

Viola Zoltán
violazoli@gmail.com
<http://parancssor.info>

Hogyan írhatasz saját programnyelvet?

Tartalomjegyzék

Bevezetés	3
1. fejezet: Milyen nyelvet készítsünk?.....	13
2. fejezet: A program vázának és főbb funkcióinak áttekintése.....	17
3. fejezet: A mau nyelvű program parancsai/utasításai, és ezek végrehajtása.....	28
4. fejezet: Pár szó a pontosvesszőről.....	40
5. fejezet: Változókezelés, meg azok a fránya címkék.....	41
6. fejezet: A programnyelvünk Turing-teljessé tétele.....	67
7. fejezet: Értékadó utasítások.....	74
8. fejezet: Műveletek végzése.....	85
9. fejezet: Memóriaműveletek.....	95
10. fejezet: If, then, else.....	98
11. fejezet: Syntax error.....	101
12. fejezet: Az „aritmetikai kifejezés” bővítése operátorokkal.....	102
13. fejezet: Névterek, és a változóink élettartama.....	115
14. fejezet: Veremkezelés.....	124
15. fejezet: Fejlett vezérlési szerkezetek.....	131
16. fejezet: Vesszőcske, avagy a paraméter-szeparátor.....	140
17. fejezet: Unáris operátorok.....	142
18. fejezet: „Normális” szintaxisú értékadás megvalósítása.....	143
19. fejezet: Tömbök.....	147
20. fejezet: Inkrementálás és dekrementálás.....	156
21. fejezet: Stringkezelés.....	159
22. fejezet: File-kezelés.....	175
23. fejezet: Rendszerfüggvények.....	180
24. fejezet: Tartalomjegyzék-kezelés.....	184
25. fejezet: Mau nyelvű függvények hívása.....	195
26. fejezet: A „maudir” program, vagyis az első, valóban hasznos mau nyelvű programunk.....	207
27. fejezet: A mau program programmemóriájának elérése.....	222
28. fejezet: A switch-szerű vezérlési szerkezet.....	223
29. fejezet: A második ténylegesen hasznos mau nyelvű programunk, ami névsorba rendezi egy fájl sorait.....	226
30. fejezet: Változó hosszúságú paraméterlista kezelése.....	227
31. fejezet: „Igazi” függvények és rekurzív függvények.....	231
32. fejezet: Bencsmarkok.....	235

Bevezetés

Ez a leírás arról szól, hogyan lehet megalkotni egy új programnyelvet, olyat ami még nem volt, nem létezett, s ami ezért természetesen épp olyan, amilyennek mi magunk azt látni szeretnénk.

Nyilvánvaló persze, hogy mint mindennek a világon, a programnyelveknek is vannak általános jellemvonásaik, azaz AKÁRMILYEN azért mégsem lehet - ez olyan, hogy ha el is jut valaha az Emberiség addig hogy a genetikusok képesek legyenek bármiféle állatokat is teremteni, azért azok se hághatnak át bizonyos alapvető természeti törvényeket, azaz merőben valószínűtlen, hogy olyan állatot teremtsenek, ami akkora, mint egy mammut, olyan nehéz is, ugyanakkor mégis képes repülni a levegőben!

Abban azonban senki nem kételkedik remélem, hogy ha akad is mondjuk vagy száz programnyelv a világon, ez a lehetséges variációk elenyésző töredékét teszi csak ki, azaz bőven van lehetőségünk olyat kiötlölni, ami még nincs, ám nekünk valamiért sokkal jobban tetszik, mint azok bármelyike, amik már léteznek!

Ráadásul egészen biztos, hogy kivétel nélkül minden, már létező programnyelvnek van egy óriási hibája, még azoknak is, amiknek a létéről se hallottunk: Tudniillik ezek mindegyikéről elmondható az az iszonyatos, óriási negatívum, hogy NEM MI ÍRTUK ŐKET!

Na és hát ez azért eléggé ciki. Tök „gáz”, tiszta „égő” lenne úgy meghalni, hogy soha egyetlen programnyelvet sem alkottunk. Ezzel szégyent hoznánk dicső őseinkre, s még az is lehet hogy úgy megorrolnának ránk, hogy nem túrnének meg maguk közt, bezárulna előttünk mind a Menny, mind a Pokol kapuja, és büntetésből visszatoloncolnának az Életbe...!

Iszonyatos perspektíva! Ezt a veszélyt semmiképp se vállalhatjuk, nincs épeszű ember aki ezt megkockáztatná, s emiatt, meg mert különben is szeretnénk azzal hencegni hogy mi olyan „geek” fazonok vagyunk, hogy már saját programnyelvünk is van, meg kell alkotnunk a magunkét, ez tiszta sor, ez egyszerűen elkerülhetetlen és parancsoló szükségszerűség, mondhatni „a kor szava”! Manapság már egyszerűen *illik*, hogy legyen mindenkinek egy saját programnyelve, elvégre nem élünk már a középkorban, emiatt nem tűrhetünk el efféle intellektuális szegénységet, nem tespedhetünk tovább a barbár szellemi nyomorban!

Arról nem is beszélve, hogy ha megalkotunk egy mondjuk XYZ nevű programnyelvet, azután teljes joggal dicsekedhetünk azzal, hogy mi vagyunk az a személy, aki az egész Világon a legislegjobban tud programozni egy bizonyos programnyelven, tudniillik ezen az XYZ nevű nyelven! Milyen nagyszerű is lenne ez!

Ez mind rendben is volna, ugyanakkor viszont azért ennek van pár aprócska feltétele. Mindenekelőtt: akármilyen nyelvet is alkossunk, ezt valamely már létező programnyelven kell megírunk! Elkerülhetetlen hát, hogy Olvasóm, ki majd láthatatlan kísérőm lesz ezen izgalmas szellemi kalandban, tudjon már programozni valamennyire, valamilyen programnyelven. Azt se titkolom el, melyiken: Én a magam nyelvét a C és C++ nyelveken írtam meg, a következő okokból:

1. Ezeket ismerem. (Ez azért nem utolsó szempont, ismerjük el... Gondoljunk csak bele, mily roppant nehézségbe ütközött volna olyan programnyelven leprogramoznom ezt, amelyet még NEM ismerek...)

2. A „C” nyelv, az „a programnyelvek angolja”. Gyakorlatilag nem is programozó a szememben - de más, kicsit is igazi szakembernek számító „vén szaki” szemében sem! - aki nem ismeri a C nyelvet, ha talán nem is profi módra, de minimum alapszinten.

3. Talán nem is létezik olyan számítógép-architektúra, amire ne lenne már kész C fordító, azaz ezáltal leendő nyelvünk könnyedén portolható lesz a számítógépek roppant széles spektrumára.

4. A C nyelvnek van egyfajta „objektumorientált kiterjesztése”, a C++ nyelv. Ez elvileg külön programnyelvként van számontartva, de mert teljesen kompatibilis a C-vel, én egyszerűen a C egyfajta bővítésének nevezem nagyvonalúan. Azaz ha C nyelven programozok, akármikor lehetőségem van objektumorientált eszközöket és módszereket is igénybe vennem, anélkül, hogy hirtelen egy tök más nyelvre váltanék át, nem kell átírnom az addig elkészült részeket.

5. Egy programnyelv megírása kétségkívül nevezhető „rendszerközeli” programozási feladatnak, márpedig a C nyelvet eredetileg éppen efféle feladatok végrehajtására fejlesztették ki, tulajdonképpen ezen írták meg az Unix operációs rendszert is, ami a mai Linux oprendszer őse. Vélhető emiatt, hogy messze alkalmasabb a céljainkra, mint az afféle mostanában létrejött „úri huncutságok”, mint a JavaScript, Python, PHP, Haskell meg más egyébek, amiket szerintem glaszékesztyűs, unatkozó ficsúrok számára találtak ki.

Amennyiben tehát Olvasóm nem lenne tisztában a C nyelv alapjaival, sőt még a C++ alapjaival is, ne nagyon reménykedjék benne, hogy komolyabb előzetes tanulás nélkül nagy hasznát veheti e könyvnek. Még azt is pontosíthatom, MENNYI ismeret kell neki ezekből a dolgokból:

— A C++ nyelvnek elég csak az alapjaival tisztában lennie, nem kell hogy nagyon mélyen járatos legyen benne. Ha pici, egyszerű programokat tud benne elbarkácsolgatni, mint „magabiztos kezdő”, az már elég.

— A C nyelvet illetően azonban kifejezetten az „erősen haladó” kategória szükségeltetik. Nem a „profi”, de mindenképp olyan, ami nemcsak nem kezdő, de az „átlagos” szintnél is jobb. Se a C, se a C++ nyelv alapfogalmait nem fogom e leírásban magyaráztatni, és magától értetődőnek tartok majd olyasmiket, hogy az olvasónak a szeme se rebben - mert ÉRTI és TUDJA, hogy miről beszélek! - amikor esetleg olyasféle dolgokról elmélkedek majd e könyv lapjain, hogy például:

— **„Ez egy olyan tömb, mely olyan tömbök pointereit tartalmazza, mely tömbök olyan függvények címeit tárolják, mely függvények int értékkel térnek vissza, input paraméterként pedig egy F struktúra referenciáját várják el.”**

Megnyugtatóként közlöm, ennél bonyolultabb deklarációink nemigen lesznek remélhetőleg, a fenti azonban nem kitaláció, hanem ténylegesen létezik is a megvalósított nyelvemben... Mert természetesen a programnyelv tervezését úgy mutatom be, hogy lépésről-lépésre megalkotunk egy konkrét, használható programnyelvet!

Na most, amióta csak létezik a C nyelv, azóta még a pokolbeli kanördög is tudja róla, hogy minden hátulgombolós padawannak, aki e nyelv elsajátításával próbálkozik, messze a legnagyobb nehézséget e nyelvben a *mutatók* megértése okozza, pláne ha olyasmikről van szó, hogy a mutatókra mutató mutatók... (mutatók helyett írhattam volna a „pointer” szót is, ugyanezt jelenti). Hát még, ha függvényekre mutató pointerekről van szó, meg ezeknek a tömbjeiről...

És bár sokáig ellébecolhat e nyelvben egy kezdő ezek használata nélkül, de komoly programot írnia enélkül bizony lehetetlen. Ha tehát ezzel nem vagy tisztában — kár a gőzért!

A mutatók használata könyvemben és a programomban nem valami sátáni gonoszság a részemről. Bár megértésük eleinte tényleg némi nehézségbe kerül, de HA már egyszer megtanultunk velük bánni, kiderül, hogy általuk minden de minden messze sokkal KÖNNYEBB, és az elkészült kód is HATÉKONYABB!

Mindenesetre, e könyvnek nem az a célja, hogy a C (vagy a C++) programozás alapjait vagy akár magasiskoláját magyarázgassa. Minden efféle magától értetődőnek tételezek fel. Nem azt akarom bemutatni, miként és mire lehet használni a C/C++ nyelvet, hanem hogy miként lehet megalkotni egy totál tök más programnyelvet, eközben mikre kell feltétlenül figyelni, s mik azok a dolgok, amiket e témában járatlan valaki talán egetrengető fontosságúnak tart és rém nehéznek is, holott esetleg abszolút nem fontos, vagy ha fontos is, de igazából egyáltalán nem nehéz.

A könyvet elejétől kell olvasni, másképp érthetetlen lesz. Ennek ellenére, az egyes fejezetekben kevés lesz a konkrét kód. Természetesen a könyv végén megtalálható lesz a teljes, elkészült program forráskódja, de épp emiatt feleslegesnek tartottam volna korábban is újra meg újra közreadni a teljes rutinok kódlistáját, helyett mindig csak azokat a részeket idéztem be, amik épp fontosak a mondanivaló szempontjából — azaz a rutinok azon részeit, melyek magától értetődőek (szerintem...), vagy épp nem a témához tartoznak, ezeket mind kihagytam. Annak érdekében azonban, hogy mindig jól látszódjék, hol történt e „nyomdafestékekkel spórolás”, e helyeket, ahol egy vagy több kódsort kihagytam, sok-sok ponttal jelöltem, így:

.....

Hogy épp hánytal, az mindegy, addig nyomtam a billentyűt, míg elegendően figyelemfelkeltőnek nem tartottam a mennyiséget. Ez tehát jelenthet akár 1, akár 1024 kihagyott sort is...

A programnyelvemet megvalósító C/C++ nyelvű program forráskódját mindig efféle stílussal írom mint e példában látszik:

```
int main(int argc, char **argv) {
```

Azaz, ez fix szélességű betűtípussal van, és nem sorkizárt, hanem balra igazított, továbbá félkövér.

Annak a programnyelvnek, amit e leírás során megalkotunk, én a „**mau**” nevet adtam. Ha olyasmit írok le, ami e mau nyelven íródott kódrészlet, azt ugyanilyen betűtípussal írom, azt kivéve, hogy az piros színű lesz, azaz efféleképp néz majd ki:

```
=#c@c 26 // c=26, ennyiszor hajtódik majd végre a ciklus
{| #c@c // c-darabszámra fut e majd ez a ciklus
? (@c)+'@ //
-#c @c 1 // c-=1
|}(((@c)+'@) == S) // kilépünk 'S'-nél
// Ciklus vége
// Üres sor kiírása
"Ít a vége fuss el véle!" /
XX // Vége a programnak
```

Ilyen stílussal jelzem ki a mau program konzolra írt outputját, meg a mau interpreter esetleges hibaüzeneteit:

```
ZYXWVUT
Ít a vége fuss el véle!
```

Ilyennel meg az olyan mau programnyelven írt programsorokat, amik HIBÁSAK:

```
|}(((@c)+'@) == B) // kilépünk 'S'-nél
```

Fontosnak tartom megemlíteni azt is, még hozzá **HANGSÚLYOZOTTAN KI-EMELVE**, hogy ha valahol e könyvben az Olvasó talál egy, a programnyelvemet megvalósító progi részét képező függvényt, rutint, akármit, az nem okvetlenül jelenti azt, hogy az az izémizé már rögvést a VÉGLEGES változat is! Nagyonis sok olyan rutin lesz, amit ahogy haladunk a nyelv tervezésében, többször is átírunk kisebb-nagyobb mértékben. Többször csak bővítjük ezeket, de akadnak olyan esetek is, amikor alapvetően megváltoznak. Ez messzemenőleg jellemző arra, ami az „aritmetikai kifejezés” kiértékelését megvalósító függvény (illetve függvény-csoport). Ha egyáltalán nevezhető valami úgy egy programnyelv megalkotásakor (csúnya szóval élve) hogy „szívás”, sőt „hatalmas szopás”, na hát akkor EZ A RÉSZ AZ! Igazából ehhez képest az összes többi rész nem más, mint „egy laza, könnyed kézlegyintés”, amit egy picit is programozni tudó, a téma iránt érdeklődő óvodás, aki előtte vagy két hétig gyakorolt az apuci gépén, maga is összedob egy unalmas hétvégén. Vagy ha ez netán túlzás is, de azt már nem érzem annak, ha azt állítom, egy tehetséges utcalány is megoldja a többi, ha úgy dönt, hogy felhagy a kéjipari „szakmával”, s tanul előtte csak 1 hónapot is szorgalmasan C nyelven programozni.

Valójában különben nem volna ez sem olyan szörnyűségeken nehéz rész, lényegében az egész nem nagyon áll másból, mint hogy e rutin időnként meghívja önmagát rekurzívan, oszt' jóóól van, jóccakát! Sajnos azonban a dolog mégsem intézhető el ilyen könnyen, tekintve hogy nekünk egy csomó különböző adattípussal is illik foglalkozni, s ez lényegesen bonyolultabbá teszi az egészet. De nem kell megijedni, épp azért, hogy minden érthető maradjon, nem esünk neki az egésznek

hűbelebalázs módjára, fokozatosan fejlesztjük fel ezt a részt (is) addig, míg már „full extra de luxe” lesz. Azaz nem kell elkeseredni az elején, amikor még úgy indulunk hogy csak egész számokat tudunk kezelni, stb. Lesz majd ez sokkal jobb is. Apránként haladunk majd, hogy mindig érthetőek maradjunk, sajnos ez azonban azzal jár, hogy egyes rutinokat gyakran kell kicsit vagy jobban újraírni.

Mindez érvényes a leendő szépséges programnyelvünk SZINTAXISÁRA IS, azaz az olyasmikre, hogy például miként is jelölünk (vagyis nevezünk meg, azonosítunk) egy változót, miként is néznek ki nálunk egyes utasítások, melyeknek mi a **neve**, stb. Azaz ha látsz is az elején valami mau nyelven írt kódrészletet e könyvben, abszolút nem biztos, hogy az érvényes és futtatható kódnak számít majd a nyelv végleges változata szerint! Mert, tudod, ami a programnyelvünk kialakításának kezdetén esetleg remek ötletnek tűnik, mert lényegesen megkönnyíti a munkánkat, esetleg cseppet sem tűnik olyan „nyerő ötletnek” a későbbiekben. Mégsem kár hogy az elején így indulunk neki, efféle „őskőkori módszerekkel”, mert legalább már eljutunk valamerre, valameddig, azaz HALADUNK, míg ha rögvést a legmagasabb célokat tűzzük ki magunk elé, s mindenféle olyan módszerekkel kezdünk ebbe az egészbe, amiket a „nagy fejek”, komoly, ősz-szakállú professzorok javasolnak, akkor előbb vagy 10 évet tanulnunk kéne holmi egyetemen, s még azután is feltehetőleg sose készülnénk el a nyelvünkkel, mert leragadnánk a tervezgetésnél.

Ez a dolog roppantmód hasonlatos a Linux operációs rendszer elkészültéhez és karrierjéhez. Köztudott, hogy az egészet egy Linus nevű fiatal egyetemista suhanc kezdte kifejleszteni. És amikor már megvolt ez-az belőle, s tudomást szerzett róla a tanára, az egyetemi prof, aki maga is javában egy operációs rendszeren dolgozott, ezt erősen kritizálta, mert Linus úgynevezett „monolitikus” kernelt (rendszermagot) fejlesztett, nem „mikrokernelt” architektúráját, amin a professzor is dolgozott. És a prof azt mondta, Linus tákolmánya ELAVULT, korszerűtlen.

Na most itt a mi számunkra nem lényeges, hogy konkrétan mit jelent a „monolitikus” meg a „mikrokernelt” fogalma (TUDOM mit jelentenek, csak mellékes e könyv szempontjából), hanem csupán az a fontos, hogy a professzornak alapvetően IGAZA VOLT. Épp csak annyi baj volt az igazával, hogy egy mikrokerneles rendszer mag úgy tűnik nehezebben összedobható, mint a másik fajtájú. Vagy ha ez nem így volna is, hát érdekes, hogy akkor miért is lett úgy, hogy amit Linus, a bátor kísérletező alkotott, a Linux oprendszer oly látványos karriert futott be a professzora készítményéhez, a Minixhez képest! Szerintem, ha Linus akkor belebonyolódik abba, hogy valami modern „mikrokerneles” oprendszert fejlesszen, sosem készül el vele, s ma nincs olyan, hogy Linux.

Jogos lehet az igény Olvasóim részéről, hogy mielőtt e vaskos iromány tanulmányozásába mélyebben belemerülnének, előbb halljanak tőlem valami infót arra vonatkozóan, tulajdonképpen mennyire is nehéz úgy összességében egy programnyelv megalkotása!

Nos, erre vonatkozóan természetesen nem lehet objektív mércét felállítani, mert ez leginkább az Olvasó már eddig megszerzett programozói ismeretein múlik, valamint azon, hogy ÉPP AZ a programnyelv, amit meg akar alkotni, mennyire lesz bonyolult. Nagy általánosságban azonban annyit írhatok erről, hogy alapvetően egy programnyelv megalkotása MESSZE-MESSZE SOKKAL DE SOKKAL

KÖNNYEBB, mint amilyennek ezt gyakorlatilag mindenki hiszi, aki még nem foglalkozott ilyesmivel, másrészt pedig kifejezetten sokkal de sokkal **KÖNNYEBB**, mint jónéhány olyan, nagyon gyakori feladat, amikkel esetleg Olvasóm talán foglalkozott is már korábban nemegyszer. Hogy csak egyetlen példát hozzak effélére, kezdő programozók gyakorta próbálkoznak olyasmivel, hogy megírjanak valami JÁTÉKPROGRAMOT, mert azon akarnak gyakorolni, meg mert azt hiszik, az könnyű feladat, hiszen az csak *játék*... Nos, elárulom, hogy ameddig csak olyasmiről van szó, hogy megy a figura a labirintusban és össze kell szednie a kincseket, addig egy játékprogram valóban könnyebb talán, mint egy programnyelv megírása. (De még ez is csak *talán*...) Abban a pillanatban azonban, amint abba a játékba bele óhajtasz venni néhány szörnyet is, amikkel nem előnyös, ha a karaktered találkozik, pláne ha e szörnyekre lőhetsz is, de különösen ha még a szörnyek is lőhetnek rád, na abban a pillanatban egy effélének a leprogramozása szerintem máris SOKKAL BONYOLULTABB és több tudást igényel, mint a legtöbb elképzelhető programnyelv megvalósítása. Természetesen elképzelhető, hogy olyan programnyelvet találsz ki, ami felülmúlja egy efféle játékprogram bonyolultsági szintjét, de mély meggyőződésemm, hogy az esetek messze túlnyomó többségében nem ez lesz a helyzet, s ha mégis, az nem az első nyelv lesz, amit meg akarsz majd valósítani.

Egyáltalán, gondoljunk csak bele: egy programnyelv alapvetően nem más, mint egy olyan micsoda, ami egy szövegfájl értelmez. E szövegfájl ugye a forráskódot tartalmazza. Ebből kell neki előállítania gépi kódú utasításokat, ha compiler típusú nyelvet készítesz, vagy ezen utasításokat rögvést értelmezi és végrehajtja, interpreter típusú nyelvek esetében. Alapvetően tehát minden esetben *végsősoron* csak egy közönséges szövegfeldolgozásról van szó. Lényegében abszolút nem kell törődnöd se a grafika, se a hangrendszer programozásával. Sőt, még a felhasználóval se kell semminemű kommunikációt se folytatnod, nincs interaktivitás, mert ha minden oké, akkor nincs mit tenni ilyen téren, ha meg valami gáz van, egyszerűen kiírod a hibaüzenetet és megállsz. Egy játékprogramban azonban igenis VAN mindez, nagyonis, van interaktivitás is, grafika is, hang is, meg a fene tudja még mi minden is. Az tehát igenis SOKKAL DE SOKKAL BONYOLULTABB!

Miért alakult ki mégis az a hiedelem, hogy egy programnyelv megírása valami iszonyatosan nehéz és embert próbáló feladat?

Ennek egyik és talán legfőbb oka szerintem az, hogy az e témáról szóló könyvek szerzői - komoly professzorok, stb - részben szándékosan túlbonyolítják a kérdés-kör leírását, azért, hogy önmaguk fontosságát emeljék, másrészt az is közrejátszhat ebben, hogy SOHA ÉLETÜKBEN NEM VÉGEZTEK RENDSZERKÖZELI PROGRAMOZÁST! Talán még magas szintű nyelvekben se sokat programoztak, de hogy valami assembly-közeliben biztos nem, az tuti. Én ellenben egészen másként vagyok ezzel, életem első negyedében a C-64-et hackeltem gépi kódban, s a Linux alatt is voltak assembly-kalandjaim, ha nem is sok, ellenben itt a C a kedvencem, ami meglehetősen alacsonyszintű maga is.

(Egy gyors közbevetés a terminológiáról: a nagyon alapszintű „gépi kódú” nyelvet nevezzük úgy, hogy „assembly nyelv”, az ezen nyelvet lefordító programot pedig úgy, hogy „assembler”).

Na és hát egy programnyelv érthetően épp a rendszerközeli dolgokkal kell foglalkozzék elsősorban ugyebár! Akármilyen magasszintű is a nyelv, előbb-utóbb a

nyelv mégiscsak végre kell hajtson bizonyos KONKRÉT utasításokat az adott processzoron. Aki tehát gépközeli nyelveken „szocializálódott”, az jól tudja az olyan „örök igazságokat”, hogy például a processzor számára nem is létezik az a fogalom, hogy „változó”. A processzornak van pár regisztere, semmiképp se annyi, amennyi változó lehet egy adott programban, van veremtára, és van memóriája. Olyan hogy „változó”, olyan a számára egyszerűen nem létezik. Olyasmi sincs a számára többnyire, hogy adattípus. Neki minden csak bájt, vagy a bájt részei, a bitek. Ritkább esetekben persze van neki olyan képessége, hogy beépített lebegőpontos aritmetika, de ez is csak a regiszterekre korlátozódik, s amikor a számot ki kell írni valahova a memóriába, akkor az bizony igenis nem ilyen vagy olyan típus, hanem egyszerűen egy bájtsorozat. Olyant se nagyon tud a processzor, hogy ciklusok - neki nincs olyanja hogy WHILE, DO-UNTIL, FOR, neki minden csak elágazás, ugrás, esetleg szubrutinból való visszatérés. Azaz, gépi kódú szinten minden sokkal EGYSZERŰBB.

A „Magas szintű nyelvek” ehhez képest rémségesen túlbonyolítottak, holott eredetileg azért jöttek létre, hogy megkönnyítsék a programozók munkáját. Kétségtelen, akadnak is egyes területek, ahol ezek jelentős könnyebbséget nyújtanak. Másrészt, ezért cserébe azzal fizetnek az ezeken programozók, hogy egyéb területeken a munkájuk jelentősen megnehezedik, ráadásul a létrejött végrehajtható kód hatékonysága is elképesztően elmarad attól, amit valamely „alacsony szintű” nyelven kódolva kapnának.

A programnyelvek írásáról szóló művek írói valamiért úgy vélik, nekik okvetlenül azzal kell foglalkozniuk, miként lehet efféle túlbonyolított „magas szintű” nyelvet értelmező programot írni. És még eközben is igyekeznek kitérni minden lehetséges esetre és aletre, és tobzódni a szakszavakban, a legelvontabb magas szintű matematikai kifejezésekkel és algoritmusokkal dobálódznak, s azt hiszik attól komolyak és tudományosak, minél magasabb szintre emelik az elvontságot, az absztrakciót.

Egyre biztos jó is ez a megközelítés: hogy elvegye szinte mindenkinek a kedvét a programnyelvírástól...

Én egészen más megközelítést használok e könyvben. Egyrészt, ballisztikus ívben trotyyantok minden magasszintű elvontságra, absztrakcióra és matematikára! Félreértés ne essék, TISZTELEM a matematikát, sőt, kiváltképpen jól értem is és szeretem is az átlagemberhez képest, de sőt még azt is elismerem, hogy az efféle absztrakciók nagyonis hasznosak lehetnek olykor. Mégis, ez olyasmi, amikor nagyonis jól kiütöközik, amit sokszor mondogatnak: „Az elmélet nem azonos a gyakorlattal”. Ezt mindjárt megvilágítom egy példával:

Az egyetemen a programozó matematikusi szakon azzal kezdik, hogy megtanítják az oda járó diákoknak, mit is jelent az a szó, hogy „program”. A tanárok szerint:

„A program egy útvonal a probléma által reprezentált állapottéren”.

Ööö... Lehetséges volna hogy nem érted?! Pedig világos! Gondolj csak bele, hiszen az állapottér nem más, mint azon állapotok nem üres halmaza, melyek a probléma világát... izé... inkább hagyjuk! Ennek nem sok hasznát veszed akkor, ha

egy konkrét feladatot akarsz megoldani, pláne határidőre! Mindenesetre, számomra egy útvonal az, amit megteszek ha a konyhából ki akarok menni az illemhelyre! Ha a gépem előtt ülök és programozok, akkor a számomra a program nem egy útvonal, mert nem megyek vele vagy rajta sehova, hanem egy UTASÍTÁS-SOROZAT.

Nem vagyok hülye, ÉRTEM, mit akar kifejezni a professzorok definíciója, amit az egyetemen nyomnak. Képes vagyok ilyen szintű elvonatkoztatásra, nagyonis. Még azt is belátom hogy igazuk van, amennyiben lehet így is szemlélni és felfogni a dolgokat. De ANNAK, aki ténylegesen meg akar írni egy konkrét programot, annak a számára messze gyümölcsözőbb, ha nem ilyen absztrakciós magasságból szemléli a dolgokat, hanem GYAKORLATIASAN. Ami azt jelenti hogy van a masinája, ami képes bizonyos alapvető műveletek végrehajtására, és ezekből kell összekokányolnia azt a sorozatot, ami a végén a beadott bemenő paramétereket megcsócsálva remélhetőleg kiköpi neki a megfelelő eredményt!

És kész. Minden egyéb bizonyos szempontból felesleges. Már amiatt is, mert amit a jó professzorok ezen elvont megközelítés által kiöltöttek mint „programhelyességi bizonyítást”, az olyasmi, amiről maguk is elismerik, hogy az esetek csak elenyésző töredékében alkalmazható, mert olyan bonyolult és oly sokáig tart, azaz amikor mégis használható, azok annyira triviális esetek, amikor is e magas szintű matematikát nélkülözvén is többnyire belátható, hogy a kód helyes (vagy éppen helytelen).

De mert ezzel tömök a leendő programozók fejét, ezért azokba belekondicionálják azt a premisszát, hogy egy programnyelv, az valami rém bonyolult dolog, a leprogramozása, na az aztán meg pláne!

A másik ok, amiért e tévhit kialakult, az az, hogy eleinte a programnyelvek compilerei illetve interpretereik valóban bonyolultak voltak. Ennek oka főleg az volt, hogy a felmerülő problémákra nem alkalmazhatták akkoriban még a legkézenfekvőbb és legegyszerűbb megoldásokat, mert azokhoz több memória vagy nagyobb műveleti sebesség kellett volna, mint ami akkoriban rendelkezésre állt az azidőtájt létezett gépeken. Az akkori gépek lehetőségeihez alkalmazkodtak tehát a programnyelvek megvalósításai is, ezért mindenféle bonyolult és ravasz kerülőutakra kényszerültek, hogy egyáltalán működjenek valahogyan.

Mondok egy konkrét példát is. Akkoriban többnyire „batch” üzemmódban dolgoztak a gépek. A programozó levitte a lyukkártyacsomagot a gépterembe, ott beolvasták, ha sikerült a végrehajtás akkor minden oké, ha nem, akkor kapott egy nagy leporellólistát a felmerülő hibákról, ezt elvitte a szobájába, átnézte, és igyekezett kijavítani a hibákat. Ez nyilván lassú folyamat. A program beolvasása a lyukkártyákról sok idő, s a program futása is a gépen drága. Alapvető szükségesség volt minden futtatási kísérletből kinyerni a lehető legtöbb információt, azaz ha az értelmező hibát talált a forráskódban, nem állt le azonnal hanem igyekezett megvizsgálni a kód hátralevő részét is amennyire tudta, hogy minden ott esetleg még fellelhető egyéb hibáról is tájékoztassa a programozót, aki ezekből minél többet kijavít majd a következő futtatási kísérletig.

Na most, a mi esetünkben erre semmi szükség. A programozó manapság szinte mindig a saját gépe előtt ül, a programot is akárhányszor elindíthatja, ez nem jelent sok plusz költséget vagy időt. A legegyszerűbb tehát, ha a compilerünk (vagy interpreterünk) minden hiba esetén azonnal megáll (miután kijelezte azt az egyetlen hibát, amit épp megtalált), s nem is foglalkozik a kód hátralevő részével. Ez roppant mértékben leegyszerűsíti a program szerkezetét. Ráadásul így elkerülhetőek azok a kezdő programozókat frusztráló jelenségek, hogy valahol a kód elején van valami jelentéktelen szintaktikai hiba, esetleg több igazi hiba nincs is a kódban ezen kívül, de a hülye fordítóprogram azért végigvizsgálja a teljes további kódot, ám mert ezen első hiba miatt mondjuk nem jött létre egy változó, annak hiányát jelzi még 158 különböző további kódsorban is, s erre a szerencsétlen programozó, aki meglátja e hosszú listát, elszörnyed, hogy milyen bugos programot írt, sose tudja majd e sok hibát kijavítani!

Holott csak *egyetlen* hibát vétett a legelején, egyetlenegyet és nem többet, mondjuk félreütött egy karaktert. Messze logikusabb, ha ekkor csak azt az egy sort jelzi ki a program, aztán le is áll.

Mi tehát e könyvben egy EGYSZERŰ programnyelvet fogunk megvalósítani, és ezt is a lehető legegyszerűbben.

Valójában mint majd látni fogjuk, a programnyelvírás nemcsak nem különösebben nehéz, de annyira hihetetlenül könnyű, hogy rögtön már most, a bevezetésben elárulom, mi a legislegfontosabb és nélkülözhetetlen emberi tulajdonsága annak a személynek, aki efféle tevékenységre adja a fejét! Nos, a legfontosabb emberi kvalitása a hihetetlen, elképzelhetetlen, mindenekfeletti és eszméletlenül hatalmas **LUSTASÁG** kell legyen!

Ezt komolyan mondom, tényleg minden vicctől mentesen, ezt nem lehet ugyanis eléggé erősen kihangsúlyozni! E „szakmában” a lustaság igenis kifejezett ERÉNY, ami **ténylegesen** és bizonyíthatóan azzal jár, hogy a programozó **jobb** minőségű kódot hoz létre, sőt az se kizárt, hogy jóval HAMARABB, mintha szorgalmas volna!

Nemegyszer belefutottam ugyanis abba a helyzetbe, hogy a program írása közben valami bonyolult részhez értem. És volt, hogy ilyenkor nekiálltam, és **szorgalmasan**, elszántan, fogcsikorgató makacssággal addig gyűrtem-gyömöszkölttem a problémát, míg végül megoldottam!

Na és ilyenkor később mindig az derült ki, hogy esetleg nem is sikerült megoldanom, csak hittem azt hogy sikerült, mert a bonyolult problémát ugyebár érthetően csak bonyolultan sikerült megoldanom, ami azzal járt hogy egyes ritkább helyzetekben a kód nemvárt módon viselkedett, hülyeséget csinált, azaz rossz volt. De ha kétséget kizáróan jó is lett a megoldásom minden elképzelhető esetre, akkor is rosszat tettem ezzel, hogy szorgalmas voltam, amiatt, mert a programnyelv megvalósítása igenis EGYSZERŰ feladat kell legyen, hiszen mint fentebb kifejtettem, alapvetően csak egy primitív szövegfeldolgozásról van szó, amennyiben tehát mégis valami agyszikkasztóan bonyolult részhez érek, az mindig és KIVÉTEL NÉLKÜL annak a jele kell legyen, hogy valamit nem gondoltam át alaposan az előzetes tervezés során!

Na és ilyenkor hiába oldom meg nagy nehezen a bonyolult feladatot ott és akkor annál a konkrét résznél JÓL de bonyolultan, maga a probléma rossz megközelítése továbbra is fennmarad, és újra meg újra vissza fog köszönni a program fejlesztésének későbbi stádiumaiban is, azaz megint és újra és megint újra bonyolult megoldásokat kell leprogramoznom, ami mind teli van rengeteg hibalehetőséggel, a munka lassan halad - érted, ember, *azért* halad lassan mert SZORGALMAS vagyok! - ráadásul az elkészült programnyelv értelmezője vagy compilere is lassan működik majd, mert bonyolult rutinok végrehajtása nyilván lassabb, mint az egyszerűbbeké! És még ha mindebbe bele is nyugszunk, akkor is az lesz, hogy mert a létrejött megvalósítása a programnyelvednek bonyolult, emiatt ha később valamivel bővíteni akarod a nyelvedet, egyszerűen nem látod majd át, hogy mit hova kell beszúrni, mit kell megváltoztatni, mert elfelejtetted addigra a bonyolult szerkezetét, belegabalyodol az áttekinthetetlen algoritmusokba, még akkor is, ha minden kódsorhoz tíz sornyi megjegyzést írsz magyarázatként. Minden apró változtatás a későbbiekben akkora munkát jelent majd neked, mintha újraírnád a programod háromnegyedét.

A programnyelv írója tehát LUSTA KELL LEGYEN. Alapelvnek tekinthető, hogy akármennyit törje is a fejét a leendő nyelvén, de amikor leül KÓDOLNI, amikor már a gép előtt ül tehát, akkor kivétel nélkül rutinból kell dolgoznia, egyszerűen benyesni a kódot egy laza csuklómozdulattal. Előtte töprenghet akármennyit, de kódolás közben már nem. Nehezen kódolható, nehézkesen implementálható megoldások nem elfogadhatóak, ha ilyenre szükség van, valamit rosszul gondolt át előzőleg. Ha tehát úgy érzi ilyesmire volna szüksége, egyszerűen álljon fel, hagyja a francha az egészet, és ölelje meg a feleségét vagy barátnőjét hogy kikapcsolódjon és ihlete támadjon. S mert egy programnyelv megvalósítása TÉNYLEG egyszerű feladat, emiatt nem is sokára hőtzticher hogy eszébe fog jutni valami, ami a probléma olyan megközelítése, mely tizedannyi kódsorból is vígan megvalósítható! S akkor ámulni fog, hogy „hát hogy a csudába is nem ez jutott az eszembe már legelőszörre is, hiszen NYILVÁNVALÓ, hogy ennek így kell lennie”!

Általában véve, egy programnyelv interpretere vagy compilere sok kis apró vicikvacak rutinból kell álljon, meg egy rakás táblázatból. A táblázatok bár lehetnek hosszúak, de mind egy kaptafára mennek. A sok kis rutin pedig mind akkora kell legyen, hogy C vagy C++ nyelven megvalósítva mindegyik úgy nagyjából 10-15 kódsoros legyen csak (nemritkán ennél is sokkal kevesebb), egy sorba legfeljebb 3-4 utasítást írva de már ennyit is csak igen ritkán; nagyon-nagyon ritkán fordulhat elő csak olyan eset, hogy ennél hosszabb legyen egyetlen rutin, sőt, erősen kérdéses hogy egyáltalán *szabad-e* előfordulnia *bármikor is* olyan esetnek, hogy ennél hosszabb legyen! Az biztos, mintegy ökölszabályként, hogy ha BÁRMELY rutinunk is annyira hosszúra nő, hogy nem fér bele egyetlen képernyőoldalba, azaz nem vagyunk képesek áttekinteni az egészet a monitoron egyetlen pillantással, továbbgörgetés nélkül, akkor ott már „vagynak ám” nagy gondok, és „ideje elkezdenünk LUSTÁNAK LENNI”...

Ezen „egyképernyőoldalas szabály” alól tulajdonképpen csak egyetlen kivétel van: amikor olyan rutint írunk, ami rengeteg különféle típusú adatot kell kezeljen, ezt többnyire valami **switch** szerkezettel oldja meg, na és hát ha sok az ilyen típus, akkor azok mindegyike igényelni fog minimum egy sort! Ezesetben könnyen előfordulhat, hogy az elkészült rutin „kimászik a képernyőnkől” a hosszúsága

miatt, de olyan NAGYON sokkal azért ekkor se szabad kilógnia onnan, s vég-eredményben így is áttekinthető marad, mert a sok sor többsége benne mind egykaptafára megy. Többnyire mind valami „case” ágnak felel meg. Efféle eset is azonban inkább csak az aritmetikai kifejezés kiértékelő rutinoknál szabad előforduljon.

Azaz a programnyelv írójának ez kell legyen a jelmondata:

„Ha nem vagyok elég lusta, sose fogok végezni vele...”

Tehát, minden olyan gondra, ami kicsit is komolyabb nehézséget jelent a számod-ra, az a „végső megoldás”, az „overkill”, sőt, nem is „végső” megoldás, hanem az, amit már rögvest legelőszörre is bevetsz, hogy egészen egyszerűen legyintesz egyet és „belustulsz”.

Más szavakkal, mert tényleg mélyen hiszek abban, hogy ezt nem lehet eléggé ki-hangsúlyozni: **„Ha valami olyan gond merül fel, amit csak nehezen tudnál megoldani, akkor biztos lehetsz abban, hogy az olyan gond, amit NEM IS KELL MEGOLDANOD!”** Azaz, akkor egészen máshol kell keresned a probléma gyökerét, nem ott, ahol a súlyos nehézség felmerült.

1. fejezet: Milyen nyelvet készítsünk?

A programnyelveket többnyire abba a két kategóriába sorolják be, hogy „inter-preter típusúak” vagy „compiler típusúak”.

A compiler típusúak a forráskódból létrehozzák a megfelelő bináris állományt, mely azonnal végrehajtható a megfelelő számítógéparchitektúrán, mert a létrejött bináris kódokat azonnal képes értelmezni a processzor. Ennek nyilvánvalóan az a hátránya, hogy a program nem azonnal hajtható végre, hanem előbb le kell fordítani, aztán ugyebár tényleg csak azon a gépen végrehajtható, amelyre le lett fordítva a forráskód; ha a forráskód elveszik vagy nem adják oda neked (zárt forráskódú programok...) akkor iszonyú nehezen módosítható a végrehajtható állomány. Előnye viszont, hogy a létrejött bináris program tényleg olyan sebesen fut, amire csak képes az adott gép processzora.

Az interpreter típusú nyelv nem hoz létre bináris kódot: amint elér egy utasítás-hoz, azt azonnal végrehajtja. Itt ugyebár előny, hogy kvázi bármiféle gépen fut majd a program, ha azon rendelkezésre áll az adott programnyelv értelmezője; a forráskód ugyanaz mint a végrehajtható kód azaz bármikor módosítható; s természetesen le sem kell külön fordítani az első futtatás előtt. Hátránya is van azonban: ha egy utasítás egy ciklus belsejében mondjuk 10 ezerszer hajtódik végre, akkor tízezerszer fogja ellenőrizni hogy annak helyes-e a szintaktikája, holott ha előszörre helyes volt, akkor az összes többi ellenőrzés teljesen felesleges már. Ha a programban sok megjegyzés van elhelyezve, azokat mindig át kell ugornia ami szintén idő. (pláne cikluson belül). És még rengeteg effélet fel lehetne sorolni hátrányként.

Nem is tagadom épezzért, hogy alapvetően sokkal jobban kedvelem a NEM interpreter típusú nyelveket.

Az interpreterek ezen hátrányai annyira nyilvánvalóak, hogy számos olyan programnyelv született, melynek elkészítették mind a compilerét, mind az interpreterét! Sajnos, programnyelve válogatja, mennyire nehéz hozzá készíteni akár interpretert, akár compilert. Többnyire az a helyzet, hogy amely programnyelv eredetileg compiler célra íródott, annak interpretert írni általában rém macerás munka, és csak sok kompromisszummal lehetséges. De azért akadnak effélék, bár még ezek is gyakorta trükköznek: gyorsan lefordítják a forráskódot amit azonban nem írnak ki a lemezre hanem a memóriában tárolják, és azt hajtják végre. Olykor nem valami konkrét processzortípusra fordítanak, hanem valami „átmeneti állapotra”, valami „köztes nyelvre”, ami ugyan nem végrehajtható közvetlenül a processzor által, de sokkal könnyebb értelmezni, mint az eredeti forrásszöveget. Egy példa: legelőször végigszalad az interpreter a forráskódon, s megkeresi, hol vannak benne címkék. Mindegyik címkének eltárolja a nevét és címét egy belső táblázatban. Amikor aztán ugróutasításhoz érkezik, nem kell minden alkalommal végigbogarászni a teljes forráskódot hogy megtalálja a címet a címke alapján ahova ugornia kell, hanem csak a maga kis belső táblázatában kell megkeresnie a címet, s azonnal odaugorhat. Könnyű belátni, ez micsoda hatalmas időnyereséget jelent, ha a forráskód nagy, ráadásul gyakran kell benne ide-oda ugrándoizni, pláne valami sokszor végrehajtott cikluson belül!

Ez már átvezet bennünket ahhoz a programnyelvtípushoz, amit nem szoktak megemlíteni általában, holott én igenis külön programnyelvtípusnak tartom mégis: ezek a „virtuális gépek”!

A virtuális gép azt jelenti, hogy a programnyelvnek lényegében 2 különböző megjelenési formája van, de mindegyik mégis ugyanaz. Vagyis, létezik egy „magas szintű” változata, ezen írja a programozó a tulajdonképpeni programot. Ezt aztán az „interpreter” előbb lefordítja, mintha nem is interpreter de compiler lenne, ám nem egy létező processzortípus assembly nyelvére fordítja le, hanem egy „elképzelt” processzor assembly nyelvére, ami lehet egyszerűbb vagy bonyolultabb, attól függően mit talált ki a nyelv írója, de mindenféleképpen messze sokkal egyszerűbb mint az, amin a programozó a kódot írta. Aztán ezen elképzelt assembly nyelv utasításait kezdi el az interpreter végrehajtani. Azaz mindenféleképpen olyan egyszerű kell legyen ez a nyelv, hogy bár interpreter hajtja végre a kódokat, de a sebessége ne legyen sokkal lassabb, mintha igazi processzoron futna ez a virtuális assembly nyelv.

Ezen utóbbi megoldás, mint látjuk, megpróbálja ötvözni a compiler és interpreter típusú nyelvek előnyeit. Aztán persze hogy ez mennyire sikerül, az nagyon erősen függ attól, egyáltalán mi is a nyelv célja azaz miért hozták létre, mit akarnak vele csinálni, mennyire bonyolult az „igazi” nyelv amin a forráskódot írják, mennyi idő alatt várják el hogy létrehozza belőle a virtuális nyelv assembly kódját, s persze hogy mennyire ügyes a programozó aki ezt az egész mindenséget végül megvalósítja, leprogramozza!

Elárulom, ez a módszer nem is valami új dolog. Igazából annyira ősi, hogy már a réges-régi C-64 -es számítógép idejében is felbukkantak a csírái! Amikor ugyanis abba beleírtál egy sort annak BASIC programnyelvén, nos, NEM AZ tárolódott el a memóriában, amit beleírtál, hanem abban a pillanatban hogy megnyomtad a „RETURN” gombot, a gép úgymond „tokenizálta” azt, azaz megkereste benne az összes ismert utasítás kulcsszavát, amik akár jó sok betűből is állhattak, s ezek helyett beillesztett oda egy mindössze 1 bájtos kódot, úgynevezett „token”. Később, amikor végre kellett hajtania a programot, onnan ismerte fel hogy efféle tokenről van szó, hogy ezek mindig nagyobbak voltak számértékben mint 127. (Azaz a bájt legfelső bitje mindig 1 volt). De ebből te mit se vettél észre, mert mindig amikor kilistázta neked a programot, ha efféle tokenhez ért, a token helyett azt a stringet listázta ki, aminek e token a kódja volt.

Na most ez tipikus viselkedése a virtuális gépeknek, egy trükk az ő repertoárjukból.

Hogy ezeket végiggondoltuk, meg kell hoznunk a legelső fontos döntést, mely legalapvetőbben befolyásolja majd a teljes további munkánkat! Milyen típusú programnyelvet készítsünk?!

Nos, a döntés nem hiszem hogy túl nehéz volna. Gondoljunk arra, ha amolyan igazi „őseredeti” compiler típusút készítünk, akkor mindenekelőtt nem elég hogy ismernünk kell a C és C++ nyelvet (vagy tágabb értelemben valamiféle akármelyik másik „magas szintű” programnyelvet amin megírjuk a magunkét), de ismernünk kell azt az assembly nyelvet is, amire a compilerünk lefordítja a magunk programnyelvét. Bár nem logikátlan feltételezni, hogy aki effélére adja a fejét mint a programnyelv írás, az ismer már legalább egy assembly nyelvet valamennyire, de az biztos, hogy semmiképp se ismerheti az összeset. Amennyiben azt akarjuk hogy leendő programnyelvünk hordozható legyen, e módszer nem követhető, mert úgyse írhatjuk meg minden platformra a compilerét.

Elvetendő azonban az a módszer is, hogy amolyan „igazi” interpretert írjunk, a forráskód mindenfajta előzetes optimalizálása/feldolgozása nélkül. Iszonyatosan lassú lenne.

Igenám, de ha meg virtuális gépet készítünk, lényegében dupla munkát kell végeznünk, mert ez esetben két nyelvet is ki kell dolgoznunk, a „magas szintű” forrásnyelvet és a virtuális assembly nyelvet, utóbbira meg kell írni az értelmezőt, előbbire meg azt ami mint compiler lefordítja a magunk assembly nyelvére! Nem túl kecsegtető kilátás, pláne mert vörös felkiáltójellel villog a szemünkbe a „Bevezetőben” kihangsúlyozott figyelmeztetés, hogy LUSTÁNAK kell lennünk! És ugye nyilvánvaló hogy aki kétszer dolgozik egyetlen feladat megoldásáért, az nem nagyon tesz eleget a lustasági kíváncsiságnak!

A fentiekben tűnődve, én végül úgy döntöttem (de az Olvasó persze dönthet másképp a maga nyelvének megalkotásakor, ám az esetben is célszerű elolvasnia e könyvet, hasznos ötletek fellelése érdekében) hogy én bizony igenis virtuális gépet készítek, ám olyat, ami mégis eleget tesz a lustasági szabálynak, amennyiben nem kell dupla munkát végeznem, hiába hogy ez egy virtuális gép!

Miként is érhetem el ezt?

Természetesen úgy, hogy a nyelv amit készíték, elegendően egyszerű lesz ahhoz, hogy ne kelljen külön lefordítani a virtuális assemblyre a végrehajtás érdekében, hanem igenis végrehajtható legyen maga a forrás!

Azon Olvasóim, akik esetleg már szereztek valamelyes tapasztalatokat valami már létező assembly nyelv kapcsán, e fenti mondatomat olvasva bizonyára felhőrdülnek, hogy hát **hé, ácsi, nem erről volt szó!** Ők azt hitték, valami MAGAS SZINTŰ programnyelv megalkotását írom le, nem holmi elképzelt assemblyét!

Sietek megnyugtadni mindenkit, hogy erről SZÓ SINCS, igenis megfelelően magas szintű lesz a programnyelv ahhoz, hogy efféle vád ne érhesse bennünket. Nyilván persze kell majd bizonyos kompromisszumokat kötni, de rögvess előre bocsátom itt és most, hogy nyelvünk alkalmas lesz minden olyasféle feladatra, amire mondjuk a C nyelv maga is alkalmas, márpedig azt cseppet se tartja már senki se assembly nyelvnek! Vagy hogy egy a kezdők számára talán szimpatikusabb nyelvet mondjak, ott a BASIC nyelv. Nyelvünk sokkal többet fog tudni, mint a Basic.

Egyszerűen arról van szó, hogy már a tervezése közben szem előtt tartjuk azt a kívánalmat, hogy ne kelljen külön lefordítani a forráskódot valami egészen más nyelvi formába, azért, hogy végrehajtható legyen. Tulajdonképpen ezen döntésünk miatt az egész programnyelvkészítési feladat ráadásul sokkal IZGALMASABB is lesz, mintha másként döntenénk, azért, mert miközben haladunk előre párhuzamosan a nyelv megvalósításában és tervezésében egyszerre, aközben mintegy „kitapogatjuk”, hol van az a végső határ, ameddig elmehetünk egy programnyelv bonyolultságában akkor, ha közben ragaszkodunk ahhoz is, hogy azonnal végrehajtható legyen, előzetes fordítás nélkül! Ezt persze úgy is tekinthetjük, hogy megpróbáljuk megvalósítani a „legbonyolultabb” vagy inkább „legmagasabb szintű” assembly nyelvet, de úgy is tekinthetjük, hogy létre akarjuk hozni azt a magas szintű nyelvet, ami még éppen közvetlenül végrehajtható.

Amennyiben így teszünk, azt a roppant előnyt is megszerezzük, hogy nyelvünk alkalmassá válik arra, hogy könnyen lehessen rá írni igazi compilert bármi ténylegesen létező processzorra, architektúrára, s még sokkal inkább alkalmas lesz arra is, hogy mégse efféle compilert írjunk, hanem olyat, ami egyszerűen C nyelvre fordítja a programunkat, mármint C forráskódra, azt meg igazán könnyű binárisra átfordítani, mert mint a Bevezetőben is említettem, C compiler már létezik minden kicsit is komolynak számító számítógép-architektúrára.

Miután ezt eldöntöttük, következne a sok-sok többi részlet, ami arról szól, milyen is legyen a leendő nyelvünk, a „mau” programnyelv, például hány karakteresek legyenek benne maximum a változónevek, legyen-e benne számított ugróutasítás, egyáltalán a programsorok számozva legyenek-e mint a BASIC esetén, vagy nem, miként hívják az egyes utasításokat, például legyen-e olyan ciklusunk aminek az a neve hogy WHILE vagy magyarkodjunk, s ennek az legyen inkább a neve hogy CIKLUS?! Elfogadhatóak-e az utasításnevekben az ékezetes karakterek? Szabad-e egy sorba több utasítást is írni, és ha igen, ezeket kötelező legyen-e el-

választani pontosvesszővel vagy más karakterrel, avagy elég oda a whitespace is (szóköz vagy TAB)? És így tovább...

Ezzel itt most mind NEM foglalkozom, azért nem, mert e felmerülő kérdések jelentős része magától megszűnik, amint így vagy úgy döntünk valamely más kérdésben. Akadnak ugyanis kérdések, melyeket így vagy úgy megválaszolva, máris behatárolják valami másik kérdésnél is a lehetséges alternatívákat. Azaz, teljesen felesleges időpocsékolás lenne törni a fejünket ezeken a dolgokon, míg el nem jutunk odáig, ahol már muszáj is lesz döntenünk róluk így vagy úgy.

Most tehát lépünk előre a következő fejezetre, s kezdjük bele a program tervezésébe, s látni fogjuk, miként épül fel mintegy magától a váza, sőt szinte az egész kód is, ha úgy fogunk neki e feladatnak, ahogyan azt illik: a strukturált programfejlesztés és a moduláris programtervezés módszertanával!

2. fejezet: A program vázának és főbb funkcióinak áttekintése

Mindenekelőtt tudjuk jól, hogy ez aminek nekiállunk, szép nagy program lesz! Hogy ne kelljen annyit írkálnunk (emlékezzünk csak, nekünk MUNKAKÖRI KÖTELESÉGÜNK lustának lenni!) csináljunk egy **vz.h** nevű headerfájlt, pár fontosabb **#define** direktívával részben a hordozhatóság, részben a kevesebb gépelés érdekében:

```
#define MauInterpreterVersionNumber 0

#define SPACE 32
#define TAB 9
#define SORVEG 10

#define USI unsigned short int
#define USIL unsigned int
#define USC unsigned char
```

Remek, menjünk tovább.

Induljunk ki abból, miként is használjuk majd a mi mau interpreterünket! Valamiféle szövegszerkesztővel megírjuk a mau nyelvű programunkat, aminek legyen a neve mondjuk „progi”. Célszerűen ezt egy **progi.mau** nevű fájlba elmentjük. Ezután a futtatása úgy történik majd, amint az szinte hagyomány a linuxos berkekben (de Windows alatt is nagyon hasonlóan működne):

```
mau prog1.mau
```

Ha kicsit intelligensebbnek írjuk meg a **mau** nevű interpretert, megoldhatjuk azt is, hogy a proginkat úgy nevezzük el hogy „progi”, kiterjesztés nélkül, ezt is simán fel tudja dolgozni, de ha nem talál „progi” nevű állományt, keressen rá arra, van-e olyan hogy „progi.mau”, s akkor azt a fájlt olvassa. Ez már részletkérdés és csicsa, a tulajdonképpeni programnyelvíráshoz nem sok köze van. Ilyesmivel akkor kell foglalkoznunk, ha már minden egyéb készen van.

Lényeg az, hogy az interpreter kap egy állománynevet, s ezt be kell olvassa a memóriába. Ehhez persze meg kell vizsgálnia, egyáltalán létezik-e ez az állomány, mekkora a fájl, le kell foglalja neki a megfelelő nagyságú memóriaterületet, oda aztán beolvassa az egészet, majd elkezdi végrehajtani. A program váza az eddigiek szerint eddig így néz ki:

- Van megadva állománynév? Nincs: » hibajelzés, leállítás
- Létezik a megadott nevű állomány? Nem » hibajelzés, leállítás
- Állományméret meghatározása
- A szükséges memóriaterület lefoglalása
- Az állomány beolvasása a lefoglalt memóriaterületre
- Végrehajtás

Akármilyen elnagyolt vázlat is ez, már a fentiekből is látható, hogy időnként szükséges mindenféle hibaüzeneteket kiíratnunk. Pláne, mert előre tudható, hogy felmerülhetnek további gondok is, már az állomány beolvasásánál, például ha az létezik ugyan, de nem megnyitható, mert például nincs jogunk olvasni azt a fájlt; vagy ha a fájl mérete túl nagy, s emiatt nem áll rendelkezésünkre kellő nagyságú memória ahhoz, hogy beolvassuk. Szóval, már a legelején meg kell küzdenünk a hibajelzések problémakörével!

Picit továbbgondolva a témát, rájövünk, hogy nemcsak hibaüzeneteket kell tudnunk kiírni, de nagyon hasznos lehet, ha időnként olyasmit is kiírogathatunk, ami nem hiba ugyan, de valamiért fontosnak tartjuk a Felhasználó tudomására hozni. Mindezt úgy mondják szaknyelven, hogy jó, ha tudunk *logolni*. Ennek egy speciális esete lehet az, amikor kifejezetten épp hibaüzenetet logolunk.

Jó lesz e logoló rutint már most a legelején megalkotni, hogy aztán egyszersmindenkorra megfeleldkezhessünk róla. A dolog persze kicsit trükkös, mert az is előre tudható, hogy lesznek olyan pillanataink, amikor nemcsak egy egyszerű stringet akarunk kiírni, hanem mindenféle rendszerváltozókat is, s ezek száma hol ennyi-hol annyi, a típusaik is lehetnek mindenfélék, s ezt mind nem láthatjuk előre. Logoló rutinunkat tehát úgy kell megtervezni, hogy a printf -hez hasonlóan képes legyen fogadni változó számú paraméterlistát.

E rutint célszerű egy teljesen külön fájlba elmenteni, mert a világon semmi köze a tulajdonképpen mau programnyelvhez. E fájlnak én a logol.cpp nevet adtam, s így néz ki a tartalma:

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

extern char *mktimes(char *fmt);extern FILE *stdlog;extern char logflags[256];

char logfile_idoformat[]="%Y.%m.%d %H:%M:%S";

void L(const char * format, ...) {if(logflags[0]==0) return;

    va_list args; va_start (args, format);
    fprintf(stdlog,"LOG:> %s : ",mktimes(logfile_idoformat));
    vfprintf (stdlog, format, args); fprintf(stdlog,"\n");
    va_end (args);
}
```

Igazán nem valami szörnyű hosszú, ugye?

Az elején szerepel 3 darab „**extern**” deklaráció. Ez azt jelenti, hogy ezek a változók illetve függvények más fájlokban vannak igazából meghatározva. Az **mktime**s természetesen egy függvény, ami arra szolgál, hogy az aktuális rendszeridőt emberi fogyasztásra alkalmas alakúra formázza, azaz visszaadja azon string szerint, amivel a formátumot meghatároztuk neki. Illik ugyanis, hogy a logolásnál minden üzenet kapjon egy úgynevezett „időbélyeget”, hogy tudjuk, mikor is történt az az esemény, amiről tájékoztat minket a programunk.

Az **stdlog** lesz a neve annak a fájlnak, amibe a logolás történik. Ezt természetesen a főprogram nyitja majd meg minden egyéb tevékenység előtt, rögvést az elindulásakor, de ezt teljesen felesleges beleírni most a legelején, hiszen amíg fejlesztjük, úgyis az a jó ha rögvést a képernyőn látunk minden üzenetet, ezért amíg csak vége nem lesz a programunk fejlesztésének, ezt egyszerűen elintézzük majd ennyivel a main.cpp fájlban valahol e fájl elején:

```
FILE *stdlog;  
.....  
stdlog=stdout;
```

Természetesen gondolnunk kell arra is, hogy amikor kilépünk a programból EXIT_SUCCESS vagy EXIT_FAILURE értékkel, az esetben le kell zárni a logfájlt, amennyiben az nem az stdout vagy az stderr volt mégsem, emiatt meg kell írunk a kilépőrutinjainkat ezen esetekre:

```
void EXITFAILURE(void) {if((stdlog!=stdout)&&(stdlog!=stderr)) {fflush(stdlog);fclose(stdlog);}  
exit(EXIT_FAILURE);  
}  
  
void EXITSUCCESS(void) {if((stdlog!=stdout)&&(stdlog!=stderr)) {fflush(stdlog);fclose(stdlog);}  
exit(EXIT_SUCCESS);  
}
```

Elemezzük tovább a logol.cpp fájlt! Ebben látunk egy **logflags** nevű tömböt, ami 256 értékből áll. Ez azért van, mert értelmes dolognak tűnik különböző lehetőségeket biztosítani a logolásra. Például, egyszerűen letiltani minden logolást. Ezt végzi el e tömb nulladik tagja: ha ez 0 értékre van állítva, semmit se logol. A tömb többi értéke felhasználható arra, hogy parancssori kapcsolók segítségével értékekkel töltsük fel, aztán ha valamit logolni kell, a program figyeli, a megfelelő flag be van-e állítva. Ha igen, logol, ha nem: nem logol.

Tudom hogy 256 lehetőség logolási szintnek baromisoknak tűnik, de nincs értelme kevesebbet megadni, akkor ugyanis illene azt is ellenőrizni, az index ami meg van adva logolási szintnek, nem nagyobb-e mint a lehetséges maximális érték. Efféle ellenőrzés mind idővesztés. Ha a tömb 256 elemű, semmi szükség erre, mert nem csordulhat ki a tömb határaiból, ha az index csak egy unsigned char változó. És ez nem számottevő memóriavesztés, mert csak 256 darab bájt-ról van szó. Ellenben kapunk érte jókora szabadságot, hogy mit logoljunk és mikor.

Az **L** lesz a logoló függvényünk neve. Semmi értelme sokkarakteres nevet adni, ha a név több karakter, könnyebb eltéveszteni, és sokat kell gépelni. Kis **l** betűt viszont nem célszerű névnek adnunk, mert rosszul látszik, és könnyen össze-téveszthető az **l**-es számjeggyel.

Az **mktimes** függvény:

```
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>
#include <iostream>

extern void EXITFAILURE(void);extern char HET[];
static const char hetnapjai[][4]={"V ", "H ", "K ", "Sze", "Cs ", "P ", "Szo"};
char buf[129];

char * mktimes(char *fmt) { using namespace std;

time_t *ido;time_t rawtime;struct tm ideiglenes;
time(&rawtime);ido=&rawtime;ideiglenes = *localtime(ido);

if (!strftime(buf, sizeof(buf)-1, fmt, &ideiglenes)) {
cerr << "ERROR: strftime == 0\n";EXITFAILURE();}

HET[0]=hetnapjai[ideiglenes.tm_wday][0];HET[1]=hetnapjai[ideiglenes.tm_wday][1];HET[2]=hetnapjai[ideiglenes.tm_wday]
[2];HET[3]=0;
return buf;
}
```

Nem tévedés, hogy itt nem az **L0** függvényünket hívtuk meg hiba esetén! Ezen **mktimes** függvényünket az **L0** függvény hívja meg általában, azaz hiba esetén ő, az **mktimes** nem közölheti a gondját velünk a logoló rutin által, mert az végtelen ciklushoz vezetne. Ezért neki a baját közvetlenül a hibacsatornára kell írnia, máshova nem teheti.

Remek, okos fiúk vagyunk, tulajdonképpen megvan a programunk teljes kerete! Nyilván persze ezt lehetne cifrázni, feltupírozni, például egyelőre csak a lehetősége van meg a többszintű logolásnak, de nincs rutin arra, hogy ezeket beolvassuk valamiféle parancssori paraméterekből. Ez azonban nagyon várhat, mert a továbbiakban dől el majd az is, egyáltalán mi a csudát is akarunk majd logolni!

Hátra van még, hogy miként is olvassa be a programfájlt, meg ellenőrizgesse meg ilyesmi. Ám itt torpanjunk meg egy pillanatra, s vessük figyelő szemünket a jövőbe!

Képzeljük el azt a boldog időszakot, amikor hiperszuper és full-extra-de-luxe programnyelvünk már tökéletesen működik, mindenki bennünket csodál érte, és az egész világon szinte kizárólag őt használják, még a legbonyolultabb feladatokra is! Egészen biztos, hogy ezesetben programnyelvünk okvetlenül kerül majd olyan szituációba, amikor is valamely, e nyelven írt program a futása közben valamikor meg kell hívjon egy MÁSIK, szintén mau nyelven megírt programot...

Ezt le kell tudnunk kezelni, s jó erre már most a legelején gondolni, mert könnyű belátni, hogy ennek abszolút semmi köze a programnyelvünk egyéb szempontok szerint való milyenségéhez és mélyebb részleteihez. Ez olyan általános dolog, amit minden programnyelv kell tudjon. (Legalábbis illene, hogy tudjon...)

Annak a programnak, ami a hívó, s annak ami a hívott, egymástól függetlenül kell tudnia működni, nem zavarhatják egymást, ugyanakkor mégis kell hogy a hívó átadhasson bizonyos paramétereket a hívott programnak, s attól annak sikeres vagy sikertelen befejeződése után visszakaphasson valamiféle eredményeket, visszatérési értékeket. Na és hát sajnos az a helyzet, hogy ez nem igazán egyszerű feladat... Nemcsak programozástechnikailag nagyon bonyolult, hanem

már maga az is nehéz, hogy van pár lehetőség, amik közül dönteni, választani kell. Itt a legelső pillanat, amikor nagyonis célszerű lesz „belustulnunk”...

Az első ötlet, ami eszünkbe jut az, hogy hívja meg a mau interpreterünk önmagát a **shell** segítségével, a **system** parancsot használva, azaz ha az A.mau program hívni akarja a B.mau programot, akkor adja ki e parancsot vagy valami effélét:

```
system("mau B.mau (esetleges_paraméterek)");
```

E módszer kétségkívül működni fog, és ahhoz hogy megvalósulhasson, még csak semmi különös dolgot nem is kell tennünk, ugyanis mindenféleképp illik lehetőséget adnunk rá, hogy programnyelvünkől meghívható legyen tetszőleges rendszerparancs! Ha azt megoldjuk (ami pofonegyszerű programozástechnikailag, ezt előre borítékolhatjuk) akkor a mau interpreter nyilvánvalóan meghívhatja eképp akár önmagát is, ez tiszta sor! Ezzel tehát itt és most nem is kell foglalkoznunk.

A baj az, hogy ez nekünk cseppet sem elégséges lehetőség! Azért nem elég, mert először is, eképp csak string típusú paramétereket adhatunk át a B.mau programnak, holott ha mondjuk afféle „librarykat” akarunk írni a mau nyelvhez, akkor elengedhetetlenül szükséges nemcsak string, de akármiféle más típusú paraméterek átadhatósága is! Ennél is nagyobb gond, hogy a B.mau semmiféle adatot nem adhat át az A.mau programnak, legfeljebb egyetlen aprócska infót a shellen keresztül, hogy ő maga a B.mau rendben fejeződött-e be, vagy hibásan ért véget a futása! Amely pillanatban ennél több adatot óhajtanánk kinyerni belőle, mindenféle ravasz kerülőutakra kényszerülnénk: például hogy a B.mau a maga outputját írja egy ideiglenes fájlba, amit aztán az őt hívó A.mau proggi beolvas — ez nyilvánvalóan nem szép megoldás, ráadásul iszonyatosan lassú is, helypazarló is! Vagy pedig a B.mau programnak a **DBus** üzenetküldő alrendszeren keresztül továbbíthatunk üzeneteket, paramétereket, s nyilván az ő válaszait is ezen keresztül kaphatjuk meg!

Ez az utóbbi megoldás bár nyilvánvalóan profibb, mint az ideiglenes fájlok használata, de van egypár óriási hibája! Sorolom.

1. Abban a pillanatban hogy ebbe az irányba kötelezzük el magunkat, a mau nyelvünknek FÜGGŐSÉGE lesz a DBus. Na most ha netán van valami, amit jobban gyűlölök mint egy világháború vagy valamely nőrokonom megerőszkolása, akkor egészen bizonyos, hogy az a FÜGGŐSÉG! A FÜGGŐSÉG a Linux-programőkoszisztéma legnagyobb rákfeneje, messze több szívást, gondot, bajt, idegességet, idegrángást, gyomorfekélyt és fogcsikorgatást okoz, mint az összes többi mindenféle probléma együttvéve! Az a szememben a LEGSÁTÁNIKUSABB FŐGONOSZ!

2. A DBus, az több mindenféle részből áll, még holmi démonból is, sőt, ahogy olvasom, egyszerre 2-féle démont is futtat, (forrás: <http://unixlinux.tmit.bme.hu/D-Bus>) és ez a számomra nem tűnik valami takarékos megoldásnak. Egyszerűen nem tartom megengedhetőnek, hogy egy esetleg picike mau program futtatásáért el kelljen indítani egy ekkora monstrumot. Nekem az a véleményem, hogy nem illik verébre atombombával lőni!

3. Legsúlyosabb érvként végül megemlítem, hogy én magam egyszerűen nem értek a Dbus programozásához! Tényleg fogalmam sincs róla, hogyan kell azt végrehajtani! Nyilván persze képes lennék megoldani ezt, meg tudnám tanulni, de LUSTA VAGYOK ehhez! Én a magam mau nyelvét akarom megalkotni, és nem örökké csak tanulni! Ha állandóan csak tanulok, sosem alkotok semmit! És ez hogy ilyen vagyok, itt előny kell legyen, hiszen mit is fejtegettem a Bevezetőben számos bekezdésen át: hogy előny lustának lenni, sőt, ez nemcsak előny, de alapvető követelmény!

Na de akkor mit csináljunk?!

Hát, malmozzunk az ujjainkkal, várjunk az isteni sugallatra... S lőn csoda! Hiszen rendelkezésünkre áll nemcsak a C nyelv, de a C++ is! Ami objektumorientált nyelv! Márpedig ha akad egyáltalán valami a programozásban ami méltó arra a névre hogy „objektum”, akkor az bizvást nem más, mint egy teljes, „mau” nyelven megírt program!

Ezen fellelkesülve, rögvést elhatározzuk, hogy egy mau nyelvű program nálunk mindig egy objektum lesz, amihez nyilván kell készítenünk egy „osztályt” a C++ programban. Ezt illik egy **mau.h** nevű fejlécállományban deklarálni. Nevezzük ezen mau nyelvű programok osztályát el úgy, hogy **PGM**, hogy megkülönböztethetők legyenek azon esetektől amikor más kontextusban emlegetjük a „program” szót, s máris írjuk bele a mau.h fájlba:

```
class PGM {  
.....  
}
```

Tehát leend nekünk máris egy „class PGM”. Ugye milyen klassz? Ez a „class” nyilván megoldja majd a fentebb ecsetelt súlyos gondjainkat, hiszen e megközelítésben a PGM osztálynak kell legyen valamiféle olyan metódusa, ami egyrészt beolvassa az adott forráskódot a memóriába, másrészt végrehajtja azt! Amikor tehát az A.mau meg akarja hívni a B.mau programot, akkor meghívja a B.mau konstruktorát, aminek paraméterként átadja a megfelelő állománynevet, erre az elkészíti a B.mau programpéldányt, ezután az A.mau meghívja a B.mau megfelelő metódusát ami a futtatást végzi, s ezt akár úgy is megírhatjuk, hogy képes legyen az A.mau programtól bizonyos paraméterek fogadására, illetve olyasmiket vissza is tudjon adni! Sőt, előre tudjuk, hogy mivel a C és C++ függvényeknek csak 1 visszatérési értékük lehet, de nekünk ez nem elég, hiszen a franc se tudhatja előre hogy épp hány eredményt akar majd az a B program visszaadni outputként, továbbá mert a visszatérési érték amúgy is kell nekünk annak jelzésére hogy a program egyáltalán sikeresen ért-e véget vagy hibajelzéssel, emiatt ez nem lehet másként, mint hogy a paramétereket amiket a B.mau kap, referenciaként adjuk át! Abban a pillanatban azonban hogy így döntünk, máris tudjuk, hogy barbár pocséklás lenne több paraméter átadásával lassítani a program működését (a paraméterátadás a veremtárban fog történni nyilvánvalóan, a veremműveletek pedig nem a gyorsaságukról híresek), hanem egyetlen paraméter referenciáját adjuk csak át, ami emiatt szükségszerűen egy struktúra lesz! Abban aztán elférnek akár egyéb mindenfélék referenciái is, vagy amit csak akarunk. Az tartalmazhatja nemcsak az input paraméterek, de az output adatok

helyét vagy mutatóit is. Nyilvánvaló persze, hogy kell valami hasonló struktúra a maga belső adatai számára is, mert sokkal kényelmesebb ha adott esetben egyszerre adhatunk át valamely függvénynek minden fontosabb adatot, mintha darabonként.

Nekünk kell tehát egy struktúra minden PGM osztályba, ami „minden fontos adatot” tárol. Nevezzük el ezt úgy, hogy „F”, azért e betűvel, mert ez tisztelgés drága barátom és tanítómesterem, Mr. Fossil Codger előtt. És azért egyetlen betű e struktúra típusa, mert ezt rém gyakran kell majd leírunk.

A PGM osztály deklarációja tehát így fest most:

```
class PGM {  
    struct F f;  
    .....  
    public:  
  
    PGM(USIL phossza); // konstruktor  
    ~PGM(); // destruktor  
    void beolvas(char *filename); // beolvassa a p memóriaterületre a megadott nevű programfájlt  
    void futtat(USIL honnan=0L); // futtatja a programot az adott pozíciótól vagy az elejétől  
    void folytat(void); // folytatja a program végrehajtását az aktuális pozíciótól  
}
```

A fentebb látható függvények szükségszerűsége nyilvánvaló. Minden osztálynak illik legyen destruktora például. Konstruktor is kell legyen, ami semmi mást nem kaphat paraméterül, mint azt a számot ami megmondja, milyen hosszú lesz (bájtban mérve) a leendő mau programnak az a memóriaterülete, ahova majd őt a programot beolvassuk, s mely memóriaterületet a konstruktor kötelessége lefoglalni. Kell aztán olyan rutin, ami a lefoglalt memóriaterületre beolvassa a megfelelő programfájlt, kell olyan is ami elkezd futtatni a progit onnan ahonnan kezdeni akarjuk, alapértelmezés szerint természetesen a legelejétől, s olyan is kell, ami folytatja onnantól, ahol valamiért abbahagyta korábban. Ez nyilván jó lesz, ha építünk bele valami debugger lehetőséget netán, lépésenkénti programvégrehajtást vagy ilyesmit. Ugyan gőzöm nincs róla még, ez miként lesz majd megvalósítva, de legalább adjuk meg erre a lehetőséget a későbbiekben, elvégre semmibe se kerül ez nekünk most még!

Ha azt emlegettük, a konstruktor le kell foglaljon valamekkora memóriát, rögvest tudjuk, hogy ennek kell egy mutató, meg kell valami változó ami megjegyzi ennek méretét, sőt, előre tudható hogy illik megjegyeznünk azon állomány nevét is amit beolvastunk mint mau forráskódot, bölcs dolog tárolni e név hosszát is, ráadásul vagyunk annyira előrelátóak hogy megsejtsük, kell majd nekünk még legalább egy memóriaterület, ami a mau program munkamemóriája lesz, az adatok számára! Ezt mind az F struktúrában illik tárolni. Ennek megfelelően:

```
struct F {  
  
    USC *p; // Ez a pointer mutat a programutasításokat tartalmazó memóriaterületre  
    USIL phossz; // A p terület hossza bájtban  
    USIL P; // Az aktuális pozíció a végrehajtandó programban, értéke mindig 0 és phossz közé esik, azaz 0 <= P < phossz !  
    .....  
    USC *m; // Ez a pointer mutat a program számára lefoglalt adatmemóriaterületre  
    USIL mhossz; // Az m terület hossza bájtban  
  
    char *programfileneve; // A végrehajtandó program fájljának neve  
    int programfilenevehossz; // A programfile nevének a hossza  
    USIL programfilehossz; // A programfile hossza bájtban  
};
```

Előre tudjuk persze, hogy ennyivel nem ússzuk meg, de ezek mindenképp szükségesek az biztos. Később majd bővítjük az F struktúrát.

Ez mind világos is, legfeljebb az a kérdés merülhet fel Olvasómban, mi a fenét bohóckodom én azzal, hogy a memóriaterületeket USC, azaz mint a **vz.h** fájlban meghatároztam e rövidítés jelentését, „unsigned char” típussal definiáltam? Miért nem jó nekem a közönséges „char” változó?!

Nos azért, mert nem óhajtok félreértést vagy kétértelműséget. A C nyelv a kedvencem, nagyon szeretem, de azért akad benne pár dolog amit nem tartok szerencsésnek. Egyik például, hogy a hexadecimális számjegyek sorozatának kezdetét nem egyetlen karakterrel jelöli, hanem kettővel: „0x”. Erre a kérdésre majd később még vissza is térek. A másik, ennél súlyosabb dolog a szememben, hogy a „char” típus nála lehet előjeles is, és előjel nélküli is! Sőt, alapértelmezettként kifejezetten előjeles!

Na most ezzel egyszerűen nem tudok megbékélni. Számomra a char, az egy 0 és 255 közötti POZITÍV egész szám, ami bizonyos körülmények közt megfeleltethető egy karakter (ASCII) kódjának is, vagy akármi másnak, amiből nincs több, mint legfeljebb 256 darab. Egy ilyen kis intervallumú változónál semmi értelmét nem látom hogy a tartomány felét negatív értékek jelzésére használjuk fel. Egyetlen esetben van csak ennek jelentősége; ha valami gépi kódú utasítássorozatnál rövid távú ugrást akarunk végrehajtani, s ott jelzi az ugróutasítás után, hogy előre ugrunk-e vagy hátra (visszafelé) - utóbbi esetben negatív a jelzett távolság értéke.

Na de ez rémségesen speciális eset, a char változó felhasználási eseteinek 99.99999999%-ában nem erről van szó! Azaz még ha meg is csinálták a lehetőséget a C/C++ nyelv konstruálásakor hogy ezt lehessen negatívként is értelmezni, de legalább alapértelmezettként jelentené a 0-255 tartományban való megfeleltetést!

Sajnos nem így van. Oké, akkor vezettek volna be egy „byte” típust ami mindig 0-255 közé esik!

Ezt sem tették meg. Nos emiatt vezettem én be az USC jelölést, ami mindig kifejezetten 0-255 közé esik mint unsigned char típus, és egyáltalán soha sem lehet negatív! Ez teljesen egyértelmű. Általában véve, én a „szigorúan típusos” programnyelveket szeretem. Ezalatt nem azt értem, hogy ne legyen lehetőség a típusok közti konverzióra, nagyonis legyen erre lehetőség, bőségesen, mindenféle variációban, beleértve a legképtelenebb eseteket is — de ezek EGYIKE SE TÖRTÉNJEN MEG AUTOMATIKUSAN, hanem mindig CSAK és KIZÁRÓLAG akkor, ha én azt explicite, szándékosan, direkt és szántszándékkal, előre megfontolt nagy aljas gonosszággal jelzem a forráskódban, ha félreérthetetlenül előírom, hogy márpedig én most kifejezetten épp ezt akarom!

Még azt se tartom jó ötletnek, ha a fordítóprogram ilyenkor csak egy „warning” üzenetet dob nekem. NEM. Kifejezetten azt akarom, hogy álljon le egy bődületes-nagy „error message” kíséretében! Egyáltalán ne legyen hajlandó lefordítani semmiféleképpen sem, amíg azt én határozottan bele nem írom a forráskódba,

hogy igenis azt akarom! Azaz, nekem a mau nyelv megalkotása közben ne csinál-
gasson olyat, hogy szerinte a char az negatív is lehet néha, emiatt elfogad oda
negatív számokat is, aztán ezt viszont néha pozitívként értelmezi ha karakternek
tekintem stringműveleteknél meg még a tökömtudja (sőt az se...) mi mindenféle
bonyodalmak támadhatnak ebből. NEM. Az én programnyelvem forráskódjában a
karakterek márpediglen egyszersmindenkorra UNSIGNED char értékekből állnak,
mind kifejezetten előjel nélküli, s ilyen karakterekből áll a mau program számára
lefoglalt leendő adatterület is. Majd ha valamikor netán mégis szükségem lenne
ezek előjeles értelmezésére (fehér holló ritkaságú lesz ezen esetek száma, már ha
egyáltalán előfordul valamikor, ami nem is biztos...) majd akkor megoldom azt
valamiképpen.

Ha már ennyire belemerültünk e szomorú témába, itt említtem meg, hogy a fenti
megfontolásokból a leendő mau programnyelvet lefordító Makefile elején nálam e
sor szerepel:

```
CFLAGS = -funsigned-char -funsigned-bitfields -Wall -Wno-long-long -Wextra -pedantic -Wunused
```

Ebből az egyes értékek a következőt jelentik:

-funsigned-char : A „char” akkor is előjel nélküli azaz unsigned, ha nem írom
külön ki. Ennek ellenére, nem tartom feleslegesnek az USC bevezetését, mert
biztos ami biztos, „az ürdüng nem aluszik”, de ha mégis, akkor is nyitva van a fél
szeme, sőt amelyik szeme csukva van, annak is átlát a szemhéján...

-funsigned-bitfields : A bitmezők is előjel nélküliek. Ezek előjeles értelmezése a
szememben még nagyobb örülség, mint a char előjeles alapértelmezése.

-Wall -Wextra -pedantic -Wunused : Bekapcsolnak mindenféle figyelmeztető
jelzéseket amik csak léteznek (warningok). Ez persze nem azt jelenti hogy nálunk
fordítás közben lesznek warningok. NEM. *Nem szeressük őket.* De épp EMIATT
mondjuk azt a gépnek, hogy rikoltozzon csak minden apró fasztság miatt nyugod-
tan (már bocs a szóért...) aztán ha MÉGSE rikoltozik, akkor megnyugodhatunk
hogy ügyes gyerekek vagyunk. Ez a legbiztosabb módja a hibamentesség
elérésének. Azaz bár a lehetőséget megadjuk a warningok létének, ennek ellenére
nem szabad hogy legyen nekünk azokból akár egy is!

Sosem értettem, miként is tűrhet meg egy programozó a programjában akár
EGYETLEN warningot is fordítás közben! A lustaság alatt én úgy tűnik NEM azt
értem, amit ezek a nemtörődöm... Ööö... *Izék...* Maradjunk ennyiben hogy „izék”,
nem akarom hogy bepereljenek valamiféle sértegetés címén, hogy nem vagyok
elég „politikailag korrekt”, mindenesetre annyit azért kihangsúlyozok, hogy ez a
részemről feljűk nem ám dicsérő jelző... (hogy finoman fogalmazzak...)

-Wno-long-long : Ez viszont KIKAPCSOLJA a „long long” változó használatakor
keletkező warningot. Azt azért rakták bele, mert a 32 bites gépeken annak a
változónak nemigen volt létjogosultsága, nem lévén akkora memóriaterület ami
azzal címezhető. De már a 64 bites gépek korát éljük... És nekünk erre kife-
jezetten szükségünk is lesz a mau programnyelvben, mert az kell hogy kezelni
tudjon ekkora számokat is.

Ezek után megalkothatjuk a konstruktort és destruktort:

```
PGM::PGM(USIL phossza) { // KONSTRUKTOR
f.phossz=phossza;f.p = new USC[f.phossz];if(!f.p) {L("Nincs elég memória!");EXITFAILURE();}
f.p=0L;
.....
f.m=NULL;f.mhossz=0L; // adatmemória nullára állítása
.....
}
// -----
PGM::~PGM() { // DESTRUKTOR
if(f.p!=NULL) {f.phossz=0L;delete[] f.p;f.p=NULL;}
if(f.m!=NULL) {f.mhossz=0L;delete[] f.m;f.m=NULL;}
}
```

Mint láthatjuk, a konstruktor nem foglal még le memóriát az adatterületnek is, hiszen előre nem tudhatjuk, hogy az mennyi kell legyen. Elképzelhető olyan mau program is extrém esetben, hogy nem kell neki egyetlen bájt se ilyen célra. De ha kell is, csak maga a program tudhatja, mennyi kell neki, nyilván erre kell majd valami utasítást kreálnunk mau nyelvünkbe. A destruktorkor azonban fel kell szabadítsa a memóriaterületeket, persze csak ha van egyáltalán lefoglalva ilyesmi, ezért előbb ezt ellenőriznie kell. Nem illik ugyanis hogy megpróbálkozzék olyasminak a felszabadításával, amit korábban le se foglalt, az ilyesmi az Álmoskönyv szerint mindenféle, a Felhasználót frusztráló hibaüzenetet szül, amikor is Ő, a Felhasználó Őfelsége nemcsak a mi nevünket kezdi sűrűn emlegetni olyan kontextusban amit ha hallanánk, fülig pirulnánk, de még az anyukánk is eszébe juthat neki. Sőt, efféle otrombaságot ha megengedünk a programunk számára, az rosszabb esetben a rendszer teljes lefagyását is okozhatja. Nem illene hogy azt okozza, elvileg egy jól megírt operációs rendszer kezelni kéne hogy tudja egy program efféle idiótaságát, de hát jól tudjuk, hogy sajnos nem a Lehetséges Világok Legjobbikában élünk. (Vagy netán mégis?! Nagy Egek, lehetséges volna hogy *nincs is ennél jobb világ?!* Iszonyatos!)

Egyesek a fenti kódsorok láttán kételkedhetnek az elmém épségében, e sor miatt:

```
if(f.p!=NULL) {f.phossz=0L;delete[] f.p;f.p=NULL;}
```

Mert ugye, minek is ellenőrzöm én a **p** pointer értékét, hogy az nullpointer-e, hiszen a destruktorkor kizárólag olyan objektum esetén futhat le, melynek valamikor már végrehajtódott a konstruktora, az pedig garantálja, hogy e pointer értéke igenis nem maradhat NULL!

Nos, e gondolatmenet természetesen helyes is, sajnos azonban kizárólag a jelenlegi helyzetben. Egy objektumnak ugyanis lehet akárhány konstruktora is, köztük a híres-hírhedt „default” konstruktor, mellyel mi nem rendelkezünk még perpillanat, s amelynek nincs semmiféle bemenő paramétere, azaz ha valamikor ezen konstruktor létrehozására vetemednénk, ez nem lenne képes lefoglalni semekkora memóriaterületet sem, mert nem tudná, mekkorára volna szükség! Ezokból a PGM osztályunk konkrét példányosítása kapcsán a szerencsétlennek csak valamiféle „vázát” alkothatná meg, memóriaterület nélkül, memóriáról másképp kéne gondoskodni neki valahogy a későbbiek során. Destruktorkor azonban lenne ezen esetben is e PGM példánynak, még hozzá ugyanaz a destruktorkor... Ez igen veszélyes helyzet, ezokból szeretném is elkerülni efféle „default” konstruktor megalkotását, hátha nem is lesz rá szükség. De mert lehet hogy mégis, emiatt, meg amiatt hátha figyelmetlenségből olyan idiótaságot követelek el valamely későbbi rutinban ami lenullázná a **p** pointert, belevettem a destruktorkorba ezen

ellenőrzést. Nem lassítja le érdemleges mértékben a program futását, mert a destruktorkor nem lesz gyakran meghívva - lényegében csak egyszer, a mai programunk legvégén, amikor már úgymint megcsinált nekünk mindent, és épp befejezi a futását. Ennyit bőven megengedhetünk magunknak a fokozott biztonság érdekében.

Ha már itt tartunk, rittyentsük ide egy laza csuklómozdulattal azt a rutint is, ami beolvassa (természetesen a már lefoglalt memóriaterületre) a megfelelő mai nyelvű forrásfájlt:

```
void PGM::beolvas(char *filename) { // beolvassa a fájlt a memóriaterületre
f.programfile=filename;f.programfilehossz=strlen(f.programfile);
f.programfilehossz=get_file_size(f.programfile);
if(f.phossz<f.programfilehossz) {L("A lefoglalt memóriaterület (%lu byte) kisebb mint a beolvasandó programfile mérete "
"(Fájlnév: %s, = %lu byte)!",f.phossz,filename,f.programfilehossz);EXITFAILURE();}
FILE *fp=fopen(filename,"rb");
if(!fp) {L("A megadott \"%s\" állomány nem megnyitható (nem tudom beolvasni)",filename);
EXITFAILURE();}
register USIL i;for(i=0L;i<f.programfilehossz;i++) {f.p[i]=(USC)fgetc(fp);} // for i vége
fclose(fp);
}
```

A fentiekből csak a **get_file_size** függvény szorul magyarázatra. Íme:

```
USIL get_file_size(char *filename) // path to file
{
FILE *p_file = NULL; p_file = fopen(filename,"rb");
if(!p_file) {L("A megadott \"%s\" állomány nem megnyitható! (Esetleg nem létezik?)",filename);EXITFAILURE();}
fseek(p_file,0,SEEK_END); USIL size = ftell(p_file); fclose(p_file);
return size;
}
```

Kihangsúlyoznám, hogy bár elképzelhető amit a Tisztelt Olvasó gondol rólam, azaz nem kizárt hogy valóban kőbölcsőben lettem ringatva még annó, de nem emiatt nyitom meg a beolvasandó állományt az **"rb"**-vel jelzett módon, azaz kifejezetten bináris megnyitást előírva, annak ellenére, hogy a mai nyelvű forráskód valami szövegfájlban lesz elérhető feltehetőleg! A dolog úgy áll, hogy sosem értettem, mi az a perverzió, hogy van valami „szöveges” megnyitásra is lehetőség. Nem látom értelmét. Az csak arra jó, hogy bizonytalanságot vigyen bele az egész feldolgozási folyamatba. Mindenesetre, egyedül a binárisként megnyitás garantálja, hogy istenbizony minden bájtot pontosan úgy kapunk meg a fájlból, ahogyan az ott szerepel eredetileg, s beolvasás közben nem történnek mindenféle aljas és számunkra nem várt konverziók/átalakítások, amelyeket a „cúf, mockosz kisz hobbitorok” beleépítettek a beolvasórutinba. Márpedig efféléket nem engedhetünk meg magunknak. Először is azért nem, mert fogalmunk se lesz róla, leendő mai nyelvünk interpretere miféle architektúrán fog futni, másrészt ha tudnánk is hogy tutira kizárólag Linux környezetben, az se védene azellen, hogy olyan mai forráskódot kapjunk, amelyet valami más környezetben működtetett text editorral kreáltak. Márpedig sajnos léteznek a szoftvervilágban olyan aljasságok, hogy például nincs egységesítve a sorvégek jelölése (sem...):

- azt jelölheti a 0x0A karakter (Ennek az elfogadott beceneve az LF, és ez van nálunk „Linuxéknál”),
- a 0x0D karakter (ennek a beceneve a CR, és ez van a MacOS világában, meg ez volt régen a C-64 számítógépnél is),
- vagy mindkettő egyszerre: CR LF, ez van a Windows uralta „Sötét Oldalon”, ebből is látszik hogy igenis a Windows a legislegjobb, legislegsziperebb és legislegtutibb operációs rendszer vitathatatlanul, mert az annyira nagyon de nagyon jó

már, hogy még azt is megengedheti magának hogy pazaroljon memóriában, állományméretben és műveletvégző sebességben, hiszen minden egyes sor végére odabiggyeszt egy plusz bájtot teljesen feleslegesen.

Mindenesetre, én annak vagyok a híve, hogy csak ne bízzunk abban, mennyire lesz okos holmi MÁSOK által megírt szövegszerkesztő vagy beolvasórutin, az a legtisztább ha mindent mi magunk tartunk kézben. Azaz olvassuk csak be azt a fájlt úgy ahogy az ott van, aztán majd a maga idejében megküzdünk a sorvégekkel meg esetleges más mindenféle elképzelhető inkompatibilitással.

Na tehát ott tartunk, hogy TÉNYLEG megvan valamiféle kerete/csontváza a programnyelvünknek, most azonban illik e csontvázat „hússal feltölteni”. Be van olvasva a mau forráskód, ami valamiféle utasításokból áll, s ezeket kell végrehajtanunk. De miféle utasítások is legyenek ezek, s hogyan hajtassanak végre? Ezzel foglalkozunk a következő fejezetben.

3. fejezet: A mau nyelvű program parancsai/utasításai, és ezek végrehajtása

Mau programunk kb a következőképp fog indulni:

```
int main(int argc, char **argv) {
.....
if(argc==1) {L("Nem adtad meg a futtatandó állomány nevét!"); EXITFAILURE();}
// argv[1] a végrehajtandó file neve!
USIL hossza=get_file_size(argv[1]);

PGM program(hossza);
program.beolvas(argv[1]);
program.futtat();
} // main vége
```

Ez persze fentebb csak egy durva váz, amennyiben nem foglalkoztunk pld semmiféle esetleges parancssori kapcsolók beolvasásával, holott ha azok netán léteznek, az se biztos hogy tényleg az **argv[1]** tartalmazza majd a mau forráskód-állomány nevét. Ettől eltekintve azonban, a fenti megoldás igenis MŰKÖDIK, kipróbáltam! Ki hinné pedig, hogy egy programnyelv főprogramja, a „main” rész ennyire egyszerű is lehet... de hát már a Bevezetőben azt emlegettem ugyebár, hogy egy programnyelv megalkotása alapvetően KÖNNYŰ feladat illik hogy legyen!

Egyes „mockosz kisz hobbitkák” ha akadnak Olvasóim közt, esetleg elkezdenek trollkodni amiatt, hogy micsoda dolog is, hogy fentebb beolvastatom a program hosszát a **get_file_size** függvénnyel, a konstruktor számára, azután pedig meghívom a **beolvas**-függvényt, mely megint meghívja a **get_file_size** függvényt a hossz meghatározására! Eképp az végrehajtódik másodszor is, feleslegesen!

Ez teljesen igaz. Sajnos, a konstruktornak mindenképp meg kell adnunk méretet, ami megmondja, mekkora memóriaterületet kell lefoglalnia! A beolvasó rutin viszont mindenképp illik hogy ellenőrizze, nem akarunk-e nagyobb fájlt beolvasni, mint amekkorára elegendő a hely.

A fenti problémát kizárólag úgy tudnánk megoldani, ha csinálnánk egy másik konstruktort is, aminek bemenő paramétere nem a lefoglalandó memória mérete, hanem csak az állománynév. Ekkor meghívna ő maga a **get_file_size** függvényt (egyetlenegyszer), az azáltal beolvasott hosszak megfelelően lefoglalja a memóriát, majd beolvassa oda a fájlt.

Mindez könnyű lenne, ezt igazán nem gond megírni, egyszerűen berzenkedem a gondolat ellen, hogy szaporítsuk a konstruktorok számát. Kivétel nélkül minden egyes rutin ugyanis amit megalkotunk, növeli a kód méretét, komplexitását, újabb hibalehetőségeknek nyújt terepet azaz rizikófaktor, s legfőképpen fennáll az inkonzisztencia veszélye. Ez azt jelenti, hogy ha a programunk későbbi fejlesztésekor valamivel ki akarjuk bővíteni a konstruktort, akkor esetleg ezt csak az EGYIK konstruktor esetén végezzük el, s a másiktól megfélekedünk. Na és hát akkor aztán „meg leszünk löve”, kereshetjük majd a hibát sokáig mire megeljük, mert az úgy fog jelentkezni, hogy bizonyos esetekben a programunk remekül működik majd, más esetekben azonban vagy leáll valami hibajelzéssel, vagy ami még rosszabb, működik majd, csak éppen rosszul. Na és kifejezetten épp az ilyen hibák a legislegrohadtabbak, ezeket a legislegnehezebb ugyanis felderíteni.

Azaz nekem elvem, hogy minden rutinból csak egy legyen, egy és nem több, amennyiben ez megvalósítható. Ennek érdekében nem tartom nagy áldozatnak a forráskódfájl hosszának kétszeri ellenőrzését beolvasáskor, mert ez úgyis csak egyszer történik meg, az induláskor. Nyilván persze másképp vélekednék, ha ez egy ciklus közepén hajtódna végre, 30 milliószor. Szerencsére erről szó sincs.

Na tehát, bé vagyon olvasva az mau prograny, ami valamiféle utasításokból áll ugyebár. Ezeket kell végrehajtani. E végrehajtást a PGM osztályunk „futtat” nevű metódusa végzi. E metódus igazán egyszerű, mert semmiféleképp se nézhet ki másképp, mint eképpen:

```
void PGM::futtat(USIL honnan) { // futtatja a programot
f.P=honnan;if(f.P>=f.phossz) {L("Kísérlet a programnak kiosztott memóriaterület határának átlépésére");EXITFAILURE();}
folytat();
}
```

Mint azt az előző fejezetben láthattuk, e futtat metódusnak van alapértelmezett paramétere, mely megmondja, honnan, melyik bájtól induljon a futtatás. Ha nem adunk meg külön valami más értéket, ez a nulladik bájt. Beállítja erre a megfelelő utasításslámlát, majd azzal kezdi, hogy szépen leellenőrzi, ez nem nagyobb-e mint a progi maximális hossza (mert ugye nem akarjuk hogy az a modortalán, bunkó Felhasználó a szájára vegye szegény időseinkat...), majd ha minden oké, egyszerűen meghívja azt a rutint, mely elvileg arra van, hogy onnan folytassa a végrehajtást, ahol azt korábban netán abbahagytuk valamiért. A lényeg tehát abban a rutinban lesz elrejtve.

Na és hát itt bizony megintcsak ideje vagyon a belustulásnak, de nagyon ám...!!! Mert e rutin rém sokfélefépp megírható, s hogy épp miként is kell megírnunk, az attól függ, miként döntünk leendő szépséges mau nyelvünk milyenségével kapcsolatban. Azaz például miféleképpen legyenek az utasítások, számozva legyenek-e a sorok mint a Basicben, meg más mindenféleképpen. Azaz, ELÉRKEZTÜNK A PROGRAM- NYELV-TERVEZÉS TALÁN LEGIZGALMASABB RÉSZÉHEZ!

Az egyetlen dolog, amit tutira tudhatunk előre a jelen pillanatban e **folytat()** metódusról, az sajnos csak ennyi:

```
void PGM::folytat(void) { // Folytatja a program végrehajtását az aktuális helytől
while(1) { // ÖRÖKCIKLUS !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
if(f.P>=f.phossz) {if(logflags[3]) {L("A program befejeződött!");} exit(EXIT_SUCCESS);}
.....
} // while(1) vége
}
```

A lényeg nyilvánvalóan az lenne, ami a sok ponttal jelzett sor helyett kell álljon...

Mint fentebb látható, e metódus alapvetően egy örökciklus. Ebből akkor ugrunk ki, ha a P nevű utasításszámláló (mely kezdetben a legelső végrehajtandó utasításra mutat) elérkezik a program végéhez. Elvárható ugyanis, hogy egy program abbahagyja a tevékenységét, ha elérkezik az utolsó utasításhoz. E tényt illik logolnia is valamiképp, amennyiben megengedjük neki a logolást, e logolási szintnek a 3-as számot választottam. Tuti hogy erre épp a 3-as szám a legmegfelelőbb, ezt ékesen bizonyítja ugyanis, hogy ez a szám úgy jött nekem ki, hogy megvakartam a köldökömet, s e fantáziafejlesztő ténykedés után ez volt az első számjegy ami az elmémben felbukkant. Ez tehát teljesen biztos hogy jó, és minden más érték csakis rosszabb lehet.

A P értékét nyilván minden egyes utasítás végrehajtása után meg kell növelnünk legalább 1-el, de lehet hogy többel, valamint az se kizárt hogy maga az épp végrehajtott utasítás is változtatja majd P értékét, tipikusan ez lesz a helyzet például a mindenféle ugróutasítások és ciklusok esetén.

Az is nyilvánvaló, hogy kell legyen majd valami olyan utasításunk is, ami kifejezetten épp arra szolgál, hogy a programból máshol is kiléphessünk, ne csak a legvégén. Ez nyilván úgy történik majd, hogy minden egyes végrehajtott utasítás vissza kell adjon e **folytat()** rutinnak egy visszatérési értéket, ami egy kód. Ha ez a megfelelő értékű, ki kell lépnünk a ciklusból, hibajelzéssel vagy anélkül. Egyezünk meg abban, hogy e kód az „errorcode” nevű változóba kerül, amit deklarálni kell a PGM osztály meghatározásánál is eképp:

```
int errorcode; // A programutasítások visszatérési értékei. 0=normál; -1=végetért a program hibával;
// 1=végetért a program rendben; 2=BREAK
```

Ennek megfelelően a **folytat()** metódust eképp fejleszthetjük tovább:

```
void PGM::folytat(void) { // Folytatja a program végrehajtását az aktuális helytől
while(1) { // ÖRÖKCIKLUS !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
if(f.P>=f.phossz) {if(logflags[3]) {L("A program befejeződött!");} exit(EXIT_SUCCESS);}
.....
if(errorcode==0) continue;
if(errorcode==1) {EXITSUCCESS();}
if(errorcode==2) {return;} // BREAK utasítás
if(errorcode==1) {EXITFAILURE();}
L("Váratlan hiba: Hibakód: %u, token: %u, cím: %lu",errorcode,f.a,f.P);
EXITFAILURE();
} // while(1) vége
}
```

Hogy a fentiekben mit jelent az **f.a** nevű úgynevezett „token”, az könnyen kitalálható: Ez maga az utasítás, ami holmi galibát okozott, vagy legalábbis az utasítás-

nak valamiféle kódja, ami alapján az beazonosítható. Illik ugyanis, hogy váratlan hiba esetén minden létező infót a Felhasználó (vagy a programfejlesztő) tudomására hozzunk. Na de mi is legyen ezen utasítás kódja, illetve miféleképpen legyenek az utasítások, most már semmiféleképpen se halogathatjuk tovább ennek a lényegi, minden túlzás nélkül fundamentálisnak nevezhető kérdésnek a megválaszolását!

Kezdjük a tünődést mindenekelőtt azzal, számozva legyenek-e a sorok! Nos, ezen ötletet szerintem vessük el gyorsan. Efféle megoldás rém sok macerát szül a napi gyakorlatban. Mert először is, sok plusz munka minden utasítássor elé sorszámot írni, ha pedig ezt automatizálni akarjuk, a program csak valami erre képes speciális szövegszerkesztőben írható meg. Aztán meg, mindig tünődni kéne azon, a sorszámoknak mennyi legyen a lépésköze, s ha két meglevő sor közé annyi sort szúrunk be hogy kifogytunk a számtartományból, akkor át kell sorszámoztatni az egész programot, ami bonyolult feladat. A soreleji sorszámok ráadásul a forráskód méretét is megnövelik. Egyáltalán, efféle megoldás a programnyelvünket a BASIC-hez teszi hasonlatossá, ami nem volna előnyös, mert a BASIC-nek nincs túl jó híre a komolyabb programozók közt.

Szóval, sorszámok nem lesznek. Ezen esetben biztosra vehetjük, hogy a **p[P]** mindig a forráskód valamely fontos karakterét adja vissza a számunkra. Miféle főbb csoportokba is eshet e karakter?

1. Jelentheti valamely kulcsszó egy karakterét, például a „**print**” utasítás „p” betűje lehet.
2. Lehet valamely változó nevének egy karaktere, mondjuk az a „p” betű, ami a **p=0** utasításban szerepel.
3. Lehet valamely függvény nevének a része, például annak, hogy **pirosra_szinezd(...**
4. Lehet valamely címke nevének része, mondjuk abban az utasításban, hogy **goto pointer_kiszamol**
5. Vagy lehet a címke nevének része akkor, amikor épp megjelöl a címke valamely helyet a forráskódban:
pointer_kiszamol:

Hát ez mindenesetre nem kedvez annak az elvnek, hogy amint elér egy utasításhoz az interpreterünk, azonnal végre kell hajtsa! Ott van nála a „p” karakter, és fogalma sincs róla, mit kezdjen vele. Sőt, legtöbbször még ha a teljes szót be is olvasta valamely whitespace karakterig, akkor se tudja hogy az a string micsoda is. Kéne neki a teljes utasítássor, hogy ezt kikalkulálja, s ez egy iszonyatosan bonyolult és ocsmány szövegfeldolgozási módszer megírását tenné szükségessé a számunkra, ami teli van hibalehetőségekkel, lassú is mind megírni, mind aztán végrehajtani, és véres veritékkel lehetne csak bővíteni később, ha eszünkbe jut valami.

A dolgon csak kevésbé segítené, ha a Pascal vagy a C nyelv azon módszerét követnénk, hogy a felhasznált változókat és függvényeket deklaráljuk valahol a program elején. A „p” karakter beolvasásakor ezesetben se tudhatja még, hogy az a „print” kulcsszó része-e, vagy a „papa” változóé. Sőt miután beolvasta a teljes szót, és tudja már hogy ez nem „print” hanem „papa”, ezután még illik leellenőriznie, hogy a „papa” egy érvényes változónév-e amit korábban deklaráltunk, s ha

nem, hibajelzést kell produkálnia. S ha e változó egy cikluson belül van ami 10 ezerszer hajtódik végre, ezt az ellenőrzést elvégzi tízezerszer. Nem tűnik gyorsnak és hatékonynak...

Mindez érvényes a függvénynevekre is.

Sőt, maradjunk egyelőre csak a programutasításoknál! Attól hogy beolvasta azt a szót hogy „print”, még nem fogja tudni, mi az ördögöt is kell tennie. Végig kell nyálaznia a kulcsszavak listáját tartalmazó tömböt, hogy szerepel-e benne ez a kulcsszó. Ha nem, dob valami hibajelzést és megáll. Ha szerepel, attól függően hányadik helyen találta meg a tömbben, valahonnét máshonnan előszedi a megfelelő sorszámú függvény címét, s meghívja azt. Az fog printelni nekünk, az lesz a tulajdonképpeni „print” parancs. És itt is igaz, hogy ha ez az utasítás egy cikluson belül van, ami lefut rengetegszer, akkor mindegyik alkalommal elvégzi ezt a tömbben való keresgélést. Az efféle dolgok miatt oly lassúak az interpreterek.

Ezt mi nyilvánvalóan nem engedhetjük meg magunknak! De akkor mit is tehetnénk?!

Induljunk ki abból, hogy bár a legtöbben azt hiszik, hogy a számítógép egy „nagyon okos masina”, mi akik értünk hozzá, tudjuk hogy ez egyáltalán nem igaz. A számítógép egy nagyon hülye valami. Nem tud gondolkodni, nincs benne előrelátás, mindent a „szájába” kell rágni. Emiatt akkor a leghatékonyabb a munkája, ha tényleg nem is kell nemhogy gondolkodnia, de még csak úgy tennie sem mintha gondolkodna! Vagyis, ha nem szükséges minden feladat végrehajtása előtt végigvizsgálnia egy csomó mindenféle adatot, s kikalkulálnia hogy mitévő legyen, hanem az hogy épp mit kell tennie, teljesen világosan következik egyrészt abból, épp hol tart a program végrehajtásában, másrészt abból, hogy épp milyen karaktert olvasott be!

Mindebből a „következők következnek”:

- feltételezhető, hogy a program az ő indulásakor legelőször valamely programutasításra bukkan. Ezen utasítás vagy csupán 1 karakterből áll, vagy több karakterből. A mi fentebb bemutatott örökciklusunknak azonban nem szükséges tudnia, az az utasítás hány karakteres. Ő csak azt tudja, ami triviális: EGYETLEN karakterből biztos hogy áll, mert nullakarakteres utasítás nem létezik! Ezekből egy táblázatból (amit e karakter ASCII kódjával indexel) kiválasztja egy függvény címét, s meghívja azt. Ez a függvény kell tudja, hogy léteznek-e olyan utasítások amelyek e karakterrel kezdődnek de vannak más karaktereik is a névben, e karakter után, vagy ez tényleg egy olyan utasítás, aminek a neve csak és kizárólag ezen egyetlen karakterből áll!

E tömb amit a mi örökciklusunk indexel, logikus hogy épp 256 elemből álljon, mert így megspóroljuk annak ellenőrzését, hogy a tömbindex túllépi-e a tömb méretét, ez pedig fontos, mert e ciklus lefut kivétel nélkül minden utasításunknál, azaz ezen ellenőrzés megspórolása jelentős sebességnövekedést eredményez a számunkra! Továbbá, így lehetőség nyílik rá, hogy a nem kiiratható ASCII kódokhoz is rendeljünk utasításokat, ami tök jó poén, akármikor a hasznunkra

válhat, pláne ha tényleg valamely létező processzor gépi kódjára akarunk emulátort írni e módszerrel. Harmadrészt ennek az a haszna is megvan, hogy nem kell foglalkozni olyasmivel, hogy érvénytelen utasításkód vagy éppen whitespace karakterek átugrása és keresése - egyszerűen csinálunk egyetlen függvényt, aminek legyen az a neve hogy „semmi”, ez nevéhez méltón semmit se fog csinálni, ő lesz az „üres utasítás”, ennek címét kezdetben beírjuk a táblázat mind a 256 helyére, aztán ha olyan kódra bukkan a ciklusunk/interpreterünk amihez még nem gyártottunk semmi értelmes függvényt, akkor ott e „semmi” függvényt találja meg, azt hajtja végre, azaz szorgalmasan és elszántan csinálja majd ekkor a semmit.

Oké, tehát akkor nekünk most azon kell tünődnünk, milyen legyen a típusa annak a függvénynek, amiből e 256 elemű tömb áll! Nos, ilyen, mint amit az alanti sor mutat:

```
typedef int fuggveny(struct F& f);
```

Ugyanis a végrehajtott függvény (parancs, utasítás...) nyilván kell hogy vissza tudjon adni egy kódot a maga befejeződésének milyensége felől, annál is inkább, mert ez kerül a fent már említett errorcode változóba! Amit pedig paraméterként vár, az az F struktúra kell legyen, amit direkt úgy emlegettünk e könyv elejétán, hogy kifejezetten ez az a struktúra, amibe „minden fontos adat” kerül, mármint minden olyasmi, ami egy efféle függvénynek fontos lehet. Tekintve azonban hogy a struktúra baromi nagy lehet, ezért csakis referenciaként adhatjuk át.

Ezek után deklaráljunk egy tömböt eképp:

```
fuggveny *fuggvenyek[256] = {
/*      0 */ semmi,
/*      1 */ semmi,
/*      2 */ semmi,
.....
/*     30 */ semmi,
/*     31 */ semmi,
/*     32 SPACE      */ semmi,
/*     33 !          */ semmi,
/*     34 idézőjel   */ semmi,
/*     35 #          */ semmi,
/*     36 $          */ semmi,
/*     37 %          */ semmi,
/*     38 &          */ semmi,
.....
/*    251 */ semmi,
/*    252 */ semmi,
/*    253 */ semmi,
/*    254 */ semmi,
/*    255 */ semmi
};
```

A felépítése gondolom világos. Persze, ahhoz ez működjön, deklarálni és megalkotni kell a „semmi” nevű függvényt. Ez eképp „csinálódik”:

```
fuggveny semmi;
.....
int semmi(F& f) {if(logflags[7]) L("Üres utasítás: kód: %u, cím: %lu",f.p[f.P-1],f.P-1);
return 0;}
```

Értelemszerűen, a „semmi” hatására nem csinál semmit. Illetve... Illetve logolja a tényt, hogy itt neki semmit se kellett csinálnia, amennyiben ezt engedélyeztük a

hetes logolási szinttel (mely hetes érték csak jó három másodperces elszánt újabb köldökvakarás után ötlött eszembe).

Ennek hogy logolhatja a semmit, 2 nagy haszna van:

1. Mindenekelőtt, ha tényleg semmit se csinálna, értve ezalatt azt hogy nem használná fel a függvény a paraméterül átadott **f** struktúrát, akkor a fordító rikoltozna egy warninggal, hogy itt van nekünk egy „unused variable”. Bár ezt az idegbaját épp mi kapcsoltuk be direkt a Makefileben a **-Wunused** kapcsolóval, mert ez többnyire nagyonis hasznos, de azért mi „nem szeressük” a warningokat, így hogy olyan nekünk ne is legyen, kell valamit kezdeni a függvény input paraméterével.

2. Ténylegesen is hasznunkra lehet a semmi logolásának lehetősége, ha nem is a kész program napi használata során, de amikor fejlesztjük a programnyelvünket. Jó ha tudjuk, épp melyik kódot tekinti üres utasításnak, mert ezekre definiálhatunk valamely hasznos funkciót.

Oké, ügyes gyerekek vagyunk, de nekünk nem elég mindössze 256 lehetséges utasítás, sőt, ennek kevesebb mint fele, ha csak a kiiratható karaktereket vesszük alapul. Tudniillik a 127-nél nagyobb kódokra jobb ha alaphól nem számítunk, mert hogy az miféle karaktert jelez, az egyrészt attól függ, a szövegfájl UTF-8 kódolásban lett-e megszerkesztve, vagy UTF-16 kódolással, vagy netán ezek egyikében sem hanem valami „nemzeti” kódtáblát használva, ami lehet ISO-8859-x, ahol az „x” helyén kis túlzással mondva bármiféle szám megtalálható. Magyarok esetében az x többnyire 2, de erre se lehet azért mérget venni. És az ISO-n felül is akad még pár elfuserált kódolás. Szóval, mi jobb ha „első körben” maradunk az ASCII által meghatározott karaktereknél, s ha csinálunk is netán valamikor utasításokat a 127-nél nagyobb kódokra, akkor azt eleve annak tudatában tesszük, hogy valami rém sepcialis célra készülnek nálunk, valami olyasmire, hogy nem is szükséges őket megjeleníteni kiiratható karakterrel. Így e pillanatban mikor e sorokat írom, nem ugrik be, miféle helyzetben lenne ez jó vagy akár csak elfogadható, de nem kizárható hogy előfordulhat ilyen eset, és csak én vagyok fantáziahiányos.

A kiiratható karakterek a 33-126 kódúak, azaz ezekből 94 van. Ennyi pláne nem elég nekünk utasításkódnak. Ennek négyzete azonban (94*94) már 8836 darab, ami esetleg elegendő mennyiség lesz. Sőt, lehetséges hogy ezek nem is mindegyikét óhajtjuk felhasználni többkarakteres utasításként.

Alkossuk meg tehát a lehetőséget amiről fentebb már elmélkedtem, a több karakterből álló utasításnevek használatára!

Egy efféle utasításcsoportot persze szintén egy „függvény” típusú függvény kezel nekünk, s egy ilyen függvény nevét érdemes úgy megválasztanunk, hogy kitésszék belőle, utasításcsoportot kezel, s jó ha benne van a nevében a kezdő karakter ASCII kódja is. Ennek megfelelően mondjuk a / azaz „per” jellel kezdődő utasításokat (mint amilyen például a // és a /*) a „csoport_per_47” nevű függvény dolgozza fel nálam, s ezt így kell deklarálni:

```
fuggveny csoport_per_47;          // A / jellel kezdődő utasítások
```

E függvény megvalósítása viszont sajnos már rémségesen bonyolult lesz programozástechnikailag! Nem is biztos hogy belefér egy kisebb képernyőbe, ugyanis ilyen rengetegsok kódsorból áll:

```
int csoport_per_47(F& f) {return masodikkarakter(f);}
```

S elárulom, elenyésző számú kivételtől eltekintve (ezekre természetesen részletesen is kitérek majd a későbbiekben) MINDEN efféle függvény mely több karakteres nevet kezel, azaz aminek függvényneve úgy kezdődik hogy „csoport_”, az mind pontosan ugyanígy néz ki a megvalósítást illetően, azaz csak a nevük különbözik, de ugyanúgy ezt a titokzatos „masodikkarakter” nevű függvényt hívják meg!

Hogyan is lehetséges ez?!

Idemásolom e függvény kódját, aztán elemezzük csak ki szépen!

