A C programozási nyelv

B. W. Kernighan - D. M. Ritchie

Tartalomjegyzék

Előszó a magyar kiadáshoz

A C programnyelvet eredetileg a Bell Laboratóriumban az UNIX operációs rendszerhez, az alatt fejlesztették ki PDP-11 számítógépen. A kifejlesztése óta eltelt évek során bebizonyosodott, hogy a C nem egyszerűen egy a napjainkban gombamód szaporodó programnyelvek közül. Korszerű vezérlési és adatszerkezetei, rugalmassága, könnyű elsajátíthatósága széles alkalmazási területet biztosított számára, különösen a 16 bit-es mikroprocesszorok megjelenése óta rendkívül sok gépen dolgoznak C nyelven. C fordító készült olyan gépekre, mint az IBM System/370, a Honeywell 6000 és az Interdata 8/32. A nyelv a kutatás-fejlesztési, tudományos célú programozás népszerű eszközévé vált.

Magyarországon szintén egyre több olyan számítógép működik, amely alkalmas a C megvalósítására. Ilyenek a hazai gyártmányok közül a TPA-11 sorozatú, az R-11 , a szocialista gyártmányok közül az SZM-4 számítógépek, de meg kell említenünk a hazánkban ugyancsak elterjedt PDP-11 sorozat tagjait is. Így érthetően a magyar számítástechnikai szakemberek körében mind nagyobb az érdeklődés a C nyelv iránt, egyre többen szeretnének megtanulni programozni ezen a nyelven. Ebben szeretnénk segíteni e könyv megjelentetésével, amely didaktikusan, bő példa- és gyakorlatanyaggal kiegészítve szól a C összetevőiről, jellemzőiről, de tartalmazza a nyelv referencia-kézikönyvét is. Az Olvasó a legavatottabb forrásból meríthet : a világhírű szerzőpáros egyik tagja, Dennis Ritchie a C nyelv tervezője, a másik, Brian W. Kernighan több, magyarul is megjelent nagy sikerű szakkönyv szerzője. Reméljük, mind a kezdő, mind a gyakorlott C-programozók haszonnal forgatják majd a művet.

A Kiadó

Előszó

A C általános célú programnyelv. Tömörség, a korszerű vezérlési és adatstruktúrák használata, bőséges operátorkészlet jellemzi. Nem nevezhető sem nagyon magas szintű, sem nagy nyelvnek, és nem kötődik egyetlen speciális alkalmazási területhez sem. Ugyanakkor a megkötések hiánya, az általános jelleg sok magas szintű nyelvnél kényelmesebbé és hatékonyabbá teszi.

A C nyelvet tervezője, Dennis Ritchie eredetileg az UNIX operációs rendszer programnyelvének szánta. Az operációs rendszer, a C fordító és lényegében az összes UNIX alkalmazási program (a könyv eredetijének a nyomdai előkészítéséhez használt szoftver is) C nyelven íródott. Dennis Ritchie az első C fordítót PDP- 11-en írta meg, de azóta néhány más gépre, így az IBM System/370-re, a Honeywell 6000-re és az Interdata 8/32-re is készült C fordító:

A C nyelv nem kötődik szorosan egyetlen hardverhez vagy rend szerhez sem, könnyen írhatunk olyan programokat, amelyek változtatás nélkül futnak bármely más, a C nyelvet támogató gépen. Könyvünkkel a C nyelvű programozás elsajátításához szeretnénk segítséget adni. Az olvasó már az Alapismeretek c. fejezet megértése után elkezdhet programozni. A könyv ezután külön-külön fejezetben ismerteti a C nyelv fő összetevőit, majd referencia-kézikönyv formájában is összefoglalja a nyelvet. Az anyag túlnyomórészt példaprogramok írásából, olvasásából és módosításából áll, nem száraz szabálygyűjteményt adunk az olvasó kezébe. A legtöbb példa teljes, ellenőrzött, működőképes program, és nem csupán el szigetelt programrész.

Könyvünkben nemcsak a nyelv hatékony használatát kívántuk ismertetni. Törekedtünk arra is, hogy jó stílusú, áttekinthető, hasznos algoritmusokat és programozási elveket mutassunk be. A könyv nem bevezető jellegű programozási segédkönyv; feltételezi, hogy az olvasó ismeri a programozás olyan alapfogalmait, mint: változók, értékadó utasítások, ciklusok, függvények. [A C nyelvben használatos terminológia szerint a szubrutinokat függvényeknek (functions) nevezik. A ford.] Ugyanakkor a könyv alapján egy kezdő programozó is megtanulhatja a nyelvet, bár szüksége lehet jártasabb kolléga segítségére.

Tapasztalataink szerint a C sokféle feladat megfogalmazására alkalmas, kellemes, kifejező és rugalmas nyelv. Könnyen elsajátítható, és aki megismerte, szívesen használja. Reméljük, hogy könyvünk segítséget nyújt a nyelv hatékony használatában. A könyv megszületéséhez és a megírásakor érzett örömünkhöz nagyban hozzájárultak barátaink, kollégáink gondolatgazdag bírálatai és javaslatai.

Különösen hálásak vagyunk Mike Bianchinak, Jim Blue-nak, Stu Feldmannek, Doug Mcllroynak, Bill Roome-nak, Bob Rosinnek és Larry Roslernek, akik figyelmesen elolvasták a könyv több változatát is. Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pins on, Bill Plauger, Jerry Spivack, Ken Thompson és Peter Weinberger megjegyzéseikkel sokat segítették munkánkat különböző fázisaiban.

Köszönet illeti továbbá Mike Lesket és Jim Ossanát a könyv szedésében való értékes közreműködésükért.

Bevezetés

A C általános célú programnyelv. Történetileg szorosan kapcsolódik az UNIX operációs rendszerhez, mivel ezen rendszer alatt fejlesztették ki és mivel az UNIX és szoftvere C nyelven készült. A nyelv azonban nem kötődik semmilyen operációs rendszerhez vagy géphez. Noha rendszerprogramnyelvnek szokás nevezni, mivel operációs rendszerek írásában jól használható, ugyanolyan célszerűen alkalmazható nagyobb numerikus, szövegfeldolgozó és adatbázis-kezelő programok esetében is.

A C viszonylag alacsony szintű nyelv. Ez nem lebecsülést jelent, csupán azt, hogy a C nyelv - mint a legtöbb számítógép – karakterekkel, számokkal és címekkel dolgozik. Ezek kombinálhatók, és az adott gépen rendelkezésre álló aritmetikai és logikai operátorokkal mozgathatók. A C nyelvben nincsenek olyan műveletek, amelyekkel összetett objektumokat, pl. karakterláncokat, halmazokat, listákat vagy tömböket egy egésznek tekinthetnénk. Hiányzik például azoknak a PL/1 műveleteknek a megfelelője, amelyek egy egész tömböt vagy karakterláncot kezelnek. A nyelvben nincs más tárfoglalási lehetőség, mint a statikus definíció és a függvények lokális változóinál alkalmazott verem elv.

Nincs továbbá olyan, hulladék tárterületek összegyűjtésére alkalmas mechanizmus (garbage collection), mint amilyet az ALGOL 68 nyújt. Végül pedig maga a C nyelv nem biztosít be- és kiviteli szolgáltatásokat: nincsenek read és write utasítások, sem rögzített állományelérési (file-elérési) módszerek. Az összes ilyen magasabb szintű tevékenységet explicit módon hívott függvényekkel kell megvalósítani. Hasonlóképpen a C nyelv csak egyszerű, egy szálon futó vezérlésátadási struktúrákat tartalmaz: ellenőrzéseket, ciklusokat és alprogramokat, de nem teszi lehetővé a multiprogramozást, a párhuzamos műveleteket, a szinkronizálást vagy párhuzamos rutinok (korutinok) használatát. Bár e szolgáltatások némelyikének hiánya súlyos hiányosságnak tűnhet, a nyelv szerény méretek közé szorítása valójában előnyökkel járt. A C nyelv viszonylag kicsi, ezért kis helyen leírható és gyorsan elsajátítható.

A C fordító egyszerű és tömör lehet, ugyanakkor könnyen megírható: a jelenlegi technológiával egy új gépen futó fordító néhány hónap alatt elkészíthető, és kódjának 80%-a várhatólag közös lesz a már létező fordítók kódjával. Ez nagyban segíti a nyelv terjedését, a programok cseréjét. Mivel a C nyelvben definiált adattípusokat és vezérlési szerkezeteket a legtöbb mai számítógép közvetlenül támogatja, kicsi lesz az önálló programok megvalósításához futási időben szükséges rutinkönyvtár, amely a PDP-11-en például csak a 32 bites szorzást és; osztást végrehajtó rutinokat, illetve a szubrutinba való belépést és az onnan való kilépést szolgáló szekvenciákat tartalmazzák.

Természetesen a nyelv valamennyi megvalósítását kiterjedt, az adott géphez illeszkedő függvénykönyvtár teszi teljessé. A függvények biztosítják a be- és kiviteli műveletek elvégzését, a karakterláncok kezelését és a tárfoglalási műveletek végrehajtását. Mivel azonban csak explicit módon hívhatók, szükség esetén elhagyhatók, ezenkívül C programként gépfüggetlen módon is megírhatók.

Minthogy a C a mai számítógépek képességeit tükrözi, a C nyelvű programok általában elég hatékonyak ahhoz, hogy ne kelljen helyettük assembly programokat írni. Ennek legjellemzőbb példája maga az UNIX operációs rendszer, amely majdnem teljes egész ében C nyelven íródott. 13000 sornyi rendszerkódból csak a legalacsonyabb szinten elhelyezkedő 800 sor készült assemblyben. Ezenkívül majdnem minden UNIX alkalmazási szoftver forrásnyelve is a C; az UNIX felhasználók túlnyomó többsége (beleértve e könyv szerzőinek egyikét is) nem is ismeri a PDP-11 assembly nyelvet. A C nyelv sok számítógép képességeihez illeszkedik, minden konkrét számítógép-architektúrától független,

így könnyen írhatunk gépfüggetlen, tehát olyan programokat, amelyek különféle hardvereken változtatás nélkül futtathatók. A szerzők környezetében m a már szokássá vált, hogy az UNIX alatt kifejlesztett szoftvert átviszik a helyi Honeywell, IBM és Interdata rendszerekre. Valójában az ezen a négy gépen működő C fordítók és futtatási szolgáltatások egymással sokkal inkább kompatibilisek, mint az ANSI-szabványos FORTRAN megfelelő változatai. Maga az UNIX operációs rendszer jelenleg mind a PDP-11-en, mind pedig az Interdata 8/32-n fut.

Azokon a programokon kívül, amelyek szükségszerűen többé-kevésbé gépfüggőek, mint a C fordító, az assembler vagy a debugger, a C nyelven írt szoftver mindkét gépen azonos. Magán az operációs rendszeren belül az assembly nyelvű részeken és a periféria handlerek-en kívüli 7000 sornyi kód mintegy 95%-a azonos.

Más nyelveket már ismerő programozók számára az összehasonlítás kedvéért érdemes megemlíteni a C nyelv néhány történeti, technikai és filozófiai vonatkozását. A C nyelv sok alapötlete a nála jóval régebbi, de még ma is élő BCPL nyelvből származik, amelyet Martin Richards fejlesztett ki. A BCPL a C nyelvre közvetett módon, a B nyelven keresztül hatott, amelyet Ken Thompson 1970-ben a PDP-7-esen futó első UNIX rendszer számára írt.

Bár a C nyelvnek van néhány közös vonása a BCPL-lel, mégsem nevezhető semmilyen értelemben a BCPL egyik változatának. A BCPL és a B típus nélküli nyelvek: az egyetlen adattípus a gépi szó és másféle objektumokhoz való hozzáférés speciális operátorokkal és függvényhívásokkal történik. A C nyelvben az alapvető adat-objektumok a karakterek, a különféle méretű egész (integer) típusok és a lebegőpontos számok. Ehhez járul még a származtatott adattípusok hierarchiája, amelyek mutatók (pointerek), tömbök, struktúrák, **union**ok és függvények formájában hozhatók létre.

A C nyelv tartalmazza a jól strukturált programok készítéséhez szükséges alapvető vezérlési szerkezeteket: az összetartozó utasítássorozatot, a döntéshozatalt (if), a programhurok tetején (**while** for) vagy alján (do) vizsgálatot tartalmazó ciklust és a több eset valamelyikének kiválasztását (**switch**). (Ezek mindegyike rendelkezésre állt a BCPL-ben is, szintaxisuk azonban némileg különbözött a C-belitől; a BCPL néhány évvel megelőzte a strukturált programozás elterjedését.)

A C nyelv lehetővé teszi a mutatók használatát és a címaritmetikát. A függvények argumentumainak átadása az argumentum értékének lemásolásával történik, és a hívott függvény nem képes megváltoztatni az aktuális argumentumot a hívóban. Ha név szerinti hívást akarunk megvalósítani, egy mutatót adhatunk át explicit módon és a függvény megváltoztathatja azt az objektumot, amire a mutató mutat. A tömbnév úgy adódik át, mint a tömb kezdőcíme, tehát tömbargumentumok átadása név szerinti hívással történik. Bármely függvény rekurzív módon hívható és lokális változói rendszerint automatikusak, azaz a függvény minden egyes meghívásakor újra létrejönnek. A függvénydefiníciók nem skatulyázhatók egymásba, a változók azonban blokkstruktúrában is deklarálhatók. A C programokban szereplő függvények külön is fordíthatók.

Megkülönböztethetünk egy függvényre nézve belső, külső (csak egyetlen forrásállományban ismert) és teljesen globális változókat. A belső változók automatikusak és statikusak lehetnek. Az automatikus változók a hatékonyság növelése érdekében regiszterekbe helyezhetők, de a **register** deklaráció csak ajánlás a fordítónak és nem vonatkozik adott gépi regiszterekre. A PASCAL-lal vagy az ALGOL 68-cal összehasonlítva a C nem szoros típusmegkötésű nyelv, viszonylag engedékeny az adatkonverziókat illetően, de az adattípusok konverziója nem a PL/1-re jellemző szabadossággal történik.

A jelenlegi fordítók nem ellenőrzik futás közben a tömbindexeket, argumentumtípusokat stb. Ha

szigorú típusellenőrzés szükséges, a C fordító egy speciális változatát, a lint-et kell használni. A lint nem generál kódot, hanem a fordítás és töltés során lehetséges legtöbb szempontból igen szigorúan ellenőriz egy adott programot. Jelzi a nem illeszkedő típusokat, a következetlen argumentum-használatot, nem használt vagy nyilvánvalóan inicializálatlan változókat, az esetleges gépfüggetlenségi problémákat stb. Azok a programok, amelyekben a lint nem talál hibát, ritka kivételektől eltekintve körülbelül ugyanolyan mértékben mentesek a típushibáktól, mint például az ALGOL 68 programok. A megfelelő helyen a lint további szolgáltatásait is ismertetjük.

Végezetül a C-nek, mint minden más nyelvnek, megvannak a maga gyengeségei. Némelyik operátorának rossz a precedenciája; a szintaxis bizonyos részei jobbak is lehetnének; a nyelvnek több, kismértékben eltérő változata él. Mindezzel együtt a C nyelv széles körben alkalmazható, rendkívül hatékony és kifejezőképes nyelvnek bizonyult.

A könyv felépítése a következő:

Az 1. fejezet a nyelv megtanulását segítő bevezetés a C nyelv központi részébe. Ennek az a célja, hogy az olvasó minél hamarabb elkezdhessen programozni, mivel a szerzők hite szerint egy új nyelv megtanulásának egyetlen módja, ha programokat írunk az illető nyelven. A fejezet feltételezi, hogy az olvasó rendelkezik a programozás alapjainak aktív ismeretével; az anyag nem magyarázza meg, hogy mi a számítógép, mi a fordítás, sem pedig az olyan kifejezések jelentését, mint n = n + 1 . Bár lehetőleg mindenütt hasznos programozási módszereket próbáltunk bemutatni, nem szántuk művünket az adatstruktúrák és algoritmusok kézikönyvének: kényszerű választás esetén elsősorban a nyelvre koncentráltunk.

A 2. ... 6. fejezet részletesen, az 1. fejezetnél precízebben tárgyalja a C nyelv különböző elemeit, a hangsúly azonban itt sem a részleteken, hanem a teljes, a gyakorlatban alkalmazható példaprogramokon van.

A 2. fejezet az alapvető adattípusokat, operátorokat és kifejezéseket ismerteti.

A 3. fejezet a programvezérléssel: **if-else**, **while**, **for** stb. foglalkozik.

A 4. fejezet témái : a függvények és a program felépítése, külső változók, az érvényességi tartomány szabályai stb.

Az 5. fejezet a mutatókkal és a címaritmetikával, a 6. fejezet a struktúrákkal és **union**ok-kal kapcsolatos tudnivalókat tartalmazza.

A 7. fejezet a szabványos be- és kiviteli (I/o) könyvtárat ismerteti, amely közös csatlakozófelületet képez az operációs rendszer felé. Ezt a be- és kiviteli könyvtárat minden olyan gép támogatja, amely a C-t támogatja, tehát azok a programok, amelyek ezt használják bevitel, kivitel és más rendszerfunkciók céljából, lényegében változtatás nélkül vihetők át egyik rendszerről a másikra.

A 8. fejezet a C programok és az UNIX operációs rendszer közötti csatlakozásokat írja le, elsősorban a be- és kivitelre, az állományrendszerre és a gépfüggetlenségre koncentrálva. Bár e fejezet egy része UNIX-specifikus, a nem UNIX-ot használó programozó k is hasznos tudnivalókat találhatnak benne - megtudhatják pl., hogyan valósították meg a szabványos könyvtár adott verzióját, és hogyan nyerhetünk gépfüggetlen programkódot.

Az A. függelék a C nyelv referencia-kézikönyvét, a C szintaxisának és szemantikájának hivatalos leírását tartalmazza. Ha az előző fejezetekben esetleg kétértelműségekre vagy hiányosságokra

bukkanunk, mindig ezt kell végső döntőbírónak tekinteni. Mivel a C olyan, még fejlődésben levő nyelv, amely számos rendszeren fut, előfordulhat, hogy a könyv egy-egy része nem felel meg valamely adott rendszer fejlődése pillanatnyi állapotának. Igyekeztünk elkerülni az ilyen problémákat, és megpróbáltuk felhívni a figyelmet a lehetséges nehézségekre. Kétes esetekben azonban általában a PDP-11 UNIX rendszer esetében érvényes helyzet leírását választottuk, mivel a C programozók többségének ez a munkakörnyezete.

Az A. függelékben ismertetjük a fontosabb C rendszerek megvalósításaiban mutatkozó különbségeket is.

1. Alapismeretek

A C nyelv tanulását kezdjük az alapismeretek gyors elsajátításával. Célunk az, hogy működőképes programokon, de a részletekbe, formális szabályokba és kivételekbe való belebonyolódás nélkül mutassuk be a nyelv legfontosabb elemeit.

Nem törekszünk tehát teljességre, sőt pontosságra sem (eltekintve attól, hogy a példáknak helyeseknek kell lenniük). Az olvasónak a lehető leggyorsabban el kell jutnia addig a pontig, ahol már használható programokat tud írni. Éppen ezért bevezetőnk az alapvető tudnivalókra koncentrál: a változókra, az állandókra, az aritmetikára, a vezérlésátadásra, a függvényekre, a be-és kivitellel kapcsolatos elemi ismeretekre.

Tudatosan kihagytuk ebből a fejezetből a C nyelv olyan összetevőit, amelyek nagyobb programok írásánál létfontosságúak. Ilyenek a mutatók, a struktúrák, a C nyelv gazdag operátorkészletének legnagyobb része, néhány vezérlésátadó utasítás és még ezernyi részlet.

Ennek a megközelítésnek megvannak természetesen a maga hátrányai is. Ezek közül a legfontosabb, hogy a nyelv valamely összetevőjét leíró összes információ nem egy helyen található, és a bevezető fejezet rövidségénél fogva félrevezető lehet.

Továbbá, mivel nem használható a C nyelv egész fegyvertára, a példák nem olyan tömörek és elegánsak, mint lehetnének. Ezeket a hátrányokat igyekeztünk a minimálisra csökkenteni, de minderről azért ne feledkezzünk meg.

További hátrány, hogy a bevezető egyes részei a későbbiekben szükségszerűen megismétlődnek. Reméljük, hogy ez az ismétlés inkább segíti, mintsem bosszantja az olvasót. Tapasztalt programozók mindenesetre már ennek a fejezetnek az anyagából ki tudják következtetni a számukra szükséges programozási információt.

Kezdőknek ajánljuk, hogy maguk is írjanak az itt bemutatottakhoz hasonló, kisméretű programokat. A bevezető mindkét csoportnak keretként szolgálhat a későbbi fejezetek anyagának befogadásához.

1.1. Indulás

Egy új programnyelv elsajátításának egyetlen módja, ha programokat írunk az adott nyelven. Az első megírandó program minden nyelv tanulásakor hasonló szokott lenni : Nyomtassuk ki a

Figyelem, emberek!

szavakat. Ez az első akadály. Ahhoz, hogy átugorjuk, képesnek kell lennünk arra, hogy (valahol) létrehozzuk a programszöveget, sikeresen lefordítsuk, betöltsük, lefuttassuk, és ki kell találnunk, hová kerül a kivitt szöveg. Ha ezeken a_mechanikus részleteken túljutottunk, minden más viszonylag könnyen fog menni.

C nyelven a "Figyelem, emberek!" szöveget kinyomtató program a következő:

```
main ()
{
    printf ("Figyelem, emberek! \ n");
}
```

A program futtatásának módja az éppen használt rendszertől függ. Az UNIX operációs rendszerben pl. a forrásprogramot olyan állomány alakjában kell létrehozni, amelynek a neve .c-re végződik, mint például figyel.c, majd ezt a

```
cc figyel.c
```

paranccsal le kell fordítani. Ha nem követtünk el semmilyen hibát, pl. nem hagytunk ki egy karaktert, vagy nem írtunk valamit hibásan, a fordítás rendben végbemegy és egy végrehajtható állomány keletkezik, amelynek neve a.out . Ha ezt az a.out paranccsal lefuttatjuk, akkor a

```
Figyelem, emberek!
```

szöveg jelenik meg a kimeneten. Más operációs rendszerekben a szabályok eltérőek, ilyen esetben forduljunk megfelelő szakemberhez.

1.1. Gyakorlat. Futtassa le a fenti programot a saját rendszerén!

Kísérletezzen a program egyes részeinek elhagyásával, hogy meglássa, milyen hibaüzenetek érkeznek!

Most pedig néhány megjegyzés magáról a programról. A C programok méretüktől függetlenül egy

vagy több függvényt tartalmaznak, amelyek meghatározzák az éppen elvégzendő számítási műveleteket. A C-beli függvények a FORTRAN függvényeihez vagy szubrutinjaihoz, ill. a PL/1 ,a PASCAL stb. eljárásaihoz hasonlítanak. Példánkban a main ilyen függvény.

A függvény neve általában tetszőleges lehet, de a main speciális név - programunk végrehajtása mindig a main elején kezdődik. Ebből az következik, hogy minden programban elő kell hogy forduljon egy main valahol. A main a feladat végrehajtása érdekében általában más függvényeket fog meghívni, amelyek közül egyesek ugyanabban a programban szerepelnek, míg mások előzőleg megírt függvénykönyvtárakból származnak.

A függvények közötti adatátadás egyik módja az argumentumok használata. A függvénynevet követő zárójelek az argumentumlistát határolják: jelen esetben main argumentum nélküli függvény, amit () jelöl. A { } kapcsos zárójelek a függvényt alkotó utasításokat zárják közre; szerepük a PL/1beli do-end-del, az ALGOL és PASCAL begin-end-jével azonos. A függvény hívása a függvény megnevezésével történik, amit az argumentumok zárójelezett listája követ. A FORTRAN-tól vagy PL/1-től eltérően itt nincs call utasítás . A zárójeleknek akkor is szerepelniük kell, ha egyetlen argumentum sincs. A

```
printf ("Figyelem, emberek!\ n");
```

sor nem más, mint függvényhívás, amely a printf nevű függvényt hívja a "Figyelem, emberek!\ n" argumentummal. printf könyvtári függvény, amely az "eredményt" - esetünkben az argumentumát alkotó karakterláncot - (egyéb periféria megadása híján) a terminálra írja. Egy tetszőleges számú karakterből álló, idézőjelek (") közé zárt karaktersorozatot karakterláncnak, karakter-állandónak (stringnek, ill. stringkonstansnak) nevezünk. Pillanatnyilag a karakterláncokat csak a printf és más függvények argumentumaiként használjuk. A karakterláncban előforduló \n karaktersorozat az újsor karakter C-beli jelölésmódja. Hatására a kiírás a következő sor bal szélén folytatódik. Ha a \n-et elhagyjuk (érdemes megkísérelni), azt tapasztaljuk, hogy a kiírás végén a kocsi vissza -soremelés elmarad. Az újsor karaktert csakis egy \n segítségével iktathatjuk be a printf-be: ha valami olyasmivel próbálkozunk, mint

```
printf ("Figyelem, emberek!
");
```

akkor a C fordító barátságtalan üzeneteket fog küldeni bizonyos hiányzó idézőjelekről. A printf sohasem helyez el újsor karaktert automatikusan, így többszöri hívás segítségével egyetlen kimeneti sort fokozatosan rakhatunk össze. Első programunkat így is írhattuk volna:

```
main ()
{
    printf ("Figyelem, ");
    printf ("emberek!");
    printf ("\ n");
}
```

miáltal a korábbival azonos kimenetet kaptunk volna. Megjegyezzük, hogy \n egyetlen karaktert jelent. A \n-hez hasonló, ún. escape jelsorozatok általánosan használható és bővíthető mechanizmust alkotnak nehezen előállítható vagy láthatatlan karakterek jelölésére. A C nyelvben ilyenek még a \t a tabulátor , a \b a visszaléptetés (backspace), a \" az idézőjel és a \\ magának a fordított törtvonalnak (backslash) a jelölésére.

1.2. Gyakorlat. Próbálja ki, mi történik, ha a printf argumentum-karakterlánca \ x-et tartalmaz, ahol x egy, a fenti listában nem szereplő karakter!

1.2. Változók és aritmetika

Az alábbi program a következő Fahrenheit-hőmérsékleteket és a megfelelő Celsius-értékeket tartalmazó táblázatot nyomtatja ki a

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

képlet alkalmazásával.

```
  0:     17.8
 20:     -6.7
 40:     4.4
 60:     15.6
 ...   ...
260:     126.7
280:     137.8
300:     148.9
```

Íme maga a program:

```
/* Fahrenheit-Celsius táblázat kinyomtatása  f = 0, 20, . . ., 300
értékekre */
 main ()
 {
    int lower, upper, step;
    float fahr, celsius;
    lower = 0; /* A hőmérséklet-táblázat alsó határa */
    upper = 300;  /* felső határ */
    step = 20; /* lépésköz */
    fahr = lower;
    while (fahr <= upper) {
         celsius = (5.0 / 9.0) * (fahr - 32.0);
         printf ("%4.0f %6.1f \n", fahr, celsius);
         fahr = fahr + step;
    }
 }
```

Az első két sor:

```
/* Fahrenheit-Celsius táblázat kinyomtatása
 f = 0, 20, . . ., 300 értékekre */
```

egy megjegyzés (comment), amely esetünkben röviden elmondja, hogy mit csinál a program. A fordító minden, a **/*** és ***/** között előforduló karaktert figyelmen kívül hagy; így ide tetszőleges, a program megértését segítő szöveget beírhatunk. Megjegyzések mindenütt előfordulhatnak, ahol szóköz vagy újsor előfordulhat. A C nyelvben használat előtt minden változót deklarálni kell, általában a függvény elején, az első végrehajtható utasítás előtt. Ha erről megfeledkezünk, hibaüzenetet kapunk a fordítótól. A deklaráció egy típus megadásából és az illető típusú változók felsorolásából áll. Példa erre az előbbi program két sora:

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

Az **int** típus azt jelenti, hogy a felsorolt változók egész (**integer**) típusúak. **float** jelöli a lebegőpontos (floating point) változókat, vagyis az olyan számokat, amelyeknek tört részük is van. Mind az **int**, mind a **float** számok pontossága az adott számítógéptől függ. A PDP-11-en például az **int** 16 bit-es előjeles szám, vagyis olyan szám, amelynek értéke -32768 és +32767 között van. A **float** szám 32 bites mennyiség, ami körülbelül 7 értékes számjegyet jelent, 10^{-38} és 10^{38} közötti nagyságrendben. A 2 .

fejezet más gépekre is közli a számábrázolási tartományokat. Az int és float mellett a C nyelvben más alapvető adattípusok is vannak:

- **char** - karakter egyetlen byte,
- **short** - rövid egész,
- **long** - hosszú egész,
- **double** - dupla-pontosságú lebegőpontos szám.

Ezen objektumok méretei ugyancsak gépfüggőek, a részleteket a 2. fejezet tartalmazza. Ezekből az alapvető típusokból tömbök, struktúrák és **union-**ok képezhetők, mutatók mutathatnak rájuk, függvények térhetnek vissza a hívóhoz ezekkel a típusokkal: mindezekkel rövidesen találkozunk. A hőmérséklet-átszámító programban a tényleges számítás a

```
lower = 0; /* A hőmérséklet-táblázat alsó határa */
upper = 300;  /* felső határ */
step = 20; /* lépésköz */
fahr = lower;
```

értékadó utasításokkal kezdődik, amelyek a változók kezdeti értékét állítják be. Az egyes utasításokat pontosvessző zárja le. A táblázat minden sorát azonos módon kell kiszámítani, ezért egy ciklust használunk, amely táblázatsoronként egyszer ismétlődik; ez a célja a **while** utasításnak :

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

Programfutás közben a gép megvizsgálja, teljesül-e a zárójelek közötti feltétel. Ha az értéke igaz (fahr kisebb vagy egyenlő, mint upper), akkor végrehajtja a ciklustörzs (a { és } kapcsos zárójelek közé zárt) utasításait. Ezután ismét megvizsgálja a feltételt, és ha az értéke igaz, újra végrehajtja a törzset. Ha a vizsgálat a hamis logikai értéket szolgáltatja (fahr meghaladja upper-t), akkor a ciklus lezárul és a végrehajtás a ciklust követő első utasításon folytatódik. Az adott program nem tartalmaz több utasítást, tehát a program véget ér. A **while** törzse egy vagy több, kapcsos zárójelek közé zárt utasítás lehet, mint a hőmérséklet-átszámító programban, vagy egyetlen, kapcsos zárójel nélküli utasítás, mint pl.:

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

A **while** által vezérelt utasításokat mindkét esetben két pozícióval beljebb írtuk, hogy első pillantásra világos legyen, mely utasítások helyezkednek el a cikluson belül. A bekezdés a program logikai szerkezetét hangsúlyozza. Bár a C nyelv meglehetősen kötetlen az utasítások pozicionálását illetően, ha azt akarjuk, hogy programunk könnyen olvasható legyen, nagyon fontos a megfelelő bekezdések és üres helyek használata. Célszerű, ha egy sor egy utasítást tartalmaz, és (általában) hagyjunk egy-egy szóközt az operátorok előtt és után. A zárójelek pozíciója kevésbé lényeges: e tekintetben a többféle divatos stílus egyikét választottuk. Az olvasó bármilyen neki megfelelő stílus mellett dönthet, célszerű azonban, ha ezt azután következetesen használja. A munka nagyja a ciklus törzsében készül el. A

```
celsius = (5.0 / 9.0) * (fahr - 32.0);
```

utasítással kiszámítjuk a Celsius-fokokban kifejezett hőmérsékletet, és értékét hozzárendeljük a celsius változóhoz. Az ok, ami miatt 5.0 /9.0-át használtunk, 5 / 9 helyett az, hogy a C nyelv csakúgy, mint sok más nyelv, az egész számokkal végzett osztásnál az eredmény tört részét elhagyja. Tehát 5/9 értéke 0, és 0 lenne az összes hőmérséklet is. Az állandón belüli tizedespont jelzi, hogy az illető állandó lebegőpontos, így 5.0 / 9.0 értéke 0.555..., amire szükségünk van. Ugyancsak 32.0-át írtunk 32 helyett, noha mivel a fahr változó **float** 32 automatikusan **float**-tá (32.0-vá) alakulnak át a kivonás előtt.

Bölcsebb azonban azt a stílust követni, hogy a lebegőpontos állandókat tizedesponttal írjuk akkor is, ha az értékük egész: ezzel az olvasó számára hangsúlyozzuk ezek lebegőpontos természetét, és biztosítjuk, hogy a fordító is eszerint kezelje őket. A 2. fejezet részletesen tartalmazza annak szabályait, hogy az egész számok mikor alakulnak át lebegőpontossá. Egyelőre csak annyit jegyzünk meg, hogy a

```
fahr = lower;
```

értékadó utasítás és a

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

vizsgálat egyaránt a várt módon működik (az **int** a művelet elvégzése előtt **float**-tá alakul át). Ez a példa a printf működéséből is valamivel többet mutat meg. A printf általános célú formátumkonvertáló függvény, amelyet teljes egészében majd a 7. fejezetben ismertetünk. Első argumentuma a kinyomtatandó karakterlánc, ahol az egyes %-jelek mutatják, hogy hová kell a további (második, harmadik, . . .) argumentumokat behelyettesíteni és milyen formátumban kell azokat kinyomtatni. Például a

```
printf ("%4.0f %6.1f \n", fahr, celsius);
```

utasításban a %4.0f konverzió-előírás szerint a lebegőpontos számot egy legalább négy karakter széles helyre kell beírni úgy, hogy a tizedespontot nem követik számjegyek. %6.1f egy másik számot ír le, amely legalább 6 szóközt foglal el és a tizedespont u tán 1 számjegyet tartalmaz - hasonlóan a FORTRAN-beli F6.1 vagy a PL/1-beli F(6,1) alakhoz. A specifikáció egyes részei elhagyhatók:

- %6f azt írja elő, hogy a szám legalább 6 karakter széles;
- %.2f legalább két helyet igényel a tizedespont után, de a szélességet nem korlátozza, és
- %f egyszerűen azt mondja, hogy a számot lebegőpontosként kell kinyomtatni.

Hozzátehetjük, hogy a printf a

- %d-t decimálisként,
- %o-t oktálisként,
- %x-et hexadecimálisként,
- %c-t karakterként,
- %s-et karakterláncként és
- %%-ot %-ként értelmezi.

A printf első argumentumában található minden egyes % konstrukcióhoz hozzárendelődik a neki megfelelő második, harmadik stb. argumentum; ezeknek szám szerint és típus szerint is meg kell egyezniük, ellenkező esetben értelmetlen válaszokat kapunk.

Egyébként a printf nem része a C nyelvnek: a C nyelven belül a be- és kivitel nincs definiálva. A printf-ben nincs semmi rendkívüli: csupán egy hasznos függvény, amely része a C programok által közönségesen elérhető szabványos rutin-könyvtárnak. Azért, hogy magára a C nyelvre koncentrálhassunk, nem foglalkozunk túl sokat a be- és kivitellel, egészen a 7. fejezetig. Elsősorban a formátumozott adatbevitel kérdését halasztjuk el addig.

Ha számokat kell beolvasnunk, olvassuk el a 7. fejezet 7.4. szakaszának a scanf függvényről szóló részét. A scanf - az adatmozgás irányától eltekintve - nagyon hasonló a printf függvényhez.

1.3. Gyakorlat. Módosítsuk úgy a hőmérséklet-átszámító programot, hogy az a táblázat fölé fejlécet is nyomtasson!

1.4. Gyakorlat. Írjunk programot a korábbi példának megfelelő Celsius-Fahrenheit táblázat kinyomtatására!

1.3. A for utasítás

Az olvasó is nyilván tudja, hogy egy programot sokféleképpen meg lehet írni: próbálkozzunk meg a hőmérséklet-átszámító program egy másik változatával :

```
 / * Fahrenheit-Celsius táblázat*/
main ()
 {
    int fahr;
    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
         printf ("%4d %6.1f \n", fahr, (5.0 / 9.0) * (fahr - 32));
}
```

Ez ugyanazokat az eredményeket adja, de láthatóan másképp néz ki. Az egyik fő eltérés, hogy a legtöbb változó szükségtelenné vált: csak a fahr maradt meg **int** változóként (azért, hogy mutassa a printf-ben a %d konverziót). Az alsó és a felső határ, valamint a lépésköz csak állandóként jelenik meg a **for** utasításban, amely maga is új számunkra. A Celsius-hőmérsékletet számító kifejezés most nem külön értékadó utasításként, hanem mint a printf harmadik argumentuma szerepel. Ez az utóbbi módosítás egy egészen általános C-beli szabályon alapul, amely kimondja, hogy minden olyan összefüggésben, ahol valamely adott típusú változó értékét használhatjuk, ugyanolyan típusú kifejezést is használhatunk. Minthogy a printf harmadik argumentumának a %6.1f-re való illeszkedés érdekében lebegőpontosnak kell lennie, tetszőleges lebegőpontos kifejezés is előfordulhat ezen a helyen. Maga a **for** egy ciklusutasítás, a **while** általánosítása. Ha az előbbi **while**-lal összehasonlítjuk, működése rögtön világossá válik.

Három részt tartalmaz, amelyeket pontosvesszők választanak el. Az első rész, vagyis

```
fahr = 0
```

egyszer hajtódik végre a ciklusba való belépés előtt. A második rész a ciklust vezérlő ellenőrzés vagy feltétel:

```
fahr <= 300
```

A gép megvizsgálja a feltételt; ha igaz, akkor végrehajtja a ciklus törzsét (itt egyetlen printf), amit az újrainicializáló lépés, azaz

```
fahr = fahr + 20
```

és újabb feltételvizsgálat követ. A ciklus akkor ér véget, amikor a feltétel hamissá válik. Csakúgy, mint

a **while** esetében, a törzs vagy egyetlen utasítás, vagy pedig kapcsos zárójelek közé zárt utasítások csoportja. Az inicializáló és újrainicializáló r ész egy-egy tetszőleges kifejezés lehet. A **while** és a **for** között szabadon választhatunk aszerint, hogy mi tűnik világosabbnak. A **for** alkalmazása általában olyan ciklusok esetében célszerű, amelyekben az inicializálás és újrainicializálás egy-egy logikailag összefüggő utasítás, mivel a **for** sokkal tömörebb, mint a **while** és egy helyen tartja a ciklusvezérlő utasításokat.

1.5. Gyakorlat. Módosítsuk úgy a hőmérséklet-átszámító programot, hogy az a táblázatot fordított sorrendben, tehát 300 foktól 0 fokig nyomtassa ki!

1.4. Szimbolikus állandók

Még egy megjegyzés, mielőtt elbúcsúznánk a hőmérséklet-átszámítástól. Nem jó gyakorlat, ha "bűvös számokat", például 300-at vagy 20-at építünk be a programba: ezek nem sokat mondanak annak, aki később olvassa majd a programot, és megváltoztatásuk is nagyon nehéz. Szerencsére a C nyelv lehetőséget ad az ilyen bűvös számok elhagyására. A #define szerkezet segítségével a program elején szimbolikus nevet vagy szimbolikus állandót rendelhetünk egy-egy megadott karakterlánchoz. Ezek után a fordító a név minden nem idézőjelezett előfordulását a megfelelő karakterlánccal helyettesíti. A név nemcsak számot, hanem tetszőleges szöveget is helyettesíthet, pl. :

```
#define LOWER 0 /* A táblázat alsó határa*/
#define UPPER 300  / * A táblázat felső határa*/
#define STEP 20 /* Lépésnagyság*/

/*Fahrenheit-Celsius táblázat*/
main () {
   int fahr;
   for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
   printf("%4d %6.1f\n", fahr, (5.0/9.0) * (fahr-32));
}
```

A LOWER, UPPER és STEP mennyiségek állandók, így deklarációban nem jelennek meg. A szimbolikus neveket nagybetűkkel szokás írni, így azonnal megkülönböztethetők a kisbetűs változónevektől. Ügyeljünk arra, hogy a definíciók után nincs pontosvessző! Mivel a nevet követő teljes sor behelyettesítődik, a for-ban túl sok pontosvessző lenne.

1.5. Néhány hasznos program

A következőkben áttekintünk néhány egymással összefüggő programot, amelyek karakteradatokon végeznek egyszerű műveleteket. Ki fog derülni, hogy sok program csupán az itt közölt prototípusok bővített változata.

1.5.1. Karakterek be- és kivitele

A szabványos könyvtárban rendelkezésre állnak olyan függvények, amelyekkel egyszerre egy karaktert lehet írni vagy olvasni. A getchar() minden egyes hívásakor beolvassa a következő bemeneti karaktert és a visszatérési értéke ez a karakter lesz. Tehát

```
c = getchar()
```

után a c változó a következő bemeneti karaktert tartalmazza. A karakterek közönséges esetben a terminálról érkeznek, de ezzel a 7. fejezetig nem kell törődnünk. A putchar(c) függvény a getchar ellentéte:

```
putchar(c)
```

a c változó tartalmát valamilyen kimeneti perifériára írja ki, ami általában ismét a terminál. A putchar és a printf hívásai keverhetők: a kivitt karakterek a hívás sorrendjében fognak megjelenni. Hasonlóan a

printf-hez, a getchar és putchar függvényekben sincs semmi rendkívüli. Ezek nem részei a C nyelvnek, de mindenütt rendelkezésre állnak. Állománymásolás getchar és putchar birtokában meglepően sok hasznos programot írhatunk anélkül, hogy ezen kívül bármi egyebet tudnánk a be-és kivitelről. A legegyszerűbb példa az a program, amely a bemenetet karakterenként a kimenetre másolja. A program váza:

egy karakter beolvasása

while (a beolvasott karakter nem az állomány vége jel)

az éppen beolvasott karakter kimenetre írása

egy új karakter beolvasása

Mindezt C nyelven kifejezve:

```
/* A bemenet átmásolása a kimenetre. 1. változat*/
main()
{
   int c ;
   c = getchar();
   while (c != EOF) {
        putchar(c);
        c = getchar();
   }
}
```

A != relációs operátor jelentése : "nem egyenlő ". A fő probléma a bemenet végének az érzékelése. Megállapodás szerint a getchar az állomány végének megtalálásakor olyan értékkel tér vissza, amely nem valamely érvényes karakter kódja: ílymódon a program észlelni tudja, hogy mikor fogytak el a bemeneten a karakterek. Az egyetlen probléma - ami azonban igen bosszantó -, hogy kétféle megállapodás is közforgalomban van arra nézve, hogy valójában mi az állomány vége érték.

Ezt a problémát egyelőre azzal kerültük ki, hogy a számszerű érték helyett az EOF szimbolikus nevet használtuk, függetlenül a tényleges értéktől. A gyakorlatban EOF vagy -1 , vagy 0, a programot ennek megfelelően vagy

```
#define EOF -1
```

vagy

```
#define EOF 0
```

kell, hogy megelőzze ahhoz, hogy helyes működést kapjunk. Azáltal, hogy az EOF szimbolikus állandót használtuk annak az értéknek a jelölésére, amit állomány vége esetén a getchar visszaad, elértük, hogy az egész programban csupán egyetlen dolog függ az állomány vége tényleges értékétől. A c változót int-nek és nem char-nak deklaráltuk, így abban tárolható a getchar által visszaadott érték. Mint azt a 2. fejezetben látni fogjuk, ez a változó azért **int** típusú, mert alkalmasnak kell lennie arra, hogy az összes lehetséges karakteren kívül a z EOF értéket is felvegye. Gyakorlott C programozók a másolóprogramot tömörebben írnák le. A C nyelvben az olyan értékadások, mint

```
c = getchar()
```

kifejezésekben is használhatók; a kifejezés értéke egyszerűen a bal oldalhoz hozzárendelt érték. Ha a c-re vonatkozó értékadás egy **while** feltételvizsgáló részének belsejébe kerül, akkor az állománymásoló program a következőképpen írható :

```
/* A bemenet átmásolása a kimenetre - 2. változat*/
main()
{
   int c;
   while ((c = getchar()) != EOF)
        putchar (c);
}
```

A program beolvas egy karaktert, hozzárendeli c-hez, majd ellenőrzi, hogy a karakter azonos-e az állomány vége jellel. Ha nem, akkor a programfutás a **while** törzsének végrehajtásával, azaz a karakter kinyomtatásával folytatódik. Ezután a **while** ciklus ismétlődik. Ha a program végül eléri a bemeneti karaktersorozat végét, akkor a **while** és vele együtt a main is befejeződik. Ez a változat egy helyre vonja össze a beolvasást - most csak egy getchar hívás van -, és egyben le is rövidíti a programot. Az értékadás behelyezése a feltételvizsgálatba az egyik olyan eset, amikor a C nyelv hasznos tömörítést tesz lehetővé. (Megvan persze a lehetősége annak, hogy ezt túlzásba vigyük és áttekinthetetlen programkódot hozzunk létre, de ezt igyekszünk elkerülni.) Lényeges látnunk, hogy az értékadás körüli zárójelek a feltételen belül tényleg szükségesek. A != precedenciája magasabb, mint az = szimbólumé ami azt jelenti, hogy zárójelek hiányában a != relációvizsgálat megelőzné az = értékadási művelet végrehajtását . Így a

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

utasítás egyenértékű a

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

utasítással. Ez azzal a nemkívánatos eredménnyel jár, hogy c 0 vagy 1 lesz, attól függően, hogy a getchar hívásakor állomány vége jel érkezett-e vagy sem. (Erről részletesebben a 2. fejezetben

szólunk.)

1.5.2. Karakterszámlálás

A következő program, amelyet a másolóprogram kis módosításával kaptunk, megszámlálja a beolvasott karaktereket:

```
 /* Megszámlálja a bemeneten érkező karaktereket*/
main()
{
    long nc;
    nc = 0;
    while (getchar () != EOF)
         ++nc;

    printf("%ld\n", nc);
}
```

A

```
 ++nc;
```

utasítás egy új operátort mutat be, amelynek jele ++, és a jelentése: inkrementálj eggyel. Írhatnánk azt is, hogy nc = nc + 1, de ++nc tömörebb és gyakran hatékonyabb is. Létezik egy ennek megfelelő -- operátor, amely 1-gyel dekrementál. A ++ és a -- egyaránt lehet prefix (előtag) operátor (++nc) vagy postfix (utótag) operátor (nc++)- e két alakhoz kifejezésekben különböző értékek tartoznak, amint azt a 2. fejezetben látni fogjuk, de ++nc és nc++ egyaránt inkrementálja nc-t. Egyelőre megmaradunk a prefix operátornál. A karakterszámláló program a karakterek számát **int** helyett egy **long** típusú változóban tárolja. A PDP-11-en egy **int** mennyiség maximális értéke 32767, így a számláló viszonylag kevés bemenő érték esetén is túlcsordulna, ha int-nek deklarálnánk. A Honeywell és IBM C-ben a **long** és az **int** ugyanaz, de a maximális érték sokkal nagyobb. A %ld konverziómegadás azt jelzi printf-nek, hogy a megfelelő argumentum egy hosszú egész (**long** integer). Ennél is nagyobb számok esetén a double típus (duplahosszúságú lebegőpontos szám) használható. A **while** helyett **for** utasítást fogunk használni, hogy bemutathassuk a ciklusszervezés egy másik lehetőségét.

```
 /*Megszámlálja a bemeneten érkező karaktereket*/
 main()
{
    double nc;
    for (nc = 0; getchar() != EOF; ++nc)
         ;

     printf ("%.0f \n", nc);
 }
```

A printf mind **float**, mind double esetén %f-et használ; a %.0f elnyomja a nemlétező tört rész kiírását. A **for** ciklus törzse üres, mivel az egész feladat a feltételvizsgáló és újrainicializáló részben hajtódik végre. A C nyelvtani szabályai azonban megkívánják, hogy a **for** utasításnak legyen törzse. Az egymagában álló pontosvessző, vagyis a nulla (üres) utasítás e követelmény kielégítése miatt szerepel. Külön sorba írtuk, hogy feltűnőbb legyen. Mielőtt befejeznénk a karakterszámláló program elemzését, felhívjuk a figyelmet, hogy ha a bemeneten nincsenek karakterek, akkor getchar legelső hívásakor a while vagy a for feltételvizsgálata hamis értéket eredményez, és így a program eredménye elvárásunknak megfelelően 0 lesz. Ez lényeges megfigyelés. A **while** és a **for** egyik előnyös tulajdonsága, hogy a feltételvizsgálat a ciklus fejében van, megelőzi a törzset. Ha tehát semmit sem kell csinálni, akkor tényleg semmi sem történik, még akkor sem, ha emiatt a program sohasem halad át a ciklus törzsén.

A programoknak akkor is értelmesen kell működniük, ha a bemenet "üres". A **while** és a **for** segítségével a programok határesetekben is ésszerűen viselkednek.

1.5.3. Sorok számlálása

A következő program megszámlálja a bemenetére érkező sorokat. Feltételezzük, hogy a bemenő sorok a \n újsor karakterrel fejeződnek be, amely szigorúan minden kiírt sor végén megjelenik.

```
 /*A bemenetre érkező sorok számlálása*/
 main()
 {
  int c, nl ;
  nl = 0;
  while ((c = getchar()) != EOF)
  if (c == '\n')
   ++nl;
  printf ("%d\n", nl );
 }
```

A **while** törzse most egy **if**-et tartalmaz, amely pedig a ++ nl inkrementáló műveletet vezérli. Az if utasítás elvégzi a zárójelezett feltétel vizsgálatát, ha ennek eredménye igaz, akkor végrehajtja a rá következő utasítást (vagy kapcsos zárójelek közötti utasításcsoportot). A sorokat ismét úgy rendeztük el, hogy világos legyen, mit mi vezérel. A C nyelv jelölésmódjában az == (kettős egyenlőségjel) jelentése: egyenlő . . .-vel (hasonlóan a FORTRAN-beli .EO.-hoz). Ezzel a szimbólummal különböztetjük meg az egyenlőség vizsgálatát a szimpla = jeltől, amit értékadásra használunk.

Minthogy tipikus C programokban az értékadás körülbelül kétszer olyan gyakran fordul elő, mint az egyenlőségvizsgálat, ésszerű, hogy az értékadó operátor fele olyan hosszú legyen. Bármely egymagában álló karakter aposztrófok közé írva az illető karakternek a gép karakterkészletében szereplő numerikus értékét jelenti: ezt karakterállandónak nevezzük.

Így például 'A' karakterállandó; az ASCII karakterkészletben ennek értéke 65, vagy is az A karakter belső ábrázolása. Természetesen kényelmesebb 'A'-t írni, mint 65-öt: 'A' jelentése világos és független az adott karakterkészlettől. A karakterállandókban a karakterláncokban használt escape jelsorozatok is megengedettek, így a feltételvizsgálatokban és aritmetikai kifejezésekben '\n' az újsor karakter kódértékét jelenti. Ne feledjük, hogy '\n' egyetlen karakter, amely kifejezésekben egy egész számmal egyenértékű, \n viszont karakterlánc, amely az adott esetben egyetlen karaktert tartalmaz! A karakterek és karakterláncok témáját a 2. fejezetben folytatjuk.

1.6. Gyakorlat. Írjunk olyan programot, amely megszámlálja a szóközöket, tab és újsor karaktereket!

1.7. Gyakorlat. Írjunk olyan programot, amely a bemenetet átmásolja a kimenetre, miközben az egy vagy több szóközből álló karakterláncokat egyetlen szóközzel helyettesíti!

1.8. Gyakorlat. Írjunk olyan programot, amely minden egyes tab karaktert a > , visszaléptetés (backspace), - háromkarakteres sorozattal helyettesít, ami áthúzott > -ként fog megjelenni, továbbá, amely a visszaléptetés karaktereket a hasonlóan át húzott < szimbólummal helyettesíti! Ezáltal a tab karakterek és visszaléptetések láthatóvá válnak.

1.5.4. Szavak számlálása

Negyedik hasznos programunk sorokat, szavakat és karaktereket számlál annak a laza definíciónak az alapján, amely szónak tekint minden olyan karaktersorozatot, amely nem tartalmaz szóközt, tab vagy újsor karaktert. (Az alábbi program az UNIX wc segédprogramjának a váza.)

```
/*A bemenet sorainak, szavainak, karaktereinek számlálása*/

#define YES 1
#define NO 0

main ()
{
   int c, nl, nw, nc, inword;
   inword = NO;
   nl = nw = nc = 0;
   while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
             ++nl;
        if (c == ' ' ||c == '\n' ||c == '\t')
             inword = NO;
        else if (inword == NO) {
             inword = YES;
             ++nw;
        }
   }
   printf ("%d %d %d\n", nl, nw, nc);
}
```

Ahányszor a program egy szó első karakterével találkozik, növeli a számlálót. Az inword változó jelzi, hogy a program pillanatnyilag egy szón belül van-e vagy sem; kezdetben nincs szón belül, miáltal a NO érték rendelődik hozzá. Előnyben részesítjük a YES és NO szimbolikus állandókat az 1 és 0 számértékekkel szemben, mivel olvashatóbbá teszik a programot. Természetesen egy ilyen kis programban, mint ez, ennek nemigen van jelentősége, de nagyobb programokban az érthetőség javulása sokszorosan megéri azt a szerény plusz fáradságot, ami, az ilyen stílusú programíráshoz szükséges. Módosítani is könnyebb az olyan programot, ahol a számok csupán szimbolikus állandóként jelennek meg. Az

```
nl =nw=nc=0;
```

sor mindhárom változót kinullázza. Ez nem speciális eset, hanem annak a ténynek a következménye, hogy az értékadások jobbról balra mennek végbe. Ez valójában ugyanaz, mintha azt írtuk volna, hogy

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

A || operátor jelentése VAGY, tehát az

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

sor azt jelenti, hogy ha "c szóköz vagy c újsor vagy c tab karakter . .. ". (Mint mondottuk, a \t escape szekvencia a tab karakter látható megjelenési formája.) Létezik az ennek megfelelő && operátor is az ÉS kifejezésére. Az && vagy || operátorokkal összekapcsolt kifejezések kiértékelése balról jobbra történik, és a kiértékelés rögtön abbamarad, amint az egész kifejezés igaz vagy hamis volta nyilvánvalóvá válik. Ha tehát c szóköz karakter, nincs szükség annak megállapítására, hogy c újsort vagy tab-ot tartalmaz-e, tehát ezek a vizsgálatok nem mennek végbe. Itt most ez nem különösen lényeges, de bonyolultabb esetekben nagyon is fontos lehet, amint azt nemsokára látni fogjuk. A példában a C nyelv **else** utasítása is szerepel, amely megadja azt az alternatív tevékenységet, amit akkor kell elvégezni, ha az **if** feltételrésze hamis értékű. Általános alakja:

```
if (kifejezés)
    1.utasítás
else
    2.utasítás
```

Az **if-else**-hez tartozó két utasítás közül egy és csakis egy, mégpedig ha a kifejezés értéke igaz, akkor az 1. utasítás, ha hamis, akkor a 2. utasítás hajtódik végre. Mindkét utasítás valójában egészen bonyolult is lehet. A szavakat számláló programban pl. az else utáni utasítás egy újabb if, amely a kapcsos zárójelek közötti két utasítást vezérli.

1.9. Gyakorlat. Hogyan ellenőrizhetjük a szavakat számláló programot? Mik lehetnek szóhatárok.

1.10. Gyakorlat. Írjunk olyan programot, amely külön-külön sorokban nyomtatja ki a bemenetére érkező szavakat!

1.11. Gyakorlat. Módosítsuk a szavakat számláló programot úgy, hogy jobban definiáljuk a szó fogalmát, például a szó legyen betűk, számjegyek és aposztrófok olyan sorozata, amely betűvel kezdődik!

1.6. Tömbök

Írjunk olyan programot, amely megszámlálja, hogy hányszor fordulnak elő az egyes számjegyek, hány üres helyet adó karakter (szóköz, tab, újsor) és hány egyéb karakter van a beolvasott állományban! Ez a

feladat nyilván mesterkélt, de lehetővé teszi, hogy egyetlen programban szemléltessük a C több jellegzetességét. Tizenkétféle bemeneti karaktert kell megkülönböztetnünk, így érdemes az egyes számjegyek előfordulásainak számát egy tömbben nyilvántartani ahelyett, hogy tíz külön változónk lenne. A program egyik lehetséges változata:

```
 /*Számjegyek, üres helyek és egyéb karakterek számlálása*/
main()
{
      int c, i, nwhite, nother;
      int ndigit [10];
      nwhite = nother = 0;
      for (i = 0; i < 10; ++i)
           ndigit [i] = 0;

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

     printf ("számjegyek=");
    for (i = 0; i < 10; ++i)
          printf ("%d", ndigit [i]);

    printf ("\n üres hely = %d, egyéb = %d\n", nwhite, nother);
}
```

Az

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

deklaráció azt fejezi ki, hogy az ndigit egy 10 egészből álló tömb. A tömbindexek a C nyelvben mindig 0-val kezdődnek (és nem 1-gyel, mint a FORTRAN-ban és a PL/1-ben), így a tömb elemei: ndigit[0], ndigit [1], .. ., ndigit[9]. Ezt tükrözi a két for ciklus: az egyik inicializálja, a másik kiíratja a tömböt. Az index tetszőleges egész típusú kifejezés. Így természetesen lehet az index egész típusú változó, mint pl. i, valamint egész értékű állandó is. Az adott program lényeges módon kihasználja a számjegyek karakterábrázolásának tulajdonságait. Például az

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

vizsgálat eldönti, hogy a c-ben levő karakter számjegy-e. Ha az, akkor az illető számjegy numerikus értéke

```
c - '0'
```

Ez a módszer csak akkor alkalmazható, ha '0', '1', . . . növekedő sorrendű pozitív számok és '0' és '9' között csak számjegyek vannak. Szerencsére ez minden szokásos karakterkészlet esetében így van. A char-okat és int-eket tartalmazó kifejezésekben definíció szerint kiértékelés előtt minden int-té konvertálódik, így a **char** változók és állandók aritmetikai szempontból lényegében az **int** mennyiségekkel azonosak. Ez egészen természetes és kényelmes megoldás: például c – '0' egész típusú kifejezés, amelynek értéke 0 és 9 között van a c-ben tárolt '0' és '9' közötti karaktereknek megfelelően, és így érvényes indexe az ndigit tömbnek. Annak eldöntése, hogy a karakter számjegy, üres hely vagy valami más, az

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

programrész segítségével történik. Az

```
if (feltétel)
    utasítás
else if (feltétel)
         utasítás
    else
         utasítás
```

programszerkezetet gyakran alkalmazzák többbutas elágazások leírására. A programszöveg beolvasása felülről kezdve mindaddig folytatódik, amíg a gép igaz feltételt nem talál. Ekkor végrehajtja az odatartozó utasítás részt, és az egész művelet végetér. (Az utasítás természetesen több, kapcsos

zárójelek közé zárt utasítás is lehet.) Ha egyik feltétel sem igaz, a gép az utolsó **else** utáni utasítást hajtja végre, amennyiben van ilyen. Ha az utolsó **else** és a hozzátartozó utasítás hiányzik (mint a szavakat számláló programban), semmi sem történik. A kezdő **if**, valamint a záró **else** között tetszőleges számú

```
else if (feltétel)
   utasítás
```

csoport fordulhat elő. Stiláris szempontból célszerű a bemutatott módon megszerkeszteni ezt a programrészt, hogy a hosszú döntési láncok ne nyúljanak túl a papír jobb szélén. A 3. fejezetben fogunk szólni a **switch** utasításról, amely szintén többutas programelágaztatások leírására ad lehetőséget. A **switch** alkalmazása különösen akkor előnyös, amikor azt vizsgáljuk, hogy egy adott egész vagy karakter típusú kifejezés értéke egyenlő-e egy állandókból álló halmaz valamelyik elemével. Az összehasonlítás céljából a 3. fejezetben bemutatjuk az előbbi a program **switch** utasítással megírt változatát.

1.12. Gyakorlat. Írjunk olyan programot, amely kinyomtatja a bemenetén előforduló szavak hosszúságának hisztogramját! A legegyszerűbb, ha a hisztogramot vízszintesen rajzoljuk; a függőleges irányú rajzolás nehezebb feladat.

