már nagy lépést tettünk afelé, hogy a jövőben elkerüljük.

A könyvnek nem célja az sem, hogy megtanítson C-ben programozni. (Erre ott van Kernighan és Ritchie *A C programozási nyelv* című klasszikusa, például Prentice-Hall 1988.) Nem is referenciakönyv (mint Harbison és Steele kötete, a *C: A Reference Manual*, Prentice-Hall 1987). Nem tárgyal algoritmusokat vagy adatszerkezeteket (mint Van Wyk a *Data Structures and C Programs*-ban, Addison-Wesley 1988), és csak érintőlegesen foglalkozik a hordozhatósággal (ezzel kapcsolatban lásd inkább Horton: *How to Write Portable Programs in C*, Prentice-Hall 1989), illetve az operációs rendszeri felületekkel (Kernighan és Pike: *The UNIX Programming Environment*, Prentice-Hall 1984). A bemutatott problémák mind a valós életből származnak, bár gyakran lerövidítettem azokat. (Összetettebb C nyelvű problémákat Feuer könyvében, a *The C Puzzle Book*-ban találunk, Prentice-Hall 1982.) A kötet nem szótár, és nem enciklopédia: szándékosan rövid, hogy senkit ne tartson vissza attól, hogy elolvassa.

Reflektorfényben az Olvasó

Biztosan kihagytam néhány gyakori buktatót. Ha az Olvasó tud ilyet, lépjen velem kapcsolatba az Addison-Wesley kiadón keresztül. Egy későbbi kiadásban lehet, hogy viszontláthatja felfedezését, valamint a nevét a köszönetnyilvánítások között.

Néhány szó az ANSI C-ről

Amikor a kötetet írtam, az ANSI C szabvány még nem volt végleges, ezért tulajdonképpen helytelenül hivatkoztam „az ANSI C"-re, hiszen a bizottság még dolgozott rajta. Az ANSI szabványt azonban a gyakorlatban már jó ideje alkalmazzák, így nem valószínű, hogy bármi, amit az ANSI C-ről állítok, meg fog változni. A bizottság által javasolt jelentősebb fejlesztések nagy részét a könyv írásakor elérhető C fordítókba már beépítették.

Ne aggódjunk, ha az általunk használt megvalósítás nem támogatja a könyvben szereplő ANSI függvényszintaxist. Elég, ha a példák lényegi részeit értjük: a hibákba a használt C-változattól függetlenül bele lehet esni.

Köszönetnyilvánítás

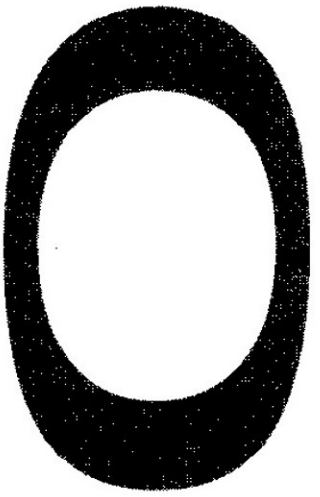
Egy ilyen gyűjteményt nem lehetett volna elkészíteni egyedül. A következő emberek konkrét programozási hibákra mutattak rá: Steve Bellovin (6.3), Mark Brader (1.1), Luca Cardelli (4.4), Larry Cipriani (2.3), Guy Harris és Steve Johnson (2.2), Phil Karn (2.2), Dave Kristol (7.5), George W. Leach (1.1), Doug McIlroy (2.3), Barbara Moo (7.2), Rob Pike (1.1), Jim Reeds (3.6), Dennis Ritchie (2.2), Janet Sirkis (5.2), Richard Stevens (2.5), Bjarne Stroustrup (2.3), Ephraim Vishniac (1.4), és még valaki, aki névtelen szeretne maradni (2.3). A rövidség kedvéért csak az első személyeket említettem meg, akik egy-egy problémára felhívták a figyelmemet. Természetesen nem valószínű, hogy az adott hibákat az említett személyek követték el *először*, és még ha úgy is lenne, nemigen ismernék be... Magam is számos, a könyvben ismertetett hibát elkövettem, egyeseket többször is.

A szerkesztésben Steve Bellovin, Jim Coplien, Marc Donner, Jon Forrest, Brian Kernighan, Doug McIlroy, Barbara Moo, Rob Murray, Bob Richton, Dennis Ritchie, Jonathan Shapiro és számos névtelen kritikus nyújtott hasznos segítséget. Lee McMahon és Ed Sitar a kínos tipográfiai hibák kiszűrésében segítettek, Dave Posser pedig az ANSI C néhány homályosabb pontját tette számomra világosabbá. Brian Kernighan felbecsülhetetlen értékű tördelőeszközökkel látott el, és ugyanilyen nélkülözhetetlen volt a személyes segítsége is.

Nagyszerű volt együtt dolgozni az Addison-Wesley kiadó munkatársaival, akik közül meg kell említenem Jim DeWolf, Mary Dyer, Lorraine Ferrier, Katherine Harutunian, Marshall Henrichs, Debbie Lafferty, Keith Wollman és Helen Wythe nevét, de biztos vagyok benne, hogy munkájukat nagyban segítették azok is, akikkel én személyesen nem találkoztam.

Külön hálával tartozom az AT&T Bell Laboratories felvilágosult vezetőinek, akik egyáltalán lehetővé tették számomra, hogy megírjam ezt a könyvet: Steve Chappell, Bob Factor, Wayne Hunt, Rob Murray, Will Smith, Dan Stanzione és Eric Sumner – köszönöm.

Tartalomjegyzék



Bevezetés

A legelső számítógépes programomat Fortran nyelven írtam 1966-ban. A feladata az lett volna, hogy 10 000-ig kiírja az összes Fibonacci számot, vagyis az 1, 1, 2, 3, 5, 8, 13, 21, ... sorozat elemeit, amelynek soron következő eleme mindig az előző két elem összegeként áll elő. Persze nem működött:

```
      I = 0
      J = 0
      K = 1
    1 PRINT 10, K
      I = J
      J = K
      K = I + J
      IF (K - 10000) 1, 1, 2
    2 CALL EXIT
   10 FORMAT (I10)
```

A Fortran-programozók bizonyára rögtön kiszúrják, hogy a programból hiányzik egy `END` utasítás. A programot viszont akkor sem tudtam lefordítani, amikor beleírtam a hiányzó `END` utasítást. Ehelyett kaptam egy rejtélyes `ERROR 6` hibaüzenetet.

Ezután töviről hegyire átolvastam a kézikönyvet, és rájöttem, hogy mi a gond. Az a Fortran fordítóprogram, amit használtam, nem tudta kezelni a négynél több jegyből álló egész állandókat. A `10000` helyére `9999`-et tettem, és ezzel a gond megoldódott.

Legelső C programomat 1977-ben írtam. Persze nem működött:

```
#include <stdio.h>

main()
{
        printf("Hello world")
}
```

A program fordítása első próbálkozásra sikerült, az eredmény azonban elég furcsa lett. A terminál képernyőjén valami ilyesmi jelent meg:

```
% cc prog.c
% a.out
Hello world%
```

A `%` karakter a rendszer *készenléti jele* volt, tehát az a karakterlánc, amivel a rendszer a tudtomra hozta, hogy nekem kell begépelnem valamit. A `%` jel közvetlenül a `Hello world` üzenet után jelent meg, mert elfelejtettem megmondani a rendszernek, hogy kezdjen egy új sort. A 3.10. részben a program egy ennél is finomabb hibájával fogunk megismerkedni.

A két probléma közötti különbség ég és föld. A Fortran példában két hiba is volt, de a fordító elég jól működött, és figyelmeztetett rájuk. A C programmal technikailag semmi gond nem volt, a gép szemszögéből tökéletesen működött, ezért hibaüzenetet sem kaptam. A gép azt tette, amit mondtak neki, csak az nem egészen az volt, amire én gondoltam.

Ez a könyv a második fajtába tartozó problémákkal foglalkozik, vagyis az olyan programokkal, amelyek nem úgy viselkednek, ahogy azt a programozó elképzelte. Ezen belül is olyan bakikkal foglalko-

zunk majd, amelyek kimondottan a C nyelven írt programokban fordulhatnak elő. Vegyük például a következő programrészletet, amivel egy N elemű tömb kezdőértékeit szeretnénk beállítani:

```
int i;
int a[N];
for (i = 0; i <= N; i++)
            a[i] = 0;
```

Sok olyan C megvalósítás létezik, amelyen ez a program végtelen ciklust eredményez! A 3.6. részből azt is megtudhatjuk, miért.

A hibák a program azon részeit jelentik, ahol annak működése eltér attól, amit a programozó elképzelt. Tehát éppen a természetüknél fogva nehéz rendszerezni őket. Én a programok különféle szemszögből való vizsgálatát vettem a csoportosítás alapjaként.

A legalacsonyabb szinten egy program nem más, mint *szimbólumok*, más néven *tokenek* (alapelemek) sorozata, éppen úgy, ahogy a könyveket betűk sorozata alkotja. Azt a folyamatot, amelynek során a programot szimbólumokra tördeljük, *lexikai elemzésnek* nevezik. Az 1. fejezetben olyan problémákat veszünk szemügyre, amelyek abból adódnak, ahogy a C a lexikai elemzést végzi.

A programokat felépítő tokenekre utasításokként és meghatározásokként is tekinthetünk, éppen úgy, ahogy a szavak sorozatából előáll egy könyv. Mindkét esetben attól függ a program, illetve a könyv jelentése, hogy a tokenek, illetve a szavak mely kombinációiból építünk fel nagyobb egységeket. A 2. fejezetben azokat a hibákat vizsgáljuk meg, amelyek a *nyelvtan* (szintaxis) félreértéséből származnak.

A 3. fejezet a jelentés (szemantika) fogalomzavaraival foglalkozik. Számtalanszor előfordul, hogy a program jelentése teljesen eltér attól, amit a programozó eredetileg mondani szeretett volna. Ebben a fejezetben abból indulunk ki, hogy a nyelv lexikális és szintaktikai elemei teljesen világosak, és csak a *jelentésre* összpontosítunk.

A 4. fejezet arra világít rá, hogy a C programok gyakran több részből állnak, és ezeket a részeket előbb külön-külön fordítjuk le, majd ezután kapcsoljuk össze a részeket. Ezt a folyamatot *összekapcsolásnak* nevezzük, ami része a program és annak környezete között létrejövő kölcsönhatásnak.

Ebben a környezetben találjuk a *könyvtári függvényeket* is. Habár ezek szigorúan véve nem részei a nyelvnek, minden valamirevaló C programnak szüksége van a könyvtári függvényekre. Létezik néhány olyan könyvtári függvény, amit majdnem minden C program felhasznál, és amelyek elég sok gondot okoznak ahhoz, hogy ezzel kiérdemelték, hogy az 5. fejezetben kizárólag róluk essék szó.

A 6. fejezetből megtudhatjuk, hogy a program, amit megírunk, nem pont ugyanaz, mint a program, amit futtatunk, mert azt először az előfeldolgozó veszi kezelésbe. Az előfeldolgozó különféle megvalósításai nem egyformák, de elég sok közös vonásuk van ahhoz, hogy hasznos tanácsokat adjunk velük kapcsolatban.

A 7. fejezetben a hordozhatóság problémáit vesszük górcső alá, vagyis azt, hogy egy program miért működik az egyik gépen tökéletesen, a másikon meg egyáltalán nem. Meg fogunk lepődni, hogy milyen nehézkes még az olyan egyszerű dolgok megvalósítása is, mint az egészekkel végzett számtani műveletek.

A 8. fejezetben egy csokor megszívlelendő jótanácsot nyújtunk át az olvasónak, amelyek segítenek megelőzni a programozási hibákat. Ebben a fejezetben találjuk a többi fejezet gyakorlatainak megoldásait is.

Végül a mellékletben három közismert, de gyakran félreértésekre okot adó könyvtári szolgáltatást mutatunk be.

0.1. gyakorlat. Vásárolnánk-e autót egy olyan gyártótól, akiről tudjuk, hogy sokszor visszahívta már felülvizsgálatra a korábban gyártott modelljeit? Megváltozna-e a véleményünk, ha a gyártó azt állítaná, hogy ez már a múlté? Mi a *valódi* ára annak, ha a felhasználóink találják meg a hibákat a programjainkban?

0.2. gyakorlat. Hány darab egymástól 10 méterre levert kerítésoszlopra van szükségünk ahhoz, hogy kifeszítsünk egy 100 méteres kerítést?

0.3. gyakorlat. Volt-e rá példa, hogy megvágtuk magunkat egy késsel főzés közben? Hogyan lehetne biztonságosabbá tenni a főzéshez használt késeket? Használnánk-e egy ekként átalakított kést?

1

Lexikális buktatók

Amikor egy mondatot olvasunk, általában nem figyelünk oda a mondatot alkotó szavak minden egyes betűjére. Ez érthető, hiszen a betűk önmagukban nem sokat jelentenek. Szavakat alkotunk belőlük, és a szavakat ruházzuk fel jelentéssel.

Így van ez a C és más nyelveken írt programoknál is. A programból kiragadott karakterek önmagukban semmit sem jelentenek, csak ha valamilyen környezetben használjuk őket. Így a

```
p->s = "->";
```

kifejezésben a - karakter két példánya két különböző dolgot jelent. Precízebben fogalmazva a - karakter példányai különböző lexikális elemek, idegen szóval *tokenek* részei. Az első példány a -> műveleti jel, a második egy karakterlánc része, a -> elem jelentése pedig az őt felépítő mindkét karakter jelentésétől eltér.

A *token* szó a program egy olyan elemére vonatkozik, ami a szerepét tekintve a mondat szavaira hasonlít, tehát bizonyos értelemben minden egyes előfordulásakor ugyanazt jelenti. Ugyanaz a karaktersorozat két különböző környezetben két teljesen eltérő lexikális elemhez is tartozhat. A fordítóprogramnak azt a részét, ami a programot tokenekre tördeli, gyakran *lexikális elemzőnek* is nevezik.

Nézzünk most egy másik példát:

```
if (x > big) big = x;
```

Ebben az utasításban az első lexikális elem az `if` kulcsszó. A következő elem a nyitó zárójel, amit az `x` azonosító, a „nagyobb mint" jel, a `big` azonosító stb. követnek. A C nyelvben mindig megtehetjük, hogy elválasztó karaktereket (szóköz, tabulátor vagy újsor karaktereket) teszünk a tokenek közé, tehát ezt is írhattuk volna:

```
if
(
x
>
big
)
big
=
x
;
```

Ebben a fejezetben néhány gyakori félreértést tisztázunk a tokenek jelentésével, egymáshoz való viszonyukkal és az őket felépítő karakterekkel kapcsolatban.

1.1. Az = nem ==

Az Algol nyelv legtöbb utódja, mint a Pascal és az Ada, a `:=` jelet használja az értékadáshoz és az `=` jelet az összehasonlításhoz. A C értékadó jele ezzel szemben az `=`, összehasonlító jele pedig az `==`, ami kényelmes megoldás: az értékadást sokkal gyakrabban használjuk, mint az összehasonlítást, így a rövidebb jelet kell gyakrabban leírnunk. Ezen kívül a C az értékadást műveletként (operátorként) kezeli, így egyszerre több értékadást is könnyen leírhatunk (például `a=b=c`), illetve összetettebb kifejezésekbe is beágyazhatunk értékadásokat.

A kényelemnek azonban egy lehetséges problémaforrás az ára: az ember akaratlanul is értékadás jelet írhat ott, ahol összehasonlítást

szeretne elvégezni. Így a következő utasítás, ami látszólag végrehajt egy `break` utasítást, ha `x` egyenlő `y` értékével,

```
if (x = y)
        break;
```

valójában `x` értékét egyenlővé teszi `y` értékével, majd ellenőrzi, hogy az érték különbözik-e nullától. Vagy nézzük meg a következő ciklust, aminek az lenne a feladata, hogy átugorja a fájlban található szóköz, tabulátor és újsor karaktereket:

```
while (c = ' ' || c == '\t' || c == '\n')
        c = getc(f);
```

A ciklus hibásan az = jelet használja az == jel helyett a `' '` ellenőrzésénél. Mivel az = művelet a `||` műveleti jel után következik a műveletek végrehajtási sorrendjében, az „összehasonlítás" valójában a `c` változóba menti az alábbi kifejezés értékét:

```
' ' || c == '\t' || c == '\n'
```

A `' '` értéke nullától különböző, így a kifejezés értéke `1` lesz, függetlenül attól, hogy a `c` értéke (korábban) mi volt. A ciklus így beszippantja az egész fájlt. Hogy mi történik ezután, az csak azon múlik, hogy az adott megvalósítás megengedi-e, hogy a program folytassa az olvasást, miután elérte a fájl végét. Ha igen, akkor végtelen ciklust kapunk.

Néhány C fordító megpróbál segíteni a felhasználóknak, és figyelmeztetést küld, ha *e1* = *e2* formájú feltétellel találkozik. Ha az a célunk, hogy értéket adjunk egy változónak, és utána ellenőrizzük, hogy az érték nulla-e vagy sem, akkor érdemes félreértést kizáró módon megadni az összehasonlítást, hogy elkerüljük a figyelmeztetést, amit ezek a fordítók küldenének. Egyszóval ehelyett:

```
if (x = y)
        foo();
```

írjuk ezt:

```
if ((x = y) != 0)
        foo();
```

Ezzel a szándékaink is nyilvánvalóbbak lesznek. A 2.2. részben megbeszéljük, miért van szükség az `x = y` kifejezést körülvevő zárójelekre.

Fordítva is könnyen összekeverhetjük a dolgokat:

```
if ((filedesc == open(argv[i], 0)) < 0)
        error();
```

A fenti példa `open` függvényének visszatérési értéke `-1` lesz, ha hibát észlel, és nullát vagy pozitív számot ad vissza, ha sikerrel járt. A kódrészlet szándékaink szerint a `filedesc` változóba menti az `open` függvény eredményét, és egyben ellenőrzi a művelet sikerességét is. Az első `==` jel helyett azonban `=` jelet kellett volna írnunk. A jelenlegi formájában a kód összehasonlítja az `open` függvény eredményét a `filedesc` értékével, és ellenőrzi, hogy az összehasonlítás értéke negatív-e. Persze sosem lesz az, hiszen az `==` eredménye mindig `0` vagy `1`, tehát nem negatív. Az `error` függvény így sohasem hívódik meg. Bár úgy tűnik, hogy minden rendben, a `filedesc` értéke ugyanaz marad, ami volt, és semmi köze sem lesz az `open` eredményéhez. Néhány fordító jelezheti, hogy a nullával történő összehasonlításnak nincs hatása, de ne nagyon számítsunk erre.

1.2. Az & és a | nem && és ||

Könnyen elnézhetjük az `=` és az `==` jelek akaratlan felcserélését, hiszen sok más nyelvben az `=` jelet használják az összehasonlításhoz. Az `&` és `&&` jeleket, illetve a `|` és a `||` jeleket is könnyen összecserélhetjük, különösen azért, mert az `&` és a `|` jeleket a C nyelvben másra használjuk, mint néhány másik nyelvben. E műveleti jelek pontos jelentését a 3.8. részben vizsgáljuk meg.

1.3. Mohó lexikális elemzés

Néhány C token, például a /, a * és az = hosszúsága mindössze egy karakter. Néhány más C token, például a /*, az == és az azonosítók több karakter hosszúak. Amikor a C fordító találkozik egy / jellel, amit egy * jel követ, el kell tudnia dönteni, hogy a két karaktert két külön vagy egyetlen elemként kell-e kezelnie. A C ezt a kérdést egy egyszerű szabály segítségével oldja meg: *újra és újra válasszuk le a lehető legnagyobb darabot.* Vagyis a C programot úgy bonthatjuk szét tokenekre, ha balról jobbra haladva minden egyes alkalommal a lehető leghosszabb tokent vesszük. Ezt gyakran *mohó* stratégiának, vagy az angol szakzsargonban *maximális majszolásnak* (maximal munch) is nevezik. Kernighan és Ritchie így fogalmaztak: „Ha a bemeneti adatfolyamot egy adott karakterig tokenekre bontottunk, akkor a következő token az a leghosszabb karakterlánc lesz, ami egy tokenként értelmezhető." A tokenek (a karakterláncok és a karakterállandók kivételével) sohasem tartalmazhatnak üres elválasztó karaktereket (szóközöket, tabulátorokat és újsor karaktereket).

Így az == jel például egyetlen token, míg az = = kifejezés két külön token, az alábbi kifejezés pedig

```
a---b
```

ugyanazt jelenti, mint az

```
a -- -b
```

és nem azt, hogy

```
a - -- b
```

Hasonlóképpen, ha egy / jel a token első karaktere és utána rögtön egy * következik, akkor a két karakter után egy megjegyzés áll, *függetlenül* a környezet további részeitől.

A következő utasítás látszólag az y változóba menti azt az értéket, amit úgy kapunk meg, ha x értékét elosztjuk azzal az értékkel, amire a p mutat:

```
y = x/*p;        /* p az osztóra mutat */
```

Valójában azonban a /* egy megjegyzés kezdetét jelzi, így a fordító egészen a */ megjelenéséig átugrik a kódon. Más szóval, az utasítás az y változóba menti x értékét, a p értékére pedig rá se hederít.
Ha átírjuk a kódot így:

```
y = x / *p;      /* p az osztóra mutat */
```

vagy így:

```
y = x/(*p);      /* p az osztóra mutat */
```

akkor már elvégzi azt az osztást, amire a megjegyzés utal.

Ez a fajta többé-kevésbé kétértelmű helyzet máshol is bajt okozhat. Korábban például a C a =+ jelet használta arra, amit ma a += jellel jelölünk. Néhány C fordító továbbra is elfogadja az elavult formát. Egy ilyen fordító számára az

```
a=-1;
```

jelentése megegyezik azzal, hogy

```
a =- 1;
```

ami ugyanaz, mint

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

Ez komoly meglepetést okozhat annak a programozónak, aki azt akarta írni, hogy

```
a = -1;
```

Ez a fajta régimódi fordító ezt a kifejezést

```
a=/*b;
```

is úgy értelmezi, hogy

```
a =/ * b ;
```

annak ellenére, hogy a `/*` megjegyzésnek néz ki.

Ezek a régebbi fordítók az összetett értékadásokat is két tokenként kezelik. Egy ilyen fordítónak az

```
a >> = 1;
```

kezelése nem okoz gondot, de egy szabványos ANSI C fordító elutasítja ezt a kifejezést.

1.4. Egész állandók

Ha egy egész állandó első karaktere a `0` számjegy, akkor a szám nyolcas számrendszerű (oktális). A `10` és a `010` tehát két teljesen különböző számot jelöl. Számos C fordító gond nélkül elfogadja a `8`-as és a `9`-es számjegyeket „oktális" számokban. Egy ilyen szám értékét a nyolcas számrendszer szabályaiból vezethetjük le. Például: a `0195` jelentése $1x8^2+9x8^1+5x8^0$, aminek az értéke a tízes (decimális) számrendszerben `141`, a nyolcas (oktális) számrendszerben pedig `0215`. Természetesen nem javasoljuk ezt a fajta használatot, az ANSI C szabvány pedig egyenesen tiltja.

Figyeljünk oda arra is, nehogy akaratunk ellenére használjunk oktális értékeket az alábbihoz hasonló helyzetekben:

```
struct {
          int part_number;
          char *description;
} parttab[] = {
          046, "left-handed widget"
```

```
            047,  "right-handed widget"   ,
            125,  "frammis"
};
```

1.5. Karakterláncok és karakterek

Az idézőjelek és az aposztrófok egymástól teljesen eltérő dolgokat jelentenek a C nyelvben, és ha összekeverjük őket, akkor a jutalmunk hibaüzenetek helyett vaskos meglepetés lehet.

Ha aposztrófok közé zárunk egy karaktert, az ugyanaz, mintha azt az egész értéket írnánk le, ami az adott karakter sorszáma az adott megvalósítás jelsorrendjében. Egy ASCII megvalósításban például az `'a'` jelentése ugyanaz, mint a `0141` vagy a `97`.

Az idézőjelekkel körbevett karakterlánc viszont egy olyan mutató rövidítésére szolgál, ami egy olyan név nélküli tömb első karakterére mutat, amely kezdetben az idézőjelek közötti karaktereket tartalmazza, valamint még egy karaktert, amelynek bináris értéke nulla.

Az alábbi utasítás tehát

```
printf("Hello world\n");
```

egyenértékű ezzel:

```
char hello[] = {'H', 'e', 'l', 'l', 'o', ' ',
                'w', 'o', 'r', 'l', 'd', '\n', 0};
printf(hello);
```

Mivel az aposztrófok közötti karakter egy egész értéket rejt, az idézőjelek közti karakter pedig egy mutatót, a fordító a típusellenőrzés során általában észreveszi, ha valahol az egyiket használjuk a másik helyett. Ha például azt írjuk, hogy

```
char *slash = '/';
```

akkor hibaüzenetet kapunk, mert a `'/'` nem karaktermutató. Néhány fordító azonban nem ellenőrzi a paraméterek típusait, ami különösen gyakori a `printf` függvénynek átadott értékeknél. Így ha azt írjuk, hogy

```
printf('\n');
```

ahelyett, hogy

```
printf("\n");
```

azzal kellemetlen meglepetést okozhatunk magunknak futásidőben, ahelyett, hogy a fordító lefülelné a problémát. A 4.4. részben részletesen foglalkozunk más hasonló esetekkel is.

Mivel egy egész érték általában elég nagy ahhoz, hogy több karakter is elférjen benne, néhány C fordító megengedi, hogy egyszerre több karaktert tároljunk egy karakterállandóban és a karakterlánc-állandókban is. Ez egyben azt is jelenti, hogy egy `"yes"` helyett szereplő `'yes'` észrevétlenül maradhat a programban. Az előbbi azt jelenti, hogy „az `y`, `e`, `s` és egy null karaktert tartalmazó, négy egymást követő memóriahely közül az elsőnek a címe". A `'yes'` jelentése nincs egyértelműen meghatározva, de sok C megvalósítás számára a jelentése egyenlő „azzal az egész számmal, amely valahogyan az `y`, `e`, és `s` karakterek értékéből áll elő". Ha a két érték egybeesik, az kizárólag a véletlen műve.

1.1. gyakorlat. Néhány C fordító megengedi a megjegyzések beágyazását. Írjunk egy C programot, ami *hibaüzenetek nélkül* megvizsgálja, hogy ilyen fordítónk van-e vagy sem. Más szóval a programnak működnie kell mind a kétféle megjegyzésszabály esetén, de mindkét esetben mást kell tennie. *Tipp:* a `/*` megjegyzésjel egy idézőjeles karakterlánc belsejében csak a karakterlánc része, egy megjegyzés belsejében az `" "` idézőjelek között lévő szöveg pedig a megjegyzés része.

1.2. gyakorlat. Ha nekünk kellene írni egy C fordítót, engedélyeznénk-e a beágyazott megjegyzéseket? Ha olyan C fordítót használunk, ami megengedi a megjegyzések beágyazását, használnánk-e ezt a szolgáltatást? A második kérdésre adott válasz befolyásolja az elsőre adott választ?

1.3. gyakorlat. Miért egyenértékű az `n-->0` kifejezés azzal, hogy `n-- >`, és nem azzal, hogy `n- ->`?

1.4. gyakorlat. Mit jelent az `a+++++b` kifejezés?

2

Szintaktikai buktatók

Az még nem elegendő a C programok megértéséhez, ha az őket alkotó lexikális elemek (tokenek) jelentésével tisztában vagyunk. Azt is tudnunk kell, hogy a tokenek milyen kombinációi alkotnak egyes kifejezéseket, utasításokat és programokat. Habár ezek a kombinációk általában elég jól meghatározottak, az is előfordul, hogy egy meghatározás logikátlan és csak összezavarja az embert. Ebben a fejezetben néhány kevésbé nyilvánvaló nyelvtani (szintaktikai) szerkezetet vizsgálunk meg.

2.1. A függvénydeklarációk értelmezése

Egyszer beszéltem egy C programozóval, aki olyan C programot írt, ami önállóan futott egy mikroprocesszoron. Amikor bekapcsolták a gépet, a hardver meghívta azt az eljárást, aminek a címét a nulla sorszámú helyen tárolták.

Ahhoz, hogy szimulálni tudjuk a gép bekapcsolását, olyan C utasítást kellett kitalálnunk, ami közvetlenül meghívja az eljárást. Némi gondolkodás után a következő kifejezést hoztuk össze:

```
(*(void(*)())0)();
```

Az efféle kifejezésektől a legtöbb C programozónak feláll a szőr a hátán. Ennek ellenére mi magunk is egész könnyedén elkészíthetünk egy ilyen kifejezést, ha betartunk egy egyszerű szabályt: *aszerint írjuk meg, amire használni fogjuk.*

A C változók típusának meghatározására mindig két részből áll: egy típusból és egy sor kifejezésszerű dologból, amiket *bevezetőknek* (deklarátorok, declarator) hívnak. A bevezetők valahogy úgy néznek ki, mint egy kifejezés, aminek adott típusú értéke a kiértékelés során. A legegyszerűbb deklarátorok a változók. A

```
float f, g;
```

azt jelenti, hogy az `f` és `g` kifejezések kiértékelésükkor lebegőpontos (`float`) típusúak lesznek. Mivel a deklarátorok olyanok, mint a kifejezések, szabadon használhatjuk velük a zárójeleket is. A

```
float ((f));
```

kifejezés azt jelenti, hogy az `((f))` lebegőpontos lesz, amiből az következik, hogy az `f` szintén lebegőpontos.

A függvény- és mutatótípusok is hasonló logikát követnek. Az, hogy

```
float ff();
```

például azt jelenti, hogy az `ff()` kifejezés lebegőpontos típusú, következésképpen az `ff` függvény visszatérési értéke szintén lebegőpontos szám. Hasonlóképpen, a

```
float *pf;
```

kifejezés azt jelenti, hogy a `*pf` lebegőpontos típusú, azaz a `pf` egy lebegőpontos számra mutat. Ezeket az alakokat a változók bevezetésénél úgy kombinálhatjuk, mint a kifejezésekben. A

```
float *g(), (*h)();
```

szerint tehát a `*g()` és a `(*h)()` lebegőpontos kifejezések. Mivel a `()` előrébb van a műveletek sorrendjében, mint a `*`, a `*g()` ugyanazt jelenti, mint a `*(g())`. A `g` olyan függvény, ami egy lebegőpontos számot címző mutatót ad vissza, a `h` pedig olyan mutató, ami egy lebegőpontos értéket visszaadó függvényre mutat.

Ha már tudjuk, hogyan kell bevezetnünk egy adott típusú változót, akkor könnyen írhatunk típusátalakítást is ahhoz a típushoz. Ehhez egyszerűen távolítsuk el a változó nevét és a típusmeghatározását lezáró pontosvesszőt, majd az egészet tegyük zárójelek közé. Mivel a

```
float (*h)();
```

bevezetés szerint a `h` mutató olyan függvényre mutat, ami lebegőpontos értéket ad vissza, a

```
(float (*)())
```

kifejezés egy típusátalakítás lesz, amivel olyan mutatótípusra válthatunk, ami egy lebegőpontos értéket visszaadó függvényre mutat.

Most már két lépcsőben elemezhetjük a `(*(void(*)())0)()` kifejezést. Először is tegyük fel, hogy van egy `fp` változónk, ami egy függvénymutatót tartalmaz, és meg akarjuk hívni azt a függvényt, amire az `fp` mutat. Ezt így tehetjük meg:

```
(*fp)();
```

Ha `fp` egy függvényre mutat, akkor `*fp` maga a függvény, tehát azt a `(*fp)()` kifejezéssel hívhatjuk meg. Az ANSI C megengedi, hogy ezt az `fp()` kifejezéssel rövidítsük, de ne felejtsük, hogy ez csak rövidítés.

A `*fp` körüli zárójelek a `(*fp)()` kifejezésben létfontosságúak, hiszen a függvényhívás előrébb van az elsőbbségi (műveleti) sorrendben, mint az egytényezős operátorok. A zárójelek nélkül a `*fp()` ugyanazt jelenti, mint a `*(fp())`. Az ANSI C ezt a `*((*fp)())` kifejezés rövidítéseként értelmezi.

Most már csak egy megfelelő kifejezésre van szükségünk, amit az `fp` helyére írhatunk, és meg is oldottuk a problémát. Ez lesz az elemzésünk második része. Ha a C képes lenne a gondolatainkból kiolvasni az adattípusokat, akkor írhatnánk azt, hogy:

```
(*0)();
```

Ez így nem működik, mert a * műveleti jel megköveteli, hogy egy mutatóval együtt használjuk. A mutatónak ezen felül egy függvényre kell mutatnia, hogy a * művelet eredményét meghívhassuk. Ezért a `0` értéket olyan típusra kell alakítanunk, amit nagyjából úgy fogalmazhatunk meg, hogy „egy mutató, ami egy üres elemet (`void`) visszaadó függvényre mutat".

Ha `fp` olyan mutató, ami üres elemet visszaadó függvényre mutat, akkor a `(*fp)()` egy üres elem, aminek a bevezetése így nézne ki:

```
void (*fp)();
```

Így azt is írhatjuk, hogy

```
void (*fp)();
(*fp)();
```

egy fiktív változó bevezetése árán. De ha már egyszer be tudjuk vezetni a változót, akkor egy állandót is át tudunk alakítani erre a típusra, csak hagyjuk el a bevezetésből a változó nevét. A `0` típusát tehát a következőképpen alakíthatjuk olyan mutatóra, ami üres elemet visszaadó függvényre mutat:

```
(void(*)())0
```

Most pedig az `fp` helyére beírhatjuk a `(void(*)())0` kifejezést:

```
(*(void(*)())0)();
```

A sor végét lezáró pontosvessző segítségével pedig utasítás lesz a kifejezésből.

Abban az időben, amikor megoldottam ezt a problémát, még nem léteztek a `typedef` típusdeklarációk. A részletek elemzéséhez nem árt ugyan, ha a `typedef` nélkül megyünk végig a példán, de a `typedef` segítségével minden sokkal világosabb:

```
typedef void (*funcptr)();
(*(funcptr)0)();
```

Ennek a rendetlen példának számos társa akad, amelyekkel a C programozók gyakran találkozhatnak. Itt van például a `signal` könyvtári függvény. Azokban a C-megvalósításokban, amelyekben szerepel ez a függvény, két paramétere van. Az egyik egy egész típusú kód, ami az elfogandó jelzést azonosítja, a másik pedig egy mutató, ami egy olyan üres elemet visszaadó felhasználói függvényre mutat, ami a jelzést kezeli. Az 5.5. részben részletesen is foglalkozunk ezzel a függvénnyel.

A programozók általában nem saját maguk vezetik be a `signal` függvényt, hanem a rendszer `signal.h` fejlécállományában található deklarációt használják. Hogy vezeti be a `signal` függvényt ez a fejlécállomány?

A legegyszerűbb, ha azzal kezdjük, hogy átgondoljuk a felhasználói jelzéskezelő függvényt, aminek a meghatározása valahogy így nézhet ki:

```
void
sigfunc(int n)
{
        /*itt történik a jelzések kezelése*/
}
```

A `sigfunc` függvény paramétere egy egész szám, ami a jelzést azonosítja. Ezzel egyelőre nem foglalkozunk.

A fenti (képzeletbeli) függvény törzse tartalmazza a `sigfunc` meghatározását. Bevezetni így tudjuk a függvényt:

```
void sigfunc(int);
```

Most tegyük fel, hogy be akarjuk vezetni az `sfp` változót, ami a `sigfunc` függvényre mutat. Ha az `sfp` a `sigfunc` függvényre mutat, akkor a `*sfp` maga a `sigfunc` függvény, s így a `*sfp` kifejezést meg lehet hívni. Ha `sig` egész (`int`) érték, `(*sfp)(sig)` üres elem, az `sfp` bevezetése tehát így néz ki:

```
void (*sfp)(int);
```

Ez azt is rögtön megmutatja, hogyan kell bevezetnünk a `signal` függvényt. Mivel a `signal` ugyanolyan típusú értéket ad vissza, mint az `sfp`, a bevezetésének így kell kinéznie:

```
void (*signal(valami))(int);
```

A *valami* a `signal` paramétereinek típusára vonatkozik, aminek a megírásával még meg kell ismerkednünk. Ezt a bevezetést úgy értelmezhetjük, hogy ha a megfelelő paraméterekkel meghívjuk a `signal` függvényt, és a visszaadott mutató által kijelölt függvényt meghívjuk egy egész paraméterrel, akkor egy üres elemet kapunk vissza. A `signal` tehát minden bizonnyal olyan függvény, ami olyan mutatót ad vissza, ami üres elemet visszaadó függvényre mutat.

Mi a helyzet magának a `signal` függvénynek a paramétereivel? Azt szeretnénk, hogy a `signal` két paramétert fogadjon. Egy, a jelzést azonosító egész értéket és egy, a jelzést kezelő felhasználói függvényt címző mutatót. Eredetileg a jelzést kezelő függvényt címző mutatót így vezettük be:

```
void (*sfp)(int);
```

Az `sfp` típusát úgy kapjuk meg, ha elhagyjuk az `sfp` kifejezést annak bevezetéséből, hogy csak a `void(*)(int)` kifejezés maradjon. A `signal` függvény továbbá egy mutatót ad vissza, ami az adott típusú jelzést kezelő előző függvényre mutat. Ez a mutató szintén egy `sfp` lesz. A `signal` függvényt tehát így vezethetjük be:

```
void (*signal(int,void(*)(int)))(int);
```

A typedef segítségével nagyban egyszerűsíthetjük a bevezetést:

```
typedef void (*HANDLER)(int);
HANDLER signal(int,HANDLER);
```

2.2. A műveleti sorrend nem mindig az, amit szeretnénk

Tegyük fel, hogy FLAG néven megadtunk egy állandót, ami egy olyan egész érték, aminek pontosan egy bitjét kapcsoltuk be kettes számrendszerbeli ábrázolásában (más szóval az érték a kettő hatványa). Azt szeretnénk megvizsgálni, hogy a flags nevű egész változónak be van-e kapcsolva ez a bizonyos bitje. A szokásos módon ezt így írjuk le:

```
if (flags & FLAG) ...
```

Ennek a kifejezésnek a jelentése a legtöbb C programozó számára egyértelmű: az if utasítás megvizsgálja, hogy a zárójelek között lévő kifejezést kiértékelve az eredmény 0-e. A pontosabb dokumentálás kedvéért nem árthat, ha még egyértelműbben fogalmazzuk meg ezt a vizsgálatot:

```
if (flags & FLAG != 0) ...
```

Az utasítás így már könnyebben érthető. Sajnos hibás is, mert a != előrébb van a műveleti sorrendben, mint a & művelet, a kifejezést tehát így kell értelmeznünk:

```
if (flags & (FLAG != 0)) ...
```

Ez csak akkor fog működni ha (tiszta véletlenségből) a FLAG értéke 1, egyébként nem.