```
int masodikkarakter(F& f) {// Ezt a függvényt hívja meg minden olyan függvény, mely nem egykarakteres tokenű.
fuggveny **fuggtomb;

if(f.P>=f.phossz) {if(logflags[3]) {L("A program befejeződött!");} EXITSUCCESS();}
f.a2=f.p[f.P++];
fuggtomb=FUGGVENYTOMB[f.a];
if(fuggtomb==fuggvenyek) return 0; // Ezesetben valójában üres utasításról van szó, helykitöltőnek került csak bele,
// és semmit se kell csinálni

if(f.a2<32) {if(wspc(f.a2)) {f.a2=SPACE;f.P--;} else return 0; // Nem kiiratható karakterek esetén semmit se csinál,
kivéve ha
// az a karakter whitespace, ezesetben ugyanazt csinálja, mintha space lenne.
// Vagyis a tömb legelső függvénye az legyen, amit akkor csinál, ha a parancstokent semmi más karakter nem követi,
// tehát ha egyedül áll.
}

if(f.a2>126) return 0; // Nem ASCII karakterek esetén semmit se csinál
return fuggtomb[f.a2-32](f); // végrehajtja az utasítást és visszatér annak visszatérési értékével.
}
```

E függvény megértésében az a nehéz, hogy hivatkozik pár tömbre meg más függvényre, melyeket még nem említettünk. Mindenekelőtt: Az **f.a** és **f.a2** változók az F struktúrában eképp deklaráltak:

```
USC a;          // az aktuális végrehajtandó utasítás kódjának első karaktere
USC a2;         // az aktuális végrehajtandó utasítás kódjának második karaktere
```

Ez tiszta sor.

A **wspc** függvény azt ellenőrzi, a karakter úgynevezett „whitespace” karakter-e:

```
int wspc(unsigned char a) { // visszatér 1-el ha az "a" karakter whitespace, különben 0-val tér vissza

switch(a) {
case SPACE:
case TAB:
case 0:
case 10:
case 11:
case 12:
case 13: return 1; break;
}
return 0;
}
```

Ezek után már csak azt kell megtudnunk, miféle tömbökre is hivatkozik ez a **masodikkarakter** függvény! Nos, a

```
fuggveny **fuggtomb;
```

sor azt mondja, hogy a **fuggtomb** változó egy olyan pointer, ami egy „fuggveny”-re mutató pointerre mutat. (Hm... itt kezd a kedves Olvasó emlegetni szegény idősanyámat, mert eddig tök könnyű volt a könyvem, de most mintha kezdene bonyolódni... Sajnálom édes öregem, én leírtam a Bevezetőben, hogy a mutatókkal tisztában kell lenni, ahhoz hogy ezt meg tudd emésztetni...!)

Ezen változónak így ad értéket:

```
fuggtomb=FUGGVENYTOMB[f.a];
```

Ez persze feltételezi, hogy kell legyen nekije valahol egy **FUGGVENYTOMB** nevű tömb megalkotva. Van is, s az efféleképp van adatokkal feltöltve:

```
fuggveny ** FUGGVENYTOMB[256] = { // Ebben a tömbben van azon tömbök címe, melyek azokat a függvényeket (utasításokat)
tárolják
// (azok címeit) melyeket a parancstoken második karakterétől függően kell végrehajtani.
// Tehát ha mondjuk az 'A' parancs nem egybetűs, de van olyan is hogy Aa, AA, A=, AB, stb, akkor e tömbből kell venni az
// FUGGVENYTOMB['A'] -adik pointert, ami egy tömbre mutat, s onnan kell végrehajtani az x-edik függvényt, aholis az 'x'
az
// 'A' karaktert követő karakter.

/*      0 */ fuggvenyek,
/*      1 */ fuggvenyek,
/*      2 */ fuggvenyek,
/*      3 */ fuggvenyek,
/*      4 */ fuggvenyek,
/*      5 */ fuggvenyek,
.....
/*    47 /          */ fuggveny_per,
.....
/*    252 */ fuggvenyek,
/*    253 */ fuggvenyek,
/*    254 */ fuggvenyek,
/*    255 */ fuggvenyek
};
```

Azaz, a **fuggtomb** változó, miután megkapta a maga értékét e **FUGGVENYTOMB** nevű tömbből, azáltal, hogy abból kiolvasta az első karaktert indexként használva, ezután egy tömb címét tartalmazza, amiből majd a második karaktert indexként használva olvashatja ki a tulajdonképpeni végrehajtandó függvény címét! Mint látjuk, a **FUGGVENYTOMB** kezdetben a „**fuggvenyek**” függvénytömb címével van feltöltve, de ez csak „fake”, azaz helykitöltő szerepet játszik, kicsit hasonlóan a „**semmi**” függvényünkhöz. Kell valami ami itt lefoglalja a helyet a nem használt variációknak, s ezt raktam be, mert a típusa megfelelő volt. A **masodikkarakter** függvény mint látjuk ellenőrzi is, hogy a kapott pointer ezzel egyenlő-e, s ha igen, abból tudja, hogy semmit se kell csinálnia ezesetben.

A „/” azaz „per” jel esetében azonban kell csinálnia, tudniillik mert azon a helyen a **fuggveny_per** név található, ami tulajdonképpen kicsit becsapós név, mert ő nem függvény, hanem függvényeket tartalmazó tömb. Eképp van szegénykém meghatározva:

```
fuggveny *fuggveny_per[95] = {
/* 0   32 SPACE    */ sorkiir, // Az egyetlen / jel kiír egy üres sort az stdout-ra
/* 1   33 !        */ semmi,
/* 2   34 idézőjel  */ semmi,
/* 3   35 #        */ semmi,
/* 4   36 $        */ semmi,
/* 5   37 %        */ semmi,
/* 6   38 &        */ semmi,
.....
/* 10  42 *        */ percsillag,
```

```

.....
/* 15 47 /          */ perper,
/* 16 48 0          */ semmi,
.....
/* 26 58 :          */ semmi,
/* 27 59 ;          */ sorkiir,
/* 28 60 <          */ semmi,
.....
/* 94 126 ~         */ semmi
};

```

Mint látható, takarékosági okokból e függvénytömb csak a kiiratható karakterek számára tartalmaz bejegyzéseket.

Egyelőre itt csak három utasításnak van még tényleges bejegyzés készítve, a `//` nevűnek, a `/*` nevűnek, amik természetesen ugyanúgy megjegyzést jelentenek nálam mint a C és C++ nyelvekben, valamint a `/*` utasításnak, ami egy üres sort ír ki az stdoutra. Ez sok esetben hasznos lehet. Ugyanezt teszi akkor is, ha csak egyetlen „per” jelet talál, amit valamely whitespace karakter követ (azaz többnyire szóköz). A `/*` variáció azért csinálja maga is ugyanezt, mert a legtöbb programozó (én is) hajlamos automatikusan pontosvesszőt rakni az utasítások után, emiatt rakna ilyet a `/` után is, na és hát nem szép dolog ha ilyenkor a programunk hirtelen nemcsak hibajelzést írna, de még azt sem, hanem egyszerűen üres utasításnak értelmezné a `/*` kombinációt, és nem írna ki üres sort, amikor pedig mi azt akartuk!

A **sorkiir** függvény rém egyszerű:

```

int sorkiir(F& f) {if(logflags[4]) L("Üres sor kiírása: kód: %u, cím: %lu",f.p[f.P-1],f.P-1);
printf("\n");return 0;}

```

A `//` és `/*` megvalósítása:

```

int perper(F& f) {if(logflags[7]) L("Komment sor: %c%c, cím: %lu", f.a, f.a2, f.P-2);
ujrsorig(f);return 0;
}
// -----
int percsillag(F& f) { // megjegyzés a következő /* párosig
USI percsillagdarab;
USC a; USC b;
percsillagdarab=1;a=0;b=f.p[f.P]; // A b az aktuális karakter (A /* utáni első)
while(1) {
a=b;
if(f.P >= (f.phossz - 1) ) return 1;
b=k(f); // A következő karakter bekérése
if((a=='/'))&&(b=='*')) {percsillagdarab++;continue;}
if((a=='*'))&&(b=='/')) {percsillagdarab--;if(percsillagdarab==0) {f.P++;return 0;} else continue;}
} // while(1) vége
}

```

Igen, a „percsillag” kicsit bonyolult, amiatt, mert értelmes dolognak látszott úgy megcsinálni, hogy lehessen akárhány `/* ... */` csoportot egymásbaskatulyázni.

E függvény hivatkozik egy bizonyos „**k**” nevűre, ami visszaadja a következő karaktert. Ennek felépítése:

```

USC k(F& f) { // visszaadja a következő bármilyen karaktert a programkódból. A P mutató e visszaadott karakterre mutat!
f.P++;
if(f.P>=f.phossz) {L("Indextúlcserülés a programmemóriában!");EXITFAILURE();}
return f.p[f.P];
}

```

E fenti függvény létét kizárólag az igazolja, hogy minden alkalommal amikor léptetjük a P pointert, illik leellenőrizni azt is, hogy nem léptük-e túl a számunkra

kiosztott memóriaméretet. Ezt macerás és pazarló lenne minden alkalommal külön leírni, jobb ezt egyetlen helyen elintézni, legalábbis amikor mód van erre.

A // rutinunk hivatkozik még egy „ujssorig” függvényre, az így néz ki:

```
void ujsorig(F& f) { // Ellépteti a P pointert a következő újsor karakter utánig
while(f.p[f.P]!=SORVEG) {f.P++;if(f.P==f.phossz) goto ujsorig_kampec;
} // while vége
f.P++;if(f.P<f.phossz) return;
ujssorig_kampec:
L("A program befejeződött az utolsó utasításnál!");EXITSUCCESS();
}
```

Az eddigiek alapján már gondolom mindenki el tudja képzelni, miként is valósítatják meg akármilyenféle más kétkarakteres utasításnév-kombináció! Elvileg egy újabb szint bevezetésével akár 3 karakteres neveket is kezelni tudnánk, s ez nem kerülne semmi különösebb további programozási nehézségbe. De szerintem ennyi egyelőre elég.

Valaki Olvasóim közt talán úgy gondolhatja, rém szegényes lesz a programnyelvünk, ha benne egy utasítás csak 1 vagy 2 karakterből állhat! Nos, ez olyasmi, ami „ízlés kérdése”. Legelsőbbben is hadd emeljem ki, hogy akár szegényes ez akár nem, az biztos hogy RETTENTŐEN GYORS végrehajtást eredményez, tudniillik mert nem kell minden utasítás nevét kikeresgélne bonyolult és LASSÚ string-műveletekkel egy táblázatból! Az egykarakteres utasítások esetén azonnal tudja melyik rutint kell végrehajtania, a 2 karaktereseknél pedig csak egyetlen indexelést kell végeznie egy táblázatból, s megintcsak tud mindent. Ez icipicikét lassabb csak, mintha közvetlenül gépi kódban lenne megírva a programunk, holott nem compilert használunk, hanem interpretert! Más interpreterekhez képest a programunk SZÁGULDANI FOG!

Aztán, már bocs, de én egy programnyelvtől elsősorban nem azt várom el hogy benne az utasításnevek „szépek” legyenek (ez különben is egyénfüggő, ki mit tart szépnek...), azt se várom el hogy a programozó munkáját megkönnyítse mondjuk 2% mértékben, azzal, hogy az utasításokra jobban tud emlékezni ha azok sok karakteresek. A programozónak az a dolga hogy megküzdjön az efféle nehézségekkel, s megtanulja az utasítások nevét, akárhány karakterből álljanak is azok. Ezért kapja elvégre a FIZETÉSÉT! A programnyelv akkor jó, ha az elkészült program HATÉKONY, azaz lehetőleg villámgyors, még azon az áron is, ha az elkészítése netán 20%-kal több időbe telik. Tudniillik elkészíteni csak egyszer kell, futtatni viszont rengetegszer. Cseppet se vagyok híve annak a manapság elburjánzó szemléletnek, hogy a programozók leszárják az optimalizálást, s kimondatlanul is ahhoz az elvhez tartják magukat, hogy „ha használni akarod a programomat, paraszt, akkor végy nagyobb gépet”!

S ezt olyasmivel indokolják, hogy a hardware s általában a gépi erőforrások manapság már olcsóak, az „emberi munka” viszont drága.

DRÁGA?! Mióta?! Tele van a világ állástalan programozókkal, főleg Indiában és Dél-Amerikában meg az arab országokban, de lassan Afrika is felzárkózik ezek soraiba! Ezek a munkanélküli programozók írják a sok vírust és kémprogramot, mert másképp nem tudnak bevételhez jutni a tudásukból, nem lévén normális állásuk... Nem, cseppet se drága a programozói munka!

Erről tehát ennyit.

Olvasóm esetleg arra is gondolhat, e megoldás memóriapazarló a sok táblázat miatt. Nos végezzünk egy számítást! A 94 kiiratható karakter minden létező párosításban kiad 8836 variációt. Ez ugyanennyi függvénybejegyzés. 64 bites gépen egy pointer 8 bájtot foglal el, azaz ehhez kell akkor 70688 bájt. Ehhez még hozzájön a FUGGVENYTOMB számára további 256*8 bájt, ami 2048 bájt. Ugyanennyi kell a „függvények” tömb számára is. Ez összesen 70688+2048+2048 azaz 74784 bájt, ami alig több, mint 73 kilobájt! És ez már a maximális szükséglet, akkor, ha a létező összes név-variációra írtunk valami okos függvényt - ami alig hiszem, hogy be fog következni!

Azaz, ez nem sokkal haladja meg a régi C-64 számítógép memóriakapacitását, ami kb 30 éves technológia... hát ENNYI hardware-szükségletet azért talán nyugodtan kiköthetünk a programjaink számára! Arról nem is beszélve, hogy akármi más módozatot találnánk is ki a programutasítások meghatározására, akkor se tudnánk elkerülni, hogy minden megvalósított rutinnak le ne tárolnánk valahol a címét, azaz más megvalósítások se lennének nulla memóriaigényűek, az egészen biztos!

Emellett leírom azt is, hogy a 2 karakteres utasításnevek begépelése is sokkal könnyebb, kevesebb ideig tart, a forráskód mérete is kisebb lesz, és könnyebben elemezhető is valamely esetleg már létező vagy később elkészítendő debuggerrel, automatikus kódoptimalizáló szoftverrel vagy ilyesmivel. Egyáltalán, maga a kódunk is így kevesebb helyet foglal el a memóriában! Aki ugyanis amiatt aggódik hogy e módszerünk helypazarló, mert micsoda dolog, hogy felhasználunk 70 kilobájtot, az végezze csak el a következő számítást:

Tegyük fel, ír egy programot, ami nem is nagyon hosszú, de van benne mondjuk 100 „print” utasítás, 30 „goto”, meg még 300 más egyéb utasítás, amik átlagosan 5 karakter hosszúak. Ez összesen 500+120+1500 karakter, ami 2120 karakter. Ezzel szemben ha nálam minden utasítás maximum 2 karakteres, akkor ez a 430 utasítás legfeljebb 860 karakterbe kerül, azaz megspóroltam 2120-860=1260 karaktert, azaz több mint 1 kilobájtot máris! És 430 utasítás, az igazán minden csak nem hosszú program... A legtöbb program TÖBB EZER utasításból áll! Szóval, szerintem egyértelmű, hogy az e könyvben bemutatott módszer nagyonis előnyös. Ráadásul én azt hiszem nem is nehezebb megtanulni a két karakteres neveket, mint az olyasféle szófosó szintaxist, ami mondjuk a COBOL nyelvet jellemzi, ahol ilyesfélék az utasítások:

```
IDENTIFICATION DIVISION  
CONFIGURATION SECTION
```

E rész befejezéseként tehát annyit mondhatunk, hogy eljutottunk addig, amikor már tudunk utasításokat írni, mert az azok végrehajtásához szükséges váz/keret készen van! Jó lesz észben tartanunk azt is, hogy függetlenül attól, hogy 1 vagy 2 karakterből áll-e az utasításunk, amikor bekerülünk a tulajdonképpeni munkát elvégző függvényünkbe, ott kezdetben a P mutató mindig éppen pontosan az utasítás karakterét vagy karaktereit követő legelső karakterre mutat, teljesen függetlenül attól, hogy az „whitespace”-e vagy valami más, „értékesebb” karakter.

Már csak egyetlen ellenvetéssel kell megküzdennem. Azzal, hogy mit görcsölök én azért, hogy legyen minden két karakterből álló utasításnak egy külön függvénye, amikor ez pazarlás - a **csoport_per_47** függvény meg a többi hasonló helyett amik úgyse csinálnak mást, csak a **masodikkarakter** függvényt hívják, rögtön írhattam volna a megfelelő tömbbe a **masodikkarakter** függvényt is, még picit gyorsabb is lenne a programunk, mert kimaradna egy paraméterátadási és egy függvényhívási procedúra!

Ez teljesen igaz, de mint rögvést említettem is fentebb e téma kapcsán, azért lesznek kivételek, ha nem is sokan, amikor egy efféle függvény picit mást is csinál majd, nemcsak annyit, hogy meghívja a **masodikkarakter** függvényt! Emiatt, és csak emiatt van ez így megvalósítva. Ettől persze még azon esetekben, ahol nincs szó ilyen „kivételről”, hívhatnám rögvést a „**masodikkarakter**” függvényt, de ezt meg azért nem tartom jó ötletnek, mert szeretem ha a dolgok „egy kaptafára mennek”, ez csökkenti a tévesztés lehetőségét, valamint ha minden 2 karakteres függvénynek külön neve van, akkor elég velük foglalkozni, amikor esetleg valami bővítést akarunk végrehajtani, ami által ők is a „kivételek” közé tartoznak majd, s nem kell akkor a táblázatunkat is piszkálni. E kivételek ismertetése azonban kicsit később történik meg, előbb lépünk a következő fejezetre, ahol megismerkedünk azzal, elborult elmém miféle iszonyatosságot ötlött ki „változókezelés” címén... Illetve, az előtt is lesz még egy rövid fejezet. Mert előbb még pontosvesszőgetünk egy kicsit.

4. fejezet: Pár szó a pontosvesszőről

Ez rövid fejezet lesz.

Ezúttal, mielőtt valami „fogósabb” kérdésbe kezdenénk, előbb térjünk ki a „pontosvessző” kérdéskörére! Azaz, arra a micsodára, ami az egy sorba írt utasításokat elválasztja egymástól. (A Basic nyelv esetén ez a kettőspont). Nos a programnyelvek eltérnek egymástól abból a szempontból, mennyire és hol követelik meg valami efféle szeparátor kiírását a forráskódban. Egyáltalán, egy efféle valamit mint a pontosvessző, kétféleképp is lehet szemlélni. Egyrészt tekinthetjük UTASÍTÁSLEZÁRÓ jelnek. Ebben az esetben minden utasítás után le KELL írunk e karaktert. Akkor is, ha a sor utolsó utasításáról van szó. Tudniillik ha nem rakjuk ki ezesetben, akkor a compiler vagy interpreter úgy tekintené, hogy az utasítás folytatódik a következő sorban.

Másrészt tekinthetjük e jelet egyszerűen SZEPARÁTORNAK - ezesetben nem kell kitenni minden utasítás után, csak azon esetekben, amikor két utasítást el kell választani egymástól, mert másképp nem lenne egyértelmű, mi tartozik az egyikhez, és mi a másikhoz. Ebben a variációban ráadásul külön kérdés, hogy mi van akkor, ha e jelet nem lenne muszáj kitenünk, de mégis kitesszük. Ezt hibának kell tekinteni, vagy egy warningot dobni rá, vagy meg se mukkanni...

Ami engem illet, ezen elgondolkodva úgy találtam, hogy különösebb jelentősége nincs egy efféle jel bevezetésének, mert a programsor feldolgozása során úgyis mindig egyértelmű, mikor van vége az utasításnak, tudniillik abból derül ez ki, hogy olyan karaktert olvasunk be, ami nem illik az épp feldolgozott parancs

kontextusába. Például ha egy decimális számsorozatot olvasunk be, s egy számjegyet egy R betű követ, nyilvánvaló, hogy ott vége van a számsorozatnak, s az azt igénylő utasításnak.

Nem tagadható azonban, sokan hajlamosak az utasításokat pontosvesszővel lezárni - én is. Emellett e jel jelentősen megnöveli a kód olvashatóságát. Úgy döntöttem, legyen a pontosvessző egyszerűen egy ÜRES UTASÍTÁS, ami eképp ki-tehető, de a használata nem kötelező! S csináljuk meg olyanná, hogy a funkciója mindössze az legyen, hogy előlépteti a programszámlálót a következő whitespace karakterig. Eképp ha egymagában áll, vagy szóköz, tab stb követi, semmit se csinál - ha azonban valami kulcsszó előtt áll, akkor azt nem veszi figyelembe az interpreter, azaz semmit se értelmez, ami pontosvesszővel kezdődik! Íme a rutin:

```
int pontosvesszo(F& f) { // Ellépteti a P pointert a következő whitespace karakterhez
while(f.P<f.phossz) {
if(wspc(f.p[f.P])==0) {f.P++;continue;}
break;} // while vége
return 0;
}
```

E megoldás amiatt is nagyon jó, mert így megvan a lehetőségünk rá, hogy olyan karaktereket is definiáljunk utasításként, amiket majd később valamikor operátornak is használunk. Például, mondjuk a „/” jelet. Amennyiben ezt egy szám után írjuk, az interpreterünk feltehetőleg úgy gondolná, egy osztásjelről és nem egy utasításról van szó. Mindjárt más azonban a helyzet, ha a szám után van egy pontosvessző, majd valami whitespace, s utána következik a per-jel. Ezesetben egyértelmű, hogy most nem operátorként, hanem utasításként használjuk e per-jellet. Azaz nálunk a pontosvessző SZEPARÁTORKÉNT működik, ám használata többnyire csak opcionális, kivéve az egészen speciális eseteket.

5. fejezet: Változókezelés, meg azok a fránya címkék

Minden ismertebb programnyelvben létezik az a fogalom, hogy „változó”. Ennek fényében talán meglepő lehet Olvasóm számára az a tény, amit már a Bevezetőben is említettem, hogy gépi kódú szinten, azaz a processzor számára ilyesmi egyszerűen nem létezik! Hacsak talán a processzor regisztereit nem tekintjük változóknak, de azokból semmi esetre sincs annyi, amennyi egy szokványos, „hétköznapi”, „átlagos” programban elő szokott fordulni, ami meg az ismertebb programnyelveket illeti, ezekben szinte kivétel nélkül potenciálisan végtelen a felhasználható változók száma!

Ez úgy lehetséges, hogy a „változó”, az semmi más, csak egy címke, ami egy valahány bájt nagyságú memóriaterületet azonosít. Többnyire ez úgy működik, hogy a programozó a program elején felsorolja a programban majd használatos változókat, ezek típusával együtt (a típusból következik, hány bájt hosszúságú memóriaterület kell a változónak), ezeknek a compiler lefoglalja a kellő nagyságú memóriaterületet, aztán minden alkalommal amikor a megfelelő nevű változóra hivatkozás történik a programban, a név helyett e memóriacímet használja.

Amennyiben interpreterről van szó és nem compilerről, a megoldás hasonló, azzal a különbséggel, hogy sokszor megengedett, hogy a változót ne kelljen a legelején „deklarálni”, hanem annak legelső használatakor foglalt le neki memóriaterületet

az interpreter. Ellenben akár így történjék akár úgy, egy interpreter kénytelen a változókat úgy kezelni, hogy minden alkalommal amikor hivatkozás történik rá, azaz felbukkan a neve a forráskódban, ki kell keresse a megfelelő táblázatból, az adott névhez miféle memóriacím is tartozik, hogy azt a memóriaterületet használja mint „változót”. Illetve persze ha nem találja meg a változó bejegyzését a táblázatában, akkor vagy hibát jelez, vagy egyszerűen kreál neki egy új bejegyzést - hogy melyik megoldást választja, attól függ, miként írták meg azt az interpretert. (Mindegyik megoldásnak vannak előnyei/hátrányai: új bejegyzés készítésénél kényelmes, hogy nem kell előre deklarálni a változót, ugyanakkor viszont jelentős hibalehetőség. Tegyük fel az ASZTAL nevű változóval dolgozunk, s ennek már van érték adva. Ám valahol véletlenül úgy írjuk hogy ASTAL. Kihagytuk a Z betűt. S erre e változónak kreál a program egy új bejegyzést, s azzal számol tovább, holott ezen új változó kezdeti értéke feltehetően nulla. És még ez is a jobbik eset, ha tényleg nulla a kezdeti értéke, mert sokkal nehezebb felderíteni az ilyen hibát, ha még a kezdeti érték se garantált egy változónál, hanem az lehet akármiféle odakeveredett véletlenszerű memóriaszemét...)

Na most, könnyű belátni, hogy fordítóprogram esetén nem történik nagy baj ha megengedünk sokkarakteres neveket, mert csak egyszer kell lefordítani azt, azután már úgyis semmi jelentősége a változóneveknek, mindegyik név helyére memóriacím kerül. Interpreter esetén azonban a helyzet ennél sokkal de sokkal rosszabb! A baj nem is annyira az, hogy a név esetleg sok karakterből áll, hanem az, hogy akárhány karakterből álljon is, de minden használat esetén kivétel nélkül végig kell bogarásznia a táblázatot, hogy a változónevet behelyettesítse a megfelelő memóriacímmel! Képzeld el, ha van akár csupán 100 változónk is, s egy olyat használunk ami a lista vége felé helyezkedik el, s ezt egy 10 ezerszer lefutó ciklus közepén használjuk, ráadásul a ciklusmagon belül is 5 alkalommal hivatkozunk rá! Ez azt jelenti, hogy ha csak 1 karakterből áll is a változó neve, mire lefut a ciklus, 5 millió összehasonlítást végez az interpreterünk, még legjobb esetben is! Ha a változónév ráadásul nem is 1 karakteres hanem több, akkor meg pláne ennél is több az összehasonlítások száma!

És ez csak EGYETLEN változó esetén lett számolva, most képzeljük el, hogy effélék történnek minden változó minden egyes használata során... Az efféle dolgok miatt lassúak az interpreterek, ezért van az, hogy kicsit is sebességigényesebb feladatokat nem ilyen nyelveken írnak.

Ezt nyilvánvalóan nem engedhetjük meg magunknak. **Ha van valami amit MINDENFÉLEKÉPPEN el kell kerülnünk, KERÜL AMIBE KERÜL, a legdrasztikusabb eszközökkel is, na hát akkor az ÉPPEN PONTOSAN EZ!** Egészen egyszerűen SEMMI sincs egy interpreter-típusú programnyelvben, ami ennél kritikusabb tényező lenne a sebességet illetően. Lehet még rengeteg mindenféle más akármivel pöcsölni egy programnyelv megalkotásánál, de ha az a nyelv interpreter típusú, akkor teljesen biztos, hogy az összes többi mindenféle együtt sem fontos tizedannyira se a sebességet illetően, mint ez a változókezelési rész!

A dolgon az sem segítene, ha az interpreter indulás előtt átvizsgálná a programot, azután kutakodva, miféle változókra is történik ott hivatkozás. Ezzel csak annyit érne el, hogy előre tudná, hány változó lesz, így mennyi memóriaterületet kell lefoglalni nekik összesen illetve külön-külön. Ettől azonban még végrehajtás

során ugyanúgy minden alkalommal amikor változónévhez ér, meg kéne keresnie az ahhoz tartozó memóriacímet valamiképp.

Mit is kezdhethünk ezzel a problémával?!

Vizsgáljuk meg először is, mire használjuk egy programban a változókat! Nos, olyan adatok tárolására, amik nem állandóak, nem konstansok, mert megváltoznak, vagy legalábbis megváltozhatnak. Lényegében tehát ugyanarra használjuk őket, mint a tömböket, azzal a különbséggel, hogy a nem tömb típusú változókhoz valamiképpen „személyesebben kötődünk”, ezek valamiféle értelemben „egyéni-séggel bírnak”, rájuk nem annyira jellemző hogy csak „egy a sokból”, mint egy olyan változóra ami mondjuk az A[345], vagyis az A tömb 345-ödik eleme. Utóbbi esetén tudjuk, hogy efféle változóból nagyon sok van, és mind „egy kaptafára megy”. Ha azonban van egy SZIN, egy EVSZAM, meg egy NEV nevű változónk, meg esetleg egy I nevű is amit ciklusváltozónak használunk, akkor tudjuk, hogy ezekből csak egyetlenegy van a programban, vagy legalábbis abban a függvényben ahol épp dolgozunk velük.

Tegyük fel azonban, hogy legalábbis a SZIN, az I és az EVSZAM is egyaránt numerikus érték, ami a 0-65535 közé esik, azaz mindegyik egy **unsigned short int** típusú változóval jellemezhető. (A SZIN ezesetben természetesen a konkrét szín egy számkódja). Könnyű belátni, hogy ha ezek nem egyedi változókként volnának deklarálva a programunkban, hanem volna egyetlen tömb eképp:

unsigned short int d[3];

akkor is működőképes maradna a programunk, amennyiben a SZIN helyett azt írnánk, hogy **d[0]**, az EVSZAM helyett azt írnánk hogy **d[1]**, és az I helyett azt írnánk, hogy **d[2]**. Abszolút semmit se vesztené a programunk se funkcionalitásában, se gyorsaságban, annál is inkább, mert ha ezt netán egy compiler fordítaná le, az úgyszintén úgy tenne, hogy lefoglalna egy memóriaterületet az összes unsigned short int változónak, aztán ebben valahányadik értéket feleltetné meg minden egyedileg elnevezett változónknak. Azaz telibepontosan épp olyanná alakítaná át a programunkat fordításkor, mint ahogy azt leírtam fentebb a „tömbösítés” esetén.

Ez a megfontolás már közel visz bennünket a problémánk megoldásához. Amennyiben ugyanis az interpreterünk tudná már induláskor, hol kezdődik mondjuk az unsigned int változóknak lefoglalt memóriaterület, az esetben ha nem egyedi változóneveket használunk csak efféle tömböket, aholis a tömb neve azt mondja meg milyen típusú változók vannak benne, akkor ha ilyenhez ér az interpreter, nem kell semmit se keresgélnie holmi változónevek közt, nem kell a név karaktereivel összehasonlításokat végeznie, hanem csak fogja a forráskódban a megfelelő helyen talált számot, ami a tömbindex, ezt beszorozza a változó típusához tartozó mérettel, s máris megkapja, az adott típusú változóknak fenntartott memóriaterület hányadik bájtyán kezdődik a változóhoz tartozó adatok sora!

E módszerrel szemben ellenérvként vetődhet fel Olvasómban, hogy a SZIN változó könnyebben megjegyezhető, mint az, hogy mondjuk U[17].

Ez igaz. Nem is vitás, hogy amíg csak egy nagyon rövid programot írsz, amiben nincs több mint mondjuk 10-15 változó, addig az egyedi nevek messze sokkal jobban megjegyezhetőek. Abban a pillanatban azonban, hogy valami igazán komoly programba kezdesz, ahol SOK változó van, akár csak már mindössze 2-3 tucatnyi is, máris úgy véled majd, nem is olyan nagyon könnyű olyan nevet kitalálnod, mely nem túl hosszú, ugyanakkor egyedi és még kellően informatív is, azaz könnyen megjegyezhető! Maradjunk csak a színek mellett: tegyük fel, terminálemulátort írsz, ahol illik lehetőséget adnod mindenféle színek beállítása. Hányféle színed is lesz?

ALAPSZIN
IRASSZIN
HATTERSZIN
KURZOR_ELOTETSZIN
KURZOR_HATTERSZIN
INVERZSZIN
KIEMELT_ELOTETSZIN
KIEMELT_HATTERSZIN
KONYVTARSZIN
KONYVTAR_HATTERSZIN
KIEMELT_KONYVTARSZIN
KIEMELT_KONYVTARHATTERSZIN
ERRORMESSAGE_SZIN
ERRORMESSAGE_HATTERSZIN
NEMAKTIVKERET_SZIN
AKTIVKERET_SZIN
SYMLINKSZIN
SYMLINKHATTERSZIN
SYMLINK_KIEMELT_SZIN
SYMLINK_KIEMELT_HATTERSZIN
STATUSBAR_IRASSZIN
STATUSBAR_ALAPSZIN
STATUSBAR_INVERZSZIN
CLOCK_SZIN
CLOCK_HATTERSZIN
BUTTON_ALAPSZIN
BUTTON_IRASSZIN
ACTIVEBUTTON_ALAPSZIN
ACTIVEBUTTON_IRASSZIN
NEMKIVALASZTHATO_IRASSZIN
NEMKIVALASZTHATO_HATTERSZIN

Ennyi színféleséged mindenképp lesz, sőt minden bizonnyal még sokkal több is! És remélem elismered, nem csaltam: igyekeztem annyira egyértelmű neveket kitalálni, amennyire az csak lehetséges (magyar anyanyelvű ember számára). Ennek ellenére, szerintem minden alkalommal amikor elérkezel valamely olyan programrészhez ahol valamelyik színre kell hivatkoznod, kénytelen leszel utána-nézni valahol - valamely magadnak készített listában, jegyzetben vagy a program korábban megírt részeiben - hogy most hogyan is nevezted el pontosan azt a változót?! Mi is annak a pontos neve: NEMKIVALASZTHATO_HATTERSZIN, vagy

NEMKIVALASZTHATOHATTERSZIN, vagy HATTERSZIN_NEMKIVALASZTHATO, vagy valami más, és akkor még nem is emlegettem azt a rengeteg lehetséges variációt ami abból adódik, hogy a változónév mely karaktereit írtad **NAGY**, s melyiket **kis** betűvel!

Azaz semmiféleképp sem tudod megúszni kicsit is komolyabb esetekben, hogy táblázatot készíts magadnak a felhasznált nevekről, s arról, melyiknek pontosan mi is a funkciója. Mert ez itt a színekkel még egészen „istenes eset”, mert tényleg eléggé egyértelmű, mire használod őket. De ha van neked egy függvényen belül akár csak féltucat olyan változód is, amiknek az a nevük hogy:

I, J, C0, C1, La3, STORE8,

nos, akkor már mérget veszek rá hogy ha most nem is gabalyodol bele, melyik mit jelent, de már 2 hónappal a program elkészülte után halovány fogalmad se lesz ezek jelentéséről, ami azért elég ciki, ha hirtelen eszedbe jut hogy valamit módosítanál a progin. Mert mondjuk ezt kívánja tőled a Főnök, vagy a Megrendelő...

Azaz mindenféleképp kell írni részletes listát úgyis arról, melyik változó mire való, akkor meg már tökmindegy, akkor akár úgy is kinézhet a lista, hogy:

.....
U[17] = keretszín
U[18] = írásszín

.....
U[132] = a beolvasott karaktereket számolja a FILEBEOLVAS függvényben
.....
U[182] = a futamidő hónapokban számolva
.....

vagy valami más hasonló módszert alkalmazol. Ezesetben viszont meglesz az az előnyöd is, hogy mire így elkészíted a programot, majdhogynem kész leszel a teljes dokumentációval is!

Sőt, e módszernek van még egy cseppet sem elhanyagolható előnye, melyre ugyan csak ritkán van szükség, de akkor aztán jaj de nagyon jól is jön! Így ugyanis, hogy nincsenek is egyedi változód, hanem mindegyik egy tömbnek a része, így könnyedén megteheted, hogy indexeled e tömböt, és egy ciklussal a tömböt végigjárva, sorra minden változóddal elvégzel valami műveletet. Például kiíratod egy kereszt-referencia-táblázatba, a képernyőre vagy egy lemezes állományra, dumpolsz, vagy ha hirtelen eszedbe jut hogy írsz egy alprogramot ami meg kell kapja e program összes változóját valamiért, vagy csak sokat belőlük... Hej de jó is akkor, hogy mindez megoldható egyetlen ciklussal, de bezzeg ha 60 egyéni változód van mindenféle egymástól teljesen eltérő, kinnal szült névvel ellátva/beazonosítva, na akkor főhet a fejed miként oldd is meg ezt, mert mindet egyedileg kell lekezelni, vagy pedig írhat sz át mindent, hogy mégis tömb legyen minden, ahogy azt én már most is javasolom...

Maradjunk tehát abban, hogy leendő szépséges mau programnyelvünkben egyáltalán nem lesznek holmi egyedi változók, csak efféle tömbszerűségek. Ennek

fényében már csak arra kell valami módot találnunk, hogy az interpreterünk tudja, melyik típusú változóknak pontosan mekkora tömböt is kell kreálnia induláskor, azaz melyik típusú változóból mennyi lehet!

Erre lehetne azt a módszert választani, hogy valamely speciális utasítással ezt a programozó megmondja a program elején az interpreternek, például így:

V USI 200

ami azt jelentené (mondjuk) hogy az **unsigned int** típusú változókból nem lehet több, mint legfeljebb 200 darab, nyilván 0-199 közti indextartománnyal.

Végeredményben ez már majdnem jó megoldás is volna, csak ennek azért van pár apró kényelmetlensége. Mert ugye mi van akkor, ha mégis több változó kéne, ekkor a programozó mindig kéne hogy módosítgassa e bejegyzést. Aztán, mi van ha valamiért efféle bejegyzésből máshova is keveredik a programban, s képzeljük csak, egyszer már lefoglalt területet az interpreter 200 efféle változóra, erre most jön neki az utasítás, hogy foglaljon le 187 ilyen típusú változónak memória-területet! Ciki. Muszáj volna ellenőrizni, történt-e már lefoglalás.

A legnagyobb baj mégiscsak az, hogy ezesetben kivétel nélkül minden alkalommal amikor egy változóra hivatkozik a program, az interpreternek kutya kötelessége volna ellenőrizni, a változó indexe nem lépi-e túl a lefoglalt tartományt. Kétségtelen, ez nem akkora időveszteség, mintha egy több karakteres nevet ellenőrizgetne valami táblázatban, de azért akkor is lassító tényező. Csekélységnek tűnhet, hiszen csak egyetlen, előjel nélküli számot kell összehasonlítani egy másik efféle számmal, azonban ezt TÉNYLEG el kell végeznie kivétel nélkül minden egyes változó minden egyes használatakor! Igen, cikluson belül is... Azért ez már meggondolandó. Mert mégsem csak csupán arról az egyetlen összehasonlító műveletről van szó ilyenkor, de ehhez elő kell szednie valamely memóriarekeszből azt a számot is, ami a maximális index mérete, ami e fenti példa esetén tehát a „200”-as szám, azaz ilyenkor nemcsak összehasonlító-, de memóriaműveletet is végezne. Időveszteség, na.

Gondolkozzunk tehát, mielőtt ebbe beleugranánk, nem kerülhetjük-e el ezt az ellenőrizgetést! Nos, IGEN, elkerülhetjük, de csak az esetben, ha garantált, hogy a változó indexe semmi esetre sem lépheti túl amúgysem a változó lehetséges maximális indexét! Azaz, ha az index olyan típustartományba esik, amelynek maximális értéke is olyan, hogy mindig van lefoglalva annyi darab változónak hely!

Ez az indextartomány kizárólag az unsigned char érték lehet, a 0-255 közti tartomány, mert ami ennél nagyobb, az az unsigned short int, s az már a 0-65535 értéket jelentené, ami nyilvánvalóan túl sok. A 256 azonban nem tűnik soknak. De túl kevésnek sem, szerencsére! Fontoljuk csak meg, nem jellemző egyetlen programra se, hogy több mint 256 egyedi változót használunk. Ha ennél több mindenféle izémizénk van, amúgy is tömböket szoktunk használni, nemigaz?

Mert gondolkozzunk csak: hányféle „alaptípusról”, „elemi típusról” is lehet szó? A C nyelv esetén ezek a következők:

- unsigned char
- signed char
- unsigned short int
- signed short int
- unsigned int
- signed int
- unsigned long long
- signed long long
- float
- double
- long double

Ez 11 típus. Igaz ugyan hogy vannak pointerok is, azaz mutatók, de minden pointer úgyis vagy 4 bájtos (32 bites rendszerek esetén) vagy 8 bájtos (64 bites rendszereknél), eképp azok simán megfeleltethetők egy unsigned int vagy egy unsigned long long értéknek.

Azaz ha úgy döntünk hogy eleve minden mau programhoz induláskor lefoglalunk fixen minden elemi típus számára 256 értéket, az nem lesz olyan nagyon nagy memóriapocséklás!

Sőt, ezt is csökkenthetjük. Többnyire azért cseppet se kell ennyi rengeteg változó minden típusból. Miért is ne lehetne úgy, hogy ha nekünk nem kell mondjuk ennyi long double változó, akkor a feleslegesen e célra lefoglalt tárterület bájtjait mondjuk unsigned char változók számára használjuk fel?!

Azaz, vegyük csak fel a következő deklarációt:

```
union V { // Az univerzális változó
    unsigned char    c[16]; // Mérete 16 bájt
    signed char      C[16]; // Mérete 16 bájt
    unsigned short int i[ 8]; // Mérete 16 bájt
    signed short int I[ 8]; // Mérete 16 bájt
    unsigned int      L[ 4]; // Mérete 16 bájt
    signed int        l[ 4]; // Mérete 16 bájt
    float             f[ 4]; // Mérete 16 bájt
    unsigned long long g[ 2]; // Mérete 16 bájt
    signed long long  G[ 2]; // Mérete 16 bájt
    double            d[ 2]; // Mérete 16 bájt
    long double D; // Mérete 16 bájt
};
```

// A V unionban elhelyezkedő mezők egymáshoz viszonyított pozíciói:

```
// | c[0] | c[1] | c[2] | c[3] | c[4] | c[5] | c[6] | c[7] | c[8] | c[9] | c[10] | c[11] | c[12] | c[13] | c[14] | c[15] |
// | c[0] | c[1] | c[2] | c[3] | c[4] | c[5] | c[6] | c[7] | c[8] | c[9] | c[10] | c[11] | c[12] | c[13] | c[14] | c[15] |
// | i[0] | i[1] | i[2] | i[3] | i[4] | i[5] | i[6] | i[7] | i[8] | i[9] | i[10] | i[11] | i[12] | i[13] | i[14] | i[15] |
// | l[0] | l[1] | l[2] | l[3] | l[4] | l[5] | l[6] | l[7] | l[8] | l[9] | l[10] | l[11] | l[12] | l[13] | l[14] | l[15] |
// | f[0] | f[1] | f[2] | f[3] | f[4] | f[5] | f[6] | f[7] | f[8] | f[9] | f[10] | f[11] | f[12] | f[13] | f[14] | f[15] |
// | g[0] | g[1] | g[2] | g[3] | g[4] | g[5] | g[6] | g[7] | g[8] | g[9] | g[10] | g[11] | g[12] | g[13] | g[14] | g[15] |
// | G[0] | G[1] | G[2] | G[3] | G[4] | G[5] | G[6] | G[7] | G[8] | G[9] | G[10] | G[11] | G[12] | G[13] | G[14] | G[15] |
// | d[0] | d[1] | d[2] | d[3] | d[4] | d[5] | d[6] | d[7] | d[8] | d[9] | d[10] | d[11] | d[12] | d[13] | d[14] | d[15] |
// | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D |
```

Sajnos az union változóinak fentebb felvázolt egymáshoz viszonyított elhelyezkedése csak ROPPANT VALÓSZÍNŰ (mármint Intel processzoros PC gépek esetében, 64 bites gépen), de igazából nem garantált automatikusan, a fordító által. Azért nem, mert valami perverz indítástól vezettetve a C és C++ nyelveket úgy konstruálták meg, hogy ezen alaptípusoknak már a MÉRETE SEM GARANTÁLT,

hanem architektúra- és implementációfüggő. Elméletileg - azt hallottam legalábbis - még az se garantált, hogy egy char változó 8 bites. Mondjuk ez elég valószínű persze, de akkor se TOTÁL biztos azért. Mindenesetre, többnyire azért biztosak lehetünk a fent felvázolt elhelyezkedésben, s abban, hogy 1 ilyen union összesen 16 bájtot foglal el, eképp a teljes unsigned char tartományra eső 256 darab efféle mindösszesen 4096 bájtot, vagyis pont 4 kilobájtot, ez majdnem biztos hogy így lesz igaz minden épeszű architektúrán. De akkor is csak MAJDNEM lesz biztos és nem TELJESEN, mert vannak nem épeszű architektúrák és fordítóprogramok is, vagy legalábbis effélék nem kizárhatóak. Mint említettem, mindezt Intel processzoros gépen próbáltam ki, 64 bites architektúrán, konkrétan egy Lenovo ThinkPad T530 -as gépen, s természetesen Linux operációs rendszerrel, a GCC csomag 4.8.1 -es verziójával, annak g++ fordítóprogramjával.

Ami a 32 bites rendszereket illeti, ott esetleg elképzelhető, hogy a long int felel meg a 4 bájtos méretnek, és a long double mérete 10 bájt, miközben a közönséges (signed vagy unsigned) int mérete csak 2 bájt. Hogy szarjon sündisznót de ne is egyet az a seggfej, aki efféle kavarást megengedhetőnek tartott és így találta ki e nyelvet! (Emlékeztetőül írtam ezen értékeket a 32 bites rendszerekre, abból az időből valók az emlékeim amikor még Win98 alatt írtam asszem BorlandC fordítóval programot, ezelőtt kb 10 évvel vagy még többel. Akkor e típusoknak ezek voltak a méreteik nálam. Majdnem biztos, hogy egy mai linuxos 32 bites rendszernél ezen értékek nem lesznek igazak.)

Mindenesetre itt és most kijelentem, hogy e „mau” programnyelvemben GARANTÁLT a V unionban szereplő mezők elhelyezkedése és mérete. Ez azt jelenti, hogy bár ez automatikusan nem garantált a C fordító részéről pusztán a típusokat megadva, amikor más architektúrára portoljuk a mau nyelvet, de ezesetben a mau interpreter forráskódján kell változtatni úgy, hogy ez a mau nyelvben már igenis garantált legyen ott is! Magyarán: **A V struktúra fentebb bemutatott felépítése SZABVÁNY a mau nyelvben!** Szabvány, amit ha maga a C fordító nem is garantál alaphól, de addig kell buherálni az interpreter forráskódját, amíg ez létre nem jön. Ami nem felel meg ennek, az definíciószerűen NEM a mau nyelv hanem valami más. Ebből következik, hogy az is garantált, hogy a **sizeof(V)** pontosan 16 bájt méretű legyen.

Vegyük aztán fel ezt az F struktúra deklarációjába:

```
V v[256];    // A változók tömbje. Induláskor garantáltan nulla mindegyik változó mindegyik bájtja!  
  
// A v tömbben levő változók unionjának indexelésére. Kezdetben ez mind 0.  
USC vindexc[256];  
USC vindexC[256];  
USC vindexi[256];  
USC vindexI[256];  
USC vindexl[256];  
USC vindexL[256];  
USC vindexf[256];  
USC vindexg[256];  
USC vindexG[256];  
USC vindexd[256];
```

Ha már azt írjuk hogy garantáljuk a változók kezdeti nulla értékét, tartsuk is be ezen ígéretünket, azaz a PGM konstruktorába vegyük bele e sorokat:


```

register int i,j;for(i=0;i<256;i++) { // A változóindexek lenullázása
f.vindexc[i]=0;f.vindexC[i]=0;f.vindexi[i]=0;f.vindexI[i]=0;f.vindexl[i]=0;f.vindexL[i]=0;f.vindexf[i]=0;f.vindexg[i]=0;
f.vindexG[i]=0;f.vindexd[i]=0;
for(j=0;j<16;j++) { // A változók lenullázása
f.v[i].c[j]=0;}}

```

Vagyis, egy úgymond „változó” a számunkra mindig egy 16 bájtos memóriaterületet azonosít be, mely 16 bájtot hol ilyen, hol olyan típusú változók sorozatának („mini-tömbjének”) tartunk.

Minthogy az unionon belül az egyes változótípusokból több is van, kell valami ami tárolja ezek konkrét indexét, ami az „aktuális”-ra mutat, ezt a feladatot látják el a „**vindex**”-szel kezdődő nevű tömbváltozók. Nem véletlen, hogy a long double típusúra nincs **vindexD** nevű tömb: minthogy abból eleve csak egyetlen lehet, emiatt annak nem kell indexet tárolni.

Na de hogyan is hivatkozunk e változókra, s pláne a típusaikra, indexükre...?

Minthogy nekünk maximum 256 változónk van, egy változót értelemszerűen a sorszáma azonosít, mely 0-255 közé esik. Ennél nagyobb sose lehet, s ez épp belefér egy unsigned char értékbe. Ha megegyezünk abban, hogy minden változó neve mondjuk a @ karakterrel kezdődik, akkor ennek megfelelően mondjuk a

@65

egy tökéletes változónév nekünk: A 65-ödik változónkat jelenti!

Igenám, de ez akkor is azt jelenti, hogy mindig amikor e változóra hivatkozunk, az interpreterünk kénytelen a 65-ös szám mindkét karakterét feldolgozni, azaz egyrészt beolvasni, másrészt szorozni-osztani, azaz kikalkulálni, ez miféle számot is jelent a tízes számrendszer szerint, azért, hogy megkapja a változónk indexét! Továbbá, hát azért egy PICI könnyebbséget már csak engedjünk meg magunknak, ha ez nem lassítja le az interpretert... pláne, ha még gyorsítja is! Azaz: Miért is ne lehetne úgy, hogy amikor egy @ jelhez ér az interpreter, s ebből tudja hogy valami változóról lesz szó, akkor veszi a következő karaktert, s megnézi, az merő véletlenségből nem valamely kis- vagy nagybetű-e! Ha az, vagyis az A-Z, a-z tartományok valamelyikébe esik, akkor egyszerűen ezen karakter ASCII kódja lesz a változó indexe, azaz úgymond „neve”! Ez már egészen barátságos: a **@A**, **@k** stb típusú nevek könnyen megjegyezhetőek! S ezekből lehet összesen 52, mert alkalmazható az angol ABC mind a 26 kis- és mind a 26 nagybetűje! Többnyire ennyi egyedi változó elég is, ritkán van szükség ennél többre.

Amikor viszont mégis kéne több, az se gond, akkor használunk ugyebár számokat. Na de hoppá, ha már számokat írkálunk a @ jel után, miért is kéne ragaszkodnunk ahhoz, hogy ott okvetlenül csak decimális számok lehessenek?! Simán elintézhethetjük, hogy interpreterünk képes legyen egy egész rakás számrendszer kezelésére egész számok esetén... Ez könnyű, csak ki kell találnunk valami logikus jelölést, melynek hála már a legelső karakter beolvasásakor eldöntheti, egy szám miféle számrendszerben értelmezendő!

Erre a következőt találtam ki:

- Ha a szám első karaktere a 0-9 karakterek valamelyike, akkor az DECIMÁLIS szám.
- Ha a szám első karaktere a % jel, akkor az egy BINÁRIS szám.
- Ha a szám első karaktere az o vagy O betű (Figyelem! BETŰ és nem a nulla számjegy!) akkor az egy OKTÁLIS szám.
- Ha a szám első karaktere a \$ jel, akkor az egy HEXADECIMÁLIS, azaz 16-os számrendszer beli szám.

Ha már elemzi az interpreter az első karakter jelentését, azt is megtehetjük, hogy az ' azaz aposztróf jel pedig azt jelenti, hogy az az ASCII kód, amit az aposztróf utáni karakter képvisel, akkor is ha az nem kiiratható karakter vagy akármi más is. Ennek megfelelően az alábbi jelölések mind ugyanazt a változót határozzák meg:

```
@A
@'A
@65
@$41
@%1000001
```

Az természetesen nyilvánvaló, hogy e fenti variációkból a legelső, a **@A** stílusú a leggyorsabban feldolgozható az interpreter számára, de az csak akkor használható, ha a változó kódja egy kiiratható karakter, sőt kifejezetten épp betű.

E két variáció:

```
o101
0101
```

szintén a decimális 65 értéknek felel meg, de a változókra utaló @ jel után nem használható, mert ott az „o” vagy „O” karakter önmaga ASCII kódját jelenti.

Felmerülhet a kérdés a Tisztelt Olvasóban, miért is egyénieskedem azzal, hogy az oktális számokat az „o” vagy „O” karakterrel vezetem be, s nem a már „megszokottá vált” azon konvenciót alkalmazom, hogy amely szám nullával kezdődik, az oktálisnak értelmeztessek?!

Azért nem ezt teszem, mert ez rengeteg randa hiba forrása lehet. Épp ma amikor e sorokat írom, történt egy bejegyzés pont erről az egyik számítástechnikai portálra, itt a forrás linkje, de idézem is:

http://hup.hu/node/130044?comments_per_page=9999

Egy apró kellemetlenség, amit bash-ben könnyű elkövetni, nekem legalább is sikerült. Alkönyvtár neve ilyesmi:

20140024.123

Ebből az első 4 számjegy az év, utána a pontig az, hogy az év hanyadik napja van. Igen ám, de ha ez nullával kezdődik, a bash oktálisan értelmezi, így aztán a 0008-ra hibát fog dobni. Amire nem, az is helytelenül lesz feldolgozva.

Így aztán le kellett csippentennem a vezető nullákat. Amíg nem kaptam hibaüzenetet, elkerülte a figyelmem.

Na az ilyesmi miatt nem alkalmazom ezt a módszert, hogy a vezető nulla oktális számot jelentsen. Nem e fenti blogbejegyzés miatt döntöttem így, réges-rég elhatároztam már ezt magamban, már ÉVEKKEL KORÁBBAN, amikor csak távoli kódös vágy volt hogy írjak egy programnyelvet, de mély meglepéssel nyugtáztam e

blogbejegyzés megszületését, mert megerősítette azon nézetemet, hogy nagyonis bölcsen döntöttem, s jó úton járok!

Az meg hogy miért a \$ a jele nálam a hexa számoknak... Nos, korábban már megígértem, visszatérek majd arra a témára, hogy a C nyelv a „0x” karakter-sorozatot használja a hexa számok bevezetésére, s ez nekem nem tetszik! Miért is nem tetszik ez nekem?

1. Mindenekelőtt, én még a C-64 (Commodore-64) idejében megszoktam, hogy a hexa számok ott a \$ karakterrel kezdődnek, márpedig ha írok egy nyelvet a magam örömeire, az legyen már olyan, amilyennek szeretem! Ez persze szubjektív meg érzelmi indok csak, de mindjárt jönnek racionális érvek is.

2. A „0x” nem 1 hanem 2 karakter, emiatt feldolgozása lassabb, mintha csak 1 karakter kéne ehhez.

3. A „0x” kétkarakteres jelölésmód szembemegy azzal az elvvel, hogy az interpreter tudja lehetőleg rögvest minden egyes karakter beolvasása után is már azonnal, mit kell tennie.

4. A „0x” esetén meg kéne engednem a „0X” jelölést is, ami plusz egy összehasonlítás, ez lassítja a programot és bonyolítja a kódot. Nekem épp emiatt már az se tetszik hogy az oktális számoknál is megvan egyaránt az „o” és az „O” is engedve, csak nem volt jobb ötletem, mert sajnos a speciális, nem alfabetikus karakterek száma csekély az ASCII kódkészletben.

Ebből már rögvest következik, nagyjából miként is néz majd ki nálunk valami olyasféle függvény, amit holmi „általános értékbeolvasó rutinnak” nevezhetnénk jobb elnevezés híján:

Megvizsgálja az aktuális karaktert, hogy az a @ jel-e. Ha igen, tudja hogy változóról van szó, aminek értékét csak akkor tudja meghatározni, ha beolvassa az e @ jel után következő számértéket, ami meghatározza a változó indexét. Ezért előre lép egy karaktert, **ÉS MEGHÍVJA ÖNMAGÁT REKURZÍVAN e függvény**, elvégre miért is ne határozhatná meg a változó indexét egy másik változó ügyebár...

Amennyiben a karakter nem a @ jel, akkor sorra megvizsgálja hogy az a 0-9 tartományba esik-e, vagy az A-Z, a-z tartományba, vagy azonos-e az o, O, \$, ' karakterek valamelyikével, s ha igen, kiszámolja azt, ami kell, s visszaadja a megfelelő eredményt. Minthogy nem lenne valami takarékos megoldás megírni e függvényt minden létező egész típusra, így nemes egyszerűséggel legyen ennek a visszatérési értéke unsigned long int, abba kábé minden más szám belefér, ha ennél kisebb érték kell, majd átkonvertálja (castolja) magának az a függvény, mely őt meghívta. A lebegőpontos formátumokkal perpillanat még ne foglalkozunk, majd később, ha az egészekre már jól működik a programunk, akkor...

Mielőtt nekiesnénk e rutin lekódolásának, törjük egy picit az okos kis buksikánkat, milyen körülmények közt is hívjuk meg majd ökelmét! Ugye az tuti hogy minden olyan esetben, amikor valamiféle egész számra van szükségünk a program során. Na de mely esetek is lesznek ezek? Ezt sajnos még nem tudhatjuk előre, vélhetőleg temérdek ilyen eset lesz, emiatt inkább úgy módosítsunk töpren-

gésünk irányát, hogy miféle értékeket akarunk még mi meghatározni egész számmal, vagy valami olyasmivel ami megfeleltethető valamiféle egész számnak! Mert jobb az összes ilyesmit itt és most elintézni, e függvény belsejében. Találunk-e még valami olyasmit, ami egész számmal jellemezhető, s fontos a programunk számára?

Találunk, bizony! S ezek nem mások, mint a CÍMKÉK...

Olvasóm most bizonyára nagyot néz. Elment talán az eszem?! Már ugyan hogy is lehetne egy címke azonos egy számmal?! Még ha beszámoztam volna a program-sorokat mint a Basic esetében, de arról már lemondtam ezen iromány legelején... De még ha valamiképp megfeleltetek is egy-egy címkét egy-egy számnak, már ugyan miért is volna az változó, mi köze volna ezeknek a változókhöz?!

Megmondom. És rögvést elárulom, hogy e kérdéskörhöz elérve, bizony rém sokáig „lustálkodtam”, mert ez irtó komoly kérdéskör, nem lehet elintézni egy kézlegyintéssel nemtörődöm módon, ugyanis a címkék nagyonis sok mindenre kihatással bírnak, az olyan fontos strukturális elemekre mint az elágazások és ciklusok, függvények és szubrutinok... Ja, hogy mi köze van a címkéknek egy szubrutinhoz, ezt kérdezed? Már bocsánat, de a címke ugyebár végsősoron egy NÉV. Na most, egy szubrutinnak és egy függvénynek is van tudtommal NEVE... Tudom, persze, a „normálisabb” programnyelvekben tök másképp néz ki a kettő. SZERINTED. Szerintem ugyanis nem. Vegyük csak alapul a Basic nyelvet mondjuk. Címkehasználat:

goto 3245

Szubrutinhívás:

gosub 3245

Nem látok különbséget köztük...

A C nyelvben:

Címkehasználat:

goto ideugorj;

Függvényhívás:

ez_egy_alprogram();

Itt sincs sok különbség: az „ideugorj” és az „ez_egy_alprogram” is egyaránt egy NÉV, és valami trükkös módon holtbiztos hogy a memória egy pontját címkézi meg, ahova el kell ugornia, vagy ahol a megfelelő függvény kezdődik. Mindkét esetben annyi biztos, hogy azon a ponton amit e név, azaz a CÍMKE megjelöl, a programkód valamelyik része kezdődik (illetve folytatódik). Ez tökugyanaz mint a Basic esetében, csak ott minden címke egyszerűen egy decimális szám, és kivétel nélkül minden sornak van ilyen címkéje.

Na most én olyan fosszilis vén állat vagyok, hogy már akkor is programozgattam, amikor a C-64-es még „nagy” számítógépnek számított. És bár annak Basic interpretere alaphól nem tette lehetővé, hogy úgynevezett „számított ugróutasítást” használjunk, mert a goto és gosub után csak konstans számokat fogadott el, de azért „meg lehetett hackelni” ezt annak aki nagyon akarta... Én természetesen megtettem. S máris lehetett ilyesmiket írni a programjába, hogyszongya:

gosub 4000+60*A

És ez bizony nagyonis jó volt sok esetben... Mert vedd észre kérlek, ez épp olyasmi, mint amikor a C nyelvben egy függvényeket tartalmazó tömböt használunk, az abban levő valamelyik függvényt hívjuk meg! Ez roppantul kitágította akkoriban a C-64 programozási lehetőségeit.

Valami effélére biztos hogy módot kell nyújtanunk a mi mau nyelvünkben is, ha nem akarjuk, hogy csak „játéknyelv” maradjon, s a dedó-színvonalra értékeljék akik megismerik!

Márpedig ha a program egy adott helyén ahol címke szerepel, ott az érték változó is lehet nemcsak konstans, akkor az biza azt jelenti, hogy nálunk a címkék igenis változók! Természetesen nem azon a helyen változók, ahol megjelölik a program-kód valamely pozícióját, hanem ott, ahol hivatkozunk rájuk!

El kell tehát most döntenünk, hogy miként is jelöljük a címkéket, azon a helyen, ahol ő a címke jelöli meg a program egy pontját, s azt is el kell döntenünk, e címkére miként is hivatkozunk később, a program akárhány más pontjáról! Amint ebbe jobban belegondolunk, rájövünk, hogy tényleg nagyon hasonló problémákkal kell megküzdenuünk, mint a változók esetében. Például hogy honnan tudja az interpreterünk, hány címkénk lesz, ezek hány karakterből állhatnak, miféle karakterekből, amikor egy címkére hivatkozunk hogyan találja meg a hozzá tartozó memóriacímet, ha ezt táblázatból kell kikeresgélne az örültlassú lesz, stb.

Ha a probléma hasonló, logikus hogy a megoldás is hasonló lesz! Csinálunk majd valamiféle tömböt, ami annyi elemből áll, ahány címke maximálisan lehetséges a programunkban... E címkéknek lesz egy előtétkarakterük... Melyik is legyen ez?!

Valami olyasféle karaktert illenék választani, ami azért nem betű... Hagyjuk már meg a betűket a programutasításoknak, parancsoknak, kulcsszavaknak... Valami speciális karakter kéne, de a 127-nél kisebb kódú karakterekből (ASCII) oly szájalmasan kevés van!

Tényleg nem használhatunk ennél nagyobb kódú karaktereket?!

Úgy tűnik, nem. Az ami 128 vagy nagyobb kód, szinte akármit is jelenthet, attól függően, miféle kódtáblát használunk épp.

Na de hoppá! Mi egy új programnyelvet alkotunk! Miért is ne szerepelhetne e programnyelv SZABVÁNYÁBAN, hogy ez a programot tartalmazó fájl miféle kódolásban várja el?! Nyilván persze ha előírunk neki egy speciális kódolást, az megnehezíti a hordozhatóságot, meg persze nem okvetlenül lesz szerkeszthető a fájl minden text editorral. Ez azonban nem olyan szörnyű nagy negatívum mint elsőre hinnék, mert rengeteg mindenféle konvertáló program létezik egyrészt, másrészt nyugodtan választhatunk olyan kódolást, mely azért manapság a legtöbb helyen mégiscsak illik hogy elérhető legyen!

Ebbe belegondolva, bizony az egyetlen ami szóba jöhet, az az UTF-8 kódolás! Az egész számítástechnikai világ is abba az irányba halad, hogy ez legyen maga „A” szabvány! Az elég durva volna, ha az ISO-8859-2 lenne az előírás a részünkről,

ami a „magyar” kódlap. De abba senki nem köthet bele, ha a nemzetközi, s már igencsak el is terjedt UTF-8 szerint várja a forráskódot az interpreterünk!

Ezzel csupán egy baj van: egy karaktert több bájtal tudunk csak meghatározni! Na de ez még mindig jobb, mint belenyugodni az ASCII rém szegényes karakterkészletébe... És legalább mindig tudható, mikor kell megvizsgálni további karaktereket: ha az épp beolvasott karakter kódja nagyobb, mint 127! Azaz ha a legfelső bitje 1 !

Rögvest csináljunk is egy jó kis táblázatot ide magunknak, ami felsorolja egyrészt az összes nekünk, magyaroknak fontos ékezetes karakter UTF-8 bájtrepresztansait, másrészt néhány további jópofa karaktert, amik hasznosak lehetnek számunkra a programnyelvünkhöz!

UTF-8 kódok

Karakter	decimális bájtsorrend	hexa bájtsorrend	Nemzetközi elnevezés	Egyéb nevek
Á	195 129	c3 81	Aacute	
á	195 161	c3 a1	aacute	
Ä	195 132	c3 84	Adiaeresis	nagy umlaut
ä	195 164	c3 a4	adiaeresis	umlaut
É	195 137	c3 89	Eacute	
é	195 169	c3 a9	eacute	
Í	195 141	c3 8d	Iacute	
í	195 173	c3 ad	iacute	
Ó	195 147	c3 93	Oacute	
ó	195 179	c3 b3	oacute	
Ö	195 150	c3 96	Odiaeresis	
ö	195 182	c3 b6	odiaeresis	
Ő	197 144	c5 90	Odoubacute	
ő	197 145	c5 91	odoubacute	
Ú	195 154	c3 9a	Uacute	
ú	195 186	c3 ba	uacute	
Ü	195 156	c3 9c	Udiaeresis	
ü	195 188	c3 bc	udiaeresis	
Ű	197 176	c5 b0	Udoubacute	
ű	197 177	c5 b1	udoubacute	
– (nagykötőjel)	226 128 148	e2 80 94	emdash	(kvirtminusz)
(nem törhető szóköz)	194 160	c2 a0		
§	194 167	c2 a7	section	paragrafusjel
»	194 187	c2 bb	guillemotleft	

Karakter	decimális bájtrend	hexa bájtrend	Nemzetközi elnevezés	Egyéb nevek
«	194 171	c2 ab	guillemotright	
- (gondolatjel)	226 128 147	e2 80 93	endash	félkvirtminusz
„ (nyitó idézőjel)	226 128 158	e2 80 9e	doublelowquotemark	
” (záró idézőjel)	226 128 157	e2 80 9d	rightdoublequotemark	
ß	195 159	c3 9f	ssharp	sárfesz-esz
... (három pont)	226 128 166	e2 80 a6	ellipsis	
×	195 151	c3 97	multiply	
¢	194 162	c2 a2	cent	
±	194 177	c2 b1	plusminus	
÷	195 183	c3 b7	division	
£	194 163	c2 a3	sterling	font, pound
¤	194 164	c2 a4	currency	

(Igen, rendes gyerek voltam, felsoroltam a németeknek fontos *umlaut* és *sárfesz-esz* kódjait is...)

Csodálkoznék, ha a kedves Olvasóm nem értene velem egyet abban, hogy a címkék előtétkarakterének magától adódik a paragrafus-jel, tehát ez: **§**
Elvégre, ez a jel épp arról híres, hogy valamiféle új szövegegységet, gondolat-egységet, logikai egységet vezet be. És nyilván ilyesminek tekinthető minden olyan hely is, amit meg óhajtunk címkézni a programunkban!

Most akkor azt kell eldöntenünk, hány címkénk is legyen, s ezek mely karakterekből állhassanak.

Hány címke is kell egy programba? Hát izé... Ez erősen attól függ, mire is használjuk a címkéket! A C és C++ nyelvekben nem sok ilyesmi van, amennyiben ahhoz a nézethez ragaszkodunk - a „címke” fogalmának szigorú értelmezéseként - hogy a címke csak az, amire „ugrani” lehet. E nyelvekben ritka a „goto” használata. Él is amúgy egy olyan babona a programozók bizonyos köreiből, hogy tűzzel-vassal irtják a goto használatát, mert az úgymond áttekinthetlenné teszi a programot, és rontja a strukturáltságot. És igen, nem tagadható hogy e megfontolásban tényleg rejtezik némi igazság, de mint minden elvet, ezt se szabad túlzásba vinni. Ha van egy többszörösen egymásbaágyazott cikluscsoportunk, s ennek közepéből szeretnénk kiugrani, akkor bizony oda épp a jó öreg goto a jó eszköz, mert ha másképp akarjuk megoldani e gondot, akkor mindenféle kinnal szült nyakatekert megoldásokra kényszerülünk, például ideiglenes változók („flag”-ek) bevezetésére, amiket több helyen is ellenőrizni kell! Az meg az Álmoskönyv szerint lelassítja a programfutást, és szintén áttekinthetlenné teszi a kódot, csak másképp mint a goto...

Ellenben a Basic nyelvben rengeteg címke van: minden programsor fel van címkézve egy számmal... Az assembly nyelvekben meg szintén minden utasításnak van címkéje: a memóriacím, amin ő elhelyezkedik! Na jó, ez mondjuk nem az

„igazi” assembly hanem maga a „gépi kód”. De ha egy valódi assembly program forráskódját nézzük meg, abban is temérdek címkét találunk.

Általában elmondható, minél több minél magasabb szintű eszközt tartalmaz a programnyelv, annál kevesebb benne az efféle „ugrási célokat szolgáló” címkék használata. Ellenben annál több egyéb mindenfélét kell benne elnevezni: függvényeket, osztályokat, sablonokat, makrókat...

Még nem tudjuk, a nyelvünk hány és miféle „magasabb szintű eszközt” fog tartalmazni, de azt azért nyugodtan megtippelhetjük, hogy címkéből azért kellhet annyi mint változóból, sőt, több is! Mennyi legyen ez a „több”?

2 bájtön 65536 lehetőség ábrázolható. Ez mintha túl sok lenne. A 256 viszont nyilvánvalóan túl kevés. Miután az interpreterünk bájtonként halad előre a forráskód értelmezésében, ezért ha a 256 kevés, akkor 2 bájtot kell fenntartanunk a címke nevére, de az e bájtokon megengedett karakterek tartományát úgy kell megválasztanunk, hogy az ne legyen túl sok! Nos, az angol ABC kis- és nagybetűit mindenképp illik megengednünk címkenevnek. $52 \cdot 52 = 2704$, ez már érzésünk szerint elegendő lenne. De mert a címkék használata is gyakori lesz vélhetőleg, nem veszi ki jól magát, hogy minden címkebeolvasáskor bonyolult tesztelgetést kell végezni azért hogy eldöntsük, érvényes-e a címkenev, s hogy a karaktereket átszámítsuk valamiféle számértékké! Sajnos ugyanis a kis-és nagybetűs abc ASCII kódjai nem közvetlenül követik egymást... Egyezzünk meg inkább abban, hogy a hexa 41 és hexa 7F közt elhelyezkedő bármely karakter szerepelhet a címkenevben!

Ekkor minden karakter 63 lehetséges variációt tartalmazhat. $63 \cdot 63 = 3969$, ez elégnek tűnik. Miután szeretjük a kerek kilobájtokra kijövő értékeket, legyen ez inkább $64 \cdot 64 = 4096$ darab variáció, azaz ennyi lehetséges címkének foglalkunk majd le tömböt.

Természetesen így is garantálni kell, hogy a **\$** jel utáni két karakterből kiszámított index semmi esetre se legyen nagyobb, mint a tömb maximális értéke, azaz a 4096. Abban nem bízhatunk, hogy nem keveredik oda valami illegális karakter, mert a Felhasználó, az nagyon aljas. Akkor is ha nem felhasználó hanem programozó... (Itt az esetünkben a Felhasználó=programozó, mert aki felhasználja a programnyelvünket, az arra használja hogy programot ír vele...) Annak érdekében, hogy ne kelljen **if** utasításokkal szöszmögni (Az nem igazán gyors) tegyük azt, hogy levonunk a karakterkódból hexa 40-et, majd egyszerűen levágjuk a karakter legfelső bitjét (pontosabban azt kinullázzuk). De mert nem kizárt hogy a karakter már eleve kisebb kódú mint 0x40, ezért előbb ennek a legfelső bitjét 1-re állítjuk, majd levonjuk a 0x40 értéket belőle, s ezután nullázzuk a legfelső 2 bitet. Tehát egy tetszőleges „a” karakter esetén:

```
a |= 128; a -= 0x40; a &= 63;
```

Ezzel garantáltan biztosítva van, hogy a karakterből képzett kód soha nem lesz nagyobb, mint 63. És nem kellett semmiféle összehasonlító műveletet se végezni...

E módszernek persze hátránya, hogy az esetben is érvényes címkecímet képez, ha olyan karakter van megadva a címke nevében, ami amúgy nem „legális” az

esetünkben. Ekkor nem ad hibajelzést. Ennek esélye azonban csekély, ráadásul úgy vagyok ezzel, hogy a programot egyszer kell csak megírni, ellenben vélhetőleg utána nagyon sokszor fog futni. Nem lenne helyes az állandó ellenőrizgetésekkel lelassítani a kész program rengetegszer való futását azért, hogy a programozó munkáját megkönnyítsük azon időszakra amíg írja azt, ráadásul feltehetően feleslegesen, mert a címkék csak 2 karakterből állnak, s ilyen „rengeteg” gépelés során csak nem lesz olyan hülye, hogy mondjuk a „3” számjegyet írja le oda az „e” betű helyett...

A § jel két UTF-8 -as bájtjának értéke 194, 167. Természetesen kell külön utasítást is kreálnunk e 194-es bájt számára, s az azt követő 167-es kódra is. Legyen e 194-es kódú utasításcsoport neve **csoport_utf8_194_0xc2**. Ezt szépen be kell írunk a **fuggvenyek** tömb 194-es helyére, ezzel nincs baj. Azaz:

```
fuggveny csoport_utf8_194_0xc2; // A 194 kódú karakterrel kezdődő utasítások
.....
fuggveny *fuggvenyek[256] = {
.....
/* 194 */ csoport_utf8_194_0xc2,
```

A baj az, hogy ha a korábbi fejezetben eltervezett utasítás-azonosító metódusainkat használjuk, akkor ehhez szükségünk volna valami efféle tömbre is:

```
fuggveny *fuggveny_utf8_194_0xc2[95] = {
```

És ebbe kéne belevennünk azt a függvényt, ami a tulajdonképpeni „§” utasításnál szükséges teendőket hajtja végre nekünk, s e függvény címe e tömb 167-es pozíciójára kerülne, mert a § karakter második bájtjának ennyi a kódja! Na most ez rém kínos, mert e tömbünk viszont láthatóan csak 95 értéket tartalmazhat, ami némileg mintha kevesebb lenne a 167-nél...

Szerencsére azonban, tudjuk, hogy 194-es kódú bájtjal kizárólag azok a karakterek kezdődhetnek amik UTF-8 kódolásúak, s ezeknél szabály, hogy minden további bájtjuk nagyobb mint 128. Nekünk tehát csak arra kell majd ügyelni, hogy e második bájt (s esetleges további bájtok) legfelső bitjét lenullázzuk, s ezesetben semmi baj, amennyiben e tömböt mégse 95, hanem 128 eleműnek vesszük fel! Az elemszám-eltérés nem gond, mert attól a pointer típusa, amit a FUGGVENYTOMB tömbbe felvesszünk, még nem változik. A helyes deklaráció tehát így néz ki esetünkben:

```
fuggveny *fuggveny_utf8_194_0xc2[128] = {
```

Ez természetesen még „nem elég az üdvösséghez”, mert ugye nekünk a 2 karakterből álló utasításoknál a második karakter tesztelését és az az által jelölt utasítás végrehajtását a „**masodikkarakter**” nevű függvény végzi, aminek korábban meg is adtam a leírását. Ezt kell módosítanunk a megfelelőképpen! Íme az új változat, kékkel kiemelve a pótlólag beszúrt sorokat:

```
int masodikkarakter(F& f) { // Ezt a függvényt hívja meg minden olyan függvény, mely nem egykarakteres tokenű.
fuggveny **fuggtomb;

if(f.P>=f.phossz) {if(logflags[3]) {L("A program befejeződött!");} EXITSUCCESS();}
f.a2=f.p[f.P++];
fuggtomb=FUGGVENYTOMB[f.a];
if(fuggtomb==fuggvenyek) return 0; // Ezesetben valójában üres utasításról van szó, helykitöltőnek került csak bele,
// és semmit se kell csinálni
```

```

if(f.a>127) // Valami UTF-8 kódolású karakterről van szó, mert az első bájtja nagyobb mint 127
{return fuggtomb[f.a2 & 127](f); // végrehajtja az utasítást és visszatér annak visszatérési értékével.
// A második bájt legfelső bitjét törli, úgy szedi ki a tömbből a végrehajtandó függvény indexét!
} // if vége

if(f.a2<32) {if(wspc(f.a2)) {f.a2=SPACE;f.P--;} else return 0; // Nem kiiratható karakterek esetén semmit se csinál,
kivéve ha
// az a karakter whitespace, ez esetben ugyanazt csinálja, mintha space lenne.
// Vagyis a tömb legelső függvénye az legyen, amit akkor csinál, ha a parancstokent semmi más karakter nem követi,
// tehát ha egyedül áll.
}

if(f.a2>126) return 0; // Nem ASCII karakterek esetén semmit se csinál
return fuggtomb[f.a2-32](f); // végrehajtja az utasítást és visszatér annak visszatérési értékével.
}

```

Ezek után ajánlatos lesz táblázatot készíteni róla, a bennünket érdeklő UTF-8 kódolású karaktereknek mi a (decimális) bájtértéke a második és további bájtokra, a 128-al való csökkentés után. Íme az előző táblázat, pirossal kiemelve a megfelelő oszlopot:

Karakter	decimális bájtsorrend	a második és további bájtok értéke a 128-al csökkentés után	Nemzetközi elnevezés	Egyéb nevek
Á	195 129	1	Aacute	
á	195 161	33	aacute	
Ä	195 132	4	Adiaeresis	nagy umlaut
ä	195 164	36	adiaeresis	umlaut
É	195 137	9	Eacute	
é	195 169	41	eacute	
Í	195 141	13	Iacute	
í	195 173	45	iacute	
Ó	195 147	19	Oacute	
ó	195 179	51	oacute	
Ö	195 150	22	Odiaeresis	
ö	195 182	54	odiaeresis	
Ő	197 144	16	Odoubacute	
ő	197 145	17	odoubacute	
Ú	195 154	26	Uacute	
ú	195 186	58	uacute	
Ü	195 156	28	Udiaeresis	
ü	195 188	60	udiaeresis	
Ű	197 176	48	Udoubacute	
ű	197 177	49	udoubacute	
– (nagykötőjel)	226 128 148	0 20	emdash	(kvirtminusz)
(nem törhető szóköz)	194 160	32		
§	194 167	39	section	paragrafusjel
»	194 187	59	guillemotleft	
«	194 171	43	guillemotright	
– (gondolatjel)	226 128 147	0 19	endash	félkvirtminusz

Karakter	decimális bájtrend	a második és további bájtok értéke a 128-al csökkentés után	Nemzetközi elnevezés	Egyéb nevek
„ (nyitó idézőjel)	226 128 158	0 30	doublelowquotemark	
” (záró idézőjel)	226 128 157	0 29	rightdoublequotemark	
ß	195 159	31	ssharp	sárfesz-esz
… (három pont)	226 128 166	0 38	ellipsis	
×	195 151	23	multiply	
¢	194 162	34	cent	
±	194 177	49	plusminus	
÷	195 183	55	division	
£	194 163	35	sterling	font, pound
¤	194 164	36	currency	

És itt van a § utasítás is, ami tulajdonképpen semmit se csinál, csak logolja a tényt hogy átlépett egy címkét, amennyiben a megfelelő logolási szint engedélyezve van. Ez a logolási lehetőség a legkifejezettebben hasznos, ha debuggolni akarunk... Fontos tudni, hogy ezen utasítás abszolúte semmit se ellenőriz azt illetően, a címke karakterei érvényesek-e. Nem tölti ilyesmivel a drága időt. Sőt, feltételezi azt is, hogy a címke egészen bizonyosan áll legalább 1 karakterből. A második karakternek pedig elfogad bármit, a whitespace kivételével. Amennyiben az whitespace, akkor feltételezi hogy a címke 1 karakteres volt. Íme:

```
int paragrafus(F& f) {USC cimke1;USC cimke2;
if(f.P>f.phossz) return 0;
cimke1=f.p[f.P]; // A címke első karaktere
cimke2=k(f); // A címke második karaktere
if(logflags[1]) L("Címke: %c%c, cím: %lu", cimke1, cimke2, f.P-3); // Logol, ha kell
// Most a f.P mutató erre a cimke2 karakterre mutat, de lehet hogy ez nem érvényes címkekarakter hanem whitespace
if(wspc(cimke2)) return 0; // Ha a címke csak 1 karakterből áll, akkor épp azután állunk, s így
// folytatódhat a program végrehajtása, mert már átugrottuk a címkét
f.P++; // Ellenkező esetben megnöveljük a programutasítás-számlálót
return 0; // és minden kész is van.
}
```

Megjegyezném, hogy egy tisztességesen megírt programnak - akármi legyen is a feladata - nagyjából így kell kinéznie mint e fenti kis rutin: AZAZ TÖBB KELL LEGYEN BENNE A KOMMENT-SOR (A MEGJEGYZÉS) MINT A KÓDSOR!

Na de még mindig nem tudjuk, miként is HASZNÁLJA majd az interpreterünk a címkéket... Mert jó-jó, átugorni már tudja őket, de ennek semmiképp sincs így önmagában több értelme, mint egy üres utasításnak, megjegyzésféleségnek!

Nos, nyilvánvalóan szükségeltetik, hogy az interpreter még a program végrehajtásának elkezdése előtt tudja valahonnan minden létező címke helyét, azaz azt a pointert, ahol az elhelyezkedik a programban! Ennek érdekében egyszer végig kell „nyálaznia” a teljes forráskódot. Na de mi nem szeretnénk erre plusz időt áldozni (bár igazából ez tulajdonképpen nem tartana valami soká, de mi olyan smucigok vagyunk hogy azt is meg óhajtjuk spórolni...) s ezért eszünkbe jut, hogy hiszen ezt már elvégezhetjük azon idő alatt is, amíg beolvassa a memóriába a forráskódot!

Ezzel nincs is semmi baj, remek ötlet, ám van egy kis bökkenő magának a kódnak az átvizsgálásával. Mert ugye, ha a címkéinket később használni akarjuk mondjuk valami ugróutasításban, azt kb 2 féleképp használhatjuk:

1. goto ab
2. goto \$ab

Na most a 2. variáció esetén, amikor az interpreter átnyalazza a kódot, keresve a címkéket jelző \$ karaktert, honnan is tudhatná, hogy az ottani \$ jel nem azt jelzi hogy ott „kezdődik” egy címke, azaz hogy épp azt a helyet jelöli az „ab” címke, hanem itt csak utal arra a helyre, amit másutt jelöl meg?! Észben kéne tartania, hogy van e \$ jel előtt valami ugróutasítás vagy más akármi... Rengeteg mindenféle lehet pedig a \$ jel előtt, ez rémségesen megbonyolítaná a címkék kigyűjtését!

Az 1. variáció esetén pedig, honnan tudná a VÉGREHAJTÁS során, hogy a goto után okvetlenül egy címke 2 karaktere áll, s nem valami más, mondjuk egy változó, amibe nagy aljasul elrejtettük valahogyan azt a címkét, amire épp ugrani akarunk? Mondhatnánk persze, onnan tudja hogy változó esetén annak lenne előtétkaraktere:

goto @ab

Igenám, de még rengeteg mindenféle disznóságot belevehetünk később a programunkba, nem látunk előre a jövőbe, fene se tudja mi minden ötletszaltóink lesznek, hogy miféle bámulatos kacifántosságokat engednénk meg a címkék felhasználására... Legtisztább, ha úgy döntünk, a címkét az ő minden felhasználásánál igenis jelölje a \$ jel! Na de eszerint mégis meg kéne könnyítenünk valahogy az interpreter dolgát, amikor legelőször kigyűjti a címkék helyét. Mit is tegyünk, mit is tegyünk... Megvan! Lustálkodunk egy cseppet... Megfürdünk, megiszunk egy teát...

Mire ezzel végzünk, kész is a jó ötlet: Felhasználjuk azt, hogy szöveges állomány tartalmazza a forráskódot, egy efféle állományra pedig jellemző, hogy SOROKBÓL áll, s mindnek a végét jelzi az „újsor” karakter!

Ha már ott van, használjuk is fel, ne csúfoskodjék ott feleslegesen... Azaz, ugyebár egy sorVÉG után következik a sor ELEJE! Tehát egyezzünk meg abban, hogy címke márpedig kizárólag egy sor elején állhat! Illetve, állhat nyugodtan akárhol máshol is, de minek, mert ha másutt van, az interpreterünk csak átugorja... Amikor azonban beolvassa a programot, figyeli, elérkezett-e egy sorvég-karakterhez. Ha igen, s a következő karakter egy \$ jel (ami eképp ugyebár egy sornak pontosan a legislegelején áll) azesetben tudja, hogy itt egy címke kezdődik. Ekkor kiszámolja ennek karaktereiből a címke sorszámát, azaz a helyét a címke-tömbben, s eltárolja oda, melyik programmemóriacím is tartozik hozzá!

Az új állománybeolvasó rutinunk ennek megfelelően a következőképp néz ki:

```
void PGM::beolvas(char *filename) { // beolvassa a fájlt a memóriaterületre
    USC sorvegflag; // Ha 0: nem sorvég után vagyunk. Ha 1: sorvég után vagyunk. Ha 194: a $ jel első bájta után vagyunk.
    // Ha 167: a $ jel második bájta után vagyunk. Ha 11: A címke első karaktere után vagyunk
    USC cinkeikarakter;USC kar;sorvegflag=1; // Az elején 1-re állítjuk hogy a program legelső sorában is lehessen címkét
    megadni.
```

```

f.programfileeneve=filename;f.programfilenehossz=strlen(f.programfileeneve);
f.cimkedb=0; // Lenullázzuk a címkeszámológát
f.programfilehossz=get_file_size(f.programfileeneve);
if(f.phossz<f.programfilehossz) {L("A lefoglalt memóriaterület (%lu byte) kisebb mint a beolvasandó programfile mérete "
"(Fájlnév: %, = %lu byte)!",f.phossz,filename,f.programfilehossz);EXITFAILURE();}
FILE *fp=fopen(filename,"rb");
if(!fp) {L("A megadott \"%s\" állomány nem megnyitható (nem tudom beolvasni)",filename);EXITFAILURE();}
register USIL i;for(i=0L;i<f.programfilehossz;i++) {kar=f.p[i]=(USC)fgetc(fp);
if(kar==SORVEG) {sorvegflag=1;continue;}
if(kar==194) {if(sorvegflag==1) sorvegflag=194; else sorvegflag=0;continue;} // § jel első bájta!
if(kar==167) // § jel második bájta!
    {if(sorvegflag==194) sorvegflag=167; // Most már biztos, hogy címkét olvasunk be.
     else sorvegflag=0; // if 194 vége
    continue;
} // if 167 vége
// Ide akkor jutunk ha valami „közönséges” karaktert olvastunk be, ami lehet egy címke karaktere is.
if(sorvegflag==167) {cimke1karakter=kar;cimke1karakter |= 128; cimke1karakter -= 0x40; cimke1karakter &= 63;
    sorvegflag=11;continue;} // Eltároltuk a címke első karakterét
if(sorvegflag==11) { sorvegflag=0; // Egy karakter már biztos megvan a címkéből
if(wspc(kar)) { // A címke csak 1 karakteres mert a második karakter whitespace
if(f.cimke[cimke1karakter]) {L("Többször használod ugyanazt az 1 karakteres címkét! Pozíció: %lu
",i);fclose(fp);EXITFAILURE();}
f.cimke[cimke1karakter]=i+1;f.cimkedb++;sorvegflag=0;continue;} // if wspc vége
// Nem whitespace a második karakter
kar |= 128; kar -= 0x40; kar &= 63;
if(f.cimke[((unsigned int)cimke1karakter) * 64 + (unsigned int)kar])
{L("Többször használod ugyanazt a 2 karakteres címkét! Pozíció: %lu ",i);fclose(fp);EXITFAILURE();}
f.cimke[((unsigned int)cimke1karakter) * 64 + (unsigned int)kar]=i+1;f.cimkedb++;sorvegflag=0;continue;
} // if 11 vége
sorvegflag=0;
} // for i vége
fclose(fp);
}

```

Igen, kicsit hosszúnak tűnik, elismerem, de még mindig kifér egy képernyőre, legalábbis nálam mert szép széles képernyőm van, és ebben is eléggé sok a komment, ezt ne feledjük el!

Persze, ehhez a konstruktort is módosítanunk kell: ki kell egészíteni azzal a résszel, ami a címketömböt lenullázza. Ezt kell belevenni a legvégére:

```

f.cimkedb=0;for(i=0;i<4096;i++) f.cimke[i]=0; // címkék lenullázása

```

Na most, meg kell jegyezni, amiatt használhatjuk a 0 számot annak jelzésére, hogy az adott címke nem tartalmaz érvényes ugrási címet, mert gondoljunk csak bele: Semmiféleképpen sem tudunk a nulladik bájtra ugrani a forráskódunkban, mert még ha a legislegelejen is van a címke a programnak, akkor is ott maga a § jel kell álljon, ez eleve 2 bájta, a nulladik és első bájta a programunknak, s utána van még minimum 1 karakter, ami a címke maga! Az ezutáni pozíció a legislegkisebb amit eltárol a rutinunk egy címkéhez, azaz a 3-as szám.

Természetesen az F struktúrát is ki kell egészítenünk:

```

unsigned int cimke[4096]; // A címkék tömbje
unsigned short int cimkedb; // A programunkban ténylegesen felhasznált címkék száma

```

A **cimkedb** változó az F struktúrában gyakorlatilag csak statisztikai célokat szolgál, számolja hogy a programunkban ténylegesen hány címkét használtunk fel, de érdemes lehet megtartani, mert nem foglal el sok helyet - 2 bájtot mind-össze - és különben is csak a fájl beolvasásakor foglalkozik vele a program, ezért aztán a futási sebességet sem csökkenti.

Na most ezek után tegyük fel, hogy van már nekünk készen egy függvényünk, ami így kezdődik:

```

USIL ertekeUSIL(F& f) { // A következő „Aritmetikai kifejezés” értékét adja vissza USIL számként.
// A P mutató mutat a kifejezés legelső karakterére.

```

Most e pillanatban még ne törődjünk vele, konkrétan mit is értünk „aritmetikai kifejezés” alatt, a lényeg az hogy abban tutira biztosan szerepel majd olyasmi, hogy „változó”, meg „konstans”, meg esetleg még más izémizék is - miért is ne szerepelhetne benne olyan is, hogy „címké”?! A címké, az természetesen egy aritmetikai kifejezésben pontosan azt a számot jelenti, ami a programkódnak azon helye, melyet ő megjelöl, ahova például ugorni kell egy ugróutasítás esetén!

Ezen fellelkesülve, azonnal meg is írhatjuk a mi „goto” utasításunkat. Persze ennek se az lesz a neve igazából, hogy „goto”, mert az több mint 2 karakter. Lehetne esetleg csak annyi, hogy „go”, de ha már úgyis megoldottuk a 2 bájtos UTF-8 karakterek kezelését, válasszunk e célra inkább a » karaktert, ami nem két egymás utáni >> jel, hanem a korábban közreadott táblázatban szereplő azon jel, aminek a kódbájttjai a 194, 187 értékekkel bírnak. Ez a jel annyira szépen mutat...

S ezesetben a mi „goto” utasításunk csak ennyiből fog állni:

```
int duplajobbranyil(F& f) { // A » utasítás, ugróutasítás
unsigned int cinke;cinke=ertekUSIL(f);
if(cinke==0) {L("Nem létező címkére megkísérelt ugrás! cím: %lu",f.P);return -1;}
f.P=cinke;
return 0;
}
```

Ezek után a mau programunkban ezt a következőképp használhatjuk:

```
»$ab // ugrás az ab címkére
»100 // ugrás a századik bájtra
»$15 // Ugrás a 21-es bájtra (mert a $15 hexa szám értéke a decimális 21)
```

Sőt még sok másféleképp is használhatjuk majd nyilván. A lényeg az, hogy a » jel után abszolúte bármi állhat, ami egy egész számnak megfeleltethető - méghozzá nem egy konstansnak, mert ugye ezután állhat változó is, megcsináljuk majd:

```
»@17 // A 17-es változónk által tartalmazott számértékre ugrás
```

Úgy néz ki tehát, a címkék kérdését megoldottuk. A változókkal kapcsolatban azonban még hátra van egy jó vaskos adósságunk! Mert ugye, bizonyos adattípusokból nálunk több is benne foglaltatik egy változóban, és kéne valami módszer, amivel megmondjuk hogy épp melyikre hivatkozunk ezek közül...

Nos, ezen célra szolgál a korábban már említett néhány azon tömb az F struktúrában, amiknek a neve úgy kezdődik, hogy **vindex**. Ezen tömbök mindegyikének is épp 256 eleme van, s mindegyik elem azt a számot tartalmazza, amely a megfelelő sorszámú változó aktuális indexe. Mint látható, mindegyik adattípushoz külön tömb tartozik. Amikor kivesszük belőlük az indexet, nem ellenőrizzük hogy ezen index nem lépi-e túl a lehetséges maximális nagyságot, ez csak időfecsérlés lenne, ezen index nagyságának érvényességét úgy garantáljuk inkább, hogy akkor ellenőrizzük le, amikor ezen tömbbe BErakjuk az indexet. Kezdetben mindegyik index értéke 0.

Ezenfelül felveszünk két speciális változót is az F struktúrába, a B és J nevűt:

```
USC B; // A "balérték" változó indexe. Kezdetben garantáltan 0.
USC J; // A "jobbérték" változó indexe. Kezdetben garantáltan 1.
```

Ezekkel magát a változótömböt indexeljük majd, a B-edik változó lesz az, amit egy értékadó utasítás bal oldalára képzelünk el, a J -edik változó lesz ellenben a jobb oldalon. Más szavakkal: A B-edik változó értéke fog megváltozni, s a J-edik változóból olvassuk ki az adatot.

Természetesen ahhoz hogy ez működjön, jó sok mindenféle utasítást kell majd gyártanunk... Az értékadó utasítások azonban egy későbbi fejezet témáját képezik. Most egyelőre oldjuk meg azt, hogy egyáltalán ki tudjunk iratni némely adatokat, még ha nem is minden típusú adatot! Meg szöveget is!

Na, ez hosszú rész lesz, sok programmal...

Kezdjük azzal, hogy bemutatom azt a rutint, ami egyelőre a legáltalánosabb formában beolvassa a következő sok mindenfélét a programból, aminek a végeredménye egy USIL szám lesz. Ezt fogja meghívni gyakorlatilag minden olyan utasítás, ami valami egész értéket igényel:

```
USIL ertekUSIL(F& f) { // A következő „Aritmetikai kifejezés” értékét adja vissza USIL számként.
// A P mutató mutat a kifejezés legelső karakterére.
USC karakter;USC cimkeikarakter;USC kar;
nemspace(f); // előlépteti a pointert a következő nem whitespace karakterig

karakter=f.p[f.P]; // Az aktuális feldolgozandó karakter
if(karakter=='@') { return vl(f);} // Változó értékét kell beolvasni
if(karakter=='?') { return systemvariable(f); } // Rendszerváltozó tartalmát kell visszaadni
if(karakter==194) { // $ karakter első bájttja
karakter=k(f);if(karakter!=167) return 0;
cimkeikarakter=k(f);kar=k(f);
cimkeikarakter |= 128; cimkeikarakter -= 0x40; cimkeikarakter &= 63;
if(wspc(kar)) { // A címke csak 1 karakteres mert a második karakter whitespace
return f.cimke[cimkeikarakter];}
// A címke 2 karakteres
f.P++; // Azért, hogy a P mutató a következő karakterre mutasson
kar |= 128; kar -= 0x40; kar &= 63;
return f.cimke[(unsigned int)cimkeikarakter * 64 + (unsigned int)kar];
}
// Különbben itt valami számféleségnek kell állnia
return number(f.p,f.P,f.phossz);
}
```

Látható, e rutin bizony számos más rutinra hivatkozik! Például a „number” nevűre, ami a számok (konstansok) beolvasását végzi:

```
USIL number(USC *p, USIL& P, USIL PMAX) { // beolvassa a számot a stringből, ami a p pointer P-edik karakterénél
kezdődik.
// A p[P] -edik karaktertől függ, a konverzió milyen számrendszerben hajtódik végre.
// Ha ez 0-9 : decimális szám. Ha % : bináris szám. Ha $ : hexadecimális szám. Ha o vagy 0 : oktális szám
// 'x' : Az x karakter ASCII kódja
// A visszatéréskor a P mindig az első olyan karakterre fog mutatni, ami már nem tartozik a megfelelő számrendszer
// tartományába.