1.7. Függvények

A C nyelvben a függvény ugyanaz, mint a FORTRAN-ban a szubrutin, ill. függvény, vagy a PL/1-ben, a PASCAL-ban és más nyelvekben az eljárás. A függvény kényelmes lehetőséget nyújt számunkra, hogy valamely számítási részt "fekete dobozba" zárjunk, amelyet azután használhatunk anélkül, hogy tartalmával törődnünk kellene. Valójában csak a függvények segítségével birkózhatunk meg nagy és bonyolult programokkal.

Helyesen tervezett függvények esetében teljesen figyelmen kívül hagyhatjuk, hogyan keletkezik a függvény értéke (eredménye); elegendő a feladat és az eredmény ismerete. A C nyelv egyszerű, kényelmes és hatékony függvényhasználatot tesz lehetővé. Gyakran fogunk olyan függvényekkel találkozni, amelyek csupán néhány sorból állnak és amelyeket csak egyszer hívunk meg: ezeket kizárólag a program világosabbá tétele érdekében használjuk.

Ezidáig csak olyan függvényeket használtunk, mint a printf, a getchar vagy a putchar, amelyeket készen kaptunk; itt az ideje, hogy magunk is írjunk néhányat. Mivel a C nyelvnek nincs olyan hatványozó operátora, mint a ** a FORTRAN-ban vagy a PL/1-ben, szemléltessük a függvénydefiniálás technikáját a power(m, n) függvény megírásával, amely az m egész típusú változót a pozitív egész n hatványra emeli. Tehát a power(2,5) függvény értéke 32.

Ez a függvény nyilvánvalóan nem tudja mindazt, amit ** tud, mivel csak kis egész számok pozitív hatványait tudja kezelni, de legjobb, ha egyszerre csak egy problémára összpontosítunk. Az alábbiakban a power függvényt egy főprogramba ágyazva mutatjuk be. Ne feledjük, hogy a main() maga is függvény!

```
/*Hatványozó függvény tesztelése*/

/*x n-dik hatványra emelése; n >0*/
power (int x,int n)
{
   int i, p;
   p = 1;
   for (i = 1; i <= n; ++i)
        p = p * x;
   return (p);
}

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

Mindkét függvény az alábbi alakú:

```
név (opcionális argumentumlista)
{
   deklarációk
   utasítások
}
```

A függvények tetszőleges sorrendben szerepelhetnek, és egy vagy két forrásállományban egyaránt állhatnak. Természetesen, ha a forrás két állományban található, bonyolultabb a fordítás és a töltés, mintha minden egyetlen állományban van, de ez az operáció s rendszer kérdése és nem a nyelvjellegzetessége. Pillanatnyilag feltesszük, hogy a két függvény ugyanabban az állományban van, tehát mindaz, amit a C programok futtatásáról megtanultunk, nem változik. A power függvényt a

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

sorban kétszer hívtuk meg. Mindkét hívás két argumentumot ad át a power függvénynek, amely mindkét alkalommal visszaad egy-egy egész számot, amit a hívó program formátumoz és megjelenít. Kifejezésen belül power(2,i) ugyanolyan egész, mint 2 és i. (Nem minden függvény eredményez egész

értéket: ezt a témát a 4. fejezetben folytatjuk.) A power argumentumait megfelelőképpen deklarálni kell ahhoz, hogy a típusuk ismert legyen. Ez a függvény nevét követő argumentumlistájában megadott

```
(int x, int n)
```

sorban történik. A power függvény által a saját argumentumai számára használt nevek teljes mértékben lokálisak a power függvényre nézve, azokhoz semmilyen más függvény sem férhet hozzá: más rutinok veszélytelenül használhatják ugyanezeket a neveket. Ez a p és az i változóra is vonatkozik: a power-beli i változónak semmi köze a main-ben használt i-hez. A power függvény által kiszámított értéket a **return** utasítás adja vissza a main-nek. A zárójelek között tetszőleges kifejezés előfordulhat. Egy függvénynek nem feltétlenül szükséges értéket visszaadnia: egy kifejezés nélküli **return** utasítás átadja a vezérlést, de nem ad át hasznos értéket a hívónak - ez történik olyankor, amikor a vezérlés a függvény végét átlépi azáltal, hogy eléri a jobb oldali záró kapcsos zárójelet.

1.13. Gyakorlat. Írjunk olyan programot, amely a beolvasott szöveget kisbetűssé alakítja át egy olyan lower(c) függvény segítségével, amely c-vel tér vissza, ha c nem betű, és c kisbetűs megfelelőjét adja vissza,ha c betű!

1.8. Argumentumok; érték szerinti hívás

A C függvények egyik tulajdonságát más nyelvekben - különösen a FORTRAN-ban vagy PL/1-ben - járatos programozók szokatlannak találhatják: a C-ben mindig érték szerinti függvényargumentum-átadás történik. Ez azt jelenti, hogy a hívott függvény az argumentumainak nem a címét, hanem - ideiglenes változóban (valójában egy veremben) – az értékét kapja meg. Ez bizonyos eltérő tulajdonságokhoz vezet az olyan név szerint hívó nyelvekhez képest, mint amilyen a FORTRAN és a PL/1, amelyekben a hívott rutin az argumentum címét, nem pedig az értékét kapja meg. A fő különbség az, hogy a C-ben a hívott függvény nem tudja megváltoztatni a hívó függvény változóinak értékét, csak a saját, ideiglenes változópéldányainak tud új értéket adni. Az érték szerinti hívás azonban előny, nem pedig hátrány. Általa legtöbbször tömörebb programokat állíthatunk elő, kevesebb segédváltozót kell használnunk, mivel a hívott rutinban az argumentumok ugyanolyan módon kezelhetők, mint a hagyományosan inicializált változók. Nézzük például a power következő változatát, amely kihasználja ezt a tényt:

```
/*x n-edik hatványra emelése; n > 0; 2. változat*/
power (int x,int n)
{
   int p;
   for (p = 1; n > 0; --n)
        p = p * x;
   return (p);
}
```

Az n argumentumot ideiglenes változóként használtuk és addig dekrementáltuk, amíg el nem érte a 0-t; így nincs szükség az i változóra. Mindannak, ami az n-nel a power-en belül történik, nincs befolyása arra az argumentumra, amellyel eredetileg a függvényt meghívtuk. Szükség esetén megoldható, hogy a függvény módosítani tudja az őt hívó rutin valamelyik változóját. A hívónak meg kell adnia a módosítandó változó címét (gyakorlatilag egy, a változót megcímző mutatót), és a hívott függvénynek az argumentumot mutatóként kell deklarálnia, a tényleges változóra ezen keresztül, indirekt módon kell hivatkoznia. Ezzel az 5. fejezetben foglalkozunk. Ha egy tömb nevét használjuk argumentumként, akkor a függvénynek átadott érték ténylegesen a tömb kezdetének helye vagy címe. (A tömbelemek nem másolódnak át). Ezt az értéket indexelve a függvény a tömb tetszőleges elemét elérheti és megváltoztathatja. Ezzel a következő fejezet foglalkozik.

1.9. Karaktertömbök

A C nyelvben leggyakoribb tömbtípus valószínűleg a karaktertömb. A karaktertömbök és az őket kezelő függvények használatát egy olyan programmal szemléltetjük, amely sorokat olvas be és közülük a leghosszabbat megjeleníti. Az alapstruktúra meglehetősen egyszerű :

```
while (van még sor)
   if (hosszabb, mint az eddigi leghosszabb sor)
        tárold a sort és a hosszát
nyomtasd ki a leghosszabb sort
```

Ez a struktúra világossá teszi a program természetes tagozódását. Az egyik rész beolvassa és megvizsgálja az újsort, a másik tárolja, a harmadik vezérli a folyamatot. Minthogy a feladatok ilyen szépen elkülöníthetők, helyes, ha a programot is eszerint írjuk meg. Ennek megfelelően először írjunk egy külön getline függvényt, amely beolvassa a bemenetről a következő sort,_ ez a getchar függvény általánosítása. Szeretnénk ha a függvény más környezetben is használható lenne, ezért igyekszünk a lehető legrugalmasabbá tenni. A minimális igény, hogy a getline jelezze vissza az esetleges állományvéget; Általánosabban használható lesz a függvény, ha a sor hosszát adja vissza , vagy pedig nullát, ha elérte az állomány végét. A nulla bizonyosan nem valódi sorhossz, mivel minden sor legalább egy karaktert kell, hogy tartalmazzon, még a csupán egyetlen soremelést tartalmazó sor hossza is 1. Ha

azt találjuk, hogy egy sor hosszabb, mint az addigi leghosszabb sor, valahová el kell mentenünk. Logikus, hogy ez egy második függvény, a copy feladata legyen, amely az új sort biztos helyre menti.

Végezetül szükségünk van egy főprogramra, amely vezérli a getline-t és a copy-t.

Íme az egész program:

```
 #define MAXLINE 1000 /*A beolvasott sor maximális mérete*/


 /*Sor beolvasása s-be, a hosszát adja vissza*/
getline (char s[],int lim)
{
    int c, i;
    for (i = 0; i < lim - 1 && (c =getchar ()) != EOF
         && c != '\n'; ++i)
         s [i] = c;

    if (c == '\n') {
         s [i] = c;
         ++i;
    }
    s [i] = '\0';
    return (i);
}

/*s1 másolása s2-be; s2-t elég nagynak feltételezi*/
copy (s1 [], s2 [])
{
  int i;
  i = 0;
  while ((s2 [i] = s1 [i]) != '\0')
  ++i;
}

/*A leghosszabb sor kiválasztása*/
main ()
{
    int len; /*A pillanatnyi sor hossza*/
    int max; / *Az eddigi maximális hossz*/
    char line [MAXLINE]; /*A pillanatnyilag olvasott sor*/
    char save [MAXLINE]; /*A leghosszabb sor mentésére*/
    max = 0;
    while ((len = getline (line,MAXLINE)) > 0)
         if (len > max) {
              max = len;
              copy (line,save);
         }

    if (max > 0) /*Volt sor*/
         printf ("%s", save);
}
```

main csakúgy, mint getline, két argumentumon és egy visszaadott értéken keresztül kommunikál. A getline-ban az argumentumokat a

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

sorok deklarálják, amelyek előírják, hogy az első argumentum tömb, a második pedig egész típusú legyen. Az s tömb hossza getline-ban nincs megadva, mivel azt a main-ben határozzuk meg. A getline a **return** utasítás segítségével küld vissza értéket a hívónak úgy, ahogy azt a power függvénynél láttuk. Egyes függvények hasznos értéket szolgáltatnak, míg másokkal, így a copy-val is valamely adott feladatot végeztetünk el, és nem adnak vissza hasznos értéket. A getline az általa létrehozott tömb végére, a karakterlánc végének jelzésére egy \0 karaktert (egy nulla karaktert, amelynek értéke 0) helyez el. Így működik a C fordító is: amikor egy karakterlánc állandó, mint például

```
"hello\n"
```

van a C programban, a fordító a lánc karaktereit tartalmazó karaktertömböt hoz létre, amelyet egy \0-val zár le. A függvények, pl. a printf, így képesek a karakterlánc végének az érzékelésére:

```
h e l l o \n \0
```

A printf %s formátumspecifikációja egy ilyen formában ábrázolt karakterláncot vár. Ha megvizsgáljuk a copy függvényt, észrevehetjük, hogy az is arra támaszkodik, hogy az s1 bemeneti argumentumot \0 zárja le, és ezt a karaktert is átmásolja az s2 kimenet i argumentumra.

(Mindez azt feltételezi, hogy \0 nem része a normál szövegnek.) Futólag érdemes megjegyeznünk, hogy még egy ilyen kis program is felvet néhány kényes tervezési problémát. Például mit csináljon main, ha olyan sorral találkozik, amely hosszabb, mint a megadott korlát? A getline helyesen működik: amikor a tömb megtelt, leáll, még akkor is, ha nem talált újsort. A hosszat és az utolsónak visszaadott karaktert ellenőrizve a main el tudja dönteni, hogy a sor túl hosszú volt-e, majd tetszés szerint cselekedhet. A rövidség kedvéért ezt a problémát figyelmen kívül hagytuk. Aki a getline függvényt használja, nem tudhatja előre, hogy milyen hosszú lehet egy beolvasott sor, így a getline ellenőrzi a túlcsordulást. Másfelől a copy használója már tudja (vagy kideritheti), hogy mekkorák a karakterláncok, ezért úgy döntöttünk, hogy ezt a függvényt nem egészítjük ki hibaellenőrzéssel.

1.14. Gyakorlat. Módosítsuk a leghosszabb sort kereső program fő rutinját oly módon, hogy helyesen írja ki tetszőlegesen hosszú bemeneti sorok hosszát és a szövegből annyit, amennyi csak lehetséges!

1.15. Gyakorlat. Írjunk programot, amely minden olyan sort megjelenít, amely 80 karakternél

hosszabb!

1.16. Gyakorlat. Írjunk olyan programot, amely eltávolítja a sorvégi szóközöket és tab karaktereket a bemenet minden sorából és törli a teljesen üres sorokat!

1.17. Gyakorlat. Írjunk olyan reverse(s) függvényt, amely megfordítja az s karakterláncot! Használjuk fel ezt a függvényt olyan program megírásához, amely soronként megfordítja a beolvasott szöveget!

1.10. Érvényességi tartomány; külső változók

A main-en belüli változók (line, save stb.) a main saját változói, vagyis a main-re nézve lokálisak. Mivel ezeket a main-en belül deklaráltuk, egyetlen más függvény sem tud közvetlenül hozzájuk férni. Ugyanez mondható más függvények változóiról; például a getline függvényen belüli i változó független a copy i változójától. A függvények lokális változói csak meghívásukkor jönnek létre, és megsemmisülnek, amikor a vezérlés a függvényből kilép. Az ilyen dinamikus lokális változókat ezért - más nyelvek szó használatához hasonlóan - automatikus változóknak nevezzük. A 4. fejezetben tárgyaljuk az ún. static tárolási osztályt, amelyben a lokális változók megtartják az értéküket két függvényhívás között. Minthogy az automatikus változók élettartama arra az időre korlátozódik, amíg a vezérlés a függvényen van, értéküket nem őrzik meg egyik hívástól a másikig, így minden belépéskor explicit módon értéket kell adni nekik. Ha ezt elmulasztjuk, tartalmuk bizonytalan. Az automatikus változók mellett olyan változókat is definiálhatunk, amelyek az összes függvényre nézve külsők, így értékük függvényhívásoktól függetlenül fennmarad. Ezeket a globális változókat bármelyik függvény név szerint elérheti. Globális hozzáférhetőségük miatt a függvények közötti adatátadást argumentumlisták helyett külső változókon keresztül is megoldhatjuk. A külső változókat az összes függvényen kívül kell definiálni: ezzel tárolóhelyet foglalunk le számukra. A változókat minden olyan függvényben, ahol használni akarjuk, vagy explicit módon az **extern** alapszóval, vagy implicit módon értelemszerűen, de deklarálnunk is kell. Mindez bizonyára érthetőbb lesz, ha példaként újra megírjuk a leghosszabb sort kereső programot úgy, hogy a line, a save és a max külső változó legyen. Ehhez mindhárom függvényben meg kell változtatnunk a hívásokat, a deklarációkat és a függvények törzseit.

```
#define MAXLINE 1000 /*A beolvasott sor maximális mérete*/
char line [MAXLINE]; /* A beolvasott sor*/
char save [MAXLINE]; /*A leghosszabb sor mentésére*/
int max; /*Az eddigi maximális hossz*/

/* Speciális változat*/
getline ()
{
    int c, i;
    extern char line [];
    for (i = 0; i < MAXLINE - 1 && (c=getchar ()) != EOF
         && c !='\n'; ++i)
         line (i] = c;

    if (c == '\n') {
         line [i] = c;
         ++i;
    }
    line [i] = '\0';
    return (i);
}

/*Speciális változat*/
copy ()
{
    int i;
    extern char line [], save [];
    i = 0;
    while ((save [i] = line [i]) != '\0')
         ++i;
}

/*A leghosszabb sor kiválasztása; speciális változat*/
main ()
{
    int len; /*A pillanatnyi sor hossza*/
    extern int max;
    extern char save [ ];
    max = 0;
    while ((len = getline ()) > 0)
    if (len > max) {
         max = len;
         copy ();
    }
```

```
    if (max > 0) /*Volt sor*/
         printf ("%s", save);
}
```

Példánkban a main, a getline és a copy függvényben előforduló külső változókat az első sorokban definiáltuk, itt határoztuk meg típusukat és foglaltuk le a szükséges tárterületet. Ezek a külső definíciók ugyanolyan felépítésűek, mint a korábban látott deklarációk, de mivel függvényeken kívül fordulnak elő; külső változókat adnak meg. Függvényben külső változót csak akkor használhatunk, ha előzőleg közöljük a függvénnyel a változó nevét. Ennek egyik módja, hogy a függvényben egy **extern** deklarációt helyezünk el, amely mindössze abban különbözik az eddigi deklarációktól, hogy az **extern** alapszóval kezdődik. Bizonyos körülmények között az **extern** deklaráció elhagyható; ha a forrásszövegben egy változó külső definíciója megelőzi a változó használatát valamely függvényben, akkor e függvényben nincs szükség **extern** deklarációra. Így a main, a getline és a copy függvényben az **extern** deklarációk feleslegesek.

Gyakorlott C-programozók általában a forrásszöveg elején definiálják az összes külső változót, és nem használnak **extern** deklarációkat. Kötelező azonban az **extern** deklaráció, ha forrásprogramunk több állományra tagolódik, és egy változót, mondjuk az A állományban definiálunk, de B-ben használunk, hiszen ilyenkor a változó két előfordulása között csak a B-ben elhelyezett **extern** deklarációval teremthetünk kapcsolatot. Ezt a témát bővebben a 4. fejezetben fejtjük ki.

Nem szabad összetévesztenünk a külső változók deklarációját és definícióját! A definíció az a programsor, ahol a változót ténylegesen létrehozzuk, számára tárhelyet foglalunk le; a deklaráció viszont olyan programrész, ahol csupán leírjuk a változó tulajdonságait, de tárhelyfoglalás nem történik. Megjegyezzük, hogy az ember hajlamos az égvilágon mindent külső változóként megadni, mivel az látszólag egyszerűsíti az adatátadást - az argumentumlisták rövidek, és a változók mindig rendelkezésre állnak, amikor csak akarjuk. Csakhogy a külső változók akkor is ott vannak, ha nem akarjuk! Ez a programozási stílus súlyos veszélyeket hord magában. Az így írt programokban az adatátadások áttekinthetetlenek - a változók váratlanul, sőt a programozó szándékától teljesen eltérő módon megváltozhatnak, és a program igen nehezen módosítható. Emiatt a leghosszabb sort kereső program második változata gyengébb az elsőnél, de hibája az is, hogy a változók nevének rögzítésével két hasznos függvény elveszti általános jellegét.

1.18. Gyakorlat. Az előbbi getline függvény **for** utasításában a feltételvizsgálat meglehetősen ügyetlen. Javítsunk rajta, de úgy, hogy az állomány végén vagy puffertúlcsorduláskor a program az eddigi módon működjön! Biztos, hogy ez a legjobb szervezés?

1.11. Összefoglalás

Az 1. fejezetben áttekintettük a C nyelv legfontosabb elemeit. Ebből a néhány építőelemből is tekintélyes méretű, hasznos programokat írhatunk, és valószínűleg jó gondolat, ha ennek érdekében az olvasó most megfelelő szünetet tart a könyv olvasásában. A z alább következő gyakorlatokban programötleteket szeretnénk adni olyan programokra, amelyek bonyolultabbak mint azok, amelyeket

ez a fejezet bemutatott. Ha az olvasó már elsajátította a C nyelv eddig ismertetett elemeit, folytassa az olvasást, mivel a következő néhány fejezetben olyan jellegzetességekről szólunk, amelyek nagyban hozzájárulnak a nyelv erejéhez és kifejezőképességéhez.

1.19. Gyakorlat. Írjunk detab néven programot, amely a bemeneten talált tab karakterek mindegyikét annyi szóközzel helyettesíti, amennyi a következő tabulátor stop-ig hátra van! Tételezzünk fel egy rögzített tabulátorstop készletet, a stop-ok mondjuk minden n-edik pozíción találhatók.

1.20. Gyakorlat. Írjuk meg az entab programot, amely a szóközökből álló karakterláncok helyébe a minimális számú tab karaktert és szóközt írja úgy, hogy a távolság ne változzon! Használjuk ugyanazokat a tab stop-okat, mint a detab-nál!

1.21. Gyakorlat. Írjunk olyan programot, amely a sor n-edik pozíciója előtt előforduló utolsó, nem szóköz karakter után "összehajtja" a hosszú bemeneti sorokat (n paraméter)! Győződjünk meg róla, hogy a program tényleg értelmesen működik nagyon hosszú sorok esetén, de akkor is, ha a megadott pozíció előtt egyáltalán nem szerepel szóköz vagy tab!

1.22. Gyakorlat. Írjunk olyan programot, amely egy C programból az összes megjegyzést eltünteti! Ne felejtkezzünk meg az idézőjelezett karakterláncok és karakterállandók helyes kezeléséről!

1.23. Gyakorlat. Írjunk olyan programot, amely a C programban megtalálja az olyan alapvető szintaktikai hibákat, mint a nem azonos számú nyitó és záró kerek, szögletes, ill. kapcsos zárójelek! Ne felejtkezzünk meg az aposztrófokról, idézőjelekről , valamint a megjegyzésekről sem! (Ezt a programot teljes általánosságban nehéz elkészíteni.)

2. Típusok, operátorok és kifejezések

A programok alapvető adatobjektumai a változók és az állandók. A deklarációk felsorolják a használni kívánt változókat, közlik a típusukat, valamint az esetleges kezdeti értéküket. Az operátorok azt határozzák meg, hogy mit kell tenni a változókkal. A ki fejezések a változókból és állandókból új értékeket hoznak létre. Fejezetünkben ezekkel foglalkozunk.

2.1. Változónevek

Bár eddig erről nem beszéltünk, a változók és szimbolikus állandók neveire nézve vannak bizonyos megkötések. A nevek betűkből és számjegyekből állnak: az első karakter betű kell, hogy legyen. Az aláhúzás karakter (_) betűnek számít: ezzel javíthatjuk a hosszú változónevek olvashatóságát. A nagy- és a kisbetű különbözőnek számít; a hagyományos C programozási gyakorlat szerint a változónevek kisbetűsek, a szimbolikus állandók csupa nagybetűből állnak. A belső nevekben csupán az első nyolc karakter értékes, bár ennél hosszabb nevek is használhatók. Külső nevek esetén, így függvényneveknél és külső változóknál ez a szám nyolcnál kevesebb is lehet, mivel a külső neveket különféle assemblerek és töltőprogramok (loaderek) is használják. Ennek részleteit az A. függelék ismerteti.

Ezenkívül az olyan kulcsszavak, mint **if**, **else**, int, **float** stb. fenntartott szavak; változónévként nem használhatók. (Kisbetűseknek kell lenniük.) Természetesen ésszerű olyan változóneveket választani, amelyek jelentenek valamit, kapcsolódnak a változó funkciójához, és tipográfiailag nem zavarók.

2.2. Adattípusok és méretek

A C-ben csak néhány alapvető adattípus van:

char egyetlen byte, amely az érvényes karakterkészlet egy elemét tartalmazhatja.

int egész szám, amely tipikusan a befogadó gépre jellemző egész szám ábrázolási méretet tükrözi.

float egyszeres pontosságú lebegőpontos szám.

double kétszeres pontosságú lebegőpontos szám.

Ezen kívül van néhány minősítő szimbólum, amely az **int** mennyiségekre alkalmazható: **short**, **long**, valamint **unsigned**. **short** (rövid), ill. **long** (hosszú) különböző méretű egész számot jelöl. Az **unsigned** (előjel nélküli) számokra a modulo 2n aritmetika szabályai vonatkoznak, ahol n az **int** típust ábrázoló bit-ek száma; az **unsigned** számok mindig pozitívak. A minősítők deklarációjának alakja:

short int x;

long int y;

unsigned int z;

Ilyen esetekben az **int** szó elhagyható, és el is szokás hagyni. Ezeknek az objektumoknak a pontossága a rendelkezésre álló géptől függ; a következő táblázat néhány - bitekben megadott – jellemző értéket mutat.

	DEC PDP-11	Honeywell6000	IBM 370	Interdata 8/32	**x86**
	ASCII	ASCII	EBCDIC	ASCII	ASCII
char	8	8	8	8	8
int	16	36	32	32	32
short	16	36	16	16	16
long	32	36	32	32	32
float	32	36	32	32	32
double	64	72	64	64	64

A cél az, hogy ahol kívánatos, a **short**, ill. a **long** különböző hosszúságú egészeket hozzon létre; **int** általában az adott gépnek megfelelő legtermészetesebb méret. Látható, hogy minden fordító a saját hardverjének megfelelően szabadon értelmezheti a **short**, ill. **long** minősítőket, az azonban bizonyos, hogy a **short** nem hosszabb, mint a **long**.

2.3. Állandók

Az **int** és **float** állandókkal már végeztünk, csupán azt tesszük még hozzá, hogy a szokásos

```
123.456e-7
```

vagy a

```
0. 123E3
```

jelölésmód a **float** számok esetében egyaránt megengedett. Minden lebegőpontos állandó **double**-nak számít, ezért az "e" jelölés a **float** és a **double** számokra egyaránt megfelelő. A **long** állandók írásmódja: 123L. Azok a közönséges egész állandók, amelyek hosszabbak annál, hogy egy int-be beleférjenek, ugyancsak **long**-nak számítanak. Külön jelölésmódja van az oktális és a hexadecimális állandóknak:

ha egy **int** típusú állandó 0-val (nullával) kezdődik, a szám nyolcas (oktális) számrendszerben értendő; a vezető 0x vagy 0X pedig azt jelenti, hogy hexadecimális (tizenhatos számrendszerbeli) számról van szó.

Például a decimális 31 ugyanannyi, mint az oktális 037 vagy a hexadecimális 0x1F, ill. 0X1F. A hexadecimális és oktális állandókból az utánuk írt L-lel szintén képezhetünk **long** mennyiséget.

A karakterállandó egyetlen, aposztrófok közé írt karakter, például 'x'. A karakterállandó értéke a karakternek a gép karakterkészletén belüli numerikus értéke. Például a nulla karakter, vagyis '0' értéke az ASCII karakterkészletben 48, az EBCDIC-ben pedig 240, mindkét érték teljesen különböző a 0 numerikus értéktől.

Ha számértékek, mint 48 vagy 240 helyett '0'-t írunk, akkor a program függetlenné válik a karakter adott értékétől. A karakterállandók ugyanúgy vesznek részt a numerikus műveletekben, mint bármilyen más szám, bár leggyakrabban más karakterekkel való összehasonlításra használjuk őket.

A konverziós szabályokkal egy későbbi fejezet foglalkozik. Bizonyos nemgrafikus karakterek escape szekvenciák segítségével ábrázolhatók karakterállandóként, mint például \n (újsor), \t (tab), \0 (nulla), \\ (fordított törtvonal), \' (aposztróf) stb., amelyek két karakternek látszanak, de valójában mindegyik csak egy karakter.

Ezenkívül tetszőleges, egy byte méretű bit-minta hozható létre a '\ddd' alak segítségével, ahol ddd egy, kettő vagy három oktális számjegy, pl.:

```
#define FORM_FEED '\014' /* ASCII lapdobás karakter*/
```

A '\0' karakterállandó a nulla értékű karaktert jelöli. 0 helyett gyakran írunk '\0'-át, amivel valamely kifejezés karakter jellegét hangsúlyozzuk. Az állandó kifejezés olyan kifejezés, amely csak állandókat

tartalmaz. Az ilyen kifejezések kiértékelése fordítási időben történik, nem pedig futási időben, és így egyszerű állandónak felelnek meg. Például:

```
#define MAXLINE 1000
char line [MAXLINE + 1];
```

vagy

```
seconds = 60 * 60 * hours;
```

A karakterlánc-állandó (string konstans) idézőjelek közé zárt, nulla vagy több karakterből álló sorozat, pl.

```
"ez itt egy karakterlánc"
```

vagy

```
""  /* Üres karakterlánc*/
```

Az idézőjelek nem részei a karakterláncnak, csupán annak határolására szolgálnak. A karakterláncokban ugyanazok az escape szekvenciák használhatók, mint amelyeket a karakterállandóknál láttunk; \" az idézőjel karaktert jelöli. Gyakorlatilag a karakterlánc olyan tömb, amelynek minden eleme egy-egy karakter. A fordító automatikusan elhelyezi a \0 nulla karaktert minden ilyen karakterlánc végére, így a programok kényelmesen megtalálhatják a karakterlánc végét. Ez a fajta ábrázolás azt jelenti, hogy nincs tényleges határa a karakterlánc hosszának, de egy adott karakterlánc hosszának megállapításához a programnak végig kell mennie az illető karakterláncon. A szükséges fizikai tárhely nagysága egy tárhellyel több, mint az idézőjelek közé írt karakterek száma. Az alábbi strlen(s) függvény az s karakterlánc hosszát adja vissza, kizárva ebből a záró \0-t.

```
/* s hosszának kiszámítása*/
strlen (char s [])
{
    int i;
    i = 0;
    while (s [i] != '\0')
         ++i;

    return (i);
}
```

Vigyázat! A karakterállandó és az egyetlen karaktert tartalmazó karakterlánc két különböző dolog: 'x' nem ugyanaz, mint "x". Az előbbi egyetlen karakter, amely az x betűnek a gép karakterkészlete szerint megfelelő számérték előállítására szolgál, az utóbbi egy karakterlánc, amely egy karaktert (az x betűt) és egy \0-át tartalmaz.

2.4. Deklarációk

Használat előtt minden változót deklarálni kell, bár bizonyos deklarációk implicit módon, értelemszerűen keletkeznek. A deklaráció meghatároz egy típust, amelyet az illető típusú változó(ka)t megadó lista követ, mint például:

```
int lower, upper, step;
char c, line [1000];
```

A változók tetszőleges módon oszthatók szét a deklarációk között;

az előző listákat így is írhattuk volna:

```
int lower;
int upper;
int step;
char c;
char line [1000];
```

az utóbbi forma több helyet igényel, de így pl. minden deklarációhoz vagy az azt követő módosításokhoz megjegyzést fűzhetünk. A változók saját deklarációikban inicializálhatók is, bár ezzel kapcsolatban vannak megkötések. Ha a nevet egy egyenlőségjel és egy állandó követi, akkor az az

illető változó kezdeti értékének megadását (inicializálását) jelenti:

```
char backslash = '\\';
int i = 0;
float eps = 1.0e-5;
```

Külső vagy statikus változó esetén az inicializálás csak egyszer - értelemszerűen a program végrehajtásának megkezdése előtt - történik meg. Az explicit módon inicializált automatikus változók minden alkalommal inicializálódnak, amikor az őket tartalmazó függvényt egy program meghívja. Az explicit inicializálás nélküli automatikus változók értéke határozatlan. A külső és statikus változók kezdeti értéke alapértelmezés szerint nulla, de stilárisan helyesebb, ha minden esetben megadjuk a kezdeti értéket. Az inicializálás témáját akkor folytatjuk, amikor a további adattípusokról lesz szó.

2.5. Aritmetikai operátorok

Az aritmetikai operátorok a +, -, *, / és a % (moduló) operátor. Van egyoperandusú -, de nincs egyoperandusú +. Az egész típusú (integer) osztás levágja a tört részt. Az

```
x % y
```

kifejezés az x-nek y-nal történő osztásakor keletkező maradékot jelenti, tehát értéke nulla, ha x pontosan osztható y-nal. Például egy év általában akkor szökőév, ha az évszám 4-gyel osztható, de nem osztható 100-zal. Kivételt jelentenek a 400-zal oszt ható évek, amelyek szintén szökőévek. Így

```
if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
    szökőév van
else
    nincs szökőév
```

A % operátor **float** és **double** mennyiségekre nem alkalmazható. A + és - operátorok precedenciája azonos és alacsonyabb, mint a * , / és % (egymással szintén azonos) precedenciája, amely viszont alacsonyabb, mint az egyoperandusú mínuszé. Az aritmetikai operátorok balról jobbra kötnek. (A 2. fejezet végén közölt táblázat összefoglalja az összes operátor precedenciáját és kötési módját.) A kiértékelés sorrendje olyan asszociatív és kommutatív operátoroknál, mint a * és +, nincs meghatározva; a fordító átrendezheti az olyan zárójelezett számításokat, amelyek ezek valamelyikét tartalmazzák. Így a+(b+c) azonos (a+b)+c-vel. Ennek ritkán van jelentősége, de ha adott sorrendre van szükség, akkor explicit ideiglenes változókat használhatunk. A túlcsordulás és alul-csordulás esetének

kezelése az adott géptől függ.

2.6. Relációs és logikai operátorok

A relációs operátorok:

> >= < <= =

Ezek mindegyikének azonos a precedenciája. Eggyel alacsonyabb – és egymás közt egyező - precedenciájúak az egyenlőségoperátorok:

== !=

A relációs operátorok precedenciája alacsonyabb, mint az aritmetikaiaké, így a várakozásnak megfelelően i < lim - 1 ugyanaz, mint i < (lim - 1). Még érdekesebbek a && és || logikai összekapcsoló műveletek. A && vagy || szimbólumokkal összekapcsolt kifejezések kiértékelése balról jobbra történik, és a kiértékelés azonnal megáll, amint az eredmény igaz vagy hamis volta kiderül. Ezek a tulajdonságok lényegbevágóak, ha jól működő programokat akarunk írni. Itt van például az 1. fejezetben írt getline sorbeolvasó függvény egyik ciklusa:

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

új karakter beolvasása előtt nyilvánvalóan ellenőriznünk kell, hogy a beolvasandó karakter tárolásához van-e elég hely az s tömbben, így az i < lim - 1 vizsgálatot kell elsőként végrehajtani! Sőt, ha a feltétel nem áll fenn, újabb karaktert már nem szabad beolvasni! Ugyancsak nem volna szerencsés, ha a c-nek az EOF-fel történő összehasonlítása a getchar hívása előtt történne meg, a hívásnak meg kell előznie a c-ben található karakter vizsgálatát! && magasabb precedenciájú ||-nél, de mindketten alacsonyabb precedenciájúak, mint a relációs és egyenlőségoperátorok, így az olyan kifejezések, mint

```
i < lim - 1 && (c = getchar()) != '\n' && c != EOF
```

külön zárójeleket nem igényelnek. De mivel a != precedenciája magasabb, mint az értékadásé, a kívánt eredmény elérése érdekében a

```
(c = getchar()) != '\n'
```

kifejezésben zárójelekre van szükség. A ! egyoperandusú negáló operátor a nem nulla, más szóval igaz operandusból 0-t, a nulla, azaz hamis operandusból pedig 1-et csinál. A ! operátort általában olyan

szerkezetekben használják, mint pl.

```
if (! inword),
```

s ezzel helyettesítjük az

```
if (inword == 0)
```

formát. Nehéz általánosságban megmondani, hogy melyik alak a jobb. Az előbbi általában jól olvasható ("ha nem szó belsejében vagyunk"), bonyolultabb esetben azonban nehezen érthető.

2.1. Gyakorlat. Írjunk az előző, **for** ciklussal egyenértékű ciklust, amely a &&-et használja!

2.7. Típuskonverziók

Ha egy kifejezésben különböző típusú operandusok fordulnak elő, a kifejezés kiértékeléséhez az operandusokat azonos típusúakká kell alakítani. Általában csak az értelmes konverziók történnek meg

automatikusan, például egész típusú mennyiségek átalakítása lebegőpontossá olyan kifejezésekben, mint f + i, ahol f **float**, i pedig **int** típusú. Az értelmetlen kifejezések, mint például a **float** indexként való használata, nem megengedettek. A **char** és **int** típusú mennyiségek aritmetikai kifejezésekben szabadon keveredhetnek: a kifejezésekben előforduló minden **char** automatikusan int-té alakul át. Ez nagymérvű rugalmasságot tesz lehetővé bizonyos karaktertranszformációkban. Példa erre az atoi függvény, amely egy számjegyekből álló karakterláncot a megfelelő numerikus értékké alakít át:

```
/ * s egésszé alakítása*/
atoi (char s [])
{
    int i, n;
    n = 0;
    for (i = 0; s [i] >= '0' && s [i] <= '9'; ++i)
         n = 10 * n + s [i] - '0';

    return (n);
}
```

Amint az 1. fejezetben említettük, az

```
s [i] – '0'
```

kifejezés előállítja az s[i] -ben tárolt karakter numerikus értékét, mivel a '0', '1' stb. értékek folytonosan növekvő pozitív sorozatot alkotnak. A char-ból int-té történő átalakítás másik példája az alábbi lower függvény, amely egyetlen karaktert alakít át kisbetűssé, kizárólag ASCII karakterkészlet esetén. Ha a karakter nem nagybetű, a lower változatlanul adja vissza.

```
/*c konvertálása kisbetűssé; csak ASCII*/
lower ( int c)
{
    if (c >= 'A' && c <= 'Z')
         return (c + 'a' – 'A');
    else
         return (c);
}
```

Ez a program csak az ASCII kódkészlet használata esetén működik helyesen, mivel abban a megfelelő kis- és nagybetűk távolsága rögzített, mind a kisbetűs, mind a nagybetűs ábécé numerikus értékei folytonosan követik egymást - A és Z között csak betűk vannak. Az EBCDIC karakterkészletre (IBM 360/370) ez az utóbbi tulajdonság nem érvényes, így lower nem működne helyesen - nem csak betűket konvertálna. A karaktereknek egész számokká történő átalakításával kapcsolatban megemlítjük a nyelv egy finomságát.

A C nyelv nem határozza meg, hogy a **char** típusú változók előjeles vagy előjel nélküli mennyiségek-e. Kérdés tehát, hogy egy char mennyiség int típusúvá alakításakor létrejöhet-e negatív egész is? Sajnos ez az architektúrától függően gépről gépre változik. Bizonyos gépeken (például a PDP-11-en) az olyan char, amelynek legbaloldalibb bitje 1,negatív egésszé alakul át (előjel-kiterjesztés: sign extension). Más gépeken a **char** oly módon válik **int** mennyiséggé, hogy a számítógép a szó bal oldalához nullákat illeszt, és így a keletkező érték mindig pozitív.

A C nyelv definíciója garantálja, hogy a gép szabványos karakterkészletében található egyetlen karakter sem lesz negatív, így ezeket a karaktereket szabadon használhatjuk kifejezésekben pozitív mennyiségekként. Ha azonban más, tetszőleges bit-mintákat tárolunk karakter típusú változókban, azok egyes gépeken pozitív számként, másokon negatív számként jelenhetnek meg. Tipikus példája ennek, amikor EOF-nak a -1 értéket használjuk.

Tekintsük a

```
char c;
c = getchar();
if (c == EOF)
  ...
```

programrészt! Olyan gépen, amely nem végez előjel-kiterjesztést, c mindig pozitív, mivel char-nak deklaráltuk, EOF viszont negatív. Így a feltétel sohasem teljesül. Ennek elkerülése érdekében ügyeltünk arra, hogy minden olyan változót int-nek és ne char-nak deklaráljunk, amely a getchar függvény által visszaadott értéket tartalmaz. Valójában persze nem csak az esetleges előjel-kiterjesztés miatt használunk int-et **char** helyett. Egyszerűen arról van szó, hogy a getchar függvénynek minden lehetséges karaktert vissza kell adnia (oly módon, hogy az bármilyen újabb programbemenethez felhasználható legyen), de vissza kell adnia az ezektől különböző EOF értéket is!

Így a getchar függvény értéke nem jelenhet meg char-ként, hanem azt int-ként kell tárolni. Az automatikus típuskonvertáló másik hasznos formája, hogy a relációs kifejezések (pl. i > j) és az &&, ill. || szimbólumokkal összekapcsolt logikai kifejezések értéke definíciószerűen 1 , ha a kifejezés igaz, ill. 0, ha hamis. Így az

```
isdigit = c >= '0' && c <= '9';
```

értékadás az isdigit változónak az 1 értéket adja, ha c számjegy és a 0 értéket ha nem az. (Az **if**, **while**, **for** stb. feltételvizsgálatában az igaz jelentése egyszerűen: nem nulla.) Az implicit aritmetikai konverziók működése teljesen értelemszerű. Általában, ha egy kétoperandusú operátor, mint a + vagy a * operandusai különböző típusúak, a program a művelet elvégzése előtt az alacsonyabb típusú változót magasabb típusúvá alakítja át. Az eredmény a magasabb típusú. Pontosabban szólva az aritmetikai operátorok az alábbi konverziós szabályok szerint hatnak: A **char** és **short** mennyiségek **int** típusúvá, a **float** mennyiségek **double** típusúvá alakulnak át.

Ezután ha az egyik operandus **double**, a másik is **double** típusúvá alakul át, és az eredmény is **double**. Egyébként ha az egyik operandus **long**, a másik is **long** típusúvá alakul át, és az eredmény is **long** lesz. Egyébként ha az egyik operandus **unsigned**, a másik is **unsigned** típusúvá alakul át, és az eredmény is **unsigned** lesz. Egyébként az operandusoknak **int** típusúaknak kell lenniük, és az eredmény int.

Jegyezzük meg, hogy egy kifejezésben előforduló minden **float** mennyiség **double**-lá alakul át: a C-ben minden lebegőpontos művelet kétszeres pontosságú! Az értékadás is típuskonverzióval jár: a jobb oldal értéke átalakul bal oldali típusúvá, és ez lesz egyben az eredmény típusa is. A karakterek egésszé alakulnak át - akár előjel-kiterjesztéssel, akár anélkül -,amint azt az előbbiekben ismertettük. Az ellentétes irányú művelet, az int-ből char-ba történő konverzió egyértelmű – a felesleges magas helyiértékű bit-ek egyszerűen elmaradnak. Így

```
int i;
char c;
i = c;
c = i;
```

esetében c értéke nem változik. Ez mindig igaz, függetlenül attól, hogy van-e előjel-kiterjesztés vagy sem. Ha x **float** és i int, akkor: x=i valamint i= x egyaránt konverzióhoz vezet; a **float**-ból int-be történő konverzió a tört rész levágását eredményezi. A **double** kerekítéssel alakul át **float**-tá. A hosszabb int-ek rövidebbekké vagy char-okká úgy alakulnak át, hogy a program a felesleges magas helyiértékű bit-eket levágja. Mivel a függvényargumentumok kifejezések, a függvényeknek történő argumentumátadás ugyancsak típuskonverziókkal jár.

Konkrétan a **char** és a **short** int-té válik, a **float** pedig **double** mennyiséggé. Ezért deklaráltuk a függvényargumentumokat int-nek és **double**- nak még akkor is, amikor a függvényt char-ral és **float**-tal hívtuk meg. Végezetül tetszőleges kifejezésben is kiválthatunk, kikényszeríthetünk típuskonverziót, ha ún. típusmódosító (cast) szerkezetet használunk. A (típusnév) kifejezés szerkezetben a kifejezés az előző szabályok alkalmazásával az előírt típusúvá alakul át, úgy mintha a kifejezés hozzá lenne rendelve egy, a megadott típusú változóhoz, amelyet azután az egész szerkezet helyett használunk. Például az sqrt (gyökvonó) könyvtári rutin **double** típusú argumentumot vár, és értelmetlen eredményt ad, ha véletlenül valami mást kap. Ha tehát n egész típusú, akkor

```
sqrt ((double) n)
```

az n-et a sqrt-nek történő átadás előtt **double**-lá konvertálja. (Jegyezzük meg, hogy a típusmódosító szerkezet n értékét a kívánt típusban szolgáltatja; n tényleges tartalma azonban nem változik.)

A típusmódosító operátor precedenciája ugyanaz, mint a többi egyoperandusú operátoré, amint azt a fejezet végén közölt összefoglaló táblázat is mutatja.

2.2. Gyakorlat. Írjuk meg a htoi(s) függvényt, amely egy hexadecimális számjegyekből álló karakterláncot a neki megfelelő egész értékké alakít át! A megengedett számjegyek; 0...9, a...f és A...F.

2.8. Inkrementáló és dekrementáló operátorok

A C nyelv tartalmaz két szokatlan operátort, amelyekkel változók inkrementálhatók és dekrementálhatók. A ++ inkrementáló operátor operandusához 1 -et ad hozzá, a -- dekrementáló operátor pedig 1-et von le belőle. A ++-t gyakran használjuk változók inkrementálására, például:

```
if (c == '\n')
  ++nl;
```

A szokatlanság abban rejlik, hogy a ++ és a – egyaránt használható prefix operátorként (a változó előtt, mint a ++n esetében) vagy postfix operátorként (a változó mögé írva: n++). Az eredmény mindkét esetben n inkrementálása. De míg a ++n kifejezés n-et az előtt növeli, hogy felhasználná annak értékét, n++ csak azt követően inkrementál. Eszerint olyan esetekben amikor nemcsak az inkrementáló tulajdonságot, hanem n értékét is felhasználjuk, ++n és n++ különböznek egymástól. Ha n értéke 5, akkor

```
x = n++;
```

az x-et 5-re állítja, de

```
x = ++n;
```

x-et 6-ra állítja. n mindkét esetben 6 lesz. Az inkrementáló és dekrementáló operátorok csak változókra alkalmazhatók; az olyan kifejezés, mint

```
x = (i + j)++
```

nem megengedett !

Ha az értékre nincs szükség, csak az inkrementáló hatásra, pl.

```
if (c == '\n')
    nl++;
```

esetében, a prefix vagy a postfix operátort tetszés szerint választhatjuk meg. Vannak azonban olyan feladatok, amikor speciálisan az egyikre vagy a másikra van szükség. Tekintsük például a squeeze(s, c) függvényt, amely az összes előforduló c karaktert törli az s karakterláncból:

```
/*Valamennyi c karakter törlése s-ből*/
squeeze (char s [], int c)
{
    int i, j;
    for (i = j = 0; s [i] != '\0'; i++)
    if (s [i] != c)
         s [j++] = s [i];

    s [j] = '\0';
}
```

Minden alkalommal, amikor a program az s karakterláncban c-vel nem azonos karaktert talál, bemásolja azt a pillanatnyi j pozícióba, és csak ezután inkrementálja j-t, hogy fogadhassa a következő karaktert. Hatása pontosan azonos az

```
if (s [i] != c) {
     s [j] = s [i];
     j++;
}
```

alakéval. Hasonló példa fordult elő az 1. fejezetben látott getline függvényben, ahol az

```
if (c == '\n') {
    s [i] = c;
    ++i;
}
```

sorokat az ennél tömörebb

```
if (c == '\n')
    s [i++] = c;
```

alakkal helyettesíthetjük.

Harmadik példánk az strcat(s, t) függvény, amely a t karakterláncot az s karakterlánc végéhez illeszti

(konkatenálja). Az strcat feltételezi, hogy s-ben elég hely van ahhoz, hogy ott az összeillesztett karakterlánc elférjen.

```
/*t illesztése s végéhez*/
/ * s-nek elég nagynak kell lennie * /
strcat ( char s[], t [])
{
    int i, j;
    i = j = 0;
    while (s[i] != '\0')  /*Keresi s végét*/
         while (s [i] = '\ ) / keresi s végét /
              i++;

    while ((s [i++] = t[j++]) != '\0') /*t átmásolása*/
         ;
 }
```

Miközben a program az egyes karaktereket t-ből s-be másolja, a ++ postfix operátor mind i, mind pedig j értékét növeli, hogy azok a következő ciklusban a megfelelő pozícióra mutassanak.

2.3. Gyakorlat. Írjuk meg az squeeze(s1, s2) egy másik változatát, amely s1-ből minden olyan karaktert töröl, amely megegyezik bármelyik s2 beli karakterrel!

2.4. Gyakorlat. Írjuk meg az any(s1, s2) függvényt, amely megadja az s1 karakterláncnak azt a legelső pozícióját, ahol bármelyik, s2 karakterláncbeli karakter előfordul, és -1 értéket szolgáltat, ha s1 egyetlen s2-beli karaktert sem tartalmaz!

2.9. Bitenkénti logikai operátorok

A C nyelvben több bitmanipulációs operátor van; ezek a **float** és **double** típusú változókra nem alkalmazhatók.

- & bitenkénti ÉS,
- | bitenkénti megengedő (inkluzív) VAGY,
- ^ bitenkénti kizáró (exkluzív) VAGY,
- << bitléptetés (shift) balra,
- >> bitléptetés (shift) jobbra,
- ~ egyes komplemens (egyoperandusú).

A bitenkénti ÉS operátort gyakran használjuk valamely bithalmaz maszkolására. Például:

```
c = n & 0177;
```

mindent nulláz, az n kis helyiértékű bitjeinek kivételével. A | bitenkénti VAGY operátorral lehet biteket 1 -re állítani.

```
x = x | MASK;
```

ugyanazokat a biteket állítja 1-be x-ben, mint amelyek 1-be vannak állítva MASK-ban. Gondosan meg kell különböztetnünk az & és | bitenkénti operátorokat az && és || logikai műveletektől, amelyek egy igazságérték balról jobbra történő kiértékelését írják elő. Ha például x értéke 1 és y értéke 2, akkor x & y értéke 0, x&&y értéke pedig 1 .

A << és >> léptető (shift) operátorok bal oldali operandusukon annyi bitléptetést hajtanak végre, ahány bitpozíciót a jobb oldali operandusuk előír. Így x <<2 az x-et két pozícióval balra lépteti, a megürült biteket pedig 0-val tölti fel; ez 4-gyel való szorzással egyenértékű. **unsigned** mennyiség jobbra léptetése esetén a felszabaduló bitekre nullák kerülnek. Előjeles mennyiség jobbra léptetése esetén bizonyos gépeken, így a PDP-11-en a felszabaduló bitekre az előjel kerül (aritmetikai léptetés), m ás gépeken 0 bitek (logikai léptetés). A ~ bináris operátor egész típusú mennyiség 1-es komplemensét képezi, vagyis minden 1-es bitet 0-ra állít és viszont. Ezt az operátort leggyakrabban olyan kifejezésekben használjuk, mint

```
x & ~077
```

amely x utolsó 6 bitjét 0-ra maszkolja. Vegyük észre, hogy x & ~077 független a szóhosszúságtól, és így előnyösebb, mint például x & 0177700, amely feltételezi, hogy x 16 bites mennyiség. A gépfüggetlen alak nem növeli a futási időt, mivel ~077 állandó kifejezés, és mint ilyen, fordítási időben értékelődik ki. Következő programpéldánkban néhány bitoperátor működését szemléltetjük. A getbits(x, p, n) függvény x-nek a p-edik pozíción kezdődő n-bites mezőjét adja vissza (jobbra igazítva). Feltételezzük, hogy a 0 bitpozíció a jobb szélen van és hogy n és p értelmes pozitív értékek. Például getbits (x,4,3) a 4, 3 és 2 pozíción levő három bitet szolgáltatja, jobbra igazítva.

```
/*n bit a p pozíciótól kezdve*/
getbits (unsigned int x, unsigned int p, unsigned int n )
{
    return ((x >> (p + 1 - n)) & ~(~0 << n));
}
```

```
x >> (p + 1 - n)
```

a kívánt mezőt a szó jobb szélére mozgatja. Az x argumentumot **unsigned int** mennyiségnek deklarálva biztosítjuk, hogy a jobbra léptetéskor a felszabaduló bitek ne előjelbitekkel, hanem nullákkal töltődjenek fel, függetlenül attól, hogy a program éppen milyen gépen fut. ~0 csupa 1 bitet jelent, amelyet az ~0 << n utasítás segítségével n bitpozícióval balra léptetve a jobb oldali n biten csupa nullákból álló, a többi pozíción egyesekből álló maszk jön létre. Ezt a ~ operátorral komplementálva olyan maszk keletkezik, amelyben a jobb oldali biteken állnak egyesek.

2.5. Gyakorlat. Módosítsuk a getbits függvényt úgy, hogy a bitpozíciók sorszáma balról jobbra nőjön!

2.6. Gyakorlat. Írjunk olyan wordlength() függvényt, amely kiszámítja a befogadó gép szóhosszúságát, azaz meghatározza, hogy egy **int** mennyiségben hány bit van! A függvény legyen gépfüggetlen, vagyis a forráskód minden gépen működjön!

2.7. Gyakorlat. Írjunk olyan rightrot(n, b) függvényt, amely b számú bitpozícióval jobbra történő bitrotációt végez az n egész típusú mennyiségen!

2.8. Gyakorlat. Írjunk olyan invert(x, p, n) függvényt, amely az x-ben a p pozíciótól kezdve n bitet invertál(vagyis az 1-eseket 0-ra, a 0-kat 1-esekre cseréli fel), miközben a többi bit változatlan marad!

2.10. Értékadó operátorok és kifejezések

Az olyan kifejezések, mint

```
i = i + 2
```

amelyekben a bal oldal a jobb oldalon megismétlődik, a += értékadó operátor segítségével az

```
i += 2
```

tömörített alakban is írhatók. A C-ben a legtöbb kétoperandusú operátornak megvan az op= alakú értékadó megfelelője, ahol op a

+ - * / % << >> & |

szimbólumok egyike. Ha e1 és e2 kifejezés, akkor

e1 op= e2

jelentése:

e1 = (e1) op (e2).

Az egyetlen eltérés, hogy az előbbi esetben a gép e1-et csak egyszer számítja ki. Ügyeljünk az e2 körüli zárójelekre:

```
x *= y + 1
```

jelentése

```
x = x * (y + 1)
```

nem pedig

```
x = x * y + 1
```

Az alábbi példában a bitcount függvény megszámlálja az egész típusú argumentumában található 1-es bitek számát.

```
/*1-es bitek megszámlálása n-ben*/
bitcount (unsigned  int n)
{
    int b;
    for (b = 0; n != 0; n >>= 1)
         if (n & 01)
    b++;
    return (b);
}
```

Tömörségük mellett az értékadó operátoroknak előnye az is, hogy jobban megfelelnek az emberi gondolkodásmódnak. Azt mondjuk:

"adj 2-t i-hez" vagy "növeld i-t 2-vel" (tehát: i += 2),

nem pedig:

"vedd i-t, adj hozzá 2-t majd tedd vissza az eredményt i-be" (i = i+ 2).

Bonyolult kifejezésekben mint

```
yyval [yypv [p3 + p4] + yypv [p1 + p2]] += 2
```

az értékadó_operátor érthetőbbé teszi a kódot, mivel az olvasónak nem kell körülményesen ellenőriznie, hogy két hosszú kifejezés tényleg megegyezik-e; ha pedig nem egyezik meg, nem kell azon tűnődnie, hogy miért nem. Ezenkívül az értékadó operátor még a fordítónak is segíthet a hatékonyabb kód előállításában. Korábban már kihasználtuk azt a tényt, hogy az értékadó utasításnak értéke van és kifejezésekben is előfordulhat; a legközönségesebb példa:

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

Ugyanúgy, a többi értékadó operátort használó értékadások is szerepelhetnek kifejezésekben, bár ezek ritkábban fordulnak elő. Az értékadó kifejezés típusa megegyezik bal oldali operandusának típusával.

2.9. Gyakorlat. 2-es komplemensű aritmetikában x & (x-1) törli x legjobboldalibb 1-es bitjét. (Miért?) Kihasználva ezt a megfigyelést, írjuk meg a bitcount egy gyorsabb változatát!

2.11. Feltételes kifejezések

Az

```
if (a > b)
    z = a;
else
    z = b;
```

feltételes utasítás eredményeként z a és b közül a nagyobbik értékét veszi fel. A C-ben a háromoperandusú ?: operátor segítségével az ilyen szerkezeteket sokkal rövidebben leírhatjuk.

Legyen e1, e2, e3 három kifejezés. Az

```
e1 ? e2 : e3
```

feltételes kifejezésben a gép először e1-et értékeli ki. Ha értéke nem nulla (igaz), akkor e2, egyébként e3 kiértékelése következik, és a kapott érték lesz a feltételes kifejezés értéke. A program e2 és e3 közül tehát csak az egyiket értékeli ki. Így z-b e a és b közül a nagyobbat az alábbi feltételes kifejezéssel tölthetjük:

```
z = (a > b) ? a : b;    /* z = max(a,b) */
```

Megjegyezzük, hogy a feltételes kifejezés is igazi kifejezés, és ugyanúgy használható, mint bármilyen más kifejezés. Ha e2 és e3 különböző típusú, az eredmény típusát a fejezetünkben korábban ismertetett konverziós szabályok határozzák meg. Ha például f **float** és n int, akkor az

```
(n > 0) ? f : n
```

kifejezés **double** lesz, függetlenül attól, hogy n pozitív-e vagy sem. A feltételes kifejezésben az első kifejezést nem kötelező zárójelbe tenni, mivel ?: precedenciája igen alacsony (pontosan az értékadás fölötti). Zárójelezéssel azonban érthetőbbé tehetjük a kifejezés feltételrészét. A feltételes kifejezések használata gyakran tömör és világos kódot eredményez. Az alábbi ciklus például soronként tízesével kinyomtatja egy tömb N elemét oly módon, hogy az egyes oszlopokat egy-egy szóköz választja el, és minden sort (az utolsót is beleértve) pontosan egy újsor karakter zár le.