Tegyük fel, hogy van két egész változónk, a hi és a low, amelyeknek az értéke 0 és 15 közötti (a két határértéket is beleértve), és az r egész változónak egy olyan 8 bites értéket akarunk adni, amelynek az

alacsony helyiértékű bitjei megegyeznek a `low` változó alacsony helyiértékű bitjeivel, a magas helyiértékű bitjei pedig a `hi` változó magas helyiértékű bitjeivel. A következő írásmód szinte kínálja magát:

```
r = hi<<4 + low;
```

Sajnos ez így helytelen. Az összeadás előrébb van a műveleti sorrendben, mint az eltolás, tehát a fenti példa az alábbi kifejezéssel egyenértékű:

```
r = hi << (4 + low);
```

Nézzünk meg két helyes megoldást is. A második azt sugallja, hogy a gondok gyökere a számtani és logikai műveletek keveréséből ered. Az eltolásnak és a logikai műveleteknek a műveleti sorrendben elfoglalt egymáshoz viszonyított helyzete ésszerűbb magyarázatnak tűnik:

```
r = (hi << 4) + low;
r = hi << 4 | low;
```

Az egyik módja annak, hogy elkerüljük ezeket a problémákat, ha mindenhová zárójeleket teszünk, a túl sok zárójelet tartalmazó kifejezések azonban nehezen érthetők. Ezért célszerű megjegyezni a műveletek végrehajtási sorrendjét a C nyelvben.

Sajnos tizenöt különböző elsőbbségi szint van, ami bizony megnehezíti a dolgunkat. A teljes táblázatot az alábbiakban közöljük.

Könnyebben megjegyezhetjük a táblázat tartalmát, ha a műveleti jeleket csoportokra osztjuk, és megértjük a műveletek egymáshoz viszonyított sorrendje mögött meghúzódó logikát.

A műveletek végrehajtási sorrendjének táblázata (a táblázatban feljebb lévő műveletek megelőzik a lejjebb lévőket)

művelet	társíthatóság
`() [] -> .`	balról
`! ~ ++ -- - (típus) * & sizeof`	jobbról
`* / %`	balról
`+ -`	balról
`<< >>`	balról
`< <= => >`	balról
`== !=`	balról
`&`	balról
`^`	balról
`|`	balról
`&&`	balról
`||`	balról
`?:`	jobbról
értékadó műveletek	jobbról
`,`	balról

A táblázat és a műveleti sorrend legelején olyan dolgokat találunk, amik nem is igazán műveletek: az indexelés, a függvényhívás és a struktúrák kijelölése. Ezek mindegyike balról társítható, vagyis az `a.b.c` ugyanazt jelenti, mint az `(a.b).c`, és nem azt, hogy `a.(b.c)`.

Ezután következnek az egytényezős műveletek (operátorok). A valódi műveletek közül ezeket hajtja végre először a program. Mivel a függvényhívások előrébb vannak az egytényezős műveleteknél, így azt a függvényt, amire a `p` mutat, a `(*p)()` kifejezéssel hívhatjuk meg. A `*p()` ugyanazt jelenti, mint a `*(p())`. A típusátalakítás egytényezős művelet, ezért a végrehajtási sorrendben a többi egytényezős művelettel egy szinten találjuk. Az egytényezős műveletek jobbról társíthatók, tehát a `*p++` helyes értelmezése a `*(p++)` (vegyük azt az objektumot, amire a `p` mutat, majd növeljük meg eggyel `p` értékét) kifejezés lesz, és nem a `(*p)++` (növeljük meg annak az objektumnak az értékét eggyel, amire a `p` mutat) kifejezés. A 3.7. részben látni fogjuk, hogy a `p++` kifejezés alkalmazása néha meglepő eredményhez vezethet.

Ezután jönnek az igazi kéttényezős műveletek. A számtani műveleteké az elsőbbség a végrehajtás során, amiket az eltolási műveletek, az összehasonlító műveletek, a logikai műveletek, az értékadó műveletek és végül a feltételes művelet követ. A két legfontosabb dolog, amiről nem szabad megfeledkeznünk:

1. Minden összehasonlító művelet megelőzi az összes logikai műveletet a végrehajtási sorrendben.
2. Az eltolási műveletek előrébb vannak, mint az összehasonlító műveletek, de hátrébb, mint a számtani műveletek.

A különféle műveletcsoportok belsejében kevés meglepetés ér minket. A szorzás, osztás és a maradékos osztás egy szinten vannak, mint ahogy az összeadás és a kivonás, valamint a két eltolási művelet is egy szinten van. Vannak, akik meglepődnek azon, hogy az `1/2*a` jelentése $\frac{1}{2} \times a$ és nem $\frac{1}{2 \times a}$, de a C ebben a tekintetben pontosan úgy viselkedik, mint a Fortran, a Pascal, és a legtöbb más programnyelv.

Némi meglepetést az okozhat, hogy a hat összehasonlító műveletet nem egy szinten találjuk a műveletek végrehajtási sorrendjében. A `==` és a `!=` előrébb van a többi összehasonlító műveletnél. Ez lehetővé teszi, hogy a következő kifejezéssel ellenőrizzük, hogy a és `b` egymáshoz viszonyított sorrendje megegyezik-e `c` és `d` egymáshoz viszonyított sorrendjével:

```
a < b == c < d
```

A logikai műveletek között nincs két olyan, ami ugyanazon az elsőbbségi szinten helyezkedne el. A bitműveletek megelőzik a sorrendi műveleteket, mindegyik *és* művelet megelőzi a megfelelő *vagy* műveletet, a bitenkénti *kizáró vagy* művelet (`^`) pedig a bitenkénti *és* és a bitenkénti *vagy* közé esik.

Ezeknek a műveleteknek a végrehajtási sorrendje történeti okokra vezethető vissza. A C nyelv ősében, a B nyelvben olyan logikai műveletek voltak, amelyek durván a C `&` és `|` műveleteinek feleltek meg. Meghatározásuk szerint bitműveletek voltak ugyan, de ha egy feltételben szerepeltek, akkor a fordító úgy kezelte őket, mint a mai

`&&` és `||` műveleteket. Amikor a két felhasználási mód kettévált a C nyelvben, túl veszélyesnek ítélték a műveletek végrehajtási sorrendjének jelentős módosítását.

A háromtényezős feltételes művelet a végrehajtási sorrendben az eddig említett összes művelet után áll. Ez lehetővé teszi, hogy az eldöntendő kifejezés összehasonlító műveletek logikus kombinációját tartalmazza, ahogy itt is:

```
tax_rate = income > 40000 && residency < 5? 3.5: 2.0;
```

Ez a példa azt is megmutatja, hogy annak is megvan az értelme, hogy a végrehajtási sorrendben az értékadást megelőzi a feltételes művelet. Ezen kívül az összes értékadó művelet egy szinten helyezkedik el, és jobbról balra társíthatók, tehát a

```
home_score = visitor_score = 0;
```

kifejezés jelentése annyi, mint:

```
visitor_score = 0
home_score = visitor_score;
```

A sorrend legalján a vessző (felsorolás) műveletet találjuk. Ezt könnyű megjegyezni, mert a vesszőt gyakran használjuk a pontosvessző helyett, amikor utasítás helyett kifejezésre van szükség. A vessző műveleti jel különösen hasznos a makrók meghatározásánál (ennek részletes tárgyalását lásd a 6.3. részben).

Az értékadás gyakran áll a műveleti sorrendből fakadó zűrzavar hátterében. Nézzük meg például az alábbi ciklust, amivel egy fájlt akartunk egy másikba másolni:

```
while (c=getc(in) != EOF)
            putc(c,out);
```

Úgy tűnik, hogy a `while` utasításban található kifejezésben először a `c` változóba kellene menteni a `getc(in)` értékét, majd összehasonlítani az `EOF` értékkel, és egyezés esetén befejezni a ciklust. Legnagyobb sajnálatunkra az összehasonlító műveletek megelőzik az értékadást a műveleti sorrendben, így a `c` változó értéke a majd eldobandó `getc(in)` érték és az `EOF` érték összehasonlításának eredménye lesz. A fájl „másolata" így olyan bájtok sorozata lesz, amelyeknek (kettes számrendszerbeli) értéke `1`. A fenti példa helyesen:

```
while ((c=getc(in)) != EOF)
            putc(c,out);
```

Az ilyesfajta hibák kiszúrása elég nehéz feladat lehet a bonyolultabb kifejezésekben. A 4.0. részben említett `lint` program egyik kiadása például ezt a hibás sort tartalmazta:

```
if( (t=BTYPE(pt1->aty)==STRTY) || t==UNIONTY ){
```

A cél az volt, hogy értéket adjanak a `t` változónak, majd megvizsgálják, hogy `t` egyenlő-e az `STRTY` vagy az `UNIONTY` értékével. A tényleges eredmény azonban teljesen más. A `t` értéke `1` vagy `0` lesz, attól függően, hogy a `BTYPE(pt1->aty)` egyenlő-e az `STRTY` értékével. Ha `t` nulla, akkor a `t` értékét összehasonlítjuk az `UNIONTY` értékével.

2.3. Figyeljünk oda a pontosvesszőkre!

Egy felesleges pontosvessző egy C programban teljesen ártalmatlan is lehet. Lehet üres utasítás, aminek semmilyen hatása nincs, vagy kiválthat egy hibaüzenetet a fordítóprogramtól, ami megkönnyíti az eltávolítását. Kivétel ez alól az `if` és a `while` utasítás, amelyeket mindig pontosan egy utasításnak kell követnie. Nézzük meg a következő példát:

```
if (x[i] > big);
        big = x[i];
```

A fordítóprogram gond nélkül megemészti az első sorban lévő pontosvesszőt, de éppen ezért ezt a programrészletet egészen másképp fogja kezelni, mint ezt:

```
if (x[i] > big)
        big = x[i];
```

Az első példa ennek felel meg:

```
if (x[i] > big) { }
big = x[i];
```

ami persze ugyanaz, mint a[1]

```
big = x[i];
```

Egy lefelejtett pontosvessző is komoly gondokat okozhat szép csendben:

```
if (n < 3)
        return
logrec.date = x[0];
logrec.time = x[1];
logrec.code = x[2];
```

Itt a `return` utasítás mögül ugyan hiányzik egy pontosvessző, de nagyon valószínű, hogy a kódrészlet fordítása hibajelzés nélkül végbemegy, és a teljes alábbi utasítást

```
logrec.date = x[0];
```

úgy kezeli, mintha az a `return` utasításhoz tartozna. Ez ugyanaz, mintha azt írnánk, hogy

```
if (n < 3)
        return logrec.date = x[0];
logrec.time = x[1];
logrec.code = x[2];
```

[1] Hacsak az `x`, az `i`, vagy a `big` nem valamilyen mellékhatással rendelkező makró.

Ha ez a kódrészlet egy üres értéket (`void`) visszaadó függvény része lenne, akkor elvárhatnánk, hogy a fordító hibaüzenettel reagáljon rá. Azoknál a függvényeknél azonban, amelyek nem adnak vissza semmilyen értéket, sokszor a visszatérési érték típusát sem adják meg, ami hallgatólagosan egy egész érték visszaadását jelenti. Ez a hiba tehát könnyen rejtve maradhat. A hatása viszont annál alattomosabb lehet. Ha $n \geq 3$, akkor a három értékadó utasítás közül az első végrehajtása egyszerűen elmarad.

Egy másik hely, ahol a pontosvessző megfelelő használata rendkívül fontos, ha az egy olyan deklaráció után áll, ami közvetlenül egy függvény meghatározása előtt található. Nézzük meg az alábbi kódrészletet:

```
struct logrec {
        int date;
        int time;
        int code;
}

main( )
{
        . . .
}
```

Egy pontosvessző hiányzik az első } és a közvetlen utána következő `main` függvény meghatározása közül. Ennek az lesz a hatása, hogy a `main` függvény egy `struct logrec` típusú értéket ad vissza, amit a függvény bevezetésének részeként adtunk meg. Vagyis a fenti kódot vehetjük így is:

```
struct logrec {
        int date;
        int time;
        int code;
} main( )
{
        . . .
}
```

Ha a helyén lenne a pontosvessző, akkor a `main` függvény meghatározása alapértelmezés szerint az lenne, hogy egy egész értéket adjon vissza.

Annak elképzelését, hogy mit eredményezhet, ha egész érték helyett `struct logrec` típusú értéket ad vissza a `main` függvény, gyengébb idegzetűeknek nem ajánlom.

2.4. A switch utasítás

A C nyelv egyik formabontó tulajdonsága, hogy a `switch` utasítás egyes esetei egymásba folyhatnak. Hasonlítsuk össze például a következő C és Pascal nyelven írt kódrészleteket:

```
switch (color) {
case 1: printf("red");
        break;
case 2: printf("yellow");
        break;
case 3: printf("blue");
        break;
```

```
case color of
1: write( 'red');
2: write( 'yellow');
3: write('blue')
end
```

Mindkét programrészlet ugyanazt teszi. Attól függően, hogy a `color` változó értéke `1`, `2` vagy `3`, (új sor kezdése nélkül) kiírják a képernyőre a `red`, `yellow` vagy `blue` szavak egyikét. A programrészletek minden tekintetben hasonlóak, egy kivétellel. A Pascal programban nem találunk olyan részt, ami megfelelne a C `break` utasításának. Ennek az az oka, hogy a C nyelven az esetek címkéi abban a tekintetben valódi címkék, hogy a program folyását nem akadályozzák, az zavartalanul folytatódhat. A Pascal nyelvben ezzel szemben minden eset címkéje egyben az előző eset végét is jelzi.

Egy kicsit másképp nézve a dolgot, a példa kedvéért most hasonlítson még jobban a C kódrészlet a Pascal kódhoz és tegyük fel, hogy a `color` változó értéke `2`.

```
switch (color) {
case 1: printf{"red");
case 2: printf{"yellow");
case 3: printf{"blue");
}
```

Ekkor a program ezt írja ki:

```
yellowblue
```

A program vezérlése ugyanis a második `printf` hívása után természetes módon az azt követő utasításra ugrik.

Ez a C nyelv `switch` utasításának egyik erőssége és egyben gyengesége is. Gyengeség, hiszen könnyen megfeledkezhetünk a `break` utasításról, ami a program rendellenes viselkedéséhez vezethet. Erősség, mert a `break` utasítás szándékos elhagyásával olyan vezérlési szerkezetek valósíthatók meg, amelyek kivitelezése másképpen igen bonyolult feladat lenne. Konkrétan, a nagyobb `switch` utasításoknál találkozunk gyakran azzal, hogy az egyik eset feldolgozása egy kis átalakítás után egy másik esetté egyszerűsödik.

Képzeljünk el például egy olyan programot, ami egy képzeletbeli gép parancsértelmező egysége. Egy ilyen program tartalmazhat egy `switch` utasítást, ami a különféle műveletek kódjait kezeli. Gyakran megesik, hogy egy kivonási művelet megegyezik egy összeadással, ha a második tag előjelét megfordítjuk. Ilyenkor jó, ha lehetőségünk van egy ehhez hasonló kód megírására:

```
case SUBTRACT:
        opnd2 = -opnd2;
        /*nincs break*/
case ADD:
        . . .
```

A fenti példában látható megjegyzés természetesen nagyon hasznos. A program olvasója így egyből látja, hogy a `break` utasítás elhagyása szándékosan történt.

Szintén jó példa egy fordítóprogramnak az a része, amelyik átugorja az üres helyközöket a lexikális elemek keresése közben. Ebben az esetben a szóköz, tabulátor és újsor karaktereket hasonlóan akarjuk kezelni, azzal a különbséggel, hogy az újsor karakternél eggyel növelnünk kell egy sorszámláló változó értékét:

```
case '\n':
       linecount++;
       /*nincs break*/
case '\t':
case ' ':
          . . .
```

2.5. Függvényhívások

Néhány más programnyelvtől eltérően a C nyelvben akkor is ki kell írnunk a paraméterek listáját, ha nincsenek paraméterek. Tehát ha `f` egy függvény, akkor az

```
f();
```

utasítás meghívja a függvényt, a

```
f;
```

viszont nem csinál semmit. Pontosabban, kiértékeli a függvény címét, de nem hívja meg.

2.6. A levegőben lógó else problémája

Habár ez a közismert probléma nem a C nyelv sajátsága, számos sokéves tapasztalattal rendelkező C programozót megtréfált már.

Vizsgáljuk meg a következő programrészletet:

```
if (x == 0)
        if (y == 0) error();
else {
        z = x + y;
        f(&z);
}
```

A programozó szándéka ezzel a kódrészlettel két fő eset meghatározása volt: x=0 és x≠0. Az első esetben a programnak semmit sem kell tennie, kivéve, ha y=0, amikor is meghívja az error függvényt. A második esetben a z változónak az x+y értékét adja, majd meghívja az f függvényt, aminek paraméterként átadja a z változó címét.

A programrészlet azonban valami egészen mást tesz. Ennek az a szabály az oka, miszerint az else egy elágazáson belül mindig a legközelebbi pár nélküli if utasításhoz tartozik. Ha átalakítjuk a behúzásokat a programban aszerint, ahogy a végrehajtása történik, akkor ezt kapjuk:

```
if (x == 0) {
        if (y == 0)
                error();
        else {
                z = x + y;
                f(&z);
        }
}
```

Vagyis ha x≠0, akkor semmi sem történik. Az eredeti példa behúzásai által sugallt hatás eléréséhez ezt kell írnunk:

```
if (x == 0) {
        if (y == 0)
                error();
```

```
} else {
                z = x + y;
                f(&z);
}
```

Az `else` itt már az első `if` utasításhoz tartozik, annak ellenére, hogy a második `if` közelebbi, mivel a második most már kapcsos zárójelek között van.

Néhány programozási nyelv határozott határolójeleket használ az `if` utasításokhoz. A fenti példa az Algol 68 nyelvben például így nézne ki:

```
if x = 0
then   if y = 0
       then error
       fi
else   z := x + y;
       f(z)
fi
```

A határolójelek kötelezővé tétele tökéletesen megoldja a levegőben lógó `else` problémáját, azon az áron, hogy a program kissé hosszabb lesz. Néhány C-felhasználó makrók segítségével próbál hasonló hatást elérni:

```
#define IF          {if(
#define THEN        ) {
#define ELSE        } else {
#define FI          }}
```

Ezekkel a fenti utolsó C példát így írhatnánk le:

```
IF x == 0
THEN   IF y == 0
       THEN   error();
       FI
ELSE   z = x + y;
       f(&z);
FI
```

Azoknak a C-felhasználóknak, akiknek az Algol 68 nem kimondottan a második anyanyelvük, nehézséget okozhat ennek a kódnak az értelmezése. Ez a megoldás tehát néha kellemetlenebb lehet, mint maga az eredeti probléma.

2.1. gyakorlat. A C nyelvben a kezdőértékek felsorolásakor megadhatunk még egy vesszőt:

```
int days[] = { 31, 28, 31, 30, 31, 30,
               31, 31, 30, 31, 30, 31, };
```

Miért hasznos ez?

2.2. gyakorlat. Számos problémát láttunk, ami abból adódott, hogy a C utasításokat pontosvesszővel kell lezárni. Ezen persze már nem változtathatunk, de szórakozásképpen megpróbálhatunk kitalálni más módokat az utasítások elkülönítésére. Hogy csinálják ezt más programnyelvekben? Vajon ezeknek a megoldásoknak is megvannak a maguk buktatói?

3

Szemantikai buktatók

Ha egy mondatban a helyesírás tökéletes, a nyelvtani felépítése pedig kifogástalan, attól az még mindig lehet kétértelmű, vagy hordozhat valamilyen szándékolatlan jelentést is. Ebben a fejezetben olyan programokat veszünk szemügyre, amelyek látszólag egyértelműek, de igazi jelentésük teljesen más, mint amit várnánk.

Ezen kívül szót ejtünk néhány olyan helyzetről is, amikor a felszínen értelmesnek tűnő dolgok valójában előre meghatározhatatlan eredményt hozhatnak a C nyelv összes megvalósításában. Azokról a dolgokról, amelyek egy-egy megvalósításban működnek, másokban viszont nem, a 7. fejezetben lesz szó, ahol a hordozhatósággal foglalkozunk.

3.1. Mutatók és tömbök

A C nyelvben a mutatók és a tömbök fogalma olyannyira elválaszthatatlan egymástól, hogy csak úgy érthetjük meg az egyiket teljesen, ha a másikkal is tökéletesen tisztában vagyunk. A C egyébiránt néhány szempontból teljesen eltérő módon kezeli ezeket a fogalmakat mint a többi ismertebb programnyelv.

A C tömbökkel kapcsolatban két dolgot kell kiemelnünk:

1. A C nyelvben csak egydimenziós tömbök léteznek, a tömb méretének pedig állandónak kell lennie. Ezt a méretet még a fordítás előtt meg kell adnunk. A tömb azonban bármilyen típusú elemekből állhat, akár más tömbökből is, így a többdimenziós tömbök létrehozása igen egyszerű.
2. Mindössze két dolgot tehetünk egy tömbbel. Megállapíthatjuk a méretét, és lekérdezhetjük a 0. elemet címző mutatót. Az *összes* többi tömbműveletet tulajdonképpen mutatókkal végezzük el, még akkor is, ha a program írásakor látszólag tömbindexeket használunk. Vagyis minden indexművelet egy mutatóműveletnek felel meg, tehát az indexek viselkedése teljes egészében leírható a mutatók viselkedésével.

Ha ezt a két pontot és a következményeiket egyszer megértettük, akkor sokkal egyszerűbb lesz a C tömbműveleteit használni. Addig viszont nagy zűrzavart okozhatnak. Különösen fontos, hogy megértsük a tömbműveletek és a nekik megfelelő mutatóműveletek felcserélhetőségét. A tömbök indexelését a legtöbb más programnyelvben megtalálhatjuk. A C nyelvben ennek alapját a mutatókkal végzett műveletek adják.

Ahhoz, hogy megérthessük a tömbök működését, előbb meg kell értenünk, hogyan vezethetjük be őket. Például az

```
int a[3];
```

azt jelenti, hogy az `a` egy három egész elemből álló tömb. Hasonlóképpen, a

```
struct {
        int p[4];
        double x;
} b[17];
```

azt jelenti, hogy a `b` egy 17 elemből álló tömb, amelynek minden eleme egy olyan szerkezet, ami egy (`p` nevű) négy egész értéket tartalmazó tömbből és egy (`x` nevű) `double` típusú változóból áll.

Nézzük most a következőt:

```
int calendar[12][31];
```

Eszerint a `calendar` olyan `12` elemű tömb, amelynek az elemei `31` egész értéket tartalmazó tömbök (tehát nem `31` elemű tömb, aminek az elemei `12` egész értéket tartalmazó tömbök). A `sizeof(calendar)` kifejezés értéke tehát `372`-szer (`31x12`) `sizeof(int)` lesz.

Ha a `calendar` változó nevét nem a `sizeof` függvény paramétereként adjuk meg, hanem szinte bárhol máshol, akkor az átalakul a `calendar` tömb első elemére irányuló mutatóvá. Ahhoz, hogy ez világos legyen, előbb meg kell értenünk néhány dolgot a mutatókkal kapcsolatban.

Minden mutató *valamilyen típusra mutat.* Ha például azt írjuk, hogy

```
int *ip;
```

akkor azzal azt mondjuk, hogy az `ip` egy egész értékre mutat.
Ha most azt írjuk, hogy

```
int i;
```

akkor az alábbi módon rendelhetjük hozzá az `i` címét az `ip` mutatóhoz:

```
ip = &i;
```

Ezután úgy is megváltoztathatjuk `i` értékét, ha az `*ip` kifejezésnek adunk értéket:

```
*ip = 17;
```

Ha egy mutató egy tömb egyik elemére mutat és hozzáadunk `1`-et a mutatóhoz, akkor megkapjuk a tömb következő elemére irányuló mutatót. Hasonlóképpen, ha kivonunk `1`-et a mutatóból, akkor megkapjuk a tömb előző elemére irányuló mutatót, és ez így megy tovább a többi egész számmal is.

Ebből az következik, hogy egy egész szám hozzáadása egy mutatóhoz általában nem ugyanaz, mintha a mutató bitekből álló ábrázolásához adtuk volna hozzá az egész számot! Ha `ip` egy egész számra mutat, akkor `ip+1` a gép memóriájában soron következő egészre mutat, ami a legtöbb modern számítógép esetében nem a következő memóriahely lesz. A mutatókat ki is vonhatjuk egymásból, feltéve, hogy mindkettő ugyanannak a tömbnek az elemeire mutat. Ez így logikus is. Ha azt írjuk, hogy

```
int *q = p + i;
```

akkor úgy helyes, ha a `q-p` művelet eredménye `i` lesz. Figyeljük meg, hogy ha `p` és `q` nem ugyanannak a tömbnek az elemeire mutatnak, akkor még arra sincs semmilyen garancia, hogy a `p` és a `q` közötti távolság egy tömbelem címének egész számú többszöröse lesz!

Az `a` változót már bevezettük egy három egész elemet tartalmazó tömbként. Ha a tömbök nevét olyan helyen használjuk, ahol a program mutatót vár, akkor a tömb neve a tömb `0` sorszámú elemét címző mutató lesz. Ha tehát azt írjuk, hogy

```
p = a;
```

azzal a `p` értékét az `a` tömb `0` sorszámú elemének címére állítjuk. Vegyük észre, hogy nem azt írtuk, hogy

```
p = &a;
```

Ez az ANSI C szabványban nem megengedett, mert az `&a` kifejezés egy tömbre mutat, a `p` viszont egy egész számra. A C korábbi változataiban nem létezett olyan fogalom, hogy egy tömb címe. Az `&a` kifejezés vagy szabálytalan, vagy egyenlő az `a` változóval.

Most, hogy a `p` az `a` `0` sorszámú elemére mutat, a `p+1` az `1` sorszámú elemre, a `p+2` a `2` sorszámú elemre, és így tovább. Ezért ha azt akarjuk, hogy a `p` az `1` sorszámú elemre mutasson, akkor ezt írjuk:

```
p = p + 1;
```

ami persze megegyezik azzal, hogy

```
p++;
```

Egy kivétellel az `a` változónév a `0` sorszámú elem címére vonatkozik. A kivétel az, amikor a `sizeof` függvény paramétereként használjuk. Ott a `sizeof(a)` azt teszi, amit jogosan elvárunk tőle, vagyis megadja a teljes a tömb méretét, és nem csak az egyik elemét címző mutató méretét.

Ebből az egészből az következik, hogy a `*a` kifejezéssel hivatkozhatunk az `a` tömb `0` sorszámú elemére. Ha azt írjuk, hogy

```
*a = 84;
```

akkor az `a` tömb `0` indexű elemének értéke `84` lesz. Tovább folytatva a gondolatmenetet, a `*(a+1)` az `a` tömb `1` indexű elemére vonatkozik, és így tovább. Általánosítva, az `a` tömb `i` indexű elemére a `*(a+1)` kifejezéssel hivatkozhatunk. Ezt olyan gyakran kell használnunk, hogy az `a[i]` kifejezést használjuk a rövidítésére.

Pontosan ez az, amit a kezdő C programozók oly nehezen értenek meg. Arról nem is beszélve, hogy mivel az `a+i` egyenértékű az `i+a` kifejezéssel, ezért az `a[i]` és az `i[a]` *szintén* ugyanazt jelentik. Az utóbbi használatot határozottan nem javasoljuk, habár néhány assembly nyelvben jártas programozónak ismerős lehet ez a forma.

Most már rátérhetünk a „kétdimenziós tömbökre", amelyek, amint azt láttuk, igazából tömbökből álló tömbök. Ha csak egydimenziós tömbökkel van dolgunk, akkor nem különösebben nehéz egy olyan program elkészítése, amiben a tömböket kizárólag mutatókkal kezeljük. A kétdimenziós tömböknél azonban az indexek használata a tömbök elemeinek jelölésére kényelmi szempontból létfontosságúvá válik. Ha csak mutatókat használunk a kétdimenziós tömbök kezelésére, azzal betekintést nyerhetünk a nyelv sötét bugyraiba, ahol a fordítóhibák baljós árnyai leselkednek ránk.

Vegyük ismét a következő változókat:

```
int calendar[12][31];
int *p;
int i;
```

Tegyük fel magunknak a kérdést, mit jelent tulajdonképpen a `calendar[4]` kifejezés?

Mivel a `calendar` egy olyan, `12` elemből álló tömb, amelynek minden eleme egy `31` egész értékből álló tömb, a `calendar[4]` kifejezés egyszerűen ennek a tömbnek a `4` indexű eleme. A `calendar[4]` tehát a `12` darab `31` egész értéket tartalmazó tömb egyike, és *pontosan* úgy is viselkedik. Tehát a `sizeof(calendar[4])` értéke például `31`-szerese lesz egy egész érték méretének, és ha azt írjuk, hogy

```
p = calendar[4];
```

azzal azt érjük el, hogy a `p` a `calendar[4]` tömb `0` indexű elemére fog mutatni.

Ha viszont a `calendar[4]` egy tömb, akkor indexelhetjük is, tehát írhatjuk, hogy

```
i = calendar[4][7];
```

És bizony ezt meg is tehetjük. Ez az utasítás pontosan azonos azzal, hogy

```
i = *(calendar[4] + 7);
```

ami viszont nem más, mint

```
i = *(*(calendar+4) + 7);
```

A szögletes zárójelek használata ennél nyilvánvalóan kényelmesebb.

Nézzük most azt, hogy

```
p = calendar;
```

Ez a kifejezés nem megengedett, mert a `calendar` tömbökből álló tömb. A `calendar` nevet használva ebben a környezetben ezért egy tömbmutatót kapunk. Mivel `p` egy egész értékre mutat, két eltérő típusú mutatót próbálunk egyenlővé tenni.

Úgy tűnik, szükségünk lesz egy olyan módszerre, amivel bevezethetünk egy tömbmutatót. Ha már átverekedtük magunkat a 2. fejezeten, akkor ez nem okoz komoly nehézséget:

```
int (*ap)[31];
```

Lényegében ezzel azt mondjuk, hogy a `*ap` egy `31` egész elemet tartalmazó tömb, tehát az `ap` egy ilyen tömbre mutat. Ezért azt is írhatjuk, hogy

```
int calendar[12][31];
int (*monthp)[31];
monthp = calendar;
```

és ekkor a `monthp` a `calendar` tömb `12` darab `31` elemű tömbje közül az elsőre fog mutatni.

Tegyük fel, hogy új év kezdődik, és ki szeretnénk üríteni a naptárat. Ezt az indexek segítségével könnyedén megtehetjük:

```
int monthp;
for (month = 0; month < 12; month++) {
            int day;
            for (day = 0; day < 31; day++)
                        calendar[month][day] = 0;
}
```

Mi történik itt a mutatók szemszögéből? Az alábbi sort

```
calendar[month][day] = 0;
```

könnyen kezelhetjük így is:

```
*(*(calendar + month) + day) = 0;
```

De mi történik tulajdonképpen? Ha `monthp` egy `31` egész értéket tartalmazó tömbre mutat, akkor a `monthp` végigmehet a `calendar` összes értékén, mint bármely más mutató esetében:

```
int (*monthp)[31];
for (monthp = calendar; monthp < &calendar[12];
        ➥ monthp++)
    /* Az adott hónap kezelése */
```

Hasonlóképpen, az egyik `monthp` által kijelölt tömb elemeit is kezelhetjük úgy, mint bármely más tömböt:

```
int (*monthp)[31];
for (monthp = calendar; monthp < &calendar[12];
        ➥ monthp++){
    int *dayp;
    for(dayp = *monthp; dayp<&(*monthp)[31];
        ➥ dayp++)
            *dayp = 0;
}
```

Most már elég szépen begyalogoltunk az ingoványos terület közepére ahhoz, hogy gyorsan megforduljunk, mielőtt elnyelne minket. Bár az utolsó példa szabályos ANSI C kód, csak üggyel-bajjal sikerült olyan fordítót találnom, amelyik elfogadta. Kis kirándulásunknak az volt a célja, hogy szemléltesse a C nyelvben található tömbök és mutatók között fennálló különleges kapcsolatot, miközben mindkettővel közelebbről is megismerkedhettünk.

3.2. A mutatók nem tömbök

A C nyelvben a karakterlánc-állandók egy adott című memóriaterületre vonatkoznak, ahol az állandó karaktereit és azt követően egy null

karaktert (`'\0'`) találunk. Mivel a nyelv előírja a null lezárót a karakterlánc-állandókhoz, a programozók a többi karakterláncnál is követik ezt a szokást.

Tegyük fel, hogy `s` és `t` két ilyen karakterlánc, amelyeket össze szeretnénk fűzni az `r` karakterlánccá. Ehhez rendelkezésünkre állnak a szokásos `strcpy` és `strcat` könyvtári függvények. Az alábbi nyilvánvalónak tűnő módszer azonban nem működik:

```
char* r;
strcpy(r, s);
strcat(r, t);
```

Azért nem működik, mert az `r` nem mutat sehová. Ahhoz pedig, hogy az `r` mutathasson valahová, lennie kell valaminek, amire mutathat. Ezt a memóriahelyet valahogy le kell foglalnunk.

Próbáljuk meg újra úgy, hogy lefoglalunk némi memóriát az `r` számára:

```
char r[100];
strcpy(r, s);
strcat(r, t);
```

Ez egészen addig működik is, amíg nem túl hosszúak azok a karakterláncok, amikre az `s` és a `t` mutatnak. Sajnos a C nyelvben kötelezően állandóként kell megadnunk a tömbök méretét, így nem lehetünk biztosak benne, hogy az `r` elég nagy lesz-e. A legtöbb C megvalósítás azonban tartalmaz egy `malloc` nevű függvényt. Ez lefoglal egy akkora memóriaterületet, ahová annyi karakter fér, amekkora számot paraméterként átadtunk neki. Általában van egy `strlen` nevű függvény is, ami megmondja, hogy hány karakterből áll egy karakterlánc. Úgy tűnhet tehát, hogy írhatjuk a következőt:

```
char *r, *malloc();
r = malloc(strlen(s) + strlen(t));
strcpy(r, s);
strcat(r, t);
```

Ez a példa azonban három okból is kudarcra van ítélve. Először is nem biztos, hogy a `malloc` le tudja foglalni a szükséges mennyiségű memóriát, amit azzal jelez, hogy egy cím nélküli (null) mutatót ad vissza.

Másodszor, nem szabad megfeledkeznünk az `r` számára lefoglalt memória felszabadításáról, ha már nem használjuk azt. Mivel az előző programrészletben az `r` helyi változó volt, a memória felszabadítása automatikusan történt. Az átalakított programban azonban külön foglaljuk le a memóriát, ezért külön fel is kell szabadítanunk azt.

A harmadik és egyben legfontosabb ok, hogy a `malloc` nem igazán foglal le elegendő memóriát. Emlékezzünk vissza arra a szokásra, hogy a karakterláncokat egy null karakterrel zárják le. Az `strlen` függvény ugyan visszaadja a karakterlánc hosszát, de ebben *nincs* benne a karakterláncot lezáró null karakter. Ha tehát az `strlen(s)` értéke `n`, akkor az `s` tárolásához ténylegesen `n+1` karakternyi helyre van szükség. Az `r` számára tehát eggyel több karakternyi helyet kell foglalnunk. Ha ezzel megvagyunk, és azt is ellenőrizzük, hogy sikerrel jár-e a `malloc`, akkor ezt kapjuk:

```
char *r, *malloc();
r = malloc(strlen(s) + strlen(t) + 1);
if (!r) {
        complain();
        exit(1);
}
strcpy(r, s);
strcat(r, t);

/* Valamivel később */
free(r);
```

3.3. Tömbök használata paraméterként

A tömböket közvetlenül nem adhatjuk át egy függvénynek paraméterként. A tömb nevét használva rögtön egy mutatót kapunk, ami a tömb kezdő elemére mutat. Ha például azt írjuk, hogy

```
char hello[] = "hello";
```

akkor a `hello` egy karakterekből álló tömb lesz. Ha átadjuk egy függvénynek ezt a tömböt:

```
printf("%s\n", hello);
```

az éppen olyan, mintha a függvény az első karakter címét kapná meg:

```
printf("%s\n", &hello[0]);
```

Ezért soha nincs értelme annak, ha egy tömböt adunk át paraméterként egy függvénynek. Ebből kifolyólag a C a tömb paramétereket automatikusan átalakítja mutató paraméterré. Más szóval, ha azt írjuk, hogy

```
int
strlen(char s[])
{
        /* A lényeg... */
}
```

az pontosan olyan, mintha azt írnánk, hogy

```
strlen(char *s)
{
        /* A lényeg... */
}
```

A C programozók gyakran hibásan feltételezik, hogy ez az automatikus átalakítás más környezetekben is működik. A 4.5. részben részletesen kitérünk egy különösen gyakori hibára. Az, hogy:

```
extern char *hello;
```

soha *nem* ugyanaz, mint az

```
extern char hello[];
```

Ha egy mutató paraméter nem tömböt ábrázol, akkor a szögletes zárójelek használata félrevezető, bár technikailag helyes. Mi a helyzet azok-

kal a mutató típusú paraméterekkel, amelyek tömböket címeznek? Az egyik példa erre a `main` függvény második paramétere:

```
main(int argc, char *argv[])
{
        /* További műveletek... */
}
```

Ez ugyanaz, mint a

```
main(int argc, char **argv)
{
        /* További műveletek... */
}
```

de az előző példában kihangsúlyoztuk, hogy az `argv` olyan mutató, ami egy karaktermutatókat tartalmazó tömb első elemére mutat. Mivel a két jelölés egyenértékű, használjuk azt, amelyik a legjobban kifejezi a szándékainkat.

3.4. Hagyjuk a szinekdochéket másra

A szinekdoché olyan költői eszköz, ami kicsit a hasonlathoz vagy a metafórához hasonlít. Az *Idegen szavak és kifejezések szótára* szerint a szinekdoché, vagy másképpen névcsere, egy „fogalomköri kapcsolatokon alapuló szókép, amely az egész és a rész, az anyag és a belőle készült tárgy (...) felcserélésén alapul".

Ez a leírás pontosan ráillik arra a gyakori buktatóra, amikor a C nyelvben összekeverik a mutatót és az adatot, amire mutat. Ez leggyakrabban a karakterláncoknál fordul elő. Például:

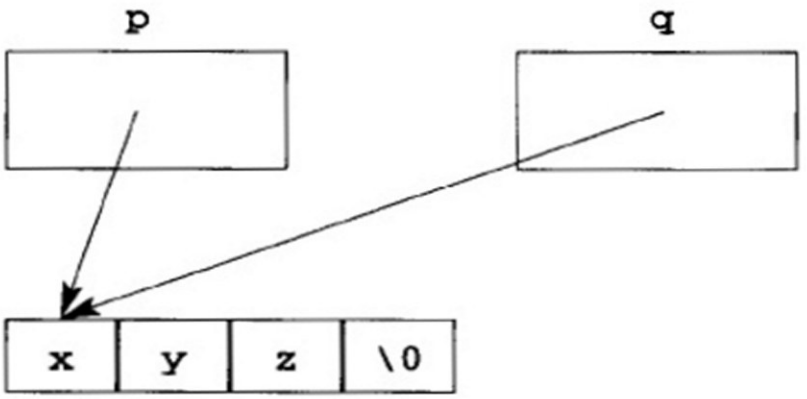
```
char *p, *q;
p = "xyz";
```

Fontos, hogy tisztán lássuk, hogy az értékadás után néha ugyan hasznos lehet úgy gondolni a `p` értékére, mint `xyz`, de ez így nem telje-

sen helyes. Ehelyett a p értéke egy *mutató*, ami az x, y, z és '\0' karaktereket tartalmazó, négyelemű tömb 0 indexű elemére mutat. Ha tehát most végrehajtjuk ezt az utasítást:

```
q = p;
```

akkor a p és a q két olyan mutató lesz, ami ugyanarra a memóriaterületre mutat. A memória ezen területén található karakterek azonban nem másolódtak le az értékadás következtében. A helyzet most a következő:



Jegyezzük meg, hogy a *mutató másolásával azt nem másoljuk le, amire mutat.*

Ha tehát ezután végrehajtjuk a

```
q[1] = 'Y';
```

utasítást, akkor a q egy olyan memóriaterületre fog mutatni, ami az xYz karakterláncot tartalmazza. A p mutatóval ugyanez lesz a helyzet, hiszen a p és a q ugyanarra a memóriahelyre mutat.