USIL num=0L;USC c;

c=p[P];

if((c>='0')&&(c<='9')) goto dec;
if(c=='$') goto hex;
if(c=='%') goto bin;
if(c=='\') goto kar;
if(c=='o') goto oct;
if(c=='0') goto oct;
return num; // Ha érvénytelen a karakter, nulla a visszatérési érték.

kar:
P++;if(P>=PMAX) return 0L;
num=(USIL)p[P];P++;
return num;

dec:
while(P<PMAX) {
num=num*10+(c-'0');
}
```

```

c=p[++P];
if((c<'0')||(c>'9')) break;
} // dec while vége
return num;

bin: c=p[++P];
if((c<'0')||(c>'1')) return num;
while(P<PMAX) {
num = (num << 1) + (c-'0');
c=p[++P];
if((c<'0')||(c>'1')) break;
} // bin while vége
return num;

oct: c=p[++P];
if((c<'0')||(c>'7')) return num;
while(P<PMAX) {
num = (num << 3) + (c-'0');
c=p[++P];
if((c<'0')||(c>'7')) break;
} // oct while vége
return num;

hex: c=p[++P];
if((c>='0')&&(c<='9')) goto okay;
if((c>='a')&&(c<='f')) goto okay;
if((c>='A')&&(c<='F')) goto okay;
return num;
okay:
while(P<PMAX) {
if((c>='0')&&(c<='9')) num = (num << 4) + (c-'0');      else {
if((c>='a')&&(c<='f')) num = (num << 4) + (c-'a')+10;    else if((c>='A')&&(c<='F')) num = (num << 4) + (c-'A')+10;
}

c=p[++P];

if((c>='0')&&(c<='9')) continue;
if((c>='a')&&(c<='f')) continue;
if((c>='A')&&(c<='F')) continue;
break;
} // hex while vége
return num;
}

```

Elismerem, ez nálam se fér már rá egy képernyőre, de nem lóg le róla sok. És mindenképp könnyen áttekinthető, mert jól áttekinthető apró részekre bomlik, attól függően, milyen számrendszerű számokat olvasunk be, vagy esetleg valami karaktert. Különben meg simán ráférhetne 1 képernyőre, ha több utasítást is írnék egy sorba, mert most majdnem mindegyikben csak 1 van... Magyarázni a működését felesleges, a kódból minden kiderül, teljesen világos az egész.

Ha majd szöveget (stringet) akarunk kiírni, előre tudható, hogy lesznek benne „escape szekvenciák”, s akkor jól jön egy rutin, ami a \ jel után következő karaktert leteszteli, s visszaadja ahelyett a ténylegesen kiírandó karaktert:

```

USC specchars(USC c) {USC k;k=c;
switch (c) {
case 'v': k=SORVEG; break; // sorvég karakter (újsor), ez Linux alatt az ASCII 10 karakter
case 'n': k=10; break; // NL (LF) karakter (újsor)
case '0': k=0;break; // ASCII 0 karakter
case 't': k=9;break; // Tabulátor karakter
case 'b': k=8;break; // Backspace karakter
case 'r': k=13;break; // CR karakter
case 'f': k=12;break; // FF karakter
case 'a': k=7;break; // Csengő karakter (alert)
} // switch vége
return k;
}

```

Az **ertekUSIL** függvény által hívott függvények, s azok amiket ezek hívnak:

```

// =====
USC vc(F& f) {/// A programkódban soron következő változó aktuális értékét adja vissza USC típusként.
// A P mutató okvetlenül egy @ jelre kell mutasson!
USC index=INDEX(f);
return f.v[index].c[f.vindexc[index]];
}

```



```

// -----
USIL vl(F& f) { // A programkódban soron következő változó aktuális értékét adja vissza USIL típusként.
// A P mutató okvetlenül egy @ jelre kell mutasson!
USC index=INDEX(f);
return f.v[index].l[f.vindexl[index]];
}
// -----
USC INDEX(F& f) { // A programkódban soron következő változó indexének értékét adja vissza
// A P mutató okvetlenül egy @ jelre kell mutasson!
USC kovkar;USC index;
if(f.p[f.P]!='@') {L("\@" jel nélküli változóra hivatkozás! Pozíció: %lu",f.P);EXITFAILURE();}
kovkar=k(f);
// Itt kell meghatározni a változó indexét
if(kovkar=='@') { // indirekció, másik változóra hivatkozás!
index=vc(f);goto INDEXokay;
}
// Ha a karakter valamely betű, az index ezen betű ASCII kódja.
if(((kovkar>='a')&&(kovkar<='z'))||((kovkar>='A')&&(kovkar<='Z'))){index=kovkar;goto INDEXokay;}
// Különbön itt valami számféleségnek kell lenni („aritmetikai kifejezésnek”)
index=(USC)ertekUSIL(f);
INDEXokay:
return index;
}
// =====
USIL systemvariable(F& f) { // Valamely fontos mau rendszerváltozó értékét adja vissza
USC k1;USIL value;value=0;

k1=k(f); // A következő karakter beolvasása
if(wspc(k1)) return value; // whitespace esetén a visszatérési érték nulla
f.P++; // Ugrás a következő karakterre
switch(k1) {
case 'p': value=f.phossz;break; // A programmemória hossza
case 'P': value=f.P;break; // Az aktuális cím a végrehajtandó programban
case '1': value=(USIL)f.a; break; // Az épp végrehajtott utasítás kódjának első karaktere
case '2': value=(USIL)f.a2; break; // Az épp végrehajtott utasítás kódjának második karaktere
case 'm': value=f.mhossz;break; // Az adatmemória hossza
case 'B': value=f.B;break; // A B változó, a „balérték-index” értéke
case 'J': value=f.J;break; // A J változó, a „jobbérték-index” értéke
// Változók belső indexeinek visszaadása:
case 'c': value=(USIL)(f.vindexc[(USC)ertekUSIL(f)]);break;
case 'C': value=(USIL)(f.vindexC[(USC)ertekUSIL(f)]);break;
case 'i': value=(USIL)(f.vindexi[(USC)ertekUSIL(f)]);break;
case 'I': value=(USIL)(f.vindexI[(USC)ertekUSIL(f)]);break;
case 'l': value=(USIL)(f.vindexl[(USC)ertekUSIL(f)]);break;
case 'L': value=(USIL)(f.vindexL[(USC)ertekUSIL(f)]);break;
case 'f': value=(USIL)(f.vindexf[(USC)ertekUSIL(f)]);break;
case 'g': value=(USIL)(f.vindexg[(USC)ertekUSIL(f)]);break;
case 'G': value=(USIL)(f.vindexG[(USC)ertekUSIL(f)]);break;
case 'd': value=(USIL)(f.vindexd[(USC)ertekUSIL(f)]);break;

case 'h': value=(USIL)f.programfilenevehossz;break; // A programfile nevének a hossza
case 'H': value= f.programfilehossz;break; // A programfile hossza
case 'b': value=(USIL)f.BRAINFUCKFlag;break; // A BRAINFUCK flag értéke
case 'E': value=(USIL)f.cimkdb;break; // A programban ténylegesen használt cínek darabszáma
case 'a': value=(USIL)returnargc();break; // A mau programnak megadott parancssori paraméterek száma
case 'V': value=(USIL) SORVEG;break; // Az a karakter, amit a sor végének jelzésére használ a rendszer (Linux alatt a 10-es kódú)
case 'M': value=MauInterpreterVersionNumber;break; // A mau interpreter verziószáma
case 'X': value=2;if(stdlog==stdout) return 0;if(stdlog==stderr) return 1;break;
case 'x': value=(USIL)logflags[(USC)ertekUSIL(f)];break; // A megfelelő sorszámu logflag értékét adja vissza

//default:
}

return value;
}

```

Végül pedig a kiíratási rutinok:

```

int textkiir(F& f) {USC c;
c=f.p[f.P]; // Az aktuális, idézőjel utáni karakter
while(c!=34) { // A következő idézőjelig mindent kiir

switch(c) {
case 92: // backslash karakter
printf("%c",specchars(k(f)));break;
default: printf("%c",c);break;
} // switch vége

c=k(f); // A következő karakter
} // while vége
f.P++;
return 0;
}
// =====

```

```

int csoport_kerdojel_63(F& f) {return masodikkarakter(f);}
// -----
int kerdojel(F& f) {USC kar;
namespace(f);
kar=f.p[f.P];
if(kar==34) {k(f);return textkiir(f);} // szövegkiírás
printf("%lu", (unsigned long int)ertekUSIL(f));
return 0;
}
// -----
int kerdojelpont(F& f) {
printf("%c", (USC)ertekUSIL(f));
return 0;}
// =====

```

A textkiir rutint természetesen az idézőjel karakterhez illesztjük be:

```

fuggveny *fuggvények[256] = {
/*      0 */ semmi,
/*      1 */ semmi,
.....
/*     31 */ semmi,
/*     32 SPACE */ semmi,
/*     33 ! */ semmi,
/*     34 idézőjel */ textkiir,
/*     35 # */ semmi,
.....

```

Most már tudunk adatmemóriát is lefoglalni:

```

int mem_lefoglal(F& f) {namespace(f);adatmemlefooglal(f,ertekUSIL(f));return 0;}
// -----
.....
void adatmemlefooglal(F& f,USIL hossz) {
if(f.m!=NULL) {L("Az adatmemória újbóli lefoglalására történt kísérlet, annak előzetes felszabadítása
nélkül!");EXITFAILURE();}
f.mhossz=hossz;f.m = new USC[f.mhossz];if(!f.m) {L("Nincs elég memória!");EXITFAILURE();}
USIL i;for(i=0L;i<hossz;i++) f.m[i]=0;
}

```

A **mem_lefoglal** parancsunkat természetesen az „M” karakterre kötjük rá:

```

fuggveny *fuggvények[256] = {
.....
/*     77 M */ mem_lefoglal,

```

Ha valakit az izgatna a fenti rutinok átnézése után, mi az a titokzatos „BRAINFUCK-flag”, ígérem hogy azt is megtudja a következő fejezetből, de annyit már most előre bocsátok, hogy ez nem holmi káromkodás a részemről.

Ezek után viszont nagyonis itt az ideje, hogy írjunk már végre egy pici kis mau nyelvű programot, ami ugyan még semmi hasznosat nem csinál, de letesztel nekünk pár funkciót:

```

M 30000
"Ezt a szöveget iratom ki!"
/ ? "Ezt is!" /
$AA
? "A programmemória mérete: " ? ?p /
"A programmutató aktuális értéke: " ? ?P /
? "Első karakter: " ? ?1 /
? "Második karakter: " ? ?2 /
? "Adatmemória hossza: " ? ?m /
? "Programnévhossz: " ? ?h /
? "Programhossz: " ? ?H /
? "Címkek száma: " ? ?E /

```

```

»$id // Elugrik az „id” címkehez
"Ezt nem szabad kiírnia, mert át kell ugrania!"
$id // Ide ugrik
"Ez \"HÜLYE\\SÉG\"!" /
? "stdlog: " ? ?X /
? "argc: " ? ?a /
? $AA /
?. $42 /
?.$AA /

?. 'j /
?. 65 /
?.$44 /

/

```

A program futtatásának eredménye valami ilyesmi kell legyen:

```

Ezt a szöveget iratom ki!
Ezt is!
LOG:> 2014.02.07 17:56:49 : Címke: $AA, cím: 62
A programmemória mérete: 562
A programmutató aktuális értéke: 151
Első karakter: 63
Második karakter: 32
Adatmemória hossza: 30000
Programnévhossz: 5
Programhossz: 562
Címkek száma: 2
Ez "HÜLYE\SÉG"!
stdlog: 0
argc: 2
66
B
B
j
A
D

```

A LOG:> kezdetű sor csak akkor látszik, ha a megfelelő logolási szint be van kapcsolva. A „66” szám helyett lehet más is, attól függően, hogy az \$AA címke pontosan hányadik sorba van elhelyezve, ugyanis e címke értékét írja itt ki. Ugyanez igaz az ez alatti második B betűre, mert itt szintén ezen címke értékét írja ki, de KARAKTERBE ÁTKONVERTÁLVA... Ennek természetesen tényleg a világon semmi értelmes haszna nincs bármi „józan esetben”, egyedül tesztelési célra jó most nekünk.

Amint látjuk, egy sorba egymás után több utasítást is beírhatunk, és nem muszáj közéjük pontosvesszőket tennünk. Viszont minden értéket amit ki akarunk iratni, külön „?” karakterrel kell jelölnünk, azaz ezzel megparancsolni a kiírását, kivéve az idézőjelek közé zárt szöveget, aminél nem muszáj kérdőjel az elejére, mert az idézőjel maga is szövegkiíró utasításként működik.

Na ez jó nagy meló volt drága hölgyeim és uraim, a többit hagyjuk a következő fejezetre...

6. fejezet: A programnyelvünk Turing-teljessé tétele

Eljutottunk hát oda szépséges és szívünknek oly kedves mau nyelvünk fejlesztésében, hogy tudunk kiírni vele mindenféle konstans szövegeket, meg konstans számokat is különböző számrendszerekben, e számoknak megfelelő karaktereket, tudunk pontosvesszőt használni, s többféleképp is megjegyzéseket

írni a programba, valamint vannak már címkéink is, amiknek lekérdezhetjük (kiirathatjuk) az értékét, sőt ugorhatunk is rájuk, ami kétségtelenül a címkék leghagyományosabb felhasználási területe!

Most e fejezetben eljutunk odáig, hogy programnyelvünk alkalmas legyen kivétel nélkül minden olyan feladatra, amire általában véve alkalmas minden (azaz bármelyik) jelenleg létező programnyelv, úgy a legegyszerűbbek mint a legislegmagasabb szintűek! Ez sokkal egyszerűbb lesz, mint azt bárki is hinné...

Amit ugyanis a fenti bekezdés jelent, hogy a programnyelv alkalmas bármi olyan feladatra amit manapság egy programnyelvtől elvárnak, azt szaknyelven azzal a fogalommal fejezik ki, hogy a programnyelv úgynevezett „Turing-teljes”. Ez a „Turing-teljes” fogalom természetesen megmagyarázható lenne ennél sokkal tudományosabban is, de ezt mellőzöm, mert már a Bevezetőben megígérttem, hogy kerülni fogom a felesleges „felHOMÁLYosításokat”... Lényeg az, hogy ha valamely programnyelvre a nagyokosok azt mondják hogy ez „Turing-teljes”, akkor ezzel elismerték, hogy az a nyelv alkalmas mindarra, amire bármelyik eddig létrejött programnyelv, bár természetesen elképzelhető, hogy valami feladatot abban a nyelvben sokkal nehezebben lehet leprogramozni, mint egy másik nyelvben. Azaz, bármire is legyen képes egy tetszőleges mai programnyelv, azt bármelyik másik programnyelv is képes megcsinálni, amennyiben az egy Turing-teljes programnyelv.

Természetesen ha a mi programnyelvünk azzal az igénnyel lép fel, hogy legalább valamely csekélyke kis mértékben komolyan vegyék, akkor kutya kötelességünk gondoskodni arról, hogy Turing-teljes legyen!

Nem hinném hogy tévedek, amikor úgy vélem, Olvasóm nem rendelkezik olyan komoly matematikai ismeretekkel, melyek által élvezne egy Turing-teljességi matematikai bizonyítást, vagy akárcsak annak kísérletét... Szerencsére azonban nincs is szükségünk ennek lefolytatására! A Turing-teljességnek ugyanis van egy számunkra módfelett előnyös szabálya, ami így hangzik:

Minden olyan programnyelv Turing-teljes, ami részhalmazként tartalmaz valamely másik Turing-teljes programnyelvet!

Ez különben logikus. Gondoljunk csak bele: Egy programnyelv lényegében utasítások egy halmaza. Ha van mondjuk 30 utasításunk, s azzal meg tudunk oldani bármely problémát, akkor ha ezen 30 utasításhoz hozzáveszünk még újabb mondjuk 47-et (vagy akármennyit), nyilvánvaló hogy akkor is meg tudjuk oldani az előző problémákat! Tudniillik ha olyan gondba ütközünk amik nem megoldhatóak az újabb utasításainkkal, akkor megoldjuk azokat legfeljebb a régi 30 utasításunk közül néhányal...

Nekünk tehát most nincs más dolgunk, mint hogy megvalósítsuk programnyelvünkben valamelyik olyan már létező másik programnyelv utasításait, melyről már korábban elismerték a „nagyokosok”, hogy az Turing-teljes. Ha ez sikerül nekünk, akkor teljesen mindegy, ezen felül mi mindent építünk még be a nyelvünkbe: az egyszersmindenkorra Turing-teljesnek fog számítani, efelől vita se lehet!

Természetesen, minthogy mi azért mégiscsak a magunk nyelvét óhajtjuk fejleszteni, e munkát afféle kitérőnek kell tekintenünk, s ezért hogy erőfeszítéseinket ezen irányba a minimálisra redukáljuk, a lehető legegyszerűbb Turing-teljes nyelvet valósítjuk meg. Ez pedig, amennyire megítélhetem, az úgynevezett „brainfuck” nyelv. Ténylegesen ez a neve, ami magyarul talán „agybaszónak” fordítandó, már elnézést a csúnya szóért... És nem véletlenül ez a neve, ugyanis irtózatosan nehéz benne programokat írni! Ennek ellenére azonban Turing-teljes. És bár e nyelven egy programot megírni tényleg nehéz, de a programnyelv interpreterének a megalkotása nagyonis könnyű... Épp emiatt választottam!

A nyelv mindössze csak 8 utasítást tartalmaz ugyanis... A nyelvet *Urban Müller* készítette, azzal a céllal, hogy olyan Turing-nyelvet hozzon létre, amire a lehető legkisebb fordítóprogramot tudja megírni. Mi azonban interpretert készítünk rá.

A brainfuck nyelvről itt található részletes Wikipédia-szócikk:
<https://hu.wikipedia.org/wiki/Brainfuck>

A *Brainfuck* nyelvnek egy univerzális byte-mutatója van, aminek a neve „pointer”, és ami szabadon mozoghat az adatmemóriában, mely legalább 30000 byte nagyságú tömb illik hogy legyen, és amelynek alapértékei nullák. A pointer a tömb elején indul. Nem tudom, miért van ez a fétisizmus a brainfuck programozók körében, hogy a memóriatömb értéke éppen 30000 bájt nagyságú legyen - még ha legalább 32768 lenne a mérete, megérteném, mert az „kerek” szám kettes számrendszerben írva. De 30000?!

Na mindegy.

A nyelv nyolc parancsát egy-egy karakter reprezentálja:

- > A pointer növelése eggyel
- < A pointer csökkentése eggyel
- + A pointernél levő byte növelése eggyel
- A pointernél levő byte csökkentése eggyel
- . A pointernél levő byte kiírása
- , Byte bekérése és a pointernél tárolása
- [Ugrás a következő, megfelelő] jel utánig, ha a pointer alatti byte nulla.
-] Ugrás az előző, megfelelő [jelig.

Ez igazán egyszerűnek tűnik, e 8 utasítás leprogramozása talán nem haladja meg szerény képességeinket...

A baj csak az, hogy e 8 utasítás 8 karaktere nagyonis olyannak tűnik, melyeket szívesen használnánk más parancsok számára is, nem engedhetjük meg, hogy elfogyasszuk őket efféle játszadozásra, mint a brainfuck programnyelv! De van megoldás! Vegyük fel az F struktúrába a brainfuck flaget:

USC BRAINFUCKflag; // Ha 1: A "+ - > < . , []" karaktereket brainfuck módban értelmezi, ha 0: másképp.

E flag kezdetben 0 értékű, erre állítja be a konstruktorunk. Kell tehát nekünk megfelelő utasítás e flag 1-re állításához (bekapcsolásához) és nullára állításához, azaz a kikapcsolására. Legyen ez a **/+** és a **/-** utasítás:

```
int perplusz(F& f) { // Brainfuck értelmezés bekapcsolása
f.BRAINFUCKflag=1;return 0;
}
// -----
int perminusz(F& f) { // Brainfuck értelmezés kikapcsolása
f.BRAINFUCKflag=0;return 0;
}
```

Ezek után meg kell alkotnunk a megfelelő utasításcsoportokat:

```
int csoport_plusz_43(F& f) {if(f.BRAINFUCKflag) return brainfuck_plusz(f);
return masodikkarakter(f);}
// -----
// =====

int csoport_vesszo_44(F& f) {if(f.BRAINFUCKflag) return brainfuck_vesszo(f);
return masodikkarakter(f);}
// -----
// =====

int csoport_minusz_45(F& f) {if(f.BRAINFUCKflag) return brainfuck_minusz(f);
return masodikkarakter(f);}
// -----
// =====

int csoport_pont_46(F& f) {if(f.BRAINFUCKflag) return brainfuck_pont(f);
return masodikkarakter(f);}
// -----
// =====

int csoport_kisebbjel_60(F& f) {if(f.BRAINFUCKflag) return brainfuck_kisebbjel(f);
return masodikkarakter(f);}
// -----
// =====

int csoport_nagyobbjel_62(F& f) {if(f.BRAINFUCKflag) return brainfuck_nagyobbjel(f);
return masodikkarakter(f);}
// -----
// =====

int csoport_nyitozarojel_91(F& f) {if(f.BRAINFUCKflag) return brainfuck_nyitozarojel(f);
return masodikkarakter(f);}
// -----
// =====

int csoport_csukozarojel_93(F& f) {if(f.BRAINFUCKflag) return brainfuck_csukozarojel(f);
return masodikkarakter(f);}
// -----
```

Mindegyik karakter számára természetesen meg kell csinálnunk a megfelelő függvénytömböt a mau.h fejlécállományunkban, amint azt az összes előző utasításunknál is megcsináltuk korábban, ezeket most nem másolom be ide, ez rutinból kell menjen már, most korlátozódjunk csak a lényegre! A teljes forráskód úgyis szerepel a könyv végén. (Vagy esetleg külön fájlban mellékelve, attól függ, miként jut el ez a kis leírás az Olvasóhoz).

A tulajdonképpeni **Brainfuck** utasítások:

```
// ===== BRAINFUCK utasítások =====
int brainfuck_kisebbjel(F& f) { // Brainfuck <
if(f.v[f.B].l[0]==0L) {L("Adatmemória alulcsordulás!");EXITFAILURE();}
f.v[f.B].l[0]--;return 0;}

int brainfuck_nagyobbjel(F& f) { // Brainfuck >
f.v[f.B].l[0]++;if(f.v[f.B].l[0]>=f.mhossz) {L("Adatmemória indextúlcsordulás!");EXITFAILURE();}
```

```

return 0;}

int brainfuck_plusz(F& f) { // Brainfuck +
if(f.m==NULL) {L("Nincs lefoglalva memória a BRAINFUCK interpreter számára!");EXITFAILURE();}
f.m[f.v[f.B].l[0]]++;return 0;}

int brainfuck_minusz(F& f) { // Brainfuck -
if(f.m==NULL) {L("Nincs lefoglalva memória a BRAINFUCK interpreter számára!");EXITFAILURE();}
f.m[f.v[f.B].l[0]]--;return 0;}

int brainfuck_pont(F& f) {printf("%c",f.m[f.v[f.B].l[0]]);return 0;} // Brainfuck .

int brainfuck_vesszo(F& f) {f.m[f.v[f.B].l[0]]=getchar();return 0;} // Brainfuck ,

int brainfuck_nyitozarojel(F& f) { // Brainfuck [
USIL darab=0L;USC c;
if(f.m[f.v[f.B].l[0]]==0) {
while(1) {
c=f.p[f.P++];
if(f.P>=f.phossz) return -2;
if(c=='[') darab++;
if(c==']') {if(darab==0L) {f.P++;
if(f.P>=f.phossz) {return -2;; return 0;}
darab--;
}
} // while vége
} // if ==0 vége
return 0;
} // funkció vége

int brainfuck_csukozarojel(F& f) { // Brainfuck ]
USIL darab=0L;USC c;
f.P--;if(f.P==0L) return -3;
f.P--; // Most az aktuális utasítást megelőző bájton van.
while(1) {
if(f.P==0L) return -3;
c=f.p[f.P--];if(f.P==0L) return -3;
if(c==']') darab++;
if(c=='[') {if(darab==0L) {f.P++;return 0;}
darab--;
}
} // while vége
return 0;
} // funkció vége

// =====

```

Készen is vagyunk az egésszel! Látható, hogy a pointernek azt a változót neveztük ki, amelyiknek az indexét a korábban emlegetett B nevű „balérték” jelzőnk tartalmazza. Ez kezdetben nulla kell legyen, erről a konstruktor gondoskodik. E B-edik változónk is ugyebár egy union, ennek „l” jelű mezője tartalmazza a pointert, mert a kellő nagyságú szám ebbe fér bele.

Ezek után ha valahol az Internet sötét virtuális sikátorában találunk valami cuki brainfuck programot, mi is lefuttathatjuk, csak egy icipici kis átalakítást kell végezni rajta. Konkréten, itt van például ez a program:

```
++++++[>++++++<-]>+++++.---.+++++..+++.
```

Ez annyit csinál, hogy kiírja, hogy
HELLO

Na most ezt a fenti formában találhatod meg az Interneten. Ahhoz hogy nálunk is működjön, ki kell egészíteni egy olyan utasítással, hogy lefoglalja neki a kellő nagyságú adatmemóriát, egy olyannal ami bekapcsolja a brainfuck-értelmezést, és... És a legvégére illik rakni egy üres sor kiírását megvalósító utasítást, mert különben a legutolsó karakter kiírása után ott jelenik meg a prompt, ami nem mutat szépen...

Azaz e program a mi esetünkben így kell kinézzen:

```
M 30000
/+
++++++[>++++++<-]>+++++.---.+++++.+++. /
```

Mint látható ezt már piros színnel írtam, mert ez egy teljesen érvényes, legális, korrekt program a mi „mau” programnyelvünkben is!

A „Hello World!” szöveget kiíró programot így találtam meg az Interneten:

```
+++++ +++++
[
  > +++++ ++
  > +++++ +++++
  > +++
  > +
  <<<< -
]
> ++ .           'H'
> + .           'e'
+++++ ++ .       'l'
.               'l'
+++ .           'o'
>+ .           'i'
<< +++++ +++++ +++++ . 'W'
> .             'o'
+++ .           'r'
----- - .      'l'
----- - - .    'd'
> + .           '!'
```

Elvileg ez is működik a mi interpreterünkkel, ha az elejére beszurjuk a

```
M 30000
/+
```

utasításokat, de nem javaslom. Amiatt működik csak, mert az összes, jobboldalt aposztrófok közt álló betű, meg persze az aposztrófok is jelenleg üres utasítások nálunk. Ki tudja azonban, később miféle funkciókat bütykölünk rájuk... Helyesebb ezeket kitörölni az eredeti brainfuck forráskódból, s ekkor ez (a megfelelő kiegészítésekkel) így néz ki a mi esetünkben:

```
M 30000
/+
+++++ +++++
[
  > +++++ ++
  > +++++ +++++
  > +++
  > +
  <<<< -
]
> ++ .
> + .
+++++ ++ .
.
+++ .
>+ .
<< +++++ +++++ +++++ .
> .
+++ .
----- - .
----- - - .
> + .

/
```

Mindezt persze így is írhatjuk:


```
M 30000  
/+  
+++++[>+++++>+++++>+++++>+++++<<<-]>+.+.+++++. .++>+. <+++++++>+.+.+.----->+.
```

Vagy így:

M 30000 / + ++++++[Y++++>++++>+><<<-]>+.>+.+++++.++>+.<<+++++++>+.>+.
+----->+.>. /

(A fenti mind EGYETLEN sorba van írva!)

Vagy nézzük csak e programot:

M 30000 /+ +[>-----<-[+ +++++ +++++.<-] /

E fenti kis ügyesség annyit csinál, hogy indítás után vár amíg leütünk néhány karaktert. Amikor a karakterek bevitelét befejeztük (azaz Entert nyomtunk) akkor kiírja nekünk a betáplált karaktereket FORDÍTOTT SORRENDEN! Íme egy futás eredménye:

```
vz@Csiszila /Releases/2014/U/Common/vz/MAU=>./mau brainfuck_kiirja_forditva.txt
asdfgh
hqfdsa
```

Ez meg kiírja a teljes ASCII karakterkészletet:

M 30000 /+ .+[.+] /

Ez pedig kiírja a 0-1000 tartományba eső négyzetszámokat:

```

M 30000
/+

++++[>+++++<-]>[<+++++>-]+<+<+
>[>+>+<<-]+>>>[<<+>>-]>>>[-]+>[-]+
>>>+[[ - ]++++++>>>]<<<[[<++++++<+>>-]+<.<[>---<-]<]
<<[>>>>]>>>[-]+++++++<[>-<-]>++++++>[-[<->-]+[<<<]]<[>+<-]>]<<-]<<-
]
[ /* Outputs square numbers from 0 to 10000 */.
// Daniel B Cristofani (cristofdathevanetdotcom)
// http://www
. // hevanet
. // com/cristofd/brainfuck/
]

```

A fenti példákban a tulajdonképpeni brainfuck-részeket nem én találtam ki, hanem úgy leltem fel itt-ott az Interneten. Én csak hozzájuk csaptam azt a 2-3 mau utasítást körítésként, ami kell nekünk. (Lefoglalni a memóriát, stb). Általában ha efféle brainfuck progival találkozunk, azt a következőképp adaptálhatjuk a mau nyelvünkhöz:

1. Kitörölünk belőle minden karaktert, ami nem a 8 brainfuck utasítás, azaz nem a `.,<>+-[]` karakterek valamelyike, vagy pedig megjegyzéssé fokozzuk le őket a `//` karakterpáros eléréseivel. (A sortörések, szóközök, tabulátorok maradhatnak).

2. Beszúrjuk az elejére:

M 30000
(Ez foglalja le a szükséges memóriaterületet)

3. Beszúrjuk az elejére:

/+ Ez kapcsolja be a brainfuck-értelmezést.

4. Szükség esetén, ha a program futása után a prompt rossz helyen jelenne meg, a legvégére szúrjunk be egy

/ jelet, hogy kiírjon egy üres sort még.

Most már boldogok lehetünk, a nyelvünk részhalmazként tartalmazza a **brain-fuck** nyelvet, ami Turing-teljes, TEHÁT ezek után a mi mau nyelvünk is Turing-teljes, efelől nem lehet vita, s ez azt jelenti, hogy minden szaktekintély által elismerten alkalmas a nyelvünk BÁRMI feladat megoldására, ami bármi más programnyelvvél is megoldható egyáltalán!

7. fejezet: Értékadó utasítások

Az gondolom nyilvánvaló, hogy az értékadó utasítások mindegyikében kell szerepeljen az = jel. Igenám, de mi mindennek is lehet értéket adnunk?

1. Változóknak
2. Indexeknek
3. Az adatmemória bájtjainak

Ennyi fajta izémizének biztos kell tudnunk értéket adni, de cseppet se kizárt, hogy ezek száma a későbbiekben még szaporodni fog! Logikusnak tűnik úgy megválasztani az értékadó utasításainkat, hogy mindegyiknek az első karaktere az = jel legyen, a második karakter pedig azt mutassa, épp miféle micsodának is adunk értéket.

Csináljuk tehát meg az = utasításcsoportnak a megfelelő tömböt a szokásos módon, majd íme ízelítőnek 2 utasítás:

```
int csoport_egyenlo_61(F& f) {return masodikkarakter(f);} // Értékadó utasítások
// -----
int egyenloB(F& f) {f.B=(USC)ertekUSIL(f);return 0;}
```

Ezek után már tudunk értéket adni az F struktúra B indexváltozójának, s ezt a korábbi utasításunkkal ki is tudjuk iratni, ami efféleképp történhet a mau programunkban:

```
=B $41
?. ?B
/
```

Ez egy nagy A betűt kell kiírjon, mert a ?. utasításunk karaktert ír ki, márpedig az „A” karakternek a kódja a hexa 41.

Ez pedig:

```
=B $41
?. ?B
/
```

annyit kell kiírjon, hogy

65

mert a sima ? utasítás, pont nélkül, számot ír ki, márpedig a \$41 hexa szám a decimális 65-tel egyenértékű.

Hasonlóképp csináljuk meg az =J utasítást is, amit már nem másolok ide.

Csináljuk meg a vindex - névkezdetű tömbök értékadó utasításait is! Csak az elsőt közlöm itt:

```
int egyenloc(F& f) {USC index;
index=(USC)ertekUSIL(f);
indexerrorMessage(f,15,index);
f.vindexc[f.B]=index;return 0;
}
```

Az e függvényben említett indexerrorMessage függvény kizárólag amiatt jött létre, hogy kevesebbet kelljen gépelni, s rövidebb legyen a forráskód:

```
void indexerrorMessage(F& f,USC hatar,USC index) {
if(index>hatar) {L("Egy értékadó utasításban túllépted a megengedett maximális indexet!\nMax. felső határ: %u akt.érték: %u\n"
"Pozíció a forráskódban: %lu",hatar,index,f.P);EXITFAILURE();}
}
```

Most kéne megalkotnunk a változók értékadását, de rájövünk, hogy itt mégse lehet könnyen megoldani, hogy ezek az utasítások egyenlőségjellel kezdődjenek! Ugyanis minden utasításnál külön kéne valahogy jelölni azt, hogy ez értékadó utasítás, másrészt hogy változónak adunk értéket, harmadrészt hogy a változó unionjának melyik mezőjére vonatkozik az értékadás! Ez 3 karaktert igényelne... Maradjunk tehát annyiban, hogy a változókra vonatkozó értékadó utasítások egyszerűen a @ jellel kezdődnek. Ennek megfelelően tehát:

```
int csoport_kukac_64(F& f) {return masodikkarakter(f);} // Változóknak szóló értékadó utasítások
// -----
int kukacc(F& f) {f.v[f.B].c[f.vindexc[f.B]]=(USC)ertekUSIL(f);return 0;}
// -----
int kukacC(F& f) {f.v[f.B].C[f.vindexC[f.B]]=(signed char)ertekUSIL(f);return 0;}
// -----
int kukaci(F& f) {f.v[f.B].i[f.vindexi[f.B]]=(unsigned short int)ertekUSIL(f);return 0;}
// -----
int kukacI(F& f) {f.v[f.B].I[f.vindexI[f.B]]=(signed short int)ertekUSIL(f);return 0;}
// -----
int kukacL(F& f) {f.v[f.B].l[f.vindexl[f.B]]=(unsigned int)ertekUSIL(f);return 0;}
// -----
int kukacL(F& f) {f.v[f.B].L[f.vindexL[f.B]]=(signed int)ertekUSIL(f);return 0;}
// -----
int kukacf(F& f) {f.v[f.B].f[f.vindexf[f.B]]=(float)ertekUSIL(f);return 0;}
// -----
int kukacg(F& f) {f.v[f.B].g[f.vindexg[f.B]]=(unsigned long long)ertekUSIL(f);return 0;}
// -----
int kukacG(F& f) {f.v[f.B].G[f.vindexG[f.B]]=(signed long long)ertekUSIL(f);return 0;}
// -----
int kukacd(F& f) {f.v[f.B].d[f.vindexd[f.B]]=(double)ertekUSIL(f);return 0;}
// -----
int kukacD(F& f) {f.v[f.B].D=(long double)ertekUSIL(f);return 0;}
```

Tehát például a

@c 36

utasítás a 36-os számot fogja berakni azon változó „c” mezőjébe (azaz oda, ahol az **unsigned char** értékeket tároljuk) amelynek indexét a **B** balérték-változónk tartalmazza. A lehetséges 16 (0-15 közt sorszámozódó) **unsigned char** mező közül pedig abba rakja, amelynek számát a **vindexc[B]** tartalmazza. Amennyiben típuskonverzióra van szükség, azt is végrehajtja. Természetesen észben kell tartanunk, hogy így csak egész értékeket adhatunk egyelőre a változóinknak.

Amennyiben e módszert bonyolultnak találod, sietek elismerni, hogy ebben igazad van, és simán meg lehetett volna oldani az egészet messze sokkal egyszerűbben is! Más se kellett volna hozzá, csak hogy kijelentsük, az interpreterünkben kizárólag a legnagyobb számtartományú lebegőpontos típusnak vannak változói, mindig mindent ebben a típusban számol, s legfeljebb amikor a memória egyes bájtaiba akar értékeket írni, akkor konvertálja át a megfelelő bájthosszúságba beleférfő valamelyik egész típusba a számot! Gyakorlatilag e módszer szerint dolgozik szinte mindegyik interpreter-nyelv.

No igen, csakhogy ez rémségesen lassú...

Ha bonyolult is, de valami módszerünk már van az értékadásra. Ez haladás.

Jó lenne azért a dolgunkon könnyíteni! Mi volna, ha legalább a konkrét változó-indexet is megadhatnánk külön? Valami efféle módon mondjuk:

vl \$41 69

vagy így:

vl 'A 67

és minthogy \$41='A'=65, ezért ez azt jelentené, hogy a 65-ödik változónk l-típusú mezői közül abba, amelynek az indexét a vindexl[65] tárolja, belerak 69-et (az első esetben), vagy 67-et (a második esetben).

Kezd egész barátságos lenni... Ugye sorban így jönnek a betűk a jelentésükkel:

v (változónak adunk értéket)

l (l-típusúnak adunk értéket)

'A (Az „A”-adik, azaz az „A”-nevű változónak adunk értéket)

s ezután jön, hogy mekkora értéket adunk neki.

Nosza írjuk is meg a rutinokat:

```
int csoport_v_118(F& f) {return masodikkarakter(f);} // Változóknak szóló értékadó utasítások
// -----
int v_c(F& f) {USC index;USIL erte;index=(USC)erteUSIL(f);erte=erteUSIL(f);
f.v[index].c[f.vindexc[index]]=(USC)erte;return 0;}
// -----
int v_C(F& f) {USC index;USIL erte;index=(USC)erteUSIL(f);erte=erteUSIL(f);
f.v[index].C[f.vindexC[index]]=(signed char)erte;return 0;}
// -----
int v_i(F& f) {USC index;USIL erte;index=(USC)erteUSIL(f);erte=erteUSIL(f);
f.v[index].i[f.vindexi[index]]=(unsigned short int)erte;return 0;}
// -----
int v_I(F& f) {USC index;USIL erte;index=(USC)erteUSIL(f);erte=erteUSIL(f);
f.v[index].I[f.vindexI[index]]=(signed short int)erte;return 0;}
// -----
int v_l(F& f) {USC index;USIL erte;index=(USC)erteUSIL(f);erte=erteUSIL(f);
f.v[index].l[f.vindexl[index]]=(unsigned int)erte;return 0;}
// -----
int v_L(F& f) {USC index;USIL erte;index=(USC)erteUSIL(f);erte=erteUSIL(f);
f.v[index].L[f.vindexL[index]]=(signed int)erte;return 0;}
// -----
int v_f(F& f) {USC index;USIL erte;index=(USC)erteUSIL(f);erte=erteUSIL(f);
f.v[index].f[f.vindexf[index]]=(float)erte;return 0;}
// -----
int v_g(F& f) {USC index;USIL erte;index=(USC)erteUSIL(f);erte=erteUSIL(f);
f.v[index].g[f.vindexg[index]]=(unsigned long long)erte;return 0;}
// -----
int v_G(F& f) {USC index;USIL erte;index=(USC)erteUSIL(f);erte=erteUSIL(f);
f.v[index].G[f.vindexG[index]]=(signed long long)erte;return 0;}
// -----
int v_d(F& f) {USC index;USIL erte;index=(USC)erteUSIL(f);erte=erteUSIL(f);
f.v[index].d[f.vindexd[index]]=(double)erte;return 0;}
// -----
int v_D(F& f) {USC index;USIL erte;index=(USC)erteUSIL(f);erte=erteUSIL(f);
f.v[index].D=(long double)erte;return 0;}
```

Próbáljuk ki egy kis mau programmal:

```
M 30000

=B 'A
"B értéke : " ? ?B /
vL 'A 67

"Karakter: " ? . @?B /
"Számmérték: " ? @?B /

"Karakter: " ? . @'A /
"Számmérték: " ? @'A /

"Karakter: " ? . @65 /
"Számmérték: " ? @65 /

"Karakter: " ? . @A /
"Számmérték: " ? @A /

vL $41 69

"Karakter: " ? . @?B /
"Számmérték: " ? @?B /

"Karakter: " ? . @'A /
"Számmérték: " ? @'A /

"Karakter: " ? . @65 /
"Számmérték: " ? @65 /

"Karakter: " ? . @A /
"Számmérték: " ? @A /

/
```

E fenti progi a következő kimenetet kell produkálja:

```
B értéke : 65
Karakter: C
Számérték: 67
Karakter: C
Számérték: 67
Karakter: C
Számérték: 67
Karakter: C
Számérték: 67
Karakter: E
Számérték: 69
Karakter: E
Számérték: 69
Karakter: E
Számérték: 69
Karakter: E
Számérték: 69
```

Nagyon pöpec! Van azonban egy kis bökkenő. Ez az egész az **ertekUSIL** függvényre épül, ami mindenféleképpen USIL típusú számot ad nekünk vissza. Ez még önmagában nem lenne baj, mert ebbe belefér minden egész típus, s már e könyv legelején mondtam, hogy a lebegőpontos számokat halasszuk későbbre. Szóval ez visszaad nekünk egy USIL számot, csak hogy ha változóról van szó, akkor abból is mindenféleképpen annak „1” mezőjéből olvassa ki az értéket! Ez persze logikus, hiszen e függvénynek már a nevében is benne van, hogy USIL típusról van szó... Nekünk azonban ki kell tudnunk szedni a változóból a többi mező tartalmát is!

A megoldás természetesen egy „flag” lesz, azaz valami olyan jel, amit ha az „aritmetikai kifejezés” kiértékelése közben beolvas, akkor tudja, hogy a következő változóból milyen típusú adatot kell kiszednie. Erre a célra a „:” karaktert szemeltem ki.

Az új **ertekUSIL** függvényünk:

```
USIL ertekUSIL(F& f) { // A következő „Aritmetikai kifejezés” értékét adja vissza USIL számként.
// A P mutató mutat a kifejezés legelső karakterére.
USC karakter;USC cimke1karakter;USC kar;USC valtozotipus;valtozotipus='l';// Alapértelmezett USIL típus
nemspace(f); // előlépteti a pointert a következő nem whitespace karakterig

karakter=f.p[f.P]; // Az aktuális feldolgozandó karakter
if(karakter=='\') { // Változótípus beolvasása
    valtozotipus=k(f); // A következő karakter beolvasása
    k(f);nemspace(f);karakter=f.p[f.P];}

if(karakter=='@') { // Változó értékét kell beolvasni
    switch(valtozotipus) {
        case 'c': return (USIL)vc(f);break;
        case 'C': return (USIL)vC(f);break;
        case 'i': return (USIL)vi(f);break;
        case 'I': return (USIL)vI(f);break;
        case 'l': return vl(f);break;
        case 'L': return (USIL)vL(f);break;
        case 'f': return (USIL)vf(f);break;
        case 'g': return (USIL)vg(f);break;
        case 'G': return (USIL)vG(f);break;
        case 'd': return (USIL)vd(f);break;
        case 'D': return (USIL)vD(f);break;
        default: return vl(f);break;
    } // switch vége
}
if(karakter=='!') { return felkialtojel(f); } // Memóriacím tartalmát kell beolvasni USIL számként
if(karakter=='#') { return (USIL)kereszt(f); } // Memóriacím tartalmát kell beolvasni USC számként
if(karakter=='?') { return systemvariable(f); } // Rendszerváltozó tartalmát kell visszaadni
if(karakter==194) { // $ karakter első bájta
    karakter=k(f);if(karakter!=167) return 0;
    cimke1karakter=k(f);kar=k(f);
    cimke1karakter |= 128; cimke1karakter -= 0x40; cimke1karakter &= 63;
    if(wspc(kar)) { // A címke csak 1 karakteres mert a második karakter whitespace
        return f.cimke[cimke1karakter];}
    // A címke 2 karakteres
    f.P++; // Azért, hogy a P mutató a következő karakterre mutasson
    kar |= 128; kar -= 0x40; kar &= 63;
    return f.cimke[(unsigned int)cimke1karakter * 64 + (unsigned int)kar];
}
// Különben itt valami számféleségnek kell állnia
return number(f.p,f.P,f.phossz);
}
```

A **vc**, **vC**, **vl** stb függvények közül amikre hivatkozik, néhány példának, a többi mind ugyanerre a kaptafára megy:

```
// =====
USC vc(F& f) { // A programkódban soron következő változó aktuális értékét adja vissza USC típusként.
// A P mutató okvetlenül egy @ jelre kell mutasson!
USC index=INDEX(f);return f.v[index].c[f.vindexc[index]];
}
// -----
signed char vC(F& f) { // A programkódban soron következő változó aktuális értékét adja vissza signed char típusként.
// A P mutató okvetlenül egy @ jelre kell mutasson!
USC index=INDEX(f);return f.v[index].C[f.vindexC[index]];
}
// -----
USIL vl(F& f) { // A programkódban soron következő változó aktuális értékét adja vissza USIL típusként.
// A P mutató okvetlenül egy @ jelre kell mutasson!
USC index=INDEX(f);return f.v[index].l[f.vindexl[index]];
}
```

Az **ertekUSIL** függvény természetesen most is USIL típusként adja a maga eredményét, de most már legalább megszabhatjuk, a változó melyik mezőjét adja vissza USIL típusként! Ez pedig eként történik, amint ebben a mau programban láthatjuk:

```
M 30000
=B 'A
"B értéke : " ? ?B /
=c 10
vc 'A 67

"Karakter: " ? . :c@?B /
"Számtértek: " ? . :c@?B /
"Karakter: " ? . :c@'A /
"Számtértek: " ? . :c@'A /
"Karakter: " ? . :c@65 /
```

```

"Számérték: " ? :c@65 /
"Karakter: " ? :c@A /
"Számérték: " ? :c@A /
/
=i 7; vi 'A 452

"i Szám: " ? :i @?B /
"i Szám: " ? :i @'A /
"i Szám: " ? :i @65 /
"i Szám: " ? :i @$41 /
/
"Karakter: " ? :c@?B /
"Számérték: " ? :c@?B /
"Karakter: " ? :c@'A /
"Számérték: " ? :c@'A /
"Karakter: " ? :c@65 /
"Számérték: " ? :c@65 /
"Karakter: " ? :c@A /
"Számérték: " ? :c@A /
/

vc $41 69

"Karakter: " ? :c @?B /
"Számérték: " ? :c @?B /
"Karakter: " ? :c @'A /
"Számérték: " ? :c @'A /
"Karakter: " ? :c @65 /
"Számérték: " ? :c @65 /
"Karakter: " ? :c @A /
"Számérték: " ? :c @A /
/

```

A program futásának eredménye:

```

B értéke : 65
Karakter: C
Számérték: 67
Karakter: C
Számérték: 67
Karakter: C
Számérték: 67
Karakter: C
Számérték: 67

i Szám: 452
i Szám: 452
i Szám: 452
i Szám: 452

Karakter: C
Számérték: 67
Karakter: C
Számérték: 67
Karakter: C
Számérték: 67
Karakter: C
Számérték: 67

Karakter: E
Számérték: 69
Karakter: E
Számérték: 69
Karakter: E
Számérték: 69
Karakter: E
Számérték: 69

```

Ha már az értékadásnál tartunk, illik ezt megvalósítani az alapvető index-regiszterek közt is:

```

int csoport_B_66(F& f) {return masodikkarakter(f);}
// -----
int BJ(F& f) {f.B=f.J;return 0;}
// =====
int csoport_J_74(F& f) {return masodikkarakter(f);}
// -----
int JB(F& f) {f.J=f.B;return 0;}
// -----
// =====
int csoport_b_98(F& f) {return masodikkarakter(f);}
// -----
int bj(F& f) {register USC c;c=f.J;f.J=f.B;f.B=c;return 0;}
// =====
int csoport_j_106(F& f) {return masodikkarakter(f);}

```

Azaz a **BJ** azt jelenti: B=J; A **JB** azt jelenti: J=B;

A **bj** pedig felcseréli a B és J tartalmát. Ugyanezt csinálja a **jb** utasítás is, de azt nem kellett külön megírni, mert egyszerűen a **bj** függvény címét beírtam a megfelelő tömbbe ami a j-vel kezdődő utasításokat tárolja.

Sajnos, még mindig nehézkesnek érezzük a fenti megoldást! Tudjuk, hogy ha állandóan el vannak tárolva az indexek egy belső változóban, az gyorsabb futást eredményez, mintha minden alkalommal külön kéne beolvasni a forráskódból, de ez akkor se tekinthető „felhasználóbarátnak”. Ráadásul, még az se biztos, hogy tényleg gyorsabb úgy... Gondoljunk csak bele, ha a változó nevét egyetlen karakter ASCII kódja azonosítja, akkor teljesen mindegy hogy azt az 1 bájtnyi karaktert a forráskódból olvassa be, vagy az F struktúrában tárolt B változóból!

Nosza, írjuk csak át a „number” függvényünk elejét így, csak egy sort kell kicserélni benne, amit itt kékkel emeltem ki:

```
USIL num=0L;USC c;
c=p[P];
if((c>='0')&&(c<='9')) goto dec;
if(c=='$') goto hex;
if(c=='%') goto bin;
if(c=='\') goto kar;
if(c=='o') goto oct;
if(c=='O') goto oct;
P++;
return (USIL)c; // Különben visszatér a karakter kódjával
```

Ezek után pedig az alábbi sorok mind ugyanazt jelentik:

```
vcA 69
vc65 69
vc'A 69
vc$41 69
vc0101 69
vcO101 69
vc A 69
vc 65 69
vc 'A 69
vc $41 69
vc o101 69
vc O101 69
```

Tehát egy változóra hivatkozhatunk egyszerűen az egykarakteres nevét megadva - kivéve ha az a karakter az „o” vagy az „O” karakter. Hát ezt ki fogjuk bírni... Na de hoppá, várjunk csak! Miért is kéne ezt kibírnunk?! Simán módosíthatjuk a „number” függvényt úgy, hogy amennyiben az „o” vagy „O” karakter után nem oktális karaktert talál, a visszatérési értéke az „o” vagy „O” ASCII kódja legyen! Íme a függvény oktális beolvasást végző részének új változata:

```
oct: num=(USIL)c;c=p[++P];
if((c<'0')||((c>'7')) return num; else num=0L;
while(P<PMAX) {
num = (num << 3) + (c-'0');
c=p[++P];
```



```

if((c<'0')||(c>'7')) break;
} // oct while vége
return num;

```

Mint látható, csak az első 2 sor változott benne, abból is alig valami (a kék színnel kiemelt részeket szúrtam be).

Remek! Határozottan haladunk. Már csak arra kéne valami megoldást találnunk, hogy azt a fránya indexet, ami az union mezőit indexelgeti, azt tudjuk megadni valami „human readable”, azaz „emberi fogyasztásra alkalmas”, barátságos módon... Például mit szólnánk egy efféle megoldáshoz:

vcA.12 — ahol a pont után természetesen tetszőleges „aritmetikai kifejezés” állhat, aminek az értékét beolvasás után nyilván USC típusra konvertálja! És amennyiben NINCS ott pont, na akkor van az, hogy tudja, hogy az indexet a megfelelő „vindex” névkezdetű tömbből kell előszednie! Amennyiben pedig VAN ott pont, akkor az azutáni számot beolvassa ugyan, és fel is használja indexként, de nem tárolja el a vindex-tömbben, hanem csak ezen egyetlen utasítás kiértékelése alatt használja!

Nosza, írjuk is csak át a rutinokat ennek megfelelően! Íme az egyik, a többi mind e kaptafára megy:

```

int v_c(F& f) {USC index;USIL ertek;USC unionindex;
index=(USC)ertekUSIL(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat
if(f.p[f.P]=='.') {k(f); // A következő karakterre ugrás
unionindex=(USC)ertekUSIL(f); // Beolvassa az union indexét
indexerrorMessage(f,15,unionindex); // Ellenőrzi, hogy a megengedett határon belül van-e
} else unionindex=f.vindexc[index];
ertek=ertekUSIL(f);
f.v[index].c[unionindex]=(USC)ertek;return 0;}

```

Továbbá, ezesetben illik megcsinálni az **ertekUSIL** beolvasófüggvényünket is úgy, hogy a : jel után megadott mezőtípus után is megadható legyen mezőindex! Mint majd látni fogjuk, ezesetben kicsit bele kell nyúlnunk a **vc**, **vl** stb függvények kódjába is, ellenben szerencsére felesleges lesz az **INDEX**(nevű függvény.

Az új **ertekUSIL** függvény, meg a **vc** példának, és két segédfüggvény:

```

USC valtozohatar(USC valtozotipus) {USC hatar;
switch(valtozotipus) { // Ellenőrzi, hogy a megengedett határon belül van-e
case 'c': hatar=15;break;
case 'C': hatar=15;break;
case 'i': hatar=7;break;
case 'I': hatar=7;break;
case 'l': hatar=3;break;
case 'L': hatar=3;break;
case 'f': hatar=3;break;
case 'g': hatar=1;break;
case 'G': hatar=1;break;
case 'd': hatar=1;break;
default: hatar=255;break;
} // switch vége
return hatar;
}
// -----
USIL usilvariable(USC valtozotipus, F& f, USC unionindex) { // Visszaadja a változó értékét USIL típusként.
// A P mutató egy @ jelre kell mutasson! A változóból a valtozotipus-nak megfelelő
// mezőt olvassa ki, aminek az indexét az unionindex kell tartalmazza.
// Ha az unionindex=255, akkor az indexértéket a megfelelő vindex-tömbből veszi.
switch(valtozotipus) {
case 'c': return (USIL)vc(f,unionindex);break;
case 'C': return (USIL)vC(f,unionindex);break;
case 'i': return (USIL)vi(f,unionindex);break;
case 'I': return (USIL)vI(f,unionindex);break;
case 'l': return (USIL)vl(f,unionindex);break;
}
}

```

```

case 'L': return (USIL)vL(f,unionindex);break;
case 'f': return (USIL)vf(f,unionindex);break;
case 'g': return (USIL)vg(f,unionindex);break;
case 'G': return (USIL)vG(f,unionindex);break;
case 'd': return (USIL)vd(f,unionindex);break;
case 'D': return (USIL)vD(f);break;
} // switch vége
return vl(f,unionindex);
}

// =====
USIL ertekUSIL(F& f) { // A következő „Aritmetikai kifejezés” értékét adja vissza USIL számként.
// A P mutató mutat a kifejezés legelső karakterére.
USC karakter;USC cimke1karakter;USC kar;USC változotípus;USC unionindex;USC pont;USC hatar;

változotípus='l'; // Alapértelmezett USIL típus
unionindex=255; // Ez azt jelenti, hogy ez esetben az igazi indexet a vindex-tömbökből kell kiolvasni
nemspace(f); // előlépteti a pointert a következő nem whitespace karakterig
karakter=f.p[f.P]; // Az aktuális feldolgozandó karakter
if(karakter=='.') {változotípus=k(f); // Változótípus beolvasása
pont=k(f);if(pont=='.') {k(f); // A következő karakterre ugrás
unionindex=(USC)ertekUSIL(f); // Beolvassa az union indexét
hatar=változohatar(változotípus);indexerrorMessage(f,hatar,unionindex); // Ellenőrzi, hogy a megengedett határon belül
van-e
} // pont-teszt vége
nemspace(f);karakter=f.p[f.P]; } // kettőspont-teszt vége

if(karakter=='@') { return usilvariable(változotípus,f,unionindex); // Változó értékét kell beolvasni
} // változóbeolvasás vége
if(karakter=='!') { return felkialtojel(f); } // Memóriacím tartalmát kell beolvasni USIL számként
if(karakter=='#') { return (USIL)kereszt(f); } // Memóriacím tartalmát kell beolvasni USC számként
if(karakter=='?') { return systemvariable(f); } // Rendszerváltozó tartalmát kell visszaadni
if(karakter==194) { // $ karakter első bájta
karakter=k(f);if(karakter!=167) return 0;
cimke1karakter=k(f);kar=k(f);
cimke1karakter |= 128; cimke1karakter -= 0x40; cimke1karakter &= 63;
if(wspc(kar)) { // A címke csak 1 karakteres mert a második karakter whitespace
return f.cimke[cimke1karakter];}
// A címke 2 karakteres
f.P++; // Azért, hogy a P mutató a következő karakterre mutasson
kar |= 128; kar -= 0x40; kar &= 63;
return f.cimke[(unsigned int)cimke1karakter * 64 + (unsigned int)kar];
} // $ teszt vége
// Különbben itt valami számféleségnek kell állnia
return number(f.p,f.P,f.phossz);
}
// =====
USC vc(F& f,USC unionindex=255) { // A programkódban soron következő változó aktuális értékét adja vissza USC típusként.
// A P mutató okvetlenül egy @ jelre kell mutasson!
USC index; k(f); index=ertekUSIL(f);if(unionindex==255) unionindex=f.vindexc[index];
return f.v[index].c[unionindex];
}

```

Lássunk megint egy rövid mau progit, ami demonstarálja, miként megy most az értékadás meg a kiíratás:

```

vcA.15 67
"Karakter: " ? . :c.15@A /
"Számmérték: " ? :c.15@A /
"Karakter: " ? . :c.$f@A /
"Számmérték: " ? :c.$f@A /
"Karakter: " ? . :c.$F@A /
"Számmérték: " ? :c.$F@A /
"Karakter: " ? . :c.o17@A /
"Számmérték: " ? :c.o17@A /
"Karakter: " ? . :c.O17@A /
"Számmérték: " ? :c.O17@A /
"Karakter: " ? . :c.%1111@A /
"Számmérték: " ? :c.%1111@A /
/
"Karakter: " ? . :c.15 @A /
"Számmérték: " ? :c.15 @A /
"Karakter: " ? . :c.$f @A /
"Számmérték: " ? :c.$f @A /
"Karakter: " ? . :c.$F @A /
"Számmérték: " ? :c.$F @A /
"Karakter: " ? . :c.o17 @A /
"Számmérték: " ? :c.o17 @A /
"Karakter: " ? . :c.O17 @A /
"Számmérték: " ? :c.O17 @A /
"Karakter: " ? . :c.%1111 @A /
"Számmérték: " ? :c.%1111 @A /
/
vib.3 69
"Karakter: " ? . :i.3@b /
"Számmérték: " ? :i.3@b /

```

```

"Karakter: " ? . :i.$3@b /
"Számtérték: " ? :i.$3@b /
"Karakter: " ? . :i.o3@b /
"Számtérték: " ? :i.o3@b /
"Karakter: " ? . :i.03@b /
"Számtérték: " ? :i.03@b /
"Karakter: " ? . :i.%11@b /
"Számtérték: " ? :i.%11@b /
/
vck 66
"Karakter: " ? . :c@k /
"Számtérték: " ? :c@k /
"Karakter: " ? . :c @k /
"Számtérték: " ? :c @k /
/

```

A program futásának eredménye:

```

Karakter: C
Számtérték: 67
Karakter: C
Számtérték: 67
Karakter: C
Számtérték: 67
Karakter: C
Számtérték: 67
Karakter: C
Számtérték: 67
Karakter: C
Számtérték: 67
Karakter: C
Számtérték: 67
Karakter: C
Számtérték: 67
Karakter: C
Számtérték: 67
Karakter: E
Számtérték: 69
Karakter: E
Számtérték: 69
Karakter: E
Számtérték: 69
Karakter: E
Számtérték: 69
Karakter: E
Számtérték: 69
Karakter: B
Számtérték: 66
Karakter: B
Számtérték: 66

```

Egy másik progi demonstrálja, hogy változóknak címkét is értékül adhatunk... És e mau program már részletesen dokumentált is kommentekkel:

```

vcA.15 101 // e betű
"Karakter: " ? . :c.15@A /
"Számtérték: " ? :c.15@A /
/
viA.2 $ii // Az A változó második unsigned int mezőjébe megy az ii címke értéke
»:i.2@A // Ugrás a változóban tárolt címkére
"Ezt nem szabad kiírnia, mert át kell ugorja!"
$ii // Itt kezdődik az ii címke által jelölt kódrész
vck 66 // A k változóba megy a B karakter
"Karakter: " ? . :c@k / // Mindenféle kiíratások
"Számtérték: " ? :c@k /
"Karakter: " ? . :c @k /
"Számtérték: " ? :c @k /
/ // Újsor kiírása
vc$41 66 // A $41 is az A változó, ebbe tettük a 66 kódú B karaktert
"Karakter: " ? . :c@A /
"Számtérték: " ? :c@A /
/

```

A futásának az eredménye:

```
Karakter: e
Számérték: 101
```

```
Karakter: B
Számérték: 66
Karakter: B
Számérték: 66
```

```
Karakter: B
Számérték: 66
```

A

```
vcA.15 101
```

sort megadhatnánk e variációkban is:

```
vc65.15 101
vc$41.15 101
vc'A.15 101
vco101.15 101
vc0101.15 101
vc%1000001.15 101
```

De nem adhatjuk meg így:

```
vcA .15 101 // EZ EGY HIBÁS UTASÍTÁS, AMIATT IS ILYEN A SZÍNE!
```

Azaz a változó indexét rögvest kell kövesse a pont, nem lehet a kettő közt whitespace karakter! Megengedett azonban e változat:

```
vcA. 15 101
```

Tehát a pont után lehet már whitespace.

Ilyet is művelhetünk:

```
vcA. 15 101 // e betű
"Karakter A: " ? . :c.15@A /
"Számérték A: " ? :c.15@A /
/
vcB.1 :c.15@A // A B változó 1 indexű unsigned char mezőjébe belerakjuk
// az A változó 15-ös unsigned char mezőjének a tartalmát
"Karakter B: " ? . :c.1@B /
"Számérték B: " ? :c.1@B /
/
```

A futtatás eredménye:

```
Karakter A: e
Számérték A: 101
```

```
Karakter B: e
Számérték B: 101
```

Persze, a dolgot tetszőlegesen megcifrázhatjuk, akárhány mélységbe egymásba ágyazva az indirekciókat:

```
vcA. 15 101 // e betű megy az A változó 15-ös unsigned char mezőjébe.
// Az „A” karakter értéke mint tudjuk a 65.
"Karakter A: " ? . :c.15@A / "Számérték A: " ? :c.15@A / // Kiíratjuk az A tartalmát

vcz.0 15 // 15 megy a z változó nulladik unsigned char mezőjébe.
"Számérték z: " ? :c.0@z / // Kiíratjuk a z tartalmát

vcB.5 65 // A B változó 5-ös unsigned char mezőjébe 65-ös értéket teszünk.
"Karakter B: " ? . :c.5@B / "Számérték B: " ? :c.5@B / // Kiíratjuk a B tartalmát
/
vcx.1 5 // 5 megy az x változó 1 indexű unsigned char mezőjébe.
"Számérték x: " ? :c.1@x / // Kiíratjuk az x tartalmát

vcC.7 :c.:c.0@z@c.:c.5@B
// a C változó 7-es unsigned char mezőjébe betöltjük indirekt módon az A-ban levő „e” karaktert
"Karakter C: " ? . :c.7@C / "Számérték C: " ? :c.7@C / // és kiíratjuk
/
vcC.7 :c.:c.0@z@c.:c.1@x@B
// a C változó 7-es unsigned char mezőjébe betöltjük indirekt módon az A-ban levő „e” karaktert
"Karakter C: " ? . :c.7@C / "Számérték C: " ? :c.7@C / // és kiíratjuk
/
```

A programfutás eredménye:

Karakter A: e
Számérték A: 101

Számérték z: 15

Karakter B: A
Számérték B: 65

Számérték x: 5

Karakter C: e
Számérték C: 101

Karakter C: e
Számérték C: 101

Időzzünk el ezen a kis szörnyszüleményen:

:c.:c.0@z@c.:c.1@xB

Ideírom az egyes részek magyarázatát:

:c	unsigned char mező kell majd a változóból.
.	E karakter azt mondja, hogy ezután egy aritmetikai kifejezés jön, ami meghatározza, a változónak hányadik mezőjére van szükségünk.
:c.0@z	Ez az az aritmetikai kifejezés, mely megmondja, hogy majd a változónak hányadik mezője kell. Történetesen ami itt áll az azt jelenti, hogy e számot a z változó nulladik unsigned char mezője tartalmazza.
@	Ez a karakter jelzi, hogy innentől következik a változó indexének a megadása. Ez az index is tetszőleges aritmetikai kifejezés lehet.
:c	Ez azt jelzi, hogy a szükséges szám valamely változó unsigned char mezőjében található.
.	Egy aritmetikai kifejezés jön, mely megmondja, hogy a mezőnek mennyi lesz az indexe.
:c.1@x	A mezőindexet ez az aritmetikai kifejezés határozza meg, ami nem más mint az x változó 1 -es indexű unsigned char mezője.
@	Ez azt jelenti, hogy ezután jön a változó indexe.
B	Ez a változóindex, ami egyszerűen a B karakter ASCII kódja.

Na, ez jó mulatság, s férfimunka volt... A többit hagyjuk a következő fejezetre...

8. fejezet: Műveletek végzése

Mi is az, mit jelent az a fogalom, hogy „műveletvégzés”? Nos, azt, hogy valaminek az értékét megváltoztatjuk. Ez a valami aminek az értékét megváltoztatjuk, a programnyelvekben egy változó szokott lenni. Memóriacímek tartalmát is megváltoztathatjuk, de az úgy szokott történni, hogy onnan kiolvassuk az értéket egy változóba, ott megváltoztatjuk, majd a megváltozott értéket visszairjuk a megfelelő memóriacímre. A memóriakezeléssel azonban később foglalkozunk.

A fenti gondolatmenet azt sugallja nekünk, hogy ez rémségesen hasonló dolog az értékadó utasításokhoz! Ott is arról volt szó, hogy egy változó tartalma megváltozik - tudniillik új érték kerül bele! Most meg egyszerűen arról van szó, hogy nem csak szimplán felülírjuk a régi értéket az újjal, hanem előbb a régi és az új érték között elvégzünk valamiféle műveletet, s ennek eredményével írjuk felül a régi értéket!

Vagyis nekünk most kéne írunk egy rakás olyasféle utasítást, mint amelyeneket az értékadó utasításokhoz írtunk. Írni mondjuk a plusz, vagyis az összeadás művelethez annyi utasítást, ahány típusunk van (11 ha jól emlékszem...) meg írni ugyanennyit a kivásra, szorzásra, osztásra, bitműveletekre... BRRRRR!!!!

Ez bizony ROHADTUL SOK utasítás volna, s nekem erre semmi kedvem, mert LUSTA VAGYOK...

Tulajdonképpen miért is van ennyi rengeteg külön függvény mondjuk az értékadó utasítás esetén?! Nézzük meg, miben különbözik egymástól mondjuk a **vc** és a **vi** függvény! Kizárólag abban, hogy

- az union indexének más a megengedett felső határa
- melyik tömbből veszi az alapértelmezett unionindexet ha azt nem adjuk meg explicite neki,
- a változó unionjának melyik mezőjébe teszi az input paraméterét,
- az input paramétert milyen típusúvá castolja!

Minden más teljesen azonos mindegyik függvényben. Na de ezt mind mégse írhatjuk meg egyetlen függvényben, mert honnan tudná a szerencsétlen, mikor melyik típus szabályai szerint kell eljárnia!

Vagy mégis tudhatja?! HÁT PERSZE! Ne feledjük, az **F** struktúra tartalmaz egy **a2** nevű unsigned char változót, ami tárolja az épp végrehajtott parancs második karakterét! És nekünk szerencsére volt annyi józan eszünk, hogy ez attól függjön, milyen mezőtípussal dolgozik az utasítás... Nosza, töröljük is ki az összes régi függvényünket, s írjuk be mindegyiknek a helyére a megfelelő tömbökbe a

v_mindentípus

nevű függvényt, ami így néz ki, egy segédfüggvényével együtt:

```
USC UNIONindex(F& f, USC index) { // Visszaadja a változó érvényes indexét.  
// Ha van megadva olyan a forráskódban akkor azt, ha nincs megadva akkor a megfelelő vindex-tömbből szedi elő.  
USC unionindex;if(f.p[f.P]=='.') {k(f); // A következő karakterre ugrás  
unionindex=(USC)ertekUSIL(f); // Beolvassa az union indexét  
indexerrorMessage(f,valtozhatar(f.a2),unionindex); // Ellenőrzi, hogy a megengedett határon belül van-e  
} else {  
switch(f.a2) {  
case 'c': unionindex=f.vindexc[index];break;  
case 'C': unionindex=f.vindexC[index];break;  
case 'i': unionindex=f.vindexi[index];break;  
case 'I': unionindex=f.vindexI[index];break;  
case 'l': unionindex=f.vindexl[index];break;  
case 'L': unionindex=f.vindexL[index];break;  
case 'f': unionindex=f.vindexf[index];break;  
case 'g': unionindex=f.vindexg[index];break;  
case 'G': unionindex=f.vindexG[index];break;  
case 'd': unionindex=f.vindexd[index];break;  
default: unionindex=255;break;  
} // switch vége  
} // else vége  
return unionindex;  
}  
// -----
```

```

int v_mindentipus(F& f) {USC index;USIL ertekek;USC unionindex;
index=(USC)ertekekUSIL(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat
if(f.a2!='D') unionindex=UNIONindex(f,index); // Kiszámítja az unionindexet
ertekek=ertekekUSIL(f); // Beolvassa az értéket, amit el kell tárolni
switch(f.a2) {
case 'c': f.v[index].c[unionindex]=(USC)ertekek;break;
case 'C': f.v[index].C[unionindex]=(signed char)ertekek;break;
case 'i': f.v[index].i[unionindex]=(unsigned short int)ertekek;break;
case 'I': f.v[index].I[unionindex]=(signed short int)ertekek;break;
case 'l': f.v[index].l[unionindex]=(unsigned int)ertekek;break;
case 'L': f.v[index].L[unionindex]=(signed int)ertekek;break;
case 'f': f.v[index].f[unionindex]=(float)ertekek;break;
case 'g': f.v[index].g[unionindex]=(unsigned long long)ertekek;break;
case 'G': f.v[index].G[unionindex]=(signed long long)ertekek;break;
case 'd': f.v[index].d[unionindex]=(double)ertekek;break;
case 'D': f.v[index].D=(long double)ertekek;break;
} // switch vége
return 0;
}
// -----

```

Mekkora nagyok vagyunk már! Nem is lett hosszú ez a két függvény, de egy egész rakás mindenfélét megspóroltunk vele a programkódból!

Rögvest fellelkesülünk, s elbánunk az egyenlőségjellel kezdődő azon utasításokkal is, melyek valamely vindex-tömbbe töltenek indexet:

```

int egyenlo_mindentipus(F& f) {USC index;index=(USC)ertekekUSIL(f);indexerrorMessage(f,valtozohatar(f.a2),index);
switch(f.a2) {
case 'c': f.vindexc[f.B]=index;break;
case 'C': f.vindexC[f.B]=index;break;
case 'i': f.vindexi[f.B]=index;break;
case 'I': f.vindexI[f.B]=index;break;
case 'l': f.vindexl[f.B]=index;break;
case 'L': f.vindexL[f.B]=index;break;
case 'f': f.vindexf[f.B]=index;break;
case 'g': f.vindexg[f.B]=index;break;
case 'G': f.vindexG[f.B]=index;break;
case 'd': f.vindexd[f.B]=index;break;
} // switch vége
return 0;
}

```

Minthogy ezen utasítások különben is rövidek voltak külön-külön, mindössze egysorosak, emiatt nem hinném hogy itt bármi érdemleges mértékben csökkent volna a kódméret, ellenben tagadhatatlan, hogy most, hogy ez itt mind egy helyen van, az egész messze sokkal áttekinthetőbb!

Ezek után már könnyedén elbánunk mondjuk az összeadó utasítással. Csak a **v_mindentipus** függvényt kell lemásolnunk, s icipicit átalakítanunk. Íme ami lett belőle:

```

int plusz_mindentipus(F& f) {USC index;USIL ertekek;USC unionindex;
index=(USC)ertekekUSIL(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat
if(f.a2!='D') unionindex=UNIONindex(f,index); // Kiszámítja az unionindexet
ertekek=ertekekUSIL(f); // Beolvassa a jobbtérteket
switch(f.a2) {
case 'c': f.v[index].c[unionindex]+=(USC)ertekek;break;
case 'C': f.v[index].C[unionindex]+=(signed char)ertekek;break;
case 'i': f.v[index].i[unionindex]+=(unsigned short int)ertekek;break;
case 'I': f.v[index].I[unionindex]+=(signed short int)ertekek;break;
case 'l': f.v[index].l[unionindex]+=(unsigned int)ertekek;break;
case 'L': f.v[index].L[unionindex]+=(signed int)ertekek;break;
case 'f': f.v[index].f[unionindex]+=(float)ertekek;break;
case 'g': f.v[index].g[unionindex]+=(unsigned long long)ertekek;break;
case 'G': f.v[index].G[unionindex]+=(signed long long)ertekek;break;
case 'd': f.v[index].d[unionindex]+=(double)ertekek;break;
case 'D': f.v[index].D+=(long double)ertekek;break;
} // switch vége
return 0;
}

```

Egy mau példa arra, hogyan működik:

```
vcA.15 101 // e betű megy az A változó 15-ös unsigned char mezőjébe.
// Az „A” karakter értéke mint tudjuk a 65.
"Karakter A: " ? . :c.15@A / "Számérték A: " ? :c.15@A / // // Kiíratjuk az A tartalmát

"Összeadás következik!" /

+cA.15 1 // az A változó 15-ös unsigned char mezőjéhez hozzáadtunk 1-et
"Karakter A: " ? . :c.15@A / "Számérték A: " ? :c.15@A / // és kiíratjuk
/
```

A futás eredménye:

```
Karakter A: e
Számérték A: 101

Összeadás következik!
Karakter A: f
Számérték A: 102
```

Ezek után azt hiszem, nyilvánvaló, miként csináljuk meg a kivonás, szorzás, osztás és hasonlók függvényeit: Mindegyik a **plusz_mindentípus** másolatra lesz, csak a nevük lesz más (mondjuk **minusz_mindentípus**), és a függvényben szereplő **+=** jeleket kell átírni megfelelően, mondjuk **-=** jelekre! Minthogy ez olyan egyszerű hogy egy óvoda is végre tudja hajtani, nem is másolom be ide a kódokat.

Oké, de most már ideje azzal foglalkoznunk, hogy egyelőre csak egész számokat tudunk kezelni... Azaz meg kell valósítsuk a „lebegőpontos aritmetikát”.

Ez részben eddig is megvalósult, mert konvertálgattunk mi mindenféle típusokat, de mindet csak „USIL” értékbe, ami nem igazán jó, mert adatvesztést eredményezhet. Nekünk az kell, hogy a megadott mezőtípus szerint olvassa be és adja vissza a változó (vagy bármi egyéb) értékét.

Sajnos, ez azt eredményezi, hogy búcsút kell mondjunk eddig oly jó szolgálatot tett **ertekUSIL** függvényünknek! Lesz helyette annyi új függvény, ahány adat-típusunk van eddig: 11 darab... Csak az egyiket mutatom be itt, a többi ennek mintájára készül:

```
unsigned char ERTEKunsignedchar(F& f) { // A következő „Aritmetikai kifejezés” értékét adja vissza unsigned char
számként.
// A P mutató mutat a kifejezés legelső karakterére.
USC karakter;USC változotípus;USC unionindex;USC pont;USC hatar;
változotípus='c'; // Alapértelmezett típus
unionindex=255; // Ez azt jelenti, hogy ez esetben az igazi indexet a vindex-tömbökből kell kiolvasni
nemspace(f); // előlépteti a pointert a következő nem whitespace karakterig
karakter=f.p[f.P]; // Az aktuális feldolgozandó karakter
if(karakter=='.') {változotípus=k(f); // Változótípus beolvasása
pont=k(f);if(pont=='.') {k(f);// A következő karakterre ugrás
unionindex=ERTEKunsignedchar(f); // Beolvassa az union indexét
hatar=valtozhatar(vváltozotípus);indexerrormessage(f,hatar,unionindex); // Ellenőrzi, hogy a megengedett határon belül
van-e
} // pont-teszt vége
nemspace(f);karakter=f.p[f.P]; } // kettőspont-teszt vége

if(karakter=='@') {
switch(vváltozotípus) {
case 'c': return (unsigned char)vc(f,unionindex);break;
case 'C': return (unsigned char)vC(f,unionindex);break;
case 'i': return (unsigned char)vi(f,unionindex);break;
case 'I': return (unsigned char)vI(f,unionindex);break;
case 'l': return (unsigned char)vl(f,unionindex);break;
case 'L': return (unsigned char)vL(f,unionindex);break;
case 'f': return (unsigned char)vf(f,unionindex);break;
case 'g': return (unsigned char)vg(f,unionindex);break;
case 'G': return (unsigned char)vG(f,unionindex);break;
case 'd': return (unsigned char)vd(f,unionindex);break;
case 'D': return (unsigned char)vD(f);break;
```



```

} // switch vége
} // változóbeolvasás vége
if(karakter=='?') { return (unsigned char)systemvariable(f); } // Rendszerváltozó tartalmát kell visszaadni
if(karakter==194) { return (unsigned char)cimkeertekviszaad(f); } // 5 teszt vége
// Különbben itt valami számféleségnek kell állnia
return (unsigned char)number(f.p,f.P,f.phossz);
}

```

A **cimkeertekviszaad** függvény:

```

unsigned int cimkeertekviszaad(F& f) {USC kar;USC karakter;USC cimke1karakter;
karakter=k(f);if(karakter!=167) return 0;
cimke1karakter=k(f);kar=k(f);
cimke1karakter |= 128; cimke1karakter -= 0x40; cimke1karakter &= 63;
if(wspc(kar)) { // A cimke csak 1 karakteres mert a második karakter whitespace
return f.cimke[cimke1karakter];}
// A cimke 2 karakteres
f.P++; // Azért, hogy a P mutató a következő karakterre mutasson
kar |= 128; kar -= 0x40; kar &= 63;
return f.cimke[(unsigned int)cimke1karakter * 64 + (unsigned int)kar];
}

```

A **vc**, **vl** stb függvényeket is át kell írunk, ahogy e példa mutatja:

```

USC vc(F& f,USC unionindex=255) {// A programkódban soron következő változó aktuális értékét adja vissza USC típusként.
// A P mutató okvetlenül egy @ jelre kell mutasson!
USC index; k(f); index=ERTEKunsignedchar(f);if(unionindex==255) unionindex=f.vindexc[index];
return f.v[index].c[unionindex];
}

```

És természetesen az értékadó illetve műveletvégző utasításaink is átírandóak, amint ezt az összeadást megvalósítón bemutatom itt:

```

int plusz_mindentipus(F& f) {USC index;USC unionindex;
index=ERTEKunsignedint(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat
if(f.a2!='D') unionindex=UNIONindex(f,index); // Kiszámítja az unionindexet
switch(f.a2) {
case 'c': f.v[index].c[unionindex]+=ERTEKunsignedchar(f);break;
case 'C': f.v[index].C[unionindex]+=ERTEKsignedchar(f);break;
case 'i': f.v[index].i[unionindex]+=ERTEKunsignedshortint(f);break;
case 'I': f.v[index].I[unionindex]+=ERTEKsignedshortint(f);break;
case 'l': f.v[index].l[unionindex]+=ERTEKunsignedint(f);break;
case 'L': f.v[index].L[unionindex]+=ERTEKsignedint(f);break;
case 'f': f.v[index].f[unionindex]+=ERTEKfloat(f);break;
case 'g': f.v[index].g[unionindex]+=ERTEKunsignedlonglong(f);break;
case 'G': f.v[index].G[unionindex]+=ERTEKsignedlonglong(f);break;
case 'd': f.v[index].d[unionindex]+=ERTEKdouble(f);break;
case 'D': f.v[index].D+=ERTEKlongdouble(f);break;
} // switch vége
return 0;
}

```

Ezek után azonban át kell írunk a kiíratást megvalósító „?” utasításunkat is, hogy minden típust ki tudjon írni! Ehhez azonban meg kell neki mondanunk, épp miféle típus kiírását várjuk tőle. Legegyszerűbb, ha ezt is a már megszokott módon jelezzük, a ? után tett karakterre, s így lesznek nekünk ?c, ?I stb utasításaink, de mindegyiket ugyanazzal az egy függvénnyel valósítjuk meg, amely az F struktúra a2 változója alapján dönti el, mit kell csinálnia. Ha pedig whitespace követi a kérdőjelet, vagy bármi nem típusjelző karakter, akkor egyszerűen kiírja karakterként az értéket. Természetesen ha idézőjelet talál, a szöveget is kiírja. Ettől kezdve a **?** függvényre többé nem lesz szükségünk:

```

int kerdojel(F& f) {USC kar;
namespace(f);
kar=f.p[f.P];
if(kar==34) {k(f);return textkiir(f);} // szövegkiírás
switch(f.a2) {
case 'c': printf("%u",ERTEKunsignedchar(f));break;
case 'C': printf("%d",ERTEKsignedchar(f));break;
case 'i': printf("%u",ERTEKunsignedshortint(f));break;
case 'I': printf("%d",ERTEKsignedshortint(f));break;
case 'l': printf("%u",ERTEKunsignedint(f));break;
case 'L': printf("%d",ERTEKsignedint(f));break;

```

```

case 'f': printf("%f",ERTEKFloat(f));break;
case 'g': printf("%llu",ERTEKunsignedlonglong(f));break;
case 'G': printf("%lld",ERTEKsignedlonglong(f));break;
case 'd': printf("%g",ERTEKdouble(f));break;
case 'D': printf("%Lg",ERTEKlongdouble(f));break;
default: printf("%c",ERTEKunsignedchar(f));break;
} // switch vége
return 0;
}

```

Ezek után így megy a mau nyelven a kiíratás, ahogy ezen a példán láthatjuk:

```

vcA. 15 101 // e betű megy az A változó 15-ös unsigned char mezőjébe.
// Az „A” karakter értéke mint tudjuk a 65.
"Karakter A: " ? :c.15@A / "Számérték A: " ?c :c.15@A / // Kiíratjuk az A tartalmát

vcz.0 15 // 15 megy a z változó nulladik unsigned char mezőjébe.
"Számérték z: " ?c :c.0@z / // Kiíratjuk a z tartalmát

vcB.5 65 // A B változó 5-ös unsigned char mezőjébe 65-ös értéket teszünk.
"Karakter B: " ? :c.5@B/ "Számérték B: " ?c :c.5@B / // Kiíratjuk a B tartalmát
/
vcx.1 5 // 5 megy az x változó 1 indexű unsigned char mezőjébe.
"Számérték x: " ?c :c.1@x / // Kiíratjuk az x tartalmát

vcC.7 :c.:c.0@z@c.:c.5@B
// a C változó 7-es unsigned char mezőjébe betöltjük indirekt módon az A-ban levő „e” karaktert
"Karakter C: " ? :c.7@C / "Számérték C: " ?c :c.7@C / // és kiíratjuk
/
vcC.7 :c.:c.0@z@c.:c.1@xB
// a C változó 7-es unsigned char mezőjébe betöltjük indirekt módon az A-ban levő „e” karaktert
"Karakter C: " ? :c.7@C / "Számérték C: " ?c :c.7@C / // és kiíratjuk
/

```

Eredménye:

```

Karakter A: e
Számérték A: 101

Számérték z: 15

Karakter B: A
Számérték B: 65

Számérték x: 5

Karakter C: e
Számérték C: 101

Karakter C: e
Számérték C: 101

```

Felmerül azonban bennünk az ötlet, miért is kéne mindig megadnunk a különböző típusokat beolvasó függvényeinknek a típust, akkor, ha csak a típus mezőindexét akarjuk megadni! Hiszen ha - mondjuk - az ERTEKunsignedchar függvényt hívjuk meg, s úgy, hogy egy változó értékét olvastatjuk be vele, akkor többnyire úgyis arról van szó, hogy ezen változó unsigned char típusú értékei közül olvasuk be valamelyiket, s ezesetben csak a mező indexét kell megadnunk, hát hadd ne kelljen már feleslegesen leírkálnunk mindig a típusjelet is!

Nosza, megint átírjuk a megfelelő értékbeolvasó függvényeink elejét! Íme itt egy új változat (csak az ERTEKunsignedchar elejét közlöm, mintának):

```

unsigned char ERTEKunsignedchar(F& f) { // A következő „Aritmetikai kifejezés” értékét adja vissza unsigned char
számként.
// A P mutató mutat a kifejezés legelső karakterére.
USC karakter;USC változotípus;USC unionindex;
változotípus='c'; // Alapértelmezett típus
karakter=kettospontteszt(f,unionindex,vváltozotípus); // Az aktuális feldolgozandó karakter

if(karakter=='@') {
switch(vváltozotípus) {
case 'c': return (unsigned char)vc(f,unionindex);break;
case 'C': return (unsigned char)vC(f,unionindex);break;
.....

```

Sokkal egyszerűbb lett a függvény eleje, de azon az áron, hogy hivatkozik egy „kettospontteszt” nevű függvényre. Az meg egy „vtype” nevűre hivatkozik. Itt van mindkettő:

```

USC vtype(F& f, USC változotípus) {USC v;USC pont;
v=k(f);switch(v) {
case 'c':
case 'C':
case 'i':
case 'I':
case 'l':
case 'L':
case 'g':
case 'G':
case 'f':
case 'd':
case 'D': k(f);break;
default : v=változotípus;
} // switch vége
pont=f.p[f.P];if(pont=='.') {k(f);} // A következő karakterre ugrás
return v;
}
// =====
USC kettospontteszt(F& f,USC &unionindex,USC &változotípus) {USC karakter;
unionindex=255; // Ez azt jelenti, hogy ezesetben az igazi indexet a vindex-tömbökből kell kiolvasni
nemspace(f); // előlépteti a pointert a következő nem whitespace karakterig
karakter=f.p[f.P]; // Az aktuális feldolgozandó karakter
if(karakter=='.') {változotípus=vtype(f,változotípus); // Változótípus beolvasása
unionindex=ERTEKunsignedchar(f); // Beolvassa az union indexét
indexerrormessage(f,változohatar(változotípus),unionindex); // Ellenőrzi, hogy a megengedett határon belül van-e
nemspace(f);karakter=f.p[f.P]; } // kettospont-teszt vége
return karakter;
}

```

Egyik se különösebben hosszú, s mindegyiket csak egyszer kellett megírni, ellenben így jelentősen lerövidült mind a 11 értékbeolvasó függvényünk! Emlékezzünk csak: legelső efféle függvényünk, az egykori **ertekUSIL** nevű bizony alig fért ki egy képernyőre, szörnyen zsúfolt volt... Most meg igaz hogy több ilyenünk is van, de csak mert már minden típust tudunk kezelni, s külön-külön mindegyik rövid! S mert a típus- és mezőindex beolvasás mindegyiknél ugyanúgy kell menjen, e feladatokat természetesen „kiszerveztem” egy külön rutinba.

Egyetlen aprócska gondunk van már csak: amennyiben ezen értékbeolvasó függvények úgy találják, hogy mégse változót kell beolvasniuk, azesetben makacsul pozitív egész számot akarnának beolvasni! Ez helytelen. Ezt meg kell oldjuk mindegyiknek a végén. Íme:

```

unsigned char ERTEKunsignedchar(F& f) {
.....
// Különbben itt valami számféleségnek kell állnia
return (unsigned char)number(f.p,f.P,f.phossz);
}
.....

signed char ERTEKsignedchar(F& f) {
.....
// Különbben itt valami számféleségnek kell állnia
return (signed char)signednumber(f.p,f.P,f.phossz);
}
.....

float ERTEKfloat(F& f) {
.....
// Különbben itt valami számféleségnek kell állnia
if((karakter=='$')||(karakter=='%')||(karakter=='o')||(karakter=='0')||(karakter=='\'))
return (float)number(f.p,f.P,f.phossz);
double tort;char *vege;
tort = strtod((char *) (f.p + f.P), &vege);
f.P=vege - (char *)f.p;
return (float)tort;
}
.....

long double ERTEKlongdouble(F& f) {
.....

```

```
// Különböző karakterek ellenőrzése
if((karakter=='$')||(karakter=='%')||(karakter=='o')||(karakter=='0')||(karakter=='\'))
    return (long double)number(f.p,f.P,f.phossz);
long double tort;char *vege;
tort = strtold((char *)f.p + f.P, &vege);
f.p=vege - (char *)f.p;
return tort;
}
```

A többit felesleges lenne bemutatni, az elv érthető ennek alapján is gondolom: Ha egész típusról van szó, ami előjel nélküli, akkor a „**number**” függvényt hívjuk meg, amit már jól ismerünk, s ennek visszaadott értékét castoljuk a megfelelőképpen. Ha előjeles egész típusról van szó, akkor egy „**signednumber**” típusú függvényt hívunk meg, s annak értékét castoljuk. Végezetül pedig ha valami lebegőpontos számról van szó, akkor a számot az **strtod** rendszerfüggvénnyel olvassuk be, s castoljuk a float típus esetén, ha meg amúgy is double számot kell visszaadni, akkor nem castoljuk. Long double esetén pedig a számot eleve az **strtold** függvénnyel olvassuk be. Amennyiben lebegőpontos számot kell visszaadnunk, de a string a \$, %, ', o vagy O karakterek valamelyikével kezdődik, akkor is a **number** függvénnyel olvastatjuk be a számot, s a visszatérési értéket castoljuk.

Illik emiatt bemutatni itt a **signednumber** függvényt:

```
signed long int signednumber(USC *p, USIL& P, USIL PMAX) {
// beolvassa a számot a stringből, ami a p pointer P-edik karakterénél kezdődik.
// A p[P] -edik karaktertől függ, a konverzió milyen számrendszerben hajtódik végre.
// Ha ez 0-9 : decimális szám. Ha % : bináris szám. Ha $ : hexadecimális szám. Ha o vagy 0 : oktális szám
// 'x' : Az x karakter ASCII kódja
// A visszatéréskor a P mindig az első olyan karakterre fog mutatni, ami már nem tartozik a megfelelő számrendszer
// tartományába.
// A számértéket megelőzheti egy „-” jel mint negatív előjel, vagy egy + jel, ez utóbbit nem veszi figyelembe.
USC c;
c=p[P];
if(c=='-') {P++;if(P>=PMAX) {return 0L;} return ((-1)*(signed long int)number(p, P, PMAX));}
if(c=='+') {P++;if(P>=PMAX) return 0L;}
return (signed long int)number(p, P, PMAX);
}
```

Látható, hogy e függvény gyakorlatilag csak egy wrapper a **number** függvényhez, mindössze lecsekkolja előbb, van-e megadva negatív előjel, mert ha igen, akkor a **number** függvény visszatérési értékét megszorozza vele. Az esetleges pozitív előjelet pedig egyszerűen átugorja.

Most már tudunk negatív számokat is beolvasni, meg tört számokat, szóval most már mintha kezdenénk hasonlítani legalább úgy a távolból holmi „rendes” programnyelvre...

Nézzünk ennek bemutatására néhány picit mau programot:

```
-dd.0 45.7
"Számérték d: " ?d :d.0@d // és kiíratjuk
vdp.0 -92.7
"Számérték p: " ?d :d.0@p // és kiíratjuk
vDq -1234.1237
"Számérték q: " ?D :D@q // és kiíratjuk
/

-dd.1 645.7
"Számérték d: " ?d :1@d // és kiíratjuk
"Számérték d: " ?d :1 @d // és kiíratjuk
"Számérték d: " ?d :d.1@d // és kiíratjuk
"Számérték d: " ?d :d.1 @d // és kiíratjuk
vdp.0 -692.7
"Számérték p: " ?d @p // és kiíratjuk
vDq -12345.756
```

```

"Számérték q: " ?D @q / // és kiíratjuk
/
"Próba:"/
"Számérték d: " ?d :.0@d / // és kiíratjuk
"Számérték p: " ?d :.0@p / // és kiíratjuk
"Számérték q: " ?D :.0@q / // és kiíratjuk
/
"Próba 2:"/
"Számérték d: " ?d :0 @d / // és kiíratjuk
"Számérték p: " ?d :0 @p / // és kiíratjuk
"Számérték q: " ?D :0 @q / // és kiíratjuk
/
"Próba 3:"/
"Számérték d: " ?d :0@d / // és kiíratjuk
"Számérték p: " ?d :0@p / // és kiíratjuk
"Számérték q: " ?D :0@q / // és kiíratjuk
/
"Próba 4:"/
vCs.6 -85
"Számérték s: " ?C :6@s / // és kiíratjuk
vCs.6 -$11 // 17
"Számérték s: " ?C :6@s / // és kiíratjuk
vCs.6 -o11 // 9
"Számérték s: " ?C :6@s / // és kiíratjuk
vCs.6 -012 // 10
"Számérték s: " ?C :6@s / // és kiíratjuk
vCs.6 -%1011 // 11
"Számérték s: " ?C :6@s / // és kiíratjuk
/
vCs.6 +85
"Számérték s: " ?C :6@s / // és kiíratjuk
vCs.6 +$11 // 17
"Számérték s: " ?C :6@s / // és kiíratjuk
vCs.6 +o11 // 9
"Számérték s: " ?C :6@s / // és kiíratjuk
vCs.6 +012 // 10
"Számérték s: " ?C :6@s / // és kiíratjuk
vCs.6 +%1011 // 11
"Számérték s: " ?C :6@s / // és kiíratjuk
/
/
vCs.6 85
"Számérték s: " ?C :6@s / // és kiíratjuk
vCs.6 $11 // 17
"Számérték s: " ?C :6@s / // és kiíratjuk
vCs.6 o11 // 9
"Számérték s: " ?C :6@s / // és kiíratjuk
vCs.6 012 // 10
"Számérték s: " ?C :6@s / // és kiíratjuk
vCs.6 %1011 // 11
"Számérték s: " ?C :6@s / // és kiíratjuk
/

```

Eredménye:

```

Számérték d: -45.7
Számérték p: -92.7
Számérték q: 0

```

```

Számérték d: -645.7
Számérték d: -645.7
Számérték d: -645.7
Számérték d: -645.7
Számérték p: -692.7
Számérték q: -12345.8

```

```

Próba:
Számérték d: -45.7
Számérték p: -692.7
Számérték q: -12345.8

```

```

Próba 2:
Számérték d: -45.7
Számérték p: -692.7
Számérték q: -12345.8

```

```

Próba 3:
Számérték d: -45.7
Számérték p: -692.7
Számérték q: -12345.8

```

```

Próba 4:
Számérték s: -85
Számérték s: -17
Számérték s: -9

```

Számérték s: -10
Számérték s: -11

Számérték s: 85
Számérték s: 17
Számérték s: 9
Számérték s: 10
Számérték s: 11

Számérték s: 85
Számérték s: 17
Számérték s: 9
Számérték s: 10
Számérték s: 11

Látjuk, hogy ez a rész:

vDq -1234.1237
"Számérték q: " ?D :D@q / // és kiíratjuk

Nullát ír ki nekünk! Azaz jegyezzük meg a szintaxisról, hogy a : jel után követlenül meg kell adnunk valamilyen számot. Típust nem feltétlenül, de számot, azt igen.

Általában véve tehát nekünk egy változó leírása az úgynevezett Backus-Naur forma (röviden BNF) szintaxisa szerint (Itt a link az erről szóló szócikkre a magyar Wikipédiában: https://hu.wikipedia.org/wiki/Backus%E2%80%93Naur_forma) :

```
<WhiteSpace> ::= "0"|"9"|"10"|"11"|"12"|"13"|"32"  
<valtozo> ::= [<TipusEsIndex>] [{<WhiteSpace>}] "@" <USC>  
<TipusEsIndex> ::= ":" [[<Tipus>] "."] <USC>  
<USC> ::= (unsigned char típusra castolt)<aritmetikaikifejezes>  
<Tipus> ::= "c"|"C"|"i"|"I"|"l"|"L"|"g"|"G"|"d"|"D"|"f"  
<aritmetikaikifejezes> ::= [{<WhiteSpace>}] (<Number> | <valtozo>)  
<Number> ::= <Dec>|<Hex>|<Oct>|<Bin>|<Apostrofalt>|<Cimke>|<SystemVariable>|<EgyKarakter>  
<Dec> ::= [<Elojel>] <DecSzamjegy> [{<DecSzamjegy>}]  
<Elojel> ::= "+"|"-"  
<DecSzamjegy> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"  
<Hex> ::= [<Elojel>] "$" {<HexSzamjegy>}  
<HexSzamjegy> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"  
|"a"|"b"|"c"|"d"|"e"|"f"|"A"|"B"|"C"|"D"|"E"|"F"  
<Oct> ::= [<Elojel>] <OctSign> [{<OctSzamjegy>}]  
<OctSign> ::= "o"|"0"  
<OctSzamjegy> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"  
<Bin> ::= [<Elojel>] "%" {<BinSzamjegy>}  
<BinSzamjegy> ::= "0"|"1"  
<Apostrofalt> ::= "'" <egy_akarmiféle_tetszoleges_bajt>  
  
<Cimke> ::= "$" <CimkeKarakter> [<CimkeKarakter>]  
<CimkeKarakter> ::= <egy_akarmiféle_tetszoleges_bajt kivéve <WhiteSpace>>  
<SystemVariable> ::= "?" <SysVarChar>  
<SysVarChar> ::= "1"|"2"|"a"|"b"|"B"|"c"|"C"|"d"|"E"|"f"|"g"|"G"|"h"  
|"H"|"i"|"I"|"j"|"L"|"l"|"m"|"p"|"P"|"v"|"V"|"x"|"X"  
<EgyKarakter> ::= <Tetszoleges_karakter kivéve <JelölőFlag>>  
<JelölőFlag> ::= <DecSzamjegy>|"$"|"%"|"'"|"?"|"@"|"s"
```

A fenti leírás természetesen csak nyelvünk jelen pillanatára érvényes, hiszen mi sem akadályoz bennünket abban, hogy még akárhány számrendszert is felvegyünk nyelvünkbe, amiknek nyilván szükséges bevezetni előtét karaktert is, ezáltal bővül a <JelölőFlag> lista, vagy bevezethetünk még akárhány újabb rendszerváltozót is...

A fentiek fényében a következő kis progi szemlélteti a lehetséges szintaxisok egy részét:

```

"Variációk: " /
vCs.6 %1011 // Ez binárisan a decimális 11-nek felel meg
": 6@s " ?C : 6@s /
":6@s " ?C :6@s /
": 6 @s " ?C : 6 @s /
":6 @s " ?C :6 @s /
":C 6@s " ?C :C 6@s /
":C. 6@s " ?C :C. 6@s /
":C . 6@s NEM JÓ!" /
":C .6@s NEM JÓ!" /
":C @s " ?C :C @s /
/

```

A futási eredmény:

```

Variációk:
: 6@s 11
:6@s 11
: 6 @s 11
:6 @s 11
:C 6@s 11
:C. 6@s 11
:C . 6@s NEM JÓ!
:C .6@s NEM JÓ!
:C @s 47

```

Azon sorok melyek végére a „NEM JÓ!” van kiírva, hibásak, valószínű is hogy a progi hibajelzéssel „elhasal” majd. Ami meg a legutolsó sort illeti, látnivaló hogy teljesen hülye értéket írt ki, 47-et a 11 helyett, aminek az az oka, hogy itt a @s változó értéké a beolvasandó változó mezőindexének felelteti meg, majd megy tovább a szerencsétlen a kód értelmezésében, s amit ezután talál, abból kutyul valami értéket, amiről a fene se tudja hogy mi lesz. Vagyis az ökölszabály az kell legyen, hogy ha megadtunk „;” jelet, akkor a kettőspont után mindenféleképpen adjunk is meg valami numerikus értéket mezőindexnek! Típusjelölő karaktert nem muszáj, de mezőindexet okvetlenül!

Mégegy rövid program, illusztrációnak, hogyan megy most a műveletvégzés:

```

"Kivonás: 26-11" /
vcs.6 11 // 11-et töltünk az „s” változó 6-odik unsigned char mezőjébe
"s = " ?c :6@s // Kiíratjuk az „s” értékét
vik.2 26 // 26-ot töltünk a „k” változó 2-odik unsigned short int mezőjébe
"k = " ?i :2@k // Kiíratjuk a „k” értékét
-ik.2 :c6@s // Kivonjuk a „k”-ból az „s”-t
"k értéke a kivonás után: " ?i :2@k // Kiíratjuk a „k” új értékét
/

```

Eredménye:

```

Kivonás: 26-11
s = 11
k = 26
k értéke a kivonás után: 15

```

Na és most megint hadd térjek vissza arra, hogy az „aritmetikai kifejezés” leírására szolgáló BNF formulánk csak a jelen pillanatot tükrözi. Ugyanis nemcsak lehetséges, de egészen biztos is, hogy ez bővülni fog a jövőben! Tudniillik, mert mindenféleképpen kell nekünk tudnunk kezelni memóriaterületet is, sőt még veremtárat is! A memóriaterülettel foglalkozunk mindjárt a következő fejezetben.

9. fejezet: Memóriaműveletek

Ami a memóriaműveleteket illeti, egyelőre úgy tűnik, elég lesz belőle kettő: Az egyik ami beolvas valamit a memóriából, a másik pedig ami kiír oda valamit. Memória alatt természetesen azt értem, aminek a kezdetét az F struktúrában az

„m” pointer jelöli. És természetesen elképzelhető, hogy e két művelet mégis több lesz, mindegyik egy-egy műveletCSOPORT, mert ugye vannak mindenféle adattípusaink... Annyiból azonban jogos mégis csupán 2 műveletről beszélni, hogy nem szándékozunk más műveletet végezni a memóriával, mint a kiolvasás és a beléírás. Az olyasmi hogy „aritmetikai műveletek”, mint mondjuk az összeadás vagy a kivonás, feleslegesek. Ezeket úgy oldjuk meg, hogy a megfelelő adatot kiolvassuk a memóriából egy változóba, módosítjuk, majd visszairjuk.

Na már most, ezen írományom legeleje tájékán arról elmélkedtem, hogy a memóriában végsősoron minden adat egyszerűen egy bájt sorozat, tárolás szempontjából egyedül abban van különbség köztük, melyik típus hány bájt igényel. A típusok bájtméretét pedig nekünk a C nyelvben remekül meghatározza a „sizeof” operátor. Ennek megfelelően, a memóriából való adatkiolvasáshoz készítsünk 11 picike függvényt, a 11 adattípusunkra:

```
unsigned char felkialtojelc(F& f) { // A ! jel utáni memóriacímét olvassa be,
// A P mutató okvetlenül egy ! jelre kell mutasson!
V W;felkialtojel_elokeszit(f,W,sizeof(unsigned char));
return W.c[0];
}
// =====
signed char felkialtojelc(F& f) { // A ! jel utáni memóriacímét olvassa be,
// A P mutató okvetlenül egy ! jelre kell mutasson!
V W;felkialtojel_elokeszit(f,W,sizeof(signed char));
return W.C[0];
}
// =====
unsigned short int felkialtojeli(F& f) { // A ! jel utáni memóriacímét olvassa be,
V W;felkialtojel_elokeszit(f,W,sizeof(unsigned short int));
return W.i[0];
}
```

A többi nem másolom ide, illusztrációnak ennyi is elég. Látható, ezek mind ugyanazt a függvényt hívják meg, melynek paraméterül átadják azon típus méretét, mely típus önmaguk visszatérési értéke. Ezen függvény dolga az, hogy a referenciaként átadott W uniont (melynek felépítése pontosan ugyanolyan, mint a változóinké!) feltöltse a memóriából kiolvasott bájtokkal, az elejétől kezdve, annyi darab bájttal, ahány kell az input paraméterként megkapott típusméret szerint. Ezután a hívó függvény visszatér az union megfelelő mezőjével. Az említett segédfüggvény, meg az ő segédfüggvénye:

```
void felkialtojel_elokeszit(F& f,V& W, USC size) {USC i;USIL index;
k(f); // Előrelép a következő karakterre
index=ERTEKunsignedint(f); // Beolvassa a ! után következő aritmetikai kifejezést, ez lesz a memóriacím
indexell(f,index, size); // ellenőrzi, az index nem lépi-e túl az adatmemória méretét
for(i=0; i < size;i++) W.c[i]=f.m[index++];
}
// =====
void indexell(F& f,USIL index, USIL size) {
// ellenőrzi, az index nem lépi-e túl az adatmemória méretét
if((index+size)>f.mhossz) {L("Adatmemóriaindex-túlcsordulás! Pointer a programkódra: %lu",f.P);EXITFAILURE();}
}
```

Ezek után pedig ki kell egészítenünk az **ERTEKunsignedchar**, **ERTEKdouble**, stb függvényeinket, de mindegyiket csak 1 sorral, amint e példán bemutatom:

```
signed long long ERTEKsignedlonglong(F& f) {
.....
if(karakter==194) { return (signed long long)cimkeertekvisszaad(f); } // $ teszt vége
if(karakter=='!') return felkialtojelG(f); // Memóriában tárolt érték beolvasása
// Különbben itt valami számféleségnek kell állnia
.....
}
```

A beszúrt sort kézzel emeltem ki. Minden függvényünknel a megfelelő típusú függvényt kell meghívni természetesen: **felkialtojelc**, **felkialtojelD**, stb.

Ezek után persze égünk a vágytól, hogy kipróbáljuk amit alkottunk... De hm, izé, hoppá... még csak kiolvasni tudunk a memóriából, de beleírni nem! Vagy... vagy mégis?! Hát persze, hiszen van nekünk „Brainfuck” értelmezőnk! Nosza, mesterkedjük csak össze e programot:

```
M 30000 // Lefoglalunk 30 ezer bájt memóriát
/+      // Brainfuck értelmező bekapcsolva
>>>    // A pointert a memória 3-adik bájtjára állítjuk
+++++  // A memóriaértéket megnöveljük 5-re
/-      // Brainfuck értelmezés kikapcsolva

?c !3 / // Kiíratjuk a memória tartalmát unsigned char számként
?i !2 / // Kiíratjuk a memória tartalmát unsigned short int számként

vih.2 3 // A h változó 2-es unsigned short int mezőjébe 3-as értéket raktunk
?c !:i2@h / // Indirekció, kiíratom azon memóriacím tartalmát unsigned char -ként,
// amire a fent említett h változó mutat
-ih.2 1 // Egytel csökkentjük a h változó tartalmát
?i !:i2@h / // Indirekció, kiíratom azon memóriacím tartalmát unsigned short int -ként,
// amire a fent említett h változó mutat
/
```

Magyarázni a fentieket felesleges, tele van kommentekkel. Legfeljebb annyit teszünk hozzá, hogy a futási eredménye ez kell legyen:

```
5
1280
5
1280
```

és amiatt jön ki az 1280, mert amikor unsigned short int számként tekintjük a memóriát, akkor az alsó bájton van nulla, a felsőre kerül az 5-ös szám, márpedig $256 \cdot 5 = 1280$

Mostantól így módosult az „aritmetikai kifejezés” BNF leírása nálunk:

```
<aritmetikaikifejezes> ::= [{<WhiteSpace>}] (<Number> | <valtozo> | <Mem>)
```

aholis

```
<Mem> ::= "!" (unsigned int-re castolt)<aritmetikaikifejezes>
```

Adósok vagyunk még azon rutinokkal, melyek bepakolják a memóriába az adatokat. Ezeket nagyon egyszerűen intézzük el: olyan neveket kapnak majd, hogy !c, !i, !D stb, aholis a felkiáltójel utal arra, hogy memóriába írunk, a második karakter azt mondja meg, milyen típusú adatot teszünk be, s két paramétere lesz mindegyik függvényünknek: az első a cím, ahova be kell rakni, a második pedig a berakandó érték! Lássuk őket!