```
for (i = 0; i < N; i++)
     printf ("%6d %c", a[i],
               (i % 10_== 9 || i == n – 1) ? '\n' : ' ');
```

Minden tizedik és az N-edik elem után egy újsor karaktert ad ki a program. Minden más elemet egy-egy szóköz követ. Gyakorlásképpen próbálja meg az olvasó ugyanezt feltételes kifejezés használata

nélkül leírni!

2.10. Gyakorlat. Írjuk át a lower függvényt, amely a nagybetűs karaktereket kisbetűsekké konvertálja! Az **if-else** helyett használjunk feltételes kifejezést!

2.12. Precedencia; a kiértékelés sorrendje

A következő táblázat összefoglalja valamennyi operátor precedencia- és kötési szabályait, azokét is, amelyekről idáig nem volt szó. Az egy sorba írt operátorok precedenciája azonos; a táblázatban lefelé haladva a precedencia csökken, így például * , / és % precedenciája azonos és magasabb + és - precedenciájánál.

Operátor	Asszociativitás
() []	balról jobbra
! ~ ++ -- - (típus) * & . -> sizeof	jobbról balra
* / %	balról jobbra
+ -	balról jobbra
<< >>	balról jobbra
< <= > >=	balról jobbra
== !=	balról jobbra
&	balról jobbra
^	balról jobbra
|	balról jobbra
&&	balról jobbra
||	balról jobbra
? :	jobbról balra

= += -= stb.	jobbról balra
, (3. fejezet)	balról jobbra

A -> és . operátorok segítségével struktúrák elemeihez férhetünk hozzá, ezekkel, valamint a sizeof (objektum mérete) operátorral a 6. fejezetben foglalkozunk. A * (indirekció) és az & (valaminek a címe) operátorral az 5. fejezetben találkozunk. Ügyeljünk arra, hogy az &, ^ és | bitenkénti logikai operátorok precedenciája kisebb, mint az == és != precedenciája. Emiatt az olyan bitvizsgáló kifejezések, mint

```
if ((x & MASK) == 0)
    . . .
```

a zárójelezés nélkül nem működnek helyesen. Mint említettük, az asszociatív és kommutatív operátorokkal (*, +,&, ^, |) felépített kifejezéseket a fordítóprogram átrendezheti, még akkor is, ha zárójele(ket)t tartalmaznak. Az esetek többségében ennek nincs jelentősége; azokban az esetekben, ahol mégis van,explicit ideiglenes változók használatával gondoskodhatunk a kívánt kiértékelési sorrendről. A legtöbb nyelvhez hasonlóan a C sem határozza meg egy-egy operátor operandusainak kiértékelési sorrendjét. Az

```
x = f () + g ();
```

utasításban pl. nem tudjuk, hogy f-et g előtt vagy g után számítja ki a gép. Így, ha akár f, akár g olyan külső változót módosít, amelytől a másik függ, x értéke függhet a műveletek végrehajtásának sorrendjétől. Ha adott sorrendre van szükségünk, ezt meg **int** csak úgy biztosíthatjuk, hogy a részeredményeket ideiglenes változókban tároljuk. Ugyancsak határozatlan a függvényargumentumok kiértékelési sorrendje, így a

```
printf ("%d %d \n", ++n, power (2,n));  /* ROSSZ */
```

utasítás különböző gépeken különböző eredményeket adhat (és ad is) attól függően, hogy a gép n-et a power hívása előtt vagy után inkrementálja. A helyes megoldás természetesen:

```
++n;
printf ("%d %d \n", n, power(2,n));
```

A függvényhívások, egymásba skatulyázott értékadó utasítások, az inkrementáló és dekrementáló operátorok mellékhatásokat okozhatnak. Ez azt jelenti, hogy egy kifejezés kiszámításának - nem szándékos - melléktermékeként megváltozhat egy változó értéke. A mellékhatásokkal járó kifejezésekben sok függhet attól, milyen sorrendben tárolja a gép a kifejezésben szereplő változókat.

Szerencsétlen, de elég gyakori esetet példáz az

```
a[i] = i++;
```

utasítás. Kérdés, hogy az index i régi vagy új értékével azonos. A válasz különböző lehet aszerint, hogy a fordító hogyan értelmezi, kezeli ezt az utasítást. Mindig a fordító dönti el tehát, lesz-e mellékhatás (módosul-e a változók értéke) vagy sem, hiszen az optimális sorrend erősen függ a gép architektúrájától. A tanulság:

egy nyelven sem szabad olyan programot írni, amelynek eredménye függ a konkrét kiértékelési sorrendtől! Természetesen jó, ha tudjuk, mire vigyázzunk, ugyanakkor, ha nem tudjuk, hogy valami hogyan működik különböző gépeken, ez a tudatlanság meg is védhet bennünket. (A lint nevű C helyességvizsgáló program a legtöbb esetben felfedezi a kiértékelési sorrendtől való függést.)

3. Vezérlési szerkezetek

A nyelv vezérlésátadó utasításai a számítások végrehajtásának sorrendjét határozzák meg. A korábbi példákban már találkoztunk a C leggyakoribb vezérlésátadó utasításaival. Ebben a fejezetben teljessé tesszük a készletet és részletesen ismertetjük a már korábban említett utasításokat is.

3.1. Utasítások és blokkok

A kifejezések, pl. x = 0, i++ vagy printf(. . .) utasítássá válnak, ha pontosvessző követi őket:

```
x = 0;
i++;
printf (. . .);
```

A C-ben a pontosvessző utasításlezáró jel (terminátor) és nem elválasztó szimbólum, mint az ALGOL-szerű nyelvekben. A { és } kapcsos zárójelek felhasználásával deklarációkat és utasításokat egyetlen

összetett utasításba vagy blokkba foghatunk össze. Ez szintaktikailag egyetlen utasítással egyenértékű. Nyilvánvaló példái ennek a függvények utasításait határoló kapcsos zárójelek, vagy azok a zárójelek, amelyek egy if, else, while vagy for szimbólumot követő utasítássort vesznek körül. (Változók bármely blokkon belül deklarálhatók, erről a 4. fejezetben lesz szó.) A blokkot lezáró jobb oldali kapcsos zárójel után soha nincs pontosvessző.

3.2. Az if-else utasítás

Az **if-else** utasítással döntést, választást írunk le. Az utasítás szintaxisa formálisan :

```
if (kifejezés)
    1.utasítás
else
    2.utasítás
```

ahol az **else** rész nem kötelező. A gép a kifejezés kiértékelése után, ha annak értéke igaz (vagyis nemnulla), az 1. utasítást, ha értéke hamis (nulla), és ha van **else** rész, akkor a 2. utasítást hajtja végre. Mivel az **if** egyszerűen a kifejezés numerikus értékét vizsgálja, lehetőség van bizonyos programozási rövidítésre. A legnyilvánvalóbb, ha

if (kifejezés) -t írunk

if (kifejezés != 0)

helyett. Ez néha természetes és világos, máskor viszont nehezen megfejthető. Minthogy az **if-else** konstrukció **else** része elhagyható, sokszor nem egyértelmű, hogy az egymásba skatulyázott **if** utasítások melyikéhez tartozik **else** ág. A kétértelműséget a C más nyelvekhez hasonlóan azzal oldja fel, hogy az **else** a hozzá legközelebbi **else** nélküli **if**-hez kapcsolódik. Például az

```
if (n > 0)
    if (a > b)
         z = a;
    else
         z = b;
```

esetben az **else** a belső **if**-hez tartozik, amint azt a sorbetolással szemléltettük. Ha nem ezt akarjuk, zárójelekkel érhetjük el a helyes összerendelést:

```
if (n > 0) {
    if (a > b)
         z = a;
}
else
     z = b;
```

A kétértelműség különösen veszélyes az olyan esetekben, mint:

```
if (n > 0)
for (i = 0; i < n; i++)
    if (s[i] > 0 ) {
         printf (". . .");
         return (i);  }
else /*ROSSZ*/
      printf("hiba, n értéke nulla \n");
```

A sorbetolás ugyan félreérthetetlenül mutatja, hogy mit akarunk, de ezt a számítógép nem érzékeli, és az **else**-t a belső **if**-hez kapcsolja. Az ilyen típusú hibákat igen nehéz felfedezni. Egyébként vegyük észre, hogy az = a után pontosvessző van az

```
if (a > b )
    z = a;
else
    z = b;
```

programrészben. Ennek az az oka, hogy nyelvtanilag egy utasítás követi az **if**-et, márpedig az olyan kifejezés jellegű utasításokat is, mint z =a mindig pontosvessző zárja le.

3.3. Az else-if utasítás

Az

```
if (kifejezés)
    utasítás
else if (kifejezés)
    utasítás
  else if (kifejezés)
        utasítás
  else
    utasítás
```

szerkezet olyan gyakran fordul elő, hogy megér némi külön fejtegetést. Többszörös elágazást (döntést) általában ilyen **if**-sorozattal valósítunk meg. A gép sorban kiértékeli a kifejezéseket. Ha valamelyik kifejezés igaz, akkor a hozzá tartozó utasítást a gép végrehajtja, és ezzel az egész lánc lezárul. Az egyes utasítások helyén egyetlen utasítás vagy kapcsos zárójelek közé zárt utasításcsoport egyaránt állhat. Az utolsó **else** a "fentiek közül egyik sem" (alapértelmezés szerinti) esetet kezeli. Ha a vezérlés ide kerül, egyetlen korábbi feltétel sem teljesült. Néha ilyenkor semmit sem kell csinálni, így a záró **else** utasítás elhagyható, vagy - valamilyen tiltott feltétel figyelésével - hibaellenőrzésre használható.

Következő példánkban egy háromutas döntést láthat az olvasó. Olyan bináris kereső függvényt mutatunk be, amely egy rendezett v tömbben egy bizonyos x értéket keres. v elemeinek növekvő sorrendben kell követniük egymást. Ha x előfordul v-ben, akkor a függ vény x v-beli (0 és n-1 közötti) sorszámát szolgáltatja, ellenkező esetben értéke -1 lesz:

```
/*x keresése v[0] . . .v[n - 1]-ben*/

binary (int x, int v[], int n)
{
    int low, high, mid;
    low = 0;
    high = n - 1;
    while (low <= high) {
         mid = (low + high) / 2;

    if (x < v[mid])
         high = mid - 1;
    else if (x > v[mid])
         low = mid + 1;
           else / *Megtalálta*/
         return (mid);
    }
    return (-1);
}
```

Minden lépésben meg kell vizsgálni, hogy x kisebb, mint a v[mid] középső elem, nagyobb nála vagy egyenlő vele, ami egészen természetes módon írható le **else-if** szerkezettel.

3.4. A switch utasítás

A **switch** utasítás a többirányú programelágaztatás egyik eszköze. Megvizsgálja, hogy valamely kifejezés értéke megegyezik-e több állandó érték valamelyikével, és ennek megfelelő ugrást hajt végre. Az 1. fejezetben olyan programot láttunk, amellyel az egye s számjegyek, üres és egyéb karakterek előfordulásait számláltuk meg. Ugyanazt a programot most az **if** ... **else if**... ...**else** szerkezet helyett a **switch** utasítással írtuk meg:

```
/*Számjegyek, üres és egyéb karakterek számlálása*/
main ()  {
  int c, i, nwhite, nother, ndigit[10];
  nwhite = nother = 0;
  for (i = 0; i < 10; i++)
        ndigit [i] = 0;

  while ((c = getchar()) != EOF)
    switch (c) {
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
         ndigit [c - '0'] ++;
         break;
    case ' ':
    case '\n':
    case '\t':
         nwhite++;
         break;
    default :
         nother++;
         break;
  }

  printf ("számjegyek=");
  for (i = 0; i < 10; i++)
  {
    printf ("%d", ndigit[i]);
    printf ("\n üres hely = %d, egyéb = %d \n",nwhite, nother);
  }
}
```

A **switch** kiértékeli a zárójelek közötti kifejezést (ebben a programban ez a c karakter), és összehasonlítja az összes **case** (eset) értékével. Minden **case**-t egész értékkel, karakterállandóval vagy állandó kifejezéssel meg kell címkézni. Ha valamelyik **case** azonos a kifejezés értékével, a végrehajtás ennél a **case**-nél kezdődik. A **default** címkéjű **case**-re akkor kerül a vezérlés, ha a többi **case** egyike sem teljesül. A **default** elhagyható : ha nem szerepel és a **case**-ek egyike sem teljesül, semmi nem

történik. A case-ek és a **default** tetszőleges sorrendben követhetik egymást. A **case** utasítások címkéinek különbözniük kell egymástól. A **break** utasítás hatására a vezérlés azonnal kilép a **switch**-ből. Mivel a **case**-ek címkeként működnek, miután valamelyik **case**-hez tartozó programrész végrehajtása befejeződött, a vezérlés a következő **case**-re kerül, hacsak explicit módon nem intézkedünk a kilépésről. A **switch**-ből való kilépés legközönségesebb módja a **break** és a **return**. Ugyancsak **break** utasítással lehet kilépni a **while**, **for** és do ciklusokból, erről e fejezet későbbi részében lesz szó. Az egymást követő **case**-ekbe való belépés nem egyértelműen előnyös. A dolog pozitív oldala, hogy mint példánkban a szóköznél, az újsor és a tab karakternél is láttuk, egyetlen tevékenység számára több esetet enged meg. De ebből az is következik, hogy általában minden case-t **break**-nek kell lezárnia, nehogy a vezérlés a következő **case**-re lépjen. A **case**-ken történő lépkedés azért is veszélyes, mert a vezérlés széteshet, ha a programot módosítjuk. Azokat az eseteket kivéve, amikor ugyanahhoz a számításhoz több címke tartozik, a **case**-ek közötti átmenetek használatával célszerű takarékoskodni. A jó külalak érdekében még akkor is helyezzünk el **break**-et az utolsó eset után (az előző példánkban a **default** után), ha az logikailag szükségtelennek látszik.

Ha valamikor később a szekvencia végéhez újabb **case**-t illesztünk, ez a fajta defenzív programozási taktika fog megmenteni minket.

3.1. Gyakorlat. Írjuk meg azt az expand(s, t) függvényt, amely - miközben az s karakterláncot a t karakterláncba másolja - a láthatatlan karaktereket (pl. újsor és a tab) látható escape szekvenciákká (pl.\n és\t) alakítja át! Használjunk **switch** utasítást!

3.5. A while és a for utasítás

Már találkoztunk a **while** és **for** ciklusokkal. A

```
while (kifejezés)
     utasítás
```

szerkezetben a gép kiértékeli a kifejezést. Ha értéke nem nulla, akkor végrehajtja az utasítást és ismét kiértékeli a kifejezést. Ez a ciklus mindaddig folytatódik, amíg a kifejezés 0 nem lesz, amikor is az

utasítás után a végrehajtás végetér. A

```
for (kif1; kif2; kif3)
     utasítás
```

alakú **for** utasítás egyenértékű a

```
kif1;
while (kif2) {
    utasítás
    kif3;
}
```

alakkal. Nyelvtanilag a **for** mindhárom összetevője kifejezés. Többnyire kif1 és kif3 értékadás vagy függvényhívás, kif2 pedig relációs kifejezés. A három kifejezés bármelyike elhagyható, de a pontosvesszőknek meg kell maradniuk. Ha kif1 vagy kif3 marad el , akkor a ; egyszerűen elmarad a kifejtésből. Ha a kif2 vizsgálat nem szerepel, akkor állandóan igaznak tekintjük, így

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

végtelen ciklus, amelyből valószínűleg más módon kell kiugrani (pl. **return** vagy **break** révén). A **while** és a **for** között lényegében ízlésünk szerint választhatunk. Például a

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ;    /*Átugorja a láthatatlan karaktereket*/
```

programrészben nincs inicializálás, sem újrainicializálás, így a **while** használata a lehető legtermészetesebbnek tűnik. A **for** nyilvánvalóan előnyösebb olyankor, amikor egyszerű inicializálás és újrainicializálás fordul elő, mivel a ciklust vezérlő utasítások egymás közelében, a ciklus tetején jelennek meg. Ez a legszembetűnőbb a

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

esetben, amely egy tömb első N eleme feldolgozásának C nyelvű megfogalmazása, a FORTRAN és PL/1 DO ciklusának megfelelője. Az analógia azonban nem teljes, mivel a **for** határai a cikluson belülről változtathatók, és az i vezérlőváltozó megtartja értékét, a mikor valamilyen oknál fogva a ciklus véget ér. Minthogy a **for** összetevői tetszőleges kifejezések, a **for** ciklus nem korlátozódik aritmetikai léptetésekre. Stiláris szempontból mégis helyesebb, ha a for-ban nem helyezünk el tőle független számításokat; a for-t inkább ciklusvezérlő műveletekre tartsuk fenn. Nagyobb példaként

bemutatjuk az atoi függvény másik változatát. Az atoi függvény egy karakterláncot a neki megfelelő numerikus értékké alakít át. Az itt következő változat a korábbinál általánosabb: kezeli az esetleges vezető szóközöket és az esetleges -vagy + előjelet. (A 4. fejezet tartalmazza az atof függvényt, amely ugyanezt a konverziót lebegőpontos számokra végzi el.) A program alapstruktúrája a bemenet alakját tükrözi:

ugord át az üres közöket, ha vannak

olvasd be az előjelet, ha van

olvasd be az egész részt és konvertáld

Minden lépés elvégzi a maga feladatát, és a dolgokat tiszta állapotban adja át a következő lépésnek. Az egész folyamat az első olyan karakter előfordulásakor ér véget, amely nem lehet része számnak.

```
/*s konvertálása egésszé*/
atoi (char s [])
{
    int i, n, sign;
    for (i = 0; s [i] == ' ' || s [i] == '\n' || s [i] == '\t'; i++)
         ; /*Ugord át az üres helyet*/

    sign = 1;
    if (s [i] == '+' || s [i] == '-') /*Előjelvizsgálat*/
         sign = (s [i++] == '+') ? 1 : -1;

     for (n = 0; s [i] >= '0' && s [i] <= '9'; i++)
         n = 10 * n + s [i] - '0';

    return (sign * n);
}
```

A ciklusvezérlés tömörítésének előnyei még jobban kiütköznek, ha több, egymásba skatulyázott hurok van. A következő függvény az UNIX Shell sort funkcióját valósítja meg:feladata egy egész típusú tömb rendezése. A Shell sort alapgondolata, hogy kezdetben inkább az egymástól távoli elemek kerüljenek összehasonlításra, nem pedig szomszédosak, mint az egyszerű cserélgetős rendezőprogramokban. Ezáltal a nagyfokú kezdeti rendezetlenség várhatóan gyorsan eltűnik, így a későbbi lépéseknek kevesebb dolga akad. A z összehasonlított elemek közötti távolság fokozatosan 1-re csökken, amikor is a rendezés szomszédcserélgetési módszerré alakul át.

```
/*v[0]...v[n-1]-et növekvő sorba rendezi*/
shell (int v[], int n)
{
    int gap, i , j, temp;
    for (gap = n / 2; gap > 0; gap /= 2)
         for (i = gap; i < n; i++)
              for (j = i - gap; j >= 0
                        && v [j] > v[j + gap]; j -= gap) {
                   temp = v [j];
                   v [j] = v [j + gap];
                   v [j + gap] = temp;
              }
}
```

Három egymásba skatulyázott ciklus van. A legkülső ciklus az összehasonlított elemek közötti távolságot vezérli, amit n/2-ről minden ciklusban felére csökkent, amíg a távolság 0 nem lesz. A középső ciklus minden olyan elempárt összehasonlít, amelyek egymástól gap-nyire vannak. A legbelső ciklus minden, nem megfelelő sorrendben levő összehasonlított elempárt megfordít. Mivel gap az utolsó ciklusban 1-re csökken, végül minden elem helyes sorrendbe rendeződik. Vegyük észre, hogy a **for** utasítás általános jellegénél fogva a külső ciklus ugyanolyan alakú, mint a többi, bár nem végez aritmetikai léptetést. Az egyik utolsó C operátor a "," (vessző), amelyet legtöbbször a **for** utasításban használunk. A vesszővel elválasztott kifejezéspárok kiértékelése balról jobbra történik, és az eredmény típusa, ill. értéke megegyezik a jobb oldali operandus típusával, ill. értékével. Így a **for** utasítás egyes részeiben több kifejezést is elhelyezhetünk például azért, hogy párhuzamosan két indexet dolgozzunk fel. Ezt mutatjuk be a reverse(s) függvényben, amely az s karakterláncot helyben megfordítja:

```
/*Az s karakterlánc helyben megfordítása*/
reverse (char s [])
{
    int c, i, j;
    for (i = 0 , j = strlen (s) - 1; i < j; i++ , j--) {
         c = s [i];
         s [i] = s [j];
         s [j] = c;
    }
}
```

A függvényargumentumokat, a deklarációkban előforduló változókat stb. elválasztó vesszők nem vesszőoperátorok, és nem garantálják a balról jobbra történő kiértékelést.

3.2. Gyakorlat. Írjuk meg az expand(s1, s2) függvényt, amely az s1 karakterláncban található

rövidítéseket s2-ben teljes listává bővíti ki (pl. a-z helyett abc. . .xyz-t ír)! Engedjük meg a kis- és a nagybetűket, ill. a számjegyeket is, és készüljünk fel az olyan esetek kezelésére is, mint a-b-c és a-z 0-9 és -a-z! (Hasznos megállapodás, ha a vezető vagy záró - karaktert betű szerint vesszük.)

3.6. A do-while utasítás

Mint az 1. fejezetben mondottuk, mind a **while**, mind a **for** ciklus rendelkezik azzal a kívánatos tulajdonsággal, hogy a kiugrási feltétel teljesülését nem a ciklus végén, hanem a ciklus elején vizsgálja. A harmadik C-beli ciklusfajta, a do-while a vizsgálatot a ciklus végén, a ciklustörzs végrehajtása után végzi el; a törzs tehát legalább egyszer mindenképpen végrehajtódik. A szintaxis:

```
do
    utasítás
while (kifejezés);
```

A gép előbb végrehajtja az utasítást, majd kiértékeli a kifejezést. Ha az értéke igaz, ismét végrehajtja az utasítást, és így tovább. Ha a kifejezés értéke hamissá válik, a ciklus véget ér. Mint várható, a **do-while**-t sokkal ritkábban szokás használni, mint a **while**-t és a for-t, talán az összes ciklusok öt százalékában. Időnként azonban mégiscsak érdemes elővenni, mint például az itt következő itoa függvényben, amely egy számot karakterlánccá alakít át (atoi inverze).A feladat kicsit bonyolultabb, mint gondolnánk, mivel az egyszerű számjegygeneráló módszerek a számjegyeket rossz sorrendben hozzák létre.

Úgy döntöttünk, hogy a karakterláncot visszafelé generáljuk, majd megfordítjuk.

```
/*n karakterré konvertálása s-be*/
itoa ( char s [], int n)
{
    int i, sign;
    if ((sign = n) < 0)      /*előjelvizsgálat és tárolás*/
         n = -n;             /*n pozitív legyen*/

    i = 0;
    do {           /*számjegyek generálása fordított sorrendben */
         s [i++] = n % 10 + '0'; /*megkapja a következő számjegyet*/
    } while ((n /= 10) > 0);               /*törli*/
    if (sign < 0)
         s [i++] = '-';

    s [i] = '\0';
    reverse (s);
 }
```

Példánkban a **do-while** használata tényleg kényelmes, mivel n értékétől függetlenül legalább egy karaktert el kell helyezni az s tömbben. A **do-while** törzsét alkotó egyetlen utasítást - bár itt szükségtelen - kapcsos zárójelek közé zártuk, hogy a sietős olvasó se higgye azt, hogy a **while** egy **while** ciklus kezdete.

3.3. Gyakorlat. Kettes komplemensű számábrázolásban az itoa függvény általunk írt változata nem kezeli a legnagyobb negatív számot, tehát a (2 szóméret-1) értékű n-et. Magyarázzuk meg, hogy miért! Módosítsuk úgy a programot, hogy ezt az értéket is helyesen írja ki, függetlenül attól, hogy milyen gépen fut!

3.4. Gyakorlat. Írjuk meg azt a hasonló itob (n,s) függvényt, amely az n **unsigned** egész számot bináris karakterábrázolásban az s karakterláncba konvertálja! Írjuk meg az itoh függvényt is, amely egy egész számot hexadecimális ábrázolásmódba alakít át!

3.5. Gyakorlat. Írjuk meg az itoa függvénynek azt a változatát, amely kettő helyett három argumentumot fogad! A harmadik argumentum a minimális mezőszélesség; az átkonvertált számot szükség esetén balról üres közökkel kell kitölteni, hogy elég széles legyen.

3.7. A break utasítás

Néha kényelmes, ha a ciklusból való kilépést nem a ciklus elején vagy végén való feltételvizsgálattal vezéreljük. A **break** utasítással a vizsgálat előtt is ki lehet ugrani a **for**, **while** és do ciklusokból, csakúgy, mint a **switch**-ből. A **break** utasítás hatására a vezérlés azonnal kilép a legbelső zárt ciklusból utasítás hatására a vezérlés azonnal kilép a legbelső zárt ciklusból (vagy switch-ből). A következő program az összes sor végéről eltávolítja a szóközöket és tab karaktereket oly módon, hogy **break** utasítás segítségével kilép a ciklusból, amikor a legjobboldalibb nem - szóköz és nem - tab karaktert megtalálja.

```
#define MAXLINE 1000

/*Sorvégi szóközök és tabok eltávolítása*/
main ()
{
    int n;
    char line [MAXLINE];
    while ((n = getline (line,MAXLINE)) > 0) {
         while (--n >= 0)
              if (line [n] != ' ' && line [n] != '\t'
                   && line [n] != '\n')

         break;
         line [n + 1] = '\0';
         printf_("%s \n", line);
    }
 }
```

A getline a sor hosszát adja vissza. A belső **while** ciklus a line utolsó karakterén kezdődik (ne felejtsük el, hogy --n előbb dekrementálja n-et és csak azután használja az értékét), és visszafelé haladva keresi az első olyan karaktert, amely nem szóköz, tab vagy újsor. Ha ilyen karaktert talál, vagy ha n negatívvá válik (vagyis, ha az egész sort megvizsgálta), akkor a ciklus megszakad. Igazolja az olvasó, hogy ez akkor is helyes működés, ha az egész sor csupa üres helyeket megjelenítő karakterekből áll!

A **break** alkalmazása helyett választhatjuk azt a megoldást is, hogy a vizsgálatot magába a ciklusba tesszük:

```
while ((n = getline (line,MAXLINE)) > 0) {
    while (--n >= 0 && (line [n] == ' '
           || line [n] == '\t' || line [n] == '\n'))
         ;
     . . .
}
```

Ez a változat gyengébb, mint az előző, mivel a vizsgálat nehezebben érthető. Általában kerüljük az olyan vizsgálatokat, amelyekben keverednek az &&, ||, ! szimbólumok és a zárójelek.

3.8. A continue utasítás

A continue utasítás a **break**-hez kapcsolódik, de a **break**-nél ritkábban használjuk; a continue-t tartalmazó ciklus (for, **while**, **do**) következő iterációjának megkezdését idézi elő. A while és a do

esetében ez azt jelenti, hogy azonnal végrehajtódik a feltételvizsgálat, a for esetében pedig a vezérlés azonnal az újrainicializálási lépésre kerül. (A continue csak ciklusokra alkalmazható, **switch**-re nem. Az olyan, switch-en belüli continue, ahol a switch egy cikluson belül van, a következő ciklusiteráció végrehajtását váltja ki.) Például a következő programrész az a tömbnek csak a pozitív elemeit dolgozza fel; a negatív értékeket átugorja:

```
for (i = 0; i < n; i++) {
    if (a [i] < 0)  /*Ugord át a negatív elemeket*/
         continue;

    . . .      /*Dolgozd fel a pozitív elemeket*/
}
```

A continue utasítást gyakran használjuk olyan esetekben, amikor a ciklus további része nagyon bonyolult és ezért a feltételvizsgálat megfordítása és egy újabb programszint (sorbetolás) túl mélyen skatulyázná a programot.

3.6. Gyakorlat. Írjunk olyan programot, amely a bemenetét a kimenetére másolja, de ha a bemenetre egymás után többször érkezik ugyanaz a sor, azt csak egyszer nyomtatja ki! (Ez egyszerű változata az UNIX uniq szolgáltatásának.)

3.9. A goto utasítás; címkék

A C-ben is használhatjuk a sokat szidott **goto** utasítást, ugrathatunk címkékre. Elméletileg a **goto**-ra soha sincs szükség, és gyakorlatilag majdnem mindig egyszerűen programozhatunk nélküle is. Ebben a könyvben nem használtunk **goto**-t. Mindazonáltal bemutatunk néhány olyan esetet, ahol a **goto**-knak meg lehet a maguk helye. A leggyakoribb eset, amikor a feldolgozást valamilyen mélyen skatulyázott szerkezet belsejében akarjuk abbahagyni oly módon, hogy egyszerre két, egymásba ágyazott ciklusból lépünk ki. A **break** utasítást közvetlenül nem használhatjuk, mivel az egyszerre csak a legbelső ciklusból ugratja ki a vezérlést. Így például :

```
for ( . . . )
    for ( . . . ) {
         . . .
         if (zavar)
              goto hiba;
         . . .
    }
 hiba:
    számold fel a zavart
```

Ez a fajta szervezés célszerű, ha a hibakezelő program nem triviális és ha a hibák különböző helyeken fordulhatnak elő. A címkék alakja ugyanaz, mint a változóneveké, csak kettőspont követi őket. Ugyanazon a függvényen belül, mint ahol a goto előfordul, bármelyik utasítást megcímkézhetjük. Másik példaként tekintsük azt a problémát, amikor egy kétdimenziós tömb első negatív elemét akarjuk megtalálni. (A többdimenziós tömbökről az 5. fejezetben lesz szó.) Az egyik lehetőség:

```
for (i = 0; i < n; i++)
     for (j = 0; j < m; j ++)
         if (v [i][j] < 0)
              goto found;

/*Nem talált*/
 . . .
found:
     /*Az i, j pozíción megtalálta*/
 . . .
```

Bármely **goto**-t tartalmazó program megírható **goto** nélkül, de esetleg csak megismételt vizsgálatok vagy külön bevezetett változó árán. Például a tömbben való keresés **goto** nélkül :

```
found = 0;
for (i = 0; i < N && !found; i++)
     for (j = 0;j < M && !found; j++)
          found = v[i][j] < 0;

if (found)
     /*i-1, j-1-nél volt*/
  . . .
else
     /*Nem talált*/
  . . .
```

Bár nem kívánunk az ügyben dogmatikusak lenni, kimondjuk : minél kevesebbet használjuk a **goto**-t, annál jobb.

4. Függvények és programstruktúra

A függvények a nagy számítási feladatokat kisebbekre bontják. Így a programozó építhet arra, amit mások már megcsináltak, és nem kell mindent elölről kezdenie. A jól megírt függvények gyakran elrejtik a műveletek részleteit a program azon részei elől, amelyeknek nem is kell tudniuk róluk. Ezáltal

az egész program világosabbá válik, és a változtatások is könnyebben elvégezhetők. A C nyelvet úgy tervezték meg, hogy a függvények hatékonyak és könnyen használhatók legyenek. A C programok általában sok kis méretű függvényt tartalmaznak. Egy program több forrásállományra is tagolódhat. Az állományok külön-külön is fordíthatók, és a könyvtárakban található, már korábban lefordított függvényekkel együtt betölthetők. Ezt a folyamatot most nem tárgyaljuk, mivel a részletek a helyi operációs rendszertől függenek. A legtöbb programozó már ismeri a be- és kivitel céljára szolgáló könyvtári függvényeket (getchar, putchar) és a numerikus számítások könyvtári függvényeit (sin, cos, sqrt). Ebben a fejezetben részletesebben szólunk arról, hogyan írhatunk új függvényeket.

4.1. Alapfogalmak

Kezdetként tervezzünk és írjunk olyan programot, amely a bemenetének minden olyan sorát kinyomtatja, amely adott karakterláncból álló mintát tartalmaz! (Ez speciális esete az UNIX grep segédprogramjának.) Például a "the" minta keresése a

Now is the time /Ideje, hogy

for all good minden jó

men to come to the aid ember segítségére siessen

of their party. embertársainak./

sorokban a

Now is the time

men to come to the aid

of their party.

kimeneti szöveget fogja eredményezni. A feladat alapstruktúrája könnyen felbontható három különálló részre:

```
while (van még sor)
    if (a sor tartalmazza a mintát)
         nyomtatás
```

Bár nyilván elhelyezhetjük az egész programkódot a fő rutinban, mégis az a jobb megoldás, hogy kihasználjuk az előző természetes struktúrát és minden részből egy-egy külön függvényt készítünk. Három kis program könnyebben kezelhető, mint egy nagy, mivel az egymásra nem tartozó részletek a függvényekbe rejthetők és a nem kívánatos kölcsönhatások lehetősége minimális lesz. Mi több, az egyes részek a későbbiekben önmagukban is hasznosak lehetnek.

A **while** (van még sor) feladatot az 1. fejezetben írt getline függvény, a nyomtatás feladatát pedig a szabványos könyvtárban rendelkezésünkre álló printf függvény végzi el. Eszerint csupán azt a rutint

kell megírnunk, amely eldönti, hogy a sor tartalmazza -e a kérdéses mintát. A probléma megoldásának tervét a PL/1-ből "lophatjuk el": az index(s, t) függvény azt az s karakterláncbeli pozíciót vagy indexet adja vissza, ahol a t karakterlánc kezdődik, vagy pedig -1-gyel tér vissza, ha s nem tartalmazza t-t. s-beli kezdőpozícióként 0-t használunk, nem 1-et, mivel a tömbök a C nyelvben a 0 indexszel kezdődnek. Ha a későbbiekben bonyolultabb minta-összehasonlítási feladatot akarunk megoldani, csak az index függvényt kell kicserélnünk; a programkód többi része változatlan marad.

Ennyi tervezés után már gyorsan megírhatjuk a programot. Jól látható, hogyan illeszkednek egymáshoz az egyes részek. Ne dolgozzunk a legáltalánosabb esettel: a keresett minta egyelőre legyen csupa betűből álló karakterlánc. Nemsokára szó lesz a karakter tömbök inicializálásáról, és az 5. fejezetben megmutatjuk, hogyan tehetjük a mintát olyan paraméterré, amelyet a program futása során állítunk be.

Példánk egyben a getline függvény újabb változata is: tanulságos lesz, ha összehasonlítjuk az 1. fejezetbeli változattal!

```
#define MAXLINE 1000

/*Sor beolvasása s-be, visszatérési érték a sorhosszúság*/
getline (char s [], int lim)
{
    int c, i;
    i = 0;
    while (--lim > 0 && (c = getchar ()) != EOF && c !='\n')
         s [i++] = c;

    if (c == '\n')
         s [i++] = c;

     s [i] = '\0';
    return (i);
 }

/*Visszaadja t s-beli indexét; -1 , ha t nincs s-ben*/
index (char s [],  char t [])
{
    int i, j, k;
    for (i = 0; s [i] != '\0'; i++) {
         for (j = i , k = 0; t [k] != '\0'
              && s [j] == t [k];j++ , k++)
              ;

         if (t [k] == '\0')
              return (i);
    }
    return (-1);
}

/* Adott mintára illeszkedő összes sor megkeresése*/
main ()
{
    char line [MAXLINE];
    while (getline (line, MAXLINE) > 0)
         if (index (line, "the") >= 0)

     printf("%s", line);
}
```

Minden függvény az alábbi alakú :

```
név (argumentumlista, ha van)
{
     deklarációk és utasítások, ha vannak
}
```

Mint látható, a különféle részek hiányozhatnak; a legrövidebb függvény :

```
dummy () { }
```

ami semmit sem csinál. (Az ilyen semmit sem csináló függvény gyakran hasznos, ha a programfejlesztés során le akarjuk foglalni egy később megírandó programrész helyét.) A függvénynevet típusnév is megelőzheti, amennyiben a függvény nem egész típusú érték kel tér vissza; erről a következő szakaszban lesz szó. A program lényegében egyedi függvény-definíciók halmaza. A függvények közötti kommunikáció (ebben az esetben) argumentumokkal és a függvények által visszaadott értékekkel történik, de történhet külső változókon keresztül is. A függvények a forrásállományon belül tetszőleges sorrendben fordulhatnak elő, és a forrásprogram több állományra bontható, csak függvényeket nem szabad kettévágni. A hívott függvény meghívójának a **return** utasítás segítségével adhat vissza értéket. A **return** utasítást tetszőleges kifejezés követheti:

return (kifejezés)

A hívó függvénynek jogában áll a visszaadott értéket figyelmen kívül hagyni. Nem szükséges továbbá, hogy a **return** után kifejezés álljon, ez esetben a hívó nem kap vissza semmit. A vezérlés akkor is érték átadása nélkül tér vissza a hívóhoz, ha a végrehajtás a függvény végén eléri a záró jobb oldali kapcsos zárójelet. Ez a megoldás megengedett, de valószínűleg valamilyen bajt jelez, ha a függvény értéket ad vissza az egyik helyről és nem ad értéket egy másikról. Mindenesetre az olyan függvény értéke, amely nem ad vissza értéket, bizonyosan értelmetlen (határozatlan, hulladék). Az ilyen jellegű hibákat a C nyelv lint nevű helyességvizsgáló programja jelzi. A több állományra tagolódó C programok fordításának és betöltésének mechanizmusa rendszerről rendszerre változik. Az UNIX operációs rendszerben pl. az 1. fejezetben említett cc parancs végzi el ezt a feladatot. Tegyük fel, hogy a három függvény három állományban található, amelyeknek a neve main.c, getline.c és index.c. Ekkor a

```
cc main.c getline.c index.c
```

parancs lefordítja a három állományt, az eredményül kapott áthelyezhető formátumú tárgykódot a main.o, getline.o és index.o nevű állományokba helyezi, és betölti őket az a.out nevű végrehajtható állományba. Ha hiba fordul elő, mondjuk a main.c-ben, akkor az illető állomány önmagában újrafordítható és az eredmény betölthető a korábban kapott állományokkal együtt a

```
cc main.c getline.o index.o
```

paranccsal. A cc parancs a ".c", ill. az ".o" névadási konvenciók segítségével különbözteti meg a forrásállományokat (source) a tárgykódot tartalmazó (object) állományoktól.

4.1. Gyakorlat. Írjunk egy rindex(s, t) nevű függvényt, amely t s-beli legjobboldalibb előfordulásának pozícióját adja vissza, ill. -1-et ad, ha t nem fordul elő s-ben!

4.2. Nem egész típusú értékekkel visszatérő függvények

Idáig egyetlen programunk sem tartalmazott a függvény típusára vonatkozó deklarációt. Ennek az az oka, hogy alapértelmezés szerint a függvények implicit módon deklaráltak azáltal, hogy megjelennek valamely utasításban vagy kifejezésben, mint pl.:

```
while (getline (line, MAXLINE) > 0)
```

Ha valamely kifejezésben korábban még nem deklarált név fordul elő, amelyet bal oldali kerek zárójel követ, akkor ezt a gép a szövegkörnyezet alapján függvénynévként deklarálja. Ezenkívül alapértelmezés szerint a függvényről azt feltételezzük, hogy **int** típusú értéket ad vissza. Mivel a **char** kifejezésekben **int** mennyiséggé alakul át, a **char** típussal visszatérő függvényeket sem kell deklarálni. Ezzel az esetek többségét lefedtük, beleértve összes eddigi példánkat is. Mi történik azonban, ha a függvénynek valamilyen más típusú értéket kell visszaadnia? Sok numerikus függvény - mint pl. az sqrt, sin és cos - **double** típusú értéket ad vissza; más speciális függvények más típusokat.

Ezek alkalmazását az atof(s) függvénnyel szemléltetjük, amely az s karakterláncot a neki megfelelő dupla-pontosságú lebegőpontos számmá alakítja. Az atof az atoi kiterjesztése, amelynek több változatát is megírtuk a 2. és 3. fejezetben. Az atof kezeli az esetleges előjelet és tizedespontot , valamint a jelenlevő vagy hiányzó egész, ill. tört részt. (Ez azonban nem nevezhető jó minőségű bemeneti konverziós rutinnak; ilyen rutin megírása több helyet igényelne, mint amit most erre a célra szánunk.)

Először is az atof maga kell, hogy deklarálja az általa visszaadott érték típusát, mivel az nem int. Tekintve, hogy kifejezésekben a float double mennyiséggé alakul át, nincs értelme azt mondanunk, hogy az atof float értéket ad vissza; jól kihasználhatjuk a kétszeres pontosságot, és a függvényt **double** értékkel visszatérőnek deklaráljuk. A típus neve megelőzi a függvény nevét:

```
/*Az s karakterlánc átalakítása double-lá*/
double atof (char s [])
{
     double val, power;
    int i, sign;
    for (i = 0; s [i] == ' ' || s [i] == '\n'
              || s [i] == '\t'; i++)
         ;          /* Üres hely átugrása*/

    sign = 1;
    if (s [i] == '+' || s [i] == '-')       /*Előjel*/
         sign = (s [i++] == '+') ? 1 : -1;

     for (val = 0; s [i] >= '0' && s [i] <= '9'; i++)
         val = 10 * val + s [i] - '0';

    if (s [i] == '.')
         i++;

     for (power = 1 ; s [i] >= '0' && s [i] <= '9'; i++) {
         val = 10 * val + s [i] - '0';
          power *= 10;
    }
    return (sign * val / power);
 }
```

Másodszor is, ugyanilyen fontos, hogy a hívó rutinnak közölnie kell, hogy az atof nem egész értéket ad vissza. A deklarációt a következő primitív kalkulátor-program mutatja (a program épphogy elegendő pl. egy csekkönyv egyenlegének kiszámításához). A program soronként egy-egy számot olvas be, amelyet előjel előzhet meg, a számokat összeadja és az összeget minden beolvasás után kinyomtatja:

```
 #define MAXLINE 100
 /*Primitív kalkulátor*/
 main ()
 {
    double sum, atof();
    char line [MAXLINE];
    sum = 0;
    while (getline (line, MAXLINE) > 0)
         printf ("\t %.2f \n", sum += atof (line));
 }
```

A

```
double sum, atof ();
```

deklaráció értelmében sum **double** típusú változó, és atof olyan függvény, amely **double** értékkel tér vissza. Amennyiben atof nincs mindkét helyen explicit módon deklarálva a C feltételezi, hogy egész típusú értékkel tér vissza, és így értelmetlen válaszokat kapunk. Ha maga az atof és main-beli hívása következetlen módon fordul elő ugyanabban a forrásállományban, ezt a fordító észreveszi. Ha azonban az atof függvényt külön fordítottuk (ami valószínű), az eltérést a gép nem veszi észre, az atof **double** értéket ad vissza, amit a main **int** értékként kezel, és értelmetlen válaszokat kapunk. (A lint kimutatja az ilyen hibát!)

Az atof birtokában elvileg így is megírhatjuk az atoi függvényt (karakterlánc konvertálása int-té):

```
/*Az s karakterlánc átalakítása egész számmá*/
atoi (char s [])
{
    double atof ();
    return (atof (s));
}
```

Figyeljük meg a deklarációk és a **return** utasítás struktúráját. A kifejezés értéke a

```
return (kifejezés)
```

-ben mindig olyan típusúvá alakul át, mint amilyen a függvény típusa, még mielőtt a hívóhoz való visszatérés megtörténne. Így atof értéke, ami **double**, automatikusan **int** típusúvá alakul át a **return**-ben való megjelenéskor, mivel az atoi függvény **int** értékkel tér vissza. (A lebegőpontos érték **int** típusúvá történő konverziója levágja az esetleges tört részt, amint erről a 2. fejezetben szó volt.)

4.2. Gyakorlat. Bővítsük ki atof-ot oly módon, hogy az 123.45e-6 alakú tudományos jelölésmódot is kezelni tudja, ahol a lebegőpontos számot e vagy E és egy esetleges előjellel ellátott kitevő követheti!

4.3. További tudnivalók a függvényargumentumokról

Az 1. fejezetben megtárgyaltuk a nyelvnek azt a tulajdonságát, hogy a függvényargumentumok érték

szerint adódnak át, vagyis a hívott függvény az egyes argumentumoknak nem a címét, hanem a külön ideiglenes másolatát kapja meg. Eszerint a függvény nem képe s befolyásolni a hívó függvényben található eredeti argumentumot. A függvényen belül valójában minden argumentum lokális változó, amely azzal az értékkel inicializálódott, amivel a függvényt meghívták. Ha a függvény argumentumaként tömbnév jelenik meg, a tömb kezdőcíme adódik át; a tömbelemek nem másolódnak át. A függvény az átadott címtől kezdődő indexeléssel megváltoztathatja a tömb elemeit. A tömbök tehát név szerint adódnak át. Az 5. fejezetben elmondjuk, hogyan lehet a mutatókat úgy használni, hogy a hívó függvényekben található nem tömb jellegű változókat is befolyásolni tudjuk.

Megjegyezzük, hogy nincs teljesen kielégítő módszer olyan gépfüggetlen függvények írására, amelyek változó számú argumentumot fogadnak. Nincs ugyanis olyan gépfüggetlen eljárás, amellyel a hívott függvény meg tudná határozni, hogy adott hívás alkalmával ténylegesen hány argumentumot kapott. Így nem tudunk például olyan, igazán gépfüggetlen programot írni, amely ki tudná választani tetszőleges számú argumentum közül a legnagyobbat, amint azt a FORTRAN és a PL/1 max nevű beépített függvénye teszi.

Változó számú argumentum általában biztonságosan használható, ha a hívott függvény nem használ olyan argumentumot, amit ténylegesen nem kapott meg, továbbá ha a típusok használata következetes. A printf, amely a legközönségesebb változó-argumentumszámú C függvény, az első argumentumában található információ alapján határozza meg, hogy még hány argumentum következik és azoknak mi a típusa. Súlyos hiba lép fel, ha a hívó nem ad elegendő számú argumentumot, vagy ha a típusok nem azonosak azzal, ami t az 1. argumentum mond. A printf sem gépfüggetlen, és különböző környezetek esetében módosítani kell. Másik lehetőség, hogy amennyiben az argumentumok ismert típusúak, valamilyen megállapodás szerint, pl. egy speciális argumentumértékkel (ami gyakran a nulla) meg lehet jelölni az argumentumlista végét.

4.4. Külső változók

A C program külső objektumok halmaza. Ezek változók vagy függvények lehetnek. A külső jelzőt a belső fogalommal való szembeállítás kedvéért használjuk, amely utóbbi a függvényeken belül definiált argumentumokat és automatikus változókat írja le. A külső változókat függvényeken kívül definiáljuk, így sok függvény számára elérhetők. Maguk a függvények mindig külsők, mivel a C-ben nem lehet függvényeket más függvényeken belül definiálni. Megállapodás szerint a külső változók egyben globális változók is tehát minden, az ilyen változóra ugyanazzal a névvel történő hivatkozás (még a teljesen külön fordított függvényekből is) ugyanarra a fizikai objektumra történő hivatkozást jelent. Ebben az értelemben a külső változók a FORTRAN vagy PL/1 **extern**al-jainak felelnek meg. Később látni fogjuk, hogyan definiálhatunk olyan külső változókat és függvényeket, amelyek globálisan nem hozzáférhetők, hanem csupán egyetlen forrásállományon belül láthatók. Mivel a külső változók globálisan hozzáférhetők, helyettesíthetik a függvényargumentumokat és a függvények közötti kommunikáció céljait szolgáló visszatérési értékeket. Bármelyik függvény hozzáférhet külső változóhoz az illető változó nevére történő hivatkozással, ha a nevet korábban deklarálták. Ha függvények között nagy számú változót kell megosztani, a külső változók használata kényelmesebb és hatékonyabb, mint a hosszú argumentumlistáké. Amint az 1. fejezetben rámutattunk, ezt az okoskodást fenntartással kell fogadnunk, mivel az ilyen megoldás rontja a program áttekinthetőségét és olyan programokat eredményez, amelyekben sok a függvények közötti adatkapcsolat.

A külső változók használatának második oka az inicializálással kapcsolatos. Különösen lényeges, hogy a külső tömbök inicializálhatók, az automatikus tömbök azonban nem. E fejezet vége felé foglalkozunk az inicializálással.

A harmadik ok - ami miatt külső változókat használunk – érvényességi tartományuk és fennmaradási idejük. Az automatikus változók valamely függvényre nézve belső változók : akkor jönnek létre, amikor a vezérlés belép a rutinba, és megszűnnek az onnan való kilépéskor.

A külső változók viszont állandóan megmaradnak: nem jönnek-mennek, így az egyik függvényhívástól a másikig megtartják értéküket. Ha tehát két függvénynek meg kell osztoznia valamilyen adathalmazon és egyik függvény sem hívja a másikat, gyakran az a legkényelmesebb, ha a közösen használt adatokat külső változókban tartjuk és nem adogatjuk át ide-oda argumentumokon keresztül. Vizsgáljuk tovább ezt a kérdést egy nagyobb példán keresztül! A feladat egy újabb, az előzőnél jobb kalkulátorprogram írása. Ez a program már megengedi a +, -, * , / és =műveleteket. A kalkulátor az infix jelölésmód helyett a fordított lengyel (reverse Polish) jelölésmódot használja, mivel az utóbbi kényelmesebb. (Így működnek pl. a Hewlett Packard gyártmányú zsebszámológépek.) Ebben a jelölésmódban minden operátor az operandusai után áll; az olyan infix kifejezést, mint pl.

```
(1 - 2) * (4 + 5) =
```

úgy írjuk be, hogy:

```
1 2 - 4 5 + * =
```

Zárójelekre nincs szükség. A megvalósítás egészen egyszerű. Minden operandust egy verembe tolunk; operátor érkezésekor a megfelelő darabszámú operandus (kétoperandusú operátorok esetében kettő) kilép a veremből, elvégezzük rajtuk az operátor által meghatározott műveletet, majd az eredményt ismét visszaírjuk a verembe. A fenti esetben pl. előbb 1 és 2 a verembe kerül, majd a helyükbe a kettő különbségét, vagyis -1-et írjuk. Ezután 4-et és 5-öt toljuk a verembe, amelyeket azután az összegük, vagyis 9 helyettesít. Végül a szorzás után -1 és 9 helyére szorzatuk, vagyis -9 kerül a verembe. Az = operátorral kinyomtatjuk a verem legfelső elemét anélkül, hogy onnan elmozdulna (így egy számítás részeredményei is ellenőrizhetők). Bár a verembe tolás és az onnan történő kiléptetés (push és pop) műveletei egyszerűek, mire a hibafigyelést és javítást is hozzáfűzzük, elég hosszú programot kapunk ahhoz, hogy mindent külön függvénybe tegyünk ahelyett, hogy ugyanazt a programkódot ismételgetnénk az egész programon keresztül.

Szükség van továbbá egy külön függvényre, amely beolvassa a következő bemenő operátort vagy operandust. Így a program felépítése:

```
while (a következő operátor vagy operandus és nem az állomány vége)
    if (szám)
         told a verembe
    else if (operátor)
         léptesd ki az operandusokat
         végezd el a műveletet
         told a verembe az eredményt
     else
         hiba
```

Nem döntöttünk még a fő kérdésben - hol legyen a verem, vagyis mely rutinok férhessenek hozzá közvetlenül. Az egyik lehetőség, hogy a vermet a main rutinban tartjuk, és a vermet és a pillanatnyi verempozíciót átadjuk a verembe írást és az onnan történő kiléptetést végző rutinoknak. A main rutinnak azonban nem kell tudnia a vermet vezérlő változókról; csupán a verembe történő írásra és az onnan történő kiléptetésre kell ügyelnie. Ezért úgy döntöttünk, hogy a vermet és a hozzá kapcsolódó információt olyan külső változókkal ábrázoljuk, amelyekhez a push és pop függvények hozzáférhetnek, a main azonban nem. Ezt a megoldást egyszerűen lefordíthatjuk a programozás nyelvére. A főprogram lényegében az operátorok és operandusok típusára vonatkozó nagy **switch**-ből áll, ez talán tipikusabb használata a **switch** utasításnak, mint amit a 3. fejezetben láttunk:

```
#define MAXOP 20   /*Operandus és operátor max.mérete*/
#define NUMBER '0'      /*Szám észlelésének jelzése*/
#define TOOBIG '9'      /*Jelzi, hogy a karakterlánc  túl nagy*/

/*A verem kiürítése*/
clear ()
{
    sp = 0;
}

#define MAXVAL 100          /*Értékverem max. mélysége*/
int sp = 0;                 /*Veremmutató*/
double val [MAXVAL];        /* Értékverem*/

/*f írása az értékverembe*/
double push (double  f)
{
    if (sp < MAXVAL)
         return (val [sp++] = f);
    else {
         printf ("hiba: a verem megtelt\n");
         clear ();
         return (0);
    }
}

/*A legfelső érték kiemelése a veremből*/
double pop ()
{
    if (sp > 0)
         return (val [--sp]);
    else {
         printf ("hiba: a verem üres\n");
         clear ();
         return (0);
    }
}

/*Fordított lengyel logikájú kalkulátor*/
main ()
{
    int type;
    char s [MAXOP];
    double op2, atof(), pop(), push();
```

```
    while ((type = getop (s, MAXOP)) != EOF)
         switch (type) {
         case NUMBER:
               push (atof(s));
              break;
         case ' +' :
              push (pop() + pop());
              break;
         case '*' :
              push (pop() * pop());
              break;
         case '-' :
              op2 = pop ();
              push (pop() - op2);
              break;
         case '/':
              op2 = pop ();
              if (op2 != 0.0)
                        push (pop () / op2);
              else
                        printf ("az osztó nulla\n");
              break;
         case '=':
              printf ("\t %f \n", push(pop()));
              break;
         case 'c':
              clear ();
              break;
         case TOOBIG:
              printf ("%.20s . . .túl hosszú\n", s);
              break;
         default:
              printf ("ismeretlen parancs %c \n", type);
              break;
         }
}
```

A c parancs annak a clear függvénynek a segítségével üríti ki a vermet, amit hiba esetén a push és a pop is használ. A getop függvénnyel rövidesen foglalkozunk. Amint arról az 1. fejezetben már szó volt, egy változó akkor külső, ha az összes függvény törzsén kívül definiáljuk. Így a push, a pop és a clear által használt vermet és veremmutatót e három függvényen kívül definiáltuk. Maga a main azonban nem hivatkozik a veremre és a veremmutatóra - a verem ábrázolását gondosan elrejtettük. Így az = operátorra vonatkozó programrésznek a

```
push (pop ());
```

utasítást kell használnia ahhoz, hogy a verem tetejét a verem megváltoztatása nélkül meg lehessen vizsgálni. Figyeljük meg továbbá, hogy mivel a + és a * kommutatív operátorok, a kiléptetett operandusok kombinálásának sorrendje közömbös, a - és a / operátorok esetében azonban meg kell különböztetni a bal oldali és a jobb oldali operandust.

4.3. Gyakorlat. Az alapvető programkeret megtartásával egyszerűen kibővíthetjük a kalkulátorprogramot. Vezessük be a moduló (_%) és az egyoperandusú mínusz operátorokat! Vezessük be továbbá az erase parancsot, amely törli a verem legfelső elemét! Vez essünk be változónevek kezelését lehetővé tevő parancsokat! (A huszonhat egybetűs változónévre egyszerűen megoldható.)

4.5. Az érvényességi tartomány szabályai

Nem szükséges egyszerre lefordítani a C programot alkotó összes függvényt és külső változót: a program forrásszövege több állományban tárolható, és könyvtárakból már előzőleg lefordított rutinok is betölthetők. Ezzel kapcsolatban két érdekes kérdés merül fel:

- Hogyan lehet a deklarációkat úgy megírni, hogy a fordítás során a változók helyesen deklarálódjanak?
- Hogyan kell elkészíteni a deklarációkat ahhoz, hogy a program betöltésekor az összes részlet helyesen kapcsolódjon össze?

Egy név érvényességi tartománya a programnak az a része, amelyre vonatkozóan a nevet definiáltuk. A függvény elején definiált automatikus változó érvényességi tartománya az a függvény, amelyben a nevet deklaráltuk, és a más függvényekben ugyanilyen n éven létező változókat ez nem érinti. Ugyanez igaz a függvény argumentumaira. A külső változó érvényességi tartománya ott kezdődik, ahol a forrásállományban a változót deklaráltuk és az illető állomány végéig tart. Ha pl. a val, sp, push, pop és clear ebben a sorrendben, egyetlen állományban vannak definiálva, vagyis:

```
int sp = 0;
double val [MAXVAL];
double push (f) { . . . }
double pop () { . . . }
clear () { . . . }
```

akkor a val és sp változók a push, pop és clear függvényekben egyszerűen megnevezésükkel használhatók, és nincs szükség további deklarációkra. Ha viszont egy külső változóra még annak definiálása előtt kell hivatkozni, vagy ha egy külső változót más forrásállományban definiálunk, mint ahol használunk, akkor kötelezően **extern** deklarációt kell alkalmazni. Lényeges, hogy különbséget tegyünk valamely külső változó deklarációja és definíciója között! A deklaráció a változó

tulajdonságait írja le (típusát, méretét stb.), míg a definícióval tárterületet is lefoglalunk. Ha az

```
int sp;
double val [MAXVAL];
```

sorok minden függvényen kívül jelennek meg, akkor definiálják az sp és val nevű külső változókat, tárterületet foglalnak le, és az adott forrásállomány többi része számára deklarációként is szolgálnak.

Másrészt az

```
extern int sp;
extern double val [];
```

sorok deklarálják, hogy sp **int** típusú, val pedig **double** típusú tömb (amelynek méretét máshol határozzuk meg), de ezek a sorok nem hozzák létre a változókat és nem foglalnak le számukra tárterületet. A forrásprogramot alkotó állományok között csupán egyben kell a külső változó definíciójának szerepelnie; a többi állományban **extern** deklarációval biztosítjuk a változó elérését. (A definíciót tartalmazó állományban is lehet **extern** deklaráció.) Külső változót csak definiáláskor lehet inicializálni. A tömbméreteket a definícióban kell megadni, de opcionálisan **extern** deklarációban is szerepelhetnek. Bár az előbbi programban az ilyenfajta szervezés nem valószínű, elképzelhető, hogy a val és sp változókat az egyik állományban definiáljuk és inicializáljuk, míg a push, pop és clear függvényeket egy másikban. Ekkor összekapcsolásukhoz a következő definíciók és deklarációk szükségesek:Az 1. állományban:

```
int sp = 0;  /* Veremmutató*/
double val [MAXVAL];  /* Értékverem*/

A 2. állományban:

extern int sp;
extern double val [];
double push (f) { . . . }
double pop () { . . . }
clear () { . . . }
```

Minthogy a 2. állományban található **extern** deklarációk a három függvény előtt és azokon kívül fordulnak elő, ezért mindegyikükre vonatkoznak, tehát egyetlen deklarációkészlet elegendő lesz az

egész 2. állományhoz. Fejezetünkben szó lesz még a nagyobb programoknál előnyös #include szolgáltatásról, amely lehetővé teszi, hogy csak egyszer írjuk le az **extern** deklarációkat, amelyek azután fordítás közben minden forrásállományba beillesztődnek.

Nézzük most a getop megvalósítását, amely a következő operátort vagy operandust olvassa be. Az alapfeladat egyszerű: a szóközök, tabok és újsorok átugrása. Ha a következő karakter nem számjegy és nem tizedespont, akkor getop visszaadja az illető karakter t. Egyébként összegyűjti a számjegyekből álló karakterláncot (amely tizedespontot is tartalmazhat) és NUMBER-rel tér vissza, jelezve, hogy a bemenetre szám érkezett. A rutin elég bonyolult, mivel arra törekedtünk, hogy azt az esetet is helyesen kezelje, amikor a beolvasott szám túl hosszú. A getop mindaddig számjegyeket olvas be (esetleg közben egy tizedespontot is), amíg azok el nem fogynak, de csupán azokat tárolja , amelyek elférnek. Ha nem volt túlcsordulás, akkor NUMBER-rel és a számjegyek karakterláncával tér vissza. Ha azonban a szám túl hosszú volt, akkor figyelmen kívül hagyja a beolvasott sor hátralevő részét, és így a felhasználó a hiba helyétől kezdve egy szerűen újraírhatja a sort. A függvény a túlcsordulást a TOOBIG-gel való visszatéréssel jelzi:

```
/*A köv. operátor vagy operandus beolvasása*/
getop (char s [], int lim)
{
    int i, c;
    while ((c = getch()) == ' ' || c == '\t' || c == '\n')
         ;

    if (c != '.' && (c < '0' || c > '9'))
         return (c);

    s [0] = c;
    for (i = 1; (c = getchar()) >= '0' && c <= '9'; i++)
         if (i < lim)
              s [i] = c;

    if (c == '.') {                    /*A tört rész beolvasása*/
         if (i < lim)
              s [i] = c;

          for (i++; (c = getchar()) >= '0' && c <= '9'; i++)
              if (i < lim)

          s [i] = c;
    }
    if (i < lim) {                      /*A szám rendben van*/
         ungetch(c);
         s [i] = '\0';
         return (NUMBER);
    }
    else {               /*Túl nagy, a sor többi részét átugorja*/
         while (c != '\n' && c != EOF)
              c = getchar();

         s [lim - 1] = '\0';
         return (TOOBIG);
    }
}
```

Mit jelent getch és ungetch? Gyakran az a helyzet, hogy a bemenetet olvasó program csak akkor jön rá, hogy eleget olvasott, amikor már a kelleténél több karaktert olvasott be. Ilyen eset pl., amikor egy számot alkotó karaktereket kell beolvasni: amíg a program nem észleli az első nem-számjegyet, a szám nem teljes. Ehhez azonban a programnak a szükségesnél eggyel több karaktert kell beolvasnia, egy olyan karaktert, amelyre nincs felkészülve.

Valahogy tehát nem beolvasottá kellene tenni a nem kívánt karaktert. Amikor a program a

szükségesnél eggyel több karaktert olvasott be, vissza kellene helyezni azt a bemenetre, így a program a továbbiakban úgy viselkedhetne, mintha ezt a felesleges karaktert sohasem olvasta volna be. Szerencsére mindezt két, egymással együttműködő függvény megírásával könnyen megoldhatjuk. getch szállítja a következő megvizsgálandó bejövő karaktert; ungetch visszaír egy karaktert a bemenetre oly módon, hogy a következő getch hívás ismét ezt a karaktert szolgáltatja. Az együttműködés módja egyszerű.

Az ungetch a felesleges karaktereket egy megosztott pufferbe - egy karaktertömbbe -írja vissza. A getch kiolvassa a puffert, amennyiben abban van valami, ill. ha üres, meghívja a getchar függvényt.

Szükség van egy olyan indexváltozóra is, amely az éppen vizsgált karakter pufferbeli pozícióját mutatja. Mivel a getch és az ungetch a puffert és az indexet közösen használja, az utóbbiaknak a hívások között meg kell tartaniuk értéküket, mindkét rutinra nézve külső változóknak kell lenniük. Így a getch, ungetch és az általuk megosztva használt változók az alábbi módon írhatók:

```
#define BUFSIZE 100
char buf [BUFSIZE];          /*Az ungetch puffere*/
int bufp = 0;                /*A következő szabad pozíció buf-ban*/

/*Kiolvas egy (esetleg visszaírt)  karaktert*/
getch ()
{
    return ((bufp > 0) ? buf [--bufp] : getchar());
}

/* Karakter visszahelyezése a bemenetre*/
ungetch (int c)
{
    if (bufp > BUFSIZE)
         printf("ungetch, túl sok karakter\n");
    else
         buf [bufp++] = c;
}
```

A pufferbe történő visszaírásra egyetlen karakter helyett tömböt használtunk, mivel ez az általánosítás a későbbiekben még jól jöhet.