3.5. A cím nélküli mutatók nem üres karakterláncok

Az, hogy mi lesz az eredménye annak, ha egy egész értéket átalakítunk mutatóvá, az adott megvalósítástól függ, egyetlen fontos kivétellel. Ez a kivétel a 0 állandó, amiből garantáltan olyan mutató lesz, ami nem egyenlő egyetlen érvényes mutatóval sem.

A dokumentálás kedvéért gyakran jelképesen meg is adják ezt:

```
#define NULL 0
```

de a hatás ugyanaz. A legfontosabb dolog, amiről nem szabad megfeledkeznünk, ha a 0 értékből mutató lesz, hogy soha ne próbáljuk elérni az általa hivatkozott helyet. Más szóval, ha egy mutatónak a 0 értéket adtuk, akkor sosem szabad megkérdeznünk, hogy mi van azon a memóriahelyen. Helyes, ha azt írjuk, hogy:

```
if (p == (char*) 0) ...
```

de azt írni helytelen, hogy:

```
if (strcmp(p, (char *) 0) == 0) ...
```

mert az `strcmp` mindig megnézi a paraméterei által hivatkozott memóriacímet.

Ha `p` cím nélküli mutató (null mutató), akkor a

```
printf(p);
```

és a

```
printf("%s", p);
```

hatása is meghatározhatatlan. Az ilyen utasítások hatása a különböző gépeken más és más lehet. A 7.6. részben még szó lesz erről.

3.6. A számolás és az aszimmetrikus korlátok

Ha egy tömbnek 10 eleme van, akkor milyen indexeket használhatunk az elemek jelölésére?

A különféle nyelvek más-más választ adnak erre a kérdésre. A Fortran, a PL/I és a Snobol4 nyelvekben például a kezdő index az 1, de a prog-

ramozó választhat más kezdőértéket is. Az Algol és Pascal nyelvekben nincs alapbeállítás, a programozónak minden tömbnél meg kell adnia az alsó és a felső korlátot is. A szabványos Basic nyelvben egy 10 elemből álló tömb bevezetése valójában egy 11 elemű tömböt hoz létre, amiben az elemek sorszáma 0-tól 10-ig bezárólag tart!

A C nyelvben az elemek indexelése 0-tól 9-ig tart. Egy 10 elemből álló tömbben van 0. elem, de nincs 10. elem. Egy `n` elemből álló C tömbnek nincs olyan eleme, aminek az indexe `n`, mivel az elemek sorszámozása `0`-tól `n-1`-ig tart. A más nyelvekhez szokott programozóknak ezért különösen óvatosnak kell lenniük a tömbök használatakor.

Nézzük meg közelebbről a bevezetőben már említett példát:

```
int i, a[10];
for(i=1; i<=10; i++)
        a[i] = 0;
```

Ebben a példában a programozó szándéka az volt, hogy az `a` tömb elemeinek a `0` értéket adja, de váratlan szövődmények léptek fel. Mivel a `for` utasításban az `i<=10` összehasonlítás szerepel a helyes `i<10` változat helyett, a nem létező `10` indexű elem értékét nullára állítja a program, ami alaposan belelép az `a` után következő memóriaszó lelkivilágába. Ha ezt a programot egy olyan fordítóval futtatjuk, amelyik csökkenő sorrendben foglalja le a memóriacímeket a változók számára, akkor az `a` után álló memóriaszó éppen `i` lesz. Ha pedig az `i` értékét nullára állítjuk, azzal a ciklusból végtelen ciklust csinálunk.

Bár a frissen felkent C programozóknak néha gondjuk akad a tömbökkel, a tömbök megvalósítása a nyelv egyik legnagyobb erősségét jelenti. Ahhoz, hogy ezt igazán értékelni tudjuk, egy kis magyarázatra lesz szükségünk.

A leggyakoribb programhibák közül a legnehezebben a *„kerítésoszlop"* hibákat, vagy más néven *„eggyel kevesebb"* hibákat a legnehezebb megtalálni. A 0.2. gyakorlat nevén nevezi a gyermeket: azt a kérdést tettük fel, hogy hány darab, egymástól 10 méterre lévő oszlop szükséges egy 100 méter hosszú kerítés kifeszítéséhez. Kapásból azt

válaszolnánk, hogy „nyilvánvalóan" 10, amit úgy kapunk, hogy a 100-at elosztjuk 10-zel. Ez persze hibás válasz, a helyes megoldás a 11.

Ezt legegyszerűbben úgy láthatjuk be, ha észrevesszük, hogy egy 10 méter hosszú kerítés kifeszítéséhez két oszlopra van szükségünk, mindkét végén egyre. Más szemszögből úgy is nézhetjük a dolgot, hogy minden 10 méteres résznek van egy oszlop a *bal* oldalán. Ezzel egy kivétellel az összes oszlopot megkapjuk. A fennmaradó oszlop a kerítés jobb szélső részének jobb oldalára kerül.

A probléma fenti két megközelítéséből két általános elvet szűrhetünk le, amelyekkel elkerülhetjük a kerítésoszlopos hibákat:

1. Mindig egy egyszerű esetből induljunk ki.
2. Mindig figyelmesen számoljunk.

Ha ezt jól bevéstük az eszünkbe, akkor nézzük meg az egészekkel történő számítások korlátait. Hány olyan `x` egész szám található például, amire igaz, hogy `x≥16` és `x≤37`? Vagyis hány eleme van a `16, 17, ..., 37` sorozatnak? A válasz nyilván valahol a `37-16` érték, vagyis a `21` közelében van, de most `20`, `21` vagy `22`?

Ha a felső és alsó korlát megegyezne, a megoldás magától értetődően az lenne, hogy egy olyan `x` egész van, amire igaz, hogy `x≥16` és `x≤16`, és ez nem más, mint a `16`. Ha tehát az alsó és felső korlát egyenlő, akkor a sorozatnak egy tagja van.

Legyen az alsó korlát jele `l`, a felső korlát jele pedig `h`. Ha ekkor azt mondjuk, hogy az alsó és a felső korlát megegyezik, azzal azt mondjuk, hogy `l=h`, vagy hogy `h-l=0`. Ebből látható, hogy a sorozat elemeinek száma `h-l+1`, vagyis a mi példánkban `22`.

A legtöbb kerítésoszlopos hiba forrása a `h-l+1` kifejezésben található `+1`. Nehéz ellenállni a kísértésnek, ami azt mondatja velünk, hogy ha egy karakterlánc egy másik karakterlánc 16-ik karakterétől annak 37-ik karakteréig tart, akkor a részlánc hossza 21 karakter lesz. Ez felvet egy kérdést is: létezik valamiféle programozási módszer, amellyel csökkenthetjük az ilyen hibák előfordulásának esélyét?

Bizony, hogy létezik, és ehhez mindössze egyetlen szabályt kell betartanunk: *egy tartományt mindig annak első elemével és az utána következő első elem segítségével adjunk meg.* Más szóval, ahelyett, hogy olyan x értékről beszélünk, amire igaz, hogy $x \geq 16$ és $x \leq 37$, inkább fogalmazzunk úgy, hogy egy olyan x érték, amire igaz, hogy $x \geq 16$ és $x < 38$. Használjunk bezárólagos alsó korlátot és kizárólagos felső korlátot. Ez az aszimmetria matematikailag nem túl szép, de meglepően leegyszerűsíti a programírást:

1. A tartomány mérete a korlátok különbsége lesz. 38–16 az 22, éppen annyi, ahány elemet találunk az aszimmetrikus 16 és 38 korlátok között.
2. A korlátok egyenlők, amikor a tartomány üres. Ez egyenes következménye az 1. pontnak.
3. A felső korlát sosem kisebb az alsó korlátnál, még akkor sem, ha a tartomány üres.

Az aszimmetrikus korlátok olyan nyelvben használhatók igazán kényelmesen, mint a C, ahol a tömbök indexelése nullától kezdődik. Egy ilyen tömb kizárólagos felső korlátja pontosan megegyezik a tömb elemeinek számával! Amikor tehát egy 10 elemű C tömböt határozunk meg, akkor a tömb indexeinek a 0 lesz a bezárólagos alsó korlátja, és 10 a kizárólagos felső korlátja. Ezért ezt kell írnunk:

```
int a[10], i;
for(i=1; i < 10; i++)
        a[i] = 0;
```

ahelyett, hogy

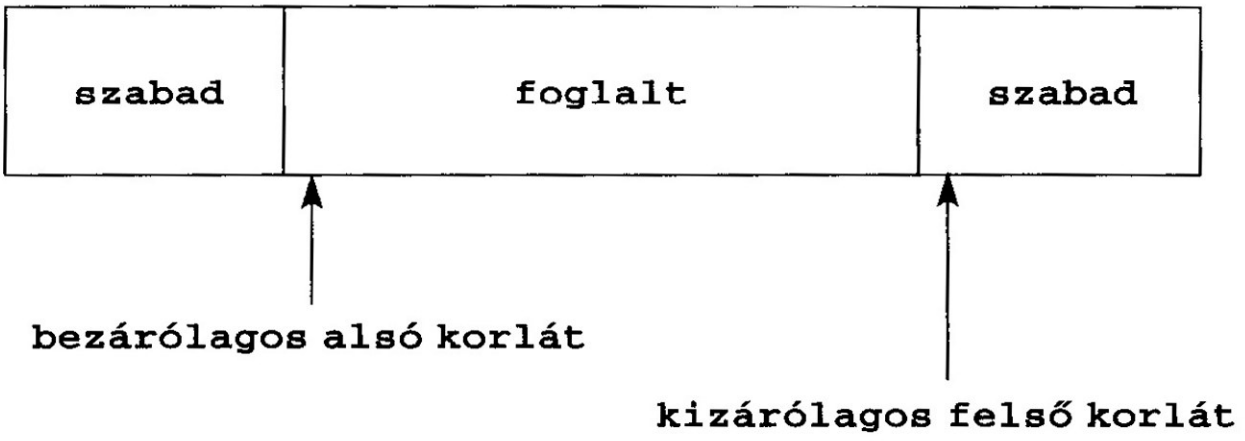
```
int a[10], i;
for(i=1; i <= 9; i++)
        a[i] = 0;
```

Ha a C nyelv `for` utasítása az Algol vagy a Pascal stílusát követné, akkor újabb buktatóhoz jutnánk. Mit jelentene ez?

```
for (i = 0 to 10)
        a[i] = 0;
```

Ha a 10 bezárólagos korlát lenne, akkor az `i` változó 11 különböző értéket venne fel, és nem 10-et. Ha a 10 kizárólagos korlát lenne, akkor a más nyelveken felnőtt programozókat érné meglepetés.

Másképpen úgy is tekinthetjük az aszimmetrikus korlátokat, hogy azok az *első foglalt* és az *első szabad* elemet jelölik egy sorozatban:



Ez a szemléletmód különösen hasznos, ha valamilyen átmeneti tárral, más néven pufferrel van dolgunk. Vegyünk például egy olyan függvényt, amelynek az a feladata, hogy valamilyen változó hosszúságú bemenetet `N` karakter hosszú blokkokba gyűjtsön, majd írja ki a tár tartalmát, ha az megtelt. Az átmeneti tár bevezetése például így nézhet ki:

```
#define N 1024
static char buffer[N];
```

Ehhez még hozzáadunk egy mutatót is, ami a tár aktuális elemére mutat:

```
static char *bufptr;
```

Mi legyen a `bufptr` feladata? Csábítóan hangzik, hogy a `bufptr` mindig mutasson a tár legutolsó foglalt karakterére, de mivel mi az aszimmetrikus korlátokat részesítjük előnyben, a `bufptr` a tár első szabad karakterére fog mutatni.

Ezt a megállapodást követve a `c` karaktert a következő módon tehetjük a tárba:

```
*bufptr++ = c;
```

Ha ez megvan, a `bufptr` ismét az első szabad karakterre mutat.

Az aszimmetrikus korlátokkal kapcsolatos megfigyeléseink alapján elmondhatjuk, hogy az átmeneti tár akkor üres, ha a `bufptr` és a `&buffer[0]` megegyezik, tehát kezdetben úgy írhatjuk le, hogy a tár üres, hogy

```
bufptr = &buffer[0];
```

vagy egyszerűbben

```
bufptr = buffer;
```

A tárban található karakterek száma mindig `bufptr-buffer` lesz, tehát úgy ellenőrizhetjük, hogy teljesen megtelt-e a tár, ha megnézzük hogy ennek a kifejezésnek az értéke egyenlő-e `N` értékével. A tárban található szabad karakterek száma ezért `N-(bufptr-buffer)` lesz.

Most, hogy tisztáztuk ezeket az előzetes megfigyeléseket, készen állunk, hogy megírjuk a programunkat, aminek a neve legyen `bufwrite`. A paraméterei az első kiírandó karakterre irányuló mutató és a kiírandó karakterek száma lesznek. Azt pedig a példa kedvéért tegyük fel, hogy meghívhatunk egy `flushbuffer` nevű függvényt, ami kiírja a tár tartalmát, majd visszaállítja a `bufptr` értékét a tár elejére.

```
void
bufwrite(char *p, int n)
{
        while(--n >= 0) {
                if (bufptr == &buffer[N])
                        flushbuffer();
                *bufptr++ = *p++;
        }
}
```

A `--n>=0` kifejezés segítségével n-szer tudunk végrehajtani egy műveletet. Hogy lássuk miért, vegyük azt az alapesetet, amikor `n=1`.[2]

Mivel a ciklus `n` alkalommal fut le, és minden alkalommal egy karaktert olvasunk be a bemeneti tárból, tudjuk, hogy minden bemeneti karakterre sor kerül, de egy darabbal sem többre.

Figyeljük meg az összehasonlítást az `&buffer[N]` értékével. Ilyen elem nincs is! A `buffer` elemeinek sorszáma `0`-tól `N-1`-ig tart. Azt írtuk, hogy

```
if (bufptr == &buffer[N])
```

a vele tulajdonképpen egyenértékű

```
if (bufptr > &buffer[N-1])
```

helyett, mert ragaszkodunk az aszimmetrikus korlátokkal kapcsolatos elveinkhez. A `bufptr` értékét a tár *után* következő első karakter címével akarjuk összehasonlítani, ami éppen a `&bufptr[N]` által megadott cím. De mi értelme egy olyan elemre hivatkozni, ami nem is létezik?

Szerencsére magára az elemre nem is kell hivatkoznunk, csak annak címére, ami viszont *igenis* létezik, legalábbis azokban a C megvalósításokban, amelyekkel eddig találkoztam. Az ANSI C kifejezetten engedélyezi ezt a használatot. A közvetlenül a tömb vége után következő nem létező elem címe kiolvasható és felhasználható összehasonlításokhoz és értékadásokhoz. Magára az elemre természetesen *tilos hivatkozni*!

A programunk a fent leírt formájában is működik, de kicsit még gyorsíthatunk rajta. Bár a (teljesítményfokozás) optimalizálás tárgyalása túlmutat a könyv hatáskörén, ezt az esetet mégis érdemes megvizsgálni a számolással kapcsolatos nézőpontból.

---

[2] A C legtöbb megvalósításában a `--n>=0` valószínűleg legalább olyan gyors lesz, mint az `n-->0` változat, sőt néhány megvalósításnál még gyorsabb is. Az első kifejezésnél kivonunk egyet `n` értékéből, és az eredményt összehasonlítjuk nullával. A második változat menti `n` értékét, majd kivon `n` értékéből egyet, majd a mentett értéket összehasonlítja nullával. Néhány fordító felismeri, hogy az így leírt kifejezést hatékonyabbá lehet tenni, de miért bíznánk a fordítóra magunkat, ha biztosra is mehetünk?

A program némi többletmunkát végez, mert a ciklus minden ismétlésekor *két* dolgot ellenőriz: azt, hogy a ciklus számlálója elérte-e a határt, illetve hogy megtelt-e a tár. Ez annak a következménye, hogy a karaktereket egyesével tesszük a tárba.

Tegyük fel, hogy módunkban áll egyszerre k karaktert tenni a tárba. A legtöbb C megvalósításban (és minden szabványos ANSI C megvalósításban) találunk egy memcpy nevű függvényt, ami éppen ezt teszi. Ezt a függvényt gyakran assembly nyelven írják meg, hogy még gyorsabb legyen. A függvényt egyébként könnyedén elkészíthetjük azokban a megvalósításokban is, amelyekből hiányzik.

```
void
memcpy(char *dest; const char *source, int k)
{
        while (--k >= 0)
                *dest++ = *source++;
}
```

A bufwrite úgy használja ki a memcpy előnyeit, hogy a karaktereket nem egyesével, hanem nagyobb csoportokban tehetjük vele a tárba. Így a ciklus minden egyes ismétlésekor kiüríti a tárat, ha szükséges, kiszámítja az áthelyezhető karakterek számát, majd áthelyezi azokat, és ennek megfelelően frissíti a számlálók értékét:

```
void
bufwrite(char *p, int n)
{
          while (n > 0) {
                  int k, rem;
                  if (bufptr == &buffer[N])
                          flushbuffer();
                  rem = N - (bufptr - buffer);
                  k = n > rem? rem: n;
                  memcpy(bufptr, p, k);
                  bufptr += k;
                  p += k;
                  n -= k;
          }
}
```

Sok programozó tart attól, hogy megírjon egy ilyen programot, nehogy hibát kövessen el valahol. Vannak olyanok is, akik bátran belevágnak, és el is szúrják. Ez a dolog bizony elég trükkös, és csak akkor próbálkozzunk vele, ha jó okunk van rá. De ha megvan a jó ok, akkor jó, ha értjük, hogyan lehet elkészíteni a programot. Ha végigvesszük az alapvető eseteket és óvatosan bánunk a számokkal, akkor biztosak lehetünk benne, hogy próbálkozásunkat siker koronázza.

A ciklus legelején az `n` a tárba helyezendő karakterek számát jelöli. Világos tehát, hogy az eljárást addig folytathatjuk, amíg `n>0`. Minden alkalommal, amikor végrehajtjuk a ciklust, valamely `k` számú karaktert helyezünk a tárba. Ezt az utolsó négy sor végzi el úgy, hogy (1) bemásol `k` karaktert a tárba az első szabad karaktertől kezdődően, (2) az első szabad helyet `k` karakterrel továbblépteti, (3) a bemeneti mutatót `k` karakterrel odébb viszi és (4) az áthelyezendő karakterek számát `k`-val csökkenti. Könnyen belátható, hogy ezek az utasítások helyesen járnak el.

A ciklus elején található ellenőrzést megőriztük az előző változatból. Ha a tár megtelt, akkor kiürítjük (és visszaállítjuk a `bufptr` értékét). Ezután az ellenőrzés után tehát biztosan lesz valamennyi hely a tárban.

A dolog neheze az, hogy meghatározzuk `k` értékét, ami az a lehető legnagyobb szám kell, hogy legyen, ahány karaktert még biztonságosan el tudunk helyezni a tárban. Ez az érték a következő két mennyiség közül a kisebbik lesz: a bemenetből még hátralévő karakterek száma és a tárban található szabad karakterek száma (amit a `rem` változóba teszünk).

A `rem` értékét kétféleképpen kaphatjuk meg. A példánkban ebből az egyiket láthatjuk. A jelenleg szabad karakterek számát megkapjuk, ha a jelenleg foglalt karakterek számát (`bufptr-buffer`) kivonjuk a tárban található összes karakter számából (`N`), vagyis az `N-(bufptr-buffer)` érték kiszámításával.

A másik módszer az, ha a tár üres részét egy tartományként kezeljük, és közvetlenül kiszámítjuk az értékét. Ha így teszünk, akkor a `bufptr` lesz a tartomány eleje, a `buffer+N` (ami egyenértékű a `&buffer[N]`

kifejezéssel) pedig a tartomány vége (után következő karakter). Ebből a szemszögből azt mondhatjuk, hogy a tárban még `(buffer+N)-bufptr` karakternyi szabad hely van. Egy kis spekuláció után rájöhetünk, hogy a

```
(buffer+N)-bufptr
```

és az

```
N-(bufptr-buffer)
```

egyenértékűek.

Nézzünk meg egy másik számolós példát. Legyen adva egy olyan program, ami valamilyen sorrendben egész számokat állít elő. Írjuk ki oszlopokba ezeket az egész számokat. Pontosabban, a kimenet álljon néhány olyan oldalból, amelyek mindegyikén `NCOLS` oszlopban `NROWS` sornyi elem található. Az egymást követő értékek az oszlopokban felülről lefelé kövessék egymás, ne a sorokban balról jobbra.

Könnyítésképpen néhány feltevésből indulunk ki, hogy a probléma számolós részére összpontosíthassunk. Először is induljunk ki abból, hogy a programunk két függvényből épül fel. Legyenek ezek a `print` és a `flush`. Azt, hogy milyen értékeket kell kiírnunk, egy külső program fogja eldönteni. Ez a program minden alkalommal meghívja a `print` függvényt, ha ismertté válik a következő érték, az utolsó érték előállítása után pedig a `flush` függvényt is meghívja egyszer.

Másodszor, tegyük fel, hogy három függvény áll rendelkezésünkre a kiírás elvégzéséhez: a `printnum`, ami egyetlen érték kiírására szolgál az adott lap aktuális helyén, a `printnl`, ami új sort kezd és a `printpage`, ami új oldalt kezd. Minden sor kiírása után meg kell hívni a `printnl` függvényt, még akkor is, ha a lap utolsó soráról van szó. Ezek a függvények balról jobbra haladva töltik meg a kimenet sorait. Ha egy sort már kiírtunk, akkor nincs mód rá, hogy visszamenjünk és módosítsuk.

Az első, amit észrevehetünk a feladattal kapcsolatban, hogy a megoldásához valamilyen átmeneti tárra lesz szükségünk. A második osz-

lop első elemét egész addig nem ismerjük, amíg meg nem kaptuk az első oszlop összes elemét. Az első sort viszont már azelőtt teljes egészében ki kell írnunk, hogy az első oszlop második elemét kiírnánk!

Mekkora legyen az átmeneti tár mérete? Első látásra elég nagynak kell lennie ahhoz, hogy elférjen benne egy egész lapnyi szám, de ha közelebbről megnézzük a dolgot, akkor rájövünk, hogy erre nincs szükség. Az utolsó oszlop elemeit azonnal kiírhatjuk, amint megkapjuk azokat, hiszen eljárásunk természeténél fogva ehhez minden információ rendelkezésünkre fog állni. A tár ezért kihagyhatja az utolsó oszlopot:

```
#define BUFSIZE (NROWS*(NCOLS-1))
static int buffer[BUFSIZE);
```

A tárat statikus változóként vezetjük be, megelőzve ezzel azt, hogy a program egy másik része hozzáférjen. A 4.3. részben részletesen szó lesz a statikus változók bevezetéséről.

A `print` létrehozásánál nagyjából a következő stratégiát fogjuk alkalmazni: tegyük az értéket a tárba, hacsak meg nem telt, amikor is ki kell írnunk az egész sort, ami ezt az értéket tartalmazza. Ha ennek a sornak a kiírásával kiürül a tár, az azt jelenti, hogy az oldal végéhez értünk.

Figyeljük meg, hogy az értékek nem olyan sorrendben jönnek ki a tárból, ahogy belekerültek. Az értékeket oszlopokban kapjuk, de soronként kell kiírnunk őket. Ez felveti azt a kérdést, hogy a sorok vagy az oszlopok legyenek egymás mellett a tárban? Mi most önkényesen úgy döntünk, hogy a tárban az azonos oszlopban lévő elemek kerüljenek egymás mellé. Ez azt jelenti, hogy a bejövő elemek egyszerűen egymás után bekerülnek a tárba, de a kifelé vezető útjuk már bonyolultabb lesz. Ahhoz tehát, hogy nyomon követhessük a tárba bemenő elemeket, elég lesz egy egyszerű mutató, ami kezdetben a tár első elemére mutat:

```
static int *bufptr = buffer;
```

Ezen a ponton már van némi elképzelésünk a `print` szerkezetéről. Paraméterként kap egy egész értéket, és ha van hely a tárban, akkor

beleteszi ezt az értéket. Ezenkívül azonban valami rejtélyeset is csinál. Írjuk is le azt, ami már megvan:

```
void
print(int n)
{
        if (bufptr == &buffer[BUFSIZE]) {
                /*valami rejtélyes*/
        } else
                *bufptr++ = n;
}
```

A „valami rejtélyes" nem más, mint az aktuális sor összes elemének kiírása, a sor növelése eggyel, és új lap kezdése, ha a jelenlegi oldal összes sorát kiírtuk. Ehhez nyilvánvalóan fel kell jegyeznünk az aktuális sor számát, amit a statikus `row` változóba teszünk.

Hogyan írjuk ki az aktuális sor összes elemét? Ez először elég bonyolultnak tűnik, de egyáltalán nem az, ha a megfelelő oldalról közelítjük meg a kérdést. Tudjuk, hogy a `row` sorszámú sor első eleme egyszerűen a `buffer[row]` értéke, és azt is, hogy a `buffer[row]` elem létezik, hiszen nem lennénk itt, ha az első oszlop nem lenne teljesen tele. Azt is tudjuk, hogy egy sor egymás mellett lévő elemeit `NROWS` elem választja el egymástól. Végül tudjuk, hogy a `bufptr` közvetlen a tár utolsó foglalt eleme utáni helyre mutat. Az aktuális sorhoz tartotó összes elemet tehát az alábbi ciklussal írhatjuk ki:

```
int *p;
for (p = buffer+row; p < bufptr; p += NROWS)
        printnum(*p);
```

A tömörség kedvéért itt a `buffer+row` kifejezést használtuk a `&buffer[row]` helyett.

A „valami rejtélyes" többi része már egyszerű. Írjuk ki az aktuális értéket, zárjuk le a sort, és kezdjünk új lapot, ha éppen egy lap utolsó sorát írtuk ki:

```
printnum(n);
printnl();
```

```
if (++row == NROWS) {
        printpage();
        row = 0;
        bufptr = buffer;
}
```

A kész `print` függvény tehát így néz ki:

```
void
print(int n)
{
        if (bufptr == &buffer[BUFSIZE]) {
                static int row = 0;
                int *p;
                for (p = buffer+row; p < bufptr;
                                p += NROWS)
                        printnum(*p);
                printnum(n);
                printnl();
                if (++row == NROWS) {
                        printpage();
                        row = 0;
                        bufptr = buffer;
                }
        } else
                *bufptr++ = n;
}
```

Most már majdnem készen is vagyunk. Csak a `flush` függvényt kell megírnunk, aminek az lesz a feladata, hogy kiírja azt a töredék oldalt, ami a tárban maradt. Ezt egy olyan belső ciklussal végezzük el, ami azonos a `printnum` függvénnyel, és ezt minden sorra megismételjük:

```
void
flush()
{
        int row;
        for (row = 0; row < NROWS; row++) {
                int *p;
                for (p = buffer+row; p < bufptr;
                                p += NROWS)
```

```
                printnum(*p);
            printnl();
        }
        printpage();
    }
```

A `flush` függvénynek ez a változata túlságosan is szó szerint veszi a dolgát. Ha az utolsó oldal csak egy (töredék) oszlopból áll, akkor is kitölti az egész oldalt, üres sorokkal kiegészítve. Sőt, ha az utolsó oldal üres, akkor is kiírja azt, teletűzdelve üres sorokkal. Bár ez nem volt a feladat része, az esztétikai érzékünk azt mondatja velünk, hogy ha már nincs mit kiírni, akkor fejezzük be a kiírást. Ezt úgy tehetjük meg, hogy kiszámoljuk, hány elem van még a tárban. Ha már nincs mit kiírni, akkor új oldalt sem kezdünk:

```
void
flush()
{
        int row;
        int k = bufptr - buffer;
        if (k > NROWS)
            k = NROWS;
        if (k > 0) {
            for (row = 0; row < k; row++) {
                int *p;
                for (p = buffer+row;
                  p < bufptr; p += NROWS)
                    printnum(*p);
                printnl();
            }
            printpage();
        }
}
```

3.7. A kiértékelési sorrend

A 2.2. részben a műveletek (elsőbbségi) sorrendjéről volt szó. A kiértékelési sorrend egészen más tészta. A műveleti sorrend az, ami meghatározza, hogy az

```
a + b * c
```

kifejezést úgy kell érteni, hogy

```
a + (b * c)
```

és nem úgy, hogy

```
(a + b) * c
```

A kiértékelési sorrend arról gondoskodik, hogy a

```
if (count != 0 && sum/count < smallaverage)
        printf ("average < %g\n", smallaverage);
```

akkor se okozzon „nullával való osztás" hibát, ha a `count` értéke nulla.

Néhány C műveletnél a tényezők kiértékelése előre ismert, meghatározott módon történik. Másoknál viszont nem. Nézzük meg például a következő kifejezést:

```
a < b && c < d
```

A nyelv leírása kimondja, hogy először az `a<b` kifejezést értékeli ki a program. Ha az `a` tényleg kisebb a `b` változó értékénél, akkor a `c<d` kifejezést is ki kell értékelni a teljes kifejezés értékének meghatározásához. Ha viszont `a` nagyobb vagy egyenlő a `b` változó értékénél, akkor a `c<d` kifejezést már nem értékeli ki a program.

Az `a<b` kiértékelésekor viszont a fordító lehet, hogy az `a`, de az is lehet, hogy a `b` értékét határozza meg előbb. Néhány gépen az is előfordulhat, hogy a kiértékelésük párhuzamosan történik.

A C nyelvben csak négy művelet esetében, az `&&`, a `||`, a `?:` és a `,` műveleteknél létezik előre megadott kiértékelési sorrend. Az `&&` és a `||` műveletek először a bal oldali tényezőt értékelik ki, aztán szükség esetén a jobb oldalit. A `?:` művelet három tényezőből áll. Az `a?b:c` művelet először az `a` értékét állapítja meg, azután az `a` értékétől függően vagy a `b`, vagy a `c` kiértékelését végzi el. A `,` műve-

let először a bal oldalán álló tényezőt értékeli ki, majd ezt az értékét eldobva kiértékeli a jobb oldalon álló tényezőt.[3]

A többi C műveletnél a tényezők kiértékelése meghatározatlan sorrendben történik. Az értékadó műveleteknél különösképpen nincs semmilyen garancia a kiértékelés sorrendjét illetően.

Az `&&` és a `||` műveleteknél fontos, hogy az összehasonlítások a megfelelő sorrendben történjenek. Itt például:

```
if (y != 0 && x/y > tolerance)
        complain();
```

életbevágó, hogy az `x/y` kiértékelésére csak akkor kerüljön sor, ha az `y` értéke különbözik nullától.

A következő módszer azért nem megfelelő az `x` tömb első `n` elemének átmásolására az `y` tömbbe, mert túl sokat feltételez a kiértékelési sorrendről:

```
i = 0;
while (i < n)
            y[i] = x[i++];
```

A gond az, hogy semmiféle garancia nincs arra nézve, hogy az `y[i]` címét még azelőtt kiértékeli a program, hogy megnövelné az `i` értékét. Van olyan megvalósítás, ahol így történik, de van, ahol nem. A következő hasonszőrű változat ugyanezen oknál fogva bukik meg:

```
i = 0;
while (i < n)
            y[i++] = x[i];
```

[3] A függvények paramétereit elválasztó vesszők nem vessző operátorok. Az `x` és `y` lehívása például meghatározatlan sorrendben történik az `f(x,y)` esetben, de a `g((x,y))` esetében nem ez a helyzet. Az utóbbi példában a `g` függvénynek egy paramétere van. A paraméter értékének meghatározása során a példakód először kiértékeli az `x` változót, majd eldobja ezt az értéket, és kiértékeli az `y` változót.

Az alábbi megoldás viszont szépen működik:

```
i = 0;
while (i < n) {
        y[i] = x[i];
        i++;
}
```

Ezt persze rövidebben így is írhatjuk:

```
for (i = 0; i < n; i++)
        y[i] = x[i];
```

3.8. Az &&, || és ! műveletek

A C nyelvben a logikai műveleteket két csoportra oszthatjuk, és ezek néha felcserélhetők. Külön csoportba tartoznak az `&`, a `|` és a `~` logikai bitműveletek és az `&&`, a `||` és a `!` logikai műveletek. Ha egy programozó felcseréli az egyik műveletet a másik csoportba tartozó művelettel, akkor jó kis meglepetésben lehet része. Úgy tűnhet, hogy a program a csere ellenére is helyesen működik, de valójában lehet, hogy ez csak a véletlen műve. Az `&`, `|` és `~` műveletek bitek sorozataként kezelik a művelet tagjait, és a műveletet az egyes biteken végzik el.
A `10&12` értéke például `8` (kettes számrendszerben `1000`), mert az `&` művelet a `10` és a `12` kettes számrendszerbeli ábrázolását veszi alapul (kettes számrendszerben `1010` és `1100`), majd eredményül egy olyan bitsorozatot ad vissza, amiben azok a bitek lesznek bekapcsolva, amelyek mindkét számnál be vannak kapcsolva ugyanazon a helyen. Hasonlóképpen, a `10|12` művelet eredménye `14` (kettes számrendszerben `1110`), a `~10` eredménye pedig `-11` (kettes számrendszerben `11...110101`), legalábbis a kettes komplemenst használó gépeken.

Az `&&`, `||` és `!` műveletek viszont „igaz" vagy „hamis" értékként kezelik a művelet tagjait, méghozzá úgy, hogy a szokásos jelölés szerint a `0` érték „hamisnak", minden más érték pedig „igaznak" minősül. A művelet eredménye vagy `1` lesz, ami az „igazat" jelenti, vagy `0`, ami a „hamisat". Az eredmény csak `1` vagy `0` lehet, és az `&&`, illetve a `||`

műveleteknél a jobb oldali tényező kiértékelése meg sem történik, ha a művelet értéke meghatározható pusztán a bal oldalon álló tényező kiértékelésével.

A `!10` eredménye tehát 0, mert `10` különbözik nullától, a `10&&12` eredménye `1`, mert a `10` és a `12` is különbözik nullától, és a `10||12` eredménye szintén `1`, mert a `10` különbözik nullától. Sőt az utóbbi kifejezésben a `12` kiértékelése meg sem történik, éppen úgy, ahogy az `f()` kiértékelése sem a `10||f()` kifejezésben.

Nézzük meg az alábbi programrészletet, ami egy táblázatban keres egy adott elemet:

```
i = 0;
while (i < tabsize && tab[i] != x)
          i++;
```

A ciklus alapgondolata az, hogy ha a ciklus végén az `i` egyenlő a `tabsize` változó értékével, akkor nem találtuk meg a keresett elemet. Máskülönben az `i` a keresett elem indexével lesz egyenlő. Figyeljük meg az aszimmetrikus korlátok használatát ebben a ciklusban.

Tegyük fel, hogy az `&&` jel helyére véletlenül az `&` jelet írjuk:

```
i = 0;
while (i < tabsize & tab[i] != x)
          i++;
```

Ekkor a ciklus továbbra is működni látszik, de csak azért, mert két szempontból is mázlink van.

Először is azért, mert a példában szereplő mindkét összehasonlítás olyan fajta, hogy ha a feltétel hamis, akkor az eredmény `0`, ha a feltétel igaz, akkor az eredmény `1` lesz. Amíg mind az `x` mind az `y` változók értéke `1` vagy `0`, addig az `x&y` és az `x&&y` értéke egyenlő lesz. Ha azonban az egyik összehasonlítást kicserélnénk egy olyannal, ami nem az eggyel, hanem egy másik nem nulla értékkel jelezné az igaz értéket, akkor a ciklus már nem működne.

A második mázlink az, hogy ha csak egy elemmel megyünk túl a tömb végén, az általában nem jár komoly következményekkel, különösen ha a program nem változtatja meg ezt az elemet. A módosított program azért szalad túl a tömb végén, mert az `&` művelet az `&&` művelettől eltérően mindig mindkét oldalt kiértékeli. Így amikor utoljára megyünk végig a cikluson, a `tab[i]` értékét attól függetlenül megnézi a program, hogy az `i` értéke egyenlő a `tabsize` értékével. Ha `tabsize` egyenlő a `tab` elemeinek számával, akkor ezzel a tömb egy nem létező elemét olvassuk be.

Emlékezzünk vissza, hogy a 3.6. részben azt mondtuk, hogy az nem baj, ha a tömb utolsó elemét követő elem címét próbáljuk elérni. Itt azonban magát az elemet próbáljuk elérni, aminek beláthatatlan következményei lehetnek, és csak nagyon kevés C megvalósítás veszi észre ezt a hibát.

3.9. Az egész értékek túlcsordulása

A C nyelvben kétféle műveletet végezhetünk az egész értékekkel: előjeleset és előjel nélkülit. Az előjel nélküli műveleteknél nem létezik túlcsordulás. Az összes előjel nélküli művelet modulo 2^n történik, ahol az n az eredményül kapott érték bitjeinek száma. Ha egy számtani művelet egyik tényezője előjeles, a másik pedig előjel nélküli, akkor az előjeles tényező átalakul előjel nélkülivé, a túlcsordulás tehát így sem lehetséges. Ha viszont mindkét tényező előjeles, akkor a túlcsordulás már bekövetkezhet, aminek az eredménye *megjósolhatatlan*. Egy olyan műveletről, aminek az eredménye túlcsordul, semmit sem tételezhetünk fel teljes biztonsággal.

Tegyük fel például, hogy `a` és `b` két egész változó, amelyekről tudjuk, hogy nem negatívak és szeretnénk ellenőrizni, hogy `a+b` túlcsorduláshoz vezet-e. A következő megoldás elég nyilvánvalónak tűnik:

```
if (a + b < 0)
        complain();
```

Csakhogy ez így nem működik. Ha az a+b értéke túlcsordult, akkor annak eredménye megjósolhatatlan. Vannak olyan gépek például, amelyeken az összeadási műveletek egy belső regiszter állapotát négy érték egyikére állítják be: pozitív, negatív, nulla vagy túlcsordult. Egy ilyen gépen a fordítónak minden joga megvan arra, hogy a fenti példát úgy valósítsa meg, hogy előbb összeadja a és b értékét, majd utána megvizsgálja ezt a belső regisztert, hogy negatív-e az állapota. Ha a művelet túlcsordulást okozott, akkor a regiszter állapota túlcsordult lesz, a vizsgálat tehát sikertelen.