```
fuggveny felkialtojel_mindentipus; // Ezekre az utasításokra: !c, !C, !i, !I, !l, !L, !f, !g, !G, !d !D
.....
USC sizeof(USC valtozotipus) { // Visszaadja az adott típus méretét
switch(valtozotipus) { // Ellenőrzi, hogy a megengedett határon belül van-e
case 'c': return sizeof(unsigned char);break;
case 'C': return sizeof(signed char);break;
case 'i': return sizeof(unsigned short int);break;
case 'I': return sizeof(signed short int);break;
case 'l': return sizeof(unsigned int);break;
case 'L': return sizeof(signed int);break;
case 'f': return sizeof(float);break;
case 'g': return sizeof(unsigned long long);break;
case 'G': return sizeof(signed long long);break;
case 'd': return sizeof(double);break;
case 'D': return sizeof(long double);break;
} // switch vége
return 0;
}
.....
```

```

int felkialtojel_mindentipus(F& f) {unsigned int index;V W;USC size;
size=sizeof(f.a2);index=ERTEKunsignedint(f); // Beolvasta a memóriacím értékét, ahova az adat kerül majd
indexell(f,index, size); // ellenőrzi, az index nem lépi-e túl az adatmemória méretét
switch(f.a2) {
case 'c': W.c[0]=ERTEKunsignedchar(f);break;
case 'C': W.C[0]=ERTEKsignedchar(f);break;
case 'i': W.i[0]=ERTEKunsignedshortint(f);break;
case 'I': W.I[0]=ERTEKsignedshortint(f);break;
case 'l': W.l[0]=ERTEKunsignedint(f);break;
case 'L': W.L[0]=ERTEKsignedint(f);break;
case 'f': W.f[0]=ERTEKfloat(f);break;
case 'g': W.g[0]=ERTEKunsignedlonglong(f);break;
case 'G': W.G[0]=ERTEKsignedlonglong(f);break;
case 'd': W.d[0]=ERTEKdouble(f);break;
case 'D': W.D[0]=ERTEKlongdouble(f);break;
} // switch vége
register USC i;for(i=0; i < size;i++) f.m[index++]=W.c[i];
return 0;
}

```

Ez igazán nem volt nehéz... Próbáljuk ki:

```

M 30000 // Lefoglalunk 30 ezer bájt memóriát

vii.5 $fce2 // Ez decimálisan 64738. 2 bájtos érték: $fc=252, $e2=226
"Az i szám értéke: " ?i :5@i / // kiírjuk
!i$21 :5@i // Betesszük e számot a memória $21=33-as bájtjától kezdve
"A szám értéke: " ?i !33 / // kiírjuk
!i100 $fce2 // Betesszük e számot a 100-as címtől kezdve is
"A szám értéke: " ?i !100 / // kiírjuk
!i255 :5@i // Betesszük e számot a memória 255-ös bájtjától kezdve
"A szám értéke: " ?i !255 / // kiírjuk
!i4300 :5@i // Betesszük e számot a memória 4300-as bájtjától kezdve
"A szám értéke: " ?i !4300 / // kiírjuk
"Első bájt értéke: " ?c !4300 /
"Második bájt értéke: " ?c !4301 /
"Első bájt karaktere: " ? !4300 /
/

```

A mau progink futásának eredménye:

```

Az i szám értéke: 64738
A szám értéke: 64738
A szám értéke: 64738
A szám értéke: 64738
Első bájt értéke: 226
Második bájt értéke: 252
Első bájt karaktere:  

```

Megvagyunk tehát a memóriakezeléssel is.

10. fejezet: If, then, else...

Ez egy nagyon rövid fejezet lesz...

Az if, then, else utasítások (vagy valami más olyasmik amik hasonló funkciót látnak el) nélkül nemigen létezhet komolyabb célra használható programnyelv. Alapvetően azonban nem amiatt, mintha ezek azért lennének fontosak, mert elágazásokat valósítanak meg velük legtöbbször! Az elágazás lényegében nem más, mint egy ugróutasítás, az pedig már van nekünk: ez a » parancs a nyelvünkben. Az if, then, else arra kell, hogy FELTÉTELT értékeljen ki, hogy megmondja, bizonyos utasításokat most épp végre kell-e hajtani vagy sem. Az merőben más kérdés, hogy az az utasítás amit végre kell (vagy nem kell) hajtani, legtöbbször épp egy ugróutasítás. Lehetne azonban akármi más is, és rengeteg példa van is rá tényleg, hogy az valóban nem ugróutasítás!

A lényeg tehát az if, ami ki kell értékeljen egy feltételt. Ez végeredményben könnyű: döntsünk úgy, hogy az if után egyszerűen egy „aritmetikai kifejezés” kell álljon, ennek eredményét az if castolja valami „unsigned char” értékre, s ha ez nulla, a feltétel hamis. Ellenkező esetben igaz. Ezesetben pedig majd minden olyasmit ami tényleges összehasonlítás, a későbbiek során beépítünk majd az „aritmetikai kifejezés” kiértékelő rutinjainkba. Igen, épp arról van szó, hogy újra átírjuk majd őket... De az a következő fejezet tárgya lesz. Egyelőre a lényeg az, hogy az if kap valami számot, ami vagy nulla, vagy nem nulla. De mit is kezdjen vele? Hogyan lesz ebből elágazás vagy más akármi?

SEHOGY! Legalábbis, ami az if-et illeti. Felesleges ugyanis bonyolítani a dolgokat... Hanem csináljuk azt, hogy kibővítjük az F struktúrát eképpen:

```
unsigned char ifflag; // A legutolsó végrehajtott „if” utasítás eredményét tárolja. 1 ha az igaz volt, 0 ha hamis
```

Ezek után az if rutinunk csak ennyi:

```
int fuggveny_if(F& f) {if(ERTEKunsignedchar(f)) {f.ifflag=1;} else {f.ifflag=0;} return 0;}
```

Hát ez tényleg rövid...

Na és ennek miként vesszük hasznát? Hát, megcsináljuk a magunk „then” és „else” utasításait, s mert nálunk egy utasításnév maximum 2 karakter lehet, ezek neve legyen egyszerűen **T** és **E**. S a T csinálja azt, hogy leellenőri ezen ifflag flaget, s ha ez nem nulla, akkor semmit se csinál. Ha ellenben nulla, akkor... Nos, akkor ugyanazt csinálja mint a // utasításunk, azaz elugrik a következő sor elejére! Ez nyilván azzal jár, hogy ha a feltétel igaz volt, végrehajtodik a T utáni programrész. Ha ellenben hamis volt, az utasításvégrehajtás a következő sor elején folytatódik. Persze, oda is tehetünk nyugodtan egy újabb T utasítást, meg a következőbe is, akárhova akármennyit, mindaddig, míg egy újabb if parancsot végre nem hajt a programunk! Az E utasításunk meg pont ugyanaz mint a T, csak akkor ugrik, ha az ifflag értéke 1.

Íme a két rutin:

```
int fuggveny_else(F& f) {if(f.ifflag) {ujsorig(f);}return 0;}  
// -----  
int fuggveny_then(F& f) {if(f.ifflag==0) {ujsorig(f);}return 0;}
```

Természetesen a konstruktorban illik gondoskodni róla, hogy az ifflag értéke kezdetben garantáltan nulla legyen.

Rutinjaink tényleg pofonegyszerűek... Legfeljebb az szorul magyarázatra, miért nem teszem rögvést magát az aritmetikai kifejezés eredményének számító unsigned char értéket az ifflag változóba, miért tesztelem le előbb, mennyi az értéke, azaz, ha az érték nem nulla, akkor miért redukálom le azt mindössze 1-re!

Nos, bevallom, egyszerűen a „szépérzékemre hagyatkozom”. Nekem meggyőződés, hogy a programozás - ha helyesen végzik - a művészetekkel rokon foglalkozás (vagy hobby), s eképp egy program akkor jó, ha valamilyen értelemben „szép”. Ez mindenre vonatkozik: Ha szép a forráskód külalakja (bevallom, e téren nálam sajnos jelentősek a hiányosságok, sorry...), akkor például könnyebb azt megérteni, és karbantartani, bővíteni, fejleszteni, módosítani. Ha a program egyes

részei nagyon hasonló felépítésűek, s így valamiféle részleges „szimmetriát” mutatnak, vagy ha az egyes kisebb részek felépítése hasonló a nagyobb egységekéhez („fraktál-mintázat”), akkor pedig jó az esély rá hogy kellően „strukturált”, s így nemigen követtünk el komoly hibát a logikai felépítésben, s ez szintén segít a megértésében és fejlesztésében is. Az is a szépséget fokozza, ha a fordítás során nem kapunk semmiféle „warning” üzenetet. Az a minimum, hogy kigyomláljunk belőle minden „nem használt változót” például, mert felesleges szemét ne legyen a forráskódban.

Na most, e gondolatok úgy kapcsolódnak e témakörhöz, hogy az ifflag változó lényegében egy logikai értéket kell tároljon: IGAZ vagy HAMIS értéket. Más programnyelvekben van is külön logikai adattípus, s az ténylegesen csak két értéket vehet fel. Mint a bit: az is csak nulla vagy egy lehet! Emiatt cselekedtem én is így az if utasításomban. Mindjárt be is mutatom, ennek miféle haszna van a számunkra!

Ugye, van nekünk az a szubrutinunk, ami a fontos rendszerváltozók értékét adja vissza. Illik, hogy ezt kibővítsük olyan lehetőséggel, ami az ifflag értékét mondja meg nekünk. Íme:

```
USIL systemvariable(F& f) { // Valamely fontos mau rendszerváltozó értékét adja vissza
switch(k1) {
// Felhasznált karakterek:
// 012  abcd fgh i lm p v x  BC EFGH IJ L P V X
.....
case '0': value=(USIL)(f.ifflag+'0');break;
case 'F': value=f.ifflag; break;
```

Mint látható fent, 2 variációban is lekérdezhetjük e fontos flaget: A **?0** egy konkrét KARAKTERT ad vissza, a "0" vagy az "1" karaktert, ahol természetesen az "1" felel meg az IGAZ értéknek, vagy lekérdezhetjük ezt a **?F** szimbólummal is, ami egyszerűen visszaadja e flag számkódját.

Fontos lehet, hogy közvetlenül is befolyásolhassuk e flag értékét. Erre megfelelne egyszerűen annyi, hogy:

```
if 0
Vagy
if 1
```

de „felesleges atombombával löni verébre”, azaz minek hívjunk meg ilyesminek a kedvéért egy komplett aritmetikai kifejezés kiértékelő rutint?! Csináljunk inkább 3 pici utasítást:

```
int per0(F& f) {f.ifflag=0;return 0;}
// -----
int per1(F& f) {f.ifflag=1;return 0;}
// -----
int perF(F& f) {if(f.ifflag) {f.ifflag=0;} else {f.ifflag=1;} return 0;}
```

A **/0** illetve **/1** nullára illetve 1-re állítja a flagunkat, a **/F** pedig invertálja.

Mindez egy példán bemutatva:

```
if 3 T "Igaz" /
? ?0 /
"ifflag számértéke: " ?c ?F /
T "Ez is igaz" /
```

```

E "Nem igaz" /
T »$ug
"Ezt nem írom ki" /
$ug "Ide ugrott" /

/0
T "1. ifflag=0, nem írom ki ezt a sort (THEN)" /
/1
T "2. ifflag=1, kiírom ezt a sort (THEN)" /
/0
E "3. ifflag=0, kiírom ezt a sort (ELSE)" /
/1
E "4. ifflag=1, nem írom ki ezt a sort (ELSE)" /
/F
T "5. Most ifflag=0, nem írom ki e sort (THEN)" /
E "6. Most ifflag=0, kiírom e sort (ELSE)" /
/F
T "7. Most ifflag=1, kiírom e sort (THEN)" /
E "8. Most ifflag=1, nem írom ki e sort (ELSE)" /

```

A progí futásának eredménye:

```

Igaz
1
ifflag számértéke: 1
Ez is igaz
Ide ugrott
2. ifflag=1, kiírom ezt a sort (THEN)
3. ifflag=0, kiírom ezt a sort (ELSE)
6. Most ifflag=0, kiírom e sort (ELSE)
7. Most ifflag=1, kiírom e sort (THEN)

```

11. fejezet: Syntax error

Ez még az előzőnél is rövidebb fejezet lesz...

Eddig azt a gyakorlatot követtük, hogy azon tömbjeinkbe melyeket létrehoztunk az utasításokat megvalósító függvények címeinek, a fel nem használt üres helyekre egyszerűen beírtuk helykitöltőnek a „**semmi**” nevű függvényünket, ami semmit se csinált. Ez megfelelt a fejlesztés első időszakában, „hamari” megoldásként, hogy ne tököljünk sokat minden apró baromságon, s elinduljunk valamerre, eljussunk valameddig. Most már azonban meglehetősen sok utasításunk van, s az aritmetikai kifejezés kiértékelő függvényeink szintaxisa is egyre bonyolultabbá válik. Rengeteg fura hibát szülhet, ha a forráskódba beleírunk valami karaktert, ami nem oda való, s nem jelez hibát az interpreter, hanem csak átugorja azt. Aztán meg keressük majd a hibát az interpreterünk C/C++ nyelvű forráskódjában, azt híve hogy valami rutint rosszul írtunk meg, holott esetleg ott minden jó, csak a mau nyelvű programban írtunk el valamit... Azaz, az összes eddigi tömbünkben le kell cserélni a „**semmi**” függvényt egy olyanra, ami hibát jelez ha azt a tokenet hívjuk meg parancsként. A kivétel ahol megmaradhat a „**semmi**”, azok a whitespace-nek fenntartott karakterek.

Ilyen hibajelző függvényből két fajtát is kell csinálnunk, mert kell egy - célszerűen „**syntaxerror**” néven - ami akkor kiabál ha olyan karaktert használunk aminek semmiféle jelentése nincs még, s kell egy másik, „**syntaxerror2**” néven, ami akkor szól, ha kétbájtos hívást hajtunk végre, ahol az első bájttnak van ugyan létrehozva tömb, de azon belül a második bájt már értelmezhetetlen. A „**syntaxerror**” függvénnyel töltjük fel a „**fuggvenyek**” tömb megfelelő helyeit, a másikkal az összes többi tömb üres helyeit (de kivéve a **FUGGVENYTOMB** tömböt mert az más célra való)!

A két függvény:

```
int syntaxerror(F& f) { // Érvénytelen utasítástoken
L("Syntax error: Érvénytelen 1 karakteres utasítástoken!\n")
"Nincs utasítás meghatározva erre a bájtra!\n"
"Kód= %u; Pozíció a forráskódban: %lu",f.a,f.P-1);
EXITFAILURE();
return 0;}
// =====
int syntaxerror2(F& f) { // Érvénytelen 2 karakteres utasítástoken
L("Syntax error: Érvénytelen 2 karakteres utasítástoken!\n")
"Utasításcsoport számkódja (az első token, aminek létezik a csoportja) = %u\n"
"A második token számkódja, amihez nincs meghatározva utasítás =: %u\n"
"Pozíció a forráskódban: %lu",f.a,f.a2,f.P-1);
EXITFAILURE();
return 0;}
```

Kicsit átírtam a „kérdőjel” rutint is, hogy igazodjék az eddig bevezetett változtatásokhoz:

```
int kerdojel(F& f) {
switch(f.a2) {
case 'c': printf("%u",ERTEKunsignedchar(f));break;
case 'C': printf("%d",ERTEKsignedchar(f));break;
case 'i': printf("%u",ERTEKunsignedshortint(f));break;
case 'I': printf("%d",ERTEKsignedshortint(f));break;
case 'l': printf("%u",ERTEKunsignedint(f));break;
case 'L': printf("%d",ERTEKsignedint(f));break;
case 'f': printf("%f",ERTEKfloat(f));break;
case 'g': printf("%llu",ERTEKunsignedlonglong(f));break;
case 'G': printf("%lld",ERTEKsignedlonglong(f));break;
case 'd': printf("%g",ERTEKdouble(f));break;
case 'D': printf("%Lg",ERTEKlongdouble(f));break;
case ' ': nenspace(f);if(f.p[f.P]=='\n') {k(f);return textkiir(f);} else {printf("%c",ERTEKunsignedchar(f));} break;
default: return 3;break;
} // switch vége
return 0;
}
```

Hát e fejezet csak ennyi volt.

12. fejezet: Az „aritmetikai kifejezés” bővítése operátorokkal

Már kezd a nyelvünk használható lenni. Igazából már megoldhatunk vele minden feladatot, legfeljebb 2 függvényt kéne írunk hozzá: egyik ami beolvas egy fájlból valami adatot, a másik ami kiírja belé. Efféle nyalánkságokat azonban későbbre halasztunk, előbb ismerjük el, hogy bár nyelvünk tényleg használható, de számos „gyermekbetegséggel” küzd! Mindenekelőtt: bűnronda, mert logikátlan! Amiatt az, mert hát micsoda dolog is az, hogy az értékadó utasítás bal oldalán úgy határozunk meg egy változót, hogy

vcA.3

de bezzeg ha pontosan ugyanezen változó a „jobb oldalon” szerepel, azaz ha őt adjuk értéknek, akkor már így kell írjuk:

:c.3@A

???????!!!!!!

Ez bizony tényleg logikátlan, és ha semmi másért nem is, de már ezen logikátlanság miatt is iszonyú ronda! Sőt, mi az hogy ronda — kimondottan OCSMÁNY!

Oké, persze, kijelentettem már a legelején, hogy kb mindent feláldozunk e nyelv esetén a hatékonyság oltárán, na de hát azért mégis... Talán csak meg lehetne ezt oldani szebben, de úgy, hogy hatékonyak maradjunk!

Mindez persze érthető, nem véletlen hogy ilyenné alakult - abból fakad, hogy nyelvünk kialakításának nem úgy fogtunk neki, mint a „nagy fejek”, akik általában tervezni szokták az ismertebb programnyelveket. Azok kiindulnak valami elméleti koncepcióból, ami azt jelenti hogy a korábban ismertetett BNF formulával kezdik, vagy az legalábbis nagyon hamar felbukkan náluk már a tervezés korai stádiumában, aztán azt szépen lépésről-lépésre leprogramozzák. Ennek kétségtelenül megvan az a nem elhanyagolható előnye, hogy amennyiben a pofa aki teveszi a nyelvet, képes elgondolni valami logikus szintaxist, azt meg is valósítja, s a nyelve olyan lesz, amilyennek ő azt kigondolta a kezdetektől fogva. Tehát ha szépnek gondolta ki a szintaxist, akkor az egy szép nyelv lesz!

Sajnos azonban, e módszernek hátránya is van. Ez pedig az, hogy ezen módszer által a nyelv valamely előre kigondolt elméleti célt tart szem előtt, természetesen maga a szintaxis is ezt szolgálja, s emiatt aztán a kód ami ezt megvalósítja, szerényen fogalmazva is nem lesz optimális a sebesség, méret, futásidő sőt akár a karbantarthatóság szempontjából! Az egész módszer arra hasonlít, mint amikor valaki papíron, a szobájában, 10 ezer kilométerre a leendő helyszíntől megtervez egy szép katedrálist, aztán azt majd valahol másutt a csapat meg kell építse épp olyanná. És lehet hogy ehhez ott egy csomó eszköz meg nyersanyag hiányzik. Akkor az egész projekt leáll vagy késik, és iszonyatos erőfeszítésbe telik a megfelelő alapanyagok és szerszámok/munkagépek beszerzése, meg persze temérdek pénzbe is!

Az a módszer ellenben amit eddig mi követtünk, más. Mi abból indultunk ki, amit „helyben találtunk”: abból, amit a számítógép „alpból tud”, s azt vizsgáltuk, mit lehet ebből kihozni úgy, hogy ne nagyon pocskoljuk a memóriaterületet illetve a műveletvégző sebességet. Ez arra hasonlít a fenti példánál maradva, mint amikor a Nagyfőnök nem kezdi el odahaza tervezgetni a katedrálist, hanem csak telefonál az alkalmazottainak: „Hé fiúk, nézzetek körül, miféle anyagok meg szerszámok vannak nagyon olcsón azon a környéken, és építsetek belőle nekem egy BAROMINAGY épületet, minél nagyobb, minél szebbet, de semmiképp se legyen nagyon drága”!

És most annál a pontnál tartunk, amikor a Nagyfőnök ellátogat az épülethez, s azt mondja:

—„Hm, látom már hogy jó lesz ez, de néhány apróságon változtatni illene azért! Például, itt ezt a randa lépcsőt építsetek át lejtőre, hogy a kerekesszékekben élők is be tudjanak jutni az épületbe”!

Térjünk vissza a hasonlatok ékes mezejéről a valóságba! Változtatnunk illene tehát a szintaxison. Előbb azonban gondolkodjunk el egy cseppet a továbbiakon! Azon, mi más szépséghibák is vannak a nyelvünkben. Mert oké, most kicsit átírunk benne ezt-azt (nem kell megijedni, nem lesz ez nagyon vészes, megnyugtatók előre mindenkit!) de azért efféle műtéteket nem szeretnénk gyakran elvégezni újra meg újra a jövőben vele! (Szeretünk ugye lustának megmaradni...!)

A legislegnagyobb gond talán az, hogy amit mi „aritmetikai kifejezés” alatt értünk, az igazából nagyon nem az, amit megszoktunk e fogalom alatt a programnyelvekben eddig. Gondoljuk csak el: a mi „kifejezésünk” valójában nem kifejezés, csak EGYETLEN valami! Elég sok mindenféle bizbasz lehet ez az egyetlen valami,

az igaz, mert lehet változó, vagy konstans, vagy címke, de ezek elég korlátozott mértékben ágyazhatóak csak egymásba. Arról például szó se lehet, hogy műveleteket végezzünk három - mondjuk A, B és C nevű - kifejezés közt, hogy mondjuk $A + B * C$

Na és hát ez igencsak ciki! Ezen változtatnunk kell. A BNF forma szerint ezt úgy írhatnánk (leegyszerűsítve), hogy:

```
<aritmetikaikifejezes> ::= (megadjuk hogy az micsoda) [{<operator> <aritmetikaikifejezes>}]
```

Azaz, egy aritmetikai kifejezés minden olyan akármilyen, ami egy aritmetikai kifejezéssel kezdődik, s azt esetleg egy „operátor” nevű valami követi, s ezután egy újabb aritmetikai kifejezés áll. Fontos kihangsúlyozni, hogy az aritmetikai kifejezés után nem muszáj hogy álljon operátor, de HA áll, akkor azt okvetlenül kell kövesse egy másik aritmetikai kifejezés! Az interpreterünk épp onnan tudja hogy vége van az aritmetikai kifejezésnek, hogy azt nem követi semmiféle érvényes operátor!

Az „operátor”, az természetesen épp azt jelenti, amit szokásosan a „műveleti jel” alatt értünk. Például a „+” jel, az egy operátor: az összeadás operátora.

Igazából ezt nem lenne nagyon nehéz megvalósítanunk, mert gondoljunk csak bele, semmi az egész: beolvas valami aritmetikai kifejezést, aztán megnézi, a következő karakter holmi operátor-e. Ha nem, visszaadja az előbb beolvasott aritmetikai kifejezés eredményét. Ha ellenben operátor, beolvassa a következő kifejezést is (azaz meghívja önmagát rekurzívan), s az előző meg a mostani érték közt elvégzi az operátor által meghatározott műveletet, majd visszaadja az így kapott eredményt.

Mindez tényleg „nem nagy durranás” - sajnos azonban ez csak a „lehetséges világok legjobbjában” működik ilyen szépen és egyszerűen. Ne feledjük ugyanis, hogy nekünk nem csak 1 -féle adattípusunk van, hanem 11, na és hát ugye, ennek megfelelően 11 különböző „aritmetikai kifejezés kiértékelő” függvényünk! Ez eddig úgy működött, hogy minden parancs eleve tudta, ő miféle típusú paramétert vár, és az annak megfelelő függvényt hívta meg. Például az, amely egy változó indexét óhajtotta kiszámítani, az az **ERTEKunsignedchar** nevűt hívta meg.

Abban a pillanatban azonban hogy e függvény önmagát rekurzívan hívogatni óhajtja, felmerül a kérdés, honnan tudja, a következő része a kifejezésnek miféle típusú is, azaz melyik kifejezéskiértékelő függvényt kell meghívnia! Mert az rendben van hogy akármelyiket is hívja meg, az eredményt majd castolja olyan típusúvá amilyen ő maga a hívó, de nem mindegy, a castolás előtt milyen típusúként érkezik az meg hozzá!

Mindez igaz a változóinkra is. Végősoron minden változónk igazából 11 típusú lehet, ő maga is egy miniatűr aritmetikai kifejezés, és cseppet se mindegy, melyik típusú értékét adja vissza!

Minderre a következőt találtam ki, ami a szintaxist is meghatározza:

Egy változó, az mindig olyan típusú értéket ad vissza, amilyen aritmetikai kifejezés kiértékelő függvény hívja őt. Az hogy melyik mezőjének értékét adja vissza, attól függ, ezt konkrétan specifikáltuk-e neki:

@v — Ez a „v” változónak azon értékét adja vissza, mely abban a sorszámú mezőben található, amit a megfelelő (típustól függő) vindex tömb tartalmaz a „v”-edik helyen.

:3@v — Ez a „v” változónak konkrétan épp a 3-as mezőjén található értékét adja vissza.

A „3” és a „v” a fenti példákban csak afféle „jelkép”, valójában mindegyik helyén állhat tetszőleges aritmetikai kifejezés.

Mindkét fenti példa önmagában véve is egy aritmetikai kifejezés. Egy aritmetikai kifejezés konkrét értéke mindig a típusától függ. A típust mindig előírja az, hogy melyik kiértékelő függvénnyel hívjuk meg, de ez csak a „default”, az „alapértelmezett” típus. Ezen mi ugyanis módosíthatunk, ha konkrétan előírjuk a kifejezés típusát, erre szolgál a # jel:

#i@z — a „z” változó azon értékét jelenti, mely a vindexi tömb z-edik elemében található érték által meghatározott mezőn van. A #i ugyanis unsigned short int típust jelent.

#i:3@z — Ez a „z” változó unsigned short int értékei közül a 3-as mezőn levőt jelenti.

Természetesen a „3” és a „z” helyén itt is állhat tetszőleges aritmetikai kifejezés, de mindegyik a végén unsigned char értékre lesz castolva. Az „i” helyén azonban nem állhat aritmetikai kifejezés - az egy konkrét karakter kell legyen! A karakterek jelentései:

Karakter	jelentés	az adattípus bájtmérete (sizeof)	A maximális érvényes index
c	unsigned char	1	15
C	signed char	1	15
i	unsigned short int	2	7
I	signed short int	2	7
l	unsigned int	4	3
L	signed int	4	3
f	float	4	3
g	unsigned long long	8	1
G	signed long long	8	1
d	double	8	1
D	long double	16	nem vesz figyelembe indexet

E „#” jel nemcsak változókra érvényes - érvényes a teljes, utána következő aritmetikai kifejezésre! Ez tehát tulajdonképpen egy úgynevezett „egyoperandusú operátor”, más néven „unáris operátor”. Tulajdonképpen semmi mást nem csinál, mint a „castolást” végzi el: meghívja a kifejezés hátralevő részére a megfelelő típusú aritmetikai kifejezés kiértékelő rutint.

A fentiek miatt, az aritmetikai kifejezések alapértelmezés szerint jobbról balra értékelődnek ki. NEM TÉVEDÉS! Az utasításfeldolgozás természetesen balról jobbra halad továbbra is, de mert a kifejezéskiértékelő rutinok minden operátor-nál önmagukat rekurzívan meghívják, emiatt legelsőként a legutolsó operátor lesz végrehajtva. Azaz:

$6*3+10$ értéke 78 lesz, nem 28, ahogy azt a matematikában általában megszoktuk. Ezt azonban előírhatjuk zárójelekkel:
 $(6*3)+10$

Természetesen akárhány zárójel is egymásba ágyazható.

Ami a precedenciaszabályokat illeti, nem sok értelmét látom annak, hogy ilyesminek a megvalósításával görcsöljek. Akad pár más programnyelv is, ha nem is a legismertebbek, melyekben nincsenek precedenciaszabályok, sőt, még ilyen zsebszámológépek is vannak. Továbbá, még talán rá is szánnám magamat erre, ha csak arról lenne szó, hogy a „+” és „-” jelek alacsonyabb precedenciájúak mint a „*” és a „/”, de hát ugyebár vélhető hogy lesz nekünk e négyen kívül még egy egész rakás más operátorunk is, no és ezek közt cseppet se egyszerű meghatározni valami logikus precedenciát! Ott van példának okáért a C nyelv is, amin épp írjuk az interpretert, e nyelv tele van operátorokkal, és ezeknek rengeteg szintje létezik. Ember legyen a talpán, aki fejből vágja, melyiknek épp mi a precedenciaszintje! Még én magam is úgy vagyok vele hogy nagy ívben tojok az egészre, és ha kicsit is bonyolultabb esetről van szó, egyszerűen úgy zárójelezem a kifejezést, hogy az teljesen egyértelmű legyen. Minden más megoldás kizárólag arra jó, hogy rengeteg tévedésre adjon lehetőséget!

Arról már nem is beszélve, hogy más programnyelvekben ugyanazon operátorok precedenciája teljesen eltérő is lehet... Gyakorlatilag annyi precedenciaszabály létezik, ahány programnyelv. Teljesen felesleges e zűrzavart fokoznunk azzal, hogy mi bevezetünk valami megint újabb rendszert. Mi a szösznek, hogy még mi magunk is belekeveredjünk?!

Nálunk, a mi mau nyelvünkben ilyesmi nem fordulhat elő. Egyetlen szabály van csak, amit emiatt rém könnyű megjegyezni: MINDIG jobbról balra értékelődik ki a kifejezés, s bármi más sorrendet óhajtunk, azt zárójelezéssel érhetjük el! Ez világos és egyértelmű.

Ajánlatos azért észben tartani pár dolgot ennek ellenére is. Tegyük fel, egy karakteres változóhoz (a nulladik mezőjéhez), aminek „d” a neve, hozzá akarunk adni 1-et, s az eredményt kiírni:

?c :0@d+1

Nos, ez esetben nem azt kapjuk eredményül, ami eggyel nagyobb annál ami a „d” változó nulladik karakteres mezőjében van, hanem azt kapjuk eredményül, ami az „e” változó nulladik karakteres mezőjében van! Ugyanis a @ jel után ő egy aritmetikai kifejezést vár el, s a „d+1” egy teljesen érvényes aritmetikai kifejezés, aholis a „d” ASCII kódjához szépen hozzáad majd 1-et, eredményül „e”-t kap, s ennek értékét adja vissza!

Ha tényleg a „d” változó nulladik rekeszének tartalmához akarunk egyet hozzáadni, zárójeleznünk kell:

```
?c (:0@d)+1
```

Ugyanis zárójel nélkül ez annak felel meg, mintha így lenne zárójelezve:

```
?c :0@(d+1)
```

Ami a balérték szintaxisát illeti, megszabadulunk a @c, @d, stb utasításainktól, meg a vc, vi, stb utasításoktól is. Nem kellene, nem szép a szintaxisuk. Ehelyett csinálunk egyetlen utasítást, =# néven, ez csinál majd mindent. Ennek az a szabálya, hogy ez után okvetlenül egy típusazonosító karakternek kell következni, nem előzheti meg azt whitespace. Azért nem, mert a castolásra használt # karakter is elvárja hogy őt közvetlenül kövesse a típusjelző karakter. Simán megcsinálhattam volna, hogy lehessen akárhány whitespace is mindkét esetben a # után, de direkt nem tettem ezt, mert messze sokkal olvashatóbb a mau forráskód, ha a # jelek után mindig ott van közvetlenül a típusjelölő.

Az új utasításunk bemutatása egy példán:

=#c@d 7 — Ez a „d” változó aktuális sorszámú (a vindexc tömb „d”-edik eleme által meghatározott) unsigned char mezőjébe tölt 7-es értéket.

=#c:3@d 7 — Ez a „d” változó 3-as unsigned char mezőjébe tölt 7-es értéket.

A fenti példákban a „c” egy konkrét, a forráskódba beleírt karakter kell legyen, ellenben a „3”, a „d” és a „7” helyén tetszőleges aritmetikai kifejezés állhat.

Nézzük meg, milyen szép szimmetrikus most egy efféle értékadás, ha változó van a jobboldalon:

```
=#c:3@d :6@e
```

Ez az előbb említett „d” változó 3-as unsigned char mezőjébe betölti az „e” változó 6-os unsigned char mezőjének értékét. Nem kell külön jelölni hogy az „e” esetén is unsigned char értékről van szó, mert az =# utasításunk úgyis azt kéri majd az „e” változótól, hiszen a legelső karakter az =# után a „c”. Megadhatjuk azonban, bár felesleges:

```
=#c:3@d #c:6@e
```

Látható, a szintaxis teljesen egyforma a bal és a jobboldalon.

Megtehetjük azonban, hogy castolt adatot rakunk valahova:

```
=#c:3@d #f:2@e
```

A fenti példa azt mondja, hogy szedje ki az „e” változó 2-es float típusú mezőjének tartalmát, majd ezt rakja be a „d” változó 3-as unsigned char mezőjébe. Berakás előtt természetesen castolja a floatként érkező számot unsigned char értéké.

Az ezt megvalósító függvényünk:

```
int egyenlokereszt(F& f) {USC index;USC unionindex;USC változotípus;
változotípus=vtype(f); // rögtön a következő karakter
nemspace(f);kettospontteszt(f,unionindex,változotípus); // változóindex beolvasása, ha meg van adva
nemspace(f);
if(f.p[f.P]!='@') return SERROR(f,3); // Ha az aktuális feldolgozandó karakter nem kukac, syntax error
k(f); // Ugrás a következő karakterre
index=ERTEKunsignedchar(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat
switch(változotípus) {
case 'c': f.v[index].c[unionindex]=ERTEKunsignedchar(f);break;
case 'C': f.v[index].C[unionindex]=ERTEKsignedchar(f);break;
case 'i': f.v[index].i[unionindex]=ERTEKunsignedshortint(f);break;
case 'I': f.v[index].I[unionindex]=ERTEKsignedshortint(f);break;
case 'l': f.v[index].l[unionindex]=ERTEKunsignedint(f);break;
case 'L': f.v[index].L[unionindex]=ERTEKsignedint(f);break;
case 'f': f.v[index].f[unionindex]=ERTEKfloat(f);break;
case 'g': f.v[index].g[unionindex]=ERTEKunsignedlonglong(f);break;
case 'G': f.v[index].G[unionindex]=ERTEKsignedlonglong(f);break;
case 'd': f.v[index].d[unionindex]=ERTEKdouble(f);break;
case 'D': f.v[index].D=ERTEKlongdouble(f);break;
} // switch vége
return 0;
}
// -----
USC kettospontteszt(F& f,USC &unionindex,USC &változotípus) {USC karakter;
karakter=f.p[f.P]; // Az aktuális feldolgozandó karakter
if(karakter=='') {k(f);unionindex=ERTEKunsignedchar(f); // Beolvassa az union indexét
indexerrorMessage(f,változohatar(változotípus),unionindex); // Ellenőrzi, hogy a megengedett határon belül van-e
nemspace(f);karakter=f.p[f.P]; // kettőspont-teszt vége
return karakter;
}
```

Hasonlóképp búcsút mondunk a +c, +i, *c, *d stb függvényeinknek, s lesz mindegyik műveletre egy olyan, hogy +#, -#, *#, /#. Ezek mindegyike a fenti == mintájára készül, felesleges lenne bemutatni őket, mert teljesen megegyeznek azzal, csak annyi a különbség, hogy a switch blokkban az „=” jel helyett a megfelelő műveleti jel van, azaz +=, *=, -= vagy /=. Tehát például e sor helyett:

```
case 'C': f.v[index].C[unionindex]=ERTEKsignedchar(f);break;
Ez:
case 'C': f.v[index].C[unionindex]+=ERTEKsignedchar(f);break;
```

Ennél érdekesebbek az új aritmetikai kifejezés kiértékelő függvényeink! Ebből is azonban csak az egyiket közlöm itt:

```
unsigned char ERTEKunsignedchar(F& f) { // A következő „Aritmetikai kifejezés” értékét adja vissza unsigned char
számként.
USC karakter;USC változotípus;USC unionindex;unsigned char A;
unionindex=255;változotípus='c'; // Alapértelmezett index és típus
nemspace(f);karakter=f.p[f.P];
if(karakter=='(') {k(f);A= ERTEKunsignedchar(f);nemspace(f);karakter=f.p[f.P];
if(karakter!=')') {L("Hiányzó csukózárájel!");return SERROR(f,3);}
k(f);goto ERTEK_c_OPERATOR;
} // Nyitózárájel-teszt vége
if(karakter=='#') {változotípus=k(f);k(f);
switch(változotípus) {
case 'c': A= (unsigned char)ERTEKunsignedchar(f); break;case 'C': A= (unsigned char)ERTEKsignedchar(f); break;
case 'i': A= (unsigned char)ERTEKunsignedshortint(f);break;case 'I': A= (unsigned char)ERTEKsignedshortint(f); break;
case 'l': A= (unsigned char)ERTEKunsignedint(f); break;case 'L': A= (unsigned char)ERTEKsignedint(f); break;
case 'f': A= (unsigned char)ERTEKfloat(f); break;case 'g': A= (unsigned char)ERTEKunsignedlonglong(f);break;
case 'G': A= (unsigned char)ERTEKsignedlonglong(f); break;case 'd': A= (unsigned char)ERTEKdouble(f); break;
case 'D': A= (unsigned char)ERTEKlongdouble(f); break;default: return SERROR(f,3);break;
} // switch vége
goto ERTEK_c_OPERATOR;} // A # teszt vége
karakter=kettospontteszt(f,unionindex,változotípus); // Az aktuális feldolgozandó karakter
if(karakter=='@') {switch(változotípus) {
case 'c': A= (unsigned char)vc(f,unionindex);break;case 'C': A= (unsigned char)vC(f,unionindex);break;
case 'i': A= (unsigned char)vi(f,unionindex);break;case 'I': A= (unsigned char)vI(f,unionindex);break;
case 'l': A= (unsigned char)vl(f,unionindex);break;case 'L': A= (unsigned char)vL(f,unionindex);break;
case 'f': A= (unsigned char)vf(f,unionindex);break;case 'g': A= (unsigned char)vg(f,unionindex);break;
```

```

case 'G': A= (unsigned char)vG(f,unionindex);break;case 'd': A= (unsigned char)vd(f,unionindex);break;
case 'D': A= (unsigned char)vd(f);break;
} // switch vége
goto ERTEK_c_OPERATOR;} // változóbeolvasás vége
switch(karakter) {
case '?': A= (unsigned char)systemvariable(f);goto ERTEK_c_OPERATOR; break; // Rendszerváltozó tartalmát kell
visszaadni
case 194: A= (unsigned char)cimkeertekvisszaad(f);goto ERTEK_c_OPERATOR; break; // $ teszt vége
case '!': A= felkialtojelc(f);goto ERTEK_c_OPERATOR; break; // Memóriában tárolt érték beolvasása
} // switch karakter vége
A= (unsigned char)number(f.p,f.P,f.phossz);// Mert különben itt valami számféleségnek kell állnia

ERTEK_c_OPERATOR:
nemspace(f); // ugrás a következő értékes karakterre
karakter=f.p[f.P]; // P most a beolvasott műveleti jelre mutat
A=Operatorc(A,karakter,f);
return A;
}

```

A hibajelző függvény:

```

int ERROR(F& f, int errorcode) {L("Syntax error: Hibakód: %u, token: %u, cím:
%lu",errorcode,f.a,f.P);EXITFAILURE();return 3;}

```

A megfelelő műveletet végrehajtó operátorfüggvény:

```

unsigned char Operatorc(unsigned char A,USC muveletijel,F& f) {
switch(muveletijel) {
case '+': k(f);return (A + ERTEKunsignedchar(f));break;
case '-': k(f);return (A - ERTEKunsignedchar(f));break;
case '*': k(f);return (A * ERTEKunsignedchar(f));break;
case '/': k(f);return (A / ERTEKunsignedchar(f));break;
} // switch vége
return A;
}

```

E függvényből természetesen minden típusra van különböző, azaz létezik OperatorC, Operatori, Operatord, stb is, és mindegyik ilyen függvényben sajnos annyi case-sor kell legyen, ahány különböző műveletet megvalósítunk, azaz ahány kétoperandusú operátorunk van a programnyelvünkben.

Meg kell még említeni, hogy ezek után nem működik majd minden esetben az üres sor kiírására szolgáló egyetlen „/” jeltől álló utasításunk! Ez ugyanis az interpreterünk számára immár operátorként (is) értelmezve van. A helyes megoldás az, ha a használata előtt tesztünk elé egy pontosvesszőt, így:

```

; /

```

Fontos észben tartani, hogy a pontosvesszőt legalább 1 darab whitespace karakter (például szóköz) kell kövesse, különben egyszerűen nem veszi majd figyelembe a pontosvesszőt követő per-jelet...

Azt is tudomásul kell vennünk, hogy ezentúl a „Brainfuck” programjainkban picit másképp kell megadnunk a legelső utasítást, ami a memóriát foglalja le. Nem elég annyi, hogy mondjuk:

```

M 30000

```

Hanem ezt így kell írunk:

```

M 30000;

```

amiatt, mert ezután többnyire egy

```

/+

```

Utasítás áll, ami bekapcsolja a „Brainfuck” értelmezést ugyebár. Csakhogy az egy „/” jellel kezdődik... ezért ezt operátornak értelmezné az interpreter, s megkísérelné beolvasni a következő aritmetikai kifejezést. Az feltételezhetően „+” jellel kezdődne, mert a „Brainfuck” progik teli vannak ezzel a jellel. E jelet numerikus

konstans bevezetésének tartaná a szegény interpreterünk, s átugorja. A következő plusz jelet is, meg az azutánit is, a mínusz jeleket is, míg végül eljutna valami olyan jelhez amit már nem tud értelmezni. S ekkor hibajelzéssel leáll.

Hasonló igaz a `//` és a `/*` megjegyzésjeleinkre is.

Bár ez olyasmi amivel „együtt lehet élni”, de e problémák legnagyobb részét könnyedén kiküszöbölhetjük. S ennek érdekében még csak nem is kell sok mindent tennünk, csak azt, ami amúgy is elkerülhetetlen: képessé tenni interpreterünket a 2 karakteres operátorok kezelésére! Ezek a jólismert `==`, `!=`, `<=`, `>=` operátorok, s javasolom vezessük be ezek mellé a `<>` jelet is, ami csinálja ugyanazt, mint a `!=`.

Hogy ez megvalósuljon, mindenekelőtt készítsük el ezt a segédfüggvényt:

```
unsigned char Operator2kar(F& f) { // 2 karakterből álló operátor kódjának meghatározása
USC a;USC kod;USC b; // b a második operátorkarakter
a=f.p[f.P]; // Az első operátorkarakter
kod=a; switch(a) { case '+': case '-': case '*': break; // Egykarakteres utasítástokenek, kódjuk az eredeti marad
default : b=k(f);
if(a=='/') {
if(b=='/') {f.P--;kod=0;goto Operator2karVege;} // A // utasítás nem operátor
if(b=='\n') {f.P--;kod=0;goto Operator2karVege;} // A \n utasítás nem operátor
if(b=='\t') {f.P--;kod=0;goto Operator2karVege;} // A \t utasítás nem operátor
if(b=='-') {f.P--;kod=0;goto Operator2karVege;} // A - utasítás nem operátor
if(b=='*') {f.P--;kod=0;goto Operator2karVege;} // A * utasítás nem operátor
f.P--;goto Operator2karVege;} // if a=='/' vége
if(a=='=') {
if(b=='=') {kod=128;goto Operator2karVege;} // Az == operátor kódja 128
f.P--;goto Operator2karVege;} // if a=='=' vége
if(a=='!') {
if(b=='=') {kod=129;goto Operator2karVege;} // Az != operátor kódja 129
f.P--;goto Operator2karVege;} // if a=='!' vége
if(a=='<') {
if(b=='>') {kod=129;goto Operator2karVege;} // Az <> operátor kódja 129, mert ugyanaz mint a !=
if(b=='=') {kod=130;goto Operator2karVege;} // Az <= operátor kódja 130
if(b=='<') {kod=134;goto Operator2karVege;} // Az << operátor kódja 134
f.P--;kod=131; // A < operátor kódja 131
goto Operator2karVege;} // if a=='<' vége
if(a=='>') {
if(b=='=') {kod=132;goto Operator2karVege;} // Az >= operátor kódja 132
if(b=='>') {kod=135;goto Operator2karVege;} // Az >> operátor kódja 135
f.P--;kod=133; // A > operátor kódja 133
goto Operator2karVege;} // if a=='>' vége
if(a=='&') {
if(b=='&') {kod=136;goto Operator2karVege;} // Az && operátor kódja 136
f.P--;goto Operator2karVege;} // if a=='&' vége
if(a=='|') {
if(b=='|') {kod=137;goto Operator2karVege;} // Az || operátor kódja 137
f.P--;goto Operator2karVege;} // if a=='|' vége
if(a=='~') {
if(b=='~') {kod=138;goto Operator2karVege;} // Az ~ operátor kódja 138 (EXOR, „kizáró VAGY”)
f.P--;goto Operator2karVege;} // if a=='~' vége

f.P--;break; // default vége
} // switch vége
Operator2karVege:
return kod;
}
```

Működésének lényege, hogy az operátorokhoz hozzárendel egy kódot. Ha az operátor 1 karakteres, mármint ÚGY 1 karakteres hogy egyáltalán semmi se állhat utána amitől érvényes operátorrá válhatna, akkor ez a kód megegyezik a karakter ASCII értékével. Ellenkező esetben kap egy 127>nél nagyobb értéket. Ezenkívül a függvény leteszteli, a 2 karakter amiről szó van, nem valami olyasféle páros-e, ami nem operátor - ezek a `//`, `/*`, `/+`, `/-`, `/*` párok. Ezen esetekben **0** értéket ad kódként, amit majd letesztel az a függvény, ami ezt hívja.

A kétkarakteres operátoroknak azért adtam 127-nél nagyobb kódot, mert az a számérték már garantáltan nem ASCII kód, vagyis ha a mau forráskódunk UTF-8 szerint van kódolva, akkor e 127-nél nagyobb kódok egyszerűen nem jelentenek kiiratható karaktereket. Márpedig az operátorainkat nyilván a kiiratható karakterek közül választjuk. Azaz, így a későbbiek során bármi más operátorral is könnyen bővíthetjük szeretett programnyelvünket, és mégse kerülünk összeütközésbe a korábbiakkal, mert egyetlen, később kiválasztott karakter kódja se lesz olyan, amilyent most adunk az operátorainknak.

E függvényt természetesen a korábban bemutatott **Operatorc**, **OperatorI**, stb függvények hívják, amik egy kissé ki lettek bővíve! Egyet mutatok be belőlük, a többi mind egy kaptafára megy:

```
unsigned char Operatorc(unsigned char A,USC muveletijel,F& f) {
muveletijel=Operator2kar(f);if(muveletijel==0) return A;
switch(muveletijel) {
case '+': k(f);return (A + ERTEKunsignedchar(f));break;
case '-': k(f);return (A - ERTEKunsignedchar(f));break;
case '*': k(f);return (A * ERTEKunsignedchar(f));break;
case '/': k(f);return (A / ERTEKunsignedchar(f));break;
case 128: k(f);if(A == ERTEKunsignedchar(f)) return 1; else return 0;
case 129: k(f);if(A != ERTEKunsignedchar(f)) return 1; else return 0;
case 130: k(f);if(A <= ERTEKunsignedchar(f)) return 1; else return 0;
case 131: k(f);if(A < ERTEKunsignedchar(f)) return 1; else return 0;
case 132: k(f);if(A >= ERTEKunsignedchar(f)) return 1; else return 0;
case 133: k(f);if(A > ERTEKunsignedchar(f)) return 1; else return 0;
case 134: k(f);return (A << ERTEKunsignedchar(f));break; // <<
case 135: k(f);return (A >> ERTEKunsignedchar(f));break; // >>
case 136: k(f);return (A & ERTEKunsignedchar(f));break; // &&
case 137: k(f);return (A | ERTEKunsignedchar(f));break; // ||
case 138: k(f);return (A ^ ERTEKunsignedchar(f));break; // ~ (EXOR, „kizáró VAGY”)
case 139: k(f);return (A % ERTEKunsignedchar(f));break; // % (maradékképzés)
} // switch vége
return A;
}
```

Megjegyzendő, hogy a bitműveletek (<<, >>, &&, ||, ~~) csak az egész típusokra értelmezettek, azaz nem működnek a következő típusok esetén: **float**, **double**, **long double**.

Már látom lelki szemeimmel, hogy egyes ~~trólok~~ szörszálhasogató kritikusok belém kötnek, miért vagyok olyan hülye, hogy ilyen írok például:

```
if(A == ERTEKunsignedchar(f)) return 1; else return 0;
amikor ehelyett írhatnék ilyen is:
return (A == ERTEKunsignedchar(f));
```

Szerintük ekkor a kód szebb lenne, rövidebb, elegánsabb, gyorsabb...

Nos, nekem az a véleményem, hogy a kód épp így szebb, ahogy én csináltam. Amiatt szebb, mert sajnos az == művelet nem garantálja, hogy a visszatérési érték „igaz” eredmény esetén éppen pontosan 1 lesz! Márpedig én olyannak óhajtom a nyelvemet, hogy az összehasonlító operátoraim eredménye kizárólag csak 0 vagy 1 lehessen, egyáltalán semmi más se. A tököm tele van vele, hogy manapság a programozási nyelvek mindenféle bizonytalanságot megengednek maguknak. Láttunk már erre más példát is, e könyv elején, hogy például az egyes változó-típusok bájtmérete sem garantált...

Minthogy azonban a mi nyelvünkben az összehasonlító műveletek „IGAZ” értéke mindig és garantáltan pontosan 1, emiatt megkapjuk azt a hatalmas előnyt, hogy nem kell bohóckodnunk olyasmivel, mint a C nyelv esetén, hogy van külön & operátor meg && operátor, meg külön | operátor és || operátor,

az egyik a bitműveletekre, a másik a logikai értékekre! Amennyiben ugyanis csak 1 bitnyi értékekről van szó, mint nálunk az összehasonlító műveleteknél, úgy mindegyiknek az eredménye azonos:

```
1&1 = 1&&1 = 1,  
1&0 = 1&&0 = 0,  
0&0 = 0&&0 = 0,  
1|1 = 1||1 = 1,  
1|0 = 1||0 = 1,  
0|0 = 0||0 = 0
```

Vagyis nálunk a logikai operátor lehet ugyanaz, mint a bitműveletekre szolgáló. E célra a dupla jeleket választottam, a kettős & és | jeleket, mert jobban olvashatóak a forráskódban. Figyelemfelkeltőbbek.

Ezt kipróbálhatjuk e programmal:

```
"Összehasonlítások:" ; /  
=#c:0@t 3//  
"t = " ?c :0@t; /  
"3 == 4"; /; ?c 3==4; /;  
"3 == 3"; /; ?c 3==3; /;  
"3 == t"; /; ?c 3==:0@t /;  
"3 != 4"; /; ?c 3!=4 /;  
"3 != 3"; /; ?c 3!=3 /;  
"3 != t"; /; ?c 3!=:0@t; /;  
"3 <> 4"; /; ?c 3<>4 /;  
"3 <> 3"; /; ?c 3<>3; /;  
"3 <> t"; /; ?c 3<>:0@t; /;  
"3 <= 4"; /; ?c 3<=4; /;  
"3 <= 3"; /; ?c 3<=3; /;  
"3 <= t"; /; ?c 3<=:0@t; /;  
"4 <= 3"; /; ?c 4<=3; /;  
"3 < 4"; /; ?c 3<4; /;  
"3 < 3"; /; ?c 3<3; /;  
"3 < t"; /; ?c 3<:0@t; /;  
"4 < 3"; /; ?c 4<3; /;  
"3 >= 4"; /; ?c 3>=4; /;  
"3 >= 3"; /; ?c 3>=3; /;  
"3 >= t"; /; ?c 3>=:0@t; /;  
"4 >= 3"; /; ?c 4>=3; /;  
"3 > 4"; /; ?c 3>4; /;  
"3 > 3"; /; ?c 3>3; /;  
"3 > t"; /; ?c 3>:0@t; /;  
"4 > 3"; /; ?c 4>3; /;
```

Eredménye:

```
Összehasonlítások:  
t = 3  
3 == 4  
0  
3 == 3  
1  
3 == t  
1  
3 != 4  
1  
3 != 3  
0  
3 != t  
0  
3 <> 4  
1  
3 <> 3  
0  
3 <> t  
0  
3 <= 4  
1  
3 <= 3  
1  
3 <= t  
1  
4 <= 3  
0  
3 < 4  
1  
3 < 3
```



```

0
3 < t
0
4 < 3
0
3 >= 4
0
3 >= 3
1
3 >= t
1
4 >= 3
1
3 > 4
0
3 > 3
0
3 > t
0
4 > 3
1

```

Egy másik példa, ami már egész hasonlatosan néz ki egy „normális” programnyelvhez:

```

"Kezdem innen" /
if 3>2 T "Igaz" /
T "Ez is igaz" /
E "Nem igaz" /
T »$ug
"Ezt nem írom ki" /
$ug "Ide ugrott" /

```

Eredménye:

```

Kezdem innen
Igaz
Ez is igaz
Ide ugrott

```

Ellentéte:

```

"Kezdem innen" /
if 1>=2 T "Igaz" /
T "Ez is igaz" /
E "Nem igaz" /
T »$ug
"Ezt nem írom ki" /
$ug "Ide ugrott" /

```

Eredménye:

```

Kezdem innen
Nem igaz
Ezt nem írom ki
Ide ugrott

```

Ugyanez még két variációban:

```

=#c:0@j 5
"Kezdem innen" /
if (3>=2)&&((#c:0@j)<17) T "Igaz" /
T "Ez is igaz" /
E "Nem igaz" /
T »$ug
"Ezt nem írom ki" /
$ug "Ide ugrott" /

```

Eredménye:

```

Kezdem innen
Igaz
Ez is igaz
Ide ugrott

```

Ellentéte:

```
=#c:0@j 20
"Kezdem innen" /
if (3>=2)&&((#c:0@j)<17) T "Igaz" /
T "Ez is igaz" /
E "Nem igaz" /
T »$ug
"Ezt nem írom ki" /
$ug "Ide ugrott" /
```

Eredménye:

```
Kezdem innen
Nem igaz
Ezt nem írom ki
Ide ugrott
```

Egy harmadik példa:

```
M 30000;
!i 20 $fce2
"mem i = " ?i !20 /;
"mem c1 = " ?c !20 /;
"mem c2 = " ?c !21 /;
```

Eredménye:

```
mem i = 64738
mem c1 = 226
mem c2 = 252
```

Még valamit alakítunk kis aranyos kinézetén, vagyis a szintaktikáján. Valamiért természetellenesnek érzem, hogy a változó mezőindexét a neve ELŐTT adjuk meg! Próbáltam megszokni, nem ment. Amint azonban nézegettem a kódot, rájöttem, hogy könnyedén megoldható, hogy megadható legyen egyaránt a @ jel előtt is, vagy a változó neveként szolgáló aritmetikai kifejezés után is. S ehhez még csak olyan nagyon sok mindent nem is kell csinálnunk...

Mindenekelőtt, az értékadó utasításunkat írjuk át eként:

```
int egyenlokereszt(F& f) {USC index;USC unionindex;USC valtozotipus;
valtozotipus=vtype(f); // rögtön a következő karakter
namespace(f);kettospontteszt(f,unionindex,valtozotipus); // változóindex beolvasása, ha meg van adva
namespace(f);
if(f.p[f.P]!='@') return SERROR(f,3); // Ha az aktuális feldolgozandó karakter nem kukac, syntax error
k(f); // Ugrás a következő karakterre
index=ERTEKunsignedchar(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat
namespace(f);kettospontteszt(f,unionindex,valtozotipus); // változóindex beolvasása, ha meg van adva
switch(valtozotipus) {
case 'c': f.v[index].c[unionindex]=ERTEKunsignedchar(f);break;
case 'C': f.v[index].C[unionindex]=ERTEKsignedchar(f);break;
case 'i': f.v[index].i[unionindex]=ERTEKunsignedshortint(f);break;
case 'I': f.v[index].I[unionindex]=ERTEKsignedshortint(f);break;
case 'l': f.v[index].l[unionindex]=ERTEKunsignedint(f);break;
case 'L': f.v[index].L[unionindex]=ERTEKsignedint(f);break;
case 'f': f.v[index].f[unionindex]=ERTEKfloat(f);break;
case 'g': f.v[index].g[unionindex]=ERTEKunsignedlonglong(f);break;
case 'G': f.v[index].G[unionindex]=ERTEKsignedlonglong(f);break;
case 'd': f.v[index].d[unionindex]=ERTEKdouble(f);break;
case 'D': f.v[index].D=ERTEKlongdouble(f);break;
} // switch vége
return 0;
}
```

Látható, kizárólag annyival bővült, amennyit itt színessel kiemeltem, ami csak 1 sor.

E módosítást vezessük át a **perkereszt**, **csillagkereszt** stb függvényekbe is.

Ezután a **vc**, **vi**, stb függvényeinket írjuk át e példa szerint:

```
USC vc(F& f,USC unionindex=255) { // A programkódban soron következő változó aktuális értékét adja vissza USC típusként.
// A P mutató okvetlenül egy @ jelre kell mutasson!
USC index;USC változotípus;
változotípus='c';k(f);index=ERTEKunsignedchar(f);if(unionindex==255) { // Ha nem volt a baloldalon definiálva mezőindex
nemspace(f);kettospontteszt(f,unionindex,változotípus); // Az aktuális feldolgozandó karakter
if(unionindex==255) unionindex=f.vindexc[index];
}
return f.v[index].c[unionindex];
}
```

És ezek után bár a régi szintaxis is működik, de lehet effélét is csinálni:

```
"Összehasonlítások:" ; /
=#c:5@t 3//
"t = " ?c @t:5; /
"3 == t"; /; ?c 3==@t:5 /;
"?c @t:5 =" ?c @t:5 /;
"=#c@t:8 9" /;
=#c@t:8 9//
"t = " ?c @t:8; /
"9 == t"; /; ?c 9==@t:8 /;
"?c @t:8 =" ?c @t:8 /;
```

Vagy ilyesmit:

```
M 30000;
=#c:2@g 20
!i 20 $fce2
"mem i = " ?i !#c@g:2 /;
"mem i = " ?i !#c:2@g /;
"mem c1 = " ?c !20 /;
"mem c2 = " ?c !21 /;
```

A kétféle szintaxist vegyesen is használhatjuk.

13. fejezet: Névterek, és a változóink élettartama

Nem csodálnám, ha Olvasóm, ki követ engem e szellemi kalandban, már a kezdetektől fogva aggódott volna amiatt, hogy mégiscsak nagyon kevés lesz az a 256 darab változólehetőség a programnyelvünkben! Még akkor is, ha egy változó neve egyszerre több tényleges változót is jelölhet, mert több mezőből áll. Nos, ezen a gondon jelentős mértékben segítünk ezúttal, méghozzá ROPPANT ELŐKELŐEN! Olyan lehetőséget építünk be ezúttal a nyelvünkbe - s hozzá úgy, hogy nem is kell emiatt temérdek kódsort írunk! - ami igazából számos programnyelvből hiányzik, bár a legelterjedtebbekben kétségtelenül benne van. Ez pedig a „névterek” lehetősége. Ez tulajdonképpen azt mondja meg, egy adott nevű változó meddig „él”, azaz létezik. Ez szoros összefüggésben van az olyan furmányos huncutságokkal is, mint a függvények hívásakor történő input paraméterátadás, meg szintén a függvények befejezésekor történő output paraméterek átadása.

Márpedig merem állítani, én erre olyan megoldást találtam, ami lekörözi még az oly nagy becsben tartott, híres, tekintélyes „C” nyelvet is!

A programnyelvek efféle kérdés szempontjából mint a névtér, illetve paraméterátadás, nagyjából a következő csoportokra oszthatóak:

1. Egyetlen névtér van, minden változó mindig mindenhol látható. Ilyen a Basic nyelv legtöbb változata, például a régi C-64 -es számítógépen megvalósított Basic is. És eddig a pillanatig még a mi mau nyelvünk is ilyen. (De már nem sokáig...!)

2. Több névtér van. Hogy hány, az az adott programnyelvtől függ. Némelyikben olyan rengeteg, hogy cseppet se könnyű észben tartani, melyik változónak meddig terjed az élettartama.

Amikor a nyelvben több névtér van, az tipikusan azt jelenti, hogy egy meghívott függvény nem látja az őt hívó program változóit, ezért akár ugyanolyan néven használhat ő is változókat, melyeknek azonban lehet teljesen más jelentésük is, eképp más értékük is, s ezen változók értékének változása semmiféle módon nem befolyásolja a hívó program hasonló nevű változóinak értékét. Ez természetesen remek dolog, mert ha ezt mi megvalósítjuk, máris nem tűnik kevésnek az a változónév-mennyiség, amivel rendelkezünk!

Igenám, de ekkor merül fel az a gond, hogy ha a hívott program egyetlen árva változót se lát a hívó névtérében szereplők közül, akkor miként is kaphat onnan input paramétereket! S ennek a gondnak a fordítottja is megjelenik: mikor befejezi a futását a hívott program, akkor oké hogy a legtöbb változója nyugodtan megsemmisülhet, de NÉHÁNY változó eredményét mégis vissza kéne juttatnia a hívónak, mert hát ugye nyilván azért hívtuk meg a hívott rutint hogy valami hasznos tevékenységet folytasson, s annak igényelnénk az eredményét!

Erre a gondra aztán az eddig elkészült programnyelvek a legkülönbözőbb technikákat fejlesztették ki. A legismertebb talán a C módszere, mely (kissé leegyszerűsítetten elmagyarázva a dolgokat) abból áll, hogy vannak úgynevezett „globális” változók, amiket minden függvény láthat, ezen kívül azonban a függvény csak a maga változóit láthatja. Input paraméterként egy változólistát kap, aminek értékeit megfelelően formálisan a maga bizonyos változóinak, érték szerint. Eredményül a függvény csak egyetlen értéket adhat vissza, máskülönben kénytelen a globális változókat használni, vagy pointerekkel bűvészkedni. Utóbbira a C++ nyelv ad némi „emberközelibb” megoldást, a „referencia típusú paraméterátadást”, de igazából az is csak pointer, csupán ezesetben a pointerekre való hivatkozás kissé el van rejtve és automatizálttá téve. Emellett pedig a függvények mind egyenrangúak, nem lehet olyasmit csinálni, hogy az egyik függvény belsejébe beépítünk egy másik függvényt.

Nálunk eddig csak globális változók voltak. Én azonban a következőt találtam ki:

Függvényeink még nincsenek ugyan, de majd lesznek. (Nem kell aggódni...!) Ellenben minden függvény úgy indul majd, hogy alapértelmezés szerint ő csak egy szubrutin, azaz olyan utasításrészlet, aminek nincsenek se input, se output paraméterei, ellenben láthatja az őt hívó program (alap esetben ez a főprogram) minden változóját. Illik azonban, bár nem kötelező, hogy a függvény azzal kezdődjék, hogy „saját névtérét hoz létre”. E saját névtér akkor is létrehozható, ha nem függvényről van szó amiből „vissza kell térni”, csak a programunk egy egyszer végrehajtandó részét akarjuk valamiért jól elkülöníteni a többi résztől.

Ez utasítások szempontjából a következőt jelenti, vázlatosan bemutatva:

{ A dupla nyitó kapcsos zárójellel kezdődik a névtér meghatározása. Ekkor létrejön egy új „mau” nyelvű program adatterülete, azonban a programkód nem változik. Ez a gyakorlatban azt jelenti, hogy e dupla nyitó kapcsos zárójel után - mindaddig míg egy SZIMPLA, azaz NEM DUPLA csukó kapcsos zárójelet nem talál - minden teljesen ugyanúgy működik mint korábban, EGYETLEN kivételtől eltekintve: hogy az **=#** utasításunk, és KIZÁRÓLAG ez, tehát a **+#**, a **-#**, ***#**, **/#** az NEM ! — szóval tehát ez az **=#** értékadó utasítás innentől kezdve úgy fog dolgozni, hogy a „jobbérték”-et amikor kiszámítja, tehát azt amit majd egy változó eredményéül kell adnia, azt továbbra is úgy számolja, hogy a kifejezés kiértékelésénél minden adatot onnan vesz ahonnan eddig, azaz a „szülő” névtérből, de ezen adatokat, az eredményt már nem a szülő névtér változóiba teszi bele, hanem az új névtér változóiba, mert neki már az a „balérték”!

} Az egy darab csukó kapcsos zárójel az, ami az egész mindenség lényege: innentől kezdődik az új névtérben végrehajtódni a program! Ez tehát tulajdonképpen a „gyermekprocessz”. Vegyük észre ugyanis, hogy gyakorlatilag „forkoltuk” a programunkat: a végrehajtandó kód ugyanaz maradt, de az adatterület teljesen megváltozott, kivéve hogy néhány változóba beletöltöttünk input paramétereket. A többinek az értéke azonban nulla.

{ Innentől kezdve - tehát a szimpla nyitó kapcsos zárójeltől kezdve - egészen addig míg egy dupla csukó kapcsos zárójellel nem találkozunk, még mindig a „gyermekprocesszben” dolgozik, azaz az új névtérben, de úgy, hogy az **=#** értékadó utasításunk a kifejezés kiértékelésénél a változókat a maga új névtéréből veszi, ellenben az eredményt a szülő program névtérének változóiba pakolja.

}} Ez fejezi be az új névtérben folyó munkát, innentől visszatér minden a szülő-processz névtérébe.

Kissé rövidebben megfogalmazva, más szavakkal, ez a következőt jelenti:

SZÜLŐ NÉVTÉR

{ INPUT PARAMÉTEREK ÁTADÁSA } Ezt nevezhetjük a névtér vagy függvény fejlécének

Program az új névtérben Ezt a névtér fő része, itt folyik a „munka”, ez a függvény törzse.

{ EREDMÉNYEK ÁTADÁSA } Ez a névtér vagy függvény „lábléce”

SZÜLŐ NÉVTÉR

Az ezt megvalósító rutinjaink igazán pofonegyszerűek, ahhoz képest, milyen óriási könnyebbséget jelentenek nekünk! Nézzük is őket!

Mindenekelőtt, bővítsük ki az F struktúra elejét így:

```
struct F {  
    PGM *CHILD;  
    F *parentF; // Mutató a szülőnévtér struktúrájára  
    unsigned int NamespaceDeep; // A névtér mélységét számolja. A főprogram szintje 0.  
    USC parentflag; // Ha ez 0, minden a „szokásos”.  
    // Ha 1, akkor az =# utasítás a jobbértéket nem az „f” struktúrába pakolja, hanem a CHILD f struktúrába!  
    // Ha 255, akkor az =# utasítás a jobbértéket nem az „f” struktúrába pakolja, hanem a parentF struktúrába!
```

A PGM osztályunk eleje így nézzen ki:

```
class PGM {  
private:  
    struct F f;
```

```

int errorcode; // A programutasítások visszatérési értékei. 0=normál; -1=végetért a program hibával;
                // 1=végetért a program rendben; 2=BREAK, stb
friend int csukokapcsoszarojel(F& f);
friend int nyitcsukkapcsoszarojel(F& f);
friend int egyenlokereszt(F& f);
// -----

```

Erre azért van szükség, mert a **csukokapcsoszarojel** és az **egyenlokereszt** függvények el kell érjék a „f” struktúrát és az **errorcode** változót. Lehetett volna egyszerűen „public” minősítést is megadnom e két adattagnak, de nem látom be miért kéne úgy csinálnom, amikor az osztályok létének épp az a legnagyobb értelme, hogy lehetőleg senki se birizgálja az adatmezőket, akinek nincs erre múlt-hatatalanul szüksége.

Csináljunk alapértelmezett konstruktort a PGM osztályunknak:

```

PGM::PGM() { // Alapértelmezett KONSTRUKTOR
Fzero(f); // A struktúra lenullázása
}

void Fzero(F& f) { // Egy F struktúrát beállít a kezdeti értékekre
f.phossz=0;f.p=0L;f.p=NULL; // Programmemória mutatója kezdetben NULL
f.parentF=NULL; // szülőnévtér mutató lenullázása
f.CHILD=0; // Gyermekprocessz mutató nulla
f.m=NULL;f.mhossz=0L; // adatmemória nullára állítása
f.mems[0]=&f.p;f.mems[1]=&f.m;
f.parentflag=0;f.BRAINFUCKflag=0;
register int i,j;for(i=0;i<256;i++) { // A változóindexek lenullázása
f.vindexc[i]=f.vindexC[i]=f.vindexi[i]=f.vindexI[i]=f.vindexl[i]=f.vindexL[i]=
f.vindexf[i]=f.vindexg[i]=f.vindexG[i]=f.vindexd[i]=0;
for(j=0;j<16;j++) {f.v[i].c[j]=0;}} // A változók lenullázása
f.B=0;f.J=1; // Balérték és jobbérték indexek beállítása
f.cimkedb=0;for(i=0;i<4096;i++) f.cimke[i]=0; // címkek lenullázása
f.ifflag=0; // Összehasonlító flag értéke kezdetben 0
f.NamespaceDeep=0;
}

```

És írjuk át a régi konstruktorunkat (sokkal egyszerűbb lesz mert belőle majdnem minden kikerült az **Fzero** függvénybe):

```

PGM::PGM(USIL phossza) { // KONSTRUKTOR
Fzero(f); // A struktúra lenullázása
f.phossz=phossza;
f.p = new USC[f.phossz];if(!f.p) {L("Nincs elég memória!");EXITFAILURE();}
}

```

Az új értékadó utasításunk:

```

int egyenlokereszt(F& f) {USC index;USC unionindex;USC valtozotipus;F *ff;
ff=&f;
if(f.parentflag==1) {
if(f.CHILD==NULL) {L("Érvénytelen, NULL értékű CHILD névtér! Pozíció: %lu",f.P);EXITFAILURE();}
ff=&f.CHILD->f;
if(ff==NULL) {L("Érvénytelen, NULL pointerű F struktúra a CHILD névtérben! Pozíció: %lu",f.P);EXITFAILURE();}
}
if(f.parentflag==255) {if(f.parentF==NULL) {L("Érvénytelen, NULL értékű szülőnévtér! Pozíció: %lu",f.P);EXITFAILURE();}
ff=f.parentF;}

valtozotipus=vtype(f); //rögtön a következő karakter
namespace(f);kettospontteszt(f,unionindex,valtozotipus); // változóindex beolvasása, ha meg van adva
namespace(f);
if(f.p[f.P]!='@') return ERROR(f,3); // Ha az aktuális feldolgozandó karakter nem kukac, syntax error
k(f); // Ugrás a következő karakterre
index=ERTEKunsignedchar(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat
namespace(f);kettospontteszt(f,unionindex,valtozotipus); // változóindex beolvasása, ha meg van adva
switch(valtozotipus) {
case 'c': (*ff).v[index].c[unionindex]=ERTEKunsignedchar(f);break;
case 'C': (*ff).v[index].C[unionindex]=ERTEKsignedchar(f);break;
case 'i': (*ff).v[index].i[unionindex]=ERTEKunsignedshortint(f);break;
case 'I': (*ff).v[index].I[unionindex]=ERTEKsignedshortint(f);break;
case 'l': (*ff).v[index].l[unionindex]=ERTEKunsignedint(f);break;
case 'L': (*ff).v[index].L[unionindex]=ERTEKsignedint(f);break;
case 'f': (*ff).v[index].f[unionindex]=ERTEKfloat(f);break;
case 'g': (*ff).v[index].g[unionindex]=ERTEKunsignedlonglong(f);break;
case 'G': (*ff).v[index].G[unionindex]=ERTEKsignedlonglong(f);break;
}

```

```

case 'd': (*ff).v[index].d[unionindex]=ERTEKdouble(f);break;
case 'D': (*ff).v[index].D=ERTEKlongdouble(f);break;
} // switch vége
return 0;
}

```

A „folytat” rutinunk örökciklusa ki kell bővüljön e sorral:

```

if(errorcode==255) {return;} // Visszatérés a forkból (a gyermekfolyamatból)

```

A kapcsolószerűjeles függvényeink:

```

int duplanyitokapcsoloszarojel(F& f) { // Új névtér létrehozása
if(f.CHILD!=NULL) {L("Új \"{}\" parancsot adtál ki, mielőtt az előzőt lezártad volna egy \"{}\" jellel!\n"
"Pozíció: %lu",f.P);EXITFAILURE();}
f.CHILD = new PGM; // Gyermekfolyamat inicializálása
f.parentflag=1;
return 0;
}
// -----
int nyitokapcsoloszarojel(F& f) {f.parentflag=255;return 0;}
// =====
void PGM::PrepareFork(F& REGI_f) {
f.parentF = &REGI_f;
f.NamespaceDeep = REGI_f.NamespaceDeep + 1;
f.p = REGI_f.p;
f.phossz = REGI_f.phossz;
f.P = REGI_f.P;
f.a = REGI_f.a;
f.a2 = REGI_f.a2;
f.programfileneve = REGI_f.programfileneve;
f.programfilenevehossz = REGI_f.programfilenevehossz;
f.programfilehossz = REGI_f.programfilehossz;
f.cimkedb = REGI_f.cimkedb;
register unsigned short int i;for(i=0;i<4096;i++) {f.cimke[i] = REGI_f.cimke[i];}}
}

// =====
int csukokapcsoloszarojel(F& f) { // Névtér fork
f.parentflag=0;
// Innentől fel kell töltsük az új struktúrát a réginek a szükséges adataival!
f.CHILD->PrepareFork(f); // Átnásoljuk a szükséges adatokat
// Gyermekfolyamat indítása
f.CHILD->folytat(); // fork !!!
// Itt jön vissza a vezérlés a gyermekfolyamatból !
if(f.CHILD->errorcode != 255) {
L("Nagy gáz van: a gyermekfolyamat nem szabályosan ért véget!\n"
"A pozíció ahonnan indult a gyermekfolyamat: %lu\n"
"A pozíció ahol a gyermekfolyamat végetért: %lu",f.P,f.CHILD->f.P);EXITFAILURE();
}
f.P = f.CHILD->f.P;
f.a = f.CHILD->f.a;
f.a2 = f.CHILD->f.a2;
if(f.CHILD->f.m == f.m) f.CHILD->f.m=NULL; // Ha átadtuk a mi memóriamutatónkat a gyerekfolyamatnak, akkor
// azt most lenullázzuk, nehogy a delete törölje nekünk...
f.CHILD->f.p=NULL; // különben a CHILD destruktora törölné a delete-vel, ami nem jó ötlet, mert a
// szülőprocessznek is ugyanez a memóriarésze, ahol a forráskód van...

delete f.CHILD; // A gyermekfolyamatot megdöglesztjük
f.CHILD=NULL; // biztos ami biztos...
return 0;
}
// -----
int duplacsukokapcsoloszarojel(F& f) {
if(logflags[255]) L("Befejeződött egy gyermekfolyamat! Mélységi szint = %u",f.NamespaceDeep);
return 255;
}

```

Ezek után persze illik beépíteni lehetőséget a névtérmélység lekérdezésére is. Ehhez a **systemvariable** függvénybe vegyük be ezen egyetlen sort:

```

case 'Y': value=(USIL)f.parentflag; break;

```

Ezenfelül, tudatosítanunk kell magunkban, hogy ezen új névtérben nem áll rendelkezésünkre automatikusan az „**m**” mutató által megcímzett memória-terület! Ez előny, mert a gyermekprocesszben lefoglalhatunk magunknak új, saját

memóriaterületet ha ehhez van kedvünk, ami automatikusan felszabadul a gyermekfolyamatból (névtérből) való kilépéskor. Amennyiben azonban szükségünk van a szülőnévtér memóriaterületére, ezt át kell adnunk a gyermeknévtérnek. E célra szolgál ez a parancs:

```
{}
```

Igen, egymás után egy nyitó és egy csukó kapcsoszárojel. Semmiféle karakter nem állhat köztük! Ezt azonban nem adhatjuk ki máshol, kizárólag a

{ és a } között, ott tehát, ahol amúgy is átadogatjuk az input paramétereket. Elvégre ez is egy input paraméter... E funkció megvalósítása:

```
int nyitcsukkapcsoszarojel(F& f) {
if(f.parentflag!=1) {L("Illegális helyen használod a \"{}\" parancsot! Pozíció: %lu",f.P);EXITFAILURE();
}
f.CHILD->f.m=f.n;f.CHILD->f.mhossz=f.mhossz; // A memóriamutató és hossz átmásolása a gyerekfolyamatnak
return 0;}
```

Mindez bemutatva egy példán:

```
"Főprogram indul!" /
=#c@f:3 5; "f értéke 5 kell legyen: f = " ?c @f:3; /
=#c@a:2 7; "a értéke 7 kell legyen: a = " ?c @a:2; /
=#c@c:1 8; "c értéke 8 kell legyen: c = " ?c @c:1; /
{{
"Input adatok megadása a gyermekfolyamatnak:" /;
"=#c@a:2 @a:2 « CHILD-a = PARENT-a " =#c@a:2 @a:2 ; /
"=#c@c:1 @c:1 « CHILD-c = PARENT-c " =#c@c:1 @c:1 ; /
} "Indul a gyermekprocessz!" /;

"f értéke 0 kell legyen: f = " ?c @f:3; /
"a értéke 7 kell legyen: a = " ?c @a:2; /
"c értéke 8 kell legyen: c = " ?c @c:1; /
"Most a gyermekfolyamatban értékeket adunk az „a” és az „f” változóknak:" /;
=#c@f:3 9; "f értéke 9 kell legyen: f = " ?c @f:3; /
=#c@a:2 50; "a értéke 50 kell legyen: a = " ?c @a:2; /
{ }}
"Végetért a gyermekprocessz!" /;
"Semmi adatot nem adtunk át belőle a hívó programnak, annak változói nem szabad hogy megváltozzanak:" /
"f értéke 5 kell legyen: f = " ?c @f:3; /
"a értéke 7 kell legyen: a = " ?c @a:2; /
"c értéke 8 kell legyen: c = " ?c @c:1; /
"Most ugyanezt a gyermekfolyamatot megismételjük, de a végén adunk át paramétereket!" /
{{
"Input adatok megadása a gyermekfolyamatnak:" /;
"=#c@a:2 @a:2 « CHILD-a = PARENT-a " =#c@a:2 @a:2 ; /
"=#c@c:1 @c:1 « CHILD-c = PARENT-c " =#c@c:1 @c:1 ; /
} "Indul a gyermekprocessz!" /;

"f értéke 0 kell legyen: f = " ?c @f:3; /
"a értéke 7 kell legyen: a = " ?c @a:2; /
"c értéke 8 kell legyen: c = " ?c @c:1; /
"Most a gyermekfolyamatban értékeket adunk az „a” és az „f” változóknak:" /;
=#c@f:3 9; "f értéke 9 kell legyen: f = " ?c @f:3; /
=#c@a:2 50; "a értéke 50 kell legyen: a = " ?c @a:2; /
{
"Ít adjuk vissza az input adatokat:" /;
"=#c@a:2 @a:2 « = PARENT-a = CHILD-a" =#c@a:2 @a:2 ; /
}}
"Végetért a gyermekprocessz!" /;
"Az „a” értéke szabad csak hogy megváltozott legyen:" /
"f értéke 5 kell legyen: f = " ?c @f:3; /
"a értéke 50 kell legyen: a = " ?c @a:2; /
"c értéke 8 kell legyen: c = " ?c @c:1; /
"Most ugyanezt a gyermekfolyamatot megismételjük, de belőle újabb gyermekfolyamatot indítunk:" /
{{
"Input adatok megadása az 1. szintű gyermekfolyamatnak:" /;
"=#c@a:2 @a:2 « CHILD-a = PARENT-a " =#c@a:2 @a:2 ; /
"=#c@c:1 @c:1 « CHILD-c = PARENT-c " =#c@c:1 @c:1 ; /
} "Indul az 1. szintű gyermekprocessz!" /;
"f értéke 0 kell legyen: f = " ?c @f:3; /
"a értéke 7 kell legyen: a = " ?c @a:2; /
"c értéke 8 kell legyen: c = " ?c @c:1; /
"Most az 1. szintű gyermekfolyamatban értékeket adunk az „a” és az „f” változóknak:" /;
=#c@f:3 9; "f értéke 9 kell legyen: f = " ?c @f:3; /
=#c@a:2 50; "a értéke 50 kell legyen: a = " ?c @a:2; /

{ } "Indul a 2. szintű gyermekprocessz!" /;
{ }
=#c@a:2 @a:2; "2.gyermekfolyamat „a” értéke = 1. gyermekfolyamat „a” értéke." /
```



```

}
"Most vagyunk a 2-es gyermekfolyamat belsejében." /
"Értéket adunk az „a” változónak." /
=#c@a:2 111; "Az „a” értéke 111 kell legyen: a = " ?c @a:2; /
{
"Most a 2-es gyermekfolyamat „a” értékét visszaadjuk az öt szülő 1-es folyamat „h” változójába!" /
=#c@h:9 @a:2
}}
"Végetért a 2-es gyermekfolyamat" /
"Most az 1-es gyermekfolyamatban vagyunk. A változók értékei:" /
"az „a” 50 kell legyen: " ?c @a:2 ; /
"a „c” 8 kell legyen: " ?c @c:1 ; /
"az „f” 9 kell legyen: " ?c @f:3 ; /
"a „h” 111 kell legyen: " ?c @h:9 ; /

{
"Itt adjuk vissza az input adatokat az 1. szintről a 0 szintre:" /;
"=#c@a:2 @a:2 « = PARENT-a = CHILD-a" =#c@a:2 @a:2 ; /
}}
"Végetért az 1 szintű gyermekprocessz!" /;
"Az „a” értéke szabad csak hogy megváltozott legyen:" /
"f értéke 5 kell legyen: f = " ?c @f:3; /
"a értéke 50 kell legyen: a = " ?c @a:2; /
"c értéke 8 kell legyen: c = " ?c @c:1; /
"h értéke 0 kell legyen: c = " ?c @h:9; /

```

E fenti példában a memóriaterület átadását végző {} parancs kék színnel van kiemelve, tudniillik teljesen felesleges ide mert anélkül is működik minden, csak bemutattam, hova lehetne elhelyezni például, ha szükséges volna.

A program futásának eredménye:

```

Főprogram indul!
f értéke 5 kell legyen: f = 5
a értéke 7 kell legyen: a = 7
c értéke 8 kell legyen: c = 8
Input adatok megadása a gyermekfolyamatnak:
=#c@a:2 @a:2 « CHILD-a = PARENT-a
=#c@c:1 @c:1 « CHILD-c = PARENT-c
Indul a gyermekprocessz!
f értéke 0 kell legyen: f = 0
a értéke 7 kell legyen: a = 7
c értéke 8 kell legyen: c = 8
Most a gyermekfolyamatban értékeket adunk az „a” és az „f” változóknak:
f értéke 9 kell legyen: f = 9
a értéke 50 kell legyen: a = 50
Végetért a gyermekprocessz!
Semmi adatot nem adtunk át belőle a hívó programnak, annak változói nem szabad hogy megváltozzanak:
f értéke 5 kell legyen: f = 5
a értéke 7 kell legyen: a = 7
c értéke 8 kell legyen: c = 8
Most ugyanezt a gyermekfolyamatot megismételjük, de a végén adunk át paramétereket!
Input adatok megadása a gyermekfolyamatnak:
=#c@a:2 @a:2 « CHILD-a = PARENT-a
=#c@c:1 @c:1 « CHILD-c = PARENT-c
Indul a gyermekprocessz!
f értéke 0 kell legyen: f = 0
a értéke 7 kell legyen: a = 7
c értéke 8 kell legyen: c = 8
Most a gyermekfolyamatban értékeket adunk az „a” és az „f” változóknak:
f értéke 9 kell legyen: f = 9
a értéke 50 kell legyen: a = 50
Itt adjuk vissza az input adatokat:
=#c@a:2 @a:2 « = PARENT-a = CHILD-a
Végetért a gyermekprocessz!
Az „a” értéke szabad csak hogy megváltozott legyen:
f értéke 5 kell legyen: f = 5
a értéke 50 kell legyen: a = 50
c értéke 8 kell legyen: c = 8
Most ugyanezt a gyermekfolyamatot megismételjük, de belőle újabb gyermekfolyamatot indítunk:
Input adatok megadása az 1. szintű gyermekfolyamatnak:
=#c@a:2 @a:2 « CHILD-a = PARENT-a
=#c@c:1 @c:1 « CHILD-c = PARENT-c
Indul az 1. szintű gyermekprocessz!
f értéke 0 kell legyen: f = 0
a értéke 7 kell legyen: a = 50
c értéke 8 kell legyen: c = 8
Most az 1. szintű gyermekfolyamatban értékeket adunk az „a” és az „f” változóknak:
f értéke 9 kell legyen: f = 9
a értéke 50 kell legyen: a = 50
Indul a 2. szintű gyermekprocessz!
2.gyermekfolyamat „a” értéke = 1. gyermekfolyamat „a” értéke.
Most vagyunk a 2-es gyermekfolyamat belsejében.

```

```

Értéket adunk az „a” változónak.
Az „a” értéke 111 kell legyen: a = 111
Most a 2-es gyermekfolyamat „a” értékét visszaadjuk az öt szülő 1-es folyamat „h” változójába!
Végetért a 2-es gyermekfolyamat
Most az 1-es gyermekfolyamatban vagyunk. A változók értékei:
az „a” 50 kell legyen: 50
a „c” 8 kell legyen: 8
az „f” 9 kell legyen: 9
a „h” 111 kell legyen: 111
Itt adjuk vissza az input adatokat az 1. szintről a 0 szintre:
=#c@a:2 @a:2 « = PARENT-a = CHILD-a
Végetért az 1 szintű gyermekprocessz!
Az „a” értéke szabad csak hogy megváltozott legyen:
f értéke 5 kell legyen: f = 5
a értéke 50 kell legyen: a = 50
c értéke 8 kell legyen: c = 8
h értéke 0 kell legyen: c = 0

```

Na szerintem ha valamire mondható hogy „nagy durranás” volt, akkor EZ bizony az! Gondoljuk csak el: az sok programnyelvben megoldott, hogy a meghívott függvénynek átadjanak akárhány paramétert is. Az azonban korántsem, hogy annak befejeződésekor is lehessen visszaadni akárhány paramétert a hívónak! Ezenfelül, nálunk a függvények nem egyenrangúak, hanem akárhány mélységben egymásbaágyazhatóak!

Oké, erre mondhatod persze, hogy ezek még nem igazi függvények, mert nem térnek vissza oda, ahonnan meghívtuk őket!

Mindenekelőtt, nem győzöm hangsúlyozni, hogy már így is mekkora roppant előnyei vannak ennek a lehetőségnek! Mindazonáltal, igazán nem nagy művészet ezek után kialakítani egy tisztességes szubrutin-lehetőséget belőle. Nézzük csak, mi is kell hozzá... Mindenekelőtt, bővítsük ki az F struktúrát ezzel:

```

PGM *THIS;
unsigned int szubrutinmelyseg;

```

Az Fzero szubrutint pedig ezzel:

```

f.szubrutinmelyseg=0;

```

A konstruktoraink mostantól így néznek ki:

```

PGM::PGM(USIL phossza) { // KONSTRUKTOR
f.THIS=this;
Fzero(f); // A struktúra lenullázása
f.phossz=phossza;
f.p = new USC[f.phossz];if(!f.p) {L("Nincs elég memória!");EXITFAILURE();}
}
// -----
PGM::PGM() { // Alapértelmezett KONSTRUKTOR
f.THIS=this;
Fzero(f); // A struktúra lenullázása
}

```

Nem változtak sokat...

A **folytat** - örökciklusunkba bekerül ez a sor:

```

if(errorcode==4) {return;} // Visszatérés szubrutinból

```

Illik csinálnunk egy utasítást arra is, hogy bárhol kiléphessünk a programból úgymond „sikeresen”, legyen ennek a neve az, hogy „**XX**”:

```

int fuggveny_XX(F& f) {
if(logflags[3]) {L("Befejeződött sikeresen a program, egy XX utasításnál!\n"
"Mélységi szint = %u, pozíció: %lu",f.NamespaceDeep, f.P);}

```

```
EXITSUCCESS();
return 0;
}
```

Maga a szubrutinhívást megvalósító rutinunk így néz ki, ennek az utasításnak egyszerűen „**G**” a neve, emlékül a jó öreg „gosub”-ra a Basic nyelvből:

```
int fuggveny_G(F& f) {unsigned int cimke;unsigned int regiP;
cimke=ERTEKunsignedint(f);if(cimke==0) {L("Nem létező címkére megkísérelt ugrás! cím: %lu",f.P);return -1;}
f.szubrutinmelyseg++;
regiP=f.P;f.P=cimke;
f.THIS->folytat();
f.P=regiP;
f.szubrutinmelyseg--;
return 0;
}
```

És már csak valami olyan utasítást kell kreálnunk, ami a „**return**” megfelelője, azaz ami a szubrutin végén visszavisz bennünket a kiindulási pontra! Erre semmi sem tűnik alkalmasabb szimbólumnak, mint a „**«**” jel. Mert ugye, elugorni valahova, azt a „**»**” jellel csináljuk. Hát akkor a visszaugrásnak legyen a jele ennek a tükörképe... Íme a rutin:

```
int duplابلرانييل(F& f) { // A « utasítás, szubrutinból való visszatérés
if(f.szubrutinmelyseg==0) {L("Kísérlet szubrutinból visszatérésre, a főprogramból! Pozíció: %lu",f.P);
EXITFAILURE();}
return 4;
}
```

Látható, hogy csak egy hibaellenőrzésből áll. Nem is csoda: a tulajdonképpeni munkát a „**G**” rutin végzi helyette, a „**«**” csak jelképnek kell.

Íme egy példa arra, hogyan lehet ezt használni:

```
"Főprogram indul!" /
"Kétszer kiíratom vele mennyi 7*2" /
=#c@a:2 7; G $ch; G $ch;
=#c@a:2 5; G $ch;
=#c@a:2 10; G $ch;

XX // vége a programnak

"Ezt sosem szabad kiírnia!" /

$ch // Itt egy „ch” nevű alprogram kezdődik
{{ =#c@a:2 @a:2 } // Input adatok átadása
"a értéke = " ?c @a:2; " ... a*2 értéke = " ?c (@a:2)*2; /
{
// Itt adjuk vissza az eredményeket. Most épp semmit.
}} // Végeért a „ch” gyermekprocessz és szubrutin!