4.4. Gyakorlat. Írjuk meg az ungets(s) nevű rutint, amely egy teljes karakterláncot ír vissza a bemenetre! Szükséges, hogy az ungets függvénynek tudomása legyen buf-ról és bufp-ről, vagy csak egyszerűen használja az ungetch függvényt?

4.5. Gyakorlat. Tegyük fel, hogy sohasem helyezünk vissza egynél több karaktert. Módosítsuk a getch és ungetch függvényeket ennek megfelelően !

4.6. Gyakorlat. getch és ungetch rutinjainak a visszahelyezett EOF-ot gépfüggő módon kezelik.

Határozzuk meg, hogyan viselkedjenek rutinjaink, amikor EOF-ot írunk vissza, majd valósítsuk meg ezt a megoldást!

4.6. Statikus változók

A már korábban megismert **extern** és automatikus változók mellett a statikus (static) változók jelentik a harmadik tárolási osztályt. A static változók akár belsők, akár külsők lehetnek. A belső static változók ugyanúgy lokálisak valamely függvényre nézve, mint az automatikus változók, de az automatikusaktól eltérően állandóan fennmaradnak és nem jönnek létre, ill. szűnnek meg a függvény minden egyes aktivizálása alkalmával. Eszerint a belső static változók a függvényen belül saját, állandó tárat képeznek . A függvényeken belül megjelenő karakterláncok, mint pl. a printf argumentumai, belső static változók. A külső static változó annak a forrásállománynak a további részében lesz ismert, amelyben deklarálták, de érvényességi tartománya nem terjed ki egyetlen más állományra sem. A külső static változók segítségével lehetőségünk van arra, hogy az olyan neveket mint buf és bufp elrejtsük a getch-ungetch kombinációban. A változóknak külsőknek kell lenniük ahhoz, hogy megoszthatók legyenek, ugyanakkor rejtve kell maradniuk a függvények felhasználói elől, mivel így kizárjuk a konfliktus lehetőségét. Ha a két rutint és a két változót egyetlen állományba szerkesztjük:

```
static char buf [BUFSIZE];  /*Az ungetch puffere*/
static int bufp = 0  /*A következő szabad pozíció  buf-ban*/
getch () { . . . }
ungetch (c) { . . . }
```

akkor egyetlen más rutin sem férhet hozzá a buf és bufp változókhoz; de nem kerülhetnek összeütközésbe a változók az ugyanezen program más állományaiban előforduló ugyanilyen nevekkel sem. A statikus tárolást, legyen az akár belső, akár külső, úgy definiáljuk, hogy a közönséges deklaráció elé a static szót írjuk. A változó külső, ha az összes függvényen kívül, ill. belső, ha valamelyik függvényen belül definiálják. A függvények általában külső objektumok, a nevük globálisan ismert. Ugyanakkor a függvények static típusúnak is deklarálhatók, az ilyen függvények neve ismeretlen lesz azon az állományon kívül, ahol deklarálták. A C nyelvben a static deklaráció nem csupán állandóságot rejt magában, hanem bizonyos mértékű elzártságot is. A belső static objektumok csupán az adott függvényen belül ismertek; a külső static objektumok (változók vagy függvények) pedig csak abból a forrásállományból hozzáférhetők, amelyben megjelennek, és neveik nem kerülnek összeütközésbe a más állományokban előforduló ugyanilyen nevű változókkal, ill. függvényekkel. Külső static változók és függvények segítségével elrejthetjük az adatobjektumokat és a velük dolgozó belső rutinokat, így más rutinok és adatok ezekkel még véletlenül sem kerülhetnek összeütközésbe.

A getch és az ungetch függvény pl. karakterbeolvas és -visszaíró modult alkot; buf és bufp pedig static kell, hogy legyen ahhoz, hogy kívülről ne legyen elérhető. Hasonlóképpen push, pop és clear egy veremkezelő modult alkot; val-nak és sp-nek ugyancsak külső static-nak kell lennie.

4.7. Regisztervváltozók

A negyedik és egyben utolsó tárolási osztály neve **register**. A **register** deklaráció közli a fordítóprogrammal, hogy a kérdéses változóra nagyon gyakran történik hivatkozás. Amennyiben lehetséges, a register típusú változók a gép regisztereibe kerülnek, miáltal rövidebb és gyorsabb programok jönnek létre. A **register** deklaráció alakja:

```
register int x;
register char c;
```

és így tovább; az **int** rész elhagyható. A **register** deklaráció csupán automatikus változókra, valamint függvények formális paramétereire alkalmazható. Utóbbi esetben a deklaráció alakja:

```
f (register int c, register int n)
{
    register int i;
    . . .
}
```

A gyakorlatban a regisztervváltozókra nézve olyan megszorítások állnak fenn, amelyek az adott hardver tulajdonságait tükrözik. Az egyes függvényeknek csupán néhány változója tárolható regiszterekben és csak bizonyos típusok megengedettek. A fölös számú, ill. meg nem engedett deklarációk esetében a **register** szót a gép figyelmen kívül hagyja. Nem hivatkozhatunk továbbá valamely regisztervváltozó címére (ezzel a témával az 5. fejezetben foglalkozunk). A speciális megkötések gépről gépre változnak: például a PDP- 11 számítógépen csupán a függvényen belüli első három regiszterdeklaráció hatásos, a típus pedig int, char vagy mutató lehet.

4.8. Blokkstruktúra

A PL/1, ill. az ALGOL értelmében a C nem blokk-struktúrált nyelv, amennyiben függvények nem definiálhatók más függvények belsejében. Változókat azonban definiálhatunk blokk-struktúrált módon. Nyitó kapcsos zárójel után - amely nemcsak függvény, hanem mindenfajta összetett utasítás kezdetét jelzi - változódeklarációk és inicializálások egyaránt állhatnak. Az ily módon deklarált változók felülbírálják a külső blokkokban ugyanilyen név alatt előforduló változókat és a vonatkozó jobb oldali kapcsos zárójelig érvényben maradnak. Pl. az

```
if (n > 0) {
    int i; /* új i deklarálása*/
    for (i = 0; i < n; i++)
         . . .
}
```

programban az i változó érvényességi tartománya az **if** utasítás igaz ága; ennek az i-nek semmi köze a programban előforduló bármely egyéb i-hez. A blokkstruktúra külső változókra is alkalmazható. Ha adottak az

```
int x;
f ()
{
    double x;
    . . .
}
```

deklarációk, akkor az f függvényen belül az x előfordulásai a belső, **double** típusú változóra, f-en kívül a külső **int** változóra vonatkoznak. Ugyanez igaz a formális paraméterek neveire :

```
int z;
f (double z)
{
  . . .
}
```

Az f függvényen belül z a formális és nem a külső paraméterre vonatkozik.

4.9. Inicializálás

Az inicializálásról futólag már többször is szóltunk, de mindig csak mellékesen valamely más téma kapcsán. Ebben a fejezetben összefoglaljuk a szabályok egy részét, miután már megtárgyaltuk a különféle tárolási osztályokat. Explicit inicializálás hiányában a külső és a statikus változók kezdeti értéke garantáltan nulla lesz; az automatikus és a regiszterváltozók értéke határozatlan. Az egyszerű változók (nem a tömbök és a struktúrák) a deklarálásukkor inicializálhatók oly módon, hogy a nevüket egyenlőségjel és egy állandó kifejezés követi :

```
int x = 1;
char squote = '\'';  /*Aposztróf*/
long day = 60 * 24;  /*Percek száma a napban*/
```

Külső és statikus változók esetében az inicializálás egyszer, mégpedig értelemszerűen a fordítási időben történik meg. Az automatikus és a regisztervváltozók minden alkalommal inicializálódnak, amikor a vezérlés belép a függvénybe vagy blokkba. Automatikus és regisztervváltozók esetében az inicializálás jobb oldalán nemcsak egy állandó állhat -tetszőleges, korábban definiált értékeket, akár függvényhívásokat tartalmazó kifejezés is megengedett. A 3. fejezetben említett bináris keresőprogram kezdeti érték beállításai pl. a következő módon írhatók :

```
binary (int x, int v [], int n)
{
    int low = 0;
    int high = n-1;
    int mid;
    . . .
}
```

a már látott:

```
binary (int x, int v [], int n)
{
    int low, high, mid;
    low = 0;
    high = n - 1;
    . . .
}
```

alak helyett. Az automatikus változók inicializálásai valójában értékadó utasítások rövidített formái. Lényegében csupán ízlés kérdése, hogy valaki melyik alakot részesíti előnyben. Általában explicit értékadásokat használtunk, mivel a deklarációkban előfordul ó inicializálások nehezebben követhetők. Automatikus tömbök nem inicializálhatók. Külső és statikus tömbök úgy inicializálhatók, hogy a deklarációt a kezdeti értékek kapcsos zárójelek közé zárt és vesszőkkel elválasztott listája követi. Az 1. fejezetben ismertetett karakterszámláló program, amelynek kezdete

```
/*Számjegyek, üres közök és egyebek számlálása*/
main ()
{
    int c, i, nwhite, nother;
    int ndigit [10];
    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
         ndigit [i] = 0;

    . . .
}
```

volt, ehelyett így is írható:

```
int nwhite = 0;
int nother = 0;
int ndigit [10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, } ;

/*Számjegyek, üres közök és egyebek számlálása*/
main ()
{
    int c, i;
    . . .
}
```

Az adott esetben ezek az inicializálások szükségtelenek, mivel mindegyik kezdeti érték nulla, ennek ellenére célszerű explicit alakban megadni őket. Ha a megadott méretnél kevesebb számú kezdeti érték van, akkor a többi kezdeti érték nulla lesz. Túl sok kezdeti érték megadása hibát jelent. Sajnos nincs lehetőség valamely kezdeti érték ismétlődésének megadására, sem pedig arra, hogy a tömb valamely közbenső elemét az összes többi kezdeti érték megadása nélkül inicializáljuk. A karaktertömbök a kezdetiérték-beállítás speciális esetét jelentik: a kapcsos zárójelekkel és vesszőkkel történő jelölésmód helyett karakterlánc is használható :

```
char pattern [] = "the";
```

Ez rövidítése a hosszabb, de ezzel egyenértékű írásmódnak:

```
char pattern [] = { 't', 'h', 'e', '\0'};
```

Ha egy - tetszőleges típusú - tömb méretét elhagyjuk, a fordító a tömb-hosszúságot a megadott kezdeti értékek darabszámából számítja ki. A fenti esetben a méret négy (három karakter és a záró \0).

4.10. Rekurzió

A C megengedi a függvények rekurzív használatát, vagyis a függvények (közvetlenül vagy közvetve) saját magukat is hívhatják. Ennek hagyományos példája valamely számnak karakterláncként történő nyomtatása. Amint korábban említettük, a számjegyek rossz sor rendben generálódnak: a kis helyiértékű számjegyek a nagyobb helyiértékű számjegyek előtt jönnek létre, de a nyomtatás fordított sorrendben kell, hogy történjék. A problémának két megoldása van. Az egyik megoldás szerint a számjegyeket a generálás sorrendjében tároljuk egy tömbben, majd fordított sorrendben nyomtatjuk ki őket, ahogy ezt a 3. fejezetben az itoa függvény tette. A printd első változata ezt a megoldást követi.

```
/*n nyomtatása decimális alakban*/
printd (int n)
{
    char s [10];
    int i;
    if (n < 0) {
         putchar ('-');
         n = -n;
    }
    i = 0;
    do {
         s [i++] = n % 10 + '0';  /*Veszi a következő karaktert*/
    } while ((n /= 10) > 0);           /*Elhagyja*/
    while (--i >= 0)
         putchar (s [i]);
}
```

A másik lehetőség a rekurzív megoldás, amelyben printd minden hívásakor először saját magát hívja meg - feldolgozza az összes vezető számjegyet, majd kinyomtatja az utolsó számjegyet.

```
/*n nyomtatása decimális alakban (rekurzív)*/
printd (int n)
{
    int i;
    if (n < 0) {
         putchar ('-');
         n = -n;
    }
    if ((i = n / 10) != 0)
         printd (i);

     putchar (n % 10 + '0');
}
```

Amikor a függvény önmagát rekurzív módon meghívja, minden hívás az automatikus változók friss készletét kapja meg, függetlenül a korábbi készlettől. Így printd(123) esetében az első printd függvényben n = 123. Átadja a 12-t az újabb printd függvénynek, é s 3-at nyomtat ki annak visszatérése után. Ugyanígy a második printd 1 -et ad át a harmadiknak (amely kinyomtatja), majd kiírja a 2-t. A rekurzió általában nem gyorsabb, és tár-megtakarítást sem jelent, mivel valahol létre kell hozni egy vermet a feldolgozott értékek számára. A rekurzív programkód azonban tömörebb és gyakran könnyebben leírható és megérthető. Mint a 6. fejezetben látni fogjuk, a rekurzió különösen kényelmes a rekurzív módon definiált adatstruktúrák, pl. a fastruktúrák esetében.

4.7. Gyakorlat. A printd-ben alkalmazott megoldások felhasználásával

írjuk meg az itoa rekurzív változatát, vagyis rekurzív rutin segítségével konvertáljunk egy egész számot karakterlánccá!

4.8. Gyakorlat. Írjuk meg az s karakterláncot megfordító reverse(s) függvény rekurzív változatát!

4.11. A C előfeldolgozó

A C nyelv egy egyszerű makro-előfeldolgozó (preprocesszor) segítségével bizonyos nyelvi kiterjesztéseket is nyújt számunkra. E kiterjesztések közül a legközönségesebb a már látott #define, de ide tartozik az a nyelvi eszköz is, amely állományok tartalmának a fordítás során történő beiktatását teszi lehetővé.

4.11.1. Állományok beiktatása

A #define szimbólumok, deklarációk és más nyelvi objektumok kezelését is megkönnyíti a C állománybeiktatási szolgáltatása. Minden sor, amelynek alakja:

```
#include "állománynév"
```

az állománynév nevű állomány tartalmával helyettesítődik. (Az idézőjelek kötelezők!) Gyakran minden forrásállomány elején megjelenik egy vagy két ilyen alakú sor, amely a közös #define utasításoknak és a globális változók **extern** _deklarációinak beiktatás ára szolgál. A #include parancsok egymásba skatulyázhatók. Az #include a legelőnyösebb módja annak, hogy egy nagy méretű program deklarációit összegyűjtsük. E megoldás eredményeképpen az összes forrásállomány ugyanazokat a definíciókat és változódeklarációkat kapja meg, és ezáltal különösen kellemetlen hibákat kerülhetünk el. Természetesen olyankor, amikor valamelyik beiktatott állomány megváltozik, az összes ettől függő állományt újra le kell fordítani.

4.11.2. Makrohelyettesítés

A

```
#define YES 1
```

alakú definícióval a legegyszerűbb típusú makrohelyettesítést valósítjuk meg - egy nevet egy karakterlánccal helyettesítünk. A #define-ban előforduló nevek alakja azonos a C azonosítókéval; a helyettesítő szöveg tetszőleges. A helyettesítő szöveg általában a sor további része; hosszú definíciók úgy folytathatók, hogy a folytatandó sor végére \-t helyezünk el. A #define szimbólummal definiált név érvényességi tartománya a definíció helyétől az adott forrásállomány végéig tart. A nevek újradefiniálhatók; a definíciók korábbi definíciókat használhatnak. Idézőjelek közé írt karakterláncokra nézve, tehát amikor pl. YES definiált név, a

```
printf("YES")
```

-ben nem történik helyettesítés. Minthogy a #define-t egy makro-előfeldolgozó, nem pedig maga a fordító dolgozza fel a definíciókra csak nagyon kevés nyelvtani megkötés vonatkozik. Az ALGOL hívei pl. a

```
#define then
#define begin {
#define end; }
```

definíciók után azt írhatják, hogy:

```
 if (i > 0) then begin
    a = 1;
    b = 2;
end
```

Lehetőség van argumentumokkal rendelkező makrók definiálására, amikor is a helyettesítő szöveg a makró hívásának módjától függ. Példaként definiáljuk a max nevű makrót az alábbi módon :

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Ekkor az

```
x = max(p+q, r+s);
```

sort az

```
x =((p + q) > (r + s) ? (p + q) : (r + s));
```

sor fogja helyettesíteni. Olyan maximum függvényt kaptunk tehát, amelynek nem függvényhívás, hanem soron belüli programkód a kifejtése. Ha az argumentumokat következetesen kezeljük, ez a makro bármilyen adattípusra meg fog felelni: nem szükséges különféle max-okat készítenünk a különböző adattípusokra, mint ahogy azt függvényhívások esetében tennénk. Amennyiben közelebbről megvizsgáljuk a max előző kifejtését, találhatunk benne néhány buktatót. A kifejezések kétszer értékelődnek ki; ez nem jó olyankor, amikor mellékhatásuk pl. függvényhívás vagy inkrementáló operátorok alkalmazása. Ügyelnünk kell továbbá a zárójelek használatára, nehogy megváltozzon a kiértékelés sorrendje! (tekintsük a

```
#define square(x) x * x
```

makrót, amikor azt square(z + 1) alakban hívjuk.) Vannak tisztán jelölésbeli problémák is: nem szabad, hogy a makro neve és az argumentumlistáját bevezető bal oldali zárójel között szóköz legyen! Mindezek ellenére a makrók igen hasznosak. Erre nézve gyakorlati példa a 7. fejezetben ismertetendő szabványos be- és kiviteli (I/O) könyvtár, amelyben a getchar és putchar függvényeket makrókként definiáljuk (putchar nyilván argumentumot igényel). Ezáltal elkerüljük azt a plusz terhelést, amit a minden egyes feldolgozandó karakter esetében történő függvényhívás jelentene. A makroprocesszor további szolgáltatásait az "A" függelék ismerteti.

4.9. Gyakorlat. Definiáljuk a swap(x, y) makrót, amely megcseréli a két **int** típusú argumentumát! (A blokkstruktúra segítségünkre lesz.)

5. Mutatók és tömbök

A mutató (pointer) olyan változó, amely egy másik változó címét tartalmazza. A C-ben gyakran használunk mutatókat, részben azért, mert néha csupán így tudunk kifejezni valamilyen számítást, részben pedig azért, mert használatuk általában tömörebb és hatékonyabb kódot eredményez, mint amelyet más módon kaphatnánk. Azt szokták mondani, hogy a mutató, csakúgy, mint a **goto** utasítás, csak arra jó, hogy összezavarja és érthetetlenné tegye a programot. Ez biztos így is van, ha ész nélkül használjuk, hiszen könnyűszerrel gyárthatunk olyan mutatókat, amelyek valamilyen ne m várt helyre mutatnak. Kellő önfegyelemmel azonban a mutatókat úgy is alkalmazhatjuk, hogy ezáltal programunk világos és egyszerű legyen. Ezt kíséreljük meg bemutatni a következőkben.

5.1. Mutatók és címek

Mivel a mutató valamilyen objektum címét tartalmazza, rajta keresztül az illető objektum indirekt módon érhető el. Tegyük fel, hogy x – pl. **int** - változó, px pedig mutató, amelyet valamilyen eddig még nem ismertetett módon hoztunk létre. Az & egyoperandusú operátor valamely objektum címét adja meg, tehát a

```
px = &x;
```

utasítás x címét rendeli hozzá a px változóhoz; ilyenkor azt mondjuk, hogy px az x értékre mutat (azt címzi meg). Az & operátor csupán változókra és tömbelemekre alkalmazható; az olyan konstrukciók, mint

```
&(x + 1)
```

vagy

```
&3
```

nem megengedettek. Ugyancsak tilos valamely register változó címére hivatkozni! A * egyoperandusú operátor úgy kezeli az operandusát, mint a keresett érték címét, és megkeresi ezt a címet, hogy tartalmát behozza. Ha tehát y is int, akkor

```
y = *px;
```

annak a tartalmát rendeli y-hoz, amire px mutat. Így a

```
px = &x;
y = *px;
```

szekvencia ugyanazt az értéket rendeli y-hoz, mint

```
y = x;
```

A műveletekben részt vevő változókat deklarálnunk is kell :

```
int x, y;
int *px;
```

x és y deklarációja ugyanolyan, mint eddig. A px mutató deklarációja azonban új.

```
int *px;
```

azt fejezi ki, hogy a *px kombináció **int** típusú mennyiség, vagyis ha px a *px környezetben fordul elő, akkor egy **int** típusú változónak felel meg. Gyakorlatilag a változók deklarációjának szintaxisa azoknak a kifejezéseknek a szintaxisát utánozza, amelyekben az illető változók előfordulhatnak. Ez a

meggondolás mindig hasznos, még bonyolult deklarációk esetében is. Pl.

```
double atof (), *dp;
```

azt fejezi ki, hogy kifejezésekben atof() és *dp **double** típusú értékkel rendelkeznek. Vegyük észre továbbá, hogy a deklaráció azt is kimondja, hogy a mutatónak mindig a megadott típusú objektumra kell mutatnia. Mutatók kifejezésekben is előfordulhatnak. Ha pl. px az egész típusú x-re mutat, akkor *px minden olyan szövegkörnyezetben előfordulhat, ahol x előfordulhat.

```
y = *px + 1
```

hatására y 1-gyel nagyobb lesz, mint x;

```
printf ("%d \n", *px)
```

kinyomtatja x pillanatnyi értékét, továbbá

```
d = sqrt((double)*px)
```

d-ben előállítja x négyzetgyökét, ahol x **double** típusúvá alakul át, mielőtt átadódna az sqrt függvénynek (l. a 2. fejezetet). Az olyan kifejezésekben, mint

```
y = *px + 1
```

a * és & egyoperandusú operátorok szorosabban kötnek, mint az aritmetikai operátorok, így a kifejezést kiértékelő program először kiolvassa azt, amire px mutat, majd hozzáad 1 -et, és hozzárendeli y-hoz. Rövidesen visszatérünk arra, hogy mit jelenthet

```
y = *(px + 1)
```

Mutatóhivatkozások értékadások bal oldalán is előfordulhatnak. Ha px és x-re mutat, akkor

```
*px = 0
```

az x-et kinullázza és

```
*px += 1
```

vagy

```
(*px)++
```

inkrementálja. Az utóbbi példában a zárójelek szükségesek: nélkülük a kifejezés px-t inkrementálná, nem pedig azt, amire px mutat, mivel az egyoperandusú operátorok, esetünkben * és ++ jobbról balra értékelődnek ki. Végül, mivel a mutatók változók, ugyanúgy kezelhetők, mint a többi változó. Ha py egy másik, **int** típusú mennyiségre mutat, akkor

```
py = px
```

a px tartalmát py-ba másolja, miáltal py ugyanarra fog mutatni, mint px.

5.2. Mutatók és függvényargumentumok

Mivel a C nyelv az argumentumokat érték szerinti hívás formájában adja át a függvényeknek, a hívott függvény semmilyen közvetlen módon nem tudja megváltoztatni a hívó függvény változóit. Mit tegyünk, ha tényleg meg kell változtatnunk valamelyik közönsége s argumentumot? Példának

okáért a rendezőrutin megcserélhet két, rossz sorrendben levő elemet a swap nevű függvény segítségével. Nem elég, ha azt írjuk, hogy

```
swap (a, b);
```

ahol a swap függvény definíciója:

```
/*ROSSZ*/
 swap (int x, int y)
 {
    int temp;
    temp = x;
    x = y;
    y = temp;
 }
```

Az érték szerint történő hívás miatt a swap nem képes az őt meghívó rutinban előforduló a és b argumentumokat megváltoztatni. Szerencsére van lehetőség a kívánt hatás elérésére. A hívó rutin a megváltoztatandó értékeket megcímző mutatókat ad át:

```
swap (&a, &b);
```

Mivel az & operátor a változó címét állítja elő, &a az a változót megcímző mutató lesz. Magában a swap rutinban az argumentumokat mutatókként deklaráljuk, és az aktuális operandusokat ezeken keresztül érjük el:

```
/*px és py megcserélése*/
 swap (int *px, int *py)
 {
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
 }
```

A mutatóargumentumokat gyakran alkalmazzák az olyan függvényekben, amelyeknek egynél több értéket kell visszaadniuk. (Mondhatjuk pl., hogy swap két értéket ad vissza, tudniillik argumentumainak az új értékeit.) Példaként tekintsük a getint függvényt, amely szabad formátumú bemeneti konverziót végez oly módon, hogy egy karaktersorozatot egész típusú értékekre tördel, hívásonként egy-egy egész értéket szolgáltatva. A getint függvénynek vagy már általa talált értéket kell visszaadnia, vagy pedig - amennyiben nincs több beolvasandó karakter az állomány vége jelet. Az értékeknek külön-külön objektumokként kell visszatérniük, függetlenül attól, hogy milyen értéket használunk EOF-ként, amely maga is egy beolvasott egész mennyiség lehet. Az egyik megoldás szerint, amely a 7. fejezetben ismertetésre kerülő scanf beolvasófüggvényen alapul, getint az EOF-ot mint saját függvényértékét adja vissza olyankor, amikor megtalálta az állomány végét; minden más visszatérő érték közönséges egész szám ra utal. A megtalált egész szám numerikus értéke argumentumon keresztül adódik vissza, amelynek egész számot megcímző mutatónak kell lennie. Ez a fajta szervezés elválasztja az állomány vége állapotot a numerikus értékektől. A következő ciklus a getint hívásai segítségével egész számokkal tölt fel egy tömböt:

```
int n, v, array [SIZE];
for (n = 0; n < SIZE && getint (&v) != EOF; n++)
   array [n] = v;
```

Minden egyes hívás beírja a v változóba a bemeneten talált következő egész számot. Vegyük észre, hogy a getint argumentumaként &y-t kell írnunk v helyett. Ha a puszta v-t használjuk, akkor valószínűleg címzési hibát követünk el, mivel getint azt hiszi, hogy érvényes mutatót kapott.

Maga a getint a már korábban megírt atoi természetes módosítása:

```
/*A következő egész beolvasása a bemenetről*/
getint (int *pn)
{
   int c, sign;
   while ((c = getch ()) == ' ' || c == '\n' || c == '\t')
        ;                   /*Az üres közt átugorja*/

    sign = 1;
   if (c == '+' || c == '-') {        /*Feljegyzi az előjelet*/
        sign = (c == '+') ? 1 : -1;
        c = getch ();
   }
   for (*pn = 0; c >= '0' && c <= '9'; c = getch ())
        *pn = 10 * *pn + c - '0';

   *pn *= sign;
   if (c != EOF)
        ungetch (c);

   return (c);
}
```

A getint függvényben a *pn végig közönséges **int típusú** változóként szerepel. Felhasználtuk a geth és ungeth függvényeket is (leírásukat l. a 4. fejezetben), így azt az egy plusz karaktert, amit még ki kell olvasni, vissza lehet helyezni a bemenetre.

5.1. Gyakorlat. Írjuk meg a getfloat függvényt, amely a getint lebegőpontos megfelelője! Milyen típust ad vissza getfloat függvényértékként?

5.3. Mutatók és tömbök

A C nyelvben szoros kapcsolat van a mutatók és a tömbök között. Ez indokolja, hogy a mutatókkal és a tömbökkel egyidejűleg foglalkozzunk. Valamennyi művelet, amely tömbindexeléssel végrehajtható, mutatók használatával éppúgy elvégezhető. Általában az utóbbi változat gyorsabb, de különösen a kezdők számára első ránézésre nehezebben érthető. Az

```
int a [10]
```

deklaráció definiálja azt a tömböt, amelynek mérete 10, vagyis egy tíz, egymást követő objektumból, az a[0], a[1], . . ., a[9] nevű elemekből álló blokkot határoz meg. Az a[i] jelölésmód a tömbnek a kezdettől

számított i-edik pozícióját fejezi ki. Ha pa egészt megcímző mutató, amelyet

```
int *pa
```

deklarál, akkor a

```
pa =&a[0]
```

értékadás úgy állítja be pa-t, hogy az az a nulladik elemére mutasson, vagyis pa az a[0] elem címét tartalmazza. Ekkor az

```
x = *pa
```

értékadás a[0] tartalmát x-be másolja. Ha pa az a tömb adott elemére mutat, akkor definíció szerint pa + 1 a tömb következő elemére mutat. Általában pa - i i elemmel pa elé, pa + i pedig i elemmel pa mögé mutat. Így ha pa az a[0]-ra mutat, akkor

```
*(pa + 1)
```

a[1] tartalmát szolgáltatja, pa+i az a[i] elem címe és *(pa+i) az a[i] elem tartalma. Ezek a megjegyzések az a tömbben elhelyezkedő változók típusától függetlenül mindig igazak. Az "adj 1-et a mutatóhoz" és ennek kiterjesztéseként az egész mutatóaritmetika alapdefiníciója, hogy a növekmény mértékegysége annak az objektumnak a tárbeli mérete, amire a mutató mutat. Így pa+i esetében a pa-hoz hozzáadás előtt i azoknak az objektumoknak a méretével szorzódik, amire pa mutat. Az indexelés és a mutatóaritmetika között láthatóan nagyon szoros kapcsolat van.

Gyakorlatilag a tömbre való hivatkozást a fordító a tömb kezdetét megcímző mutatóvá alakítja át. Ennek hatására a tömb neve nem más, mint egy mutatókifejezés, amiből számos hasznos dolog következik. Mivel a tömb neve ugyanaz, mint az illető tömb nulladik elemének címe, a

```
pa = &a[0]
```

értékadás úgy is írható, mint

```
pa = a
```

Legalábbis első ránézésre még meglepőbb az a tény, hogy az a[i]-re történő hivatkozás *(a+i)-ként is írható. a[i] kiértékelésekor a C fordító azonnal átalakítja ezt *(a+i)-vé; a két alak teljesen egyenértékű. Ha az ekvivalencia mindkét elemére alkalmazzuk az & operátort, akkor azonnal következik, hogy &a[i] és a+i szintén azonosak: a+i az a-t követő i-edik elem címe. Az érem másik oldala viszont az, hogy ha pa mutató, akkor azt kifejezések indexelhetik: pa[i] azonos *(pa+i)-vel. Röviden, bármilyen töm b vagy indexkifejezés leírható, mint egy mutató plusz egy eltolás és viszont, akár egy utasításon belül is.

Van azonban egy fontos különbség a tömbnév és a mutató között, amire ügyelnünk kell. A mutató változó, így pa=a és pa++ értelmes műveletek. A tömbnév azonban állandó, nem pedig változó: az olyan konstrukciók, mint a=pa vagy a++, vagy p=&a nem megengedettek! Amikor a tömbnév egy függvénynek adódik át, a függvény valójában a tömb kezdetének címét kapja meg. A hívott függvényen belül ez az argumentum változó, ugyanúgy, mint a többi, a tömbnévargumentum tehát csakugyan mutató, vagyis egy címet tartalmazó változó. E tényt kihasználva megírhatjuk az strlen karakterlánchossz-számító függvény új változatát:

```
/*Visszaadja az s karakterlánc hosszát*/
 strlen (char *s)
 {
    int n;
    for (n = 0; *s != '\0'; s++)
         n++;

    return (n);
}
```

s inkrementálása teljesen megengedett, mivel a mutatók változók;

s++-nak nincs hatása az strlen hívó függvénybeli karakterláncra – csupán a címnek az strlen-ben található másolatát inkrementálja. Függvénydefinícióban

```
char s [];
```

és

```
char *s;
```

egyaránt szerepelhet formális paraméterként; azt, hogy melyiket használjuk, nagymértékben az dönti el, hogy miként írjuk le a kifejezéseket a függvényen belül. Amikor a tömbnév adódik át valamelyik függvénynek, a függvény tetszése szerint hiheti azt, hogy tömböt vagy mutatót kapott, és ennek megfelelően kezelheti azt. Akár mindkét típusú műveletet használhatja, ha ez célszerűnek és világosnak látszik.

Lehetőség van arra, hogy a tömbnek csupán egy részét adjuk át valamelyik függvénynek oly módon, hogy a résztömb kezdetét megcímző mutatót adunk át. Ha pl. a egy tömb neve, akkor

```
f(&a[2])
```

és

```
f(a+2)
```

egyaránt az a[2] elem címét adja át az f függvénynek, mivel &a[2] és a+2 egyaránt mutatókifejezés, mindkettő az a tömb harmadik elemére vonatkozik. f-en belül az argumentumdeklaráció akár

```
f(int arr [])
{
  . . .
}
```

akár

```
f (int *arr)
{
  . . .
}
```

is lehet. Ami f-et illeti, az a tény, hogy az argumentum valójában egy nagyobb tömb egy részére vonatkozik, semmiféle következménnyel sem jár.

5.4. Címaritmetika

Ha p mutató, akkor p++ oly módon inkrementálja p-t, hogy az a megcímzett tetszőleges típusú objektum következő elemére, p += i pedig úgy, hogy az a pillanatnyilag megcímzett elem utáni i-edik elemre mutasson. Az ilyen és hasonló szerkezetek a mutató- vagy címaritmetika legegyszerűbb és legközönségesebb formái. A C nyelv következetes és szabályos módon közelít a címaritmetikához; a mutatók, tömbök és a címaritmetika egységes kezelése a nyelv egyik legfőbb erénye.

Szemléltessük ezt azzal, hogy megírunk egy elemi tárfoglaló programot (amely azonban egyszerűsége ellenére is használható)! Két rutinunk van:

az alloc(n) rutin n egymást követő karakterpozíciót megcímző p mutatót ad vissza, amelyet az alloc hívója karakterek tárolására használhat;

továbbá free(p), amely felszabadítja az alloc rutinnal nyert tár területet későbbi használat céljára. A rutinok "elemiek", mivel free hívásai fordított sorrendben kell, hogy történjenek, mint az alloc hívásai. Így az alloc és a free által kezelt tárterület egy verem, vagyis egy "utolsó-be, első-ki" (last-in, first-out) lista. A szabványos C könyvtárban rendelkezésre állnak az ezeknek megfelelő függvények, amelyekre azonban nem vonatkoznak ilyen megszorítások, és a 8. fejezetben is bemutatunk javított változatokat. Addig azonban számos alkalmazáshoz megfelel a triviális alloc is, ha arra van szükségünk, hogy előre nem látható méretű kisebb tárterületek előre nem látható időpontokban rendelkezésre álljanak. A legegyszerűbb megvalósításban az alloc egy allocbuf-nak nevezett nagy karaktertömb darabjait szolgáltatja. Ez a tömb az alloc és a free kizárólagos tulajdona.

Mivel ez a két rutin mutatókat és nem tömbindexeket használ, a tömb nevét egyetlen más rutinnak sem kell ismernie, így az static extern-ként deklarálható, vagyis csak az alloc és a free függvényeket tartalmazó állományban lesz érvényes és azon kívül láthatatlan. A gyakorlatban akár nem is kell, hogy névvel rendelkezzen a tömb: ehelyett úgy is előállítható, hogy a program az operációs rendszertől elkér valamilyen név nélküli tárblokkot megcímző mutatót. Tudnunk kell azt is, hogy mennyi került felhasználásra allocbuf-ból. E célból egy, a következő szabad elemet megcímző mutatót használunk, amelynek neve allocp. Ha valaki az alloc-tól n karaktert kér, akkor az ellenőrzi, hogy maradt-e még ennyi hely alloc buf-ban. Ha igen, akkor alloc visszaadja az allocp pillanatnyi értékét (vagyis a szabad blokk kezdőcímét), majd azt n-nel inkrementálja, hogy a következő szabad területre mutasson. free(p) egyszerűen p-re állítja be allo_p-t, ha p az allocbuf-on belül van.

```
#define NULL 0         /*Mutató a hibajelzéshez*/
#define ALLOCSIZE 1000 /*A rendelkezésre álló terület  mérete*/
static char allocbuf [ALLOCSIZE];    /*Tárhely alloc-nak*/
static char *allocp = allocbuf;      /*Köv. szabad hely*/
char *alloc (int n)   /*n karaktert megcímző mutatót ad vissza*/
{
   if (allocp + n <= allocbuf + ALLOCSIZE) {   /*Befér*/
        allocp += n;
        return (allocp - n);                    /*Régi p*/
   } else                                  /*Nincs elég hely*/
        return (NULL);
}
```

```
/*p által megcímzett terület felszabadítása*/
free (char *p)
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
         allocp = p;
}
```

Néhány megjegyzés: Általában a mutató éppen úgy inicializálható, mint bármilyen más változó, noha közönséges esetben értelmes érték csupán a NULL (l. a továbbiakban) vagy olyan kifejezés lehet, amely a korábban definiált megfelelő típusú adatok címeit tartalmazza. A

```
static char *allocp = allocbuf;
```

deklaráció úgy definiálja allocp-t, hogy az karaktermutató legyen, és úgy inicializálja, hogy allocbuf-ra mutasson, amely a következő szabad pozíció a program indításakor. Ezt úgy is írhattuk volna, hogy:

```
static char *allocp = &allocbuf [0];
```

mivel a tömb neve egyben a nulladik elemének a címe; mindig a természetesebb változatot használjuk! Az

```
if (allocp + n <= allocbuf + ALLOCSIZE)
```

vizsgálat ellenőrzi, hogy van-e elegendő hely az n számú karakter elhelyezésére vonatkozó kérés teljesítésére. Ha igen, akkor az allocp új értéke legfeljebb eggyel mutat túl az allocbuf végén. Ha a kérés kielégíthető, az alloc közönséges mutatóval tér v issza (figyeljük meg magának a függvénynek a deklarációját). Ha a kérés nem teljesíthető, akkor az alloc-nak valamilyen jel visszaadásával kell jeleznie, hogy nem maradt hely. A C nyelv gondoskodik arról, hogy semmiféle olyan mutató, amely érvényes módon adatra mutat, nem tartalmazhat nullát, így a nulla visszatérési érték használható az abnormális esemény jelzésére, adott esetben annak közlésére, hogy nincs hely. Számszerű nulla helyett azonban NULL-t írunk, hogy ezzel világosabban jelezzük : ez a mutató különleges értéke. Általában egész számok nem rendelhetők értelmes módon mutatókhoz, a nulla speciális eset. Az olyan vizsgálatok, mint

```
if (alloc + n <= allocbuf + ALLOCSIZE)
```

és

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

a mutatóaritmetika lényeges sajátosságaira világítanak rá. Először is, a mutatók bizonyos körülmények között összehasonlíthatók. Ha p és q ugyanannak a tömbnek az elemeire mutat, akkor az olyan relációk, mint <, >= stb. megfelelően működnek.

```
P < q
```

pl. akkor igaz, ha p a tömb kisebb sorszámú elemére mutat, mint q. Az == és != relációk ugyancsak alkalmazhatók. Bármilyen mutató nullával való egyenlősége vagy nemegyenlősége értelmes módon ellenőrizhető. Vigyázat! Ne végezzünk viszont különböző tömbök et megcímző mutatókkal aritmetikai műveleteket vagy összehasonlításokat! Ha szerencsénk van, akkor minden gépen nyilvánvaló értelmetlenséget kapunk. Ha azonban nincs szerencsénk, akkor a programunk működni fog az egyik gépen, de rejtélyes módon össze fog omlani egy másikon. Másodszor láttuk, hogy az egész számot megcímző mutatókkal összeadás és kivonás végezhető.

```
p + n
```

a p által éppen megcímzett objektumot követő n-edik objektumot jelenti. Ez igaz, függetlenül attól, hogy a p-t milyen típusú objektumot megcímző mutatónak deklaráltuk: a fordító n-et olyan egységekben számlálja, amelyek megfelelnek a p által megcímzett objektum méretének, amely utóbbit p deklarációja határozza meg. A PDP-11-en például a méretfaktor **char** esetében 1 **short**-nál 2, **long** és **float** esetén 4 és **double** esetén 8. Mutatók kivonása szintén megengedett: ha p és q ugyanannak a tömbnek az elemeire mutatnak, akkor p - q a p és q közötti elemek darabszáma. E tényt kihasználva megírhatjuk az strlen újabb változatát:

```
/*Visszaadja az s karakterlánc hosszát*/
strlen (char *s)
{
    char *p = s;
    while (*p != '\0')
         p++;

    return (p – s);
}
```

A deklarációban p kezdeti értékeként s-et adtuk meg, vagyis p kezdetben az s első karakterére mutat. A **while** ciklusban addig vizsgáljuk az egymást követő karaktereket, amíg a véget jelző \0 elő nem kerül. Mivel \0 értéke nulla, és mivel a **while** csupán az t vizsgálja, hogy a kifejezés nulla-e, elhagyható az explicit vizsgálat. Az ilyen ciklusokat gyakran az alábbi alakban írják:

```
while (*p)
    p++;
```

Minthogy p karakterekre mutat, p++ minden alkalommal a következő karakterre lépteti p-t, és p - s az átlépett karakterek számát, vagyis a karakterlánc hosszát adja meg. A mutatóaritmetika következetes: ha **float** mennyiségekkel dolgoznánk, amelyek a char-oknál több tárterületet foglalnak el, és ha p **float**-ot megcímző mutató lenne, akkor p++ a következő **float**-ra léptetne. Így az alloc másik változatát, amely pl. char-ok helyett **float** változókkal dolgozik, egyszerűen úgy írhatjuk meg, hogy az alloc-ban és free-ben végig **float**-okra cseréljük a **char** változókat. Az összes mutatóművelet automatikusan számításba veszi a megcímzett objektum méretét, így semmi egyebet nem kell megváltoztatni.

Az említett műveleteken kívül (mutató és integer összeadása és kivonása, két mutató kivonása és összehasonlítása) minden más mutatóművelet tilos! Nincs megengedve két mutató összeadása,

szorzása, osztása, mutatók léptetése, maszkolása, sem pedig **float** vagy double mennyiségeknek mutatókhoz történő hozzáadása.

5.5. Karaktermutatók és függvények

Az

```
"Ez itt egy karakterlánc"
```

alakú karakterlánc-állandó nem más, mint egy karaktertömb. A fordító a belső ábrázolásban a tömböt a \0 karakterrel zárja le, hogy a programok megtalálhassák a karakterlánc végét. A tárterület hossza tehát eggyel nagyobb, mint az idézőjelek közötti karakterek száma. A karakterlánc-állandó leggyakrabban talán függvényargumentumokban fordul elő, mint pl.

```
printf ("Figyelem, emberek\n");
```

A programban ily módon megjelenő karakterlánc karaktermutatón keresztül érhető el; printf a karaktertömböt megcímző mutatót kap. A karaktertömböknek természetesen nem kell feltétlenül függvényargumentumoknak lenniük. Ha message-et

```
char *message;
```

deklarálja, akkor a

```
message = "now is the time"; /*Ideje*/
```

utasítás message-hez a tényleges karaktereket megcímző mutatót rendel hozzá. Ez nem karakterlánc-másolás, a dolog csak a mutatókat érinti. A C nyelvben nincsen olyan operátor, amellyel teljes karakterláncot egy egységként dolgozhatnánk fel. A mutatók és tömbök további vonatkozásait a 7. fejezetben ismertetendő szabványos be- és kiviteli (I/O) könyvtár két hasznos függvényén keresztül mutatjuk be. Az első függvény az strcpy(s, t), amely a t karakterláncot az s karakterláncba másolja. Az argumentumokat az értékadáshoz való hasonlóság miatt írtuk ebben a sorrendben, hiszen a t karakterláncnak az s karakterlánchoz történő hozzárendelésekor azt mondanánk, hogy

```
s = t
```

Először a tömbös változatot mutatjuk be:

```
/*t másolása s-be*/
strcpy (char s [], char t [])
{
    int i;
    i = 0;
    while ((s [i] =t [i]) != '\0')
         i++;
}
```

Összehasonlításul íme az strcpy mutatóval írt változata:

```
/*t másolása s-be;    1. mutatót alkalmazó változat*/
strcpy (char *s, char *t)
{
    while ((*s = *t) != '\0') {
         s++;
         t++;
    }
}
```

Mivel az argumentumok átadása érték szerint történik, strcpy tetszés szerinti módon használhatja s-t és t-t. Az adott esetben ezek alkalmas módon inicializált mutatók, amelyek karakterenként végighaladnak a tömbökön, amíg a t-t lezáró \0 át nem másolódik s-be. A gyakorlatban strcpy-t nem az előbb bemutatott módon írnánk meg. Egy másik lehetőség pl. :

```
/*t másolása s-be; 2. mutatót alkalmazó változat*/
strcpy (char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
         ;
}
```

Ez a programkód s és t inkrementálását a feltételvizsgálatba helyezi át. *t++ értéke az a karakter, amire t inkrementálás előtt mutatott; a ++ postfixum mindaddig nem változtatja meg t-t, amíg ez a karakter

feldolgozásra nem került. Hasonlóképpen, s inkrementálása előtt a karakter a régi s pozícióban tárolódik. Egyben ez a karakter lesz az az érték, amelyet a ciklusvezérlés érdekében \0-val összehasonlítunk. Végeredményben a karakterek a záró \0-ig, a záró \0-t is beleértve átmásolódnak t-ből s-be. Végső lerövidítésként vegyük ismét észre, hogy a \0-val való összehasonlítás redundáns, ezért a függvény gyakran így jelenik meg:

```
/*t másolása s-be;  3. mutatót alkalmazó változat*/
strcpy (char *s, char *t)
{
    while (*s++ = *t++)
         ;
}
```

Bár első ránézésre titokzatosnak tűnhet, ez a jelölés nagyon kényelmes, és már csak azért is el kell sajátítanunk, mert gyakran találkozunk vele C programokban. A másik rutin az strcmp(s, t), amely összehasonlítja az s és t karakterláncokat, és negatív számot, nullát vagy pozitív számot ad vissza aszerint, hogy s lexikografikusan kisebb, mint t, egyenlő t-vel vagy nagyobb, mint t. A visszaadott értéket úgy nyerj ük, hogy az első olyan pozíción, ahol s és t nem egyeznek meg, kivonjuk egymásból a karaktereket.

```
/*A visszatérő érték < 0, ha s < t; 0, ha s == t, > 0, ha s > t*/
strcmp ( char s [], char t [])
{
    int i;
    i = 0;
    while (s [i] == t [i])
         if (s [i++] == '\0')
              return (0);

    return (s [i] - t [i]);
}
```

Az strcmp mutató alkalmazásával:

```
/*A visszatérő érték < 0, ha s < t;  0, ha s == t; > 0, ha s > t*/
strcmp (char *s, char *t)
{
     for (; *s == *t; s++,t++)
         if (*s == '\0')
              return (0);

    return (*s - *t);
}
```

Mivel ++ és -- akár prefix, akár postfix operátorok lehetnek, ritkábban ugyan, de a * és ++, ill. -- más kombinációi is előfordulhatnak. Pl.

```
*++p
```

p-t még azelőtt inkrementálja, hogy a p által megcímzett karakterhez való hozzáférés megtörténne.

```
*--p
```

először dekrementálja p-t.

5.2. Gyakorlat. Írjuk át a 2. fejezetben bemutatott strcat függvényt mutató alkalmazásával (strcat(s, t) a t karakterláncot az s karakterlánc végére másolja)!

5.3. Gyakorlat. Írjunk makrót strcpy-ra!

5.4. Gyakorlat. Írjuk át a korábbi fejezetek erre alkalmas programjait és gyakorlatait úgy, hogy tömbindexelés helyett mutatókat használunk! Jó lehetőség pl. a getline (l. az 1. és 4. fejezetet), az atoi az itoa és változataik (l. a 2., 3., 4. fejezete t), valamint az index és a getop (4. fejezet).

5.6. A mutatók nem egész számok

Régebbi C programok igen liberálisan kezelték a mutatók másolásának kérdését. Általában a legtöbb gépen a mutatót hozzá lehetett rendelni egy egész típusú_ mennyiséghez és viszont anélkül, hogy maga a mutató megváltozott volna sem méretszámítás, sem konverzió nem történt, nem vesztek el bitek. Sajnos azonban ez oda vezetett, hogy sokan szabadosan kezelték a mutatókat visszaadó rutinokat, és a kapott mutatókat egyszerűen más rutinoknak adták át gyakran elmulasztva a szükséges mutatódeklarációkat. Tekintsük pl. az strsave(s) függvényt, amely az alloc hívásával kapott biztos

helyre másolja az s karakterláncot, majd az azt megcímző mutatóval tér vissza. E program helyesen így fest :

```
/*Elmenti az s karakterláncot*/
char *strsave (char *s)
{
    char *p, *alloc ();
    if ((p = alloc (strlen (s) + 1)) != NULL)
         strcpy (p, s);

    return (p);
}
```

5.7. Többdimenziós tömbök

A C nyelv mátrixjellegű többdimenziós tömbök használatát is megengedi, noha a gyakorlatban ilyeneket sokkal ritkábban használunk, mint mutatótömböket. Ebben a szakaszban ezek néhány tulajdonságát mutatjuk be. Tekintsük a dátumkonverzió problémáját: a hónap adott napjának az év egy napjává és vissza történő alakítását. Példának okáért a március 1. nem szökőévekben a 60., szökőévekben a 61. nap. Az átalakítások elvégzésére definiáljunk két függvényt: day_of_yea r a hónapot és napot az év napjává, míg month_day az év adott napját hónappá és nappá alakítja át. Mivel az utóbbi függvény két értékkel tér vissza, a hónap és a nap argumentum mutató lesz:

```
month_day (1977, 60, &m, &d)
```

hatására m 3 és d 1 lesz (március 1 .). Mindkét függvénynek ugyanarra az információra van szüksége, tudniillik az egyes hónapok napjainak számát tartalmazó táblázatra. Mivel a hónapokban levő napok száma eltér a szökőévekben és a nem szökőévekben, egyszerűbb, ha ezeket egy kétdimenziós tömb két sorában elkülönítjük, mint ha a számítás során próbálnánk nyomon követni, hogy mi is történik februárban. A tömb és az átalakításokat végző függvények a következők :

```
static int day_tab [2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/*Az éven belüli napsorszám kiszámítása a hónapból és napból*/
day_of_year (int year, int month, int day)
{
    int i, leap;
    leap = year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
    for (i = 1; i < month; i++)
         day += day_tab [leap][i];

    return (day);
}

/*Hónap és nap kiszámítása az
   év adott sorszámú napjából*/
month_day (int year, int yearday, int *pmonth, int *pday)
{
    int i, leap;
    leap = year % 4 == 0 && year % 100 != 0   || year % 400 == 0;
    for (i = 1; yearday > day_tab [leap][i]; i++)
         yearday -= day_tab [leap][i];

    *pmonth = i;
    *pday = yearday;
}
```

A day_tab tömbnek kívül kell lennie mind a day_of_year, mind pedig a month_day függvényen, hogy mindketten használhassák. A day_tab az első kétdimenziós tömb, amellyel eddig találkoztunk. A C-ben definíció szerint a kétdimenziós tömb valójában olyan egydimenziós tömb, amelynek minden eleme tömb. Ezért írjuk az indexeket

```
day_tab [i][j]
```

nem pedig

```
day_tab [i,j]
```

alakban, mint a legtöbb más nyelvben. Ettől eltekintve a kétdimenziós tömb ugyanúgy kezelhető, mint a többi nyelvben. Az elemek soronként tárolódnak, vagyis a legjobboldalibb index változik a leggyorsabban, amikor az elemekhez a tárolás sorrendjében történik hozzáférés. A tömböt a kezdeti értékek kapcsos zárójelek közé zárt listájával inicializáljuk; a kétdimenziós tömb minden sorát a megfelelő allista inicializálja. A day_tab tömböt egy nullákat tartalmazó oszloppal kezdtük, hogy a hónapszámok ne 0-tól 11-ig, hanem a megszokott módon 1-től 12-ig fussanak. Mivel az adott esetben az elfoglalt tárhely mennyisége nem lényeges, ez a megoldás egyszerűbb, mint az indexek kiigazítása. Ha a kétdimenziós tömböt függvénynek kell átadni, a függvénybeli argumentumdeklarációnak tartalmaznia kell az oszlopméretet; a sorméret közömbös, mivel mint korábban, most is mutatót adunk át. Az adott esetben ez a 13 int-et tartalmazó tömbökre mint objektumokra mutat. Így, ha a day_tab tömböt kell átadni az f függvénynek, akkor f deklarációja:

```
f (int day_tab [2][13])
{
  . . .
}
```

Az f-beli argumentumdeklaráció lehetne

```
int day_tab [][13];
```

is, mivel a sorok száma közömbös, vagy lehetne

```
int (*day_tab)[13];
```

amely azt fejezi ki, hogy az argumentum egy 13 egészből álló tömböt jelölő mutató. A zárójelek szükségesek, mivel [ ] (szögletes zárójelek) precedenciája nagyobb, mint a * szimbólumé, így zárójelek nélkül az

```
int *day_tab [13];
```

deklaráció egy 13 darab, egészt megcímző mutatóból álló tömböt jelent, amint ezt a következőkben látni fogjuk.

5.8. Mutatótömbök; mutatókat megcímző mutatók

Mivel a mutatók maguk is változók, joggal várható, hogy mutatókból álló tömbök is használhatók. Ez valóban így van. E lehetőséget olyan program megírásán keresztül mutatjuk be, amely egy szövegsorokból álló halmazt alfabetikus sorrendbe rendez: ez a UNIX -beli sort rendező segédprogram egyszerűsített változata. A 3. fejezetben bemutattuk a Shell sort függvényét, amely egészekből álló tömböt rendez. Ugyanez az algoritmus fog itt is működni, attól eltekintve, hogy most szövegsorokkal kell foglalkoznunk, amelyek különböző hosszúságúak, és az egészektől eltérően ne m hasonlíthatók össze, és egyetlen művelettel nem mozgathatók. Olyan adatábrázolásra van szükségünk, amely hatékonyan és kényelmesen birkózik meg változó hosszúságú szövegsorokkal. Itt lépnek be a mutatókból álló tömbök. Ha a rendezendő sorokat elejétől végig egyetlen hosszú karaktertömbben tároljuk (amelyet esetleg az alloc kezel), akkor minden sor elérhető az első karakterét megcímző mutatón keresztül. Maguk a mutatók egy tömbben tárolhatók. Két sor oly módon hasonlítható össze, hogy mutatóikat átadjuk strcmp-nek. Ha két, nem megfelelő sorrendben levő sort meg kell cserélni, akkor a mutatótömbben levő mutatók cserélődnek fel, nem pedig maguk a szövegsorok. Ezáltal elkerüljük azt a két összetartozó problémát, amit a bonyolult tárkezelés és a tényleges szövegsorok mozgatásával együtt járó nagy megterhelés jelentene. A rendezés folyamata három lépésből tevődik össze:

- az összes bemeneti sor beolvasása,
- a beolvasott sorok rendezése,
- a helyes sorrendben történő kinyomtatás.

Szokás szerint legcélszerűbb, ha a programot olyan függvényekre bontjuk fel, amelyek ezt a természetes felosztást követik, és a főrutin végzi a folyamat vezérlését. Egy pillanatra hagyjuk a rendezési lépést és összpontosítsunk az adatstruktúrára, valamint a be- és kivitelre. A bemeneti rutinnak össze kell gyűjtenie, majd tárolnia kell az egyes sorok karaktereit és össze kell állítania a sorokat megcímző mutatók tömbjét. Ugyancsak a bemeneti rutinnak kell megszámlálnia a beérkező sorokat, mivel erre az információra a rendezéskor és nyomtatáskor szükség lesz. Tekintve, hogy a beolvasófüggvény csak véges számú bemeneti sorral tud megbirkózni, valamiféle nemlétező sor -darabszámot, pl. -1-et ad vissza, ha túl sok szöveg érkezett. A kimeneti rutinnak abban a sorrendben kell kinyomtatnia a sorokat, amelyben azok a mutatók tömbjében megjelennek.

```
 #define NULL 0
 #define LINES 100          /*A rendezendő sorok max. száma*/
 #define MAXLEN 1000

/*Rendezéshez Sorok beolvasása*/
readlines (char *lineptr [], int maxlines)
{
    int len, nlines;
    char *p, *alloc (), line [MAXLEN];
    nlines = 0;
    while ((len = getline (line, MAXLEN)) > 0)
         if (nlines >= maxlines)
              return (-1);
         else if ((p = alloc (len)) == NULL)
                   return(-1);
                     else {
                   line [len - 1] = '\0'; /*Az újsort levágja*/
                   strcpy (p, line);
                   lineptr [nlines++] = p;
                     }

    return (nlines);
}

/*Beolvasott sorok rendezése*/
main ()
{
    char * lineptr [LINES];  /*A szövegsorokatmegcímző mutatók*/
    int nlines;              /*A beolvasott sorok száma*/
    if ((nlines = readlines (lineptr, LINES)) >= 0) {
         sort (lineptr, nlines);
         writelines (lineptr, nlines);
    }
    else
         printf ("a bemenet túl nagy a rendezéshez \n");
 }
```

A sorok végén található újsor karakterek törlődnek, így nem befolyásolják a rendezési sorrendet.

```
/*A kimenetre kerülő sorok kiírása*/
writelines (char *lineptr [], int nlines)
{
    int i;
    for (i = 0; i < nlines; i++)
         printf ("%s \n", lineptr [i]);
}
```

A legfontosabb új dolog lineptr deklarációja:

```
char *lineptr [LINES];
```

azt jelenti, hogy a lineptr egy LINES számú elemből álló, mutatókat tartalmazó tömb, amelynek minden eleme egy-egy char-ra mutat. Más szóval lineptr [i] karaktermutató, és *lineptr [i] egy karakterhez fér hozzá.

Mivel lineptr tömb, amelyet writelines-nak adunk át, pontosan ugyanúgy kezelhetjük mutatóként, mint azt a korábbi példákban láttuk. A függvény tehát így is írható:

```
/*A kimenetre kerülő sorok kíírása*/
writelines (char *lineptr [], int nlines)
{
    while (--nlines >= 0)
          printf ("%s \n", *lineptr++);
}
```

A *lineptr kezdetben az első sorra mutat; minden inkrementálás a következő sorra lépteti, miközben nlines-t leszámláljuk. Most, hogy a be- és kimenet a kezünkben van, rátérhetünk a rendezésre. A 3. fejezetben látott Shell rendezőprogramot kismértékben meg kell változtatnunk: módosítani kell a deklarációkat, és az összehasonlítás műveletét külön függvényben kell elhelyezni. Az alapvető algoritmus változatlan, ezért bízhatunk abban, hogy a program továbbra is működni fog.

```
/*A v [0] ... v [n - 1] karakterláncok  rendezése növekvő
sorrendben*/
sort ( char *v [], int n)
{
    int gap, i, j;
    char *temp;
    for (gap = n/2; gap > 0; gap /= 2)
         for (i = gap; i < n; i++)
              for (j = i - gap; j >= 0; j -= gap) {
                   if (strcmp (v [j], v [j + gap]) <=0)
                             break;

                   temp = v [j];
                   v [j] = v [j + gap];
                   v [j + gap] = temp;
              }
}
```

Mivel v (azaz lineptr) minden egyes eleme karaktermutató, temp-nek is annak kell lennie, hogy az egyik a másikba másolható legyen. A programot a lehető legegyszerűbbre írtuk meg, hogy minél gyorsabban el tudjuk indítani. Lehetne azonban gyorsabb is, ha pl. a bejövő sorokat közvetlenül a readlines által karbantartott tömbbe másolnánk, nem pedig először a line-ba, majd az alloc által kezelt, rejtett helyre. Az első változatot azonban bölcsebb úgy megírni, hogy minél érthetőbb legyen. A hatékonysággal ráérünk később foglalkozni. Programunkon valószínűleg nem sokat gyorsítana, ha kiküszöbölnénk a bemeneti sorok szükségtelen átmásolását . Lényeges javulást csak az hozhat, ha a Shell sort programját valami jobbal, pl. a quicksort programmal cseréljük fel. Az 1. fejezetben rámutattunk, hogy mivel a **while** és a **for** ciklusok a végfeltételt a ciklustörzs első végrehajtása előtt vizsgálják, hozzájárulnak ahhoz, hogy a programok a lehető leggyorsabban működjenek, különösen, ha nincs bemenet. Tanulságos, ha végigmegyünk a rendezőprogram függvényein, és megvizsgáljuk, mi történik, ha egyáltalán nincs bemeneti szöveg.

5.5. Gyakorlat. Írjuk újra a readlines függvényt oly módon, hogy a sorokat a main által adott tömbben hozzuk létre és nem az alloc-ot hívjuk a tár karbantartása céljából! Mennyivel gyorsabb így a program?