Helyes megoldáshoz úgy jutunk, ha a és b változókat előjel nélkülivé alakítjuk:

```
if ((unsigned) a + (unsigned) b > INT_MAX)
    complain();
```

Itt az INT_MAX egy előre megadott állandó, ami a legnagyobb lehetséges egész értéket tárolja. Az ANSI C nyelvben ennek meghatározását a <limits.h> állományban találjuk. Más megvalósításoknál elképzelhető, hogy nekünk kell meghatározni ezt az értéket.

Egy másik lehetőséghez egyáltalán nincs szükség az előjel nélküli műveletekre:

```
if (a > INT_MAX - b)
    complain();
```

3.10. Érték visszaadása a main függvényből

A legegyszerűbb C program, vagyis a

```
main()
{
}
```

tartalmaz egy mákszemnyi hibát. A többi függvényhez hasonlóan a main esetében is azt feltételezzük, hogy egy egész értéket ad vissza, ha nem határozzuk meg visszatérési értékének a típusát. Ebben a programban pedig ilyen típusmeghatározást nem találunk.

Ez általában nem okoz gondot. Egy egész függvény, ami nem ad vissza semmilyen értéket, hallgatólagosan egy valahonnan összeszedett egész értékkel tér vissza. Addig nincs is gond, amíg senki sem használja ezt az értéket.

Van viszont néhány olyan helyzet, ahol a `main` függvény visszatérési értéke fontos szerepet kap. Számos C megvalósításban ezt az értéket arra használják, hogy a program elárulja az operációs rendszernek, hogy a végrehajtása sikeres vagy sikertelen volt-e. A visszaadott `0` érték jellemzően a sikert, minden más érték pedig a kudarcot jelenti. Egy olyan program végrehajtása, aminek a `main` függvénye semmilyen értéket nem ad vissza, valószínűleg sikertelennek tűnik majd. Ez meglepő következményekkel járhat, ha olyan környezetben – például egy szoftverfelügyelő rendszerben – használják, ahol bizony számít, hogy egy program futása sikeres volt-e vagy sem.

Szigorúan véve tehát, a mi kis C programunk így néz ki helyesen:

```
main()
{
        return 0;
}
```

Vagy így:

```
main()
{
        exit(0);
}
```

A klasszikus `"hello world"` programnak pedig így kellene kinéznie:

```
#include <stdio.h>

main()
{
        printf ("Hello world\n");
        return 0;
}
```

3.1. gyakorlat. Tegyük fel, hogy létre sem hozhatjuk egy, a tömb korlátain kívül található elem címét. Hogy néznének ki ekkor a 3.6. részben található `bufwrite` programok?

3.2. gyakorlat. Hasonlítsuk össze a `flush`-nak a 3.6. részben található utolsó változatát ezzel:

```
void
flush()
{
        int row;
        int k = bufptr - buffer;
        if (k > NROWS)
                k = NROWS;
        for (row = 0; row < k; row++) {
                int *p;
                for (p = buffer+row; p < bufptr;
                                p += NROWS)
                        printnum(*p);
                printnl();
        }
        if (k > 0)
                printpage();
}
```

3.3. gyakorlat. Írjunk olyan függvényt, ami bináris keresést hajt végre egy egészeket tartalmazó rendezett táblázatban. A függvény paraméterei a következők legyenek: egy mutató, ami a táblázat elejére mutat, a táblázatban található elemek száma és a keresett érték. A függvény visszatérési értéke vagy a keresett elemet címző mutató legyen, vagy egy `NULL` mutató, ha a keresett elem nincs a táblázatban.

4

Szerkesztés

Egy C program több olyan részből állhat, amelyeket külön-külön fordítunk le, majd egy *összekötő* vagy *összekapcsoló* (linker), *összeszerkesztő* (linkage editor) vagy *betöltő* (loader) elnevezésű program segítségével egyetlen egésszé kovácsoljuk a program részeit. Mivel a fordítóprogram általában egyszerre csak egy fájlt lát, az olyan hibákat képtelen észrevenni, amelyekhez egyszerre kellene ismernie több forrásfájl tartalmát. Ezen kívül a linker sok rendszeren nem is tartozik az adott C megvalósítás felügyelete alá, így a C hibák azonosítására sem képes.

Némelyik C megvalósításban, de nem mindegyikben, találunk egy `lint` nevű programot, ami elcsípi ezeket a hibákat. Ha rendelkezésünkre áll egy ehhez hasonló program, akkor nem lehet elégszer elmondani, hogy feltétlenül használjuk is.

Ebben a fejezetben szemügyre veszünk egy jellegzetes összekapcsolót, megfigyeljük, hogyan kezeli a C programokat, majd levonjuk a megfelelő következtetéseket azokkal a hibalehetőségekkel kapcsolatban, amelyek a linker természetéből adódnak.

4.1. Mi fán terem a linker?

A C nyelv egyik fontos vonása az *önálló fordítás* (külön fordítás) lehetősége. A külön-külön lefordított programok később összeköthetők. Az összekapcsolást végző program azonban nem a C fordító része, így a C nyelvet sem ismeri igazán. De akkor honnan tudja, hogyan kell összeszerkeszteni a C programokat? Habár a szerkesztők nem értenek C nyelven, a gépi kódot és a memória szerkezetét rendkívül jól ismerik. A C fordítóknak pedig az a feladata, hogy úgy fordítsák le a C programokat, hogy azt a linker megértse.

Egy hétköznapi linker feladata általában a C vagy assembly fordítók által létrehozott *tárgymodulok* egyesítése. Ennek a folyamatnak az eredménye egy *betölthető modul* vagy egy *végrehajtható állomány* lesz, amit az operációs rendszer már önállóan is képes futtatni. Az összekapcsoló bemenetként megkapja a tárgymodulok egy részét, a többit pedig szükség szerint egy *könyvtárból* hívja le. Ez utóbbiban olyan tárgykódok vannak, mint például a `printf` és társai.

Az összekapcsoló *külső objektumok* gyűjteményeként kezeli a tárgymodulokat. Minden egyes külső objektum a gép memóriájának egy részletét ábrázolja, és egy *külső név* segítségével azonosítható. Tehát minden olyan függvény külső objektum, amit nem statikusként (`static`) vezettünk be, ahogyan minden olyan külső változó is az, amelyik nem statikus. Néhány megvalósításban a statikus függvényekből és változókból is külső objektumok lesznek. Ilyenkor az elnevezésüket úgy választja meg a program, hogy ne ütközzenek a többi forrásfájl hasonló nevű változóival.

A legtöbb linker nem engedi, hogy két különböző külső objektumnak ugyanaz legyen a neve egyazon betölthető modulban. A sok egyesítendő tárgymodulban azonban előfordulhatnak azonos nevű külső objektumok. A linker egyik legfontosabb feladata, hogy kezelje a nevek ütközését.

Az ilyen ütközések kezelésének legegyszerűbb módja, ha nem engedélyezzük őket. A függvények esetében ez teljesen rendben is van, hiszen egy olyan programot, amiben két függvénynek ugyanaz a ne-

ve, vissza kell utasítanunk. Nehezebb a dolgunk, ha változókról van szó. A különféle linkerek más-más módon kezelik ezt a helyzetet. Később majd látni fogjuk, hogy ennek mi a jelentősége.

Ezeknek az információknak a birtokában már kezd kialakulni bennünk egy kép az összekapcsolók működéséről. Az összekapcsoló bemenetként tárgymodulokat és könyvtárakat kap. Kimenetként egy betölthető modult ad vissza, amit a bemenet beolvasása közben épít fel. Minden tárgymodul összes külső objektumánál ellenőrzi, hogy létezik-e már az adott néven objektum a betölthető modulban.
Ha nem, akkor hozzáadja a külső objektumot a betölthető modulhoz.
Ha igen, akkor valamilyen megoldást keres az ütközés elhárítására.

A tárgymodulokban a külső objektumok mellett *hivatkozásokat* is találhatunk más modulokban lévő külső objektumokra. Ha például egy C programból előállított tárgymodul meghívja a `printf` függvényt, akkor tartalmazni fog egy hivatkozást a `printf` függvényre, ami feltehetőleg egy külső objektum lesz valamelyik könyvtárban. A betölthető modul építése közben az összekapcsolónak nyomon kell követnie ezeket a külső hivatkozásokat. Amikor beolvas egy tárgymodult, feloldja a modulban található összes hivatkozást a külső objektumokra, azáltal, hogy megjegyzi, hogy ezeknek az objektumoknak már van meghatározása.

Mivel az összekapcsolást végző program nemigen ismeri a C nyelvet, számos olyan hiba létezik, amit nem vesz észre. Ha az adott megvalósításban találunk egy `lint` programot, akkor feltétlenül azt használjuk!

4.2. Deklaráció és definíció

Az alábbi deklarációt

```
int a;
```

ami a függvényeken kívül található, az `a` külső objektum *meghatározásának* (definíciójának) nevezzük. A fenti sor szerint `a` egy külső egész érték, aminek a tárolásához szükséges helyet lefoglaljuk a me-

móriában. Mivel kezdőértéket nem adunk neki, az értéke 0 lesz (azoknál a linkereknél, amelyek ezt nem garantálják más nyelven íródott programok esetében, a C fordító feladata, hogy a megfelelő varázsszavak segítségével rávegye erre a szerkesztőprogramot).

Az alábbi deklaráció

```
int a = 7;
```

meghatározza a változót, és egyértelmű kezdőértéket is ad neki. Ezzel nemcsak lefoglalja a helyet számára a memóriában, hanem azt is megmondja, hogy a memóriába milyen érték kerül. A következő deklaráció

```
extern int a;
```

nem határozza meg az a változót. Itt is azt állítjuk, hogy a egy külső egész változó, de az extern kulcsszó segítségével azt is egyértelműen megmondjuk, hogy az a tárolásához szükséges helyet valahol máshol foglaljuk le. A linker szemszögéből egy ilyen deklaráció hivatkozás az a külső objektumra, és nem annak meghatározása. Mivel ez a bevezetés egyértelműen külső objektumra hivatkozik, egy függvény belsejében is ugyanez lesz a jelentése. Az alábbi srand függvény egy random_seed nevű külső változóban tárolja egész paraméterének értékét:

```
void
srand(int n)
{
        extern int random_seed;
        random_seed = n;
}
```

Minden külső objektumot meg kell határoznunk valahol. Egy olyan programban tehát, amiben ez áll:

```
extern int a;
```

annak is ott kell lennie valahol, hogy

```
int a;
```

vagy ugyanabban, vagy egy másik programfájlban.

Mi a helyzet egy olyan program esetében, ami egynél többször határozza meg ugyanazt a külső változót? Vagyis mi van akkor, ha mondjuk az

```
int a;
```

kifejezés kettő vagy több különböző forrásfájlban is szerepel? És ha az egyik fájlban az

```
int a = 7;
```

kifejezés, a másikban pedig az

```
int a = 9;
```

kifejezés található? A különféle rendszerek ebben az esetben másképp viselkednek. Van azonban egy szigorú szabály, miszerint minden külső változót *pontosan egyszer* kell meghatározni. Ha több külső meghatározás is kezdőértéket rendel a változóhoz, ami például

```
int a = 7;
```

lesz az egyik fájlban és

```
int a = 5;
```

a másikban, akkor a legtöbb rendszer visszadobja a programot.
Ha viszont egy külső változó meghatározása több különböző fájlban is szerepel kezdőérték nélkül, akkor lesz olyan rendszer, amelyik elfogadja a programot, és lesz, amelyik nem. Az egyetlen mód, hogy az összes C megvalósításban elkerüljük az ezzel kapcsolatos gondokat, ha minden külső változót pontosan egyszer határozunk meg.

4.3. A névütközések és a static módosító

Két azonos nevű külső objektum ugyanaz az objektum lesz, akkor is, ha a programozónak nem ez volt a szándéka. Tehát ha két különböző forrásfájlban is szerepel, hogy

```
int a;
```

akkor az vagy hibához vezet (ha a linker tiltja a megkettőzött külső változókat), vagy megosztoznak az a egyetlen példányán, ha tetszik, ha nem.

Ez még abban az esetben is igaz, ha az `a` egyik meghatározása a rendszerkönyvtárban található. Egy átgondolt könyvtárban persze nem külsőként határozzák meg az `a` változót, de nem könnyű megjegyezni az összes olyan változót, amelynek meghatározása szerepel a könyvtárban. A `read` és a `write` és a hozzájuk hasonló elnevezések könnyen kitalálhatók, míg mások csak nehezen.

Az ANSI C szabvány azzal könnyíti meg a névütközések elkerülését, hogy felsorolja azokat a függvényeket, amelyek ilyen ütközésekhez vezethetnek. Ha egy könyvtári függvény egy olyan másik könyvtári függvényt hív meg, ami nem szerepel ebben a listában, akkor azt egy „rejtett név" segítségével teszi. Ezáltal a programozó nyugodtan meghatározhat egy `read` nevű függvényt anélkül, hogy azon kellene tépelődnie, hogy a `getc` majd ezt a `read` függvényt hívja meg a rendszerfüggvény helyett. A legtöbb C megvalósítás azonban még nem így működik, így ezek az ütközések továbbra is gondot jelenthetnek.

Hasznos eszköz az efféle ütközések számának csökkentésére a `static` módosító. Az alábbi deklaráció például

```
static int a;
```

ugyanazt jelenti, mint az

```
int a;
```

egyetlen forrásfájlon belül, de az előbbi esetben az a rejtve marad a többi fájl elől. Ha tehát több függvénynek kell osztoznia néhány külső objektumon, akkor tegyük az összes függvényt egyetlen fájlba, és ugyanebben a fájlban vezessük be a szükséges objektumokat a `static` használatával.

Ugyanez igaz a függvényekre is. Ha az `f` függvény meghívja a `g` függvényt, és csak az `f` függvénynek kell képesnek lennie a `g` meghívására, akkor az `f` és `g` függvényeket ugyanabba a fájlba tehetjük, és a `g` függvényt statikusként vezethetjük be:

```
static int
g(int x)
{
        /*valami*/
}
void f()
{
        /*még valami*/
        b = g(a);
}
```

Ekkor még számos olyan fájlunk lehet, amelyiknek saját `g` függvénye van, ha mindegyiket (vagy egy kivételével mindegyiket) statikusként vezettük be. Összefoglalva: az olyan függvényeket, amelyeket csak az adott fájlon belül található többi függvény hívhat meg, mindig statikusként vezessük be, hogy elkerüljük a véletlen névütközéseket.

4.4. Argumentumok, paraméterek és visszatérési értékek

Minden C függvénynek van egy sor *paramétere*. Ezek olyan változók, amelyek a függvény meghívásakor kapnak kezdőértéket. Ennek a függvénynek egy darab egész típusu paramétere van:

```
int
abs(int n)
{
        return n<0? -n: n;
}
```

Néhány függvény paraméterlistája üres:

```
void
eatline()
{
        int c;
        do c = getchar();
        while (c != EOF && c != '\n');
}
```

A függvényeket *argumentumokkal* (átadott értékekkel) hívjuk meg. Ebben a példában az `abs` függvénynek az `a-b` kifejezés értékét adjuk át:

```
if (abs(a-b) > n)
        printf("difference is out of range\n");
```

Az üres paraméterlistával rendelkező függvényeknél üres lesz az argumentumlista is:

```
eatline();
```

Minden C függvénynek van egy *eredménytípusa* is, ami vagy `void` (üres), vagy a függvény által visszaadott érték típusa. Az eredmény típusának fogalmát könnyebb megérteni, mint az átadott értékek típusát, ezért előbb ezzel foglalkozunk.

Ha egy függvény deklarációja vagy meghatározása minden fájlban megelőzi annak első meghívását, akkor az eredmény típusa nem okoz gondot. Vegyünk például egy `square` nevű függvényt, ami négyzetre emeli a számára átadott `double` típusú értéket:

```
double
square(double x)
{
        return x * x;
}
```

Nézzünk egy olyan programot is, ami a fenti `square` függvényt használja:

```
main()
{
        printf("%g\n", square(0.3));
}
```

Ahhoz, hogy ez a program helyesen működjön, vagy meg kell határoznunk a `square` függvényt még a `main` függvény előtt:

```
double
square(double x)
{
        return x * x;
}

main()
{
        printf("%g\n", square(0.3));
}
```

vagy be kell vezetnünk azt még a `main` meghívása előtt:

```
double square(double);

main()
{
        printf ("%g\n", square(0.3));
}

double
square(double x)
{
        return x * x;
}
```

Ha meghívunk egy függvényt, mielőtt deklaráltuk vagy meghatároztuk volna, akkor a visszatérési értékét egész (`int`) típusúnak veszi a program. Tehát ha a `main` függvényt magára hagyjuk a fájlban:

```
main()
{
        printf("%g\n", square(0.3));
}
```

akkor helytelen értéket fog visszaadni, amikor összeszerkesztjük a `square` függvénnyel, hiszen a `main` úgy veszi, hogy a `square` egész értéket ad vissza, pedig az a valóságban `double` típusú értékkel tér vissza.

Mi a helyzet akkor, ha két külön fájlban szeretnénk meghatározni a `main` és a `square` függvényt? A `square` függvénynek természetesen csak egy meghatározása lehet. Ha a hívás és a meghatározás két külön fájlban van, akkor a hívó fájlban deklarálnunk kell a `square` függvényt:

```
double square(double);

main()
{
        printf("%g\n", square(0.3));
}
```

Ennél kissé bonyolultabbak az argumentumok és paraméterek párosítására vonatkozó szabályok. Az ANSI C lehetővé teszi a programozók számára, hogy a függvények bevezetésekor megadják a függvény paramétereinek típusát. Az alábbi deklaráció szerint

```
double square(double);
```

a `square` függvény `double` típusú értéket vár, és `double` típusú értéket ad vissza. Ezután a `square(2)` függvényhívás már megengedett. A `2` egész számértéket a program átalakítja `double` típusra, mintha a programozó azt írta volna, hogy `square((double)2)` vagy `square(2.0)`.

Ha egy függvénynek nincsenek `float`, `short` és `char` típusú paraméterei, akkor a paraméterek típusait teljes egészében elhagyhatjuk a függvény deklarációjából (a meghatározásából viszont nem). Tehát még az ANSI C is megengedi a `square` ilyenforma bevezetését:

```
double square();
```

Ezzel a *hívóra* hárul a felelősség, hogy a megfelelő számú és típusú értéket adja át a függvénynek. A *megfelelő* jelentése nem feltétlen *egyenlő*. A `float` típusú értékekből automatikusan `double`, a `short` és `char` típusú értékekből pedig egész típusú értékek lesznek. Tehát az

```
int
isvowel(char c)
{
```

```
        return c == 'a' || c == 'e' || c == 'i' ||
                c == 'o' || c == 'u';
}
```

függvényt így kell bevezetni minden olyan fájlban, amiben meghívjuk:

```
int isvowel(char);
```

Máskülönben az `isvowel` egész típusúra alakítaná a neki átadott értékeket, ami különbözne a paraméterének típusától. Ha az alábbi módon határoznánk meg az `isvowel` függvényt:

```
int
isvowel(int c)
{
        return c == 'a' || c == 'e' || c == 'i' ||
                c == 'o' || c == 'u';
}
```

a függvényt meghívónak akkor sem kellene deklarálnia a függvényt, ha éppen `char` típusú értéket ad át neki.

Az ANSI C szabvány előtti időkből származó fordítók nem mindegyike támogatja ezt a stílust. Ha ilyen fordítót használunk, elképzelhető, hogy az `isvowel` függvény deklarációjának így,

```
int isvowel();
```

meghatározásának pedig így kell kinéznie:

```
int
isvowel(c)
        char c;
{
        return c == 'a' || c == 'e' || c == 'i' ||
                c == 'o' || c == 'u';
}
```

Hogy a korábbi használatnak megfeleljen, az ANSI C a deklarációknak és a meghatározásoknak ezt a régebbi formáját is támogatja, ami felvet egy problémát: ha egy fájl, ami meghívja az `isvowel` függvényt, nem adhatja meg a paramétereinek típusát (azért, hogy a régebbi fordítókon is működjön), akkor honnan tudja a fordító, hogy a paraméter típusa `char` és nem `int`? A válasz az, hogy a két különböző meghatározási forma két különböző dolgot jelent. Az `isvowel` legutóbbi meghatározása tulajdonképpen ezzel egyenértékű:

```
int
isvowel(int i)
{
    char c = i;
    return c == 'a' || c == 'e' || c == 'i' ||
           c == 'o' || c == 'u';
}
```

Most, hogy már láttunk pár dolgot a függvények meghatározásával és deklarációjával kapcsolatban, nézzük meg, hogyan ronthatjuk el ezeket. Az alábbi egyszerű programocska két okból sem fog működni:

```
main()
{
        double s;
        s = sqrt(2);
        printf("%g\n", s);
}
```

Az első ok az, hogy a `sqrt` egy `double` típusú értéket vár, de ehelyett egész értéket kap. A második ok, hogy `double` típusú értéket ad vissza, pedig a deklarációjában nem ez szerepelt.

Itt láthatjuk az egyik módszert a hibák kijavítására:

```
double sqrt(double)

main()
{
        double s;
        s = sqrt(2);
        printf("%g\n", s);
}
```

Ez pedig egy másik módszer, ami az ANSI előtti fordítókon is működik:

```
double sqrt();

main()
{
        double s;
        s = sqrt(2.0);
        printf("%g\n", s);
}
```

De a legjobb megoldás a következő:

```
#include <math.h>

main()
{
        double s;
        s = sqrt(2.0);
        printf("%g\n", s);
}
```

Ebben a programban semmi konkrétum nincs az `sqrt` függvénynek átadott és az általa visszaadott értékekkel kapcsolatban. Ezt az információt a rendszer `math.h` fejlécállományából gyűjti be a program. Az ANSI fordítókon ezzel még azt is biztosítjuk, hogy a `2.0` a megfelelő típusra lesz átalakítva, bár a példával a régebbi fordítókra is gondoltunk, mivel az átadott érték típusa `double` és nem egész (`int`).

Mivel a `printf` és a `scanf` függvényeknek különböző helyzetekben különböző típusú értékeket adunk át, ezekkel könnyen gondunk támadhat. Álljon itt egy rendkívül látványos példa:

```
#include <stdio.h>

main()
{
        int i;
        char c;
        for (i=0; i<5; i++) {
                scanf("%d", &c);
```

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

Ez a program látszólag beolvas öt számot a szabványos bemenetről, majd a szabványos kimenetre kiírja, hogy

```
0 1 2 3 4
```

Valójában azonban nem mindig ezt teszi. Van olyan fordító, ahol ezt írja ki:

```
0 0 0 0 0 1 2 3 4
```

Miért? A dolog nyitja, hogy a `c` deklarációja szerint `char`, és nem `int` típusú. Amikor arra kérjük a `scanf` függvényt, hogy olvasson be egy egész értéket, akkor az egy olyan mutatót vár, ami egy egész értékre mutat. Ebben az esetben azonban olyan mutatót kap, ami karakterre mutat. A `scanf` függvény persze nem képes arra, hogy felismerje, hogy nem azt kapta, amit várt. A bemenetet egész mutatóként kezeli, és egész értéket tárol az adott helyen. Mivel egy egész több helyet foglal a memóriában, mint egy karakter, ezzel a memória `c` közelében lévő területére lépünk.

Hogy pontosan mi is van a `c` közelében, az a fordító dolga. Ebben az esetben történetesen az `i` alacsony helyiértékű része lesz ott. Ezért minden alkalommal, amikor egy új értéket olvasunk a `c` változóba, azzal lenullázzuk az `i` értékét. Amikor a program végre-valahára eléri a fájl végét, a `scanf` többé már nem próbálkozik azzal, hogy új értéket adjon a `c` változónak, így végre növekedni kezdhet az `i` értéke, és ezáltal a ciklus is véget érhet.

4.5. Külső típusok ellenőrzése

Tegyük fel, hogy van egy két fájlból álló C programunk. Az egyik fájlban az

```
extern int n;
```

deklaráció, a másikban pedig a

```
long n;
```

meghatározás szerepel. Mindkét esetben függvényen kívül történt a bevezetés, tehát azok külső hatókörrel rendelkeznek.

Ez így nem lesz szabályos C program, mert ugyanazt a külső nevet két különböző típussal vezettük be a két fájlban. Ez a hiba azonban számos megvalósításban észrevétlen marad. A fordító mindkét fájlt önállóan kezeli, és az is lehet, hogy hónapok telnek el a két fájl lefordítása között. Miközben tehát a fordító az egyik fájlon dolgozik, semmit sem tud a másik fájl tartalmáról. A linker viszont valószínűleg nem ismeri a C nyelvet, ezért nem képes összehasonlítani az `n` két különböző meghatározásának típusát.

Mi történik akkor, ha futtatjuk ezt a programot? Több lehetőség is van:

1. Az adott megvalósítás elég okos ahhoz, hogy észrevegye a típusok ütközését. Ekkor valószínűleg hibaüzenetet kapunk, ami jelzi, hogy az `n` típusa eltérő a két fájlban.
2. Az adott megvalósításban az `int` és `long` típusú értékek belső ábrázolása megegyezik. Ez általában olyan gépeken fordul elő, amelyeknél a 32 bites aritmetika használata a legtermészetesebb. Ebben az esetben a program úgy fog működni, mintha mindkét helyen `long` (vagy `int`) típust használtunk volna. Ez jó példa arra, amikor egy program helyes működése pusztán a véletlenül műve.
3. Az `n` két példányának tárolásához különböző méretű helyre van szükség, de történetesen úgy osztoznak meg a tárhelyen, hogy az egyik értéke érvényes lesz a másiknál is. Ez úgy történhet meg, ha például a linker úgy osztotta ki a tárhelyet, hogy az `int`

érték a `long` érték alacsony helyiértékű részével osztozik a helyen, és a `long` csak olyan értékeket kap, amelyek elférnek egy `int` méretű helyen is. Ez még az előzőnél is jobb példa arra, amikor egy program helyes működése pusztán a véletlen műve.

4. Az `n` két példánya úgy osztozik a tárhelyen, hogy amikor az egyiknek értéket adunk, akkor annak olyan hatása lesz, mintha a másik példánynak másik értéket adtunk volna. Ebben az esetben a program valószínűleg nem fog működni.

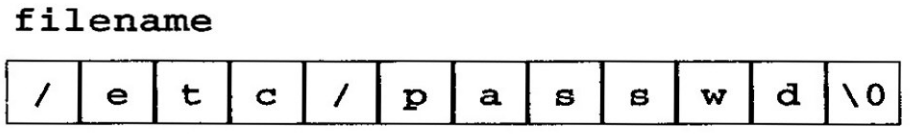
Általában tehát a programozó felel azért, hogy az azonos nevű külső meghatározásoknak minden tárgymodulban ugyanolyan típusa legyen. Az „ugyanolyan típust" nagyon komolyan kell vennünk. Nézzünk meg például egy olyan programot, ahol az egyik fájlban a

```
char filename[] = "/etc/passwd";
```

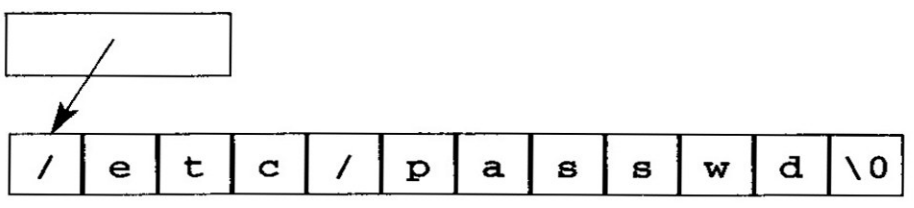
meghatározás, a másik fájlban pedig az

```
extern char *filename;
```

deklaráció szerepel. Bár a tömbök és a mutatók sok környezetben nagyon hasonlítanak egymásra, a *kettő nem ugyanaz.* Az első esetben a `filename` egy karaktertömb neve. Bár egy olyan utasítás, ami a `filename` értékére hivatkozik, egy mutatót kap a tömb első elemére, a `filename` típusa ettől még „karaktertömb" lesz, és nem „karaktermutató". A második deklarációnál ezzel szemben azt állítjuk, hogy a `filename` mutató. A `filename` kétféle bevezetése különbözőképpen használja a tárhelyet. A józan ész pedig kizárja, hogy a két változat megférjen egymással. Míg az első példa így néz ki:



a második így:



A példánkat úgy javíthatjuk ki, ha a `filename` változónak vagy a bevezetését vagy a meghatározását úgy módosítjuk, hogy az megfeleljen a másiknak. Tehát vagy azt írjuk az egyik fájlban, hogy

```
char filename[] = "/etc/passwd";
```

és a másikban azt, hogy

```
extern char filename[];
```

vagy pedig

```
char *filename = "/etc/passwd";
```

kerül az egyik fájlba, és

```
extern char *filename;
```

a másikba.

Úgy is könnyedén bajba kerülhetünk a külső típusokkal, ha hányaveti módon elfelejtjük megadni egy függvény visszatérési értékének típusát, vagy éppen rossz típust adunk meg. Emlékezzünk vissza például a 4.4. részben látott programra:

```
main()
{
        double s;
        s = sqrt(2);
        printf("%g\n", s);
}
```

Ebből a programból hiányzik az `sqrt` függvény bevezetése, tehát a típusát csak a használatából lehet kikövetkeztetni. A C nyelvben van egy szabály, miszerint ha egy még be nem vezetett azonosítót egy nyitó zárójel követ, akkor az egész értéket visszaadó függvényt jelöl. A programunknak tehát éppen az lesz a hatása, mintha ezt írtuk volna:

```
extern int sqrt();

main()
{
        double s;
        s = sqrt(2);
        printf("%g\n", s);
}
```

Ez így persze hibás. Az `sqrt` visszatérési értékének típusa `double`, és nem `int`. Az tehát megjósolhatatlan, hogy mit művel majd a program. Ez annyira így van, hogy néhány gépen akár működhet is! Képzeljünk el például egy olyan gépet, ami ugyanazt a regisztert használja az egész és a lebegőpontos visszatérési értékekhez is. Egy ilyen gép fogja az `sqrt` által visszaadott biteket, és minden további vizsgálat nélkül átadja azokat a `printf` függvénynek. Ha a `printf` a megfelelő biteket kapja, akkor minden bizonnyal a helyes eredményt fogja kiírni. Vannak olyan rendszerek, amelyeken az egészek és a mutatók tárolása különböző regiszterekben történik. Az ilyen gépeken még akkor is hibás lehet a program működése, ha nem használunk lebegőpontos műveleteket.

4.6. Fejlécállományok

A hasonló problémák elkerülése végett érdemes betartani egy egyszerű szabályt: *minden külső objektumot csak egy helyen vezessünk be.* Ez a hely általában egy fejlécállományban lesz, amit minden olyan modulba bele kell foglalni, amelyik használja az adott külső objektumot. Különösen fontos, hogy szerepeljen abban a modulban, amelyik meghatározza az objektumot.

Nézzük meg például még egyszer a `filename` példánkat, ami több modulnak is a része lehet, amelyek mindegyikének ismernie kell egy adott fájl nevét. Az lenne a jó, ha egyetlen helyen módosíthatnánk a fájl nevét úgy, hogy ezután az összes modul az új nevet használhassa. Ezt úgy érhetjük el, ha létrehozunk egy fájlt, aminek legyen a neve mondjuk `file.h`, és ami az alábbi deklarációt tartalmazza:

```
extern char filename[];
```

Minden olyan forrásfájlban, amelynek szüksége van erre a külső objektumra, szerepeljen a következő:

```
#include "file.h"
```

Végül pedig kiválasztjuk az egyik C forrásfájlt, amiben megadjuk a `filename` kezdőértékét. Ennek a fájlnak legyen mondjuk `file.c` a neve:

```
#include "file.h"
char filename[] = "/etc/passwd";
```

Figyeljük meg, hogy a `file.c` állományban tulajdonképpen két deklarációját találjuk a `filename` változónak. Az `include` utasítás végrehajtása után a `file.c` így néz ki:

```
extern char filename[];
char filename[] = "/etc/passwd";
```

Ez így teljesen szabályos, egészen addig, amíg a deklarációk következetesek, és legfeljebb az egyikük lesz meghatározás.

Nézzük meg ennek a hatását. A `filename` típusát a `file.h` állományban vezetjük be, tehát minden olyan modulban automatikusan helyesen fog szerepelni, amelyikbe belefoglaljuk a `file.h` állományt. A meghatározást tartalmazó `file.c` állományba belefoglaltuk a `file.h` állományt, tehát a meghatározás típusa automatikusan meg fog egyezni a bevezetésnél használt típussal. Ha egyszer lefordítjuk az egészet, a típusokkal biztosan nem lesz gondunk!

4.1. gyakorlat. Tegyük fel, hogy egy program egyik állományában a

```
long foo;
```

egy másik állományában pedig az

```
extern short foo;
```

deklaráció található. Azt is tegyük fel, hogy ha egy kis értéket – mondjuk 37-et – adunk a `foo` változó `long` típusú változatának, akkor annak eredményeképpen a `short` típusú változat értéke szintén 37 lesz. Mire következtethetünk ebből a hardvert illetően? Mi a helyzet akkor, ha a `short` változat értéke 0 lesz?

4.2. gyakorlat. Ez itt a 4.4. rész egyik hibás programja kissé leegyszerűsítve:

```
#include <stdio.h>

main()
{
        printf("%g\n", sqrt(2));
}
```

Vannak olyan rendszerek, ahol ez a következőt írja ki:

```
%g
```

Miért?

5

Könyvtári függvények

Minden valamirevaló C programnak szüksége van könyvtári függvényekre, mert a C nyelvben nincsenek bemeneti-kimeneti utasítások. Az ANSI C szabvány erre külön figyelmet fordít, és egy sor olyan szabványos könyvtári függvény meghatározását tartalmazza, amelyeknek minden C megvalósításban szerepelnie kell. A függvények gyűjteménye korántsem teljes. Például gyakorlatilag az összes C megvalósításban megtaláljuk a `read` és `write` függvényeket az alacsonyszintű I/O műveletekhez, bár ezek az ANSI szabványban nem szerepelnek. Másrészt nem minden szabványos függvényt fogunk megtalálni az összes C megvalósításban. Ehhez a könyv írásakor az ANSI C szabvány még túlságosan új volt.

A legtöbb könyvtári függvénnyel általában kevés gondunk akad. Elég egyértelműek ahhoz, hogy a legtöbben helyesen tudják használni őket. Vannak azonban olyan esetek, amikor néhány gyakran használt könyvtári függvény nem egészen úgy működik, ahogy a felhasználóik várnák. A programozóknak különösképpen a `printf` függvénycsaláddal, és a változó paraméterlistával rendelkező függvények létrehozását megkönnyítő `varargs.h` szolgáltatással gyűlik meg a bajuk. A könyv függelékében részletesen bemutatjuk ezt a két szolgáltatást, és az `stdarg.h` szolgáltatást is (ami a `varargs.h` ANSI C változata).

Talán a legjobb tanács, amit a könyvtári függvények használatához adhatunk, hogy *ha lehet, mindig használjuk a rendszer fejlécállományait.* Ha már egyszer a könyvtár szerzője megírt egy olyan fejlécállományt, ami tökéletesen leírja a könyvtár függvényeit, akkor nagy butaság lenne nem használni azt. Ez különösen fontos az ANSI C szabványnál, ahol a fejlécállományok a függvények paramétereinek és visszatérési értékeinek típusait is tartalmazzák. Sőt, van olyan eset is az ANSI C szabványban, ahol csak akkor lehetünk biztosak a helyes eredményben, ha a rendszer fejlécállományát használjuk.

A fejezet további részében olyan gyakran használt könyvtári függvényekkel kapcsolatos problémákat vizsgálunk meg, amelyek a legtöbb gondot okozzák a programozóknak.