« // Visszatérés a szubrutinból
```

A futás eredménye:

```
Főprogram indul!
Kétszer kiíratom vele mennyi 7*2
a értéke = 7 ... a*2 értéke = 14
a értéke = 7 ... a*2 értéke = 14
a értéke = 5 ... a*2 értéke = 10
a értéke = 10 ... a*2 értéke = 20
```

Azt hiszem, a fenti példa nem igényel semmi külön plusz magyarázatot.

Egy hasonló példa, bőven kommentelve, amiben már van ciklusszervezés is:

```
"Főprogram indul!" /

=#c@a:2 0; // a=0
$ci // Ciklus elejének címkéje
```

```

G $ch;      // szubrutin hívása
+#c@a:2 1 // a+=1
if (#c@a:2) < 8 T »$ci // Ha a<8 visszaugrik, azaz folytatja a ciklust

XX // vége a programnak

"Ezt sosem szabad kiírnia!" / // mert ide sosem jut el a vezérlés

$ch // Itt egy „ch” nevű alprogram kezdődik
// Ez annyit csinál, hogy a megadott számot megszorozza kettővel
// Itt még a főprogram névterében vagyunk, de csak a következő
// dupla kapcsos nyitózáráig. Az léptet be a saját névtérbe.
{{ =#c@a:2 @a:2 } // Input adatok átadása. Különböztet nem találja meg az „a” változót a saját névterében
// Az előző szimpla csukó kapcsos zárjólal a dolgok lényege,
// az gyakorlatilag forkolja a programot.
// Innentől kezdődik a „lényegi munka” a szubrutinban (vagy függvényben, mindegy mi a neve)

"a értéke = " ?c @a:2; " ... a*2 értéke = " ?c (@a:2)*2; /

{ // Előkészület a névtérből kilépésre
// Itt adjuk vissza a szubrutin eredményeit, ha át akarunk menteni
// valami adatot belőle a hívó program változói számára. Most ilyesmit nem művelünk.
}} // Végetért a „ch” gyermekprocessz és szubrutin! Ezzel kiléptünk a rutin névteréből
// Innentől már újra a hívó program névterében vagyunk
« // Visszatérés a szubrutinból

```

A futási eredmény:

```

Főprogram indul!
a értéke = 0 ... a*2 értéke = 0
a értéke = 1 ... a*2 értéke = 2
a értéke = 2 ... a*2 értéke = 4
a értéke = 3 ... a*2 értéke = 6
a értéke = 4 ... a*2 értéke = 8
a értéke = 5 ... a*2 értéke = 10
a értéke = 6 ... a*2 értéke = 12
a értéke = 7 ... a*2 értéke = 14

```

Fontos megemlíteni, hogy a címkék ugyanolyan értékűek maradnak az új névtérben, a gyermekprocesszben is, azok értéke ugyanis a forráskód beolvasásakor dől el.

Látható, gyakorlatilag semmi szükségünk holmi külön ciklusszervező utasításokra, mint a for, next, do, until, while... ezek nélkül is képesek vagyunk megoldani mindent! Persze, szebb lenne az élet ezekkel... És simán meg is oldhatóak. Ezekhez azonban előbb meg kell csinálnunk azt, ami amúgy is illendő hogy legyen nekünk: a veremkezelést! Ez lesz a következő fejezet témája.

14. fejezet: Veremkezelés

Szerintem a címtől sokakat a hideg ráz ki. Jellemző a kezdő programozókra, hogy valami általam érthetetlen oknál fogva hideglelősen borzonganak a veremtárnak már a gondolatától is. Ha ezt csak megemlítik is nekik, már olyan arcot vágunk, mintha agyérgörcs kínozná őket, vagy ha udvariasak, akkor fízológias reakcióik bemutatását lecsökkentik olyan szintre, hogy mindössze olyannak látsszanak, mintha citromot ennének két pofára, cukor nélkül.

Nem tudom pedig tényleg, miért: Számomra legelőször is már amikor hallottam e fogalomról, teljesen érthetőnek és könnyűnek tűnt - A verem egy olyan memóriaterület, amit úgy kell elképzelni, mint egy szűk torkú zsákot, vagy egy lyukat a földben, szűk nyílással: nem kotorászhat sz benne, csak mindig azt szedheted ki belőle, ami legfelül van, mert a többi, alább levő cüchöz nem férhetsz hozzá! Ennek megfelelően tényleg egyszerű a kezelése, mert lényegében

csak 2 művelete van: betehetsz egy adatot a verembe, és kiveheted onnan! És természetesen ami adatot legelsőnek raktál bele, azt veheted ki utoljára.

A legtöbb programozási nyelv nem ad beépített lehetőséget veremváltozók kezelésére - mi mégis megoldjuk ezt. Nem mintha olyan szörnyen fontosak lennének, mert ugye akinek mégis kell, megcsinálhatná magának a maga programjában. Mégis megcsináljuk, egyszerűen amiatt, mert mindenféleképp szükséges nekünk magához az interpreterhez, azért, hogy csinálhassunk olyan cuki kis programvezérlő szerkezeteket a későbbiekben, mint például a for-next páros.

Mindenekelőtt alkossunk egy VEREM osztályt, ami valahogy efféleképp kezdődik majd:

```
class VEREM {
private:
    unsigned int veremmutato; // Ha 0, a verem üres, azaz mindig az első szabad bájtra mutat.
    // A verem tehát „alulról felfelé” telítődik! Kezdetben az értéke 0.
    unsigned int vmax; // A veremnek lefoglalt memóriaterület max hossza. Kezdetben ez is 0.
    // Mindig igaz kell legyen, hogy veremmutato < vmax
    USC *v; // Ez mutat a verem számára lefoglalt memóriaterületre

    // Privát tagfüggvények:

    void verembepakol(V& W,unsigned char size);

public:
    VEREM(unsigned int vm); // konstruktor
    VEREM(); // alapértelmezett konstruktor
    ~VEREM(); // destruktorktor

    unsigned char operator=(unsigned char c);
    signed char operator=(signed char c);
    unsigned int operator=(unsigned int c);
    signed int operator=(signed int c);
    unsigned short int operator=(unsigned short int c);
    signed short int operator=(signed short int c);
    unsigned long long operator=(unsigned long long c);
    signed long long operator=(signed long long c);
    float operator=(float c);
    double operator=(double c);
    long double operator=(long double c);

    operator unsigned char();
    operator signed char();
    operator unsigned int();
    operator signed int();
    operator unsigned short int();
    operator signed short int();
    operator unsigned long long();
    operator signed long long();
    operator float();
    operator double();
    operator long double();
}
```

A konstruktor és destruktork megvalósítása:

```
VEREM::VEREM(unsigned int vm) { // VEREM osztály konstruktora
    veremmutato=0;vmax=vm;v = new unsigned char [vm];}
// .....
VEREM::VEREM() { // VEREM osztály alapértelmezett konstruktora
    veremmutato=0;vmax=0;v = NULL;}
// .....
// VEREM::~~VEREM() { // VEREM osztály destruktora
    if(v != NULL) {delete[] v;} v=NULL;veremmutato=vmax=0;}
```

Az „=” operátor túlterhelésével oldjuk meg az értékadást a veremnek, látható. Ezen függvényekből csak egyet mutatok be illusztrációnak:

```

unsigned short int VEREM::operator=(unsigned short int c) {V W; unsigned char size; size=sizeof(c);
if((veremmutato+size)>vmax) {L("Veremtúlcsoordulás! Típusméret: %u",size);EXITFAILURE();}
W.i[0]=c;verembepakol(W,size);
return c;
}

```

Ezek segédfüggvénye:

```

void VEREM::verembepakol(V& W,unsigned char size) { // A VEREM osztály privát tagfüggvénye
register unsigned int i;for(i=0;i<size;i++) {v[veremmutato++]=W.c[i];}
}

```

Érdekességképpen megjegyzem, hogy eredetileg e segédfüggvény nem létezett, ez fixen bele volt írva mindegyik **operator=** függvénybe! Ekkor a lefordított binárisom mérete 124024 bájt volt. Amikor azonban ezt kiszedtem belőlük, s egyetlen privát tagfüggvényként valósítottam meg mint a fenti példán látható, a mau interpreter lefordítva, binárisként már csak 123725 bájt méretű lett, azaz e trükkal spóroltam 299 bájtot, ami jócskán több mint negyed kilobájt! És szerintem a kód is áttekinthetőbb lett.

Azután a kódot nézegetve rájöttem, hogy ezt a sort:

```

if((veremmutato+size)>vmax) {L("Veremtúlcsoordulás! Típusméret: %u",size);EXITFAILURE();}

```

szintén felesleges beleírnom mindegyik **operator=** tagfüggvénybe, elég ha egyszer szerepel, a „verembepakol” függvényben! Áthelyeztem oda, s íme, máris megint csökkent a bináris mérete újabb 480 bájttal! Azaz pusztán ezek a két kis trükkel háromnegyed kilobájtot spóroltam meg a kódméretben! Ami az eredeti méret 0.6%-a. Ez így talán csekélységnek tűnhet, de gondoljunk csak bele: Több mint fél százalék! Ha csak 20 alkalommal is ügyel az ember efféle apró nüansznyi részletekre, máris simán megspórolhat akár 10 százalékot is a kódméreten, és ezért egyáltalán semmit se kell feladni a funkcionalitásból!

A veremből való érték kiszedést egyszerűen a casting operátorok meghatározásával oldottam meg, mint látható. Egy példa egy ilyenre:

```

VEREM::operator signed short int() {V W; unsigned char size;// casting operátor függvény!
size=sizeof(signed short int);verembolkiszed(W,size);return W.I[0];}

```

És ezen casting operátorok segédfüggvénye:

```

void VEREM::verembolkiszed(V& W,unsigned char size) { // A VEREM osztály privát tagfüggvénye
if(size>veremmutato) {L("Veremalulcsordulás! Típusméret: %u",size);EXITFAILURE();}
register unsigned int i;for(i=size;i>0;i--) {W.c[i-1]=v[--veremmutato];}
}

```

Ugye, nem is bonyolultak... de hát megjósoltam már e könyv elején, hogy kicsi kis függvényecskéket írunk majd...

Mindezt azonban egyelőre csak mi használhatjuk programírás közben. Ahhoz hogy ez a mau nyelven írt programokban is rendelkezésünkre álljon, be kell vennünk (be kell vezetnünk) egy új változótípust oda! Ugye, eddig minden típust külön karakterrel azonosítottunk: c,C,i,I,l,L,g,G,f,d,D - most ezek mellé fel- veszünk egy újabb típusjelölőt, ennek a jele az lesz, hogy „v”. Azért épp „v”, mert e betűvel kezdődik a verem-szó, s eképp ez könnyen megjegyezhető. Elébbemenve a kötekedő ~~trükk~~ szörszálhasogató kritikusoknak, sietve megjegyzem, nem holmi magyarkodásból használom e betűt, s nem pedig az „s” karaktert ami a

„nemzetközi”, azaz inkább angol „stack” szó kezdőkaraktere lenne, hanem amiatt, mert lesznek nekünk később stringjeink is, és azok jele lesz majd az „s” karakter. Stringeket ugyanis vélhetőleg gyakrabban használunk majd, mint vermeket... Ezenfelül a „v” karakter amiatt is jó, mert ez ugye kinézetre egy lefelé mutató nyílhegyre hasonlít, ami vizuálisan utal arra, hogy mi „lefelé”, a „mélybe” teszünk valamit, vagyis egy gödör, egy verem aljába dobjuk le/bele, a „mélybe”...

Vegyük fel ezt az F struktúrába:

```
VEREM verem[256];
```

Minthogy a VEREM osztály adatterülete egyelőre (64 bites gépen!) 16 bájtot foglal el, e 256 elemű tömb összesen 4 kilobájt memóriefogyasztást indukál. Veremből szerintem nem volna szükség ennyire, de ez nem olyan szörnyű nagy pazarlás, és érdemes 256 lehetőséget fenntartani a vermeinknek is, hogy kompatibilisek maradjunk a „rendes”, „normális” változóink darabszámával. Különbösen is, élek a gyanúperrel, ha elkezdenék variálni a kóddal hogy kevesebb lehetőség legyen vermek számára, s emiatt mindig ellenőrizgetni kéne az index méretét, az ezeket megvalósító extra kódmennyiség egyhamar maga is 4 K fölé nőne, akkor meg semmit nem értem az egészel.

Ne feledjük azonban, e vermeink kezdetben még nemcsak üresek, olyan értelemben hogy nincs bennük semmiféle adat, de ráadásul még csak hely sincs lefoglalva nekik arra, hogy bármiféle adatot is képesek legyenek befogadni! Kell utasítást csinálnunk arra, hogy egy veremnek adatterület legyen lefoglalva. A lefoglalt terület felszabadításával nem kell törődnünk, arra ha akarunk csinálhatunk éppenséggel utasítást, de lényegében felesleges, mert a destruktor úgyis elvégzi helyettünk ezt a munkát.

Minthogy nekünk már úgyis van egy **M** nevű utasításunk, ami lefoglal bizonyos memóriaterületet adattárolási célokra, teljesen logikus hogy a vermeknek memóriát lefoglaló utasítás neve **MV** legyen. Ennek 2 paramétere kell legyen, mindegyik egy-egy aritmetikai kifejezés:

MV (a verem neve) (a verem számára szükséges terület bájtban)

A „verem neve” természetesen unsigned char típusú aritmetikai kifejezés kell legyen, a szükséges terület nagyságát meghatározó aritmetikai kifejezés pedig unsigned int.

Ehhez azonban meg kell írunk a VEREM osztály memórialefoglaló függvényét:

```
void VEREM::veremmemorialefoglal(unsigned int mekkora) {
    if(v != NULL) {
        L("Memóriát akartál lefoglalni olyan veremnek, aminek területét korábban már lefoglaltad, és nem szabadítottad fel!");
        EXITFAILURE();
    }
    veremmutato=0; vmax=mekkora; v = new unsigned char [mekkora];
}
```

Ha azonban ez kész van, simán leegyszerűsíthetjük a konstruktorát:

```
VEREM::VEREM(unsigned int mekkora) { // VEREM osztály konstruktora
    v=NULL; veremmemorialefoglal(mekkor);
}
```

Ezek után az MV utasításunk:

```
int fuggveny_MV(F& f) {USC index; unsigned int mekkora;  
index=ERTEKunsignedchar(f);mekkora=ERTEKunsignedint(f);  
f.verem[index].veremmemorialefooglal(mekkora);  
return 0;  
}
```

Kell csinálnunk egy **v#** utasítást, az **=#** utasítás mintájára:

```
int vkereszt(F& f) { // v# utasítás, érték betétele veremtárba  
// Szintaxis: v# (érték típusát jelölő karakter) (a verem azonosítója) (érték)  
USC index;USC változotípus;F *ff;  
ff=&f;  
if(f.parentflag==1) {  
if(f.CHILD==NULL) {L("Érvénytelen, NULL értékű CHILD névtér! Pozíció: %lu",f.P);EXITFAILURE();}  
ff=&f.CHILD->f;  
if(ff==NULL) {L("Érvénytelen, NULL pointerű F struktúra a CHILD névtérben! Pozíció: %lu",f.P);EXITFAILURE();}  
}  
if(f.parentflag==255) {if(f.parentF==NULL) {L("Érvénytelen, NULL értékű szülőnévtér! Pozíció: %lu",f.P);EXITFAILURE();}  
ff=f.parentF;}  
  
változotípus=vtype(f); //rögtön a következő karakter a v# után!  
namespace(f);  
index=ERTEKunsignedchar(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat  
namespace(f);  
switch(vváltozotípus) {  
case 'c': (*ff).verem[index]=ERTEKunsignedchar(f);break;  
case 'C': (*ff).verem[index]=ERTEKsignedchar(f);break;  
case 'i': (*ff).verem[index]=ERTEKunsignedshortint(f);break;  
case 'I': (*ff).verem[index]=ERTEKsignedshortint(f);break;  
case 'l': (*ff).verem[index]=ERTEKunsignedint(f);break;  
case 'L': (*ff).verem[index]=ERTEKsignedint(f);break;  
case 'f': (*ff).verem[index]=ERTEKfloat(f);break;  
case 'g': (*ff).verem[index]=ERTEKunsignedlonglong(f);break;  
case 'G': (*ff).verem[index]=ERTEKsignedlonglong(f);break;  
case 'd': (*ff).verem[index]=ERTEKdouble(f);break;  
case 'D': (*ff).verem[index]=ERTEKlongdouble(f);break;  
} // switch vége  
return 0;  
}
```

Ennek szintaxisa természetesen a következő:

v#x index érték

Ahol az „x” azt mondja meg, milyen típusú adatot óhajtunk a verembe rakni. Ez az „x” egy karakter a már ismert típusjelölőink közül, azaz lehet a c,C,i,I,l,L,g,G,f,d,D közül bármelyik. Az „index” pedig a verem neve, ami ugye egy 0-255 közt értelmezhető azaz unsigned char típusú aritmetikai kifejezés, az „érték” pedig az a szám, amit beteszünk a verembe, na ezen „érték” típusa kell legyen az, amit az „x” meghatároz.

Sajnos ez is csak akkor működik, ha a PGM osztályban „friend”-ként jelöljük meg. Igenám, de nem szeretjük ha sok ott a friend... Különbö meg a vak is látja hogy ennek eleje hajszálla megegyezik az **=#** utasításunk elejével! Nosza, szervezzük ki ezt egy külön rutinba, és ezek után csak azt kell friendként megjelölni! Íme a rutin:

```
F *PGMfriend_for_f(F& f) {F *ff;ff=&f;  
if(f.parentflag==1) {  
if(f.CHILD==NULL) {L("Érvénytelen, NULL értékű CHILD névtér! Pozíció: %lu",f.P);EXITFAILURE();}  
ff=&f.CHILD->f;  
if(ff==NULL) {L("Érvénytelen, NULL pointerű F struktúra a CHILD névtérben! Pozíció: %lu",f.P);EXITFAILURE();}  
}  
if(f.parentflag==255) {if(f.parentF==NULL) {L("Érvénytelen, NULL értékű szülőnévtér! Pozíció: %lu",f.P);EXITFAILURE();}  
ff=f.parentF;}  
  
return ff;
```



```
}
```

Ezek után a **v#** eleje így néz ki:

```
int vkereszt(F& f) { // v# utasítás, érték betétele veremtárba
// Szintaxis: v# (érték típusát jelölő karakter) (a verem azonosítója) (érték)
USC index;USC változotípus;F *ff;
ff=PGMfriend_for_f(f);
változotípus=vtype(f); //rögtön a következő karakter a v# után!
```

Az **=#** eleje pedig eként:

```
int egyenlokereszt(F& f) {USC index;USC unionindex;USC változotípus;F *ff;
ff=PGMfriend_for_f(f);
változotípus=vtype(f); //rögtön a következő karakter
```

Mindjárt minden egyszerűbb... És természetesen, a **v#** utasításunk is épp olyan szabályok szerint működik a **{** utasítás után, az új névtérbe való belépéskor, mint az **=#** utasításunk.

Ezzel persze még nincs vége a mókának: fontos hogy a veremből KI IS TUDJUK OLVASNI az oda betett adatot, azaz hogy szerepeltethessük aritmetikai kifejezésben! Ehhez mindekelőtt bővítenünk kell a vtype függvényünket, hogy érvényes típusnak ismerje el a „v” karaktert:

```
USC vtype(F& f) {USC v;v=f.p[f.P];
switch(v) {
case 'c': case 'C': case 'i': case 'I': case 'l': case 'L': case 'g': case 'G': case 'f': case 'd': case 'D':
case 'v': k(f);break;
default : return ERROR(f,3);break;
} // switch vége
return v;
}
```

Ezután pedig az **ERTEKunsignedchar**, **ERTEKsignedshortint** stb függvényeinket kell egy picit kibővítenünk, így, ahogy azt az egyiken bemutatom, kékkel kiemelve a pótlólag beszúrt részt:

```
double ERTEKdouble(F& f) { // A következő „Aritmetikai kifejezés” értékét adja vissza double számként.
USC karakter;USC változotípus;USC unionindex;double tort;char *vege;double A;
unionindex=255;változotípus='d'; // Alapértelmezett index és típus
nemspace(f);karakter=f.p[f.P];
if(karakter=='(') {k(f);A= ERTEKdouble(f);nemspace(f);karakter=f.p[f.P];
if(karakter!=')') {L("Hiányzó csukózárójel!");return ERROR(f,3);}
k(f);goto ERTEK_d_OPERATOR;
} // Nyitózárrójel-teszt vége
if(karakter=='#') {változotípus=k(f);k(f);
switch(vváltozotípus) {
case 'c': A= (double)ERTEKunsignedchar(f); break;case 'C': A= (double)ERTEKsignedchar(f); break;
case 'i': A= (double)ERTEKunsignedshortint(f);break;case 'I': A= (double)ERTEKsignedshortint(f); break;
case 'l': A= (double)ERTEKunsignedint(f); break;case 'L': A= (double)ERTEKsignedint(f); break;
case 'f': A= (double)ERTEKfloat(f); break;case 'g': A= (double)ERTEKunsignedlonglong(f);break;
case 'G': A= (double)ERTEKsignedlonglong(f); break;case 'd': A= (double)ERTEKdouble(f); break;
case 'D': A= (double)ERTEKlongdouble(f); break;
case 'v': nemspace(f);karakter=f.p[f.P];if(karakter=='@') k(f); // Veremváltozó esetén átugorja a kukac jelet
A= (double)(f.verem[ERTEKunsignedchar(f)]);break;
default: return ERROR(f,3);break;
} // switch vége
goto ERTEK_d_OPERATOR;} // A # teszt vége
karakter=kettospontteszt(f,unionindex,vváltozotípus); // Az aktuális feldolgozandó karakter
if(karakter=='@') {switch(vváltozotípus) {
case 'c': A= (double)vc(f,unionindex);break;case 'C': A= (double)vC(f,unionindex);break;
case 'i': A= (double)vi(f,unionindex);break;case 'I': A= (double)vI(f,unionindex);break;
case 'l': A= (double)vL(f,unionindex);break;case 'L': A= (double)vL(f,unionindex);break;
case 'f': A= (double)vf(f,unionindex);break;case 'g': A= (double)vg(f,unionindex);break;
case 'G': A= (double)vG(f,unionindex);break;case 'd': A= (double)vd(f,unionindex);break;
case 'D': A= (double)vD(f);break;
} // switch vége
goto ERTEK_d_OPERATOR;} // változóbeolvasás vége
switch(karakter) {
case '?': A= (double)systemvariable(f);goto ERTEK_d_OPERATOR; break; // Rendszerváltozó tartalmát kell visszaadni
case 194: A= (double)cinkeertekviisszaad(f);goto ERTEK_d_OPERATOR; break; // $ teszt vége
case '!': A= felkialtozajeld(f);goto ERTEK_d_OPERATOR; break; // Memóriában tárolt érték beolvasása
```

```

} // switch karakter vége
// Különben itt valami számféleségnek kell állnia
if((karakter=='$')||(karakter=='%')||(karakter=='o')||(karakter=='0')||(karakter=='\'))
    A= (double)number(f.p,f.P,f.phossz);
tort = strtod((char *) (f.p + f.P), &vege); f.P=vege - (char *)f.p;
A= tort;
ERTEK_d_OPERATOR:
nemspace(f); // ugrás a következő értékes karakterre
karakter=f.p[f.P]; // P most a beolvasott műveleti jelre mutat
A=Operator(A,karakter,f);
return A;
}

```

És ezek után már tudunk a mau programunkban veremeket is kezelni, amint azt bemutatom e kis példán:

```

MV x 2000 // 2000 bájtot lefoglalunk az „x” nevű verem számára
v#c x 4    // Beteszünk az „x” nevű verembe 4-et
v#c x 5    // Beteszünk az „x” nevű verembe 5-öt
v#c x 18   // Beteszünk az „x” nevű verembe 18-at
v#c x 20   // Beteszünk az „x” nevű verembe 20-at

v#i x $fce2 // Beteszünk a verembe egy 2 bájtos hexa számot is, ez a 64738

// Kiírás

?c #v@x ; / // Itt kiírom a hexa fce2 2 bájta közül a felső bájtot, értéke 252
?c #v@x ; / // Itt kiírom a hexa fce2 2 bájta közül az alsó bájtot, értéke 226
?c #v@x ; / // Kiírom a 20-at
?c #v@x ; / // Kiírom a 18-at
?c #v@x ; / // Kiírom az 5-öt
?c #v@x ; / // Kiírom a 4-et

// Most a verem kiürült. Rakunk bele megint számokat:

v#c x 4      // 4-et
v#i x $fce2  // 64738-at, ez 2 bájtos
v#c x 20     // 20-at

?c #v x ; / // Kiírom a 20-at
?i #v x ; / // Kiírom a 64738-at, mert most unsigned short intként vettem ki a veremből
?c #v x ; / // Kiírom a 4-et

```

Futásának eredménye:

```

252
226
20
18
5
4
20
64738
4

```

A példa a kommenteknek hála magáért beszél, legfeljebb annyit teszek hozzá, hogy mint látható, verem esetén MINDIG meg kell adnunk a # jellel típusként, hogy ő verem. Különben a kifejezésértékelő függvény nem tudná ezt róla, és közönséges változóként kezelné. Ellenben ha a #v castolási operátor már szerepel, akkor a következő kifejezést okvetlenül veremindexként értelmezi, azaz az ezután esetleg szereplő @ jelet átugorja. Helyesebbnek tartom azonban ha ott áll a kukac jel, mert így jobban látszik, mi tartozik egy veremváltozóhoz.

Végül pedig felülkerekedtem a lustaságomon, és kreáltam egy utasítást a verem lenullázására is, azaz arra, hogy ha korábban foglaltunk le neki területet, azt felszabadíthassuk:

```
void VEREM::veremdelete(void) {if(v != NULL) {delete[] v;} v=NULL;veremmutato=vmax=0;}
```

Ezek után a destruktorként ilyen egyszerű lett:

```
VEREM::~VEREM() {veremdelete();} // VEREM osztály destruktora
```

Az ezeket meghívó mau parancs:

```
int v0(F& f) {f.verem[ERTEKunsignedchar(f)].veremdelete();return 0;} // verem lenullázása
```

Ez tényleg egyszerű... Természetesen így kell használni mondjuk az „x” nevű veremre:

v0 x

Az utasítástoken nevének logikája az, hogy a **veremnek** ezután **0** lesz a mérete.

Értelmes dolognak tűnik még egy függvényt készíteni, ami azt mondja meg nekünk, van-e egyáltalán még adat a veremünkben, vagy az már üres! E célra kiválóan alkalmasnak látszik a korábbi „?” függvényünk, mellyel mindenféle „rendszerváltozókat” csekkolhatunk le. Ezt bővítsük ki e sorral:

```
case 'v': value=(USIL)f.verem[ERTEKunsignedchar(f)].vanebenne(); break;
```

Az ehhez szükséges **vanebenne** függvény:

```
unsigned char VEREM::vanebenne(void) {if(veremutato) return 1; else return 0;}
```

E ezt például eként használhatjuk:

```
if(?v x) T "Az „x” nevű verem nem üres 3." /  
E "Az „x” nevű verem üres" /
```

Az is hasznos lehet, ha konkrétan is le tudjuk kérdezni, van-e lefoglalva a veremnek valamekkora terület, s ha igen, mennyi! Ezt oldjuk meg a

?S x

függvénnyel, ahol az x egy unsigned char típusú aritmetikai kifejezés, s a verem nevét határozza meg. A függvény visszaadja azt a darabszámot, amennyi érték számára le lett foglalva veremterület. Ha nem lett még lefoglalva, nulla az amit visszaad.

Példa a használatára:

```
"Az x verem mérete: " ?l ?S x; /;
```

15. fejezet: Fejlett vezérlési szerkezetek

Korábban már láttuk, miként valósíthatunk meg a jelenlegi eszközeinkkel ciklust. Minthogy van lehetőségünk elágazásra az **if** paranccsal, meg van „goto” utasításunk a » parancs képében, ezért szigorúan véve tulajdonképpen nincs is szükségünk semmi másra ezeken kívül. Na de hát ez azért nem nagyon „felhasználó-programozó-barát”...

Más programnyelvekben többfajta ciklusféleségre van lehetőség. Mindenféle szintaxissal. Lényegében azonban mindegyik besorolható mindössze 2 kategóriába, s ezek az „előltesztelő” és a „hátteltesztelő”. Ezek a következő blokk-sémát követik vázlatosan:

ELŐLTESZTELŐ CIKLUS:

(előkészület a ciklushoz)

(Feltételvizsgálat: ha a feltétel igaz, végrehajtom a ciklusmagot)

{CIKLUSMAG}

(visszaugrás a feltételvizsgálathoz)

A program további része

HÁTULESZTELŐ CIKLUS:

(előkészület a ciklushoz)

(Ciklus innen indul. Ezt a helyet megjegyzi valamiképp a program)

{CIKLUSMAG}

(Feltételvizsgálat. Ha a feltétel igaz, visszaugrik a ciklus elejére)

A program további része

Ez a 2 variáció van csak. Semmi több! A LÉNYEGI különbség ezek közt, hogy a hátultesztelő ciklusok esetében a ciklusmag MINDENKÉPPEN VÉGREHAJTÓDIK LEGALÁBB EGY ALKALOMMAL, az előltesztelők esetében azonban nem okvetlenül.

Fontos megérteni, hogy az „előltesztelő” névnek a világon SEMMI KÖZE ahhoz, hogy a tesztelendő feltétel a programkódban miféle szabályok szerint hova van írva: a ciklusmag elé vagy után, s hasonlóképp teljesen mindegy ez a hátultesztelő ciklusok esetében is. Az ilyesmi csak csicsa, csak külalak. A C nyelv **for** ciklusánál például az egész mindenség a ciklus legelejére van írva, s a ciklusmagot csak a zárójelek határolják.

Ha már a **for** ciklusnál tartunk: az is teljesen ezen kaptafára épül, csak ott a „feltétel” amit vizsgálgatunk, egy speciális változó értékével kapcsolatos, mely változó nagyságát csökkentjük vagy növeljük a ciklusmagon belül.

Ami egyaránt közös mindegyik típusban, az holtbiztos hogy az, hogy valamiképp meg kell jegyezze a ciklusmag elejét, hiszen oda neki vissza kell tudnia térnie majd! Na most erre természetesen az a megoldás, hogy meg kell jegyezze a programkód ottani címét. Igenám, de miként is jegyezze meg ezt?

Erre is 2 megoldás létezik. Az egyik a „címkézéses” megoldás, a másik a „veremtárra épülő”.

A címkézéses egyszerű: van egy címke a ciklusmag elejénél, s ha a ciklus végéhez ér az interpreter, s úgy véli meg kell ismételje a ciklust, hát ott is meg van adva valamiképp e címke neve, s ebből tudja hova kell ugrania.

A veremtárra épülő ellenben úgy működik, hogy a ciklus elejénél beteszi a címet a veremtárba, majd a ciklus végén onnan veszi ki, megtudandó, hova kell ugrania.

Lássuk is ezt az utóbbi variációt, hiszen van már lehetőségünk vermeket használni! Ehhez kiindulásként vegyünk fel egy rendszervermet az F struktúrába:

VEREM CIKLUSVEREM;

A PGM konstruktorait bővítsük ki ezzel:

```
f.CIKLUSVEREM.veremmemorialefooglal(CIKLUSVEREMSIZE);
```

A destruktort meg ezzel:

```
f.CIKLUSVEREM.verendelete();
```

Elvileg ugyan a fenti veremmemóriatörlést végrehajtja automatikusan a VEREM osztályunk destruktora, ami illik hogy meghívott legyen a PGM destruktornak meghívása közben, de jobbnak tartom ha az ilyen huncutságokat explicite is jelöljük. Ebből baj nem lehet, mert a VEREM destruktort eleve úgy írtuk meg, hogy csak akkor szabadítja fel a lefoglalt veremmemóriát, ha az egyáltalán le lett már foglalva.

A CIKLUSVEREMSIZE értékét egy **#define** direktívával határozzuk meg, hogy könnyen változtatható legyen:

```
#define CIKLUSVEREMSIZE 8000
```

Ezután csináljunk egy ilyen függvényt:

```
int nyitokapcsosnyitorendeszarojel(F& f) {f.CIKLUSVEREM=f.P;return 0;}
```

Meg egy ilyet is:

```
int csukosimacsukokapcsoszarovel(F& f) {USIL regip;USC feltetel;  
regip=f.CIKLUSVEREM;nemspace(f);feltetel=ERTEKunsignedchar(f);  
if(feltetel) {f.P=regip;f.CIKLUSVEREM=regip;}  
return 0;  
}
```

S máris készen vagyunk, van egy pompás hátultesztelő ciklusunk! Azaz, ez egyszer mindenképpen lefut. A használatát gyönyörűen bemutatja az alábbi kis példaprogram:

```
=#i@W 4 // unsigned int W=4  
{( // Kezdődik a külső ciklus  
=#i@K 5 // unsigned int K=5  
{( // Kezdődik a belső ciklus  
"W=" ?i @W " , K=" ?i @K " , W*K=" ?i (@W)*(@K) /; // Kiiratás  
+#i@K 1 // K+=1  
)} ((#i@K) <= 8) // Ha (K <= 8) folytatja a ciklust  
+#i@W 1 // W+=1  
)} ((#i@W) <= 10) // Ha (W <= 10) folytatja a ciklust  
XX // Végeért a program
```

Amint a fenti példán látható, a ciklusaink egymásba ágyazhatóak. Minthogy jelenleg az **f.P** programmutatónk típusa olyan, hogy 4 bájt a mérete, ezért a 8000 bájt elemű ciklusvermünk „csak” 2000 mélységben engedi a ciklusainkat egymásba ágyazni... Ezen változtathatnánk a CIKLUSVEREMSIZE megnövelésével, de azt hiszem ez is bőven elég lesz... Nem mellékesen megjegyzem, hogy a ciklusverem az F struktúra része, emiatt ha új névtérbe lépünk be a {{ paranccsal,

akkor új ciklusverem jön létre, ami azt jelenti, hogy ott újra egymásba skatulyázható újabb maximum 2000 ciklus...

A fenti program futásának eredménye:

```
W=4 , K=5, W*K=20
W=4 , K=6, W*K=24
W=4 , K=7, W*K=28
W=4 , K=8, W*K=32
W=5 , K=5, W*K=25
W=5 , K=6, W*K=30
W=5 , K=7, W*K=35
W=5 , K=8, W*K=40
W=6 , K=5, W*K=30
W=6 , K=6, W*K=36
W=6 , K=7, W*K=42
W=6 , K=8, W*K=48
W=7 , K=5, W*K=35
W=7 , K=6, W*K=42
W=7 , K=7, W*K=49
W=7 , K=8, W*K=56
W=8 , K=5, W*K=40
W=8 , K=6, W*K=48
W=8 , K=7, W*K=56
W=8 , K=8, W*K=64
W=9 , K=5, W*K=45
W=9 , K=6, W*K=54
W=9 , K=7, W*K=63
W=9 , K=8, W*K=72
W=10 , K=5, W*K=50
W=10 , K=6, W*K=60
W=10 , K=7, W*K=70
W=10 , K=8, W*K=80
```

Ez szép munka volt, pompás funkcióval bővült a rendszerünk abszolút minimális kódolás árán! Nézzük most az előtesztelő ciklust, mert ugye szükségünk lehet olyan ciklusfajtára is, ami esetleg egyszer se kell hogy lefusson!

Az előtesztelő ciklussal az a baj, hogy amennyiben tényleg nem kell egyszer se lefutnia, akkor azzal kell kezdenie, hogy azonnal elugrik a végére. Igenám, de a végén még addig sosem járt! Honnan tudhatja a szerencsétlen, hogy hová kell hát ugrania, a program mely pontjára?!

Nincs más megoldás, azt bizony meg KELL jelölnünk már jóelőre egy címkével! Ráadásul e címke értékét szerepeltetnünk kell valamiképp a ciklus fejrészában, hiszen ott már tudnia kell, hova fog ugorni ha muszáj, sőt ugyanott kell szerepeljen a feltétel is, aminek teljesülésétől függ, hogy ugrik-e!

Mindez meglehetősen bonyolult szintaktikát igényel, hovatovább jó lenne az egész ciklust még valamiképp zárójelezni is, hogy az egész miskulancia áttekinthetőbb legyen! Bevallom, itt viszonylag sokat lustálkodtam, míg kiötöltem valamit amit elfogadhatónak tartok.

Mindenekelőtt a fenti hátulatesztelő ciklus szintaktikáján SZIGORÍTSUNK egy picit! A `}}` utasítás csak és kizárólag akkor értékeljen ki feltételt, ha éppen pontosan egy nyitózároljel követi őt magát, bármiféle whitespace megelőzése nélkül! Azaz így:

```
}}(feltétel)
```

De így már nem:

```
}} (feltétel)
```

Az utóbbi esetben semmiféle feltételt nem értékel ki, hanem mindenféleképpen visszaugrik a ciklus elejére. Ez első pillantásra hülyeségnek tűnik, mert mi a

fenének szigorítani?! Holott nagy értelme van ám ennek, mert így csinálhatunk könnyedén „örökciklusokat”, amikből csak úgy lehet kijutni, hogy a cikluson belül gondoskodunk valami feltétel vizsgálatáról. Ezesetben persze illik, hogy a kiugráskor töröljük a ciklusverem legfelső értékét, mert nem jó ám az, ha felesleges szemetet hagyunk a veremben... Ezt legegyszerűbb úgy elvégezni, hogy csinálunk 2 utasítást, a **then T** és az **else E** párját, amik pontosan úgy működnek mint elődeik, épp csak közben elvégzik e veremtisztítást is nekünk! Íme:

Az új **}}** utasításunk:

```
int csukosimacsukokapcsoszarozel(F& f) {USIL regip;USC feltetel;
regip=f.CIKLUSVEREM;
if(f.p[f.P]!='(') goto csukosimacsukokapcsoszarozel_vege;
namespace(f);feltetel=ERTEKunsignedchar(f);
if(feltetel==0) return 0;
csukosimacsukokapcsoszarozel_vege:
f.P=regip;f.CIKLUSVEREM=regip;
return 0;
}
```

A „speciális then és else” utasításaimnak egyszerűen a **TT** illetve az **EE** neveket adtam. Ez könnyen megjegyezhető, mert mindegyik a **T** illetve **E** párja. Ezeknek szükségük van egy olyan veremfüggvényre, ami töröl a verem tetejéből valahány bájtot. Ezt a **-=** operátorra terheltem rá:

```
void VEREM::operator-=(unsigned char c) { // Töröl a verem tetejéről „c” darab bájtot
if(c>veremmutato) {L("Veremalulcsordulás! Típusméret: %u",c);EXITFAILURE();}
veremmutato-=c;
}
```

A **TT** és az **EE** függvény:

```
int fuggveny_TT(F& f) {USC karakter;
namespace(f); // whitespace karakterek átugrása
karakter=f.p[f.P]; if(karakter!=194) return(3); // Ha nem a » első bájtja, syntax error
karakter=k(f); if(karakter!=187) return(3); // Ha nem a » második bájtja, syntax error
f.P--;
if(f.ifflag==0) {ujsorig(f);return 0;}
f.CIKLUSVEREM=(sizeof(f.P));
return 0;
}
// -----
int fuggveny_EE(F& f) {USC karakter;
namespace(f); // whitespace karakterek átugrása
karakter=f.p[f.P]; if(karakter!=194) return(3); // Ha nem a » első bájtja, syntax error
karakter=k(f); if(karakter!=187) return(3); // Ha nem a » második bájtja, syntax error
f.P--;
if(f.ifflag) {ujsorig(f);return 0;}
f.CIKLUSVEREM=(sizeof(f.P));
return 0;
}
```

És így működnek:

```
=#i@W 400 // unsigned int W=4
{( // Kezdődik a ciklus
+#i@W 1 // W+=1
"W = " ?i @W /; // kiiratás
if ((#i@W) > 410) TT »$in // Ha (W <= 410) folytatja a ciklust
)}
$in
"Vége!" /
XX // Végetért a program
```

A progi outputja:

```
W = 401
W = 402
W = 403
W = 404
W = 405
W = 406
W = 407
W = 408
W = 409
W = 410
W = 411
Vége!
```

Amint az a kódból kiderül, a **TT** és az **EE** szigorúan ellenőrzi, hogy őutánuk okvetlenül CSAK és KIZÁRÓLAG pontosan a „**»**” jel állhat! (előtte akárhány whitespace karakterrel). Ennek az az oka, hogy míg a **T** és **E** lehet többsoros, addig a **TT** és **EE** már egyáltalán nem, mert gondoljuk csak el, mi történne, ha többsorosak volnának, és mindegyik megkísérelne törölni a verem tetejéről valahány bájtot, mint szerinte már felesleges visszatérési címet... Természetesen így se lehet kiküszöbölni minden hibalehetőséget, de olyan programnyelv még nem született, amely eleve lehetetlenné tesz minden hibaelkövetést. Ez amit ide beleépítettem, nem igazán erőforrásigényes, ugyanakkor mégis nyújt valamekkora biztonságot.

Ez mind nagyon szép, de még mindig nincs előltesztelő ciklusunk! Nos, annak érdekében hogy legyen, picit módosítsunk a ciklus elején, a **{{** utasításon! Mindenekelőtt, ne az aktuális f.P értéket tegye be a verembe, hanem annál kettővel kevesebbet. Emellett a **}}** parancsból szedjük ki azt a részt, ami visszapakolja a veremtárba a visszatérési címet. Továbbá... De lássuk is inkább a kódot a magyarázat előtt!

```
int csukosimacsukokapcsoszarozel(F& f) {USIL regip;USC feltetel;
regip=f.CIKLUSVEREM;
if(f.p[f.P]!='(') goto csukosimacsukokapcsoszarozel_vege;
nemspace(f);feltetel=ERTEKunsignedchar(f);
if(feltetel==0) return 0;
csukosimacsukokapcsoszarozel_vege:
f.P=regip;
return 0;
}
// -----
int nyitokapcsosnyitorendeszarozel(F& f) { // Ciklusfejléc bevezető utasítása
USIL paragrafuscim;

f.CIKLUSVEREM=f.P-2; // Berak a verembe kettővel kevesebbet, mint az aktuális programmutató.
// Most a veremben az a programmutató cím van, ami a „{” utasítástoken „{” karakterére mutat.

if(f.p[f.P] != 194) return 0; // Ha nem a § jel első bájtja a következő karakter
if(f.p[f.P+1] != 167) return 0; // Ha nem a § jel második bájtja a következő karakter
// A fenti két esetben hátultesztelő ciklusról van szó.
// Különben előltesztelő ciklus, ami így kezdődik:
// {($x
// vagy így:
// {($xy
// ahol „x” és „y” tetszőleges címkekarakterek lehetnek. Az „y” karakter mindenképp létezik, legfeljebb
// valami whitespace karakter, ami emiatt nyugodtan átugorható. Ezért előltesztelő ciklus esetén e 2 karakter
// után folytatható a kódvégrehajítás

paragrafuscim=f.P; // Eltároljuk a „§” karakter címét.
f.P+=4; // Most az f.P az „y” címkekarakter utáni első bájtra mutat. Ha a címke 1 karakteres, akkor
// e címkekarakter s az f.P jelezte mostani karakter közt 1 db valamilyen whitespace áll, ami nem számít.
if(ERTEKunsignedchar(f)) { // Ha a következő feltétel igaz
// Akkor ugranunk kell a fejlécben specifikált címkére
f.P=paragrafuscim; // Programmutató a paragrafuskarakterre
f.CIKLUSVEREM=(sizeof(f.P)); // Töröljük a veremből az imént belerakott címet
return duplajobbbranyil(f); // és végrehajtjuk az ugróutasítást
} // if vége
return 0; // Különben belemegyünk a ciklusba
}
```


Nem, ez az utóbbi CSEPPET SE HOSSZÚ rutin, csak marha sok benne a komment. Emiatt azonban nem is kell pótlólag magyaráznom a működését, remélem! Lássuk azonban egy példán, hogy „miként is műxi ez magát”!

```
=#i@W 1 // unsigned int W=1

{( // Kezdődik a külső ciklus. Ez hátultesztelő

=#i@K 5 // unsigned int K=5

// Ez itt egy előltesztelő ciklus
{($ki ((#i@K) == 8) // Ha (K == 8) kiugrik a ciklusból a „ki” címkére
// Kezdődik a belső ciklus

"W=" ?i @W " , K=" ?i @K " , W*K=" ?i (@W)*(@K) /; // Kiíratás

+#i@K 1 // K+=1

)}} // Itt végetért a belső ciklus, ami előltesztelő volt
$ki // Erre a címkére kerül a vezérlés az előltesztelő ciklus után

+#i@W 1 // W+=1

)}((#i@W) <= 5) // Ha (W <= 5) folytatja a külső ciklust

XX // Végetért a program
```

A fenti példa 2 ciklust mutat be, melyek egymásba vannak ágyazva. A külső egy hátultesztelő, ami emiatt mindig végrehajtódik legalább egyszer, a belső ellenben egy előltesztelő ciklus. A program eredménye:

```
W=1 , K=5, W*K=5
W=1 , K=6, W*K=6
W=1 , K=7, W*K=7
W=2 , K=5, W*K=10
W=2 , K=6, W*K=12
W=2 , K=7, W*K=14
W=3 , K=5, W*K=15
W=3 , K=6, W*K=18
W=3 , K=7, W*K=21
W=4 , K=5, W*K=20
W=4 , K=6, W*K=24
W=4 , K=7, W*K=28
W=5 , K=5, W*K=25
W=5 , K=6, W*K=30
W=5 , K=7, W*K=35
```

Látható, hogy a belső ciklus a K==8 esetén már nem hajtódik végre. Na most megjegyzem, trükközhetünk is, mert attól hogy a ciklus fejlécét megcsináljuk mint előltesztelőt, ettől még senki nem tiltja meg nekünk, hogy a láblécét mint hátultesztelőt készítsük el, s ekkor is tökéletesen fog működni, legfeljebb tesztel mindegyik helyen:

```
=#i@W 1 // unsigned int W=1

{( // Kezdődik a külső ciklus. Ez hátultesztelő

=#i@K 5 // unsigned int K=5

// Ez itt egy előltesztelő ciklus
{($ki ((#i@K) == 8) // Ha (K == 8) kiugrik a ciklusból a „ki” címkére
// Kezdődik a belső ciklus

"W=" ?i @W " , K=" ?i @K " , W*K=" ?i (@W)*(@K) /; // Kiíratás

+#i@K 1 // K+=1

)}}((#i@K)<=6) // Trükkös fazonok vagyunk, előltesztelő ciklust hátul is tesztelünk...
// Itt végetért a belső ciklus, ami előltesztelő volt
$ki // Erre a címkére kerül a vezérlés az előltesztelő ciklus után

+#i@W 1 // W+=1

)}((#i@W) <= 5) // Ha (W <= 5) folytatja a külső ciklust

XX // Végetért a program
```

Az output:

```
W=1 , K=5, W*K=5
W=1 , K=6, W*K=6
W=2 , K=5, W*K=10
W=2 , K=6, W*K=12
W=3 , K=5, W*K=15
W=3 , K=6, W*K=18
W=4 , K=5, W*K=20
W=4 , K=6, W*K=24
W=5 , K=5, W*K=25
W=5 , K=6, W*K=30
```

Mindez jó meg cucci, de tudjuk, hogy az esetek óriási többségében a ciklusok konkrétan előre meghatározható fix darabszámúak kell lefussanak! Erre nem ártana valami igazán könnyű megoldást találnunk. Nos, mit szólnánk e két utasításhoz?

```
int nyitokapcsoszarozelvonat(F& f) { // Fix darabszámú lefutó ciklus
unsigned long long darabszam; // Ennyiszer fog lefutni a ciklus
darabszam=ERTEKUNSIGNEDLONG(f); // Beolvassuk a darabszámot
if(darabszam==0) {
L("Hátultesztelő fix darabszámú lefutó ciklust nullaszer próbáls végrehajtani! Pozíció: %lu",f.P);EXITFAILURE();
}
f.CIKLUSVEREM=f.P; // Elmentjük a verembe a programmutatót
f.CIKLUSVEREM=darabszam; // Elmentjük a verembe a darabszámot is
return 0; // Belépünk a ciklusmagba
}

// -----
int vonalkapcsoscsukozarozel(F& f) { // Visszatérés fix darabszámú ciklusból
unsigned long long darabszam;
USIL regip;
darabszam=f.CIKLUSVEREM; // Kiolvassuk a darabszámot a veremből
regip=f.CIKLUSVEREM; // Kiolvassuk a visszatérési címet a veremből
darabszam--;
if(darabszam==0) return 0; // Ha nulla, kilépünk a ciklusból
f.CIKLUSVEREM=regip; // Visszatesszük a visszatérési címet a verembe
f.CIKLUSVEREM=darabszam; // Visszatesszük a darabszámot a verembe
f.P=regip; // Programmutató beállítása a visszatérési címre
return 0; // Visszaugrás a ciklusba
}
```

Ezeknek hála van már fix ciklusunk is, mégpedig hátultesztelő. Nincs értelme ebből előltesztelőt csinálni, mert nem hinném hogy gyakori volna az esetleg nullaszer végrehajtandó ciklus esete...

Ja, és így használandóak mint e példa mutatja, ami kiírja az angol ABC kis-betűit:

```
==c@a a // a='a'
{| 26 // 26-szor fut e majd ez a ciklus

? @a // Az „a” változó kiírása karakterként

+#c @a 1 // a+=1

|} // Ciklus vége
/ // Üres sor kiírása
"Itt a vége fuss el véle!" /

XX // Vége a programnak
```

Eredménye:

```
abcdefghijklmnopqrstuvwxyz
Itt a vége fuss el véle!
```

A fenti példában a „26” helyett természetesen tetszőleges aritmetikai kifejezés állhat, mert mindegy neki: csak egyszer értékeli ki, amikor legelőször ugrik be e ciklusba. Azután soha nem foglalkozik már vele. Azaz ha a „26” helyén egy változó

van, annak értékét is csak egyszer olvassa ki, utána már ha az a változó megváltozik, az se számít neki, nem befolyásolja hogy hányszor hajtódik végre a ciklus. Íme:

```
#c@c 26 // c=26, ennyiszor hajtódik majd végre a ciklus
{| #c@c // c-darabszám-szor fut e majd ez a ciklus
? (@c)+'@ //
-#c @c 1 // c-=1
}| // Ciklus vége
// Üres sor kiírása
"Itt a vége fuss el véle!" /
XX // Vége a programnak
```

Eredménye:

```
ZYXWVUTSRQPONMLKJIHGFEDCBA
Itt a vége fuss el véle!
```

Mint látható a példaprogramban, a „c” változó értékét egyre csökkentjük, mégis 26-szor hajtódik végre a ciklus.

Mégjobbá tehetjük e ciklusunkat! Megcsinálhatjuk úgy, hogy csak LEGFELJEBB a megadott fix számszor fusson le. Azaz betehetünk ide is egy feltételtesztelést a ciklusvégre:

```
int vonalkapcsoscsukozarojel(F& f) { // Visszatérés fix darabszámú ciklusból
unsigned long long darabszam;USIL regip;
darabszam=f.CIKLUSVEREM; // Kiolvassuk a darabszámot a veremből
regip=f.CIKLUSVEREM; // Kiolvassuk a visszatérési címet a veremből
if(f.p[f.P]==') { // Ha a közvetlenül következő karakter nyitózárrójel,
if(ERTEKunsignedchar(f)) return 0; // akkor kiolvassuk a feltételt, s ha az igaz, kilépünk a ciklusból
}
darabszam--;
if(darabszam==0) return 0; // Ha nulla, kilépünk a ciklusból
f.CIKLUSVEREM=regip; // Visszatesszük a visszatérési címet a verembe
f.CIKLUSVEREM=darabszam; // Visszatesszük a darabszámot a verembe
f.P=regip; // Programmutató beállítása a visszatérési címre
return 0; // Visszaugrás a ciklusba
}
```

Ez mindazt pontosan tudja amit az előző változat. Ám ha a |} jeleket közvetlenül követi egy nyitó rendes kerek zárójel, akkor kiértékeli annak a feltételét, s ha az igaz, kiugrik a ciklusból, akkor is, ha az még nem futott le a specifikált darabszám-szor.

Íme:

```
#c@c 26 // c=26, ennyiszor hajtódik majd végre a ciklus
{| #c@c // c-darabszám-szor fut le majd ez a ciklus
? (@c)+'@ //
-#c @c 1 // c-=1
}|(((@c)+'@) == S) // kilépünk 'S'-nél
// Ciklus vége
// Üres sor kiírása
"Itt a vége fuss el véle!" /
XX // Vége a programnak
```

Eredménye:

```
ZYXWVUT
Itt a vége fuss el véle!
```

Gyakran hasznos lehet, ha le tudjuk kérdezni a fix darabszámú lefutó ciklus esetén, épp mennyi a ciklusváltozó értéke - még ha nincs is ez egy külön változóval jelölve a ciklusfejrészben! Íme ennek megoldása:

```
unsigned long long ciklusvaltozo_erteke(F& f) { // visszaadja az aktuális (azaz legbelső)
// fix darabszámú lefutó ciklusváltozó értékét
unsigned long long darabszam;
darabszam=f.CIKLUSVEREM; // Kiolvassuk a darabszámot a veremből
f.CIKLUSVEREM=darabszam; // Visszatesszük a darabszámot a verembe
return darabszam;
}
```

Használata:

```
{| 6
?l ?l; /;
|}
```

Eredménye:

```
6
5
4
3
2
1
```

Mint látható, a ciklusváltozó értéke rendre CSÖKKEN, erre ÜGYELJÜNK! Első értéke konkrétan az amit megadtunk neki, s a nulla értéket sosem éri el!

Valamint jó tudni, hogy bár a {| után a ciklus fejlécében egy unsigned long long értéket tárol el a kis aranyos, de a ?l rendszerváltozó ezt nekünk simán unsigned int értéként adja vissza... Ez többnyire nem baj, általában ezen értéktartomány is megfelelő nekünk. Ha azonban a típushelyesség fontos, akkor sincs semmi gond, lekérdezhetjük unsigned long long értéként is, azaz a maga eredeti típusában, a következő módon:

```
{| 6
?g ?#g "?l"; /;
|}
```

Eredménye természetesen ugyanaz, mint az előzőnek.

16. fejezet: Vesszőcske, avagy a paraméter-szeparátor

Most már tényleg jó sok minden van a programnyelvünkben, s ezen „sokmindenek” közt bizony bőségesen akadnak olyan utasítások, melyek több paramétert is várnak, olyan paramétereket ráadásul, amelyek mindegyike tetszőleges aritmetikai kifejezés lehet! Na és hát eléggé sokféle bizbasz lehet nálunk ám egy aritmetikai kifejezésben, s el kell ismerni, ezek jelölése nálunk néha meglehetősen „egzotikus”! Például egy változó meghatározása ugyebár, ahol a változó indexe (=neve) is, meg a mezőindexe is lehet maga is tetszőleges aritmetikai kifejezés! Ez egyrészt remek dolog, mert lehetővé teszi az akármilyen mélységű többszörös indirekciót, másrészt bizony áttekinthetetlenné teheti a programot! Kéne valami, amivel egyértelműen elkülöníthetjük egymástól az egyes paramétereket - ugyanakkor azonban a megadása nem kötelező!

Erre a célra a „vessző” karaktert szemeltem ki, ami más programnyelvekben is hasonló feladatkört lát el.

Rutinja igen egyszerű:

```
USC vesszoteszt(F& f) { // átugorja a soronkövetkező vesszőt
USC karakter;nemspace(f);karakter=f.p[f.P];if(karakter=='') {k(f);nemspace(f);karakter=f.p[f.P];}
return karakter;
}
```

Ezek után az aritmetikai kifejezést kiértékelő rutinok eleje így néz ki, kékkel kiemelve a módosult részt:

```
unsigned char ERTEKunsignedchar(F& f) { // A következő „Aritmetikai kifejezés” értékét adja vissza unsigned char
számként.
USC karakter;USC változotípus;USC unionindex;unsigned char A;
unionindex=255;változotípus='c'; // Alapértelmezett index és típus
karakter=vesszoteszt(f);
if(karakter=='') {k(f);A= ERTEKunsignedchar(f);nemspace(f);karakter=f.p[f.P];
.....}
```

Fontosnak tartom megjegyezni, hogy e módosítás által a programom binárisának mérete nemcsak hogy nem nőtt sokkal, de egyenesen **CSÖKKENT**, tudniillik teljes 6 bájtal a korábbi állapothoz képest! Ami ugyan nem sok, elismerem, de mégis szép, hogy a kód nemcsak nem nő hanem csökken, s közben a funkció-gazdagság növekszik!

Ezek után a vessző szerepe nálunk hasonló a pontosvesszőéhez, csak míg az utóbbi UTASÍTÁSOKAT választ el egymástól mint üres utasítás, addig a vessző PARAMÉTEREKET választ el egymástól! Megadása opcionális, nem kötelező. Az alábbi utasítások például mind egyenértékűek:

```
-#c @c ,1 // c-=1
-#c @c , 1 // c-=1
-#c @c, 1 // c-=1
-#c @c 1 // c-=1
```

Megadása annyira opcionális, hogy ilyesmi is megengedhető:

```
}((((@c)+,'@) == B) // kilépünk 'B'-nél
```

Azaz még operátor (műveleti jel, jelen esetben a „+”) után is megadható! Így viszont már nem adható meg, hibát fog jelezni:

```
}((((@c)+,'@) == B) // kilépünk 'S'-nél
```

Azaz operátor UTÁN megadható, de operátor ELŐTT már NEM!

Ilyesmi se megengedhető:

```
-#c @c 1, // c-=1
```

Azaz csak olyan helyen állhat, ahol nem újabb utasítást vár, hanem egy másik aritmetikai kifejezést.

Na ez rövid fejezet volt...

17. fejezet: Unáris operátorok

Operátorok nemcsak kétoperandusúak lehetnek, hanem egyoperandusúak is, ezek az úgynevezett „unáris” operátorok. Ilyen például az, ami egy bájt bitenként negáltját képezi, erre a célra a C nyelvben a „~” jel szolgál.

Ha belegondolunk e jel működésébe, szintaktikájába, rájövünk, hogy nekünk máris van egy unáris operátorunk: ez a „#” jel, az utána következő karakterrel együtt, mely megmondja, a következő aritmetikai kifejezést miféle típus szerint kell kiértékelni, azaz ez felel meg nálunk a „casting” operátornak - ami értelem-szerűen csakis egyoperandusú lehet! Bár, tulajdonképpen ez se igaz, el tudnék én képzelni 2 operandusú casting operátort is... de itt a mienk egyoperandusú, épp csak nem egyetlen casting operátorunk van hanem 11.

Ennek mintájára aztán gyárthatunk még akárhány más unáris operátort is. Csináljuk is meg a bitenkénti negációt, persze csak a nem lebegőpontos típusokra! És csináljuk meg a negatív előjel operátort is, ezt meg persze csak a nem „unsigned” típusokra! Csak az egyik függvényen mutatom be, mivel bővült, az **ERTEKsignedlonglong** függvényen, kékkel kiemelve a most beszúrt részt:

```
default: return ERROR(f,3);break;
} // switch vége
goto ERTEK_G_OPERATOR;} // A # teszt vége

if(karakter=='~') {k(f);A=~ERTEKsignedlonglong(f);goto ERTEK_G_OPERATOR;} // A ~ teszt vége
if(karakter=='-') {k(f);A=(-1)*ERTEKsignedlonglong(f);goto ERTEK_G_OPERATOR;} // A - teszt vége, mínusz eggyel szorzás

// ***** Unáris operátorok tesztjének a vége

karakter=kettospontteszt(f,unionindex,valtozotipus); // Az aktuális feldolgozandó karakter
if(karakter=='@') {switch(valtozotipus) {
```

Egy picit kis példaprogram erre:

```
// Bitenkénti negáció
=#c@a %00101001 // a=41
?c ~@a // Most a %11010110 számot kell kiírnia, azaz a 214-es számot
/

// Előjelváltás
=#C@b %00101001 // b=41
?C -@b // Most a -41 számot kell kiírnia
/

XX
```

A futási eredmény természetesen:

```
214
-41
```

Azaz, ezt a részt is könnyen kivégeztük.

18. fejezet: „Normális” szintaxisú értékadás megvalósítása

Nem csodálnám, ha az eddigi értékadó utasításunkat, az `=#` tokenűt, túl nehézkesnek éreznéd. Nos, ezen segíthetünk valamelyest! Ugye, az az utasítás egy típusjelölő karaktert vár rögvest önmaga után. Ez oké is, erre szüksége van. De mit keres a legelején az „=” jel?! Miért is volna arra szüksége okvetlenül?! De jól van, legyen neki ilyen jele, mutassa valami, hogy értékadásról van szó — hanem akkor már tehetjük azt sokkal jobb helyre is! Íme:

```
int fuggveny_kereszt(F& f) {USC index;USC unionindex;USC valtozotipus;F *ff;
ff=PGMfriend_for_f(f);
valtozotipus=vtype(f); //rögtön a következő karakter
namespace(f);kettospontteszt(f,unionindex,valtozotipus); // változóindex beolvasása, ha meg van adva
namespace(f);
if(f.p[f.P]!='@') return ERROR(f,3); // Ha az aktuális feldolgozandó karakter nem kukac, syntax error
k(f); // Ugrás a következő karakterre
index=ERTEKunsignedchar(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat
namespace(f);kettospontteszt(f,unionindex,valtozotipus); // változóindex beolvasása, ha meg van adva

namespace(f);if(f.p[f.P]=='=') {k(f);} else {ERROR(f,3);}
switch(valtozotipus) {
case 'c': (*ff).v[index].c[unionindex]=ERTEKunsignedchar(f);break;
case 'C': (*ff).v[index].C[unionindex]=ERTEKsignedchar(f);break;
case 'i': (*ff).v[index].i[unionindex]=ERTEKunsignedshortint(f);break;
case 'I': (*ff).v[index].I[unionindex]=ERTEKsignedshortint(f);break;
case 'l': (*ff).v[index].l[unionindex]=ERTEKunsignedint(f);break;
case 'L': (*ff).v[index].L[unionindex]=ERTEKsignedint(f);break;
case 'f': (*ff).v[index].f[unionindex]=ERTEKfloat(f);break;
case 'g': (*ff).v[index].g[unionindex]=ERTEKunsignedlonglong(f);break;
case 'G': (*ff).v[index].G[unionindex]=ERTEKsignedlonglong(f);break;
case 'd': (*ff).v[index].d[unionindex]=ERTEKdouble(f);break;
case 'D': (*ff).v[index].D=ERTEKlongdouble(f);break;
} // switch vége
return 0;
}
```

Ezek után ez az értékadás:

```
#i@W 400 // unsigned int W=400
```

bár továbbra is érvényes, de így is írható:

```
#i@W = 400 // unsigned int W=400
```

Hát ez a fenti kétségtelenül messze sokkal áttekinthetőbb, ismerősebb, BARÁTSÁGOSABB! És az hogy ezt így megcsinálhattuk, annak köszönhető, hogy az összehasonlító művelet operátorának nem a sima akarommondani „szimpla” egyenlőségjelet választottuk, hanem a duplát, az „==” jelet, miként az a C nyelvben is van. Emiatt tudja a kis aranyos, hogy a beolvasott aritmetikai kifejezés csak az egyenlőségjelig tart.

Igenám, de észre kell vegyük, hogy ez a rutin hajszálra ugyanaz, mint az `=#` rutinunk, mindössze egyetlen nyamvadt sor van beleszúrva pótlólag a switch utasítás fölé! Az ellenőrzi le az egyenlőségjel meglétét. Elég durva dolog márpedig ilyesmi miatt duplán szerepeltetni egy ekkora rutint... Nem is tűrünk meg ilyesmit! Íme az új, közös rutin, s az ezeket használó értékadó utasításaink új formájú kódba öltözöttek:

```

int ertekadas (F& f, USC módszer) {USC index;USC unionindex;USC változotípus;F *ff;
ff=PGMfriend_for_f(f);
változotípus=vtype(f); //rögtön a következő karakter
namespace(f);kettospontteszt(f,unionindex,vározotípus); // változóindex beolvasása, ha meg van adva
namespace(f);
if(f.p[f.P]!='@') return ERROR(f,3); // Ha az aktuális feldolgozandó karakter nem kukac, syntax error
k(f); // Ugrás a következő karakterre
index=ERTEKunsignedchar(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat
namespace(f);kettospontteszt(f,unionindex,vározotípus); // változóindex beolvasása, ha meg van adva
if(módszer=='=') {namespace(f);if(f.p[f.P]=='=') {k(f);}else {ERROR(f,3);}}
switch(vározotípus) {
case 'c': (*ff).v[index].c[unionindex]=ERTEKunsignedchar(f);break;
case 'C': (*ff).v[index].C[unionindex]=ERTEKsignedchar(f);break;
case 'i': (*ff).v[index].i[unionindex]=ERTEKunsignedshortint(f);break;
case 'I': (*ff).v[index].I[unionindex]=ERTEKsignedshortint(f);break;
case 'l': (*ff).v[index].l[unionindex]=ERTEKunsignedint(f);break;
case 'L': (*ff).v[index].L[unionindex]=ERTEKsignedint(f);break;
case 'f': (*ff).v[index].f[unionindex]=ERTEKFloat(f);break;
case 'g': (*ff).v[index].g[unionindex]=ERTEKunsignedlonglong(f);break;
case 'G': (*ff).v[index].G[unionindex]=ERTEKsignedlonglong(f);break;
case 'd': (*ff).v[index].d[unionindex]=ERTEKdouble(f);break;
case 'D': (*ff).v[index].D=ERTEKlongdouble(f);break;
} // switch vége
return 0;
}
// -----
int egyenlokereszt(F& f) {return ertekadas(f,0);}

// -----
int fuggveny_kereszt(F& f) {return ertekadas(f,'=');}

```

Hát, alaposan lerövidült ez a két rutin, nemigaz?! S maga a kódméret is: Ezen új variációnak hála az interpreter binárisának mérete 1047 bájtal csökkent, ami akárhogy nézem is, de több mint 1 kilobájt! Márpedig ez a mennyiség az eredeti méret 0.7%-a! Megintcsak magamat ismételhetem: Sokan az ilyesmit elintézik egy legyintéssel, hogy mi a francokat görcsölök effélével, ez SEMMI, mert olyan picike érték. Igen, az, de a közmondás is úgy tartja, hogy sok kicsi sokra megy! Egy efféle projekt mint egy programnyelv megírása, elég nagy dolog ami sokáig tart, ezért közben TEMÉRDEK alkalom van rá, hogy odafigyeljünk ilyesmire. Cseppet se tartom túlzásnak, hogy némi kis odafigyeléssel az elkészült progi akár több mint egyharmadával is kevesebb memóriát igényelhet.

A legnagyobb előnye azonban nem ez a fentieknek. Hanem az, hogy ezentúl egyetlen rutinban van lekódolva MINDEN értékadó művelet, TEHÁT ha netán valamikor majd változtatni akarnánk rajta valamit, akkor azt csak egyetlen helyen kell megtennünk!

Menjünk tovább a szintaxis emberivé tételével. Ahogy fejlesztem e nyelvet, egyre inkább úgy találom, hogy soha nem használom a mezőindexnek a @ jel bal oldalán történő megadását. Akkor pedig minek az oda?! Csak elbonyolítja a kódot! Ezt egyetlen sor végzi az aritmetikai kifejezés kiértékelő függvényeinkben, azt kiirtjuk onnan, s ezek után az csak akkor lesz meghívva, ha a @ jelet már beolvastuk, s az azutáni változóindexet is. Emellett gondolkodjunk el azon, tényleg szükségünk van-e nekünk a **vindexc**, **vindexl** stb tömbökre! Nos, úgy találtam, nem. Nemigen fordul elő olyan eset, hogy ezek használata különösebben hasznos lenne, ellenben módfelett elbonyolítják ezek is a kódot! Amennyiben úgy döntünk, hogy a mezőindexnek a ":" „operátorral” való explicit megadása nélkül a mezőindex automatikusan „0” értéknek van tekintve, akkor e tömbök simán nélkülözhetőek.

Még tovább trükközünk. Ha az értékadás operátorát megcsináltuk emberi szintaxisúra, simán megcsinálhatjuk ezt a +#, *# stb függvényekkel is! Lássuk csak,

hogyan írtuk át ezeket, úgy, hogy megmarad a régi szintaxisuk (ami továbbra is érvényes marad). Ellenben ezen átírásnak hála lesz egy alternatív, sokkal barátságosabb formánk is:

```
void mplusz(USC változotípus, USC index, USC unionindex, F& f) {
switch(változotípus) {
case 'c': f.v[index].c[unionindex]+=ERTEKunsignedchar(f);break;
case 'C': f.v[index].C[unionindex]+=ERTEKsignedchar(f);break;
case 'i': f.v[index].i[unionindex]+=ERTEKunsignedshortint(f);break;
case 'I': f.v[index].I[unionindex]+=ERTEKsignedshortint(f);break;
case 'l': f.v[index].l[unionindex]+=ERTEKunsignedint(f);break;
case 'L': f.v[index].L[unionindex]+=ERTEKsignedint(f);break;
case 'f': f.v[index].f[unionindex]+=ERTEKfloat(f);break;
case 'g': f.v[index].g[unionindex]+=ERTEKunsignedlonglong(f);break;
case 'G': f.v[index].G[unionindex]+=ERTEKsignedlonglong(f);break;
case 'd': f.v[index].d[unionindex]+=ERTEKdouble(f);break;
case 'D': f.v[index].D+=ERTEKlongdouble(f);break;
} // switch vége
}
// -----
void mminusz(USC változotípus, USC index, USC unionindex, F& f) {
switch(változotípus) {
case 'c': f.v[index].c[unionindex]-=ERTEKunsignedchar(f);break;
case 'C': f.v[index].C[unionindex]-=ERTEKsignedchar(f);break;
case 'i': f.v[index].i[unionindex]-=ERTEKunsignedshortint(f);break;
case 'I': f.v[index].I[unionindex]-=ERTEKsignedshortint(f);break;
case 'l': f.v[index].l[unionindex]-=ERTEKunsignedint(f);break;
case 'L': f.v[index].L[unionindex]-=ERTEKsignedint(f);break;
case 'f': f.v[index].f[unionindex]-=ERTEKfloat(f);break;
case 'g': f.v[index].g[unionindex]-=ERTEKunsignedlonglong(f);break;
case 'G': f.v[index].G[unionindex]-=ERTEKsignedlonglong(f);break;
case 'd': f.v[index].d[unionindex]-=ERTEKdouble(f);break;
case 'D': f.v[index].D-=ERTEKlongdouble(f);break;
} // switch vége
}
// -----
void mszor(USC változotípus, USC index, USC unionindex, F& f) {
switch(változotípus) {
case 'c': f.v[index].c[unionindex]*=ERTEKunsignedchar(f);break;
case 'C': f.v[index].C[unionindex]*=ERTEKsignedchar(f);break;
case 'i': f.v[index].i[unionindex]*=ERTEKunsignedshortint(f);break;
case 'I': f.v[index].I[unionindex]*=ERTEKsignedshortint(f);break;
case 'l': f.v[index].l[unionindex]*=ERTEKunsignedint(f);break;
case 'L': f.v[index].L[unionindex]*=ERTEKsignedint(f);break;
case 'f': f.v[index].f[unionindex]*=ERTEKfloat(f);break;
case 'g': f.v[index].g[unionindex]*=ERTEKunsignedlonglong(f);break;
case 'G': f.v[index].G[unionindex]*=ERTEKsignedlonglong(f);break;
case 'd': f.v[index].d[unionindex]*=ERTEKdouble(f);break;
case 'D': f.v[index].D*=ERTEKlongdouble(f);break;
} // switch vége
}
// -----
void mper(USC változotípus, USC index, USC unionindex, F& f) {
switch(változotípus) {
case 'c': f.v[index].c[unionindex]/=ERTEKunsignedchar(f);break;
case 'C': f.v[index].C[unionindex]/=ERTEKsignedchar(f);break;
case 'i': f.v[index].i[unionindex]/=ERTEKunsignedshortint(f);break;
case 'I': f.v[index].I[unionindex]/=ERTEKsignedshortint(f);break;
case 'l': f.v[index].l[unionindex]/=ERTEKunsignedint(f);break;
case 'L': f.v[index].L[unionindex]/=ERTEKsignedint(f);break;
case 'f': f.v[index].f[unionindex]/=ERTEKfloat(f);break;
case 'g': f.v[index].g[unionindex]/=ERTEKunsignedlonglong(f);break;
case 'G': f.v[index].G[unionindex]/=ERTEKsignedlonglong(f);break;
case 'd': f.v[index].d[unionindex]/=ERTEKdouble(f);break;
case 'D': f.v[index].D/=ERTEKlongdouble(f);break;
} // switch vége
}
// -----
int muveletvegrehajt(F&f, USC muvelet) {USC i; char kod;
USC index; USC unionindex; USC változotípus;
változotípus=vtype(f); // rögtön a következő karakter
nemspace(f);
if(f.p[f.P]!='@') return SERROR(f,3); // Ha az aktuális feldolgozandó karakter nem kukac, syntax error
k(f); // Ugrás a következő karakterre
index=ERTEKunsignedchar(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat
nemspace(f); kettospontteszt(f, unionindex, változotípus); // változóindex beolvasása, ha meg van adva
kod=(char) muvelet;
for(i=0; i<muveletikoddarab; i++) {
if(kod==muveletikodok[i]) {muveletek[i](változotípus, index, unionindex, f); return 0;}
}
return SERROR(f,3);
}
// -----
int pluszkereszt(F& f) {return muveletvegrehajt(f, '+');}
// -----
```

```

int minuszkereszt(F& f) {return muveletvegrehajt(f, '-');}
// -----
int perkereszt(F& f) {return muveletvegrehajt(f, '/');}
// -----
int csillagkereszt(F& f) {return muveletvegrehajt(f, '*');}
// -----

```

valamint a fejlécállományunkba bele kell venni ezt is:

```

typedef void muvelet(USC valtozotipus, USC index, USC unionindex, F& f);

muvelet mplusz;
muvelet mminusz;
muvelet mszor;
muvelet mper;

muvelet *muveletek[] = {mplusz, mminusz, mszor, mper}; // műveletek függvénytömbje

const char muveletikodok[] = "+-*/";

unsigned char muveletikoddarab = (sizeof(muveletikodok)/sizeof(char))-1;

```

Ez még így önmagában nem sokat segít nekünk, sőt a kód is hosszabb lett ha jól emlékszem vagy 8-10 bájtal. Ellenben most jön a trükk: a korábban említett „ertekadas” függvényünket írjuk át egy picikét - tényleg nem nagyon, csak a közepébe kerül egy kicsinyke rész:

```

int ertekadas (F& f, USC modszer) {USC index; USC unionindex; USC valtozotipus; F *ff; char kod;
ff = PGMfriend_for_f(f);
valtozotipus = vtype(f); // rögtön a következő karakter
nemspace(f);
if(f.p[f.P] != '@') return ERROR(f, 3); // Ha az aktuális feldolgozandó karakter nem kukac, syntax error
k(f); // Ugrás a következő karakterre
index = ERTEKunsignedchar(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat
nemspace(f); kettospontteszt(f, unionindex, valtozotipus); // változóindex beolvasása, ha meg van adva
if(modszer == '=') {nemspace(f); if(f.p[f.P] == '=') {k(f);
} else {register USC i; kod = (char)f.p[f.P]; // Az aktuális karakter
for(i = 0; i < muveletikoddarab; i++) {
if(kod == muveletikodok[i]) {if(k(f) == '=') {k(f); muveletek[i](valtozotipus, index, unionindex, f); return 0;}
else {ERROR(f, 3);}
}
}
ERROR(f, 3);}}
switch(valtozotipus) {
case 'c': (*ff).v[index].c[unionindex] = ERTEKunsignedchar(f); break;
case 'C': (*ff).v[index].C[unionindex] = ERTEKsignedchar(f); break;
case 'i': (*ff).v[index].i[unionindex] = ERTEKunsignedshortint(f); break;
case 'I': (*ff).v[index].I[unionindex] = ERTEKsignedshortint(f); break;
case 'l': (*ff).v[index].l[unionindex] = ERTEKunsignedint(f); break;
case 'L': (*ff).v[index].L[unionindex] = ERTEKsignedint(f); break;
case 'f': (*ff).v[index].f[unionindex] = ERTEKfloat(f); break;
case 'g': (*ff).v[index].g[unionindex] = ERTEKunsignedlonglong(f); break;
case 'G': (*ff).v[index].G[unionindex] = ERTEKsignedlonglong(f); break;
case 'd': (*ff).v[index].d[unionindex] = ERTEKdouble(f); break;
case 'D': (*ff).v[index].D = ERTEKlongdouble(f); break;
} // switch vége
return 0;
}

```

Ennek hála már ilyesmit is megengedhetünk magunknak:

```
#c@a:2 += 1 // a+=1
```

Vagyis lettek szabályos, a "C" nyelvben megszokott olyan operátoraink, mint a **+=**, **-=**, ***=**, **/=** — hát nem nagyszerű?!

Igaz, ehhez kissé ki kell bővítenünk az operátorok érvényességét ellenőrző függvényünk elejét is, mert az szegény eddig úgy tudta, a **+**, **-**, *****, **/** karakterek érvényesek, és csak 1 karakterből álló operátorok! Le kell tesztelje, ha ilyennel találkozunk, hogy a második karakter nem az egyenlőségjel-e. Ha igen, tudja hogy nem operátorról van szó. E függvény eleje most így néz ki:

```

unsigned char Operator2kar(F& f) { // 2 karakterből álló operátor kódjának meghatározása
    USC a; USC kod; USC b; // b a második operátorkarakter
    a=f.p[f.P]; // Az első operátorkarakter
    kod=a; switch(a) { case '+': case '-': case '*':
    if(f.p[f.P+1]=='') { kod=0; goto Operator2karVege; break; } // +=, -=, /=, *= nem operátor
    break; // Egykarakteres utasítástokenek, kódjuk az eredeti marad
    default : b=k(f);
    if(a=='/') {
    if(b=='=') { f.P--; kod=0; goto Operator2karVege; } // A /= utasítás nem operátor
    if(b=='/') { f.P--; kod=0; goto Operator2karVege; } // A // utasítás nem operátor
    if(b==';') { f.P--; kod=0; goto Operator2karVege; } // A /; utasítás nem operátor
    if(b=='+') { f.P--; kod=0; goto Operator2karVege; } // A /+ utasítás nem operátor
    .....

```

A függvény többi része nem változott.

19. fejezet: Tömbök

Ideje, hogy végre tömböket is tudjunk kezelni. Ez elég ocsmány dolog lesz, nem mintha egy tömb önmagában véve valami rémségesen nehéz ügy volna, de sajnos ezt is el kell készítenünk az összes kezelni kívánt típusunkra, márpedig abból 11 darab van...

A lényeg, hogy olyasmit akarunk, hogy a közönséges változók mellett tudjunk kezelni tömböket is. Legyen mondjuk minden adattípusra is 256 lehetséges különböző tömbünk, melyek nevére ugyanazok a szabályok érvényesek, mint a „rendes” változók nevére, de azt hogy tömbről van szó, a szokásos szögletes zárójelekkel jelezzük, azaz például az „x” nevű unsigned char típusú változó 3-adik elemére így hivatkozhatunk:

```
#c@x[3]
```

ahol persze az „x” és a „3” helyén is állhat tetszőleges aritmetikai kifejezés.

E változónak persze semmi köze a közönséges

```
#c@x
```

változóhoz, aminek az esetében nincsenek szögletes zárójelek.

Na most a tömbök kezdetben természetesen nem rendelkeznek memóriaterülettel, mert honnan is tudhatná szegény interpreter ELŐRE, hogy hány darab szám kerül beléjük! Lényeg az hogy az index típusa unsigned int kell legyen, persze bármi más típust is megadhatunk a szögletes zárójelek közt, de ő majd azt arra castolja. És az első használat előtt le kell foglalni neki a memóriaterületet. Meg arra is kell utasítás, hogy ha már nem kell nekünk az a memóriaterület, felszabadíthassuk. Aztán azt persze senki se tiltja meg nekünk, hogy felszabadítás után akármikor lefoglaljuk újra, más értékkel... Ezekre is kell utasítás, ami nézzen ki mondjuk eképp:

Memória foglálás:

```
[#x @ k = 100]
```

A fenti utasítás a „k” nevű „x” típusú változónak foglalt le memóriaterületet, akkorát, ami elég 100 darab változónak az „x” típusból. Azaz nekünk nem kell bájt méreteket számolgatni, ezt elvégzi az interpreter nekünk, mert tudja a típus alapján, hogy egy darab olyan változónak hány bájt kell, ezt beszorozza a darabszámmal ami itt e példában 100, s lefoglalja amennyi kell. Az indexek nullától

számozódnak, ahogy az illik. Az „x” karakter a következők valamelyike lehet, ahogy a nem tömb változóknál is megszoktuk:

c,C,i,I,l,L,g,G,f,d,D

Továbbá, kifejezetten igényeljük és elvárjuk, hogy ha lefoglal valahány darab elem számára memóriát az interpreterünk, akkor azt mind töltsen fel nulla értékkel, azaz garantálja, hogy kezdetben minden változónk nulla értékű a tömbben, s nem valami véletlenszerűen odatrottyantott szemét!

A tömb törlése és a lefoglalt memóriaterület felszabadítása:

[#x @ k]

Ígyeljük természetesen, hogy a tömbjeinket ne csak aritmetikai kifejezésekben szerepeltethessük, de értékadó utasítások bal oldalán is, eféleképp:

#c@t[30] = akármí

Illetve:

#c@t[30] += akármí

Azaz, röviden, hogy abszolút mindent megtehecssünk velük, mint a közönséges változókkal!

E komoly igényeknek komolyan kell nekiállnunk, azaz mindenekelőtt kreáljunk egy TOMB osztályt:

```
class TOMB {
    POINTER t[256];

public:
    TOMB(); // Alapértelmezett konstruktor
    ~TOMB(); // destruktör
    void lefoglal(USC típus, USC index, USI mennyit); // Lefoglalni a tömbnek a memóriaterületet
    void felszabadít(USC típus, USC index); // felszabadítja a tömbnek memóriaterületét
    void tmbdelele(void);
    void BERAK_c(USC index, USI hova, USC k);
    void BERAK_C(USC index, USI hova, signed char k);
    void BERAK_i(USC index, USI hova, unsigned short int k);
    void BERAK_I(USC index, USI hova, signed short int k);
    void BERAK_l(USC index, USI hova, unsigned int k);
    void BERAK_L(USC index, USI hova, signed int k);
    void BERAK_g(USC index, USI hova, unsigned long long k);
    void BERAK_G(USC index, USI hova, signed long long k);
    void BERAK_f(USC index, USI hova, float k);
    void BERAK_d(USC index, USI hova, double k);
    void BERAK_D(USC index, USI hova, long double k);

    unsigned char KIOLVAS_c(USC index, USI honnan);
    signed char KIOLVAS_C(USC index, USI honnan);
    unsigned short int KIOLVAS_i(USC index, USI honnan);
    signed short int KIOLVAS_I(USC index, USI honnan);
    unsigned int KIOLVAS_l(USC index, USI honnan);
    signed int KIOLVAS_L(USC index, USI honnan);
    unsigned long long KIOLVAS_g(USC index, USI honnan);
    signed long long KIOLVAS_G(USC index, USI honnan);
    float KIOLVAS_f(USC index, USI honnan);
    double KIOLVAS_d(USC index, USI honnan);
    long double KIOLVAS_D(USC index, USI honnan);

};
```

és vegyünk fel az F struktúrába egy tömböt:

TOMB t;

Ezen függvények megvalósításai:

```
TOMB::TOMB() {int i;for(i=0;i<256;i++) {
// Maximális indexeket lenullázzuk
t[i].cmax=t[i].Cmax=t[i].imax=t[i].Imax=t[i].lmax=t[i].Lmax=t[i].gmax=t[i].Gmax=t[i].fmax=t[i].dmax=t[i].Dmax=0;
```

```

// pointereket nullára állítjuk
t[i].c=NULL;t[i].C=NULL;t[i].i=NULL;t[i].I=NULL;t[i].l=NULL;t[i].L=NULL;t[i].g=NULL;t[i].G=NULL;t[i].f=NULL;
t[i].d=NULL;t[i].D=NULL;
} // for i vége
} // konstruktor
// -----
TOMB::~TOMB() {tombdelete();} // destruktör
// -----
void TOMB::tombdelete(void){int i;
for(i=0;i<256;i++) {
// Maximális indexeket lenullázzuk
t[i].cmax=t[i].Cmax=t[i].imax=t[i].Imax=t[i].lmax=t[i].Lmax=t[i].gmax=t[i].Gmax=t[i].fmax=t[i].dmax=t[i].Dmax=0;
// pointereket nullára állítjuk
if(t[i].c!=NULL) delete [] t[i].c;
if(t[i].C!=NULL) delete [] t[i].C;
if(t[i].i!=NULL) delete [] t[i].i;
if(t[i].I!=NULL) delete [] t[i].I;
if(t[i].l!=NULL) delete [] t[i].l;
if(t[i].L!=NULL) delete [] t[i].L;
if(t[i].g!=NULL) delete [] t[i].g;
if(t[i].G!=NULL) delete [] t[i].G;
if(t[i].f!=NULL) delete [] t[i].f;
if(t[i].d!=NULL) delete [] t[i].d;
if(t[i].D!=NULL) delete [] t[i].D;
} // for i vége
}
// -----
void TOMB::felszabadit(USC tipus, USC index) {
switch(tipus) {
case 'c' : if(t[index].c!=NULL) delete [] t[index].c;t[index].cmax=0;break;
case 'C' : if(t[index].C!=NULL) delete [] t[index].C;t[index].Cmax=0;break;
case 'i' : if(t[index].i!=NULL) delete [] t[index].i;t[index].imax=0;break;
case 'I' : if(t[index].I!=NULL) delete [] t[index].I;t[index].Imax=0;break;
case 'l' : if(t[index].l!=NULL) delete [] t[index].l;t[index].lmax=0;break;
case 'L' : if(t[index].L!=NULL) delete [] t[index].L;t[index].Lmax=0;break;
case 'g' : if(t[index].g!=NULL) delete [] t[index].g;t[index].gmax=0;break;
case 'G' : if(t[index].G!=NULL) delete [] t[index].G;t[index].Gmax=0;break;
case 'f' : if(t[index].f!=NULL) delete [] t[index].f;t[index].fmax=0;break;
case 'd' : if(t[index].d!=NULL) delete [] t[index].d;t[index].dmax=0;break;
case 'D' : if(t[index].D!=NULL) delete [] t[index].D;t[index].Dmax=0;break;
} // switch vége
}
// -----
void tomberror(void) {
L("Olyan tömbnek kíséreltél meg lefoglalni memóriát, aminek korábban már foglaltál le területet!");EXITFAILURE();
}
void tombindexerror(void) {
L("Indexhatárátlépés tömb esetében!");EXITFAILURE();
}
// -----
void TOMB::lefoglal(USC tipus, USC index, USI mennyit) { // lefoglalja a tömbnek a memóriaterületet.
switch(tipus) {
case 'c' : if(t[index].c!=NULL) tomberror();break;
case 'C' : if(t[index].C!=NULL) tomberror();break;
case 'i' : if(t[index].i!=NULL) tomberror();break;
case 'I' : if(t[index].I!=NULL) tomberror();break;
case 'l' : if(t[index].l!=NULL) tomberror();break;
case 'L' : if(t[index].L!=NULL) tomberror();break;
case 'g' : if(t[index].g!=NULL) tomberror();break;
case 'G' : if(t[index].G!=NULL) tomberror();break;
case 'f' : if(t[index].f!=NULL) tomberror();break;
case 'd' : if(t[index].d!=NULL) tomberror();break;
case 'D' : if(t[index].D!=NULL) tomberror();break;
} // switch vége
switch(tipus) {register USI i;
case 'c' : t[index].c = new unsigned char [mennyit];t[index].cmax=mennyit;for(i=0;i<mennyit;i++){t[index].c[i]=0;}
break;
case 'C' : t[index].C = new signed char [mennyit];t[index].Cmax=mennyit;for(i=0;i<mennyit;i++){t[index].C[i]=0;}break;
case 'i' : t[index].i = new unsigned short int [mennyit];t[index].imax=mennyit;for(i=0;i<mennyit;i++)
{t[index].i[i]=0;}break;
case 'I' : t[index].I = new signed short int [mennyit];t[index].Imax=mennyit;for(i=0;i<mennyit;i++)
{t[index].I[i]=0;}break;
case 'l' : t[index].l = new unsigned int [mennyit];t[index].lmax=mennyit;for(i=0;i<mennyit;i++){t[index].l[i]=0;}break;
case 'L' : t[index].L = new signed int [mennyit];t[index].Lmax=mennyit;for(i=0;i<mennyit;i++){t[index].L[i]=0;}break;
case 'g' : t[index].g = new unsigned long long [mennyit];t[index].gmax=mennyit;for(i=0;i<mennyit;i++)
{t[index].g[i]=0;}break;
case 'G' : t[index].G = new signed long long [mennyit];t[index].Gmax=mennyit;for(i=0;i<mennyit;i++)
{t[index].G[i]=0;}break;
case 'f' : t[index].f = new float [mennyit];t[index].fmax=mennyit;for(i=0;i<mennyit;i++){t[index].f[i]=0;}break;
case 'd' : t[index].d = new double [mennyit];t[index].dmax=mennyit;for(i=0;i<mennyit;i++){t[index].d[i]=0;}break;
case 'D' : t[index].D = new long double [mennyit];t[index].Dmax=mennyit;for(i=0;i<mennyit;i++){t[index].D[i]=0;}break;
} // switch vége
}
// -----
void TOMB::BERAK_c(USC index, USI hova, USC k) {if(hova>=t[index].cmax) tombindexerror();
t[index].c[hova]=k;
}
// -----
void TOMB::BERAK_C(USC index, USI hova, signed char k) {if(hova>=t[index].Cmax) tombindexerror();

```

```

t[index].C[hova]=k;
}
// -----
void TOMB::BERAK_i(USC index, USI hova, unsigned short int k) {if(hova>=t[index].imax) tombindexerror();
t[index].i[hova]=k;
}
// -----
void TOMB::BERAK_I(USC index, USI hova, signed short int k) {if(hova>=t[index].Imax) tombindexerror();
t[index].I[hova]=k;
}
// -----
void TOMB::BERAK_l(USC index, USI hova, unsigned int k) {if(hova>=t[index].lmax) tombindexerror();
t[index].l[hova]=k;
}
// -----
void TOMB::BERAK_L(USC index, USI hova, signed int k) {if(hova>=t[index].Lmax) tombindexerror();
t[index].L[hova]=k;
}
// -----
void TOMB::BERAK_g(USC index, USI hova, unsigned long long k) {if(hova>=t[index].gmax) tombindexerror();
t[index].g[hova]=k;
}
// -----
void TOMB::BERAK_G(USC index, USI hova, signed long long k) {if(hova>=t[index].Gmax) tombindexerror();
t[index].G[hova]=k;
}
// -----
void TOMB::BERAK_f(USC index, USI hova, float k) {if(hova>=t[index].fmax) tombindexerror();
t[index].f[hova]=k;
}
// -----
void TOMB::BERAK_d(USC index, USI hova, double k) {if(hova>=t[index].dmax) tombindexerror();
t[index].d[hova]=k;
}
// -----
void TOMB::BERAK_D(USC index, USI hova, long double k) {if(hova>=t[index].Dmax) tombindexerror();
t[index].D[hova]=k;
}
// -----
unsigned char TOMB::KIOLVAS_c(USC index, USI honnan) {if(honnan>=t[index].cmax) tombindexerror();
return t[index].c[honnan];}
// -----
signed char TOMB::KIOLVAS_C(USC index, USI honnan) {if(honnan>=t[index].Cmax) tombindexerror();
return t[index].C[honnan];}
// -----
unsigned short int TOMB::KIOLVAS_i(USC index, USI honnan) {if(honnan>=t[index].imax) tombindexerror();
return t[index].i[honnan];}
// -----
signed short int TOMB::KIOLVAS_I(USC index, USI honnan) {if(honnan>=t[index].Imax) tombindexerror();
return t[index].I[honnan];}
// -----
unsigned int TOMB::KIOLVAS_l(USC index, USI honnan) {if(honnan>=t[index].lmax) tombindexerror();
return t[index].l[honnan];}
// -----
signed int TOMB::KIOLVAS_L(USC index, USI honnan) {if(honnan>=t[index].Lmax) tombindexerror();
return t[index].L[honnan];}
// -----
unsigned long long TOMB::KIOLVAS_g(USC index, USI honnan) {if(honnan>=t[index].gmax) tombindexerror();
return t[index].g[honnan];}
// -----
signed long long TOMB::KIOLVAS_G(USC index, USI honnan) {if(honnan>=t[index].Gmax) tombindexerror();
return t[index].G[honnan];}
// -----
float TOMB::KIOLVAS_f(USC index, USI honnan) {if(honnan>=t[index].fmax) tombindexerror();
return t[index].f[honnan];}
// -----
double TOMB::KIOLVAS_d(USC index, USI honnan) {if(honnan>=t[index].dmax) tombindexerror();
return t[index].d[honnan];}
// -----
long double TOMB::KIOLVAS_D(USC index, USI honnan) {if(honnan>=t[index].Dmax) tombindexerror();
return t[index].D[honnan];}
// -----

```

A **vc**, **vc** stb függvényeinket is ki kell bővítenünk kissé, mint ezt bemutatom itt alább, de csak az egyiken, kékkel kiemelve a beszúrt részeket:

```

USC vc(F& f,USC unionindex=0) {// A programkódban soron következő változó aktuális értékét adja vissza USC típusként.
// A P mutató okvetlenül egy @ jelre kell mutasson!
USC index;USC változotípus;USI tombindex;
változotípus='c';k(f);index=ERTEKunsignedchar(f);
namespace(f);
if(f.p[f.P]=='[') {// Tömbről van szó
k(f);tombindex=ERTEKunsignedint(f);namespace(f);
if(f.p[f.P]!='']') {SERROR(f,3);}
k(f);return f.t.KIOLVAS_c(index,tombindex);
}
}

```

```
kettospontteszt(f,unionindex,valtozotipus); // Az aktuális feldolgozandó karakter
return f.v[index].c[unionindex];
}
```

Meg kell csinálnunk a [# utasítást is, ami így néz ki:

```
int nyitoszegleteskereszt(F& f) {
// [#tipus @ index = darabszám amit le kell foglalni] // Memórialefogalás tömbnek
// [#tipus @ index] // Tömb törlése
USC index;USC tipus;
tipus=f.p[f.P]; // Közvetlenül a következő karakter
k(f);namespace(f);
if(f.p[f.P]!='@') {ERROR(f,3);}
k(f);index=ERTEKunsignedchar(f);namespace(f);
if(f.p[f.P]=='|') {k(f);f.t.felszabadit(tipus,index);return 0;} // Tömb törlése
if(f.p[f.P]!='=') {ERROR(f,3);}
k(f);f.t.lefoglal(tipus,index,ERTEKunsignedint(f));namespace(f);
if(f.p[f.P]=='|') {k(f);} else {ERROR(f,3);}
return 0;
}
```

Végezetül pedig az „ertekadas” nevű függvényünket sajnos eszméletlenül sok mindenfélével ki kell bővítenünk! Íme:

```
int ertekadas (F& f, USC módszer) {USC index;USC unionindex;USC valtozotipus;F *ff; char kod;
USI tombindex;USC muveletikod;
ff=PGMfriend_for_f(f);
valtozotipus=vtype(f); //rögtön a következő karakter
namespace(f);
if(f.p[f.P]!='@') return ERROR(f,3); // Ha az aktuális feldolgozandó karakter nem kukac, syntax error
k(f); // Ugrás a következő karakterre
index=ERTEKunsignedchar(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat
namespace(f);

if(f.p[f.P]=='|') { // tömbváltozóról van szó
k(f); // következő karakterre ugrás
tombindex=ERTEKunsignedint(f); // tombindex beolvasása
namespace(f);
if(f.p[f.P]!='|') ERROR(f,3); // syntax error
k(f); // következő karakterre ugrás
namespace(f);

if((f.p[f.P]=='+')||(f.p[f.P]=='-')||(f.p[f.P]=='*')||(f.p[f.P]=='/')) {muveletikod=f.p[f.P];

if(f.p[f.P+1]=='=') {k(f);k(f);

switch(muveletikod) {

case '+': { switch(valtozotipus){
case 'c': f.t.BERAK_c(index,tombindex,f.t.KIOLVAS_c(index,tombindex)+ERTEKunsignedchar(f));return 0; break;
case 'C': f.t.BERAK_C(index,tombindex,f.t.KIOLVAS_C(index,tombindex)+ERTEKsignedchar(f));return 0; break;
case 'i': f.t.BERAK_i(index,tombindex,f.t.KIOLVAS_i(index,tombindex)+ERTEKunsignedshortint(f));return 0; break;
case 'I': f.t.BERAK_I(index,tombindex,f.t.KIOLVAS_I(index,tombindex)+ERTEKsignedshortint(f));return 0; break;
case 'l': f.t.BERAK_l(index,tombindex,f.t.KIOLVAS_l(index,tombindex)+ERTEKunsignedint(f));return 0; break;
case 'L': f.t.BERAK_L(index,tombindex,f.t.KIOLVAS_L(index,tombindex)+ERTEKsignedint(f));return 0; break;
case 'g': f.t.BERAK_g(index,tombindex,f.t.KIOLVAS_g(index,tombindex)+ERTEKunsignedlonglong(f));return 0; break;
case 'G': f.t.BERAK_G(index,tombindex,f.t.KIOLVAS_G(index,tombindex)+ERTEKsignedlonglong(f));return 0; break;
case 'f': f.t.BERAK_f(index,tombindex,f.t.KIOLVAS_f(index,tombindex)+ERTEKfloat(f));return 0; break;
case 'd': f.t.BERAK_d(index,tombindex,f.t.KIOLVAS_d(index,tombindex)+ERTEKdouble(f));return 0; break;
case 'D': f.t.BERAK_D(index,tombindex,f.t.KIOLVAS_D(index,tombindex)+ERTEKlongdouble(f));return 0; break;
} // switch vége
} // case + vége

case '-': { switch(valtozotipus){
case 'c': f.t.BERAK_c(index,tombindex,f.t.KIOLVAS_c(index,tombindex)-ERTEKunsignedchar(f));return 0; break;
case 'C': f.t.BERAK_C(index,tombindex,f.t.KIOLVAS_C(index,tombindex)-ERTEKsignedchar(f));return 0; break;
case 'i': f.t.BERAK_i(index,tombindex,f.t.KIOLVAS_i(index,tombindex)-ERTEKunsignedshortint(f));return 0; break;
case 'I': f.t.BERAK_I(index,tombindex,f.t.KIOLVAS_I(index,tombindex)-ERTEKsignedshortint(f));return 0; break;
case 'l': f.t.BERAK_l(index,tombindex,f.t.KIOLVAS_l(index,tombindex)-ERTEKunsignedint(f));return 0; break;
case 'L': f.t.BERAK_L(index,tombindex,f.t.KIOLVAS_L(index,tombindex)-ERTEKsignedint(f));return 0; break;
case 'g': f.t.BERAK_g(index,tombindex,f.t.KIOLVAS_g(index,tombindex)-ERTEKunsignedlonglong(f));return 0; break;
case 'G': f.t.BERAK_G(index,tombindex,f.t.KIOLVAS_G(index,tombindex)-ERTEKsignedlonglong(f));return 0; break;
case 'f': f.t.BERAK_f(index,tombindex,f.t.KIOLVAS_f(index,tombindex)-ERTEKfloat(f));return 0; break;
case 'd': f.t.BERAK_d(index,tombindex,f.t.KIOLVAS_d(index,tombindex)-ERTEKdouble(f));return 0; break;
case 'D': f.t.BERAK_D(index,tombindex,f.t.KIOLVAS_D(index,tombindex)-ERTEKlongdouble(f));return 0; break;
} // switch vége
} // case - vége

case '*': { switch(valtozotipus){
case 'c': f.t.BERAK_c(index,tombindex,f.t.KIOLVAS_c(index,tombindex)*ERTEKunsignedchar(f));return 0; break;
case 'C': f.t.BERAK_C(index,tombindex,f.t.KIOLVAS_C(index,tombindex)*ERTEKsignedchar(f));return 0; break;
case 'i': f.t.BERAK_i(index,tombindex,f.t.KIOLVAS_i(index,tombindex)*ERTEKunsignedshortint(f));return 0; break;
case 'I': f.t.BERAK_I(index,tombindex,f.t.KIOLVAS_I(index,tombindex)*ERTEKsignedshortint(f));return 0; break;
}
```

```

case 'l': f.t.BERAK_l(index,tombindex,f.t.KIOLVAS_l(index,tombindex)*ERTEKunsignedint(f));return 0; break;
case 'L': f.t.BERAK_L(index,tombindex,f.t.KIOLVAS_L(index,tombindex)*ERTEKsignedint(f));return 0; break;
case 'g': f.t.BERAK_g(index,tombindex,f.t.KIOLVAS_g(index,tombindex)*ERTEKunsignedlonglong(f));return 0; break;
case 'G': f.t.BERAK_G(index,tombindex,f.t.KIOLVAS_G(index,tombindex)*ERTEKsignedlonglong(f));return 0; break;
case 'f': f.t.BERAK_f(index,tombindex,f.t.KIOLVAS_f(index,tombindex)*ERTEKfloat(f));return 0; break;
case 'd': f.t.BERAK_d(index,tombindex,f.t.KIOLVAS_d(index,tombindex)*ERTEKdouble(f));return 0; break;
case 'D': f.t.BERAK_D(index,tombindex,f.t.KIOLVAS_D(index,tombindex)*ERTEKlongdouble(f));return 0; break;
} // switch vége
} // case * vége

case '/': { switch(valtozotipus){
case 'c': f.t.BERAK_c(index,tombindex,f.t.KIOLVAS_c(index,tombindex)/ERTEKunsignedchar(f));return 0; break;
case 'C': f.t.BERAK_C(index,tombindex,f.t.KIOLVAS_C(index,tombindex)/ERTEKsignedchar(f));return 0; break;
case 'i': f.t.BERAK_i(index,tombindex,f.t.KIOLVAS_i(index,tombindex)/ERTEKunsignedshortint(f));return 0; break;
case 'I': f.t.BERAK_I(index,tombindex,f.t.KIOLVAS_I(index,tombindex)/ERTEKsignedshortint(f));return 0; break;
case 'l': f.t.BERAK_l(index,tombindex,f.t.KIOLVAS_l(index,tombindex)/ERTEKunsignedint(f));return 0; break;
case 'L': f.t.BERAK_L(index,tombindex,f.t.KIOLVAS_L(index,tombindex)/ERTEKsignedint(f));return 0; break;
case 'g': f.t.BERAK_g(index,tombindex,f.t.KIOLVAS_g(index,tombindex)/ERTEKunsignedlonglong(f));return 0; break;
case 'G': f.t.BERAK_G(index,tombindex,f.t.KIOLVAS_G(index,tombindex)/ERTEKsignedlonglong(f));return 0; break;
case 'f': f.t.BERAK_f(index,tombindex,f.t.KIOLVAS_f(index,tombindex)/ERTEKfloat(f));return 0; break;
case 'd': f.t.BERAK_d(index,tombindex,f.t.KIOLVAS_d(index,tombindex)/ERTEKdouble(f));return 0; break;
case 'D': f.t.BERAK_D(index,tombindex,f.t.KIOLVAS_D(index,tombindex)/ERTEKlongdouble(f));return 0; break;
} // switch vége
} // case / vége

} // switch muveletikod vége

} // = teszt vége
} // hosszu if vége ( + - * / teszt )

else {
if(f.p[f.P]=='=') {k(f);

switch(valtozotipus){
case 'c': f.t.BERAK_c(index,tombindex,ERTEKunsignedchar(f));return 0; break;
case 'C': f.t.BERAK_C(index,tombindex,ERTEKsignedchar(f));return 0; break;
case 'i': f.t.BERAK_i(index,tombindex,ERTEKunsignedshortint(f));return 0; break;
case 'I': f.t.BERAK_I(index,tombindex,ERTEKsignedshortint(f));return 0; break;
case 'l': f.t.BERAK_l(index,tombindex,ERTEKunsignedint(f));return 0; break;
case 'L': f.t.BERAK_L(index,tombindex,ERTEKsignedint(f));return 0; break;
case 'g': f.t.BERAK_g(index,tombindex,ERTEKunsignedlonglong(f));return 0; break;
case 'G': f.t.BERAK_G(index,tombindex,ERTEKsignedlonglong(f));return 0; break;
case 'f': f.t.BERAK_f(index,tombindex,ERTEKfloat(f));return 0; break;
case 'd': f.t.BERAK_d(index,tombindex,ERTEKdouble(f));return 0; break;
case 'D': f.t.BERAK_D(index,tombindex,ERTEKlongdouble(f));return 0; break;
} // switch vége

} // = teszt vége
} // else vége
} // tömbváltozó vége

ketospontteszt(f,unionindex,valtozotipus); // változóindex beolvasása, ha meg van adva
if(modszer=='=') {namespace(f);if(f.p[f.P]=='=') {k(f);
}else {register USC i;kod=(char)f.p[f.P];// Az aktuális karakter
for(i=0;i<muveletikoddarab;i++) {
if(kod==muveletikodok[i]) {if(k(f)=='='){k(f);muveletek[i](valtozotipus,index,unionindex,f);return 0;}
else {ERROR(f,3);}
}
}
ERROR(f,3);}}
switch(valtozotipus) {
case 'c': (*ff).v[index].c[unionindex]=ERTEKunsignedchar(f);break;
case 'C': (*ff).v[index].C[unionindex]=ERTEKsignedchar(f);break;
case 'i': (*ff).v[index].i[unionindex]=ERTEKunsignedshortint(f);break;
case 'I': (*ff).v[index].I[unionindex]=ERTEKsignedshortint(f);break;
case 'l': (*ff).v[index].l[unionindex]=ERTEKunsignedint(f);break;
case 'L': (*ff).v[index].L[unionindex]=ERTEKsignedint(f);break;
case 'f': (*ff).v[index].f[unionindex]=ERTEKfloat(f);break;
case 'g': (*ff).v[index].g[unionindex]=ERTEKunsignedlonglong(f);break;
case 'G': (*ff).v[index].G[unionindex]=ERTEKsignedlonglong(f);break;
case 'd': (*ff).v[index].d[unionindex]=ERTEKdouble(f);break;
case 'D': (*ff).v[index].D=ERTEKlongdouble(f);break;

} // switch vége
return 0;

}

```

Na hát ez már TÉNYLEG nem fér rá egy képernyőre a legnagyobb jóindulattal sem; — ennek azonban az az oka, hogy vegyük észre, ez lényegében egy egész csomó feladatot ellát nekünk! Ez több függvény, amik szépen takaros rendben követik egymást: A +=, a -=, a *=, a /=, meg a közönséges = függvény. Egyelőre nem látom értelmét, hogy ezeket külön függvényekbe pakoljam.

Tömbjeink működését demonstrálja e program:

```
[#c@t=100] // Memória foglalás 100 darab unsigned char értéknek a "t" nevű tömbbe

#i@i = 0; // i=0

{| 10 // ciklus 10-szer

#c@t[#i@i]=#i@i ; // t[i] = i

?i @i          // print i
"-edik elem = "
?c @t[#i@i] /;