5.9. Mutatótömbök inicializálása

Írjuk meg a month_name(n) függvényt, amely egy olyan mutatót ad vissza, amely az n-edik hónap nevét tartalmazó karakterláncra mutat. Ez a belső **static** tömb ideális alkalmazása. A month_name a karakterláncok számára külön tömböt tartalmaz, és hívás után a megfelelő karakterláncot megcímző mutatót adja vissza. E szakasz témája az, hogy hogyan kell a nevek tömbjét inicializálni. A szintaxis hasonlít a korábbi inicializálásokra:

```
/*Visszaadja az n-edik hónap nevét*/
char *month_name (int n)
{
    static char *name [] = {
              "nem létező hónap",
              "január",
              "február",
              "március",
              "április",
              "május",
              "június",
              "július",
              "augusztus",
              "szeptember",
              "október",
              "november",
              "december"
    }
    return ((n < 1 || n > 12) ? name [0] : name [n]);
}
```

A name karaktermutatókból álló tömb. Deklarációja ugyanaz, mint lineptr-é a rendezési példában. A kezdeti érték egyszerűen a karakterláncok listája; mindegyik hozzá van rendelve a tömb megfelelő pozíciójához. Pontosabban szólva az i-edik karakterlánc kar akterei valamilyen más helyre kerülnek, és az őket megcímző mutató name [i]-ben található. Mivel a tömb mérete nincs megadva, maga a fordító számlálja meg a kezdeti értékeket és tölti be a helyes darabszámot.

5.10. Mutatók és többdimenziós tömbök

A kezdő C programozókat gyakran megzavarja a kétdimenziós tömbök és az olyan mutatótömbök közötti különbség, mint amilyen a fenti példában name volt. Pl. az

```
int a [10][10];
int *b [10];
```

deklarációkban a és b használata ugyan hasonló, amennyiben a[5][5] és b[5] [5] egyaránt egy bizonyos int-re történő megengedett hivatkozások. a azonban igazi tömb : 100 tárrekeszt foglaltunk le számára, egy adott elemét a hagyományos mátrixszerű index-számítással választhatjuk ki. b esetében azonban a deklaráció csupán 10 mutatót foglal le: ezek mindegyikét úgy kell beállítani, hogy egy-egy egészekből álló tömbre mutasson. Feltéve, hogy mindegyikük egy-egy tízelemű tömbre mutat, 100 tárrekeszt foglaltunk le, ezenkívül további tíz rekeszre van szükség a mutatók számára. Így a mutatókból álló

tömb valamivel több területet foglal el, és explicit inicializálást igényelhet. Van azonban két előnye: egyrészt az elemeket nem szorzással és összeadással, hanem mutatón keresztül, indirekt módon érjük el, másrészt a tömb sorai különböző hosszúságúak lehetnek. Így nem szükséges, hogy b minden eleme

egy-egy tízelemű vektorra mutasson: lehet köztük amelyik két elemre, esetleg húszra, sőt amelyik egyetlen elemre sem mutat. Bár az előbbi fejtegetésben egészekről beszéltünk, a mutatótömbök használatának messze leggyakoribb esete az, amelyet a month_name-ben mutattunk be: különböző hosszúságú karakterláncok kezelése.

5.6. Gyakorlat. Írjuk át a day_of_year és a month_day rutint oly módon, hogy indexelés helyett mutatókat használjunk!

5.11. Parancssor-argumentumok

A C nyelvet támogató környezetekben lehetőség van arra, hogy a végrehajtás megkezdésekor a programnak parancssor-argumentumokat vagy paramétereket adjunk át. Amikor a végrehajtás megkezdésekor main-t meghívjuk, a hívásban két argumentum szerepel. Az első (amit szokás szerint argc-nek nevezünk) azoknak a parancssor-argumentumoknak a darabszáma, amelyekkel a programot meghívtuk. A második argumentum (argv) egy mutató: ez arra a karakterlánc-tömbre mutat, amely az előbbi argumentumokat tartalmazza. Egy karakterlánc egy argumentumnak felel meg. E karakterláncok kezelése a többszörös mélységű mutatóhasználat tipikus esete. A többszintű mutatóhasználathoz szükséges deklarációknak, a módszer alkalmazásának legegyszerűbb szemléltető példája az echo program, amely egyszerűen megismétli (visszhangozza) az egy sorban megjelenő, szóközökkel elválasztott parancssor-argumentumokat. Vagyis, ha kiadjuk az

```
echo Figyelem, emberek
```

parancsot, akkor a kimeneten

```
Figyelem, emberek
```

jelenik meg. Megállapodás szerint argv [0] az a név, amellyel a programot hívták, így argc legalább 1 . Az előbbi példában argc 3 és argv[0], argv [1], ill. argv [2] sorra echo, Figyelem és emberek. Az első igazi argumentum argv [1] és az utolsó argv[n-1]. Ha argc értéke 1, akkor a program nevét nem követik parancssor-argumentumok. Mindezt az echo programban mutatjuk be:

```
/*Visszhangozza az argumentumokat 1. változat*/
main (int argc, char *argv [])
{
    int i;
    for (i = 1; i < argc; i++)
         printf ("%s %c", argv [i], (i < argc-1) ? ' ' : '\n');
}
```

Mivel argv mutatótömböt megcímző mutató, többféleképpen is megírhatjuk ezt a programot úgy, hogy a tömb indexelése helyett a mutatót kezeljük.

Lássunk két változatot :

```
/*Visszhangozza az argumentumokat  2. változat*/
main (int argc, char *argv [])
{
    while (--argc > 0)
         printf ("%s %c", *++argv,(argc > 1) ? ' ' : '\n');
}
```

Mivel argv az argumentum-karakterláncok tömbjének kezdetét megcímző mutató, 1-gyel történő inkrementálásának (++argv) hatására argv[0] helyett az eredeti argv[1]-re fog mutatni. Az egymást követő inkrementálások hatására mindig a következő argumentumra l ép; *argv az illető argumentumot megcímző mutató. Ezzel egyidejűleg argc dekrementálódik: amikor nullává válik, nincs több kinyomtatandó argumentum. Egy másik változat :

```
/*Visszhangozza az argumentumokat 3. változat*/
main (int argc, char *argv [])
{
    while (--argc > 0)
         printf ((argc > 1) ? "%s" : "%s \n", *++argv);
}
```

Ez a változat azt mutatja be, hogy a printf formátum-argumentuma éppúgy lehet kifejezés, mint bármilyen más függvényé. Ez a fajta használat nem túl gyakori, de érdemes rá emlékezni. Második példaként végezzünk néhány javítást a 4. fejezet mintakereső programjában. Talán emlékezünk arra, hogy a keresési minta mélyen a programon belül helyezkedett el, ami nem valami kellemes megoldás. Az UNIX grep segédprogramjának fonalát követve változtassuk meg a programot oly módon, hogy az

összehasonlítandó mintát a parancssor első argumentuma adja meg.

```
#define MAXLINE 1000

/*Megkeresi az 1. argumentum  szerinti mintát*/
main (int argc, char *argv [])
{
    char line [MAXLINE];
    if (argc != 2)
         printf ("Mintakeresés \n");
    else
         while (getline (line, MAXLINE) > 0)
              if (index (line, argv [1]) >= 0)
                        printf ("%s", line);
}
```

Most kidolgozhatjuk az alapmodellt, amivel a további mutatóalkalmazásokat szemléltethetjük. Tegyük fel, hogy két opcionális (szabadon választható) argumentumot engedünk meg. Az egyik azt mondja, hogy "nyomtass ki minden sort, kivéve azokat, amelyek illeszkednek a mintára", míg a második azt mondja, "írd minden kinyomtatott sor elé annak sorszámát". A C programokban a mínusz jellel kezdődő argumentumok megállapodásszerűen opcionális jelzőt (flaget) vagy paramétert jelentenek. Ha tehát az inverzió (a kivéve, a fordítottság) jelölésére -x-et választunk, és a sorszámozást -n-nel kérjük, akkor a

```
find –x –n the
```

parancs a

Now is the time

for all good men

to come to the aid

of their party.

bemeneti szöveg esetén a

```
2: for all good men
```

kimenetet fogja eredményezni. Az opcionális argumentumok sorrendje tetszőleges kell, hogy legyen, és a program további részének nem szabad függenie a ténylegesen megadott argumentumok számától. Az adott esetben az index hívásának nem szabad argv[2]-re hivatkoznia olyankor, amikor csupán egyetlen opcionális argumentum volt, vagy argv[1]-re, ha egyáltalán nem volt ilyen argumentum. A felhasználók számára kényelmes továbbá, ha az opcionális argumentumok konkatenálhatók, mint pl.:

```
find –nx the
```

Íme a program:

```
#define MAXLINE 1000

/*Megkeresi az 1. argumentum szerinti mintát*/
main (int argc, char *argv [])
{
    char line [MAXLINE], *s;
    long lineno = 0;
    int except = 0, number = 0;
    while (--argc > 0 && (*++argv) [0] == '-')
         for (s = argv [0] + 1; *s != '\0'; s++)
              switch (*s) {
              case 'x':
                        except = 1;
                        break;
              case 'n':
                        number = 1;
                        break;
              default:
                        printf ("keresés: illegális opció %c \n",
                                 *s);
                        argc = 0;
                        break;
              }

         if (argc != 1)
               printf ("Mintakeresés: -x -n \n");
         else
              while (getline (line, MAXLINE) > 0) {
                   lineno++;
                   if ((index (line, *argv) >= 0) != except) {
                             if (number)
                                  printf ("%1d: ", lineno);
                             printf ("%s", line);
                   }
              }
}
```

argv minden egyes opcionális argumentum előtt inkrementálódik, argc pedig dekrementálódik. Ha nincsenek hibák, akkor a ciklus végén argc értéke 1 és *argv a mintára mutat. Vegyük észre, hogy *++argv argumentum-karakterláncot megcímző mutató: (*++argv) [0] az első karaktere. A zárójelek szükségesek, mivel nélkülük a kifejezés *++(argv[0]) lenne, aminek az értelme teljesen más (és rossz).

Megengedett alak lenne még *++argv is.

5.7. Gyakorlat. Írjuk meg az add nevű programot, amely kiértékel egy, a parancssorban szereplő

fordított lengyel alakú kifejezést! Például

add 2 3 4 + *

2 * (3+4)-et számítja ki.

5.8. Gyakorlat. Módosítsuk az entab és detab programokat (amelyeket az 1. fejezetben gyakorlatként írtunk meg) oly módon, hogy az a tabulátor stop-ok listáját argumentumokként fogadja! Argumentum megadásának hiányában a normál tabulátorbeállításokat használjuk!

5.9. Gyakorlat. Bővítsük az entab és detab programokat úgy, hogy elfogadják az entab m +n rövidített jelölést, amely az m-edik oszloptól kezdve minden n-edik oszlopon egy-egy tabulátor stop-ot jelent. Válasszunk (a felhasználó számára) kényelmes magatartást az alapesetre (default)!

5.10. Gyakorlat. Írjuk meg a tail nevű programot, amely kinyomtatja az utolsó n bemeneti sort! Az n alapértelmezése legyen pl. 10, amit opcionális argumentum változtathat meg, tehát tail – n az utolsó n sort nyomtatja ki. A programnak elfogadhatóan kell viselkednie, függetlenül attól, hogy mennyire értelmetlen a bemenet vagy n értéke. Írjuk meg úgy a programot, hogy a legjobban kihasználja a rendelkezésre álló tárterületet: a sorokat tároljuk úgy, mint a sort-ban, nem pedig rögzített méretű kétdimenziós tömbben!

5.12. Függvényeket megcímző mutatók

A C nyelvben maga a függvény nem változó, de mód van függvényt megcímző mutató definiálására, amellyel műveletek végezhetők, függvényeknek átadható, tömbökbe helyezhető stb. Mindezt a fejezet korábbi részében bemutatott rendezőprogram módosításával szemléltetjük: ha megadjuk a -n opcionális argumentumot, akkor a program nem lexikografikusan, hanem numerikusan rendezi a bejövő sorokat. A rendezés gyakran három részből áll - összehasonlításból, amely tetszőleges objektumpár sorrendjét határozza meg, cseréből, amely megfordítja ezek sorrendjét és rendezőalgoritmusból, amely mindaddig végzi az összehasonlításokat és cseréket, amíg az objektumok a helyes sorrendbe nem kerülnek.

Maga a rendezőalgoritmus független az összehasonlítási és felcserélési műveletektől, így különböző összehasonlító és felcserélő függvényeket átadva különböző kritériumok szerint rendezhetünk. Ezt az elvet alkalmazzuk az új rendezőprogramban.

A korábbiaknak megfelelően két sor lexikografikus összehasonlítását az strcmp, felcserélését pedig a swap végzi. Szükségünk lesz még a numcmp rutinra, amely numerikus értékük alapján hasonlít össze két sort, és strcmp-hez hasonlóan valamiféle feltételjelzést ad vissza. Ezt a három függvényt a mainben deklaráljuk, és az őket megcímző mutatókat adjuk át sort-nak. A sort viszont a mutatókon keresztül hívja a függvényeket. Az argumentumokkal kapcsolatos hibafeldolgozást elhanyagoltuk, hogy a fő feladatokkal foglalkozhassunk.

```
#define LINES 100  /*A rendezendő sorok maximális száma*/
/*Beolvasott sorok rendezése*/
main (int argc, char *argv [])
{
    char *lineptr [LINES];        /*Mutatók a szövegsorokra*/
    int nlines;                   /*A beolvasott sorok száma*/
    int strcmp (), numcmp ();     /*Összehasonlító   függvények*/
    int swap ();                  /*Felcserélő függvény*/
    int numeric = 0;              /*1, ha a rendezés numerikus*/
    if (argc > 1 && argv [1][0] == '-' && argv [1][1] =='n')
         numeric = 1;

    if ((nlines = readlines (lineptr, LINES)) >= 0) {
         if (numeric)
              sort(lineptr, nlines, numcmp, swap);
         else
              sort (nlineptr, lines, strcmp, swap);
          writelines (lineptr, nlines);
    } else
         printf ("a bemenet túl nagy a rendezéshez \n");
 }
```

Az strcmp, numcmp és swap - függvények címei; mivel ezek bizonyosan függvények, az & operátor ugyanúgy nem szükséges, mint ahogy nincs rá szükség a tömbök neve előtt sem. A fordító gondoskodik a függvény címének átadásáról. Második lépésben a sort-ot módosítjuk:

```
/*A v[0] ... v[n-1] karakterláncok  rendezése növekvő sorrendbe*/
sort (char *v [], int n, int (*comp) () , (*exch) ())
{
    int gap, i, j;
    for (gap = n / 2; gap > 0; gap /= 2)
         for (i = gap; i < n; i++)
              for (j = i - gap; j >= 0; j -= gap)
              {
                   if ((*comp) (v [j], v [j + gap]) <= 0)
                             break;
                    (*exch) (&v [j], &v [j + gap]);
              }
}
```

Vizsgáljuk meg a deklarációkat!

```
int (*comp) ()
```

azt fejezi ki, hogy a comp olyan függvényt megcímző mutató, amely int-et ad vissza. Az első zárójelpár szükséges, nélkülük

```
int *comp ()
```

azt fejezné ki, hogy a comp olyan függvény, amely int-et megcímző mutatót ad vissza, ami egészen más dolog. comp használata az

```
if ((*comp) (v [j],v [j + gap]) <= 0)
```

sorban összhangban van a deklarációval: comp a függvényt megcímző mutató, *comp a függvény, és

```
 (*comp) (v [j], v [j + gap])
```

annak hívása. Ahhoz. hogy az összetevők helyesen kapcsolódjanak, szükség van a zárójelekre. Korábban már láttuk strcmp-t, amely két karakterláncot hasonlít össze. Íme numcmp amely vezető numerikus értékeik szerint hasonlít össze két karakterláncot :

```
/*s1 és s2 numerikus összehasonlítása*/
numcmp ( char *s1 , char *s2)
{
    double atof (), v1 , v2;
    v1 = atof (s1);
    v2 = atof (s2);
    if (v1 < v2)
         return (-1);
    else if (v1 > v2)
              return (1);
          else
              return (0);
}
```

Az utolsó lépés a két mutató felcserélését végző swap függvény megírása. Ezt közvetlenül arra alapozhatjuk, amit a fejezet korábbi részében már közöltünk:

```
/* *px és *py felcserélése*/
swap (char *px [], char *py [])
{
    char *temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

A rendezőprogram további opciók egész sorával egészíthető ki; ezek közül néhány érdekes gyakorlat lehet.

5.11. Gyakorlat. Módosítsuk a sort programot oly módon, hogy kezelje -r-t, amely ellentétes irányú (csökkenő) rendezést ír elő! -r-nek természetesen -n-nel is működnie kell!

5.12. Gyakorlat. Illesszük hozzá a -f opciót, amellyel egyesítjük a kis- és nagybetűket úgy, hogy azokat a rendezés során nem különböztetjük meg: a kis- és nagybetűs adatokat együtt rendezzük, tehát a és A szomszédosként jelennek meg, nem választja el ők et az egész ábécé.

5.13. Gyakorlat. Illesszük a függvényhez a -d (szótári rendezés) opciót, amely csak betűket, számokat és szóközöket hasonlít össze! Ügyeljünk arra, hogy -f-fel együtt is működjön!

5.14. Gyakorlat. Egészítsük ki a függvényt mezőkezelő szolgáltatással, amely lehetővé teszi, hogy sorokon belül kijelölt mezőkön is végezhető legyen rendezés, mégpedig minden mezőn egymástól független opciókészlet szerint! (E könyv angol eredetijének tárgymutatóját kulcsszavakra -dffel lapszámokra -n-nel rendezték.)

6.fejezet : Struktúrák

A struktúra egy vagy több, esetleg különböző típusú változó együttese, amelyeket az egyszerű kezelhetőség érdekében gyűjtünk össze. A struktúrákat egyes nyelvekben, legismertebben a PASCAL-ban rekordoknak nevezik. A struktúrát szemléltető hagyományos példa a bérfizetési lista:

a dolgozót egy attribútum-készlettel jellemezzük, amelyben helyet kap az illető neve, címe, társadalombiztosítási száma, bére stb. Ezek némelyike akár struktúra is lehet: a név, a cím vagy éppen a bér maga is több részből állhat. A struktúrák különösen a nagy méretű programok esetében nyújtanak segítséget bonyolult adathalmazok szervezésében, mivel sokszor lehetővé teszik, hogy az összetartozó változók együttesét egy egységként, nem pedig különálló elemekként kezeljük.

Ebben a fejezetben megkíséreljük bemutatni a struktúrák használatát. Mintaprogramjaink a könyvünkben megszokottaknál terjedelmesebbek lesznek, de még mindig elég szerény méretűek.

6.1. Alapfogalmak

Vegyük ismét elő az 5. fejezetben tárgyalt dátumkonverziós rutinokat. Egy-egy adat több részből áll, ilyenek a nap, a hónap és az év, továbbá esetleg a nap sorszáma az évben, és a hónap neve. Ez az öt változó az alábbi módon egyetlen struktúrába foglalható:

```
struct date {
    int day;
    int month;
    int year;
    int yearday;
    char mon_name [4];
}
```

A **struct** kulcsszó a struktúra deklarációját vezeti be, amely nem más, mint egy kapcsos zárójelek közé zárt deklarációlista. A **struct** kulcsszót struktúracímke követheti (mint pl. az előbb a date). Ez egy név, amely megnevezi az adott típusú struktúrát, és a továbbiakban rövidítésként használható a részletes deklaráció helyett. A struktúrában előforduló elemeket, ill. változatokat tagoknak nevezzük. Valamely struktúratagnak, ill. -elemnek és egy közönséges (vagyis nem tag) változónak lehet ugyanaz a neve: ebből nem származik zavar, mivel ezek a szövegkörnyezet alapján mindig megkülönböztethetők. Az már természetesen stílus kérdése, hogy az ember ugyanazokat a neveket csak szorosan összetartozó objektumok esetében használja. A tagok listáját lezáró jobboldali zárójelet a változólista követheti ugyanúgy, mint minden alaptípusnál. Eszerint

```
struct { . . . } x, y, z;
```

szintaktikusan hasonló az

```
int x, y, z;
```

sorhoz abban az értelemben, hogy mindkét utasítás a megnevezett típusú változóként deklarálja x-et, y-t és z-t, ill. helyet foglal számukra. Az olyan struktúradeklaráció, amit nem követ változólista, nem foglal tárhelyet, csupán a struktúra alakját (template) írja le. H azonban a deklaráció címkézett, akkor ez a címke a későbbiekben a struktúra konkrét előfordulásakor a definíciókban használható. Ha pl. adott a date előbbi deklarációja, akkor

```
struct date d;
```

úgy definiálja a d változót, hogy az date típusú struktúra legyen.

A külső, ill. a statikus struktúra úgy inicializálható, hogy definícióját az elemek kezdeti értékeiből álló lista követi:

```
struct date d = {4, 7, 1776, 186, "júl"};
```

Valamely struktúra adott tagjára kifejezésen belül a

```
struktúranév.tag
```

alakú szerkezettel lehet hivatkozni. A "." struktúratag operátor a struktúra nevét és a tag nevét kapcsolja össze. Ha pl. a d struktúrabeli dátum alapján akarjuk leap-et (a szökőévet) beállítani, akkor a kód

```
leap = d.year % 4 == 0 && d.year % 100 != 0   || d.year % 400 == 0;
```

lesz. A hónap nevének vizsgálata:

```
if (strcmp (d.mon_name, "aug") == 0) . . .
```

Ha a hónapneveket az angol helyesírás szerint nagy kezdőbetűkkel írtuk volna, kisbetűssé alakításuk a következőképpen történhetne:

```
d.mon_name [0] = lower (d.mon_name [0]);
```

A struktúrák egymásba skatulyázhatók; a fizetési jegyzék pl. így nézhet ki:

```
struct person {
    char name [NAMESIZE];
    char address [ADRSIZE];
    long zipcode;
    long ss_number;
    double salary;
    struct date birthdate;
    struct date hiredate;
};
```

A person nevű struktúra két dátumot tartalmaz. Ha az emp-et mint

```
struct person emp;
```

deklaráljuk, akkor

```
emp.birthdate.month
```

a születés hónapjára vonatkozik. A .struktúratag operátor balról jobbra köt.

6.2. Struktúrák és függvények

A C nyelvbeli struktúrákra számos megkötés vonatkozik. Ezek közül a leglényegesebb, hogy struktúrán

csak kétféle művelet végezhető – a struktúra címének előállítása az & szimbólum használatával, és a struktúra valamelyik tagjához való hozzáférés. Ebből következőleg a struktúrákat egy egységként nem lehet semmihez sem hozzárendelni (értékül adni), ill. másolni, nem adhatók át függvényeknek, és a függvények sem adhatnak vissza struktúrákat. (A C nyelv később megjelenő változataiból ezek a megkötések ki fognak maradni.) A struktúrákat megcímző mutatókra már nem vonatkoznak ezek a korlátozások, így a struktúrák és a függvények kényelmesen együtt tudnak működni. Végezetül, az automatikus tömbökhöz hasonlóan az automatikus struktúrák sem inicializálhatók, ez csak a külső és a statikus struktúrák esetében lehetséges.

Vizsgáljunk meg e jellegzetességek közül néhányat! Példaként írjuk át az előző fejezetben látott dátumkonverziós rutint, struktúrák használatával! Mivel a szabályok nem engedik meg, hogy struktúrát függvénynek közvetlenül átadjunk, vagy külön-külön az elemeket, vagy az egészt megcímző mutatót kell átadnunk. Az első lehetőség úgy használja day_of_year-t, ahogy az 5. fejezetben leírtuk:

```
d.yearday = day_of_year(d.year, d.month, d.day);
```

A másik lehetőség a mutatóátadás. Ha a hiredate-et

```
 struct date hiredate;
```

alakban deklaráltuk és day_of_year-t átírtuk, akkor

```
 hiredate.yearday = day_of_year (&hiredate);
```

segítségével a hiredate-et megcímző mutatót átadhatjuk day_of_year-nek. A függvényt módosítani kell, mivel argumentuma a korábbi változólista helyett most mutató lett:

```
/*Az év napjának előállítása a hónapból és a napból*/
day_of_year (struct date *pd)
{
    int i, day, leap;
    day = pd->day;
    leap = pd->year % 4 == 0 && pd->year % 100 != 0
          || pd->year % 400 == 0;
    for (i = 1; i < pd->month; i++)
    day += day_tab [leap][i];
    return (day);
}
```

A

```
struct date *pd;
```

deklaráció szerint pd olyan mutató, amely date típusú struktúrára mutat. A

```
pd->year
```

példa szerinti jelölésmód új. Ha p struktúrát megcímző mutató, akkor

```
p->struktúratag
```

az adott tagra vonatkozik. (A -> operátor mínuszjelből és az azt követő >-ból áll.) Mivel pd a struktúrára mutat, a year tagra a

```
(*pd).year
```

alakban is hivatkozhatunk, de struktúrákat jelölő mutatókat olyan gyakran használunk, hogy kényelmes rövidítésként a -> jelölésmód is rendelkezésre áll. A (*pd).year alakban a zárójelek szükségesek, mivel a . struktúratag operátor precedenciája magasabb, mint a * operátoré. Mind ->, mind pedig . balról

jobbra köt, így

```
p->q->memb
emp.birthdate.month
```

értelme:

```
(p->)->memb
(emp.birthdate).month
```

A teljesség kedvéért íme a másik függvény, month_day, amit szintén a struktúra használatával írunk át :

```
/*Hónap és nap előállítása az év napjából*/
month_day (struct date *pd)
{
    int i, leap;
    leap = pd->year % 4 == 0 && pd->year % 100 != 0
              || pd->year % 400 == 0;
    pd->day = pd->yearday;
    for (i = 1; pd->day > day_tab [leap][i]; i++)
         pd->day -= day_tab [leap][i];

    pd->month = i;
}
```

A -> és . struktúraoperátorok, valamint az argumentumlistákat közrefogó () és az indexet tartalmazó [] a precedenciahierarchia csúcsán áll, és ezért nagyon szorosan köt. Ha pl. adott a

```
struct {
    int x;
    int *y;
} *P;
```

deklaráció, akkor

```
++p->x
```

nem p-t, hanem x-et inkrementálja, mivel az alapértelmezés szerinti zárójelezés:

```
++(p->x)
```

Zárójelek használatával a kötés megváltoztatható: (++p)->x az x-hez való hozzáférés előtt növeli p-t, míg (p++)->x azt követően inkrementál. (Az utóbbi zárójelkészlet felesleges.) Hasonlóképpen, *p->y

behozza, amire p mutat; *p->y++ azután inkrementálja y-t, miután hozzáfért ahhoz, amire mutat (éppúgy, mint *s++); (*p->y) ++ azt növeli, amire y mutat, míg *p++->y azután inkrementálja p-t, hogy hozzáfért ahhoz, amire y mutat.

6.3. Struktúratömbök

A struktúrák különösen alkalmasak egymással összefüggő változók tömbjeinek kezelésére. Tekintsük pl. azt a programot, amely a C nyelv kulcsszavainak előfordulásait számlálja. A nevek tárolásához szükségünk lesz egy karakterlánc-tömbre, a darabszámok tárolásához pedig egy egészekből álló tömbre. Az egyik megoldás szerint párhuzamosan két tömböt használunk. Legyenek ezek pl. keyword és keycount:

```
char *keyword [NKEYS];
int keycount [NKEYS];
```

Azonban maga a tény, hogy a tömbök párhuzamosak, jelzi, hogy lehetőségünk van másfajta szervezésre is. Valójában minden kulcsszóbejegyzés egy pár:

```
char *keyword;
int keycount;
```

Hozzuk létre a párok tömbjét! A

```
struct key {
     char *keyword;
    int keycount;
} keytab [NKEYS];
```

struktúradeklaráció az ilyen típusú struktúrák keytab nevű tömbjét definiálja és tárterületet foglal le számára. A tömb minden eleme struktúra. Ezt így is írhatjuk:

```
struct key {
    char *keyword;
    int keycount;
};

struct key keytab [NKEYS];
```

Mivel a keytab struktúra jelen esetben a nevek állandó halmazát tartalmazza, legegyszerűbb, ha definiáláskor egyszer és mindenkorra inicializáljuk. A struktúrák inicializálása mindenben hasonlít a

korábbiakhoz - a definíciót a kezdeti értékek kapcsos zárójelek közé zárt listája követi:

```
struct key {
    char *keyword;
    int keycount;
} keytab [] = {
    "break", 0,
    " case", 0,
    " char", 0,
    " continue", 0,
    " default", 0,
    /* ... */
    "unsigned" , 0,
    "while", 0,
};
```

A kezdeti értékeket a struktúratagoknak megfelelően páronként soroltuk fel. Pontosabb lenne, ha minden sor vagy struktúra kezdeti értékeit zárnánk kapcsos zárójelek közé

```
{"break", 0},
{" case", 0},
. . .
```

szerint, de a belső kapcsos zárójelekre nincs szükség, ha a kezdeti értékek egyszerű változók vagy karakterláncok, mint a jelen esetben is. Amennyiben kezdeti értékek vannak megadva, és a [] üresen maradt, a fordító most is ezek alapján számítja ki a key tab tömb elemeinek számát.
A kulcsszó-számláló program a keytab definiálásával kezdődik. A főrutin a getword függvény ismételt hívásával szavanként olvassa a bemenetet. A program minden szót megkeres a keytab-ben a bináris keresőfüggvénynek a 3. fejezetben látott változatával. (A helyes működés érdekében a kulcsszavak listáját természetesen növekvő sorrendben kell megadnunk.)

```
#define MAXWORD 20

/*Szó megkeresése tab[0] . . . tab[n-1]-ben*/
binary (char *word, struct key tab [], int n)
{
    int low, high, mid, cond;
    low = 0;
    high = n - 1;
    while (low <= high) {
         mid = (low + high) / 2;
         if ((cond = strcmp (word, tab [mid].keyword)) < 0)
              high = mid - 1;
         else if (cond > 0)
                   low = mid + 1;
                     else
                   return (mid);
    }
    return (-1);
}

/*C kulcsszavak számlálása*/
main ()
{
    int n, t;
    char word [MAXWORD];
    while ((t = getword (word, MAXWORD)) != EOF)
         if (t == LETTER)

    if ((n = binary (word, keytab, NKEYS)) >= 0)
              keytab [n].keycount++;

    for (n = 0; n < NKEYS; n++)
         if (keytab [n].keycount > 0)
               printf ("%4d %s \n",
                        keytab [n].keycount, keytab [n].keyword);
}
```

Rövidesen sor kerül a getword függvény bemutatására; egyelőre elég annyit tudnunk, hogy minden alkalommal, amikor megtalál egy kulcsszót, a LETTER értéket adja vissza és az illető szót az első argumentumába másolja. Az NKEYS mennyiség a kulcsszavak száma a keytab-ban. Bár ezt magunk is megszámlálhatnánk, sokkal könnyebb és biztonságosabb, ha a gépre bízzuk, különösen, ha a lista változhat.

Az egyik lehetőség az lenne, hogy a kezdeti értékek listáját a nulla mutatóval zárjuk le,majd ciklusban

addig haladunk keytab-on, amíg a végét meg nem találtuk. Ez azonban több, mint amire szükség van, mivel a fordító a tömb méretét fordítás közben pontosan meghatározza. A bejegyzések száma ebből:

keytab mérete / **struct** key mérete

A C-ben rendelkezésünkre áll a sizeof egyoperandusú operátor, amelynek segítségével bármilyen objektum mérete fordítási időben meghatározható. A

```
sizeof (objektum)
```

kifejezés eredménye olyan egész szám, amely megegyezik a megadott objektum méretével. (A méretet byte-nak nevezett specifikálatlan egységekben kapjuk, amelynek mérete ugyanaz, mint egy char-é.) Az

objektum valamilyen aktuális változó, tömb vagy struktúra , vagy pedig valamilyen alap-, ill. leszármaztatott típus (l. **int** vagy **double**, ill. struktúrák) neve lehet. Esetünkben a kulcsszavak száma a tömbméret osztva egy tömbelem méretével. Ezt a számítást #define utasításban használva, állítjuk be az NKEYS értékét:

```
#define NKEYS (sizeof (keytab) / sizeof (struct key))
```

Most pedig nézzük a getword függvényt. A getword-nek az adott program számára szükségesnél általánosabb változatát írtuk meg, amely azonban nem lényegesen bonyolultabb, getword a bemeneten soron következő szót adja vissza, ahol a szó vagy betűk és számok betűvel kezdődő lánca, vagy pedig egyetlen karakter. Az objektum típusát függvényértékként kapjuk meg; ez az érték LETTER, ha az adott egység szó, EOF az állomány végén, vagy maga a karakter, ha az nem alfabetikus.

```
/*Vedd a következő szót a bemenetről*/
getword (char *w, int lim)
{
    int c,t;
    if (type (c = *w++ = getch ()) != LETTER) {
         *w = '\0';
         return (c);
    }

    while (--lim > 0) {
         t = type (c = *w++ = geth ());
         if (t != LETTER && t != DIGIT) {
              ungetch (c);
              break;
         }
    }
    *(w - 1) = '\0';
    return (LETTER);
}
```

A getword a getch és ungetch rutinokat használja, amelyeket a 4. fejezetben írtunk meg. Amikor egy alfabetikus szövegegység begyűjtése befejeződik, a getword már a szükségesnél egy karakterrel többet olvasott be. Az ungetch hívásával ezt a karaktert a getword következőhívásáig visszaírjuk a bemenetre. A getword a type hívásával állapítja meg az egyes bemeneti karakterek típusát. Az alábbi változat csak az ASCII karakterkészletben működik:

```
/*ASCII karakter típusának visszaadása*/
type(int c)
{
    if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
         return (LETTER);
    else if (c >= '0' && c <= '9')
              return (DIGIT);
           else
              return (c);
}
```

A LETTER és a DIGIT szimbolikus állandó minden olyan értéket felvehet, amely nem ütközik a nem-alfanumerikus karakterekkel és az EOF-fal, kézenfekvő pl. az alábbi választás:

```
#define LETTER 'a'
#define DIGIT '0'
```

A getword-öt felgyorsíthatjuk, ha a type függvény hívásait valamely alkalmas type [] tömbre vonatkozó hivatkozásokkal helyettesítjük. A szabványos C könyvtárban rendelkezésünkre állnak az isalpha és isdigit nevű makrók, amelyek ily módon működnek.

6.1. Gyakorlat. Végezzük el a getword-nek ezt a módosítását, és mérjük meg a program sebességének változását!

6.2. Gyakorlat. Írjuk meg a type olyan változatát, amely független a karakterkészlettől!

6.3. Gyakorlat. Írjuk meg a kulcsszószámláló program olyan változatát, amely nem számlálja az idézőjelek közé zárt karakterláncokban előforduló kulcsszavakat!

6.4. Struktúrákat megcímző mutatók

A mutatókkal és struktúratömbökkel kapcsolatos megfontolásaink szemléltetésére írjuk újra a kulcsszószámláló programot, ezúttal tömbindexek helyett mutatók használatával! A keytab külső deklarációját nem kell megváltoztatnunk, a main és a binary azonban módosításra szorul.

```
/*Szó keresése  tab [0] . . . tab [n-1]-ben*/
struct key *binary (char *word, struct key tab [], int n)
{
    int cond;
    struct key *low = &tab [0];
    struct key *high = &tab [n - 1];  struct key *mid;
    while (low <= high) {
         mid = low + (high - low) / 2;
         if ((cond = strcmp (word, mid->keyword)) < 0)
              high = mid - 1;
         else if (cond > 0)
                   low = mid + 1;
                     else
                   return (mid);
    }
    return (NULL);
}

/*C kulcsszavak számlálása;   mutatót alkalmazó változat*/
main ()
{
    int t;
    char word [MAXWORD];
    struct key *binary (), *p;
    while ((t = getword (word, MAXWORD)) != EOF)
         if (t == LETTER)
              if ((p = binary (word, keytab, NKEYS)) != NULL)
                        p->keycount++;

     for (p = keytab; p < keytab + NKEYS; p++)
         if (p->keycount > 0)
              printf ("%4d %s \n", p->keycount, p->keyword);
}
```

Ebben a programban több dolog is említésre méltó. Először is binary deklarációjának jeleznie kell, hogy key típusú struktúrát megcímző mutatót ad vissza, és nem egész típusú mennyiséget; ezt mind a main-ben, mind binary-ban deklaráltuk. Ha binary megtalálta a szót, az azt kijelölő mutatót adja vissza; ha a keresés eredménytelen, a visszaadott érték NULL. Másodszor, a keytab elemeihez történő minden hozzáférés mutatókkal történik. Emiatt a binary rutin jelentősen megváltozik: a középső elem kiszámítása már nem lehet egyszerűen

```
mid = (low + high) / 2
```

mivel két mutató összeadása semmiféle értelmes választ nem eredményez (még a 2-vel való osztáskor sem), sőt ez a művelet tiltott ! Ehelyett a

```
mid = low +(high – low) / 2
```

alakra van szükség, amely úgy állítja be mid-et, hogy az a low és a high közötti terület felezőpontjára mutasson. Figyeljük meg low és high kezdeti értékeit is! Lehetőség van arra, hogy egy mutatót valamely korábban definiált objektum címével inicializáljunk: itt éppen ezt tettük. A main-ben azt írtuk, hogy

```
for (p = keytab; p < keytab + NKEYS; p++)
```

Ha p struktúrát megcímző mutató, akkor minden p-re vonatkozó aritmetikai számítás figyelembe veszi a struktúra tényleges méretét, így p++ a p-t a megfelelő mennyiséggel inkrementálja ahhoz, hogy előálljon a struktúrák tömbjének következő eleme. Ne higgyük azonban, hogy a struktúra mérete megegyezik az elemei méretének az összegével – a különböző jellegű objektumok elhelyezkedési követelményeinek folytán lyukak lehetnek a struktúrában. Végezetül egy megjegyzés a program alakjával kapcsolatban. Ha egy függvény bonyolult típust ad vissza, mint a

```
struct key *binary(word, tab, n)
```

esetben, akkor a függvény nevét esetleg nehéz észrevenni, vagy szövegszerkesztő programmal (text editorral) megtalálni. Emiatt néha más formát szokás használni:

```
struct key
binary (word, tab, n)
```

Ez természetesen leginkább személyes ízlés kérdése: válasszuk ki a nekünk tetsző alakot, és ahhoz tartsuk magunkat.

6.5. Önhivatkozó struktúrák

Tegyük fel, hogy általánosabb feladatként a bemeneti szöveg összes szavának előfordulásait akarjuk megszámlálni. Mivel a szavak listája előzetesen nem ismert, azt nem tudjuk alkalmas módon rendezni és nem használhatunk bináris keresést. Lineáris keresést azonban végre tudunk hajtani minden beérkező szóra, amivel megnézzük, hogy volt-e már ilyen szó: a program futása azonban így örökké fog tartani. (Pontosabban szólva a várható futási idő a beolvasott szavak számával négyzetesen nő.) Hogyan szervezzük meg az adatokat ahhoz, hogy hatékonyan meg tudjunk birkózni a tetszőleges szavakból álló listával? Az egyik megoldás szerint állandóan rendezett állapotban tartjuk a már megvizsgált szavakat oly módon, hogy a beérkezés sorrendjében minden szót a neki megfelelő helyre teszünk. Ezt azonban nem úgy végezzük el, hogy a szavakat egy lineáris tömbben tologatjuk, mivel ez is túl sokáig tartana. Ehelyett a bináris fa nevű adatstruktúrát fogjuk használni. A fa minden különböző szóhoz egy-egy csomópontot rendel, amelynek tartalma

a szó szövegét megcímző mutató,

a szó előfordulásainak száma,

a bal oldali gyermek (leszármazott) csomópontot megcímző mutató,

a jobb oldali gyermek csomópontot megcímző mutató.

Egyetlen csomópontnak sem lehet kettőnél több gyermeke; lehet viszont nulla vagy egy gyermeke. A csomópontokat úgy hozzuk létre, hogy minden egyes csomópont esetében a bal oldali részfa csupa olyan szót tartalmaz, amely kisebb, mint a csomópontbeli szó, míg a jobb oldali részfában csupa olyan szó van, amely nála nagyobb. Ha el akarjuk dönteni, hogy egy új szó már rajta van-e a fán, a vizsgálatot a fa gyökerénél kezdjük, és az új szót az illető csomópontban tárolt szóval hasonlítjuk össze. Ha megegyeznek, a válasz igenlő. Ha az új szó kisebb, mint a csomópontbeli szó, a keresés a bal oldali, ellenkező esetben a jobb oldali gyermek csomópontban folytatódik. Ha a kiválasztott irányban nincs leszármazott, a szó nincs a fán, és éppen a hiányzó leszármazottnak megfelelő csomópontba kell beírnunk. Ez a keresési eljárás rekurzív, hiszen bármelyik csomóponttól induló keresés tartalmazza a valamelyik leszármazottjától induló keresést is. Ennek megfelelően a legtermészetesebb az, ha a beillesztésre és kiírásra is rekurzív rutinokat használunk. Visszatérve a csomópont leírására, a csomópont nyilván struktúra lesz, amely négy összetevőből áll:

```
struct tnode { /*Alapcsomópont*/
    char *word;  /*A szövegre mutat*/
    int count;  /*Előfordulások száma*/
    struct tnode *left;  /*Bal oldali gyermek*/
    struct tnode *right; /*Jobb oldali gyermek*/
}
```

A csomópontnak ez a rekurzív deklarációja talán bizonytalanul fest, de valójában teljesen helyes és pontos. A struktúra nem tartalmazhatja saját megnevezését, de

```
struct tnode *left;
```

a left-et a csomópontot megcímző mutatónak, nem pedig csomópontnak deklarálja. Az egész program meglepően rövid, mivel már korábban megírt segédrutinokat használ. Ezek: a getword, amellyel az egyes bemeneti szavakat olvassuk be, és az alloc, amellyel helyet biztosítunk a szavak arrébb csúsztatásához. A főrutin egyszerűen a getword segítségével beolvassa a szavakat és a tree használatával elhelyezi azokat a fán.

```
#define MAXWORD 20
main () /*Szavak gyakoriságának számlálása*/
{
    struct tnode *root, *tree ();
    char word [MAXWORD];
    int t;
    root = NULL;
    while ((t = getword (word, MAXWORD)) != EOF)
         if (t == LETTER)

    root = tree (root, word);
    treeprint (root);
}
```

Maga a tree egyszerű. A main a fa tetején (a gyökér szintjén egy szót ad át. Ezt a szót minden lépésben összehasonlítjuk az adott csomópontnál már tárolt szóval és a tree rekurzív hívásai révén vagy a bal oldali, vagy a jobb oldali részfa irányában haladunk tovább. Végül is a szó vagy megegyezik valamelyik, már a fán tárolt szóval (amikor is előfordulásainak számát növeljük), vagy pedig a program a NULL mutatót találja meg, ami azt jelzi, hogy új csomópontot kell létrehozni és a fát azzal ki kell bővíteni. Új csomópont létrehozásakor a tree az azt megcímző mutatóval tér vissza, ami bekerül a szülő csomópontba:

```
/*w elhelyezése p-nél vagy p alatt*/
struct tnode *tree (struct tnode *p, char *w)
{
    struct tnode *talloc ();
    char *strsave ();
    int cond;
    if (p == NULL) {                    /*új szó érkezett*/
         p = talloc ();                      /*új csomópont készül*/
         p->word = strsave (w);
         p->count = 1;
         p->left = p->right = NULL;
    }
    else if ((cond = strcmp (w, p->word)) == 0)
          p->count++;              /*Már volt ilyen szó*/
                else if (cond < 0) /*Kisebb - a bal részfába kerül*/
                   p->left = tree (p->left, w);
                else          /*Nagyobb - a jobb részfába kerül*/
                   p->right = tree (p->right, w);

    return (p);
}
```

Az új csomópont számára szükséges tárhelyet a talloc szolgáltatja, amely a már korábban megírt alloc módosított változata. A talloc rutin a fa csomópontjának tárolására alkalmas szabad területet megcímző mutatót ad vissza. (erről röviden részletesebben i s szólunk.) Az új szót az strsave másolja be egy rejtett helyre, a darabszám inicializálódik, és a két leszármazott nulla lesz. A programkódnak ezt a részét csupán a fa szélein hajtjuk végre, amikor új csomópontot iktatunk be. A strsave és a talloc által visszaadott értékek hibaellenőrzését elhagytuk (ami élesben használt program esetében nem bölcs dolog). A treeprint a fát a bal oldali részfa sorrendjében nyomtatja ki; minden egyes csomópontnál kinyomtatja a bal oldali részfát (minden olyan szót, amely a kérdéses szónál kisebb), majd magát a szót és végül a jobb oldali részfát (minden olyan szót, amely nagyobb). Ha az olvasó bizonytalannak érzi magát a rekurziós technikával kapcsolatban, rajzoljon le egy fát és nyomtassa ki a treeprint-tel: kevés ennél áttekinthetőbb rekurzív rutint találhatunk.

```
/*A p fa rekurzív kinyomtatása*/
treeprint (struct tnode *p)
{
    if (p != NULL) {
         treeprint (p->left);
          printf ("%4d %s \n", p->count, p->word);
          treeprint (p->right);
    }
}
```

Gyakorlati megjegyzés: ha a fa "kiegyensúlyozatlanná" válik, mert a szavak nem véletlenszerű sorrendben érkeznek, a program futási ideje túl gyorsan növekedhet. A legrosszabb eset az, amikor a szavak már sorrendben vannak, ilyenkor ez a program igen költséges módon szimulálja a lineáris keresést. A bináris fának vannak általánosításai, mégpedig a 2-3 fák és az AVL fák, amelyek mentesek ettől a legrosszabb esetben bekövetkező viselkedéstől, könyvünkben azonban ezeket nem ismertethetjük. Mielőtt befejeznénk a példát, érdemes rövid kitérőt tennünk a tárterület-lefoglalással kapcsolatos egyik problémára. Nyilván jó lenne, ha a programban csak egy tárfoglaló függvény lenne, még akkor is, ha az különféle jellegű objektumok számára foglal helyet. Ha azonban ugyanaz a függvény foglal helyet pl. char-okat és **struct** tnode-okat megcímző mutatók számára, két kérdés merül fel. Először is, hogyan elégíti ez ki a legtöbb valóságos gépnek azt a követelményét, hogy bizonyos típusú objektumok adott elhelyezési előírásoknak kell, hogy eleget tegyenek? (pl. az egész típusú mennyiségeket gyakran páros címen kell elhelyezni.) Másodszor, milyen deklarációk tudnak megbirkózni azzal a ténnyel, hogy az alloc szükségszerűen különböző típusú mutatókat ad vissza?

Az elhelyezési előírásoknak általában - némi hely elvesztése árán -egyszerűen úgy tehetünk eleget, ha gondoskodunk arról, hogy a helyfoglaló mindig olyan mutatót adjon vissza, amely az összes elhelyezési követelménynek eleget tesz. A PDP-11-en pl. elegendő, ha az alloc mindig páros mutatót ad vissza, mivel páros címen bármilyen típusú objektum tárolható. Ennek ára csupán egyetlen elvesztett karakterpozíció páratlan_ hosszúságú mennyiség esetén. Hasonló intézkedés tehető más gépeken is. Így lehet, hogy az alloc megvalósítása nem gépfüggetlen, a használata azonban az. Az 5. fejezetben ismertetett alloc semmiféle megkülönböztetett elhelyezkedést sem garantál, a 8. fejezetben bemutatjuk, hogyan kell helyesen megoldani ezt a feladatot. Az alloc típusdeklarációjának kérdése minden olyan nyelvben gondot okoz, amely komolyan veszi a típusellenőrzést. A C-ben a legjobb eljárás az, ha az alloc-ot úgy deklaráljuk, hogy char-t megcímző mutatót adjon vissza, majd típusmódosító szerkezettel explicit kényszerrel változtatjuk a mutatót a kívánt típusúvá. Ha tehát p deklarációja

```
char *p;
```

akkor

```
(struct tnode *)p
```

egy kifejezésben p-t tnode mutatóvá alakítja át. Így a talloc leírása:

```
struct tnode *talloc ()
{
    char *alloc ();
    return ((struct tnode *) alloc (sizeof (struct tnode)));
}
```

Ez már több, mint amire a jelenlegi fordítók esetében szükség van, azonban jelzi a jövőre nézve a legbiztosabb irányt.

6.4. Gyakorlat. Írjunk olyan programot, amely beolvas egy C programot, és alfabetikus sorrendben kinyomtatja a változóneveknek azokat a csoportjait, amelyek első 7 karakterükben megegyeznek, azonban ezt követően valahol különböznek! Ügyeljünk arra, hogy a 7 paraméter legyen!

6.5. Gyakorlat. Írjunk elemi, keresztbe hivatkozó (cross-referencing) programot, amely kinyomtatja egy dokumentumban előforduló szavak listáját, és minden szóra megadja azoknak a soroknak a sorszámát, ahol az illető szó előfordul!

6.6. Gyakorlat. Írjunk olyan programot, amely a bemenetén előforduló szavakat az előfordulás csökkenő sorrendjébe rendezve nyomtatja ki! Minden szó elé írjuk oda az előfordulások számát!

6.6. Keresés táblában

Ebben a fejezetben egy táblakereső (table-lookup) programcsomag belsejét írjuk meg, amivel a struktúrák további vonatkozásait illusztráljuk. Tipikusan ilyen programkód található a makroprocesszorok vagy fordítók szimbólumtábla-kezelő rutinjaiban. Tekints ük pl. a C #define utasítását. Ha egy olyan sor, mint

```
#define YES 1
```

fordul elő, a YES név és az 1 helyettesítő szöveg bekerül egy táblázatba. A későbbiekben, amikor a YES név utasításokban fordul elő, pl.

```
inword = YES;
```

azt 1-gyel kell helyettesíteni. Két főrutin kezeli a neveket és a helyettesítő szövegeket. Az install(s, t) beírja az s nevet és a t helyettesítő szöveget egy táblázatba; az s és a t egyszerűen karakterláncok. A lookup(s) megkeresi s-et a táblázatban, és egy mutatót ad vissza, amely arra a helyre mutat, ahol s-et megtalálta, vagy pedig NULL, ha s nincs a táblázatban. Az ún. hash keresési algoritmust használjuk - a program a beérkező nevet kis pozitív egész számmá alakítja át, amelyet később azután egy mutatótömb indexelésére használ. A tömb egy eleme olyan blokkok láncának kezdetére mutat, amelyek az illető hash értékű neveket írják le. A tömbelem NULL, ha egyetlen név sem rendelkezik az adott hash értékkel. A láncban egy blokk olyan struktúra, amely a nevet megcímző mutatókat, a helyettesítő szöveget és a következő láncbeli blokkot megcímző mutatót tartalmazza. A lánc végét a következő blokk mutatójának nulla értéke jelzi :

```
/*Elemi tábla bejegyzés*/
struct nlist {
    char *name;
    char *def;
    struct nlist *next;  /*Következő bejegyzés a láncban*/
}
```

A mutatótömb:

```
#define HASHSIZE 100
static struct nlist *hashtab [HASHSIZE];      /*Mutatótábla*/
```

A lookup és az install által egyaránt használt hash értékképző függvény egyszerűen összeadja a láncbeli karakterértékeket és képezi az összegnek a tömbmérettel vett maradékát. (Ez nem a lehető legjobb algoritmus, de megvan az az előnye, hogy rendkívül egyszerű.)

```
/*Az s string hash értékének képzése*/
hash (char *s)
{
    int hashval;
    for (hashval = 0; *s != '\0'; ;)
         hashval += *s++;

    return (hashval % HASHSIZE);
}
```

A hash-eljárás a hashtab tömbben létrehoz egy kezdőindexet; ha a karakterlánc egyáltalán megtalálható, akkor az itt kezdődő blokkláncban kell lennie. A keresést a lookup végzi. Ha lookup megtalálja a bejegyzést, a megfelelő mutatót adja vissza; ha nem, akkor NULL-lal tér vissza.

```
/*s keresése hashtab-ben*/
struct nlist *lookup (char *s)
{
    struct nlist *np;
    for (np = hashtab [hash (s)]; np != NULL; np = np->next)
         if (strcmp (s, np->name) == 0)
              return (np); /*Megtalálta*/

    return (NULL);  /*Nem találta meg*/
}
```

Az install a lookup-ot használja annak eldöntésére, hogy a beállított név már jelen van-e. Ha igen, akkor az új definíció felülbírálja a régit, egyébként pedig teljesen új bejegyzés keletkezik. Az install NULL-t ad vissza, ha valamilyen oknál fogva nincs hely az új bejegyzés számára.

```
/*(name, def)  elhelyezése hashtab-ben*/
struct nlist *install (char *name, char *def)
{
    struct nlist *np, *lookup ();
    char *strsave (), *alloc ();
    int hasval;
    if ((np = lookup (name)) == NULL) {          /*Nem találta meg*/
         np = (struct nlist *) alloc (sizeof (np));
         if (np == NULL)
              return (NULL);

         if ((np->name = strsave (name)) == NULL)
              return (NULL);

         hashval = hash( np->name);
         np->next = hashtab [hashval];
         hashtab [hashval] = np;
    } else                                    /*Már ott van*/
         free (np->def);     /*Felszabadítja az előző definíciót*/

    if ((np->def (strsave (def))) == NULL)
         return (NULL);

    return (np);
}
```

Az strsave egyszerűen átmásolja az argumentumában megadott karakterláncot valamilyen biztos helyre, amit az alloc hívásával kapott. Ezt a programot az 5. fejezetben láttuk. Mivel az alloc és a free hívásai tetszőleges sorrendben előfordulhatnak, továbbá minthogy az elhelyezkedés is számít, az alloc-nak az 5. fejezetben közölt egyszerű változata itt nem elegendő (l. a 7. és 8. fejezetet).

6.7. Gyakorlat. Írjunk olyan rutint, amely a lookup és install által kezelt táblából töröl egy nevet és egy definíciót!

6.8. Gyakorlat. Az ebben a fejezetben közölt rutinokat, ill. getch-t és ungetch-t alapul véve valósítsuk meg a #define processzor egyszerű változatát, amely C programok számára használható!

6.7. Mezők

Ha szűkében vagyunk a tárhelynek, előfordulhat, hogy több objektumot egyetlen gépi szóban kell elhelyeznünk. Tipikus esete ennek az egybites feltételjelzők (flagek) alkalmazása, pl. a fordítóprogramok szimbólumtábláiban. Kívülről kényszerített adatformátumok, pl. hardvereszközök illesztésekor, gyakran igénylik azt a lehetőséget, hogy a gépi szó egyes darabjaihoz is hozzáférhessünk. Képzeljük el a fordítónak azt a részét, amely a szimbólumtáblát kezeli. Minden programbeli azonosítóhoz bizonyos információ társul, pl. hogy kulcsszó-e vagy sem, hogy külső és/vagy statikus

stb. mennyiségről van-e szó. Az ilyen információ kódolásának legtömörebb módja az egybites feltételjelzők készletének használata egyetlen char-on vagy int-en belül. Ez általában úgy történik, hogy a választott bitpozícióknak megfelelően egy maszk-készletet definiálnak, mint

```
#define KEYWORD 01
#define EXTERNAL 02
#define STATIC  04
```

(A számoknak kettő hatványainak kell lenniük.) Ezek után a biteket a 2. fejezetben ismertetett léptető, maszkoló és komplementáló operátorokkal már könnyen elérhetjük. Bizonyos fordulatok különösen gyakoriak:

```
flags = EXTERNAL | STATIC;
```

1-re állítja a flags-ben az EXTERNAL és STATIC biteket, míg

```
flags &= ~(EXTERNAL | STATIC);
```

ugyanezeket a biteket kinullázza, és

```
if ((flags &(EXTERNAL | STATIC)) == 0) . . .
```

igaz, ha mindkét bit nulla.

Bár ez a forma gyorsan elsajátítható, a C nyelv azt is lehetővé teszi, hogy valamely szón belül ne bitenkénti logikai operátorokkal, hanem közvetlenül definiáljunk és érjünk el egyes mezőket. A mező (field) szomszédos bitek halmaza egyetlen int-en belül. A meződefiníció és -elérés szintaxisa a struktúrákon alapul. Pl. az előbbi #define sorok három mező definiálásával helyettesíthetők:

```
struct {
    unsigned is_keyword : 1;
    unsigned is_extern : 1;
    unsigned is_static : 1;
} flags;
```

Ez a flags nevű változót definiálja, amely három 1-bites mezőt tartalmaz. A kettőspontot követő szám jelenti a mezőszélességet bitekben. A mezőket **unsigned**-nak deklaráltuk annak hangsúlyozására, hogy azok ténylegesen előjel nélküli mennyiségek. Az egyes mezőkre flags.is_keyword, flags.is_extern stb. alakkal hivatkozhatunk, éppúgy, mint más struktúratagok esetében. A mezők úgy viselkednek, mint kis, előjel nélküli egész számok, és éppúgy részt vehetnek aritmetikai műveletekben, mint más egészek. Így a fenti példákat természetesebb módon a következőképpen írhatjuk:

```
flags.is_extern = flags.is_static = 1;
```

amely 1-re állítja;

```
flags.is_extern = flags.is_static = 0;
```

amely kinullázza, és

```
if (flags.is_extern == 0 && flags.is_static == 0) . . .
```

amely megvizsgálja a biteket.

A mező nem lépheti át az **int** határát; ha a megadott szélesség erre vezetne, a mező a következő **int** határra fog illeszkedni. A mezőknek nem kell feltétlenül névvel rendelkezniük; név nélküli mezőket (csak egy kettőspont és a szélesség) használunk kitöltés re. A speciális 0 szélesség előírásával a következő **int** határra való illeszkedést kényszeríthetjük ki. A mezők használatával kapcsolatban néhány dologra ügyelnünk kell! Talán a leglényegesebb, hogy bizonyos gépeken a mezők hozzárendelése balról jobbra, más gépeken jobbról balra történik, ami az eltérő hardverfelépítést tükrözi. Ebből következőleg, bár a mezők igen hasznosak belsőleg definiált adatstruktúrák kezelésére, mielőtt külsőleg definiált adatok szétbontására használnánk őket, alaposan meg kell fontolni, milyen is lesz, hol kezdődik a mezőkiosztás. További megjegyzendő megkötések : a mezők előjel nélküliek; csak int-ekben tárolhatók (vagy az ezzel egyenértékű **unsigned**-okban); a mezők nem tömbök; nincsen

címük, így rájuk az & operátor nem alkalmazható.

6.8. Unionok

A **union** olyan változó, amely (különböző időpontokban) különféle típusú és méretű objektumokat tartalmazhat oly módon, hogy a fordító ügyel a méretre és illeszkedésre vonatkozó követelmények teljesülésére. A **union**ok lehetővé teszik, hogy ugyanazon a tárterületen különbözőfajta adatokkal dolgozzunk anélkül, hogy a programban gépfüggő információt kellene elhelyeznünk. Példánkat ismét a fordító szimbólumtáblájából véve tegyük fel, hogy állandóink int-ek, **float**-ok vagy karaktermutatók lehetnek. Valamely adott állandó értékét a megfelelő típusú változóban kell tárolnunk, ugyanakkor a táblakezelés szempontjából a legkényelmesebb, ha az érték ugyanannyi tárterületet foglal el és ugyanazon a helyen tárolódik, a típusától függetlenül. Ez a **union** használatának célja - olyan változót létrehozni, amely megengedett módon több típus bármelyikét tartalmazhatja. A mezőkhöz hasonlóan a szintaxis a struktúrákon alapul.