5.1. A getchar egész értéket ad vissza

Nézzük meg az alábbi programot:

```
#include <stdio.h>

main()
{
        char c;

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

A `getchar` függvény a szabványos bemeneti állomány következő karakterét adja vissza, vagy ha nincs több adat, akkor az `EOF` értéket (aminek a meghatározását a `stdio.h` tartalmazza, és ami különbözik minden más karaktertől). Úgy tűnhet tehát, hogy a program a szabványos bemenetről érkező adatokat a szabványos kimenetre másolja. Valójában azonban nem egészen ezt teszi.

Ennek oka, hogy a `c` változót karakter típusúként vezettük be, és nem egészként. Ez azt jelenti, hogy a `c` nem képes az összes lehetséges karakter és az `EOF` tárolására.

Tehát két lehetőség van. Vagy az történik, hogy egy szabályos bemeneti karakter hatására a `c` olyan értéket kap, ami a csonkolás után az `EOF` érték lesz, vagy a `c` értéke sosem lesz `EOF`. Az előző esetben a program egyes fájlok másolását valahol félúton abbahagyja, az utóbbi esetben végtelen ciklusba kerül.

Tulajdonképpen van egy harmadik eset is. Egy véletlen folytán a program látszólag működhet is. Bár a `getchar` visszatérési értékét megcsonkítjuk, amikor a `c` változóba tesszük, és bár az összehasonlítást a `c` változóban található csonkolt értékkel és nem a `getchar` eredményével kellene elvégeznie a programnak, meglepő módon léteznek olyan fordítók, amelyekben ennek megvalósítása helytelenül történik. Ezek a fordítók a `getchar` függvény eredményének alacsony helyiértékű bitjeit helyesen teszik a `c` változóba, de ahelyett, hogy ezután a `c` értékét hasonlítanák össze az EOF értékkel, a `getchar` teljes értékét használják az összehasonlításban! Egy ilyen fordítón a fenti mintaprogram látszólag „helyesen" fog működni.

5.2. Soros elérésű állományok módosítása

Sok olyan rendszer létezik, amely megengedi, hogy ugyanazt a fájlt egyszerre írásra és olvasásra is megnyissuk:

```
FILE *fp;
fp = fopen(file, "r+");
```

Ez a kód azzal a szándékkal nyitja meg azt az állományt, amelynek a neve a `file` változóban található, hogy olvasson belőle és írjon is bele.

Azt gondolnánk, hogy ha ez megvan, akkor már kedvünkre váltogathatjuk az írási és olvasási műveleteket. Sajnos ez nem így van, köszönhetően azoknak az erőfeszítéseknek, amelyekkel megpróbálják fenntartani az együttműködési képességet azokkal a programokkal, amelyek ennek a lehetőségnek a megjelenése előtt készültek. Ezért egy olvasási művelet soha nem követhet közvetlenül egy írási műveletet és fordítva, anélkül hogy előbb meghívnánk az `fseek` függvényt.

A következő programrészlet látszólag néhány előre kiválasztott rekordot módosít egy soros elérésű (szekvenciális) állományban:

```
FILE *fp;
struct record rec;
. . .
while (fread((char *) &rec, sizeof(rec), 1,
                fp) == 1) {
        rec módosítása
        if (a rec értékét át kell írni) {
                fseek(fp, -(long)sizeof(rec), 1);
                fwrite((char *)&rec, sizeof(rec),
                  1, fp);
        }
}
```

Elsőre úgy tűnik minden rendben. A `&rec` típusát figyelmesen átalakítjuk, hogy `char *` legyen, mielőtt átadnánk az `fread` és az `fwrite` függvényeknek. A `sizeof(rec)` értékét pedig `long` típusúra alakítjuk (az `fseek` második paraméterként egy `long` értéket vár, mert egy `int` érték nem biztos, hogy elég nagy ahhoz, hogy elférjen benne a fájlméret; a `sizeof` egy előjel nélküli értéket ad vissza, ezért csak úgy lehet az ellentettjét venni, ha előbb előjelessé alakítjuk). A program azonban hibázik, és az is lehet, hogy ezt egész észrevétlenül teszi.

A gond ott van, hogy amikor egy rekordot átírunk (vagyis az `fwrite` függvény végrehajtásakor), a következő dolog, ami a fájllal történik, a ciklus elején található `fread` lesz. Ez így nem jó, hiszen közben nincs ott a szükséges `fseek`. Ezt úgy oldhatjuk meg, ha a következőképpen írjuk át a programot:

```
while (fread((char *) &rec, sizeof(rec), 1,
                fp) == 1) {
        rec módosítása
        if (a rec értékét át kell írni) {
                fseek(fp, -(long)sizeof (rec), 1);
                fwrite((char *)&rec, sizeof(rec),
                  1, fp);
                fseek(fp, 0L, 1);
        }
}
```

A második `fseek` látszólag semmit nem csinál, de valójában olyan állapotba hozza a fájlt, hogy így már sikeresen olvashatunk belőle.

5.3. Késleltetett kiírás és a memória lefoglalása

Amikor egy programnak valamilyen kimenete van, mennyire fontos, hogy a felhasználó azonnal lássa is ezt a kimenetet? Programja válogatja.

Ha a kimenet például egy terminálra érkezik, és a terminál mögött ülő embertől várunk választ egy kérdésre, akkor rendkívül fontos, hogy a felhasználó lássa a kimenetet, hogy tudja, mit kell begépelnie. Másrészt viszont, ha a kimenet célja egy fájl, majd pedig egy sornyomtató, akkor csak az számít, hogy valahogy eljusson odáig.

Gyakran költségesebb egyből megjeleníteni valamit, mint elraktározni és később nagyobb adagban kiírni. A C megvalósítások ezért általában megengedik a programozónak, hogy eldöntse, mennyi kimenet gyűljön össze, mielőtt a tényleges kiírás megtörténne.

A döntés hatalmával általában a `setbuf` könyvtári függvény ruházza fel a programozót. Amennyiben a `buf` egy megfelelő méretű karaktertömb, akkor a

```
setbuf(stdout, buf);
```

utasítás azt jelenti az I/O könyvtár számára, hogy az `stdout` kimenetre érkező adatokat innentől kezdve a `buf` kimeneti tár közbeiktatásával kell kiírni, vagyis a tényleges kiírás csak akkor történik meg, amikor a `buf` megtelik, vagy ha a programozó kimondottan kéri azt az `fflush` függvény meghívásával. A tár megfelelő méretét a `BUFSIZ` tartalmazza, amelynek meghatározását az `<stdio.h>` állományban találjuk.

A következő program tehát azt szemlélteti, hogyan lehet a legkézenfekvőbb módon átmásolni a szabványos bemenetre érkező adatokat a szabványos kimenetre:

```
#include <stdio.h>

main()
{
        int c;

        char buf[BUFSIZ];
        setbuf(stdout, buf);

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

Egy mákszemnyi hiba miatt azonban ez a program így helytelen. A `setbuf` hívásával azt kérjük az I/O könyvtártól, hogy átmenetileg a `buf` tárba kerüljenek a szabványos kimenetre küldött karakterek. Hogy lássuk, hol a gond, tegyük fel azt a kérdést, hogy mikor ürítjük utoljára a `buf` átmeneti tárat. A válasz az, hogy a főprogram befejezése után, annak a tisztogatási műveletnek a részeként, amit a könyvtár végez el, mielőtt visszaadná a vezérlést az operációs rendszernek. Addigra viszont a `buf` változóhoz tartozó helyet már felszabadítottuk a memóriában!

Az efféle problémák megakadályozásának két módja van. Az egyik, hogy az átmeneti tárból statikus változót készítünk, vagy úgy, hogy eleve statikusként deklaráljuk,

```
static char buf[BUFSIZ];
```

vagy úgy, hogy teljesen a főprogramon kívülre visszük a bevezetését. Egy másik lehetőség, ha dinamikusan foglalunk le memóriát a tár számára, amit soha nem szabadítunk fel:

```
char *malloc();
setbuf(stdout, malloc(BUFSIZ));
```

Aki vonzódik a felületes programozási megoldásokhoz, az észreveheti, hogy itt szükségtelen ellenőriznünk, hogy sikerrel járt-e a `malloc` függvény. Ha nem, akkor egy cím nélküli (null) mutatót kapunk vissza. Ezt a `setbuf` elfogadja második paraméterként, és hatására az adatok ki-

írása a stdout kimenetre késleltetés nélkül megy végbe. A program ettől lassabban fog működni, de legalább működni fog.

5.4. Az errno használata a hibák azonosítására

A könyvtári függvények között vannak olyanok, különösen az operációs rendszerrel foglalkozók, amelyek hibajelzést adnak vissza egy errno nevű külső változóban, ha hívásuk sikertelen volt. A legkézenfekvőbb megoldás ennek kihasználására sajnos helytelen:

```
meghívjuk a könyvtári függvényt
if (errno)
        panaszkodunk
```

A gond az, hogy azoknak a könyvtári függvényeknek, amelyek az errno értékét használják a hibák jelzésére, nem kell lenullázniuk az értékét abban az esetben, ha nem történt hiba. Látszólag tehát az alábbi megoldásnak működnie kell, de még ez is helytelen:

```
errno = 0;
meghívjuk a könyvtári függvényt
if (errno)
        panaszkodunk
```

Bár hiba hiányában nem kötelező a könyvtári függvényeknek lenullázni az errno értékét, az sincs megtiltva, hogy valamilyen értéket adjanak neki. Hogy lássuk mi értelme ennek, képzeljük el, mi történhet az fopen belsejében. Ha írásra nyitunk meg egy fájlt, akkor az fopen törli a fájlt, ha már létezett, majd megnyitja. Ehhez lehet, hogy meghív egy másik könyvtári függvényt, amivel ellenőrzi, hogy létezik-e már a fájl.

Tegyük fel, hogy ez a könyvtári függvény valamilyen értéket ad az errno változónak, ha a fájl még nem létezik. Ekkor minden alkalommal, amikor az fopen olyan fájlt nyit meg, ami még nem létezett, mellékhatásként akkor is értéket ad az errno változónak, ha hiba valójában nem is történt.

Tehát amikor egy könyvtári függvényt hívunk meg, létfontosságú, hogy az esetleges hibák ellenőrzésekor *előbb* vizsgáljuk meg a függvény által visszaadott értéket, és csak azután próbáljuk megállapítani a hiba okát az `errno` értékének segítségével:

```
meghívjuk a könyvtári függvényt
if (hibát kapunk)
        errno vizsgálata
```

5.5. A signal függvény

Jóformán minden C megvalósítás tartalmazza a `signal` függvényt, amit az aszinkron események kezelésére szántak. Ha használni akarjuk, akkor a

```
#include <signal.h>
```

sorral emelhetjük be a megfelelő deklarációkat. Egy adott jel (esemény) kezeléséhez a

```
signal(jelzés_típusa, kezelőfüggvény);
```

formát kell követnünk, ahol a *jelzés_típusa* az elfogandó esemény típusát azonosító állandó lesz, amelynek meghatározása a `signal.h` állományban található, a *kezelőfüggvény* pedig az a függvény, amit akkor kell meghívni, ha bekövetkezik az adott esemény.

Az események sok megvalósításban ténylegesen aszinkron módon következnek be, ami azt jelenti, hogy egy esemény szó szerint bármikor bekövetkezhet a C program végrehajtása közben. Különösen fontos hangsulyozni, hogy olyan bonyolult könyvtári függvények végrehatása közben is bekövetkezhetnek, mint amilyen a `malloc`. Ezért tehát az eseménykezelők számára az ilyen könyvtári függvények meghívása nem biztonságos.

Tegyük fel például, hogy a `malloc` futását megszakítja egy esemény. Ekkor nagyon valószínű, hogy azoknak az adatoknak a frissítése csak részben történik meg, amelyekkel a `malloc` nyomon követi a memória szabad területeit. Ha az eseményt kezelő függvény ismét meghívja a `malloc` függvényt, akkor az a `malloc` adatszerkezeteinek teljes felborításával, és az ezt követő súlyos fejfájással járhat.

Hasonló okokból általában nem tanácsos a `longjmp` függvénnyel kilépni egy eseménykezelő függvényből. Elképzelhető ugyanis, hogy az esemény bekövetkezésekor a `malloc` vagy egy másik könyvtári függvény éppen egy adatszerkezet módosításával volt elfoglalva, amivel még nem végzett. Úgy tűnik tehát, hogy ha biztosra akarunk menni, akkor az egyetlen, amit az eseménykezelő tehet, hogy beállít egy jelzőértéket és kilép, abban a reményben, hogy a főprogram ellenőrzi azt, és így tudomást szerez az eseményről.

De még ez sem teljesen biztonságos. Néhány gép újra elvégzi a hibás műveletet az eseménykezelő visszatérése után, ha olyan aritmetikai hiba bekövetkeztéről van szó, mint a túlcsordulás vagy a nullával történő osztás. A művelet tényezőinek megváltoztatására az újbóli próbálkozásnál nincs hordozható módszer. A legvalószínűbb eredmény ebben az esetben az esemény ismételt kiváltása lesz. Tehát az egyetlen hordozható és nagyjából biztonságos dolog, amit egy eseménykezelő egy aritmetikai hibával kapcsolatban tehet, hogy kiír egy hibaüzenetet és kilép (a `longjmp` vagy az `exit` segítségével).

Ebből az a tanulság, hogy az eseményekkel vigyáznunk kell, és hogy olyan belső tulajdonságaik is lehetnek, amelyek nem hordozhatók. A legjobb védekezés ezekkel a problémákkal szemben, ha az eseménykezelőket a lehető legegyszerűbbre írjuk meg, és egy csoportba tesszük őket. Így könnyebben módosíthatók, ha a szükség úgy hozza, hogy egy új rendszer követelményeinek kell megfelelniük.

5.1. gyakorlat. Ha egy program valamilyen hiba folytán ér véget, akkor a kimenetének néhány utolsó sora elveszhet. Miért? Mit lehet ez ellen tenni?

5.2. gyakorlat. Az alábbi program a bemenetét a kimenetére másolja:

```
#includeo <stdio.h>

main()
{
        register int c;
```

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

Ha eltávolítjuk az `#include` utasítást a programból, akkor nem tudjuk lefordítani, mert az `EOF` értéke meghatározatlan lesz. Nem szép dolog ilyet tenni, de tegyük fel, hogy „kézzel" megadjuk `EOF` értékét:

```
#define EOF -1

main()
{
        register int c;

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

Ez a program számos rendszeren működni fog, de lesznek olyanok is, ahol csak nagyon lassan. Miért?

6

Az előfeldolgozó

A programokat nem abban a formában futtatjuk, ahogy megírtuk azokat. A C előfeldolgozó először átalakítja őket. Az előfeldolgozó segítségével olyan dolgokat rövidíthetünk le, amelyek két fő (és több kevésbé fontos) okból is lényegesek.

Először is, előfordulhat, hogy egy mennyiség, például egy tábla méretének összes előfordulását meg akarjuk változtatni úgy, hogy csak egyetlen helyen kelljen elvégeznünk a módosítást, aztán újrafordíthassuk a programot. Az előfeldolgozóval mindez pofonegyszerű, még akkor is, ha az adott mennyiség a program számos helyén előfordul. Mindössze annyit kell tennünk, hogy a mennyiséget valahol *szimbolikus állandóként* (manifest constant) határozzuk meg, és már használhatjuk is a megfelelő helyeken. Az előfeldolgozó használata ezen kívül azt is lehetővé teszi, hogy egy helyre gyűjtsük össze az állandók meghatározásait, amiket ezután könnyedén megtalálhatunk.

Másodszor, a legtöbb C megvalósításban jelentős többletterhet jelent a program számára minden egyes függvényhívás. Ezért előfordul, hogy valami olyan, függvényhez hasonló dolgot szeretnénk meghatározni, aminél a függvényhívással kapcsolatos teher nem jelentkezik. A `getchar` és `putchar` megvalósítása például általában makróként történik, hogy ne kelljen minden bejövő és kimenő karakterhez meghívni egy függvényt.

Amilyen hasznosak a makrók, éppen annyira össze is keverhetik azt a programozót, aki nem ismeri fel, hogy *a makrók a program szövegén végeznek műveleteket.* Vagyis a makrók segítségével a C programot alkotó karaktereket módosíthatjuk, a programban található objektumokat viszont nem. A makrókkal tehát egy nyelvtanilag látszólag helytelen valamiből szabályos C programot készíthetünk, de az is megeshet, hogy egy ártatlan kis semmiségből szörnyet varázsolunk.

6.1. A makrók meghatározásában számítanak a szóközök

A paraméterek nélküli függvényeket úgy hívjuk meg, hogy a nevük után egy zárójelpárt teszünk. A paraméter nélküli makrókat ezzel szemben a nevük leírásával használjuk, a zárójelek nem kellenek. Ha a makró meghatározása megtörtént, akkor ez teljesen rendben van. Az előfeldolgozó a makró meghatározásából tudja, hogy számítania kell-e paraméterekre vagy sem.

A makrókat kicsit nehezebb meghatározni, mint meghívni. Az `f` alábbi meghatározásában például

```
#define f (x) ((x)-1)
```

kell paraméter vagy sem? Mindkét válasz elképzelhető. Lehet, hogy az `f(x)` jelentése ez:

```
((x) -1)
```

de az is lehet, hogy ez:

```
(x) ((x)-1)
```

Ebben az esetben az utóbbi válasz a helyes, hiszen egy szóköz áll az `f` és az őt követő `(` jel között. Ha tehát az `f(x)` meghatározásán az `((x)-1)` kifejezést értjük, akkor azt kell írnunk, hogy

```
#define f(x) ((x)-1)
```

Ez a szabály csak a makrók *meghatározására* érvényes, a makrók *hívására* nem. Tehát a fenti utolsó meghatározást követően, az `f(3)` és az `f (3)` értéke is 2 lesz.

6.2. A makrók nem függvények

Mivel a makrók kinézetre majdnem olyanok, mint a függvények, a programozók néha engednek a csábításnak és egy kalap alá veszik őket. Az ember így néha ilyenekkel találkozik:

```
#define abs(x)  (((x)>=0)?(x):-(x))
```

vagy ilyenekkel

```
#define max(a,b)  ((a)>(b)?(a):(b))
```

Figyeljük meg a makrók törzsében található zárójeleket, amelyek a műveletsorrenddel kapcsolatos hibáktól védenek. Tegyük fel például, hogy az `abs` függvényt az alábbi módon határoztuk meg:

```
#define abs(x) x>0?x:-x
```

majd képzeljük el, hogy milyen eredményt kapunk az `abs(a-b)` kifejezés kiértékelésekor. Az

```
abs(a-b)
```

kifejezés kibontva így néz ki

```
a-b>0?a-b:-a-b
```

ami helytelen megoldáshoz vezet. A `-a-b` részkifejezés a `(-a)-b` kifejezéssel egyenértékű, és nem a `-(a-b)` kifejezéssel, ahogy azt eredetileg terveztük. Ezen oknál fogva nem árt minden paramétert zárójelek közé tenni. Az is fontos, hogy az egész kifejezést is zárójelek közé tegyük, mert ezzel megelőzhetjük a makró felhasználását egy nagyobb kifejezésben.

Máskülönben az

```
abs(a)+1
```

kibontva ez lesz:

```
a>0?a:-a+1
```

ami nyilvánvalóan hibás. Ha az `abs` meghatározása helyes,

```
#define abs(x) ((x)>0?(x):-(x))
```

akkor ennek eredményeképp az

```
abs(a-b)
```

kibontva helyesen

```
((a-b)>0?(a-b) :-(a-b))
```

az

```
abs(a)+1
```

pedig helyesen kibontva

```
((a)>0?(a):-(a))+1
```

Egy kétszer felhasznált tényező kiértékelése akkor is kétszer mehet végbe, ha mindenhol használtunk zárójeleket a makró meghatározásában. A `max(a,b)` kifejezésben tehát, ha `a` nagyobb, mint `b`, akkor az `a` kiértékelése kétszer történik meg. Először az összehasonlításkor, másodszor pedig a `max` függvény visszatérési értékének meghatározásakor. Ez nem csak a hatékonyságot ronthatja el, hanem a programot is:

```
biggest = x[0];
i = 1;
while (i < n)
        biggest = max(biggest, x[i++]);
```

Ezzel nem lenne semmi gond, ha a max igazi függvény lenne, de nem fog működni, mert a max most egy makró. Hogy lássuk is ezt, tegyünk néhány kezdőértéket az x tömbbe:

```
x[0] = 2;
x[1] = 3;
x[2] = 1;
```

Nézzük meg mi történik a ciklus első végrehajtásakor. Az értékadás művelet kibontva így néz ki:

```
biggest = ((biggest)>(x[i++])?(biggest):(x[i++]));
```

Először a biggest változó értékét összehasonlítjuk az x[i++] értékével. Mivel i értéke 1, az x[1] értéke pedig 3, az összehasonlítás eredménye hamis lesz. Az i értéke eközben mellékhatásként 2-re változik.

Mivel az összehasonlítás eredménye hamis, a biggest változóhoz az x[i++] elem értékét rendeljük. Az i értéke azonban most már 2, tehát a biggest változónak az x[2] értékét adjuk, ami 1. Az i értéke most már 3.

Az egyik mód arra, hogy elkerüljük ezeket a problémákat, ha gondoskodunk róla, hogy a max makró paramétereinek ne legyenek mellékhatásai:

```
biggest = x[0];
for (i = 1; i < n; i++)
        biggest = max(biggest, x[i]);
```

Egy másik megoldás az, ha a max-ot függvénnyé tesszük, vagy ha magunk végezzük el a számításokat:

```
biggest = x[0];
for (i = 1; i < n; i++)
        if (x[i] > biggest)
                biggest = x[i];
```

Nézzünk egy másik példát a makrók és a mellékhatások keverésével járó kockázatokra. Ez itt a `putc` makró szokásos meghatározása:

```
#define putc(x,p) \
(--(p)->_cnt>=0?(*(p)->_ptr++=(x)):_flsbuf(x,p))
```

A `putc` első paramétere egy karakter, amit kiír egy fájlba. A második paraméter egy mutató, ami egy külső adatszerkezetre mutat, ami ennek a fájlnak a leírását tartalmazza. Figyeljük meg, hogy az első, `x` paramétert, ami akár egy `*z++` értéket is kaphat, csak egyszer értékeljük ki, annak ellenére, hogy két külön helyen is megtalálható a makró törzsében. Ez a két hely nem más, mint a `:` műveleti jel két oldala.

Ezzel szemben a második, `p` paramétert, ami azt a fájlt jelöli, ahová írunk, mindig kétszer értékeljük ki. Mivel a `putc` fájlparaméterének szinte soha nincsenek mellékhatásai, ez ritkán okoz gondot. Ennek ellenére az ANSI szabvány figyelmeztet arra, hogy előfordulhat, hogy a `putc` kétszer is kiértékeli második paraméterét. Más C megvalósítások kevésbé óvatosak, és lehetőséget adnak egy olyan `putc` megvalósítására, ami az *első* paraméterét is egynél többször értékeli ki. Ha a `putc` számára olyan értéket adunk át, aminek mellékhatásai vannak, akkor nem árt, ha tisztában vagyunk ezekkel a gyanútlan megvalósításokkal.

Egy másik példaként vegyük a `toupper` függvényt, amit sok C könyvtárban megtalálhatunk. Ez a kisbetűket alakítja át nagybetűkké, a többi karaktert pedig érintetlenül hagyja. Ha feltételezzük, hogy az összes kisbetű, illetve az összes nagybetű egymás után következik a gép jelsorrendjében (a kis- és nagybetűk között esetleg „hézag" lehet), akkor a következő függvényt kapjuk:

```
toupper(int c)
{
        if (c >= 'a' && c <= 'z')
                c += 'A' - 'a';
        return c;
}
```

A legtöbb C megvalósításban az eljáráshívással járó többletmunka sokkal nagyobb, mint maga az eredmény kiszámítása, így a megvalósítást végző személy rögtön arra gondol, hogy makrót készít:

```
#define toupper(c) \
      ((c)>='a' && (c)<='z'? (c)+('A'-'a'): (c))
```

Sok esetben ez tényleg sokkal gyorsabb, mint a függvény, de komoly meglepetésben lesz része annak, aki a `toupper(*p++)` kifejezést próbálja használni.

A makrók használatának egy másik kockázatos oldala, hogy előfordulhat, hogy rendkívül terjedelmes kifejezéseket hoznak létre, és ezzel több helyet foglalnak el, mint ahogy azt a felhasználójuk szerette volna. Nézzük meg például még egyszer a `max` meghatározását:

```
#define max(a,b)  ((a)>(b)?(a):(b))
```

Tegyük fel, hogy ezt a meghatározást használva szeretnénk megtalálni `a`, `b`, `c` és `d` közül a legnagyobbat. Ha az alábbi kézenfekvő megoldást választjuk:

```
max(a,max(b,max(c,d)))
```

akkor az kibontva így fog kinézni,

```
((a)>(((b)>(((c)>(d)?(c):(d)))?(b):(((c)>(d)?(c):(d)))))?
 (a):(((b)>(((c)>(d)?(c):(d)))?(b):(((c)>(d)?(c):(d))))))
```

ami meglepően hosszú. A tényezők jobb elosztásával némileg csökkenthetjük a méretét:

```
max(max(a,b),max(c,d))
```

ami kibontva annyi, mint:

```
(((((a)>(b)?(a):(b)))>(((c)>(d)?(c):(d)))?
 (((a)>(b)?(a):(b))):(((c)>(d)?(c):(d))))
```

Az embernek valahogy mégis az az érzése, hogy egyszerűbb azt írni, hogy:

```
biggest = a;
if (biggest < b) biggest = b;
if (biggest < c) biggest = c;
if (biggest < d) biggest = d;
```

6.3. A makrók nem utasítások

Bár csábító a gondolat, meglepően nehéz olyan makrót írni, ami utasításként működik. Vegyük például az `assert` makrót. Paraméterként egy kifejezést kap, és ha ennek az értéke nulla, akkor a megfelelő hibaüzenettel befejezi a program végrehajtását. Makróról lévén szó, lehetőségünk van arra, hogy a hibaüzenet tartalmazza a hibát kiváltó eljárást tartalmazó fájl nevét és a sor számát. Más szóval, az

```
assert(x>y);
```

ne csináljon semmit, ha az `x` nagyobb, mint `y`, egyébként pedig állítsa le a programot.

Íme, az első kísérletünk a makró megírására:

```
#define assert(e) if (!e)
        assert_error(__FILE__,__LINE__)
```

Az `assert` felhasználójának kell majd a pontosvesszőt kitennie, így az a meghatározásban nem szerepel. A `__FILE__` és `__LINE__` makrókat beépítették a C előfeldolgozóba. Kibontva annak a fájlnak a nevét és annak a sornak a számát jelenítik meg, amelyikben használtuk őket. Ez a meghatározás egy alig észrevehető hibát tartalmaz, ami egy egészen egyszerű esetben rögtön kiderül:

```
if (x > 0 && y > 0)
        assert (x > y);
else
        assert (y > x);
```

Ez így teljesen logikus, de kibontva ez lesz belőle:

```
if (x > 0 && y > 0)
      if (!(x > y)) assert_error("foo.c", 37);
else
      if (!(y > x)) assert_error("foo.c", 39);
```

Ha úgy módosítjuk a behúzásokat, hogy azok megmutassák a tényleges (és nem a szándékolt) szerkezetet, akkor ezt kapjuk:

```
if (x > 0 && y > 0)
      if (!(x > y))
            assert_error("foo.c", 37);
      else
            if (!(y > x))
                  assert_error("foo.c", 39);
```

Ezt a problémát elkerülhetjük, ha az `assert` makró törzsét kapcsos zárójelek közé tesszük:

```
#define assert(e) \
 { if (!e) assert_error(__FILE__, __LINE__); }
```

Ez viszont felvet egy új problémát. A példánk kibontva most így néz ki:

```
if (x > 0 && y > 0)
  { if (!(x > y)) assert_error("foo.c", 37); };
else
  { if (!(y > x)) assert_error("foo.c", 39); };
```

az `else` előtti pontosvessző pedig nyelvtani hibához vezet. Ezt megoldhatjuk úgy, hogy megköveteljük, hogy az `assert` meghívását soha ne kövesse pontosvessző, de ez nagyon furcsán fog kinézni:

```
y = distance(p, q);
assert(y > 0)
x = sqrt(y);
```

Az `assert` meghatározásának helyes módja közel sem nyilvánvaló. Úgy kell elkészítenünk az `assert` törzsét, hogy az egy kifejezésre és ne egy utasításra hasonlítson:

```
#define assert(e) \
        ((void)((e)||_assert_error(__FILE__,
                                   __LINE__)))
```

Ez a meghatározás a `||` művelet sorrendiségén alapszik. Ha `e` igaz, akkor a

```
(void)((e)||_assert_error(__FILE__, __LINE__))
```

értékéről anélkül is tudjuk, hogy igaz, hogy ezt itt kiértékelnénk:

```
_assert_error(__FILE__, __LINE__)
```

Ha `e` hamis, akkor az

```
_assert_error(__FILE__, __LINE__)
```

értékét meg kell határozni. Az `assert_error` meghívása kiírja a megfelelő „hiba az eljárásban" (assertion failed) üzenetet.

6.4. A makrók nem típusmeghatározások

Gyakran arra használják a makrókat, hogy különböző változók típusát egy helyen adják meg:

```
#define FOOTYPE struct foo
FOOTYPE a;
FOOTYPE b, c;
```

Ennek segítségével a programozónak elég egyetlen programsort átírnia ahhoz, hogy megváltoztassa az `a`, `b` és `c` típusát, még akkor is, ha az `a`, `b` és `c` bevezetése teljesen más helyen található.

Egy makrómeghatározás használatának ebben az esetben a hordozhatóság lesz a legnagyobb előnye, hiszen minden C fordító támogatja. Ennek ellenére a típusmeghatározás használata a jobb megoldás:

```
typedef struct foo FOOTYPE;
```

A fenti sor egy új, `FOOTYPE` nevű típust határoz meg, ami megegyezik a `struct foo` típussal.

A típusok elnevezésének ez a két módja azonosnak tűnhet, de a `typedef` sokkal általánosabb. Vegyük például a következőt:

```
#define T1 struct foo *
typedef struct foo *T2;
```

Ezen meghatározások alapján a `T1` és a `T2` elvben `struct foo` típusú mutató lesz. De nézzük, mi történik, ha egynél több változóval próbáljuk használni őket:

```
T1 a, b;
T2 c, d;
```

Az első bevezetés kibontva így néz ki:

```
struct foo * a, b;
```

E meghatározás szerint `a` struktúramutató lesz, `b` viszont struktúra (tehát nem mutató). A második bevezetésnél ezzel szemben a `c` és a `d` is struktúramutató lesz, mivel a `T2` igazi típusként viselkedik.

6.1. gyakorlat. Írjuk meg a `max` makróváltozatát úgy, hogy egész paraméterei legyenek, és a makró csak egyszer értékelje ki azokat.

6.2. gyakorlat. Lehet-e a 6.1. részben említett

```
(x)  ((x)-1)
```

„kifejezés" érvényes C kifejezés?

7

Hordozhatósággal kapcsolatos buktatók

A C nyelv megvalósítását már sokan sokféle gépre elkészítették. Éppen ez az egyik fő ok, amiért sokan C nyelven írják meg a programjaikat, hiszen azokat így könnyen át lehet vinni az egyik fejlesztői környezetből a másikba.

A sok megvalósítás azonban azt is jelenti, hogy nem pontosan ugyanazt kapjuk a különböző rendszereken. Már az első két C fordító is jelentősen eltért egymástól. A különféle rendszerek ezen felül különböző követelményeket is támasztanak, tehát joggal számíthatunk arra, hogy a különféle gépeken található C megvalósítások között apróbb eltérések lesznek. Az ANSI szabvány elterjedése nagy segítség, de csodát ne várjunk tőle.

Mivel a korai C megvalósításoknak egy közös ősük volt, ezekben nagyjából ez az örökség formálta a C könyvtárat. Amikor aztán elkezdték különféle operációs rendszereken megvalósítani a C nyelvet, megpróbálták úgy formálni a könyvtárat, hogy az ismerős legyen a korai megvalósításokhoz szokott programozók számára is.

Ezek a próbálkozások nem mindig jártak sikerrel. Sőt, ahogy azt várni lehetett, amint a világ különböző tájain egyre többen kezdtek dolgozni különféle C megvalósításokon, néhány könyvtári függvény

viselkedése egyre jobban eltért a különféle megoldásokban. Ha manapság egy C programozó olyan programokat akar írni, amelyek több környezetben is használhatók, akkor tisztában kell lennie ezekkel a sokszor igen finom különbségekkel.

A hordozhatóság éppen ezért rendkívül terjedelmes témakör. Általános tárgyalása messze túlmutat ennek a könyvnek a keretein. Mark Horton *How to Write Portable Software in C* (Hogyan írjunk hordozható programokat C nyelven, Prentice-Hall) című könyve részletesen foglalkozik ezzel a kérdéssel. Ebben a fejezetben mindössze a leggyakoribb hibaforrások közül vizsgálunk meg néhányat, különös tekintettel a nyelv (és nem a könyvtár) jellemzőire.

7.1. Alkalmazkodás a változásokhoz

Amint ezt írom, az ANSI bizottság éppen az utolsó simításokat végzi a legújabb C szabványon. Ez a szabvány számos olyan nyelvi fogalmat tartalmaz, amelyek még egyáltalán nem általánosak a C fordítókban. Sőt, annak ellenére, hogy a C fordítók készítői várhatóan alkalmazkodnak az új szabványhoz, egyáltalán nem biztos, hogy a C nyelven programozók azonnal frissítik is a fordítóprogramjaikat. Az új fordítók beszerzése pénzbe kerül, a telepítésük pedig időt vesz igénybe. És egyáltalán minek cserélnének le egy olyan fordítót, ami működik?

Ezek a változások komoly dilemma elé állítják a C programok szerzőit. Használja-e a program az új elemeket vagy sem? Ha a válasz igen, azzal könnyebbé válik a programozás, és csökken a hibák lehetősége is, viszont a régebbi megvalósításokon használhatatlan lesz az eredmény.

A 4.4. részben láthattunk is erre egy példát, a függvények prototípusával kapcsolatban. Emlékezzünk vissza a `square` függvényre abból a részből:

```
double square(double x)
{
        return x * x;
}
```

A fenti módon megírva nem sok C fordító tudja majd lefordítani. Ha átírjuk a régebbi stílust használva, azzal hordozhatóbbá tesszük, mert az ANSI szabvány a régi forma használatát is engedélyezi:

```
double
square(x)
        double x;
{
        return x * x;
}
```

A hordozhatóságnak azonban ára van. Ha következetesek vagyunk a régebbi használatot illetően, akkor a függvényt a hívó programban így kell deklarálnunk:

```
double square();
```

A paraméter típusának ilyetén elhagyása az ANSI C szabványban is megengedett. Emlékezzünk vissza, hogy egy ilyen bevezetés semmit nem árul el a paraméterek típusairól. Ez azt jelenti, hogy a nem megfelelő paraméter használata szép csendben hibához vezet:

```
double square();

main()
{
        printf("%g\n", square(3));
}
```

Mivel a `square` deklarációja semmit nem árul el a paraméterek típusáról, a `main` fordításakor lehetetlen megállapítani, hogy a `square` függvény paraméterének `double` típusúnak, és nem `int`-nek kellene lennie. A program így viszont csak értelmetlen adatokat ír ki.
Az ilyen problémákra a 4.0 részben már említett `lint` program használatával deríthetünk fényt, ha az rendelkezésünkre áll. Ha a programot így írtuk volna meg:

```
double square(double);

main()
{
        printf("%g\n", square(3));
}
```

akkor a 3 automatikusan átalakult volna `int` típusúról `double` típusúra. Egy másik megoldás, ha a program kimondottan `double` típusú értéket ad át a függvénynek:

```
double square();

main()
{
        printf("%g\n", square(3.0));
}
```

Ez így is működőképes. Az utóbbi stílus ráadásul az olyan régebbi fordítókon is működik, amelyek nem teszik lehetővé, hogy a függvények meghatározása tartalmazza a paraméterek típusait.

A hordozhatósággal kapcsolatos döntéseink sokszor nagyon hasonlóak a fentiekhez. Használja-e a programozó az újabb vagy kifinomultabb szolgáltatást vagy sem? Ha a válasz igen, az a kényelem szempontjából számos előnnyel járhat, de ezért azt az árat kell fizetnünk, hogy szűkítjük a program lehetséges felhasználóinak táborát.

Ezekre a kérdésekre nem létezik egyszerű válasz. A programok általában tovább élnek, mint ahogy azt a készítőik valaha is álmodni merték volna, még akkor is, ha a szerző csak saját felhasználásra készítette őket. Ezért nem tehetjük meg, hogy programírás közben csak a jelennel foglalkozunk, és figyelmen kívül hagyjuk a jövőt. Viszont éppen az imént láttuk, hogy a hordozhatóság elérése sokba kerülhet, hiszen a ma előnyeit kell feláldoznunk azért, hogy a tegnap eszközeit használhassuk. A legtöbb, amit tehetünk ez ügyben, hogy felismerjük, hogy *nekünk kell meghoznunk* ezeket a döntéseket, és nem hagyjuk, hogy a véletlen hozza meg azokat helyettünk.

7.2. Mit rejt egy név?

Vannak olyan C megvalósítások, amelyekben az azonosítók összes karakterének jelentősége van. Vannak viszont olyanok is, amelyek szép csendben lenyisszantják a hosszabb azonosítók végét.

A linkereknek is lehetnek saját követelményei a neveket illetően. Előfordulhat például, hogy csak csupa nagybetűből álló külső azonosítókat fogadnak el. Ha egy ilyen megkötéssel találkozik valahol az adott rendszeren a megvalósítás készítője, akkor jogosan követelheti meg a külső nevek csupa nagybetűvel való írását. Az ANSI C egyébként mindössze annyit garantál, hogy a megvalósításai különbséget tesznek azon külső nevek között, amelyeknél a név első hat karaktere eltérő. Ennek a meghatározásnak a szempontjából, azonban a kis- és nagybetűk nem számítanak eltérőnek.

Emiatt nagyon óvatosnak kell lennünk a külső azonosítók kiválasztásával az olyan programoknál, amelyeknél fontos a hordozhatóság. Nem szerencsés például, ha van mondjuk egy `print_fields` és egy `print_float` függvényünk is, vagy ha a `State` és a `STATE` együtt szerepelnek. Gondoljuk végig az alábbi meglepő példát:

```
char *
Malloc(unsigned n)
{
        char *p, *malloc(unsigned);
        p = malloc(n);
        if (p == NULL)
                panic("out of memory");
        return p;
}
```

Ezzel az egyszerű módszerrel elérhetjük, hogy nem fogyunk ki észrevétlenül a memóriából. Az alapgondolat itt az, hogy ha a program memóriát szeretne lefoglalni, akkor a `Malloc` függvényt hívja meg a `malloc` helyett. Ha a `malloc` nem járna sikerrel, akkor meghívjuk a `panic` függvényt, ami feltehetőleg a megfelelő hibaüzenet kíséretében leállítja a programot. Ezáltal a függvényt felhasználó programnak nem kell minden egyes `malloc` hívás után külön ellenőriznie az eredményt.

Gondoljunk bele azonban, hogy mi történik, ha egy olyan C megvalósításon használjuk ezt a függvényt, amely nem veszi figyelembe a külső azonosítókban található kis- és nagybetűk közti különbsége-

ket. Ekkor a `malloc` és a `Malloc` elnevezés gyakorlatilag ugyanaz lesz. Más szóval, a `malloc` könyvtári függvényt felváltja a fenti `Malloc` függvény, ami ha meghívja a `malloc` függvényt, akkor tulajdonképpen saját magát hívja meg. Ennek az eredménye persze az lesz, hogy amint megpróbálunk lefoglalni valamennyi memóriát, az önhívó függvényhívások sorozatához, ez pedig fogcsikorgatáshoz és jajveszékeléshez vezet, annak ellenére, hogy a függvény a kis- és nagybetűs azonosítókat megkülönböztető megvalósítások alatt kitűnően fog működni.

7.3. Mekkora egy egész szám?

A C nyelvben három különböző méretű egész típus áll a programozók rendelkezésére. Ezek a `short`, a sima `int` és a `long`. Ott vannak ezen kívül a karakterek is, amelyek tulajdonképpen kis egész számokként viselkednek. A nyelv meghatározása néhány dolgot garantál a különböző egész értékek egymáshoz viszonyított méretével kapcsolatban:

1. A háromféle egész mérete nem csökkenhet. Vagyis a `short` egész csak olyan értékeket vehet fel, amelyek belefférnek egy sima egészbe is, a sima egészek pedig csak olyan értékeket vehetnek fel, amelyek belefférnek egy `long` egészbe. A megvalósításoknak nem kötelező mindhárom mérettípust támogatniuk, de a `short` egészek nem lehetnek nagyobbak a sima egészeknél, a sima egészek pedig nem lehetnek nagyobbak a `long` egészeknél.
2. Egy átlagos egész mérete elég nagy lesz ahhoz, hogy bármely tömbindex belefférjen.
3. A karakterek mérete mindig az adott hardver számára természetes érték.

A legtöbb modern gépen a karakterek 8 bitesek, bár vannak 9 bites karakterek is. Egyre több 16 bites karaktereket használó megvalósítás jelenik meg azonban, hogy képesek legyenek a japán nyelvéhez hasonló, nagyméretű karakterkészletek kezelésére.

Az ANSI szabvány megköveteli, hogy a `long` egészek legalább 32 bit méretűek, a `short` és a sima egészek pedig legalább 16 bit méretűek

legyenek. Mivel a legtöbb gép 8 bites karaktereket használ, amihez a legkényelmesebb a 16 és 32 bites egészek használata, jóformán az összes régebbi C fordító is betartja ezeket a korlátokat.

Mit jelent ez a gyakorlatban? A legfontosabb, hogy nem lehet biztosra venni egyik pontosság meglétét sem. Félhivatalosan annyit valószínűleg elvárhatunk, hogy lesz 16 bites `short` vagy sima egész, és 32 bites `long` típusú egész, de még ezekre sincs semmilyen garancia. A normál egészeket ugyan használhatjuk a táblázatok méreteinek és indexeinek megadására, de mi van akkor, ha olyan változóra van szükségünk, ami tízmilliós nagyságrendig kaphat értékeket?

A leghordozhatóbb mód egy ilyen változó meghatározására valószínűleg az, ha `long` típusúként vezetjük be, de ilyenkor is gyakran az a legtisztább, ha meghatározunk egy „új" típust:

```
typedef long tenmil;
```

Ha ezt a típust adjuk meg az összes ilyen méretű változó meghatározásánál, akkor a legrosszabb esetben is mindössze egyetlen helyen kell megváltoztatnunk a típus leírását, hogy minden változó típusa helyes legyen.

7.4. Előjelesek-e a karakterek vagy sem?

A legtöbb modern számítógép támogatja a 8 bites karaktereket, ezért a legtöbb modern C fordítónál a karakterek megvalósítása 8 bites egészként történik, viszont nem minden ilyen fordító értelmezi ugyanúgy ezeket a 8 bites mennyiségeket.

Ez a kérdés csak akkor válik fontossá, amikor egy `char` típusú mennyiséget egy nagyobb egész típusra alakítunk át. Visszafelé egyértelműen működik a dolog, a felesleges biteket egyszerűen eldobjuk. De ha egy `char` típusú értéket alakít `int` típusúvá, akkor a fordítónak el kell döntenie, hogy előjelesként vagy előjel nélküliként kezelje azt. Az előbbi esetben úgy kell kibővítenie a `char` értéket `int` típusúvá, hogy lemásolja az előjelbitet. Az utóbbi esetben nullákkal kell feltöltenie a további biteket.

Ennek a döntésnek a következményei jóformán mindenkit érintenek, aki olyan karakterekkel dolgozik, amelyeknek a magas helyiértékű bitjei be vannak kapcsolva. A döntés meghatározza, hogy a 8 bites karakterek értéke `-128` és `127` között vagy `0` és `255` között legyen. Ez pedig hatással lesz arra, hogy a programozó miként tervezi meg a hasítótáblákat és a fordítótáblákat.

Ha számít, hogy a program negatív számként kezelje az olyan karakterértékeket, amelyeknél be van kapcsolva a magas helyiértékű bit, akkor valószínűleg a legjobb, ha `unsigned char` típusúként vezetjük be őket. Az ilyen értékeknél garantáltan megtörténik a nullákal való kiegészítés, ha egészre alakítjuk őket, míg a sima `char` típusú értékek előjelesek lehetnek az egyik megvalósításban és előjel nélküliek a másikban.

Mellesleg létezik egy elég gyakori félreértés, miszerint ha `c` egy karakterváltozó, akkor az `(unsigned) c` kifejezéssel megkaphatjuk a `c` előjel nélküli egész megfelelőjét. Ez azért nem működik, mert amikor `unsigned` típusúra alakítunk egy `char` értéket, akkor előbb a program `int` típusúra alakítja azt, ami váratlan eredményt hozhat.

A helyes módszer az `(unsigned char) c` kifejezés használata. Ha egy `unsigned char` típusú értéket alakítunk egész típusúvá, akkor rögtön az `unsigned` típusú értékhez jutunk, anélkül, hogy eközben az értéket `int` típusra alakítanánk.

7.5. Léptető műveletek

Általában két kérdés megválaszolása okoz gondot a léptető műveletek használatakor:

1. A jobbra léptetésnél a megüresedett bitek helyére nullák vagy az előjelbit másolatai kerüljenek?
2. Milyen értékeket vehet fel a léptető számláló?

Az első kérdésre egyszerű a válasz, de néha az adott megvalósításon múlik. Ha `unsigned` típusú az elem, amin a léptetést végezzük, akkor nullákat léptetünk a megüresedett helyekre. Ha a típusa `signed`, akkor az adott megvalósítástól függ, hogy a megüresedett bitekbe

nullák vagy az előjelbit másolata kerül. Ha a jobbra léptetésnél fontos számunkra, hogy mi kerül a megüresedett bitekbe, akkor `unsigned` típusúként vezessük be a kérdéses változót. Ha így teszünk, akkor számíthatunk rá, hogy a megüresedett bitekbe nullák kerülnek.

A második kérdésre szintén egyszerű a válasz. Ha egy *n* bit hosszúságú elemen végezzük el a léptető műveletet, akkor a léptetések száma nagyobb vagy egyenlő nullánál, és *szigorúan* kisebb, mint *n*. Tehát egy művelettel lehetetlen kiléptetni az összes bitet egy elemből. Ennek a megszorításnak a lényege, hogy a nyelv megvalósítása egyszerű legyen a hasonló korlátokkal rendelkező hardveren.

Ha például egy egész érték 32 bites, és `n` egy egész érték, akkor azt írhatjuk, hogy `n<<31` és `n<<0`, de azt nem, hogy `n<<32` vagy `n<<-1`.

Jegyezzük meg, hogy egy előjeles egész jobbra léptetése általában nem egyenértékű a kettő egy hatványával való osztással, még akkor sem, ha az adott megvalósításnál a megüresedett bitekbe az előjelbit másolata kerül. Hogy ezt be is bizonyítsuk, gondoljunk bele, hogy a `(-1)>>1` kifejezés értéke nem lehet nulla, a `(-1)/2` kifejezés értéke viszont a legtöbb megvalósítás esetében nulla. Ez azt sejteti, hogy ha egy léptető műveletet egy osztással helyettesítünk, az lényegesen lelassíthatja a programot. Sokkal gyorsabb például, ha azt írjuk, hogy

```
mid = (low + high) >> 1;
```

a vele egyenértékű

```
mid = (low + high) / 2;
```

helyett, ha biztosak vagyunk benne, hogy a `low+high` értéke nem negatív.

7.6. A nulla memóriacím

A cím nélküli (null) mutatók semmilyen objektumra nem mutatnak. Ezért van az, hogy az értékadásokon és az összehasonlításokon kívül semmi másra ne lehessen használni őket. Nem tudhatjuk például, hogy mi lesz az `strcmp(p,q)` értéke, ha a `p` vagy a `q` cím nélküli mutató.

Hogy ilyenkor mi történik, az az egyik C megvalósításról a másikra változik. Léteznek olyan megvalósítások, amelyeknél hardveres védelmet alkalmaznak, hogy meggátolják a 0 cím olvasását. Ha egy program helytelenül használ egy cím nélküli mutatót egy ilyen megvalósításon, akkor az azonnali hibához vezet. Vannak olyan megvalósítások is, amelyek a 0 cím olvasását engedélyezik, de az írását nem. Ebben az esetben a cím nélküli mutató látszólag egy karakterláncra fog mutatni, ami általában valamilyen értelmetlen adat lesz. Léteznek továbbá olyan megvalósítások is, amelyeknél a 0 címet egyaránt írni és olvasni is lehet. Ha egy ilyen megvalósításon hibásan használunk egy cím nélküli mutatót, akkor azzal akár az operációs rendszert is könnyen átírhatjuk, ami teljes káoszhoz vezet.

A szó szorosan vett értelmében ez a kérdés nem a hordozhatóság problémakörébe tartozik. A cím nélküli mutatók helytelen használata az összes C programban kiszámíthatatlan következményekhez vezethet. Előfordulhat azonban, hogy egy ilyen program látszólag tökéletesen fog működni az adott rendszeren, és a hibára egészen addig nem is derül fény, amíg át nem visszük egy másik gépre.

A legkönnyebben úgy találhatjuk meg ezeket a hibákat, ha egy olyan gépen futtatjuk a programunkat, amely tiltja a 0 cím olvasását.
Az alábbi program megvizsgálja, hogy az adott megvalósítás hogyan kezeli a 0 címet:

```
#include <stdio.h>

main()
{
        char *p;

        p = NULL;
        printf("Location 0 contains %d\n", *p);
}
```

Ez a program hibát okoz egy olyan gépen, ahol tiltva van a 0 cím olvasása, egyébként pedig tízes számrendszerben megjeleníti azt az értéket, ami látszólag a 0 címen található.

7.7. Hogyan csonkol az osztás?

Tegyük fel, hogy a `q` értékét az `a` és `b` értékek hányadosaként kapjuk meg, ahol az osztás maradéka `r`:

```
q = a / b;
r = a % b;
```

A példa kedvéért azt is tegyük fel, hogy `b>0`.

Milyen összefüggés legyen az `a`, `b`, `p` és `q` értékek között?

1. A legfontosabb, hogy a `q*b + r == a` egyenlőség igaz legyen, hiszen ez az összefüggés nem más, mint a maradékos osztás meghatározása.
2. Ha megváltoztatjuk `a` előjelét, akkor `q` előjelének is meg kell változnia, de nagyságának nem.
3. Ha `b>0`, akkor fontos, hogy `r>=0` és `r<b` igaz legyen. Ha például a maradékot egy hasítótábla indexének megadására használjuk, akkor fontos, hogy biztosak lehessünk benne, hogy mindig érvényes indexet kapunk.

Világos, hogy mindhárom tulajdonság kívánatos az egészekkel való maradékos osztásoknál. Sajnos *egyszerre mindegyik nem lehet igaz.*

Vegyük a `3/2` hányadost, ami `1` lesz, az osztás maradéka pedig szintén `1`. Ez megfelel az 1. pontban leírtaknak. De mi legyen a `(-3)/2` értéke? A fenti 2. tulajdonság azt sugallja, hogy a helyes érték a `-1`, de ha ez igaz, akkor a maradék *szintén* `-1` lesz, ami ellentmond a 3. számú tulajdonságnak. Ha viszont a 3. előírásnak akarunk eleget tenni, és a maradék `1` lesz, akkor az 1. tulajdonság miatt a hányados `-2`, ami megsérti a 2. szabályunkat.

Tehát a C és a többi nyelv, amelyben megtaláljuk az egész értékek csonkoló osztását, kénytelen legalább az egyik elvet feladni a fenti háromból. A legtöbb programozási nyelv a 3. elvet nem tartja be, és inkább arra figyel, hogy a maradék előjele megegyezzen az osztandó előjelével. Ez lehetővé teszi, hogy megtartsuk az 1. és 2. számú elveket. A legtöbb C megvalósítás is ezt a gyakorlatot követi.

A C nyelv meghatározása azonban csak az 1. elvben foglaltakra nyújt garanciát, és arra, hogy ha `a=0` és `b>0`, akkor `|r|<|b|` és `r=0`. Ez a tulajdonság kevésbé szigorú, mint a 2. és a 3. elv.

Időnként felesleges rugalmassága ellenére a C szabvány elegendő ahhoz, hogy a maradékos osztás úgy működjön, ahogy akarjuk, feltéve, hogy tudjuk, mit akarunk. Tegyük fel például, hogy adva van az `n` érték, ami egy azonosító karaktereinek függvényeként jön létre, és egy osztással akarjuk megkapni a `h` bejegyzést egy hasítótáblában úgy, hogy a *0=h<HASHSIZE* igaz legyen. Ha tudjuk, hogy `n` sosem lesz negatív, akkor egyszerűen ezt írhatjuk:

```
h = n % HASHSIZE;
```

Ha viszont előfordulhat, hogy `n` negatív lesz, akkor ez nem elég, mert akkor a `h` is lehet negatív. Tudjuk viszont, hogy *h>-HASHSIZE*, tehát írhatjuk azt, hogy

```
h = n % HASHSIZE;
if (h < 0)
        h += HASHSIZE;
```

Vagy ami még jobb, eleve úgy tervezzük meg a programot, hogy az `n` értéke ne lehessen negatív, és `unsigned` típusúként vezetjük be az `n` változót.

7.8. Mekkora egy véletlenszám?

Hajdanán, amikor az egyetlen C megvalósítás még csak a PDP-11 számítógépeken futott, volt egy `rand` nevű függvény, ami egy (ál-)véletlen nem negatív egész értéket adott vissza. A PDP-11 egészei 16 bit hosszúak voltak, beleértve az előjelet is, így a `rand` egy `0` és $2^{15}-1$ közötti értéket adott vissza.

Amikor a VAX-11 rendszerre készült a C megvalósítása, az egészek már 32 bit hosszúak voltak, ami felvetette azt a kérdést, hogy mi legyen ezeken a `rand` függvény értékkészlete.

A kérdésre két párhuzamosan folyó fejlesztés két különböző választ adott. A kaliforniai Berkeley egyetem szakemberei a saját C megvalósításuk kidolgozása során arra az álláspontra helyezkedtek, hogy a `rand` visszatérési értékei öleljék fel az összes nem negatív egész értéket, így az ő változatukban a `rand` egy `0` és $2^{31}-1$ közötti egészet ad vissza.

Az AT&T szakemberei viszont úgy gondolták, hogy könnyebb lesz átvinni a VAX-11 rendszerekre azokat a PDP-11 rendszerre készült programokat, amelyek a `rand` függvény visszatérési értékeként egy 2^{15}-nél kisebb értéket vártak, ha a `rand` a VAX-11 rendszeren is egy `0` és 2^{15} közötti értéket ad vissza.

Ennek eredményeképpen, ha manapság olyan programot írunk, ami a `rand` függvényt használja, akkor mindig az adott megvalósításhoz kell igazítanunk a programot. Az ANSI C ugyan meghatározza a `RAND_MAX` állandót, ami a legnagyobb véletlenszámot ábrázolja, de ezt a régebbi C megvalósításokban általában nem találjuk meg.

7.9. Kis- és nagybetűk közti átalakítások

A `toupper` és `tolower` függvényeknek hasonló a történetük. Eredetileg mindkettő makró volt:

```
#define toupper(c) ((c)+'A'-'a')
#define tolower(c) ((c)+'a'-'A')
```

Ha a `toupper` egy kisbetűs karaktert kap bemenetként, akkor a megfelelő nagybetűs karaktert adja vissza. A `tolower` függvény ennek éppen az ellenkezőjét teszi. Mindkét makró működése abból a szempontból az adott megvalósítás karakterkészletétől függ, hogy megkövetelik, hogy egy nagybetű és a neki megfelelő kisbetű közötti távolság állandó legyen minden betű esetében. Ez a feltevés az ASCII és az EBCDIC karakterkészletek esetében is igaz, és valószínűleg nem is túl kockázatos, hiszen a két makró meghatározásából eredő hordozhatatlanságot arra az egyetlen fájlra korlátozhatjuk, amelyikben benne vannak.

Ezeknek a makróknak azonban van egy hátrányuk is. Ha olyan karaktert adunk át nekik, ami nem a működésüknek megfelelő, vagyis kisbetűt akarunk kicsivé alakítani, illetve nagyot naggyá, akkor értelmetlen adatot adnak vissza. Az alábbi ártatlan programocska tehát, ami egy fájl tartalmát alakítaná csupa kisbetűsre, a makrók miatt nem fog működni:

```
int c;
while ((c = getchar()) != EOF)
            putchar(tolower(c));
```

Ehelyett azt kell írnunk, hogy:

```
int c;
while ((c = getchar()) != EOF)
            putchar(isupper(c)? tolower(c): c);
```

Egy napon az AT&T egyik vállalkozó szellemű fejlesztője felfigyelt rá, hogy a `toupper` és a `tolower` használatát általában megelőzte annak ellenőrzése, hogy a nekik átadott értékek megfelelőek voltak-e. Gondolt egyet, és az alábbi módon írta át a makrókat:

```
#define toupper(c)
      ((c)>='a'&&(c)<='z'?(c)+'A'-'a':(c))
#define tolower(c)
      ((c)>='A'&&(c)<='Z'?(c)+'a'-'A':(c))
```

Aztán rájött, hogy a `c` kiértékelése így minden híváskor valahol egy és három közötti alkalommal történik meg, ami kockázatos lehet az olyan kifejezéseknél, mint a `toupper(*p++)`. Ezért úgy határozott, hogy a `toupper` és a `tolower` makrókat függvényként írja újra. A `toupper` függvény most így nézett ki:

```
int
toupper(int c)
{
      if (c >= 'a' && c <= 'z')
            return c + 'A' - 'a';
      return c;
}
```

A `tolower` szintén hasonló lett.

A változtatás előnye a nagyobb hibatűrés volt, amiért viszont a függvény meghívásával járó többletmunkával kellett fizetni. Hősünk felismerte, hogy nem mindenki lesz hajlandó megfizetni a többletmunka árát, így hát új neveken ismét bevezette a makrókat:

```
#define _toupper(c) ((c)+'A'-'a')
#define _tolower(c) ((c)+'a'-'A')
```

A felhasználók így választhattak a kényelem és a sebesség között.

Ezzel az egésszel csak egy gond volt. A Berkeley szakemberei és néhány más C megvalósítás sem követte ezt a példát. Ez azt jelenti, hogy egy AT&T rendszeren készült C program, amiben ott a `toupper` vagy a `tolower`, és a programozó feltételezi, hogy ezek nem a működésüknek megfelelő értékeket is nyugodtan kaphatnak, akkor lesz olyan C megvalósítás, amelyen a program nem fog működni. Ezt a fajta hibát pedig rendkívül nehezen találja meg olyasvalaki, aki sosem hallotta ezt a történetet.

7.10. Előbb felszabadítani, és csak azután újrafoglalni?

A legtöbb C megvalósítás három memóriakezelő függvényt bocsát a felhasználók rendelkezésére. Ezek a `malloc`, a `realloc`, és a `free`. A `malloc(n)` hívása egy mutatót ad vissza, ami `n` karakternyi frissen lefoglalt memóriára fog mutatni, amit a programozó szabadon felhasználhat. Ha a `free` függvénynek átadunk egy olyan mutatót, amit korábban a `malloc` függvény meghívásával kaptunk, akkor az adott memóriaterület felszabadul, és újra felhasználhatóvá válik. Ha meghívjuk a `realloc` függvényt egy lefoglalt területet kijelölő mutatóval és egy új mérettel, akkor a memóriaterület az új méretre zsugorodik vagy bővül, és eközben lehet, hogy átkerül egy másik helyre.

Ez nem mindig volt így. A UNIX rendszer kézikönyvének hetedik kiadása egy kissé más viselkedést ír le:

> A *`realloc`* függvény *`size`* bájt méretűre módosítja annak a blokknak a méretét, amire a *`ptr`* mutat, és visszaad egy muta-

tót a (közben esetleg áthelyezett) blokkra. A blokk tartalma egy akkora részen változatlan marad, amelynek mérete megegyezik a régi és az új méret közül a kisebbikkel.
A `realloc` akkor is működik, ha a `ptr` egy olyan blokkra mutat, ami a `malloc`, `realloc` és `calloc` legutolsó meghívása után szabadult fel, tehát a `free`, `malloc` és `realloc` egymás utáni használatával a tárhely tömörítésére használhatjuk a `malloc` keresési stratégiáját.

Más szóval, ez a megvalósítás lehetővé tette, hogy újra lefoglaljunk egy memóriaterületet, *azután hogy felszabadítottuk*, ha az újbóli lefoglalás elég gyorsan történt. A „Hetedik Kiadásnak" megfelelő rendszereken tehát megengedett a következő:

```
free(p);
p = realloc(p, newsize);
```

Egy ilyen sajátos rendszeren, a listaelemekhez tartozó memóriát az alábbi különös módszer segítségével is felszabadíthatjuk:

```
for (p = head; p != NULL; P = p->next)
      free((char *) p);
```

Mindezt anélkül tehetjük, hogy azon kellene gondolkoznunk, hogy a `free` meghívása érvényteleníti-e a `p->next` hivatkozást.

Mondanom se kell, ezt a megoldást nem javaslom, már csak azért sem, mert nem minden C megvalósítás őrzi meg elég hosszú ideig a felszabadított memóriát. A kézikönyv hetedik kiadása azonban egy dologról hallgat. A `realloc` egy korábbi megvalósításában *megkövetelte*, hogy a számára átadott területet előbb felszabadítsuk. Ebből kifolyólag számos olyan C program forog még közkézen, amiben az újrafoglalás előtt felszabadítják a memóriát. Erre érdemes odafigyelni, amikor egy régi C programot viszünk át egy új rendszerre.

7.11. Példa a hordozhatósági problémákra

Nézzünk meg egy példát, amit sokan és sokszor megoldottak már. Az alábbi függvénynek két paramétere van, egy hosszú egész, és egy függvénymutató. Először tízes számrendszerbe teszi az egész értéket, majd a tízes számrendszerbeli alak minden egyes karakterével meghívja a függvényt:

```
void
printnum(long n, void (*p)())
{
        if (n < 0) {
                (*p) ('-');
                n = -n;
        }
        if (n >= 10)
                printnum(n/10, p);
        (*p) ((int)(n % 10) + '0');
}
```

Ez a kód elég egyértelmű. Először megnézzük, hogy `n` negatív-e. Ha igen, akkor kiírjuk az előjelet, és pozitívra változtatjuk `n` értékét. Ezután megnézzük, hogy `n≥10` igaz-e. Ha igen, akkor tízes számrendszerbeli alakja legalább két jegyből áll, tehát a `printnum` függvényt ismétlődően meghívva kiírjuk az összes számjegyet az utolsó kivételével. Végül, az `n%10` típusát `int` típusúra alakítva az utolsó számjegyet is kiírjuk, biztosítva ezzel, hogy a megfelelő értéket adjuk át a `*p` számára.[4]

Ez felesleges az ANSI C szabványnál, de véd az ellen, ha valaki esetleg úgy próbálná meg egy régebbi rendszerre átvinni a programot, hogy egyszerűen átírja a függvény fejlécét.

Ez a program, egyszerűségének ellenére, számos gondot okozhat a hordozhatóság szemszögéből. Először is ott van az a módszer,

[4] Abban a dolgozatban, amelyre ez a könyv épül, ez volt az utolsó utasítás a `printnum` függvényben:

```
(*p) (n % 10 + '0');
```

Ez csak olyan gépeken működik, amelyeken a `long` és az `int` típusok belső ábrázolása megegyezik.

ahogy az n alacsony helyiértékű tízes számrendszerbeli számjegyét alakítja át karakterré. Azzal nincs gond, hogy az n%10 kifejezéssel kapjuk meg az alacsony helyiértékű tízes számrendszerbeli számjegyet, de azzal igen, hogy úgy állítjuk elő a megfelelő karakteres ábrázolást, hogy hozzáadjuk ehhez a '0' karaktert. Ez az összeadás azt feltételezi, hogy a gép jelsorrendjében szünet nélkül követik egymást a számjegyek, vagyis a '0'+5 értéke megegyezik az '5' értékével, és így tovább. Habár ez a feltevés az ASCII és EBCDIC karakterkészleteket használó gépeken, továbbá az ANSI szabványhoz igazodó megvalósításokon igaz, vannak olyan gépek, ahol nem ez a helyzet. A problémát egy táblázat használatával kerülhetjük meg. Mivel a karakterlánc-állandók tulajdonképpen karaktertömbök, használhatjuk őket tömbök nevei helyett. Az alábbi meglepő kifejezést tehát

```
"0123456789"[n % 10]
```

nyugodtan használhatjuk a következő példában:

```
void
printnum(long n, void (*p)())
{
        if (n < 0) {
                (*p)('-');
                n = -n;
        }
        if (n >= 10)
                printnum(n/10, p);
        (*p)("0123456789"[n % 10]);
}
```

A következő gond akkor jelentkezik, ha n<0. A program ilyenkor kiírja a negatív előjelet, majd n értéke -n lesz. Ennek az értékadásnak túlcsordulás lehet a következménye, mivel a kettes komplemensű gépek általában több negatív, mint pozitív szám ábrázolását teszik lehetővé. Nevezetesen, ha egy (hosszú) egész k bit plusz az előjelbit hosszúságú, akkor a -2^k értéket lehet ábrázolni, de a 2^k értéket nem.

Ennek a problémának számos megoldása létezik. A legkézenfekvőbb, ha a -n értéket egy unsigned long típusú változóba tesszük,

és kész. No de a -n nem értékelhető ki, mert ha megpróbáljuk, az túlcsorduláshoz vezethet!

Az egyes és kettes komplemenst használó gépeken egyaránt biztosak lehetünk benne, hogy ha egy pozitív szám előjelét változtatjuk meg, az nem fog túlcsordulást okozni. Gond csak a *negatív számok* előjelének megváltoztatásánál van, tehát elkerülhetjük a bajt, ha vigyázunk, nehogy pozitívra próbáljuk változtatni n értékét.

Persze, ha már egyszer kiírtuk egy negatív érték előjelét, akkor szeretnénk ugyanúgy kezelni a negatív és a pozitív számokat. Ezt úgy tehetjük meg, hogy az előjel kiírása után kikényszerítjük, hogy n negatív legyen, és ezután csak negatív értékekkel számolunk. Ha így teszünk, akkor gondoskodnunk kell róla, hogy csak egyszer hajtsuk végre a programnak azt a részét, amelyik az előjelet kiírja. Ezt a legkönnyebben úgy tehetjük meg, ha két függvényre bontjuk a programot. A printnum függvény csak azt ellenőrzi majd, hogy a kiírandó szám negatív-e, és ha az, akkor kiírja a negatív előjelet. Ezen kívül mindenképpen meghívja a printneg függvényt, az n abszolút értékének negatív megfelelőjével. A printneg függvény így már biztosítja, hogy az n értéke mindig negatív vagy nulla lesz:

```
void
printneg(long n, void (*p)())
{
        if (n <= -10)
                printneg(n/10, p);
        (*p)("0123456789"[-(n % 10)]);
}

void
printnum(long n, void (*p)())
{
        if (n < 0) {
                (*p)('-');
                printneg(n, p);
        } else
                printneg (-n, p);
}
```

Még ez sem az igazi. Az `n` első és utolsó számjegyének ábrázolásához az `n/10`, illetve az `n%10` kifejezéseket használtuk (az előjel megfelelő módosításával). Emlékezzünk vissza, hogy az egészek osztása során kapott eredmény függhet az adott megvalósítástól, ha az egyik tag negatív, ezért lehet, hogy `n%10` pozitív lesz! Ebben az esetben `-(n%10)` negatív lenne, és túlszaladnánk a számjegyeket tartalmazó tömb végén.

Ennek a problémának két átmeneti változó létrehozásával vesszük elejét, amelyekbe a hányadost és a maradékot tesszük. Az osztás elvégzése után ellenőrizzük, hogy a maradék a határokon belül van-e, és ha nem, akkor mindkét változót módosítjuk. A `printnum` függvényen ezúttal nem változtattunk, ezért csak a `printneg` függvényt mutatjuk be:

```
void
printneg(long n, void (*p)())
{
        long q;
        int r;

        q = n / 10;
        r = n % 10;
        if (r > 0) {
                r -= 10;
                q++;
        }
        if (n <= -10)
                printneg(q, p);
        (*p)("0123456789"[-r]);
}
```

Szép kis munkát kellett végeznünk a hordozhatóság biztosítása végett!

De mi értelme ennek? Hát az, hogy a mai világban folyton változó fejlesztői környezetekben kell boldogulnunk. Megfoghatatlanságuk ellenére a programok általában túlélik azt a hardvert, amin futnak. A jövő hardverének természetét pedig nehéz előre megjósolni. A hordozható programok azonban mindig tartósabbak.

A hordozható programok ezen kívül általában kevesebb hibát is tartalmaznak. A példánk esetében a legtöbb erőfeszítést annak érdekében tettük, hogy a `printnum` akkor is helyesen működjön, ha a legnagyobb negatív értéket adjuk át neki. Sok olyan forgalomban lévő programmal találkoztam már, ami éppen ilyen helyzetekben mondja be az unalmast.

7.1. gyakorlat. A 7.3. részben azt mondtuk, hogy egy 8 bites karaktereket használó gépen az egészek valószínűleg 16 vagy 32 bitesek lesznek. Miért?

7.2. gyakorlat. Írjuk meg az `atol` függvény hordozható változatát. A függvény paramétere egy null karakterrel lezárt karakterláncra irányuló mutató, visszatérési értéke pedig a karakterláncnak megfelelő `long` érték legyen. Tegyük fel a következőket:

- A bemenet mindig egy érvényes hosszú egész ábrázolása lesz, tehát az `atol` függvénynek nem kell ellenőriznie, hogy a határokon belül van-e az érték.
- Csak számjegyeket, illetve a + és a – előjelet adhatjuk át a függvénynek. A bemenet az első érvénytelen karakternél véget ér.

8

Tippek és megoldások

Elérkeztünk kis túránk végéhez, amelynek során láttuk, hogyan kerülhetnek bajba a C programozók. Sokaknak most bizonyára ugyanaz a kérdés motoszkál a fejében, mint azoknak, akik a könyv vázlatát olvasták annak idején: „Hogyan kerülhetném el ezeket a problémákat?".

Talán a legjobb megoldás erre, *ha tudjuk, hogy mit csinálunk.* A legbosszantóbb bonyodalmakat mindig az olyan programok okozzák, amelyek látszólag működnek, de valójában hibákat rejtenek. Mivel rejtett hibákról van szó, a legkönnyebben úgy vehetjük észre őket, ha már jó előre gondolunk rájuk. Ha addig játszadozunk egy hibás programmal, amíg látszólag működni fog, azzal szinte biztosan olyan programhoz jutunk, ami csak majdnem lesz működőképes.

Ezt a legszebben egy csembalókészítésről szóló kézikönyvben (hol máshol) fogalmazták meg. A mű szerzője egy bizonyos David Jacques Way, aki minden bizonnyal nagyra értékelte a magabiztos tudást, és akit most szíves engedelmével idézek is:

> A hibákhoz a hiten át vezet az út. Az a tény is igazolja ezt, hogy aki hibázik, az mindig azt mondja, hogy „De én azt hittem...." Rá se hederítsünk az ilyen badarságra. Mielőtt bármit összeragasztanánk, tudnunk kell, mit csinálunk. Állítsuk össze a részeket ra-

> gasztó nélkül (ezt „száraz próbának" nevezik), majd tanulmányozzuk át, hogyan illeszkednek egymáshoz, és nézzük meg a tervrajzunkat, ami megmutatja, hová kerülnek a részek. Miután összeragasztottunk valamit, ismét ellenőrizzük. Számtalanszor hallottam már elkeseredett történeteket, amelyek mind így kezdődtek: „Tegnap éjjel ezt meg ezt csináltam, és amikor reggel megnéztem...."
> Drága barátom, ha tegnap éjjel is megnézted volna, akkor még szétszedhetted volna, és a helyére tehetted volna a részeket. Sokan közülünk a szabadidőnkben barkácsolunk, így nagy a kísértés, hogy éjszakába nyúlóan dolgozzunk. De ha hinni lehet azoknak a telefonhívásoknak, amelyeket kapok, akkor a legtöbb hibát a lefekvés előtti legutolsó művelettel követjük el. A legokosabb tehát, ha még az utolsó művelet előtt lefekszünk aludni.

Mennyire igaz ez a programozásra is, ha az „összeragasztás" alatt azt értjük, amikor számos apró részből állítunk össze egy nagyobb programot! A megbízható programok írásának kulcsa az, ha már azelőtt megértjük, hogyan fognak egymáshoz kapcsolódni az elemek, hogy ténylegesen összeillesztenénk azokat.

Ezek az ismeretek különösen fontosak, ha időszűkében vagyunk. Egy hosszúra nyúlt hibakeresés vége felé egyre nagyobb a kísértés, hogy szinte véletlenszerűen próbáljunk ki dolgokat, és ha valami működni látszik, akkor többet ne is foglalkozzunk vele. Ez az út bizony a káosz felé visz!

8.1. Tippek

Álljon itt néhány gondolat a hibák elkerülésével kapcsolatban.

Ne beszéljük be magunknak, hogy látunk valamit, ha az nincs ott! A hibák néha nagyon meggyőzőek. Jó példa erre az 1.1. részben látható kód, ami egy kicsit másképpen nézett ki abban a dolgozatban, amelyik e könyvvé nőtte ki magát:

```
while (c == '\t' || c = ' ' || c == '\n')
        c = getc(f);
```

A fenti formájában ez a kód nem szabályos C. A `while` utasításban található műveletek közül a = végrehajtása a legutolsó, tehát a fentieket így kell értelmeznünk:

```
while ((c == '\t' || c) = (' ' || c == '\n'))
        c = getc(f);
```

Ez persze helytelen, hiszen a

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

nem állhat egy értékadás bal oldalán. Emberek ezrei nézték meg a példát, de egészen addig senki nem vette ezt észre, amíg Rob Pike fel nem hívta rá a figyelmemet.

Amikor nekiálltam a könyvnek, az eredeti dolgozat olvasóinak megjegyzéseit félretettem egészen addig, amíg már majdnem végeztem a munkával. A hibás példa így bekerült abba a vázlatba, ami bejárta a Bell laboratóriumot, és abba is, amit az Addison-Wesley kiküldött a bírálóknak. Egyetlen olyan bíráló sem akadt, aki felfedezte volna a hibát.

Legyenek világosak a szándékaink! Ha valami olyat írunk, aminek több jelentése is lehet, akkor zárójelekkel vagy valamilyen más módon gondoskodjunk róla, hogy egyértelmű legyen, hogy mit is akartunk. Ezzel nemcsak a saját dolgunkat könnyítjük meg, amikor később a programhoz visszatérve meg kell értenünk, hogy mit is akartunk, hanem másoknak is egyszerűbb lesz a dolga, ha később el kell olvasniuk a programunkat.

Néha lehetőségünk van rá, hogy úgy írjunk meg valamit, hogy azzal megelőzzük a gyakori hibákat. Vannak olyan programozók például, akik az állandókat az egyenlőséget vizsgáló összehasonlítások bal oldalára teszik. Vagyis ahelyett, hogy

```
while (c == '\t' || c == ' ' || c == '\n')
        c = getc(f);
```

azt írják, hogy

```
while ('\t' == c || ' ' == c || '\n' == c)
        c = getc(f);
```

Így, ha véletlenül = kerül a == helyére, a fordítóprogram hibát jelez, hiszen a

```
while ('\t' = c || ' ' == c || '\n' == c)
        c = getc(f);
```

szabálytalan, mert a `'\t'` kifejezésnek próbál értéket adni.

Vizsgáljuk meg a legegyszerűbb eseteket! Ez a programok tesztelésére és működésük elemzésére is vonatkozik. Oly sok program mondja fel a szolgálatot, amikor a bemenete üres vagy csak egy elemből áll, hogy ezeket az eseteket érdemes legelőször kipróbálnunk.

Ugyanez a helyzet a programtervezéssel is. Amikor megtervezünk egy programot, mindig tegyük fel magunknak a kérdést, hogy mit fog az tenni, ha a számára átadott bemeneti adatok halmaza üres.

Használjunk aszimmetrikus korlátokat! Olvassuk át újra a 3.6. részt, amiben a tartományok ábrázolásáról volt szó. Az a tény, hogy a C indexek nullától indulnak, rendkívül leegyszerűsít mindenféle számolós problémát, ha megértjük ezeknek a működését.

A programhibák a sötét zugokból leselkednek ránk. Minden C megvalósítás kicsit más, mint a többi. Próbáljunk megmaradni a nyelv jól ismert részeinél. Ha betartjuk ezt a szabályt, akkor sokkal könnyebben tudjuk majd áttenni egy új gépre a kódunkat, és kevesebb esélyt hagyunk a fordítási hibáknak.

Biztosan emlékszünk még a 3.1. részre, ahol a tömbök és mutatók tárgyalását egy tisztázatlan kérdéssel fejeztük be. Ha egy program működése tényleg azon múlna, hogy az adott megvalósításnál az összes részlet stimmel-e, akkor az a program előbb-utóbb csütörtököt mondana.

Az sem árt, ha megpróbálunk védekezni a felületesen megírt könyvtárak ellen. Egyszer rendesen megszenvedtem egy programmal, amit át kellett tennem egy másik gépre, mert a program bízott benne, hogy egy több ezer karakterből álló formázott karakterlánccal meghívhatja a `printf` függvényt. Ezzel nem is lenne baj, csak hát a `printf` függvénynek létezik olyan megvalósítása, ami ezzel nem boldogul.

Ez a tipp különösen fontos lehet, ha olyan szolgáltatás használatán gondolkozunk, amit csak egyetlen gyártó támogat. Sose felejtsük el, hogy a program túléli a gépet.

Alkalmazzuk a védekező programozást! Soha ne tételezzünk fel többet a felhasználókról és az adott megvalósításról, mint amennyit feltétlenül szükséges. Emlékszem, egyszer beszélgettem valakivel, akinek egy rendszert építettem. Az eszmecserénket valahogy így folytattuk le:

– Milyen kódok kerülhetnek a mezőnek ebbe a részébe?
– Csak az `X`, az `Y` vagy a `Z`.
– És mi van akkor, ha valami más kerül oda?
– Olyan nincs.
– Nos, a programnak *valamit* csinálnia kell, ha ez mégis bekövetkezik. Mit csináljon?"
– Mindegy.
– Tényleg mindegy?
– Igen.
– Akkor nem gond, ha töröljük a teljes adatbázist, ha a program nem az `X`, `Y` vagy `Z` értéket kapja itt?
– Ne mondjon már ilyen badarságot! Nem törölheti csak úgy a teljes adatbázist!
– Akkor tehát *mégsem mindegy*, hogy mit csinál a program. Akkor mi legyen, mit csináljon?

Az „olyan nincs" esetek néha mégis bekövetkeznek, de a hibatűrő programokat ez nem éri váratlanul.

Jó lenne, ha az egyes C megvalósítások még több programhibát el tudnának csípni. Sajnos ez több okból is meglehetősen nehéz feladat. A legfontosabb ok a C nyelv történetére vezethető vissza, hiszen so-

kan olyan feladatok megoldására használták ezt a nyelvet, amelyeknek a megoldása korábban assembly nyelven történt. Ezért sok olyan C program létezik, amelynek bizonyos részei szándékosan olyan dolgokat művelnek, amik szigorúan véve kívül esnek a nyelv határain. Ilyenek például az operációs rendszerek. Egy szigorúan ellenőrzött C megvalósításban kellene, hogy létezzen egy „menekülési útvonal", ami lehetővé tenné az ilyen programok számára, hogy elvégezzék a géptípushoz kötődő tennivalójukat, miközben a program hordozható részeit továbbra is szigorú ellenőrzés alatt tarthatnánk.

Vannak olyan dolgok is, amelyek természetüknél fogva nehezen ellenőrizhetők. Vegyük például ezt a függvényt:

```
void
set(int *p, int n)
{
        *p = n;
}
```

Ez most szabályos vagy sem? A válasz természetesen az, hogy anélkül lehetetlen megmondani, hogy ismernénk a környezetét. Ha így hívjuk meg:

```
int a[10];
set(a+5, 37);
```

akkor szabályos, de ha így:

```
int a[10];
set(a+10, 37);
```

akkor nem. Ennek ellenére az utóbbi részlettel önmagában semmi gond. Az ANSI C szabvány megengedi, hogy a program előállítsa egy tömb végét közvetlenül követő hely címét. Ha tehát ezt a fajta hibát nyakon tudja csípni egy C megvalósítás, akkor le a kalappal előtte.

Ezzel nem azt akarom mondani, hogy lehetetlen olyan C megvalósításokat készíteni, amelyek alaposabban megvizsgálnának egy programot. Természetesen lehet, és kapható is néhány ilyen. De olyan egy sem létezik, ami az összes hibát megtalálja egy programban.

8.2. Megoldások

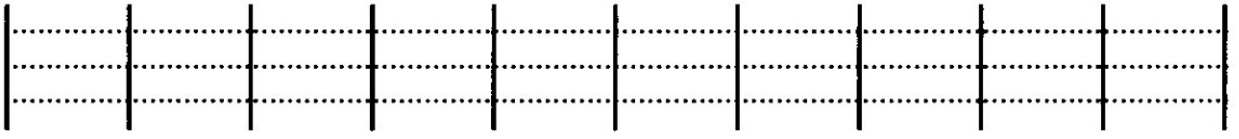
0.1. gyakorlat

A jó hírnév nagyon fontos, amikor több termék közül kell választanunk. És ha egyszer oda a jó hírnév, akkor nagyon nehéz visszaszerezni. Idő kell ahhoz, hogy az emberek elhiggyék, hogy a cég legfrissebb termékeinek kiváló minősége nem csak a véletlen műve.

A legtöbben sohasem vásárolnának olyan terméket, amiről tudják, hogy komoly tervezési hibákkal rendelkezik. Az egyetlen kivétel ez alól, ha valaki számítógépprogramot vásárol. A legtöbb programozó legalább néhány programját másoknak írja. Az emberek számítanak rá, hogy a programok nem működnek tökéletesen. Okozzunk nekik kellemes csalódást!

0.2. gyakorlat

Tizenegy. 10 kerítésrész lesz, de 11 oszlop. Számoljuk meg. A 3.6. részből többet is megtudhatunk arról, hogy ennek a kérdésnek mi köze a programozási hibákhoz.



0.3. gyakorlat

Sokkal könnyebben lehet egy bonyolult eszközt biztonságosabbá tenni, mint egy egyszerűt. A konyhai robotgépeken mindig van egy biztonsági zár, nehogy az ujjainkkal fizessünk egy jó ebédért. A késeken nincs semmi. Ha megpróbálnánk valamilyen ujjvédő szerkezetet készíteni egy ilyen egyszerű és rugalmas eszközhöz, azzal éppen az egyszerűségét tennénk tönkre. Az eredmény valószínűleg jobban is hasonlítana egy robotgépre, mint egy késre.

Ha megakadályozzuk, hogy hülyeséget csináljunk, azzal gyakran az ügyes ötletek megvalósítását is megnehezítjük.

1.1. gyakorlat

Ahhoz, hogy eldönthessük, hogy működik-e a megjegyzések egymásba ágyazása, egy olyan jelsorozatot kell találnunk, ami mindkét megvalósítás alatt működik, de mindegyikben mást fog jelenteni. Egy ilyen jelsorozat szükségszerűen tartalmazni fog legalább egy beágyazott megjegyzést. Kezdjük tehát ezzel:

```
/* /* */
```

A beágyazást támogató rendszereken bármi is következzen ez után, az egy megjegyzés része lesz, a többi rendszeren pedig a valódi jelentése lesz a mérvadó. Rögtön arra gondolunk, hogy tegyünk a kifejezés mögé egy idézőjelek között lévő, a megjegyzés végét jelző jelet:

```
/* /* */ " */ "
```

Ha a megjegyzések egymásba ágyazása működik, akkor ez egy idézőjelet fog eredményezni. Ha nem, akkor egy karakterlánc-literált kapunk. Ezt tehát folytathatjuk egy másik megjegyzést megnyitó jellel, és még egy idézőjellel:

```
/* /* */ " */ " /* "
```

Ha a megjegyzések egymásba ágyazása működik, akkor ez egy idézőjelek között lévő, egy megjegyzést megnyitó jel lesz. Ha nem, akkor viszont egy idézőjelek között lévő, megjegyzést bezáró jelet látunk, amit egy lezáratlan megjegyzés követ. Már csak annyi a teendőnk, hogy bezárjuk ezt a megjegyzést:

```
/* /* */ " */ " /* " /* */
```

A kifejezés értéke `" */ "` lesz, ha a megjegyzések egymásba ágyazása működik, és `" /* "`, ha nem.

Miután megoldottam ezt a problémát nagyjából a fent leírtak szerint, Doug McIlroy az alábbi elképesztő megoldással állt elő:

```
/*/*/0*/**/1
```

Ez a megoldás a „maximális majszolás" szabályára épül. Ha a megjegyzések egymásba ágyazása működik, akkor a kifejezés értelmezése a következő:

```
/* /* /0 */ * */ 1
```

A két `/*` jelnek a két `*/` jel lesz a párja, tehát a kifejezés értéke `1`. Ha a megjegyzések egymásba ágyazása nem működik, akkor a megjegyzés belsejében lévő `/*` jelet figyelmen kívül hagyja a fordító. Még a `/` jelnek sincs semmilyen különleges szerepe a megjegyzésen belül. A kifejezés értelmezése tehát a következő lesz:

```
/* / */ 0 * /* */ 1
```

A `0*1` értéke pedig `0` lesz.

1.2. gyakorlat

Az egymásba ágyazott megjegyzésekkel átmenetileg könnyen eltávolíthatunk a programból egy kódrészletet. Tegyünk a kérdéses kód elé egy megjegyzésjelet, és a kód után zárjuk be. Ennek megvan a maga hátránya is. Ha terjedelmes részletet távolítunk el ezzel a módszerrel, akkor könnyen előfordulhat, hogy nem vesszük észre, hogy eltávolítottuk azt.

A C nyelv meghatározása azonban nem engedélyezi az egymásba ágyazott megjegyzéseket, ezért a hűséges programozónak nincs is más választása. Annak a programozónak a kódja, aki egymásba ágyazott megjegyzéseket használ, egyébként sem fog működni egy csomó fordítóprogrammal. Tehát csak olyan programokban szabad egymásba ágyazott megjegyzéseket használni, amelyeket forrásfájl formában nem kívánunk terjeszteni. Az is elképzelhető, hogy ezek a programok az újabb vagy átdolgozott C megvalósításokon sem fognak működni.

Ebből kifolyólag én bizonyosan nem engedélyezném az egymásba ágyazott megjegyzések használatát, ha nekem kellene megírnom egy C fordítót, és akkor sem használnám őket, ha a fordítóm megengedné ezt. Persze a végső döntést mindenkinek magának kell meghoznia.

1.3. gyakorlat

A maximális majszolás szabálya miatt a `--` jelet már azelőtt egy tokennek veszi a program, hogy elérné a `>` jelet.

1.4. gyakorlat

Az egyetlen értelmes jelentés, amit ebből kibogozhatunk, a következő:

```
a ++ + ++ b
```

A maximális majszolás szabálya miatt azonban így kell darabokra tördelnünk a kifejezést:

```
a ++ ++ + b
```

Ez nyelvtanilag helytelen, mert az alábbi kifejezéssel egyenértékű:

```
((a++)++) + b
```

Viszont az `a++` nem balérték, és így nem végezhetjük el rajta a `++` műveletet. A *lexikális* kétértelműség feloldására vonatkozó szabályok miatt lehetetlen olyan formára hozni ezt a példát, hogy az *szintaktikailag* helyes legyen. A gyakorlatban persze célszerű kerülni az efféle szerkezeteket, hacsak nem vagyunk teljesen biztosak benne, hogy mi a pontos jelentésük.

2.1. gyakorlat

Egy kicsit másképpen írva a példát

```
int days[] = { 31, 28, 31, 30, 31, 30,
               31, 31, 30, 31, 30, 31, };
```

már jól látható, hogy minden kezdőértékeket tartalmazó sor végén vessző áll. Mivel a sorok formailag hasonlók, sokkal egyszerűbben kezelhetjük a hosszú, kezdőértékből álló listákat az olyan automatikus fejlesztőeszközökkel, mint a szerkesztőprogramok.

2.2. gyakorlat

A Fortran és a Snobol nyelvekben az utasításokat lezárja a sor vége. Mindkét nyelv lehetőséget ad arra is, hogy egy utasítás több soron is átíveljen, amit az utasítás *második* és soron következő soraiban jeleznünk kell. A Fortran nyelvben a jel a sor 6. oszlopába tett nem üres karakter (a 0–5. oszlopokat fenntartották a címkéknek), a Snobol nyelvben pedig egy . vagy egy + a sor 1. oszlopában.

Kissé különösnek tűnik, hogy egy sor jelentését befolyásolhatja az utána következő sor. Néhány nyelvben ezért az *n*. sorban találunk valamilyen jelzést arra nézve, hogy még az *n+1*. sor is az utasításhoz tartozik. A UNIX rendszer parancsértelmezőjének például a sor végén álló \ jelzi, hogy a következő sor is még az utasítás része. A C szintén ezt a szokást követi az előfeldolgozóban és a karakterláncoknál. Vannak olyan nyelvek, például az Awk vagy a Ratfor, amelyekben akkor folytatódik egy utasítás a következő sorban, ha valami olyasmire végződik a sor, amit feltétlenül folytatni kell, mondjuk egy műveleti jelre vagy egy nyitó zárójelre. Úgy tűnik, ezek az eljárások mind jól működnek a gyakorlatban, habár nem könnyű precízen meghatározni őket.

3.1. gyakorlat

A `bufwrite` programok azon a feltevésen alapszanak, hogy visszatérhetünk, ha teljesen tele az átmeneti tár és csak akkor ürítjük ki, amikor legközelebb meghívjuk a `bufwrite` függvényt. Ha a `bufptr` nem mutathat a táron túlra, akkor az jelentősen megnehezíti a helyzetünket. Hogyan jelezzük, hogy megtelt a tár?

A legkevésbé fájdalmas megoldásnak az tűnik, hogy megpróbáljuk elkerülni, hogy a tár tele legyen, amikor a `bufwrite` visszatér. Ehhez a tárba kerülő utolsó karaktert különleges esetként kell kezelnünk.

Arra is vigyáznunk kell, hogy csak akkor növeljük meg a `p` értékét, ha biztosak vagyunk benne, hogy nem egy tömb utolsó elemére mutat. Ez tulajdonképpen azt jelenti, hogy `p` értékét azután nem növelhetjük meg, hogy beolvastuk az utolsó bemenő karaktert. Ezt itt most egy további ellenőrzéssel oldjuk meg, amit a ciklus minden végrehajtásakor elvégzünk.

A másik megoldás az lenne, ha megkettőznénk a ciklust:

```
void
bufwrite(char *p, int n)
{
        while (--n >= 0) {
                if (bufptr == &buffer[N-1]) {
                        *bufptr = *p;
                        flushbuffer() ;
                } else
                        *bufptr++ = *P;
                if (n > 0)
                        p++;
        }
}
```

Figyeljünk rá, nehogy megnöveljük p értékét, ha már tele a tár, elkerülve ezzel a `buffer[N]` címének létrehozását, ami nem megengedett.

A `bufwrite` második változata még ennél is bonyolultabb. A belépéskor tudjuk, hogy legalább egy karakter lesz a tárban, így kezdetben soha nem kell ürítenünk. A végén viszont lehet, hogy szükség lesz az ürítésre. A ciklus utolsó végrehajtásakor pedig ismét vigyáznunk kell, nehogy megnöveljük p értékét:

```
void
bufwrite(char *p, int n)
{
        while (n > 0) {
                int k, rem;
                rem = N - (bufptr - buffer);
                k = n > rem? rem: n;
                memcpy(bufptr, p, k);
                if (k == rem)
                        flushbuffer();
                else
                        bufptr += k;
                n -= k;
                if (n)
                P += k;
        }
}
```

A k értékét, vagyis az aktuális ismétlésben másolt karakterek számát összevetjük a rem értékével, vagyis a tárban még szabadon maradt karakterek számával. Ezzel ellenőrizzük, hogy lesz-e még szabad hely a tárban a másolás után, vagy ürítenünk kell. A p értékét csak azután növeljük meg, ha n értékét összehasonlítva nullával ellenőriztük, hogy a ciklus utolsó ismétlésénél járunk-e.

3.2. gyakorlat

```
void
flush()
{
        int row;
        int k = bufptr - buffer;
        if (k > NROWS)
                k = NROWS;
        for (row = 0; row < k; row++) {
                int *p;
                for (p = buffer+row; p < bufptr;
                                p += NROWS)
                        printnum(*p);
                printnl();
        }
        if (k > 0)
                printpage();
}
```

A két változat között az a különbség, hogy az itt található példában csak a printpage hívása követi a k>0 összehasonlítást, míg a 3. fejezetben található példában a teljes for ciklus szerepel. Azt a változatot így fordíthatnánk le magyarra: „Ha ki kell még valamit nyomtatni, akkor nyomtassuk ki, majd kezdjünk új oldalt". Az itt található változat ehelyett azt jelenti, hogy „Nyomtassunk ki mindent, ami megmaradt, és ha van ilyesmi, akkor kezdjünk új oldalt". A k szerepe a fenti példa for ciklusában nem olyan egyértelmű, mint a 3. fejezetben található változatban. Ott rögtön világos volt, hogy amennyiben k értéke nulla, akkor átugorjuk a ciklust.

Bár a két program technikailag egyenértékű, valamelyest más szándékot fejeznek ki. Az lesz a jobb választás, amelyik jobban kifejezi a programozó szándékát.

3.3. gyakorlat

A bináris keresések elméletben nagyon egyszerűek, de a gyakorlatban sokan eltévesztik őket. Most két változatot mutatunk be, mindkettőben aszimmetrikus korlátokat használva. Az elsőben indexeket, a másodikban mutatókat használunk.

Tegyük fel, hogy *x*, a keresett elem, a *k*. eleme a tömbnek, ha egyáltalán benne van. Kezdetben csak annyit tudunk a *k* értékről, hogy *0=k<n*. A célunk az, hogy addig szűkítsük ezt a tartományt, amíg meg nem találjuk a keresett elemet, vagy amíg meg nem tudjuk mondani, hogy nincs ilyen elem a tömbben.

Ehhez *x* értékét összehasonlítjuk a tartomány közepén található elem értékével. Ha egyenlők, akkor meg is vagyunk. Máskülönben leszűkíthetjük a tartományt, ha kizárjuk azokat az elemeket, amelyek a vizsgált elem „rossz" oldalán vannak. Az alábbi ábrán a keresés közbeni helyzetet láthatjuk:

Bármely adott időpontban a *lo* és a *hi* által közrefogott aszimmetrikus tartományban fogunk keresni. Vagyis megköveteljük, hogy *lo=k<hi* legyen. Ha *lo=hi*, akkor nullára szűkítettük a tartomány méretét, tehát biztosak lehetünk benne, hogy az *x* nincs a táblázatban.

Ha *lo<hi*, akkor legalább egy elem van a tartományban. Ekkor *mid* értékét a tartomány közepére állítjuk, majd összehasonlítjuk *x* értékét a táblázat *mid* sorszámú elemének értékével. Ha *x* kisebb, mint ez az elem, akkor a *mid* lesz a tartományon túli legkisebb elem, tehát elvégezzük a *hi=mid* értékadást. Ha *x* a nagyobb, akkor *mid+1* lesz a leszűkített tartomány legkisebb indexe. Végül, ha *x* egyenlő az elemmel, akkor készen vagyunk.

Nincs azzal semmi gond, hogy *mid=(hi+lo)/2*? Addig biztosan nincs, amíg a *hi* és a *lo* jó messze vannak egymástól, de mi van akkor, ha a két érték közel azonos?

A *hi=lo* eset nem okoz gondot, hiszen ekkor már tudjuk, hogy a tartomány üres, és így nem is állítjuk be *mid* értékét. A *hi=lo+2* eset szintén nem okoz gondot, hiszen ekkor a *hi+lo* értéke *2xlo+2* lesz, ami páros szám, így a *(hi+lo)/2* értéke *lo+1* lesz. Mi a helyzet, ha *hi=lo+1*? Ebben az esetben a tartomány egyetlen eleme a *lo* sorszámú elem lesz, tehát jó lenne, ha *(hi+lo)/2=lo* igaz lenne.

Szerencsére ez így is van, mert a *hi+lo* érték mindig pozitív, és ebben az esetben az osztás garantáltan lefelé csonkol. A *(hi+lo)/2* tehát egyenlő azzal, hogy *((lo+1)+lo)/2*, ami másképpen *(2xlo+1)/2*, ami éppen *lo*.

A program ezért így fog kinézni:

```
int *
bsearch(int *t, int n, int x)
{
        int lo = 0, hi = n;
        while (lo < hi) {
                int mid = (lo + hi) / 2;
                if (x < t[mid])
                        hi = mid;
                else if (x > t[mid])
                        lo = mid + 1;
                else
                        return t + mid;
        }
        return NULL;
}
```

Figyeljük meg, hogy a

```
int mid = (lo + hi) / 2;
```

kiértékelése tartalmaz egy olyan osztást, amit léptető műveletként is írhatunk:

```
int mid = (lo + hi) >> 1;
```

Ez bizony igencsak felgyorsítja a programot. Előbb azonban próbáljunk megszabadulni a címszámításoktól, hiszen sok gépen az indexekkel végzett műveletek lassabbak a mutatóműveleteknél. A címszámítások mennyiségét kissé csökkenthetjük, ha a *t+mid* értékét egy helyi változóba mentjük, ahelyett, hogy újra ki kellene számolnunk:

```
int *
bsearch(int *t, int n, int x)
{
        int lo = 0, hi = n;
        while (lo < hi) {
                int mid = (lo + hi) / 2;
                int *p = t + mid;
                if (x < *p)
                        hi = mid;
                else if (x > *p)
                        lo = mid + 1;
                else
                        return p;
        }
        return NULL;
}
```

Tegyük fel, hogy még ennél is jobban le szeretnénk csökkenteni a szükséges címszámítások mennyiségét, azáltal, hogy az egész programban mutatókat használunk az indexek helyett. Első látásra úgy tűnik, hogy nem kell mást tennünk, mint átírni a programot:

```
int *
bsearch(int *t, int n, int x)
{
        int *lo = t, *hi = t + n;
        while (lo < hi) {
                int *mid = (lo + hi) / 2;
                if (x < *mid)
                        hi = mid;
                else if (x > *mid)
                        lo = mid + 1;
                else
                        return mid;
                }
                return NULL;
}
```

És ez majdnem működik is. A gond csak annyi, hogy a

```
mid = (hi + lo) / 2;
```

utasítás érvénytelen, mert két mutatót próbál összeadni. Előbb ki kell számítanunk a *lo* és a *hi* közti távolságot, majd a távolság felét kell hozzáadnunk a *lo* mutatóhoz:

```
mid = lo + (hi - lo) / 2;
```

A *hi-lo* kiszámításában egy osztás is szerepel, de a legtöbb megvalósítás elég okos ahhoz, hogy léptető műveletként valósítsa meg azt. Ahhoz viszont már nem elég okosak, hogy a kettővel osztást is lecseréljék egy léptető műveletre. A fordítóprogram szemszögéből a *hi-lo* értéke akár negatív is lehet, és ebben az esetben a léptetés és a kettővel való osztás eredménye nem ugyanaz. A léptetést ezért nekünk kell megírnunk:

```
mid = lo + (hi - lo) >> 1;
```

Sajnos még ez a megoldás sem helyes. Emlékezzünk vissza, hogy a számtani műveletek megelőzik a léptető műveleteket a műveleti sorrendben! Ezért tehát azt kell írnunk, hogy

```
mid = lo + ((hi - lo) >> 1);
```

és ekkor a kész programunk így néz ki:

```
int *
bsearch(int *t, int n, int x)
{
        int *lo = t, *hi = t + n;
        while (lo < hi) {
                int *mid = lo + ((hi - lo) >> 1);
                if (x < *mid)
                        hi = mid;
                else if (x > *mid)
                        lo = mid + 1;
                else
                        return mid;
        }
        return NULL;
}
```

A bináris keresésekhez egyébként gyakran szimmetrikus korlátokat használnak. A szimmetria miatt az így előállított program valamivel rendezettebb lesz:

```
int *
bsearch(int *t, int n, int x)
{
        int lo = 0, hi = n - 1;
        while (lo <= hi) {
                int mid = (lo + hi) / 2;
                if (x < t[mid])
                        hi = mid - 1;
                else if (x > t[mid])
                        lo = mid + 1;
                else
                        return t + mid;
        }
        return NULL;
}
```

Csakhogy amikor megpróbáljuk „csupa mutatós" formára alakítani ezt a programot, rögtön felmerül egy probléma. A gond az, hogy a *hi* kezdőértéke nem lehet csak úgy a *t+n-1* kifejezés értéke, mert amennyiben *n* nulla, a *t+n-1* érvénytelen cím lesz! Ha tehát a mutatókat akarjuk használni, akkor azt is külön ellenőriznünk kell, nehogy *n=0* legyen. Ez ismét azt sugallja, hogy jobban járunk az aszimmetrikus korlátokkal.

4.1. gyakorlat

Ha a `short` típusú érték is `37` lesz, amikor a `long` típusú értékbe `37` kerül, az azt sugallja, hogy a `short` érték ugyanazt a memóriahelyet foglalja el, mint amit a `long` érték használ a `37` értékes bitjeinek tárolására. Az is egy eshetőség, hogy a `long` és a `short` típusok megegyeznek, de ekkor egy nagyon ritka C megvalósítással állunk szemben. Sokkal valószínűbb, hogy a `long` alacsony helyiértékű bitjei a `long` azon részén találhatók, amelyik a `short` értékkel osztozik a tárhelyen. Ez általában a legalacsonyabb memóriacímen található rész. A következtetés ekkor az, hogy valószínűleg egy olyan gépen vagyunk, ami a legkisebb helyiértékű bájtot a legkisebb memóriacímen tárolja (*little-endian*). Hasonlóképpen, ha a `short` értéke `0`

lesz, amikor a `long` értékbe `37` kerül, akkor valószínűleg olyan gépen dolgozunk, ami fordított bájtsorrendű (*big-endian*) számtárolást alkalmaz.

4.2. gyakorlat

Léteznek olyan C megvalósítások, amelyeken két változata is van a `printf` függvénynek. Ezek közül az egyikben megtaláljuk a `%e`, `%f` és `%g` lebegőpontos formátumok megvalósítását, a másikban nem. Ez a kettősség a könyvtárban helyet takarít meg az olyan programokban, amelyek nem használnak lebegőpontos műveleteket, mert az ilyen programok használhatják a `printf` függvénynek azt a változatát, amelyik nem támogatja azokat.

Vannak olyan rendszerek, amelyeken a programozónak egyértelműen meg kell mondania a linkernek, hogy a program használ-e lebegőpontos műveleteket vagy sem. Más rendszerek ezt automatikusan próbálják eldönteni úgy, hogy a fordítóprogramnak kell megmondani a linkernek, hogy vannak-e lebegőpontos műveletek a programban.

Ebben a programban nincsenek lebegőpontos műveletek! Nem része a `math.h`, és nincs benne az `sqrt` bevezetése, így a fordítóprogram nem tudhatja, hogy az `sqrt` lebegőpontos függvény. Még lebegőpontos értéket sem adtunk át az `sqrt` függvénynek. A fordító tehát joggal jelentheti a linkernek, hogy nem lebegőpontos programról van szó!

Mi a helyzet az `sqrt` könyvtári függvénnyel? Az a tény, hogy az `sqrt` függvényt a könyvtárból hívtuk be, már biztosan elég bizonyíték arra, hogy a program végez lebegőpontos műveleteket. Ez persze igaz, de lehet, hogy a linker már azelőtt eldöntötte, hogy a `printf` melyik változatát fogja használni, hogy lehívta volna az `sqrt` függvényt a könyvtárból.

5.1. gyakorlat

Ha egy program valamilyen hiba miatt ér véget, akkor lehet, hogy már nem lesz lehetősége kiüríteni a kimenetét tartalmazó átmeneti tárolóit. Előfordulhat tehát, hogy a program már előállított valamilyen

kimenetet, ami ott csücsül a memóriában, a kiírása viszont még nem történt meg. Vannak olyan rendszerek, ahol ez a kimenet akár több oldalra is felduzzadhat.

Ha így veszik el a kimenet, az nagyon megtévesztő lehet a hibakeresés folyamán, mert úgy tűnhet, hogy a program sokkal előbb elromlott, mint valójában. A megoldás ilyen esetekben az, ha kikényszerítjük, hogy ne legyen késleltetett a kiírás. Az ehhez szükséges varázsige rendszerről rendszerre változó, de általában valahogy így néz ki:

```
setbuf(stdout, (char *) 0);
```

Ezt az utasítást még azelőtt kell végrehajtanunk, hogy bármit az `stdout` kimenetre írnánk, beleértve a `printf` hívásait is. A legmegfelelőbb hely az utasítás számára a főprogram első sora.

5.2. gyakorlat

A függvényhívások sok időt emészthetnek fel, ezért a `getchar` megvalósítása gyakran makróként történik. A makró meghatározása az `stdio.h` állományban található, tehát ha nem foglaljuk bele az `stdio.h` állományt a programba, akkor a fordító számára ismeretlen lesz a `getchar` meghatározása. Így azt fogja feltételezni, hogy a `getchar` egy egész értéket visszaadó függvény.

Számos C megvalósításnál találunk egy `getchar` függvényt a könyvtárban, részben az ilyen figyelmetlenségek kivédése végett, részben pedig kényelmi szempontból azok számára, akiknek szükségük lehet a `getchar` címére. Ha tehát nem foglaljuk bele a programba az `stdio.h` állományt, azzal azt érjük el, hogy a `getchar` makrót lecseréljük a függvény változatra. A program a függvényhívással járó többletmunka miatt lelassul. Pontosan ugyanez az érvelés igaz a `putchar` esetében is.

6.1. gyakorlat

A `max` részére átadott értékek felhasználására kétszer is sor kerülhet. Először az összehasonlításban, másodszor pedig eredményként. Ezért fontos, hogy az átadott értékeket egy-egy átmeneti változóba mentsük.

Sajnos a C kifejezéseken belül nem tudunk közvetlenül bevezetni átmeneti változókat, ezért ha a max makrót egy kifejezésen belül fogjuk használni, akkor máshol kell bevezetnünk a változókat, valószínűleg a makrómeghatározás közelében, és nem annak részeként. Az átmeneti változókat statikusként vezetjük be, hogy elkerüljük a névütközéseket, ha a max makrót több fájlban is felhasználjuk. A meghatározásokat feltehetően egy fejlécállományba tennénk:

```
static int max_temp1, max_temp2;
#define max(p,q) (max_temp1=(p),max_temp2=(q), \
    max_temp1>max_temp2? max_temp1: max_temp2)
```

Ez egészen addig működni is fog, amíg a max hívásait nem ágyazzuk egymásba. Annak megvalósítása, hogy a program az utóbbi esetben is működjön, talán nem is lehetséges.

6.2. gyakorlat

Egy lehetséges eset, ha az x egy típus neve, amit például így határoztunk meg:

```
typedef int x;
```

Ebben az esetben az

```
(x) ((x)-1)
```

kifejezés egyenértékű az

```
(int) ((int)-1)
```

kifejezéssel, ami fogja a -1 állandót és kétszer egész típusúra alakítja. Ugyanezt a hatást érhetjük el, ha az előfeldolgozóval határozzuk meg az x típust:

```
#define x int
```

Felmerülhet az a lehetőség is, hogy x függvénymutató. Emlékezzünk vissza, hogy ha függvénymutatót használunk egy olyan helyen, ahol

függvényre van szükség, akkor a program automatikusan meghívja azt a függvényt, amire a függvénymutató irányul, és azt fogja használni a mutató helyett. Tehát úgy is értelmezhetjük ezt a kifejezést, hogy meghívjuk azt a függvényt, amire az x mutat, és aminek az `(x)-1` lesz a paramétere. Ahhoz, hogy az `(x)-1` tényleg egy szabályos kifejezés legyen, fontos, hogy az x egy függvénymutatókat tartalmazó tömb egyik elemére mutasson.

Mi lesz az x teljes típusa? Az egyszerűség kedvéért legyen `T` az x típusa, hogy így vezethessük be az x-et:

```
T x;
```

Úgy tűnik, hogy az x minden bizonnyal egy olyan függvényre mutat, aminek a paramétere `T` típusú. Ez kissé megnehezíti a `T` meghatározását. A kézenfekvő megoldás nem működik:

```
typedef void (*T)(T);
```

mert a `T` meghatározása egészen addig nem létezik, amíg fel nem dolgozzuk a deklarációt! Nem feltétlenül kell azonban ragaszkodnunk ahhoz, hogy az x olyan függvényre mutasson, amelynek a paramétere `T` típusú. A paraméter típusát illetően elegendő azt kikötnünk, hogy azt át lehessen alakítani `T` típusúra. Nevezetesen, a `void *` megoldás biztosan működni fog:

```
typedef void (*T)(void *);
```

Ebből a gyakorlatból az a legnagyobb tanulság, hogy nem mindig szabad egyből hibásnak titulálni egy furcsa kinézetű kifejezést.

7.1. gyakorlat

Vannak olyan számítógépek, amelyek egyedi memóriacímet rendelnek minden karakterhez, míg mások memóriaszavakat használnak. A memóriaszavas gépeken sokszor gondot okoz a karakterek hatékony feldolgozása, hiszen egy karakter kiolvasásához előbb ki kell olvasni a teljes memóriaszót, majd el kell dobni a felesleges részeit.

Mivel sokkal hatékonyabban dolgoznak a karakterekkel, az elmúlt években a karakteres címzésű gépek lényegesen népszerűbbek lettek a memóriaszavas címzésű gépeknél. Az egészekkel végzett műveletek szempontjából azonban még a karakteres címzésű gépek számára is fontos maradt a memóriaszó fogalma. Mivel a karakterek tárolása egymást követő memóriacímeken történik, az egymást követő memóriaszavak címei közti különbség megegyezik a memóriaszavakban található karakterek számával.

A hardver számára sokkal egyszerűbb átalakítani a karaktercímeket memóriaszócímekké, ha a memóriaszóban található karakterek száma a 2 hatványa, hiszen a szorzás a 2 egy hatványával nem más, mint léptetés. Ezért logikus, ha egy memóriaszó karakterben kifejezett hossza a 2 hatványa.

De miért nincsenek 64 bites egészek? Bizony néha határozottan jól jönnének. A lebegőpontos képességekkel rendelkező gépeken azonban nincs nagy jelentőségük, és a megvalósításuk elég költséges ahhoz képest, hogy milyen ritkán van ténylegesen szükségünk ekkora pontosságra az egész számoknál. Alkalmi használatra pedig szoftveresen is elég hatékonyan utánozhatjuk a 64 bites (vagy nagyobb) egészeket.

7.2. gyakorlat

Feltehetjük, hogy a számjegyek egymással határosak a gép jelsorrendjében, hiszen minden modern gép így viselkedik, az ANSI C pedig elő is írja ezt. A nehézséget tehát az jelenti, hogy elkerüljük a túlcsordulást a köztes eredményeknél, habár a végeredmény a korlátokon belül marad.

Ahogy a `printnum` esetében, itt is gond lehet, ha a legkisebb negatív és a legnagyobb pozitív hosszú egész értéke különböző. Pontosabban akkor kerülhetünk bajba, ha az értéket pozitívként kapjuk meg, majd később az ellentettjét vesszük. Ezzel a legkisebb negatív értéknél sok gépen túlcsordulást idézünk elő.

Az alábbi változat úgy kerüli el ezeket a túlcsordulást, hogy csak negatív (vagy nulla) értékeket használ eredményének meghatározásához:

```
long
atol(char *s)
{
        long r = 0;
        int neg = 0;

        switch (*s) {
        case '-':
                neg = 1;
                /* nincs break */
        case '+':
                s++;
                break;
        }

        while (*s >= '0' && *s <= '9') {
                int n = *s++ - '0';
                if (neg)
                        n = -n;
                r = r * 10 + n;
        }
        return r;
}
```

FÜGGELÉK

PRINTF, VARARGS és STDARG

Ez a függelék három olyan általános C szolgáltatást mutat be, amelyekkel kapcsolatban gyakran merülnek fel félreértések. Ezek a `printf` családba tartozó könyvtári függvények, valamint a `varargs` és a `stdarg`, amelyek segítségével hívásról hívásra más-más számú, illetve típusú paraméterrel rendelkező függvényeket írhatunk. Sokszor találkozom olyan programokkal, amelyek olyan tulajdonságait használják a `printf` függvénynek, amelyek már évekkel ezelőtt eltűntek a nyelvből. Gyakran látok olyan programokat is, amelyeket mindenféle hordozhatatlan foltozgatással tűzdeltek tele, hogy elérjék azt, amit a `varargs` vagy a `stdarg` segítségével szép tisztán megoldhattak volna.

A.1. A printf család

Az alábbi program a 0. fejezet első C példájára hasonlít:

```
#include <stdio.h>

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

A program ezt írja ki:

```
Hello world
```

Ezt egy újsor karakter (`\n`) követi. A `printf` első paramétere egy *formázó*, vagyis egy karakterlánc, ami meghatározza a kimenet formáját. A szokásos C hagyományt követve ezt a karakterláncot mindig egy null karakter (`\0`) zárja le. Ha állandóként adjuk meg a karakterláncot, a lezáró karakter garantáltan a helyén lesz.

A `printf` függvény átmásolja a formázó karaktereit a szabványos kimenetre, egészen addig, amíg el nem éri a formázó végét, vagy bele nem ütközik egy `%` karakterbe. Ha a `printf` `%` karakterrel találkozik, akkor azt nem írja ki, hanem megnézi, hogy milyen utasítást tartogat számára a `%` jelet követő néhány karakter, majd ennek megfelelően alakítja át a következő paraméterét. A függvény az így átalakított paramétert írja ki a `%` jel és az azt követő néhány karakter helyett. Mivel a fenti példában található formázóban nincs `%` jel, a `printf` kimenete pontosan meg fog egyezni a formázóban megadott karakterekkel. A formázó és a benne található paraméterek határozzák meg a kimenet minden egyes karakterét, *beleértve a minden sort lezáró újsor karaktert is.*

A `printf` függvénnyel rokon függvények az `fprintf` és az `sprintf`. Míg a `printf` a szabványos kimenetre ír, az `fprintf` bármilyen kimeneti állományba képes írni. Az állományt a függvénynek átadott első paraméter adja meg, ami egy `FILE` mutató típusú érték. A

```
printf(stuff);
```

utasítás és az

```
fprintf(stdout, stuff);
```

utasítás tehát ugyanazt jelenti.

Az `sprintf` függvényt akkor használjuk, amikor a kimenet nem fájlba, hanem valahová máshová kerül. Az `sprintf` első paramétere egy karaktervektorra mutat, ahová a `sprintf` a kimenetét helyezheti. A programozó felel azért, hogy ez a tömb elég nagy legyen ahhoz, hogy beleférjen a `sprintf` által előállított kimenet. A függvény többi paramétere megegyezik a `printf` paramétereivel. Az `sprintf` kimenetét mindig egy null karakter zárja le. A null karaktert ezen kívül csak úgy lehet megjeleníteni, ha a `%c` formázóelemmel kifejezetten elrendeljük a kiírását.

Mindhárom függvény az átvitt karakterek számával tér vissza. Az `sprintf` esetében ez a szám nem tartalmazza a kimenetet lezáró null karaktert. Ha a `printf` vagy az `fprintf` írás közben I/O hibába ütközik, akkor negatív értéket ad vissza. Ebben az esetben lehetetlen megállapítani a sikeresen átvitt karakterek számát. Mivel az `sprintf` nem végez bemeneti-kimeneti műveleteket, sohasem szabadna negatív értéket visszaadnia (bár semmi kétség, hogy lesz olyan megvalósítás, ami erre is talál majd valamilyen okot).

Mivel a formázó karakterlánc határozza meg a többi paraméter típusát, és mivel azt futás közben is felépíthetjük, a C megvalósításnak rendkívül nehéz dolga van, amikor ellenőrzi, hogy a `printf` paramétereinek típusa megfelelő-e. Ha tehát azt írjuk, hogy

```
printf("%d\n", 0.1);
```

vagy azt, hogy

```
printf("%g\n", 2);
```

azzal értelmetlen adatokat állítunk elő, és elég valószínű, hogy ezt már csak akkor fogjuk észrevenni, amikor ténylegesen futtatjuk a programot.

A legtöbb megvalósítás figyelmét ez is elkerüli:

```
fprintf("error\n");
```

A programozó itt az `fprintf` segítségével a `stderr` kimenetre szeretett volna írni, csakhogy elfelejtette megemlíteni a `stderr` kimenetet az utasításban. Ennek valószínűleg az lesz a vége, hogy a program kiírja a memória teljes tartalmát, mivel az `fprintf` fájlként értelmezi a formázó karakterláncot.

Egyszerű formázótípusok

Minden formázóelemet egy `%` jel vezet be, amit mindig, de nem mindig azonnal, egy *formátumkód* elnevezésű karakter követ. A `%` jel és a formátumkód között más karakterek is szerepelhetnek, amelyek befolyásolhatják az átalakítást a később bemutatott módokon. A formátumkód mindig a formázóelem végén található.

A leggyakoribb formázóelem minden bizonnyal a `%d`, ami tizedes formában ír ki egy egész számot. A

```
printf("2 + 2 = %d\n", 2 + 2);
```

utasítás például azt írja ki, hogy

```
2 + 2 = 4
```

amit egy újsor karakter fog követni (a további példákban nem említjük meg külön az újsor karaktert a kimenetben).

A `%d` formázóval egy egész szám kiírását kérjük. Kell, hogy legyen tehát egy hozzá tartozó egész paraméter. Amikor a formázó a kimenetre íródik, a `%d` helyére az egész érték kerül, előtte és utána szóközök nélkül. Ha negatív egészről van szó, akkor a kimenet első karaktere egy – jel lesz. A `%u` formázóelem `unsigned` típusúként kezeli az egész értékeket. Például a

```
printf("%u\n", -37);
```

utasítás azt írja ki, hogy

```
4294967259
```

a 32 bites egész értékeket használó gépeken.

Emlékezzünk vissza, hogy a `char` és a `short` típusú paramétereket automatikusan `int` típusúra bővíti a program. Ez meglepő eredményhez vezethet az olyan gépeken, amelyek a `char` értékeket előjelesként kezelik. Egy ilyen gépen például a

```
char c;

c = -37;
printf("%u\n", c);
```

programrészlet azt írja ki, hogy

```
4294967259
```

mivel a `char` típusú `-37` értéket átalakítja `int` típusú `-37` értékké. Ezt a problémát úgy kerülhetjük el, ha az `%u` formázóelem használatát az `unsigned` típusú értékekre korlátozzuk.

A `%o`, `%x` és a `%X` formázóelemek 8-as vagy 16-os számrendszerben írják ki az egész értékeket. A `%o` elem egy oktális (nyolcas számrendszerbeli) érték kiírására utasít, a `%x` és a `%X` elemek pedig hexadecimális (tizenhatos számrendszerbeli) értékek kiírását eredményezik. A `%x` és a `%X` között mindössze annyi a különbség, hogy a `%x` elem az `a`, `b`, `c`, `d`, `e` és `f` betűket használja a 10-től 15-ig terjedő számjegyek kiírásához, míg a `%X` elem az `A`, `B`, `C`, `D`, `E` és `F` betűket használja erre a célra. Az oktális és hexadecimális értékek mindig előjel nélküliek.