#i@i += 1 ; // i += 1

|}
"Növelés után" /;
#i@i = 3; // i=3

{| 5; #c@t[#i@i] += 2; #i@i += 1; |} // Itt növelem a 3 - 7 indexű elemeket 2-vel

#i@i = 0; // i=0

{| 10 // ciklus 10-szer
if((#i@i) == 3) T "Innentől nőttek meg az értékek 2-vel" /;
if((#i@i) == 8) T "Eddig volt növelve, innentől nem növeltük meg őket" /;

?i @i          // print i
"-edik elem = "
?c @t[#i@i] /;

#i@i += 1 ; // I += 1

|}

XX
```

A program futásának eredménye:

```
0-edik elem = 0
1-edik elem = 1
2-edik elem = 2
3-edik elem = 3
4-edik elem = 4
5-edik elem = 5
6-edik elem = 6
7-edik elem = 7
8-edik elem = 8
9-edik elem = 9
Növelés után
0-edik elem = 0
1-edik elem = 1
2-edik elem = 2
Innentől nőttek meg az értékek 2-vel
3-edik elem = 5
4-edik elem = 6
5-edik elem = 7
6-edik elem = 8
7-edik elem = 9
Eddig volt növelve, innentől nem növeltük meg őket
8-edik elem = 8
9-edik elem = 9
```

Véleményem szerint azonban ezek után semmi szükségünk az F struktúrában lefoglalt külön memóriaterületre amire az „m” pointer mutat, meg az ezeket kezelő függvényekre! Ha kell nekünk ilyesmi, egyszerűen lefoglalunk egy unsigned char típusú tömbnek kellő elemszámmra memóriát. Ezeket tehát kiirtjuk a nyelvünkéből. Ezáltal a kódméret ráadásul 3676 bájtal csökken, azaz több mint 3.5 kilobájtal...

Ezek után azonban át kell írunk a brainfuck-értelmező részt is a programnyelvünkben, mert már nem létezik az a memóriaterület, amire hivatkozik! Nos, semmi baj, hivatkozzon ehelyett a nullás sorszámú tömb unsigned char típusú

értékeire! Majd ennek kell lefoglalnunk helyet minden brainfuck program elején.
Ez nem az eddigi
M 30000
utasítással történik majd, hanem így:
`[#c@0=30000]`

Ez jobban illik is a „brainfuck” stílushoz szellemiségben, mert titokzatosabb...

Az átírt brainfuck rész így néz ki a programunkban:

```
// ===== BRAINFUCK utasítások =====
int brainfuck_kisebbjel(F& f) { // Brainfuck <
if(f.v[0].l[0]==0L) {L("Adatmemória alulcsordulás!");EXITFAILURE();}
f.v[0].l[0]--;return 0;}

int brainfuck_nagyobbjel(F& f) { // Brainfuck >
f.v[0].l[0]++;
return 0;}

int brainfuck_plusz(F& f) { // Brainfuck +
f.t.BERAK_c(0,f.v[0].l[0],f.t.KIOLVAS_c(0,f.v[0].l[0])+1);return 0;}

int brainfuck_minusz(F& f) { // Brainfuck -
f.t.BERAK_c(0,f.v[0].l[0],f.t.KIOLVAS_c(0,f.v[0].l[0])-1);return 0;}

int brainfuck_pont(F& f) {printf("%c",f.t.KIOLVAS_c(0,f.v[0].l[0]));return 0;} // Brainfuck .

int brainfuck_vesszo(F& f) {f.t.BERAK_c(0,f.v[0].l[0],getchar());return 0;} // Brainfuck ,

int brainfuck_nyitozarojel(F& f) { // Brainfuck [
USIL darab=0L;USC c;
if(f.t.KIOLVAS_c(0,f.v[0].l[0])==0) {
while(1) {
c=f.p[f.P++];
if(f.P>=f.phossz) return -2;
if(c=='[') darab++;
if(c==']') {if(darab==0L) { // f.P++;
if(f.P>=f.phossz) {return -2;}; return 0;}
darab--;
}
} // while vége
} // if ==0 vége
return 0;
} // funkció vége

int brainfuck_csukozarojel(F& f) { // Brainfuck ]
USIL darab=0L;USC c;
f.P--;if(f.P==0L) return -3;
f.P--; // Most az aktuális utasítást megelőző bájton van.
while(1) {
if(f.P==0L) return -3;
c=f.p[f.P--];if(f.P==0L) return -3;
if(c==']') darab++;
if(c=='[') {if(darab==0L) {f.P++;return 0;}
darab--;
}
} // while vége
return 0;
} // funkció vége
```

Ha már úgyis belenyúltunk a változókat kiértékelő rutinokba, hogy lecsekkolja, van-e nekije olyanja a végén hogy '[' tömbindex-jel, miért is ne lehetne úgy, hogy ha a két szögeletes-zárójel, a nyitó és a csukó közvetlenül követi egymást, és nincs közte semmi se megadva (whitespace sem!) akkor ebből tudja, hogy nem tömbről van szó, hanem veremről?! Így megszabadulnánk attól a nyavalyás „v” típusjelölőtől. Ennek tesztjét persze kiirtanánk az aritmetikai kifejezések kiértékeléséből is. Sőt, a tömbökhöz hasonlóan megoldhatnánk ugyanilyen szintaxissal a veremtárba való betétel kérdését is „emberbarát” szintaxissal, a "[" karakter-párost figyelve. Ekkor pedig feleslegessé válik a v# utasítás is! Íme a módosult **ertekadas** rutin egy része, kékkel kiemelve a beszúrt soroka:

```

int ertekadas (F& f, USC modszer) {USC index;USC unionindex;USC valtozotipus;F *ff; char kod;
USI tombindex;USC muveletikod;
ff=PGMfriend_for_f(f);
valtozotipus=vtype(f); //rögtön a következő karakter
namespace(f);
if(f.p[f.P]!='@') return SERROR(f,3); // Ha az aktuális feldolgozandó karakter nem kukac, syntax error
k(f); // Ugrás a következő karakterre
index=ERTEKunsignedchar(f); // Beolvasta a változó indexét. Most a P a következő bármilyen karakterre mutat
namespace(f);

if(f.p[f.P]=='[') { // ..... tömb- vagy veremváltozóról van szó
k(f); // következő karakterre ugrás
if(f.p[f.P]==']') { // ..... Veremváltozóról van szó
k(f); // következő karakterre ugrás
namespace(f);if(f.p[f.P]!='=') {SERROR(f,3);}
k(f); // következő karakterre ugrás
switch(valtozotipus) {
case 'c': (*ff).verem[index]=ERTEKunsignedchar(f);break;
case 'C': (*ff).verem[index]=ERTEKsignedchar(f);break;
case 'i': (*ff).verem[index]=ERTEKunsignedshortint(f);break;
case 'I': (*ff).verem[index]=ERTEKsignedshortint(f);break;
case 'l': (*ff).verem[index]=ERTEKunsignedint(f);break;
case 'L': (*ff).verem[index]=ERTEKsignedint(f);break;
case 'f': (*ff).verem[index]=ERTEKfloat(f);break;
case 'g': (*ff).verem[index]=ERTEKunsignedlonglong(f);break;
case 'G': (*ff).verem[index]=ERTEKsignedlonglong(f);break;
case 'd': (*ff).verem[index]=ERTEKdouble(f);break;
case 'D': (*ff).verem[index]=ERTEKlongdouble(f);break;
} // switch vége
return 0;
} // if [] vége

// Itt már biztos hogy nem veremről hanem tömbről van szó

tombindex=ERTEKunsignedint(f); // tombindex beolvasása
namespace(f);
.....

```

És itt a változók értékét visszaadó függvények közül az egyik, példának, kékkel kiemelve a beszúrt sorokat:

```

USC vc(F& f,USC unionindex=0) {// A progrankódban soron következő változó aktuális értékét adja vissza USC típusként.
// A P mutató okvetlenül egy @ jelre kell mutasson!
USC index;USC valtozotipus;USI tombindex;
valtozotipus='c';k(f);index=ERTEKunsignedchar(f);
namespace(f);
if(f.p[f.P]=='[') {// Tömbről van szó
k(f); // ugrás a következő karakterre
if(f.p[f.P]==']') { // Veremről van szó
k(f); // ugrás a következő karakterre
return (unsigned char)f.verem[index];
}

tombindex=ERTEKunsignedint(f);namespace(f);
.....

```

Ezek után a korábban már bemutatott példaprogram a veremkezelésre a következőképp néz ki:

```

MV x 2000 // 2000 bájtot lefoglalunk az „x” nevű verem számára

#c@x[] = 4 // Beteszünk az „x” nevű verembe 4-et
#c@x[] = 5 // Beteszünk az „x” nevű verembe 5-öt
#c@x[] = 18 // Beteszünk az „x” nevű verembe 18-at
#c@x[] = 20 // Beteszünk az „x” nevű verembe 20-at

#i@x[] = $fce2 // Beteszünk a verembe egy 2 bájtos hexa számot is, ez a 64738

// Kiírás

if(?v x) T "A verem nem üres 1." /
?c @x[] ; / // Itt kiírom a hexa fce2 2 bájta közül a felső bájtot, értéke 252
?c @x[] ; / // Itt kiírom a hexa fce2 2 bájta közül az alsó bájtot, értéke 226
?c @x[] ; / // Kiírom a 20-at
?c @x[] ; / // Kiírom a 18-at
if(?v x) T "A verem nem üres 2." /
?c @x[] ; / // Kiírom az 5-öt
?c @x[] ; / // Kiírom a 4-et

// Most a verem kiürült. Rakunk bele megint számokat:
if(?v x) T "A verem nem üres 3." /
E "A verem üres" /
#c@x[] = 4 // 4-et

```

```
#i@x[] = $fce2      // 64738-at, ez 2 bájtos
#c@x[] = 20         // 20-at

?c @x[] ; / // Kiírom a 20-at

?i @x[] ; / // Kiírom a 64738-at, mert most unsigned short intként vettem ki a veremből
?c @x[] ; / // Kiírom a 4-et
```

Az outputja természetesen ugyanaz kell legyen, nem szabad hogy megváltozzék:

```
A verem nem üres 1.
252
226
20
18
A verem nem üres 2.
5
4
A verem üres
20
64738
4
```

Ha már a szegletes (szögletes) zárójelekkel bohóckodunk memóriát foglalandó, alakítsuk át a vermeknek történő memórialefoglalás és felszabadítás szintaxisát is! Valahogy nekem nem tetszik a régi. Az új legyen ez, amint itt látszik a forrás-kódból:

```
int nyitoszegleteskukac(F& f) { // Memóriafoglalás veremnek, vagy annak felszabadítása
// [@ index [ byteszám_amit_le_kell_foglalni ] ] // Memóriafoglalás
// [@ index [] ] // Felszabadítás
USC index;
namespace(f); // Következő értékes karakterre ugrás
index=ERTEKunsignedchar(f);namespace(f); // Index beolvasása és a köv. értékes karakterre ugrás

if(f.p[f.P]!='{') {ERROR(f,3);} // Syntax error
k(f);
if(f.p[f.P]=='}') { // Felszabadítás
k(f);f.verem[index].veremdelete();namespace(f);if(f.p[f.P]!='}') {ERROR(f,3);} // Syntax error
k(f);return 0;} // Felszabadítás vége
// Lefoglalás

f.verem[index].veremmemorialefoglal(ERTEKunsignedint(f));
namespace(f);
if(f.p[f.P]!='{') {ERROR(f,3);} // Syntax error
k(f);namespace(f);
if(f.p[f.P]!='}') {ERROR(f,3);} // Syntax error
k(f);return 0;
}
```

Ezek után természetesen törlendő az **MV** és a **v0** utasításunk.

20. fejezet: Inkrementálás és dekrementálás

Egy változó értékének eggyel növelése vagy csökkentése nagyon gyakori művelet a programokban, nem véletlen hogy a C nyelv is beépített funkciót tartalmaz e feladatra. Nekünk is illik megoldanunk.

E feladatra ugyanúgy a ++ és -- jeleket használjuk majd, mint a C nyelv. 2 rutin(csoport)ban kell a kódot változtatnunk ehhez: az „**ertekadas**” nevűben, és azokban amik az unáris operátorokat tesztelik le.

Feleslegesnek tartom e helyütt a kódot bemásolni, lényeg az, hogy a szintaxis látható a következő példaprogramokból:

```

#c@c:2=3;
#c@c=7;
"Inkrementálás:\n"
{ | 6
?c @c:2; /;
?c @c; /;
/;
#c++@c:2;
#c++@c;
|}

#c@c:2=20;
#c@c=27;
"Dekrementálás:\n"
{ | 4
?c @c:2; /;
?c @c; /;
/;
#c--@c:2;
#c--@c;
|}

```

A futási eredmény:

```

Inkrementálás:
3
7

4
8

5
9

6
10

7
11

8
12

Dekrementálás:
20
27

19
26

18
25

17
24

#s@S="ABCD\n";
?s @S;
"Inkrementálás:\n"
#s++@S[2];
?s @S;
"Dekrementálás:\n"
#s--@S[1];
?s @S;

```

Eredménye:

```

ABCD
Inkrementálás:
ABDD
Dekrementálás:
AADD

[#c@t=100] // Memória foglалás 100 darab unsigned char értéknek a "t" nevű tömbbe
#i@i = 0; // i=0
{ | 10 // ciklus 10-szer
#c@t[#i@i]=#i@i
?i @i // print i
"-edik elem = "
?c @t[#i@i] /;
#i++@i // i++
|}
"Növelés után" /;

```

```

#i@i = 3; // i=3
{[ 5; #c++@t[#i@i]; #i++@i; ]} // Itt növelem a 3 - 7 indexű elemeket 1-el

#i@i = 0; // i=0
{[ 10 // ciklus 10-szer
if((#i@i) == 3) T "Innentől nőttek meg az értékek 1-el" /;
if((#i@i) == 8) T "Eddig volt növelve, innentől nem növeltük meg őket" /;
?i @i // print i
"-edik elem = "
?c @t[#i@i] /;
#i++@i
]}

XX

```

Eredménye:

```

0-edik elem = 0
1-edik elem = 1
2-edik elem = 2
3-edik elem = 3
4-edik elem = 4
5-edik elem = 5
6-edik elem = 6
7-edik elem = 7
8-edik elem = 8
9-edik elem = 9
Növelés után
0-edik elem = 0
1-edik elem = 1
2-edik elem = 2
Innentől nőttek meg az értékek 1-el
3-edik elem = 4
4-edik elem = 5
5-edik elem = 6
6-edik elem = 7
7-edik elem = 8
Eddig volt növelve, innentől nem növeltük meg őket
8-edik elem = 8
9-edik elem = 9

```

Mint látható tehát, a ++ és -- operátorokat használhatjuk normál változókra és tömbváltozókra is egyaránt, valamint stringváltozókra is. Utóbbi egy kicsit „előreszaladás” a témában, mert a stringekkel részletesebben a következő fejezetben foglalkozunk. Sajnos azonban az úgy volt, hogy elkezdtem írni azt a fejezetet, de közben eszembe jutott hogy illik betenni az inkrementálás és dekrementálás lehetőségét is a nyelvbe, s ez messze könnyebb volt mint a stringek kezelését kibővíteni egyéb más, de szükséges funkcióval, s így mert épp volt pár szabad percem, megcsináltam azt a mintegy negyed órás munkaközi szünetemben... S akkor már megoldottam a stringekre is.

Unáris operátorként használva őket:

```

"Inkrementálás:\n"
#c@c=7; "c=" ?c @c; /;
#c@d=++@c;
"d=" ?c @d; /;
"c=" ?c @c; /;
"c+1=" ?c ++@c; /;
/;

"Dekrementálás:\n"
#c@c=7; "c=" ?c @c; /;
#c@d=--@c;
"d=" ?c @d; /;
"c=" ?c @c; /;
"c-1=" ?c --@c; /;
/;

```

Eredménye:

```

Inkrementálás:
c=7
d=8

```

```
c=7
c+1=8
```

```
Dekrementálás:
c=7
d=6
c=7
c-1=6
```

```
"Inkrementálás:\n"
#c@c:2=7; "c=" ?c @c:2; /;
#c@d=++@c:2;
"d=" ?c @d; /;
"c=" ?c @c:2; /;
"c+1=" ?c ++@c:2; /;
/;

"Dekrementálás:\n"
#c@c:2=7; "c=" ?c @c:2; /;
#c@d=--@c:2;
"d=" ?c @d; /;
"c=" ?c @c:2; /;
"c-1=" ?c --@c:2; /;
/;
```

Eredménye ugyanaz mint az előzőé.

Nagyon FONTOS MEGÉRTENI, hogy ezen 2 utóbbi esetben, amikor a ++ és/vagy -- operátoraink „jobbértékben” szerepelnek, ekkor tehát NEM változtatják meg a változónak az értékét, hanem azt csinálják, hogy egyszerűen az egész, tőlük jobbra eső aritmetikai kifejezés értékét kiszámolják, majd azt megnövelik (vagy épp csökkentik) 1-el, s az így kapott eredményt adják vissza annak a rutinnak, ami meghívta őket! Ezesetben tehát a változó eredeti tartalma NEM módosul, mert nem is módosulhat - inkrementáló vagy dekrementáló unáris operátorunknak halvány fogalma sincs róla, ami értéket kap az egy szimpla változóból származik, vagy egy 30 soros rém bonyolultan összetett aritmetikai kifejezés végeredménye!

Az előbbi esetek azonban mások, azoknál ténylegesen magának a változónak a tartalma módosul, mert az balérték, s nem jobbérték.

21. fejezet: Stringkezelés

Nem minden programnyelv ad beépített lehetőséget stringek kezelésére — például a C sem. Idegenkednek is ám tőle emiatt a kezdő programozók... Mi is lehetnének enélkül, mert tömbjeink már vannak, megoldhatnánk azokkal is a stringkérdést. Na de elég baj nekünk már az is, hogy a gyorsaság érdekében a változóinkat ilyen furán kell jelölnünk, nem fogjuk magunkat még külön azzal is szopatni, hogy a stringeket se kezelhessük kényelmesen, amikor pedig azok épp az az adattípus, amit talán a legislegtöbbször használunk egy programnyelvben!

Nálunk tehát igenis lesz külön olyan adattípus, hogy „string”. Ezt természetesen az „s” karakterrel jelöljük majd kedvenc casting operátorunk, a „#” után. De egyelőre még ne szaladjunk ilyen messzire, előbb meg kell alkossunk magunknak egy string-osztályt! Hogy semmiképp se keveredhessen ezen osztály neve valami más C++ osztály nevével, legyen a neve az, hogy MAUSTRING.

Ami ezen osztály konkrét felépítését illeti, meg a hozzá tartozó programokat, e helyütt nem közlöm őket, egyetlen nagyobb, s pár icipici kivétellel. Ennek oka az, hogy a stringkezelés megoldása az eddigi talán legislegocsmányabb, legutálatosabb, legrohadtabb, legundorítóbb „favágó munka”, amivel csak találkoztam a programnyelvem fejlesztése során! Különben előre sejtettem, hogy ez így lesz, igazam is lett, épp csak rémségesen gyűlölöm, amikor ennyire igazam van...

Ezen tény, hogy ez efféle „favágó” munka, ez nem abból fakad, mintha ezt valami rém bonyolult dolog volna megalkotni. Épp ellenkezőleg: Pont arról van szó, hogy a világon semmiféle különös képességet sem igényel a megcsinálása, nincs benne semmi „szép”, semmi „kihívás”, semmi szellemi izgalom, gyakorlatilag e melóból minden hiányzik, ami öröm a programozásban. Ugyanakkor ezen unalmas melóból van nemcsak bőven, de DÖGIVEL, mert gondoljunk csak bele: erre a mocskos, undorító stringtípusra meg kell alkotnunk gyakorlatilag minden létező műveletet, mint amit bármi más típus is tud, csak még ezenfelül sokkal de sokkal több minden ótvar szarságot is, mert ugye stringeket nemcsak konstansokból gyárthatunk, de minden létező numerikus típusból is; stringjeinket át is kell tudnunk konvertálni efféle számokká; össze kell tudnunk fűzni stringeket, stringeket kell tudnunk képezni más stringek egy részéből, stringeket össze kell tudnunk hasonlítani, le kell tudnunk kérdezni a hosszukat, meg még a franc se tudja mi mindent jó ha tudunk csinálni a stringekkel. Elég ehhez csak megnézni valamely ismertebb programnyelvnek a stringosztályát, mennyi temérdek függvénye van rá, mondjuk ilyen nyelv a C++ maga is, vagy a PHP, vagy akármelyik másik! Ez nem is véletlen, mert tényleg a stringek a legsokrétűbben használt adattípus minden nyelvben.

Valójában, a stringekkel kapcsolatos képességeink fejlesztését csak abbahagyni lehet, befejezni soha, mert soha nem lesz vége. Akármennyire is úgy érezzük, nyelvünk már „full-extra-de-luxe”, bármikor felbukkanhat valami igény, hogy jaj de jó is lenne, ha még ezt is tudná, és hétszentség, hogy ezen esetek 99%-a olyan lesz, hogy e felmerülő igény stringekkel lesz kapcsolatos!

Ugyanakkor azonban ezen igények kielégítése programozástechnikailag annyira nem egy nagy „vasziszdasz”, hogy kifejezetten unalmas és lélekölő leprogramozni őket. Nélkülözhetetlenek, de dögunalmasak. Mondom, ezt tudtam már előre, nem is véletlen, hogy halogattam amíg csak tehettem...

Most sem vagyok kész mindennel, csak a legfontosabbakkal. Majd bővítem a stringosztály képességeit még a későbbiekben, ha lesz rá lelkeröm, de már nagyon tele van a hócipőm vele, pihenésképpen mással foglalkozom, azonban mert rend a lelke mindennek, legalább nagyjából leírom ide, mire jutottam velük. A rutinokat azonban már azért se érdemes becopyzni ide, mert ahogy fejlődnek a képességeik, változhat a kódjuk is, ha nem is alapvetően, de úgy, hogy beszűrődik beléjük egy-egy újabb sor, mint például egy újabb case-ág, vagy hasonló.

Lényeg az, hogy az F struktúrába felvettem e sort:

```
MAUSTRING S[256];
```


Itt vannak a stringjeink, nevükre ugyanazon szabályok érvényesek, mint a változónevekre. Mindegyik egy MAUSTRING struktúrát tartalmaz. E struktúra nemcsak a stringre mutató pointert tartalmazza, de a string hosszát is, mert arra nagyon gyakran van szükség, s így hogy nem kell minden alkalommal végigszaladni a stringen a záró nullabájtot keresve, gyorsabb a végrehajtás. Ennek ellenére, a string mindig tartalmazza a záró nullabájtot is, de az nem számít bele a hosszba. A string karakterei nullától számozódnak, tehát ha a hossz 3, akkor a karakterek a 0,1 és 2 pozíciókon tárolódnak, a 3-adik pozíción már maga a záró nullabájt van.

Ezen osztály továbbá tartalmazza a string úgynevezett „maximális” hosszát is. Ez nem azt jelenti hogy annál nagyobb hosszúságú stringet ne lehetne értékül adni neki, csak azt jelenti, hogy ha olyan stringet akarunk értékül adni amelynek hossza kisebb mint ezen maximális hossz, akkor nem foglal le újabb memória-területet, mert minek, akkor csak bemásolja a régi helyére! Ez is gyorsítja a végrehajtást.

Azt is fontos tudni, hogy a konstruktor garantálja, hogy kivétel nélkül minden létrejövő stringnek van legalább 1 bájt lefoglalva, a nulladik pozíció, amin természetesen egy CHR\$(0) bájt szerepel, azaz a string végét lezáró nullabájt. Azaz minden string már kezdetben egy szabályos, nulla bájjal lezárt üres string. Aminek a hossza nullaként van visszaadva nekünk, mert e hosszba nem számítódik bele a záró nullabájt.

Példák a használatra:

```
#s@a = "macska" // értékadás a stringnek konstanssal
?s @a; // A stringváltozó kiírása
```

Két stringváltozó összefűzése:

```
#s@c = (@a) + (@b);
"Összeg:" /;
?s @c; /;
```

Másik példa:

```
#s@a = "Egy \"jó\" macska"
?s @a /;
"A string hossza: " ?l !@a /;

#s@b = "kutya"
?s @b; /;
"A string hossza: " ?l !@b /;

#s@c = (@a) + " az kérlekalássan egy " + (@b);
"Összeg:" /;
?s @c; /;
```

A program eredménye:

```
Egy "jó" macska
A string hossza: 16
kutya
A string hossza: 5
Összeg:
Egy "jó" macska az kérlekalássan egy kutya
```

Látható a fenti példából, hogy a string hosszát a „!” unáris operátorral képezzük, pontosabban kérdezhetjük le, s az is látható, hogy a szám amit visszaad, nem a

string KARAKTERszámát jelenti, hanem a BÁJTszámát, ami nagyon nem mindegy, ugyanis UTF-8 kódolás esetén az ékezetes betűink bizony egynél több bájton vannak letárolva...

A stringet meg is fordíthatjuk a ~ unáris operátorral:

```
#s@c = ~@c;
```

Ennél is ügyeljünk rá, hogy ez bizony bájtonként fordítja meg a stringet, s emiatt kivétel nélkül minden UTF-8 kódolású több-bájtos karakter elromlik... nyilván kell csinálni majd valami rutint, ami egy így elrontott stringet „kijavít”. Majd ha lesz rá türelmem, megoldom ezt is, jelenleg vannak fontosabb dolgaim. Ez igazán csak egy százhuszonhatodrangú mellékes részletkérdés. Csak bemutatom, hogy ilyen esetben ebből a szövegből:

```
Egy "jó" macska az kérlekalássan egy kutya  
Ez lesz:  
aytuk yge nass;ĀlakelrĀk za akscam "³Āj" ygE
```

Stringekből nem csak 2 darabot adhatunk össze egymás után hanem akármennyit, s ezt vegyíthetjük konstansokkal is:

```
#s@c = (@a) + " az kérlekalássan egy " + (@b);
```

Stringek egyes karaktereire a tömbökhöz hasonlóan hivatkozhatunk. Természetesen minden stringünk unsigned char típusú bájtokból áll:

```
#s@c[2]='2;  
#s@c[3]=64;
```

Konkrét példa:

```
#s@S = "123456789";  
?s @S; /;  
#i@i=0;  
{| 9  
#s@S[#i@i] +=1;  
#i@i = ++#i@i;  
|}  
?s @S; /;  
XX
```

A program eredménye:

```
123456789  
23456789:
```

E fenti programot így is írhattam volna, zöld színnel kiemelem a megváltoztatott sort:

```
#s@S = "123456789";  
?s @S; /;  
#i@i=0;  
{| 9  
#s++@S[#i@i];  
#i@i = ++#i@i;  
|}  
?s @S; /;  
XX
```

Érdemes tudnunk, hogy amennyiben egy stringnek annyiadik elemére hivatkozunk eképp, azaz tömbindexszel, amennyiedik elem momentán nem is létezik

az ő esetében, mert a string jelenlegi hossza kisebb nála, akkor kibővíti a stringet olyan hosszúságúra, hogy létezzék annyiadik karaktere is. Eközben a string régi tartalma nem vész el, hanem megőrződik! Az új, megnövelt hosszúságú string esetében is garantált, hogy a legvége egy nullabájttal lesz lezárva, de amennyiben ez az automatikus stringhossznövelés olyankor következik be, amikor a stringet jobbtértékként szerepeltetjük, akkor bizony az így visszaadott karakter tartalma véletlenszerű hogy micsoda, mert ott tetszőleges memóriaszemét lehetséges.

Vagyis, string egy karakterére jobbtértékként is hivatkozhatunk a [] indexoperátorral, s azt is jó tudni, hogy stringet számként értelmezve azt a számot kapjuk eredményül, ami a legelején szerepel:

```
#s@s="234kutya";
#c@c=#s@s;
?c @c; /;
#c@d=#s@s[4];
? @d; /;
XX
```

A program eredménye:

```
234
u
```

Nézzük aztán e programot:

```
#l@i=8;
{
? ("52cd"+"efghijk"[#l@i]) ; /;
#l--@i;
}((#l@i)>0)
? "65cdefghijk" ; /;
? ("65cdefghijk") ; /;
XX
```

Eredménye:

```
i
h
g
f
e
d
c
2
65cdefghijk
A
```

Ugye milyen cuki?! A hátultesztelő ciklusban nem is képeztünk igazi stringváltót, csak egy konstansokból álló stringkifejezésünk van, de EZT IS indexelhetjük, ennek is kiolvashatjuk a tetszőleges karakterét! Láthatjuk továbbá, mi a különbség a stringkonstans kétfajta kiíratása közt:

Ez:

```
? "65cdefghijk" ; /;
```

kiírja magát az egész stringet.

Ez azonban:

```
? ("65cdefghijk") ; /;
```

azt a KARAKTERT írja ki, aminek az ASCII kódját az utána következő unsigned char típusú aritmetikai kifejezés határozza meg! Az egy zárójeles kifejezés. Azon belül egy konstans string van. Ezt az interpreter kötelességszerűen átalakítja unsigned char értéké, ami úgy megy, hogy kiolvassa, miféle numerikus

konstans van a string első valahány bájtján. Ott „65” van, ezt adja vissza. Tehát az a karakter íródik ki, aminek az ASCII kódja 65, ez pedig az „A” karakter.

```
#s@s="234kutya";  
?s @s; /;  
?s (@s)-2; /;  
XX
```

E fenti program eredménye:

```
234kutya  
234kut
```

Ugyanis mint látható a progi utolsó előtti sorából, azt a stringet iratjuk ott ki, amiből előbb levontuk a „2” numerikus értéket. Na most, a mau nyelvben az hogy egy stringből levonunk egy számot, az azt jelenti, hogy annyi bájtal csökkentjük a string hosszát.

Fontos tudni, a fenti példában az **@s** stringünk maga változatlan maradt, csak így lett kiírva! Ha ténylegesen változtatni szeretnénk az **@s** string hosszán, kiírás előtt, azt így kéne megoldanunk:

```
#s@s="234kutya";  
?s @s; /;  
#s@s=(@s)-2;  
?s @s; /;  
XX
```

Eredménye:

```
234kutya  
234kut
```

Ez a program pedig:

```
#l@i=8;  
{(  
? ("52cd"+"efghijk"[#l@ -- --k]) ; /;  
#l--@i;  
)}((#l@i)>0)  
XX
```

azt mutatja meg, hogy a dekrementáló operátorunk (persze az inkrementáló is) egymás után többször is alkalmazható. Valamint, hogy a változó neve a mau nyelvben ténylegesen lehet tetszőleges aritmetikai kifejezés! Erről a sorról van szó:

```
? ("52cd"+"efghijk"[#l@ -- --k]) ; /;
```

Itt ugyanis a stringkifejezés indexe ez:

```
[#l@ -- --k]
```

Ez pedig azt mondja, hogy az indexet jelölő számot egy „**l**” típusú változó tárolja, aminek a neve az a szám, ami a „**k**” karakter ASCII kódjának kettővel való csökkentése!

A progi kimenete ez:

```
i  
h  
g  
f  
e  
d  
c  
2
```

Rész-stringek képzésére példaprogram:

```

#s@s="101kiskutya";
#s@g="film"
?s @s; /; // Kiirja: 101kiskutya
?s @g; /; // Kiirja: film
#s@r=[,5]@s;
?s @r; /; // Kiirja: 101ki
#s@r=[,13](@s)+(@g);
?s @r; /; // Kiirja: 101kiskutyafi
#s@r=[0,4]@s;
?s @r; /; // Kiirja: 101k
#s@r=[5,]@s;
?s @r; /; // Kiirja: skutya
#s@r=[,]@s;
?s @r; /; // Kiirja: 101kiskutya
?s [4,9](@s)+(@g); /; // Kiirja: iskutyafi
?s [4,59](@s)+(@g); /; // Kiirja: iskutyafiln
?s [700,59](@s)+(@g); /; // Semmit nem ír ki
?s [2,0](@s)+(@g); /; // Semmit nem ír ki
XX

```

Eredménye:

```

101kiskutya
film
101ki
101kiskutyafi
101k
skutya
101kiskutya
iskutyafi
iskutyafiln

```

Látható a fentiekből a **[,]** operátor működése: A teljes, öt követő stringkifejezésre vonatkozik, valamint, ha a vessző előtt semmi sincs megadva, akkor a string elejét tekinti kezdőpozíciónak, ha pedig a vessző után nincs megadva semmi, vagy ha ott akkora szám van ami „túlérne” a string hosszán, akkor a string végéig vesz mindent. Ha azonban kezdőpozíciónak van megadva túl nagy szám, vagy ha a veendő karakterek számának kifejezetten épp nullát adunk meg, akkor üres stringgel tér vissza, s eképp semmit se ír ki.

Stringek összehasonlítása:

```

#s@s="Álom";
#s@f="álom";
"s=" ?s @s; " f=" ?s @f; /;
if(#s((@f)==(@s))) T "Egyenlőek!\n"
E "Nem egyenlőek!\n"
if(#s((@f)==(@s))) T "Egyenlőek!\n"
E "Nem egyenlőek!\n"
if(#s((@f)!=(@s))) T "f != s !\n"
E "f == s !\n"
if(#s((@f)<(>(@s))) T "f != s !\n"
E "f == s !\n"
if(#s((@f)>(@s))) T "f>s !\n"
E "f<=s !\n"
if(#s((@f)<(>(@s))) T "f<s !\n"
E "f>=s !\n"
if(#s((@f)<=(>(@s))) T "f<=s !\n"
E "f>s !\n"
if(#s((@f)>=(>(@s))) T "f>=s !\n"
E "f<s !\n"
XX

```

Eredménye:

```

s=Álom f=álom
Nem egyenlőek!
Nem egyenlőek!
f != s !
f != s !
f>s !
f>=s !
f>s !
f>=s !

```

Na most, nyilván felmerül a kérdés a Tisztelt Olvasóban, miért van ilyen „bonyolultan” megadva az összehasonlítás az „if” után. Nem bonyolult pedig az, csak tisztában kell lenni nyelvünk alapelveivel... Ez pedig az, hogy minden utasítás megszabja, miféle típusú paramétert vár el. Az **if**, az egy unsigned char számot. Ha tehát mi stringeket akarunk összehasonlítani, kell a **#s** casting operátor, ami azt mondja meg, hogy egy stringkifejezés jön, azt kell kiértékelnie. E stringkifejezés természetesen egy stringet ad vissza eredményül, mi mást is amikor épp azért STRINGkifejezés... de mert összehasonlító műveletről van szó e stringkifejezésben, ezért az eredmény egy olyan string lesz, ami csak 2 bájtól áll: az **[1]** indexű bájt mindig a stringzáró nullabájt, a **[0]** indexű pedig egy CHR\$(0) bájtot tartalmaz ha az összehasonlítás eredménye HAMIS, és CHR\$(1) bájtot, ha az összehasonlítás IGAZ. E bájtot aztán a kifejezéskiértékelő rutin annak rendje szerint unsigned char értékké alakítva adja vissza az **if** utasításunknak, ami ennek megfelelően cselekszik.

A fenti példában az **if**-et követő legelső zárójel és a párja tulajdonképpen felesleges, nem szükséges, de szerintem nem árt kitenni, mert akkor már igenis szükséges, ha további összehasonlítások is vannak, amiket netán az **&&** vagy a **||** operátorokkal kombinálni akarunk.

Na és most egy fontos, lényegi kérdés, hogy ugyan miféle szabályok szerint is hasonlítja össze a stringjeinket a „rencer”, azaz a mau interpreterünk... Mert annak nem lenne jó vége, ha bájtonként, tekintve hogy abból csak rémséges összevisszaság „eredményeződne”, mert ékezetes karaktereink az UTF-8 kódolásban ugyebár több-bájtosan vannak kódolva... Most jön mindjárt az a rutin amit a stringekkel kapcsolatban mégis beleírok ide a könyvbe, s ez bizony elég hosszú lesz!

Meg kell oldanunk tehát az UTF-8 kódolású karaktereket tartalmazó stringek összehasonlítását, mely ahhoz is kell, sőt, alapvető követelmény, hogy ha valamiért valahogy majd rendezni akarunk stringeket, az a nekünk tetsző sorrendet eredményezze. Na de miféle sorrend is tetsszen nekünk? Talán a „hivatalos”? Az bizony így hangzik, idézem a magyar Wikipédiából a „szabályzatot”:

d) A magánhangzók rövid és hosszú változatát jelölő betűk (a – á, e – é, i – í, o – ó, ő – ö, u – ú, ü – ű) a kialakult szokás szerint mind a szavak elején, mind pedig a szavak belsejében azonos értékűnek számítanak a betűrendbe sorolás szempontjából. A magánhangzó hosszú változatát tartalmazó szó tehát meg is előzheti a rövid változatút.

Na most e „szabályzattól” én hideglelést kapok, herótom és hányingerem van tőle, s azóta is hogy valamikor gyermekkoromban megtudtam hogy ez lenne a magyar „szabály” a névsorra, azóta is ha csak ez az eszembe jut, idegrángás környékez, és közepes súlyosságú agyérgörcs! Szerintem ezt csak valaki IQ-negatív idióta találhatta ki, miután a maradék eszét is elitta félig. Számomra már az i-í is teljesen külön betű, de az a-á aztán pláne! Engem ZAVAR egy szöszedetben ez a keveredés. Ha vége az a-val kezdődő szavaknak s jön egy á-val kezdődő, azt hiszem nincs is több az a-val kezdődő szavakból, s erre kiderül hogy ja de mégis! Megesküdtem rá, hogy az ÉN programnyelvemben nem ilyen zagyva összevissza ~~rend~~ OCSMÁNY RENDETLENSÉG lesz. KERÜL AMIBE KERÜL programozásra fordított időben, „szopásban”, munkában, kínlódásban, de NEM ez lesz! Sőt: nálam jöjjenek előbb a NAGY betűk, aztán a kicsik, nem vegyesen! Azaz, efféle sorrendet akartam felállítani:

AaÁäÄbBcCcDdEeÉéFf... azt hiszem érthető.

Ez a rendezés semmiféleképp sem zavarhatja a nyelvem esetleges angol felhasználóit, mert nekik teljesen mindegy, az ő nyelvükben nincsenek ékezetes karakterek. Ugyanakkor e rend jó szerintem a németeknek is.

Egy efféle rendezésnek a lelke egy olyan rutin, ami a 2 stringet várja paraméterül, vagy legalábbis a stringekre mutató pointert, majd visszaad egy számot ami negatív ha az első string kisebb mint a második, nullát ha egyenlők, és pozitív számot ha az első nagyobb mint a második. Íme a rutin, sajnos ez valóban hosszú, de mit se lehet tenni, mert az UTF-8 kódolás is igencsak bonyolult...:

```
int hasonlit(const void *elso, const void *masodik) { // Ez az összehasonlító függvény a qsorthoz,
// vagy a MAUSTRING-ek „if” műveleteihez

union unijon {char utf8byte;
struct bitmezo {
unsigned int bit0:1;
unsigned int bit1:1;
unsigned int bit2:1;
unsigned int bit3:1;
unsigned int bit4:1;
unsigned int bit5:1;
unsigned int bit6:1;
unsigned int bit7:1;
} utf8;
};

union unijon bitek;

char *a;
char *b;

a= *((char **)elso); // Az "elso" és a "masodik" ugyanis egy karakterláncra mutató pointerre mutató pointer...
b= *((char **)masodik);

int aindex,bindex; // A stringek indexelésére
char akta, aktb; // az aktuális beolvasott bájtok
char akta1,akta2,akta3,akta4,akta5;// további utf-8 kódbájtok
char aktb1,aktb2,aktb3,aktb4,aktb5;
char akod,bkod; // az általam adott kódok
aindex=0;bindex=0;

while(1) { // !!!!!!!!!!!!! Nem szokványos ciklus! !!!!!!!!!!!!!

akta=a[aindex];
aktb=b[bindex];

if((akta==0)|| (aktb==0)) {
if(akta>aktb) return 1;
if(akta<aktb) return -1;
return 0;
} // if vége

akod=255;if(akta<128) akod=kodjain[(int)akta]; else {
akta1=akta2=akta3=akta4=akta5=0;
bitek.utf8byte=akta;

if((bitek.utf8.bit7)&&(bitek.utf8.bit6)&&(bitek.utf8.bit5==0)) // A kód kétbájtos
{
// beolvassuk a következő bájtot
akta1=a[++aindex];if(akta1==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
} // kétbájtos kód if vége

if((bitek.utf8.bit7)&&(bitek.utf8.bit6)&&(bitek.utf8.bit5)&&(bitek.utf8.bit4==0)) // A kód 3 bájtos
{
// beolvassuk a következő bájtokat
akta1=a[++aindex];if(akta1==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
akta2=a[++aindex];if(akta2==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
} // 3 bájtos kód if vége

if((bitek.utf8.bit7)&&(bitek.utf8.bit6)&&(bitek.utf8.bit5)&&(bitek.utf8.bit4)&&(bitek.utf8.bit3==0)) // A kód 4 bájtos
{
// beolvassuk a következő bájtokat
akta1=a[++aindex];if(akta1==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
akta2=a[++aindex];if(akta2==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
akta3=a[++aindex];if(akta3==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
} // 4 bájtos kód if vége
```

```

if((bitek.utf8.bit7)&&(bitek.utf8.bit6)&&(bitek.utf8.bit5)&&(bitek.utf8.bit4)&&(bitek.utf8.bit3)&&(bitek.utf8.bit2==0))
// A kód 5 bájtós
{
// beolvassuk a következő bájtokat
akta1=a[++aindex];if(akta1==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
akta2=a[++aindex];if(akta2==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
akta3=a[++aindex];if(akta3==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
akta4=a[++aindex];if(akta4==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
} // 5 bájtós kód if vége

if((bitek.utf8.bit7)&&(bitek.utf8.bit6)&&(bitek.utf8.bit5)&&(bitek.utf8.bit4)&&(bitek.utf8.bit3)&&(bitek.utf8.bit2)&&(bitek.utf8.bit1==0)) // A kód 6 bájtós
{
// beolvassuk a következő bájtokat
akta1=a[++aindex];if(akta1==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
akta2=a[++aindex];if(akta2==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
akta3=a[++aindex];if(akta3==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
akta4=a[++aindex];if(akta4==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
akta5=a[++aindex];if(akta5==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
} // 6 bájtós kód if vége

// Á = 12 #c3 81
// á = 13 #c3 a1
// Ä = 14 #c3 84
// ä = 15 #c3 a4
// Ê = 24 #c3 89
// é = 25 #c3 a9
// Í = 34 #c3 8d
// í = 35 #c3 ad
// Ò = 48 #c3 93
// ó = 49 #c3 b3
// Õ = 50 #c3 96
// ö = 51 #c3 b6
// Ö = 52 #c5 90
// ô = 53 #c5 91
// Ú = 66 #c3 9a
// ú = 67 #c3 ba
// Û = 68 #c3 9c
// ü = 69 #c3 bc
// Ű = 70 #c5 b0
// ű = 71 #c5 b1
// - = 5 #e2 80 94 ----- nagyköjtőjel
// = 2 #c2 a0 ----- nem törhető szóköz

{register unsigned int index; index=(unsigned int)akta1;akod=utf8kodjain[index];} // Ez az ocsmányság a fordítóprogram
miatt van:
// ha ugyanis char értékkel indexelek egy tömböt, és nem int-tel, akkor ez a barom amit biztos valami IQ-negatív ürge
programozott,
// warningokkal idegesít engem! Nem tudom, miből vonta le a készítője azt a meglepő következtetést, hogy csak int
értékkel lehet indexelni, s aki
// char értékkel akar az tutira elcseszett valamit! Igenis gyakran van úgy, hogy nekem egy tömbben 256-nál kevesebb
elemem van, s akkor nagyonis elég a
// char változó! S mert utálok a warningokat, inkább így mondom meg a fordítónak hogy nem vagyok bolond és tényleg ezt
akarom.
// Más megoldás is lehetséges volna, nekem ez tetszik.

if((akta==0xe2)&&(akta1==0x80)&&(akta2==0x94)) {akod=5;} // nagyköjtőjel
if((akta==0xc2)&&(akta1==0xa0)) {akod=2;} // nem törhető szóköz
// A többi engem érdeklő kód biztos hogy 0xc3-mal vagy 0xc5-tel kezdődik.
// De mivel mindegyik 2 bájtós kód, valamint mivel a 0xc5-tel kezdődő kódok második bájtja nem egyezik meg egyetlen
0xc3-mal kezdődő
// kód második bájtjával sem, emiatt az első bájtokat egyáltalán nem is kell figyelembe vennem.
} // else vége

bkod=255;if(aktb<128) bkod=kodjain[(int)aktb]; else{

aktb1=aktb2=aktb3=aktb4=aktb5=0;
bitek.utf8byte=aktb;

if((bitek.utf8.bit7)&&(bitek.utf8.bit6)&&(bitek.utf8.bit5==0)) // A kód kétbájtós
{
// beolvassuk a következő bájtot
aktb1=b[++bindex];if(aktb1==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
} // kétbájtós kód if vége

if((bitek.utf8.bit7)&&(bitek.utf8.bit6)&&(bitek.utf8.bit5)&&(bitek.utf8.bit4==0)) // A kód 3 bájtós
{
// beolvassuk a következő bájtokat
aktb1=b[++bindex];if(aktb1==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
aktb2=b[++bindex];if(aktb2==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
} // 3 bájtós kód if vége

if((bitek.utf8.bit7)&&(bitek.utf8.bit6)&&(bitek.utf8.bit5)&&(bitek.utf8.bit4)&&(bitek.utf8.bit3==0)) // A kód 4 bájtós
{
// beolvassuk a következő bájtokat
aktb1=b[++bindex];if(aktb1==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
aktb2=b[++bindex];if(aktb2==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
aktb3=b[++bindex];if(aktb3==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
} // 4 bájtós kód if vége

```



```

if((bitek.utf8.bit7)&&(bitek.utf8.bit6)&&(bitek.utf8.bit5)&&(bitek.utf8.bit4)&&(bitek.utf8.bit3)&&(bitek.utf8.bit2==0))
// A kód 5 bájtós
{
// beolvassuk a következő bájtokat
aktb1=b[++bindex];if(aktb1==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
aktb2=b[++bindex];if(aktb2==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
aktb3=b[++bindex];if(aktb3==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
aktb4=b[++bindex];if(aktb4==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
} // 5 bájtós kód if vége

if((bitek.utf8.bit7)&&(bitek.utf8.bit6)&&(bitek.utf8.bit5)&&(bitek.utf8.bit4)&&(bitek.utf8.bit3)&&(bitek.utf8.bit2)&&(bitek.utf8.bit1==0)) // A kód 6 bájtós
{
// beolvassuk a következő bájtokat
aktb1=b[++bindex];if(aktb1==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
aktb2=b[++bindex];if(aktb2==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
aktb3=b[++bindex];if(aktb3==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
aktb4=b[++bindex];if(aktb4==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
aktb5=b[++bindex];if(aktb5==0) {printf("Érvénytelen UTF-8 kód!\n");exit(EXIT_FAILURE);}
} // 6 bájtós kód if vége

[register unsigned int index; index=(unsigned int)aktb1;bkod=utf8kodjaim[index];] // Lásd fentebb, miért van ez így

if((aktb==0xe2)&&(aktb1==0x80)&&(aktb2==0x94)) {bkod=5;} // nagyköötjel
if((aktb==0xc2)&&(aktb1==0xa0)) {bkod=2;} // nem törhető szóköz
// A többi engem érdeklő kód biztos hogy vagy 0xc3-mal vagy 0xc5-tel kezdődik.
// De mivel mindegyik 2 bájtós kód, valamint mivel a 0xc5-tel kezdődő kódok második bájtja nem egyezik meg egyetlen
0xc3-mal kezdődő
// kód második bájtjával sem, emiatt az első bájtokat egyáltalán nem is kell figyelembe vennem.
} // else vége

if(akod>bkod) return 1;
if(akod<bkod) return -1;

aindex++;bindex++;

} // örök ciklus vége
return 0; // Csak mert biztos ami biztos...
}

```

Ezenfelül kell nekünk 2 statikus tömb is, a kódoknak ugyebár:

```

USC kodjaim[256];
USC utf8kodjaim[256];

```

Meg egy rutin, amit a **main** program legelején hívunk meg, s mely rutin beállítja a tömb megfelelő értékeit:

```

void UTF8KODBEALLIT(void) {
kodjaim[0]=0;
kodjaim[10]=0;

kodjaim[' ']=1; // szóköz
kodjaim[9] =0; // TAB
// Nem törhető szóköz lesz a 2
kodjaim['.']=3;
kodjaim['-']=4; // mínuszjel
// Nagyköötjel lesz az 5
kodjaim['A']=10;
kodjaim['a']=11;
// Á 12
// á 13
// Ä 14
// ä 15
kodjaim['B']=16;
kodjaim['b']=17;
kodjaim['C']=18;
kodjaim['c']=19;
kodjaim['D']=20;
kodjaim['d']=21;
kodjaim['E']=22;
kodjaim['e']=23;
// É 24
// é 25
kodjaim['F']=26;
kodjaim['f']=27;
kodjaim['G']=28;
kodjaim['g']=29;
kodjaim['H']=30;
kodjaim['h']=31;
kodjaim['I']=32;
kodjaim['i']=33;

```

```

// í 34
// i 35
kodjain['j']=36;
kodjain['j']=37;
kodjain['k']=38;
kodjain['k']=39;
kodjain['l']=40;
kodjain['l']=41;
kodjain['m']=42;
kodjain['m']=43;
kodjain['n']=44;
kodjain['n']=45;
kodjain['o']=46;
kodjain['o']=47;
// ö 48
// ó 49
// ő 50
// ø 51
// õ 52
// ô 53
kodjain['p']=54;
kodjain['p']=55;
kodjain['q']=56;
kodjain['q']=57;
kodjain['r']=58;
kodjain['r']=59;
kodjain['s']=60;
kodjain['s']=61;
kodjain['t']=62;
kodjain['t']=63;
kodjain['u']=64;
kodjain['u']=65;
// ű 66
// ú 67
// Ű 68
// ü 69
// Ū 70
// ů 71
kodjain['v']=72;
kodjain['v']=73;
kodjain['w']=74;
kodjain['w']=75;
kodjain['x']=76;
kodjain['x']=77;
kodjain['y']=78;
kodjain['y']=79;
kodjain['z']=80;
kodjain['z']=81;
// Számok:
kodjain['0']=100;
kodjain['1']=101;
kodjain['2']=102;
kodjain['3']=103;
kodjain['4']=104;
kodjain['5']=105;
kodjain['6']=106;
kodjain['7']=107;
kodjain['8']=108;
kodjain['9']=109;

utf8kodjain[0x81]=12; // Á = 12 #c3 81
utf8kodjain[0xa1]=13; // á = 13 #c3 a1
utf8kodjain[0x84]=14; // Ä = 14 #c3 84
utf8kodjain[0xa4]=15; // ä = 15 #c3 a4
utf8kodjain[0x89]=24; // É = 24 #c3 89
utf8kodjain[0xa9]=25; // é = 25 #c3 a9
utf8kodjain[0x8d]=34; // Í = 34 #c3 8d
utf8kodjain[0xad]=35; // í = 35 #c3 ad
utf8kodjain[0x93]=48; // Ò = 48 #c3 93
utf8kodjain[0xb3]=49; // ó = 49 #c3 b3
utf8kodjain[0x96]=50; // Ò = 50 #c3 96
utf8kodjain[0xb6]=51; // ö = 51 #c3 b6
utf8kodjain[0x90]=52; // Õ = 52 #c3 90
utf8kodjain[0x91]=53; // ô = 53 #c3 91
utf8kodjain[0x9a]=66; // Ū = 66 #c3 9a
utf8kodjain[0xba]=67; // ú = 67 #c3 ba
utf8kodjain[0x9c]=68; // Ű = 68 #c3 9c
utf8kodjain[0xbc]=69; // ü = 69 #c3 bc
utf8kodjain[0xb0]=70; // Ū = 70 #c3 b0
utf8kodjain[0xb1]=71; // ů = 71 #c3 b1
}

```

Ezek után a stringösszehasonlító operátoraink többi kódsora triviális, felesleges lenne e helyütt részletezni őket. Annyit jegyzek csak meg, hogy a fenti „**hasonlit**” függvény kifejezetten azért várja a paramétereit éppen így, ahogy, hogy ő maga

alkalmas legyen arra is, hogy a **qsort** nevű rendezőrutin hívja meg, ha valamikor esetleg szükségesnek találják azt. Mert minek akkor megírni egy másik összehasonlító rutint, felesleges munka lenne...

Stringjeink tehát vannak már, ezek után megírhatunk egy felettebb hasznos parancsot, azt, ami nem más mint a C nyelv „system” függvénye, s amely tetszőleges parancs végrehajtását éri el az operációs rendszernél! A parancs egy karakterlánc által kell meghatározva legyen. Nos, e célra a **SY** tokent választottam. Így működik mint ezt e 2 kis példa mutatja:

```
"Tartalomjegyzék-lista:\n"
SY "ls -l";
"Tartalomjegyzék-lista vége.\n"
```

Másik példa:

```
"Tartalomjegyzék-lista:\n"
#s@p="ls ";
#s@s="-l";
SY (@p)+(@s);
"Tartalomjegyzék-lista vége.\n"
```

Gondolom, e példák maguktól értetődőek.

Számot stringgé természetesen a casting operátorok segítségével alakítunk:

```
"Unsigned char értékek:\n"
#c@a = 243; ?c @a; /; #s@d = #c@a;
"d értéke stringként=" ?s @d; "<==\n"
#c@a = 5; ?c @a; /; #s@d = #c@a;
"d értéke stringként=" ?s @d; "<==\n"
#c@a = 0; ?c @a; /; #s@d = #c@a;
"d értéke stringként=" ?s @d; "<==\n"
#c@a = 17; ?c @a; /; #s@d = #c@a;
"d értéke stringként=" ?s @d; "<==\n"
"Signed char értékek:\n"
#C@a = 126; ?C @a; /; #s@d = #C@a;
"d értéke stringként=" ?s @d; "<==\n"
#C@a = 5; ?C @a; /; #s@d = #C@a;
"d értéke stringként=" ?s @d; "<==\n"
#C@a = 0; ?C @a; /; #s@d = #C@a;
"d értéke stringként=" ?s @d; "<==\n"
#C@a = 17; ?C @a; /; #s@d = #C@a;
"d értéke stringként=" ?s @d; "<==\n"
#C@a = -126; ?C @a; /; #s@d = #C@a;
"d értéke stringként=" ?s @d; "<==\n"
#C@a = -5; ?C @a; /; #s@d = #C@a;
"d értéke stringként=" ?s @d; "<==\n"
#C@a = -17; ?C @a; /; #s@d = #C@a;
"d értéke stringként=" ?s @d; "<==\n"
```

Eredménye:

```
Unsigned char értékek:
243
d értéke stringként= 243<==
5
d értéke stringként= 5<==
0
d értéke stringként= 0<==
17
d értéke stringként= 17<==
Signed char értékek:
126
d értéke stringként= 126<==
5
d értéke stringként= 5<==
0
d értéke stringként= 0<==
17
d értéke stringként= 17<==
-126
```

```
d értéke stringként=-126<==
-5
d értéke stringként=- 5<==
-17
d értéke stringként=- 17<==
```

Mint látható, a **char** típusú értékek egyformán 4 karakter hosszú számstringgé vannak alakítva, melynek első helye szóköz a pozitív értékek esetén, illetve egy „-” negatív előjel a negatív számok esetén. Elvárom egy szám stringgé konvertálása-kor ugyanis, hogy ne kelljen nekem vizsgálgatni azt a programban, hogy hát jaj-istenkém, ez vajon most épp milyen hosszú, s nekem kell emiatt „varázsolgatni”, hogy a helyiértékek szépen olvashatóan egymás alá kerüljenek... Francokat! Ezt intézze el a program. Minden szám olyan hosszú stringbe legyen pakolva, amilyen hosszú kell az adott számtípusba beleférő legislegnagyobb elképzelhető értéknek. Ha ugyanis annál sokkal kisebb számok is elegendőek nekünk, használjunk más számtartományt lefedő számtípust.

Sajnos, ilyen rutinokat (konstruktorokat) kénytelenek vagyunk gyártani az összes létező numerikus típusunk stringgé alakításához... Bevallom, én csak a nem lebegőpontos típusokra csináltam meg. A float, double és long double esetében tehát nem, mert ott cseppet se egyértelmű, mennyi is a szükséges karakterszám, mert attól is függ, normálalakban akarjuk-e megjeleníteni, meg hány tizedesjegy pontosságig, stb. Ez pedig nagyonis annak függvénye, milyen célra akarjuk használni a számot. Ezt majd annak kell megírni, aki programot ír magának mau nyelven. Célszerűen kell erre írnia valami külön függvényt. Most ugyan még nincs lehetőség függvények írására, de ne aggódjunk, eljön annak is majd az ideje...

A rendszerváltozóink értékét is átalakíthatjuk stringekké:

```
#s@H=?H;
"A programfile hossza: " ?s @H; /;
```

Mégegy fontos parancs: Ez kissé érdekesebb, mint az eddigi unalmas „favágó” munkák. Arról van szó, hogy legyen lehetőségünk, tetszőleges string VÉGRE-HAJTÁSÁRA!

Rendszerparancsot már tudunk végrehajtani a **SY** utasítással. Csináljuk most meg, hogy stringbe letárolt „mau” programnyelven írt parancsokat is végre tudjunk hajtani!

Ez efféleképp működik:

```
#c@h=3;
#s@s="{| 5; ?c @h; /; #c++@h; |} xx"
"Kezdődik a stringparancs végrehajtása!\n"
MAU @s;
"Befejeződött a stringparancs végrehajtása!\n"
"A \"h\" értéke most: " ?c @h; /;
```

A program kimenete:

```
Kezdődik a stringparancs végrehajtása!
3
4
5
6
7
Befejeződött a stringparancs végrehajtása!
A "h" értéke most: 8
```

A string végrehajtását, mint látható, a „**MAU**” nevű parancs végzi. Ez ugyan 3 karakter látszólag, illetve valóságosan is, igazából azonban ezt a „**MA**” nevű parancs végzi a rendszerben, épp csak legelső teendőjeként leellenőrzi, a legislegelső karakter a programkódban őutána az „**U**” betű-e. Ha nem az, akkor syntax error hibaüzenettel kilép.

Nem vagyok a híve a felesleges ellenőrizgetéseknek, de efféle egzotikus dolgot mint egy string végrehajtása, feltehetőleg nem fogunk gyakran végezni, s akkor egyetlen plusz bájtt ellenőrzése még talán elfogadható, azért, hogy a forráskódból egyértelműen kiderüljön, mit is csinálunk ott. Azaz tudható legyen ez rögvest a parancs nevéből. Pláne mert azért az ilyesmi rizikós dolog, s esetleg biztonságügyileg is aggályosnak mondható. Jó azonban ha van, csak ésszel kell használni, mert néha igenis aranyat érhet!

Mint látható a példaprogram stringjéből, azt egy eddig ismeretlen, „**xx**” nevű utasítással kell lezárni. A dupla nagy X, vagyis az **XX** mint tudjuk a program végét jelzi, e dupla kis ixek pedig a „**BREAK**” utasítás. Ez direkt arra van hogy visszatérjünk rendben valami efféle huncutságból. Ezzel kell lezárni az efféle paraméterstringeket.

A **MAU** és az **xx** utasítások kódja:

```
int fuggveny_xx(F& f) { // BREAK utasítás
if(logflags[8]) {L("xx parancs: BREAK! pozíció: %lu, token:%c%c", f.P, f.a, f.a2);} return 2;
}
// -----
int fuggvenyMAU(F& f) { // stringben letárolt mau parancs végrehajtása
MAUSTRING maustring;USIL aktP;USIL aktphossz;USC *aktP;
if(f.p[f.P]!='U') {SERROR(f,3);} k(f);
maustring=ERTEKstring(f); // A string beolvasása
aktP=f.P;aktphossz=f.phossz;aktP=f.p;
f.P=0;f.phossz=maustring.hossz;f.p=maustring.s;
f.THIS->folytat();
f.P=aktP;f.phossz=aktphossz;f.p=aktP;
return 0;
}
```

Fontos tudni, hogy efféle, stringben tárolt parancs-sorozat végrehajtása közben nem működnek a címkéink! Az értékük kiolvasható ugyan, de hiába próbálunk ugrani rájuk, mert a programmemória mutatója ideiglenesen át van állítva a string kezdetére, azaz nem találja meg a címkék által jelölt pontokat. Lehet azonban ugrálni a stringen belül, tudniillik ha konkrét numerikus értéket adunk az ugróutasításunk paraméteréül, amely a string valahányadik bájtyát jelöli. Vég-eredményben egy ciklus esetén is ugróutasításokról van szó, s látjuk a fenti példán, hogy a ciklus remekül működik!

Igenám, de előre tudható, hogy hajlamosak leszünk elfeledkezni a string végébe beleírni az „xx” karaktereket... Jó lenne „bolondbiztossá” tenni e parancsot! Nos, ennek semmi akadálya: mindössze picit kell kibővítsük a rutint, úgy, hogy a paraméterül kapott string végéhez mindenféleképp fűzze hozzá magától még a végrehajtás előtt a szükséges „végződést”! Az nem baj ha mégis beleírtuk mi is az „iksz-ikszeket”, akkor legfeljebb marad a string végén pár karakter amit sosem hajt végre. Ez nem számít. Íme a módosított kód:

```
int fuggvenyMAU(F& f) { // stringben letárolt mau parancs végrehajtása
MAUSTRING maustring;USIL aktP;USIL aktphossz;USC *aktP;char xx[ ]=" ; xx";
if(f.p[f.P]!='U') {SERROR(f,3);} k(f);
maustring=ERTEKstring(f); // A string beolvasása
```

```

mauststring+=xx;
aktP=f.P;aktphossz=f.phossz;aktp=f.p;
f.P=0;f.phossz=mauststring.hossz;f.p=mauststring.s;
f.THIS->folytat();
f.P=aktP;f.phossz=aktphossz;f.p=aktp;
return 0;
}

```

STRINGTÖMBÖK

Voltak már tömbjeink, vannak már stringjeink, természetesen kell legyenek stringtömbjeink is... Na most, erre bevezethetnénk egy külön adattípust, ami bizonyos szempontokból megkönnyítené az életünket, sajnos azonban sok más szempontból nem. Emiatt úgy döntöttem, nem lesz külön adattípus. Hanem azt, hogy nem közönséges, „single” stringről van szó, hanem egy stringtömb egy tagjáról, azt az interpreterünk onnan ókumulálja ki, hogy van-e megadva a változónév után zárójel. Mármint szögletes zárójel. Természetesen stringtömbből is 256 különböző lehet nekünk, a szokott módon nevezve őket, s ezen stringtömböknek semmi közük az ugyanolyan nevű közönséges stringekhez amik nem tömbök!

Igenám, csak hogy szögletes zárójeleket már használtunk, tudniillik a string egy karakterének a jelzésére! Nos, semmi baj: ha egy stringtömb egy tagjára utalunk, mint a teljes stringre, azt dupla szögletes zárójellel jelöljük! Az egész mindjárt érthetővé válik a következő néhány példán:

```

ö#s@t="Ez egy sima string, nem tömb!\n"
// Feltöltöm értékekkel a tömböt
#s@t[[0]]="kutya";
#s@t[[1]]="macska";
#s@t[[2]]="vadbarom";
#s@t[[3]]="cica";
"Kiíratom a 4 elemű stringtömb mindegyik értékét:\n"
#i@i=0; { | 4 ?i @i; ". = " ?s @t[[#i@i]]; /; #i++@i; }
"Kiíratom a sima t string értékét is:\n"
?s @t;
"Kiíratom a 4 elemű stringtömb mindegyik értékének 3-as indexű karakterét.\n"
"Ez valójában a 4-edik karakter, mert a karakterek nullától számozzódnak.\n"
#i@i=0; { | 4 ?i @i; ". = " ?s @t[[#i@i]][3]; /; #i++@i; }
"Névsorba rendezzük a t stringtömböt (e tömb első 4 értékét).\n";
QS t, 4;
"Kiíratom a 4 elemű stringtömb mindegyik értékét a rendezés után:\n"
#i@i=0; { | 4 ?i @i; ". = " ?s @t[[#i@i]]; /; #i++@i; }
"Az 1 indexű string 3 indexű karakterét kicserélem egy kukacjelre:\n"
#s@t[[1]][3]='@';
?s @t[[1]]; /; // és kiíratom
"Felszabadítom a stringtömb memóriaterületét.\n"
[@t[[[]]];
"Most megpróbálom kiíratni belőle a nulladik indexű stringet,\n"
"de ha jól működik az interpreterem, akkor most hibajelzést kell kapjak:\n"
?s @t[[0]];

```

A program eredménye:

```

Kiíratom a 4 elemű stringtömb mindegyik értékét:
0. = kutya
1. = macska
2. = vadbarom
3. = cicá
Kiíratom a sima t string értékét is:
Ez egy sima string, nem tömb!
Kiíratom a 4 elemű stringtömb mindegyik értékének 3-as indexű karakterét.
Ez valójában a 4-edik karakter, mert a karakterek nullától számozzódnak.
0. = y
1. = s
2. = b
3. = a
Névsorba rendezzük a t stringtömböt (e tömb első 4 értékét).
Kiíratom a 4 elemű stringtömb mindegyik értékét a rendezés után:
0. = cicá
1. = kutya
2. = macska
3. = vadbarom

```

Az 1 indexű string 3 indexű karakterét kicserélem egy kukacjelre:
kut@
Felszabadítom a stringtömb memóriaterületét.
Most megpróbálok kiíratni belőle a nulladik indexű stringet,
de ha jól működik az interpreterem, akkor most hibajelzést kell kapjak:
LOG:> 2014.02.25 18:42:36 : Indexhatárátlépés tömb esetében!

Másik példa:

```
[@s[[20]]]; // Memória foglalás 20 stringnek az „s” nevű tömbbe. Feleslegesen, mert csak 1-et használunk.  
#s@s[[0]]="101kiskutya";  
#s@g="film"  
?s @s[[0]]; // Kiírja: 101kiskutya  
?s @g; // Kiírja: film  
#s@r=[,5]@s[[0]];  
?s @r; // Kiírja: 101ki  
#s@r=[,13](@s[[0]])+(@g);  
?s @r; // Kiírja: 101kiskutyafi  
#s@r=[0,4]@s[[0]];  
?s @r; // Kiírja: 101k  
#s@r=[5,]@s[[0]];  
?s @r; // Kiírja: skutya  
#s@r=[,]@s[[0]];  
?s @r; // Kiírja: 101kiskutya  
?s [4,9](@s[[0]])+(@g); // Kiírja: iskutyafi  
?s [4,59](@s[[0]])+(@g); // Kiírja: iskutyafil  
?s [700,59](@s[[0]])+(@g); // Semmit nem ír ki  
?s [2,0](@s[[0]])+(@g); // Semmit nem ír ki  
#s@s[[0]][1]=A;  
//#s@s[[0]][1]+=5;  
?s @s[[0]]; // Kiírja: 1A1kiskutya  
#s++@s[[0]][1];  
?s @s[[0]]; // Kiírja: 1B1kiskutya  
#s--@s[[0]][2];  
?s @s[[0]]; // Kiírja: 1B0kiskutya  
#s@s[[0]][2]+=7;  
?s @s[[0]]; // Kiírja: 1B7kiskutya  
#s@s[[0]][2]-=3;  
?s @s[[0]]; // Kiírja: 1B4kiskutya  
XX
```

Eredménye:

```
101kiskutya  
film  
101ki  
101kiskutyafi  
101k  
skutya  
101kiskutya  
iskutyafi  
iskutyafil
```

```
1A1kiskutya  
1B1kiskutya  
1B0kiskutya  
1B7kiskutya  
1B4kiskutya
```

Látható a fentiekből, hogy stringtömböt névsorba is tudunk már rendezni, a **qs** paranccsal, ami természetesen a qsort rendezőrutint hívja meg.

22. fejezet: File-kezelés

A file-kezelés alatt természetesen fájlok kezelését értem, olyan fájlokét, melyek (többnyire...) a merevlemezen tárolódnak. Mindenesetre olyan helyen, ami meghatározható egy úgynevezett „path”, azaz elérési útvonal által. E fájlokat kell tudnunk megnyitni, lezárni, adott pozícióra ugrani bennük, olvasni és írni. No meg, azért nem árt ám, ha le tudjuk kérdezni akár rögvést megnyitás után is a

fájl méretét, le tudjuk azt is csekkolni egy beolvasási kísérlet után, nem értük-e el véletlenül az állomány végét...

Szerintem itt se nagyon fontos közreadnom minden apró rutint amit megírtam, a lényeg a lényeg: Nálam a fájlok kizárólag „bináris” módon nyithatóak meg — erről a mániámról már alaposan értekeztem e könyv legelején!

Aztán, fájlból alapvetően kétféle létezik: amit olvasni akarunk, s amit írni. Nem bölcs dolog vegyíteni a két típust... (Szerintem). Emiatt két külön osztályt készítettünk e célra: az egyik neve az lesz, hogy „**inputfile**”, a másiké hogy „**outputfile**”.

Az input fájlok is külön adattípus nálam, természetesen ebből is 256 változónk lehetséges, és e változótípust a „**B**” karakter jelöli, mert ez olyasmi, amit **Beolva-**sunk. Az „i” és „I” karakterek amik az „input” szóra utalnának, már foglaltak más változótípusnak, ugye.

Íme rögvest egy kis mau program az input fájlok kezelésére. Előbb közlöm a teljes progit, majd utána soronként a magyarázatot, ahol szükséges:

```
#B@b="ido.cpp"; // Megnyitjuk a fájlt
if(#B@b) T "A file sikeresen meg lett nyitva!\n"
E "A file nem megnyitható!\n" XX

#l@m=#B@b; // A file mérete
"A file mérete = " ?s #l@m; " bájt\n"
{ | 30 ; // beolvassuk az első 30 karaktert
#L@c=#B@b;
if(#L((#L@c)==-1)) T "\nItt a file vége!\n" XX
if(#L((#L@c)==-2)) T "\nOlyan fájlból próbáltál olvasni, ami nem is lett megnyitva!\n" XX
? @c; // és kiírjuk
} } // Ciklus vége
/;
"Most lezárjuk a fájlt\n"
[#B@b] // Itt zártuk le, ezzel az utasítással
"Most megnyitjuk a fájlt újra\n"
#B@b="ido.cpp"; // Megnyitjuk a fájlt újra
[#B@b=6] // Elugrunk a 6-odik bájtjához
{ | 14 ; // beolvassuk innentől az első 14 karaktert
#L@c=#B@b;
? @c; // és kiírjuk
} } // Ciklus vége
/;
"Vége!\n"
XX
```

Magyarázat:

```
#B@b="ido.cpp"; // Megnyitjuk a fájlt
```

Mint látható, a fájl a „b” nevű változó által lett megnyitva, azaz ezen a változón keresztül hivatkozhatunk rá ezentúl. Itt egy konstans string adja meg neki a fájl nevét, de stringváltozón át is megadhatnánk azt, eképpen:

```
#s@b="ido.cpp";
#B@b=#s@b; // Megnyitjuk a fájlt
```

E fenti két sor is ugyanúgy megnyitná. Sőt, ennyi is elég volna:

```
#s@b="ido.cpp";
#B@b=@b; // Megnyitjuk a fájlt
```

Ugyanis egy **#B** típusú változónak kizárólag stringet adhatunk értékül, amit ő meg akar majd nyitni, na most miután amúgy is stringet vár, ezért a **@b** által jelölt változót mindenképp stringként próbálja értelmezni, akkor is ha nincs előtte explicit módon megadva a **#s** casting operátor. A fentiekből az is látszik, hogy bár a stringváltozónknak is meg a fileváltozónknak is egyaránt **@b** a neve, de e két változónak a világon semmi köze egymáshoz.

```
if(#B@b) T "A file sikeresen meg lett nyitva!\n"
E "A file nem megnyitható!\n" XX
```


E két fenti sorból az látszik, hogy az inputfile változóink átkonvertálhatóak unsigned char értéké is. (Az **if** ugyanis azt várja el). Ezesetben az inputfile típus eredménye mindig egy 0 vagy egy 1 értékű bájt - nulla, ha a változóhoz nem tartozik megnyitott állomány, és 1, ha tartozik hozzá, azaz ha a fájl épp meg van nyitva. Az **if** sorát így is írhatnám, zárójelek nélkül:

```
if #B@b T "A file sikeresen meg lett nyitva!\n"
```

Csak szerintem zárójelezve az inputfile változót olvashatóbb a kód.

```
#l@m=#B@b; // A file mérete
```

```
"A file mérete = " ?s #l@m; " bájt\n"
```

A fenti két sorból az látszik, hogy az inputfile típus castolható unsigned int értéké is, ez esetben pedig egyszerűen az épp megnyitott fájl méretét adja vissza (bájtokban). Ezen értéket aztán itt kiíratjuk stringként a **?s** utasítással.

```
{| 30 ; // beolvassuk az első 30 karaktert
```

Itt fentebb egy fix számszor lefutó ciklust indítottunk.

```
#L@c=#B@b;
```

E fenti sorban történik a fájlból a beolvasás, pontosan 1 bájt beolvasása. E beolvasás azonban nem signed vagy unsigned char, hanem signed int értéként történik, mert - amint az a következő sorban látható - le kell tudjuk tesztelni valamiképp azt is, hogy elértük-e az állomány végét.

```
if(#L((#L@c)==-1)) T "\nItt a file vége!\n" XX
```

Itt fentebb teszteljük hogy nem EOF-ot kaptunk-e, azaz nem értük-e el az állomány végét. Az **if** utáni **#L** casting operátor nélkülözhetetlen, mert az **if** alaphoz olyan kiértékelő rutint hív ami unsigned char értéket ad eredményül, de nekünk az kell, hogy a változónkat és a mínusz 1 számot még mint signed int értékeket hasonlítsa össze! Ellenben lehet mindezt kevesebb zárójellel is írni, ez is jó lenne:

```
if #L (#L@c)==-1 T "\nItt a file vége!\n" XX
```

Viszont, bár a **#L** casting operátor használata mulhatatlanul szükséges, nem kell kétszer kitenni, elég egyszer is, így:

```
if #L (@c)==-1 T "\nItt a file vége!\n" XX
```

Ugyanis eképp az egész, utána következő kifejezésre vonatkozik, s emiatt a **@c** változót is signed int-ként értelmezi. Az nem lenne jó, ha így szerepelne:

```
if (#L@c)==-1 T "\nItt a file vége!\n" XX
```

ugyanis ekkor a **@c** változót igaz hogy signed int-ként értelmezné, de a TELJES aritmetikai kifejezésre az unsigned char vonatkozna, mert az **if** azt várja el, emiatt aztán a „-1” értéket unsigned char -ként akarná beolvasni, annak viszont nem lehet előjele. Így megakad már a mínuszjelnél, annak ASCII kódját adja vissza. Ezt összehasonlítja a **#L@c** változó értékével, visszaadja az eredményt az **if**-nek, ami eltárolja ahova illik, majd menne tovább a parancsértelmezésben, de a következő karakter neki az „1”, mert csak az előtte álló mínuszjelet értékelte ki előzőleg, s mert az „1” karakterre nem definiáltunk még parancsot, így *syntax error* hibajelzéssel megáll.

Az is látható a fenti sorokból, hogy ha elértük a fájl végét, csak nagy lazán kilépünk a programból, nem tökölödünk olyasmivel, hogy a fájl lezárása. Az efféle rutinmelót előzékenyen elvégzi nekünk maga az interpreterünk: amint ugyanis kilépünk a programból, pontosabban, amint bármiért is de megsemmisülnek az inputfile típusú változóink, előbb még - ha volt hozzájuk rendelve még le nem zárt fájl - azt lezárja. Ezt természetesen azért teszi, mert az inputfile osztály destruktóra így lett megírva.

```
if #L (@c)==-2 T "\nOlyan fájlból próbáltál olvasni, ami nem is lett megnyitva!\n" XX
```

A fenti sor azt mutatja meg, hogy azt is tesztelhetjük, hogy nem épp egy olyan fájlból akarunk-e olvasni, ami meg se lett nyitva.

```
? @c; // és kiírjuk
```

A fenti sor amilyen rövid, olyan érdekes. Ugye, a beolvasás a **#L@c** változóba történt. Ide egy signed int került. Amennyiben azonban normális beolvasás történt, s nem valami hibajelzés, akkor ezen signed int érték valójában teljesen megegyezik egy közönséges unsigned char értékével - azzal a karakterrel, amit várunk a fájlból! S tekintve hogy elég bölcsek voltunk ahhoz, hogy a különböző single, azaz nem tömb változóinkat unionban helyezzük el, emiatt ezen **#L@c** signed int változónk első bájtja épp azonos a **#c@c** unsigned char változóval, de mert a „?” kiíró utasításunk amúgy is unsigned char értéket vár el, ezért ezen változónévből még a **#c** típusjelölőt se muszáj kiírunk, elég a változó neve!

```
} } // Ciklus vége
/;
"Most lezárjuk a fájlt\n"
[#B@b] // Itt zártuk le, ezzel az utasítással
```

Gondolom, a fenti utasítás szintaxisa nem szorul különösebb magyarázatra: egyszerűen a két szögletes zárójel közt meg kell adnunk az inputfile változót. Itt kötelező használni a **#B** típusjelölőt!

```
"Most megnyitjuk a fájlt újra\n"
#B@b="ido.cpp"; // Megnyitjuk a fájlt újra
[#B@b=6] // Elugrunk a 6-odik bájtjához
```

A fenti utasításnál is kötelezően használandó a **#B** típusjelölő! Az egyenlőségjel után megadandó pozíció mindig a fájl legelejétől számolódik, a fájl legelső karaktere természetesen a nulladikként van sorszámozva, hasonlóan mint a stringeknél.

```
{| 14 ; // beolvassuk innentől az első 14 karaktert
#L@c=#B@b;
? @c; // és kiírjuk
}| // Ciklus vége
/;
"Vége!\n"
XX
```

Ami az output fájlok kezelését illeti, azt is egy példaprogrammal magyarázom el. Az alábbi program beolvas egy input fájlból pár karaktert, s ezt kiírja egy output fájlba, majd visszaolvassa:

```
#s@b="proba.txt"; // Az input file neve
#s@o="mau_output_file.txt"; // Az output file neve
#B@b=@b; // Megnyitjuk az input fájlt
if #B@b T "Az input file sikeresen meg lett nyitva!\n"
E "Az input file nem megnyitható!\n" XX
#l@m=#B@b; // Az input file mérete
"Az input file mérete = " ?s #l@m; " bájt\n"
#K@o=@o; // Megnyitjuk az output file-ot.
if #K@o T "Az output file sikeresen meg lett nyitva!\n"
E "Az output file nem megnyitható!\n" XX
<s @o "Ezt a szöveget írom a fájl első sorába!\n";
{| 50 ; // beolvassuk az input file első 50 karakterét
#L@c=#B@b;
if #L (@c)==-1 T "\nItt az input file vége!\n" XX
if #L (@c)==-2 T "\nOlyan fájlból próbáltál olvasni, ami nem is lett megnyitva!\n" XX
<c @o @c; // és kiírjuk az imént beolvasott karaktert az output file-ba.
}| // Ciklus vége
<c @o 10; // Kiírunk egy sorvég-karaktert az output fileba
"Most lezárjuk az input fájlt\n"
[#B@b] // Itt zártuk le, ezzel az utasítással
<s @o "Ezt a sort az output file legvégére írom ki!\n";
"Lezárom az output fájlt!\n";
[#K@o]; // Lezárom az output file-ot ezzel az utasítással
"Most megnyitjuk az imént lezárt output fájlt újra, ezúttal olvasásra\n"
#B@b="mau_output_file.txt"; // Megnyitjuk a fájlt újra
{| #l#B@b // beolvasó ciklus indul, annyit olvasunk belőle ahány bájt a file mérete
#L@c=#B@b;
? @c; // és kiírjuk
}| // Ciklus vége
/;
"Vége!\n"
XX
```

Remélem, a sok komment miatt a program jórészt önmagát magyarázza; néhány kiegészítést fűznék csak hozzá:

```
<s @o "Ezt a szöveget írom a fájl első sorába!\n";
```

A fenti sor mutatja, hogy a „<” jel a fájlba kiíró utasításunk. Közvetlenül ezután kell álljon a típusjelölő. Ha ez az „s”, akkor stringet jelent. Természetesen nem csak konstans stringet adhatunk meg, hanem akármiféle stringváltozót is. Amennyiben a karakter nem „s” hanem „c”, akkor unsigned char értéket vár el, más típusjelölő karakterek esetén pedig más típusokat. Viszont a numerikus típusokat NEM STRINGKÉNT fogja kiírni, hanem úgy ahogy azok a memóriában le vannak tárolva, bájtonként! Például az unsigned short int típus decimális számok stringjeként akár 5 karaktert is elfoglalhat a képernyőn, mert lehet akár 65535 is egy ilyen szám értéke, mindazonáltal ha mi ezt a

```
<i @o 65535;
```

utasítással írjuk ki az „o” nevű outputfile-változónkba, akkor a fájlba csak 2 bájt kerül, s mindegyiknek az értéke 255 lesz.

```
[#K@o]; // Lezárom az output file-ot ezzel az utasítással
```

A fenti sor a fájl-lezárást mutatja be. Nem kell szórakoznunk a „C” nyelvben megismert „fflush” utasítással, azt magától is megcsinálja ilyen esetben, automatikusan.

```
{| #l#B@b // beolvasóciklus indul, annyit olvasunk belőle ahány bájt a file mérete
```

A fenti sor egy érdekes trükköt mutat be. Ugye, itt egy „fix számszor lefutó” ciklust alkalmazunk. A ciklus fejlécében szereplő számot, mely megmutatja hányszor kell lefusszon, az interpreter csak egy alkalommal olvassa be (illetve értékeli ki az ezt meghatározó aritmetikai kifejezést). Mi ide azt a számot óhajtjuk írni, ami a fájl mérete. Na most, ezt nekünk nem kell külön elmentenünk egy plusz változóba, ha amúgy nincs rá szükségünk, mert mi a csudának?! Egyedül arra kell ügyelnünk, hogy az inputfile-változónk az esetben adja vissza ezen méretet nekünk, ha unsigned int értékre castoljuk. Emiatt kell elé a #1 unáris casting operátor.

Itt mutatok be egy olyan utasítást s két függvényt, amiknek nincs közük az inputfile osztályunkhoz, de a fájlkezeléshez azért igen. Ugye sokszor jó lenne tudni, hány sor van egy fájlban, mert soronként akarjuk feldolgozni! Azt se ártana tudni esetleg, hány bájt hosszú a leghosszabb sor belőle. Nos, ehhez sajnos muszáj végigolvasni a fájlt. Azaz megnyitjuk, végigolvassuk, megszámloljuk amit kell, majd bezárjuk. Milyen jó is lenne ezt automatizálni... Íme!

```
int kisebbjel_kerdojel(F& f) { // Megszámolja, hogy egy fájlban hány sor van, s ezek közül
// visszaadja a leghosszabbnak a hosszát bájtokban számolva.
// Szintaxis:
// <? s
// ahol az s egy MAUSTRING kifejezés. A program ahhoz hogy ezen adatokat megtudja,
// természetesen kénytelen a fájlt megnyitni, végigolvasni, majd lezárni. A tartalmát nem módosítja.
// Ezen utasítás után a két adat a következő függvényekkel érhető el:
// unsigned int sorok száma = ?n;
// unsigned int leghosszabb sor hossza = ?!;
MAUSTRING filename;int kar;unsigned int i;int utolsoelem;
filename=ERTEKstring(f);f.inputfilesormaxhossz=0;f.inputfilesordb=0;i=0;
FILE *fp=fopen((char *)filename.s,"rb");
if(!fp) {L("File open error a \"<?\" utasításnál: A megadott \"%s\" állomány nem megnyitható (nem tudom beolvasni)!\\n"
"Megnyitási kísérlet helye: %lu",(char *)filename.s,f.P);EXITFAILURE();}
utolsoelem=0;
kisebbjel_kerdojel_ciklus:
kar=fgetc(fp);i++;
if(kar==SORVEG) {utolsoelem=SORVEG;f.inputfilesordb++;if(i>f.inputfilesormaxhossz) {f.inputfilesormaxhossz=i;}
i=0;} else {if(kar!=EOF) {utolsoelem=0;}}
if(kar!=EOF) goto kisebbjel_kerdojel_ciklus;
if(utolsoelem==0) f.inputfilesordb++;
fclose(fp);
return 0;
}
```

A használata bele van írva a kód elejébe, de egy kis példát is bemutatok, íme:

```

#!mau
<? "filesorokszama.mau";
"A file sorainak száma: " ?l ?n ; /;
"A file leghosszabb sora " ?l ?!; " bájttal hosszú.";
/;
XX

```

Ami a visszaadott eredményeket illeti, azok a **<?** utasítás következő kiadásáig nem módosulnak.

23. fejezet: Rendszerfüggvények

Rendszerfüggvény alatt olyan függvényeket értek, amiket nem a Felhasználó ír ma nyelven, hanem fixen bele vannak építve az interpreterbe. A dolog ugyanis úgy áll, hogy bár eddig még csak elboldogultunk valahogy a casting operátorok segítségével, de ez nem mehet örökké. Például, most hogy már vannak fájlkezelő műveleteink, jó lenne kibővíteni az input fájl kezelésének lehetőségét úgy, hogy legyen olyan utasításunk is, ami egy stringet olvas be a fájlból, tudniillik egészen addig olvas, míg egy sorvég-karaktert nem talál, de maximum egy bizonyos előre meghatározott darabszámú karaktert!

Na és hát ugye, e függvénynek minimum 2 paramétere kell legyen, amik eltérő típusúak: az input fájl változójának neve, és a megadott maximális darabszám! Ehhez olyan módszert kell megalkotnunk, mely képes eltérő típusú paramétereket kezelni.

E célra a jó öreg kérdőjel-karaktert vetjük be. Ez már ismerős nekünk, amikor fontos rendszerváltozók állapotát/értékét lehet lekérdeznünk vele. Annyit kell csak tennünk, hogy leteszteljük, közvetlenül utána a „#” jel áll-e. Ha ugyanis igen, az esetben ez egy rendszerfüggvény, melynek visszatérési értéke olyan típusú, amilyent a # utáni karakter határoz meg. Ekkor az interpreter egyszerűen meghívja az ezen típusú függvénykiértékelő rutint, ami elvéggez mindent. Többek közt, megnézi, egyáltalán miféle nevű függvényt kell meghívnia, mert ugye többféle is létezhet ami mind ugyanazon típusú visszatérési értéket szolgáltat. Na most a függvény neve legyen stringváltozóban megadva... Ez nagyon jó lesz, mert így még azt is variálhatjuk ha nagyon akarjuk, adott helyen a programban milyen függvényt hajtunk végre!

Íme a rutinok:

Az ERTEKstring függvényünket a következő résszel kell kibővíteni (zölddel a beszúrt sorok):

```

.....
case 's': A= vs(f);break;
} // switch vége
goto ERTEK_s_OPERATOR;} // változóbeolvasás vége

if((karakter=='?')&&(f.p[f.P+1]=='#')) {k(f);k(f);karakter=f.p[f.P];k(f);
switch(karakter) {
case 's': return kerdojelkereszt_s(f);break;

default: break;
} // switch vége
ERROR(f,3);
} // '?' teszt vége

```

```
switch(karakter) {
case '?': A= (MAUSTRING)systemvariable(f);goto ERTEK_s_OPERATOR; break; // Rendszerváltozó tartalmát kell visszaadni
case 194: A= (MAUSTRING)cinkeertekvisszaad(f);goto ERTEK_s_OPERATOR; break; // $ teszt vége
} // switch karakter vége
.....
```

Nyilván persze ahogy fejlődik a nyelvünk, több case-ágat is belevehetünk a zölddel jelzett switch részbe, ami mind egy-egy meghatározott visszatérési típust reprezentál.

A **kerdojelkereszt_s** függvény:

```
MAUSTRING kerdojelkereszt_s(F& f) {static const MAUSTRING semmi;
MAUSTRING fuggvenyneve;
fuggvenyneve=ERTEKstring(f);
if(fuggvenyneve.hossz==2) {
if((fuggvenyneve[0]=='N')&&(fuggvenyneve[1]=='L')) return inputfilebeolvas_NL(f);
} // hossz==2 vége

return semmi;
}
```

S végül az a függvény, mely megvalósítja a fejezet elején említett sorbeolvasást a fájlból:

```
MAUSTRING inputfilebeolvas_NL(F& f) {MAUSTRING ideiglenes;
USC inputfileindex;USI maxsorhossz;signed int be;
inputfileindex=ERTEKunsignedchar(f);
if(((unsigned char)f.B[inputfileindex])==0) {
L("Sort akartál beolvasni input fájlból, de a file még nincs megnyitva! Pozíció: %u",f.P);EXITFAILURE();}
maxsorhossz=ERTEKunsignedint(f);if(maxsorhossz==0) {
L("Sort akartál beolvasni input fájlból, de max. sorhosszméretnek 0 bájtot adtál meg! Pozíció: %u",f.P);EXITFAILURE();
}
ideiglenes[maxsorhossz]=0;ideiglenes.hossz=0;
while(ideiglenes.hossz<maxsorhossz) {
be=f.B[inputfileindex];
if(be==(-1)) return ideiglenes;
ideiglenes[ideiglenes.hossz]=(unsigned char)be;
ideiglenes.hossz++;
ideiglenes[ideiglenes.hossz]=0;
if(be==SORVEG) return ideiglenes;
}
return ideiglenes;
}
```

E funkció használatára példaprogram:

```
#B@b="ido.cpp"; // Megnyitjuk a fájlt
if(#B@b) T "A file sikeresen meg lett nyitva!\n"
E "A file nem megnyitható!\n" XX

#l@m=#B@b; // A file mérete
"A file mérete = " ?s #l@m; " bájt\n"

#i@i=0; // sorszámláló
{( // Ciklus indul
#s@b=?s "NL", b, 100; // Beolvassuk a sort, de max. 100 karaktert
if(!#s@b)==0 T [#B@b]; "Vége!\n"; XX; // Ha üres sort olvastunk be, vége a fájlnek, kilépünk
?i @i ". sor=> " ?s @b; "<===\n" ; // Kiírjuk a sort
#i++@i; // sorszámláló növelése
)} // Ciklus vége
```

Látható tehát, miként kell nálunk meghívni egy rendszerfüggvényt. Általános alakja:

?#x "neve" paraméterek

Ahol az „**x**” a visszatérési típust jelenti, a „**neve**” a függvény neve, a paraméterek száma és milyensége pedig attól függ, a függvény épp mit igényel.

Csináljunk is eképp egy nagyon hasznos függvényt, „**ST**” néven, ami név a „**S**ystem **T**ime” rövidítése, s az aktuális rendszeridőt adja vissza nekünk stringként:

```

MAUSTRING systime_ST(void) {using namespace std;MAUSTRING ID0;
static const char hetnapjai[4]={"V ", "H ", "K ", "Sze", "Cs ", "P ", "Szo"};
char idoformatum[]="%Y.%m.%d %H:%M:%S";
time_t *ido;time_t rawtime;struct tm ideiglenes;char buf[50];
time(&rawtime);ido=&rawtime;
ideiglenes = *localtime(ido);
if (!strftime(buf, sizeof(buf)-1, idoformatum, &ideiglenes)) {L("ERROR: strftime == 0");EXITFAILURE();}
buf[19]=SPACE;
buf[20]=hetnapjai[ideiglenes.tm_wday][0];buf[21]=hetnapjai[ideiglenes.tm_wday][1];buf[22]=hetnapjai[ideiglenes.tm_wday]
[2];
buf[23]=0;
ID0+=buf;
return ID0;
}

```

Ez majdnem teljesen megegyezik az „**mktimes**” függvénnyel, ami a logolásnál segít, de jó dolog külön megcsinálni, mert így könnyen módosíthatjuk, ha valami extrát akarunk beleépíteni, valamint mert ő nem azonos az mktimes-sel, ha valami baja van, azt már a log fájlba írhatja ki nekünk.

A rutint alkalmazó példaprogram:

```

#s@t=?#s "ST";
"Aktuális rendszerdátum és idő: "
?s @t; "<==\n"

```

Kimenete:

```
Aktuális rendszerdátum és idő: 2014.02.26 10:07:38 Sze<==
```

Íme 2 másik hasznos kis függvény, melyek 1 bájt értékét adják vissza hexadecimális formátumban (A bevezető „\$” karakter nélkül!):

```

MAUSTRING hex_x(F& f) {MAUSTRING hex;USC c;USC also;USC felso;
c=ERTEKunsignedchar(f);also=c & 0x0f;felso=c>>4;
//hex[2]=0;
if( also<10) hex[1]= also+'0'; else hex[1]= also+'a'-10;
if(felso<10) hex[0]=felso+'0'; else hex[0]=felso+'a'-10;
return hex;
}
// =====
MAUSTRING hex_X(F& f) {MAUSTRING hex;USC c;USC also;USC felso;
c=ERTEKunsignedchar(f);also=c & 0x0f;felso=c>>4;
//hex[2]=0;
if( also<10) hex[1]= also+'0'; else hex[1]= also+'A'-10;
if(felso<10) hex[0]=felso+'0'; else hex[0]=felso+'A'-10;
return hex;
}

```

Példaprogram:

```

#c@c=0;
{| 17
#s@c=?#s "$x" @c;
#s@C=?#s "$X" @c;
"dec " ?c @c " = $" ?s @c; " = $" ?s @C; "<==" /;
#c++@c;
|}

```

Eredménye:

```

dec 0 = $00 = $00<==
dec 1 = $01 = $01<==
dec 2 = $02 = $02<==
dec 3 = $03 = $03<==
dec 4 = $04 = $04<==
dec 5 = $05 = $05<==
dec 6 = $06 = $06<==
dec 7 = $07 = $07<==
dec 8 = $08 = $08<==
dec 9 = $09 = $09<==

```

```
dec 10 = $0a = $0A<==
dec 11 = $0b = $0B<==
dec 12 = $0c = $0C<==
dec 13 = $0d = $0D<==
dec 14 = $0e = $0E<==
dec 15 = $0f = $0F<==
dec 16 = $10 = $10<==
```

Mint látható, a 10 vagy annál nagyobb számértékek a „\$x” függvény esetén kisbetűkkel, a „\$X” függvénynél nagybetűkkel vannak visszaadva.

Ideje megoldanunk a billentyűzetről való beolvasást is! Ezt természetesen egy olyan függvénnyel oldjuk meg, ami unsigned char értéket szolgáltat nekünk. A neve az lesz, hogy „0”, azaz a nyitó és csukó kerek zárójelek egymás után, ez jó, mert utal arra hogy e beolvasás a getchar() függvénnyel történik, ami szintén e jelekkel végződik. Valamint, így senki nem köthet bele hogy miért „BE” a neve mondjuk és nem angolosan „IN”, vagy más akármi. Íme a függvény:

```
unsigned char karakterbeolvas(void) {return getchar();}
```

És egy példaprogram rá:

```
{(
#c@c=?#c "(");
? @c;
)}((@c)!=10);
/;
```

A fenti progi egyszerűen addig várja a karaktereket amíg Entert nem nyomunk, s mindet visszaírja a képernyőre.

Csináljunk egy másik rövid, de roppantul hasznos okosságot, ami visszaadja nekünk a fontos, operációsrendszer által meghatározott környezeti változók értékét! Íme:

```
MAUSTRING GetEnvironmentString(F& f) { // Visszaadja valamely fontos operációsrendszer-változó
// stringértékét
// Szintaxis: "ENV" x
// ahol az „x” egy stringváltozó, mely megmondja melyik oprendszer-stringet kell visszaadnia.
// Például: "ENV" "EDITOR"
MAUSTRING A;
A=getenv((const char *)ERTEKstring(f).s);
return A;
}
```

Használata:

```
#s@e=?#s "ENV" "EDITOR";
#s@s=?#s "ENV" "SHELL";
"Editor: " ?s @e; /;
"Shell : " ?s @s; /;
```

Eredménye nálam perpillanat:

```
Editor: mcedit
Shell : /bin/bash
```

Egy másik nagyon hasznos kis funkció, ha tudunk előre megadott hosszúságú stringet készíteni, olyat, amibe egy másik string balra van igazítva, s a fennmaradó helyek szóközökkel vannak feltöltve! Ezt ezzel a függvénnyel érhetjük el:

```
?#s "SB" string darab
```


ahol a „string” természetesen bármiféle stringkifejezés lehet, a „darab” pedig egy unsigned int típusú aritmetikai kifejezés. E függvénynél jó azonban tudnunk, ha a megadott string hosszabb mint „darab”, akkor NEM CSONKOLJA, hanem a függvény visszatérési értéke maga a „string” lesz!

Még egy hasznos függvény, mely stringként adja vissza a processz-ID értéket:

```
MASTRING pidstring(F& f) { // Visszaadja a processz id -nek megfelelő értékét stringként, a megfelelő hossza
// igazítva/csonkolva.
// Szintaxis:
// "PID" x c
// ahol az „x” egy unsigned char érték, a stringhosszat határozza meg,
// c pedig szintén egy unsigned char érték, az a karakter, amivel a string üres helyeit feltölti majd.
MASTRING A; unsigned char hossz; pid_t pid; char pidstring[60]; unsigned char c;
hossz=ERTEKunsignedchar(f); c=ERTEKunsignedchar(f);
if(hossz<1) {L("A pidstringnek 0 hosszat határozta meg! Nem valószínű, hogy ebbe beleférne... Hívási kísérlet helye:
%lu",f.P);
EXITFAILURE();}
// beolvassuk a process id értékét
if ((pid = getpid()) < 0) { L("Nem tudom beolvasni a pid-et! Hívási kísérlet helye: %lu",f.P);EXITFAILURE();}
sprintf(pidstring, "%d", pid); // stringgé konvertáljuk a pid értéket
A=pidstring;
if(hossz<A.hossz) {A[hossz]=0;A.hossz=hossz;}
if(hossz>A.hossz) {unsigned int regihossz;regihossz=A.hossz;A[hossz]=0;
register unsigned int i;for(i=regihossz;i<hossz;i++) A.s[i]=c;
}
A.hossz=hossz;A[hossz]=0;
return A;
}
```

Ez olyan esetekben lehet jó, amikor egy ideiglenes fájlt akarunk készíteni, s ekkor jó ötlet ha annak nevéhez hozzáfűzzük a pidstringet, hogy így megkülönböztethető legyen azoktól a hasonló nevű fájloktól, amiket esetleg ugyanezen program más példányai kreálnak ezen időtartam alatt ugyanabban a könyvtárban. Erre az esély ugyan kicsi, de ha mégis bekövetkezik, irtó nagy kavarodás lehet belőle... Továbbá ez is egy hasznos funkció:

```
#l@x=#s ?#s "SP" @h;
```

A fenti parancs egy konkrét példán mutatja be, miként lehet lecsípni egy string elejéről a felesleges dolgokat. E példában az „x” nevű unsigned int változó tartalma az a szám lesz, amit a „h” nevű string tárol. Igenám, de baj van ha a „h” string nem valami értékes számjeggyel kezdődik, hanem mondjuk szóközzel... ezesetben ugyanis az „x” változóba a 32 szám kerülne bele, mert az a szóköz kódja! Na most, emiatt nem adhatjuk értékül az x változónak csak úgy simán a „h” stringet, hogy:

```
#l@x=#s@h;
```

hanem előbb le kell csippantani a string bal oldaláról az összes whitespace karaktert. Na ezt végzi el az "SP" függvény.

24. fejezet: Tartalomjegyzék-kezelés

Ez egy meglehetősen szövevényes és bonyolult ügy. Arról van szó, hogy az Unix/Linux tartalomjegyzék-struktúrája nem épp az egyszerűségéről híres! Rengetegsok mindenféle adata, értéke, attribútuma van egy-egy fájlnak ami a tartalomjegyzékben szerepel, már amiatt is, mert egy tartalomjegyzék nemcsak fájlokat tartalmazhat, hanem mindenféle más izémizéket is, ami lényegében csak a nevében „fájl”, valójában azonban cseppet se, mert lehet például maga is tartalomjegyzék, vagy szimbolikus link (ami lehet „törött” vagy létező fájlra mutató), lehet közönséges fájl, lehet holmi „socket”, meg egy rakás egyéb mindenféle

bizbasz is még. Meg vannak mindenféle attributumok amik meghatározzák, hogy mit csinálhat vele a tulajdonos, a csoport vagy mindenki más, van az állományoknak méretük, meg mindenféle időbélyegeik, és... és még rengeteg minden.

Gusztustalan és undorító mindezt kezelni, főleg mert nehéz ennek a sokmindennek a megjelenítéséhez egy egyszerű, átlátható, konzisztens rendszert tervezni, amiben minden benne van amit épp látni akarunk, és ki is fér a képernyőre... Ez gyakorlatilag lehetetlen is, mert hiszen már egyetlen fájlnev is lehet akár 255 karakter hosszú, s hol van még az elérési útvonal?! És symlinknél azt is illő kijelezni, mire mutat.

Mindez kb teljességgel megoldhatatlan, s erre bizonyíték, hogy maga az „ls” parancs is többféle módon képes listázni, hogy az épp felmerülő igényekhez alakítsuk a megjelenítési formáját. Kár, hogy az „ls -l” formán kívül a többi kb semmire se jó, s még ez a forma is csak alig alkalmas valamire...

Rég felmerült bennem az ötlet, hogy írok egy saját listázórutint, aztán a dolog egyre csak halasztódott... De most amikor saját programnyelvet írok, már nincs mese: ezt is meg kell oldanom! Amiatt is, mert ha e rutint a magam „mau” nyelvén írom meg, azzal bebizonyítom a nyelvem használhatóságát is.

Nem írok ide minden kódot, csak megjegyzem, hogy akit részleteiben érdekel a megvalósítás, az a forráskódban az „**ADAT**” nevű osztály metódusainak nézzen utána. Ebben van egy tartalomjegyzék-bejegyzés minden adata, tehát a fájlnev, típus, jogosultságok, időadatok, stb. Ezen ADAT típusú bejegyzések egy-egy tömbben szerepelnek, típusonként különválogatva, azaz külön tömbben vannak az altartalomjegyzékeket tartalmazó ADAT értékek, külön tömbben a symlinkekre vonatkozó ADAT értékek, külön tömbben a közönséges fájlokra vonatkozó ADAT értékek, stb. Ezen tömbök együttesen alkotnak egy **MAPPA** nevű objektumot. És természetesen programnyelvünkben e MAPPA nevű adattípusból is 256 különböző lehetséges változónk van, amiket a **#T** típus jelöl. Amint egy ilyen változónak értéket adunk, amely egy tartalomjegyzék útvonala, ő azonnal beolvassa a megfelelő tömbökbe az összes, azon típusnak megfelelő bejegyzést, azaz az ADAT értékeket, sajnos azonban ahhoz hogy tudja melyikből hány van, előbb végigjárja a teljes tartalomjegyzéket. Ezután tudja melyikből mennyi kell, lefoglalja nekik a kellő memóriát, sőt egy picit többet a biztonság kedvéért, hátha épp most nyit a directoryban valami másik program egy fájl... Ezután újra beolvassa a tartalomjegyzék-struktúrát, de most már teljes részletességgel, s elhelyezi a megfelelő tömbökben az adatokat, azaz rögvest szétszortírozza őket típus szerint. Majd névsorba is rendezi őket. Ezek után már a mi dolgunk, a „mau” nyelven programozóké, hogy miként listázzuk ki e kutyulmányt, természetesen a különböző adatok lekérdezésére vannak mindenféle függvényeink.

Itt csak a legfontosabb rutint mutatom be, mely magának az egész mappának a beolvasását végzi, lehet rajta szörnyülködni:

```
void MAPPA::operator=(MAUSTRING mappaneve) {struct stat statisztika;struct stat linkstat;MAUSTRING erremutatasymlink;
DIR *dr;struct dirent *ent;char linke;MAUSTRING neve;unsigned long flagek;
char per[]="/";
int fd, r, f = 0;
unsigned int aktB,aktC,aktD,aktF,aktR,aktS,aktL,aktM;

if(D!=NULL) delete [] D;
```

```

if(R!=NULL) delete [] R;
if(B!=NULL) delete [] B;
if(C!=NULL) delete [] C;
if(F!=NULL) delete [] F;
if(S!=NULL) delete [] S;
if(LINK!=NULL) delete [] LINK;
if(MINDENEGYEB!=NULL) delete [] MINDENEGYEB;

maxD=maxR=maxB=maxC=maxF=maxL=maxM=maxS=0;mappaujneve=mappaneve;
LINK=B=C=D=F=R=S=MINDENEGYEB=NULL;

if(mappaneve.hossz==0) return;

// ***** Megszámoljuk, hogy melyik típusú bejegyzésből hány darab van
ent = 0;   dr = opendir((char *)mappaujneve.s);
if(dr==NULL) return;
do{
    ent = readdir(dr);
linke=0; // kezdetben a flaget beállítjuk arra, hogy "nem link".
    if(ent){
        if(strcmp(ent->d_name, ".") == 0) continue;
        if(strcmp(ent->d_name, "..") == 0) continue;
if(ent->d_type == DT_LNK) maxL++; else {
neve=mappaujneve;
if(neve[neve.hossz-1]!='/') neve+=per;
neve+=ent->d_name;
stat((char *)neve.s, &statisztika);
        switch (statisztika.st_mode & S_IFMT) { // megszámláljuk, melyik típusból hány van
            case S_IFBLK: maxB++; break;
            case S_IFCHR: maxC++; break;
            case S_IFDIR: maxD++; break;
            case S_IFIFO: maxF++; break;
            case S_IFREG: maxR++; break;
            case S_IFSOCK: maxS++; break;
            default: maxM++; break;
        } // switch vége
    } // else vége

}} while(ent);
closedir(dr);
// Memória foglалás az ADAT típusú tömböknek
if(maxL) LINK = new ADAT[maxL+10];
if(maxB) B = new ADAT[maxB+10];
if(maxC) C = new ADAT[maxC+10];
if(maxD) D = new ADAT[maxD+10];
if(maxF) F = new ADAT[maxF+10];
if(maxR) R = new ADAT[maxR+10];
if(maxS) S = new ADAT[maxS+10];
if(maxM) MINDENEGYEB = new ADAT[maxM+10];
// ***** Eddig tartott a számolás
aktB=aktC=aktD=aktF=aktR=aktS=aktL=aktM=0;
ent = 0;
dr = opendir((char *)mappaujneve.s);
if(dr==NULL) return;
do{
    ent = readdir(dr);
linke=0; // kezdetben a flaget beállítjuk arra, hogy "nem link".
    if(ent){
        if(strcmp(ent->d_name, ".") == 0) continue;
        if(strcmp(ent->d_name, "..") == 0) continue;
// statisztikai adatok beolvasása
        neve=mappaujneve;
        if(neve[neve.hossz-1]!='/') neve+=per;
        neve+=ent->d_name;
// ***** ITT DOLGOZZUK FEL A SYMLINKET
linke=0;if(ent->d_type == DT_LNK) {
linke=1; // Beállítjuk a flaget arra, hogy ez link

        if (lstat((char *)neve.s, &linkstat) != -1) { // Szimbolikus linkről (symlinkről) van szó !!!
erremutatasmlink[1024]=0;
        r = readlink((char *)neve.s, (char *)erremutatasmlink.s, 1024);
// if(r<0) {printf("Nem sikerült az %s symlink beolvasása!\n",ent->d_name);exit(1);}
        if(r<0) {linkstat.st_size=0;
linke=2; // törött symlink flag beállítása!!!
        }
if(r>1023) {L("Túl hosszú a neve annak a fájlnak, amire a %s symlink mutat!",ent->d_name);EXITFAILURE();}
erremutatasmlink.s[linkstat.st_size]=0;
erremutatasmlink.hossz=strlen((char *)erremutatasmlink.s);
        } // Symlink feldolgozásának vége
    }
// ***** EDDIG TART A SYMLINK FELDOLGOZÁSA
if (stat((char *)neve.s, &statisztika) == -1) {if(linke==1) linke=2; }
// ***** IMMUTABLE, APPEND bitek meg egyéb ext2-specifikus nyalánságok
beolvasása
fd = open ((char *)neve.s, OPEN_FLAGS);if (fd) {r = ioctl (fd, EXT2_IOC_GETFLAGS, &f);flagek = f;close (fd);} else f=0;
// ***** EDDIG TART az ext2-specifikus részek beolvasása

if(linke!=0) {LINK[aktL++](ent->d_name, statisztika,erremutatasmlink, linke, flagek);} // ha élő vagy törött symlink
else {
    switch (statisztika.st_mode & S_IFMT) { // betesszük a beolvasott tételt a megfelelő tömbbe

```

```

    case S_IFBLK: B[aktB++] (ent->d_name, statisztika,erremutatasymlink, linke, flagek); break;
    case S_IFCHR: C[aktC++] (ent->d_name, statisztika,erremutatasymlink, linke, flagek); break;
    case S_IFDIR: D[aktD++] (ent->d_name, statisztika,erremutatasymlink, linke, flagek); break;
    case S_IFIFO: F[aktF++] (ent->d_name, statisztika,erremutatasymlink, linke, flagek); break;
//    case S_IFLNK: LINK(adat); break;
    case S_IFREG: R[aktR++] (ent->d_name, statisztika,erremutatasymlink, linke, flagek); break;
    case S_IFSOCK: S[aktS++] (ent->d_name, statisztika,erremutatasymlink, linke, flagek); break;
    default: MINDENEGYEB[aktM++] (ent->d_name, statisztika,erremutatasymlink, linke, flagek); break;
} // switch vége
} // else vége
    } // if (ent) vége
} while(ent);
closedir(dr);
// Névsorbarendezés

if(aktB) qsort(B,aktB,sizeof(ADAT),ADATHasonlit);
if(aktC) qsort(C,aktC,sizeof(ADAT),ADATHasonlit);
if(aktD) qsort(D,aktD,sizeof(ADAT),ADATHasonlit);
if(aktF) qsort(F,aktF,sizeof(ADAT),ADATHasonlit);
if(aktR) qsort(R,aktR,sizeof(ADAT),ADATHasonlit);
if(aktS) qsort(S,aktS,sizeof(ADAT),ADATHasonlit);
if(aktL) qsort(LINK,aktL,sizeof(ADAT),ADATHasonlit);
if(aktM) qsort(MINDENEGYEB,aktM,sizeof(ADAT),ADATHasonlit);

}

```

Nem, nem hazudom azt, drága gyermekeim, hogy ez beleférne egy képernyőbe! Cseppet sem. Jó azonban tisztában lenni azzal, hogy ez már tulajdonképpen NEM a programnyelv része úgy „istenigazából”! Ez már bizony afféle „extra” nyaláncság, olyasmi, ami szinte minden „normális” programnyelvből hiányzik, úgy értem nincs benne „alapból”, hanem mindenféle külső függvényekkel oldják meg. És ez igazán egy rém bonyolult adattípus. Nem én találtam ki, hogy ilyen bonyolult legyen...

Rögvest mutatok egy kis mau programot a használatát demonstrálandó, ami egyelőre még nem tud túl sokat, de egy primitív kis listázórutinak azért már megfelel. Utána közlöm megint, ahol kell, soronként megmagyarázva a működését.