```
union u_tag {
    int ival;
    float fval;
    char *pval;
} uval;
```

Az uval változó elég nagy lesz ahhoz, hogy a három típus közül a legnagyobbat is tartalmazhassa, függetlenül attól a géptől, amelyen a fordítás történik - a programkód független a hardver jellemzőitől. E típusok bármelyike hozzárendelhető uval-hoz, majd kifejezésekben használható mindaddig, amíg a használat következetes: a visszanyert típus a legutoljára tárolt típussal kell, hogy megegyezzen. A programozó feladata annak követése, hogy éppen mit tárolt a union-ban; ha valamit adott típusként tárolunk és más típusúként olvassuk ki, akkor az eredmények gépfüggőek. A **union** tagjaihoz való hozzáférés szintaxisa:

```
unionnév.tag
```

vagy

```
unionmutató->tag
```

csakúgy, mint a struktúrák esetében. Ha az utype változó segítségével követjük az uval-ban tárolt aktuális típust, akkor az alábbihoz hasonló kódot kapunk:

```
if (utype == INT)
    printf ("%d \n", uval.ival);
else if (utype == FLOAT)
        printf ("%f \n", uval.fval);
        else if (utype == STRING)
              printf ("%s \n", uval.pval);
                else
              printf ("rossz típus %d az utype-ban\n", utype);
```

Unionok előfordulhatnak struktúrákban és tömbökben, ill. viszont. A struktúrabeli **union** (vagy megfordítva) egyik tagjához való hozzáférést leíró jelölésmód azonos az egymásba skatulyázott struktúrák jelölésmódjával. Pl. a

```
struct {
    char *name;
    int flags;
    int utype;
    union {
              int ival;
              float fval;
              char *pval ;
    } uval;
} symtab [NSYM];
```

által definiált struktúratömbben az ival változóra a

```
symtab [i].uval.ival
```

alakban, míg a pval karakterlánc első karakterére a

```
*symtab [i].uval.pval
```

alakban hivatkozhatunk. Valójában a **union** olyan struktúra, amelyben a tagok közötti eltolás nulla, és amely elég nagy ahhoz, hogy a legszélesebb tagot is tartalmazhassa úgy, hogy az illeszkedés a benne előforduló összes típus számára megfeleljen. A struktúrákhoz hasonlóan a union-okra jelenleg csak két

művelet megengedett: valamelyik tagjához való hozzáférés, ill. a cím előállítása. A unionok-hoz semmit sem lehet hozzárendelni, nem lehet őket függvényeknek átadni, és függvények sem adhatnak vissza **union**okat. A **union**okat megcímző mutatók ugyanúgy használhatók, mint a struktúrák mutatói. A 8. fejezetben bemutatásra kerülő tárterület-lefoglaló szemlélteti, hogyan lehet **union** használatával kikényszeríteni, hogy egy változó adott típusú tárterület határára illeszkedjen.

6.9. Típusnévdefiníciók

A C nyelv typedef-nek nevezett szolgáltatásának segítségével új adattípus-neveket hozhatunk létre. Pl. a

```
typedef int LENGTH;
```

deklaráció hatására a LENGTH név az **int** szinonimája lesz. A LENGTH típus deklarációban, típusmódosító szerkezetben stb. pontosan ugyanúgy használható, mint az **int** típus:

```
LENGTH len, maxlen;
LENGTH *lengths [];
```

Hasonlóképpen, a

```
typedef char * STRING;
```

deklaráció hatására a STRING a **char *** , vagyis a karaktermutató szinonimája lesz, amit azután olyan deklarációkban használhatunk, mint

```
STRING p, lineptr [LINES], alloc ();
```

Figyeljük meg, hogy a typedef-ben deklarált típus a változónév helyén jelenik meg, nem pedig közvetlenül a typedef szó után. A typedef szintaktikusan olyan, mint az **extern**, **static** stb. tárolási osztályok. A nevek hangsúlyozása érdekében nagybetűket használtunk. Bonyolultabb példaként typedef-eket készítünk az ebben a fejezetben korábban bemutatott facsomópontok számára:

```
typedef struct tnode { /*Alapcsomópont*/
         char *word;  /*A szövegre mutat*/
         int count;  /*Előfordulások száma*/
         struct tnode *left;  /*Bal oldali gyermek*/
         struct tnode *right; /*Jobb oldali gyermek*/
} TREENODE, *TREEPTR;
```

Ezzel két új típuskulcsszó, TREENODE (struktúra) és TREEPTR (a struktúramutatója)jön létre. Ekkor a talloc rutinból

```
TREEPTR talloc ()
{
    char *alloc ();
    return ((TREEPTR) alloc (sizeof (TREENODE)));
}
```

lesz. Hangsúlyozzuk, hogy a typedef deklaráció semmilyen értelemben sem hoz létre új típust; egyszerűen új nevet ad valamilyen, már létező típusnak. Szemantikailag sincs benne semmi új : az ily módon deklarált változók pontosan ugyanolyan tulajdonságúak, mint azok a változók, amelyeknek deklarációit explicit módon leírtuk. Valójában typedef olyan, mint #define, attól eltekintve, hogy mivel a fordító értelmezi, olyan szöveges helyettesítésekkel is meg tud birkózni, amelyek meghaladják a C makroprocesszor képességeit. Pl.:

```
typedef int (*PFI) ();
```

létrehozza a PFI típust az int-et visszaadó függvényt megcímző mutató számára, amely olyan összefüggésekben használható, mint

```
PFI strcmp, numcmp, swap;
```

az 5. fejezet rendezőprogramjában. A typedef deklarációk használatának két fő oka van. Az első ok a programok paraméterezése a gépfüggőségi problémák kivédésére. Ha a typedef-eket olyan adattípusokra használjuk, amelyek gépfüggőek, akkor a program áthelyezésekor csupán a typedef-eket kell megváltoztatni. Az egyik szokásos eset az, amikor különféle egész mennyiségek számára használunk typedef neveket, majd minden egyes befogadó gépre elkészítjük a **short**, **int** és **long**

választékából álló megfelelő készletet. A typedef-ek használatának másik célja a programdokumentálás javítása – a TREEPTR-nek nevezett típust könnyebb megérteni, mint azt, amelyiket csupán egy bonyolult struktúra mutatójaként deklaráltunk. Végezetül, mindig megvan annak a lehetősége, hogy a jövőben a fordító vagy valamelyik másik program, mint pl. a lint fel tudja használni a typedef deklarációkban tárolt információt a program valamilyen külön ellenőrzése céljából.

7. Bevitel és kivitel

A be- és kiviteli (I/O) szolgáltatások nem részei a C nyelvnek, ezért ezekre idáig nem fordítottunk különösebb figyelmet. Kétségtelen azonban, hogy a valódi programok sokkal bonyolultabb módon állnak kapcsolatban környezetükkel mint ahogy azt idáig bemutattuk. Ebben a fejezetben a szabványos (standard) be- és kiviteli könyvtárat ismertetjük; ez a függvényeknek olyan készlete, amelyek a C programok szabványos be- és kiviteli rendszerét képezik.

E függvények feladata, hogy kényelmes programozási csatlakozást biztosítsanak, ugyanakkor csupán olyan műveleteket valósítsanak meg, amelyek a legtöbb modern operációs rendszerben rendelkezésre állnak. A rutinok - függetlenül attól, hogy milyen kritikus alkalmazásról van szó - elég jól működnek, a felhasználók ritkán érezhetik úgy, hogy a nagyobb hatékonyság érdekében más megoldást kell alkalmazniuk. Végül a rutinok gépfüggetlenek abban az értelemben, hogy kompatibilis formában működnek minden olyan rendszeren, amelyen a C létezik, és azok a programok, amelyek a rendszerrel folytatott párbeszédüket a szabványos könyvtár által nyújtott szolgáltatásokra korlátozzák, lényegében változtatás nélkül vihetők át egyik gépről a másikra. Ezen a helyen nem kíséreljük meg a teljes be- és kiviteli könyvtár leírását; sokkal fontosabbnak tartjuk, hogy bemutassuk, hogyan kell az operációs rendszerbeli környezetükkel kapcsolatot tartó C programokat írni.

7.1. Hozzáférés a szabványos könyvtárhoz

Minden olyan forrásállománynak, amely valamelyik szabványos könyvtárbeli függvényre hivatkozik, valahol az állomány elején tartalmaznia kell az

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

sort. Az stdio.h állomány bizonyos, a be- és kiviteli könyvtár által használt makrókat és változókat definiál. A < és > könyökzárójeleknek a szokásos idézőjelek helyetti használata hatására a fordító az állományt abban a katalógusban (directory-ban) fogja keresni, amely a szabványos fej (header) információt tartalmazza (a UNIX-ban tipikusan /usr/include). A program betöltésekor szükséges lehet továbbá a könyvtár explicit megadása, a PDP-11 UNIX rendszeren pl. a program fordítását előíró parancs

```
cc forrásállományok stb. ls
```

ahol ls jelzi a szabványos könyvtárból történő betöltést. (Az l karakter az "el" betű, load = betölteni.)

7.2. Szabványos be- és kivitel; getchar és putchar

A legegyszerűbb beviteli mechanizmus az, amikor getchar-ral karakterenként olvasunk a szabványos bemenetről (standard inputról), általában a felhasználói terminálról. getchar() minden hívása után a következő bemeneti karaktert adja vissza. A legtöbb olyan környezetben, amely a C-t támogatja, a terminált egy állománnyal helyettesíthetjük a C konvenció segítségével: ha a program a getchar-t használja, akkor a

```
prog < infile
```

parancssor hatására a prog az infile-t fogja olvasni a terminál helyett. A bemenet átkapcsolása oly módon történik, hogy maga a prog érzéketlen a változtatásra; közelebbről, az <infile karakterlánc nem

kerül be az argv-beli parancssor-argumentumok közé. Hasonló a helyzet, ha a bemenet parancsláncon (pipe) keresztül valamelyik másik programtól érkezik; az

```
otherprog > prog
```

parancssor két programot futtat, mégpedig az otherprog-ot és a prog-ot, és úgy intézkedik, hogy a prog szabványos bemenete az otherprog szabványos kimenetéről jöjjön. A getchar az EOF értéket adja vissza, amikor az általa éppen olvasott, bármiféle bemenet végére ért. A szabványos könyvtár az EOF szimbolikus állandót -1-nek definiálja (egy #define-nal az stdio.h állományban), a vizsgálatokat ennek ellenére EOF-ra és ne -1-re végezzük, hogy ezáltal az adott értéktől függetlenek maradjunk. Ami a kimenetet illeti, a putchar (c) a c karaktert szabványos kimenetre (standard outputra) teszi, ami alapértelmezés szerint szintén a terminál. A kimenet > használatával irányítható állományba; ha prog a putchar-t használja, akkor

```
prog > outfile
```

a szabványos kimenetet a terminál helyett az outfile-ra írja. A UNIX rendszerben parancsláncot (pipe) is használhatunk:

```
prog < anotherprog
```

a prog szabványos kimenetét az anotherprog szabványos bemenetére teszi. A prog ebben az esetben sem vesz tudomást az átirányításról. A printf által létrehozott kimenő szöveg szintén a szabványos kimenetre kerül. A putchar és a printf hívásai keverhetők. Meglepően nagy azoknak a programoknak a száma, amelyek csupán egyetlen bemeneti folyamot olvasnak és csupán egyetlen kimeneti folyamot írnak. Ilyen programok esetében a be- és kivitel getchar, putchar, ill. printf függvényekkel történő megvalósítása teljesen megfelelő, és az induláshoz feltétlenül elég. Ez különösen igaz akkor, ha az egyik program kimenetének a következő program bemenetével történő összekapcsolása céljából rendelkezésre áll az állomány-átirányítás és a parancslánc-mechanizmus. Tekintsük pl. a lower programot, amely a bemenetet kisbetűssé képezi le:

```
/*A bemenet kisbetűssé alakítása*/
#include < stdio.h>
main ()
{
    int c;
    while ((c = getchar ()) != EOF)
         putchar(isupper(c)) ? tolower(c) : c);
}
```

Az isupper és tolower függvények valójában az stdio.h-ban definiált makrók. Az isupper makró ellenőrzi, hogy az argumentum nagybetű-e és nem nullát ad vissza, ha az, ill. nullát, ha nem. A tolower makró a nagybetűket kisbetűkké alakítja. Függetlenül attól , hogy az adott gépen ezek a függvények hogyan vannak megvalósítva, kívülről nézve egyformán viselkednek, így az azokat használó programoknak nem kell ismerniük a karakterkészletet. Több állomány konvertálásakor az állományok összegyűjtésére olyan programot használhatunk, mint a UNIX cat segédprogramja:

```
cat file1, file2 . . . > lower > output
```

így nem kell megtanulnunk, hogyan lehet állományokat programból elérni. (A cat-ot e fejezet későbbi részében mutatjuk be.) Mellékesen megjegyezzük, hogy a szabványos be- és kiviteli könyvtárban a getchar és putchar függvények valójában makrók lehetnek, így elkerülhető a karakterenkénti függvényhívás miatti terhelés (overhead). A 8. fejezetben fogjuk ennek tényleges megvalósítását megmutatni.

7.3. Formátumozott kimenet; printf

A kivitel céljából használt printf és a beolvasást végző scanf rutin (l. a következő szakaszt) numerikus mennyiségek karakteres ábrázolásra és karakteres mennyiségek numerikus ábrázolásra történő átalakítását, formátumozott sorok létrehozását és értelmezését teszi lehetővé. A printf függvényt az előző fejezetekben kötetlenül használtuk, íme a teljesebb és pontosabb leírás:

```
printf(control, arg1 , arg2, . . .)
```

A printf az argumentumait konvertálja, formátumozza és a szabványos kimenetre nyomtatja a control karakterlánc vezérlete alatt. A vezérlő karakterlánc kétféle típusú objektumot tartalmaz: közönséges karaktereket, amelyeket egyszerűen a kimeneti folyamra másol és konverzió-specifikációkat, amelyek mindegyike a printf soron következő argumentumának konvertálását és kinyomtatását írja elő. Minden konverzió-specifikációt a % karakter vezet be, és konverziós karakter zár le. A % és a konverziós karakter között a következők állhatnak:

- Mínuszjel, amely az ebbe a mezőbe konvertált argumentum balra igazítását írja elő.
- Számjegyekből álló karakterlánc, amely a minimális mezőszélességet határozza meg. Az átalakított szám legalább ilyen széles vagy szükség esetén szélesebb mezőbe nyomtatódik ki. Ha a konvertált argumentum kevesebb karakterből áll, mint a mezőszélessé g, akkor bal oldalon (vagy, ha a balra igazítás jelző szerepel, akkor jobb oldalon) a mező kitöltődik, hogy ezáltal az előírt mezőszélesség meglegyen. A kitöltő karakter közönséges esetben szóköz, ill. amennyiben a mezőszélességet elősnullával adtuk meg , akkor nulla (ez a zérus nem jelent oktálisan értelmezett mezőszélességet).
- Pont, amely a mezőszélességet a rá következő számjegysorozattól választja el.
- Számjegysorozat (a pontosság), amely a láncból kinyomtatásra kerülő karakterek maximális számát vagy **float** és **double** esetében a tizedes- ponttól jobbra kinyomtatandó számjegyek számát határozza meg.
- Az l (el betű) hosszmódosító, amely arra utal, hogy a szóban forgó adat **int** helyett **long**.

A konverziós karakterek és jelentésük:

- d Az argumentum decimális jelölésmódúvá alakul.
- o Az argumentum előjel nélküli, oktális számmá konvertálódik (elő nulla nélkül).
- x Az argumentum előjel nélküli, hexadecimális számmá konvertálódik (vezető 0x nélkül).
- u Az argumentum előjel nélküli decimális jelölésmódúvá alakul.
- c Az argumentumot egyetlen karakternek tekinti.
- s Az argumentum karakterlánc; a láncbeli karakterek mindaddig nyomtatódnak, amíg a nulla karakter nem kerül sorra, vagy amíg a pontossági specifikáció által előírt számú karakter kiírása

meg nem történt.

- e Az argumentumot **float**-nak vagy **double**-nak tekinti, és a [-]m.nnnnnnE[+]xx decimális jelölésmódba konvertálja, ahol az n-ek karakterláncának hosszát a pontosság adja meg. Az alapértelmezés szerinti pontosság 6.
- f Az argumentumot **float**-nak vagy **double**-nak tekinti, és a [-]mmm.nnnnn decimális jelölésmódba konvertálja, ahol az n-ek karakterláncának hosszát a pontosság adja meg. Az alapértelmezés szerinti pontosság 6. Jegyezzük meg, hogy a pontosság nem határozza meg az f formátumban kinyomtatott értékes számjegyek számát!
- g %e és %f közül a rövidebbet használja; az értéktelen nullákat elhagyja.

Ha a %-ot követő karakter nem konverziós karakter, az illető karakter nyomtatásra kerül: így a % mint %% nyomtatható ki. A legtöbb formátumkonverzió jelentése nyilvánvaló, és a korábbi fejezetekben ezeket megtárgyaltuk. Ez alól az egyik kivétel az, hogy a pontosság miként vonatkozik a karakterláncokra. Az alábbi táblázat különféle specifikációknak a "halló, világ" (12 karakter) kinyomtatására gyakorolt hatását mutatja. Minden mező köré kettőspontokat helyeztünk, hogy ezzel szemléltessük a mező kiterjedését:

Figyelmeztetés: a printf az első argumentumát használja annak eldöntésére, hogy még hány argumentum következik, és azoknak mi a típusa. A program összezavarodik, és értelmetlen választ kapunk, ha nincs elég argumentum, vagy ha azok nem a megfelelő típusúak.

7.1. Gyakorlat. Írjunk olyan programot, amely tetszőleges bemenetet képes ésszerű módon kinyomtatni! Minimális feladatként a nemgrafikus karaktereket (a helyi szokásnak megfelelően) oktálisban vagy hexadecimálisban nyomtassa ki, és hajtogassa össze a hosszú sorokat!

7.4. Formátumozott bemenet; scanf

A scanf függvény a printf bemeneti megfelelője, amely az ellenkező irányban nyújt számos, a fentiekben leírt szolgáltatást:

```
scanf(control, arg1, arg2, . . .)
```

A scanf karaktereket olvas a szabványos bemenetről, a control-ban meghatározott formátum szerint értelmezi azokat, és az eredményeket a többi argumentumban tárolja. A vezérlőargumentumot az alábbiakban írjuk le; a többi argumentum, amelyek mindegyike mutató kell, hogy legyen, azt jelzi, hogy hol kell tárolni az átalakított bemenetet. A vezérlő karakterlánc általában olyan konverziós utasításokat tartalmaz, amelyek feladata a bemeneti jelsorozat közvetlen értelmezése. A vezérlő karakterlánc tartalmazhat:

- Szóközöket, tab-okat és újsorokat (üres karaktereket), amelyeket figyelmen kívül hagy.
- Közönséges karaktereket (nem %-ot), amelyek várhatóan illeszkednek a bemeneti folyam következő nem üres karakterére.

- Konverzióspecifikációkat, amelyek a % karakterből, a * hozzárendelés-elnyomó karakterből, egy, a maximális mezőszélességet meghatározó számból, valamint egy konverziós karakterből állnak, ezek közül a két középső (* és a szám) elhagyható.

A konverzióspecifikáció a következő bemeneti mező átalakítását irányítja. Közönségesen az eredmény a megfelelő argumentum által megcímzett változóba kerül. Ha azonban a * karakter a hozzárendelés elnyomását írja elő, a vezérlés a bemeneti mezőt egyszerűen átugorja, és nem történik értékadás. A beolvasott mező definíciószerűen a nem üres karakterek lánca, tehát vagy a következő üres karakterig tart, vagy addig, amíg el nem fogy az esetleg megadott mezőszélesség. Ebből következőleg a scanf sorhatárokon keresztül is olvassa a bemenetét, mivel az újsor karakterek üres helyek. A konverziós karakter a beolvasott mező értelmezésére utal; a hozzá tartozó argumentumnak mutatónak kell lennie, amint azt a C nyelv érték szerint hívó szemantikája megkívánja. A következő konverziós karakterek megengedettek:

- d A bemeneten decimális egész számot vár; a megfelelő argumentumnak egészre kell mutatnia.
- o A bemeneten oktális egész számot vár (előnullával vagy anélkül); a megfelelő argumentumnak egész mutatónak kell lennie.
- x A bemeneten hexadecimális egész számot vár (vezető 0x-szel vagy anélkül); a megfelelő argumentumnak egész mutatónak kell lennie.
- h A bemeneten **short** egész számot vár; a megfelelő argumentum **short** egészt megcímző mutató kell, hogy legyen.
- c Egyetlen karaktert vár; a megfelelő argumentum karaktermutató kell, hogy legyen; a következő bemeneti karakter a megjelölt helyre kerül. Az üres karakterek szokásos átugrása ebben az esetben letiltódik; a következő nem üres karakter beolvasásához használjunk %ls-t.
- s Karakterláncot vár; a megfelelő argumentum karaktermutató; olyan karaktertömbre mutat, amely elég nagy ahhoz, hogy befogadja a karakterláncot és a lezáró \0-t.
- f Lebegőpontos számot vár; a megfelelő argumentum **float**-ot megcímző mutató kell, hogy legyen.
- e konverziós karakter az f szinonimája. A **float**-ok bemeneti formátuma: előjel (elhagyható), számokból álló lánc, amely tizedespontot és egy (el hagyható) kitevőmezőt tartalmazhat, amely utóbbi egy E-ből vagy e-ből és az azt követő, esetleg előjeles egész számból áll.

A d, o és x konverziós karaktereket az l (el betű) előzheti meg, amely arra utal, hogy az argumentumlistában **long**-ot és nem int-et megcímző mutató jelenik meg. Az e vagy f konverziós karaktereket ugyancsak megelőzheti az l, ebben az esetben azt jelezve, hogy az argumentumlista **double**-ra és nem **float**-ra hivatkozó mutatót tartalmaz.

Például az

```
int i;
float x;
char name[50];
scanf("%d %f %s", &i, &x, name);
```

hívás a

```
25 54.32E-1 Thompson
```

bemenet esetén az i-hez a 25 értéket rendeli hozzá, az x-hez az 5.432 értéket és a name-hez a \0-val rendesen lezárt "Thompson" karakterláncot. A három bemeneti karakterláncot tetszőleges számú szóközzel, tab-bal és újsorral lehet egymástól elválasztani. Az

```
int i;
float x;
char name[50];
scanf("%2d %f %*d %2s", &i, &x, name);
```

hívás az

```
56 789 0123 45a78
```

bemenettel 56-ot rendel i-hez, 789.0-t az x-hez, átugorja a 0123-at és a 45 karakterláncot a name-be teszi. Bármelyik bemeneti rutin következő hívása az a betűnél történő kereséssel fog indulni. E két példában a name mutató, és nem előzheti meg az & szimbólum.

Másik példaként a 4. fejezetben ismertetett elemi kalkulátorprogramot most úgy írjuk át, hogy a scanf végezze a bemeneti konverziót:

```
/* Elemi kalkulátorprogram*/
#include <stdio.h>

main()
{
    double sum, v;
    sum = 0;
    while (scanf("%lf", &v) != EOF)
         printf("\t%.2f\n", sum += v);
}
```

A scanf akkor fejezi be működését, amikor kimerítette a vezérlő karakterláncát, vagy amikor valamelyik bemenet nem illeszkedik a vezérlési specifikációra. A scanf visszatérési értéke a sikeresen illesztett és hozzárendelt bemeneti tételek száma. Ebből meghatározható, hogy a scanf hány bemeneti tételt talált. Állomány vége esetén a visszaadott érték EOF; ügyeljünk arra, hogy ez nullától eltérő érték, amely azt jelenti, hogy a következő bemeneti karakter nem illeszkedik a vezérlő karakterlánc első specifikációjára! A scanf következő hívásakor a keresés közvetlenül az utoljára visszaadott karakter után folytatódik. Még egy utolsó figyelmeztetés: a scanf argumentumainak mutatóknak kell lenniük! A leggyakoribb hiba, amikor valaki azt írja, hogy

```
scanf("%d", n);
```

ahelyett, hogy

```
scanf("%d", &n);
```

-et írna.

7.5. Formátumkonverzió a táron belül

A scanf és printf függvényekkel rokon az sscanf és sprintf, amelyek ugyanezeket a konverziókat végzik, de állomány helyett karakterláncon dolgoznak. Az általános formátum:

```
sprintf(string, control, arg1, arg2, ...)
sscanf(string, control, arg1, arg2, ...)
```

Az előzőekhez hasonlóan az sprintf a control szerint formátumozza az arg1, arg2 stb.-beli argumentumokat, az eredményt azonban a szabványos kimenet helyett a string-be teszi. A string-nek természetesen elég nagynak kell lennie ahhoz, hogy befogadja az eredményt. Ha pl. string karaktertömb és n egész, akkor

```
sprintf(name, "temp %d", n);
```

a name-ben létrehoz egy temp NNN alakú karakterláncot, ahol NNN az n értéke. Az sscanf az ellentétes irányú konverziókat hozza létre – a control-ban megadott formátum szerint végighalad a karakterláncon, és a kapott eredményeket az arg1, arg2 stb.-ben helyezi el. Ezen argumentumoknak mutatóknak kell lenniük. A

```
sscanf(name, "temp%d", &n);
```

hívás az n-et annak a számjegyekből álló karakterláncnak az értékére állítja be, amely a name-ban a temp-et követi.

7.2. Gyakorlat. Írjuk meg újra a 4. fejezetben látott kalkulátorprogramot úgy, hogy a bemenetre és a számkonverzióra a scanf és/vagy sscanf függvényeket alkalmazzuk!

7.6. Állomány-hozzáférés

Az idáig megírt programok mindegyike a szabványos bemenetet olvasta és a szabványos kimenetre írt, amelyekről mindeddig feltételeztük, hogy valamilyen varázslatos módon a helyi operációs rendszer előre definiálta őket a számunkra. A be- és kivitellel való ismerkedésünk következő lépéseként olyan programot írunk, amellyel programhoz nem rendelt állományhoz férhetünk hozzá. Az ilyen műveletek szükségességét világosan bizonyító program a cat, amely megnevezett állományok halmazát gyűjti ki (konkatenálja) a szabványos kimenetre. A cat feladata állományoknak a terminálra történő kinyomtatása, valamint általános célú bemeneti információgyűjtés azon programok számára, amelyek maguk nem képesek állományokhoz név szerint hozzáférni.

Pl. a

```
cat x.c y.c
```

parancs az x.c és y.c állományok tartalmát a szabványos kimenetre nyomtatja. Kérdés, hogyan érjük el, hogy a megnevezett állományok beolvasásra kerüljenek – azaz, neveket azokhoz az utasításokhoz, amelyek ténylegesen elolvassák az adatokat. A szabályok egyszerűek. Mielőtt egy állományt olvasni vagy írni lehetne, az fopen szabványos könyvtári függvénnyel meg kell nyitni. Az fopen vesz egy külső nevet (mint x.c vagy y.c), bizonyos nyilvántartást végez, és párbeszédet folytat az operációs rendszerrel (aminek részleteivel nem kell törődnünk), és olyan belső nevet ad vissza, amelyet az állomány ezután következő olvasásai, ill. írásai során használnunk kell.

E belső név valójában mutató, amelyet állománymutatónak nevezünk, és amely egy, az állományról különböző információkat tartalmazó struktúrára mutat. Itt található pl. a puffer címe, a pillanatnyi pufferbeli karakterpozíció, annak jelzése, hogy az állomány éppen olvasás vagy írás alatt áll stb. A felhasználóknak a részleteket nem kell ismerniük, mivel az stdio.h-tól nyert szabványos be- és kiviteli definíciók egyik része a FILE-nak nevezett struktúradefiníció. Az állománymutató számára szükséges egyetlen deklarációra nézve példa a

```
FILE *fopen(), *fp;
```

Eszerint fp FILE-t megcímző mutató, és fopen szintén ilyen mutatóval tér vissza. Figyeljük meg, hogy FILE, csakúgy, mint int, típusnév, nem pedig struktúracímke; typedef-ként valósították meg. (Annak részleteit, hogy mindez miként működik a UNIX operációs rendszerben, a 8. fejezetben ismertetjük.) Az fopen tényleges hívása a programon belül így fest:

```
fp = fopen(name, mode);
```

Az fopen első argumentuma az állomány neve, amely egy karakterlánc. A második argumentum, amely szintén karakterlánc, a mód, amely azt jelzi, hogy a felhasználó hogyan akarja használni az állományt. A megengedett módok az

- olvasás (r: read),
- az írás (w: write) és a
- hozzáfüggesztés (a: append).

Ha írásra vagy hozzáfüggesztésre nem létező állományt nyitunk meg, akkor az illető állomány (ha lehet) létrejön. Létező állomány írásra történő megnyitásának hatására annak korábbi tartalma elvész. Hibát jelent, ha nemlétező állományt olvasni akarunk. Más hibaokok is előfordulhatnak (pl. ha olyan

állományt próbálunk meg olvasni, amelyre nincs engedélyünk). Bármilyen hiba esetén az fopen a NULL mutatóértékkel tér vissza (amelynek definíciója a kényelem kedvéért szintén stdio.h-ban van). A következőkben azt kell tudnunk, hogyan olvashatjuk a már megnyitott állományokat. Több lehetőség van, amelyek közül a getc és putc csak a legegyszerűbb. getc az állomány soron következő karakterével tér vissza, állománymutatóval kell megadnunk, hogy melyik állományról van szó. Így

```
c = getc(fp)
```

az fp által hivatkozott állományból a következő karaktert c-be helyezi, ill. EOF kerül c-be, ha elértük az állomány végét. A putc a getc inverze:

```
putc(c, fp)
```

a c karaktert az fp állományba helyezi és c-t adja vissza. A getchar és putchar függvényekhez hasonlóan a getc és putc is lehet függvény helyett makró. Három állomány minden program indításakor automatikusan megnyílik, és a rendszer állománymutatókat is rendelkezésre bocsát számukra. Ezek az állományok: a szabványos bemenet, a szabványos kimenet és a szabványos hibakimenet; az ezeknek megfelelő mutatók neve: stdin, stdout és stderr. Közönséges esetben ezek mindegyike a terminálhoz van rendelve, azonban az stdin és stderr mutatókat a 7.2. szakaszban leírt módon állományokba vagy parancsláncokba lehet átirányítani. A getchar és a putchar az alábbi módon definiálható a getc, a putc, az stdin és az stdout segítségével:

```
#define getchar() getc(stdin)
#define putchar(c) putc(c, stdout)
```

Állományok formátumozott beolvasására vagy kiíratására az fscanf és fprintf függvényeket használhatjuk. Ezek azonosak a scanf és printf függvényekkel, eltekintve attól, hogy az első argumentum állománymutató, amely az olvasandó vagy írandó állományt határozza meg; a vezérlő karakterlánc a második argumentum. E bevezetés után már abban a helyzetben vagyunk, hogy megírhatjuk az állományokat konkatenáló cat programot. Az alapfelépítés azonos azzal, ami már sok programban kényelmesnek bizonyult: ha vannak parancssor-argumentumok, akkor azok feldolgozása sorrendben történik.

Ha nincsenek argumentumok, akkor a szabványos bemenetet dolgozzuk fel. Ílymódon a program akár önállóan, akár valamely nagyobb feldolgozás részeként használható.

```
#include <stdio.h>
/* cat: állományok konkatenálása*/

/* Állomány másolása a szabványos kimenetre*/
filecopy(FILE *fp)
{
    int c;
    while ((c = getc(fp)) != EOF)
         putc(c, stdout);
}

main(int argc, char *argv[])
{
    FILE *fp, *fopen();
    if (argc == 1) /* Nincs arg., a szabványos bemenetet másolja*/
         filecopy(stdin);
    else
         while (--argc > 0)
              if ((fp = fopen(*++argv, "r")) == NULL) {
                   printf("cat: nem nyitható meg %s\n", *argv);
                   break;
              } else {
                        filecopy(fp);
                        fclose(fp);
              }
}
```

Az stdin, ill. stdout állománymutatók a be- és kiviteli könyvtárban szabványos bemenetként, ill. szabványos kimenetként előredefiniáltak; mindenütt használhatók, ahol FILE típusú objektumokat használni lehet. Ezek azonban állandók és nem változók, tehát ne próbáljunk semmit sem hozzájuk rendelni! Az fclose függvény az fopen inverze: megszakítja az állománymutató és a külső név között az fopen által létrehozott kapcsolatot, és így az állománymutató egy másik állomány számára szabadul fel. Mivel a legtöbb operációs rendszerben az egyidejűleg megnyitott állományok száma korlátozott, célszerű azokat felszabadítani, ha már nincs rájuk szükség, amint ezt a cat-ban is tettük. Az fclose kimeneti állományra való alkalmazásának másik oka is van: üríti azt a puffert, amelyben a putc a kimenetet gyűjti. (A program normális befejeződésekor az fclose automatikusan meghívódik minden megnyitott állományra.)

7.7. Hibakezelés; stderr és exit

A hibáknak az a fajta kezelése, amit a cat-ban használtunk, nem ideális. A baj az, hogy ha az állományok egyike valamely oknál fogva hozzáférhetetlen, a hibajelzés a konkatenált kimenet végére nyomtatódik. Ez elfogadható, ha a kimenet a terminálra irányú l, azonban rossz, ha egy állományba vagy parancsláncon keresztül egy másik programba megy. A jobb hibakezelés érdekében az stdin és

stdout állománnyal azonos módon a programhoz egy második kimeneti állomány, az stderr is hozzá van rendelve.

Ha egyáltalán lehetséges, az stderr-re írt kimenet még akkor is megjelenik a felhasználói terminálon, amikor a szabványos kimenetet átirányították. Módosítsuk a cat programot úgy, hogy a hibaüzeneteket a szabványos hibaállományra írja!

```
#include <stdio.h>

/* cat: állományok konkatenálása*/
main(int argc, char *argv[])
{
    FILE *fp, *fopen();
    if (argc == 1) /* Nincs arg., a szabványos bemenetet másolja*/
         filecopy(stdin);
    else
         while (--argc > 0) {
              if ((fp = fopen(* ++argv, "r")) == NULL) {
                        fprintf(stderr,
                                  "cat: nem nyitható meg %s\n", *argv);
                        exit(1);
              } else {
                        filecopy(fp);
                        fclose(fp);
                   }
          }
    exit(0);
}
```

A program kétféleképpen jelzi a hibákat. Az fprintf által előállított diagnosztikai kimenet az stderr-re megy, tehát a felhasználó terminájára kerül, és nem tűnik el egy parancsláncon keresztül vagy valamelyik kimeneti állományban. A program az exit szabványos könyvtári függvényt is használja. Az exit meghívása a program befejeződését eredményezi. Az exit argumentum bármilyen, az exit-et hívó folyamat rendelkezésére áll, így a program sikeres vagy sikertelen lefutását egy másik program oly módon ellenőrizheti, hogy ezt a programot mint részfolyamatot használja. Megállapodás szerint a 0 visszatérési érték azt jelzi, hogy minden rendben ment, míg a különféle nem nulla értékek a normálistól eltérő állapotokat jelzik. Az exit minden megnyitott kimeneti állományra meghívja az fclose-t az összes pufferelt kimenet kiürítése érdekében, majd meghívja az exit nevű rutint, amelynek hatására a programfutás mindenféle pufferürítés nélkül azonnal véget ér. Az exit szükség esetén természetesen közvetlenül is hívható.

7.8. Szövegsorok beolvasása és kivitele

A szabványos könyvtárban rendelkezésre áll az fgets rutin, amely meglehetősen hasonlít a könyvben végig használt getline függvényhez. Az

```
fgets(line, MAXLINE, fp)
```

hívás az fp állományból a line karaktertömbbe beolvassa a következő bemeneti sort (az újsort is beleértve); legfeljebb MAXLINE-1 sort fog olvasni. A kapott tömb \0-val zárul. Normál esetben az fgets a line-t adja vissza, állomány végén pedig NULL-t. (A getline függvényünk a sorhosszat, ill. állomány vége esetén a nullát adja vissza.) Kivitelkor az fputs függvény karaktersorozatot (amely nem kell, hogy újsort tartalmazzon) ír az állományra:

```
fputs(line, fp)
```

Annak érzékeltetésére, hogy az olyan függvények körül, mint fgets és fputs nincs semmi varázslatos, a szabványos be- és kiviteli könyvtárból közvetlenül ide másoltuk e függvények programkódját:

```
#include < stdio.h>

/* Legfeljebb n karakter olvasása iop-ról*/
char * fgets(char *s, int n,  register FILE *iop)
{
    register int c;
    register char *cs;
    cs=s;
    while (--n > 0 && (c = getc(iop)) != EOF)
         if ((*cs++ = c) == '\n')
              break;

    *cs = '\0';
    return((c == EOF && cs == s) ? NULL : s);
}

/*Az s karakterláncot az iop állományra írja*/
fputs(register char *s, register FILE *iop)
{
    register int c;
    while (c = *s++)
         putc(c, iop);
}
```

7.3. Gyakorlat. Írjunk olyan programot, amelyik összehasonlít két állományt, és kiírja az első olyan sort és karakterpozíciót, ahol az állományok eltérnek egymástól!

7.4. Gyakorlat. Módosítsuk az 5. fejezet mintakereső programját oly módon, hogy a bemenetét argumentumokként megnevezett állományok halmazából vegye, vagy ha ilyenek nincsenek, akkor a szabványos bemenetről! Ki kell-e íratni az állomány nevét, ha a program egymásra illeszkedő sorokat talál?

7.5. Gyakorlat. Írjunk olyan programot, amely több állományt nyomtat ki, minden újabb állományt új oldalon, cím kiírásával kezd, és az oldalakat állományonként folyamatosan számozza!

7.9. Néhány további függvény

A szabványos könyvtár számos függvényt bocsát rendelkezésünkre, amelyek közül néhány különösen hasznos. Már említettük az strlen, shcpy, strcat és strcmp karakterlánc-kezelő függvényeket. Íme néhány további függvény.

Karakterosztály-vizsgálat és -átalakítás Több makro végez karaktervizsgálatot és átalakítást:

- isalpha(c) nemnulla, ha c alfabetikus, 0, ha nem.
- isupper(c) nemnulla, ha c nagybetű, 0, ha nem.

- islower(c) nemnulla, ha c kisbetű, 0, ha nem.
- isdigit(c) nemnulla, ha c számjegy, 0, ha nem.
- isspace(c) nemnulla, ha c szóköz, tab vagy újsor, 0, ha nem.
- toupper(c) c átalakítása nagybetűssé.
- tolower(c) c átalakítása kisbetűssé.

Az ungetch függvény A szabványos könyvtárban megtaláljuk a 4. fejezetben általunk megírt ungetch függvény egy meglehetősen szűkített változatát; ennek neve ungetc.

Az

```
ungetc(c, fp);
```

a c karaktert az fp állományba helyezi vissza. Állományonként csak egy karakternyi visszatolás megengedett. Az ungetc minden olyan bemeneti függvénnyel és makróval együtt használható, mint a scanf, getc vagy a getchar.

7.9.1. Rendszerhívás

A system(s) függvény az s karakterláncban tartalmazott parancsot hajtja végre, majd visszatér az adott program végrehajtásához. Az s tartalma erősen függ a helyi operációs rendszertől. Triviális példaként a UNIX-ban a

```
 system("date");
```

sor hatására lefut a date nevű program; amely kinyomtatja a dátumot és a napon belüli időpontot.

7.9.2. Tárkezelés

A calloc függvény igen hasonlít a korábbi fejezetekben használt alloc függvényre.

```
calloc(n, sizeof(objektum))
```

egy mutatót szolgáltat, amely olyan helyre mutat, ahol elegendő hely van n darab megadott méretű objektum számára, ill. a NULL értéket adja vissza, ha a kérés nem teljesíthető. A tárterület kezdeti nagysága nulla. A mutató a szóban forgó objektum típusának megfelelő helyre mutat, azonban

típusmódosító szerkezettel a megfelelő típusúvá kell alakítani, pl. :

```
char *calloc();
int *ip;
ip = (int *)calloc(n, sizeof(int));
```

A cfree(p) felszabadítja a p által megcímzett helyet, ahol p-t eredetileg a calloc valamelyik hívásával nyertük. A helyfelszabadítás sorrendjére nincs megkötés, azonban végzetes hiba, ha olyasvalamit szabadítunk fel, amit nem a calloc hívásával nyertünk. A 8. fejezetben bemutatjuk a calloc-hoz hasonló tárterület-foglaló függvény megvalósítását, amelyben a lefoglalt blokkok tetszőleges sorrendben szabadíthatók fel.

8. Csatlakozás a UNIX operációs rendszerhez

E fejezet anyaga a C programok és a UNIX operációs rendszer közötti kapcsolattal foglalkozik. Mivel a legtöbb C programozó UNIX rendszer alatt dolgozik, ezek az ismeretek az olvasók többsége számára hasznosak lesznek. Sőt, még ha az olvasó a C nyelvet más gépen is használja, e példák tanulmányozása révén mélyebb betekintést nyerhet magába a C programozásba is. A fejezet három fő témakörre oszlik: bevitel/kivitel, állománykezelés és tárterület-foglalás. Az első két rész feltételezi a UNIX külső megjelenésének legalább némi ismeretét. A 7. fejezet olyan rendszer-határfelülettel foglalkozott, amely számos operációs rendszerben egyforma. Bármelyik konkrét rendszerben a szabványos könyvtár rutinjait a befogadó rendszerben rendelkezésre álló be- és kiviteli szolgáltatások figyelembevételével kell megírni. A következő néhány szakaszban a UNIX operációs rendszer be- és kiviteli rendszerének alapvető belépési pontjait ismertetjük, és azt szemléltetjük, miként lehet ezek segítségével a szabványos könyvtár egyes részeit megvalósítani.

8.1. Állományleírók

A UNIX operációs rendszerben az összes be- és kivitel állományok írásával és olvasásával valósul meg, mivel az összes periféria, még a felhasználó terminálja is egy-egy állományként jelenik meg. Ez azt jelenti, hogy egyetlen homogén csatolóprogram kezeli a program és a perifériák között az összes kapcsolatot. A legáltalánosabb esetben egy állomány írása vagy olvasása előtt értesítenünk kell a rendszert erről a szándékunkról. Ezt a folyamatot az állomány megnyitásának nevezzük. Ha írni akarunk egy állományba, akkor szükség lehet az állomány létrehozására is. A rendszer ellenőrzi, hogy minderre van-e jogunk (Létezik-e az állomány? Van-e hozzáférési engedélyünk?), és ha minden rendben van, akkor a programhoz egy állományleírónak nevezett kis egész számmal tér vissza. Minden esetben, amikor az állományon be- vagy kiviteli műveletet akarunk végezni, az állomány azonosítása céljából annak neve helyett az állományleírót használjuk (Ez nagyjából hasonlít a READ(5, ...) és WRITE(6, ...) használatára a FORTRAN-ban.) A megnyitott állományra vonatkozó összes információt a rendszer kezeli, a felhasználói program csupán az állományleírón keresztül hivatkozik az állományra. Mivel a felhasználói terminálon keresztül megvalósított be- és kivitel egészen mindennapos tevékenység, a UNIX tervezői igyekeztek ezt minél kényelmesebbé tenni. Amikor a parancsértelmező

(a shell) valamelyik programot futtatja, három állományt nyit meg a 0, 1 és 2 állományleírókkal, amelyeknek neve szabványos bemenet, szabványos kimenet és szabványos hibakimenet. Alaphelyzetben mindhárom a terminálhoz van rendelve, ha tehát egy program a 0 állományleírót olvassa, ill. az 1 és 2 állományleíróra ír, a terminálon keresztüli be/kivitel közben nem kell törődnie az állományok megnyitásával. A program felhasználója az állományokkal folytatott be- és kivitelt átirányíthatja a < és > szimbólumokkal:

```
prog < infile > outfile
```

Ebben az esetben a shell a 0 és 1 állományleíróra vonatkozó alap-hozzárendeléseket a terminálról a megnevezett állományokra irányítja. Normál esetben a 2 állományleíró továbbra is a terminálhoz lesz rendelve, így a hibaüzenetek oda íródhatnak ki. Hasonlóképpen jellemezhetjük azt az esetet, amikor a bemenet vagy a kimenet valamilyen parancslánchoz kapcsolódik. Megemlítendő, hogy az állomány-hozzárendeléseket mindig a shell változtatja meg, nem pedig a program. Maga a program mindaddig nem tudja, hogy a bemenete honnan jön és a kimenete hová megy, amíg a 0 állományt használja bevitelre és az 1 és 2 állományt kivitelre.

8.2. Alacsony szintű bevitel és kivitel; read és write

A be- és kivitel UNIX-beli legalacsonyabb szintje nem nyújt sem pufferelést, sem egyéb szolgáltatást: ez valójában az operációs rendszer közvetlen belépési pontja. Az összes bevitelt és kivitelt két függvény végzi, amelyeknek neve: read és write. Az első argumentum mindkét esetben az állományleíró. A második argumentum a programunkban elhelyezett puffer, ahonnan az adatok érkeznek, ill. ahová beíródnak. A harmadik argumentum az átvitelre kerülő byte-ok száma. A hívások:

```
n_read = read(fd, buf, n);
n_written = write(fd, buf, n);
```

Mindegyik hívás byte-darabszámot ad vissza, amely a ténylegesen átvitt byte-ok száma. Olvasáskor a visszaadott byte-szám az előírtnál (n) kisebb lehet. A nulla byte-visszatérési érték az állomány végét jelenti, míg a -1 valamilyen hibára utal. Írás eset én a visszaadott érték a ténylegesen felírt byte-ok száma; általában hibát jelent, ha ez nem egyezik meg a felírandó byte-ok előre megadott számával. Az olvasandó vagy írandó byte-ok száma teljesen tetszőleges. A két legközönségesebb érték az 1 , amely egyidőben egy karakter átvitelét jelenti (pufferelétlen átvitel) és az 512, amely a legtöbb periféria esetében a fizikai blokkméretnek felel meg. Az utóbbi méret a leghatékonyabb, de még a karakterenkénti be- és kivitel sem rendkívül költséges. Ezen ismeretek birtokában megírhatunk egy egyszerű programot, amely a bemenetét a kimenetére másolja - ez az 1. fejezetben megírt állománymásoló programnak felel meg.

UNIX alatt ez a program bármit bármire másol, mivel a bemenet és a kimenet bármilyen állományra

vagy perifériára átirányítható.

```
#define BUFSIZE 512 /* Legjobb méret a PDP-11 UNIX-ra*/

/*A bemenet másolása a kimenetre*/
main()
{
    char buf[BUFSIZE];
    int n;
    while ((n = read(0, buf, BUFSIZE)) > 0)
         write(1, buf, n);
}
```

Ha az állományméret nem a BUFSIZE többszöröse, akkor valamelyik read egy ennél kisebb számot ad át write-nak a felírandó byte-ok számaként; a read ezután következő hívása nullát fog visszaadni. Tanulságos látnunk, hogyan használható read és write, olyan magasabb szintű rutinok létrehozására, mint a getchar, putchar stb. Íme pl. a getchar egy változata, amely pufferelés nélküli olvasást végez:

```
#define CMASK 0377 /*A char-ok 0-vá tételére*/

/*Pufferelетlen egykarakteres bevitel*/
getchar()
{
    char c;
    return((read(0, &c, 1) > 0) ?c & CMASK : EOF);
}
```

A c-t char-nak kell deklarálni; mivel a read karaktermutatót fogad. A visszaadott karaktert 0377-tel maszkolni kell, hogy biztosan pozitív legyen - ellenkező esetben az előjel-kiterjesztés következtében negatívvá válhat. (A 0377 állandó a PDP-11 számára megfelelő, de nem feltétlenül jó más gépek esetén.) A getchar második változata nagy egységekben végzi az olvasást, és egyenként adja ki a karaktereket:

```
#define CMASK 0377 /*A char-ok 0-vá tételére*/
#define BUFSIZE 512

/*Pufferelt változat*/
getchar()
{
    static char buf[BUFSIZE];
    static char *bufp = buf;
    static int n = 0;
    if (n == 0) {                  /*A puffer üres*/
         n = read(0, buf, BUFSIZE);
         bufp = buf;
    }
    return ((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

8.3. Open, creat, close, unlink

Az alapértelmezés szerinti szabványos bemeneti, kimeneti és hibakimeneti állományon kívül az összes állományt explicit módon meg kell nyitnunk, ha azokat írni vagy olvasni akarjuk. Ebből a célból két rendszerbelépési pont áll rendelkezésre : az open és a creat (vigyázat, nem create!). Az open lényegében ugyanolyan, mint a 7. fejezetben tárgyalt fopen, eltekintve attól, hogy nem állománymutatót ad vissza, hanem állományleírót, ami egyszerűen egy int.

```
int fd;
fd = open(name, rwmode);
```

Az fopen-hez hasonlóan a name argumentum a külső állománynévnek megfelelő karakterlánc. Az elérés módja azonban eltérő: az rwmode értéke olvasáskor 0, íráskor 1, és egyidejű írási-olvasási hozzáférés esetén 2. Hiba előfordulásakor az open -1-et ad vissza , egyébként a visszatérési érték az érvényes állományleíró. Hibához vezet, ha nem létező állományt próbálunk megnyitni. A creat belépési pont új állományok létrehozására vagy régiek felülírására szolgál:

```
fd = creat(name, pmode);
```

állományleírót ad vissza, ha létre tudta hozni a name nevű állományt, és -1-et, ha nem. Ha az állomány már létezik, a creat nulla hosszúságúra vágja le, nem jelent tehát hibát már létező állomány creat-tel

történő létrehozása. Ha az állomány vadonatúj, a creat azt a pmode argumentumban megadott védelmi móddal hozza létre. A UNIX rendszerben minden állományhoz kilenc bitből álló védelmi információ társul. Ezek a bitek az állomány tulajdonosára, a tulajdonos csoportjára, valamint a másokra vonatkozó olvasási, írási és végrehajtási engedélyeket szabályozzák. Az engedélyeket így legkényelmesebben egy háromjegyű oktális számmal adhatjuk meg. Pl. 0755 olvasási-írási-végrehajtási engedélyt ad a tulajdonosnak, és olvasási-végrehajtás i engedélyt a csoport tagjainak és mindenki másnak. Szemléltetés céljából közöljük a UNIX cp nevű segédprogramjának egyszerűsített változatát, amely egy állományt egy másikba másol. (A fő egyszerűsítés az, hogy az itt közölt változat csak egyetlen állományt másol és nem teszi lehetővé, hogy a második argumentum katalógus (directory) legyen.)

```
/*cp: f1 másolása f2-be*/

#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW a tulajdonosnak, R a csoportnak és
másoknak*/

/*A hibaüzenetet kiírja és leáll*/
error(char *s1, char *s2)
{
    printf(s1, s2);
    printf("\n");
    exit(1);
}

main(int argc, char *argv[])
{
    int f1, f2, n;
    char buf[BUFSIZE];
    if (argc != 3)
         error("Használat: cp honnan hová", NULL);

    if ((f1 = open(argv[1], 0)) == -1)
         error("cp: nem nyitható meg %s", argv[1]);

    if ((f2 = creat(argv[2], PMODE)) == -1)
         error("cp: nem hozható létre %s", argv[2]);

     while ((n = read(f1, buf, BUFSIZE)) > 0)
         if (write(f2, buf, n) != n)
              error("cp: íráshiba", NULL);

    exit(0);
}
```

A programok által egyidejűleg nyitva tartható állományok száma korlátozott (tipikusan 15-25). Ennek megfelelően minden olyan programot, amelynek sok állományt kell feldolgoznia, úgy kell elkészíteni, hogy képes legyen az állományleírók újbóli használatár a. A close rutin megszakítja az állományleíró és a megnyitott állomány közötti kapcsolatot és felszabadítja az állományleírót, így azt a későbbiekben más állomány használhatja. A program exit hatására történő befejezése és a főprogramból való visszatérés az összes megnyitott állományt lezárja.

Az unlink(filename) függvény a filename nevű állományt törli az állományrendszerből.

8.1. Gyakorlat. Írjuk át a 7. fejezetben látott cat programot úgy, hogy a read, write, open és close

rutinokat használjuk azok szabványos könyvtárbeli megfelelői helyett! Végezzünk kísérleteket a két változat egymáshoz viszonyított sebességének meghatározására!

8.4. Véletlen hozzáférés; seek és lseek

Állományok be- és kivitele általában soros: minden read és write az állománynak azon a pozícióján történik, amely közvetlenül a megelőző be- vagy kivitel állománybeli pozícióját követi. Szükség esetén azonban az állomány tetszőleges sorrendben olvasható vagy írható. Az lseek rendszerhívás lehetővé teszi, hogy tényleges olvasás vagy írás nélkül mozoghassunk az állományban:

```
lseek(fd, offset, origin);
```

hatására az fd leírójú állományban az aktuális pozíció az offset pozícióra mozdul, amelyet az origin által meghatározott helyhez képest relatíven értelmezünk. Az ezt követő olvasás vagy írás ezen az új pozíción fog kezdődni. Az offset **long** típusú: az fd és az origin **int** típusúak. Az origin 0, 1 vagy 2 lehet, jelezve, hogy az offset-et az állomány elejétől, a pillanatnyi pozíciótól, vagy az állomány végétől kell számítani. Ha pl. az állományhoz valamit hozzá akarunk függeszteni, írás előtt keressük meg az állomány végét:

```
lseek(fd, 0L, 2);
```

Ha vissza akarunk térni az állomány elejére ("visszatekercselés"):

```
 lseek(fd, 0L, 0);
```

Figyeljük meg a 0L argumentumot, ezt (**long**)0-nak is írhatnánk. Az lseek használatával lehetőségünk van arra, hogy az állományokat – lassabb hozzáférés árán - nagy tömbökhöz hasonlóan kezeljük. Az alábbi egyszerű függvény pl. az állomány tetszőleges pontjáról tetszőleges számú byte-ot olvas be:

```
/*n byte olvasása a pos pozícióról*/
get(int fd, int n, long pos,  char *buf;)
{
    lseek(fd, pos, 0);            /*Elmegy pos-ra*/
    return(read(fd, buf, n));
}
```

A UNIX rendszer 7-est megelőző változataiban a be- és kiviteli rendszer alapvető belépési pontjának neve: seek. A seek és az lseek azonosak, attól eltekintve, hogy az előbbinek az offset argumentuma nem **long**, hanem int. Ennek megfelelően, mivel a PDP- 11 int-ek 16 bitesek, a seek-nek megadható offset felső korlátja 65535; ezért a 3, 4, 5 origin értékek hatására a seek a megadott offset értéket 512-vel (a fizikai blokkban található byte-ok számával) megszorozza, majd az origin-t úgy értelmezi, mintha az adott sorrendben 0, 1 vagy 2 lenne. Ílymódon, ha egy nagy állomány tetszőleges pontjára akarunk lépni, akkor két seek-re van szükségünk: az elsővel a blokkot választjuk ki, a másodikkal pedig, amelyben az origin értéke 1 , a blokkon belül a kívánt byte- ra mozdulunk.

8.2. Gyakorlat. Világos, hogy az lseek a seek felhasználásával megírható és viszont. Írjuk meg mindkettőt a másik felhasználásával!

8.5. Példa; az fopen és a getc megvalósítása.

Próbáljuk meg egységbe foglalva szemléltetni a mondottakat az fopen és getc szabványos könyvtári rutinok egyik megvalósításának bemutatásával. Emlékezzünk arra, hogy a szabványos könyvtár állományait nem állományleírók, hanem állománymutatók jellemzik. Ez utóbbiak olyan struktúrára mutatnak, amely az állományra vonatkozó különböző információkat tartalmaz: egy puffert megcímző mutatót, ami lehetővé teszi az információ nagy darabokban történő beolvasását; a pufferben maradt karakterek darabszámát; a következő pufferbeli karakterpozíciót megcímző mutatót; néhány jelzőt (flag-et), amelyek pl. az olvasás/írás módot írják le; és végül az állományleírót.

Az állományt leíró adatstruktúra az stdio.h állományban található, amelyet (#include-dal) minden olyan forrásállományba be kell iktatni, amely a szabványos könyvtár valamelyik rutinját használja. A könyvtárbeli függvények ugyancsak tartalmazzák. Az stdio .h-ból vett alábbi kivonatban azok a nevek, amelyeket csak a könyvtárbeli függvények használhatnak, aláhúzással kezdődnek, így kisebb annak a valószínűsége, hogy valamelyik felhasználói programbeli névvel összeütközésbe kerüljenek.

```
#define BUFSIZE 512
#define NFILE 20            /*Kezelhető állományok száma*/

typedef struct iobuf {
    char *ptr;              /*Következő karakterpozíció*/
    int cnt;                /*Megmaradt karakterek száma*/
    char *_base;            /*A puffer címe*/
    int flag;               /*Az állományelérés módja*/
    int fd;                 /*Állományleíró*/
}FILE;

extern FILE iob[NFILE];

#define stdin (&iob[0])
#define stdout (&iob[1])
#define stderr (&iob[2])
#define _READ 01        /*Állománymegnyitás olvasásra*/
#define _WRITE 02       /*Állománymegnyitás írásra*/
#define UNBUF 04        /*Az állomány pufferelетlen*/
#define BIGBUF 010      /*Nagy pufferlefoglalás*/
#define EOF 020         /*EOF fordult elő ebben az állományban*/
#define ERR 040         /*Hiba fordult elő ebben az állományban*/
#define NULL 0
#define EOF (-1)
#define getc(p) (--(p) - ) cnt >= 0 \
             ? *(p) > ptr++ & 0377 : fillbuf(p))
#define getchar() getc(stdin)
#define putc(x, p) (--(p) == cnt >= 0 \
             ? *(p) == ptr++ = (x) : flushbuf((x), p))
#define putchar(x) putc(x, stdout)
```

A getc makró normál esetben egyszerűen dekrementálja a darabszámot, előrelépteti a mutatót, és visszaadja a karaktert. (A hosszú #define-okat fordított \ törtvonallal lehet folytatni.) Ha a darabszám negatívvá válik, a getc meghívja a fillbuf függvényt, amivel újratölti a puffert, újrainicializálja a struktúra tartalmát, és egy karaktert ad vissza. A függvények rendelkezhetnek gépfüggetlen csatlakozófelülettel, akkor is, ha maguk gépfüggő konstrukciókat tartalmaznak: a getc 0377-tel maszkolja a karaktert, amely felülbírálja a PDP-11 által végrehajtott előjel-kiterjesztést, és biztosítja, hogy minden karakter pozitív legyen.

Bár nem kívánunk részletekbe menni, mégis beiktattuk a putc definícióját annak bemutatására, hogy az lényegében ugyanúgy működik, mint a getc, azaz amikor a puffere megtelt, meghívja a flushbuf függvényt. Ezek után megírhatjuk az fopen függvényt. Az fopen legnagyobb része azzal foglalkozik, hogy megnyitja az állományt, a megfelelő helyre pozicionálja, és úgy állítja be a jelzőbiteket, hogy azok a helyes állapotot mutassák. Az fopen pufferterületet nem foglal le : ezt az állomány első

olvasásakor a fillbuf végzi.

```
 #include <stdio.h>
 #define PMODE 0644 /* RWW a tulajdonosnak; R másoknak*/

/*Megnyítja az állományt, az állománymutatót adja vissza*/
FILE *fopen(register char *name, char *mode)
{
    register int fd;
    register FILE *fp;
    if (*mode != 'r' && *mode != 'w' && *mode != 'a') {
         fprintf(stderr,
              "tiltott mód %s a %s megnyitásakor \n", mode, name);
         exit( 1 );
    }

    for (fp = &iob; fp < iob + NFILE; fp++)
         if ((fp->~flag & (_READ & _WRITE)) == 0)
              break;              /*Szabad területet talált*/

    if (fp >= iob + &NFILE)       /*Nincs szabad hely*/
         return(NULL);

    if (*mode == 'w')             /*Állományhozzáférés*/
         fd = creat(name, PMODE);
    else
         if (*mode == 'a') {
              if ((fd = open(name, 1)) == -1)
                        fd = creat(name, PMODE);

              lseek(fd, 0L, 2);
         } else
              fd = open(name, 0);

    if (fd == -1)  /* Nem tudta a nevet elérni*/
         return(NULL);

    fp->_fd = fd;
    fp->cnt = 0;
    fp->_base = NULL;
    fp->_flag &= ^(_READ & _WRITE);
    fp->flag &= (*mode == 'r') ? _READ : _WRITE)_;
    return(fp);
}
```

A fillbuf függvény jóval bonyolultabb. A bonyolultság fő oka, hogy fillbuf akkor is megkísérli az állomány-hozzáférés engedélyezését, ha esetleg a be- és kivitel pufferéléséhez nincs elegendő tár. Ha a calloc-tól további hely nyerhető újabb puffer létrehozására, akkor minden rendben van. Ha nem, akkor a fillbuf pufferéletlenbe- és kivitelt végez egyetlen karakter használatával, amelyet az egyik saját tömbjében tárol.

```
 #include <stdio.h>

/*Bemeneti puffer lefoglalása és feltöltése*/
fillbuf( register FILE *fp)
{
    static char smallbuf[NFILE];       /*pufferéletlen I/O-ra*/
    char *calloc();
    if ((fp->flag& _READ) == 0 || (fp->flag& (_EOF || ERR)) != 0)
         return(EOF);

     while (fp->base == NULL)          /*Pufferterületet keres*/
         if (fp->flag& UNBUF)          /*Pufferéletlen*/
              fp->base = &smallbuf[fp->fd];
         else
              if ((fp->base = calloc(BUFSIZE, 1)) == NULL)
                        fp->flag &= UNBUF; /*Nem kap nagy puffert*/
              else
                   fp->flag &= BIGBUF; /*Nagy puffert kapott*/

     fp->ptr = fp->base;
    fp->cnt = read(fp->fd, fp->ptr, fp-> ~flag
              & _UNBUF ? 1 : _BUFSIZE);
    if (--fp->cnt < 0) {
         if (fp->cnt = -1)
              fp->flag &= &EOF;
         else
              fp->flag &= ERR;

         fp->cnt = 0;
         return(EOF);
    }
    return(*fp->*ptr++ & 0377); /*A karaktert pozitívvá teszi*/
}
```

A getc valamely állományra vonatkozó első hívásakor a darabszám 0, ami előidézi a fillbuf meghívását. Ha a fillbuf úgy találja, hogy az állomány nincs olvasásra megnyitva, azonnal az EOF értékkel tér vissza. Egyébként megkísérli a nagy puffer lefoglalását, és ha ez nem sikerül, az

egykarakteres puffert utalja ki a flag-beli pufferelési információ értelemszerű beállításával. Ha egyszer a puffer létrejött, a fillbuf annak feltöltésére egyszerűen meghívja a read rutint, beállítja a darabszámot és a mutatókat, majd a puffer kezdetén található karakterrel tér vissza. A fillbuf további hívásaikor a puffer már rendelkezésre áll.