Az

```
int n = 108;
printf("%d decimal = %o octal = %x hex\n",
        ➥ n, n, n);
```

például azt írja ki, hogy

```
108 decimal = 154 octal = 6c hex
```

Ha a %X elemet használjuk a %x elem helyett, akkor a kimenet így fog kinézni:

```
108 decimal = 154 octal = 6C hex
```

A %s formázóelem karakterláncokat ír ki. A hozzá tartozó paraméternek egy karaktermutatónak kell lennie. A karakterek kiírása a paraméter által megcímzett karakterrel kezdődik, és addig tart, amíg a program el nem ér egy null karaktert ('\0'). Egy példa a %s formázóelem használatára:

```
printf("There %s %d item%s in the list.\n",
          n!=1? "are": "is", n, n!=1? "s": "");
```

Az első %s helyére vagy az is vagy az are szó, a második %s helyére pedig vagy egy s betű, vagy egy üres karakterlánc kerül. Ha tehát n értéke 37, akkor a kimenet ez lesz:

```
There are 37 items in the list.
```

De ha n értéke 1, a program ezt fogja kiírni:

```
There is 1 item in the list.
```

Ha egy karakterláncot a %s formázóelemmel írunk ki, akkor azt mindig le *kell* zárnunk egy null karakterrel ('\0'). (Ez alól egy kivétel van, amiről később ejtünk szót.) A printf ugyanis csak így képes megtalálni a karakterlánc végét. Ha olyan karakterláncot teszünk a %s elem mellé, ami nincs megfelelően lezárva, akkor a printf egészen addig folytatja a karakterek kiírását, amíg össze nem találkozik egy '\0' értékkel a memóriában. A kimenet így bizony elég hosszúra nyúlhat! Mivel a %s formázóelem a hozzá tartozó paraméter összes karakterét kiírja, a

```
printf(s);
```

és a

```
printf("%s", s)
```

két különböző dolgot jelent. Az első példa az s karakterláncban található % jeleket formátumkódok bevezetéseként fogja értelmezni. Ha a %% jelen kívül más formátumkód is lesz a karakterláncban, az gondot okozhat, hiszen hiányozni fog a hozzá tartozó paraméter. A második példa bármely null karakterrel lezárt karakterláncot kiír.

Mivel egy NULL mutató nem mutat sehová, abban is biztosak lehetünk, hogy egy karakterláncra sem mutat. A

```
printf("%s\n", NULL);
```

utasítás tehát megjósolhatatlan eredménnyel járhat. A 3.5. részben erről részletesen szót ejtettünk.

A %c formázóelem egyetlen karaktert ír ki. A

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

megfelel annak, hogy

```
putchar(c);
```

de annyival rugalmasabb nála, hogy a c karakter értékét egy tágabb összefüggésbe helyezhetjük vele. A %c formázóelemhez tartozó paraméter egy egész típusú érték, amit a kiíráshoz karakter típusúvá alakít a program. A

```
printf("The decimal equivalent of '%c' is %d\n",
        '*', '*');
```

utasítás például ezt írja ki:

```
The decimal equivalent of '*' is 42
```

Három olyan formázóelem is van, ami lebegőpontos értékeket ír ki: a %g, a %f és a %e. Ha a lebegőpontos kimenet nem oszlopokba kerül, akkor a %g formázóelem használata a legüdvözítőbb, ami a megfelelő (float vagy double típusú) érték 6 értékes jegyét írja ki úgy, hogy eltávolítja a szám végéről a felesleges nullákat. Ha tehát belefoglaljuk a programunkba a math.h állományt, akkor a

```
printf("Pi = %g\n", 4 * atan(1.0));
```

utasítás azt fogja kiírni, hogy

```
Pi = 3.14159
```

a

```
printf("%g %g %g %g %g\n",
        1/1.0, 1/2.0, 1/3.0, 1/4.0, 0.0);
```

utasítás pedig azt, hogy

```
1 0.5 0.333333 0.25 0
```

A bevezető nullák nem számítanak bele a pontosságba, így hat darab 3-as lesz a `0.333333` értékben. A kiírt értékeket nem csonkolja, hanem kerekíti a program. A

```
printf("%g\n", 2.0 / 3.0);
```

utasítás ezt írja ki:

```
0,666667
```

Amennyiben az érték nagyobb, mint `999999`, annak kiírása az imént leírt formában csak egy hibás érték vagy hatnál több értékes számjegy megjelenítésével lehetséges. A `%g` formázóelem ezt a kérdést úgy oldja meg, hogy az ilyen értékeket tudományos alakban írja ki. A

```
printf("%g\n", 123456789.0);
```

utasítás ezt írja ki:

```
1.23457e+08
```

Az értéket itt is hat értékes jegyre kerekíti a program.

Ahogy az érték nagyságrendje egyre kisebb lesz, az ábrázolásához szükséges számjegyek száma kényelmetlenül megnő.
Elég bután néz ki például, ha a $\pi \times 10^{-10}$ értékét úgy írjuk le, hogy `0,000000000314159`. Sokkal tömörebb és jobban olvasható, ha azt írjuk, hogy `3,14159e-10`. Ha a kitevő pontosan `-4`, akkor a két alak hossza megegyezik (a `0,000314159` éppen annyi helyet foglal el, mint a `3,14159e-04`). A `%g` formázóelem ezért csak akkor kezdi el használni a tudományos jelölésmódot, ha a kitevő értéke `-5` vagy kisebb lesz. Tehát a

```
printf("%g %g %g\n", 3.14159e-3, 3.14159e-4,
    ➥ 3.14159e-5);
```

utasítás azt írja ki, hogy

```
0.00314159 0.000314159 3.14159e-05
```

A `%e` formázóelem mindig következetesen kiírja a kitevőt a lebegőpontos értékekhez. A π tehát `%e` alakra hozva `3,141593e+00` lesz. A `%e` formázóelem a *tizedesvessző után* ír ki hat számjegyet, nem pedig összesen hat értékes jegyet.

A `%f` formázóelem kikényszeríti, hogy az értéket a kitevő *nélkül* írja ki a program. A π tehát úgy jelenik meg, hogy `3,141593`. A `%f` formázóelem szintén a tizedesvessző (tizedespont) után ír ki hat jegyet. Egy nagyon kis érték ezért nullaként jelenhet meg, még akkor is, ha nem az, egy nagyon nagy érték pedig sok-sok számjegyből állhat. A

```
printf("%f\n", 1e38);
```

utasítás ezt írja ki:

```
100000000000000000000000000000000000000.000000
```

Mivel az itt kiírt számjegyek száma meghaladja a legtöbb hardver pontosságát, a végeredmény gépenként változó lehet.

A `%E` és a `%G` formázóelemek éppen úgy működnek, mint a kisbetűs megfelelőik, azzal a különbséggel, hogy ha az `e` helyett az E formázót használjuk, akkor megjelenik a kitevő.

A `%%` formázóelem egy `%` karaktert ír ki. Abban a tekintetben tehát egyedülálló, hogy *nem tartozik hozzá* paraméter. A

```
printf("%%d prints a decimal value\n");
```

utasítás tehát ezt írja ki:

```
%d prints a decimal value
```

Módosítók

A `printf` elfogad még olyan karaktereket is, amelyek a formázóelemek jelentését módosítják. Ezek a karakterek a `%` jel és a következő formázóelem közé kerülnek.

Az egészek háromféle méretűek lehetnek: vannak rövid (`short`), hosszú (`long`) és sima (`int`) egészek. Ha egy függvénynek, beleértve a `printf` függvényt is, egy rövid egész értéket adunk át, akkor azt a program automatikusan kiegészíti sima egésszé, de azt még nem tudjuk, hogyan lehet megmondani a `printf` függvénynek, hogy egy `long` típusú egész értékre számítson. Ezt úgy érhetjük el, hogy egy `l` karaktert teszünk közvetlenül a formátumkód elé, létrehozva ezzel az új `%ld`, `%lo`, `%lx` és `%lu` formátumkódokat. Ezek a módosított kódok pontosan úgy viselkednek, mint az eredeti megfelelőik, azzal a különbséggel, hogy mindegyikhez egy `long` típusú egész értéknek kell tartoznia. A `%lu` formázóelem úgy írja ki a hosszú egészeket, mintha azok előjel nélküliek lennének, még azon C megvalósításokban is, amelyek közvetlenül nem támogatják a `long unsigned` típusú értékeket. Az `l` módosítónak csak akkor van értelme, ha egész formátumkódokkal használjuk.

Sok megvalósítás az `int` és a `long` típusú értékeket ugyanolyan pontossággal tárolja. Az ilyen gépeken egészen addig nem derül fény rá, hogy lefelejtettük az `l` módosítót, amíg a program át nem kerül egy olyan gépre, ahol az `int` és `long` típusok tényleg különböznek. Például a

```
long size;
. . .
printf("%d\n", size);
```

néhány gépen működni fog, míg másokon nem.

A *szélességmódosító* leegyszerűsíti az értékek rögzített szélességű oszlopokba írását. Helye a `%` jel és a következő formátumkód között van, és azt adja meg, hogy *legalább* hány karaktert kell kiírnia a módosított formázóelemnek. Amennyiben a kiírt érték nem tölti ki teljesen az oszlopot, akkor az érték bal oldalára üres közök kerülnek, hogy

meglegyen a kívánt szélesség. Ha a kiírandó érték túl nagy az oszlophoz képest, akkor a program annak megfelelően kiszélesíti az oszlopot. *A szélességmódosító soha nem csonkolja a oszlopokat.* Ha arra használjuk a szélességmódosítót, hogy oszlopokba rendezzünk számokat, akkor ha egy érték túl nagy a saját oszlopának, akkor jobbra tolja a sorban található többi értéket.

Ez a programrészlet

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

a következőket írja ki:

```
0   0 *
1   1 *
2   4 *
3   9 *
4  16 *
5  25 *
6  36 *
7  49 *
8  64 *
9  81 *
10 100 *
```

A * ebben a példában a sorok végét jelöli. A `100` túl nagy ahhoz, hogy elférjen két karakternyi helyen, ezért a hozzá tartozó oszlop szélessége megnő, a sor többi része pedig jobbra tolódik.

A szélességmódosító minden formátumkódra hatással van, még a `%%` kódra is. A

```
printf("%8%\n");
```

utasítás tehát egy % jelet ír ki jobbra igazítva egy nyolc karakter széles oszlopba. Más szóval, a fenti utasítás hét szóközt és azt követően egy % jelet ír ki.

A *pontosságmódosító* azt szabja meg, hogy hány számjegy jelenjen meg egy adott szám kiírásakor, vagy hogy hány karakter jelenjen meg egy karakterláncból. Ez a módosító egy tizedespontból és az azt követő számjegyekből áll. A helye a formátumkód és a hosszúságmódosító előtt, de a `%` jel és a szélességmódosító után van. A pontosságmódosító pontos jelentése az adott formátumkódtól függ:

- A `%d`, `%o`, `%x` és `%u` egész formázóelemeknél azt adja meg, hogy *legalább* hány számjegyet kell kiírni. Ha az érték kiírásához nincs szükség annyi számjegyre, akkor a szám elejét nullákkal egészíti ki. A

```
printf("%.2d/%.2d/%.4d\n", 7, 14, 1789);
```

utasítás tehát azt írja ki, hogy

```
07/14/1789
```

- A `%e`, `%E` és `%f` formázóelemeknél a pontosság azt adja meg, hogy hány számjegy álljon a tizedesvessző (tizedespont) után. Hacsak a jelzőkarakterek (amelyekről rövidesen szó lesz) máshogy nem rendelkeznek, a tizedespont csak akkor jelenik meg, ha a pontosság nagyobb, mint nulla. Ha tehát belefoglaljuk a `math.h` állományt a programba, akkor a

```
double pi;
pi = 4 * atan(1.0);
printf("%.0f %.1f %.2f %.3f %.6f %.10f\n",
          pi, pi, pi, pi, pi, pi);
printf("%.0e %.1e %.2e %.10e\n",
          pi, pi, pi, pi, pi, pi);
```

programrészlet azt írja ki, hogy

```
3 3.1 3.14 3.142 3.141593 3.1415926536
3e+00 3.1e+00 3.14e+00 3.1415926536e+00
```

- A `%g` és `%G` formázóelemeknél a pontosság a kiírandó *értékes számjegyek* számát határozza meg. Hacsak a jelzőkarakterek máshogy nem rendelkeznek, az értéktelen nullákat eltávolítja a program, a tizedespontot pedig törli, ha egyetlen számjegy sem követi azt. A

```
printf("%.1g %.2g %.4g %.8g\n",
          10/3.0, 10/3.0, 10/3.0, 10/3.0);
```

utasítás tehát ezt írja ki:

```
3 3.3 3.333 3.3333333
```

- A `%s` formázóelemeknél a pontosság azt adja meg, hogy az adott karakterláncból hány karaktert kell kiírni. Ha a karakterláncban nincs annyi karakter, hogy kielégítse a megadott pontosságot, akkor a kimenet rövidebb lesz. Ebben az esetben, ha szükséges, a szélességmódosítóval hosszabbíthatjuk meg a kimenetet. Vannak olyan rendszerek, amelyek egy 14 karakteres tömbben tárolják a fájlneveket. Ha a fájlnév kevesebb, mint 14 karakterből áll, akkor a tömb többi részébe null karakterek kerülnek, de ha a fájlnév hossza a lehető legnagyobb, akkor a tömböt nem zárja le null karakter. Egy ilyen nevet így írhatunk ki:

```
char name[14];
. . .
printf("... %.14s ...", ... , name, ...);
```

  Így biztosak lehetünk benne, hogy a program helyesen fogja kiírni a fájlnevet, attól függetlenül, hogy az milyen hosszú. Ha a `%14.14s` formázóelemet használjuk, akkor garantáltan 14 karaktert ír ki a program, a név hosszától függetlenül (ha szükséges, a név bal oldalára üres karakterek kerülnek, hogy meglegyen a 14 karakter; de azt is mindjárt megnézzük, hogyan kell az üres karaktereket a jobb oldalra helyezni).

- A pontosságot a `c` és `%` formázóelemeknél figyelmen kívül hagyja a program.

Jelzőkarakterek

A `%` jel és az oszlopszélesség között feltűnhetnek olyan karakterek, amelyek kissé változtatnak a formázóelem hatásán. Ezeket *jelzőkarakereknek* (jelző, flag) nevezzük, és a jelentésük a következő:

- A `-` jelzőnek csak akkor van értelme, ha megadtunk egy szélességet (mert az érték igazítására csak akkor van szükség, ha az oszlop szélessége meghaladja az érték szélességét). Ebben az esetben az érték igazításához használt üres közök az érték jobb, és nem a bal oldalára kerülnek.

Amikor rögzített szélességű oszlopokba írunk karakterláncokat, általában jobban néz ki, ha balra igazítjuk azokat. A `%14s` formázó tehát valószínűleg tévesztés eredménye, és helyette a `%-14s` formátum lenne a helyes. Az előző példa eredménye ezért valószínűleg így szebben mutat:

```
char name[14];
. . .
printf("... %-14s ...", ... , name, ...);
```

- A + jelző gondoskodik róla, hogy minden kiírt számérték első karaktere egy előjel legyen. A nem negatív számok első karaktere tehát egy + jel lesz. A – jelzőkarakterhez semmi köze. A

```
printf("%+d %+d %+d\n", -5, 0, 5);
```

utasítás azt írja ki, hogy

```
-5 +0 +5
```

Ha szóközt használunk jelzőkarakterként, akkor egy darab szóköz fog megjelenni minden olyan kiírt számérték előtt, amelynek első karaktere nem előjel. Ez szerfelett hasznos, ha + előjelek nélkül, balra igazított oszlopokban szeretnénk megjeleníteni számokat. Ha a + és a szóköz jelzőt ugyanannál a formázóelemnél használjuk, akkor a + jelzőkarakteré az elsőbbség. Például a

```
int i;
for (i = -3; i <= 3; i++)
        printf("% d\n", i);
```

programrészlet azt írja ki, hogy

```
-3
-2
-1
 0
 1
 2
 3
```

A `% e` és a `%+e` formázóelemek sokkal megfelelőbbek a tudományos jelölésmódú számok oszlopokba írásához, mint a sima `%e` formázó. Az előjel (vagy szóköz) jelenléte az összes kimeneti értéknél garantálja, hogy a tizedespontok pontosan egymás alá kerülnek.

Például a

```
double x;

for (x = -3; x <= 3; x++)
        printf("% e %+e %e\n", x, x, x);
```

programrészlet azt írja ki, hogy

```
-3.000000e+00    -3.000000e+00    -3.000000e+00
-2.000000e+00    -2.000000e+00    -2.000000e+00
-1.000000e+00    -1.000000e+00    -1.000000e+00
 0.000000e+00    +0.000000e+00     0.000000e+00
 1.000000e+00    +1.000000e+00     1.000000e+00
 2.000000e+00    +2.000000e+00     2.000000e+00
 3.000000e+00    +3.000000e+00     3.000000e+00
```

A `%e` formázóelemmel kiírt oszlopban a tizedespontok nincsenek egy vonalban, a másik kettőben viszont igen.

- A # jelző a számértékek formátumát alakítja át egy kissé, az adott formázóelemtől függően. A `%o` formázóelem esetében a pontosságot növeli meg, ha szükséges, de csak éppen annyira, hogy az első kiírt karakter a `0` legyen. Ennek az a lényege, hogy az oktális számokat abban az alakban lehessen kiírni, ahogy azt a legtöbb C programozó megszokta. A `%#o` nem ugyanaz, mint a `0%o`, mivel az utóbbi a nullát úgy írja ki, hogy `00`. Hasonlóképpen, a `%#x` és a `%#X` formázóelemek hatására az értékek előtt megjelenik egy `0x` illetve egy `0X`.
  A # jelzőkarakternek kettős hatása van a lebegőpontos formátumokra. Egyrészt mindig megjeleníti a tizedespontot, még akkor is, ha nem áll utána semmi. Másrészt megakadályozza, hogy a `%g` és `%G` formázóelemek töröljék az értékek végén lévő nullákat. Például a

```
printf("%.0f %#.0f %g %#g\n",
            3.0, 3.0, 3.0, 3.0);
```

utasítás azt írja ki, hogy

```
3 3. 3 3.00000
```

A szóköz és a + jelzők kivételével a jelzőkarakterek mind függetlenek egymástól.

Változó oszlopszélesség és pontosság

Néhány C programban gondosan meghatározzák egy karaktertömb hosszát szimbolikus állandóként, de amikor a kiírásra kerül a sor, akkor egy egész állandóval adják meg a szélességet. Korábbi példánkat tehát kissé ostoba módon így is átírhatnánk:

```
#define NAMESIZE 14
char name[NAMESIZE];
. . .
printf("... %.14s ...", ... , name, ...);
```

A NAMESIZE meghatározásával az volt a célunk, hogy csak egy helyen kelljen megemlítenünk a 14-es értéket. Ha valaki később módosítja a NAMESIZE értékét, valószínűleg nem fogja átnézni az összes printf függvényt, hogy megváltoztassa az értékeket. A NAMESIZE állandót viszont közvetlenül nem használhatjuk a printf függvény hívásakor. Az alábbi utasítás

```
printf("... %.NAMESIZE ...", ... , name, ...);
```

nem fog működni, mert az előfeldolgozó nem fér hozzá a karakterláncok belsejéhez.

A printf függvény ezért egy közvetett módot kínál az oszlopszélesség és a pontosság beállítására. Ehhez le kell cserélnünk az oszlopszélességet vagy a pontosságot, vagy mindkettőt egy * jelre. Ilyenkor a printf a tényleges értéke(ke)t a paraméterlistájából olvassa ki, még azelőtt, hogy kiolvasná a kiírandó értéket. A fenti példát tehát úgy írhatnánk, hogy

```
printf("... %.*s ...", ... , NAMESIZE,
        ➥ name, ...);
```

Ha az oszlopszélességnél és a pontosságnál is a * jelölést használjuk, akkor először az oszlopszélesség következik, amit a pontosság követ, és utána jön a kiírandó érték. A

```
printf("%*.*s\n", 12, 5, str);
```

utasításnak tehát éppen az lesz a hatása, mint a

```
printf("%12.5s\n", str);
```

utasításnak, ami kiírja a `str` első öt karakterét (vagy kevesebbet, ha `strlen(s)<5`), amelyek elé annyi szóközt tesz, hogy a kiírt karakterek száma pontosan 12 legyen. Az alábbi rejtélyes példa

```
printf("%*%\n", n);
```

egy `%` jelet ír ki jobbra igazítva egy `n` karakter szélességű oszlopba, ami éppen `n-1` szóköz és utána egy `%` jel kiírását jelenti.

Ha a `*` jelet használjuk az oszlopszélességnél, és a hozzá tartozó érték negatív, akkor azt a hatást érjük el, mintha a – jelzőt is megadtuk volna. Vagyis az iménti példában, ha `n` negatív, akkor a kimenet egy `%` jel lesz, amit `1-n` szóköz fog *követni.*

Nyelvi újítások

Az ANSI szabvány két új formátumkódot határozott meg. A `%p` egy mutatót ír ki, az adott C megvalósításnak megfelelő alakban, a `%n` pedig megmutatja, hogy hány karaktert írtunk ki eddig, úgy, hogy ezt az értéket a hozzá tartozó paraméter által kijelölt egész értékbe *menti.* A

```
int n;

printf("hello"\n%n", &n);
```

programrészlet végrehajtása után `n` értéke `6` lesz.

Elavult nyelvi elemek

Az évek során számos dolog eltűnt a `printf` függvényből, de néhány megvalósítás még támogatja ezeket. A `%D` és `%O` formázóelemek egykor a `%ld`, illetve a `%lo` szinonimái voltak, a `%X` formázóelem pedig a `%lx` megfelelője volt. Később hasznosabbnak látták, hogy csupa nagybetűvel lehessen kiírni a hexadecimális értékeket, s így a `%X` jelentése megváltozott. A `%D` és a `%O` ugyanekkor tűnt el a nyelvből.

Régebben csak úgy lehetett bevezető nullákkal kiírni egy értéket, ha a 0 jelzőt használták. A jelentése az volt, hogy a kiírt érték igazítása szóközök helyett nullákkal történjen. Tehát a

```
printf("%06d %06d\n", -37, 37);
```

utasítás ezt írta ki:

```
-00037 000037
```

Ez a meghatározás azonban furcsa kölcsönhatásba került a balra igazítással és a hexadecimális értékek kiírásával. Az egészek pontosságmódosítója tehát sokkal jobban megfelel erre a célra. A

```
printf("%.6d %.6d\n", -37, 37);
```

utasítás azt írja ki, hogy

```
-000037 000037
```

ami elég hasonló ahhoz, hogy a legtöbb esetben a %. kódot használhassuk a %0 helyett.

A.2. Változó paraméterlisták használata a varargs.h állománnyal

Sokszor, ahogy egyre nagyobbra nő egy C program, a készítője egyre inkább azt szeretné, ha a hibakezelés valamilyen következetes módon történne. Erre kézenfekvő megoldás, ha készítünk mondjuk egy error nevű függvényt, ami valahogy úgy működik, mint a printf. Vagyis a

```
error("%d is out of bounds", x);
```

utasítás hatása ugyanaz lesz, mintha azt írnánk, hogy:

```
fprintf(stderr, "error: %d is out of bounds\n",
            ➥ x);
exit(1);
```

Egy ilyen függvény megírása egyetlen apró részletet leszámítva elég egyszerű. Az egyetlen bökkenő, hogy az `error` függvénynek átadott értékek száma és típusa hívásról hívásra változik, éppen úgy, ahogy a `printf` függvénynél. Egy jellemző és egyben helytelen megoldás erre a problémára, amikor az `error` függvényt valahogy így írják meg:

```
void error(a, b, c, d, e, f, g, h, i, j, k)
{
        fprintf(stderr, "error: ");
        fprintf(stderr, a, b, c, d, e, f, g, h,
                    ➥ i, j, k);
        fprintf(stderr, "\n");
        exit(1);
}
```

Az elgondolás itt az, hogy gyűjtsünk össze egy csomó adatot a paraméterlistából, aztán úgy, ahogy van, adjuk át az `fprintf` függvénynek. Mivel az `a-k` paraméterek egyikének sem adtuk meg a típusát, a program `int` típusúnak veszi azokat. Az `error` paraméterlistájában persze mindig lesz majd legalább egy olyan érték (a formázó karakterlánc), amelynek nem `int` lesz a típusa. A program így azon a helytelen feltevésen alapszik, hogy pusztán egész értékek segítségével tetszőleges típusú értékeket lesz képes kiírni.

Lesznek olyan gépek, amelyeken ez nem fog működni. De még ott is csak egy bizonyos pontig számíthatunk a működésére, ahol minden rendben megy. Ha elég sok paramétert kap az `error` függvény, akkor néhány biztosan elveszik közülük. Valamilyen módszernek pedig mégis csak léteznie kell arra, hogy változó paraméterlistája legyen egy függvénynek, hiszen a `printf` is ezen az elven működik.

Van egy kikötés, ami megkönnyíti a `printf` dolgát. Az első paramétere egy karakterlánc kell, hogy legyen, és ennek a karakterláncnak az alapján a függvénynek el kell tudnia dönteni, hogy hány darab és milyen típusú értéket kapott még a függvény (feltéve persze, hogy a `printf` meghívása helyesen történik). Az lenne a jó, ha mi is hozzáférhetnénk valahogy ahhoz a megoldáshoz, amin a `printf` működése alapszik, hogy mi is használhassunk változó hosszúságú paraméterlistákat.

A `printf` sikeres megvalósításához egy ilyen rendszernek a következő tulajdonságokkal kell rendelkeznie:

- Ahhoz, hogy a függvények első paraméterét elérhessük, elég, ha tudjuk annak típusát.
- Amennyiben az `n` paramétert sikeresen elértük, akkor az `n+1` paramétert is el tudjuk érni, ha ismerjük annak típusát.
- A fentiek nem vehetnek igénybe túlzottan sok időt.

Vegyük észre, hogy nincs szükség rá, hogy a paramétereket fordított sorrendben, véletlenszerűen, vagy bármilyen más, az eredetitől eltérő sorrendben olvassuk be. Továbbá általában nem szükséges és nem is lehetséges meghatározni a paraméterlista végét.

A legtöbb C megvalósítás ezt a közös néven `varargs` makróknak nevezett gyűjtemény segítségével éri el. Ezeknek a makróknak a pontos formája eltér a különböző megvalósításokban, de ha az őket felhasználó program elég óvatosan bánik velük, akkor különféle gépek széles körében tud majd változó paraméterlistákat használni.

Minden olyan program, amely használja a `varargs` makrókat, be kell, hogy hívja a megfelelő meghatározásokat tartalmazó állományt, ezzel a sorral:

```
#include <varargs.h>
```

Ebben a fejlécállományban található a `va_list`, `va_dcl`, `va_start`, `va_end` és `va_arg` azonosítók meghatározása. A `va_alist` meghatározása a programozó feladata, és rövidesen látni fogjuk ennek mikéntjét. Fontos, hogy soha ne keverjük össze a `va_list` és a `va_alist` azonosítókat.

Minden C megvalósításnak szüksége van valamilyen információra ahhoz, hogy hozzáférjen egy változó paraméterlista `n.` paraméteréhez, ha annak típusa ismert. Ez az információ az első `n-1` paraméter kiolvasásának melléktermékeként adódik, és úgy képzelhetjük el, mint a paraméterlistára irányuló mutatót, habár a tényleges kivitelezése ennél lényegesebben bonyolultabb bizonyos gépeken.

Ezt az információt egy `va_list` típusú objektum tárolja. Miután tehát bevezetünk egy `va_list` típusú `ap` változót, lehetővé válik az első paraméter értékének meghatározása, és ehhez csak `ap` értékére és az első paraméter típusára van szükségünk.

Ha egy `va_list` segítségével férünk hozzá egy paraméterhez, akkor a `va_list` egyúttal továbblép a paraméterlista következő értékére.

Mivel egy `va_list` *minden* olyan információt tartalmaz, ami az *összes* paraméter eléréséhez szükséges lehet, az `f` függvény létrehozhat egy `va_list` objektumot a paraméterei számára, és átadhatja azt egy `g` függvénynek, ami aztán végiglépkedhet az `f` függvény paraméterein.

Sok megvalósításban például a három `printf` függvény mindegyike egy közös alfüggvényt hív meg, amely számára rendkívül fontos, hogy sorra vehesse az őt meghívó függvény paramétereit.

Egy változó paraméterlistás függvénynek a `va_alist` és `va_dcl` makrókat kell használnia a meghatározásának elején, az alábbi módon:

```
#include <varargs.h>

void error (va_alist) va_dcl
```

A `va_alist` makró a kibontás után az a paraméterlista lesz, amire az adott megvalósításnak szüksége van ahhoz, hogy lehetővé tegye a függvény számára a változó számú paraméter kezelését. A `va_dcl` makró a kibontás után a paraméterlistának megfelelő deklarációkká alakul, *beleértve* a lezáró pontosvesszőt is, ha az szükséges.

Ahhoz, hogy végignézhesse a számára átadott értékeket, az `error` függvényünknek létre kell hoznia egy `va_list` változót, majd kezdőértéket kell neki adnia úgy, hogy a nevét átadja a `va_start` makrónak. Ha a program végzett a paraméterlistával, akkor meg kell hívnia a `va_end` makrót úgy, hogy paraméterként átadja neki a `va_list` nevét, jelezve ezzel, hogy már nincs szüksége a `va_list` változóra.

Az error függvényünk szépen meghízott:

```
#include <varargs.h>

void error(va_alist) va_dcl
{
        va_list ap;
        va_start(ap);

        ide jön a programnak az a része, ami az
        ap változót használja

        va_end(ap);
        ide jöhet bármi, ami nem használja az ap
        változót
}
```

Fontos, hogy ne felejtsük el meghívni a va_end makrót. A legtöbb C megvalósítást ez nem érdekli, de a va_start néhány változata dinamikusan foglal tárhelyet a paraméterlista számára, hogy könnyebb legyen végigmenni azon. Egy ilyen megvalósításban valószínűleg a va_end szabadítja fel a lefoglalt memóriát. Ezért ha megfeledkezünk a va_end meghívásáról, akkor a program látszólag működni fog egyes rendszereken, míg másokon szép lassan felfalja a memóriát.

A va_arg makróval egy paramétert érhetünk el. Két paramétere van, az egyik egy va_list neve, a másik pedig annak a paraméternek az *adattípusa*, amelyiket következőnek készül kiolvasni. Előbb kiolvassa az adott paramétert, majd továbblépteti a va_list értékét, hogy az a következő paraméterre hivatkozzon. Az error függvényünk tehát most így néz ki:

```
#include <varargs.h>

void error(va_alist) va_dcl
{
        va_list ap;
        char *format;
```

```
        va_start(ap);
        format = va_arg(ap, char *);
        fprintf(stderr, "error: ");

        várunk a csodára

        va_end(ap);
        fprintf(stderr, "\n");
        exit(1);
}
```

És itt most megakadtunk, mert a `printf` nem fogad el `va_list` típusú paramétert. Valahogy el kell ezt érnünk, ahogy azt a „várunk a csodára"" megjegyzés is jelzi. De hogyan?

Szerencsére az ANSI C előírásának megfelelően sok C megvalósításban megtaláljuk a `vprintf`, `vfprintf` és `vsprintf` függvényeket. Ezek a függvények éppen úgy működnek, mint a `printf` függvények, azzal a különbséggel, hogy egy `va_list` paramétert várnak ott, ahol a `printf` függvénynél a formázót követő paraméterlista áll. Ezek a függvények csak azért létezhetnek, mert egy `va_list` értéket átadhatunk paraméterként, és nem szükséges, hogy egy `va_arg` szerepeljen ugyanabban a függvényben, ahol megtörténik annak a `va_start` makrónak a hívása, ami létrehozta azt a `va_list` változót, amit a függvény használ.

Az `error` függvényünk végső változata tehát így néz ki:

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

void error (va_alist) va_dcl
{
        va_list ap;
        char *format;

        va_start(ap);
        format = va_arg(ap, char *);
        fprintf(stderr, "error: ");
        vfprintf(stderr, format, ap);
```

```
        va_end(ap);
        fprintf(stderr, "\n");
        exit(1);
}
```

Másik példaként nézzünk meg egy módszert a `printf` megvalósítására a `vprintf` segítségével. Ne felejtsük menteni a `vprintf` eredményét, hogy visszaadhassuk azt a `printf` hívójának:

```
#include <varargs.h>

int
printf(va_alist) va_dcl
{
        va_list ap;
        char *format;
        int n;

        va_start(ap);
        format = va_arg(ap, char *);
        n = vprintf(format, ap);
        va_end(ap);
        return n;
}
```

A varargs.h megvalósítása

A `varargs.h` megvalósítása jellemzően csupa makróból áll, leszámítva a `va_list` bevezetését egy `typedef` segítségével:

```
typedef char *va_list;
#define va_dcl int va_alist;
#define va_start(list) list = (char *) &va_alist
#define va_end(list)
#define va_arg(list,mode) \
        ((mode *) (list += sizeof(mode)))[-1]
```

Figyeljük meg, hogy a `va_alist` még makróként sem szerepel ebben a változatban. A

```
#include <varargs.h>

void error (va_alist) va_dcl
```

kódrészlet a kibontás után így néz ki

```
typedef char *va_list;
void error(va_alist) int va_alist;
```

Vagyis egy változó paraméterlistát váró függvénynek látszólag egyetlen `va_alist` nevű egész típusú paramétere van.

A példa egy olyan C megvalósításon alapszik, amely egy függvény paramétereit közvetlenül egymás mellett tárolja a memóriában, tehát csak az aktuális paraméter címére van szükség ahhoz, hogy végiglépkedjünk a paramétereken. Ebben a megvalósításban tehát a `va_list` nem más, mint egy sima karaktermutató. A `va_start` makró a `va_alist` címére állítja a paraméterét (egy típusátalakítással, nehogy a `lint` program panaszkodjon), a `va_end` pedig nem csinál semmit.

A legösszetettebb makró a `va_arg`. Azt a megfelelő típusú értéket kell visszaadnia, amire a `va_list` paramétere mutat, majd tovább kell léptetnie a paraméter értékét az adott objektumtípus hosszával. Mivel egy típusátalakítás értéke nem lehet a célja egy értékadásnak, ezt úgy oldja meg, hogy a `sizeof` függvénnyel meghatározza a megfelelő növekményt, majd azt közvetlenül hozzáadja a `va_list` értékéhez. Az így kapott mutatót a megfelelő típusra alakítja, majd, mivel az most eggyel távolabbra mutat, egy `-1` index segítségével éri el a megfelelő paramétert.

Vigyázzunk, nehogy belessünk abba a csapdába, hogy egy `char`, `short` vagy `float` típusú második értéket adjunk át a `va_arg` részére. A `char` és `short` paramétereket `int` típusúra, a `float` paramétereket pedig `double` típusúra alakítja a rendszer. Az ilyen helytelen értékek megadása komoly gondokat okozhat.

Azt például sosem írhatjuk, hogy:

```
c = va_arg(ap,char);
```

mert `char` típusú értéket nem tudunk átadni, hiszen azt automatikusan `int` típusúra alakítja a program.

Ehelyett azt kell írnunk, hogy:

```
c = va_arg(ap,int);
```

Másfelől, ha `cp` karaktermutató és a program karaktermutató paramétert vár, akkor teljesen rendben van, ha ezt írjuk:

```
cp = va_arg(ap,char *);
```

A mutató típusú paramétereket nem alakítja át a program, csak a `char`, `short` és `float` értékeket.

Jegyezzük meg azt is, hogy nincs olyan beépített módszer, amivel megállapíthatnánk az átadott paraméterek számát. Minden `varargs` szolgáltatást használó programnak saját magának kell kialakítani valamilyen szabályt, amivel meg lehet adni a paraméterlista végét. A `printf` függvények például a formázó karakterlánc segítségével állapítják meg, hogy hány darab és milyen típusú értéket kapott még a függvény.

A.3. A stdarg.h, avagy az ANSI varargs.h

A `varargs.h` 1981-ből származik, ezért számos C megvalósításban elérhető. Az ANSI C szabványban azonban a `stdarg.h` felel a változó paraméterlisták kezeléséért.

A 7.1. részben leírtak itt is érvényesek a C rendszerek készítőire és azok felhasználóira. Nagyszerű, ha egy ANSI C fordítóban bővítményként szerepel a `varargs.h`, hogy a régebbi programokkal is boldoguljon. Így a gyakorlatban, ha egy program a `varargs.h` szolgáltatást használja, akkor többféle rendszeren fog futni, mint egy hasonló program, ami a `stdarg.h` szolgáltatásra épül. De ha szabványos ANSI programot akarunk írni, akkor nincs mese, mindenképpen a `stdarg.h` szolgáltatást kell használnunk! Ez is azon esetek egyike, amikor bármelyik döntésnek megvan a maga ára.

A `varargs.h` és `stdarg.h` közötti fő különbség abból a megfigyelésből adódik, hogy a gyakorlatban a függvények *első* paraméterének típusa ugyanaz kell, hogy legyen a függvény minden egyes meghívásakor. A `printf` jellegű függvények az első paraméterükből kiindulva meg tudják állapítani a második paraméterük típusát, de a paraméterlista semmilyen információt nem nyújt az első paraméter típusát illetően. Ezért a `stdarg.h` szolgáltatást használó függvényeknek legalább egy rögzített típusú paraméterrel kell rendelkezniük, amit aztán ismeretlen számú és típusú paraméter követhet.

Példaként nézzük meg ismét az `error` függvényt. Első paramétere egy `printf` formázó, ami mindig egy karaktermutató. A függvényt tehát így vezethetjük be:

```
void error(char *, ...);
```

De mi a helyzet az `error` függvény meghatározásával? A `stdarg.h` szolgáltatás nem használja a `varargs.h` `va_arg` és `va_dcl` makróit. Azok a függvények, amelyek a `stdarg.h` szolgáltatást használják, közvetlenül vezetik be a rögzített paramétereiket, majd ezekre építik a változó paramétereiket úgy, hogy az utolsó rögzített paraméter értékét adják át a `va_start` makrónak. Az `error` függvényt tehát így határozhatjuk meg:

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

void error(char *format, ...)
{
        va_list ap;
        va_start(ap, format);
        fprintf(stderr, "error: ");
        vfprintf(stderr, format, ap);
        va_end(ap);
        fprintf(stderr, "\n");
        exit(1);
}
```

Ebben a példában a va_arg használatára nincs szükség, mert a formázó karakterlánc a paraméterlista rögzített részében található.

Egy másik példaként bemutatjuk, hogyan lehet a stdarg.h használatával megvalósítani a printf függvényt a vprintf segítségével:

```
#include <stdarg.h>

int
printf(char *format, ... )
{
        va_list ap;
        int n;

        va_start(ap, format);
        n = vprintf(format, ap);
        va_end(ap);
        return n;
}
```

Tárgymutató

A, Á

B

C, Cs

D

E, É

F

G, Gy

H

I

J

K

L, Ly

M

N, Ny

O, Ö

P

R

S, Sz

T, Ty

U, Ü

V

W

Z, Zs