```

// Konstansok meghatározása
#c@c:=30 // A neveket ennyi hosszúságú stringbe igazítja balra
#c@c:=11 // A fájlméretet jobbra igazítva írja ki, ennyi hosszú mezőbe. Ha nem fér ki, csonkol...
#s@c=" --> " // Ez a string utal arra, melyik fájlra mutat a symlink

// Előkészítő műveletek
#c@c:=21-#c@c:1

"Az aktuális tartalomjegyzék listája:\n"
#T@t="."; // Az aktuális tartalomjegyzéket jelölő string

// A directoryk listázása
#l@i=0;
if (#l (?#l "T#" t D)==0) T »SDV; // Ha nincs altartalomjegyzék, nem listáz
"Tartalomjegyzékek száma: " ?l ?#l "T#" t D; /;
{ | ?#l "T#" t D;
?s ?#s "SB" (?#s "Tn" t, #l@i, D) #c@c:0; " <=" /;
#l++@i;
|}
SDV // Directoryk vége

// A közönséges fájlok listázása
#l@i=0;
if (#l (?#l "T#" t R)==0) T »SRV; // Ha nincs közönséges fájl, nem listáz
"Közönséges fájlok száma: " ?l ?#l "T#" t R; /;
{ | ?#l "T#" t R;
?s [#c@c:1,] ?#g "FM" t, #l@i, R;
" : "
?s ?#s "SB" (?#s "Tn" t, #l@i, R) #c@c:0; " <=" /;
#l++@i;
|}
SRV // Közönséges fájlok vége

// Szimbolikus linkek listázása
#l@i=0;
if (#l (?#l "T#" t L)==0) T »SLV; // Ha nincs symlink, nem listáz
"Linkek száma: " ?l ?#l "T#" t L; /;
{ | ?#l "T#" t L;
?s ?#s "SB" (?#s "Tn" t, #l@i, L) #c@c:0; ?s @c; ?s ?#s "LINK" t, #l@i; /;

```

```
#l++@i;
}|
$LV // Symlinkek vége

// Összesítés

"Összes bejegyzés száma: " ?l ?#l "T#" t m; /;
```

Outputja e pillanatban:

```
Az aktuális tartalomjegyzék listája:
Tartalomjegyzékek száma: 8
ABIW                                     <=
Brainfuck                               <=
mauprogramok                           <=
mau00                                   <=
mau01                                   <=
mau02                                   <=
mau03                                   <=
mau04                                   <=
Közönséges fájlok száma: 14
15911 : egyebek.cpp                     <=
93 : env.mau                           <=
55751 : hh                              <=
73996 : hh.cpp                          <=
736 : ido.cpp                           <=
431 : logol.cpp                         <=
528 : Makefile                          <=
196607 : main.cpp                       <=
1384 : mappa.mau                        <=
236424 : mau                             <=
163232 : mau.h                          <=
4702 : maumainfunctions.h               <=
13966 : rendez.c                        <=
245 : vz.h                              <=
Linkek száma: 3
brainfuck_h.mau                        --> /Releases/2014/U/Common/vz/MAU/Brainfuck/brainfuck_h.mau
brainfuck_kiirja_forditva.mau          --> /Releases/2014/U/Common/vz/MAU/Brainfuck/brainfuck_kiirja_fordita.mau
01_Colors_1988                         --> /Releases/2014/U/Common/vz/Ofra/01_Colors_1988
Összes bejegyzés száma: 25
```

A progi magyarázata:

```
// Konstansok meghatározása
#c@c:0=30 // A nevetek ennyi hosszúságú stringbe igazítja balra
#c@c:1=11 // A fájlméretet jobbra igazítva írja ki, ennyi hosszú mezőbe. Ha nem fér ki, csonkol...
#s@c=" --> " // Ez a string utal arra, melyik fájlra mutat a symlink
```

A fentiek gondolom világosak: nem írunk ki minden számot külön, s így hogy a fontos adatok a program elején vannak, egyhelyütt kell csak módosítanunk őket, ha meggondoljuk magunkat.

```
// Előkészítő műveletek
#c@c:1=21-#c@c:1
```

Itt kiszámítjuk, hányadik karaktertől kell majd csonkolnunk azt a stringet, amit az állományméretből gyárt nekünk a rutin. Az a méret ugyanis 21 karakter hosszú lehet maximum, de azért ugye többnyire nem olyan hosszú nálunk egy állomány, még bájtnban mérve sem... Illik viszont, hogy a méretet jelző számok helyiérték-helyesen kerüljenek egymás alá.

```
"Az aktuális tartalomjegyzék listája:\n"
#T@t="."; // Az aktuális tartalomjegyzéket jelölő string
```

A fenti sor rögvést meg is nyitja a tartalomjegyzéket, azaz beolvasásra kerül az egész trutymó a „t” nevű tartalomjegyzék-változóba.

```
// A directoryk listázása
```

```
#l@i=0;
```

Ez itt fent csak egy ciklusváltozó, ez számolja, épp hányadik tételt írjuk ki.

```
if (#l (?#l "T#" t D)==0) T »$DV; // Ha nincs altartalomjegyzék, nem listáz
```

Összehasonlító művelet. Nem illik hogy akkor is megpróbálkozzunk kiíratással, ha netán nincs mit kiírni, mert épp egyetlen subdirectory sincs a tartalom-

jegyzékben. Az hibát eredményezne. Na most, az **if** egy unsigned char értéket vár el, ezért külön jelezni kell a **#l** casting operátorral, hogy mi most unsigned int értékeket hasonlítunk majd össze. A belső zárójelen belüli rész adja meg, hogy épp hány dir típusú bejegyzés van a beolvasott tartalomjegyzékben. E kifejezés részletes jelentése:

?#l : unsigned int típusú rendszerfüggvényt hívunk meg

"T#" : A függvény neve. Arra utal, hogy tartalomjegyzék tételszáma után tudakozódunk.

t : A „t” nevű tartalomjegyzék-változót kérdezzük le.

D : A tartalomjegyzék-változónk bejegyzései közül a directorykra vagyunk kíváncsiak.

```
"Tartalomjegyzékek száma: " ?l ?#l "T#" t D; /;
```

Kiírjuk a darabszámot.

```
{| ?#l "T#" t D;
```

Ciklus indul, annyiszor fut le, amennyi a beolvasott darabszám, azaz ahány szubdirectory van a tartalomjegyzékben.

```
?s ?#s "SB" (?#s "Tn" t, #l@i, D) #c@c:0; " <=" /;
```

Kiírás. A **?#s "SB"** függvény ismerős kell legyen az előző fejezetből: ez hoz létre balra igazított, fix hosszúságú stringeket. Első paramétere a string, amit balra kell igazítani. Ez nálunk egy zárójeles kifejezés, és nem más mint egy **"Tn"** nevű, string típusú rendszerfüggvény, ami visszaadja a „t” tartalomjegyzék-változó directory-bejegyzései közül annak a nevét, aminek a sorszámát a ciklusváltozónk határozza meg. Az **"SB"** második paramétere pedig a mezőhossz, ezt a program elején beállított egyik konstans mondja meg neki.

```
#l++@i;
```

A ciklusváltozó inkrementálása.

```
|}
```

A ciklus vége.

```
$DV // Directoryk vége
```

Ide ugrik, ha nincs altartalomjegyzék-bejegyzés.

```
// A közönséges fájlok listázása
```

```
#l@i=0;
```

```
if (#l (?#l "T#" t R)==0) T »$RV; // Ha nincs közönséges fájl, nem listáz
```

```
"Közönséges fájlok száma: " ?l ?#l "T#" t R; /;
```

```
{| ?#l "T#" t R;
```

```
?s [#c@c:1,] ?#g "FM" t, #l@i, R;
```

Itt minden olyan mint a directoryk esetében, csak a fenti sor érdekes egy picit: Ez írja ki az állományméretet. Ezt a directoryknál és a symlinkeknél nem iratom ki, mi a fenének arra a pár bájtra... A közönséges fájlknál azonban ez fontos! Na most, e méretet egy unsigned long long típusú rendszerfüggvény adja meg, aminek **"FM"** a neve nálam, mert „**F**ile **M**éret”... Azt hogy közönséges fájl méretét kérdezzük le általa, az **"R"** betű jelenti, mert ugye „**R**egular”... Na most, hogyan is állítunk elő ebből stringet? Ugye, a **?s** parancs eleve stringet akar kiírni, tehát a sokféle kifejezésértékelő függvényünk közül azt hívja meg, ami stringet akar készíteni mindenből, amiből csak tud. Ez belefut abba a kutylományba, ami a szögletes zárójelek közt található. Ebből tudja, hogy neki nem az egész stringet kell visszaadnia amit majd beolvas, hanem annak csak egy részét. Első paramétere egy változó, azt mondja meg, hányadik karaktertől. Ezt megjegyzi. Aztán jön a vessző, majd semmi, ebből tudja hogy innen a hátralevő teljes hosszat kell vennie. Oké. Ezután beolvassa ami hátra van. Igen ám, de ezután nem string-

kifejezés jön, hanem egy unsigned long long típusú szám! Nem jön zavarba a drága, rögvest tudja hogy ez maximum 21 karakter hosszú decimális számsorozat lehet, azonnal átalakítja arra. Majd az eredményül kapott stringet csonkolja úgy, ahogy a szögletes zárójelek közt előírtuk neki.

```
" : "  
?s ?#s "SB" (?#s "Tn" t, #l@i, R) #c@c:0; " <=" /;
```

A fenti sor is kiírás. A végén a <= nyíl kizárólag amiatt irattatik ki most, hogy lássuk, tényleg minden érték fix hosszúra lett igazítva.

```
#l++@i;
```

Ciklusváltozó inkrementálása.

```
}
```

Ciklus vége.

```
$RV // Közöséges fájlok vége
```

```
// Szimbolikus linkek listázása
```

```
#l@i=0;
```

```
if (#l (?#l "T#" t L)==0) T »$LV; // Ha nincs symlink, nem listáz
```

Mint látjuk, szimbolikus linkek darabszáma után tudakozandó, a "T#" függvényt az „L” paraméterrel kell meghívunk.

```
"Linkek száma: " ?l ?#l "T#" t L; /;
```

```
{| ?#l "T#" t L;
```

```
?s ?#s "SB" (?#s "Tn" t, #l@i, L) #c@c:0; ?s @c; ?s ?#s "LINK" t, #l@i; /;
```

A fenti kiírásnak csak a vége az érdekes, a többi már rutinművelet. Itt a "LINK" nevű string-típusú rendszerfüggvényt hívtuk meg, ami a „t” nevű tartalomjegyzék-változónk link-típusú bejegyzései közül a megadott sorszámúnál levő azon stringet adja vissza, mely meghatározza, melyik állományra mutat az adott symlink.

```
#l++@i;
```

```
}
```

```
$LV // Symlinkek vége
```

```
// Összesítés
```

```
"Összes bejegyzés száma: " ?l ?#l "T#" t m; /;
```

Ezen utolsó sorról azt érdemes tudnunk, hogy akkor adja vissza a "T#" függvényünk az adott tartalomjegyzék-változó összes bejegyzésének a számát, ha olyan karaktert adunk neki típusként, ami nem érvényes típus. Itt most az „m” karakterre esett a választásom, de lehetett volna sok más akármi is. Az érvényes karakterek, amik külön típusokként vannak kezelve nálam:

D : altartalomjegyzék (subdirectory)

L : szimbolikus link

R : közöséges (reguláris) fájl

B : blokk-device

C : karakteres eszköz

S : socket

F : fifo

M : Minden egyéb

Ha már kiírkálunk mindenfélét a képernyőre, illik, hogy efféle bonyolult listákat színesben tudjunk megjeleníteni! Ennek érdekében vegyük fel a **vz.h** fájlba e sorokat:

```

#define COLOR_RED      "\x1b[31m"
#define COLOR_GREEN    "\x1b[32m"
#define COLOR_YELLOW   "\x1b[33m"
#define COLOR_BLUE     "\x1b[34m"
#define COLOR_MAGENTA  "\x1b[35m"
#define COLOR_CYAN     "\x1b[36m"
#define COLOR_WHITE    "\x1b[37m"
#define COLOR_BRIGHT   "\x1b[01m"
#define COLOR_UNDERLINE "\x1b[04m"
#define COLOR_RESET    "\x1b[0m"

```

Majd írjuk be **fuggveny_S_83** tömb b,c,d,g,m,r,u,v,w,y betűihez azt, hogy „színek”, és készítsük is el e függvényt eképp:

```

int színek(F& f) {
switch(f.a2) {
case 'd' : printf(COLOR_RESET); break; // default színek visszaállítása
case 'v' : printf(COLOR_BRIGHT); break; // világosság
case 'u' : printf(COLOR_UNDERLINE); break; // aláhúzás
case 'r' : printf(COLOR_RED); break;
case 'g' : printf(COLOR_GREEN); break;
case 'y' : printf(COLOR_YELLOW); break;
case 'b' : printf(COLOR_BLUE); break;
case 'm' : printf(COLOR_MAGENTA); break;
case 'c' : printf(COLOR_CYAN); break;
case 'w' : printf(COLOR_WHITE); break;
default: SERROR(f,3); break;
} // switch vége
return 0;
}

```

Ezek után, ha egy mau programban színesen akarunk kiírni valamit, a kiírás előtt csak adjuk ki az **Sr** parancsot hogy vörössel írjunk, az **Sg** parancsot hogy zölddel írjunk, stb. Az **S** utáni lehetséges karakterek listája kiolvasható a fenti programból. Az **Sv** parancsra a szöveg világosabb színben jelenik meg, az **Sd** parancsra visszaáll az alapértelmezett szín.

Annak érdekében, hogy parancssori paramétereket is tudjunk átadni a mau programunknak, csináljunk egy

```

?#s "ARGV" x

```

nevű függvényt is, aholis az „x” egy unsigned char érték, s azt mondja meg, az „argv”-ben levő parancssori paraméterek hányadik értékét adja vissza stringként. Ennek megfelelően ha a listázó programunk neve mondjuk **mappa.mau**, akkor ezt így hívhatjuk meg egy könyvtárra, mondjuk a \$HOME könyvtárra:

```

mau mappa.mau $HOME

```

és ennek érdekében a lekérdezendő könyvtárat meghatározó sor a mau programunkban így kell kinézzen:

```

#T@t= ?#s "ARGV" 2; // A lekérdezendő tartalomjegyzéket jelölő string

```

Csináljunk lekérdezést arra is, hogy a symlink törött-e! Ezt efféleképp használhatjuk:

```

if((?#c "L?" t, #l@i, L)==2) T Sc; Sv; ?s @C; Sd; Sm; Sv;
E Sc; ?s @C; Sd; Sm;

```

Azaz mint látható, a **?#c "L?"** függvényvel lekérdezhettük valaminek a link-állapotát. A visszatérési érték 0 ha nem link, 1 ha élő link és 2 ha törött link.

Jó azonban, ha a színekkel való kiírást nem csak fix színekkel tudjuk meghatározni, hanem beleépítve a színeket a stringekbe is! Ennek érdekében készítsük el e függvényt:

```

MAUSTRING szinstring(F& f) { // Visszaad egy szinstringet.
// Formátum: ?#s "SZIN" k
// ahol a „k” egy unsigned char érték, ami meghatározza a szinstringet.
unsigned char k;MAUSTRING A;

char d[]=COLOR_RESET; char v[]=COLOR_BRIGHT;char u[]=COLOR_UNDERLINE;char r[]=COLOR_RED; char g[]=COLOR_GREEN;
char y[]=COLOR_YELLOW;char b[]=COLOR_BLUE; char m[]=COLOR_MAGENTA; char c[]=COLOR_CYAN;char w[]=COLOR_WHITE;

k=ERTEKunsignedchar(f);switch(k) {
case 'd' : A=d; break; // default szinek visszaállítása
case 'v' : A=v; break; // villogás
case 'u' : A=u; break; // aláhúzás
case 'r' : A=r; break;
case 'g' : A=g; break;
case 'y' : A=y; break;
case 'b' : A=b; break;
case 'm' : A=m; break;
case 'c' : A=c; break;
case 'w' : A=w; break;
default: SERROR(f,3); break;
} // switch vége
return A;
}

```

Használata bele van írva a rutin elejére.

Csináljunk függvényt arra is, hogy 4 karakteres oktális stringként visszaadja a fájl jogait! Igaz „Kocka” ugyanis kizárólag oktálisán szereti látni a jogosultságokat, nem afféle helypocsékoló kezdő dedósoknak való übergagyi stílusban, hogy „rwxrwxrwx”... Íme a rutin:

```

MAUSTRING JOGOSULTSAG(F& f) { // Visszaadja a tartalomjegyzékben levő valahányadik tétel jogosultságainak
// stringjét, oktálissan!
// Szintaxis: "JOG" index hanyadik x
// ahol „index” a tartalomjegyzék-változó indexe,
// „x” pedig azon karakterek valamelyike lehet, ami itt a switchben látható.
USC tipus;USC index;unsigned int hanyadik;MAUSTRING A;char ismeretlen[]="???";
char filejogai[]="0000";jogai J; speci S;jogok mode;
index=ERTEKunsignedchar(f);hanyadik=ERTEKunsignedint(f);tipus=ERTEKunsignedchar(f);
switch(tipus) {
case 'D': mode.mod=f.T[index].D[hanyadik].MODE();break;
case 'R': mode.mod=f.T[index].R[hanyadik].MODE();break;
case 'B': mode.mod=f.T[index].B[hanyadik].MODE();break;
case 'C': mode.mod=f.T[index].C[hanyadik].MODE();break;
case 'F': mode.mod=f.T[index].F[hanyadik].MODE();break;
case 'L': mode.mod=f.T[index].LINK[hanyadik].MODE();break;
case 'S': mode.mod=f.T[index].S[hanyadik].MODE();break;
case 'M': mode.mod=f.T[index].MINDENEGYEB[hanyadik].MODE();break;
default: A=ismeretlen;return A;break;
} // switch vége

S.byte=0;S.s.sticky=mode.a.sticky;S.s.groupid=mode.a.groupid;S.s.userid=mode.a.userid;filejogai[0]=S.byte+'0';
J.byte=0;J.j.x=mode.a.tx;J.j.w=mode.a.tw;J.j.r=mode.a.tr;filejogai[1]=J.byte+'0';
J.byte=0;J.j.x=mode.a.gx;J.j.w=mode.a.gw;J.j.r=mode.a.gr;filejogai[2]=J.byte+'0';
J.byte=0;J.j.x=mode.a.mx;J.j.w=mode.a.mw;J.j.r=mode.a.mr;filejogai[3]=J.byte+'0';
A=filejogai;
return A;
}

```

Kell rutin arra is, hogy lecsekkoljuk, végrehajtható-e a fájl! Ez ugye korántsem mindegy... Ez természetesen egy unsigned char értéket ad vissza, ami végrehajtható fájlok esetében 1, különben nulla. A rutin:

```

unsigned char Executable(F& f) { // Visszaad 1-et ha a tartalomjegyzékben levő valahányadik tétel
// egy végrehajtható fájl, különben nullát ad vissza.
// Szintaxis: "EXE" index hanyadik x
// ahol „index” a tartalomjegyzék-változó indexe,
// „x” pedig azon karakterek valamelyike lehet, ami itt a switchben látható.
USC tipus;USC index;unsigned int hanyadik;
jogok mode;
index=ERTEKunsignedchar(f);hanyadik=ERTEKunsignedint(f);tipus=ERTEKunsignedchar(f);
switch(tipus) {
case 'D': mode.mod=f.T[index].D[hanyadik].MODE();break;
case 'R': mode.mod=f.T[index].R[hanyadik].MODE();break;
case 'B': mode.mod=f.T[index].B[hanyadik].MODE();break;
case 'C': mode.mod=f.T[index].C[hanyadik].MODE();break;
case 'F': mode.mod=f.T[index].F[hanyadik].MODE();break;
case 'L': mode.mod=f.T[index].LINK[hanyadik].MODE();break;
case 'S': mode.mod=f.T[index].S[hanyadik].MODE();break;

```



```

case 'M': mode.mod=f.T[index].MINDENEGYEB[hanyadik].MODE();break;
default: SERROR(f,3);break;
} // switch vége
if(mode.a.tx|mode.a.gx|mode.a.mx) return 1; else return 0;
}

```

Végül pedig kell rutin mely visszaadja azon csoport nevét, aminek a tulajdonában van a fájl. Ezt nem írom ide ki, teljesen ugyanúgy néz ki mint a tulajdonos nevét visszaadó, a szintaxisa:

```

?#s "Tg" index hanyadik x

```

ahol „index” a tartalomjegyzék-változó indexe, „x” pedig azon karakterek valamelyike lehet, ami a tulajdonos stringjének meghatározásakor is használható.

Ezek után ideje, hogy e sok újabb rutin működését bemutassuk előbbi állomány-listázó programunk egy fejlettebb változatában:

```

// Konstansok meghatározása

```

```

if(?a)<3 T #s@A="."; // Ha nem adtunk meg parancssori paramétert, az aktuális könyvtárat listázza
E #s@A=?#s "ARGV" 2; // különben a megadott tartalomjegyzéket

```

```

#c@c:0=30 // A neveket ennyi hosszúságú stringbe igazítja balra
#c@c:1=11 // A fájl méretet jobbra igazítva írja ki, ennyi hosszú mezőbe. Ha nem fér ki, csonkol...
#c@c:2=10 // A csoport és tulajdonos nevét ennyi karakter hosszúságú mezőbe balra igazítva írja ki
#s@c=" ==> " // Ez a string utal arra, melyik fájlra mutat a symlink
#s@c=" --> " // Ez a string utal arra, melyik fájlra mutat a symlink, az esetben, ha a link „törött”
#s@K=(?#s "SZIN" c)+";"; // Ez határozza meg a csoportstringet és tulajstringet elválasztó karaktert
#s@G=?#s "SZIN" b; // Ez határozza meg a csoportstring és tulajdonos-string színét
#s@o=?#s "SZIN" g; // Az oktális jogosultságok színe
#s@m=(?#s "SZIN" g)+(?#s "SZIN" v); // állomány méret színe
#s@R=(?#s "SZIN" y); // A közönséges állományok színe
#s@x=(#s@R)+(?#s "SZIN" v)+(?#s "SZIN" u); // végrehajtható állományok színe
#s@l=(?#s "SZIN" m); // A symlinkek színe
#s@L=(#s@l)+(?#s "SZIN" v); // Törött symlinkek színe

```

```

// Előkészítő műveletek

```

```

#c@c:1=21-#c@c:1
#c@c:2=10+#c@c:2 // Ez miatt kell, mert a karakterszám méretet meg kell növelni 2*5 bájtal, a kettőspont elé és mögé
// beszűrt szístringek bájt mérete miatt.

```

```

"Tartalomjegyzék:\n"

```

```

#T@t=@A; // A lekérdezendő tartalomjegyzéket jelölő string. Itt olvassuk be a directoryt.

```

```

// A directoryk listázása

```

```

#l@i=0;
if (#l (?#l "T#" t D)==0) T »SDV; // Ha nincs altartalomjegyzék, nem listáz
Sw; // Directory színek fehér
"Altartalomjegyzékek száma: " ?l ?#l "T#" t D; /;
{ | ?#l "T#" t D;
Sw; // Directory színek fehér
?s ?#s "SB" (?#s "Tn" t, #l@i, D) #c@c:0; // A dir neve
?s @o; " \" ?s ?#s "JOG" t, #l@i, D; "\" " // Az altartalomjegyzék jogai
?s (#s@G) + ?#s "SB" ((?#s "Tg" t, #l@i, D) + (#s@K) + (#s@G) + (?#s "Tt" t, #l@i, D)) #c@c:2; // A dir csoportjának
és /tulajának neve
" "
/;
#l++@i;
|}
Sd; // Directory szín vége
SDV // Directoryk vége

```

```

// A közönséges fájlok listázása

```

```

#l@i=0;
if (#l (?#l "T#" t R)==0) T »SRV; // Ha nincs közönséges fájl, nem listáz
"Közönséges fájlok száma: " ?l ?#l "T#" t R; /;
{ | ?#l "T#" t R;
Sd;
?s @m; // file méret szín beállítása
?s [#c@c:1,] ?#g "FM" t, #l@i, R; // A file méret
Sd; " "
if(?#c "EXE" t, #l@i, R) T ?s @x; // Ha végrehajtható állomány, ez lesz a színe
E ?s @R; // Ha nem végrehajtható, akkor pedig ez
?s ?#s "SB" (?#s "Tn" t, #l@i, R) #c@c:0; // A file neve
Sd;
?s @o; " \" ?s ?#s "JOG" t, #l@i, R; "\" " // A file jogai
?s (#s@G) + ?#s "SB" ((?#s "Tg" t, #l@i, R) + (#s@K) + (#s@G) + (?#s "Tt" t, #l@i, R)) #c@c:2; // A file csoportjának

```

```

// és tulajának neve
" "
/;
#l++@i;
}
$RV // Közöséges fájlok vége

// Szimbolikus linkek listázása

Sd; #l@i=0;
if (#l (?#l "T#" t L)==0) T »$LV; // Ha nincs symlink, nem listáz
?s @l; // A symlinkek színe
"Linkek száma: " ?l ?#l "T#" t L; /;
{ | ?#l "T#" t L;
Sd; ?s @l; // A symlinkek színe
?s ?#s "SB" (?#s "Tn" t, #l@i, L) #c@c:0; Sd;
if((?#c "L?" t, #l@i, L)==2) T Sd; ?s @L; ?s @C;
E Sd; ?s @l; ?s @c;

?s ?#s "LINK" t, #l@i; /;
#l++@i;
}
Sd;
$LV // Symlinkek vége

// Összesítés

"Összes bejegyzés száma: " ?l ?#l "T#" t m; /;

```

Használata, ha a proginak a „mappa.mau” nevet adtuk:

```
mau mappa.mau listázandó_tartalomjegyzék
```

Ez már aztán igazán pöpec módon listáz... És a legszebb az egészben az, hogy a listázási formátum és a színek nincsenek fixen behuzalozva valami gépi kódra fordított progiba mint az „ls” esetében, hanem icipici tudás birtokában e rutint bárki kedvére alakíthatja!

Na most, ha már ilyen viszonylag komoly programokat is képesek vagyunk írni, azaz olyasmit aminek van gyakorlati haszna is a számunkra, akkor könnyítsük meg a dolgunkat azzal, hogy írunk egy roppant komplex és bonyolult olyan rutint is a nyelvünkbe, ami a **#!** névre „hallgat”, és egyszerűen annyit csinál, hogy mint-ha a pontosvessző utasításunk volna, elugrik a következő whitespace karakterig! Ennek érdekében a **fuggvény_kereszt** nevű, már készen levő rutinunkat kell módosítani, ami a módosítás után így néz ki:

```

int fuggvény_kereszt(F& f) {if(f.p[f.P]=='!') {pontosvesszo(f); return 0;}
else {return ertekadas(f,'=');}}
}

```

Na most hogy mi értelme van ennek... Messze több, mint amit e módosítás rövidege sejtetni enged! Ugyanis ezek után írhatunk ilyesmit a mau nyelvű programjaink legelső sorába:

```
#!mau
```

Ezt maga a mau interpreter figyelmen kívül hagyja. Ő igen, de nem ám a shell, például a bash... Azaz, ezek után ha a mau interpreter végrehajtható állományának logikusan a „mau” nevet adtuk, s ezt olyan helyre pakoltuk a rendszerben, mely szerepel a \$PATH rendszerváltozó útvonalai közt, akkor csak annyi a dolgunk, hogy futtatási jogot adjunk a mau nyelven írt programunknak, mondjuk a

```
chmod 755 mappa.mau
```

paranccsal, s ezek után már nem muszáj őt úgy indítanunk hogy

```
mau mappa.mau
```

hanem elég annyi is, hogy

```
./mappa.mau
```

Ugyanis a shell ilyenkor kiolvassa a fájl első sorát, s az ott levő **#!** karakterpáros után következő nevű programot próbálja meghívni, s ha megtalálta, átadja neki paraméterként az állomány nevét (meg a név után megadott esetleges egyéb parancssori paramétereket).

Hm, kezdünk igazán „nagyok” lenni... Már tökre hasonlítunk tényleg egy igazi, használható szkriptnyelvhez... Egészen jól „beépülünk” az oprendszerbe...

25. fejezet: Mau nyelvű függvények hívása

A „mau nyelvű függvény” fogalma alatt sok mindent értünk. Nagy vonalakban ez valamiféle olyan program vagy programrészlet, amit szeretett mau program-nyelvünkben írtunk meg, s ezt meg akarja hívni valamely másik program, hogy hasznos munkát végezzen neki. Na most, ennek azért van ám egy rakás variációja, s azoknak a esetetei... Mert kezdjük azzal, hogy az se teljesen világos, mármint nincs rá elfogadott terminológia, mi is az, hogy „függvény”!

Sokan ugyanis a szubrutinokat is függvényeknek tartják. Ilyesmiket mi már tudunk gyártani a meglevő eszközeinkkel, s ezzel simán elértük a régi C-64 -es számítógép BASIC-jének színvonalát és használhatóságát. Ha ehhez hozzávesszük, hogy a szubrutinon belül képesek vagyunk külön névteret generálni, aminek paramétereket is átadhatunk, sőt e névterek tetszőleges mélységben egymásba is ágyazhatóak - nos, akkor meg bőven túl is haladtuk az említett nyelv színvonalát már! Általában ugyanis azért úgy vélekednek a programozók közt a legtöbben, hogy csak az függvény, ami saját névtérrel rendelkezik. Igen ám, de a névtér fogalmába ők bele szokták érteni a függvényen belüli címkéket is, amik nálunk viszont globálisak! Továbbá, igazi függvényeknek azért szokott ám neve is lenni, ami nálunk eddig még nincs. Címkéink vannak, függvényneveink még nincsenek.

Mindez még tovább bonyolódik azzal, hogy egy „függvény”, az legalább 2 nagyon különböző valami lehet: mert lehet egyrészt ugyanabban a fájlban amiben a fő-program, vagy, általánosabb megfogalmazásban, az a program ami őt a függvényt meg óhajtja hívni, s az is lehet - legalábbis illik hogy megoldjuk ennek lehetőségét - hogy a hívott függvény egy teljesen külön forrásprogram. Utóbbi esetre nem jó megoldás az hogy a már rendelkezésünkre álló **SY** paranccsal a shellen keresztül hívjuk meg, mert úgy csak stringeket adhatnánk át neki paraméterül, visszatérési értéként meg jószerivel semmit se fogadhatnánk tőle, ha csak fájlba nem menti az eredményeket. Erről korábban már elmélkedtem.

És természetesen a meghívott függvénynek is tudnia kell hívni más függvényeket, olyanokat amik vele egy fájlban vannak, s olyanokat is amik külön fájlban vannak. Szóval az ügy cseppet se egyszerű, s itt bizony rém sokáig lustálkodtam! Cseppet se magától értetődő, miféle szintaxist találjunk ki ennek a dolognak, különösképpen, mert azt is szerettem volna megoldani, hogy akárhány vissza-térési értéke is lehessen a függvényeimnek!

Nos, abból indultam ki, amit már e könyv legelején is említettem: nálam egy mau nyelvű program egy objektum, azaz, ha meg akar hívni egy másik mau nyelvű programot, egyszerűen kreál egy ilyen új objektumot, megadja neki az állomány címét, az betölti, s ezután meghívja annak futtatási módszerét és kész! Egyszerű.

Igen, EZ egyszerű, a nagy gond az, hogy mi legyen azon függvényekkel, amik a már épp futó fájlban vannak, vele együtt töltődtek be a tárba! Itt bizony nem volt más megoldás, mint hogy figyelni kell betöltés közben, épp egy új függvény fejléce kezdődik-e, s ekkor „röptében”, betöltés, azaz az input fájl olvasása közben kreálni egy új PGM objektumot, s azontúl abba rakosgatni be a címkék címét. Ráadásul ugyebár, ezen új PGM objektum kódszegmense ott illik kezdődjön ahol ő maga a függvény, miközben pedig a program kódja valójában nem szabad hogy megduplázódjon emiatt, mert az iszonyatos memóriapocséklás lenne. Ezekből a neki letárolt címkecímek abszolút értékéből mindig le kell vonni azt az értéket ami e függvény kezdete a programkódban. A függvény továbbá kell ismerje a saját nevét, valamint a szülő program névteréből azt az infót, hogy hol lelheti fel a vele együtt betöltött többi PGM objektumot, mert hátha ő maga is meg akarna hívni más, vele egyszerre betöltött függvényeket!

Mindez cseppet se egyszerű. Ennek érdekében sajnos jócskán ki kellett bővítenem a beolvasórutint. Igazából persze a dolog nem volt olyan nagyon szörnyű amikor már rájöttem, mit kell csinálnom, az volt a „nem egyszerű”, amíg rájöttem a teendőimre... Íme az új rutin:

```
void PGM::beolvas(char *filename) { // beolvassa a fájlt a memóriaterületre
unsigned int fuggvenynevpozicio_kezd;
unsigned int fuggvenynevpozicio_veg;
unsigned int fuggvenynev_hossz;
char fuggvenynevetomb[256];
USC voltenyitoidezojel;
PGM *uj;
F *aktF; // Az aktuális F struktúra, amibe be kell olvasni a címkéket majd.
unsigned int levon; // Ennyit kell mindig levonnia a címke abszolút értékéből, ahhoz, hogy a helyes
// számot tegye be a címketömbbe

USC sorvegflag; // Ha 0: nem sorvég után vagyunk. Ha 1: sorvég után vagyunk. Ha 194: a $ jel első bájta után vagyunk.
// Ha 167: a $ jel második bájta után vagyunk. Ha 11: A címke első karaktere után vagyunk
USC cimkelarakter;USC kar; sorvegflag=1; // Az elején 1-re állítjuk hogy a program legelső sorában is lehessen címkét
megadni.

levon=0; // Kezdetben a címkék a programfájl legelejétől számíthatók.
aktF=&f; // Kezdetben a címkéket a szülőnévtér struktúrájába pakolja
voltenyitoidezojel=0;
f.programfile neve=filename; f.programfile nevehossz=strlen(f.programfile neve);
aktF->cimkedb=0; // Lenullázzuk a címkeszámálót
f.programfilehossz=get_file_size(f.programfile neve);
if(f.phossz<f.programfilehossz) {L("A lefoglalt memóriaterület (%lu byte) kisebb mint a beolvasandó programfile mérete "
"(Fájlnév: %s, = %lu byte)!",f.phossz,filename,f.programfilehossz);EXITFAILURE();}
FILE *fp=fopen(filename,"rb");
if(!fp) {L("A megadott \"%s\" állomány nem megnyitható (nem tudom beolvasni)",filename);EXITFAILURE();}
register USIL i;for(i=0L;i<f.programfilehossz;i++) {
kar=f.p[i]=(USC)fgetc(fp); // Mindenképpen eltároljuk a beolvasott karaktert
if(kar==SORVEG) {sorvegflag=1;continue;}

// *****
// Itt kezdődik a programfájl szétDarabolása különálló függvényekre

if(kar==226) {if(sorvegflag==1) sorvegflag=226; else sorvegflag=0;continue;} // A „ jel első bájta, hexa $E2
if(kar==128) { // „ jel második bájta, hexa $80
if(sorvegflag==226) {sorvegflag=128;} else {sorvegflag=0;} // if 226 vége
continue;
} // if 128 vége
if(kar==158) { // „ jel harmadik bájta, hexa $9E
if(sorvegflag==128) { // Most már biztos hogy nyitóidézőjelet találtunk, azaz
// itt egy új függvény kezdődik

// -----
// ===== Itt jön a függvénynév kezdetének meghatározása!
fuggvenynevpozicio_kezd=i+1; // Az aktuális karakter pozíciójánál eggyel nagyobb pozíción fog kezdődni a függvény
nevének
// első karaktere; ezt eltároljuk mert kellene fog nekünk.
```

```

voltenyitoidezojel=1;
// =====
// -----
                } // if 128 vége
                else {sorvegflag=0;}
continue;
} // if 158 vége

if(kar==157) { // " jel harmadik bájtja, hexa $9D
// Ha ezt a bájtot olvastuk be, elképzelhető, hogy ez a csukóidézőjel 3 bájtos UTF-8 kódjának
// utolsó bájtja. Ekkor mindenekeelőtt meg kell vizsgálni, volt-e korábban új függvénybe belépés.
if(voltenyitoidezojel==0) {continue;} // Nem volt nyitóidézőjel, semmit se kell csinálni.
// Ha ideértünk, akkor VOLT nyitóidézőjel.
// Most meg kell vizsgálnunk, e $9D bájt tényleg egy záró idézőjel harmadik bájtja-e. Ehhez megcsekkoljuk az előtte
// levő 2 bájtot.
if((f.p[i-1]!=128)||((f.p[i-2]!=226)) {continue;} // Volt ugyan nyitóidézőjel, de ez nem egy csukóidézőjel vége,
// s emiatt megintcsak semmit se kell csinálnunk.
// Ha idáig eljutottunk, akkor viszont tuti hogy volt nyitóidézőjel, és ez egy csukóidézőjel utolsó bájtja.
voltenyitoidezojel=0; // E flagot lenullázzuk.
fuggvenyevpozicio_veg=i-2; // Ez a csukóidézőjel első bájtjára mutat.
fuggvenyev_hossz=fuggvenyevpozicio_veg-fuggvenyevpozicio_kezd;
if(fuggvenyev_hossz>254) {
L("Hiba a file beolvasása közben: túl hosszú függvéynév! A név kezdő pozíciója a fájlban:%u",fuggvenyevpozicio_kezd);
EXITFAILURE();
}
{register unsigned short int J;
for(J=0;J<fuggvenyev_hossz;J++) {fuggvenyevetomb[J]=(char)f.p[fuggvenyevpozicio_kezd+J];} // átmásoljuk a nevet
fuggvenyevetomb[fuggvenyev_hossz]=0; // lezárjuk a végét egy nullabájtjal
} // register J vége
uj = new PGM; // Elkészítjük az új PGM vázát a default konstruktorral
uj->f.p=f.p+i+1; // itt fog kezdődni az új PGM kódszegmense.
uj->f.phossz=f.phossz-i-1; // A hossz a maradék.
uj->MAUprogramNEVE=fuggvenyevetomb; // beállítjuk a progí nevét
uj->f.programfileneve=f.programfileneve;
uj->f.programfilenevehossz=f.programfilenevehossz;
uj->f.parentF=&f; // beállítjuk neki a szülő PGM F struktúrájának címét is, mert az jó lesz neki esetleg ha ismeri.
uj->f.pgmtomb=f.pgmtomb; // Beállítjuk az új PGM-nél, hogy a szülőnévtérben keresse a függvényeket ő is, mert hátha
olyan
// függvényt akar meghívni, ami vele egy blokkba tartozik
levon=i+1; // Ennyit kell majd levonnia a címkek abszolút értékéből, hogy beállítsa az új proginak őket jól
aktF=&uj->f; // Ezentúl ebbe az F struktúrába pakolja a címkeket
f.PGMtomb[f.maxpgm++]=uj; // és betesszük a listába
} // if 157 vége

// Itt végződik a programfájl szétadarabolása különálló függvényekre
// *****

if(kar==194) {if(sorvegflag==1) sorvegflag=194; else sorvegflag=0;continue;} // $ jel első bájtja!
if(kar==167) // $ jel második bájtja!
    {if(sorvegflag==194) sorvegflag=167; // Most már biztos, hogy címket olvasunk be.
    else sorvegflag=0; // if 194 vége
    continue;
    } // if 167 vége
// Ide akkor jutunk ha valami „közönséges” karaktert olvastunk be, ami lehet egy címke karaktere is.
if(sorvegflag==167) {cimke1karakter=kar;cimke1karakter |= 128; cimke1karakter -= 0x40; cimke1karakter &= 63;
    sorvegflag=11;continue;} // Eltároltuk a címke első karakterét
if(sorvegflag==11) { sorvegflag=0; // Egy karakter már biztos megvan a címkéből
if(wspc(kar)) { // A címke csak 1 karakteres mert a második karakter whitespace
if(aktF->cimke[cimke1karakter]) {L("Többször használod ugyanazt az 1 karakteres címkét! Pozíció: %lu
",i);fclose(fp);EXITFAILURE();}
aktF->cimke[cimke1karakter]=i+1-levon;aktF->cimkedb++;sorvegflag=0;continue;} // if wspc vége
// Nem whitespace a második karakter
kar |= 128; kar -= 0x40; kar &= 63;
if(aktF->cimke[((unsigned int)cimke1karakter) * 64 + (unsigned int)kar])
{L("Többször használod ugyanazt a 2 karakteres címkét! Pozíció: %lu ",i);fclose(fp);EXITFAILURE();}
aktF->cimke[((unsigned int)cimke1karakter) * 64 + (unsigned int)kar]=i+1-levon;aktF->cimkedb++;sorvegflag=0;continue;
} // if 11 vége
sorvegflag=0;
} // for i vége
fclose(fp);
qsort(f.PGMtomb,f.maxpgm,sizeof(PGM*),PGMhasonlit);
}

```

Mint látható, e rutin a legvégén a qsort-tal névsorba is rendezi a beolvasott függvényeket. Az ehhez szükséges rutin egyszerű:

```

int PGMhasonlit(const void *elso, const void *masodik) {
PGM *e;PGM *n;e=(PGM **)elso;n=(PGM **)masodik;
return MAUSTRINGhasonlit((const void *)e->MAUprogramNEVE),(const void *)n->MAUprogramNEVE);
}

```

Ahhoz hogy e beolvasórutin működjék, ki kell bővítsük az F struktúrát:

```
PGM * PGMtomb[PGMTOMBSIZE];
PGM** pgmtomb;
unsigned int maxpgn;
```

A beolvasórutin működésének megértéséhez továbbá óhatatlanul szükséges már most a legelején tisztázni, a mau függvényeink miféle szintaxis szerint kell megírtak legyenek!

Nos, ez a következő:

Mindenekelőtt: szigorúan a főprogram, a „main” kell legyen a programfájl legislegelején. Őt nem muszáj elnevezni (sőt, nem is szabad...), de ettől még neki akkor is, automatikusan a „main” név lesz adva az interpreterünk által, mert tiszteljük a szívemnek oly kedves C nyelv hagyományait...

Aztán, e main függvény illik hogy valahol végetérjen egy

XX

utasítással, s ezután jöhetnek a mindenféle, fájlban belüli függvények. Ezek alakja a következő példa szerinti:

```
„Hányados és maradék megjegyzésekkel” // Két számnak adja vissza a hányadosát és a maradékát
#i@e, // első paraméter - az osztandó
#i@m // második paraméter - az osztó
#i@h=(@e)/(@m); // hányados
#i@M=(@e)>@m; // maradék
\ // Itt jön az output paraméterek átadása
// Ez egy többsoros megjegyzés!
#i@h, // Első output paraméter - a hányados
#i@M; // Második output paraméter, a maradék
xx
```

Mint látható, a függvény neve egyszerűen a magyar szabványoknak megfelelő idézőjelek közt áll, azaz alul nyitó, felül záró idézőjelek közt. Jelen esetben a függvény neve az, hogy "Hányados és maradék megjegyzésekkel". Az idézőjelek nem tartoznak a függvény nevéhez. A bal oldali, alsó nyitó idézőjel a címkéinkhez hasonlóan szigorúan csak egy sor legislegelső karaktere lehet!

Mint már a fenti példából is sejthető, a címke neve abszolút bármiféle (UTF-8 kódolás szerinti) karaktert tartalmazhat, ékezeteseket is, kivéve a 0 kódú, azaz „szakmaian” írva a **CHR\$(0)** bájtot, ami amúgy is stringek végét jelzi, valamint a magyar nyitó- és záró idézőjeleket tehát a „ és ” karaktereket, meg az „angol” idézőjelet, a " jelet.

Utóbbi, az angol idézőjelet tulajdonképpen tartalmazhatja elvileg a címke neve, nincsen azzal semmi baj, épp csak ezesetben rém macerás lesz a használata, mert speciális módon lehet csak megadni stringekben, szóval ettől inkább óvakodjunk. Teli van sok ezer karakterrel az UTF-8 kódolású UNICODE táblázat, vannak benne mindenféle gyönyörűséges vesszők, aposztrófok meg más határoló-karakterek, minden mi szem-szájnak ingere, erről az egy jelről nyugodtan lemondhatunk. Ismétlem azonban nem kötelező erről lemondani, mindenki szívathatja magát vele ha akarja, de minek ha nem muszáj... Esetleges rossz-szándékú kritikusaímnak megjegyzem, függvényeim elnevezése még e korlátozás mellett is

messze nagyobb szabadságot nyújt mint a C, C++, Pascal vagy más nyelvek elnevezési szabályai, mert ugyan melyikben is tehető ékezetes karakter is a függvény nevébe, még szóköz sem... a mau nyelvben ez mind SZABAD, még kínai, héber, koreai, japán vagy akármi más karaktereket is! Azaz, ha tegyük fel valami olyan szubrutint készítünk, ami a legendás izraeli énekesnő, Ofra Haza élettörténetéről ír ki információkat, vagy valamit az ő dalaival kapcsolatban végez, akkor az e feladatot ellátó függvénynek nyugodtan adhatjuk az ő nevét úgy, ahogy az eredetileg írva van, héber betűkkel:

„עפרה חזה” // Ofra Haza függvénye

A függvény meghívása pedig a következő szintaxis szerint történik:

(„Hányados és maradék megjegyzésekkel” 26, 7 \ #i@H, #i@M); // Függvény meghívása, 2 input és 2 output paraméterrel

Azaz, a függvényt meghívó utasítás tulajdonképpen a kerek nyitó zárójel és az utána következő alul nyitó („magyar”...) idézőjel. Ezután nem muszáj hogy egy angol idézőjelek közt levő függvénynév álljon, mert állhat ott tetszőleges string-kifejezés is, például olyasmi, hogy

#s@f

de nyilvánvaló, hogy a legtöbbször egyszerűen ki szeretnénk írni oda a függvény nevét, s nem változóban eltárolva meghívni azt. Na most amennyiben ki akarjuk írni, akkor azt angol idézőjelek közt kell megtegyük, mert az azok közti karakterláncot alakítja át az interpreterünk stringgé. Most már gondolom érthető, miért nem jó ötlet ha a függvénynek a nevében is szerepel angol idézőjel...

A függvény nevét meghatározó stringkifejezés után egy záró magyar idézőjelnek kell állnia. Ezután hogy mi jön, attól függ, milyen a meghívandó függvény: akarunk-e neki paramétereket átadni, s várunk-e tőle visszatérési értékeket. Ha ezek egyike se áll fenn, egyszerűen egy csukózárójel kell következzen.

Ha adunk át input paramétereket, akkor a csukóidézőjel után kell felsorolni azon aritmetikai vagy stringkifejezéseket, melyek a paraméterek. Ezek érték szerint lesznek átadva, azaz ha itt változókat adunk meg, azok tartalmát nem módosítja a meghívott függvény. A paramétereket egymástól elválaszthatja vessző, de ez nem kötelező.

Ahogy végetért az input paraméterek listája, következik egy \ jel, azaz egy „visszaperjel”, angolul backslash. Ez az a baromság amit mindig is utáltam a Windows alatt, az, ami ott az egyes könyvtárakat elválasztja egymástól. Linux alatt szerencsére megszabadultam ettől az idiótaságtól, ott a könyvtárszeparátor a normális per-jel. Most azonban szükségem volt valami speciális jelre, amit nem nagyon használok máshol, főleg nem aritmetikai vagy string kifejezésekben műveleti jelként, s így felhasználtam e backslash jelet, mert itt még elviselhető a jelenléte, tudniillik nem lesz nagyon gyakran szükség a használatára feltehetően. Szóval ez a jel jelzi azt, hogy nincs több átadandó input paraméter, s következnek az úgynevezett „output paraméterek”. Amennyiben azok nincsenek, akkor a backslash jelet természetesen a csukózárójel követi, sőt, az esetben magára a backslash jelre sincs szükségünk, jöhet az input paraméterek után rögvést a csukózárójel!

Na most azonban ha vannak output paraméterek, pontosabban „visszatérési értékek”, ezeket természetesen a meghívott függvény fogja szolgáltatni, ezekből a hívási helyen a \ jel után már nem állhat aritmetikai vagy string kifejezés, hanem kizárólag csak VÁLTOZÓK, méghozzá olyan változók, amik nem a rövidített

nevükön szerepelnek, hogy mondjuk `@d`, hanem teljes alakjukban, kitéve eléjük a casting operátor jelet is, hogy például `#i@d`, tudniillik másképp nem tudja majd az interpreter, a hívott függvény visszatérési értékét milyen típusú változóba helyezze el!

Természetesen az output paramétereket is elválaszthatja egymástól vessző, de ez se kötelező. Az utolsó output paramétert kell kövesse a függvényhívást lezáró kerek csukózárójel, a `)`.

A hívott függvény a nevének végét követő csukó azaz „felső állású” magyar idézőjel utáni első karakternél kezdődik, mármint a végrehajtása kezdődik ott. Itt állhat `//` jelpárossal kezdődő megjegyzés, ami megmagyarázza hogy ez miféle függvény, mit csinál, stb, sőt e komment több soron át is tarthat, persze csak úgy, ha minden megjegyzéssor a `//` jellel kezdődik. A `/*...*/` típusú kommentek viszont itt nem alkalmazhatóak.

Mindenesetre, a `"` jel után amennyiben nem kommentek állnak, akkor a hívott függvény input paramétereinek fogadásához szükséges változók kell hogy itt szerepeljenek, természetesen ezek is teljes alakjukban, a `#` casting operátorral jelezve a fajtájukat. E változókat egymástól elválaszthatja vessző, de ez nem kötelező, valamint ezen változók közt is állhatnak `//` formájú kommentek. Sőt, érdemes tudnunk, hogy bár a hívó függvélynél az input paramétereket egymástól, valamint az output paramétereket egymástól csak whitespace vagy vessző választhatja el, a hívott függvény esetében akár az input, akár az output paraméterek listájában is a paraméterek elválasztására a pontosvesszőt is használhatjuk, ha akarjuk!

Vagyis, a hívási helyen nem használhatunk kommenteket, magának a hívott függvénynek a törzsében azonban a paraméterátadási helyeken igencsak szabad kommentelési lehetőségünk van, ami nagyon jó, mert kifejezetten helyes, ha alaposan elmagyarázzuk, melyik változó mire szolgál egy függvényben.

Amint a hívott függvélynél felsoroltuk az input változókat, kezdődik a lényegi kód, a függvény törzse. Ennek kezdetét semmiféle speciális jel nem kell hogy jelölje, tudniillik azt hogy ez hol kezdődik, onnan tudja, hogy elfogytak a hívási helyen az input változók/kifejezések, azaz ha nincs már több paraméter amit átvehetne a hívott függvény, onnan tudja hogy innentől már nem folytatódik az ő paraméterlistája, TEHÁT csakis a lényegi rész következhet, a végrehajtandó kód. Ennek végét pedig vagy egy

xx utasítás jelzi (ügyeljünk rá hogy ezek **kisbetűs** ikszek!), ezesetben nem kell semmiféle visszatérési értéket visszaadnia, vagy egy `\` karakter jelzi. Utóbbi esetben ezután következnek a visszatérési értékek. Mint írtuk, ezeket is elválaszthatja egymástól vessző, pontosvessző vagy `//` stílusú komment, de ez nem kötelező. Minden visszatérési érték tetszőleges aritmetikai vagy stringkifejezés lehet. Nem baj ha ezek alapvetően más típusúak mint amiket a hívó függvény output paraméterlistájában szereplő változók elvárnak, csak az a lényeg, hogy castolhatóak legyenek arra a típusra. Na most amint vége az output paraméterlistának a hívott függvélynél, maga a függvény is végetér, ezokból ilyen esetben nem kötelező az **xx** utasítás sem hogy jelezze a függvény végét. Ennek ellenére, célszerű kiírni, hogy olvashatóbbá tegye a kódot. Valójában a hívott függvény akkor is végetér, ha szerepel a végén a `\` jel, de utána egyetlen output paraméter

sem, mert olyat nem is vár a hívó függvény vissza, vagy ha szerepel ugyan valahány output paraméter a hívott függvény listájában, de a hívó valamiért kevesebbet vár el, azaz kevesebbnek tart fenn változót a hívási helyen! Ezesetben nem történik semmi baj, nincs hibajelzés, a felesleges változókat nem adja át (sőt ki se számolja) a hívott függvény. Akkor van csak baj, ha a hívó több visszatérési értéket várna, mint amennyit a hívott szolgáltatni akarna... Ekkor hibajelzést kapunk.

NAGYON FONTOS TUDNUNK AZT IS (!!!!!!!!!), hogy bár minden függvény végrehajtása mindig, kivétel nélkül a legelején kezdődik minden meghíváskor, ám a függvény által felhasznált belső változók értéke csak az első meghíváskor garantáltan nulla! Ugyanis az egyes meghívások során NEM TÖRLŐDIK A KORÁBBAN MEGKAPOTT ÉRTÉKÜK, akármennyi legyen is az!

Meg lehetett volna oldani könnyen, hogy ez ne így legyen, s filóztam is rajta, hogy megcsináljam-e — úgy kb egyetlen rövid programsor beszúrásából állt volna csak az egész! Mégsem tettem így, azaz ez „nem bug hanem feature”, ahogy mondani szokás. Nagyonis sok esetben rendkívül hasznos ugyanis, ha a függvény képes emlékezni mindenfélére az „előző életéből”! Például ha apránként adogatunk át neki mindenféle értékeket, melyeknek nem lehet előre tudni a darabszámát, ezekkel ő bütyköl mindenfelét, azt hogy nem jön több adat neki egy speciális adatfészeség jelzi például nulla vagy negatív érték, s ekkor ő visszaad pár fontos adatot, mondjuk a kapott paraméterek darabszámát, átlagát, összegét, stb. Ha nem emlékezne az előző hívások eredményeire, akkor efféle csak rém macerás kerülőutakon lehetne leprogramozni. Ha ellenben emlékszik az előző értékekre, akkor az ilyesmi nagyon könnyű, ellenben azon esetek megvalósítása se nehéz, amikor nem szabad emlékeznie a korábbi adatokra: azon fontos változókat ugyanis, melyek nem szabad hogy korábban létrejött adatokat tartalmazzanak, egyszerűen explicit módon nullára kell állítani a program legelején, például így:

```
#c@=0;
```

Arra az esetre ha sok adatot kéne lenullázni a függvény elején, két megoldás is megkönnyíti az életünket. Egyik az, hogy valahova ahova nekünk tetszik, beírjuk ezt a kódba:

```
00;
```

Ez garantáltan minden „single” adatot lenulláz, azaz minden nem tömb változót és nem veremváltozót, ezenfelül nullára állítja az **if** utasítás eredményét tároló flagot és a BRAINFUCK flagot is. Ez egy igen durva, azonban rém gyors módszer a „tisztalappal induláshoz”. Fontos azonban észben tartani, hogy ez a kapott input paramétereket is elfelejteti a függvénnyel, azaz ha azokra mégis szükség van, előbb verembe vagy tömbbe kell menteni őket! Ezenkívül, a lenullázás csak a numerikus változókra vonatkozik, a stringváltozókra NEM.

Abban az esetben ha mindent le szeretnénk nullázni a függvény indulásakor úgy, ahogy azt fentebb a 00; utasításnál írtam, de KIVÉVE az input adatokat, azt az input adatok listája előtt kell jelölnünk, EGYETLEN nulla jellel, eképp (zölddel emeltem ki a nulla helyét az alábbi példában):

```
„Hányados és maradék” 0 // Két számnak adja vissza a hányadosát és a maradékát  
#i@e; // első paraméter - az osztandó  
#i@m // második paraméter - az osztó  
// Itt kezdődik a függvénytörzs  
#i@h=(@e)/(@m); // hányados
```

```
#i@M=(@e)><(@m); // maradék
\#i@h; // hányados
#i@M;
xx
```

Minden beolvasott függvénynek van egy sorszáma. Egy függvény mindenképpen van, ez a main, ezesetben ha nincs más függvény, az ő sorszáma természetesen a nulla. Amint azonban több függvény is van beolvasva, nem okvetlenül lesz igaz hogy a main sorszáma a nulla, tudniillik amint beolvasta a teljes forrásprogramot, a függvényeket névsorba rendezi, a stringeknél bemutatott sorrendiségi szabályok szerint. Ezesetben a „main” nagy valószínűséggel már nem a nullás számot kapja.

Na most, ez nem baj, mert az esetek óriási többségében bennünket abszolút nem érdekel, miféle sorszáma van egy függvénynek, mert úgyis név szerint hivatkozunk rájuk! Ennek hogy mi a sorszáma, csak rendkívül speciális esetekben lehet jelentősége. Ekkor azonban fontos lehet megtudni, melyik függvénynek mi a sorszáma, vagy ellenkezőleg, melyik sorszámhoz épp melyik függvény tartozik, egyáltalán, hány függvényt is olvastunk épp be?! Nost, ezt a legkönnyebben úgy tudhatjuk meg, ha a forrásfájlba valahova beírjuk e kis függvényt, aztán amikor nekünk úgy tetszik, meghívjuk:

```
„Betöltve”
„Betöltve ” ?l ?f; ” függvény!\n”
#i@i=0;
{ | ?f // A ciklus annyiszor hajtódik végre, ahány betöltött függvény van,
„Az ” ?i @i; ”. függvény neve: „” ?s ?#s ”,”” #i@i; ””\n”
#i++@i;
| }
xx
```

Szerintem aki eddig eljutott a könyvem olvasásában, az komolyan érdeklődik a mau programnyelv iránt, s eddig bőven felszedett magára annyi ismeretet e nyelvből, hogy a fenti kis függvényből külön szájbarágás nélkül is megértse, miféle szintaktikával tehet szert a betöltött függvények darabszámára, s hogy melyik számhoz melyik függvénynév tartozik.

Ennek ellenkezője, amikor ismerjük a függvény nevét, s keressük a sorszámát:

```
„A \”Betöltve\” nevű függvény sorszáma: ” ?l ?#l “[ ] ” ”Betöltve”; /;
```

Na most a sorszámokat azért lehet jó ismerni némelykor, mert van lehetőségünk függvényt meghívni nem a nevének, hanem a sorszámának a megadásával! Például, ha igaz az hogy egy függvény sorszáma 1, akkor azt így is meghívhatjuk:

```
([1]); // Meghívtunk egy függvényt paraméterek nélkül. Nincs se input, se output paraméter
```

Természetesen ez csak a legegyszerűbb példa, mert egyrészt az „1” helyén a szögletes zárójelek közt tetszőleges aritmetikai kifejezés állhat ami meghatározza a meghívandó függvény sorszámát, másrészt a szögletes zárójeles kifejezés kizárólag a függvény nevét helyettesíti, azaz a csukó szögletes zárójel után halál-nyugodtan meg lehet adni ugyanúgy az input és/vagy output paraméterlistát, mintha a „**név**” szintaktikával a függvényt a maga neve szerint hívtuk volna meg. Azaz a függvényhívó szintaxis szerint, ha a függvényre igaz az, hogy a neve NÉV és a sorszáma X, akkor a függvényhívás vagy úgy kell kezdődjön hogy

```
(„NÉV”
```

Vagy úgy, hogy

```
([X]
```

és minden más teljesen azonos. Ja és a "**NÉV**" helyén állhat tetszőleges stringkifejezés, az **X** helyén pedig tetszőleges aritmetikai kifejezés.

És hogy e sorszám szerinti hívogatásra miért van lehetőség... Hát amiatt, mert függvényből nem csak annyi lehet mint változóból, hanem akár jócskán több is! Konkrétan, jelenleg most maximum 10 ezer lehet betöltve egyszerre, de ennek értéke fordítási paraméterként állítható, a vz.h include fájlban van meghatározva ezzel az utasítással:

```
#define PGMTOMBSIZE 10000
```

Na de nem ez a legnagyobb baj, hanem hogy a függvényeknél már semmi esetre se követhettem el olyan illetlenséget, hogy a nevük csak maximum 1 vagy 2 vagy netán 3 karakterből állhasson! Amint ilyesmit előírok a mau programnyelvben, azonnal kinéztek volna minden jobb társaságból... és nem azért, mintha mondjuk a maximum 3 karakteres nevek variációs lehetősége túl kevés lett volna, mert ha csak az angol ABC kis- és nagybetűit veszem alapul, az 52*52*52 lehetőség, ami 140608, s ez már aránylag egész szép mennyiség. Hanem ez amiatt nem lett volna mégsem megfelelő, mert ha elkezd sok mindenféle ember netán majd függvénykönyvtárakat írni a mau nyelvhez, ugyan miként is dőljön el, melyiküknek jut ez vagy az a 3 karakteres név... Állandóan keveredés lett volna belőle, melyik "kis" vagy "cut" nevű függvényt hívja meg a programunk...

Szóval, muszáj volt úgy megoldani ezt, hogy itt már tényleg tetszőleges string lehessen a függvények neve. Igenám, de ez meg azzal jár, hogy nem lehet elkerülni egyáltalán semmiképpen sem, hogy bizony a függvénynek minden meghívásánál végignyálazza a belső táblázatot az interpreter, megtudni, létezik-e az a nevű függvény, s ha igen, mi az ő címe!

Na most, ez azért mégsem akkora tragédia mint a változók esetében lenne, mert függvényeket nem hívogatunk olyan gyakran, mint ahogy a változókat használjuk. Mégis, ha nem is túl gyakran, de ELŐADÓDHAT olyan „élethelyzet”, amikor a program valami olyan sebességkritikus részhez ér, ahol még ez az idővesztés is számottevő lehet. Na most, ekkor megszabadíthatjuk ezen felesleges keresgéléstől az interpreterünket, ha a sebességkritikus részbe való belépés előtt lekérdezzük az ott sokszor meghívandó függvényünk sorszámát a neve alapján EGYSZER, mondjuk az F változóba betöltve, eképp:

```
#l@F=?#l "[" "Baromisokszor meghívandó függvény neve áll itt"
```

Majd nem a neve hanem e változó alapján hívjuk meg őt a sebességkritikus részben (mondjuk egy ciklus belsejében 12 milliószor...):

```
([#l@F] input paraméterek \ visszatérési értékek );
```

Na most, ez volt az első, s kétségtávolgyakoribb variációja a függvényeknek, ez, amikor a hívó programmal egy fájlban vannak. Amikor más fájlban vannak, annak leírása alább következik.

Ezen esetről azt kell tudni, hogy kizárólag azért, mert én olyan rém gonosz vagyok, és mindenféleképpen meg akarom nehezíteni annak a sok szerencsétlennek a sorsát aki a mau programnyelv elsajátítására adja a fejét, emiatt addig

törtem a fejemet, azaz addig lustálkodtam, míg kitaláltam, hogy programnyelvem bonyolultságát növelendő, a más fájlokban megvalósított függvények meghívásának szintaxisa PONTOSAN UGYANAZ, mintha a függvény nem is más, hanem a hívó függvénnyel azonos fájlban lenne!

Ugye milyen rémségesen gonosz vagyok, de tényleg és igazán?!

Ez azonban tényleg csak azt jelenti, hogy ott ahol meghívjuk a függvényt, ugyanúgy kell írunk mindent. Azt semmiképp se kerülhetjük el, hogy a függvény első meghívása előtt valamiképp „betöltsük” őt, azaz beolvassuk az őt tartalmazó fájlt az interpreterrel. Erre a célra természetesen egy külön utasítás szolgál, aminek szintén „**mau**” a neve. Ilyen nevű utasítást már csináltunk korábban, de az MAU volt, azaz nagybetűkkel íródott, ez azonban kisbetűs mau.

Szintaxisa:

mau path név

ahol a „path” a függvény fájljának elérési útvonala, a „név” pedig az a név, aminek alapján majd meg óhajtjuk hívni őt a programunkban. Bár ezen utóbbi nevet illik feltüntetni a hívott program forrásfájljának elején valahol, de ez tulajdonképpen nem kötelező, sőt, ott lehet akármiféle más név is, ez a hívó programot baromira nem fogja izgatni, ő azon a néven keresi és hívogatja majd, amit itt a beolvasáskor adunk neki. Természetesen a „path” és a „név” helyén is állhat tetszőleges stringkifejezés.

Természetesen van azért lehetőség arra, hogy a függvény megtudja, őt épp miféle néven azonosítva hívogatják! Mondjuk nem nagyon tudom elképzelni hogy ez miért, milyen esetekben lehet fontos egy függvény vagy program számára, de a lehetőség azért megvan rá hogy ezt lekérdezze, s e lekérdezés eredményét, ami természetesen egy string, például ilyesféleképpen irathatja ki:

"Az én nevem: " ?s ?#s "I"; /;

A fenti névlekérdezési lehetőség természetesen adott azon függvények számára is, melyek a hívó programmal azonos fájlban vannak. Ezek azt is megtehetik, hogy miután így megtudták a saját nevüket, ezután lekérdezzék a sorszámukat, mondjuk így:

"Az én sorszámon: " ?l ?#l "[" ?#s "I"; /;

Bár ennek se tudom, miféle gyakorlati haszna volna. Ennek ellenére, megteheti a függvény. Azok a függvények azonban, melyek külön fájlban vannak, már ezt az utóbbit NEM tehetik meg!

A nevüket le tudják kérdezni. A sorszámukat azonban nem. Lekérdezhetik, csak hogy ami eredményt kapni fognak ekkor, az NEM LESZ AZ IGAZI sorszámuk!

Tudniillik, amely függvény külön fájlból van betöltve, az egy külön PGM objektum. A neve, az őbenne magában van letárolva, amikor beolvassa őt az interpreter, akkor beállítja neki ezt a nevet. Egészen pontosan fogalmazva, nem akkor amikor beolvassa őt, hanem amikor létrehozza azt a PGM objektumot, aminek majd meghívja a fájlbeolvasási metódusát. A sorszámot azonban ami ezen

objektumhoz hozzá lesz rendelve, nem adja át neki, tudniillik ekkor még nem tudja - azért nem, mert miután megtörtént a beolvasás, ezen PGM objektum mutatóját is elhelyezi abban a tömbben, ahol minden más PGM objektumnak is tárolja a mutatóit, még önmaga mutatóját (a main programét) is, azaz azon PGM-ekét is amik vele a hívóéval egy fájlból lettek beolvasva, majd megint meghívja e tömbre a rendezőrutint! Azaz, minden egyes programbeolvasáskor az összes addigi program jelentős részének a sorszáma is jóeséllyel megváltozik. Tökéletesen lehetséges, hogy ugyanaz a függvény két hívás közt más sorszámot kap, ha közben más program is be lett olvasva a tárba. A neve azonban nem fog megváltozni.

Ehelyütt kell kitérnem arra, hogy más fájlokból csak EGY program hívható meg ezzel a módszerrel - az, ami ott a főprogram, azaz „legelől áll”. Természetesen lehet azon fájlban akárhány további másik program is, de azokat a mi programunk, ami ezt a másik programfájlt meghívta, már nem tudja meghívni, elérni, azokat csak az a program hívogathatja, ami abban a fájlban a főprogram! Ő azonban nem képes meghívni azokat a programokat, amik az őt hívó programmal vannak azonos fájlban. Azaz, minden program azon másik programokat tudja csak meghívni, amik vele azonos fájlban szerepelnek, illetve ami egy betöltött külső programfájl legelső programja, tehát ami abban a fájlban a főprogram!

Na most, azt ugye írtuk, hogy ezen programot ami másik fájlban szerepel (s abban a fájlban a főprogram) azt ugyanolyan szintaxissal kell meghívunk mint a többi más programokat. Miként kell azonban megírni ezen más fájlban levő programokat?

Nos, nem nagy a különbség. Egy efféle például így néz ki az elején:

```
#!„mauls” // Mau nyelvű tartalomjegyzék-listázó külső függvény
// Input paraméterek:
#s@A // A listázandó tartalomjegyzék neve
```

Vagy ha le akarjuk nullázni nála a változókat a hívás kezdeténél:

```
#!„mauls” 0 // Mau nyelvű tartalomjegyzék-listázó külső függvény
// Input paraméterek:
#s@A // A listázandó tartalomjegyzék neve
```

Az első sor tulajdonképpen csak egy megjegyzés, nem kötelező. Ez csak egy konvenció, ami arra utal, hogy ez egy mau nyelven írt külső függvény. E tényre a „ ” jelpáros utal, köztük van megadva a függvény neve. Utána állhat akármennyi // stílusú komment, majd a korábban már elmagyarázott szabályok szerint a függvény input paraméterei.

A legelső input paraméter előtt azonban okvetlenül kell álljon ha nem is egy efféle névmeghatározás, de legalább egy közönséges // típusú komment, és/vagy egy vessző vagy pontosvessző. Azaz semmiképp se kezdődjön a forráskód egyszerűen az input paraméterrel, mert akkor hibát fog jelezni. Ezt könnyen ki lehetne kerülni, de nem teszem, mert helyesnek látom ha ki van kényszerítve hogy ha sehol máshol nem is, de legalább a függvények legislegelején álljon legalább egyetlen nyamvadt megjegyzéssor, ami megmondja, mi a fészkes fenét is csinál ez a függvény.

Mint azt már tudjuk eddigre, a függvények külön névtérrel rendelkeznek, azaz ha van a függvényben mondjuk egy #c@G nevű változó, annak semmi köze a függ-

vényt meghívó ugyanilyen nevű és típusú változóhoz. Azoknál a függvényeknél azonban, melyek a hívó függvénnyel EGY FÁJLBAN VANNAK, és CSAK EZEKNÉL (!!!!), lehetősége van a hívott függvénynek kiolvasni a SZÜLŐ NÉVTÉR valamely változójának tartalmát!

FIGYELMEZTETÉS! NEM A HÍVÓ VÁLTOZÓJÁT olvashatja ki, hanem annak a „főprogramnak” a változóját, mely a „main” az ő fájljában, tehát azon fájl „legfelső szintű”, azaz a fájl legelején álló programnak a változóját!

Ennek jelölése pedig úgy történik, hogy a változónevet bevezető @ jel után közvetlenül odairunk egy ^ jelet is.

Azaz például:

#s@g a függvény „g” nevű stringváltozóját jelenti.

#s@^g a függvény szülőnévterének „g” nevű stringváltozóját jelenti.

És ismétlem, ezek csak input változók lehetnek, azaz csak „jobbértékben” szerepelhetnek. Értéket adni nekik nem lehet. Valamint, ez nem vonatkozik a veremváltozókra. Irtó nagy gondok lehetnének belőle, ha egy hívott függvény elkezdene értékeket kiszedegetni a szülőnévtér vermeiből.

Használható azonban e módszer még néhány más speciális esetben is, ezek a következők amint e példák mutatják őket:

```
<s @^o @S; // Output fájlba írás

?#l "T#" ^ t @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzéseinek
darabszámát kérdezhetjük így le

?#s "Tn" ^ t, #l@i, @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzései közül
a #l@i változó által megadott sorszámúnak a nevét kérdezhetjük így le

?#s "Tg" ^ t, #l@i, @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzései közül
a #l@i változó által megadott sorszámúnak a csoportját kérdezhetjük így le

?#s "Tt" ^ t, #l@i, @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzései közül
a #l@i változó által megadott sorszámúnak a tulajdonosát kérdezhetjük így le
?#c "EXE" ^ t, #l@i, @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzései
közül a #l@i változó által megadott sorszámúnál azt kérdezhetjük le, hogy végrehajtható állomány-e.

?#c "L?" ^ t, #l@i, @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzései közül
a #l@i változó által megadott sorszámúnál azt kérdezhetjük le, hogy link-e, s ha link akkor törött-e.

?#s "JOG" ^ t, #l@i, @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzései
közül a #l@i változó által megadott sorszámúnak a jogosultságait kérdezhetjük így le oktális formában

?#s "LINK" ^ t, #l@i, @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó link típusú bejegyzései közül
a #l@i változó által megadott sorszámúnál azt a stringet kérdezhetjük így le, amely fájlra vagy könyvtárra a symlink
mutat.
```

Igazából, általában véve is kerülendőnek tartom e változóhasználatot. Hatalmas, és rém nehezen kideríthető hibák forrása lehet. Ezen ^ által hívott változók tulajdonképpen globális változóknak minősülnek minden, az adott fájlban szereplő függvény számára, márpedig jól tudjuk már a C nyelvű programozásban szerzett tapasztalataink alapján is, hogy a globális változók használata nagyonis kerülendő! Néha azonban, nagyritkán, jól jöhet e lehetőség, például olyankor, ha egy nagy tömbről van szó, amit sok függvénynek kell használnia, márpedig tömb összes elemét egyenként átadogatni egy függvényhívásnál paraméterekként, hát az izé... se nem szép, se nem hatékony, szóval akkor már legyen az inkább globális változó. De mondom, az ilyesmivel nagyon óvatosan, „ésszel” kell bánni, mert ez veszélyes lehetőség!

Nagyon fontos azt is megemlítenünk, hogy e függvényhívási módszer **ALKALMATLAN REKURZÍV HÍVÁSOK KEZELÉSÉRE!** Ennek oka, hogy egyszerűen nem veremtáron keresztül adja át illetve fogadja a hívó függvény a paramétereket. Valamint a programmutató sincs ilyenkor veremtárba mentve. Az egész függvényhívási metódus megvalósítása, sőt, már a függvények de még magának a PGM nevű mau-programosztálynak az adatszerkezete is olyan, hogy teljességgel alkalmatlan bármiféle rekurzivitás lekezelésére is!

Sokat töprengtem e dolgon, s végül oda jutottam, hogy hiába alkalmatlan rá, **MEG TUDNÁM CSINÁLNI.** Igen, meg lehetne erőszakolni a struktúráját úgy, hogy erre képes legyen, de csak azon az áron, hogy egyrészt iszonyatosan túlbonyolódna a kód, másrészt örületes memóriapocséklással járna egy rekurzív hívás lebonyolítása. Nem érné meg.

Ennek ellenére, természetesen kell hogy a nyelvünk alkalmas legyen rekurzívan megoldható feladatok elvégzésére, de ahhoz másfajta függvényhívási módszert kell megkonstruálnunk, az azonban nem ezen fejezet tárgya.

26. fejezet: A „maudir” program, vagyis az első, valóban hasznos mau nyelvű programunk

Létezik a linuxos világban egy „vidir” nevű kis program, bár nem mindenki ismeri. Ez annyit csinál, hogy elindítása után a megadott nevű tartalomjegyzéket beolvassa (illetve ha nincs megadva tartalomjegyzék, akkor az „aktuális”-at), s ebből a könyvtár- és fájlneveket kiírja egy ideiglenes fájlba, úgy, hogy mindegyiknek az elejéhez odabiggyeszt egy sorszámot (a sorszám és a név közt természetesen van valamennyi whitespace). Miután e fájlt előállította, meghívja rá a \$EDITOR változóban beállított szövegszerkesztő programot, ha ilyen nincs beállítva akkor a „vi” nevű programot próbálja meghívni. Ezzel aztán szerkeszthetjük e fájlt, amennyiben a benne levő fájl- és könyvtárneveket átírhatjuk úgy, ahogy az nekünk tetszik, csak arra kell vigyáznunk hogy a sorok elején a sorszámokat ne piszkáljuk. Ellenben az szabad, hogy egész sorokat kitöröljünk.

Amikor ezzel megvagyunk s elmentettük a fájlt, a vidir program azt beolvassa, elemzi, s ahol úgy találja hogy az adott sorszámú sorban már nem az a könyvtár- vagy állománynév áll amire ő emlékszik, akkor azt átnevezi a megfelelőképpen, az új névre. Ahol meg nincs is már meg az adott sorszámú sor, azt a könyvtárat vagy fájlt egyszerűen törli.

Ez nagyon kényelmes a fájloknak főleg az átnevezésére, ha egyszerre sok efféle műveletet akarunk végrehajtani, s pláne ha a fájlnev hosszú, s esetleg teli is van mindenféle ékezetes karakterekkel. Mikor még csak egy szimplán nekem tetsző állománylistázó progit akartam összehozni, szerettem volna abba beleépíteni e „vidir” funkciót is. Legnagyobb bánatomra amikor belenéztem a vidir kódjába, kiderült, hogy nem C vagy C++ nyelven van, hanem Perl nyelven! Pfff... Perl nyelven nem tudok, és semmi kedvem nem is volt megtanulni. Annyit értettem belőle, hogy rájöjjenek, mit csinál. Na de így nem tudom a kódot „ellopni”, azaz beleépíteni egy C nyelvű programba...

Azonban a dolog nem hagyott nyugodni, mert a vidirt szerettem, de az nem tetszett hogy ha használni akarom, okvetlenül szükségem van hozzá az egész hatalmas Perl interpreterre, meg annak mindenféle könyvtáaira, stb. Miért is ne írhatnám meg én a vidirt C nyelven?! Nem olyan marha bonyolult dolog amit csinál...

Aztán mégis egyre halasztódott e projekt, most már azonban hogy van saját programnyelvem ami még azt a mocsokságot is képes elkövetni hogy MŰKÖDIK (hát nem pofátlan?!... meg kéne neki tiltani...), most már nincs ürügy, és meg kell írnom a magam vidir progiját, aminek nem is titkoltan dicsekvési célból a „maudir” nevet adtam. Elvégre úgyis muszáj legalább egyetlen olyan progit írni a nyelvem aminek valami konkrét haszna is van, hogy bizonyítsam e nyelv használhatóságát és működőképességét, akkor meg miért is ne lehetne ez épp a szeretett vidir progij... azaz bocsánat, „maudir”!

Íme a program:

```
#!/mau // maudir program.
// Készítette: Viola Zoltán, violazoli@gmail.com
// A program a "mau" nevű programozási nyelven íródott, azt is én Viola Zoltán készítettem.
// Mindegyik GPL licenc alatt van.

// Konstansok meghatározása

#s@P="/tmp"; // Annak a könyvtárnak a PATH-ja, ahova az elkészült ideiglenes fájl kerül, az,
// amit a $EDITOR változóban meghatározott texteditorral szerkesztünk majd.
#c@c='-'; // A pidstring üres helyeit kitöltő karakter
#c@p=6; // A pidstring maximális hossza
#s@i="MAUDIR_temporary_file"; // Az ideiglenes file alapneve a pidstring nélkül
#s@k=".tmp"; // Az ideiglenes file kiterjesztése

// Előkészítő műveletek
#s@p=?#s "PID" @p @c; // A pidstring előállítás
#s@n=(@i)+"_PID-"+@p+@k; // A leendő ideiglenes file nevének előállítása

// A leendő ideiglenes fájl teljes útvonalat tartalmazó nevének előállítása:
if((#s@P[!#s@P-1])!='') T #s@N=(@P)+"/"+ (@n);
E #s@N=(@P)+(@n);
// A fenti két sor azt csinálja, hogy ellenőrizzük, e progij legelején az ideiglenes fájl helyének megjelölt könyvtár
útvonala
// egy "/" jellel végződik-e. Ha ez nincs így, pótoljuk e per-jelet.

if (?a)<3 T #s@A="."; // Ha nem adtunk meg parancssori paramétert, az aktuális könyvtárra vonatkozik a hívás,
E #s@A=?#s "ARGV" 2; // különben a megadott tartalomjegyzékre

// Tartalomjegyzék beolvasása

#T@t= @A; // A lekérdezendő tartalomjegyzéket jelölő stringet a #s@A tartalmazza. Itt olvassuk be a directoryt.

// A szerkesztendő ideiglenes fájl megnyitása írásra
#K@o=@N; // Megnyitjuk az output file-ot. Ennek neve természetesen a korábban meghatározott string lesz,
// ami a PID értékét is tartalmazza. A @N változó e nevet tartalmazza, az elején kiegészítve a könyvtárát tartalmazó
// elérési útvonallal.

if #K@o E "Az output file nem megnyitható!\n" XX // Hiba esetén kilépünk.
// Figyeljük meg, a fenti if utasítás nem is tartalmaz THEN ágat... Így egyszerűbb, nem bajlódunk az összehasonlító
// művelet eredményének negálásával.

if ((#s@A[!#s@A-1])!='') T #s@A=(@A)+"/"; // Itt ellenőrizzük, e progij legelején a beolvasandó tartalomjegyzék
útvonalának
// vége egy "/" jel-e. Ha ez nincs így, pótoljuk e per-jelet. Ha netán nem lett volna ott, az nem baj a tartalomjegyzék
// beolvasásakor még, mert ezt akkor pótolja az interpreter beolvasórutinja maga is nagy előzékenyen. De most ezt nekünk
is
// meg kell csinálni.

// Directoryk:
#l@i=0; // Ciklusszámláló kezdeti értéke
if (#l (?#l "T#" t D)==0) T »$DV; // Ha nincs altartalomjegyzék, nem ír ki azokról semmi adatot. Nem hülye ő...
[#c@D=?#l "T#" t D] // Memória foglalás annyi darab unsigned char értéknek a "D" nevű tömbbe, ahány directory bejegyzés
van.
[| ?#l "T#" t D; // Ciklus indul, annyiszor fut le, ahány altartalomjegyzék van. Legalább egy egészen biztosan akad.
(,"A fájlba kiírandó sor előállítás" "D:",#l@i,?#s "Tn" t, #l@i, D \ #s@S); // Függvényhívás
<s @o @S; // A sor kiírása
#l++@i; // Ciklusszámláló növelése
```



```

}} // A ciklus vége
$DV // Directoryk vége

// Közöséges fájlok:
#l@i=0; // Ciklusszámláló kezdeti értéke
if (#l (?#l "T#" t R)==0) T »$RV; // Ha nincs közöséges fájl, nem ír ki azokról semmi adatot. Nem hülye ő...
[#c@R=?#l "T#" t R] // Memória foglалás annyi darab unsigned char értéknek az "R" nevű tömbbe, ahány közöséges fájl van.
{ | ?#l "T#" t R; // Ciklus indul, annyiszor fut le, ahány közöséges (reguláris) fájl van. Legalább egy egészen biztosan
akad.
(,,"A fájlba kiírandó sor előállítás") "R:",#l@i,?#s "Tn" t, #l@i, R \ #s@S); // Függvényhívás
<s @o @S; // A sor kiírása
#l++@i; // Ciklusszámláló növelése
}} // A ciklus vége
$RV // Reguláris fájlok vége

// Szimbolikus linkek:
#l@i=0; // Ciklusszámláló kezdeti értéke
if (#l (?#l "T#" t L)==0) T »$LV; // Ha nincs link fájl, nem ír ki azokról semmi adatot. Nem hülye ő...
[#c@L=?#l "T#" t L] // Memória foglалás annyi darab unsigned char értéknek az "L" nevű tömbbe, ahány szimbolikus link
van.
{ | ?#l "T#" t L; // Ciklus indul, annyiszor fut le, ahány szimbolikus link van. Legalább egy egészen biztosan akad.
(,,"A fájlba kiírandó sor előállítás") "L:",#l@i,?#s "Tn" t, #l@i, L \ #s@S); // Függvényhívás
<s @o @S; // A sor kiírása
#l++@i; // Ciklusszámláló növelése
}} // A ciklus vége
$LV // Reguláris fájlok vége

[#K@o]; // Lezárom az output file-ot

SY (?#s "ENV" "EDITOR") + " " + @N; // Megnyitjuk a fájlt a $EDITOR környezeti változó által meghatározott text
editorral

#B@b=@N; // Megnyitjuk a szerkesztett fájlt input fájlként.
if #B@b E "Az ideiglenes fájl nem megnyitható beolvasásra!\n" XX

{( // beolvasóciklus kezdete. Végtelenciklus.
#s@s=?#s "NL", b, 1000; // Beolvassuk a sort, de max. 1000 karaktert
if(!#s@s)==0 TT »$tt; // Ha üres sort olvastunk be, vége a fájlnek, ekkor elugrunk a törlésellenőrzési részhez
// Itt kezdődik a sor feldolgozása.
#c@t:0=#s@s[0]; // A file típusa
#s@h=[2,11]@s; // A file sorszáma stringként
#l@h=#s #s "SP" @h; // Átalakítjuk a file sorszámát stringből unsigned int számmá. Ehhez előbb le kell vágni
// a string elejéről a szóközöket. Ezt végzi el az iménti "SP" függvény: levág minden whitespace karaktert a string
elejéről.
#s@a=[16,]@s; // A file neve
#s@a=(#s@a) - 1; // A hosszából le kell vonnunk 1-et, mert másképp a file nevéhez tartozónak tartaná a 10-es karakterkódú
// sorvégjelet is.
if(@t:0)==D T #c@D[#l@h]=1; »$qq; // Ebben a pár sorban azt vizsgáljuk, milyen típusú fájlról szól a beolvasott sor.
if(@t:0)==R T #c@R[#l@h]=1; »$qq; // Amilyen típusú, az ahhoz tartozó tömb megfelelő sorszámú elemét 1-re állítjuk. Ez
amiatt
if(@t:0)==L T #c@L[#l@h]=1; // fontos, mert így tudjuk követni, nincs-e kitörölve a fájlból az adott sorszámú fájl
sora.
// Ha ugyanis azt kitörölte a Felhasználó, majd törölni kell azt a fájlt vagy könyvtárat is!

$qq

if (#s (?#s "Tn" t, #l@h, (@t:0)) != (#s@a)) T SY "mv " + (@A) + (?#s "Tn" t, #l@h, (@t:0)) + " " + (@A) + (@a);
// Fentebb azt csináltuk, hogy ellenőrizzük, a megfelelő típusú és sorszámú fájl neve a beolvasott tartalomjegyzék-
struktúrában
// azonos-e azzal, amit épp beolvastunk az ideiglenes fájl sorából. Ha NEM azonos vele, meghívjuk a SY
rendszerparanccsal
// a shell "mv" átnevező funkcióját, s átnevezzük az új névre.
)} // Visszaugrás a beolvasóciklus elejére

$tt

[#B@b]; // Bezárjuk az ideiglenes fájlt.
SY "rm " + @N; // Töröljük az ideiglenes fájlt a shell segítségével

if (#l (?#l "T#" t D)==0) T »$tD; // Ha nincs altartalomjegyzék, nem fut le e ciklus.
{ | ?#l "T#" t D ; // Fix számszor lefutó ciklus, annyiszor fut le ahány dir bejegyzésünk van.
if #c@D[?] -1] E SY "rm -rf " + (@A) + (?#s "Tn" t, ?| -1, D)
}}
$tD

if (#l (?#l "T#" t R)==0) T »$tR; // Ha nincs reguláris fájl, nem fut le e ciklus.
{ | ?#l "T#" t R ; // Fix számszor lefutó ciklus, annyiszor fut le ahány reguláris fájl bejegyzésünk van.
if #c@R[?] -1] E SY "rm " + (@A) + (?#s "Tn" t, ?| -1, R)
}}
$tR

if (#l (?#l "T#" t L)==0) T »$tL; // Ha nincs link, nem fut le e ciklus.
{ | ?#l "T#" t L ; // Fix számszor lefutó ciklus, annyiszor fut le ahány link bejegyzésünk van.
if #c@L[?] -1] E SY "rm " + (@A) + (?#s "Tn" t, ?| -1, L)
}}
$tL

XX // A program vége

```

```
// ===== Függvények

„A fájlba kiírandó sor előállítás” // Ez a függvény állítja elő azt a stringet, amit az ideiglenes fájlba kiíratandó 1
sor.
// Input paraméterek
#s@t, // Az a string ami jelzi, a file directory-e (D) vagy közönséges fájl (R) vagy szimbolikus link (L vagy >)
#l@i, // A file sorszáma. A sorszámot jobbra igazítva, fix hosszú mezőbe írjuk ki. E tényeket az garantálja,
// hogy egész számot alakítunk át stringgé, s ezen átalakítás nálunk automatikusan így történik, hála az
interpreterünknek.
// Ez az ő egyik beépített szépsége a számunkra, fokozandó a „felhasználói élményt”...
#s@n // A kiíratandó file neve
// Most jön az output paraméter meghatározása, a visszatérési érték:
\ #s (@t)+(#l@i)+ " = "+(@n)+"\n"; // A fájlba kiírandó sor visszaadása stringkifejezésként
xx // A függvény vége
```

Na most, ez már olyan program ami TÉNYLEG használható komoly célra, TÉNYLEG végez nekünk HASZNOS funkciót! Az nyilván természetes, hogy magának a mau interpreternek a működő bináris fájlját a „mau” név azonosítsa, s kell szerepeljen valahol a rendszerünk \$PATH változója által meghatározott valamelyik elérési útvonalon. De hogyan is hívjuk meg a mau nyelven írt programokat kényelmesen?

Nos, erre a legtutibb megoldás az, ha valahova készítünk egy mondjuk „maubin” nevű könyvtárat, s abba pakoljuk be a fenti programot, aminek legyen a neve maudir.mau, meg minden más mau nyelven megírandó programunk is e könyvtárba kerüljön, s mindnek a neve legyen **.mau** kiterjesztéssel ellátva. Ezután pedig készítsük el e szkriptet:

```
#!/bin/bash
n=$1
shift
mau /_P/Szkriptjeim/-/maubin/$n.mau "$@"
```

E fenti szkript neve legyen egyszerűen csak „**m**”. És ez is legyen valahova a \$PATH-ba mentve, persze adjunk rá futtatási jogot is. S ezek után tetszőleges mau nyelvű program meghívása, ami a maubin könyvtárban van, s aminek a neve progineve.mau így néz ki:

```
m progineve paraméterek_ha_vannak
```

A fenti szkript nagy előzékenyen megkímél ugyanis bennünket a **.mau** kiterjesztések beirogatásától, s az összes parancssori paramétert is átpasszolja a mau proginak. Ennek megfelelően, a fenti maudir progí meghívása az aktuális könyvtárra:

```
m maudir
```

Valamely másik könyvtárra pedig mondjuk efféleképp lehet meghívni:

```
m maudir $HOME/munka
```

Na most a fenti maudir program ugyan MŰKÖDIK, de senkinek se ajánlom, hogy ilyen trehány stílusban programozzon. Ez ugyanis így ahogy most van, egy szőföső, bloatware szar, ami amolyan „csapjuk össze fél óra alatt” módon készült „AS IS” minőségben. („AS IS” = „**A**hogy **S**ikerül. **I**gy **S**ikerült”). Például egy csomó dolgot kétszer, sőt háromszor ír le, háromszor ír le, háromszor ír le... Azaz, tisztességes programozó, egy úriember, aki valamit is ad a szakmájára, ilyesmit egyszerűen nem követ el, mert e stílus nem használja ki a gyönyörűséges mau programnyelv megannyi számos roppant lehetőségét, például azt, hogy itt a változók NEVE is lehet tetszőleges aritmetikai kifejezés... Mi azonban Olvasóm véleménye mondjuk erről a változatról?!

```

#!mau // maudir program.
// Készítette: Viola Zoltán, violazoli@gmail.com
// A program a "mau" nevű programozási nyelven íródott, azt is én Viola Zoltán készítettem.
// Mindegyik GPL licenc alatt van.

// Konstansok meghatározása

#s@P="/tmp"; // Annak a könyvtárnak a PATH-ja, ahova az elkészült ideiglenes fájl kerül, az,
// amit a $EDITOR változóban meghatározott texteditorral szerkesztünk majd.
#c@c='-'; // A pidstring üres helyeit kitöltő karakter
#c@p=6; // A pidstring maximális hossza
#s@i="MAUDIR_temporary_file"; // Az ideiglenes file alapneve a pidstring nélkül
#s@k=".tmp"; // Az ideiglenes file kiterjesztése

#s@Z="DRL"; // Ezekkel a típusú tartalomjegyzék-bejegyzésekkel foglalkozunk a maudir programban. És ebben a sorrendben
// fognak megjelenni a szerkesztendő ideiglenes fájlban felülről lefelé:
// D azaz Directoryk,
// R azaz Reguláris (közönséges) fájlok
// L azaz Linkek.

// Előkészítő műveletek
#s@p=?#s "PID" @p @c; // A pidstring előállítása
#s@n=(@i)+"_PID-"+(@p)+(@k); // A leendő ideiglenes file nevének előállítása

// A leendő ideiglenes fájl teljes útvonalat tartalmazó nevének előállítása:
if((#s@P[(!#s@P)-1])!='') T #s@N=(@P)+"/"+ (@n);
E #s@N=(@P)+(@n);
// A fenti két sor azt csinálja, hogy ellenőrizzük, e progi legelején az ideiglenes fájl helyének megjelölt könyvtár
útvonala
// egy "/" jellel végződik-e. Ha ez nincs így, pótoljuk e per-jelet.

if (?a)<3 T #s@A="."; // Ha nem adtunk meg parancssori paramétert, az aktuális könyvtárra vonatkozik a hívás,
E #s@A=?#s "ARGV" 2; // különben a megadott tartalomjegyzékre

// A tartalomjegyzék beolvasása

#T@t= @A; // A lekérdezendő tartalomjegyzéket jelölő stringet a #s@A tartalmazza. Itt olvassuk be a directoryt.

// A szerkesztendő ideiglenes fájl megnyitása írásra
#K@o=@N; // Megnyitjuk az output file-ot. Ennek neve természetesen a korábban meghatározott string lesz,
// ami a PID értékét is tartalmazza. A @N változó e nevet tartalmazza, az elején kiegészítve a könyvtárát tartalmazó
// elérési útvonallal.

if #K@o E "Az output file nem megnyitható!\n" XX // Hiba esetén kilépünk.
// Figyeljük meg, a fenti if utasítás nem is tartalmaz THEN ágat... Így egyszerűbb, nem bajlódunk az összehasonlító
// művelet eredményének negálásával.

if ((#s@A[(!#s@A)-1])!='') T #s@A=(@A)+"/"; // Itt ellenőrizzük, e progi legelején a beolvasandó tartalomjegyzék
útvonalának
// vége egy "/" jel-e. Ha ez nincs így, pótoljuk e per-jelet. Ha netán nem lett volna ott, az nem baj a tartalomjegyzék
// beolvasásakor még, mert ezt akkor pótolja az interpreter beolvasórutinja maga is nagy előzékenyen. De most ezt nekünk
is
// meg kell csinálni.

{! #s@Z; if (#l (?#l "T#" t #s@Z[-- ?]])>0) T [#c@(#s@Z[-- ?]])=?#l "T#" t (#s@Z[-- ?])];
} // E ciklusban foglalkunk le memóriát annyi darab unsigned char értéknek a megfelelő nevű tömbbe,
// ahány tartalomjegyzéki bejegyzés van abból a típusból.

{! #s@Z; (,,"Kiíró rutin" (#s@Z[(!#s@Z) - ?]));
} // A kiíró függvény meghívása ciklusban. Ez egy fix számszor lefutó ciklus, és indexelésre a ciklusváltozót
// használjuk, amire a ?| utal. Ez azonban egyre csökken, így hogy a kiírás a típusok megfelelő sorrendjében
// történjék, a stringhosszból mindig levonjuk a ciklusváltozó értékét.

[#K@o]; // Lezárom az output file-ot

SY (?#s "ENV" "EDITOR") + " " + @N; // Megnyitjuk a fájlt a $EDITOR környezeti változó által meghatározott text
editorral

#B@b=@N; // Megnyitjuk a szerkesztett fájlt input fájlként.
if #B@b E "Az ideiglenes fájl nem megnyitható beolvasásra!\n" XX

{( // beolvasóciklus kezdete. Végtelenciklus.
#s@s=?#s "NL", b, 1000; // Beolvassuk a sort, de max. 1000 karaktert
if(!#s@s)==0 TT »$tt; // Ha üres sort olvastunk be, vége a fájlnak, ekkor elugrunk a törlésellenőrzési részhez
// Itt kezdődik a sor feldolgozása.
#c@t=#s@s[0]; // A file típusa
#s@h=[2,11]@s; // A file sorszáma stringként
#l@h=#s ?#s "SP" @h; // Átalakítjuk a file sorszámát stringből unsigned int számmá. Ehhez előbb le kell vágni
// a string elejéről a szóközöket. Ezt végzi el az iménti "SP" függvény: levág minden whitespace karaktert a string
elejéről.
#s@a=[16,]@s; // A file neve
#s@a=(#s@a) - 1; // A hosszából le kell vonnunk 1-et, mert másképp a file nevéhez tartozónak tartaná a 10-es karakterkódú
// sorvégjelet is.