Az egyetlen dolog, amit még nem tisztáztunk, hogy mindez hogyan indul. Az stdin, stdout és stderr számára definiálni és inicializálni kell az iob tömböt:

```
FILE iob[NFILE] = {
    {NULL, 0, NULL, READ, 0},          /* stdin*/
    {NULL, 0, NULL, WRITE, 1},         /* stdout*/
    {NULL, 0, NULL, WRITE, UNBUF, 2}   /* stderr*/
};
```

A struktúra flag részének inicializálása mutatja, hogy stdin-t olvasni, stdout-ot írni kell, stderr-re pedig pufferelés nélkül írunk.

8.3. Gyakorlat. Írjuk át fopen-t és fillbuf-ot úgy, hogy explicit bitműveletek helyett mezőket használunk!

8.4. Gyakorlat. Tervezzük és írjuk meg a _flushbuf és fclose rutinokat!

8.5. Gyakorlat. A szabványos könyvtárban rendelkezésünkre áll az fseek(fp, offset, origin)

függvény, amely azonos az lseek függvénnyel attól eltekintve, hogy fp állománymutató és nem állományleíró. Írjuk meg fseek-et! Gondoskodjunk arról, hogy az általunk írt fseek helyesen működjön együtt a könyvtár többi függvényei számára végzett pufferkezeléssel!

8.5. Példa; katalógusok kilistázása

Időnként az eddigiektől eltérő jellegű párbeszédet kell folytatnunk az állományrendszerrel: magára az állományra vonatkozó információra van szükségünk, nem pedig arra, hogy mit tartalmaz az állomány. Példa erre az ls (list directory) nevű UNIX parancs, a mely kinyomtatja az adott katalógusban található állományok nevét, és kívánság szerint egyéb információt is közöl, mint pl. a méreteket, az engedélyeket stb. Mivel legalábbis a UNIX esetében a katalógus maga is egy állomány, semmi különös nincs az olyan parancsokban, mint az ls: beolvas egy állományt, és kiemeli belőle a számára fontos információt.

Ennek az információnak a formátumát ugyanakkor maga a rendszer határozza meg, nem pedig a felhasználói program, így az ls-nek ismernie kell az operációs rendszer ábrázolásmódját. E megjegyzések közül néhányat az fsize program megírásával fogunk szemléltetni. Az fsize az ls olyan speciális formája, amely az argumentumlistájában megnevezett összes állomány méretét kinyomtatja. Ha az állományok valamelyike katalógus, az fsize erre rekurzívan alkalmazza önmagát. Ha egyáltalán nem adtunk meg argumentumot, az aktuális katalógust dolgozza fel. Indulásként röviden átismételjük az állománykezeléssel kapcsolatos tudnivalókat. A katalógus (directory) olyan állomány, amely állománynevek listáját tartalmazza, és utal arra, hogy a megfelelő állományok hol találhatók. Az állományok címe valójában egy másik táblázatba, az inode táblázatba mutató index. Az állomány

inode-ja az a hely, ahol a nevet kivéve az állományra vonatkozó összes információ tárolódik. A katalógus bejegyzés csupán két tételt tartalmaz: az inode-számot és az állomány nevét. A pontos specifikáció a sys/dir.h állomány beiktatásával jön létre, amelynek tartalma:

```
#define DIRSIZ 14 /*Az állománynév max. hossza*/

struct direct /*A katalógusbejegyzés struktúrája*/
{
    ino_t d_ino; /* Inode-szám*/
    char d_name[DIRSIZ]; /*Állománynév*/
};
```

Az ino_t típus olyan typedef, amely az inode-táblázatba mutató indexet ír le. A PDP 11 UNIX esetében ez **unsigned**, de ilyenfajta információt nem szokás a programba ágyazni: más rendszerben ez eltérő lehet. Innen a typedef. A rendszertípusok teljes készlete a sys/types.h-ban található. A stat függvény veszi az állomány nevét, és az annak inode-jában található összes információt (vagy hiba esetén -1-et) adja vissza. Eszerint:

```
struct stat stbuf;
char *name;
stat(name, &stbuf);
```

az állománynévre vonatkozó inode információval tölti fel az stbuf struktúrát. A stat által visszaadott értéket leíró struktúra a sys/stat.h-ban található, formája a következő:

```
struct stat             /*A stat által visszaadott struktúra*/
{
    dev_t st_dev;       /* Az inode perifériája*/
    ino_t st_ino;       /* Inode-szám*/
    short st_mode;      /* Mód bitek*/
    short st_nlink;     /* Az állományra mutató linkek száma*/
    short st_uid;       /* A tulajdonos felhasználó azonosítója*/
    short st_gid;       /* A tulajdonos csoportjának azonosítója*/
    dev_t st_rdev;      /* Speciális állományokra*/
    off_t st_size;      /* Állományméret karakterekben*/
    time_t st_atime;    /* Az utolsó hozzáférés időpontja*/
    time_t st_mtime;    /* Az utolsó módosítás időpontja*/
    time_t st_ctime;    /* Az eredeti létrehozás időpontja*/
};
```

Ezek legtöbbjét a megjegyzések megmagyarázzák. Az st_mode bejegyzés az állományt leíró jelzőket tartalmaz; a kényelem kedvéért a jelződefiníciók ugyancsak részei a sys/stat.h-nak.

```
#define S_IFMT 0160000      /*Az állomány típusa*/
#define S_IFDIR 0040000     /*Katalógus*/
#define S_IFCHR 0020000     /*Speciális karakter*/
#define S_IFBLK 0060000     /*Speciális blokk*/
#define S_IFREG 0100000     /*Szabályos*/
#define S_ISUID 04000       /*Felhasználói azonosító beállítása
                                végrehajtásra*/
#define S_ISGID 02000       /*csoportazonosító beállítása
                                végrehajtásra*/
#define S_ISVTX 01000       /*Az átvitt szöveget használat után
                                menti*/
#define S_IREAD 0400        /*Olvasási engedély*/
#define S_IWRITE 0200       /*Írási engedély*/
#define S_IEXEC 0100        /*Végrehajtási engedély*/
```

Most már meg tudjuk írni az fsize programot. Ha a stat-tól kapott mód azt jelzi, hogy az állomány nem katalógus, akkor a rendelkezésre álló méret közvetlenül kinyomtatható. Ellenkező esetben a katalógust állományonként fel kell dolgoznunk: ez maga is tar tartalmazhat alkatalógusokat, így a folyamat rekurzív. A főrutin szokás szerint elsősorban a parancssor-argumentumokkal foglalkozik: egy nagy pufferben ad át minden egyes argumentumot az fsize függvénynek.

```
#include <stdio.h>
#include <sys/types.h>      /*typedef-ek*/
#include <sys/dir.h>        /*Katalógusbejegyzés struktúra*/
#include <sys/stat.h>       /*A stat által visszaadott struktúra*/
#define BlJFSIZE 256

/*fsize: állományméretek kinyomtatása*/
main(int argc, char *argv[])
{
    char buf[BUFSIZE];
    if (argc == 1) {    /*Alapértelmezés: az aktuális katalógus*/
         strcpy(buf, ".");
         fsize(buf);
    }
    else
         while (argc > 0) {
              strcpy(buf, *++argv);
              fsize(buf);
         }
}
```

Az fsize függvény az állomány méretét nyomtatja ki. Azonban ha az állomány katalógus, akkor először az összes benne levő állomány kezelése érdekében meghívja a directory függvényt. Figyeljük meg a stat.h-ban az S_IFMT és S_IFDIR jelzőnevek használatát:

```
/*Kinyomtatja a megadott nevű állomány méretét*/
fsize(char *name)
{
    struct stat stbuf;
    if (stat(name, &stbuf) == -1) {
         fprintf(stderr, "fsize: %s nem található\n", name);
         return;
    }
    if ((stbuf.st_mode& S_IFMT) == S_IFDIR)
         directory(name);

     printf("%0ld %s\n", stbuf.st_size, name);
}
```

A directory függvény a legbonyolultabb. A legnagyobb része azonban a szóban forgó állomány teljes

elérési nevének (pathname) előállításával foglalkozik.

```
directory( char *name)
{
    struct direct dirbuf;
    char *nbp, *nep;
    int i, fd;
    nbp = name + strlen(name);
    nbp++ = '/';             /*/ hozzáadása a katalógus nevéhez*/
    if (nbp+DIRSIZ+2 >= name+BUFSIZE)  /*A név túl hosszú*/
         return;

    if ((fd = open(name, 0)) == -1)
         return;

    while (read(fd, (char *)&dirbuf, sizeof(dirbuf)) > 0 {
         if (dirbuf.d_ino == 0)   /*A rovat nincs használatban*/
              continue;

         if (strcmp(dirbuf.d_name, ".") == 0 &&
              strcmp(dirbuf.d_name, "..") ==0)
              continue;      /*Önmagát és a szülőt átugorja*/

         for (i = 0 , nep = nbp; i < DIRSIZ; i++)
              *nep++ = dirbuf.d_name[i];

         *nep++ = '\0';
         fsize(name);
    }
    close(fd);
    *nbp = '\0';   /*Név helyreállítása*/
}
```

Ha a katalógus adott rovata éppen nincs használatban (mivel az állományt átnevezték), a mód bejegyzés nulla, és ezt a pozíciót átugorjuk. Minden katalógus tartalmazza bejegyzésként önmagát a "." név alatt, valamint a szülőjét a ".." név alatt. Ezeket nyilván át kell ugrani, különben a program jó ideig futni fog. Bár az fsize program meglehetősen speciális, számos fontos gondolatot mutat be. Először is, sok program nem rendszerprogram, csupán olyan információt használ, amelynek formáját vagy tartalmát az operációs rendszer kezeli.

Másodszor, ilyen programok eset ében lényeges, hogy az információ ábrázolása csak olyan szabványos, ún. fej (header) állományokban jelenjen meg, mint stat.h és dir.h, továbbá, hogy a programok a konkrét deklarációk alkalmazása helyett ezeket az állományokat iktassák be.

8.6. Példa; tárterület lefoglalása

Az 5. fejezetben az alloc egyszerűsített változatát mutattuk be. A most megírandó változat már nem tartalmaz korlátozásokat abban az értelemben, hogy most az alloc és a free hívásai tetszőleges sorrendben követhetik egymást, szükség esetén az alloc az operációs rendszertől igényel további tárterületet. Ezek a rutinok önmagukban is hasznosak, emellett rávilágítanak: arra, hogyan lehet gépfüggő programokat viszonylag gépfüggetlen módon megírni, és a struktúrák, az **union**ok, ill. a typedef valós életből vet t alkalmazásait is bemutatják.

Az alloc a helyfoglalást nem a program részét képező, rögzített méretű tömbből végzi, hanem szükség szerint az operációs rendszertől igényel újabb tárterületet. Mivel a programban folyó egyéb tevékenységek aszinkron módon ugyancsak igényelhetnek helyet, előfordulhat, hogy az alloc által kezelt terület nem lesz folytonos. Így a szabad terület szabad blokkokból álló láncot alkot. A blokkok a tulajdonképpeni szabad hely mellett egy méretet és egy, a következő blokkot megcímző mutatót tartalmazna k. Növekvő tárcím szerint követik egymást, és az utolsó (legmagasabb című) blokk a legelsőre mutat. Ílymódon a lánc valójában gyűrűt képez. Tárkérés esetén a program átvizsgálja a szabad blokkok listáját, hogy tartalmaz-e elegendően nagy szabad blokkot. Ha a talált blokk mérete pontosan megegyezik a kért mérettel, akkor lekapcsolja a listáról és átadja a felhasználónak. Ha a-blokk túlságosan nagy, akkor a program kettévágja, és a felhasználónak csak a megfelelő méretű területet utalja ki, a maradékot pedig visszahelyezi a szabad listába.

Végül, ha nem talált elegendően nagy blokkot, akkor újabb blokkot kér az operációs rendszertől, rákapcsolja a szabad listára, majd újra kezeli a keresést. A blokkfelszabadítás szintén a szabad lista vizsgálatával indul, a programnak ugyanis keresnie kell a listában egy olyan helyet, ahová a felszabadítani kívánt blokkot beilleszthetí. Ha a felszabadított blokk bármelyik oldalán szomszédos egy listabeli blokkal, akkor a kettő egyetlen, nagyobb blokká egyesül, így a tár nem töredezik fel túlságosan.

A szomszédosság tényét könnyen megállapíthatjuk, hiszen a szabad listában a blokkokat címnövekvő sorrendben tartjuk nyilván. Az egyik probléma, amit az 5. fejezetben érintettünk annak biztosítása volt, hogy az alloc által visszaadott terület helyesen illeszkedjen azokhoz az objektumokhoz, amelyeket ott tárolni kívánunk. Bár a gépek különbözőek, minden gépen létezik egy olyan típus, amely, ha egy adott címen tárolható, akkor ott az összes többi típus is biztosan tárolható. Pl. az IBM 360/370, a Honeywell 6000 és sok más gép esetében bármilyen objektum tárolható olyan határon, amely a **double** számára, a PDP 11 esetében pedig az **int** számára megfelelő. A szabad blokkban a tulajdonképpeni szabad területet megelőző vezérlési információt (a láncban következő blokkot megcímző mutatót és a blokk méretét) fejnek nevezzük.

Az illesztés egyszerűsítése érdekében minden blokk a fejméret többszöröse, maga a fej pedig megfelelően illeszkedik. Ezt az alábbi union-nal érhetjük el, amely tartalmazza a kívánt fejstruktúrát, valamint a legnehezebben illeszthető típusra vonatkozó kitételt:

```
typedef int ALIGN;          /*Illeszkedést biztosít a PDP-11-en*/
 union header {             /*Szabad blokk fej*/
    struct {
         union header *ptr;  /*Köv. szabad blokk*/
         unsigned size;      /*Ennek a szabad blokknak a mérete*/
    } s;
    ALIGN x;                /*A blokkok illesztése*/
};

typedef union header HEADER;
```

Az alloc rutinban a karakterekben előírt méretet felkerekítjük a megfelelő számú fejméretű egységgé. A ténylegesen kiutalt blokk eggyel több ilyen egységet tartalmaz, t.i. egy egységre magának a fejnek is szüksége van, és ez a darabszám kerül a fej size mezőjébe. Az alloc által visszaadott mutató a szabad területre mutat, nem pedig magára a fejre.

```
static HEADER base;              /*Üres lista az induláshoz*/
static HEADER *allocp = NULL;    /*Az utolsó lefoglalt blokk*/

char *alloc(unsigned int nbytes) /*Általános célú tárfoglaló*/
{
    HEADER *morecore();
    register HEADER *p, *q;
    register int nunits;
    nunits = 1 + (nbytes + sizeof(HEADER) -1) / sizeof(HEADER);
    if ((q = allocp) == NULL) {       /*Még nincs szabad lista*/
         base.s.ptr = allocp = q = &base;
         base.s.size = 0;
    }
    for (p = q->s.ptr; ; q = p , p = p->s.ptr) {
         if (p->s.size >= nunits) {        /*Elég nagy*/
              if (p->s.size == nunits)     /*Pontosan akkora*/
                        q->s.ptr = p->s.ptr;
              else {                /*A hátsó felét foglalja le*/
                        p->s.size -= nunits;
                        p += p->s.size;
                        p->s.size = nunits;
              }
              allocp = q;
              return((char *)(p + 1));
         }
         if (p == allocp)    /*Körüljárt aa szabad listát*/
              if ((p = morecore(nunits)) == NULL)
                        return(NULL);   /*Nincs több*/
    }
}
```

A base nevű változót használjuk induláskor. Ha, mint alloc első hívásakor, az allocp értéke NULL, egy elfajult szabad lista jön létre: egyetlen, nulla méretű blokkot tartalmaz és saját magára mutat. Ezután a program minden esetben végigkeresi a szabad listát. A megfelelő méretű szabad blokkot azon az (allocp) ponton kezdi keresni, ahol legutoljára talált szabad blokkot; ez a stratégia elősegíti, hogy a lista homogén maradjon. Ha a program túl nagy blokkot talál, akkor a felhasználó a blokk második felét kapja meg, ílymódon az eredeti fejben csak a méretet kell helyesbíteni. A felhasználónak átadott mutató mindig a tényleges szabad területre mutat, amely egy egységgel a fej mögött helyezkedik el. Figyeljük meg, hogy p karakterré alakul át, mielőtt az alloc visszaadná.

A morecore függvény az operációs rendszertől kér tárterületet. Ennek megoldási módja természetesen operációs rendszertől függően változik. A UNIX-ban az sbrk(n) rutin olyan mutatót ad vissza, amely n byte-nyi tárterületre mutat. (A mutató minden illeszkedési megkötésnek eleget tesz.) Mivel tár kérése a rendszertől viszonylag költséges művelet, ezt nem akarjuk az alloc minden hívásakor megtenni, ezért a

morecore a kért egységek számát nagyobb értékre kerekíti fel; ezt a nagyobb blokkot aztán szükség szerint darabolhatjuk fel. A megnövelés értéke olyan paraméter, amely az igényeknek megfelelően változtatható.

```
#define NALLOC 128     /*Az egyszerre lefoglalandó egységek száma*/
static HEADER *morecore(unsigned int nu)        /*Tár kérése a
                                                rendszertől*/
{
    char *sbrk();
    register char * cp;
    register HEADER *up;
    register int rnu;
    rnu = NALLOC *((nu + NALLOC -=1) / NALLOC);
    cp = sbrk(rnu *sizeof(HEADER));
    if ((int)cp == -1)            /*Egyáltalán nincs hely*/
         return(NULL);

    up = (HEADER *)cp;
    up->s.size = rnu;
    free((char *)(up + 1));
    return(allocp);
}
```

Amennyiben nem volt hely, az sbrk = -1-et ad vissza, bár a NULL célszerűbb választás lett volna. A biztonságos összehasonlíthatóság érdekében a -1-et int-té kell alakítani. Ismét sűrűn használtuk a típusmódosítást, így a függvény viszonylag érzéketlen az egyes gépek mutatóábrázolásának különbözőségére. Maga a free utolsónak maradt. Egyszerűen átvizsgálja a szabad listát az allocp-től kezdve, miközben keresi a szabad blokk beillesztésére alkalmas helyet. Ez vagy két, már létező blokk közé esik, vagy a lista végén van. Ha a felszabadítandó blokk bármelyik esetben szomszédos valamely másik szabad blokkal, akkor a program a kettőt egyesíti. Csupán arra kell ügyelni, hogy a mutatók mindig a megfelelő helyre mutassanak és a méretek helyesek legyenek!

```
/*Az ap blokkot a szabad listába teszi*/
free(char *ap)
{
    register HEADER *p, *q;
    p = (HEADER *)ap - 1;              /*A fejre mutat*/
    for (q=allocp; !(p > q && p < q->s.ptr); q=q->s.ptr)
         if (q >= q->s.ptr && (p > q || p < q->s.ptr))
              break;                   /*Egyik vagy másik végén*/

    if (p + p->s.size == q->s.ptr) {   /*Egyesül a felső
                                            szomszéddal*/
         p->s.size += q->s.ptr->s.size;
         p->s.ptr = q->s.ptr->s.ptr;
    }
    else
         p->s.ptr = q->s.ptr;

    if (q + q->s.size == p) {     /*Egyesül az alsó szomszéddal*/
         q->s.size += p->s.size;
         q->s.ptr = p->s.ptr;
    }
    else
         q->s.ptr = p;

    allocp = q;
}
```

Bár a tárterület-foglalás lényegénél fogva gépfüggő, a bemutatott program szemlélteti, hogyan tarthatjuk kézben és korlátozhatjuk a program egészen kis részére a gépfüggő vonatkozásokat. A typedef és az **union** segítségével gondoskodhatunk az összeillesztésről (feltéve, hogy az sbrk a megfelelő mutatót szolgáltatja). A típusmódosító szerkezetek használata explicitté teszi a mutatókonverziókat, és még rosszul tervezett rendszercsatlakozással is megbirkózik. Noha az itt közölt részletek a tárterület-foglalásra vonatkoznak, az elv, a megközelítés más esetekben is alkalmazható.

8.6. Gyakorlat. A calloc(n, size) szabványos könyvtári függvény n darab size nagyságú objektumot megcímző mutatót ad vissza, a tárterület kezdeti nagysága nulla. Írjuk meg a calloc függvényt úgy, hogy az alloc-ot mintaként vagy hívott függvényként használjuk!

8.7. Gyakorlat. Az alloc a méretre vonatkozó kérést anélkül fogadja el, hogy annak jogosságát ellenőrizné. A free azt hiszi, hogy az a blokk, amelynek felszabadítását tőle kérik, érvényes méretű mezőt tartalmaz. Javítsuk e programok minőségét azzal, hogy nagyobb gondot fordítunk a hibaellenőrzésre!

8.8. Gyakorlat. Írjuk meg a bfree(p, n) rutint, amely az alloc és a free által kezelt szabad lista számára felszabadítja az n karakterből álló tetszőleges p blokkot! bfree használatával a felhasználó bármikor

beiktathat a szabad listába egy statikus vagy külső tömböt.

9.A. függelék : C referencia-kézikönyv

1.Bevezetés

A kézikönyv a DEC PDP 11 , a Honeywell 6000, az IBM System/370 és az Interdata 8/32 gépeken használható C nyelvet ismerteti. Eltérések esetén a PDP 11-es változatot helyezi előtérbe, de igyekszik rámutatni a megvalósításfüggő részletekre. Néhány kivételtől eltekintve ezek a gépfüggő részletek közvetlenül a hardver alaptulajdonságaiból következnek; a különféle fordítók általában eléggé kompatibilisek.

2.Szintaktikai egységek

A szintaktikai egységek hat osztályba sorolhatók: azonosítók, kulcsszavak, állandók, karakterláncok, operátorok és egyéb szeparátorok. A szóközöket, tabulátorokat, újsorokat, megjegyzéseket (közös nevükön üres helyeket), mint az alábbiakban is látni fogjuk, a C fordító nem veszi figyelembe, eltekintve attól, hogy feladatuk a szintaktikai egységek elválasztása. Üres helyre van szükség az egyébként szomszédos azonosítók, kulcsszavak és állandók elválasztására. Ha a beolvasott szöveg szintaktikai egységekre bontása adott karakterig megtörtént, a fordító azt a lehető leghosszabb karakterláncot tekinti a következő egységnek, amelyről feltételezhető, hogy még egyetlen szintaktikai egységet képez.

2.1. Megjegyzések

A /* karakterek megjegyzést (comment) vezetnek be, amely a */ karakterekkel zárul. A megjegyzések nem skatulyázhatók egymásba.

2.2. Azonosítók (nevek)

Az azonosító betűk és számjegyek sorozata; az első karakter betű kell, hogy legyen. A aláhúzásjel betűnek számít. A nagy- és kisbetűk különbözők. Csupán az első nyolc karakter értékes, bár több is használható. A különféle , assemblerek és betöltőprogramok által használt külső azonosítók ennél kötöttebbek:

DEC PDP 11 7 karakter, kétféle betűtípus (kis- és nagybetű).

Honeywell 6000 6 karakter, egyféle betűtípus.

IBM 360/370 7 karakter, egyféle betűtípus.

Interdata 8/32 8 karakter, kétféle betűtípus.

2.3. Kulcsszavak

Az alábbi azonosítók a nyelv kulcsszavai, így egyéb célra nem használhatók:

int extern else char register for float typedef do double static while struct goto switch union return case long sizeof default short break entry auto unsigned continue if

Az entry kulcsszót egyetlen jelenleg működő fordítóban sem valósították meg, későbbi fejlesztésekhez tartottuk fenn. Bizonyos megvalósításokban a fortran és az asm szavak is kulcsszóként szerepelnek.

2.4. Állandók

Többfajta állandó van; ezeket a következőkben soroljuk fel. A méreteket érintő hardverjellemzőket a 2.6. pontban foglaljuk össze.

2.4.1. Egész állandók

A számjegyek sorozatát tartalmazó egész típusú (integer) állandót a fordító oktálisnak tekinti, ha 0-val (a nulla számjeggyel) kezdődik, egyébként decimálisnak veszi. A 8 és 9 számjegyek oktális értéke 10, ill. 11 . Az olyan számjegysorozatot, amelyet 0X vagy 0x (a 0 a nulla számjegy) előz meg, a fordítóprogram hexadecimális egésznek tekinti. Hexadecimális számjegyek az a-tól, ill. A-tól f-ig, ill. F-ig elhelyezkedő karakterek, amelyeknek értéke 10, . . ., 15. Azt a decimális állandót, amelynek értéke meghaladja a gépen ábrázolható legnagyobb előjeles egészt, a fordítóprogram **long**-nak veszi; hasonlóképpen **long** lesz az az oktális vagy hexadecimális állandó, amelynek értéke meghaladja a legnagyobb, előjel nélküli gépi egészt.

2.4.2. Explicit long állandók

Az a decimális, oktális vagy hexadecimális egész, amelyet közvetlenül l ("el" betű) vagy L követ, **long** (hosszú) állandó. Amint arról az alábbiakban szó lesz, bizonyos gépeken az **int** és **long** értékek azonosak.

2.4.3. Karakterállandók

A karakterállandó aposztrófok (szimpla idézőjelek) közé zárt karakter, pl. 'x'. A karakterállandó értéke a karakternek a gép karakterkészletében szereplő numerikus értéke. Bizonyos nem grafikus karaktereket, pl. az aposztrófot (') vagy a fordított törtvonalat (\) az alábbi escape-szekvenciákkal ábrázolhatunk:

újsor	NL (LF) \n
vízszintes tab	HT \t
vissza-szóköz	BS \b
kocsi-vissza	CR \r
lapdobás	FF \f

fordított törtvonal	\ \\
aposztróf	' \'
bitminta	ddd \ddd

A \ddd escape-szekvencia egy fordított törtvonalat és 1 , 2 vagy 3 rá következő oktális számjegyet tartalmaz, amelyek a kívánt karakter értékét határozzák meg. E konstrukció speciális esete a \0 (amit nem követ számjegy), amely a NULL karaktert jelöli. Ha a fordított törtvonalat követő karakter nem az előbbiek egyike, a fordító a fordított törtvonalat nem veszi figyelembe.

2.4.4. Lebegőpontos állandók

A lebegőpontos állandó egész részből, tizedespontból, törtrészből, e-ből vagy E-ből és (esetleg előjeles) kitevőből áll. Mind az egész, mind a tört rész számjegyek sorozata. Akár az egész, akár a tört rész hiányozhat (de mind a kettő nem!); ill. a tizedes pont vagy az e és a kitevő közül az egyik szintén elmaradhat. Minden lebegőpontos állandó dupla pontosságú.

2.5. Karakterláncok

A karakterlánc idézőjelek közé zárt karaktersorozat: ". . .". A karakterlánc típusa szerint karaktertömb, tárolási osztálya **static** (l. a következőkben a 4. szakaszt), és a megadott karakterek inicializálják. Az egyes karakterláncok, még az azonos módon leírtak is, külön egységet képeznek. A fordító minden karakterlánc végére elhelyezi a \0 nullabyte-ot abból a célból, hogy a karakterláncot vizsgáló programok megtalálják a karakterlánc végét. A karakterláncon belül elhelyezett " idézőjelet \ kell, hogy megelőzze; a karakterállandóknál ismertetett összes escape-szekvencia használható. Végül megjegyezzük, hogy az \-t és az azt közvetlenül követő újsort a fordító nem veszi figyelembe.

2.6. Hardverjellemzők

Az alábbi táblázatban néhány olyan hardvertulajdonságot foglaltunk össze, amely gépről gépre változik. Noha ezek a programok gépfüggetlenségét érintik, mégis jóval kisebb problémát okoznak, mint azt valaki eleve gondolná. (A számok bitekben értendők.)

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
char	8	9	8	8
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32

double	64	72	64	64
értéktartomány	+-10+-38	+-10+-38	+-10+-76	+-10+-76

E négy gép esetében a lebegőpontos számoknak 8 bites kitevőjük van.

3. A szintaxis jelölése

A kézikönyvben használt szintaktikai jelölésmódban a kulcsszavakat és karaktereket - ahol az egyértelműség megkívánja - kövér szedéssel jelöljük. A választható (alternatív) kategóriák külön sorban szerepelnek. Az elhagyható (opcionális) szimbólumokat az "opc" index jelöli, így

{ kifejezésopc }

kapcsos zárójelek közé zárt elhagyható kifejezést jelöl. A szintaxist később a 18. pontban foglaljuk össze.

4. Az azonosítók értelmezése

A C nyelv az azonosítók értelmezését az azonosítók két tulajdonságára alapozza: a tárolási osztályára és a típusára. A tárolási osztály az azonosítóhoz rendelt tárhely elhelyezkedését és élettartamát, a típus az azonosítóhoz rendelt tárterületen talált értékek jelentését határozza meg. Négy deklarálható tárolási osztály van: automatikus, statikus, külső és regiszterosztály. Az automatikus változók egy blokk minden hívására nézve lokálisak (l. a 9.2. pontot) értéküket a blokkból való kilépéskor elvesztik; a statikus változók egy blokkra nézve lokálisak, de még akkor is megtartják értéküket, ha a vezérlés időközben kilépett a blokkból; a külső változók megmaradnak és megtartják értéküket az egész program végrehajtása során és függvények közötti kommunikációra használhatók, még külön-külön lefordított függvények esetében is.

A regisztervváltozók (ha lehetséges) a gép gyors regisztereiben tárolódnak; az automatikus változókhoz hasonlóan az egyes blokkokra nézve lokálisak és a blokkból való kilépéskor eltűnnek. A C nyelv több alapvető objektumtípus használatát engedi meg: A karakterként (char) deklarált objektumok elegendően nagyok ahhoz, hogy az adott implementáció karakterkészletének tetszőleges elemét tárolni tudják, és ha valóban egy, az illető karakterkészletből vett karaktert akarunk karakter típusú változóban tárol ni, annak értéke meg fog egyezni a karakter egész értékű kódjával. Más mennyiségek is tárolhatók karakter típusú változókban, de ennek megvalósítása gépfüggő. Maximum háromféle egész típusú méret áll rendelkezésre, amelyeket **short int** (rövid egész), **int** (egész) és **long int** (hosszú egész) alakban deklarálunk. A hosszabb egészek bizonyosan nem igényelnek kevesebb tárhelyet, mint a rövidebbek, de az adott ny elvi megvalósítás a **short** int-eket a **long** int-ekkel vagy akár mind a kettőt közönséges egészekkel (int) egyenlő méretűvé teheti.

A közönséges egészeknek a befogadó gép architektúrájából következő természetes méretük van; a többi méret speciális igénye k kielégítésére szolgál. Az **unsigned**-ként deklarált, előjel nélküli egészekre a modulo 2n aritmetika szabályai vonatkoznak, ahol n a bitek száma az adott megvalósításban. (A PDP-11 az előjel nélküli **long** mennyiségeket nem támogatja.) Az egyszeres pontosságú lebegőpontos (**float**) és a dupla pontosságú lebegőpontos (**double**) ábrázolás egyes gépeken azonos lehet.

Mivel az említett típusú objektumok célszerűen értelmezhetők számokként, ezekre mint aritmetikai típusokra fogunk hivatkozni. Az összes char és int típust (mérettől függetlenül) együttesen integrális típusnak, a float-ot és a double-t együttesen lebegőpontos típusnak fogjuk nevezni. Az alapvető aritmetikai típusokon kívül elvileg végtelen számú leszármaztatott típus képezhető az alaptípusokból, az alábbi módokon:

- tömbök, amelyek a legtöbb típusú objektumból képezhetők;
- függvények, amelyek adott típusú objektumot adnak vissza;
- mutatók, amelyek adott típusú objektumra mutatnak;
- struktúrák, amelyek különféle típusú objektumok sorozatát tartalmazzák;
- **union**ok, amelyek különféle típusú objektumok bármelyikét tartalmazhatják.

Az objektumok létrehozásának ezek a módszerei általában rekurzív módon alkalmazhatók.

5. Objektumok és balértékek

Az objektum a tár valamely műveletekkel kezelhető része; a balérték (lvalue) objektumra hivatkozó kifejezés. A balérték kifejezésre kézenfekvő példa az azonosító. Bizonyos operátorok balértékeket eredményeznek: ha E mutató típusú kifejezés, akkor *E olyan balérték kifejezés, amely arra az objektumra hivatkozik, amire az E mutat. A balérték elnevezés az E1 =E2 értékadó kifejezésből származik, amelyben az E1 bal oldali operandusnak balérték kifejezésnek kell lennie. Az egyes operátorok alább következő ismertetése során közöljük hogy az adott operátor balérték operandusokat vár-e és hogy balértéket ad-e eredményül.

6. Konverziók

Operandusuktól függően számos operátor válthatja ki valamelyik operandusa értékének egyik típusból valamilyen másik típusba történő átalakítását. Ebben a szakaszban az ilyen konverziók várható eredményét ismertetjük. A közönséges operátorok többsége által megkövetelt konverziókat a 6.6. pontban foglaltuk össze; ezt szükség szerint az egyes operátorok tárgyalásánál további információkkal egészítettük ki.

6.1. Karakterek és egészek

Karaktert és rövid _egészt mindenütt használhatunk, ahol közönséges egész használható. Az érték minden esetben int-té alakul. Rövidebb egész hosszabb egésszé történő konvertálása mindig előjel-kiterjesztéssel jár: az egészek előjeles mennyiségek. Az adott géptől függ, hogy karakterek esetében is történik-e előjel-kiterjesztés, de annyi bizonyos, hogy a szabványos karakterkészlet valamennyi eleme nemnegatív. Azok közül a számítógépek közül, amelyeket ez a kézikönyv figyelembe vesz, csak a PDP- 11 végez előjel-kiterjesztést. A PDP-11-en a karakter típusú változók értéktartománya -128 és 127 között van; az összes ASCII karakter pozitív. Az oktális escape-szekvencia segítségével megadott karakterállandókra előjel-kiterjesztés történik, és negatívként is megjelenhetnek, pl. '\077' értéke -1. Ha egy hosszabb egészt rövidebb egésszé vagy char-rá alakítunk, a levágás bal oldalon történik : a

felesleges bitek egyszerűen elmaradnak.

6.2. Float és double

A C-ben mindenféle lebegőpontos művelet dupla pontosságú; amikor egy kifejezésben **float** fordul elő, az a tört rész nullákkal való kitöltése révén **double**-lá hosszabbodik. Ha double-t kell float-tá alakítani, pl. értékadás során, a double először kerekítődik és csak ezután rövidül **float** hosszúságúvá.

6.3. Lebegőpontos és integrális mennyiségek

A lebegőpontos értékek integrális típusúvá alakítása általában eléggé gépfüggő művelet; különösképpen a negatív számok csonkításának iránya változik gépről gépre. Ha a rendelkezésre álló helyen az eredmény nem fér el, határozatlan lesz. Integrális értékek lebegőpontossá alakítása problémamentes. A pontosság némileg csökken, ha a célhelyen nincs elegendő bit.

6.4. Mutatók és egészek

Az **int** vagy **long int** mennyiségek a mutatókhoz hozzáadhatók vagy azokból levonhatók; ebben az esetben az előbbiek az összeadó operátornál leírtak szerint alakulnak át. Két, ugyanolyan típust megcímző mutató egymásból kivonható: ez esetben az eredmény egésszé alakul át, amint azt a kivonó operátornál tárgyaljuk.

6.5. Előjel nélküli egészek

Ha előjel nélküli (**unsigned**) és közönséges egészeket kombinálunk, a közönséges egész előjel nélkülivé alakul át, és az eredmény is előjel nélküli. Az érték az a legkisebb előjel nélküli egész, amely kongruens az előjeles egésszel (modulo 2szóméret). 2-es komplemensű ábrázolásban a konverzió csupán elvi, a bitminta valójában nem változik. Ha az előjel nélküli egész **long**-gá alakul, az eredmény értéke számszerűleg ugyanaz, mint az előjel nélküli egészé. Így a konverzió csupán a bal oldali kitöltő nullák elhelyezéséből áll.

6.6. Aritmetikai konverziók

Számos operátor hasonló konverziót vált ki, és az eredményt ugyanabban a típusban szolgáltatja. Ezt az eljárást szokásos aritmetikai konverziónak nevezni. Először is minden **char** vagy **short** típusú operandus int-té és minden **float** operandus **double**-lá alakul. Ezután, ha valamelyik operandus **double**, akkor a másik is **double**-lá alakul, és az eredmény szintén **double** lesz. Egyébként, ha valamelyik operandus **long**, a másik operandus és az eredmény típusa is **long** lesz. Egyébként, ha valamelyik operandus **unsigned**, a másik is unsigned-á alakul, és ez lesz az eredmény típusa is. Minden más esetben mindkét operandusnak int-nek kell lennie és ez lesz az eredmény típusa is.

7. Kifejezések

A kifejezésekben előforduló operátorok precedenciája ugyanaz, mint ebben a fejezetben az alfejezetek (pontok) sorrendje; a legmagasabb precedencia az első. Így pl. azokat a kifejezéseket, amelyekre mint a

+ operandusaira hivatkozunk (7.4. pont) a 7.1 ... 7.3. pontokban definiáljuk. Az egyes pontokon belül minden operátor azonos precedenciájú. Minden pontban megadjuk, hogy az ott tárgyalt operátorokra bal-, ill. jobb-irányú asszociativitás vonatkozik-e. A kifejezésekben alkalmazott operátorok precedenciáját és asszociativitását a 18. pontban közölt nyelvtan foglalja össze. Egyéb esetekben a kifejezések kiértékelésének sorrendje határozatlan. A fordítóprogram a részkifejezéseket saját megítélése szerint abban a sorrendben számítja ki, amit leghatékonyabbnak vél, még abban az esetben is, ha a részkifejezéseknek mellékhatásai k vannak. A mellékhatások előfordulásának sorrendje meghatározott. Kommutatív és asszociatív operátorokat (*, +, &, |, n~) tartalmazó kifejezések tetszés szerint rendezhetők még zárójelek jelenlétében is; ha adott sorrendben végzendő kiértékelést kívánunk előírni, explicit ideiglenes változót kell használnunk.

A kifejezések kiértékelése során a túlcsordulás és az osztás ellenőrzésének kezelése gépfüggő. A C nyelv minden létező megvalósítása figyelmen kívül hagyja az egészek túlcsordulását; a 0-val való osztás kezelése, ill. a lebegőpontos kivételek gépről gépre változnak, és általában valamilyen könyvtári függvénnyel módosíthatók.

7.1. Elsődleges kifejezések

A . és -> szimbólumokat, indexelést és függvényhívásokat tartalmazó elsődleges kifejezések csoportosítása balról jobbra történik.

- elsődleges_kifejezés:
- azonosító
- állandó
- karakterlánc (kifejezés)
- elsődleges_kifejezés [kifejezés]
- elsődleges_kifejezés [kifejezéslistaopc]
- elsődleges_balérték.azonosító
- elsődleges_kifejezés->azonosító

Kifejezéslista:

- kifejezés
- kifejezéslista, kifejezés

Az azonosító elsődleges kifejezés, feltéve, hogy az alábbi ismertetett módon helyesen deklarálták. Típusát a deklarációja határozza meg. Ha azonban az azonosító típusa valamilyen tömb, akkor az azonosító kifejezés értéke a tömb első objektumát megcímző mutató, és a kifejezés típusa a tömb alaptípusára hivatkozó mutató. A tömbazonosító továbbá nem balérték kifejezés. Hasonlóképpen a függvényként deklarált azonosító is a függvény mutatójává alakul át, kivéve, ha valamely függvényhívás függvénynév-pozícióján fordul elő. Az állandó elsődleges kifejezés. Típusa az alakjától függően lehet int, **long** vagy **double**. A karakterállandók típusa int, a lebegőpontos állandóké **double**. A karakterlánc elsődleges kifejezés. Típusa eredetileg char-ok tömbje, de az azonosítókra vonatkozó

fenti szabály értelmében az a char-mutatóvá módosul, és az eredmény a karakterlánc első karakterét megcímző mutató.

(Kivételt képeznek egyes kezdetiérték- beállítók (l. a 8.6. pontot.)) A zárójelezett kifejezés olyan elsődleges kifejezés, amelynek típusa és értéke azonos a zárójel nélküli kifejezésével. A zárójelek jelenléte nem befolyásolja azt a tényt, hogy a kifejezés balérték-e vagy sem. Az elsődleges kifejezés és az azt követő szögletes zárójelek közötti kifejezés szintén elsődleges kifejezést képez [kifejezés]. Az elsődleges kifejezés általában valamilyen mutató típusú, az index kifejezés int, és az eredmény típusa az a típus, amelyre a mutató mutat. Az E1[E2] kifejezés definíció szerint azonos a *((E1)+(E2))-vel.

Ez a pont, valamint az azonosítókkal, a +-szal, ill. *-gal foglalkozó 7.1., 7.2., ill. 7.4. pont az összes tudnivalót tartalmazza, ami ennek a jelölésmódnak a megértéséhez szükséges. Az indexelésről a 14.3. pontban szólunk. A függvényhívás olyan elsődleges kifejezés, amelyet zárójelek között a függvény aktuális argumentumait alkotó kifejezések esetleg üres, vesszőkkel elválasztott listája követ. Az elsődleges kifejezésnek "függvény, amely visszaadja . . .-t" típusúnak kell lennie, és a függvényhívás eredménye " . . . " típusú. Mint a következőkben látni fogjuk, minden korábban elő nem fordult azonosító, amelyet közvetlenül nyitó zárójel követ, a szövegkörnyezet alapján egészt visszaadó függvényként deklarálódik, így a legközönségesebb esetben az egész értékű függvényeket nem kell deklarálni.

A **float** típusú argumentumok hívás előtt **double**-lá alakulnak át; minden **char** és **short** int-té konvertálódik, és a tömbnevek, mint mindig, mutatókká alakulnak. Automatikusan semmilyen más konverzió nem történik; lényeges tudnunk, hogy a fordító az aktuális argumentumok típusát nem hasonlítja össze a formális argumentumokéval. Ha konverzióra van szükség, használjunk típusmódosító szerkezetet (l. a 7.2. és 8.7. pontot).

A függvényhívás előkészítéseképpen másolat készül minden aktuális paraméterről, így a C nyelvben minden argumentumátadás szigorúan érték szerint történik. A függvény megváltoztathatja formális paramétereinek értékét, de ezek a változtatások nem befolyásolhatják az aktuális paraméterek értékét. Lehetőség van viszont mutató átadására, tudva azt, hogy a függvény megváltoztathatja annak az objektumnak az értékét, amelyre a mutató mutat.

A tömbnév mutatókifejezés. Az argumentumok kiértékelésének sorrendjét a nyelv nem definiálja; ne feledjük, hogy a különböző fordítók eltérőek! Bármilyen függvény rekurzív módon hívható. Egy elsődleges kifejezés, az azt követő pont és az azután következő azonosító együttesen kifejezést alkot.

Az első kifejezésnek olyan balértéknek kell lennie, amely struktúrát vagy **union**t nevez meg, az azonosító pedig meg kell, hogy nevezze a struktúra v agy **union** egy tagját. Az eredmény a struktúra vagy **union** megnevezett tagjára vonatkozó balérték. Egy elsődleges kifejezés, az azt követő nyíl (amelyet egy - és egy > alkot) és az azután következő azonosító együttesen kifejezést alkot. Az első kifejezésnek struktúrát vagy **union**t megcímző mutatónak kell lennie, és az azonosítónak a struktúra vagy **union** egy tagját kell megneveznie.

Az eredmény olyan balérték, amely a mutatókifejezés által megcímzett struktúra vagy **union** megnevezett tagjára vonatkozik. Így az E1->MOS kifejezés azonos a (*E1).MOS kifejezéssel. A struktúrákkal és **union**okkal a 8.5. pont foglalkozik. A használatukra vonatkozóan itt megadott szabályokat a fordító rugalmasan alkalmazza, hogy ki lehessen lépni a típusmechanizmusból (l. a 14.1 . pontot).

7.2. Egyoperandusú operátorok

Az egyoperandusú operátorokkal alkotott kifejezések csoportosítása jobbról balra történik.

egyoper_kifejezés:

- *kifejezés
- &balérték
- -kifejezés
- !kifejezés
- ~kifejezés
- ++balérték
- --balérték
- balérték++
- balérték--
- (típusnév) kifejezés
- sizeof kifejezés
- sizeof (típusnév)

Az egyoperandusú * operátor indirekciót fejez ki: a kifejezés mutató kell hogy legyen, és az eredmény olyan balérték, amely a kifejezés által megcímzett objektumra vonatkozik. Ha a kifejezés mutató típusú, akkor az eredmény típusa a mutatóval megcímzett objektum típusa. Az egyoperandusú & operátor hatására a balérték által hivatkozott objektumot megcímző mutató keletkezik. Ha a balérték típusa ". . .", akkor az eredmény típusa "mutató . . .-ra" Az egyoperandusú – operátor az operandus negatív értékét eredményezi. A szokásos aritmetikai konverziók mennek végbe. Előjel nélküli (**unsigned**) mennyiség esetében a negatív értéket úgy kell kiszámítani, hogy 2n-ből levonjuk az operandus értékét, (n az int-beli bitek száma). Egyoperandusú + operátor nincs. A ! logikai negálóoperátor hatására az eredmény 1 lesz, ha az operandus nulla, 0 lesz, ha az operandus nem-nulla. Az eredmény típusa int.

Bármilyen aritmetikai típusra és mutatókra alkalmazható. A ~ operátor hatására az operandus 1-es komplemense jön létre. Megtörténnek a szokásos aritmetikai konverziók. Az operandus integrális típusú kell, hogy legyen. A ++ operátor balérték operandusa előtt alkalmazva inkrementálja az operandus által hivatkozott objektumot. Az érték az operandus új értéke, amely azonban nem balérték. A ++x kifejezés x+=1-gyel egyenértékű. A konverziókra vonatkozóan l. az összeadásra (7.4. pont) és értékadó operátorokra (7.14. pont) vonatkozó ismertetést.

A -- operátor, ha balérték operandusa előtt áll, az előbbiekhez hasonlóan dekrementálja az operandust. Ha a ++ operátort valamely balérték után alkalmazzuk, az eredmény a balérték által hivatkozott objektum értéke lesz. Az eredmény feljegyzése után az objektum ugyanúgy inkrementálódik, mint az elölről alkalmazott ++ operátor esetében. Az eredmény típusa ugyanaz, mint a balérték kifejezésé. Ha a – operátort valamely balérték után alkalmazzuk, az eredmény a balérték által hivatkozott objektum

értéke lesz. Az eredmény feljegyzése után az objektum ugyanúgy dekrementálódik, mint az előtag -- operátor esetében.

Az eredmény típusa ugyanaz, mint a balérték kifejezésé. Ha egy kifejezést valamelyik adattípus zárójelek közé írt neve előz meg, a kifejezés értéke a megadott típusúvá alakul át. Ezt a konstrukciót típusmódosító szerkezetnek (cast) nevezzük. A típusneveket a 8.7. pontban írjuk le. A sizeof operátor az operandusának a byte-okban kifejezett méretét állítja elő. (A byte-ot a nyelv csupán sizeof értékének segítségével definiálja. Azonban minden létező megvalósításban a byte az a terület, amely alkalmas egy **char** tárolására.) Tömbre alkalmazva az eredmény az összes tömbbeli byte-ok száma lesz. A méretet a kifejezésben előforduló objektumok deklarációi határozzák meg.

Ez a kifejezés szemantikailag egész típusú állandó, bárhol használható, ahol állandóra van szükség. Leginkább olyan rutinokkal történő kommunikáció céljaira használatos, mint pl. a tárterület-foglaló függvények és a be- és kivitel rendszerek. A sizeof operátor zárójelben álló típusnévre is alkalmazható. Ekkor egy, a megjelölt típusú objektum méretét szolgáltatja byte-okban. A sizeof(típus) szerkezet összefüggő egység, így a sizeof(típus)-2 kifejezés ugyanaz, mint (sizeof(típus))-2.

7.3. Multiplikatív operátorok

A * , / és % multiplikatív operátorok balról jobbra csoportosítanak. Megtörténnek a szokásos aritmetikai konverziók.

multiplikatív_kifejezés:

- kifejezés * kifejezés
- kifejezés / kifejezés
- kifejezés % kifejezés

A kétoperandusú * operátor a szorzást jelöli. A * operátor asszociatív, és az ugyanazon a szinten több szorzást tartalmazó kifejezéseket a fordító átrendezheti. A kétoperandusú / operátor az osztást jelöli. Pozitív egészek osztásakor a csonkítás nulla felé történik, de ha bármelyik operandus negatív, akkor a csonkítás formája gépfüggő. Az ebben a kézikönyvben figyelembe vett gépek esetében az osztandó és a marad ék előjele megegyezik. Mindig igaz, hogy

```
(a / b) * b + a % b
```

megegyezik a-val (ha b nemnulla). A kétoperandusú % operátor az első kifejezésnek a másodikkal történő osztásából származó maradékot állítja elő. A művelet szokásos aritmetikai konverziókkal jár. Az operandusok nem lehetnek **float** típusúak.

7.4. Additív operátorok

A + és - additív operátorok balról jobbra csoportosítanak. A szokásos aritmetikai konverziókat eredményezik. Mindkét operátor esetében vannak további típuslehetőségek.

additív kifejezés:

- kifejezés + kifejezés
- kifejezés - kifejezés

A + operátor alkalmazásának eredménye az operandusok összege. Egy tömbbeli objektumot megcímző mutató és bármelyik integrális típus értéke összeadható. Az utóbbi minden esetben relatív címmé alakul oly módon, hogy megszorzódik annak az objektumnak a hosszúságával, amelyre a mutató mutat. Az eredmény az eredetivel megegyező típusú mutató, amely ugyanannak a tömbnek egy másik elemére mutat, megfelelő eltolással az eredeti objektumhoz képest. Ha tehát P tömbelemet megcímző mutató, akkor a P+1 kifejezés a tömb következő elemét megcímző mutató lesz. Mutatókra semmilyen más típusú kombináció sem megengedett! A + operátor asszociatív, és az ugyanazon a szinten több összeadást tartalmazó kifejezéseket a fordító átrendezheti.

A - operátor alkalmazásának hatására a két operandus különbsége keletkezik, a szokásos aritmetikai konverziók alkalmazásával. Ezenkívül mutatókból le szabad vonni bármely integrális típusú értéket, ekkor megtörténnek ugyanazok a konverziók, mint az összeadásnál. Ha két ugyanolyan típusú objektumot megcímző mutatót vonunk ki egymásból, az eredmény (az objektum hosszával történő osztás révén) int-té alakul, és a megcímzett objektumok között elhelyezkedő objektumok darabszámát adja meg. Általános esetben ez a konverzió váratlan eredményre vezet, kivéve, ha a mutatók ugyanannak a tömbnek az elemeire mutatnak. Ennek az az oka, hogy még az ugyanolyan típusú objektumok távolsága sem feltétlenül az objektumhosszúság többszöröse.

7.5. Léptető operátorok

A << és >> léptető (shift) operátorok balról jobbra csoportosítanak. Mindkettő elvégzi az operandusokon a szokásos aritmetikai konverziókat; az operandusok mindegyike integrális kell, hogy legyen. A művelet során a jobb oldali operandus int-té alakul át; az eredmény típusa megegyezik a bal oldali operanduséval. Az eredmény határozatlan, ha a jobb oldali operandus negatív vagy nagyobb, mint az objektum bitekben mért hosszúsága, vagy pedig azzal megegyezik.

léptető_kifejezés:

- kifejezés << kifejezés
- kifejezés >> kifejezés

Az E1<<E2 értéke a bitmintaként értelmezett E1 E2 számú bittel balra léptetve; a kiürült bitek 0-val töltődnek fel. Az E1>>E2 értéke úgy áll elő, hogy E1 értéke E2 bittel balra léptetődik. A jobbra garantáltan logikai jellegű (0-val történő feltöltés), ha az E1 unsigned; más esetben aritmetikai lehet (és a PDP 11-en az is lesz) ilyenkor a feltöltődés az előjelbittel történik.

7.6. Relációs operátorok

A relációs operátorok balról jobbra csoportosítanak, de ez a tény nem különösebben hasznos: a < b < c jelentése nem az, amit gondolnánk.

relációs_kifejezés:

- kifejezés < kifejezés
- kifejezés > kifejezés
- kifejezés <= kifejezés
- kifejezés >= kifejezés

A < (kisebb, mint), > (nagyobb, mint), <= (kisebb vagy egyenlő) és >= (nagyobb vagy egyenlő) operátorok mindegyike 0-t eredményez, ha a megadott reláció értéke hamis, és 1 -et, ha igaz. Az eredmény típusa int. A műveletek a szokásos aritmetikai konverziókkal járnak. Két mutató összehasonlítható: az eredmény a megcímzett objektumok címének a címtartományban való egymáshoz képesti elhelyezkedésétől függ. A mutató összehasonlítás csak akkor gépfüggetlen, ha a mutatók ugyanabban a tömbben elhelyezkedő objektumokra mutatnak.

7.7. Egyenlőségi operátorok

egyenlőség_kifejezés:

- kifejezés == kifejezés
- kifejezés != kifejezés

A == (egyenlő) és != (nem egyenlő) operátorok pontosan ugyanolyanok, mint a relációs operátorok - csak a precedenciájuk alacsonyabb. (Így

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

értéke 1 , ha a < b és c < d igazságértéke megegyezik.) Mutató és egész összehasonlítható, de az eredmény gépfüggő, kivéve ha az egész a 0 állandó. Az a mutató, amelyhez a 0-t rendeltünk hozzá, garantáltan nem mutat semmilyen objektumra, és 0-val egyenlőként fog megjelenni; a hagyományos használatban az ilyen mutatót nullának tekintjük.

7.8. Bitenkénti ÉS operátor

és kifejezés:

- kifejezés & kifejezés

Az & operátor asszociatív, és az &-et tartalmazó kifejezések átrendezhetők. A szokásos aritmetikai konverziók mennek végbe; az eredmény az operandusok bitenkénti ÉS függvénye. Az operátor csak integrális operandusokra alkalmazható!

7.9. Bitenkénti kizáró VAGY operátor

kizáró vagy kifejezés:

- kifejezés ^ kifejezés

A ^ operátor asszociatív, és a ^-t tartalmazó kifejezések átrendezhetők. A művelet a szokásos aritmetikai konverziókkal jár; az eredmény az operandusok bitenkénti kizáró VAGY függvénye. Az operátor csak integrális operandusokra alkalmazható!

7.10. Bitenkénti inkluzív VAGY operátor

inkluzív vagy kifejezés:

- kifejezés | kifejezés

A | operátor asszociatív, és a |-ot tartalmazó kifejezések átrendezhetők. A művelet a szokásos aritmetikai konverziókkal jár; az eredmény az operandusok bitenkénti inkluzív VAGY függvénye. Az operátor csak integrális operandusokra alkalmazható!

7.11. Logikai ÉS operátor

logikai_és_kifejezés:

- kifejezés && kifejezés

Az && operátor balról jobbra csoportosít. 1-et ad vissza; ha egyik operandusa sem nulla, egyébként 0- . Az &-től eltérően az && biztosítja a balról jobbra történő kiértékelést; ezen felül a második operandus nem értékelődik ki, ha az első 0. Az operandusoknak nem kell azonos típusúaknak lenniük, de mindegyikük típusa vagy valamelyik alaptípus, vagy pedig mutató kell, hogy legyen. Az eredmény mindig int.

7.12. Logikai VAGY operátor

logikai_vagy_kifejezés:

- kifejezés || kifejezés

A || operátor balról jobbra csoportosít. 1_t ad vissza, ha valamelyik operandusa nemnulla, 0-t egyébként. A |-tól eltérően a || biztosítja a balról jobbra történő kiértékelést; ezen felül a második operandus nem értékelődik ki, ha az első nemnulla. Az operandusoknak nem kell azonos típusúaknak lenniük, de mindegyikük típusa vagy valamelyik alaptípus, vagy pedig mutató kell, hogy legyen. Az eredmény mindig int.

7.13. A feltételes operátor

feltételes_kifejezés:

- kifejezés ? kifejezés : kifejezés

A feltételes kifejezések balról jobbra csoportosítanak. Az első kifejezés kiértékelődik, és ha az értéke nemnulla, az eredmény a második kifejezés értéke lesz, egyébként pedig a harmadik kifejezésé. Lehetőség szerint megtörténnek a szokásos aritmetikai konverziók, amelyek révén a második és a harmadik kifejezés azonos típusúvá válik; egyébként, ha mindkettő ugyanolyan típusú mutató, az

eredmény típusa ez a közös típus lesz; vagy pedig az egyiknek mutatónak, a másiknak a 0 állandónak kell lennie, és az eredmény típusa a mutató típusa lesz. A második és a harmadik kifejezés közül csak az egyik értékelődik ki.

7.14. Értékadó operátorok

Több értékadó operátor van, amelyek mindegyike jobbról balra csoportosít. Bal oldali operandusként mindegyikük egy-egy balértéket igényel, az értékadó kifejezés típusa a bal oldali operandus típusával fog megegyezni. Az értékadó_ kifejezés értéke az az érték lesz, amely az értékadás után a bal oldali operandusban található. Az összetett értékadó operátor két része különálló szintaktikai egységet képez.

értékadó_kifejezés:

- balérték = kifejezés
- balérték += kifejezés
- balérték -= kifejezés
- balérték *= kifejezés
- balérték /= kifejezés
- balérték %= kifejezés
- balérték >>= kifejezés
- balérték <<= kifejezés
- balérték &= kifejezés
- balérték ^= kifejezés
- balérték |= kifejezés

A legegyszerűbb értékadásnál, ahol az = operátort alkalmazzuk, a kifejezés értéke behelyettesítődik a balérték által hivatkozott objektum értékébe. Ha mindkét operandus aritmetikai típusú, a jobb oldali operandus még az értékadás előtt bal oldali típusúvá alakul át. Az

E1 op= E2

alakú kifejezés hatását kikövetkeztethetjük, ha azt

E1 = E2 op (E2)

alakúnak tekintjük; az E1 azonban csak egyszer értékelődik ki. A += és -= esetben a bal oldali operandus mutató is lehet, ekkor az (integrális)jobb oldali operandus a 7.4. pontban mondottak szerint alakul át; minden jobb oldali operandus és az összes nem -mutató jellegű bal oldali operandus aritmetikai típusú kell, hogy legyen. A jelenlegi fordítók megengedik mutató értékül adását egésznek, egészt mutatónak, valamint mutatót más típusú mutatónak. Az értékadás tisztán másolási művelet, konverzió nélkül. Ez a fajta használat gépfüggő, és olyan mutatókat eredményezhet, amelyek használatuk során címzési problémákhoz vezetnek. Annyi azonban bizonyos, hogy a 0 állandónak mutatóhoz való hozzárendelése olyan nulla-mutatót eredményez, amely bármilyen objektumot jelölő

mutatótól megkülönböztethető.

7.15. A vessző operátor

vessző_kifejezés:

- kifejezés , kifejezés

A vesszővel elválasztott kifejezéspár balról jobbra értékelődik ki, és a bal oldali kifejezés értéke megegyezik a jobb oldali operandus típusával és értékével. Ez az operátor balról jobbra csoportosít. Olyan szövegkörnyezetben, ahol a vesszőnek speciális jelentése van, pl. függvények aktuális argumentumainak listájában (7.1. pont) és a kezdeti értékek listájában (8.6. pont), az itt ismertetett vessző operátor csak zárójelek között jelenhet meg; pl.

```
f (a, (t = 3 , t + 2), c)
```

-nek három argumentuma van; ezek közül a másodiknak az értéke 5.

8. Deklarációk

A deklarációk segítségével határozzuk meg, hogyan értelmezze a C fordító az egyes azonosítókat; a deklarációk nem feltétlenül jelentenek tárterület-foglalást az azonosító számára. A deklarációk alakja:

deklaráció:

dekl._specifikátorok deklarátorlistaopc;

A deklarátor-listában elhelyezkedő deklarátorok a deklarálandó azonosítókat tartalmazzák. A deklaráció-specifikátorok típus- és tárolásiosztály-meghatározások sorozatából állnak.

dekl._specifikátorok:

- típus-specifikátor dekl._specifikátorokopc
- t.o._specifikátor dekl._specifikátorokopc

A listát az alábbiak szerint következetesen kell megszerkeszteni.

8.1. Tárolásiosztály-specifikátorok

A tárolásiosztály-specifikátorok az alábbiak:

t.o._specifikátor:

- auto
- **static**
- **extern**

- **register**
- typedef

A typedef specifikátor nem foglal tárhelyet, és csak a szintaktikai kényelem kedvéért nevezzük tárolásiosztály-specifikátornak (l. a 8.8. pontot). A különféle tárolási osztályok jelentését a 4. pontban ismertettük. Az auto, **static** és **register** deklarációk definícióként is szolgálnak, amennyiben megfelelő nagyságú tárterület lefoglalását is előidézik. Az **extern** esetben a megadott azonosítók külső definíciójának (10. pont) is szerepelnie kell valahol azon a függvényen kívül, amelyben deklaráltuk őket.

A **register** deklarációt legcélszerűbb olyan **auto** deklarációnak tekinteni, amely még azt is jelzi a fordítónak, hogy a deklarált változókat sűrűn fogjuk használni. Csupán az első néhány ilyen deklarációnak lesz hatása. Ezenkívül csupán néhány típus tárolódik ténylegesen regiszterekben; a PDP-11-en ezek a típusok az int, a **char** és a mutató. Még egy megszorítás vonatkozik a regiszter típusú változókra: nem alkalmazható rájuk az & (címe valaminek) operátor. A regiszterdeklarációk megfelelő használatával kisebb méretű, gyorsabb programokhoz juthatunk, a kódgenerálás továbbfejlesztésével azonban lehet, hogy alkalmazásuk feleslegessé válik. Egy deklarációban legfeljebb egy t. o. -specifikátort lehet megadni. Ha a t.o._specifikátor hiányzik a deklarációból, akkor azt a fordító függvényen belül auto-nak, függvényen kívül **extern**-nek tekinti. Kivétel: a függvények sohasem automatikusak!

8.2. Típus-specifikátorok

A típus-specifikátorok az alábbiak :

típus-specifikátor:

- **char**
- **short**
- int
- **long**
- **unsigned**
- **float**
- **double**
- strukt._vagy_union_specifikátor
- typedef_név

A **long** (hosszú), **short** (rövid) és **unsigned** (előjel nélküli) szavakat jelzőknek tekinthetjük; az alábbi kombinációk fogadhatók el:

- **short int**
- **long int**

- **unsigned int**
- **long float**

Az utóbbi ugyanazt jelenti, mint a **double**. Egyébként egy deklaráción belül legfeljebb egy típus-specifikátor adható meg. Ha a deklarációból hiányzik a típus-specifikátor, akkor a deklarált változót a fordító int-nek tekinti. Struktúrák és **union**ok specifikátoraival a 8.5. pont foglalkozik; a typedef nevekkel történő deklarációkat a 8.8. pont tárgyalja.

8.3. Deklarátorok

A deklarációban megjelenő deklarátorlista deklarátorok vesszőkkel elválasztott sorozata, amelyek mindegyike kezdeti értékkel (k.é.) rendelkezhet.

deklarátorlista:

- k.é._deklarátor
- k.é._deklarátor , deklarátorlista
- k.é._deklarátor:
- deklarátor inicializálóopc

A kezdeti értékekkel a 6.6. pont foglalkozik. A deklarációbeli specifikátorok megadják azoknak az objektumoknak a típusát és tárolási osztályát, amelyekre a deklarátorok vonatkoznak. A deklarátorok szintaxisa:

deklarátor:

- azonosító
- (deklarátor)
- *deklarátor
- deklarátor ()
- deklarátor [állandó_kifejezésopc]

A csoportosítás ugyanolyan, mint a kifejezésekben.

8.4. A deklarátorok jelentése

Minden deklarátort arra vonatkozó állításnak tekinthetünk, hogy ha valamely kifejezésben a deklarátorral megegyező alakú szerkezet jelenik meg, akkor az a megjelölt típusú és tárolási osztályú objektumot fogja eredményezni. Minden deklarátor pontosan egy azonosítót tartalmaz, ez az azonosító az, amelyet deklarálunk. Ha deklarátorként bővítmény nélküli azonosító szerepel, akkor annak típusa az lesz, amit a deklarációt bevezető specifikátor megjelöl. A zárójelek közötti deklarátor azonos a zárójel nélkülivel, de az összetett deklarátorok kötési sorrendje zárójelekkel megváltoztatható (l. a következő példákat). Most képzeljük el a

T D1

deklarációt, ahol T a típus-specifikátor (mint az **int** stb.) és D1 a deklarátor. Tegyük fel, hogy e deklaráció hatására az azonosító típusa ". . .T" lesz, ahol ". . " üres, ha D1 csupán sima azonosító (tehát x típusa **int** x-ben egyszerűen int). Ha viszont D1 alakja *D akkor az általa tartalmazott azonosító típusa ". . .mutató T-re". Ha D1 alakja

D()

akkor az általa tartalmazott azonosító típusa ". . . függvény, amely T-t ad vissza". Ha D1

D[állandó_kifejezés]

vagy

D[]

alakú, akkor az általa tartalmazott azonosító típusa "T . . . tömbje". Az első esetben az állandó kifejezés olyan kifejezés, amelynek értéke fordítási időben meghatározható és amelynek típusa **int** Az állandó kifejezések pontos definíciója a 15. pontban található.) Ha több . . .tömbje specifikáció egymással szomszédos, akkor többdimenziós tömb keletkezik; a tömbhatárokat rögzítő állandó kifejezések csupán a sorozat első tagjánál hiányozhatnak. Ez az elhagyás akkor hasznos, ha külső tömbről van szó, és a tárfoglalást előidéző definíció máshol szerepel.

Az első állandó kifejezés akkor is elhagyható, ha a deklarátort kezdeti érték követi. Ilyenkor a fordító a méretet a megadott kezdeti értékek számából számítja ki. Tömböt az alaptípusok valamelyikéből, mutatókból, struktúrákból, **union**okból vagy más tömbökből (többdimenziós tömböt generálva) alkothatunk. A fenti szintaxissal definiált lehetőségek közül nem mindegyik megengedett. A megszorítások a következők : függvények nem adhatnak vissza tömböket, struktúrákat, **union**okat vagy függvényeket, de visszaadhatnak ilyeneket megcímző mutatókat; függvényekből nem képezhető tömb, de létezik függvényeket megcímző mutatókból képzett tömb.

Hasonlóképpen, a struktúrák és **union**ok sem tartalmazhatnak függvényt, legfeljebb függvényt megcímző mutatót. Például

```
int i, *ip, f (), *fip (), (*pfi) ()
```

deklarálja az i egészt, az ip egészt megcímző mutatót, az egészt visszaadó f függvényt, az egészt megcímző mutatót visszaadó fip függvényt és a pfi mutatót, amely egy egészt visszaadó függvényre mutat.

Különösen hasznos ha a két utolsót hasonlítjuk össze . A *fip() kötési sorrendje *(fip()), így a deklaráció azt írja elő, ill. egy kifejezésben előforduló ilyen szerkezet azt váltja ki, hogy a fip függvény meghívása után a (mutatójellegű) eredményen keresztüli indirekcióval egy egész álljon elő. A (*pfi)() deklarátorban (vagy a szerkezetet felhasználó kifejezésekben) a plusz zárójelek szükségesek: azt jelzik, hogy a függvényt megcímző mutatón keresztüli indirekció függvényt eredményez, amely meghívása után egészt ad vissza. Másik példaként

```
float fa[17], *afp[17];
```

egy **float** számokból álló tömböt és egy **float** számokat megcímző mutatókból álló tömböt deklarál. Végezetül

```
static int x3d[3][5][7];
```

egészek statikus, háromdimenziós tömbjét deklarálja, amelynek mérete 3 * 5 * 7. Részleteiben nézve x3d háromelemű tömb; minden elem öt tömböt tartalmaz; az utóbbiak mindegyike 7 darab egészből áll. Az x3d, x3d[i], x3d[i][j], x3d[i][j][k] alakok bármelyike előfordulhat valamely kifejezésben. Az első három tömb típusú, az utolsó típusa int.

8.5. Struktúra- és union deklarációk

A struktúra névvel ellátott tagok sorozatát tartalmazó objektum. Minden tag tetszőleges típusú lehet. Az **union** olyan objektum, amely adott időpillanatban több lehetséges tag bármelyikét tartalmazhatja. A struktúra- és az unionspecifikátorok azonos alakúak.

strukt._vagy_union_specifikátor:

- strukt._vagy_union { strukt._dekl._lista}
- strukt._vagy_union azonosító {strukt._dekl._lista}
- strukt._vagy_union azonosító
- strukt._vagy_union:
- **struct**
- **union**

A struktúradeklarátor-lista a struktúra vagy **union** tagjaira vonatkozó deklarációk felsorolása:

strukt._dekl._lista:

- strukt._deklaráció
- strukt._deklaráció strukt._dekl._lista
- strukt._deklaráció:
- típus_specifikátor strukt._deklarátor_lista
- strukt._deklarátor_lista:

- strukt._deklarátor
- strukt._deklarátor , strukt._deklarátor_lista

Közönséges esetben a strukt. deklarátor egyszerűen a struktúra vagy **union** valamely tagjának deklarátora. A struktúra tagjai adott számú bitet is tartalmazhatnak. Az ilyen tag neve mező (field), hosszát a névtől kettőspont választja el.

strukt. _deklarátor:

- deklarátor
- deklarátor : állandó_kifejezés
- : állandó_kifejezés

A struktúrán belül a deklarált objektumok címei a deklarációkban balról jobbra haladva növekednek. A struktúra minden nem-mező tagja a típusának megfelelő címhatáron kezdődik, így a struktúrában név nélküli lyukak helyezkedhetnek el. A mező jellegű tagok gépi egészekben helyezkednek el, szóhatárokon nem nyúlnak át. Az a mező, amely nem fér el egy szóban még fennmaradt helyen, a következő szóba kerül. A mező nem lehet szélesebb, mint a szó. Mezők hozzárendelése PDP-11-en jobbról balra, más gépeken balról jobbra történik.

A deklarátor nélküli, csupán kettőspontot és a szélességet tartalmazó struktúradeklarátor olyan név nélküli mezőt jelöl ki, amelyet kívülről előírt elrendezéseknek megfelelő kitöltésre használhatunk. Speciális esetben a 0 szélességű név nélküli mező a következő mező szóhatárra történő illesztését írja elő. A "következő mező" feltehetően tényleg mező, nem pedig közönséges struktúratag, mivel az utóbbi esetben ez az illesztés automatikusan megtörténne. A nyelv nem ír elő korlátozást a mezőként deklarált objektumok típusára vonatkozólag, a megvalósításoktól azonban nem várjuk el, csak az egész típusú mezők támogatását. Sőt, még az **int** mezőket is előjel nélkülinek tekinthetik. A PDP-11-en a mezőknek nincs előjelük, és csak egész értékűek lehetnek.

Egyetlen megvalósításban sincsenek mezőkből képzett tömbök, továbbá a mezőkre az & címoperátor sem alkalmazható, vagyis nincsenek mezőket megcímző mutatók sem. Az **union**t olyan struktúrának képzelhetjük, amelynek tagjai a 0 relatív címen kezdődnek, és amelynek mérete elegendően nagy ahhoz, hogy bármelyik tagját tartalmazhassa. Az **union** egyszerre legfeljebb egy tagját tartalmazhatja. A második alakú struktúra- vagy unionspecifikátor, vagyis a

```
struct azonosító {strukt._dekl._lista}
union azonosító {strukt._dekl._lista}
```

egyike, az azonosítót a lista által meghatározott struktúra struktúracímkéjeként (vagy unioncímkéjeként) deklarálja. Az ezt követő deklarációkban azután a specifikátor harmadik alakja, a

struct azonosító

union azonosító

alakok egyike használható. A struktúracímkék lehetővé teszik önhivatkozó struktúrák definiálását; megengedik, hogy a deklaráció hosszú részét csupán egyszer adjuk meg és több alkalommal használjuk. Tilos olyan struktúrát vagy **union**t deklarálni, amelyben saját maga előfordul, de a struktúra vagy **union** tartalmazhat saját magát megcímző mutatót! A tagok és címkék nevei megegyezhetnek a közönséges változók neveivel. A címkék és a tagok nevének azonban egymástól el kell térniük!

Két struktúrának lehet közös kezdeti tagsorozata, azaz ugyanaz a tag két különböző struktúrában is megjelenhet, ha mindkettőben azonos a típusa és ha az összes megelőző tag is mind a kettőben azonos. (A fordító tulajdonképpen csak azt ellenőrzi, hogy a két különböző struktúrában előforduló név típusa és relatív címe megegyezik-e, de ha a megelőző tagok különböznek, akkor a szerkezet nem gépfüggetlen.) A struktúradeklaráció egyszerű példája:

```
struct tnode {
    char tword [20];
    int count;
    struct tnode * left;
    struct tnode *right;
};
```

amely 20 karakterből álló tömböt, egy egészt és két, hasonló struktúrát megcímző mutatót tartalmaz. E deklaráció megadása után a

```
struct tnode s, *sp;
```

deklaráció szerint s a megadott jellegű struktúra lesz, és sp az ilyen jellegű struktúrát megcímző mutató. Ezeknek a deklarációknak az alapján az

```
sp->count
```

kifejezés annak a struktúrának a count nevű mezőjére mutat, amelyre az sp utal;

```
s.left
```

az s struktúra bal oldali részfájának mutatójára vonatkozik, míg

```
s.right->tword [0]
```

az s struktúra jobb oldali részfája tword nevű tagjának első karakterére utal.

8.6. Inicializálás

A deklarátor megadhatja a deklarált azonosító kezdeti értékét. Az inicializálót = előzi meg, és kapcsos zárójelek közé zárt kifejezést vagy értéklistát tartalmaz.

inicializáló:

- = kifejezés
- = { inicializáló_lista }
- = { inicializáló_lista ,}

inicializáló_lista:

- kifejezés
- inicializáló_lista , inicializáló_lista
- (inicializáló_lista)

A statikus vagy külső változók inicializálóiban kizárólag állandó kifejezések (l. a 15. pontot), vagy pedig olyan kifejezések szerepelhetnek, amelyek valamelyik korábban deklarált változó címére redukálhatók (az alábbitól egy állandó kifejezéssel való címeltolás is lehetséges). Az automatikus és regisztervváltozók esetében tetszőleges inicializálás lehetséges állandók, korábban deklarált változók és függvények bevonásával.

Inicializálatlan statikus és külső változók kezdeti értéke garantáltan nulla; az inicializálatlan automatikus és regisztervváltozókban pedig induláskor biztos hulladék van. Ha az inicializálót skalár mennyiségre (mutatóra vagy aritmetikai típusú objektumra) alkalmazzuk, tartalma egyetlen, esetleg kapcsos zárójelek közötti kifejezés.

Az objektum kezdeti értékét a gép a kifejezés alapján számítja ki; a konverziók ugyanaz ok, mint értékadásnál. Ha a deklarált változó aggregátum (struktúra vagy tömb jellegű összetett mennyiség), akkor az inicializáló az aggregátum tagjainak kapcsos zárójelek közötti, vesszőkkel elválasztott listáját tartalmazza.

Az inicializálókat az indexek vagy tagok növekvő sorrendjében adjuk meg. Ha az aggregátum részaggregátumokat tartalmaz, ugyanez a szabály vonatkozik rekurzív módon az aggregátum tagjaira. Ha a listában kevesebb inicializáló van, mint ahány tagja van az aggregátumnak, akkor az aggregátum nullákkal töltődik ki.

Unionok és automatikus aggregátumok inicializálása nem megengedett! A kapcsos zárójeleket a következő módon hagyhatjuk el. Ha az inicializáló bal oldali kapcsos zárójellel kezdődik, akkor a rá következő, vesszőkkel elválasztott inicializálólista az aggregátum tagjait inicializálja; Ha, ha itt több

inicializáló van, mint tag. Ha azonban az inicializáló nem bal oldali kapcsos zárójellel kezdődik, akkor a fordítóprogram a listából csupán az aggregátum tagjainak megfelelő számú elemet vesz figyelembe; a listában fennmaradó tagok annak az aggregátumnak a következő elemét fogják inicializálni, amelynek a szóban forgó aggregátum a része.

Végül megemlítjük, hogy a **char** tömbök röviden, karakterláncokkal inicializálhatók. Ez esetben a lánc egymást követő karakterei a tömb egyes elemeit inicializálják. Inicializálási példák:

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

az x-et olyan egydimenziós tömbként deklarálja és inicializálja, amelynek három eleme van, mivel méretet nem adtunk meg és három inicializáló van.

```
float y [4][3] ={
    {1, 3, 5},
    {2, 4, 6},
    {3, 5, 7},
};
```

teljes zárójelezett inicializálás: 1 , 3 és 5 az y[0] tömb első sorát, mégpedig az y[0][0], y[0][1] és y[0] [2] elemeket inicializálják. A következő két sor hasonló módon inicializálja y[1]-et és y[2]-t. Az inicializáló túl hamar ér véget, és ezért y[3] 0 -val inicializálódik. Pontosan ugyanezt az eredményt értük volna el

```
float y [4][3] ={
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

megadásával. y inicializálója bal oldali kapcsos zárójellel kezdődik, de y[0]-é nem, így a gép a listából három elemet használ fel. Hasonlóképpen a következő három y[1]-é, az azt követő három pedig y[2]-é lesz. Ugyanígy, ;

```
float y [4][3] ={
    {1}, {2}, {3}, {4}
};
```

a (kétdimenziós tömbnek tekintett) y első oszlopát inicializálja és a többi elemet 0 értékűnek hagyja meg. Végezetül

```
char msg [] = "Szintaktikai hiba a %s-edik sorban \n";
```

olyan karaktertömböt mutat, amelynek elemeit karakterlánccal inicializáltuk.

8.7. Típusnevek

Két összefüggésben (típusmódosító szerkezettel végzett explicit típuskonverzió esetén és a sizeof argumentumaként) kell valamilyen adattípus nevét megadnunk. Ez típusnév használatával történik, ami lényegében egy adott típusú objektum olyan deklarációja, amelyből hiányzik az objektum neve.

típus_név:

- típus_specifikátor absztrakt_deklarátor

absztrakt_deklarátor:

- üres
- (absztrakt_deklarátor)
- *absztrakt_deklarátor
- absztrakt_deklarátor ()
- absztrakt_deklarátor [állandó_kifejezésopc]

A kétértelműség elkerülése érdekében az

(absztrakt_deklarátor)

szerkezetben az absztrakt deklarátor nem lehet üres. E megszorítás figyelembevételével egyértelműen azonosítható az absztrakt-deklarátorban az a hely, ahol az azonosító megjelenne, ha a szerkezet egy deklaráción belüli deklarátor lenne. A megnevezett típus ekkor ugyanaz lesz, mint a hipotetikus azonosító típusa. Pl.

```
int
int *
int *[3]
int (*) [3]
int * ()
int (*) ()
```

sorban megnevezi az egész, egészt megcímző mutató, 3 darab egész-mutatóból álló tömb, 3 egészből álló tömböt megcímző mutató, egészt megcímző mutatót visszaadó függvény és az egészt visszaadó függvényt megcímző mutatótípusokat.

8.8. Typedef

Az olyan deklarációk, amelyeknek a tárolási osztálya typedef, nem tárterületet definiálnak, hanem olyan azonosítókat, amelyeket a későbbiekben úgy használhatunk, mintha az alapvető vagy a leszármaztatott típusokat megnevező kulcsszavak lennének:

typedef_név:

- azonosító

A typedef-et tartalmazó deklaráció érvényességi tartományán belül minden ott előforduló deklarátor részeként megjelenő azonosító szintaktikusan egyenértékű lesz azzal a típuskulcsszóval, amely a 8.4. pontban leírt módon megnevezi az azonosítóhoz társított típust. Pl.

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im;}complex;
```

után a

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

szerkezetek mindegyike megengedett deklaráció; a distance típusa int, a metricp-é int-et megcímző mutató, a z-é pedig a megadott struktúra. zp az ilyen struktúrát megcímző mutató. A typedef nem teljesen új típusokat vezet be, csupán más módon is megadható típusok szinonimáit. Így a fenti példában distance pontosan ugyanolyan típusú, mint minden más **int** objektum.

9. Utasítások

Az utasítások egymást követően, sorban hajtódnak végre, az ettől való eltérést külön jelezzük.

9.1. A kifejezés utasítás

A legtöbb utasítás kifejezés jellegű; ezek alakja:

kifejezés;

A kifejezés jellegű utasítások legtöbbször értékadások vagy függvényhívások.

9.2. Az összetett utasítás vagy blokk

Annak érdekében, hogy ott, ahol elvileg csak egy utasítás helyezhető el, több utasítás is használható legyen, rendelkezésre áll az összetett utasítás (más szóval blokk).

összetett_utasítás:

- { deklarációlistaopc utasításlistaopc}

deklarációlista:

- deklaráció

deklaráció deklarációlista

- utasításlista:
- utasítás
- utasítás utasításlista

Ha a deklarációlistában előforduló bármelyik azonosítót már korábban deklaráltuk, a külső deklaráció a blokk végrehajtásának időtartamára érvényét veszti, majd annak befejeztével visszanyeri hatályát. Az **auto** és **register** változók bármilyen inicializálása minden alkalommal újra megtörténik, amikor a vezérlés a blokkba felülről belép.

Jelenleg lehetséges (de helytelen gyakorlat) a blokk belsejébe való ugratás; ez esetben az inicializálások elmaradnak. A **static** változók kezdeti értékének beállítása csupán egyszer, a program végrehajtásának kezdetén történik meg. A blokkon belül az **extern** deklarációk hatására nincs tárfoglalás, így ezek inicializálása nem megengedett.

9.3. A feltételes utasítás

```
if (kifejezés)
    utasítás

if (kifejezés)
     utasítás
else
    utasítás
```

A gép mindkét esetben kiértékeli a kifejezést, és ha értéke nemnulla, az első alutasítást hajtja végre. A második esetben, ha a kifejezés értéke 0, a második alutasítást hajtja végre. Az **else**-vel kapcsolatos szokásos kétértelműséget a C úgy oldja fel, hogy az else az utoljára talált else nélküli **if**-hez kötődik.

9.4. A while utasítás

A **while** utasítás alakja:

```
while (kifejezés)
    utasítás
```

Az alutasítás végrehajtása mindaddig ismétlődik, amíg a kifejezés értéke nemnulla marad. A vizsgálat mindig az utasítás egyes végrehajtásai előtt történik.

9.5. A do utasítás

A do utasítás alakja

```
do
     utasítás
while (kifejezés);
```

Az alutasítás végrehajtása mindaddig ismétlődik, amíg kifejezés értéke nullává nem válik. A vizsgálat mindig az utasítás egyes végrehajtásai után történik.

9.6. A for utasítás

A **for** utasítás alakja:

```
for (1._kifejezésopc; 2._kifejezésopc; 3._kifejezésopc)
     utasítás
```

Ez az utasítás egyenértékű az

```
1._kifejezés;
while (2._kifejezés) {
     utasítás
    3._kifejezés;
}
```

alakkal. Eszerint az első kifejezés a ciklust inicializálja; a második azt a vizsgálatot határozza meg, amely minden iterációt megelőz, és a vezérlés kilép a ciklusból, ha a kifejezés nullává válik; a harmadik kifejezés gyakran az egyes iterációk után végrehajtandó inkrementálást határozza meg. A kifejezések bármelyike, vagy akár mindegyik elhagyható.

Ha a 2. kifejezés hiányzik, akkor a megfelelő **while** utasításból **while**(1) lesz; a többi hiányzó kifejezés egyszerűen elmarad az előbbi kifejtett formából.

9.7. A switch utasítás

A **switch** utasítás hatására a megadott kifejezés értékétől függően a vezérlés több utasítás valamelyikére adódik át. Alakja:

```
switch (kifejezés)
utasítás
```

A kifejezésben megtörténnek a szokásos aritmetikai konverziók, de az eredménynek int-nek kell lennie. Az utasítás általában összetett. A **switch** utasításon belül előforduló bármelyik utasítás megcímkézhető egy vagy több **case** előtaggal az alábbi módon :

```
case állandó_kifejezés:
```

ahol az állandó kifejezés **int** kell, hogy legyen. Ugyanazon a **switch**-en belül két **case** állandónak nem lehet egyforma értéke. Az állandó kifejezések pontos definícióját a 15. pont tartalmazza. Legfeljebb egy darab

```
default:
```

alakú utasítás-előtag is előfordulhat a **switch** utasításban. A **switch** utasítás végrehajtása során a gép kiértékeli a benne előforduló kifejezést és összehasonlítja minden egyes **case** állandóval. Ha a **case** állandók valamelyike megegyezik a kifejezés értékév el, a vezérlés az illeszkedő **case** előtagot követő utasításra adódik át. Ha egyik állandó sem egyezik meg a kifejezés értékével, és szerepel a **default** előtag, akkor a program végrehajtása az ezt követő utasításon folytatódik. Ha egyik **case** sem illeszkedik és nincs **default**, akkor a gép a **switch**-ben előforduló utasítások közül egyiket sem hajtja végre. A **case** és **default** előtagok egymagukban nem változtatják meg a vezérlés menetét, amely zavartalanul végighalad ezeken az előtagokon.

A **switch**ből való kilépésre vonatkozólag l. a **break** utasítást a 9.8. pontban. A **switch** tárgyát képező

utasítás legtöbbször összetett. Deklarációk szerepelhetnek ennek az utasításnak a fejében, de az automatikus és regisztervált zók inicializálásai hatástalanok.

9.8. A break utasítás

A

```
break;
```

utasítás hatására befejeződik a **break**-et körülvevő legbelső **while**, **do**, **for** vagy **switch** utasítás végrehajtása; a vezérlés a befejezett utasítást követő utasításra adódik át.

9.9. A continue utasítás

A

```
continue;
```

utasítás hatására a vezérlés a **continue**-t körülvevő legbelső **while**, **do** vagy **for** utasítás ciklusfolytató részére adódik át, vagyis a ciklus végére. Pontosabban, a

```
while (...) {
    do {
         for (...) {
              ...    ...    ...
              contin: ;
              contin: ;
              contin: ;
         }
    } while (...);
}
```

utasítások mindegyikében a **continue** utasítás egyenértékű a **goto** contin-nel. A contin: után nulla utasítás szerepel, (l. a 9.13. pontot).

9.10. A return utasítás

A függvény a hívójához a **return** utasítás segítségével tér vissza, amelynek lehetséges alakja:

```
return ;
return kifejezés;
```

Az első esetben a visszaadott érték határozatlan. A második esetben a kifejezés értéke kerül vissza a függvény hívójához. Szükség esetén az értékadáshoz hasonlóan a kifejezés olyan típusúvá alakul át, mint amilyen típusú függvényben előfordul. A függvény végének átlépése azonos a visszatérési érték nélküli **return**-nel.

9.11. A goto utasítás

A vezérlés feltétel nélkül a

```
goto azonosító;
```

utasítás segítségével adható át. Az azonosító az éppen végrehajtott függvényen belül elhelyezett címke (l. a 9.12. pontot) kell, hogy legyen.

9.12. A címkézett utasítás

Bármelyik utasítást megelőzhetik az

```
azonosító:
```

alakú előtagok, amelyek az azonosítót címkeként deklarálják. A címke egyedül a **goto** célpontjaként szolgál. A címke érvényességi tartománya az a függvény, amelyben előfordul, kivéve azokat az alblokkokat, amelyekben ugyanezt az azonosítót újra deklarálták (l. a 11. pontot).

9.13. A nulla utasítás

A nulla utasítás alakja:

```
;
```

A nulla utasítás hordozhat pl. címkét közvetlenül valamely összetett utasítás }-e előtt, vagy pedig a **while**-hoz hasonló valamelyik ciklusutasítás számára üres ciklustörzset képezhet.

10. Külső definíciók

A C program külső (**extern**al) definíciók sorozatát tartalmazza. A külső definíció a változót **extern** (ez az alapértelmezés) vagy **static** tárolási osztályúnak és megadott típusúnak deklarálja. A típus-specifikátor (l. a 8.2. pontot) lehet üres, ebben az esetben a típust int-nek tekintjük. A külső definíciók érvényességi tartománya annak az állománynak a végéig tart, amelyben deklarálták őket; hasonlóképpen a deklarációk is az állomány végéig érvényesek. A külső definíciók szintaxisa ugyanaz, mint az összes deklarációé, azzal a különbséggel, hogy a függvényeket csak ezen a szinten lehet definiálni.

10.1. Külső függvénydefiníciók

A függvénydefiníciók alakja:

függvénydefiníció:

- dekl._specifikátorokopc függvény deklarátor függvénytörzs

A deklarációspecifikátorok közül tárolásiosztály-specifikátorként csupán az **extern** és a **static** megengedett; a kettő közötti különbségre nézve l. a 11.2. pontot. A függvény-deklarátor hasonló a "függvény, amely...-t ad vissza" jellegű deklarátorhoz, azzal a különbséggel, hogy megadja a definiált függvény formális paramétereinek listáját.

függvénydeklarátor:

- deklarátor (paraméterlistaopc)

paraméterlista:

- azonosító
- azonosító , paraméterlista

A függvénytörzs alakja:

függvénytörzs:

- deklarációlista összetett_utasítás

A paraméterlistabeli azonosítók - és csakis ezek deklarálhatók a deklarációlistában. Az olyan azonosítót, amelynek típusát nem adtuk meg, a fordítás int-nek tekinti. Az egyetlen megadható tárolási osztály a **register**; ha ez szerepel, akkor a neki megfelelő aktuális paraméter, amennyiben lehetséges, a függvény végrehajtásának kezdetén valamelyik regiszterbe kerül. Egyszerű példa a teljes függvénydefinícióra:

```
int max (int a, int b, int c)
{
    int m;
    m = (a > b) ? a : b;
    return ((m > c) ? m : c);
}
```

Itt az **int** a típus-specifikátor; max(a, b, c) a függvénydeklarátor;

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

a formális paraméterek deklarációinak listája; { . . } az utasítás programkódját megadó blokk. A C az összes **float** típusú aktuális paramétert **double**-lá alakítja át, így a **float**-nak deklarált formális paraméterek deklarációi is **double**-lá módosulnak. Továbbá, mivel a tömbre történő hivatkozás bármilyen összefüggésben különösen aktuális paraméterként) olyan mutatót jelent, amely a tömb első elemére mutat, a ". . . tömbje" alakban deklarált formális paraméterek deklarációi "mutató . . .-ra" alakúra módosulnak. Végezetül, mivel a struktúrák, **union**ok és függvények nem adhatók át függvénynek, értelmetlen dolog formális paramétereket struktúrának, union-nak vagy függvénynek deklarálni (az ilyen objektumokat megcímző mutatók természetesen megengedettek).

10.2. Külső adatdefiníciók

A külső adatdefiníciók alakja:

adatdefiníció:

- deklaráció

Az ilyen adatok tárolási osztálya lehet **extern** (ez az alapértelmezés) vagy **static**, de nem lehet auto, sem pedig **register**.

11. Az érvényességi tartomány szabályai

Nem szükséges az egész C programot egyszerre fordítani: a program forrásszövege több állományban tárolható, és könyvtárakból előre lefordított rutinokat lehet betölteni. A program függvényei közötti kommunikáció akár explicit hívásokkal, akár külső adatokon keresztül megvalósítható. Ennek következtében kétféle érvényességi tartományról kell beszélnünk: először is arról, amit az azonosító lexikális érvényességi tartományának nevezünk, és ami lényegében a programnak az a része, amelyben a definiálatlan azonosító ("undefined identifier ") hibaüzenet előfordulása nélkül használhatjuk, másodszor pedig a külső azonosítókhoz tartozó érvényességi tartományról, amelyre az a szabály jellemző, hogy az ugyanarra a külső azonosítóra vonatkozó hivatkozások ugyanarra az objektumokra

való hivatkozásokat jelentenek.

11.1. Lexikális érvényességi tartomány

A külső definíciókban deklarált azonosítók lexikális érvényességi tartománya a definícióktól az őket tartalmazó forrásállomány végéig tart. A formális paraméterként előforduló azonosítók érvényességi tartománya az a függvény, amelyhez tartoznak. A blokkok fejében deklarált azonosítók érvényességi tartománya a blokk végéig terjed. A címkék érvényességi tartománya az egész függvény, amelyben előfordulnak.

Mivel az ugyanarra a külső azonosítóra utaló összes hivatkozás ugyanarra az objektumra vonatkozik (l. a 11.2. pontot), a fordítóprogram ellenőrzi, hogy ugyanannak a külső azonosítónak az összes deklarációja kompatibilis-e; valójában ezek érvényességi tartománya kiterjed az egész állományra, amelyben előfordulnak. Minden esetben fennáll azonban, hogy ha egy azonosító explicit módon egy blokk - akár függvényt alkotó blokk - fejében deklarálunk, akkor annak végéig az illető azonosító összes, a blokkon kívül előforduló deklarációja felfüggesztődik. Emlékezzünk arra is (l. a 8.5. pontot), hogy egyrészt a közönséges változókhoz, másrészt a struktúra-, ill. **union**tagokhoz és – címkékhez kapcsolódó változók két külön osztályt alkotnak, amelyek között nincs ütközés. A tagokra és címkékre ugyanazok az érvényességi tartomány szabályok vonatkoznak, mint a többi azonosítókra. A typedef nevek ugyanabba az osztályba tartoznak, mint a közönséges azonosítók, belső blokkokban újradeklarálhatók, de a belső deklarációban a típust explicit módon meg kell adni:

```
typedef float distance;
 . . .
{
    auto int distance;
 . . .
```

Az int-nek szerepelnie kell a második deklarációban, különben a fordító deklarátor nélküli, distance típusú deklarációnak tekintené.

11.2. A külső azonosítók érvényességi tartománya

Ha egy függvény **extern**-ként deklarált azonosítóra hivatkozik, akkor a teljes programot alkotó állományok, ill. könyvtárak közül valamelyikben szerepelnie kell az azonosító külső definíciójának. Egy adott programban előforduló minden olyan függvény, amely ugyanarra a külső azonosítóra hivatkozik, egyben ugyanarra az objektumra is hivatkozik, ezért ügyelnünk kell arra, hogy a definícióban megadott típus és méret kompatibilis legyen minden egyes, az adatokra hivatkozó függvényben megadott típussal és méret tel. Az **extern** kulcsszó a külső definícióban azt jelzi, hogy a deklarált azonosítók számára szükséges tárhelyet valamely másik állományban foglaljuk le.

Így több állományból álló programban **extern** specifikátor nélküli külső adatdefiníció egy és csakis egy állományban szerepelhet. Az összes többi állományban, ahol külső definícióval kívánjuk valamelyik

változót megadni, a definícióban szerepelnie kell az **extern**-nek. Az azonosító csak abban a deklarációban inicializálható, ahol a tárhely lefoglalás a történt. A legfelső szinten külső definíciókban **static**-ként deklarált azonosítók más állományokban nem láthatók. Függvények is deklarálhatók **static**-ként.

12. A fordítónak szóló vezérlősorok

A C fordító része egy előfeldolgozó program, amely makrohelyettesítésre, feltételes fordításra és megadott nevű állományok beiktatására képes. Az előfeldolgozó a # karakterrel kezdődő sorokat értelmezi. E sorok szintaxisa független a nyelv többi részétől , bárhol előfordulhatnak, és (érvényességi tartománytól függetlenül) hatásuk az adott forrásprogram-állomány végéig tart.

12.1. Szintaktikai egységek helyettesítése

A

#define azonosító szint._egységek_karakterlánca

alakú fordító vezérlő sor (vigyázat: nincs záró pontosvessző) hatására az előfeldolgozó az azonosító minden további előfordulását a szintaktikai egységek megadott karakterláncával helyettesíti. A

#define azonosító(azonosító, . . .,azonosító)

szint._egységek_karakterlánca

alakú sor, ahol az első azonosító és a (között nincs szóköz, argumentumokkal ellátott makrodefiníció. Az első azonosítónak azon további előfordulásait, ahol az azonosítót (, szintaktikai egységek vesszőkkel elválasztott sorozata és egy) követi, a definícióban megadott szintaktikai egység karakterlánccal helyettesíti. A definíció formális paraméterlistájában említett azonosító összes előfordulása helyére a hívás hatására a megfelelő szintaktikai egység karakterlánc kerül. A hívás aktuális argumentumai vesszőkkel elválasztott szintaktikai egység karakterláncok, azonban az idézőjelek közötti vagy zárójelekkel védett vesszők nem argumentumelválasztók. A formális és aktuális paraméterek darabszáma egyenlő kell, hogy legyen. Karakterláncon vagy karakterállandón belüli szövegre nem vonatkozhat a helyettesítés. A helyettesítő karakterláncot (mindkét változatban) újra átvizsgálja az előfeldolgozó, hogy megtalálja az esetleges további definiált azonosítókat. A hosszú definíciók mindkét alakban új sorban folytathatók oly módon, hogy a folytatandó sor végére \-t írunk.

A #define használat_ leginkább a hangsúlyozott funkciójú állandók definiálására előnyös, pl.:

```
#define TABSIZE 100
int table[TABSIZE];
```

Az

```
#undef azonosító
```

alakú vezérlősor hatására megszűnik az azonosító előfeldolgozó-definíciója.

12.2. Állományok beiktatása

Az

```
#include "állománynév"
```

alakú vezérlősort az előfeldolgozó program az állománynév nevű állomány teljes tartalmával helyettesíti. A megnevezett állomány keresése az eredeti forrásállomány katalógusában kezdődik, majd sorban, szabványos helyeken folytatódik. Megadhatjuk az

```
#include <állománynév>
```

alakú vezérlősort is, amikor a keresés csak a szabványos helyeken történik, és nem terjed ki a forrásállomány katalógusára. Az #include-ok egymásba skatulyázhatók.

12.3. Feltételes fordítás

Az

#if állandó_kifejezés

alakú fordításvezérlő sor ellenőrzi, hogy az állandó kifejezés (l. a 15. pontban) értéke nemnulla-e. Az

```
#ifdef azonosító
```

alakú vezérlősor megvizsgálja, hogy az azonosító pillanatnyilag definiálva van-e az előfeldolgozóban, azaz szerepelt-e már valamelyik #define vezérlősorban. Az

```
#ifndef azonosító
```

alakú vezérlősor azt ellenőrzi, hogy az azonosító pillanatnyilag definiálatlan-e az előfeldolgozóban. Mindhárom alakot tetszőleges számú, esetleg az

```
#else
```

vezérlősort is tartalmazó sor, majd az

```
#endif
```

vezérlősor követi. Ha a vizsgált feltétel igaz, akkor az #else és az #endif közötti sorok hatástalanok. Ha a vizsgált feltétel hamis, akkor az ellenőrzés és az #else vagy annak hiányában a #endif közötti sorok lesznek hatástalanok. E szerkezetek egymásba skatulyázhatók.

12.4. Sorvezérlés

Egyéb, C programokat létrehozó előfeldolgozók szempontjából hasznos a

#line állandó_azonosító

alakú sor. Hatására - diagnosztikai célokból a fordító azt hiszi, hogy a következő forrássor sorszáma az állandó által megadott érték, és a pillanatnyi bemeneti állomány az, amelyet az azonosító megnevez. Azonosító hiányában a megnevezett állománynév nem változik.

13. Implicit deklarációk

A deklarációban nem mindig kell a tárolási osztályt és az azonosítók típusát is megadnunk. A tárolási osztályt külső definíciókban és formális paraméterek, ill. a struktúratagok deklarációiban a szövegkörnyezet határozza meg. Függvényen belüli deklarációban, ha a tárolási osztályt megadtuk, de a típust nem, az azonosító feltételezés szerint int; ha típus szerepel, de tárolási osztály nem, akkor az azonosítót **auto**-nak tekinti a fordító. Az utóbbi szabály alól kivételek a függvények, mivel az **auto** függvényeknek nincs értelmük (a C nem képes kódot generálni a verembe); ha valamely azonosító típusa "függvény, amely ...-t ad vissza", akkor az implicite **extern**-nek deklarálódik. Kifejezésekben az olyan, még nem deklarált azonosítót, amelyet (követ, a szövegkörnyezet alapján a fordító int-et visszaadó függvénynek tekinti.

14. Még egyszer a típusokról

Ez a szakasz azokat a műveleteket foglalja össze, amelyeket csak bizonyos típusú objektumokon lehet elvégezni.

14.1. Struktúrák és unionok

Struktúrákkal és **union**okkal két dolgot tehetünk: megnevezhetjük valamelyik tagjukat (a . operátorral), vagy előállíthatjuk a címüket (az egyoperandusú &-tel). Az egyéb műveletek, mint a struktúrák vagy unionok valamihez történő hozzárendelése, paraméterként való átadása, vagy nekik való értékadás hibaüzenetet von maga után. Reméljük, hogy a jövőben a C, ha egyebekkel nem is, de ezekkel a műveletekkel kiegészül. A 7.1. pontban mondottak szerint a (. vagy -> segítségével történő) direkt vagy indirekt struktúrahivatkozásban a jobb oldalon álló névnek a bal oldali kifejezés által megnevezett vagy megcímzett struktúra tagjának kell lennie. A rugalmas típuskezelés érdekében ezt a megkötést a fordító követeli meg szigorúan. Valójában a . előtt bármilyen balérték megengedett, és a fordító feltételezi, hogy ez a balérték olyan alakú struktúra, mint amilyen a jobb oldali név tagja. A -> előtti kifejezésnek ugyancsak mutatónak vagy egésznek kell lennie.

Ha a kifejezés mutató, akkor feltételezés szerint arra a struktúrára mutat, amelyiknek a jobb oldalon álló név tagja. Ha a kifejezés egész típusú, akkor a fordító a megfelelő struktúra (gépi tárolási egységekben kifejezett) abszolút címének tekinti. Az ilyen konstrukciók nem gépfüggőek.

14.2. Függvények

Függvénnyel csupán két műveletet végezhetünk: meghívhatjuk vagy előállíthatjuk a címét. Ha a függvény neve kifejezésen belül nem valamely hívás függvénynév-pozícióján jelenik meg, akkor a függvényt megcímző mutató jön létre. Ha tehát egy függvényt egy másiknak akarunk átadni, azt mondhatjuk, hogy:

```
int f ();
 . . .
g (f);
```

Ekkor a g definíciója

```
g (funcp)

int (*funcp) ();

{
  . . .
  (*funcp) ();
  . . .
}
```

lehet. Jegyezzük meg, hogy f-et a hívó rutinban explicit módon deklarálni kell, mivel g (f)-beli

előfordulását nem követte (.

14.3. Tömbök, mutatók és indexelés

Minden alkalommal, amikor tömb típusú azonosító jelenik meg egy kifejezésben, az azonosító a tömb első elemét megcímző mutatóvá alakul át. E konverzió miatt a tömbök nem balértékek. Definíció szerint a [ ] indexoperátor értelmezése olyan, hogy

```
E1 [E2]
```

azonos

```
*((E1)+(E2))
```

-vel. A +-ra vonatkozó konverziós szabályok következtében, ha E1 tömb és E2 egész, akkor

```
E1 [E2]
```

az E1 tömb E2-dik elemére hivatkozik. Emiatt - aszimmetrikus megjelenése ellenére - az indexelés kommutatív művelet. A többdimenziós tömbökre következetes szabály vonatkozik. Ha E n-dimenziós, i * j * . . . * k-rangú tömb , akkor kifejezésekben (n-1)-dimenziós, j*...*k-rangú tömböt megcímző mutatóvá alakul át. Ha a * operátor akár explicit, akár indexelés következtében implicit módon erre a mutatóra alkalmazzuk, az eredmény a megcímzett (n-1)-dimenziós tömb, amely maga is azonnal mutatóvá alakul át. Tekintsük pl. az

```
int x [3][5];
```

deklarációt. Itt x 3*5-ös egész tömb. Ha x kifejezésben jelenik meg, akkor x a három darab 5-tagú egész tömb közül az elsőt megcímző mutatóvá alakul át. Az x[i) kifejezésben, amely *(x+i)-vel egyenértékű, x először az ismertetett módon mutatóvá, majd i az x típusával azonos típusúvá alakul, ami magában foglalja azt, hogy i megszorzódik annak az objektumnak a hosszával, amelyre a mutató mutat: ez jelen esetben 5 egész objektum. Az eredmények összeadódnak, és indirekció alkalmazásával (5 egészből álló) tömb keletkezik, amely viszont ezen egészek közül az elsőt megcímző mutatóvá alakul át. Ha még további index is van, ismét ugyanezt a megfontolást kell alkalmazni; esetünkben az

eredmény egész.

A fentiekből következik, hogy a C-ben a tömbök sor-folytonosan tárolódnak (az utolsó index változik a leggyorsabban), továbbá, hogy a deklarációban előforduló első index segítségével határozható meg a tömb által elfoglalt tárterület nagysága, egyéb szerep e azonban az indexszámításokban nincs.

14.4. Explicit mutatókonverziók

A mutatókra bizonyos konverziók megengedettek ugyan, de gépfüggő vonatkozásaik vannak. Valamennyi ilyen konverziót explicit típuskonverziós operátorral írhatjuk elő (l. a 7.2. és 8.7. pontot). Mutatók bármely olyan integrális típussá átalakíthatók, amelyben elférnek. Az, hogy ez a típus **int** vagy **long**-e, gépfüggő. A leképzés maga is gépfüggő, de azok számára, akik ismerik a gép címzési struktúráját, nem okozhat meglepetést. A későbbiekben néhány gépre vonatkozóan a részleteket is ismertetjük. Az integrális típusú objektumok explicit módon mutatókká alakíthatók át. A leképzés hatására a mutatókból létrejött egészek ugyanazokká a mutatókká alakulnak vissza, egyébként a folyamat gépfüggő. Adott típust megcímző mutató más típust megcímző mutatóvá alakítható. Az eredményül kapott mutató címzési zavarokat okozhat, ha a szóban forgó mutató által megcímzett objektum illeszkedése a tárban nem megfelelő.

Bizonyos azonban, hogy adott méretű objektumot megcímző mutató változatlan marad, ha először kisebb méretű objektumot, majd ismét az eredeti méretű objektumot megcímző mutatóvá alakítjuk. A tárterület-foglaló rutin pl. elfogadhatja valamely kiutalandó objektum (byte-okban megadott) méretét és **char** mutatót adhat vissza:

```
extern char *alloc ();
double *dp;
dp = (double *) alloc (sizeof (double));
*dp = 22.0 / 7.0;
```

Az alloc-nak (gépfüggő módon) biztosítani kell, hogy a visszaadott értéket át lehessen alakítani **double** mutatóvá; ebben az esetben a függvény használata gépfüggetlen. A PDP- 11 mutatóábrázolása 16 bites egésznek felel meg, egysége a byte. A char-okkal szemben nincsenek illeszkedési követelmények; minden másnak páros címűnek kell lennie. A Honeywell 6000 gépen a mutató 36 bites egésznek felel meg: a szórész a bal oldali 18 biten van, és az a két bit, amely a szón belül a karaktert választja ki, ettől közvetlenül jobbra található. Így a karaktermutatókat a 216 byte-os egységekben mérjük , minden más 218 gépi szó egységekben mérhető.

A **double** mennyiségeknek és az azokat tartalmazó aggregátumoknak páros szócímen kell elhelyezkedniük (0 mod 219). Az IBM 370 és az Interdata 8/32-es gépek hasonlóak. A címeket mindkettőn byte-okban mérjük; az elemi objektumoknak a hosszuknak megfelelő határra kell illeszkedniük, így a **short**-ot megcímző mutatóknak (0 mod 2)-nek, az int-re és **float**-ra mutatóknak (0 mod 4)-nek és a **double**-ra mutatóknak (0 mod 8)-nak kell lenniük. Aggregátum illesztése az alkotóelemeire vonatkozó illeszkedési feltételek közül a legszigorúbb szerint történik.

15. Állandó kifejezések

A C nyelvben több helyen kell alkalmaznunk olyan kifejezéseket, amelyeket kiértékelve állandó eredményt kapunk: **case** után, tömbhatárként, kezdeti értékekként. Az első két esetben a kifejezésben csupán egész állandók, karakterállandók és sizeof kifejezése k szerepelhetnek, amelyeket a

+ - * / % & | ^ << >> == != < > <= >=

két-, ill. a

- ~

egyoperandusú operátorok valamelyike vagy a háromoperandusú

? :

operátor köthet össze egymással. A zárójelek csoportosításra használhatók, függvényhívásra azonban nem. Kevesebb megkötés vonatkozik a kezdeti értékekre; az előbb tárgyalt állandó kifejezéseken kívül külső és statikus objektumokra, valamint állandó kifejezéssel indexelt külső és statikus tömbökre is alkalmazható az egyoperandusú & operátor. Implicit módon az egyoperandusú &-et indexeletlen tömbök és függvények megjelenésekor ugyancsak alkalmazhatjuk.

Az alapszabály az, hogy a kezdeti értékek kiértékelésével vagy állandót, vagy pedig valamely már korábban deklarált külső vagy statikus objektum (esetleg állandóval növelt vagy csökkentett) címét kell megkapnunk.

16. Gépfüggetlenség

A C nyelv bizonyos részei lényegüknél fogva gépfüggők. Az alábbiakban nem térhettünk ki minden problémára, csak a legfontosabbakat akartuk kiemelni. Az olyan tisztán hardverkérdések, mint a szavak mérete, a lebegőpontos aritmetika tulajdonságai és az egészek osztása a gyakorlatban nem okoztak különösebb gondot. A hardver egyéb jellegzetességei az eltérő megvalósításokban mutatkoznak meg. Ezek némelyike, különösen az előjel-kiterjesztés (negatív karakter negatív egésszé történő átalakítása), valamint a byte-ok szavakon belüli elhelyezkedési sorrendje olyan kellemetlen tényezők, amelyekre különös figyelmet kell fordítanunk.

Az egyéb gépfüggő tulajdonságok már nem jelentenek nagyobb problémát. A regiszterekben ténylegesen elhelyezkedő **register** típusú változók száma - a megengedett típuskészlethez hasonlóan - gépről gépre változik. Minden fordító helyesen végzi azonban a dolgát a saját gépe szempontjából: a fölös számú vagy érvénytelen **register** deklarációkat nem veszi figyelembe. Nehézségek csak akkor támadnak, amikor valaki rossz programozási módszereket alkalmaz.

Ne írjunk olyan programokat, amelyek az adott architektúra bármilyen specifikus tulajdonságától függetlenek! A függvényargumentumok kiértékelési sorrendjét a nyelv nem határozza meg. PDP- 11- n jobbról balra, a többi gépen balról jobbra történik. A mellékhatások érvényesülésének sorrendje ugyancsak nem meghatározott. Mivel a karakterállandók valójában **int** típusú objektumok, több karakterből álló karakterállandók használata is megengedett. Ennek megvalósítása azonban rendkívül gépfüggő, mivel a karakterek szóhoz történő hozzárendelésének sorrendje gépről gépre változik. Mezők hozzárendelése szavakhoz, karaktereké egészekhez a PDP 11-en jobbról balra, a többi gépen balról jobbra történik. Elszigetelt programok számára e különbségek láthatatlanok maradnak, hacsak

nem viszik túlzásba a típusokkal folytatott játékot (pl. az által, hogy valamely **int** mutatót **char** mutatóvá alakítanak át, majd megvizsgálják a megcímzett tárterületet). Számolnunk kell azonban e különbségekkel akkor, ha a programunkat kívülről megszabott tárterület-elrendezésekkel akarjuk összhangba hozni. A különféle fordítók által elfogadott nyelvek csupán egészen kis részletekben térnek el egymástól.

A leglényegesebb, hogy a pillanatnyilag használatos PDP-11-es fordító nem inicializálja a bitmezőket tartalmazó struktúrákat, és egyes értékadó operátorokat nem fogad el olyan környezetben, ahol ki akarjuk használni a hozzárendelés értékét.

17. Anakronizmusok

Mivel a C fejlődésben levő nyelv, egyes régebbi programokban bizonyos elavult szerkezetek találhatók. Bár a fordító legtöbb változata az ilyen anakronizmusokat is támogatja, előbb-utóbb ezek el fognak tűnni, csupán gépfüggőségi problémát hagyva maguk után. A C nyelv korábbi változatai értékadó operátorként az =op alakot használták az op= alak helyett. Ez kétértelműségekhez vezet, amelynek tipikus esete

```
x = -1
```

amely a valóságban x-et dekrementálja, mivel az = és a – szomszédosak, de amivel könnyen az lehetett a szándékunk, hogy -1-et rendeljünk x-hez. A kezdeti értékek szintaxisa megváltozott: korábban a kezdeti értéket bevezető egyenlőségjel nem szerepelt, így az

```
int x = 1;
```

alak helyett az

```
int x 1;
```

alak volt használatban. A változtatás azért történt, mert az

```
int f (1+2)
```

alakú inicializálás éppen eléggé hasonlít a függvénydeklarációra ahhoz, hogy megtévessze a fordítókat.

18. A szintaxis összefoglalása

A C nyelv szintaxisának összefoglalása sokkal inkább tömör segédletül, mintsem a nyelv rövid összefoglalásául szolgál.

18.1. Kifejezések

Az alapvető kifejezések a következők:

kifejezés:

elsődleges_kifejezés

- *kifejezés
- &kifejezés
- -kifejezés
- !kifejezés
- ~kifejezés
- ++balérték
- --balérték
- balérték ++
- balérték --
- sizeof kifejezés
- (típus_név) kifejezés
- kifejezés kétop kifejezés
- kifejezés ? kifejezés : kifejezés
- balérték értékadó_op kifejezés
- kifejezés , kifejezés

elsődleges_kifejezés:

- azonosító
- állandó
- karakterlánc
- (kifejezés)
- elsődleges_kifejezés (kifejezés_listaopc)

- elsődleges_kifejezés [kifejezés]
- balérték . azonosító
- elsődleges_kifejezés -> azonosító

balérték:

- azonosító
- elsődleges_kifejezés [kifejezés]
- balérték . azonosító
- elsődleges_kifejezés -> azonosító
- *kifejezés
- (balérték)

A

() [] . ->

elsődleges kifejezés operátorok prioritása a legmagasabb, és az ilyen operátorok balról jobbra kötnek. Az egyoperandusú

* & - ! ~ ++ -- sizeof (típusnév)

operátorok prioritása az elsődleges operátorokénál alacsonyabb, de magasabb az összes kétoperandusú operátorénál: ezek az operátorok jobbról balra kötnek. Az összes kétoperandusú operátor és a feltételes operátor balról jobbra köt, ezeket az alábbiakban csökkenő prioritási sorrendben soroljuk fel:

kétop:

- * / %
- + -
- >> <<
- < > <= >=
- == !=
- &
- ^
- |
- &&
- ||
- ? :

Az értékadó operátorok mindegyike azonos prioritású, és mindegyik jobbról balra köt.

értékadó_op:

= += -= *= /= %= >>= <<= &= ^= |=

A vessző operátor (,) prioritása a legalacsonyabb, és balról jobbra csoportosít.

18.2. Deklarációk

deklaráció:

- dekl._specifikátorok deklarátorlistaopc;

dekl._specifikátorok:

- típus_specifikátor dekl._specifikátorokopc
- t.o._specifikátor dekl._specifikátorokopc

t.o._specifikátor:

- **auto**
- **static**
- **extern**
- **register**
- **typedef**

típus_specifikátor:

- **char**
- **short**
- **int**
- **long**
- **unsigned**
- **float**
- **double**
- strukt._vagy_union_specifikátor
- typedef_név

k.é._deklarátorlista:

- k.é._deklarátor
- k.é._deklarátor , k.é._deklarátorlista

k.é._deklarátor:

- deklarátor inicializálóopc

deklarátor:

- azonosító
- (deklarátor)
- *deklarátor
- deklarátor ()
- deklarátor [állandó_kifejezésopc]

strukt._vagy_union_specifikátor:

- **struct** { strukt._dekl._lista }
- **struct** azonosító { strukt._dekl._lista }
- **struct** azonosító
- **union** { strukt._dekl._lista }
- **union** azonosító {strukt._dekl._lista }
- **union** azonosító

strukt._dekl._lista:

- strukt._deklaráció
- strukt._deklaráció strukt._dekl._lista

strukt._deklaráció:

- típus_specifikátor strukt._deklarátor_lista;

strukt._deklarátor_lista:

- strukt._deklarátor
- strukt._deklarátor , strukt._deklarátor_lista

strukt._deklarátor:

- deklarátor
- deklarátor : állandó_kifejezés
- : állandó_kifejezés

inicializáló:

- = kifejezés
- = { inicializáló_lista }

- = { inicializáló_lista, }

inicializáló_lista:

- kifejezés
- inicializáló_lista , inicializáló_lista
- { inicializáló_lista }

típus_név:

- típus_specifikátor absztrakt_deklarátor

absztrakt_deklarátor:

- üres
- (absztrakt_deklarátor)
- *absztrakt_deklarátor
- absztrakt_deklarátor ()
- absztrakt_deklarátor [állandó_kifejezésopc]

typedef_név:

- azonosító

18.3. Utasítások

összetett_utasítás:

- { deklarációlistaopc utasításlistaopc }

deklarációlista:

- deklaráció
- deklaráció deklarációlista

utasításlista:

- utasítás
- utasítás utasításlista

utasítás:

- összetett_utasítás
- kifejezés;
- if (kifejezés)
- utasítás

- **if** (kifejezés)
- utasítás
- **else** utasítás
- **while** (kifejezés)
- utasítás
- **do**
- utasítás
- **while** (kifejezés) ;
- **for** (kifejezés_1opc; kifejezés_2opc; kifejezés_3opc)
- utasítás
- **switch** (kifejezés)
- utasítás
- **case** állandó_kifejezés:
- utasítás
- **default**:
- utasítás
- **break**;
- **continue**;
- **return**;
- **return** kifejezés;
- **goto** azonosító;

azonosító:

- utasítás
- ;

18.4. Külső definíciók

program:

- külső_definíció
- külső_definíció program

külső_definíció:

- függvénydefiníció
- adatdefiníció

függvénydefiníció:

- dekl._specifikátoropc függvénydeklarátor függvénytörzs

függvénydeklarátor:

- deklarátor (paraméterlistaopc)

paraméterlista:

- azonosító
- azonosító , paraméterlista

függvénytörzs:

- deklarációlista függvény_utasítás

függvény_utasítás:

- deklarációlistaopc utasításlista

adatdefiníció:

- externopc típus_specifikátoropc k.é._deklarátorlista;
- staticopc típus_specifikátoropc k.é._deklarátorlista;

18.5. Előfeldolgozó

```
#define azonosító szint._egységek_karakterlánca
#define azonosító(azonosító, ..., azonosító)
  szint._egységek_karakterlánca
#undef azonosító
#include "állománynév"
#include <állománynév>
#if állandó_kifejezés
#ifdef azonosító
#ifndef azonosító
#else
#endif
```

#line állandó azonosító