#c@(@t)[#l@h]=1; // Ebben a sorban azt vizsgáljuk, milyen típusú fájlról szól a beolvasott sor.
// Amilyen típusú, az ahhoz tartozó tömb megfelelő sorszámú elemét 1-re állítjuk. Ez amiatt
// fontos, mert így tudjuk követni, nincs-e kitörölve a fájlból az adott sorszámú fájl sora.
// Ha ugyanis azt kitörölte a Felhasználó, majd törölni kell azt a fájlt vagy könyvtárat is!

```

```

if (#s (?#s "Tn" t, #l@h, (@t)) != (#s@a)) T SY "mv " + (@A) + (?#s "Tn" t, #l@h, (@t)) + " " + (@A) + (@a);
// Fentebb azt csináltuk, hogy ellenőrizzük, a megfelelő típusú és sorszámú fájl neve a beolvasott tartalomjegyzék-
struktúrában
// azonos-e azzal, amit épp beolvastunk az ideiglenes fájl sorából. Ha NEM azonos vele, meghívjuk a SY
rendszerparanccsal
// a shell "mv" átnevező funkcióját, s átnevezzük az új névre.
}) // Visszaugrás a beolvasóciklus elejére

$tt

[#B@b]; // Bezárjuk az ideiglenes fájlt.
SY "rm " + (@N); // Töröljük az ideiglenes fájlt a shell segítségével

{! #s@Z; (, "Töröl rutin" (#s@Z[-- ?]));
} // A töröl függvény meghívása ciklusban

XX // A program vége

// ===== Függvények

„Töröl rutin” // Törli a megfelelő fájlbejegyzést
// Input paraméterek:
#c@t // típus: D: directory, R: reguláris fájl, L: link

if (#l (?#l "T#" ^ t @t)==0) T xx; // Ha nincs a megfelelő típusú fájlból, vége a rutinnak.

if (@t)==D T #s@y="rm -rf ";
E #s@y="rm ";

{! #l "T#" ^ t @t ; // Fix számszor lefutó ciklus, annyszor fut le ahány fájlbejegyzésünk van.
if #c@^(@t)[?] -1] E SY (@y) + (@A) + (?#s "Tn" ^ t, ?| -1, @t)
}

xx

„Kiíró rutin” // Kiírja a megfelelő sort a fájlba
// Input paraméterek:
#c@t // típus: D: directory, R: reguláris fájl, L: link

#s@y=" "; #s@y[0]=@t;
#l@i=0; // Ciklusszámláló kezdeti értéke
if (#l (?#l "T#" ^ t @t)==0) T xx; // Ha nincs megfelelő típusú fájl, nem ír ki azokról semmi adatot. Nem hülye ő...
{! #l "T#" ^ t @t; // Ciklus indul, annyszor fut le, ahány fájl van az adott típusból. Legalább egy egészen biztosan
akad.
<s @^o (@y)+(#l@i)+ " = "+(?#s "Tn" ^ t, #l@i, @t)+"\n"; // A sor kiírása
#l++@i; // Ciklusszámláló növelése
} // A ciklus vége
xx

```

Na EZ már tényleg él a mau nyelv pompázatos lehetőségeinek széles skálájával!

Legalábbis az eddigiekkel... Mert ugye, épp az a legfőbb célja egy efféle projektnek, mint egy valóban értelmes célra is használható komolyabb progi megírása egy új nyelven, hogy közben felfigyeljünk azon funkciókra, melyek a munkához hiányoznak a nyelvben, holott jaj de jó lenne, ha a nyelv rendelkezne velük! És itt most épp ezen eset áll fenn. Mert figyeljük csak meg, számos alkalommal vált nekünk szükségessé, hogy egy ciklus annyszor fusson le, ahány karakter van egy stringben. Miért is ne lehetne nekünk egy olyan ciklusvariációnk is, ami ezt magától tudja?! S ami már természetesen nem csökkenő sorrendben számol, hanem nullától kezdve, mert ugyebár aki nem valami végletekig elfajzott perverz mutogatós bácsi, az egy stringet nem a végétől kezd el elejéig vizsgálni, hanem az elejétől végéig vizsgálni...

Nos, csináljuk csak meg a {! ... !} cikluslehetőséget, ahol a ... helyén áll természetesen a ciklusmag! Ez telibepontosan ugyanaz lesz mint a megszokott fix darabszámszor lefutó {! ... !} ciklusunk, csak a {! után egy stringkifejezést adunk meg szám helyett. Természetesen itt is lehet kilépési feltételt meghatározni a } jel után, zárójelek közt.

E ciklus két szükséges függvénye:

```
int nyitokapcsoszarojelfelkialtojel(F& f) { // Fix darabszámú lefutó ciklus, annyiszor fut le, amennyi
// a megadott stringkifejezés hossza.
MAUSTRING A;MAUSTRING *m;
A=ERTEKstring(f); // Beolvassuk a stringkifejezést
if(A.hossz==0) {
L("Hátultesztelő fix darabszámú lefutó ciklust nullaszer próbálsz végrehajtani! Ciklustípus: \"!\"", Pozíció:
%lu",f.P);
EXITFAILURE();
}
m = new MAUSTRING;
*m=A;

f.CIKLUSVEREM=f.P; // Elementjük a verembe a programmutatót
f.CIKLUSVEREM=(unsigned long long) 0; // Elementjük a verembe a stringpozíció kezdeti értékét
f.CIKLUSSTRINGVEREM=(unsigned long long)m;
return 0; // Belépünk a ciklusmagba
}
// -----
int felkialtojel_csukokapcsos(F& f) { // Visszatérés fix darabszámú ciklusból
unsigned long long darabszam;USIL regip;unsigned int stringhossz;
MAUSTRING *m;unsigned long long longm;
darabszam=f.CIKLUSVEREM; // Kiolvassuk a darabszámot a veremből
regip=f.CIKLUSVEREM; // Kiolvassuk a visszatérési címet a veremből
longm=f.CIKLUSSTRINGVEREM; // Kiolvassuk a másik veremből a stringmutatót
m=(MAUSTRING *)longm;
stringhossz=(m).hossz; // A string hossza
if(f.p[f.P]=='(') { // Ha a közvetlenül következő karakter nyitózároljel,
if(ERTEKunsignedchar(f)) {delete m; return 0;} // akkor kiolvassuk a feltételt, s ha az igaz, kilépünk a ciklusból
}
darabszam++;
if(darabszam==stringhossz) {delete m;return 0;} // Ha nulla, kilépünk a ciklusból
f.CIKLUSVEREM=regip; // Visszatesszük a visszatérési címet a verembe
f.CIKLUSVEREM=darabszam; // Visszatesszük a darabszámot a verembe
f.CIKLUSSTRINGVEREM=longm; // Visszatesszük a stringmutatót a másik verembe
f.P=regip; // Programmutató beállítása a visszatérési címre
return 0; // Visszaugrás a ciklusba
}
```

Természetesen a **?!** függvényünk a ciklus belsejében itt is a ciklusváltozó aktuális értékét adja vissza, ami ugye egy érvényes index a stringkifejezés valamely karakterére, de ez most nullától növekszik folyamatosan.

Ezzel felvértezve, maudir programunk immár így néz ki:

```
#!/mau // maudir program.
// Készítette: Viola Zoltán, violazoli@gmail.com
// A program a "mau" nevű programozási nyelven íródott, azt is én Viola Zoltán készítettem.
// Mindegyik GPL licenc alatt van.

// Konstansok meghatározása

#s@P="/tmp"; // Annak a könyvtárnak a PATH-ja, ahova az elkészült ideiglenes fájl kerül, az,
// amit a $EDITOR változóban meghatározott texteditorral szerkesztünk majd.
#c@c='-'; // A pidstring üres helyeit kitöltő karakter
#c@p=6; // A pidstring maximális hossza
#s@i="MAUDIR_temporary_file"; // Az ideiglenes file alapneve a pidstring nélkül
#s@k=".tmp"; // Az ideiglenes file kiterjesztése

#s@Z="DRL"; // Ezekkel a típusú tartalomjegyzék-bejegyzésekkel foglalkozunk a maudir programban. És ebben a sorrendben
// fognak megjelenni a szerkesztendő ideiglenes fájlban felülről lefelé:
// D azaz Directoryk,
// R azaz Reguláris (közönséges) fájlok
// L azaz Linkek.

// Előkészítő műveletek
#s@p=?s "PID" @p @c; // A pidstring előállítás
#s@n=(@i)+"_PID-"+(@p)+(@k); // A leendő ideiglenes file nevének előállítása

// A leendő ideiglenes fájl teljes útvonalat tartalmazó nevének előállítása:
if((#s@P[!#s@P-1])!='/') T #s@N=(@P)+"/"+ (@n);
E #s@N=(@P)+(@n);
// A fenti két sor azt csinálja, hogy ellenőrizzük, e progi legelején az ideiglenes fájl helyének megjelölt könyvtár
útvonala
// egy "/" jellel végződik-e. Ha ez nincs így, pótoljuk e per-jelet.

if (?a)<3 T #s@A="."; // Ha nem adtunk meg parancssori paramétert, az aktuális könyvtárra vonatkozik a hívás,
E #s@A=?s "ARGV" 2; // különben a megadott tartalomjegyzékre

// A tartalomjegyzék beolvasása
```

```

#T@t= @A; // A lekérdezendő tartalomjegyzéket jelölő stringet a #s@A tartalmazza. Itt olvassuk be a directoryt.

// A szerkesztendő ideiglenes fájl megnyitása írásra
#K@o=@N; // Megnyitjuk az output file-ot. Ennek neve természetesen a korábban meghatározott string lesz,
// ami a PID értékét is tartalmazza. A @N változó e nevet tartalmazza, az elején kiegészítve a könyvtárat tartalmazó
// elérési útvonallal.

if #K@o E "Az output file nem megnyitható!\n" XX // Hiba esetén kilépünk.
// Figyeljük meg, a fenti if utasítás nem is tartalmaz THEN ágat... Így egyszerűbb, nem bajlódunk az összehasonlító
// művelet eredményének negálásával.

if ((#s@A[(!#s@A)-1])!='/') T #s@A=(@A)+"/"; // Itt ellenőrizzük, e progi legelején a beolvasandó tartalomjegyzék
útvonalának
// vége egy "/" jel-e. Ha ez nincs így, pótoljuk e per-jellet. Ha netán nem lett volna ott, az nem baj a tartalomjegyzék
// beolvasásakor még, mert ezt akkor pótolja az interpreter beolvasórutinja maga is nagy előzékenyen. De most ezt nekünk
is
// meg kell csinálni.

{! #s@Z; if (#l (?#l "T#" t #s@Z[?])>0) T [#c@(#s@Z[?])=?#l "T#" t (#s@Z[?])];
!} // E ciklusban foglalunk le memóriát annyi darab unsigned char értéknek a megfelelő nevű tömbbe,
// ahány tartalomjegyzéki bejegyzés van abból a típusból.

{! #s@Z; („Kiíró rutin” (#s@Z[?]));
!} // A kiíró függvény meghívása ciklusban. Ez egy fix számszor lefutó ciklus, és indexelésre a ciklusváltozót
// használjuk, amire a ?|

[#K@o]; // Lezárom az output file-ot

SY {#s "ENV" "EDITOR"} + " " + @N; // Megnyitjuk a fájlt a $EDITOR környezeti változó által meghatározott text
editorral

#B@b=@N; // Megnyitjuk a szerkesztett fájlt input fájlként.
if #B@b E "Az ideiglenes fájl nem megnyitható beolvasásra!\n" XX

{( // beolvasóciklus kezdete. Végtelenciklus.
#s@s=?s "NL", b, 1000; // Beolvassuk a sort, de max. 1000 karaktert
if(!#s@s)==0 TT »$tt; // Ha üres sort olvastunk be, vége a fájlnek, ekkor elugrunk a törlésellenőrzési részhez
// Itt kezdődik a sor feldolgozása.
#c@t=#s@s[0]; // A file típusa
#s@h=[2,11]@s; // A file sorszáma stringként
#l@h=#s ?s "SP" @h; // Átalakítjuk a file sorszámát stringből unsigned int számmá. Ehhez előbb le kell vágni
// a string elejéről a szóközöket. Ezt végzi el az iménti "SP" függvény: levág minden whitespace karaktert a string
elejéről.
#s@a=[16,]@s; // A file neve
#s@a=(#s@a) - 1; // A hosszából le kell vonnunk 1-et, mert másképp a file nevéhez tartozónak tartaná a 10-es karakterkódú
// sorvégjelet is.

#c@(@t)[#l@h]=1; // Ebben a sorban azt vizsgáljuk, milyen típusú fájlról szól a beolvasott sor.
// Amilyen típusú, az ahhoz tartozó tömb megfelelő sorszámu elemét 1-re állítjuk. Ez amiatt
// fontos, mert így tudjuk követni, nincs-e kitörölve a fájlból az adott sorszámu fájl sora.
// Ha ugyanis azt kitörölte a Felhasználó, majd törölni kell azt a fájlt vagy könyvtárat is!

if (#s {#s "Tn" t, #l@h, (@t)} != (#s@a)) T SY "mv " + (@A) + ({#s "Tn" t, #l@h, (@t)} + " " + (@A) + (@a);
// Fentebb azt csináltuk, hogy ellenőrizzük, a megfelelő típusú és sorszámu fájl neve a beolvasott tartalomjegyzék-
struktúrában
// azonos-e azzal, amit épp beolvastunk az ideiglenes fájl sorából. Ha NEM azonos vele, meghívjuk a SY
rendszerparanccsal
// a shell "mv" átnevező funkcióját, s átnevezzük az új névre.
}) // Visszaugrás a beolvasóciklus elejére

$tt

[#B@b]; // Bezárjuk az ideiglenes fájlt.
SY "rm " + @N; // Töröljük az ideiglenes fájlt a shell segítségével

{! #s@Z; („Töröl rutin” (#s@Z[?]));
!} // A töröl függvény meghívása ciklusban

XX // A program vége

// ===== Függvények

„Töröl rutin” // Törli a megfelelő fájlbejegyzést
// Input paraméterek:
#c@t // típus: D: directory, R: reguláris fájl, L: link

if (#l (?#l "T#" ^ t @t)==0) T xx; // Ha nincs a megfelelő típusú fájlból, vége a rutinnak.

if (@t)==D T #s@y="rm -rf ";
E #s@y="rm ";

{! ?#l "T#" ^ t @t ; // Fix számszor lefutó ciklus, annyiszor fut le ahány fájlbejegyzésünk van.
if #c@^(@t)[?] -1 E SY (@y) + (@A) + ({#s "Tn" ^ t, ?| -1, @t)
!}

xx

```

```

„Kiíró rutin” // Kiírja a megfelelő sort a fájlba
// Input paraméterek:
#c@t // típus: D: directory, R: reguláris fájl, L: link

#s@y=" "; #s@y[0]=@t;
#l@i=0; // Ciklusszámláló kezdeti értéke
if (#l (?#l "T#" ^ t @t)==0) T xx; // Ha nincs megfelelő típusú fájl, nem ír ki azokról semmi adatot. Nem hülye ő...
[| ?#l "T#" ^ t @t; // Ciklus indul, annyiszor fut le, ahány fájl van az adott típusból. Legalább egy egészen biztosan
akad.
<s @^o (@y)+(#l@i)+" = "+(?#s "Tn" ^ t, #l@i, @t)+"\n"; // A sor kiírása
#l++; // Ciklusszámláló növelése
|} // A ciklus vége
xx

```

Elismerem, nem lett sokkal rövidebb, ellenben szerintem sokkal szebb, és határozottan áttekinthetőbb, világosabb, letisztultabb...

Látjuk azonban azt is, hogy a `!` típusú ciklusainkban a ciklusváltozót, ami a string indexe, jelen programunkban kizárólag épp arra használjuk, hogy indexelje a stringet, azaz a segítségével „előcsalogassuk” a string aktuális karakterét! Vélhető, hogy legtöbbször valóban épp ez is lesz a szerepe más programjainkban is. Na már most, ezesetben a string nevét bizony többször írjuk le, ami a programot áttekinthetetlené teszi, helypocsékolóvá is mert nő a forráskód mérete, s nyilván lassabb is lesz a futása. Miért is ne lehetne valami rendszerfüggvényünk, ami visszaadja a ciklus aktuális karakterét a stringből?! Íme:

```

unsigned char stringciklus_aktkar(F& f) { // Visszaadja az aktuális karaktert egy ! ... ! típusú
// ciklus belsejében
// Szintaxis: "!"
MAUSTRING *m; unsigned long long longm; unsigned long long darabszam;
darabszam=f.CIKLUSVEREM; // Kiolvassuk a darabszámot a veremből
f.CIKLUSVEREM=darabszam; // Vissza is tesszük gyorsan a darabszámot a verembe azon melegében, nehogy elfelejssük
longm=f.CIKLUSSTRINGVEREM; // kiolvassuk a ciklusstringveremből a stringmutatót
f.CIKLUSSTRINGVEREM=longm; // Vissza is tesszük gyorsan a stringmutatót a verembe azon melegében, nehogy elfelejssük

m=(MAUSTRING *)longm; // Átkonvertáljuk a mutatót a megfelelő típusú pointerre
return m->s[darabszam];
}

```

Igazán nem sok... És nézzük csak meg, mennyire leegyszerűsödtek a maudir programunkban ezek a ciklusaink:

```

.....
{! #s@Z; if (#l (?#l "T#" t, ?#c "{!}">0) T [#c@(?#c "{!}"=?#l "T#" t, ?#c "{!}"];
!} // E ciklusban foglalunk le memóriát annyi darab unsigned char értéknek a megfelelő nevű tömbbe,
// ahány tartalomjegyzéki bejegyzés van abból a típusból. Ez egy fix számszor lefutó ciklus, annyiszor fut le,
// amennyi a string hossza. A ?#c "{!}" adja vissza a string aktuális karakterét.

{! #s@Z; („Kiíró rutin” ?#c "{!}");
!} // A kiíró függvény meghívása ciklusban. Ez egy fix számszor lefutó ciklus, annyiszor fut le amennyi a string hossza.
.....

{! #s@Z; („Törlő rutin” ?#c "{!}");
!} // A törlő függvény meghívása ciklusban
.....

```

Van azonban egy kis baj. Az, hogy a Felhasználó rémségesen aljas és ravasz! Plusz teljesen idióta is. Ezt minden programozó tudja, kivétel nélkül. Vagyis, mi van ha a mi szépséges maudir programunk használata közben képes átírni a sor első karakterét valami olyasmire, ami nem a D, R vagy L betűk valamelyike?! Mit csinál ekkor a program, érvénytelen bejegyzéstípus esetén?

Ezt írja ki:

LOG:> 2014.03.09 00:36:43 : Indexhatárlépés tömb esetében!

Na most, ez nem igazán informatív üzenet... Azért nem, mert ezt az interpreter kreálja, és nem a maudir program. Illene valahogy leellenőrizni a típusazonosítót a maudir programban... Nos igen, simán megtehetnénk egy ciklusban, ami addig megy ahány karakter hosszú a **#s@Z** stringünk, és...

Na de álljon meg a menet! Rendkívül gyakoriak lesznek a programokban az olyan feladatok, hogy ellenőrizni kell, szerepel-e egy karakter egy stringben, s ha igen, az hányadik karakter. Miért is ne automatizálhatnánk e feladatot?! Sőt, mi mindenfajta tömbben akarunk keresni tudni automatikusan! Meg lehetne-e ezt oldani kényelmesen?!

Hogy a csudába ne! Íme:

```
int kerdojelkerdojel(F& f) { // Speciális összehasonlító utasítás, tömbökben keresésre.
// Szintaxis:
// ?? c S x
// ahol c egy unsigned char érték, az S pedig egy stringkifejezés, x pedig egy unsigned int érték.
// Amennyiben a c karaktert megtalálja az S stringben, annak x-edik karakterétől a string végéig levő tartományban,
// akkor az f.kerdojelkerdojelpozicio értékét beállítja arra a számra ahol a stringben megtalálta. Ez a szám
// abszolút értékű, a string elejétől számítódik! Ezesetben ugyanakkor az f.kerdojelkerdojel flag értéke 1 lesz.
// Ha nem találta meg a stringben a c karaktert, vagy x>S.hossz, akkor a pozíciószám nulla lesz,
// ellenben a flag értéke is nulla.
// Ha ebben a formában adjuk meg:
// ?? c S;
// akkor az x értékének a nullát tekinti automatikusan. Ne feledjük, hogy ezesetben MUSZÁJ lezárni az
// utasításunkat egy pontosvesszővel!
// ??#t c T x
// A fenti változat egy TÖMBBEN keres. A T tömbben. Ez tehát itt már nem lehet KIFEJEZÉS, csak tömbváltozó!
// A T itt a tömb nevét meghatározó unsigned char KIFEJEZÉS... Az x szintén egy unsigned int érték,
// a tömb ennyiedik elemétől keres. A c adatot keresi, amely olyan típusú, amilyen típusra a t karakter utal.
// A # jel közvetlenül a ?? után kell álljon, nem lehet köztük whitespace, s hasonlóképpen a t karakternek
// is közvetlenül a # után kell állnia. A t karakter lehetséges értékei:
// c,C,i,I,l,L,g,G,d,D,f,s
// Megengedett a ??#t c ^ T x alak is, ekkor a szülő névtér tömbjében keres.
// Ha ebben a formában adjuk meg:
// ??#t c T;
// akkor az x értékének a nullát tekinti automatikusan. Ne feledjük, hogy ezesetben MUSZÁJ lezárni az
// utasításunkat egy pontosvesszővel!

unsigned char tipus; unsigned char c; MAUSTRING S; unsigned int x; unsigned int i; unsigned char T;
F *ff; ff=&f; V v;

if(f.p[f.P]=='#') {tipus=k(f);k(f);

switch(tipus) { // itt olvassuk be a keresendő adatot
case 's': S=ERTEKstring(f);break;
case 'c': v.c[0]=ERTEKunsignedchar(f);break;
case 'C': v.C[0]=ERTEKsignedchar(f);break;
case 'i': v.i[0]=ERTEKunsignedshortint(f);break;
case 'I': v.I[0]=ERTEKsignedshortint(f);break;
case 'l': v.l[0]=ERTEKunsignedint(f);break;
case 'L': v.L[0]=ERTEKsignedint(f);break;
case 'g': v.g[0]=ERTEKunsignedlonglong(f);break;
case 'G': v.G[0]=ERTEKsignedlonglong(f);break;
case 'f': v.f[0]=ERTEKfloat(f);break;
case 'd': v.d[0]=ERTEKdouble(f);break;
case 'D': v.D[0]=ERTEKlongdouble(f);break;
default: L("Syntax error: ércénytelen típusazonosító a ??# rész után! A helytelen karakter: %c.\n"
"Pozíció: %lu",tipus,f.P);ERROR(f,3);break;
} // switch vége
namespace(f);if(f.p[f.P]=='^') {ff=f.parentF;k(f);}
T=ERTEKunsignedchar(f);namespace(f);
if(f.p[f.P]==';'') {x=0;} else {x=ERTEKunsignedint(f);}

switch(tipus) {
case 's': if(x>=ff->t.t[T].smax) {goto kerdojelkerdojel_nincsbenne;}
for(i=x;i<ff->t.t[T].smax;i++) {if(S==ff->t.t[T].s[i]) {f.kerdojelkerdojelpozicio=i;f.kerdojelkerdojelflag=1;return 0;}
} break; // for i vége
case 'c': if(x>=ff->t.t[T].cmax) {goto kerdojelkerdojel_nincsbenne;}
for(i=x;i<ff->t.t[T].cmax;i++) {if(v.c[0]==ff->t.t[T].c[i]) {f.kerdojelkerdojelpozicio=i;f.kerdojelkerdojelflag=1;return
0;}}
} break; // for i vége
case 'C': if(x>=ff->t.t[T].Cmax) {goto kerdojelkerdojel_nincsbenne;}
for(i=x;i<ff->t.t[T].Cmax;i++) {if(v.C[0]==ff->t.t[T].C[i]) {f.kerdojelkerdojelpozicio=i;f.kerdojelkerdojelflag=1;return
0;}}
} break; // for i vége
case 'i': if(x>=ff->t.t[T].imax) {goto kerdojelkerdojel_nincsbenne;}
```



```

for(i=x;i<ff->t.t[T].imax;i++) {if(v.i[0]==ff->t.t[T].i[i]) {f.kerdojelkerdojelpozicio=i;f.kerdojelkerdojelflag=1;return 0;}}
} break; // for i vége
case 'I': if(x>=ff->t.t[T].Imax) {goto kerdojelkerdojel_nincsbenne;}
for(i=x;i<ff->t.t[T].Imax;i++) {if(v.I[0]==ff->t.t[T].I[i]) {f.kerdojelkerdojelpozicio=i;f.kerdojelkerdojelflag=1;return 0;}}
} break; // for i vége
case 'l': if(x>=ff->t.t[T].Lmax) {goto kerdojelkerdojel_nincsbenne;}
for(i=x;i<ff->t.t[T].Lmax;i++) {if(v.l[0]==ff->t.t[T].l[i]) {f.kerdojelkerdojelpozicio=i;f.kerdojelkerdojelflag=1;return 0;}}
} break; // for i vége
case 'L': if(x>=ff->t.t[T].Lmax) {goto kerdojelkerdojel_nincsbenne;}
for(i=x;i<ff->t.t[T].Lmax;i++) {if(v.L[0]==ff->t.t[T].L[i]) {f.kerdojelkerdojelpozicio=i;f.kerdojelkerdojelflag=1;return 0;}}
} break; // for i vége
case 'g': if(x>=ff->t.t[T].gmax) {goto kerdojelkerdojel_nincsbenne;}
for(i=x;i<ff->t.t[T].gmax;i++) {if(v.g[0]==ff->t.t[T].g[i]) {f.kerdojelkerdojelpozicio=i;f.kerdojelkerdojelflag=1;return 0;}}
} break; // for i vége
case 'G': if(x>=ff->t.t[T].Gmax) {goto kerdojelkerdojel_nincsbenne;}
for(i=x;i<ff->t.t[T].Gmax;i++) {if(v.G[0]==ff->t.t[T].G[i]) {f.kerdojelkerdojelpozicio=i;f.kerdojelkerdojelflag=1;return 0;}}
} break; // for i vége
case 'f': if(x>=ff->t.t[T].fmax) {goto kerdojelkerdojel_nincsbenne;}
for(i=x;i<ff->t.t[T].fmax;i++) {if(v.f[0]==ff->t.t[T].f[i]) {f.kerdojelkerdojelpozicio=i;f.kerdojelkerdojelflag=1;return 0;}}
} break; // for i vége
case 'd': if(x>=ff->t.t[T].dmax) {goto kerdojelkerdojel_nincsbenne;}
for(i=x;i<ff->t.t[T].dmax;i++) {if(v.d[0]==ff->t.t[T].d[i]) {f.kerdojelkerdojelpozicio=i;f.kerdojelkerdojelflag=1;return 0;}}
} break; // for i vége
case 'D': if(x>=ff->t.t[T].Dmax) {goto kerdojelkerdojel_nincsbenne;}
for(i=x;i<ff->t.t[T].Dmax;i++) {if(v.D==ff->t.t[T].D[i]) {f.kerdojelkerdojelpozicio=i;f.kerdojelkerdojelflag=1;return 0;}}
} break; // for i vége
} // switch vége

goto kerdojelkerdojel_nincsbenne;
} // # teszt vége
// Itt jön az a rész, ami stringben keresi a karaktert:
c=ERTEKunsignedchar(f);
S=ERTEKstring(f);namespace(f);
if(f.p[f.P]=='\;') {x=0;} else {x=ERTEKunsignedint(f);}
if(x>=S.hossz) goto kerdojelkerdojel_nincsbenne;
for(i=x;i<S.hossz;i++) {
if(S.s[i]==c) {f.kerdojelkerdojelpozicio=i;f.kerdojelkerdojelflag=1;return 0;}
} // for i vége
kerdojelkerdojel_nincsbenne:
f.kerdojelkerdojelpozicio=0;f.kerdojelkerdojelflag=0;return 0;
}

```

Nem, ez nem nagyon hosszú rutin, csak temérdek benne a kommentsor. Ott van az elején, hogyan kell használni, emiatt ezt nem is kell kirészletezni külön.

Természetesen kell valami módszer arra is, egy efféle keresés lefuttatása után, hogy lekérdezzük, hányadik pozíción találta meg a drága a keresett adatot, illetve, hogy egyáltalán megtalálta-e! Nos, a pozíció lekérdezésére szolgál a

?

rendszerfüggvény. Használata egyszerű, mint azt itt láthatjuk:

"Tömb pozíció: " ?l ?.; /;

Tehát: először lefuttatjuk a ?? paranccsal a keresést, majd azután amíg csak újabb keresést nem futtatunk le, a ? függvény az utolsó keresés eredményét adja vissza nekünk. Ennek eredményét különben egy külön mezőben tárolja az F struktúrában, aminek a neve **f.kerdojelkerdojelpozicio**. Amennyiben a keresés sikertelen volt mert nem találta meg a megadott adatot, akkor ennek értéke nulla. Ez megtévesztő lehet, mert hogy is különböztessük meg akkor ezt az esetet attól, amikor nagyonis megtalálta az elemet, de a tömb nulladik pozícióján? Nos erre szolgál az **f.kerdojelkerdojelflag** mező, aminek értéke 0 ha nem találta meg, és 1 ha megtalálta. Ezt természetesen a ?? függvénnyel kérdezhetjük le, mint unsigned int számot.

Igen ám, de ha már ott az az érték, miért kéne külön összehasonlítani az **if** utasítással, azért, hogy utána ugorhassunk az eredménytől függően?! Messze okosabb, ha kreálunk valami speciális ugróutasítást, ami ugyanúgy működik mint a T,TT,E,EE — csak éppen nem az f.ifflag állapotát veszi figyelembe hanem ezen másik flagét!

Ezek az új ugróutasításaink legjobb ha pontosan ugyanazt a nevet viselik mint a régiek, csak bővítjük ki a nevüket egyetlen karakterrel - legyen ez a "." azaz a pont karakter! Alakjuk tehát:

T.
TT.
E.
EE.

Ezek után betehetünk egy efféle ellenőrzést a maudir programunkba (zölddel kiemelve a pótlólag beszúrt részt):

```
.....
#s@a=(#s@a) - 1; // A hosszából le kell vonnunk 1-et, mert másképp a file nevéhez tartozónak tartaná a 10-es karakterkódú
// sorvégjelet is.

?? @t @Z; E. "Ismeretlen bejegyzéstípus az ideiglenes fájlban!\n"
E. "Esetleg kitörölted vagy megváltoztattad a sor első karakterét?\n"
E. "A hibás típusazonosító karakterkódja: " ?c @t; " Maga a karakter: " ? @t; /; XX;

#c@(@t)[#l@h]=1; // Ebben a sorban azt vizsgáljuk, milyen típusú fájlról szól a beolvasott sor.
.....
```

Így már mégiscsak profibbnak tűnik a progi...

Egy rövid példaprogram annak bemutatására, miként keresgélhetünk tömbökben meg stringekben értékeket:

```
#s@S="abcdefgh";

?? c @S;
"Pozíció: " ?l ?.; /;

[@t[[4]]]; // lefoglalok területet 4 string részére egy t nevű tömbbe
#s@t="Ez egy sina string, nem tömb!\n"
// Feltöltöm értékekkel a tömböt
#s@t[[0]]="kutya";
#s@t[[1]]="macska";
#s@t[[2]]="vadbarom";
#s@t[[3]]="cica";
"Kiíratom a 4 elemű stringtömb mindegyik értékét:\n"
#i@i=0; { | 4 ?i @i; ". = " ?s @t[[#i@i]]; /; #i++@i; |}

??#s "vadbarom" t;

"Tömb pozíció: " ?l ?.; /;

[#c@t=4]; // lefoglalok területet 4 unsigned char szám részére egy t nevű tömbbe
#c@t[0]=10;
#c@t[1]=11;
#c@t[2]=12;
#c@t[3]=13;

"Kiíratom a 4 elemű unsigned char tömb mindegyik értékét:\n"
#i@i=0; { | 4 ?i @i; ". = " ?c @t[[#i@i]]; /; #i++@i; |}

??#c 11 t 8;

"Tömb pozíció: " ?l ?.; /;
```

A futási eredmény:

```
Pozíció: 2
Kiíratom a 4 elemű stringtömb mindegyik értékét:
0. = kutya
1. = macska
```

```

2. = vadbarom
3. = cica
Tömb pozíció: 2
Kiíratom a 4 elemű unsigned char tömb mindegyik értékét:
0. = 10
1. = 11
2. = 12
3. = 13
Tömb pozíció: 0

```

Igenám, de e maudir programot merészelttem elkezdni HASZNÁLNI is... Hát igen, elég nagy önbizalom kell hozzá, hogy az Embör Gyermeke a napi „éles” munkája során elkezdjén használni egy olyan bit-tákolmányt, amit ráadásul egy önmaga által összeeszkábált programnyelvben mesterkedett össze... Hiszen ilyenkor ugyebár, a hibák lehetősége hatványozódik!

Nos, hibát éppenséggel nem találtam, ám elszörnyedve tapasztaltam, hogy az én drága maudir programom egyszerűen nem működik, ha olyan könyvtár- vagy fájlnevekkel kell dolgoznia, amelyek szóközöket tartalmaznak, vagy netán aposztrófot, vagy... vagy még egy csomó más spéci karakter esetén se működik jól. Megfontolva ennek okát, arra a következtetésre kellett jussak, hogy ez amiatt történik így, mert a shell útján törölök illetve nevezek át, ami sajna bizonyos karaktereket különlegesen kezel, a szóközről meg egyenesen azt hiszi, paraméter-szeperator...

Első ötletem az volt, hogy a megfelelő útvonalneveket végigvizsgálom, s ha azokban efféle karakter szerepel, eléje szúrom a „\” jelet... Kissé jobban belegondolva azonban ezen ötletembe, inkább sürgősen kivertem a fejemből. Fenesok vizsgálgatást igényelt volna, lassú lett volna, és LUSTA is voltam ezt leprogramozni. Aztán meg az is eszembe jutott, ha a shellt sikerülne megkerülnöm, az egész mindenség gyorsabban is működne.

Nos, végül csináltam pár remek parancsot a mau nyelvemhez, íme:

```

int fuggveny_REMOVE(F& f) { // Fájl törlése
// Szintaxis:
// RM s
// ahol az s egy MAUSTRING.
MAUSTRING s;
s=ERTEKstring(f);
remove((char *)s.s);
return 0;
}
// -----
int fuggveny_RENAME(F& f) { // Fájl vagy könyvtár átnevezése
// Szintaxis:
// RN regineve ujneve
// ahol a regineve és az ujneve egy-egy MAUSTRING.
MAUSTRING regineve; MAUSTRING ujneve;
regineve=ERTEKstring(f);ujneve=ERTEKstring(f);
rename((char *)regineve.s,(char *)ujneve.s);
return 0;
}
// -----
int fuggveny_REMOVEDIRECTORY(F& f) { // Könyvtár törlése rekurzívan
// Szintaxis:
// RD s
// ahol az s egy MAUSTRING.
MAUSTRING s;s=ERTEKstring(f);remove_directory((char *)s.s);
return 0;
}
// -----
int remove_directory(const char *path) { // Segédfüggvény könyvtár rekurzív törléséhez
DIR *d = opendir(path);
size_t path_len = strlen(path);
int r = -1;

if (d)
{
    struct dirent *p;

```

```

    r = 0;

    while (!r && (p=readdir(d)))
    {
        int r2 = -1;
        char *buf;
        size_t len;

// Átugorjuk a "." és a ".." bejegyzéseket.
        if (!strcmp(p->d_name, ".") || !strcmp(p->d_name, ".."))
        {
            continue;
        }

        len = path_len + strlen(p->d_name) + 2;
        buf = (char *)malloc(len);

        if (buf)
        {
            struct stat statbuf;

            snprintf(buf, len, "%s/%s", path, p->d_name);

            if (!stat(buf, &statbuf))
            {
                if (S_ISDIR(statbuf.st_mode))
                {
                    r2 = remove_directory(buf);
                }
                else
                {
                    r2 = unlink(buf);
                }
            }

            free(buf);
        }

        r = r2;
    }

    closedir(d);
}

if (!r)
{
    r = rmdir(path);
}

return r;
}

```

A fentebbi funkciókba bele van írva a használat szintaxisa, így külön nem is rágom szájba azt újra, ehelyett bemutatom a (jelen időpillanat szerinti) legújabb maudir programot. Ebben látható már olyan trükk is, amikor szubrutint címke HELYETT egy változó aktuális tartalma alapján hívunk meg. A változó tartalma természetesen egy címke értéke, de az hogy épp melyik címkéé, a futás közben dől el... Szóval íme:

```

#!mau // maudir program.
// Készítette: Viola Zoltán, violazoli@gmail.com
// A program a "mau" nevű programozási nyelven íródott, azt is én Viola Zoltán készítettem.
// Mindegyik GPL licenc alatt van.

// Konstansok meghatározása

#s@P="/tmp"; // Annak a könyvtárnak a PATH-ja, ahova az elkészült ideiglenes fájl kerül, az,
// amit a $EDITOR változóban meghatározott texteditorral szerkesztünk majd.
#c@p='-'; // A pidstring üres helyeit kitöltő karakter
#p=6; // A pidstring maximális hossza
#s@i="MAUDIR_temporary_file"; // Az ideiglenes file alapneve a pidstring nélkül
#s@k=".tmp"; // Az ideiglenes file kiterjesztése

#s@Z="DRL"; // Ezekkel a típusú tartalomjegyzék-bejegyzésekkel foglalkozunk a maudir programban. És ebben a sorrendben
// fognak megjelenni a szerkesztendő ideiglenes fájlban felülről lefelé:
// D azaz Directoryk,
// R azaz Reguláris (közönséges) fájlok
// L azaz Linkek.

// Előkészítő műveletek
#s@p=?s "PID" @p @c; // A pidstring előállítás
#s@n=(@i)+"_PID-"+(@p)+(@k); // A leendő ideiglenes file nevének előállítása

```

```

// A leendő ideiglenes fájl teljes útvonalat tartalmazó nevének előállítás:
if((#s@P[(!#s@P)-1])!='/) T #s@N=(@P)+"/"+ (@n);
E #s@N=(@P)+(@n);
// A fenti két sor azt csinálja, hogy ellenőrizzük, e progi legelején az ideiglenes fájl helyének megjelölt könyvtár
útvonala
// egy "/" jellel végződik-e. Ha ez nincs így, pótoljuk e per-jelet.

if (?a)<3 T #s@a="."; // Ha nem adtunk meg parancssori paramétert, az aktuális könyvtárra vonatkozik a hívás,
E #s@a=?#s "ARGV" 2; // különben a megadott tartalomjegyzékre

// A tartalomjegyzék beolvasása

#T@t= @A; // A lekérdezendő tartalomjegyzéket jelölő stringet a #s@a tartalmazza. Itt olvassuk be a directoryt.

// A szerkesztendő ideiglenes fájl megnyitása írásra
#K@o=@N; // Megnyitjuk az output file-ot. Ennek neve természetesen a korábban meghatározott string lesz,
// ami a PID értékét is tartalmazza. A @N változó e nevet tartalmazza, az elején kiegészítve a könyvtárat tartalmazó
// elérési útvonallal.

if #K@o E "Az output file nem megnyitható!\n" XX // Hiba esetén kilépünk.
// Figyeljük meg, a fenti if utasítás nem is tartalmaz THEN ágat... Így egyszerűbb, nem bajlódunk az összehasonlító
// művelet eredményének negálásával.

if ((#s@a[(!#s@a)-1])!='/) T #s@a=(@A)+"/"; // Itt ellenőrizzük, e progi legelején a beolvasandó tartalomjegyzék
útvonalának
// vége egy "/" jel-e. Ha ez nincs így, pótoljuk e per-jelet. Ha netán nem lett volna ott, az nem baj a tartalomjegyzék
// beolvasásakor még, mert ezt akkor pótolja az interpreter beolvasórutinja maga is nagy előzékenyen. De most ezt nekünk
is
// meg kell csinálni.

{! #s@Z; if (#l (?#l "T#" t, ?#c "{!}")>0) T [#c@(?#c "{!}")=?#l "T#" t, ?#c "{!}"];
!} // E ciklusban foglalunk le memóriát annyi darab unsigned char értéknek a megfelelő nevű tömbbe,
// ahány tartalomjegyzéki bejegyzés van abból a típusból. Ez egy fix számszor lefutó ciklus, annyiszor fut le,
// amennyi a string hossza. A ?#c "{!}" adja vissza a string aktuális karakterét.

{! #s@Z; („Kiíró rutin” ?#c "{!}"); !} // A kiíró függvény meghívása ciklusban.
// Ez egy fix számszor lefutó ciklus, annyiszor fut le amennyi a string hossza.
// A ?#c "{!}" adja vissza a string aktuális karakterét.

[#K@o]; // Lezárom az output file-ot

SY (?#s "ENV" "EDITOR") + " " + @N; // Megnyitjuk a fájlt a $EDITOR környezeti változó által meghatározott text
editorral

#B@b=@N; // Megnyitjuk a szerkesztett fájlt input fájlként.
if #B@b E "Az ideiglenes fájl nem megnyitható beolvasásra!\n" XX

{( // beolvasóciklus kezdete. Végtelenciklus.
#s@s=?#s "NL", b, 1000; // Beolvassuk a sort, de max. 1000 karaktert
if(!#s@s)==0 TT »$tt; // Ha üres sort olvastunk be, vége a fájlnek, ekkor elugrunk a törlésellenőrzési részhez
// Itt kezdődik a sor feldolgozása.
#c@t=#s@s[0]; // A file típusa
#s@h=[2,11]@s; // A file sorszáma stringként
#l@h=#s ?#s "SP" @h; // Átalakítjuk a file sorszámát stringből unsigned int számmá. Ehhez előbb le kell vágni
// a string elejéről a szóközöket. Ezt végzi el az iménti "SP" függvény: levág minden whitespace karaktert a string
elejéről.
#s@a=[16,]@s; // A file neve
//#s@a=(#s@a) - 1; // A hosszából le kell vonnunk 1-et, mert másképp a file nevéhez tartozónak tartaná a 10-es
karakterkódú
// sorvégjelet is.
#s--@a; // A hosszából le kell vonnunk 1-et, mert másképp a file nevéhez tartozónak tartaná a 10-es karakterkódú
// sorvégjelet is.

?? @t @Z; E. "Ismeretlen bejegyzéstípus az ideiglenes fájlban!\n"
E. "Esetleg kitörölt vagy megváltoztattad a sor első karakterét?\n"
E. "A hibás típusazonosító karakterkódja: " ?c @t; " Maga a karakter: " ? @t; /; XX;

#c@(@t)[#l@h]=1; // Ebben a sorban azt vizsgáljuk, milyen típusú fájlról szól a beolvasott sor.
// Amilyen típusú, az ahhoz tartozó tömb megfelelő sorszámú elemét 1-re állítjuk. Ez amiatt
// fontos, mert így tudjuk követni, nincs-e kitörölve a fájlból az adott sorszámú fájl sora.
// Ha ugyanis azt kitörölte a Felhasználó, majd törölni kell azt a fájlt vagy könyvtárat is!

if (#s (?#s "Tn" t, #l@h, (@t)) != (#s@a)) T RN (@A) + (?#s "Tn" t, #l@h, (@t)), (@A) + (@a);
// Fentebb azt csináltuk, hogy ellenőrizzük, a megfelelő típusú és sorszámú fájl neve a beolvasott tartalomjegyzék-
struktúrában
// azonos-e azzal, amit épp beolvastunk az ideiglenes fájl sorából. Ha NEM azonos vele, átnevezzük.
}) // Visszaugrás a beolvasóciklus elejére

$tt

[#B@b]; // Bezárjuk az ideiglenes fájlt.
SY "rm " + (@N); // Töröljük az ideiglenes fájlt a shell segítségével

{! #s@Z; („Töröl rutin” ?#c "{!}"); !} // A töröl függvény meghívása ciklusban

XX // A program vége

```

```
// ===== Függvények

„Töröl rutin” // Törli a megfelelő fájlbejegyzést
// Input paraméterek:
#c@t // típus: D: directory, R: reguláris fájl, L: link

if (#l (?#l "T#" ^ t @t)==0) T xx; // Ha nincs a megfelelő típusú fájlból, vége a rutinnak.

if (@t)==D T #l@B=$rd;
E #l@B=$rm;

[| ?#l "T#" ^ t @t ; // Fix számszor lefutó ciklus, annyiszor fut le ahány fájlbejegyzésünk van.
if #c@^(@t)[?-] E G #l@B // Meghívjuk a tartalomjegyzék-bejegyzés típusától függő szubrutint
|]

xx

$rm // fájl-töröl szubrutin
RM (@^A) + (?#s "Tn" ^ t, ?-, @t)
«

$rd // könyvtárat rekurzívan töröl szubrutin
RD (@^A) + (?#s "Tn" ^ t, ?-, @t)
«

„Kiíró rutin” // Kiírja a megfelelő sort a fájlba
// Input paraméterek:
#c@t // típus: D: directory, R: reguláris fájl, L: link

#s@y=" "; #s@y[0]=@t;
#l@i=0; // Ciklusszámláló kezdeti értéke
if (#l (?#l "T#" ^ t @t)==0) T xx; // Ha nincs megfelelő típusú fájl, nem ír ki azokról semmi adatot. Nem hülye ő...
[| ?#l "T#" ^ t @t; // Ciklus indul, annyiszor fut le, ahány fájl van az adott típusból. Legalább egy egészen biztosan
akad.
<s @^o (@y)+(#l@i)+ " = "+(?#s "Tn" ^ t, #l@i, @t)+"\n"; // A sor kiírása
// A fenti 3 sorban a ^ jel azt mutatja, hogy a t nevű speciális tartalomjegyzék-objektumot(változót) a szülő-névtérből
// veszi (ott keresi) és nem a függvény saját névtérében található változók közül.
#l++@i; // Ciklusszámláló növelése
|] // A ciklus vége
xx
```

27. fejezet: A mai program programmemóriájának elérése

Néha bizony hasznosak lehetnek olyan eszközök, melyeket valamely, manapság már lebecsült, régi, primitív programnyelvben találtak ki és fel illetve vezettek be, többnyire azért, hogy az akkori gépek igencsak szűkös kapacitását ellensúlyozzák. Ezeket a módszereket a mai programnyelvek már általában nem alkalmazzák, mondván hogy divatjamúltak, meg veszélyesek meg ilyesmi. Na de miért is kéne lemondani valamiről ami ha ritkán is de jó szolgálatot tehet, pláne, ha a leprogramozása megvan pillanatok alatt, és a programkód mérete is csak elhanyagolható mennyiségben növekszik meg miatta?!

Ilyen a „data” nevű utasítás a régi C-64 -es számítógép Basic-jében. E kulcsszó után ott fel volt sorolva egy csomó szám, vesszővel elválasztva, s volt utasítás rá, ami ezekből mindig a következőt olvasta be.

Na most, ennek a lassúsága nyilvánvaló, sőt helypazarló is a kódban. Hülyeségnek tartottam már akkor is. Igen ám, de maga az alapelv, hogy néha jól jöhet ha elérhetjük a programmemória területét azért, hogy onnan adatot olvassunk be, az már nagyonis okos gondolat! Nem szükséges azonban, hogy ennek lehetőségét lekorlátozzuk speciálisan holmi egész számokra. Egyszerűen tegyük lehetővé egy függvény által, hogy a programmemória bármely bájtyát kiolvashassuk, s ebből

már a Felhasználó összeállíthat bármely, a számára kedves függvényt minden neki tetsző adattípusra!

Íme a megfelelő rutin rá:

```
unsigned char programkodja(F& f) { // visszaad egy bájtot ami a program adott címén található.  
// Szintaxis:  
// "p" c  
// ahol a c egy unsigned int érték. Ha így hívjuk meg:  
// "p" ^ c  
// akkor a szülő névtérből olvas.  
unsigned int c; F *ff; ff=&f;  
memset(f); if(f.p[f.P]=='^') {ff=f.parentF; k(f);}  
c=ERTEKunsignedint(f);  
if(c>=ff->phossz) {L("A programkód határán túlról akarsz bájtot kiolvasni a kódból! Pozíció: %lu",f.P);EXITFAILURE();}  
return ff->p[c];  
}
```

A használatát egy olyan kis progin mutatom be, ami kiírja ÖNMAGÁT, azaz önmaga kódját:

```
#!/mau  
"A progí mérete: " ?l ?p; " bájt\n";  
"Most kiírom önmagam programkódját:\n"  
{l ?p; ? ?#c "P" ?p-?|; |}  
/;
```

Látható a lényege: A **{l** egy fix darabszámszor lefutó ciklust specifikál. Ez **?p** darabszámszor fut le, ami a programmemória maximális mérete. Ezután a **?** egy kiíró utasítás, ami az utána következő bájtot bájt-ként akarja kiírni, azaz karakterként és nem a kódját. A **?#c** azt jelenti hogy olyan függvényt hívunk meg, ami unsigned char értéket ad vissza. E függvény neve a **P**. Ez egy unsigned int számot vár ami meghatározza, a programmemória hányadik bájtját adja vissza. Ezt mi úgy állítjuk elő neki, hogy a programmemória maximális értékéből, amit a **?p** függvény ad meg nekünk, levonjuk a ciklusváltozó aktuális értékét amit a **?l** ad vissza. A ciklusváltozó ugyanis mindig felülről lefelé halad, azaz egyre csökken.

Többnyire természetesen nem akarjuk elérni a programunk memóriáját az elejétől, mert mi a fittyfenéért is kellene nekünk maga a programkód! Ellenben az nagyonis jó dolog lehet, ha a legutolsó kódsor után odabiggyesztünk egy címkét, mondjuk hogy

\$dd

ami jelzi hogy innen következnek az adatok, aztán inntől kezdve olvasgatjuk a programmemóriát, úgy, hogy az aktuális címet a **(\$dd)+valami** módon állítjuk elő!

28. fejezet: A switch-szerű vezérlési szerkezet

A C nyelv switch-nevű vezérlési szerkezetét számos alkalommal használtuk már a mau programnyelv fejlesztése közben. Vélhetőleg éppily jól jönne nekünk valami efféle a mau nyelvbe is - bár úgy tűnik, hála a nyelvünk temérdek indirekciós lehetőségének, simán nélkülözhetjük ezt. Na de miért mondjunk le róla, ha nem muszáj?!

Ki kéne tehát találnunk valami normális szintaktikát rá, miként is menjen az, hogy az interpreter egy unsigned char értéket beolvas, aztán ezt sorra összehasonlítja más efféle értékekkel, s ha egyezést talál, átadja a vezérlést a megfelelő kódsorra.

A switch esetén az efféle ciklusszerűséget a switch utasítás vezeti be; az egész miskulancia végét csukókapcsoszárójel jelzi, az eseteket a case kulcsszó, ezek végét a break kulcsszó, s van neki olyan kulcsszava is hogy default. Na most ezeket sorra nekünk is meg kell valósítani ha nem is ugyanezen nevekkel — bár a break mintha felesleges volna! Tapasztaltam magam is, de sokan mások is mondják, hogy azt bizony a case-ágak végéről hajlamos lehagyni programírás közben az ember, s ez rengeteg randa hiba forrása. Ugyan megvan ennek a haszna, hogy így többféle, egymás után felsorolt esethez is hozzá lehet rendelni ugyanazon tevékenységet, de erre a lehetőségre igencsak ritkán van szükség, szóval a switch ciklus e működési megvalósításának, hogy állandóan „breakolga-tunk”, sokkal több a kára, mint a haszna. Szerintem. Mert miért is ne lehetne olyasmi, hogy ha egy case utasításhoz ér akkor, amikor már úgyis egy case ágot hajt végre, akkor azonnal a switch-blokk végére ugrik?!

Na szóval, úgy találtam, én mindezt meg tudom oldani mindössze 2 utasítással: ezek a ... és a **?! utasítások**.

Ezek pedig imígyen „műxik magukat”:

```
int kerdojel3pont(F& f) { // Összehasonlító utasítás, afféle „switch”.
// Szintaxis:
// ?! c
// ...K1
// ...K2
// .....
// ahol a c, valamint a K1, K2 stb értékek egy-egy unsigned char típusú kifejezés.
// A rutin beolvassa a c kifejezést, majd sorra megvizsgálja a következő programsorok elejét. Ha
// az a ... jellel kezdődik, kiértékeli az utána következő K1, K2 stb kifejezéseket is mint egy-egy unsigned char értéket,
és ha
// a c értéke ezzel megegyezik, a vezérlés a megfelelő K kifejezés utánra adódik. Ha nem egyezik meg, tovább keresi a
// megfelelő ... -al kezdődő sort ahol már esetleg megegyezik. A megfelelő érték keresése akkor áll csak le
// egyezés hiányában, ha olyan sorra lel, ami egymás után 2 db ... karakterrel kezdődik, ez a „default”, azaz ha
// ilyenre lel innen folytatja e 2 ... jel után a programvégrehajtást, akármilyen legyen is a c értéke.
// Amennyiben azonban 3 db ... jelet talál egymás után, akkor azután folytatódik a vezérlés.
// Azaz egy efféle vezérlési szerkezetet mindenféleképpen le kell zárunk a ..... jelsorozattal,
// különben a program végéig keresne!
unsigned char c; unsigned char K;
c=ERTEKunsignedchar(f); goto kerdojel3pont_ciklus;
kerdojel3pont_ujsorigciklus:
ujsorig(f);
kerdojel3pont_ciklus:
if(f.p[f.P]!=226) goto kerdojel3pont_ujsorigciklus; // A ... jel első bájtjának kódja
if(k(f)!=128) goto kerdojel3pont_ujsorigciklus; // A ... jel második bájtjának kódja
if(k(f)!=166) goto kerdojel3pont_ujsorigciklus; // A ... jel harmadik bájtjának kódja
k(f); namespace(f);
if((f.p[f.P]==226)&&(f.p[f.P+1]==128)&&(f.p[f.P+2]==166)) {k(f);k(f);k(f);
if((f.p[f.P]==226)&&(f.p[f.P+1]==128)&&(f.p[f.P+2]==166)) f.P+=3;
return 0;}
K=ERTEKunsignedchar(f);
if(c!=K) goto kerdojel3pont_ujsorigciklus; // Ha nem egyezik, tovább vizsgálódunk
return 0;
}
// =====
int fuggvenyharompont(F& f) { // A ... utasítás. Ez csak annyit csinál,
// hogy azonnal elugrik a következő sor elejére, s megvizsgálja,
// az nem kezdődik-e triplán ... -al. Ha igen, a vezérlés azután folytatódik, ha nem, akkor tovább keres. Amennyiben
azonban
// ő maga a tripla ..., akkor semmit se csinál.
if(f.p[f.P]!=128) ERROR(f,3); // A ... jel második bájtjának kódja
if(k(f)!=166) ERROR(f,3); // A ... jel harmadik bájtjának kódja
if(k(f)!=226) goto harompont_ciklus;
if(k(f)!=128) goto harompont_ciklus;
if(k(f)!=166) goto harompont_ciklus;
if(k(f)!=226) goto harompont_ciklus;
if(k(f)!=128) goto harompont_ciklus;
if(k(f)!=166) goto harompont_ciklus;
k(f); return 0;
harompont_ciklus:
ujsorig(f);
if(f.P+9>=f.phossz) ERROR(f,3);
```



```

if(f.p[f.P] !=226) goto harompont_ciklus;
if(f.p[f.P+1]!=128) goto harompont_ciklus;
if(f.p[f.P+2]!=166) goto harompont_ciklus;
if(f.p[f.P+3]!=226) goto harompont_ciklus;
if(f.p[f.P+4]!=128) goto harompont_ciklus;
if(f.p[f.P+5]!=166) goto harompont_ciklus;
if(f.p[f.P+6]!=226) goto harompont_ciklus;
if(f.p[f.P+7]!=128) goto harompont_ciklus;
if(f.p[f.P+8]!=166) goto harompont_ciklus;
f.P+=9; return 0;
}

```

A fentiek bemutatása egy kis példán:

```

?! g
...a "a\n";
...b "b\n";
...c "c\n";
...d "d\n";
...e "e\n";
"Ez még az e betű sora!\n"
...f "f\n";
...g "g\n";
..... "valami más!\n";
"Ez is a valami máshoz tartozik!\n"
..... "Itt a vége!\n"

XX

```

Fontos megérteni a következőket:

A „három pont” nem 3 darab különálló pont, azaz nem 3 darab ilyen: ".", hanem az a "..." karakter, aminek a kódja az UTF-8 szerint a 226,128,166 bájtsorozat! Továbbá, a fenti elágazásszervezésben amint a ?! parancs kiértékelte az utána következő aritmetikai kifejezést, azonnal abbahagyja a sor további feldolgozását, nem veszi figyelembe, ami utána van. Ezenkívül, azt meg lehet csinálni, hogy ne írjunk olyan sort, ami csak 2 db „háromponttal” kezdődik, azaz nem definiálunk „default” eseményt, amit „nem illeszkedés” esetén hajt végre az interpreter, azt azonban semmiféleképpen sem szabad elfelejtenünk, hogy legutoljára legyen egy

, azaz 3 db „háromponttal” kezdődő sorunk, ami azt jelzi hogy itt ér véget ez az egész miskulancia.

Az egyes eseményágak nem muszáj hogy beleférjenek egyetlen programsorba, látható hogy folytatódhatnak több soron keresztül is, és egyáltalán semmivel se muszáj jelölni a végüket, mert nekik az a vég, hogy egy újabb hárompont-utasításhoz érkeznek. Ellenben egy eseményen belül nem nyithatunk újabb efféle szerkezetet a ?! utasítással... Azaz, ezek nem ágyazhatóak egymásba. Viszont nyugodtan kiugorhatunk belőle bármiféle nekünk tetsző módon, például a » utasítással.

Van például a maudir programban egy ilyen összehasonlító rész:

```

if (@t)==D T #s@y="rm -rf ";
E #s@y="rm ";

```

Ezt így kéne átírnunk e szerkezetre, ha akarnánk:

```

?! @t
...D #s@y="rm -rf ";
..... #s@y="rm ";
.....

```

Csak éppenséggel ennek nem sok értelme van. Egyetlen összehasonlítás kedvéért nem érdemes ilyen szerkezetet létrehozni. Ennek akkor van értelme, ha sok különböző esetből kell választani. Minél többől, annál hasznosabb.

29. fejezet: A második ténylegesen hasznos mau nyelvű programunk, ami névsorba rendezi egy fájl sorait

Írtam korábban egy primitív kis szótárprogit, ami azonban egyszerűsége ellenére is teljesen megfelelt a céljaimnak. Azonban ennek „adatbázisa” egy szöveges fájl, minden szóhoz egy sor tartozik. Na és hát néha ugye bővül a szóállomány, s akkor azt bizony illik névsorba rendezni! Természetesen a korábban megadott sorrendiség szerint, amit a stringeknél ecseteltem, mert nekem az tetszik.

Na most, e feladatra korábban írtam én természetesen egy C nyelvű programot, ami még kiválóan működik is. Na de milyen már az hogy van saját programnyelvem, s nem abban írom meg a nekem szükséges feladatokat... Jó-jó, senki nem vethet rám követ ezért, mert amikor azt a rendezőprogit írtam, még hírehamva sem volt a mau programnyelvnek! Most már azonban van nekem olyanom hogy saját programnyelv, illik tehát e csúf hiányosságot kiküszöbölni!

Íme tehát a program, ami egy megadott fájl beolvas, a sorait névsorba rendezi (a mau programnyelv SZABVÁNYA szerinti sorrendben...) és kiírja egy másik fájlba aminek a neve azonos az eredetivel, de a végéhez hozzábiggyeszti, hogy „.rendezett”. Az eredeti fájl természetesen nem bántja.

```
#!mau // Egy file sorainak névsorba rendezése
// rendez.mau
if (?a)<3 T "Nem adtad meg a rendezendő állomány nevét!\n" XX;
    E #s@a=?#s "ARGV" 2; // A rendezendő file neve

<? @A; // Megszámoljuk a sorait
"A file sorainak száma: " ?l ?n ; /;
"A file leghosszabb sora " ?l ?!; " bájttal hosszú.\n";

#B@b=@A; // Megnyitjuk az input fájlt
if #B@b E "Az input file nem megnyitható!\n" XX

[?t[?n]]; // lefoglalok területet annyi string részére, ahány sor van a fájlban, a t nevű stringtömbbe
{[ ?n; // A ciklus annyiszor fut le, ahány sor van a fájlban
#s@t[?n-?]]]=?#s "NL", b, ?!; // Beolvassuk a sort, de max. annyi karaktert, amennyi a leghosszabb sor a fájlban.
} // vége a ciklusnak
[#B@b]; // bezárjuk az input fájlt
QS t,?n; // névsorba rendezzük a tömböt
#K@o=(@A)+".rendezett"; // Megnyitjuk írásra az output fájlt
if #K@o E "Az output file nem megnyitható!\n" XX
{[ ?n; // A ciklus annyiszor fut le, ahány sor van a fájlban
<s @o, @t[?n-?]]]; // Kiírjuk a sort a fájlba
} // vége a ciklusnak
[#K@o]; // Lezáróm az output fájlt

XX // vége a programnak
```

Érdekességképpen megjegyzem, hogy teljesen pontosan ugyanezen feladat megoldása az eredeti C nyelvű programban akkora forráskódot eredményezett, ami 13966 bájt hosszú volt, s e C progi binárisa pedig 11056 bájt méretű lett. Sőt, még rosszabb is volt az a progi, mert nem írta ki nekem hány sor van a fájlban, s mennyi a leghosszabbnak a mérete. Ezzel szemben e mostani mau nyelvű programom mérete csak 1115 bájt! Azaz, a mérete tizedakkora kb, holott telis-teli van megjegyzésekkel, nem túlzás hogy több benne a komment mint az érdemi kód! És teljesen ugyanazt a rendezési módszert alkalmazza.

Oké, most erre valaki mondhatja hogy csalok, mert hiszen szinte minden az interpreterben van igazából, ami jelenleg 270304 bájt nagyságú a bináris kódját illetően. Ami jóval nagyobb mint a C nyelven megírt rendezőprogram. IGAZ, nem

tagadom. De az interpretert csak egyszer kellett megírnom, s eztán bármikor rendelkezésemre áll! És a mau nyelvű rendezőprogram méretén teljes nyugalommal zsugoríthatok még rengeteget, mert nézzük csak mi lesz belőle, ha kiszedem a kommenteket:

```
#!mau // Egy file sorainak névsorbarendezeése
if (?a)<3 T "Nem adtad meg a rendezendő állomány nevét!\n" XX;
    E #s@a=?#s "ARGV" 2;

<? @A;
"A file sorainak száma: " ?l ?n ; /; "A file leghosszabb sora " ?l ?!; " bájttal hosszú.\n";
#B@b=@A; if #B@b E "Az input file nem megnyitható!\n" XX
[?t[[?n]]]; { | ?n; #s@t[[?n-?]]=?#s "NL", b, ?!; | } [#B@b]; QS t,?n; #K@o=(@A)+".rendezett";
if #K@o E "Az output file nem megnyitható!\n" XX
{ | ?n; <s @o, @t[[?n-?]]]; | } [#K@o];
XX
```

Ez máris csak 492 bájttal, s tulajdonképpen még ezen is lehetne faragni, mert a pontosvesszők nem igazán szükségesek. S ezen maradék hossz java része sem más, mint a kiíratandó szövegek...

Hm, bevallom egyre jobban tetszik a programnyelv, amit írtam! Hatalmas szellemi élvezet hogy alkottam egy ilyesmit, ami ennyire jól használható! Szerencsétlen nyomorúságos szellemi páriák, akik nem írtak programnyelvet, de sajnálom őket! Fogalmuk sincs róla, MEKKORA élvezetről maradnak le! Csakis és kizárólag biztathatok mindenkit hogy VÁGJON BELE, mert magasan MEGÉRI! És ne jöjjön nekem senki se azzal, hogy de ez nehéz. Nyilván nem olyan könnyű persze mint egész nap a fotelban tespedni s az idióta Valóságshow-kat bámulni, hogy a Győzike kutyája mekkorát okádik a szőnyegre. De hegyet mászni is nehéz, aztán mégis mennyien megteszik. Szórakozásból. A programnyelvírás is ilyen, csak még annál is sokkal jobb, mert ha EGYSZER megírtuk, azután már MINDIG a rendelkezésünkre áll, nem úgy mint a hegymászás. Azaz a programnyelvírással MARADANDÓT alkotunk!

30. fejezet: Változó hosszúságú paraméterlista kezelése

Programnyelvünk már nagyon szép és jó és csecse meg minden, alkalmas nagyon sok mindenre, de van két hiányossága: Az egyik hogy nem alkalmas még rekurzív függvényhívásokra, a másik pedig, hogy nem tud változó hosszúságú paraméterlistát kezelni.

Na most, a rekurzív hívási lehetőség hiánya nem olyan szörnyű nagy gond, mert meglehetősen ritkák az olyan feladatok, amik csak úgy oldhatóak meg. A rekurzív hívással történő problémamegoldás majdnem mindig kiváltható ciklusszervezéssel, s ha mégis rekurzívan oldja meg azt a programozó, hát alig van példa rá, amikor ezzel nem önmagát „szopátja”, már bocs... De mert azért valóban akad NÉHÁNY, nagyon kevés probléma amihez ez a lehetőség muszáj, majd megoldjuk ezt is később. Ez azonban tényleg ráér. Ami azonban a változó hosszúságú paraméterlista kezelését illeti, ez olyasmi, ami nélkül egy egész csomó feladat csak alig oldható meg, vagy éppen egyáltalán nem!

Gondoljunk csak rá, miként írnánk valami olyasféle függvényt, ami kb hasonló feladatot lát el, mint a C nyelvből ismert „printf”. Ennél csak egyetlen dolog biztos: első paramétere egy string. Hogy azonban van-e ezen kívül még para-

métere, s hogy hány, s milyen típusúak, az a string elemzéséből derül ki, azaz ezt egyszerűen nem tudhatja a szerencsétlen hívott függvény a hívás pillanatában még! Vagyis az eddigi függvényhívási módszerünk, amikor is a hívott függvény szépen sorban felsorolta a maga változóit név szerint, s abba a hívó belepakolta a megfelelő értékeket, ez teljességgel alkalmatlan a számunkra e feladattípusokhoz!

Mégis, azt már olyan gyönyörűre megírtuk, hogy nem szívesen mondanánk le róla. Legjobb az volna, ha megmaradna úgy ahogy van, de volna valami plusz lehetősége, amivel e gond megoldhatóvá válna! Mondjuk, hasonlóan mint a C nyelv esetén: amikor elfogytak a biztosan szükséges paraméterek, odaírnánk nagy lazán három pontot, azaz e jelet: "...", és ezek után az esetleg létező egyéb paramétereket a hívó valahova máshova pakolja, nem a hívott függvény konkrét, nevesített változóiba, de olyan helyre, hogy onnan a hívó azért elő tudja szedgetni, ha kellenek neki!

Miféle hely is legyen ez? Nyilvánvalóan csak olyan adattípus jöhet szóba, aminek a mérete meglehetősen kötetlenül változtatható, hívásról-hívásra, valamint nem igazán érzékeny arra se, miféle típusú adat kerül bele. Ilyen adattípus nálunk a VEREM. Igenám, de azért valahonnan mégis tudnia kell a hívott függvénynek, hogy hol milyen típusú adat lett átadva neki...

Bevallom, rémségesen sokáig lustálkodtam e feladat megoldása kapcsán! Bár azt hogy veremre van szükségem, tudtam az első pillanattól kezdve, de az cseppet se volt magától értetődő a számomra, e sokrétű kérdéskör kezeléséhez miféle szintaxist valósítsak meg.

Végülis azt hiszem sikerült kitalálnom egy fölöttébb rugalmas szisztémát, mely nem igényelte az eddigi függvényhívási módszerünk különösebb átdolgozását, úgy értem, programozástechnikailag. Azaz amit eddig írtam a mau nyelvű függvény-hívásokról, mind igaz maradt. Ugyanakkor attól tartok, annak elmagyarázása, miként kezeljük a változó hosszúságú paraméterlistát, kissé hosszú lesz! Vágjunk is bele.

Íme rögvest egy mau nyelvű példa, ezen magyarázok el mindent!

```
#!mau

"Meghívom a függvényt\n"

(,,"pontok" 4, 425,#c64,#c65,#c66,#c67)
(,,"pontok" 6, 512,#c68,#c69,#cz,#c70,#c71,#cu \ #l@L)

"Visszatérési érték: " ?l @L; /;

XX

,,pontok" // változó hosszúságú paraméterlistát váró függvény
// Input paraméterek:
#c@c, // Ez mondja meg hány további paramétert várunk
#i@i,
[#c = #c@c], // Memórafoglalás a megfelelő számú unsigned char értéknek
... // Minden további paraméter verembe kerül

? #c@[0]; /;
?i @i; /;

{[ #c@c; ? #c@[?]-1]; ]} /;

\ 65537
xx
```

Látható, 2 alkalommal hívtuk meg a „pontok” nevű függvényt, s tényleg különböző hosszú paraméterlistával! Na most, a hívott függvény az elején felsorol 2 (unsigned char és unsigned short int típusú, de lehetne más típus is) input változót. Ez rendben is van, mindkét alkalommal amikor ő meghívatik, ezekbe belekerül az első 2 átadott paraméter, először 4, 425, azután 6, 512. Ezután azonban egy olyan fura tudákos hogyishívják jön, hogy:

```
[#c = #c@c], // Memóriafooglalás a megfelelő számú unsigned char értékek
```

Na most, ez a fenti kutyulmány memóriafooglalás. Veremtárnak foglalunk memóriát, a **#c** azt mondja hogy unsigned char értékeknek. Az egyenlőségjel után van megadva, hány darab ilyen értéknek foglalunk vermet. Itt most ezt a darabszámot egy változó határozza meg, az, amit az imént kaptunk a hívó függvénytől! Lehetne a helyén valamilyen konstans szám is, hogy mondjuk 30, vagy a ^ karakter segítségével hivatkozhatnánk itt a szülő névtér valamely változójára is. Az a darabszám amennyinek itt helyet foglalunk, a **maximális** darabszám, tehát nem muszáj hogy mindegyik használtassék is.

Veremtárlefooglaló utasításunk ismerős kell legyen már, de van egy furcsasága: csak a típusa van megadva, de nincs NEVE! S ez nem véletlen. Ezek különleges vermek. Ezzel az utasítással kizárólag a függvény legelején, a paraméterátadás ideje alatt foglalhatunk vermet, s e vermeknek abszolút semmi közük a korábban megismert verem-típusú változóinkhoz! E vermekbe a hívott függvény nem is képes adatot tenni, ezek kizárólag „input” vermek, ezekbe semmi más nem kerülhet, mint a hívó függvény által szolgáltatott input paraméterek! Ezekből tehát e vermek adatait csak „jobbértékként” lehet felhasználni. E vermek minden híváskor újra létrejönnek, nem is okvetlenül ugyanazon mérettel, s a függvény lefutása után megsemmisülnek. Minden adattípusnak külön vermet specifikálhatunk, a megfelelő típusjelölő karakter segítségével. Azaz lehet vermünk a **#c, #C, #i, #I, #l, #L, #g, #G, #f, #d, #D** és a **#s** típusokra.

Ebben a sorban:

```
? #c@[0]; /;
```

látható, miként hivatkozunk a verembe rakott elemre. A hívó függvény által a verembe rakott első elem lesz a nulladik indexű a hívott függvény számára, azaz ahogy fel vannak sorolva a paraméterek a hívó függvény listájában balról jobbra, úgy nő az indexük nullától számítva. A verembe legelőször akkor kerül elem a paraméterátadás során, amikor a hívott függvéynél elérkezik egy „...” karakterhez. Ez jelzi azt, hogy innentől az összes további esetleges paramétert ami még van, a vermekbe kell pakolja a hívó függvény. Természetesen minden adat a maga megfelelő típusú vermébe kell kerüljön, s ha annak a típusnak nem foglalt le memóriát a hívott függvény, akkor az hibajelzéshez vezet, amint a hívó épp egy olyan típusú adatot óhajtana bepakolni.

Nézzük aztán meg ezt a sort, a függvényhívást:

```
(„pontok” 4, 425, #c64, #c65, #c66, #c67)
```

A paramétereknél a 4 és 425 számoknál nincs semmi baj. A többi azonban mintha kissé furcsán festene. Nos igen, amiatt, mert mindegyik szám előtt ott a casting operátor! Ha nem konstans számokat adnánk meg hanem változókat a teljes nevükkel, nem volna probléma, mert mondjuk a **#d@f** alakból egyértelműen kiderül, hogy ez egy double szám. Ha azonban csak egy szám áll ott, abból nem tudja az interpreter, ezt most épp melyik verembe kell tegye: az unsigned char verembe, vagy a signed long long verembe, vagy szóval most mit is kezdjen vele?!

Emiatt a verembe kerülő paramétereknél minden esetben szigorúan ki kell tenni a paraméter elejére a megfelelő casting operátort.

Nézzünk meg egy másik példát, ahol stringet is átadunk:

```
(„pontok3” 6, 2, 4, #c68,#c69,#cz,#i384#c70,#c71,#cu,#i1024,#s"String-2!",#s(#s@)+”valami” \ #s@v)
```

Látható itt a végén, hogy egy olyan kifejezést adunk át, ami úgy keletkezik, hogy a `@s` típusú stringváltozóhoz hozzáadjuk a "valami" konstans stringet. Minthogy efféle stringkonkatenáció esetén a változókat zárójelbe tesszük, így hogy rögvest tudja az interpreter hogy string következik, muszáj ezelé is odabiggyeszteni a `#s` casting operátort. Ez előtt van egy idézőjelek közt álló stringkonstans is, az elé is kell a casting operátor, láthatjuk. E stringekre a hívott függvényben így hivatkozhatunk, mondjuk egy kiíró utasításnál:

```
?s #s@[0]; /;  
?s #s@[1]; /;
```

Bár e paraméterek veremekben vannak, de attól hogy a hívott függvényben valahol eképp használjuk őket, nem kerülnek még ki a veremből. Csak akkor törlődnek, amikor a függvény teljesen végetér, s a vezérlés visszakerül a hívóhoz.

Miután már elég sok mindent jelölünk szögletes zárójelekkel, illendő egy rövid, tömör összefoglalást iderittyenteni e jel szintaktikájáról:

```
[#c = #c@c], // Memória foglalás a megfelelő számú unsigned char input paraméter értéknek  
#c@k=#c@[0]; // Kiolvasás az unsigned char értékeket tartalmazó input paraméter veremből, 0 indexszel  
#c@c=#c@v[]; // A v nevű verem aktuális unsigned char eleme.  
#c@v[]=4; // Beteszünk 4-et a v nevű verembe unsigned char számként.  
[@v[2000]] // 2000 bájtot lefoglalunk a „v” nevű verem számára  
[@v[]]; // Törölöm a v nevű vermet  
[#c@t=100]; // Memória foglalás 100 darab unsigned char értéknek a "t" nevű tömbbe  
#c@t[#i@i]=4; // Értékkadás a t nevű, unsigned char értékeket tartalmazó tömb egy elemének  
#c@c=#c@t[#i@i]; // A tömb egy értékének kiolvasása  
[#c@t]; // A t nevű unsigned char értékeket tartalmazó tömb törlése (a neki lefoglalt memóriaterület felszabadítása)  
#c@c=#s@s[4]; // A 4-es indexű karakter az s nevű stringből  
#s@s=[3,7]#s@s; // Az s nevű string egy darabjának képzése  
[@t[4]]; // lefoglalok területet 4 string részére egy t nevű stringtömbbe  
#s@t[1]="macska"; // Értékkadás stringtömb egy elemének  
#s@t[1][3]='@'; // Értékkadás egy olyan karakternek, mely egy stringtömb egyik stringjében van.  
[@t[]]; // Felszabadítom a stringtömb memóriaterületét
```

Vezessünk be azonban egy kis csinosítást nyelvünkbe, aminek ugyan semmi köze a változó hosszúságú paraméterlistához, ellenben mégis ennek apropóján jutott eszembe! Az idézett példában szerepel ugyanis e kifejezés:

```
?|-1
```

És hát ez eddig már jó sok alkalommal szerepelt pontosan ugyanígy. Ez a fix számszor lefutó ciklus ciklusváltozójának eggyel csökkentett értékét jelenti. Ha ez már eddig is ennyire gyakori, érdemes erre egy külön rendszerváltozót bevezetnünk! Legyen ennek jele a

```
?-
```

Természetesen a mínuszjel itt arra utal, hogy ez a ciklusváltozó olyan értéke, amiből már eleve le van vonva 1! Mindjárt rövidebbek és áttekinthetőbbek lesznek a mau programjaink, talán még gyorsabbak is mert 2-vel kevesebb karaktert kell beolvasniuk/feldolgozniuk, sőt, elmarad egy egész függvényhívás, amikoris korábban meghívott egy aritmetikai kifejezés kiértékelő rutint, átadva neki az F struktúra pointerét, csak azért, hogy aztán rájöjjön, mindössze egy nyavalyás 1

értéket kell levonnia! Most ez mind NINCS. Továbbá, a legtöbb billentyűzeten a mínusz karakter is könnyebben begépelhető, mint a | jel.

Hasonló okokból megcsináltam, hogy a numerikus változók értékének dekrementálásához hasonlóan lehessen dekrementálni a stringek hosszát is, eképp:

```
#s--@a;
```

Így mégiscsak egyszerűbb levágni a string utolsó karakterét, sőt (szóhoz engedem jutni ezúttal a bennem munkálkodó kis kaján ördögfiókat...): az egész program így megintcsak átláthatóbb, világosabb, könnyebben érthető, hiszen ki az, ugyan KI, akinek a fenti jelsorozat láttán nem ugrik be azonnal, hogy itt egy string utolsó karakterének elhagyásáról van szó?! Vagy létezne mégis ilyen ember?! Ja, az ne mau nyelven programozzon... Ez az ELIT programozási nyelve, kéremalássan...

31. fejezet: „Igazi” függvények és rekurzív függvények

Amikor páran megtudták hogy programnyelvet írok, s megnézték az eddigi eredményeimet, egyesek kritizálni kezdték, mondván hogy amit függvénynek nevezek, igazából nem függvény, mert nem építhető be aritmetikai kifejezésekbe! Nos, ez már attól függ, miként definiáljuk a függvény fogalmát. Mint tudjuk jól a C nyelv esetéből is, egy függvény visszatérési értékét nem okvetlenül kell fogadnunk. Sőt, lehet hogy direkt **void** visszatérési értékűre írjuk, s akkor máris nagyon hasonló az eddigi mau függvényekhez! Ettől azonban még csinálhat nekünk valami hasznosat, sőt, még értékeket is adhat vissza, mutatókon vagy referencia típusú paramétereken keresztül.

Ennek ellenére, kritikusaimnak igazuk van abban, hogy hasznos volna ha a mau nyelv rendelkezne valami „hagyományosabb” stílusú függvényhívási lehetőséggel is. Ennek persze az az ára, hogy egy efféle függvénynek tényleg csak 1 visszatérési értéke lehet, ráadásul az se mindegy, annak milyen a típusa! Természetesen e függvények egymásba ágyazhatóak kell legyenek, na és ha már megcsináljuk e huncutságot, akkor ez már legyen olyan, ami alkalmas önmaga meghívására is, azaz rekurzívan is hívható!

Nos, ezt a következő programon mutatom be. E program kiszámolja az 5 és a 6 faktoriálisát. A faktoriálisszámító függvény önmagát hívja rekurzívan:

```
#!mau
#l@i=?(#l „faktoriális” #l5); // Itt az 5 szám faktoriálisát számíttatjuk ki
?l @i; // Ki is írjuk az eredményét
#l@i=?(#l „faktoriális” #l6); // Itt meg a 6 szám faktoriálisát számíttatjuk ki
?l @i; // Ezt is kiíratjuk

XX // Itt a (főprogram) vége, fuss el véle!

„faktoriális” // Faktoriális-számoló függvény. REKURZÍV!
if (#l@^[0])<2 T \ #l; // A nulla és az 1 faktoriálisa mint tudjuk 1.
// Ha nem 0 vagy 1, az adott számot megszorozza az eggyel kisebb szám faktoriálisával, s ezt adja vissza:
\ #l@^[0]*?(#l „faktoriális” #l(--@^[0]));
xx // vége a függvénynek
```

Látható, hogy miként kell meghívni az efféle programokat. A **?{** parancs jelzi, hogy függvény következik, rögtön ezután kell álljon a # casting operátor, mely az utána álló karakterrel azt mondja meg, milyen típusú függvényhívás következik. Ez

azért fontos, mert olyan típusú paramétert ad vissza a hívott függvény a hívónak. Ezután következik a hívott függvény meghatározása. Ez egészen ugyanúgy megy mint az eddigi függvények esetében: azaz a „ és ” karakterek közt megadhatjuk a függvény nevét stringkifejezéssel, vagy használhatjuk a **[sorszám]** alakot is, ahol a sorszám egy aritmetikai kifejezés. Ezután jönnek az input paraméterek, amikből akármennyi lehet, azaz ez változó hosszúságú paraméterlista átadására is alkalmas. Természetesen amiatt, mert itt is veremtárakba kerülnek a paraméterek, s itt is muszáj jelezni mindegyik előtt a casting operátorral a paraméter típusát. Am jelen esetben ezek a veremtárak amikbe a paraméterek kerülnek, nem azonosak a korábbi függvényhívásnál említett veremtárakkal! Ezek mérete a **vz.h** fájlban van meghatározva, eképp:

```
#define gveremcmax 1000
#define gveremCmax 1000
#define gvereminax 1000
#define gveremInax 1000
#define gveremlmax 1000
#define gveremLmax 1000
#define gveremgmax 1000
#define gveremGmax 1000
#define gveremfmax 1000
#define gveremdmax 1000
#define gveremDmax 1000
#define gveremsmx 1000
#define gverempmax 1000
```

Ezen veremek globálisak, s eképp vannak deklarálva a main.cpp fájl elején:

```
// Globális adatvermek

VEREM gveremc;
VEREM gveremC;
VEREM gveremi;
VEREM gveremI;
VEREM gvereml;
VEREM gveremL;
VEREM gveremg;
VEREM gveremG;
VEREM gveremf;
VEREM gveremd;
VEREM gveremD;
VEREM gverems;
VEREM gveremp;
```

A **?(** utasításhoz érve, az aritmetikai kifejezés kiértékelő függvény beolvassa az utána következő casting operátort, s ennek megfelelően hívja meg valamelyik függvénykiértékelő rutint. Ezek ilyesféléképp vannak megoldva:

```
unsigned char Fuggvenyc(F& f) { // Amikor belép e függvénybe, az f.P a ?(#x utáni legelső karakterre mutat.
Fuggveny_mag(f);
return f.FV.c[0];
}
// -----
signed char FuggvenyC(F& f) { // Amikor belép e függvénybe, az f.P a ?(#x utáni legelső karakterre mutat.
Fuggveny_mag(f);
return f.FV.C[0];
}
// -----
unsigned short int Fuggvenyi(F& f) { // Amikor belép e függvénybe, az f.P a ?(#x utáni legelső karakterre mutat.
Fuggveny_mag(f);
return f.FV.i[0];
}
// -----
signed short int FuggvenyI(F& f) { // Amikor belép e függvénybe, az f.P a ?(#x utáni legelső karakterre mutat.
Fuggveny_mag(f);
return f.FV.I[0];
}
// -----
unsigned int Fuggvenyl(F& f) { // Amikor belép e függvénybe, az f.P a ?(#x utáni legelső karakterre mutat.
Fuggveny_mag(f);
return f.FV.l[0];
}
// -----
```



```

signed int FuggvenyL(F& f) { // Amikor belép e függvénybe, az f.P a ?(#x utáni legelső karakterre mutat.
Fuggveny_mag(f);
return f.FV.L[0];
}
// -----
unsigned long long Fuggvenyg(F& f) { // Amikor belép e függvénybe, az f.P a ?(#x utáni legelső karakterre mutat.
Fuggveny_mag(f);
return f.FV.g[0];
}
// -----
signed long long FuggvenyG(F& f) { // Amikor belép e függvénybe, az f.P a ?(#x utáni legelső karakterre mutat.
Fuggveny_mag(f);
return f.FV.G[0];
}
// -----
float Fuggvenyf(F& f) { // Amikor belép e függvénybe, az f.P a ?(#x utáni legelső karakterre mutat.
Fuggveny_mag(f);
return f.FV.f[0];
}
// -----
double Fuggvenyd(F& f) { // Amikor belép e függvénybe, az f.P a ?(#x utáni legelső karakterre mutat.
Fuggveny_mag(f);
return f.FV.d[0];
}
// -----
long double FuggvenyD(F& f) { // Amikor belép e függvénybe, az f.P a ?(#x utáni legelső karakterre mutat.
Fuggveny_mag(f);
return f.FV.D;
}
// -----
void * Fuggvenyp(F& f) { // Amikor belép e függvénybe, az f.P a ?(#x utáni legelső karakterre mutat.
Fuggveny_mag(f);
return f.FV.p[0];
}
// -----
MAUSTRING Fuggvenys(F& f) { // Amikor belép e függvénybe, az f.P a ?(#x utáni legelső karakterre mutat.
Fuggveny_mag(f);
return f.FS;
}

```

A **Fuggveny_mag** rutin a lényeg, amit mindegyik használ, ez most így néz ki:

```

void Fuggveny_mag(F& f) {
// Általános függvényhívási rutin. Ezt hívja minden más típusfüggő függvényhívó rutin. E rutin a függvény eredményét
// az f.FV union megfelelő típusú 0 sorszámú mezőjébe pakolja.
// A függvényhívás szintaxisa:
// ?(#x „NÉV” ip1, ip2, ...) vagy
// ?(#x [sorszám] ip1, ip2, ...)
// ahol az x a megfelelő típusjelző karakter, ami a visszatérési érték típusát határozza meg:
// azaz az x lehet: c,C,i,I,l,L,g,G,f,d,D,s
// az ip1, ip2 stb pedig az input paraméterek. Ezek kötelezően egy #x casting operátorral kell kezdődjenek!
// A NÉV egy stringkifejezés és a függvény nevét jelenti, a sorszám pedig a függvény sorszáma, és
// tetszőleges aritmetikai kifejezés lehet.

USIL oldP; unsigned char típus; MAUSTRING *S; unsigned char módszer; unsigned int sorszam; PGM *pgm; F *pgmf;
unsigned int veremmutatoc, veremmutatoC, veremmutatoi, veremmutatoI, veremmutatol, veremmutatoL, veremmutatog, veremmutatoG,
veremmutatof, veremmutatod, veremmutatoD, veremmutatos;

// A hívandó függvény nevének és sorszámának meghatározása
namespace(f);
módszer=f.p[f.P]; // A "?(#x" utáni első nem whitespace karakter
if(módszer!='[') { // sorszám alapján hívjuk meg a függvényt
k(f); sorszam=ERTEKunsignedint(f); namespace(f);
if(sorszam>f.maxpgm) {L("Nem létező sorszámú függvényt akartál meghívni! Pozíció: %lu",f.P);EXITFAILURE();}
if(f.p[f.P]!='[') {L("Hiányzó „[” karakter függvényhívó utasításban! Pozíció: %lu",f.P);ERROR(f,3);}
k(f);
} // if [ vége
else {if(módszer!=226) {ERROR(f,3);} módszer=k(f); if(módszer!=128) {ERROR(f,3);} módszer=k(f);
if(módszer!=158) {ERROR(f,3);} k(f);
sorszam=MAUfuggvenysorszama(f);
if(sorszam==65535) {L("Olyan nevű függvényt akartál meghívni, ami nincs betöltve! Pozíció: %lu",f.P);EXITFAILURE();}
namespace(f); módszer=f.p[f.P]; // A "[]" karakter első bájta
if(módszer!=226) {ERROR(f,3);} módszer=k(f); if(módszer!=128) {ERROR(f,3);} módszer=k(f); if(módszer!=157)
{ERROR(f,3);} k(f);
}
// Most eljutottunk odáig, hogy tudjuk, az f.P pointer a függvénynek átadandó paraméterek listájának elejére mutat.

pgm=f.pgmtomb[sorszam]; // E pgm változó csak azért lett bevezetve, hogy kényelmesebben utalhassunk a meghívandó
függvényre
pgmf=&(pgm->f); // Ez a pointer mutat a meghívandó függvény F struktúrájára, ez is csak kényelmi okokból lett bevezetve

// A veremmutatók aktuális állásának elmentése ideiglenes változókbá:
veremmutatoc=gverenc.veremmutato; veremmutatoC=gveremC.veremmutato; veremmutatoi=gveremi.veremmutato;
veremmutatoI=gveremI.veremmutato; veremmutatol=gveremL.veremmutato; veremmutatoL=gveremL.veremmutato;
veremmutatog=gvereng.veremmutato; veremmutatoG=gveremG.veremmutato; veremmutatof=gveremf.veremmutato;
veremmutatod=gverend.veremmutato; veremmutatoD=gveremD.veremmutato; veremmutatos=gverems.veremmutato;

```

```

Fuggveny_mag_inputparameter_verembetesz_ciklus:
namespace(f);
if(f.p[f.P]=='') {k(f);goto Fuggveny_mag_vegrehajt;} // Nincs több paraméter

if(f.p[f.P]!='#') {L("Hiányzó # jel függvényhívásnál egy input paraméter elején! Hely: %lu",f.P);ERROR(f,3);}
tipus=k(f);k(f);
switch(tipus) {
case 'c': gveremc=ERTEKunsignedchar(f);break;case 'C': gveremC=ERTEKsignedchar(f);break;
case 'i': gveremi=ERTEKunsignedshortint(f);break;case 'I': gveremI=ERTEKsignedshortint(f);break;
case 'l': gvereml=ERTEKunsignedint(f);break;case 'L': gveremL=ERTEKsignedint(f);break;
case 'g': gveremg=ERTEKunsignedlonglong(f);break;case 'G': gveremG=ERTEKsignedlonglong(f);break;
case 'f': gveremf=ERTEKfloat(f);break;case 'd': gveremd=ERTEKdouble(f);break;
case 'D': gveremD=ERTEKlongdouble(f);break;case 's': S = new MAUSTRING;*S=ERTEKstring(f);gverems=S;break;

default: L("Ismeretlen típusazonosító függvényhívás input paraméterénél a # jel után! Hely:
%lu",f.P);EXITFAILURE();break;
} // switch vége
goto Fuggveny_mag_inputparameter_verembetesz_ciklus;

Fuggveny_mag_vegrehajt:

// A veremmutatók korábban ideiglenes változóba mentett értékének elmentése a megfelelő vermekbe:
gveremc=veremmutatoc;gveremC=veremmutatoC;gveremi=veremmutatoi;gveremI=veremmutatoI;
gvereml=veremmutatol;gveremL=veremmutatoL;gveremg=veremmutatog;gveremG=veremmutatoG;
gveremf=veremmutatof;gveremd=veremmutatod;gveremD=veremmutatoD;gverems=veremmutatos;

oldP=f.P; // A programmutató elmentése
pgmf->P=0; // A meghívott függvény programmutatójának a függvény elejére állítása

pgm->folytat(); // ***** Itt hajtódik végre a meghívott függvény ***** !!!!!!!!
// Itt térünk vissza a meghívott függvényből! E visszatérést vagy egy "\n" parancs okozta,
// vagy akármi más. Bennünket csak az érdekel hogy "\n" lett-e a befejeződés oka,
// ekkor kell ugyanis output paramétereket visszaadni, minden más kilépő utasítás nem számít. Egyéb esetben
// különben a kilépés oka feltehetőleg egy xx utasítás. De ha nem, az is le van érdekelve.
// Most leteszteljük, \ okozta-e a meghívott progí végét
// A \ egy egykarakteres utasítás, a végrehajtásakor a P érték a következő bájtra állítódik,
// tehát a P-1 mutat a \ karakterre.

if(pgmf->p[pgmf->P - 1] == '\\') { // Ha \ okozta a kilépést, akkor kell output paramétert átadni
// Meghívjuk a \ utáni aritmetikai kifejezést kiértékelő rutint. Ehhez be kell olvasnunk az annak elején szereplő
// casting operátort.
namespace(*pgmf);
if(pgmf->p[pgmf->P] != '#') {L("Szintaktikai hiba függvényhívás közben: hiányzó # jel a hívott függvény visszatérési
értékénél!\n"
"A függvény neve: %s\nA meghívási kísérlet helye: %lu",pgm->MAUprogramNEVE.s,f.P);ERROR(f,3);
} // if # vége
tipus=k(*pgmf);k(*pgmf);
switch(tipus) {
case 'c': f.FV.c[0]=ERTEKunsignedchar(*pgmf);break;case 'C': f.FV.C[0]=ERTEKsignedchar(*pgmf);break;
case 'i': f.FV.i[0]=ERTEKunsignedshortint(*pgmf);break;case 'I': f.FV.I[0]=ERTEKsignedshortint(*pgmf);break;
case 'l': f.FV.l[0]=ERTEKunsignedint(*pgmf);break;case 'L': f.FV.L[0]=ERTEKsignedint(*pgmf);break;
case 'g': f.FV.g[0]=ERTEKunsignedlonglong(*pgmf);break;case 'G': f.FV.G[0]=ERTEKsignedlonglong(*pgmf);break;
case 'f': f.FV.f[0]=ERTEKfloat(*pgmf);break;case 'd': f.FV.d[0]=ERTEKdouble(*pgmf);break;
case 'D': f.FV.D[0]=ERTEKlongdouble(*pgmf);break;case 's': f.FS=ERTEKstring(*pgmf);break;
default: L("Ismeretlen típusazonosító függvényhívás visszatérési értékénél a # jel után! Hely:
%lu",f.P);EXITFAILURE();break;
} // switch vége

} // pgmf->P - 1 == \\ teszt vége
// Immár megszületett az eredmény, már amennyiben kellett egyáltalán eredményt visszaadni.
// Most vissza kell állítani a vermek eredeti állapotát

gveremc.veremmutato=veremmutatoc;gveremC.veremmutato=veremmutatoC;gveremi.veremmutato=veremmutatoi;
gveremI.veremmutato=veremmutatoI;gvereml.veremmutato=veremmutatol;gveremL.veremmutato=veremmutatoL;
gveremg.veremmutato=veremmutatog;gveremG.veremmutato=veremmutatoG;gveremf.veremmutato=veremmutatof;
gveremd.veremmutato=veremmutatod;gveremD.veremmutato=veremmutatoD;

// Már csak a stringverem visszaállítása van hátra, de ez szopatós, mert a belé elhelyezett stringmutatók által
// jelzett stringek területét is fel kell szabadítanunk...
{ // stringverem felszabadításának kezdete
register unsigned int stringdb;V w;register unsigned int i;register unsigned int j;
stringdb=(gverems.veremmutato-veremmutatos)/(sizeof(void *));
for(i=0;i<stringdb;i++) {
for(j=0;j<sizeof(void *)*j;j++) {w.c[j]=gverems.v[sizeof(void *)*i+j];}
delete (MAUSTRING *)w.p[0];
} // for i vége
gverems.veremmutato=veremmutatos;
} // stringverem felszabadításának vége

f.P=oldP; // Erre kizárólag azért van szükség, mert lehet hogy rekurzív hívásról volt szó...

}

```

Igen, kilóg a képernyőről, de nem olyan szörnyű sokkal, és a megjegyzés is elég sok a kódban, ne feledjük.

Ahhoz hogy ez működjön, picit ki kellett bővíteni a **vc()**, **vc()**, **vl()**, stb függvényeinket is, hogy tudják, ha **@^[]** módon hivatkozunk egy változóra, az ezen fentebb említett globális vermet jelenti.

32. fejezet: Bencsmarkok

E könyvem elején arról elmélkedtem érintőlegesen, hogy »Cseppet se vagyok híve annak a manapság elburjánzó szemléletnek, hogy a programozók leszarják az optimalizálást, s kimondatlanul is ahhoz az elvhez tartják magukat, hogy „ha használni akarod a programomat, paraszt, akkor végy nagyobb gépet”!«

Vagyis, könyvünk a vége felé tart már... Programnyelvünket lehet még sok aprósággal bővíteni, de azért nagyjából már készen van. Ideje elgondolkodnunk az optimalizálási lehetőségeken. Ehhez mindenféleképp illik hogy beépített nyelvi eszközöket biztosítsunk részben a magun kérdezte számára, azaz önmagunknak kik a programnyelvet írjuk, hogy leellenőrizhessük, mik is nyelvünk „szűk keresztmetszetei”, másrészt azoknak, akik majd mau nyelvű programokat írnak!

A legfontosabb amire optimalizálni szoktak s amire lehet, az a SEBESSÉG. Ehhez úgynevezett „bencsmarkokat” kell tudnunk készíteni, azaz olyasmit amivel lemérhetjük, egy adott programrész lefutása mennyi időbe telik. Ennek érdekében használjuk fel azt a lehetőségét a C nyelvnek, hogy le lehet kérdezni, egy adott pillanatban épp hányadik „tick”-nél tart a processzor. Hogy mi a fene az a tick bevallom, nem tudom, de nem is érdekes - ez valamiféle időegység, amit el kezd számolni amikor elindul a gép vagy legalábbis az adott program. Ha ezt le tudjuk kérdezni 2 különböző alkalommal, akkor a különbség azzal arányos, amennyi ideig a két lekérdezés közt a program futott.

A megfelelő include fájlt megnézegetve úgy találtam, ez a jelenlegi implementációk esetén akkora, hogy épp egymillió „tick” ad ki egy másodpercet. Ez nagyon remek, mert eszerint egy „tick” pont egymilliomod másodperc. Ez szerintem elegendően pontos nekünk.

Ennek öröme azonnal csináljunk az F struktúránkba egy ilyen micsodát:

```
clock_t CLOCK[256];
```

Ez nekünk 256 speciális változóra nyújt lehetőséget, melyek nevére ugyanazon szabályok érvényesek, mint a többi változóéra (azaz unsigned char értékek jelzik a nevüket). Fontos megjegyezni azonban, hogy a közönséges változóktól eltérően ezek kezdeti értékét semmi módon nem garantálja a mau programnyelv!

Ezek jó nagy számok, emiatt nemigen érdemes másképp kezelni őket mint unsigned long long értékeket, ettől függetlenül azonban tettem róla hogy átalakíthatóak legyenek más típusokká is. Használatuk:

```
#!mau
```

```
#t@a; // inicializálunk egy időváltozót
{ | 3000000
#i@i=?|;
|}
#t@b; // inicializálunk egy másik időváltozót
#g@e=(#t@b)-(#t@a);
"Eredmény: " ?g @e; /;
```

A fenti programnak persze a világon semmi értelme, hótt' ökörség számos szempontból. Arra azonban jó, hogy lecsekkoljuk, mennyi ideig tart egy 3 milliószor lefutó ciklus nála. Az eredményt milliomod másodpercekben kapjuk meg; az egyes futások eredményei közt - főleg ha sokmagú gépen futtatjuk - lehet akár 3-4% eltérés is. Nálam, T530-as gépen core i7 processzorral ez kb 1.15 másodpercig tartott.

Ez nem rossz idő szerintem, de azért elgondolkoztam rajta, nem lehetne-e javítani! Mit is csinál ez a ciklus? Ugye, előszedi a veremből a ciklusváltozót. Vissza-teszi oda. Castolja és beteszi a változóba. Jön a ciklus vége, ott újra előszedi a változót, megnézi nulla-e, ha nem nulla levon belőle egyet, visszateszi a verembe... Elgondolkodtam a ciklusváltozó veremből kiszedésén. Minek is kéne azt onnan kiszednie, ha változatlanul visszateszi rögtön?! Nézzük csak meg az ezt művelő programrészt... Íme:

```
unsigned long long ciklusvaltozo_erteke(F& f) { // visszaadja az aktuális (azaz legbelső)
// fix darabszámszor lefutó ciklusváltozó értékét
unsigned long long darabszam;
darabszam=f.CIKLUSVEREM; // Kiolvassuk a darabszámot a veremből
f.CIKLUSVEREM=darabszam; // Visszatesszük a darabszámot a verembe
return darabszam;
}
```

Nos, ez nagy pocskéklás! Erre semmi szükség. Kis gondolkodás után ebből ez lett:

```
unsigned long long ciklusvaltozo_erteke(F& f) { // visszaadja az aktuális (azaz legbelső)
// fix darabszámszor lefutó ciklusváltozó értékét
return *((unsigned long long *)(&f.CIKLUSVEREM.v+f.CIKLUSVEREM.veremmutato-sizeof(unsigned long long)));
}
```

Ennek következtében pedig máris 16 százalékot spóroltam a bencsmarkjaim szerint az előbb bemutatott mau nyelvű példán!

Na most, 16 százalék azért már cseppet sem megvetendő nyereség...

Ha már a bencsmarkoknál tartunk, ide kívánczik - mert ez is olyan „idővel összefüggő dolog” - két pici utasítás, melyek meghatározott időre felfüggesztik a program végrehajtását:

```
int fuggveny_WS(F& f) { // Várakozás adott számú másodpercig
// Szintaxis:
// WS x
// ahol x egy unsigned int kifejezés
unsigned int x;x=ERTEKunsignedint(f);
if(x) sleep(x);
return 0;
}
// -----
int fuggveny_Us(F& f) { // Várakozás adott számú mikroszekundumig
// Szintaxis:
// Us x
// ahol x egy unsigned int kifejezés
unsigned int x;x=ERTEKunsignedint(f);
if(x) usleep(x);
return 0;
}
```

Használata gondolom világos, de íme itt egy program is rá:

```
"Várok 5 másodpercet!\n"
WS 5;
"Most várok 500000 mikroszekundumot!\n"
Us 500000;
```

Na és hát a dolog úgy áll, hogy **ITT A KÖNYVEM VÉGE!** Természetesen még rengeteg mindenféle aprósággal lehet bővíteni a mau programnyelvet, s nyilván fogom is majd. De olyan igazán nagy strukturális változások már nem lesznek benne, s azt hiszem olyasmik sem, amik valami rendkívüli horderejű új képességet vezetnének be belé. Ezekből itt és most fejezem be e könyvet, mert valahol muszáj abbahagyni. A mau programnyelv a jelenlegi állapotában már használható minden általános feladatra, szerintem aki saját programnyelvet óhajt írni, ez a vágya, igencsak boldog lehet, ha elér vele egy olyan állapotba, amiben jelenleg a mau nyelv van. Innentől már igazán könnyű továbbfejleszteni, felcsicsázni mindenféle újabb apró kényelmi szolgáltatásokkal.

Most tehát neked, kedves Olvasóm, hogy idáig eljutottál, egyetlen további teendőd van csak: KEZDJ BELE A TE SAJÁT PROGRAMNYELVED MEGALKOTÁSÁBA...

Sok sikert hozzá!

Ja: hogy végülis mi lett a mau programnyelv végleges változata? Mivé nőtte ki magát? Ezt megtudhatod, ha elolvasod az erről szóló másik könyvemet, a részletes leírást, aminek címe:

A mau programozási nyelv

Az is a MEK-ből tölthető le, ingyen! A forráskód is...

<http://mek.oszk.hu/13000/13093/>